technische universität
dortmund

Bachelorarbeit

Integration von Clusterverfahren in AMUSE

Pauline Speckmann
17. Mai 2021

Gutachter:
Prof. Dr. Günter Rudolph
Dr. Igor Vatolkin

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Algorithm Engineering (LS-11)
https://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das *Advanced MUSic Explorer*[1] (kurz AMUSE) Framework [33] vereinigt verschiedene Werkzeuge zur Musikdatengewinnung in sich, um die Anwendung dieser und die Interaktion zwischen ihnen zu vereinfachen, indem eine einheitliche Nutzung ermöglicht wird. Dies bietet vielfältige Anwendungsmöglichkeiten, da eine physische Musiksammlung heutzutage eine Ausnahme darstellt [35]. Durch die voranschreitende Digitalisierung tragen viele ihre Musik zu jeder Zeit direkt auf ihrem Smartphone mit sich.
Das erlaubt vor allem eine einfache, digitale Verarbeitung durch Computer, sodass z. B. viele verschiedene Internetdienste, basierend auf favorisierten Musikstücken, automatische ein personalisiertes Radio erstellen können [35]. Andere Plattformen helfen bei der Findung eines spezifischen Liedes allein mittels einer gesummten Melodie und viele Programme zur Verwaltung und zum Abspielen von Musik versuchen, die eigene Musiksammlung anhand der Ähnlichkeit der Stücke zu organisieren. Oft bauen diese Anwendungen auf Ergebnissen einer Musikdatenanalyse auf, welche wiederum mithilfe eines Programmes wie AMUSE durchgeführt wird. Dafür führt AMUSE verschiedene Experimente durch, welche wie folgt unterschieden werden:

1. Die Extraktion von Merkmalen aus gegebenen Musikstücken
2. Die Verarbeitung der extrahierten Merkmale
3. Das Training eines Klassifizierungsmodells mithilfe der verarbeiteten Merkmale
4. Die Klassifizierung weiterer Musikstücke durch das trainierte Modell
5. Die Validierung der Klassifizierung
6. Die Optimierung der Merkmalsextraktion, der Verarbeitung und des Modells

Für die Klassifizierung (Aufgabe 4) werden bisher nur überwachte Lernmethoden bereitgestellt, was innerhalb dieser Arbeit geändert werden soll. Die überwachten Lernmethoden zielen auf die eindeutige Zuordnung der Eingabedaten in vordefinierte Klassen ab, indem sie zunächst einen Satz von Beispieldaten bzw. eine Trainingsmenge erhalten, an dem sie genau das trainieren können

[1] https://github.com/AdvancedMUSicExplorer/AMUSE, zugegriffen am 2021-04-23

(Aufgabe 3). Dazu muss zusätzlich zu den Beispieldaten auch bereits die richtige Klassifizierung vorgegeben werden, sodass die Ergebnisse der Methode verifiziert bzw. falsifiziert werden können. Die Vorhersagen der überwachten Lernmethoden beruhen entsprechend auf bereits gelösten Fällen [12]. Im Gegensatz dazu stehen unter anderem die unüberwachten Lernmethoden, zu denen auch die Clusterverfahren zählen. Deren Ziel ist es, ohne zusätzliche Angaben, die Eingabedaten anhand ihrer Beschaffenheit einer oft unbestimmten Anzahl von Klassen zuzuweisen und Ergebnisse innerhalb eines gewissen Fehlerbereiches zu produzieren.

1.1 Nutzen und Möglichkeiten

Clusterverfahren ordnen die Eingabedaten (Objekte) (semi-)automatisch in Gruppen (*Cluster*) ein [10], sodass die Objekte innerhalb eines Clusters ähnlicher zueinander sind als zu Objekten aus anderen Clustern. So können bspw. durch eine Clusteranalyse natürliche Muster innerhalb der Eingabedaten erkannt werden.

Eines der Ziele von Clusteranalysen ist das Verständnis der Daten [36], wobei automatisch ein Zusammenhang zwischen den Eingabedaten gefunden wird. So werden gemeinsame, bisher unter Umständen noch unbekannte, Charakteristiken erkannt. Darunter kann, bezogen auf den konkreten Anwendungsfall der Arbeit, jedes der extrahierten Merkmale fallen, wie z. B. das Tempo oder die Grundfrequenz eines Liedes. Ein weiteres Ziel ist das Clustern zum Nutzen der gewonnenen Informationen. Es kann etwa versucht werden, ein repräsentatives Objekt aus einer Gruppe individueller Objekte herauszuabstrahieren. Dies findet bspw. in der Musiksegmentierung Anwendung, wo innerhalb eines Musikstückes automatisch einzelne Komponenten wie Strophe und Refrain erkannt werden sollen.

Beide Ansätze stellen im Rahmen der Musikdatenanalyse interessante Vorgehen dar. Im Hinblick auf ein Musikempfehlungssystem können so z. B. Lieder, welche auf natürlicherweise ähnlich sind, unabhängig vom Nutzerverhalten oder ihren Genres vorgeschlagen werden. Dies spielt gerade heutzutage, in einer Ära der Musikstreamingdienste wie etwa Spotify, eine enorme Rolle. Eine Begutachtung dieser Stücke liegt dabei auch nahe, denn so können die genauen Ähnlichkeiten herausgefunden werden. Dies kann dabei helfen, mehr über die vorliegenden Daten und die Aussagekraft der gewählten Merkmale zu lernen. Das kann wiederum in Forschungen im Musikbereich oder aber auch in psychologischen Bereichen (wie der subjektiven Wirkung von Musik auf Individuen) zu neuen Erkenntnissen führen.

Insgesamt bietet die Erweiterung von AMUSE durch unüberwachte Lernmethoden mehr Möglichkeiten. Mit überwachten Lernmethoden können Musikdaten im Hinblick auf fest vorgegebene Kategorien analysiert werden, wobei auch dort bezüglich großer, unbekannter Datenmengen eine Clusteranalyse sinnvoll sein kann. Durch Clusterverfahren werden deutlich andere Anwendungsgebiete, wie bspw. das Finden neuer Zusammenhänge zwischen verschiedenen Datenmengen, zur Verfügung gestellt. Dies treibt insbesondere das Ziel von AMUSE, verschiedene Werkzeuge zur Gewinnung von Musikdaten in sich zu vereinigen, weiter voran.

1.2 Verwandte Arbeiten

Im Bereich der Musikdatenanalyse werden Clusterverfahren bereits auf viele unterschiedliche Arten eingesetzt. Im Folgenden werden einige Arbeiten vorgestellt, die mithilfe des Clusterns von Musikdaten interessante und relevante Ergebnisse erzielen, um Dienste bzw. Informationen zur weiteren Verarbeitung bereitzustellen oder Antworten auf Fragen zu geben.
So wird in [22] versucht, verschiedene Gruppen von Musikhörern mit ähnlichen Geschmacksmustern zu identifizieren, indem eine Clusteranalyse auf Musikstil-Ergebnisse angewandt wird. Diese beschreiben durchschnittliche Ergebnisse einer Bewertung von Musikgenres einer Gruppe von Probanden. Dabei repräsentiert jede einzelne der Stilbewertungen die individuelle Einschätzung eines Probanden eines spezifischen Musikstils. Insgesamt wurden sieben aufeinanderfolgende, hierarchische Clusteranalysen verwendet, um Gruppen zu definieren, welche die Komplexität der Mustererkennung von Musikpräferenzen reflektieren, und die danach noch weiter analysiert wurden. Schließlich konnten so angemessene Gruppen identifiziert werden, was den Nutzen der Clustermethoden belegt, um die Zusammenstellung eines Musikpublikums zu skizzieren.
Ein Framework zur Extraktion und Modellierung von Hörstimmungen durch das Clustern von Musiktiteln des Hörverlaufs eines Nutzers wird in [4] behandelt. Dabei beschreiben Hörstimmungen den Gemütszustand, der durch das Hören von Musik hervorgerufen, verstärkt oder allgemein mit ihr assoziiert wird. Dafür werden zunächst die Nutzerdaten und die Audiomerkmale, der zum Hörverlauf des Nutzers gehörenden Titel, gesammelt, wobei letztere geclustert werden. Aus diesen Clusterergebnissen werden die Hörstimmungen extrahiert und deren Dichte innerhalb des Hörverlaufs ermittelt, sodass auf dessen Basis eine *Playlist* mit angemessensten Titeln für jede Stimmung des Nutzers erstellt werden kann. Diese wird wiederum verwendet, um dem Nutzer passend zu seiner aktuellen Stimmung Lieder vorzuschlagen.

[18] stellt ein clusterbasiertes Musikempfehlungssystem vor, das neben Nutzerdaten auch Inhaltsinformationen über die Musikstücke nutzt. Insgesamt können durch die sogenannte ICHM (*Item-based Clustering Hybrid Method*) mehrere Problematiken behoben werden. Zu diesen zählt unter anderem das Kaltstartproblem, bei dem zum Start eines Empfehlungssystems bzw. nach dem Hinzufügen eines neuen Musikstückes noch keine Daten von Nutzern bereitstehen, auf dessen Basis eine Zuordnung zu ähnlich bewerteten Musikstücken geschehen könnte. Durch das Clustern von Musikstücken in verschiedene Gruppen können inhaltsbasierte Informationen für kollaborative Ähnlichkeitsberechnung bereitgestellt werden, mithilfe derer die Vorhersagen für die Musikstücke getroffen und das Problem gelöst werden kann. Ein Jahr später wurde die ICHM bereits um die Berücksichtigung von Audiomerkmalen erweitert [19], was weitere Herausforderungen adressiert und löst.
In [17] wird eine Methode zur Segmentierung einer Musikdatei in Abschnitte beschrieben, sodass die gesamte Struktur eines Liedes erkannt wird (z. B. *Intro*-Vers-Refrain-Vers-Refrain-*Bridge*-Vers-Refrain-*Outro*). Dies führt zu einem Cluster-Framework, in dem musikalische und andere Vorinformationen als einfache Bedingungen modelliert werden können. Die Experimente zeigen, dass ein Clusteransatz die musikalische Struktur effektiv als eine Menge von Cluster-Schwerpunkten erfassen kann. Dabei fasst jeder Schwerpunkt ein Gaußsches Mischmodell (*Gaussian Mixture Model*, kurz GMM) im Merkmalsraum zusammen, das einem bestimmten Segment-Typ, wie bspw. Strophe oder Refrain, entspricht. Dieses Wissen findet z. B. in der Musikzusammenfassung Verwendung, bei der automatisch eine kurze, repräsentative Musikvorschau ausgewählt werden soll. Ein anderes Beispiel ist die automatische Abschnitt-zu-Abschnitt-Ausrichtung von Musikstücken, was als Unterstützung für Suchalgorithmen dient.

1.3 Ziele der Arbeit

Die Hauptziele dieser Bachelorarbeit sind die erfolgreiche Integration von zwei bis drei Clusterverfahren der Software-Plattform RapidMiner[2], sowie eines autonomen Clusterverfahrens in das AMUSE-Framework. RapidMiner stellt dabei eine sehr mächtige, modulare Lernumgebung dar, welche in Abschnitt 2.2 besprochen wird. Während die Auswahl der Verfahren weitestgehend uneingeschränkt ist, soll es sich bei der autonomen Methode explizit um ein anderes Verfahren als die in RapidMiner verfügbaren handeln.

[2] https://rapidminer.com/, zugegriffen am 2021-04-23

Die Integration soll über ein gemeinsames Interface geschehen, über das alle unüberwachten Lernmethoden aufgerufen werden können. Zu beachten sind dabei die stark variierenden Input- bzw. Outputwerte der einzelnen Verfahren, welche jeweils aufeinander abgestimmt werden müssen, sowie die erforderlichen Anpassungen innerhalb von AMUSE. Darunter fällt insbesondere die Benutzeroberfläche (*Graphical User Interface*, kurz GUI). Schließlich soll eine Nutzung der integrierten Clusterverfahren über die Konsole und die AMUSE GUI möglich sein. Die Verfahren sollen dabei mittels eines Funktionstests validiert werden, um die korrekte Integration zu belegen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen, die zum Verständnis des späteren Hauptteiles (Kapitel 3, der Integration der Clusterverfahren in AMUSE) benötigt werden, erläutert. Dabei wird in Abschnitt 2.1 zunächst die Anwendung von AMUSE vorgestellt, um einen Überblick der Funktionen mit Fokus auf die Klassifizierung und die interne Verarbeitungskette zu geben. Als nächstes wird im Abschnitt 2.2 RapidMiner als Umgebung für Data-Mining und maschinelles Lernen kurz beschrieben, bevor auf die Bibliothek eingegangen wird, welche in dieser Arbeit verwendet wird.
Weiterhin werden neben den beiden praktischen Anwendungen auch die theoretischen Grundlagen erklärt. So wird es eine Einführung in die Thematik der Clusterverfahren und im Besonderen in die für die Integration in AMUSE gewählten Verfahren geben (Abschnitt 2.3). Schließlich werden Distanzmaße genauer betrachtet (Abschnitt 2.4), um noch einen tieferen Einblick in die Arbeitsweise von Clusterverfahren im Allgemeinen zu ermöglichen.

2.1 AMUSE

Bei AMUSE [33] handelt es sich um ein Framework, welches diverse Werkzeuge zur Musikdatengewinnung in sich vereinigt. Der *Advanced MUSic Explorer* wurde vom Lehrstuhl 11 für Algorithm Engineering der TU Dortmund zuerst im Jahre 2010 veröffentlicht und seitdem weiter ausgearbeitet.
Die Entwicklung des in Java geschriebenen Programms basiert stark auf dem Gedanken, eine Schnittstelle zwischen den bereits existierenden Werkzeugen zur Musikdatenanalyse erschaffen zu wollen. Diese haben oft verschiedene Fokuspunkte und damit auch entsprechende Vor- und Nachteile. Unter Umständen sind bspw. einige Werkzeuge zu spezifisch, um effektiv damit zu arbeiten, oder andere sind zu generisch, sodass es schwierig ist, eine angemessene Lösung für die jeweiligen Probleme zu erstellen. Da sich die Software der Wahl entsprechend des

Anwendungsgebietes ändert, ist es sinnvoll, sie zu zentralisieren und eine einheitliche Anwendung zu ermöglichen.
Neben dieser Vereinigung stellt AMUSE auch eigene, neue Anwendungsmöglichkeiten in bisher eher unterrepräsentierten Bereichen bereit. Dazu gehört z. B. einschließlich, aber nicht ausschließlich, die Merkmalsverarbeitung als Zwischenschritt nach der Extraktion der rohen Merkmale und der fertigen Eingabe zur Klassifizierung.

Die folgende Abbildung 2.1 stellt die AMUSE Benutzeroberfläche der Experiment-Übersicht innerhalb des *Experiment erstellen*-Bereiches dar. Die anderen Bereiche, wie bspw. die Einstellungen oder die Erstellungen von Annotationen, sollen hier nicht weiter betrachtet werden.

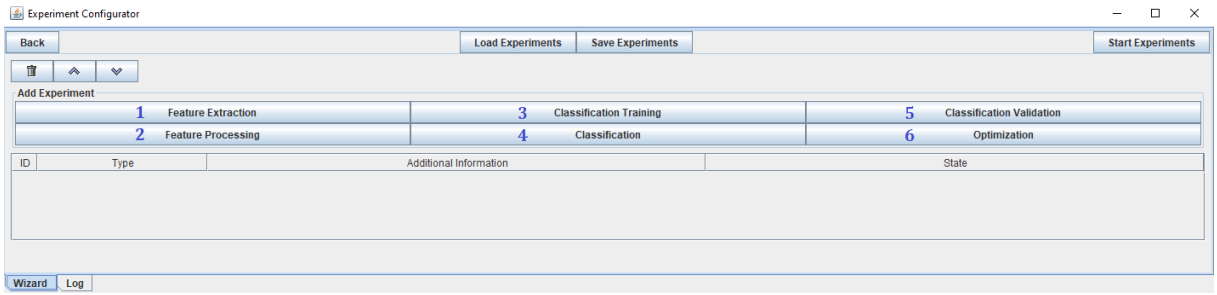

Abbildung 2.1: AMUSE GUI des Bereiches *Experiment erstellen*

Das logisch erste Experiment (**1**) besteht aus der Extraktion der gewählten Merkmale. So kann hier zunächst aus einer Liste aller aktuell 46 von AMUSE bereitgestellten Merkmale, wie bspw. des Tempos, eine unechte Teilmenge dieser gewählt werden. Die gewählten Merkmale werden nach dem Abschluss der Konfiguration aus den bereitgestellten Musikstücken extrahiert. AMUSE erstellt daraufhin für alle Musikstücke und alle gewählten Merkmale eine eigene Datei, in der diese rohen Merkmale abgespeichert werden.
Im nächsten Experiment (**2**) werden diese zuvor extrahierten, rohen Merkmale nun für die selbe Auswahl der Musikstücke verarbeitet. Dabei werden verschiedene Algorithmen zur z. B. Zeitverringerung oder Vorverarbeitung bereitgestellt, aus denen eine beliebige Kombination ausgewählt und angewendet werden kann. Zum Abschließen dieser Einstellung bzw. Konfiguration muss noch eine Methode zur Konvertierung der Matrix zu Vektoren, wie z. B. dem Gaußschen Mischmodell, gewählt werden. AMUSE führt dann entsprechend der getroffenen Entscheidungen alle gewählten Merkmale pro Lied in einer Datei zusammen.
Für überwachte Lernmethoden folgt das Klassifizierungstraining (**3**), indem ein Modell zu Erkennung einer oder mehrerer Kategorien trainiert wird. Dieser Schritt fällt für unüberwachte

Lernmethoden, die nicht mit Modellen arbeiten, weg. Das darauf folgende Experiment (4), die Klassifizierung, stellt das Hauptaugenmerk dieser Arbeit dar und besteht aus der Einordnung der gewählten Musikstücke in verschiedene Gruppen. Im Folgenden wird die Klassifizierung deswegen vor den letzten beiden Experimenten mithilfe von Abbildung 2.2 genauer betrachtet.

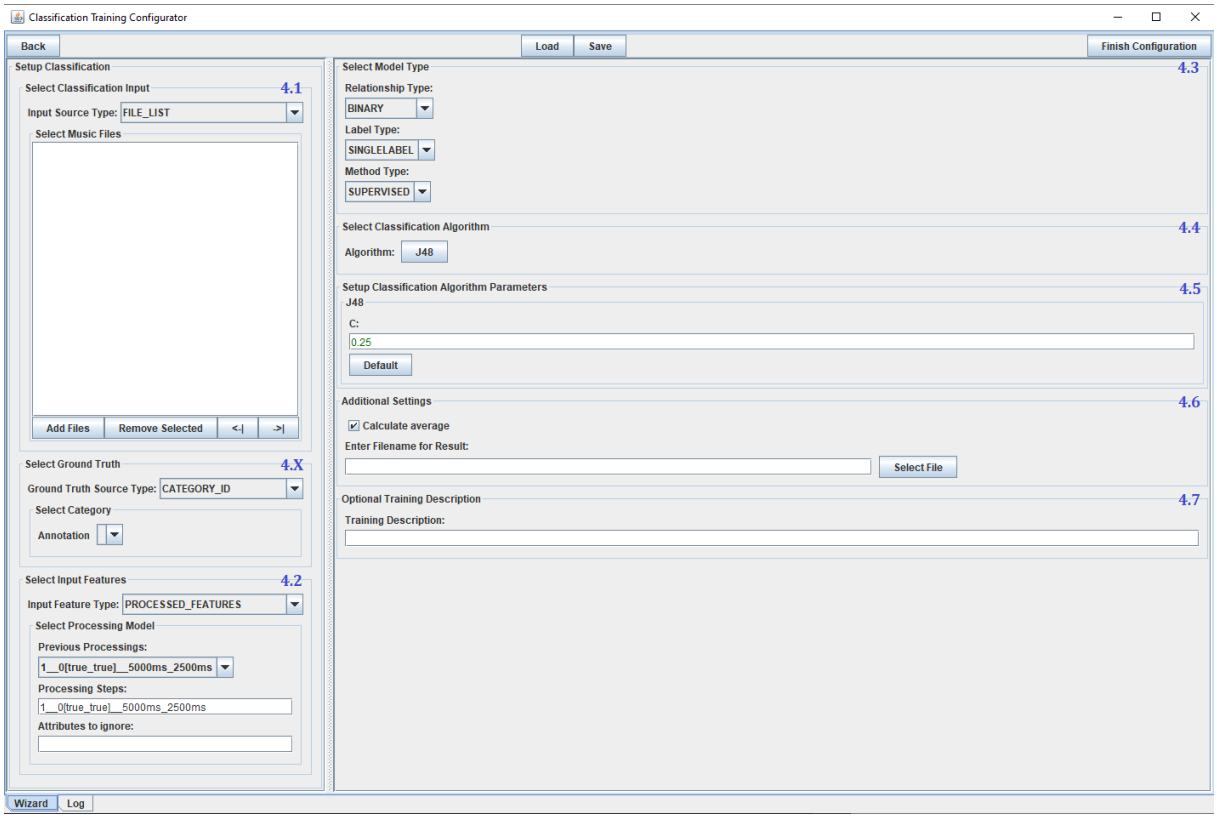

Abbildung 2.2: AMUSE GUI des Bereichs *Klassifikation*

Innerhalb der Klassifizierung müssen zunächst die Musikstücke für dieses Experiment gewählt (**4.1**) und die (Vor-) Verarbeitung der extrahierten Merkmale angegeben werden (**4.2**). Darauf folgt die Definition des Modelltyps (**4.3**), bestehend aus der hier relevantesten Unterscheidung innerhalb des Methodentyps zwischen überwachter und unüberwachter Klassifizierung. Nach der Wahl des spezifischen Algorithmus (**4.4**) können darauf folgend noch die zugehörigen Standardwerte der Parameter angepasst werden (**4.5**). Für die überwachten Lernmethoden muss zusätzlich eine sogenannte Grundwahrheit (*Ground Truth*) bereit gestellt werden (**4.X**). Diese Grundwahrheit enthält die richtige Zuordnung aller Musikstücke zu der vorgegebenen Klasse bzw. den vorgegebenen Klassen. Dies ist jedoch im Fall einer unüberwachten Lernmethode nicht nötig und dieser Bereich kann ignoriert werden. Die Konfiguration kann schließlich durch die Eingabe der weiteren Einstellungen (**4.6**), insbesondere des Namens bzw. Pfades der Ergebnisdatei,

und einer optionalen Trainingsbeschreibung (**4.7**) abgeschlossen werden. Das Experiment kann dann über den Bereich aus Abbildung 2.1 mittels der Schaltfläche *Starte Experimente* in der rechten oberen Ecke gestartet werden. Nach dessen Abschluss werden die Ergebnisse unter dem zuvor spezifizierten Pfad als Datei im *Attribute Relation File Format* (kurz ARFF) gespeichert. Eine Datei dieses Formats enthält zwei dedizierte Abschnitte (*@RELATION* und *@DATA*). Dort geben mit *@ATTRIBUTE* gekennzeichnete Attribute des ersten Abschnitts die Reihenfolge an, in der die im zweiten Abschnitt zeilenweise geschriebenen Datenwerte (durch Kommata getrennt) diesen Attributen zugeordnet werden. In einer solchen Ergebnisdatei werden die Pfade zu den Musikstücken, für jedes Lied die jeweiligen Partitionsunterteilungen, sowie das Klassifizierungsergebnis angegeben.

Zurück zur Übersicht aller Experimente: Die automatische Validierung der Klassifizierungsergebnisse (**5**) fällt, ähnlich wie das Klassifizierungstraining, bei der Verwendung von unüberwachten Lernmethoden weg. Dort werden anhand der Grundwahrheit in Kombination mit dem trainierten Modell die Ergebnisse der Klassifizierung überprüft, welche Clusterverfahren beide nicht besitzen. Das logisch letzte Experiment (**6**) bildet die Optimierung, in der die Konfigurationen der bisherigen Aufgaben geprüft werden, und so die Klassifizierungsergebnisse eventuell noch verbessert werden können.

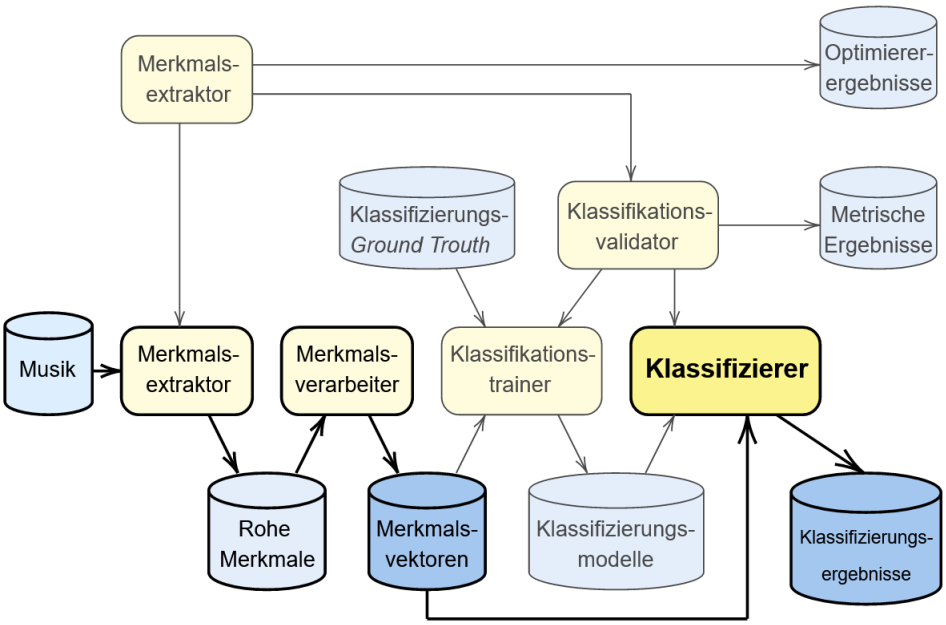

Abbildung 2.3: AMUSE Verarbeitungskette

Insgesamt ergibt sich so ein Zusammenspiel aller Experimente, welches in Abbildung 2.3 dargestellt wird. Die blauen Zylinder stellen Daten und die abgerundeten, gelben Rechtecke Experimente dar. Dabei heben die fetten Umrandungen bzw. Pfade die Verarbeitungskette der unüberwachten Lernmethoden hervor. So werden aus der gegebenen Musik die ausgewählten Merkmale extrahiert, was die rohen Merkmalsdaten ergibt, die wiederum zu Merkmalsvektoren verarbeitet werden. Diese stellen die direkte Eingabe der Klassifizierung dar, deren Ergebnisse schließlich abgespeichert werden. Farblich hervorgehoben sind weiterhin die für diese Arbeit relevanten Teile der Verarbeitungskette, was die Merkmalsvektoren als direkte Eingabedaten für die Clusterverfahren, den Klassifizierungsprozess, und die letztendlich verwertbaren Klassifizierungsergebnisse betrifft.

2.2 RapidMiner

RapidMiner wurde ursprünglich im Jahre 2001 unter dem Namen *Yet Another Learning Enviroment* (kurz YALE) [30] vom Lehrstuhl für künstliche Intelligenz an der TU Dortmund als modulare Lernumgebung entwickelt. Diese erlaubt unter anderem verschachtelte und generell den leichten Austausch von Operatoren und ermöglicht damit die systematische Auswertung und den Vergleich von Operatoren innerhalb der gleichen Aufgabe.
Heutzutage stellt RapidMiner ein Werkzeug für den gesamten Lebenszyklus von Datenwissenschaften dar [1]: Angefangen bei der Datenvorbereitung, über maschinelles Lernen und bis zum Einsatz eines vorausschauenden Modells. Dies macht RapidMiner zu einer umfangreichen Umgebung, sodass es eine der Herausforderungen innerhalb ihrer Anwendung ist, den Überblick über alle Operatoren und ihre Möglichkeiten zu behalten. Im Verlauf eines Experiments, mit diversen Unterprozessen und langen Operatorverknüpfungen, wird dies ebenfalls nicht einfacher. Neben den verschiedenen Operatoren für den Datenzugriff, ihrer Bearbeitung bzw. Vorbereitung, wie bspw. die Normalisierung, und ihrer Nutzung, worunter z. B. auch Makros oder Prozessplanung fallen, werden auch Operatoren zur Modellierung angeboten. Unter die Modellierungsoperatoren fallen insbesondere auch (un-) überwachte Lernmethoden, sodass dieser, für diese Arbeit relevante, Teil der unüberwachten Verfahren folgend genauer betrachtet wird.
Zählt man die Varianten der k-Means Methode (FastKMeans, KernelKMeans, kMeans und XMeans, beschrieben in 2.3.1) zusammen, stellt die Version 9.7.0 der in AMUSE integrier-

ten RapidMiner-Bibliothek (zu finden unter `amuse/lib/rapidminer.jar`) acht verschiedene Clusterverfahren bereit:

1) *SingleLinkageMethod*
2) *AverageLinkageMethod*
3) *CompleteLinkageMethod*
4) ***kMeans***
5) ***DBScan***
6) ***SVClustering***
7) *RandomClustering*
8) *TopDownClustering*

Die im Verlauf dieser Arbeit verwendeten Operatoren 4−6 werden in Abschnitt 2.3 erläutert. Alle Operatoren benötigen die Eingabedaten im RapidMiner spezifischen *ExampleSet*-Format zusammen mit den, für die Verfahren angepassten, Parametern. Diese Parameter können innerhalb von AMUSE über Namenskonstanten, die in der Operatorklasse definiert sind, angesteuert und gesetzt werden. Die jeweiligen Schnittstellen (*Ports*), über welche die Ein- und Ausgaben der Operatoren laufen, können auf ähnliche Weise angesteuert werden. Dabei ist die Verbindung des Operators mit dem Prozess, der den Operator aufruft und seine Ausgabe erhält, kontraintuitiv. So muss der Ausgabeport des Prozesses mit dem Eingabeport des Operators, sowie der Ausgabeport des Operators mit dem Eingabeport des Prozesses verbunden werden. Wurden schließlich die Operatorklassen definiert, die Parameter gesetzt, der Operator zum Prozess hinzugefügt, die Schnittstellen verbunden und die Eingabe in das richtige Format gebracht, kann der Prozess gestartet und die Ergebnisse abgerufen werden.

2.3 Clusterverfahren

Das Clustern von Daten kann als wohl wichtigstes Problem des unüberwachten Lernens betrachtet werden [20]. Es behandelt das Finden einer Struktur innerhalb einer Kollektion von unmarkierten Daten, indem es Datenobjekte (semi)automatisch in Teilmengen (auch Kategorien, Klassen, Gruppen und Cluster genannt) einteilt, sodass die Objekte innerhalb eines Clusters ähnlicher zueinander sind als zu Objekten anderer Cluster [10, 20, 36]. Durch die vielseitigen Anwendungsmöglichkeiten spielt das Clustern eine wichtige Rolle in Anwendungen wie z. B. dem maschinellen Lernen, der Mustererkennung und der Informationsgewinnung. Im Bereich der Musikdatenanalyse ist in der heutigen Zeit insbesondere ein Musikempfehlungssystem eine relevante Applikation.

Natürlich hat jedes Clusterverfahren Vor- und Nachteile, sodass ihre Effektivität stark von dem Kontext des Anwendungsgebietes abhängig ist [20]. Für ihren Einsatz wird zunächst

auch nur angenommen, dass verschiedene Gruppen existieren, selbst wenn eigentlich keinerlei Gruppenstruktur vorhanden ist. Diese Annahme kann schwach oder auch ganz falsch sein, sodass die Ergebnisse des Clusterns mit entsprechender Vorsicht zu genießen sind und nicht generalisiert werden sollten. Vereinfacht können Clusterverfahren jedoch zunächst in zwei Kategorien eingeteilt werden: Partitionierend und hierarchisch [20, 35]. In diesem Abschnitt werden nun weiterhin die zwei Unterkategorien, sowie im Detail die Verfahren erläutert, welche im Hauptteil dieser Arbeit (Kapitel 3) in AMUSE integriert werden sollen.

2.3.1 Partitionierende Clusterverfahren

Die partitionierenden Clusterverfahren bestimmen alle Cluster zur gleichen Zeit [20]. Oft teilen sie die Objekte auf eine vorab gewählte, meist k genannte, Anzahl von Clustern auf. Die Ermittlung eines sinnvollen Wertes für k ist ein ganz eigenes Problem, zu dem es diverse Lösungsansätze gibt (z. B. [21]), aber die hier nicht weiter betrachtet werden sollen. Eine einfache Faustregel, falls man keine gewünschte oder aus dem Kontext vorgegebene Anzahl von Clustern hat, ist: Wähle $k \approx \sqrt{n/2}$, wobei n die Anzahl aller zu clusternden Objekte darstellt. Prinzipiell können partitionierende Verfahren in viele weitere verschiedene Unterkategorien unterteilt werden [36], wovon folgend zwei für diese Arbeit relevante betrachtet werden:

***Prototypbasierte* Algorithmen** wählen einen Prototypen für jedes Cluster und formen die jeweiligen Cluster anhand der Objekte, die um den zugehörigen Prototypen herumliegen. Diese Nähe wird durch ein Distanzmaß (siehe Abschnitt 2.4) definiert. Beispielsweise kann ein Schwerpunkt (auch Mittelpunkt oder *Centroid* genannt) als Prototyp genutzt werden, sodass alle Objekte anhand ihres nächstliegenden Schwerpunktes einem Cluster zugeordnet werden.

***Dichtebasierten* Algorithmen** teilen Cluster als Regionen von eng beieinander liegenden Objekten ein, die von Bereichen mit einer sehr niedrigen Objektdichte umgeben sind. Sie werden oft verwendet, wenn Cluster eine irreguläre Form haben oder Außenseiter bzw. Rauschen vorhanden sind. Dabei sind Außenseiter Objekte, die eher abseits zwischen den Clustern liegen, und mehrere Außenseiter werden als Rauschen bezeichnet.

Weiterhin werden drei partitionierende Verfahren betrachtet, die im Verlauf der Arbeit verwendet werden. Dazu gehören das prototypbasierte *k-Means*, sowie die dichtebasierten Methoden *DBSCAN* und *Support Vector Clustering*.

k-Means

Ein Beispiel für partitionierende prototypbasierte Clusterverfahren sind die *k-Means*-Methoden [35]. Zu ihnen gehören iterative Verfahren, die zur Bestimmung der Partitionen bzw. Cluster den folgenden Schritten folgen [31, 35, 36]:

1. Wähle k initiale Schwerpunkte.
2. Alle zu clusternden Objekte werden ihrem am nächsten liegenden Schwerpunkt zugeteilt und formen so k Cluster.
3. Die Schwerpunkte werden anhand der ihnen zugewiesenen Objekte aktualisiert.
4. Wiederhole die Schritte 2 und 3, bis die maximale Anzahl der Optimierungsschritte erreicht wurde oder sich kein Cluster mehr verändert.

Die Schwerpunkte müssen dabei die gleichen Dimensionen wie die Eingabeobjekte besitzen. Während sie in Schritt 1 zufällig bestimmt werden können, kann es in praktischen Anwendungen auch zu besseren Ergebnissen führen, sie anhand des eigenen Vorwissens manuell festzulegen [35]. Innerhalb von Schritt 2 können verschiedene Distanzmaße verwendet werden, um die Distanz zwischen den Objekten und den Schwerpunkten zu berechnen. In Schritt 3 gibt es ebenfalls mehrere Wege zur Bestimmung des neuen Schwerpunkts, wie bspw. die Berechnung eines Medians oder des Durchschnitts.
Zusammengefasst ist k-Means ein simples, aber robustes und sehr effektives Clusterverfahren [36], das für viele verschiedene Datenarten angewendet werden kann. Zu den Nachteilen gehören z. B. die schlechte *Performance* bei nicht runden Clustern und die Sensibilität gegenüber Außenseitern, was aber mit einigen neuen Varianten des Verfahrens teilweise ausgeglichen werden kann.

Density Based Spatial Clustering of Applications with Noise

Density Based Spatial Clustering of Applications with Noise (kurz DBSCAN) [9] ist ein dichtebasiertes Clusterverfahren, das zur Erkennung von Clustern mit beliebiger Form entwickelt wurde. Dabei ist die Kernidee, dass für jedes Objekt die Nachbarschaft ermittelt wird, die aus einer minimalen Anzahl *MinPts* von Objekten bestehen muss, um ein eigenes Cluster zu bilden [15]. Die Nachbarschaft ist durch einen Radius *Eps* (ε) eines Objekts gegeben, innerhalb dessen, definiert anhand eines Distanzmaßes, andere Objekte liegen. Die Definition der Nachbarschaft eines beliebigen Objektes p ist in Formel 2.1 mit N als Menge aller zu clusternden Objekte

dargestellt. Objekte, die innerhalb ihrer ε-Distanz mindestens *MinPts* andere Objekte haben, werden mit Formel 2.2 als Kernobjekte definiert.

$$N_{\varepsilon,p} = \{q \in N \mid dist(p,q) < \varepsilon\} \tag{2.1}$$

$$|N_{\varepsilon,p}| > MinPts \tag{2.2}$$

Zur Clusterbildung werden zunächst für ein beliebiges Objekt p alle von ihm aus benachbarten Punkte ermittelt [9]. Falls p ein Kernobjekt ist, liefert dieses Verfahren mittels *Eps* und *MinPts* ein Cluster. Falls p jedoch keine bzw. weniger als *MinPts* ε-erreichbare Nachbarn hat, handelt es sich um einen Grenzpunkt, und DBSCAN betrachtet den nächsten Datenpunkt. Zwei Cluster C_1 und C_2 können zusammengeführt werden, wenn diese nach Formel 2.3 definiert ε-erreichbar voneinander sind. Ein jedes Cluster C ist dabei immer eine nicht leere ($C \neq \emptyset$), maximale ($\forall p,q.$ *falls* $p \in C$ *und* $dist(p,q) < \varepsilon$ *dann* $q \in C$) und verbundene ($\forall p,q \in C.\ dist(p,q) < \varepsilon$) Teilmenge von N.

$$dist(C_1,C_2) < \varepsilon \text{ mit } dist(C_1,C_2) = \min\{dist(p,q) \mid p \in C_1, q \in C_2\} \tag{2.3}$$

Im Idealfall sind die angemessenen Werte für *Eps* und *MinPts*, sowie ein Punkt eines jeden Clusters bekannt. Jedoch gibt es keinen einfachen Weg, diese Informationen vorab für alle Cluster der Datenbank zu erhalten. Es gibt allerdings eine heuristische Methode zur Bestimmung der Werte für das am wenigsten dichteste bzw. dünnste Cluster. Diese Parameter sind immer gute Kandidaten für die globalen Parameter, da diese das Cluster mit der geringsten Dichte, was kein Rauschen darstellt, beschreiben. Die restlichen Cluster mit einer höheren Dichte werden so ebenfalls entdeckt.
Zusammengefasst ist DBSCAN ein Pionier der dichtebasierten Clusterverfahren [15]. Es kann Cluster beliebiger Form und Größe erkennen, selbst wenn die Datenmenge Außenseiter oder Rauschen beinhaltet. Zu den Nachteilen des originalen Algorithmus' gehören jedoch die sehr spezifischen Parameter, welche von dem Benutzer angegeben werden müssen, die Dilemmaanfälligkeit im Falle von Clustern mit unterschiedlichen Dichten, sowie die hohe Rechenkomplexität.

Support Vector Clustering

Im *Support Vector Clustering* (kurz SVC) [5] werden Stützvektormaschinen (*Support Vector Machines*, kurz SVMs) verwendet, die für sich genommen zu den überwachten Lernmethoden

zählen [25]. SVMs bilden Datenpunkte von einem Datenraum mittels einer Kernelfunktion auf einen hoch-dimensionalen Merkmalsraum ab. Innerhalb des Merkmalraumes wird schließlich nach einer so genannten *Hyperplane* gesucht. Vereinfacht gesagt ist eine Hyperplane eine Ebene innerhalb eines hoch-dimensionalen Raumes, welche die Datenpunkte von einander trennt. Da es fast immer mehrere mögliche Hyperplanes gibt, wird mittels Stützvektoren diejenige ausgewählt, welche die Datenpunkte zwei verschiedener Gruppen, die sich am nächsten liegen, maximal separiert.
Im Anwendungsfall des Clusterns gibt es jedoch keine klare Dichotomie, also keine klare Zweiteilung der Objekte. Um eine angemessene Aufteilung zu finden wird nach Abbildung der Daten, mittels einer Funktion f auf den hoch-dimensionalen Merkmalsraum, nach einer Sphäre mit minimalem Radius gesucht, die das Datenbild umschließt [6]. Für f wird hier der Gaußsche-Kernel verwendet, auf den hier aber nicht weiter eingegangen wird. Nachdem die minimal große Sphäre gefunden wurde, wird sie aus dem Merkmalsraum zurück auf den Datenraum abgebildet. Dort entspricht sie einem Satz von Konturen, welche die Datenpunkte auf dem Eingaberaum einschließen.
Mit der Verkleinerung des Breiten-Parameters der Gaußschen-Kernelfunktion brechen die Konturen in eine sich erhöhende Anzahl von unverbundenen Teilen. Die Datenpunkte, welche von einem einzelnen dieser Teile eingeschlossen sind, werden zu einem Cluster gezählt und so können die Konturen als Clustergrenzen interpretiert werden. Im Falle von sich überschneidenden Clustern gibt es eine Konstante, welche einen *weichen* Rand der Cluster und damit Außenseiter ermöglicht. Da so Objekte, die unter Umständen zu keinem Cluster passen, nicht berücksichtigt werden müssen, können die Ergebnisse unter Umständen verbessert werden.
Abschließend findet der SVC-Algorithmus Clusterlösungen mittels Regionen hoher Dichte innerhalb der Daten [5]. So werden Cluster anhand von Lücken oder Regionen mit niedriger Dichte innerhalb des Merkmalraumes getrennt, während keine Annahmen über die Clusteranzahl oder der Formen auf dem Eingaberaum gemacht werden. Es kann weiterhin mit Außenseitern bzw. Rauschen gearbeitet werden, sodass das Verfahren diesen gegenüber relativ robust ist, aber bei stark überlappenden Clustern nur relativ kleine Clusterkerne bestimmen kann [6].

2.3.2 Hierarchische Clusterverfahren

Die hierarchischen Clusterverfahren finden Cluster sukzessiv [27], indem sie die bereits zuvor etablierten Cluster nutzen und so eine hierarchische Struktur aufbauen. Diese Struktur hat an

der Spitze ein einzelnes, allumfassendes und am Ende mehrere einelementige Cluster (für jedes individuelle Objekt eins). Innerhalb eines algorithmischen Sinnes sind diese Verfahren gierig [23, 27]. Dies bedeutet, dass die Algorithmen in jedem Schritt die aktuell beste Option wählen, die zur Verfügung steht. Es wird so eine Sequenz von unumkehrbaren Schritten ausgeführt, um die gewünschte Datenstruktur zu konstruieren. Diese Unflexibilität bedeutet aber auch, dass es insgesamt weniger Berechnungskosten gibt, da keine anderen Szenarien betrachtet werden müssen. Die hierarchischen Clusterverfahren können weiterhin in genau zwei Unterkategorien aufgeteilt werden:

***Agglomerative* Algorithmen** arbeiten von unten nach oben (*bottom-up*). Startend mit den einelementigen Clustern wird in jedem Schritt ein Paar zusammengeführt, bis es nur noch ein großes Cluster gibt, welches alle anderen subsumiert.

***Spaltende* Algorithmen** arbeiten von oben nach unten (*top-down*). Sie beginnen mit allen Objekten in einem allumfassenden Cluster und spalten dieses rekursiv Schritt für Schritt auf, bis jedes Objekt einem eigenen, einelementigen Cluster zugewiesen wurde. Dazu wird ein zweiter, flacher Clusteralgorithmus, wie z. B. k-Means, verwendet, was jedoch die Komplexität des top-down Verfahrens im Vergleich zu den bottom-up Verfahren erhöht.

Die Ausgabe eines hierarchischen Verfahrens kann als ultrametrische Topologie auf den n zu clusternden Objekten gesehen werden [23]. Eine ultrametrische bzw. Baummetrik definiert dabei eine Topologie, sodass für drei Punkte i, j und k die Eigenschaften der Symmetrie ($d(i,j) = d(j,i)$), der positiven Definiertheit ($d(i,j) > 0$ *und falls* $d(i,j) = 0$ *dann* $i = j$) und der starken Dreiecksungleichung ($d(i,j) \leq \max\{d(i,k), d(k,j)\}$) gelten.
Eine andere Sichtweise der Ausgabe ist ein binärer Baum [23, 35], der üblicherweise in diesem Kontext auch Dendrogramm genannt wird. Dabei stellen die einelementigen Cluster die sogenannten Blattknoten dar, während die Vereinigung (*Merge*) von genau zwei Clustern (Kindknoten) jeweils einen inneren Knoten (Elternknoten) darstellt. Das Cluster, das am Ende alle anderen subsumiert, wird durch den Wurzelknoten repräsentiert. Eine Kante existiert immer zwischen einem Elternknoten und seinen beiden Kindern, wobei ein Blattknoten keine Kinder hat.

Wards Agglomeration

Wards Agglomeration [34] stellt ein hierarchisches agglomeratives Clusterverfahren dar, welches darauf abzielt, den Verlust zu minimieren, der mit jeder Zusammenführung zweier Cluster

einhergeht. So werden in jedem Schritt die zwei Cluster vereinigt, welche zusammen die geringste Erhöhung der gesamten Distanz innerhalb des neuen Clusters generieren [35].

$$W(C_a, C_b) = \underbrace{\frac{w_a \cdot w_b}{w_a + w_b}}_{Kardinalität} \cdot \underbrace{\sum_{v=1}^{m} (c_{a,v} - c_{b,v})^2}_{Distanz} \tag{2.4}$$

Diese Erhöhung wird für zwei Cluster C_a und C_b mithilfe des Ward-Kriteriums (Formel 2.4) berechnet [24, 29, 34]. Während $w_{a/b}$ die jeweilige aktuelle Größe des Clusters (auch Gewicht genannt) beschreibt, stellen $c_{a,v}$ bzw. $c_{b,v}$ den mittleren Wert des jeweiligen Clusters in Dimension v dar, wobei jeder Vektor die Größe m hat. Ein kurzes Beispiel zur Berechnung des Ward-Kriteriums findet sich am Ende dieses Abschnitts.

Der Clusterprozess startet entsprechend der agglomerativen Herangehensweise mit der Zuweisung eines jeden der n Objekte zu einem eigenen Cluster, sodass n einelementige Mengen entstehen. Der nächste Schritt besteht in der Auswahl von zwei Teilmengen, welche zusammengeführt die geringste Beeinträchtigung des optimalen Wertes für die Distanzfunktion herbeiführen. Sie besitzen also, verglichen mit allen anderen möglichen Vereinigungen, den geringsten Wert des Ward-Kriteriums. Anschließend werden eben diese beiden gewählten Teilmengen vereinigt und die Anzahl der Cluster bzw. der Teilmengen wird um eins reduziert.

Danach werden die $n-1$ verbleibenden Teilmengen begutachtet, um herauszufinden, ob eine weitere, dritte Teilmenge mit den ersten beiden vereinigt werden sollte. Stattdessen kann auch ein neues Paar aus zwei anderen Teilmengen erstellt werden, sodass insgesamt der minimale Wert für $n-2$ Cluster erreicht wird. Das Verfahren kann solange fortgeführt werden, bis alle Teilmengen zu einem Cluster vereinigt wurden oder eine gewünschte Anzahl von k Clustern erreicht wurde.

Beispiel zur Berechnung von $W(C_a, C_b)$:
Sei $C_a = \{O_1, O_2, O_3\}$ und $C_b = \{O_4, O_5\}$ mit

$$O_1 = \begin{pmatrix} 6 \\ 6 \end{pmatrix}, \quad O_2 = \begin{pmatrix} 4 \\ 5 \end{pmatrix}, \quad O_3 = \begin{pmatrix} 5 \\ 4 \end{pmatrix}, \qquad O_4 = \begin{pmatrix} 2 \\ 8 \end{pmatrix}, \quad O_5 = \begin{pmatrix} 1 \\ 7 \end{pmatrix}$$

Dann ist $W(C_a, C_b) = 22,2$ wegen

$$\frac{3 \cdot 2}{3+2} \cdot \sum_{v=1}^{2} (c_{a,v} - c_{b,v})^2 = \frac{6}{5} \cdot [(\frac{6+4+5}{3} - \frac{2+1}{2})^2 + (\frac{6+5+4}{3} - \frac{8+7}{2})^2]$$

2.4 Distanzmaße

Neben der Wahl eines angemessenen Clusterverfahrens muss meistens auch ein entsprechendes Distanzmaß gewählt werden. Dieses entscheidet, wie der Abstand zwischen einzelnen Objekten gemessen wird und ist damit auch ausschlaggebend dafür, welche Objekte zu einem Cluster gehören und welche nicht [23]. Das kann über Ähnlichkeit, Unähnlichkeit oder ein anderes Maß geschehen.
Für eine beliebige Gruppe von Punkten bzw. Vektoren i, j und k erfüllt eine Distanz, ähnlich zur zuvor in Abschnitt 2.3.2 beschriebenen Baummetrik, die Eigenschaften der Symmetrie, positiven Definiertheit und der (hier nicht starken) Dreiecksungleichung ($d(i,j) \leq d(i,k) + d(k,j)$). Die Ähnlichkeit ist schließlich gegeben durch

$$s(i,j) = \max_{i,j} \{d(i,j)\} - d(i,j) \tag{2.5}$$

Falls die Dreiecksungleichung nicht berücksichtigt wird, so liegt eine Unähnlichkeit vor. Dabei hängt die Wahl des Distanzmaßes auch stark von der Art der vorliegenden Daten ab. Da die Merkmale in AMUSE exklusiv numerische Werte produzieren, sollen an dieser Stelle auch nur Distanzmaße für numerische Daten vorgestellt werden, gleichwohl z. B. RapidMiner-Clusterverfahren auch mit nominalen oder gemischten (numerischen und nominalen) Datentypen umgehen können.

$$d_p(x_a, x_b) = \left(\sum_{i=1}^{m} |x_{i,a} - x_{i,b}|^p \right)^{\frac{1}{p}} ; \ \forall \ p \in \mathbb{N} \setminus \{0\} \tag{2.6}$$

Zum Arbeiten in einem Vektorraum ist die Minkowski-Distanz (Formel 2.6) ein traditionelles und weit verbreitetes Distanzmaß [23, 31]. Die Anzahl der Dimensionen der Vektoren x_a und x_b werden durch m beschrieben. Üblicherweise praktisch verwendet werden jedoch meist die folgenden Spezialfälle der Minkowski-Distanz [20, 23, 31, 35]:

- Manhattan-Distanz

  Mit $p = 1$ stellt sie den Weg dar, der beim Reisen über ein achsenparalleles Gitter von einem Objekt zu einem anderen zurückgelegt würde. Es ist die Summe der Differenzen ihrer zugehörigen Komponenten bzw. der n Dimensionen.

  $$d_{\text{Manhattan}}(x_a, x_b) = \sum_{i=1}^{n} |x_{i,a} - x_{i,b}| \tag{2.7}$$

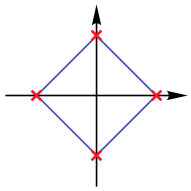

Abbildung 2.4: Minkowski-Distanz mit $p = 1$

- Euklidische-Distanz

  Als wohl am meisten gewähltes Maß mit $p = 2$ stellt es den Weg dar, den ein *Rabe beim Fliegen* von einem Objekt zu einem anderen zurückgelegen würde. Gebildet durch die Quadratwurzel der Summe der quadrierten Differenzen zwischen ihren zugehörigen Komponenten ist dieses Maß unter anderem bekannt dafür, von Außenseitern stark beeinflusst zu werden.

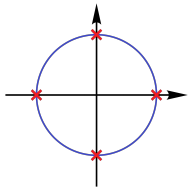

Abbildung 2.5: Minkowski-Distanz mit $p = 2$

$$d_{\text{Euklidisch}}(x_a, x_b) = \sqrt{\sum_{i=1}^{n}(x_{i,a} - x_{i,b})^2} \qquad (2.8)$$

- Maximums-Distanz

  Auch als *Chebychev-Distanz* bekannt stellt es mit $p \to \infty$ den maximalen Weg von einem Objekt zu einem anderen dar. Es berechnet die absolute Größe der Differenzen der Koordinaten von zwei Objekten.

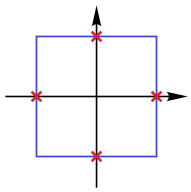

Abbildung 2.6: Minkowski-Distanz mit $p \to \infty$

$$d_{\text{Maximum}}(x_a, x_b) = \max_{i=1}^{n} |x_{i,a} - x_{i,b}| \qquad (2.9)$$

Bezüglich dieser drei Distanzmaße und mit k-Means als Verfahren bietet die Euklidische-Distanz die beste Performance [31], wohingegen die Manhattan-Distanz am schlechtesten abschneidet. Jedoch sind die bessere Robustheit gegenüber Außenseitern oder die einfache Skalierung auf höhere Dimensionen Gründe [26], um sich trotzdem für die Manhattan-Distanz als Maß zu entscheiden.

Insgesamt gibt es eine Vielzahl von verschiedenen Distanzmaßen die, wie auch die Clusterverfahren selbst, jeweils Vor- und Nachteile, besonders bestimmten Anwendungsgebieten gegenüber, besitzen. Alle über RapidMiner, und damit auch teilweise über AMUSE, zur Verfügung gestellten Maße können innerhalb der RapidMiner-Dokumentation [28] nachgeschlagen werden.

Kapitel 3

Implementierung

Dieses Kapitel umfasst den Hauptteil dieser Arbeit: Die Integration der gewählten (siehe Kapitel 2.3) Clusterverfahren in AMUSE. Während Abschnitt 2.1 AMUSE als Software aus Benutzersicht erläutert, wird in Abschnitt 3.1 der interne Aufbau beschrieben. Dazu gehören die Zusammenhänge einzelner Klassen beim Klassifizierungsprozess und alle für diese Arbeit notwendigen Änderungen. Danach werden die Einbindungen der Clusterverfahren k-Means, DBSCAN und SVC aus der RapidMiner-Bibliothek betrachtet (Abschnitt 3.2). Abschließend folgt die Implementierung von Wards Agglomeration innerhalb von AMUSE (Abschnitt 3.3). Alle Anpassungen können über `https://github.com/DiePaupi/AMUSE` eingesehen werden.

3.1 Anpassung von AMUSE

Um eine Übersicht der AMUSE-Hierarchie zu geben, wird folgend die Struktur des Dateisystems betrachtet. Über den Pfad `amuse/src/amuse/nodes/` wird der für diese Arbeit relevanteste Ordner *classifier* erreicht. In ihm befinden sich die Interfaces und Methoden der Klassifizierungsaufgabe.

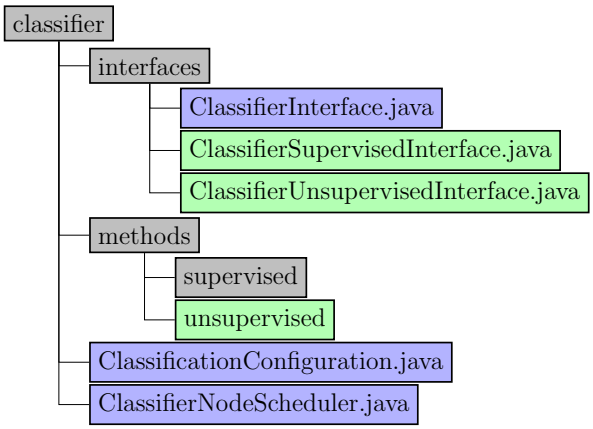

Abbildung 3.1: Ordnerstruktur der Klassifizierung

Die in Abbildung 3.1 dargestellten blauen Knoten zeigen bearbeitete, und die grünen neu erstellte Elemente. So wurde zunächst in `/classifier/methods/` neben dem bereits existierenden Ordner *supervised* ein weiterer Ordner *unsupervised* erstellt, welcher zukünftig die Adapterklassen der Clustermethoden umfassen soll. Die Adapterklassen stellen dabei die einzelnen Verfahren dar, die entweder einen RapidMiner-Prozess konfigurieren, aufrufen und die Ergebnisse speichern, oder die Klassifizierung autonom durchführen.
Das Interface der Klassifizierung *ClassifierInterface* wurde in zwei für überwachte (*ClassifierSupervisedInterface*) und unüberwachte Lernmethoden (*ClassifierUnsupervisedInterface*) spezialisierte Interfaces aufgespalten. Sie sind unter `/classifier/interfaces/` zu finden und erweitern das ursprüngliche Klassifizierungs-Interface. Diese Unterteilung war notwendig, da die Methode `classify(String path)`, die jede Adapterklasse nach dem Erben des Interfaces implementieren muss, in jedem Fall einen Pfad zum trainierten Modell als Parameter benötigte. Dieser Pfad kann bei unüberwachten Lernmethoden jedoch nicht mit angegeben werden, da ein solches Modell für die Clusterverfahren nicht existiert. Über das *ClassifierUnsupervisedInterface* implementieren die entsprechenden Klassen nun die Methode `classify()` ohne jegliche Parameter.
Innerhalb der Konfigurationsklasse *ClassificationConfiguration* war die Erstellung neuer Konstruktoren erforderlich. Die bisherigen Konstruktoren verlangten Parameter wie z. B. die Grundwahrheit, die für Clusterverfahren jedoch nicht zur Klassifizierung benötigt werden. So wurde für jeden Konstruktor ein weiterer, ohne die für das überwachte Lernen spezifischen Parameter, hinzugefügt.

Der *ClassifierNodeScheduler* erforderte mehrere Anpassungen, da dies die Klasse ist, die den Klassifizierungsprozess durchführt. Sie wird in Abbildung 3.2 ausführlicher betrachtet. Dort wird zunächst innerhalb von Schritt *i* eine passende *ClassificationConfiguration* erstellt, welche die Eingaben des Nutzers zur Klassifizierungsaufgabe enthält. In dem Zuge wird auch der *ClassifierNodeScheduler* selbst konfiguriert, indem bspw. die Interfaces deklariert werden. Eine Anpassung ist hier die Initialisierung eines Wahrheitswertes, der anzeigt, ob es sich um überwachte oder unüberwachte Klassifizierung handelt.
Der zweite Schritt (*ii*) besteht aus dem Vorbereiten der Eingaben der *ClassificationConfiguration*. Hier werden die Eingabedaten entsprechend zur Form, in der die Musikstücke und extrahierten Merkmale übergeben werden, für die weitere Verarbeitung umformatiert.

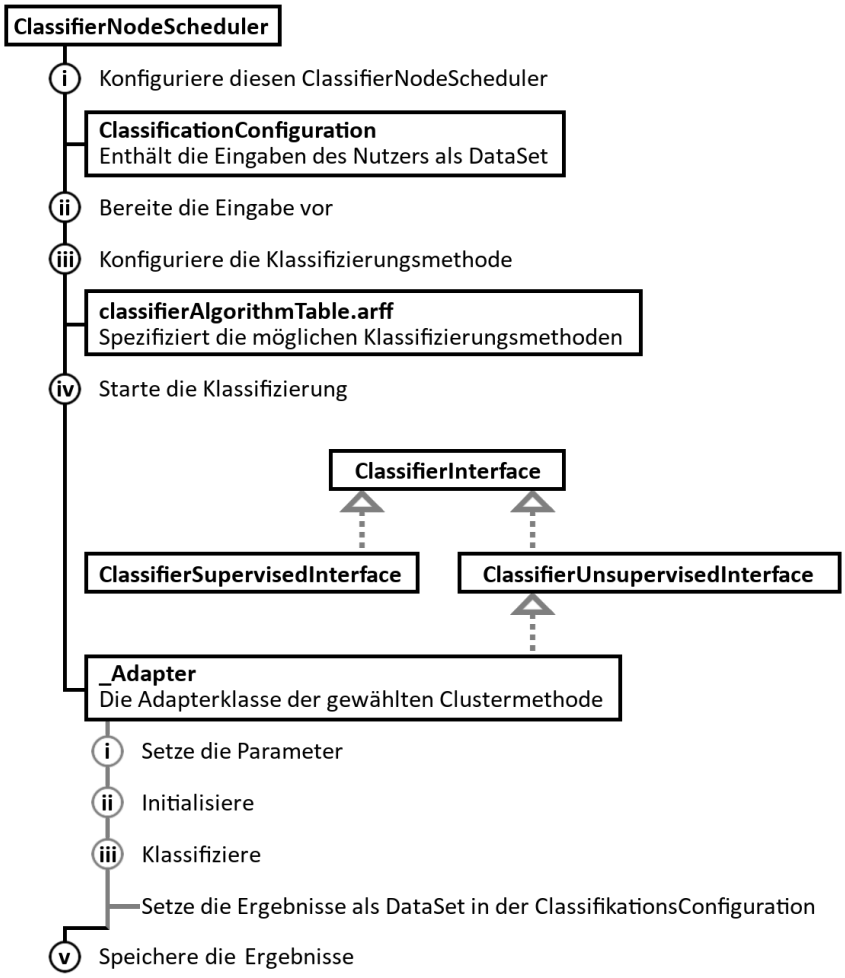

Abbildung 3.2: Klassifizierungsablauf innerhalb des `ClassifierNodeScheduler`

Um die gewählte Klassifizierungsmethode zu konfigurieren, wird in Schritt *iii* die *classifierAlgorithmTable.arff*-Datei verwendet. Diese umfasst Informationen zu allen Klassifizierungsmethoden und ihren Eigenschaften. Darunter fallen:

- Der Name des Verfahrens
- Die algorithmische Einordnung für den Nutzer (z. B. *Unsupervised>Partitional*)
- Den Pfad zur Adapterklasse
- Die benötigten Parameter mit ihren Namen, ihren Standardwerten und Beschreibungen
- Die unterstützten Typoptionen (z. B. ob unüberwachte Klassifizierung unterstützt wird)

Der *ClassifierNodeScheduler* prüft anhand dieser Informationen, ob der gewählte Algorithmus mit den vom Benutzer angegebenen Optionen kompatibel ist, bevor der nächste Schritt beginnt. Die vierte Sektion (*iv*) stellt die Klassifizierung dar, sodass über das zugehörige Interface die `classify(*)` Methode in der spezifizierten Adapterklasse aufgerufen wird. Auf deren Funktionalität wird in den folgenden Abschnitten 3.2 und 3.3 ausführlich eingegangen. Die Ergebnisse der Adapterklassen werden an deren Ende immer in der zugehörigen, in Schritt *i* erstellten, *ClassificationConfiguration* gesetzt.
Der *ClassifierNodeScheduler* greift schließlich in Schritt *v* auf diese Ergebnisse zu, um sie weiter zu verarbeiten und abschließend in einer ARFF-Datei abzuspeichern. Die Speicherung geschieht nun je nach Art der gewählten Klassifizierungsmethode, da es nicht nur zwischen den überwachten und unüberwachten Verfahren Unterschiede gibt, sondern auch zwischen den unüberwachten Methoden selbst. So muss zunächst im Anwendungsfall eines Clusterverfahrens die Anzahl der Cluster bestimmt werden, was für überwachte Lernmethoden nicht notwendig ist, da die Anzahl der Kategorien dort bereits zu Beginn bekannt ist. Für die unüberwachten Lernmethoden wurde die Hilfsmethode `updateTheNumberOfClusters (DataSet input)` angelegt, welche die Anzahl der Clusterattribute zählt und in der Variable der Kategorieanzahl speichert. Diese Variable beschreibt bei den überwachten Methoden die Anzahl der unterschiedlichen Kategorien und wird bei den Clusterverfahren analog für die Anzahl der unterschiedlichen Cluster verwendet.
Zusätzlich ist hier eine Unterscheidung von hierarchischen Clusterverfahren zu partitionierenden notwendig. AMUSE erwartet für jedes Musikstück einzelne, klassifizierte Partitionen, die auf Wunsch zusammengesetzt werden und so die wahrscheinliche Zugehörigkeit des gesamten Liedes für jedes Cluster ermitteln. Da hierarchische Methoden jedoch sukzessiv arbeiten, müssen diese Partitionen bereits vor dem Clustern zusammengesetzt werden. Zur Vereinigung bzw. Spaltung zweier Cluster müssen deren zugehörige Musikstücke bereits konkret feststehen. Dies bedeutet, dass AMUSE als Clusterergebnis hier keine Liste von einzelnen Partitionen erwartet, die noch zusammengesetzt werden müssen. Stattdessen stehen die fertig zugeordneten Musikstücke bereit, die auch entsprechend weiterverarbeitet und gespeichert werden können.

Neben der bereits bestehenden Speicherung der Klassifizierungsergebnisse in einer ARFF-Datei, erstellt AMUSE nun zusätzlich eine passende Visualisierung in Form einer LaTeX-Datei. Dort wird eine Tabelle mit den Clustern bzw. Kategorien als Spalten und den Musikstücken bzw. Partitionen, falls diese nicht zusammengefügt werden sollen, als Zeilen dargestellt. Ein Beispiel für eine solche Ausgabe sind die Ergebnistabellen der Clusterverfahren in Kapitel 4, wobei die Färbungen

nachträglich eingefügt wurden. Zur Erstellung der Tabelle ruft der *ClassifierNodeScheduler* die Methode `createVisual(...)` auf, in der zunächst der Dateikopf erstellt wird. Nach der Konstruktion eines angemessenen Tabellenkopfes müssen jedoch erst alle Namen der Musikstücke eines jeden Clusters und die maximale Clustergröße ermittelt werden, bevor die Tabelle befüllt werden kann. Da dieses umfangreiche Verfahren kein Hauptteil dieser Arbeit ist, sollte es hier nur kurz erwähnt werden und ist ausführlich in Anhang A beschrieben.

Um die Nutzbarkeit der Clusterverfahren innerhalb der Klassifizierung zu ermöglichen, musste deren Unterstützung durch eine Zeilenänderung im *ModelTypePanel* (zu finden unter `amuse/src/amuse/scheduler/gui/training/`) vermerkt werden, sodass sie innerhalb der GUI auswählbar wurden. Eine weitere Anpassung der Benutzeroberfläche war zwar für die Funktionalität nicht notwendig, jedoch hilft die Ausgrauung der Wahlfläche der Grundwahrheit, dem Benutzer im Falle der unüberwachten Lernmethoden die Übersicht zu behalten. Dies wird innerhalb der Klasse *ClassifierView* (unter `amuse/src/amuse/scheduler/gui/classifier/`) eingestellt. Weitere kleine Änderungen sind unter `amuse/src/amuse/data/` vorgenommen worden. Abschließend, nach dem Einbinden der Clusterverfahren, musste noch der *classifierAlgorithmTable* innerhalb `amuse/config/` um die entsprechenden Werte der neuen Methoden erweitert werden.

3.2 Einbindung von RapidMiner-Clusterverfahren in AMUSE

Die RapidMiner-Bibliothek stellt aktuell acht verschiedene Clusterverfahren zur Verfügung. Aus diesen wurden die klassischen Verfahren k-Means, DBSCAN und SVC ausgewählt, welche in Abschnitt 2.3.1 eingeführt wurden.
Zum Einbinden eines neuen Clusterverfahrens muss zunächst eine entsprechende Adapterklasse innerhalb von `/classifier/methods/unsupervised/` erstellt werden. Diese erweitert die abstrakte Klasse *AmuseTask* und implementiert das Interface für unüberwachte Methoden *ClassifierUnsupervisedInterface*. Die Struktur, zu sehen in Algorithmus 3.1, teilen alle Adapterklassen der unüberwachten Lernmethoden.

```
1  package amuse.nodes.classifier.methods.unsupervised;
2
3  import ...
4
5  public class xAdapter extends AmuseTask implements ClassifierUnsupervisedInterface {
6     ...
7     public void setParameters(String parameterString) throws NodeException { ... }
8     public void initialize() throws NodeException { ... }
9     public void classify() throws NodeException { ... }
10 }
```

Algorithmus 3.1: Struktur einer Adapterklasse

3.2.1 Die Parameter

Zu Beginn einer Adapterklasse müssen immer erst die, für das spezifische Verfahren benötigten, Parameter global deklariert werden. Danach können sie in der Methode `setParameters(String parameterString)` entsprechend der Benutzerangaben initialisiert werden. Welche Parameter notwendig sind und wie ihre Standardwerte aussehen, wird im weiteren Verlauf für jedes Verfahren einzeln mittels der RapidMiner-Dokumentation [28] genauer betrachtet. Am Ende der ersten Methode (bezogen auf Algorithmus 3.1) folgt abschließend noch eine Überprüfung der gesetzten Parameter, sodass es einen Fehler gibt, falls sich mindestens einer außerhalb seines gültigen Bereiches befindet.

k-Means

Das k-Means Verfahren benötigt einen Integer-Wert für k, welcher die Anzahl der Cluster vorgibt. Der Standardwert, welcher von RapidMiner vorgeschlagen wird, ist 5, wobei $2 \leq k$ gelten muss. Weiterhin können die Anzahl der Optimierungsschritte und die Anzahl der Durchläufe (Integer-Werte > 0) bestimmt werden. Sie haben die Standardwerte 100 und 10. Das Distanzmaß kann über ein *Drop-Down*-Menü von vordefinierten Strings gewählt werden. Aktuell stellt RapidMiner 13 verschiedene Distanzmaße für numerische Werte bereit, darunter fallen unter anderem die in Abschnitt 2.4 beschriebene Manhattan-, Euklidische- und Maximums-Distanz.
Zusätzlich gibt es auch die Möglichkeit, RapidMiner automatisch Startwerte für die initialen Cluster ermitteln zu lassen. Diese Startwerte können aber auch vom Nutzer selbst beeinflusst werden, indem die Verwendung eines lokalen Zufallswertes (*local random seed*) angeordnet wird. Dieser ist standardmäßig 1992, kann aber über einen Integer-Parameter (> 0) spezifiziert werden.

Density Based Spatial Clustering of Applications with Noise

DBSCAN benötigt drei verschiedene Parameter. Zum einen den Double-Wert für Epsilon (ε) mit $0,0 \leq \varepsilon$ und $1,0$ als Standardwert, welcher den Radius der Nachbarschaft bestimmt. Zum anderen den Integer-Wert für die minimale Anzahl an Punkten *MinPts*, die innerhalb von ε liegen müssen, um ein Cluster zu formen. *MinPts* hat standardmäßig den Wert 5 und es muss $1 \leq$ *MinPts* gelten. Der dritte Parameter ist das Distanzmaß, das genau wie bei k-Means über ein *Drop-Down*-Menü gewählt werden kann.

Support Vector Clustering

Die Parameterauswahl für SVC ist komplexer als bei den anderen Verfahren, da AMUSE aktuell noch keine dynamischen Parameterveränderungen unterstützt. Für ein ausgewähltes Verfahren sind daher alle Parameter fest vorgeschrieben, sodass nicht nach einer Auswahl eines bspw. ersten Parameters weitere hinzu kommen bzw. weg fallen können. Da SVC verschiedene Parameter je nach Wahl der Kernelfunktion benötigt, ist ein anderer Lösungsweg nötig, da ein simples Anzeigen aller Parameter nicht benutzerfreundlich scheint. So hat AMUSE nun zwei Adapterklassen für SVC: Eine, welche die Kernelfunktionen *dot*, *radial* und *polynomial* (kein bzw. ein Parameter) unterstützt, und eine weitere, welche die *neural*-Kernelfunktion (zwei Parameter) verwendet. Die Kernelfunktionen sind dabei wie folgt definiert:

$$radial: \; k(x,y) = \exp(-\gamma||x-y||^2) \tag{3.1}$$

$$polynomial: \; k(x,y) = (x \cdot y + 1)^d \tag{3.2}$$

$$dot: \; k(x,y) = x \cdot y \tag{3.3}$$

$$neural: \; k(x,y) = \tanh(a \cdot x \cdot y + b) \tag{3.4}$$

Die Standard-Kernelfunktion der ersten Adapterklasse ist *radial* (Formel 3.1), wobei dies ähnlich zu den vorher beschriebenen Verfahren im Bezug auf die Distanzmaße über ein *Drop-Down*-Menü verändert werden kann. Die radiale Funktion benötigt einen Double-Wert, der üblicherweise auch Gamma (γ) genannt wird und standardmäßig $1,0$ ist, wobei $0,0 \leq \gamma$ gelten muss. Die polynomiale Kernelfunktion (Formel 3.2) benötigt stattdessen einen Integer-Wert (≥ 0), der auch als Grad (d) bezeichnet wird und mit dem Wert 2 vordefiniert ist. Auf der anderen Seite benötigt die *dot*-Funktion (Formel 3.3) keinen weiteren Parameter, sodass das Eingabefeld des Kernelwertes in diesem Fall irrelevant ist. Dass dieses Parameterfeld bei der Auswahl der

dot-Funktion angezeigt wird ist nicht optimal, aber eine weitere Aufspaltung von SVC schien die Auswahl des Klassifizierungsverfahrens unübersichtlich zu machen. Dieser Umstand wird mit einer Beschreibung des Parameterfeldes klargestellt, was über die *classifierAlgorithmTable.arff* umgesetzt wird. Da das Parameterfeld des Kernelwertes von allen drei Kernelfunktionen geteilt wird, muss folglich innerhalb der Adapterklasse anhand des Namens der gewählten Kernelfunktion bestimmt werden, ob bzw. wie der Wert ausgelesen werden soll. Die zweite Adapterklasse, die nur die *neural*-Kernelfunktion (Formel 3.4) unterstützt, benötigt für die Funktionen entsprechend keine zusätzliche Unterscheidung. Dafür sind zwei beliebige Double-Werte als Parameter für Kernel a (standardmäßig $1,0$) und Kernel b (standardmäßig $0,0$) gefordert.
Die restlichen sieben Parameter sind, unabhängig der gewählten Kernelfunktionen, gleich. Dazu zählt zunächst die *Cache*-Größe in MB, die über einen Integer-Parameter (≥ 0), mit standardmäßig 200, definiert wird. Es folgt die Anzahl der Probepunkte, welche genutzt werden, um die Nachbarschaft zu überprüfen. Dies sind voreingestellt 20, aber es kann auf einen beliebigen, ganzzahligen Wert größer-gleich 0 abgeändert werden. Weiterhin gibt es einen Integer-Wert (≥ 0) für die Mindestanzahl der Punkte, welche ein Cluster ausmachen, der standardmäßig 2 beträgt. Der nächste Parameter ist ein Double-Wert ($\geq 0,0$) zur Optimierung der Präzision der Karush–Kuhn–Tucker-Konditionen [14], auch Konvergenz-Epsilon genannt, und beträgt hier üblicherweise $0,001$. Danach muss ein Integer-Wert (≥ 1) spezifiziert werden, der die maximale Anzahl der Iterationen vorschreibt und einen Standardwert von 100 000 hat. Der vorletzte Parameter wird p genannt und definiert über einen Double-Wert zwischen $0,0-1,0$ den Prozentanteil der erlaubten Außenseiter, wobei $0,0$ vordefiniert ist. Schließlich ist der letzte anzugebende Parameter r. Bei Angabe des Standardwertes $r = -1,0$ wird der im Laufe des Prozesses berechnete Radius verwendet. Falls dies unerwünscht ist, kann der Benutzer einen beliebigen anderen Double-Wert größer $0,0$ als Radius spezifizieren.

3.2.2 Vorbereiten und Durchführen des RapidMiner-Prozesses

In der Methode `initialize()` muss die RapidMiner-Bibliothek initialisiert werden, was über den internen *LibraryInitializer* mithilfe der Methode `initializeRapidMiner()` geschieht. Schließlich stellt die Methode `classify()` den konkreten Aufruf des RapidMiner-Prozesses, sowie die Umwandlung der Ergebnisse in ein von AMUSE weiter verwendbares Format dar. Dabei wird ersteres folgend und letzteres in Abschnitt 3.2.3 betrachtet.

Zu Beginn wird der Eingabedatensatz aus der zugehörigen Konfigurationsklasse, der *ClassificationConfiguration*, geladen, der die Attribute und einzelnen zugehörigen Werte jeder Partition aller zu clusternden Musikstücke enthält. Dieser Datensatz muss in das RapidMiner-kompatible Format eines Beispielsatzes (*ExampleSet*) umgewandelt werden, was die Methode `convertToRapidMinerExampleSet()` der Datensatz-Klasse übernimmt. Danach muss ein neuer RapidMiner-Prozess erstellt werden, was mithilfe der Klasse *Process* der RapidMiner-Bibliothek realisiert wird. Zusätzlich muss auch das konkrete Clusterverfahren als Operator definiert werden (*clusterer* in Algorithmus 3.2). Dies geschieht mittels des von RapidMiner bereitgestellten *OperatorService* über die Methode `createOperator(Class<T> clazz)`, wobei die entsprechende Verfahrensklasse aus der Bibliothek als Parameter übergeben wird. Dieser Operator muss auch die bereits ermittelten Parameter noch einmal intern setzen. So wird mithilfe der Methode `setParameter(String key, String value)` für jeden Parameter der, über die Verfahrensklasse vordefinierte, Schlüssel und der in dieser Adapterklasse zuvor gesetzte Parameterwert als String übergeben.

```
1 clusterer.setParameter(DistanceMeasures.PARAMETER_MEASURE_TYPES,
      DistanceMeasures.MEASURE_TYPES[DistanceMeasures.NUMERICAL_MEASURES_TYPE]);
2 clusterer.setParameter(DistanceMeasures.PARAMETER_NUMERICAL_MEASURE, measureType);
```

Algorithmus 3.2: Setzen des Distanzmaßes für RapidMiner-Verfahren

Das gewählte Distanzmaß (für die Verfahren k-Means und DBSCAN) wird analog zu den anderen Parametern gesetzt, jedoch ist zu beachten, dass zwei Variablen dazugehören, was mittels Algorithmus 3.2 dargestellt wird. Die von der Verfahrensklasse definierten Konstanten zum Ansteuern der Variablen (die Schlüssel) sind dabei blau hervorgehoben. Zuerst muss der Maßtyp (Zeile 1), der für die Anwendung über AMUSE immer numerisch ist, und danach das spezifische Distanzmaß (Zeile 2) gesetzt werden. Das Distanzmaß ist dabei für den Parameter des numerischen Maßtyps definiert, sodass dieser hier als Schlüssel dient. Damit ist der Operator fertig initialisiert und kann dem Prozess hinzugefügt werden, was mithilfe des Methodenaufrufs `getRootOperator().getSubprocess(0).addOperator(clusterer)` realisiert wird.

```
1 InputPort clustererInputPort = clusterer.getInputPorts().getPortByName("example
      set");
2 OutputPort clustererOutputPort =
      clusterer.getOutputPorts().getPortByName("clustered set");
3 InputPort processInputPort =
      process.getRootOperator().getSubprocess(0).getInnerSinks().getPortByIndex(0);
4 OutputPort processOutputPort =
      process.getRootOperator().getSubprocess(0).getInnerSources().getPortByIndex(0);
5
6 processOutputPort.connectTo(clustererInputPort);
7 clustererOutputPort.connectTo(processInputPort);
```

Algorithmus 3.3: Verbinden der Ports für RapidMiner-Verfahren

Bevor der Prozess jedoch gestartet werden kann, müssen die Schnittstellen verbunden werden. In Algorithmus 3.3 ist zu sehen, wie in den Zeilen 1 bis 4 die benötigten Schnittstellen angesteuert werden können. Auf den Eingabe- (*clustererInputPort*) und Ausgabeport (*clustererOutputPort*) des Clusteroperators kann dabei mit den definierten Namen zugegriffen werden. Für die Schnittstellen des Prozesses kann der Eingabeport (*processInputPort*) über `getInnerSinks()` und der Ausgabeport (*processOutputPort*) über `getInnerSources()` erreicht werden. Schließlich werden der Ausgabeport des Prozesses mit dem Eingabeport des Operators (Zeile 6) und der Ausgabeport des Operators mit dem Eingabeport des Prozesses (Zeile 7) verbunden. Der Prozess kann dann über die Zeile `IOContainer result = process.run(new IOContainer(exampleSet));` gestartet werden. So werden auch direkt die Ergebnisse bereitgestellt, wobei das *exampleSet* den zu Beginn umgewandelten Eingabedatensatz darstellt.

3.2.3 Umwandeln und Speichern der Ergebnisse

Um die Ergebnisse des RapidMiner-Prozesses aus dem Beispielsatz-Format in das AMUSE-interne Datensatz-Format zu bringen, auf dem weiter gearbeitet werden kann, sind mehrere Schritte notwendig. Zuerst müssen die Ergebnisse über `result.get(ExampleSet.class)` als konkreter Beispielsatz gespeichert werden, sodass danach ein neuer Datensatz direkt aus diesem Beispielsatz erstellt werden kann. Letzteres passiert über die Datensatz-Klasse, die einen entsprechenden Konstruktor bereitstellt.

RapidMiner hat jedoch im Gegensatz zum von AMUSE verwendeten Format zwei weitere Unterschiede, die bisher übernommen wurden. Zum einen wird beim Erstellen des Ergebnisses ein zusätzliches Attribut zur Identifikation der einzelnen Partitionen der Musikstücke in den Beispielsatz eingefügt. Zum anderen werden die spezifischen Clusterergebnisse innerhalb eines

Attributs *cluster* als String der Form *cluster_x* gespeichert. Diese beiden Punkte sind für die weitere Verarbeitung innerhalb von AMUSE suboptimal, da AMUSE ein eigenes Identifikationsattribut verwendet und pro Cluster ein eigenes, numerisches Attribut erwartet wird. Deswegen müssen diese Attribute auf dem bereits umgewandelten Datensatz gelöscht bzw. ersetzt werden. Die Datensatzklasse besitzt aber keine Methode zum Löschen von Attributen, sodass ein weiterer, leerer Datensatz erstellt wird, der die korrekten Attribute kopiert.
AMUSE handhabt die Clusterergebnisse mit einem Attribut pro Cluster, welches für jede Partition einen binären, numerischen Wert entsprechend der Zugehörigkeit enthält. Folglich müssen hier erst angemessen viele Clusterattribute erstellt werden, bevor sie anhand der Clusternummer x (aus der ursprünglichen Ergebnisform) befüllt werden können. Dies geschieht, indem zuerst alle Werte des ursprünglichen Attributes *cluster* durchlaufen werden und der höchste Wert für x ausfindig gemacht wird. Gleichzeitig wird im Falle von SVC geprüft, ob auch Rauschen entdeckt wurde. In diesem Fall wird in der *ClassificationConfiguration* eine entsprechende Variable gesetzt, die dies für eine spätere Ausgabe (in Anhang A beschrieben) vormerkt.
Innerhalb einer Schleife über die bekannte Clusteranzahl wird pro Iteration ein neues Clusterattribut *cluster_y* erstellt. Dabei stellt y die Nummer der aktuellen Iteration dar. Mithilfe der Schleife werden alle Partitionen durchlaufen und es wird jeweils die Gleichheit von x und y überprüft. Im Falle einer Gleichheit wird der Wert 1 und ansonsten der Wert 0 für die entsprechende Partition innerhalb des aktuellen Attributs (bzw. Clusters) gespeichert. Für den Spezialfall, dass einige Musikstücke keinem Cluster angehören, sondern Rauschen darstellen, wird dies ähnlich zu einem normalen Cluster behandelt. Es wird ein zusätzliches Attribut mit dem Namen *noise* erstellt, das ebenfalls für alle Partitionen die Werte 1 oder 0 als Indikator der Zugehörigkeit enthält. Am Ende jeder Iteration wird das fertige Clusterattribut dem Ergebnisdatensatz angefügt und sobald die Schleife beendet ist, kann der Ergebnisdatensatz in der *ClassificationConfiguration* als finales Ergebnis gesetzt werden.

3.3 Einbindung eines autonomen Clusterverfahrens in AMUSE

Für das autonome zu integrierende Verfahren ist die Wahl auf Wards Agglomeration gefallen. Dieses Verfahren grenzt sich dabei von den drei partitionierenden Verfahren des vorherigen Abschnitts ab, da es zu den hierarchischen Methoden zählt. Üblicherweise werden für hierarchische Clusterverfahren die drei bekannten *Linkage*-Algorithmen verwendet [20], die eine sehr simple und effektive Lösung anbieten, aber bereits über RapidMiner zur Verfügung stehen. Wards

Agglomeration stellt in diesem Kontext ein weiteres weit verbreitetes Clusterverfahren dar [24], welches den Anspruch, nicht in RapidMiner integriert zu sein, erfüllt.
Die Struktur dieser Adapterklasse ist die gleiche wie in Algorithmus 3.1. Denn alle Adapterklassen der unüberwachten Lernmethoden erweitern die Klasse *AmuseTask* und implementieren das Interface der unüberwachten Klassifizierung. Während die Methode `setParameters(...)` die für Wards Agglomeration angepassten Parameter enthält, die folgend betrachtet werden, ist die Methode `initialize()` leer, da nichts initialisiert werden muss.
Zu den Parametern gehört einerseits die Wahl der Methode zur Berechnung der Distanzen vor jeder Vereinigung zweier Cluster. Der Nutzer hat die Auswahl aus der iterativen, klassischen Methode [34] und der rekursiven *Lance-Williams Dissimilarity Update Formula* [16]. Beide Ansätze werden in Abschnitt 3.3.2 beschrieben. Weiterhin wird eine Angabe für k benötigt, was wie gewohnt die gewünschte Clusteranzahl angibt. Die Standardwerte sind dabei die klassische Methode sowie $k = 0$. Letzteres gibt an, dass alle Musikstücke zu einem umfassenden Cluster vereinigt werden sollen. Abweichend kann für k aber auch ein beliebiger anderer Integer-Wert zwischen 1 und der Anzahl der zu clusternden Musikstücke bzw. Partitionen angegeben werden.

3.3.1 Vorbereitung der Eingabedaten

Die Hauptmethode `classifiy()` beginnt mit dem Zugriff auf den Eingabedatensatz über die *ClassificationConfiguration*. Mithilfe dieses Datensatzes und der zugehörigen Liste von Lied-Partitions-Beschreibungen wird ein zweidimensionaler Array (`double[][] allValues`) erstellt. Die Lied-Partitions-Beschreibungen enthalten Angaben über die Anzahl der zu klassifizierenden Musikstücke, sowie aller zugehörigen Partitionen. Der Array *allValues* hat die Größe $i \times f$, wobei i die Anzahl aller Partitionen bzw. der Musikstücke darstellt, und f die Anzahl der Merkmale. Falls nur ein einzelnes Musikstück klassifiziert werden soll, werden die Partitionen beibehalten und jede wird im weiteren Verlauf als eigenständiges Musikstück angesehen (Anzahl aller Partitionen $\times$ Anzahl der Merkmale). Andernfalls wird für jedes Musikstück pro Merkmal der Durchschnittswert aller Partitionen gebildet (Anzahl aller Musikstücke $\times$ Anzahl der Merkmale). In dem Fall wird auch in der *ClassificationConfiguration* die Variable `boolean partitionsAlreadySummerized` auf `true` gesetzt. Diese Variable teilt dem *ClassifierNodeScheduler* mit, dass die Musikstücke bereits zusammengefasste Partition haben.
Im Gegensatz zu den anderen Adapterklassen, welche immer die einzelnen Werte jeder Partition aller Lieder behalten, werden diese hier zusammengeführt, falls es mehr als ein Musikstück

zum Klassifizieren gibt. Dieses Vorgehen ist notwendig, da AMUSE normalerweise jede der Partitionen einzeln clustert und das Ergebnis eines gesamten Liedes, falls erwünscht, erst nach der Rückgabe dieser einzelnen Zwischenergebnisse an die *ClassificationConfiguration* ermittelt. Da jedoch in einem hierarchischen Clusterverfahren die Cluster sukzessive über eine Schleife bzw. rekursiv ermittelt werden, kann eine Bestimmung der Clusterzugehörigkeit nicht erst abschließend außerhalb der Adapterklasse stattfinden.
Zur Befüllung von *allValues* wird gleichzeitig eine Liste aus Tupeln erstellt (`List<idAndName> songIdsAndNames`), welche jeweils die Kombination aus einer Lied-Identifikationsnummer (kurz Lied-ID) und des Lied-Namens enthält. Die Zuweisung jedes Musikstückes zu einem eigenen Cluster, entsprechend des in Abschnitt 2.3.2 beschrieben Vorgehens, wird durch eine Liste von Integer-Listen (`List<List<Integer>> clusterAffiliation`) erreicht. Jede Integer-Liste repräsentiert ein eigenes Cluster, in dem die diesem Cluster zugehörigen Musikstücke über ihre Lied-IDs gespeichert werden.

```
1 public Dendrogramm (List<List<Integer>> clusterInput, List<idAndName>
      songIdsAndNames) {
2    clusters = new ArrayList<Node>();
3    for (int clusterNumber=0; clusterNumber < clusterInput.size(); clusterNumber++) {
4       Node current = new Node(""+clusterNumber,
            songIdsAndNames.get(clusterNumber).getName(),
            clusterInput.get(clusterNumber), null, null);
5       clusters.add(clusterNumber, current);
6    }
7 }
```

Algorithmus 3.4: Konstruktor der Dendrogrammklasse

Mithilfe der *clusterAffiliation* und der *songIdsAndNames*-Listen wird dann ein initiales Dendrogramm erzeugt. Dieses beschreibt einen binären Baum, der die Vereinigungen der Cluster repräsentiert, indem genau zwei Kinderknoten (einzelne Cluster) zu einem Elternknoten (vereinigtem Cluster) führen. Innerhalb Algorithmus 3.4 ist der Konstruktor der Dendrogrammklasse dargestellt. Dort wird zunächst für jedes einzelne, an dieser Stelle einelementige, Cluster ein neuer Blattknoten erstellt und in einer globalen Liste gespeichert. Die Idee der Knoten-Liste ist es, alle aktuellen Cluster in Form eines eigenen Teil-Dendrogramms anhand ihrer Wurzelknoten zu repräsentieren. Nach einer Vereinigung zweier Cluster wird ein entsprechender neuer Elternknoten erzeugt und in die Knoten-Liste eingefügt, während die Kinderknoten aus der Liste entfernt werden.

Die Knoten selbst werden dabei über eine interne Klasse realisiert und besitzen mehrere Parameter:

String id dient der Identifikation. Er besteht initial aus der Position innerhalb der Knoten-Liste *clusters* und setzt sich im weiteren Verlauf aus den ID-Strings der beiden vereinigten Cluster (getrennt durch ein + und mit einem ; als Abschluss) zusammen, um die Reihenfolge der Vereinigungen später nachvollziehen zu können.

String name ist eine Angabe aller Pfade (durch Zeilenumbrüche getrennt) der dem, durch diesen Knoten repräsentierten, Cluster zugehörigen Musikstücke.

List<Integer> value speichert analog zu den Integer-Listen in *clusterAffiliation* die dem Cluster zugehörigen Musikstücke über ihre Lied-ID.

Node parent gibt den Elternknoten an. Er ist initial leer und wird erst bei einer Vereinigung dieses Knotens (Clusters) mit einem anderen gesetzt.

Node left gibt den linken Kinderknoten an. Er ist für Blattknoten leer und wird bei Vereinigungen zweier Cluster innerhalb des dort entstehenden, gemeinsamen Knotens dem ersten der beiden beteiligten Cluster zugewiesen.

Node right gibt den rechten Kinderknoten an. Er ist für Blattknoten leer und wird bei Vereinigungen zweier Cluster innerhalb des dort entstehenden, gemeinsamen Knotens dem zweiten der beiden beteiligten Cluster zugewiesen.

Nachdem *allValues* befüllt, *clusterAffiliation* erstellt und ein initiales Dendrogramm erzeugt wurde, ist alles fertig vorbereitet und der praktische Clusterprozess kann starten.

3.3.2 Der Vereinigungsprozess

Um die Clustervereinigungen nacheinander zu ermitteln gibt es zwei Ansätze, die jedoch beide auf einer Unähnlichkeits-Matrix basieren, anhand derer die nächstbeste Zusammenführung festgestellt werden kann. Der iterative Ansatz (die klassische Methode) berechnet innerhalb einer Schleife bei jedem Durchlauf eine neue Unähnlichkeits-Matrix. Bei dem rekursiven Ansatz (die *Lance-Williams Dissimilarity Update Formula*) wird hingegen die initiale Matrix mit jedem Aufruf aktualisiert. Dabei gilt für jede Iteration bzw. jeden rekursiven Aufruf, dass nach der Ermittlung der besten Vereinigung, diese über die *clusterAffiliation* und das Dendrogramm realisiert werden muss. Im weiteren Verlauf des Abschnittes werden zunächst die Funktionsweisen der beiden Methoden und dann die Vereinigung an sich genauer betrachtet.

Die klassische Methode

```
1 int mergeUntilThisClusterNumber;
2 if (k == 0 || k == 1) { mergeUntilThisClusterNumber = 2; }
3 else { mergeUntilThisClusterNumber = k; }
4
5 while (clusterAffiliation.size() > mergeUntilThisClusterNumber) { ... }
6 if (clusterAffiliation.size() == 2 && (k == 0 || k == 1)) { ... }
```

Algorithmus 3.5: While-Schleife im WardAdapter zur Vereinigung von Clustern

Die Zusammenführungen der einzelnen Cluster werden innerhalb der klassischen Methode schrittweise mittels einer *While*-Schleife realisiert (Algorithmus 3.5). Diese wird so lange ausgeführt, wie die Anzahl der aktuellen Cluster noch größer als die Anzahl der gewünschten Cluster ist. Die Variable zur Indikation der Clusteranzahl, bis zu der vereinigt werden soll (*mergeUntilThisClusterNumber*), ist für eine komplette Vereinigung ($k = 0$ oder $k = 1$) zwei (Zeile 2). Die Zusammenführung der letzten beiden Cluster kann dann nach der Schleife ohne jegliche Berechnungen geschehen (Zeile 6). Für spezifische Angaben von k wird *mergeUntilThisClusterNumber* auf genau dieses gesetzt (Zeile 3).

Der erste Schritt in jedem Durchlauf ist das Berechnen der neuen Unähnlichkeits-Matrix (`double[][] dissimilarityMatrix`) mithilfe der Methode `calculateDissimilarityMatrix()`. Diese Matrix nimmt in Höhe und Breite jeweils die aktuelle Anzahl der Cluster an. Jedes Element der Matrix repräsentiert die Vereinigung der zwei Cluster der entsprechenden Spalte und Zeile. Die Elemente der Diagonale können folglich mit dem Wert $0,0$ befüllt werden, da sich dort immer dasselbe Cluster trifft. Die *dissimilarityMatrix* ist auch symmetrisch, sodass aus Effizienzgründen die Felder, die rechts der Diagonalen liegen, nicht extra berechnet werden. Stattdessen werden sie einfach auf die zugehörigen Felder mit vertauschtem Spalten- und Zeilenindex links der Diagonalen gesetzt.

```
1     private double calculateWardsCriterion (double[] centroidA, double sizeA,
          double[] centroidB, double sizeB) {
2           double cardinality = (sizeA * sizeB) / (sizeA + sizeB);
3           double distance = this.squaredEuclideanDistance(centroidA, centroidB);
4           return cardinality * distance;
5     }
```

Algorithmus 3.6: Methode des *WardAdapter* `calculateWardsCriterion(...)`

Zur Berechnung eines einzelnen Unähnlichkeitswertes in einem Element der Matrix werden zunächst die Schwerpunkte (auch Mittelpunkte oder *Centroids* genannt), der zwei betrachteten Cluster ermittelt. Ein Schwerpunkt ist jeweils ein Array mit der Länge der Merkmalsanzahl, welcher für jedes Merkmal den Durchschnittswert aus allen Musikstücken des entsprechenden Clusters enthält. Die klassische Methode nutzt in jeder Iteration das in Abschnitt 2.3.2 beschriebene Ward-Kriterium zur Berechnung des Unähnlichkeitswertes, welches in Formel 2.4 dargestellt wird. Der Multiplikator (Kardinalität bzw. *cardinality*) setzt sich aus den Größen der Cluster zusammen. Sie werden in Algorithmus 3.6 durch die Parameter `double sizeA` und `double sizeB` repräsentiert. Die Berechnung der Kardinalität findet dabei in Zeile 2 satt. Die Parameter der Clustergrößen müssen an dieser Stelle als Double-Werte übergeben werden, obwohl es sich um ganzzahlige Werte handelt, da das Ergebnis der Berechnung sonst automatisch gerundet wird.

```
1     private double squaredEuclideanDistance (double[] centroidA, double[]
          centroidB) {
2        double distance = 0.0;
3        for(int featureNumber=0; featureNumber<centroidA.length; featureNumber++) {
4           double difference = centroidA[featureNumber] - centroidB[featureNumber];
5           distance += difference * difference;
6        }
7        return distance;
8     }
```

Algorithmus 3.7: Methode des *WardAdapter* `squaredEuclideanDistance(...)`

Der Multiplikand (Distanz bzw. *distance*) ergibt sich durch das definierte Distanzmaß. Dies ist für das Ward-Kriterium die quadrierte Euklidische-Distanz (zu sehen in Algorithmus 3.7). Dabei werden die beiden Schwerpunkte der Cluster durchlaufen und es wird eine Summe (`double distance`) gebildet. In dieser Summe werden für jedes Merkmal zunächst die Werte der Clusterrepräsentanten voneinander abgezogen (Zeile 4), bevor dieses Zwischenergebnis quadriert und anschließend aufsummiert wird (Zeile 5). Der gesamte Produktwert wird schließlich als Unähnlichkeitswert der beiden betrachteten Cluster im entsprechenden Feld der *dissimilarity-Matrix* vermerkt und es wird mit der Berechnung des Wertes der nächsten beiden Cluster fortgefahren.

Ist die Unähnlichkeits-Matrix befüllt, wird die später in den Algorithmen 3.9 und 3.10 genauer beschriebene Vereinigung zwischen den beiden Clustern, die innerhalb der Matrix an dem minimalen Wert beteiligt sind, gesetzt. Danach beginnt der zweite Teil des Vereinigungs-Durchlaufes, falls

die gewünschte Clusteranzahl noch nicht erreicht wurde. So werden für das neue Cluster $A \cup B$ die Unähnlichkeiten zu den verbleibenden $n-1$ ($n-2$ ohne $A \cup B$ selbst) Clustern berechnet. Analog zur Unähnlichkeits-Matrix werden diese Ergebnisse in einem Array gespeichert. Gesucht ist nun der kleinste Wert des Arrays und der zweitkleinste Wert der Unähnlichkeits-Matrix. Falls der Arraywert kleiner als der Matrixwert ist, wird das entsprechende (für diesen kleinsten Wert mitverantwortliche) Cluster C mit dem Cluster $A \cup B$ vereinigt. Weiterhin wird überprüft, ob nur noch zwei Cluster übrig sind und ein allumfassendes Cluster gewünscht ist. Diese würden an dieser Stelle direkt vereinigt werden (Zeile 6 in Algorithmus 3.5). Danach wird wieder der Kopf der While-Schleife aufgerufen, in der die initiale Bedingung, ob die gewünschte Clusteranzahl noch nicht erreicht wurde, erneut geprüft und entsprechend eine neue Iteration gestartet wird.

Die Lance-Williams Dissimilarity Update Formula

Im Gegensatz zur klassischen Methode bietet die *Lance-Williams Dissimilarity Update Formula* (kurz LWDUF) [16] einen rekursiven Ansatz, indem die initiale Unähnlichkeits-Matrix nach jeder Vereinigung aktualisiert, anstatt ganz neu berechnet wird. So können die Werte der Cluster, die nicht an der Vereinigung beteiligt sind, übernommen werden, da sie sich nicht verändert haben. Alle Werte der beteiligten Cluster werden jedoch mithilfe der LWDUF überarbeitet.

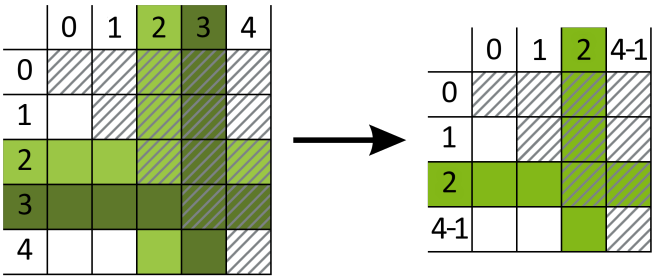

Abbildung 3.3: LWDUF Unähnlichkeits-Matrix Aktualisierung

Die in Abbildung 3.3 dargestellten Matrizen veranschaulichen eine Unähnlichkeits-Matrix vor und nach der Aktualisierung durch die Vereinigung der Cluster 2 und 3. So werden die beiden, an der Vereinigung beteiligten, Cluster der linken Matrix im Cluster mit dem niedrigeren Index der rechten Matrix zusammengefasst. Das grüne Kreuz stellt dabei die Felder da, die mittels der LWDUF aktualisiert werden müssen, wohingegen die weißen Felder übernommen werden können. Zu beachten sind die verschobenen Cluster-Indizes, sodass bspw. das Cluster 4 der linken Matrix

in der rechten zu Cluster 3 $(4-1)$ wird. Das ursprüngliche Cluster 3 der linken Matrix existiert in der rechten nicht mehr.

$$d_{\text{LWDUF}}(A \cup B, C) \;=\; \alpha(A) \cdot d(A,C) + \alpha(B) \cdot d(B,C) + \beta \cdot d(A,B) + \gamma \cdot |d(A,C) - d(B,C)| \quad (3.5)$$

Das Ward-Kriterium wird in dieser Methode nur zur initialen Berechnung der Unähnlichkeits-Matrix verwendet. Nebenbei können durch die LWDUF auch weitere agglomerative Clusterverfahren durch die Veränderung der vier Koeffizienten umgesetzt werden [29]. Dafür wird allgemein Formel 3.5 genutzt, wobei $\alpha(A)$, $\alpha(B)$, β und γ das agglomerative Kriterium definieren. Eine Übersicht der Werte der Koeffizienten für eine Auswahl gängiger Verfahren findet sich in Tabelle 3.1 [29, 2].

	„Lance-Williams Dissimilarity Update Formula“			
Clusterverfahren	$\alpha(A)$	$\alpha(B)$	β	γ
Single Link	$1/2$	$1/2$	0	$-1/2$
Average Link	$\frac{\lvert A\rvert}{\lvert A\rvert+\lvert B\rvert}$	$\frac{\lvert B\rvert}{\lvert A\rvert+\lvert B\rvert}$	0	0
Complete Link	$1/2$	$1/2$	0	$1/2$
Centroid	$\frac{\lvert A\rvert}{\lvert A\rvert+\lvert B\rvert}$	$\frac{\lvert B\rvert}{\lvert A\rvert+\lvert B\rvert}$	$-\frac{\lvert A\rvert\lvert B\rvert}{(\lvert A\rvert+\lvert B\rvert)^2}$	0
Median	$1/2$	$1/2$	$-1/4$	0
Ward	$\frac{\lvert A\rvert+\lvert C\rvert}{\lvert A\rvert+\lvert B\rvert+\lvert C\rvert}$	$\frac{\lvert B\rvert+\lvert C\rvert}{\lvert A\rvert+\lvert B\rvert+\lvert C\rvert}$	$-\frac{\lvert C\rvert}{\lvert A\rvert+\lvert B\rvert+\lvert C\rvert}$	0

Tabelle 3.1: LWDUF Koeffizientenübersicht

Die initiale Erstellung der *dissimilarityMatrix* läuft analog zum bereits beschriebenen Vorgehen in der klassischen Methode ab. Danach wird die rekursive Hilfsmethode `updateDissimilarityMatrix (double[][] dissimilarityMatrix)` aufgerufen, die als Parameter die zu aktualisierende Unähnlichkeits-Matrix erhält. Sie hat zu Beginn einen Rekursionsanker, der überprüft, ob die gewünschte Clusteranzahl bereits erreicht wurde, oder nur noch zwei Cluster verbleiben, die direkt vereinigt werden können, falls ein umfassendes Cluster gewünscht ist. Ansonsten wird das Minimum der Matrix ermittelt und die entsprechenden Cluster werden analog zur klassischen

Methode (mit den Algorithmen 3.9 und 3.10) vereinigt. An dieser Stelle beginnt die Aktualisierung der Unähnlichkeits-Matrix.

Dazu wird eine leere Matrix erstellt, welche eine Spalte und eine Zeile weniger im Vergleich zur übergebenen *dissimilarityMatrix* besitzt. Die neue Matrix wird mittels zwei *for*-Schleifen durchlaufen, wobei mehrere Fälle unterschieden werden. Zunächst wird, falls Spalten- und Zeilenindex übereinstimmen, der Wert auf $0,0$ gesetzt, da es sich um ein Diagonalfeld handelt. Falls es sich um ein Feld rechts der Diagonalen handelt, wird der Wert auf den des zugehörigen Feldes links der Diagonalen gesetzt. Ansonsten wird geprüft, ob eines der dem Feld zugehörigen Cluster an der Vereinigung beteiligt war. Wenn dies nicht der Fall ist, kann der Wert aus der übergebenen *dissimilarityMatrix* übernommen werden, da sich an den Clustern, für die dieser Wert berechnet wurde, nichts verändert hat. Falls jedoch eines der beiden, für dieses Feld relevanten, Cluster an der Zusammenführung beteiligt war, findet eine Überarbeitung des originalen Wertes statt. So wird genau das andere Cluster, welches nicht an der Vereinigung beteiligt war, innerhalb der LWDUF als drittes Cluster C verwendet. Der Unähnlichkeits-Wert wird gemäß Formel 3.5, mithilfe der für das Ward-Kriterium angepassten Koeffizienten, aktualisiert.

$$d_{\text{Ward}}(A \cup B, C) = \underbrace{\frac{|A|+|C|}{|A|+|B|+|C|} \cdot d(A,C)}_{alphaFirst} + \underbrace{\frac{|B|+|C|}{|A|+|B|+|C|} \cdot d(B,C)}_{alphaSecond} - \underbrace{\frac{|C|}{|A|+|B|+|C|} \cdot d(A,B)}_{beta} \quad (3.6)$$

```
1  private double calculateLWDissimilarity (double[] centroidA, double sizeA, double[]
       centroidB, double sizeB, double[] centroidC, double sizeC) {
2     double alphaFirst = (sizeA + sizeC) / (sizeA + sizeB + sizeC);
3     alphaFirst = alphaFirst * this.squaredEuclideanDistance(centroidA, centroidC);
4
5     double alphaSecond = (sizeB + sizeC) / (sizeA + sizeB + sizeC);
6     alphaSecond = alphaSecond * this.squaredEuclideanDistance(centroidB, centroidC);
7
8     double beta = (sizeC) / (sizeA + sizeB + sizeC);
9     beta = beta * this.squaredEuclideanDistance(centroidA, centroidB);
10
11    return alphaFirst + alphaSecond - beta;
12 }
```

Algorithmus 3.8: Methode des *WardAdapter* `calculateLWDissimilarity(...)`

Die Anpassungen der LWDUF für das Ward-Kriterium sind in Formel 3.6 zu sehen. Diese Formel wird in Algorithmus 3.8 mit den Clustern A und B, als die beiden vereinigten Cluster, und Cluster C, als das weitere betroffene Cluster, umgesetzt. Wurde jedes Feld einmal betrachtet

und gegebenenfalls aktualisiert, startet die Methode schließlich den rekursiven Aufruf mit der aktualisierten Unähnlichkeits-Matrix als Parameter. Ähnlich zur klassischen Methode werden dort wieder die Bedingungen des Rekursionsankers geprüft und entsprechend eine neue Rekursion durchgeführt.

Die Vereinigung

In beiden Ansätzen muss für jede Iteration bzw. jeden rekursiven Aufruf nach der Ermittlung der besten Vereinigung, anhand des minimalen Wertes der Unähnlichkeits-Matrix, diese Vereinigung realisiert werden. Die Methode `setMerge(...)`, zu sehen in Algorithmus 3.9, setzt die Vereinigung über die *clusterAffiliation* und das Dendrogramm um. Die Cluster A und B, die dem minimalen Wert zugehörig sind, werden zuerst in einer neuen Integer-Liste vereinigt (Zeilen 2−4) und als neues Cluster $A \cup B$ in *clusterAffiliation* hinzugefügt (Zeile 12). Danach können die beiden einzelnen Cluster A und B aus dieser Liste entfernt werden (Zeilen 13 und 14). Gleichzeitig werden diese drei Cluster (A, B, $A \cup B$) auch an das Dendrogramm übergeben (Zeile 7), welches die Zusammenführung intern mit der Methode `setNewMerge(...)` setzt.

```
1  private List<Integer> setMerge (List<Integer> clusterA, List<Integer> clusterB) {
2     List<Integer> mergedCluster = new ArrayList<Integer>();
3     mergedCluster.addAll(clusterA);
4     mergedCluster.addAll(clusterB);
5
6     try {
7        dendo.setNewMerge(clusterA, clusterB, mergedCluster);
8     } catch (Exception e) {
9        AmuseLogger.write("WardAdapter", Level.WARN, "The dendrogram coudn't set a
             new merge: " + e.getMessage());
10    }
11
12    clusterAffiliation.add(mergedCluster);
13    clusterAffiliation.remove(clusterA);
14    clusterAffiliation.remove(clusterB);
15    return mergedCluster;
16 }
```

Algorithmus 3.9: Methode des *WardAdapter* `setMerge(...)`

In Algorithmus 3.10 der Dendrogrammklasse werden so zunächst in Zeile 3 die Eingabeparameter auf Gültigkeit überprüft. Danach werden in Zeile 7 zwei Knotenreferenzen erstellt, die, falls die übergebenen Cluster in der internen Knoten-Liste (*clusters*) gefunden werden können, auf die zugehörigen Knoten gesetzt werden (Zeilen 8−14). Ob dies erfolgreich war, wird in Zeile

16 überprüft, sodass danach mit der eigentlichen Vereinigung der Cluster-Knoten fortgefahren werden kann.

```
1  public void setNewMerge (List<Integer> inputClusterA, List<Integer> inputClusterB,
       List<Integer> inputMergedCluster) throws NodeException {
2
3     if (inputClusterA.equals(null) || inputClusterB.equals(null) ||
          inputMergedCluster.equals(null)) {
4        throw new NodeException ("Dendrogram - setNewMerge(): At least one of the
             given input cluster lists was null.");
5     }
6
7     Node clusterA = null, clusterB = null;
8     for (int c=0; c < clusters.size(); c++) {
9        if (clusters.get(c).getValue().equals(inputClusterA)) {
10          clusterA = clusters.get(c);
11       } else if (clusters.get(c).getValue().equals(inputClusterB)) {
12          clusterB = clusters.get(c);
13       }
14    }
15
16    if (clusterA.equals(null) || clusterB.equals(null)) {
17       throw new NodeException ("Dendrogram - setNewMerge(): Couldn't find the
             clusters to merge in the dendrograms own cluster list.");
18    } else {
19
20       List<Integer> parentValue = new ArrayList<Integer>();
21       parentValue.addAll(inputMergedCluster);
22
23       String parentID = clusterA.getID() + "+" + clusterB.getID() +";";
24       String parentName = clusterA.getName() + "\n" + clusterB.getName();
25       Node parent = new Node(parentID, parentName, parentValue, clusterA, clusterB);
26       clusterA.setParent(parent);
27       clusterB.setParent(parent);
28
29       clusters.add(parent);
30       clusters.remove(clusterA);
31       clusters.remove(clusterB);
32    }
33 }
```

Algorithmus 3.10: Methode der Dendrogrammklasse `setNewMerge(...)`

Dazu wird eine neue Integer-Liste erstellt (Zeile 20), die später für den Cluster-Knoten der Vereinigung als *value*-Parameter dienen soll. Deswegen übernimmt diese Liste alle Werte des vereinigten Clusters $A \cup B$ (Zeile 21). Danach wird eine passende ID für den *id*-Parameter, entsprechend des bereits beschriebenen Schemas, erstellt (Zeile 23). Der String für den *name*-Parameter setzt sich aus denen der Knoten A und B zusammen, indem diese durch einen

Zeilenumbruch aneinander gereiht werden (Zeile 24). Bei der Erstellung des vereinigten Knotens in Zeile 25 werden dann die zuvor besprochenen Parameter übergeben, wobei die Kinder entsprechend auf die Knoten der Cluster A und B gesetzt werden. Schließlich müssen nur noch die einzelnen Cluster-Knoten A und B den soeben erstellten Knoten ihrer Vereinigung als Elternknoten setzen (Zeilen 26 und 27). Das neu vereinigte Cluster kann der globalen Liste aller Cluster hinzugefügt (Zeile 29), während die anderen beiden daraus entfernt werden (Zeilen 30 und 31).

3.3.3 Speichern und Ausgeben der Ergebnisse

Abschließend muss ein neuer Datensatz erstellt werden, der die Clusterergebnisse enthält und in der *ClassificationConfiguration* gesetzt wird, sodass AMUSE die Klassifikationsaufgabe beenden kann. Dazu werden zunächst die Attribute des Eingabedatensatzes, sowie all ihre Merkmalswerte entsprechend des zweidimensionalen Arrays *allValues*, übernommen. Danach wird für jedes verbleibende Cluster in *clusterAffiliation* ein weiteres Attribut mit dem Namen *cluster_y* und dem jeweiligen Index innerhalb von *clusterAffiliation* für y erstellt. Diese Attribute werden simultan befüllt, sodass sie, analog zu denen in Abschnitt 3.2.3 erstellten Attributen, pro Musikstück einen Wert von 1 enthalten, falls das Musikstück zu diesem Cluster gehört, oder 0 falls nicht. Der fertige Ergebnisdatensatz wird schließlich in der *ClassificationConfiguration* gesetzt.

Damit ist die eigentliche Arbeit der Adapterklasse bereits abgeschlossen, jedoch sind für hierarchische Clusterverfahren die Ergebnisse, besonders bei einer vollständigen Zusammenführung bzw. Aufspaltung, eher uninteressant, da sie in diesem Fall immer identisch sind. So können die hierarchischen Verfahren nach der Philosophie *Der Weg ist das Ziel* arbeiten, wobei die Dendrogramme das eigentliche Ergebnis darstellen. Mit ihnen kann die Reihenfolge der Cluster-Vereinigungen bzw.-Aufspaltungen nachvollzogen und damit eine beliebige Anzahl von Teilclustern abgelesen werden. Sie enthalten Informationen zu den (Un-) Ähnlichkeiten eines Objektes zu jedem anderen und können so nach der initialen Erstellung, z. B. innerhalb eines Musikempfehlungssystems, sehr effizient jedem Nutzer ohne Mehraufwand weitere ähnliche Titel aufzeigen. Eine vollständige Zusammenführung bzw. Aufspaltung kann in diesem Kontext sinnvoller sein als eine bestimmte Anzahl von Clustern zu verlangen, da ein vollständiges Dendrogramm die Informationen der Einzelnen subsumiert.

```
1  private String getNodeStructure (Node current, int maxDepth) {
2     String result = "";
3
4     if (current != null) {
5
6        String indent = " ";
7        for (int v = current.getValue().size(); v < maxDepth; v++) {
8           indent += "  ";
9        }
10
11       String openNode = indent + "[";
12
13       if (current.getValue().size() == 1) {
14          openNode += "(" + current.getValue().get(0) + ") " +
                this.getSongnameFromPath(current);
15          return openNode + "] \n";
16       } else if (current.getLeft() != null && current.getRight() != null) {
17          openNode += this.integerListToString(current.getValue());
18
19          String leftChildsString = this.getNodeStructure(current.getLeft(),
                maxDepth);
20          String rightChildsString = this.getNodeStructure(current.getRight(),
                maxDepth);
21
22          String endNode = indent + "] \n";
23          return openNode + "\n" + leftChildsString + rightChildsString + endNode;
24       } else {
25          AmuseLogger.write(this.getClass().getName(), Level.WARN, "Something went
                wrong while printing the dendrogram structure.");
26       }
27    }
28
29    return result;
30 }
```

Algorithmus 3.11: Methode der Dendrogrammklasse `getNodeStructure(...)`

Zu diesem Zweck kann die Dendrogrammklasse mithilfe der Methode `printTikzDendrogram (String path)` eine LaTeX-Datei erstellen, die alle Dendrogramme dieser Klassifizierungsaufgabe visualisiert (und lediglich einmal manuell kompiliert werden muss). Dabei wird nach der Initialisierung des Dateikopfes mittels einer *for*-Schleife die interne Knoten-Liste durchlaufen, in der jeweils die Wurzelknoten gespeichert sind. Bei einer vollständigen Vereinigung ist das nur ein einziger Knoten, aber im Falle mehrerer Wurzelknoten wird für jeden einzelnen eine eigene *forest*-Umgebung [37] eröffnet. Innerhalb dieser *for*-Schleife wird dann für jeden Wurzelknoten die rekursive Hilfsmethode `getNodeStructure(...)` (Algorithmus 3.11) aufgerufen. Dort wird jeweils ein passender String des (Teil-) Dendrogramms erzeugt.

Nach der Erzeugung eines leeren Ergebnis-Strings in Zeile 2, wird der übergebene Knoten geprüft (Zeile 4). Anhand der initial als Parameter übergebenen maximalen Clustertiefe und der Größe des aktuell betrachteten Knotens wird die aktuelle Zeile eingerückt (Zeilen 6−9), um die Übersichtlichkeit innerhalb des Dokuments sicherzustellen. Danach wird eine Knotenklammer geöffnet ([) und mittels der Werteliste *value* des aktuellen Knotens ermittelt, ob es sich um einen Blattknoten handelt (*values* hat genau ein Element) oder nicht (Zeile 13). Im Falle eines Blattknotens wird zunächst in Zeile 14 die ID und der Name des zugehörigen Musikstückes dem String hinzugefügt, bevor die Knotenklammer in Zeile 15 geschlossen (]) und ein Zeilenumbruch eingefügt wird. Ansonsten werden die Lied-IDs der zugehörigen Musikstücke als Liste in den String eingefügt (Zeile 17), um diesen Knoten und entsprechend die Vereinigung zu repräsentieren. An dieser Stelle werden die rekursiven Aufrufe der Methode für das linke und rechte Kind gestartet und die Ergebnis-Strings gespeichert (Zeilen 19−20). Schließlich können die Strings zusammengesetzt und zurückgegeben werden. Insgesamt werden so in Zeile 23 die Knotenöffnung (Einrückung und Klammer), ein Zeilenumbruch, das Ergebnis des linken und dann des rechten Kindes, sowie die Knotenschließung (Einrückung, Klammer und Zeilenumbruch) aneinander gereiht.

Nach Abschluss der Rekursion wird in der Methode `printTikzDendrogram(...)`, falls es sich nicht um das letzte Cluster handelt, nach dem Ende der *forest*-Umgebung noch ein wenig Abstand zur Übersicht angefügt, bevor eine neue Iteration beginnt. Abschließend wird das LaTeX-Dokument geschlossen und unter `amuse/experiments/` gespeichert. Ein Beispiel für die fertige Ausgabe eines so erstellten Dendrogramms ist Abbildung 4.1 bzw. die Dendrogramme des Anhangs B. Die Färbungen wurden jedoch nachträglich eingefügt.

Kapitel 4

Validierung

Der Prozess zum Ermitteln, wie gut die Cluster zur unterliegenden Struktur passen, wird auch (Cluster-) Validierung genannt [3]. Wenn eine korrekte Aufteilung der Daten verfügbar ist, ist das übliche Vorgehen der Vergleich zwischen dem Muster und der vom Algorithmus vorgeschlagenen Aufteilung. Dieses Vorgehen wird innerhalb dieses Kapitels mittels zwei verschiedenen Anwendungsbeispielen (Abschnitt 4.1 und 4.2) durchgeführt, bevor die Ergebnisse in Abschnitt 4.3 verglichen werden.
Für praktische Anwendungen der Clusterverfahren wird eine genauere Evaluation der Clustervalidität, aufgrund des unterschiedlichen Verhaltens der Verfahren basierend auf den verwendeten Merkmalen und initialen Annahmen, benötigt [11]. Jedoch wird in dieser Arbeit lediglich die Genauigkeit, also die Anzahl der richtig zugeordneten Musikstücke geteilt durch die Anzahl aller geclusterten Musikstücke, als Qualitätsmaß angegeben. Auf weitere Indizes, welche Datenpartitionen vergleichen, wird hier wegen der simplen Beschaffenheit dieser Funktionstests verzichtet.

4.1 Genreklassifizierung

Zur Funktionsüberprüfung der Verfahren wird hier eine Genreklassifikation von vier unterschiedlichen Genres mit jeweils zehn Musikstücken durchgeführt, wobei die optimale Einteilung der wirklichen Genrezugehörigkeit entspricht. Die gewählten Genres mit ihren Charakteristika sind in Tabelle 4.1 dargestellt.

Genre	Gesang	Instrumente	Fokus
Hörbuch	klare Stimme	0	Stimme
Piano	−−	1	Melodie
Lofi	−−	> 2	Rhythmus
Death-Metal	verzerrte Stimme	> 2	−−

Tabelle 4.1: Genreübersicht

Musikgenre-Informationen sind unter anderem innerhalb der Klangfarbe, dem Rhythmus, der Harmonie oder der Melodie zu finden, wobei die klangfarbenbasierten Merkmale als nützlicher zur Unterscheidung zwischen Genres befunden wurden [32]. Der Mel-Frequenz-Cepstrum-Koeffizient (*Mel Frequency Cepstral Coefficient*, kurz MFCC) gehört nicht nur zu den klangfarbenbasierten Merkmalen, sondern hat im Vergleich zu anderen Merkmalen der gleichen Kategorie auch genauere Ergebnisse erzielt. Deswegen wird innerhalb dieses Experiments nur der MFCC als Merkmal verwendet, während alle anderen ignoriert werden.

Zur Vorverarbeitung der Musikstücke wurde der NaN Eliminator, der alle ungültigen Werte (*Not A Number*, kurz NaN) entfernt, verwendet, da einige Klassifizierungsverfahren nicht mit dieser Art von Werten umgehen können. Zusätzlich wurden die rohen Merkmalswerte mithilfe einer Gaußschen Mischverteilung modelliert [13].

Ein Clusterverfahren soll im Folgenden als funktionsfähig gelten, wenn die Zuteilung der Musikstücke zu den Clustern durch die Verfahren besser als eine zufällige Aufteilung erfolgt. Bei vier Clustern soll folglich die Genauigkeit aller Methoden mindestens $\frac{1}{4}$ bzw. 25 % betragen. Weiterhin wird eine gute Funktionsfähigkeit als Genauigkeit von mindestens 50 % ($\frac{2}{4}$), und eine sehr gute als Genauigkeit ab 75 % ($\frac{3}{4}$) definiert.

Hörbuch	Piano	Lofi	Death Metal
Hörbuch 1	Avatar Theme	Celadon City	Free Will Sacrifice
Hörbuch 2	Beethoven Moonlight Sonata 1st Movement	Cerulean Fuchsia City	Guardians Of Asgaard
Hörbuch 3	Canon in D Piano	Cinnabar Island	Ironside
Hörbuch 4	Conquest Of Paradise Theme	Lavander Town	Ravens Flight
Hörbuch 5	Halloween Theme	Oaks Lab	Shield Wall
Hörbuch 6	Hedwigs Theme	Pewter Saffron Viridian City	Tattered Banners And Bloody Flags
Hörbuch 7	Over The Misty Mountains (Cold)	Pokemon Tower	Twilight Of The Thunder God
Hörbuch 8	Invincible Wrath Of The Lich King Piano	Route 1	Valkyria
Hörbuch 9	LOTR Main Theme	Route 24	Where Is Your God
Hörbuch 10	The Avengers Main Theme	Vermilion	Wings of Eagles

Tabelle 4.2: Datensatz mit entsprechender Genrezuteilung

Tabelle 4.2 listet den verwendeten Datensatz innerhalb seiner korrekten Aufteilung auf die vier Genres. Die Hörbücher $1-10$ sind jeweils genau fünf Minuten lange Ausschnitte des schwedischen Hörbuches *Harry Potter och de vises sten* gelesen von Björn Kjellman. Alle Stücke des Lofi-Genres wurden von dem Künstler GeanoFee[3] bereitgestellt und für das Death Metal Genre wurden Lieder der Band Amon Amarth[4] verwendet.

[3] https://www.youtube.com/GeanoFee
[4] https://www.amonamarth.com/

Die Piano-Versionen wurden von unterschiedlichen Künstlern eingespielt, wobei die genauen Informationen folgend gelistet sind:

Titel	Link	Künstler	Komponist
Avatar Theme	YouTube	Patrick Pietschmann	Benjamin Wynn und Jeremy Zuckerman
	Notenblätter		
Hedwigs Theme	YouTube	Patrick Pietschmann	John Williams
	Notenblätter		
LotR Main Theme	YouTube	Patrick Pietschmann	Howard Shore
	Notenblätter		
The Avengers Main Theme	YouTube	Patrick Pietschmann	Alan Silvestri
	Notenblätter		
Beethoven Moonlight Sonata 1st Movement	YouTube	Rousseau	Ludwig van Beethoven
	Notenblätter		
Canon in D	YouTube	Jacob Ladegaard	Johann Pachelbel
Conquest Of Paradise	YouTube	Andrew Dugros	Vangelis
Halloween Theme	YouTube	Noud van Harskamp	John Carpenter
	Notenblätter		
Over The Misty Mountains (Cold)	YouTube	Akmigone	Howard Shore
Invincible	YouTube	Jason Lam	Russell Brower, Derek Duke
	Notenblätter		und Glenn Stafford

Tabelle 4.3: Daten der verwendeten Piano-Versionen

Ergebnisse von Fast k-Means

Für den Fast k-Means-Algorithmus wurde der Parameter der Clusteranzahl k, basierend auf dem Wissen über den zugrunde liegenden Datensatz, auf 4 gesetzt. Das entspricht der Anzahl der verschiedenen Genres. Die übrigen Parameter wurden auf den von RapidMiner vorgeschlagenen Standardwerten belassen, da mit dieser Konfiguration bereits aussagekräftige Ergebnisse (Tabelle 4.4) erzielt wurden.

Fünf Death-Metal-Lieder wurden zusammen mit allen Pianostücken Cluster 0 zugeordnet, alle Hörbuchteile Cluster 1 und die anderen 5 Death-Metal-Lieder zusammen mit den Lofistücken Cluster Nr. 2. Die Lieder des Death-Metal-Genres wurden also, anstatt gemeinsam Cluster 3, innerhalb ihrer Alben zwei anderen Clustern zugewiesen. So gehören alle Metal-Stücke aus Cluster 2 dem Album *Berserker* (Amon Amarth, 2019) an, während die verbleibenden fünf Metal-Lieder aus Cluster 0 zu dem Album *Twilight of the Thunder God* (Amon Amarth, 2008) gehören. So

wurden insgesamt 30 von 40 Musikstücken korrekt gruppiert. Das ergibt eine Genauigkeit von 75 % und nach Definition besitzt Fast k-Means damit knapp eine sehr gute Funktionsfähigkeit.

Cluster 0	Cluster 1	Cluster 2	Cluster 3
Twilight Of The Thunder God	Hörbuch 1	Ironside	
Free Will Sacrifice	Hörbuch 10	Ravens Flight	
Guardians Of Asgaard	Hörbuch 2	Shield Wall	
Where Is Your God	Hörbuch 3	Valkyria	
Tattered Banners And Bloody Flags	Hörbuch 4	Wings of Eagles	
Avatar Theme*	Hörbuch 5	Celadon City*	
Beethoven Moonlight Sonata 1st Movement*	Hörbuch 6	Cerulean Fuchsia City*	
Canon in D Piano*	Hörbuch 7	Cinnabar Island	
Conquest Of Paradise Theme	Hörbuch 8	Lavander Town*	
Halloween Theme*	Hörbuch 9	Oaks Lab*	
Hedwigs Theme*		Pewter Saffron Viridian City	
Over The Misty Mountains (Cold)*		Pokemon Tower*	
Invincible		Route 1	
LotR Main Theme		Route 24	
The Avengers Main Theme		Vermilion LoFi	

Tabelle 4.4: Ergebnisse der Genreklassifizierung mit Fast k-Means

Anmerkung: Die Sterne bedeuten eine Sicherheit von mindestens 98 %, mit der dieses Musikstück zu dem zugewiesenen Cluster gehört.

Ergebnisse von Density Based Spatial Clustering of Applications with Noise

Die Auswahl der Parameter für das DBSCAN Verfahren war komplexer, da mit den Standardwerten alle Musikstücke in ein gemeinsames Cluster geordnet wurden. Mit dem Ziel von besseren Ergebnissen wurde eine kurze Vorstudie zur Findung einer geeigneten Parameterkombination durchgeführt. Innerhalb dieser Experimentreihe wurden die folgenden Kombinationen der beiden Parameter *Epsilon* und *MinPts* getestet:

Epsilon	MinPts	Epsilon	MinPts	Epsilon	MinPts	Epsilon	MinPts
1,0	6	0,1	5	0,01	5	0,001	5
1,0	5	0,1	4	0,01	4	0,001	4
1,0	4	0,1	3	0,01	3	0,001	3
1,0	3	0,1	2	0,01	2	0,001	2

Tabelle 4.5: Innerhalb der Genreklassifizierung für DBSCAN getestete Parameterkombinationen

Die getesteten Parameterwerte ergeben sich aus der theoretischen Überlegung, dass im besten Fall vier verschiedene Cluster im Ergebnis vorliegen sollten. Da die Standardwerte $Eps = 1,0$ und $MinPts = 5$ nur ein einziges Cluster hervorbrachten, wurde durch die Verminderung des Radius und der minimalen Objektzahl zur Clusterbildung versucht, die Clusteranzahl des Ergebnisses zu erhöhen.

Allerdings hat keine der in Tabelle 4.5 gelisteten Parameterkombinationen eine Veränderung des Ergebnisses (Tabelle 4.6) bewirken können. Die korrekte Funktion des Verfahrens konnte innerhalb dieses Anwendungsbeispiels folglich nicht belegt werden. Dies ist aber nur eine kleine Versuchsreihe gewesen, sodass es sehr wohl geeignetere Parameterkombinationen geben kann, die angemessenere Ergebnisse erzielen.

Die Leistung kann unter Umständen auch mit der Beschaffenheit der Daten unter den gewählten Voreinstellungen begründet werden. Für das gewählte Merkmal des MFCCs sind die jeweiligen Datenpunkte eventuell relativ gleichmäßig und damit für dichtebasierte Verfahren explizit ungeeignet verteilt. In dem Fall gäbe es keine dichteren und undichteren Bereiche, die erkannt werden könnten.

Der DBSCAN-Algorithmus wird durch RapidMiner bereit gestellt, sodass eine fehlerhafte Implementierung auszuschließen ist. Da das Verfahren weiterhin fehlerfrei durchläuft und analog zu dem funktionierenden k-Means Verfahren eingebunden wurde, scheint zumindest auch eine fehlerhafte Ausgabe der Ergebnisse ebenfalls unwahrscheinlich.

Cluster 0
Twilight Of The Thunder God*
Free Will Sacrifice*
Guardians Of Asgaard*
Where Is Your God*
Tattered Banners And Bloody Flags*
Ironside*
Ravens Flight*
Shield Wall*
Valkyria*
Wings of Eagles*
Celadon City*
Cerulean Fuchsia City*
Cinnabar Island*
Lavander Town*
Oaks Lab LoFi*
Pewter Saffron Viridian City*
Pokemon Tower*
Route 1*
Route 24*
Vermilion*
Avatar Theme*
Beethoven Moonlight Sonata 1st Movement*
Canon in D Piano*
Conquest Of Paradise Theme*
Halloween Theme*
Hedwigs Theme*
Over The Misty Mountains (Cold)* Invincible*
LotR Main Theme*
The Avengers Main Theme*
Hörbuch 1*
Hörbuch 10*
Hörbuch 11*
Hörbuch 12*
Hörbuch 13*
Hörbuch 2*
Hörbuch 3*
Hörbuch 6*
Hörbuch 7*
Hörbuch 8*

Tabelle 4.6: Ergebnisse der Genreklassifizierung mit DBSCAN

Ergebnisse von Support Vector Clustering

Das SVC-Verfahren hatte die gleiche Problematik wie DBSCAN, sodass auch hier eine Vorstudie zur Ermittlung geeigneter Parameter notwendig war. Erschwerend kam jedoch die Komplexität dieses Verfahrens hinzu. SVC besitzt deutlich mehr Parameter als alle anderen integrierten Clusterverfahren. Je nach Kernelfunktion sind es zwischen acht und zehn verschiedene Parameter, was die Anzahl der möglichen Parameterkombinationen deutlich erhöht. Getestet wurden die folgenden Parameterkombinationen:

Cache size	No. of sample points	Kernel value	Min. number of points	Convergence epsilon	Max. iterations	p	r
200	20	1,0	2	0,001	100000	0,0	−1,0
200	39	1,0	2	0,001	100000	0,0	−1,0
200	39	0,5	2	0,001	100000	0,0	−1,0
200	39	0,1	2	0,001	100000	0,0	−1,0
200	39	0,01	2	0,001	100000	0,0	−1,0
200	39	0,005	2	0,001	100000	0,0	−1,0
500	500	1,0	2	0,001	100000	0,0	−1,0
500	750	1,0	2	0,001	100000	0,0	−1,0
500	1000	1,0	2	0,001	100000	0,0	−1,0
500	3683	1,0	2	0,001	100000	0,0	−1,0
500	3683	100,0	2	0,001	100000	0,0	−1,0

Tabelle 4.7: Innerhalb der Genreklassifizierung für SVC mit der Kernelfunktion *radial* getestete Parameterkombinationen

Cache size	No. of sample points	Min. number of points	Convergence epsilon	Max. iterations	p	r
200	20	2	0,001	100000	0,0	−1,0
200	39	2	0,001	100000	0,0	−1,0
200	39	2	0,001	100000	0,0	0,1
200	39	2	0,001	100000	0,0	0,01
200	39	2	0,001	100000	0,0	0,001
500	200	2	0,001	100000	0,0	−1,0
500	500	2	0,001	100000	0,0	−1,0
500	500	2	0,001	100000	0,0	0,1
500	500	2	0,001	100000	0,0	0,01

Tabelle 4.8: Innerhalb der Genreklassifizierung für SVC mit der Kernelfunktion *dot* getestete Parameterkombinationen

Für die Kernelfunktionen *polynomial* und *neural* wurden ebenfalls die Standardparameter getestet, die jedoch wie die anderen Experimente das gleiche Ergebnis hervorbrachten. So wurden, ähnlich zur Tabelle 4.6, alle Musikstücke immer wieder in ein umfassendes Cluster geordnet. Die Funktionsfähigkeit von SVC konnte folglich ebenfalls mithilfe dieses Anwendungsbeispiels nicht festgestellt werden. Jedoch ist SVC wie DBSCAN ein dichtebasiertes Verfahren, sodass die schlechte Leistung analog hergeleitet werden kann. Unter Berücksichtigung des fehlerfreien Durchlaufs des Clusterverfahrens scheinen die Ergebnisse auch hier auf den zugrundeliegenden Daten zu gründen.

Ergebnisse von Wards Agglomeration

Für Wards Agglomeration wurde analog zu Fast k-Means der Parameter der Clusteranzahl k auf 4 gesetzt. Als Vorgehensweise wurde die Standardeinstellung der klassischen Methode beibehalten.

Cluster 0	Cluster 1	Cluster 2	Cluster 3
Twilight Of The Thunder God*	Avatar Theme*	Hörbuch 1*	Ironside*
Free Will Sacrifice*	Beethoven Moonlight Sonata 1st Movement*	Hörbuch 10*	Ravens Flight*
Guardians Of Asgaard*	Canon in D Piano*	Hörbuch 11*	Shield Wall*
Where Is Your God*	Conquest Of Paradise Theme*	Hörbuch 12*	Valkyria*
Tattered Banners And Bloody Flags*	Halloween Theme*	Hörbuch 13*	Wings of Eagles*
	Hedwigs Theme*	Hörbuch 2*	Celadon City*
	Over The Misty Mountains (Cold)*	Hörbuch 3*	Cerulean Fuchsia City*
	Invincible*	Hörbuch 6*	Cinnabar Island*
	LotR Main Theme*	Hörbuch 7*	Lavander Town*
	The Avengers Main Theme*	Hörbuch 8*	Oaks Lab*
			Pewter Saffron Viridian City*
			Pokemon Tower*
			Route 1*
			Route 24*
			Vermilion*

Tabelle 4.9: Ergebnisse der Genreklassifizierung mit Wards Agglomeration

Hier wurden, wie auch bei dem Fast k-Means Verfahren, die Metal-Lieder albumweise getrennt. Die ersten fünf Lieder des Metal-Genres sind Cluster 0 und die anderen fünf zusammen mit den Lofistücken Cluster 3 zugeordnet worden. Alle Hörbuchteile wurden Cluster 2 zugewiesen. Als Gegensatz zu dem partitionierenden Verfahren wurden die Pianostücke Cluster 1 zugeordnet, was eine Verbesserung darstellt. Wards Agglomeration hat damit 35 von 40 Musikstücken korrekt gruppiert, was eine Genauigkeit von $87,5\,\%$ ergibt. Folglich kann in diesem Fall nach Definition eine sehr gute Funktionsfähigkeit festgestellt werden.

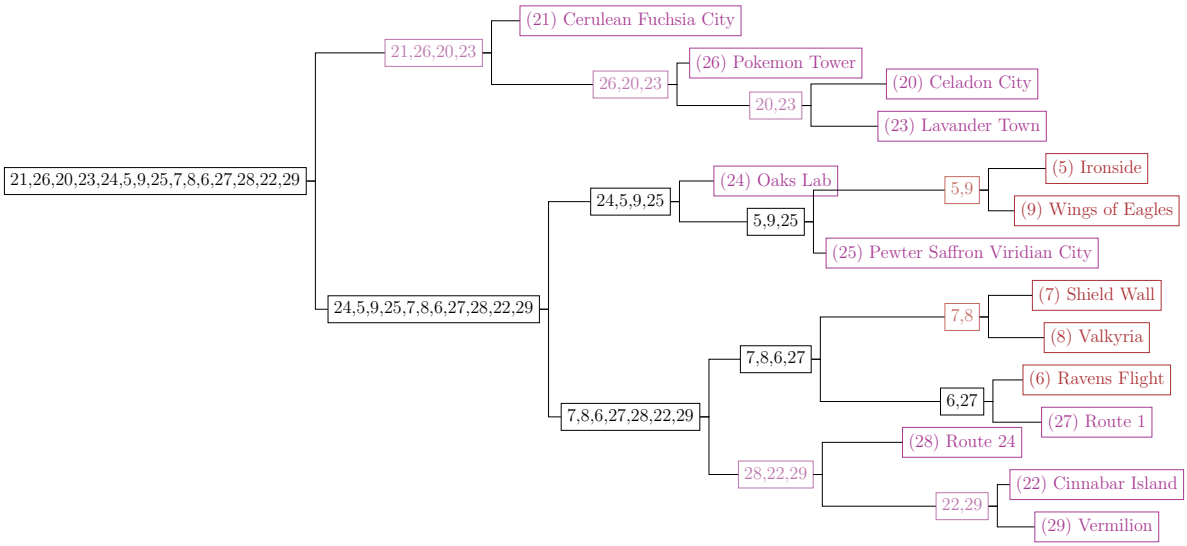

Abbildung 4.1: Dendrogramm von Wards Agglomeration für Cluster 3 der Genreklassifizierung

Anhand des Dendrogramms für Cluster Nr. 3 (Abbildung 4.1) kann an dieser Stelle auch etwas mehr Einsicht in den Clusterprozess gewonnen werden. Die anderen drei Dendrogramme können in Anhang B.1 eingesehen werden. Die Lieder, die zuerst vereint in ein gemeinsames Cluster geordnet werden, haben zusammen, wie in Abschnitt 2.3.2 und 3.3 beschrieben, die geringste Erhöhung der gesamten Distanz innerhalb dieses Clusters zur Folge. Anders gesagt bewirken die Musikstücke eines Clusters, im Vergleich zu anderen, untereinander die geringste Erhöhung der Unähnlichkeit, sodass die zuerst vereinigten Lieder anhand der verwendeten Merkmale besser zueinander passen, als die später hinzugefügten.

Die Vermischung der Genres passiert in diesem Fall bereits auf den untersten Ebenen in den Knoten $(5, 9, 25)$ und $(6, 27)$. Es steht folglich die Vermutung im Raum, dass die Lieder des Albums *Berserker* allgemein (bezüglich des MFCCs) ähnlicher zu den Liedern des Lofi-Genres als zum elf Jahre älteren Death-Metal-Album *Twilight of the Thunder God* sind. Da die Lieder des *Berserker*-Albums nicht erst zu fünft zusammen und dann mit den zehn Lofistücken, sondern teilweise auf der untersten Ebene gemischt geclustert wurden, lässt sich weiter vermuten, dass die Musikstücke dieses Albums auch untereinander sehr divers sind.

Um diese Ergebnisse zukünftig weiter zu verbessern, müsste mindestens ein weiteres, relevantes Merkmal hinzugezogen werden, welches Differenzen in genau den Knoten $(5, 9, 25)$ und $(6, 27)$ aufzeigt. Dabei ist ein Merkmal relevant [35], wenn das Entfernen des Merkmals zu einer verminderten Klassifizierungsleistung führt, und irrelevant, wenn die Leistung nicht beeinflusst wird.

4.2 Notenerkennung

Eine weitere Herangehensweise zur Validierung der integrierten Clusterverfahren ist eine Notenerkennung. Dafür sollen die Verfahren aus einem Musikstück die einzelnen Noten unterscheiden und Partitionen, in denen die gleichen Noten gespielt werden, zusammen clustern. Zu diesem Zweck wurde eine simple Melodie (Abbildung 4.2) mit einer Wiederholung, basierend auf dem Lied *Squarehammer* (2016) der Band Ghost, mit einem E-Bass gespielt und aufgenommen.

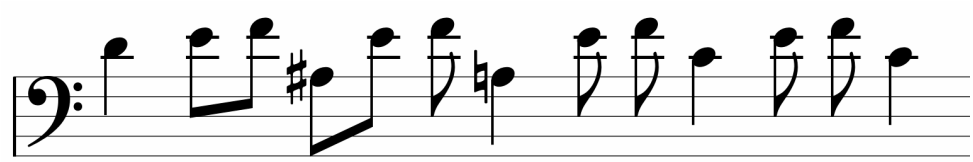

Abbildung 4.2: Melodie mit sechs verschiedenen Tönen (A, #A, C, D, E, F)

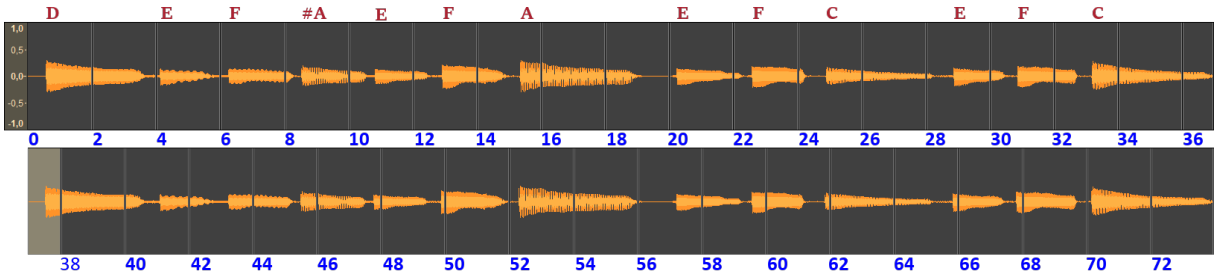

Abbildung 4.3: Wellenform der aufgenommenen Melodie

Anhand der Wellenform der Audiodatei (Abbildung 4.3) können die gespielten Noten (in rot) und die geraden Partitionen (durch die Senkrechten getrennt und deren Nummer in blau) begutachtet werden. Als Merkmal wurde für dieses Vorhaben ausschließlich der Chroma-Vektor genutzt, da Chroma-Merkmale die Intensität beschreiben, die mit den zwölf Halbtönen innerhalb einer Oktave assoziiert werden [8]. Die Partitionen sind analog zu der in Abschnitt 4.1 beschriebenen Vorverarbeitung mit dem NaN Eliminator und einer GMM entstanden. Die Partitionsgrößen betragen 500ms mit einer Überschneidung von 250ms, sodass die Viertelnoten, welche jeweils knapp 500ms lang sind, oft durch eine Partition genau abgedeckt werden. Die ungeraden Partitionen liegen dabei genau zwischen den Geraden, wobei die eingezeichneten Senkrechten jeweils deren Mitte darstellen.

Um die Ergebnisse der Clusterverfahren zu bewerten ist vorher eine klare Zuteilung der Partitionen zu den jeweiligen Noten notwendig, was durch Tabelle 4.10 dargestellt wird. Der mittlere Teil der

Tabelle enthält dabei die Partitionen, welche eine Note genau abdecken. Der obere und untere Part stellen die restlichen Partitionen dar, in denen sich zwei Noten treffen, wobei sie den Noten zugeordnet wurden, die auf ihnen länger klingen. Die Aufteilung zweier Noten auf den betroffenen Partitionen wurde grob mithilfe der den einzelnen Noten zugewiesenen Farben kenntlich gemacht. So ist bspw. auf Partition 47 ca. 125ms ($\frac{1}{4}$ der 500ms) lang die Note #A und ca. 375ms ($\frac{3}{4}$ der 500ms) die Note E zu hören. Innerhalb der Tabelle enthält der obere Teil die Partitionen, auf denen zwei Noten klingen, und die dominante Note die zweite ist. Analog enthält der untere Part diejenigen Partitionen, die aus der dominanten Note in eine andere ausklingen.

A	#A	C	D	E	F
					Partition 22
		Partition 24		Partition 10	Partition 30
		Partition 32		Partition 28	Partition 49
	Partition 08	Partition 61		Partition 40	Partition 59
Partition 14	Partition 45	Partition 69		Partition 47	Partition 67
Partition 15		Partition 25	Partition 00	Partition 04	Partition 06
Partition 16		Partition 26	Partition 01	Partition 11	Partition 13
Partition 17		Partition 27	Partition 02	Partition 19	Partition 23
Partition 18		Partition 33	Partition 37	Partition 20	Partition 31
Partition 52		Partition 34	Partition 38	Partition 29	Partition 43
Partition 53		Partition 35	Partition 39	Partition 41	Partition 50
Partition 54		Partition 62		Partition 56	Partition 68
Partition 55		Partition 63		Partition 57	
		Partition 70		Partition 65	
		Partition 71			
		Partition 72			
		Partition 73			
	Partition 09	Partition 36	Partition 03	Partition 05	Partition 07
	Partition 46	Partition 64		Partition 12	Partition 44
				Partition 21	Partition 51
				Partition 42	Partition 60
				Partition 48	
				Partition 58	
				Partition 66	

Tabelle 4.10: Zuteilung von Partitionen auf die sechs verschiedenen Töne (A, #A, C, D, E, F)

Da diese Aufteilung nicht hundertprozentig präzise erfolgt ist, werden zur Auswertung der Genauigkeit der Clusterverfahren die folgenden zwei Teilmengen der Partitionen verwendet. Zunächst die Menge der genau abdeckenden Partitionen P_{genau}, die durch das mittlere Tabellenfeld

gekennzeichnet wird. Weiterhin wird auch die erweiterte Menge der Partitionen $P_{\frac{3}{4}}$ betrachtet, in denen eine Note lediglich zu dreivierteln abgedeckt sein muss. Damit umfasst P_{genau} 42 und $P_{\frac{3}{4}}$ 61 Partitionen, wobei $P_{\text{genau}} \subset P_{\frac{3}{4}}$ gilt. Unter $P_{\frac{3}{4}}$ fallen zusätzlich die folgenden Partitionen:

$$\begin{aligned} \#\text{A} &: P_{08}, P_{45}, P_{09}, P_{46} \\ \text{C} &: P_{24}, P_{61}, P_{64} \\ \text{E} &: P_{10}, P_{28}, P_{40}, P_{47}, P_{48}, P_{58}, P_{66} \\ \text{F} &: P_{22}, P_{30}, P_{59}, P_{44}, P_{60} \end{aligned}$$

Ein Clusterverfahren soll im Folgenden als funktionsfähig gelten, wenn die Zuteilung der Partitionen zu den Clustern durch die Verfahren besser als eine zufällige Aufteilung erfolgt. Bei sechs Clustern soll folglich die Genauigkeit aller Methoden mindestens $\frac{1}{6}$ bzw. $16,67\,\%$ betragen. Weiterhin wird hier eine normale Funktionsfähigkeit als Genauigkeit von über $33,32\,\%$ ($\frac{2}{6}$), eine gute Funktionsfähigkeit als Genauigkeit von über $50,00\,\%$ ($\frac{3}{6}$) und eine sehr gute als Genauigkeit über $66,67\,\%$ ($\frac{4}{6}$) definiert.

Ergebnisse von Fast k-Means

Analog zur Genreklassifizierung wurde der Parameter der gewünschten Clusteranzahl k aufgrund des Wissens über den Datensatz gewählt. Da die Melodie sechs verschiedene Töne aufweist, wurde $k = 6$ gesetzt und die restlichen Standardparameter beibehalten.
RapidMiners Fast k-Means hat für die Teilmenge P_{genau} der Partitionen, welche die Noten genau abdecken, 34 von 42 korrekt gruppiert, was einer Genauigkeit von ca. $80,95\,\%$ entspricht. Dazu gehören 6/9 A-Partitionen (Cluster 1), 8/12 C-Partitionen (Cluster 2), 6/6 D-Partitionen (Cluster 3), 9/9 E-Partitionen (Cluster 4) und 5/7 F-Partitionen (Cluster 5).
Mit der erweiterten Menge $P_{\frac{3}{4}}$ sind es 48 von 61 korrekt zugeordneten Partitionen, was eine Quote von ca. $78,69\,\%$ ergibt. Darunter fallen 6/8 A-Partitionen (Cluster 1), 4/4 #A-Partitionen (Cluster 0), 9/15 C-Partitionen (Cluster 2), 6/6 D-Partitionen (Cluster 3), 13/16 E-Partitionen (Cluster 4) und 10/12 F-Partitionen (Cluster 5).
Probleme gab es insbesondere in Cluster 0 (siehe Tabelle 4.11), in dem neben den #A-Partitionen auch viele der anderweitig gemischten, sowie die Partitionen der ausklingenden A-Noten zugeordnet wurden. Auch wurde in Cluster 3 die Partitionen der Noten C und D, sowie in Cluster 4 die Noten E und F vermischt, was in Anbetracht der Nähe der Noten zueinander jedoch nicht unlo-

gisch scheint. Insgesamt lassen die Ergebnisse, analog zu den Definitionen der Genreklassifizierung, auf eine sehr gute Funktionsfähigkeit schließen.

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Partition 03	Partition 15	Partition 25	Partition 00	Partition 04	Partition 07
Partition 08	Partition 16	Partition 26	Partition 01	Partition 05	Partition 12
Partition 09	Partition 17	Partition 27	Partition 02	Partition 06	Partition 13
Partition 10	Partition 52	Partition 35	Partition 33	Partition 11	Partition 14
Partition 18	Partition 53	Partition 36	Partition 34	Partition 19	Partition 22
Partition 24	Partition 54	Partition 62	Partition 37	Partition 20	Partition 23
Partition 40		Partition 63	Partition 38	Partition 21	Partition 30
Partition 45		Partition 64	Partition 39	Partition 28	Partition 31
Partition 46		Partition 72	Partition 70	Partition 29	Partition 32
Partition 47		Partition 73	Partition 71	Partition 41	Partition 44
Partition 55				Partition 42	Partition 49
Partition 61				Partition 43	Partition 50
				Partition 48	Partition 51
				Partition 56	Partition 59
				Partition 57	Partition 60
				Partition 58	Partition 67
				Partition 65	Partition 68
				Partition 66	Partition 69

Tabelle 4.11: Ergebnisse der Notenbestimmung mit Fast k-Means

Ergebnisse von Density Based Spatial Clustering of Applications with Noise

Der DBSCAN-Algorithmus hat mit den Standardparametern alle Partitionen in ein großes Cluster sortiert, sodass auch in diesem Validierungsexperiment eine Vorstudie zur Findung einer geeigneten Parameterkombination durchgeführt werden musste.

Epsilon	MinPts	Epsilon	MinPts	Epsilon	MinPts	Epsilon	MinPts	Epsilon	MinPts
1,0	5	0,14	5	0,12	5	0,10	5	0,08	5
1,0	4	0,14	4	0,12	4	0,10	4	0,08	4
1,0	3	0,14	3	0,12	3	0,10	3	0,08	3
1,0	2	0,14	2	0,12	2	0,10	2	0,08	2
0,15	5	0,13	5	0,11	5	0,09	5	0,07	5
0,15	4	0,13	4	0,11	4	0,09	4	0,07	4
0,15	3	0,13	3	0,11	3	0,09	3	0,07	3
0,15	2	0,13	2	0,11	2	0,09	2	0,07	2

Tabelle 4.12: Für DBSCAN getestete Parameterkombinationen

Infolge der Experimente mit den in Tabelle 4.12 dargestellten Parameterkombinationen haben sich zwei Ergebnisse, zu sehen in den Tabellen 4.13 und 4.14, als interessant herausgestellt.

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Partition 00	Partition 04	Partition 08	Partition 10	Partition 25
Partition 01	Partition 05	Partition 09	Partition 24	Partition 26
Partition 02	Partition 06	Partition 45	Partition 36	Partition 35
Partition 03	Partition 07	Partition 46	Partition 47	Partition 62
Partition 15	Partition 11		Partition 61	Partition 63
Partition 16	Partition 12			Partition 72
Partition 17	Partition 13			Partition 73
Partition 18	Partition 14			
Partition 19	Partition 20			
Partition 27	Partition 21			
Partition 33	Partition 22			
Partition 34	Partition 23			
Partition 37	Partition 28			
Partition 38	Partition 29			
Partition 39	Partition 30			
Partition 40	Partition 31			
Partition 52	Partition 32			
Partition 53	Partition 41			
Partition 54	Partition 42			
Partition 55	Partition 43			
Partition 56	Partition 44			
Partition 64	Partition 48			
Partition 70	Partition 49			
Partition 71	Partition 50			
	Partition 51			
	Partition 57			
	Partition 58			
	Partition 59			
	Partition 60			
	Partition 65			
	Partition 66			
	Partition 67			
	Partition 68			
	Partition 69			

Tabelle 4.13: Ergebnisse der Notenbestimmung mit DBSCAN für $Eps = 0,08$ und $MinPts = 3$

Cluster 0	Cluster 1	Cluster 2	Cluster 3
Partition 00	Partition 03	Partition 25	Partition 33
Partition 01	Partition 04	Partition 26	Partition 34
Partition 02	Partition 05	Partition 27	Partition 70
Partition 15	Partition 06	Partition 35	Partition 71
Partition 16	Partition 07	Partition 62	
Partition 17	Partition 08	Partition 63	
Partition 18	Partition 09	Partition 64	
Partition 37	Partition 10	Partition 72	
Partition 38	Partition 11	Partition 73	
Partition 39	Partition 12		
Partition 52	Partition 13		
Partition 53	Partition 14		
Partition 54	Partition 19		
Partition 55	Partition 20		
	Partition 21		
	Partition 22		
	Partition 23		
	Partition 24		
	Partition 28		
	Partition 29		
	Partition 30		
	Partition 31		
	Partition 32		
	Partition 36		
	Partition 40		
	Partition 41		
	Partition 42		
	Partition 43		
	Partition 44		
	Partition 45		
	Partition 46		
	Partition 47		
	Partition 48		
	Partition 49		
	Partition 50		
	Partition 51		
	Partition 56		
	Partition 57		
	Partition 58		
	Partition 59		
	Partition 60		
	Partition 61		
	Partition 65		
	Partition 66		
	Partition 67		
	Partition 68		
	Partition 69		

Tabelle 4.14: Ergebnisse der Notenbestimmung mit DBSCAN für $Eps = 0,10$ und $MinPts = 3$

Der DBSCAN-Algorithmus hat mittels $0,08$ für Epsilon und 3 für die minimale Nachbarschaftsgröße fünf verschiedene Cluster erzeugt (Tabelle 4.13). Im Bezug auf die Teilmenge P_{genau} wurden

8/8 A- zusammen mit 5/12 C-Partitionen, 6/6 D-Partitionen und 2/9 E-Partitionen Cluster 0 zugewiesen. Cluster 1 enthält 7/9 E- und 7/7 F-Partitionen, während Cluster 4 die restlichen 7/12 C-Partitionen umfasst. Im Gegensatz dazu hat die Parameterkombination $Eps = 0,10$ und $MinPts = 3$ vier verschiedene Cluster hervorgebracht. Dort wurden 8/8 A- und 6/6 D-Partitionen Cluster 0, 9/9 E- und 7/7 F-Partitionen Cluster 1, sowie 8/12 C-Partitionen Cluster 2 und die restlichen 4/12 C-Partitionen Cluster 3 zugewiesen.

Bezüglich der erweiterten Teilmenge $P_{\frac{3}{4}}$ wurde für $Eps = 0,08$ zusätzlich eine E-Partition zu Cluster 0 geordnet, weitere vier E-Partitionen wurden zusammen mit den fünf zusätzlichen F-Partitionen in das gemeinsame Cluster 1, 4/4 #A-Partitionen Cluster 2 und schließlich eine weitere E-Partition Cluster 3 zugewiesen. Eine der drei weiteren C-Partitionen ist in Cluster 0 und die anderen beiden sind in Cluster 3 gelandet. DBSCAN mit $Eps = 0,10$ hat hingegen alle weiteren E- und F-Partitionen zusammen mit den #A-Partitionen Cluster 1 zugeordnet, während die C-Partitionen auf die Cluster 1 und 2 aufgeteilt wurden.

Interessant ist für diese beiden Ergebnisse der Unterschied der Einteilung der Partitionen 33, 34, 70, 71 und 08, 09, 45, 46. Während weiteren die C-Partitionen (33, 34, 70, 71) bei $Eps = 0,08$ zusammen mit den Partitionen der Noten A und D geclustert wurden, sind sie bei $Eps = 0,10$ in einem eigenen Cluster gelandet. Die 4/4 #A-Partitionen (08, 09, 45, 46) der erweiterten Menge $P_{\frac{3}{4}}$ sind bei $Eps = 0,08$ einzeln erkannt und alleine Cluster 2 zugewiesen worden, während sie bei $Eps = 0,10$ dem großen Cluster aus E- und F-Partitionen zugeordnet wurden. Diese Differenzen zeigen die Auswirkungen einer Veränderung des Epsilon-Parameters um nur 0.02 auf. Das veranschaulicht zusätzlich, dass mit genauer angepassten Parametern die Ergebnisse vermutlich noch verbessert werden können.

Eine Genauigkeitsangabe wird unter den Umständen, dass mehrere Noten mit verschieden vielen Partitionen ganz zusammen geclustert wurden (z. B. E- und F), im Zweifel für das Clusterverfahren gemacht. So liegt die Genauigkeit von DBSCAN mit $Eps = 0,08$ im Bezug auf die Teilmenge P_{genau} mit 22 von 42 korrekt zusammen geordneten Partitionen bei $52,38\,\%$. Dazu gehören 8/8 A-Partitionen in Cluster 0, 7/7 F-Partitionen in Cluster 1 und 7/12 C-Partitionen in Cluster 4. Für die erweiterte Teilmenge $P_{\frac{3}{4}}$ sind es 33 von 61 Partitionen mit einer Quote von $54,10\,\%$, für die 8/8 A-Partitionen in Cluster 0, 12/12 F-Partitionen in Cluster 1, 4/4 #A-Partitionen in Cluster 2, 2/16 E-Partitionen in Cluster 3 und 7/12 C-Partitionen in Cluster 4 zählen. Bei $Eps = 0,10$ ergibt sich eine Genauigkeit, mit 25 von 42 korrekt gruppierten Partitionen der Teilmenge P_{genau}, von $59,52\,\%$. Diese wurde mit den 8/8 A-Partitionen in Cluster 0, 9/9 E-Partitionen in Cluster 1

und 8/12 C-Partitionen in Cluster 2 ermittelt. Für die erweiterte Teilmenge $P_{\frac{3}{4}}$ sind es hingegen 32 von 61 Partitionen, was die Quote auf $52,46\,\%$ fallen lässt. Dazu zählen die 8/8 A-Partitionen in Cluster 0, 16/16 E-Partitionen in Cluster 1 und 8/12 C-Partitionen in Cluster 2.
Die Ergebnisse scheinen jedoch ungenau, da die Partitionen der relativ weit auseinander liegenden Töne A und D in beiden Ergebnissen zusammen geordnet wurden, während der exakt gleiche Ton C jeweils auf zwei Cluster aufgeteilt wurde. Die Theorie aus Abschnitt 4.1 über die Unangemessenheit von dichtebasierten Verfahren in diesem Anwendungszusammenhang bleibt folglich auch hier bestehen. Nichtsdestotrotz liegen die Genauigkeiten immer über $50\,\%$, was eine gute Funktionsfähigkeit bestätigt.

Ergebnisse von Support Vector Clustering

Bezüglich der Parameterwahl ergaben auch hier die Standardwerte lediglich ein umfassendes Cluster, was eine weitere Vorstudie notwendig machte. Mit dem Parameter *Number of sample points* = 73, da es insgesamt 74 verschiedene Partitionen gibt, wurde vor allem der Kernelwert und später der Radius stark variiert getestet.

Cache size	No. of sample points	Kernel value	Min. number of points	Convergence epsilon	Max. iterations	p	r
200	20	1,0	2	0,001	100000	0,0	−1,0
200	73	1,0	2	0,001	100000	0,0	−1,0
200	73	0,5	2	0,001	100000	0,0	−1,0
200	73	0,01	2	0,001	100000	0,0	−1,0
200	73	0,001	2	0,001	100000	0,0	−1,0
200	73	0,0001	2	0,001	100000	0,0	−1,0
200	73	2,0	2	0,001	100000	0,0	−1,0
200	73	3,0	2	0,001	100000	0,0	−1,0
200	73	5,0	2	0,001	100000	0,0	−1,0
200	73	10,0	2	0,001	100000	0,0	−1,0
200	73	50,0	2	0,001	100000	0,0	−1,0
200	73	100,0	2	0,001	100000	0,0	−1,0
200	73	125,0	2	0,001	100000	0,0	−1,0
200	73	150,0	2	0,001	100000	0,0	−1,0
200	73	175,0	2	0,001	100000	0,0	−1,0
200	73	200,0	2	0,001	100000	0,0	−1,0
200	73	100,0	2	0,001	100000	0,1	−1,0
200	73	100,0	2	0,001	100000	0,0	1,0
200	73	100,0	2	0,001	100000	0,0	0,9
200	73	100,0	2	0,001	100000	0,0	0,8
200	73	100,0	2	0,001	100000	0,0	0,7
200	73	100,0	2	0,001	100000	0,0	0,6
200	73	100,0	2	0,001	100000	0,0	0,5
200	73	100,0	2	0,001	100000	0,0	0,1
200	73	100,0	2	0,001	100000	0,0	0,01
200	73	100,0	2	0,001	100000	0,0	0,001

Tabelle 4.15: Innerhalb der Notenerkennung für SVC mit der Kernelfunktion *radial* getestete Parameterkombinationen

Mit der Einstellung von *Number of samlpe points* = 73, *Kernel value* = 100,0 und den restlichen Standardwerten wurden die gleichen Ergebnisse erzielt wie mit dem abgewandelten Kernelwert von 105 und 125. Dieses Ergebnis stellt insgesamt das beste Ergebnis der Versuchsreihe dar. So wurden die Partitionen auf drei verschiedene Cluster, wobei eines davon Rauschen repräsentiert, aufgeteilt.

Die Partitionen 01 und 38, die dem Cluster *Noise* zugeordnet wurden, sind genau abdeckende Partitionen der Note D. Die Partitionen aus Cluster 2 sind 6/8 der genau abdeckenden A-Partitionen. Die restlichen wurden alle dem Cluster 1 zugewiesen, während Cluster 0 komplett leer blieb.
Im Bezug auf die Teilmenge der Partitionen P_{genau} ergibt das gutwillig 20 von 42 zusammen geclusterten Partitionen, da die zwölf C-Partitionen in Cluster 1 überwiegen. Dies führt zu einer Genauigkeit von 47,62 %. Mit der erweiterten Menge von $P_{\frac{3}{4}}$ sind es jedoch lediglich 24 von 61 Partitionen, was die Quote auf 39,34 % senkt. Dabei inkludiert sind aus Cluster 1 16/16 E-Partitionen, sowie auch Cluster 2 die 6/8 A- und aus dem *Noise*-Cluster die 2/6 D-Partitionen.
Diese Leistung auch an der Komplexität des Verfahrens im Bezug auf die Menge an verschiedenen Parametern liegen.

Cluster 0	Cluster 1	Cluster 2	Noise
	Partition 00	Partition 15	Partition 01
	Partition 02	Partition 16	Partition 38
	Partition 03	Partition 17	
	Partition 04	Partition 52	
	Partition 05	Partition 53	
	Partition 06	Partition 54	
	Partition 07		
	Partition 08		
	Partition 09		
	Partition 10		
	Partition 11		
	Partition 12		
	Partition 13		
	Partition 14		
	Partition 18		
	Partition 19		
	Partition 20		
	Partition 21		
	Partition 22		
	Partition 23		
	Partition 24		
	Partition 25		
	Partition 26		
	Partition 27		
	Partition 28		
	Partition 29		
	Partition 30		
	Partition 31		
	Partition 32		
	Partition 33		
	Partition 34		
	Partition 35		
	Partition 36		
	Partition 37		
	Partition 39		
	Partition 40		
	Partition 41		
	Partition 42		
	Partition 43		
	Partition 44		
	Partition 45		
	Partition 46		
	Partition 47		
	Partition 48		
	Partition 49		
	Partition 50		
	Partition 51		
	Partition 55		
	Partition 56		
	Partition 57		
	Partition 58		
	Partition 59		
	Partition 60		
	Partition 61		
	Partition 62		
	Partition 63		
	Partition 64		
	Partition 65		
	Partition 66		
	Partition 67		
	Partition 68		
	Partition 69		
	Partition 70		
	Partition 71		
	Partition 72		
	Partition 73		

Tabelle 4.16: Ergebnisse der Notenbestimmung mit SVC

Mithilfe einer Methode zur Ermittlung guter Parameterwerte oder auch mehr Zeit für eine ausführlichere Studie sind durchaus genauere Ergebnisse denkbar. Insgesamt ist SVC mit einer Genauigkeit von über $33,32\,\%$ nach Definition normal Funktionsfähig.

Ergebnisse von Wards Agglomeration

Für Wards Agglomeration wurde der Parameter der Clusteranzahl k identisch zur Anzahl der verschiedenen Noten gewählt. Neben $k = 6$ wurde weiterhin die klassische Methode zur Berechnung der Unähnlichkeits-Matrix genutzt.

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Partition 15	Partition 04	Partition 25	Partition 06	Partition 03	Partition 00
Partition 16	Partition 05	Partition 26	Partition 07	Partition 08	Partition 01
Partition 17	Partition 11	Partition 27	Partition 12	Partition 09	Partition 02
Partition 52	Partition 19	Partition 35	Partition 13	Partition 10	Partition 32
Partition 53	Partition 20	Partition 62	Partition 14	Partition 18	Partition 33
Partition 54	Partition 21	Partition 63	Partition 22	Partition 24	Partition 34
	Partition 28	Partition 64	Partition 23	Partition 36	Partition 37
	Partition 29	Partition 72	Partition 30	Partition 40	Partition 38
	Partition 41	Partition 73	Partition 31	Partition 45	Partition 39
	Partition 42		Partition 43	Partition 46	Partition 69
	Partition 48		Partition 44	Partition 47	Partition 70
	Partition 56		Partition 49	Partition 55	Partition 71
	Partition 57		Partition 50	Partition 61	
	Partition 58		Partition 51		
	Partition 65		Partition 59		
	Partition 66		Partition 60		
			Partition 67		
			Partition 68		

Tabelle 4.17: Ergebnisse der Notenbestimmung mit Wards Agglomeration

Die Ergebnisse von Wards Agglomeration ähneln stark denen des Fast k-Means Verfahrens. Dabei wurden lediglich die Partitionen 06, 32, 36, 43, 44 und 69 anders gruppiert. Mit der Teilmenge P_{genau} der Partitionen, welche die Noten genau abdecken, als Bewertungskriterium wurden 6/8 A-Partitionen in Cluster 0, 8/12 C-Partitionen in Cluster 2, 6/6 D-Partitionen in Cluster 5, 9/9 E-Partitionen und 7/7 F-Partitionen in Cluster 3 sortiert. Dies ergibt mit 36 von 42 korrekt geclusterten Partitionen eine Genauigkeit von $85,71\,\%$.

Bezüglich der erweiterten Menge $P_{\frac{3}{4}}$, in dem alle Partitionen mit einer Notenabdeckung von drei Vierteln als relevant gelten, fällt der Wert mit 50 von 61 korrekt gruppierten Partitionen auf

81, 97 %. Darunter fallen 6/8 A-Partitionen (Cluster 0), 4/4 #A-Partitionen (Cluster 4), 9/15 C-Partitionen (Cluster 2), 6/6 D-Partitionen (Cluster 5), 13/16 E-Partitionen (Cluster 1) und 12/12 F-Partitionen (Cluster 3).
Wie auch bei Fast k-Means gab es hier Probleme in Cluster 4, bezüglich der Vermischung der #A-Partitionen mit anderen Gemischten und den ausklingenden A-Partitionen, und Cluster 5, mit einer Vermischung der D- und C-Partitionen. Für mehr Einsicht können alle Dendrogramme des Experiments in Abschnitt B.2 eingesehen werden. Die Unterscheidung der E- und F-Partitionen hat jedoch ausgesprochen gut funktioniert. Insgesamt hat dieses Verfahren eine nach Definition sehr gute Funktionsfähigkeit gezeigt.

4.3 Diskussion

Während in den vorherigen Abschnitten des Kapitels die Funktionsfähigkeit im Einzelnen für jedes Clusterverfahren gezeigt wurde, sollen nun hier die Ergebnisse verglichen und im Kontext zueinander betrachtet werden.

	Ergebnisse der Anwendungsbesipiele	
Clusterverfahren	Genreklassifizierung	Notenerkennung
Fast k-Means:	75, 00 %	Δ79, 82 %
DBSCAN:	−−	$Eps = 0,08:$ Δ53, 24 %
		$Eps = 0,10:$ Δ55, 99 %
SVC:	−−	Δ43, 48 %
Wards Agglomeration:	87, 50 %	Δ83, 84 %

Tabelle 4.18: Übersicht der Validierungsergebnisse

In Tabelle 4.18 sind die Genauigkeitswerte aller Verfahren im Bezug auf die Anwendungsbeispiele zur Validierung gelistet. Dabei bezieht sich der Deltawert der Notenerkennung auf den Durchschnitt aus den beiden einzelnen Quoten für die Teilmengen P_{genau} und $P_{\frac{3}{4}}$. Während sich Wards Agglomeration und Fast k-Means mit deutlich über drei Vierteln richtig zugeordneter Objekte absetzten, sind es mit DBSCAN noch mehr als die Hälfte. Für SVC kann dieser Erfolg leider nicht verzeichnet werden.

Wie in 4.1 bereits aufgegriffen, können die Ergebnisse der beiden dichtebasierten Verfahren jedoch verschiedene Ursachen haben. Zum einen ist es durchaus möglich, dass für die beiden gewählten Anwendungsbeispiele, welche jeweils nur ein einziges Merkmal verwendet haben, die Daten zu gleichmäßig verteilt sind. Sobald es keine erkennbaren Bereiche mit unterschiedlicher Objektdichte gibt, kann ein solcher dichtebasierter Algorithmus nur begrenzt gut arbeiten.
Weiterhin spielt wohl insbesondere die Anzahl und Komplexität der Parameter eine Rolle. Fast k-Means besitzt zwar auch mehrere Parameter, aber diese dienen nur der Optimierung der initialen Ergebnisse. Wie bei Wards Agglomeration ist der Parameter der Clusteranzahl k der Einzige, welcher maßgeblich Einfluss auf die Ergebnisse nimmt. DBSCAN besitzt hingegen zwei ausschlaggebende Parameter, wobei der des Radius (*Eps*) deutlich mehr Wissen über den Datensatz bzw. eine weitere Methode zur Bestimmung guter Parameter benötigt. Anhand des Vergleiches der Ergebnisse für DBSCAN in Abschnitt 4.2 ist erkennbar, dass solche auschlaggebenden Parameter mit hoher Präzision gewählt werden müssen. Der Unterschied zwischen den dort gewählten Werten für *Eps* beträgt nur $0,02$ und beeinflusst die Genauigkeit bereits um $7,14\,\%$ (im Bezug auf die Teilmenge P_{genau}). SVC besitzt hingegen mehrere relevante Parameter, sodass die Auswahl einer passenden Kombination noch komplexer wird.
Werden nun die Ergebnisse der beiden Anwendungsbeispiele vergleichen, kann man zwischen den Ergebnissen von Fast k-Means und Wards Agglomeration einen Unterschied beobachten. Während die Genauigkeit von Wards Agglomeration beide Male über $80\,\%$ liegt, sinkt sie um $\sim 4\,\%$ in der Notenerkennung. Bei Fast k-Means ist es exakt umgekehrt. Dort steigt die Quote um $\sim 4\,\%$ im Vergleich der Genreklassifizierung zur Notenerkennung. Ob dies an den Datensätzen selbst oder der allgemeinen Angemessenheit der Clusterverfahren für die jeweiligen Anwendungsbereiche liegt, kann ohne weitere Experimente nicht geklärt werden.

Kapitel 5

Fazit

5.1 Zusammenfassung

Innerhalb dieser Bachelorarbeit wurden in dem AMUSE-Framework mehrere Anpassungen zur Ermöglichung der Verwendung von Clusterverfahren umgesetzt. Dabei stellt AMUSE vereinfacht gesagt einen Werkzeugkoffer zur Musikdatenanalyse dar. Unter die Anpassungen fiel insbesondere die Einbindung der Fast k-Means-, DBSCAN- und SVC-Verfahren aus der RapidMiner-Bibliothek, sowie Wards Agglomeration als autonome Methode.
Die Integration der Verfahren geschah über ein gemeinsames Interface, welches durch die Aufspaltung des Ursprünglichen realisiert wurde (siehe Abschnitt 3.1). Nun gibt es zwei verschiedene Interfaces, die jeweils auf die überwachten bzw. unüberwachten Lernmethoden spezialisiert sind. Das *ClassifierUnsupervisedInterface* wird von allen Adapterklassen der Clusterverfahren implementiert, die wiederum in der `classifiy`-Methode die konkrete Klassifizierung starten. Um den reibungslosen Ablauf der Clusterverfahren zu garantieren, waren weitere Anpassungen notwendig. Darunter fiel insbesondere die durchführende Klasse des Prozesses, der *ClassifierNodeScheduler* (siehe auch Abbildung 3.2). Dort wurde unter anderem die Verarbeitung der Clusterergebnisse angepasst (Abschnitt 3.1) und zusätzlich eine angemessene Ausgabe für eben diese hinzugefügt (Anhang A).
Für die konkrete Einbindung der RapidMiner-Verfahren (Abschnitt 3.2) war zunächst der Aufruf des RapidMiner-Prozesses eine Schwierigkeit. Danach mussten die Ergebnisse in ein für AMUSE passendes Format umgewandelt werden. Die Integration von Wards Agglomeration (Abschnitt 3.3) umfasste mehrere Hürden: Zuerst mussten dort die Eingabedaten, sprich die zu klassifizierenden Musikstücke und ihre Partitionen, vorbereitet werden. Dazu gehörte insbesondere die Umsetzung der hierarchischen Struktur, was über eine Hilfsklasse *Dendrogram* realisiert wurde. Nach der Vorbereitung der Daten wurden zwei Methoden zur Berechnung der Distanz zwischen den einzelnen Objekten bzw. Objektgruppen (Clustern) implementiert. Beide verwenden eine Unähnlichkeits-Matrix, welche die Unähnlichkeit zwischen allen Clustern darstellt. Mithilfe dieser

Matrix wird der minimale Wert ausgewählt und die beiden Cluster, die zu diesem Wert gehören, werden vereinigt. Eine Möglichkeit dafür ist die klassische Variante, bei der iterativ in jedem Schritt eine komplett neue Unähnlichkeits-Matrix berechnet wird. Die andere Möglichkeit ist die *Lance-Williams Dissimilarity Update Formula*, welche die Unähnlichkeits-Matrix nach einer initialen Erstellung nur noch rekursiv aktualisiert. Abschließend werden die Ergebnisse auch hier in ein für AMUSE passendes Format überführt, jedoch findet zusätzlich auch eine Ausgabe der hierarchischen Struktur in Form eines Dendrogramms statt.
Einige Anpassungen an der Benutzeroberfläche wurden ebenfalls vorgenommen. Beispielsweise wurde der nicht benötigte Bereich zur Auswahl der Grundwahrheit, bei einer Wahl von unüberwachten Lernmethoden, ausgegraut.

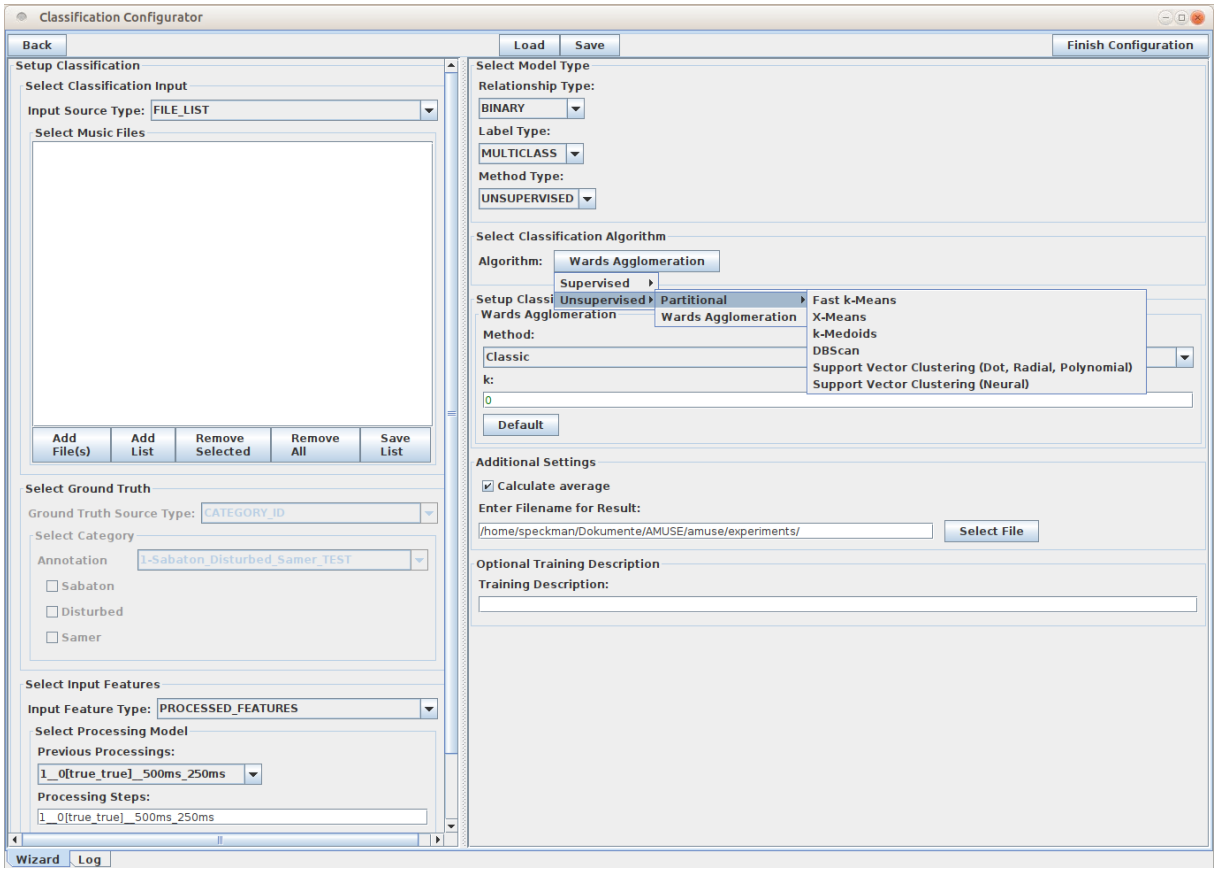

Abbildung 5.1: Aktuelle AMUSE GUI des Bereichs *Klassifikation* (vergleiche Abbildung 2.2)

In Kapitel 4 wurde schließlich die korrekte Integration der Clusterverfahren geprüft. Für zwei verschiedene Anwendungsszenarien, bei denen die idealen Ergebnisse bereits bekannt waren, wurden die Clusterverfahren angewendet und ihre Resultate mit dem Muster verglichen. Während

Fast k-Means und Wards Agglomeration in beiden Fällen sehr gute Ergebnisse erzielen konnten, hatten die beiden dichtebasierten Verfahren DBSCAN und SVC einige Probleme. Eine angemessene Funktionsfähigkeit konnte jedoch auch bei diesen beiden Verfahren abschließend bestätigt werden.

5.2 Ausblick

Wie in Kapitel 4 zu sehen ist, werfen die Parameterkombinationen der Clusterverfahren eine Problematik auf. Zukünftig wäre die Integration einer Methode sinnvoll, die automatisch gute Parameterwerte ermittelt. Dies könnte z. B. innerhalb des Trainingsbereiches von AMUSE umgesetzt werden, der bisher für unüberwachte Lernmethoden ungenutzt bleibt.
In dem Kontext scheint auch die Anpassung der AMUSE-internen Validierung sinnvoll. Diese könnte z. B. erweitert werden, sodass für unüberwachte Verfahren auch eine Grundwahrheit bereit gestellt werden kann, um die Genauigkeit er Clusterverfahren automatisch zu ermitteln. Weiterhin könnten dort auch weitere Qualitätsmaße, wie etwa der Dunn-Index [7], Integriert werden und eine nützliche Ergänzung darstellen. Die generelle Handhabung dieser Verfahren, so wie sie in dieser Arbeit umgesetzt wurde, kann jedoch ebenfalls weiter optimiert werden.
So kann neben der überwachten und unüberwachten Klassifizierung auch die Regression integriert werden. Dazu müsste zusätzlich der Wahrheitswert, welcher im *ClassifierNodeScheduler* die Unterscheidung der Verfahren umsetzt, in einen *Enum*-Typ umgewandelt werden. Dieser könnte dann über bspw. 0 = Überwacht, 1 = Unüberwacht und 2 = Regression das zu verwendende Verfahren speichern.
Ein weiter Punkt wäre die Zusammenzuführung der bisher getrennten Verfahren des SVC. Dies könnte z. B. durch Vererbung einen einheitlichen Aufruf des Clusterverfahrens ermöglichen, indem die Oberklasse das Grundgerüst enthält und die Kernelfunktionen durch eine jeweilige Unterklasse realisiert werden. Durch eine interne Unterscheidung der Unterklassen als eigene Algorithmen würden zu jeder eigenen Klasse die passenden Parameter angezeigt, wobei der eigentliche Aufruf einheitlich bliebe. Ein anderer Ansatz wäre die Anpassung der Auswahlmöglichkeiten über die GUI, sodass dynamisch weitere Parameter für die bisherige Parameterkonfiguration angezeigt werden können.
Im Bezug auf die hierarchischen Verfahren könnte auch die Dendrogrammklasse weiter ausgebaut werden. Aktuell unterstützt sie nur agglomerative Verfahren, sodass eine Erweiterung um z. B. eine Methode `setNewDivision(...)` als Gegenstück zur Methode `setNewMerge(...)`, zur Umsetzung

der spaltenden Verfahren, angemessen wäre. In diesem Zusammenhang könnte auch die interne Ausgabe des Dendrogramms in dem AMUSE-*Logger* korrigiert werden. Bisher wurde über formatierte Strings eine gleichmäßige Darstellung versucht, was sich jedoch als unwirksam herausstellte.

Mit den neuen Ausgaben der Klassifizierungsergebnisse in Form von LaTeX-Dateien scheint auch ein dedizierter Ordner zur Speicherung dieser Ergebnisse sinnvoll. Dafür müssten die AMUSE-Einstellungen eine weitere Spezifikation eines solchen Ordners zur Verfügung stellen, während in den entsprechenden Methoden der Speicherpfad auf eben diese referenziert.

Anhang A

LaTeX-Ausgabe der Klassifizierungsergebnisse

```
1  private void createVisual(List<String> songNames, List<double[]>
       clusterAffiliations, String path) {
2
3     AmuseLogger.write("ClassifierNodeScheduler - createVisual(...)", Level.DEBUG,
          "STARTING for output path: " + path);
4     File visualOutput = new File(path);
5     try {
6        BufferedWriter fileWriter;
7        fileWriter = new BufferedWriter(new FileWriter(visualOutput));
8        String sep = System.getProperty("line.separator");
9
10       String fileHead = "\\documentclass[12pt,border=10pt]{standalone} " + sep
11          + "\\usepackage{booktabs} " + sep
12          + "\\begin{document} " +sep;
13       fileWriter.append( fileHead );
14
15       String tabularStart = " \\begin{tabular}{";
16       String clusterNames = "";
17       for (int c=0; c < numberOfCategories-1; c++) {
18          tabularStart += "c";
19          clusterNames += "\\multicolumn{1}{c}{\\textbf{Cluster " + c + "}} & ";
20       }
21       tabularStart += "c} " + sep + " \\toprule " + sep;
22       if (((ClassificationConfiguration)taskConfiguration).getNoise()) {
23          clusterNames += "    \\multicolumn{1}{c}{\\textbf{Noise}} \\" + "\\ " + sep
              + " \\midrule " + sep;
24       } else {
25          clusterNames += "    \\multicolumn{1}{c}{\\textbf{Cluster " +
              (numberOfCategories-1) + "}} \\" + "\\ " + sep + " \\midrule " + sep;
26       }
27       fileWriter.append( tabularStart + sep + "  " + clusterNames );
28       ...
```

Algorithmus A.1: Beginn der Methode des *ClassifierNodeScheduler* `createVisual(...)`

Der *ClassifierNodeScheduler* speichert die Klassifizierungsergebnisse in einer für die weiteren Experimente nutzbaren ARFF-Datei ab, die jedoch zur Einsicht der Ergebnisse denkbar ungeeig-

net ist. Entsprechend wird nun zusätzlich die Methode `createVisual(...)` aufgerufen, welche die Ergebnisse in einer übersichtlichen Tabelle innerhalb einer LaTeX-Datei zur Verfügung stellt. Diese besteht aus mehreren Teilen, die folgend genauer betrachtet werden.

Zu Beginn der Methode (Algorithmus A.1) wird, neben der Ausgabe einer Startmitteilung an AMUSE (Zeile 3), in Zeile 4 eine neue Datei unter dem als Parameter übergebenen Pfad erstellt. Die Initialisierung eines *Writers*, der direkt in die eben erstellte Datei schreiben kann (Zeilen 6 und 7), sowie eines angemessenen Zeilentrenners (Zeile 8) wird zuerst vorgenommen.
In den Zeilen 10−13 wird der Dokumentenkopf erzeugt, in dem zur Erstellung der Tabellen das *booktabs*-Paket geladen werden muss. Es folgt die Erstellung des Tabellenkopfes (Zeile 15), der erstmal nur für die unüberwachten Verfahren optimiert ist. Denn folgend werden in den Zeilen 16 bis 20 entsprechend der Clusteranzahl−1 die Spaltenüberschriften mit *Cluster y* beschriftet, was im Falle der überwachten Verfahren zu dem jeweiligen Kategorienamen geändert werden könnte. Vor der Beschriftung des letzten Clusters wird geprüft, ob es sich um Rauschen handelt (Zeile 22), um die Spaltenüberschrift passend zu wählen. Über Zeile 27 wird der Tabellenkopf zusammengesetzt und in die Datei übertragen.

Um die Tabellen nun zu befüllen, müssen erst mittels Algorithmus A.2 für jedes Cluster die zugehörigen Namen der jeweiligen Musikstücke ausfindig gemacht werden (Zeilen 28−57). Dies wird mithilfe der Liste von String-Listen *allClustersAndTheirMembers* (Zeile 27) realisiert, indem jede String-Liste eine Gruppe repräsentiert und jeder String der Liste ein Name eines zugehörigen Musikstücks darstellt.
Die in Zeile 28 beginnende *for*-Schleife iteriert über die Indizes der vorhandenen Gruppen. So wird zu Beginn in Zeile 29 eine neue String-Liste für die aktuelle Gruppe erstellt, die anhand der von AMUSE berechneten Sicherheit (Zeilen 31−38), dass ein Musikstück zu einer bestimmten Gruppe gehört, befüllt wird (Zeilen 30−55). Gehört ein betrachtetes Musikstück zur aktuell betrachteten Gruppe (Zeile 40), wird weiterhin geprüft, ob es sich um die Partitionen eines einzelnen Musikstückes oder verschiedene Lieder handelt (Zeile 42). Falls es sich nicht um dasselbe Lied handelt, muss weiterhin erst der Name aus dem gegebenen Pfad zum Musikstück ausgelesen werden (Zeilen 44−48). Um dies zu erreichen, wird der Pfad-String in einen char-Array umgewandelt und vom Ende an rückwärts durchlaufen. Da die Dateiendung anhand der von AMUSE unterstützen Formate wie *.mp3* oder *.wav* immer aus vier Zeichen besteht, können diese ignoriert werden. Danach werden die Zeichen bis zum ersten Schrägstrich (unter Linux Betriebssystemen) bzw. zum ersten umgekehrten Schrägstrich (unter Windows) gelesen (Zeile

```
26  ...
27  List<List<String>> allClustersAndTheirMembers = new ArrayList<List<String>>();
28  for (int cluster = 0; cluster < numberOfCategories; cluster++) {
29     List<String> thisClustersMembers = new ArrayList<String>();
30     for (int song = 0; song < clusterAffiliations.size(); song++) {
31        int clusterOfThisSong = 0;
32        double clusterCertanty = clusterAffiliations.get(song)[0];
33        for (int clusterAgain = 0; clusterAgain < numberOfCategories;
              clusterAgain++) {
34           if (clusterCertanty < clusterAffiliations.get(song)[clusterAgain]) {
35              clusterCertanty = clusterAffiliations.get(song)[clusterAgain];
36              clusterOfThisSong = clusterAgain;
37           }
38        }
39
40        if (clusterOfThisSong == cluster) {
41           String nameOfThisSong = "";
42           if (songNames.size() == 1) { nameOfThisSong = "Partition " + song; }
43           else {
44              char[] name = songNames.get(song).toCharArray();
45              for (int positionInString = name.length -5; positionInString >= 0;
                    positionInString--) {
46                 if (name[positionInString] == '/' || name[positionInString] ==
                       '\\') { break; }
47                 else if (name[positionInString] == '_') { nameOfThisSong = "-" +
                       nameOfThisSong; }
48                 else { nameOfThisSong = name[positionInString] + nameOfThisSong; }
49              }
50           }
51           if (clusterCertanty > 0.98) {
52              thisClustersMembers.add(nameOfThisSong + "$^{\\ast}$");
53           } else { thisClustersMembers.add(nameOfThisSong); }
54        }
55     }
56     allClustersAndTheirMembers.add(thisClustersMembers);
57  }
58
59  int maxMemberCount = 0;
60  for (int clusters=0; clusters < allClustersAndTheirMembers.size(); clusters++) {
61     if (maxMemberCount < allClustersAndTheirMembers.get(clusters).size()) {
62        maxMemberCount = allClustersAndTheirMembers.get(clusters).size();
63     }
64  }
65
66  for (int clusters=0; clusters < allClustersAndTheirMembers.size(); clusters++) {
67     if (allClustersAndTheirMembers.get(clusters).size() < maxMemberCount) {
68        int difference = maxMemberCount -
              allClustersAndTheirMembers.get(clusters).size();
69           for (int diff = 0; diff < difference; diff++) {
70              allClustersAndTheirMembers.get(clusters).add(null);
71           }
72        }
73     }
74  ...
```

Algorithmus A.2: Mitte der Methode des *ClassifierNodeScheduler* `createVisual(...)`

46), da dort der Name aufhört und der Rest des Pfades beginnt. Ein eventueller Unterstrich im Namen muss auch abgefangen und in einen Bindestrich umgewandelt werden (Zeile 47), da dies sonst Probleme in LaTeX verursacht. Falls die Clustersicherheit über 98% beträgt, wird dem Namen des zugehörigen Musikstückes noch in den Zeilen 51 bis 53 ein Sternchen angehangen. Schließlich kann der nun ausgelesene Name der Liste hinzugefügt werden (Zeile 56).
In den Zeilen 59 bis 64 wird dann die maximale Größe der Gruppe bestimmt. Dies wird beim Befüllen der Tabelle relevant, wenn eine Gruppe bereits keine weiteren Mitglieder hat, aber auf den Index der Liste zugegriffen werden soll, um den nächsten Namen zu übertragen. Um dies zu verhindern werden die Listen, bis sie ebenfalls die maximale Gruppengröße erreicht haben, mit `null`-Werten befüllt (Zeilen 66−73).

Im letzten Teil der Methode (Algorithmus A.3) wird der Tabellen-String zunächst zeilenweise aufgebaut, wobei es genau so viele Zeilen geben wird, wie das größte Cluster zugehörige Musikstücke hat. Für jede Zeile werden alle Spalten, die je ein Cluster repräsentieren, bis auf die letzte durchlaufen (Zeilen 79−83) und, falls es noch Namen in der entsprechenden Gruppe gibt, werden die Namen der Zeile zusammen mit dem Spaltentrenner hinzugefügt. Falls es für ein Cluster keine weiteren Namen gibt, wird nur das Trennsymbol eingefügt. Die jeweils letzte Spalte wird ähnlich bearbeitet, nur, dass anstatt des Spaltentrenners zwei umgekehrte Schrägstriche als Zeilenumbruch innerhalb der Tabelle verwendet werden (Zeilen 84−86). Schließlich kann in Zeile 88 die Tabellenzeile dem Tabellen-String zusammen mit einem Zeilenumbruch des Dokuments angefügt werden und eine neue Iteration beginnt.
In Zeile 91 wird die befüllte Tabelle durch den *Writer* an das Dokument angehangen, bevor das Tabellenende folgt (Zeilen 93 und 94). Mit dem Ende der Methode ist die fertige Ausgabe erstellt und muss lediglich einmal mit LaTeX kompiliert werden, damit die Klassifizierungsergebnisse in einer übersichtlichen PDF-Datei begutachten werden können.

```
74      ...
75      String table = "";
76         for (int members = 0; members < maxMemberCount; members++) {
77
78            String tableLine = "    ";
79            for (int clusters=0; clusters < allClustersAndTheirMembers.size()-1;
                  clusters++) {
80               if (allClustersAndTheirMembers.get(clusters).get(members) != null) {
81                  tableLine += allClustersAndTheirMembers.get(clusters).get(members) +
                        " & ";
82               } else { tableLine += " & "; }
83            }
84            if (allClustersAndTheirMembers.get(allClustersAndTheirMembers.size()-1).
                  get(members) != null) {
85               tableLine +=
                     allClustersAndTheirMembers.get(allClustersAndTheirMembers.size()-1).
                     get(members) + " \\" + "\\";
86            } else { tableLine += " \\" + "\\"; }
87
88            table += tableLine + sep;
89         }
90
91         fileWriter.append( table );
92
93         String end = " \\bottomrule" + sep + " \\end{tabular}" + sep +
              "\\end{document}" ;
94         fileWriter.append( end );
95         fileWriter.close();
96
97      } catch (Exception e) {
98         AmuseLogger.write(ClassifierNodeScheduler.class.getName(), Level.WARN,
              "Classifier visual couldn't be created: " +e);
99      }
100  }
```

Algorithmus A.3: Ende der Methode des *ClassifierNodeScheduler* `createVisual(...)`

Anhang B

Weitere Ergebnisse

B.1 Dendrogramme der Genreklassifizierung

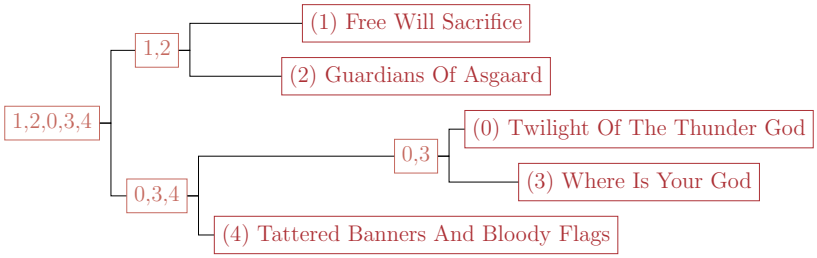

Abbildung B.1: Dendrogramm der Ward Agglomeration für Cluster 0 der Genreklassifizierung

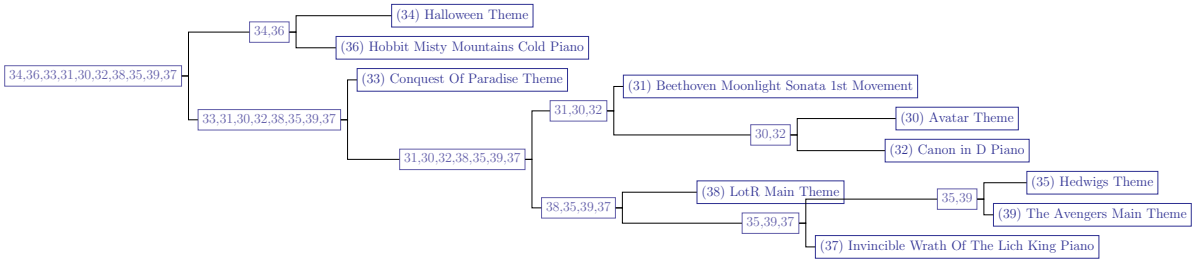

Abbildung B.2: Dendrogramm der Ward Agglomeration für Cluster 1 der Genreklassifizierung

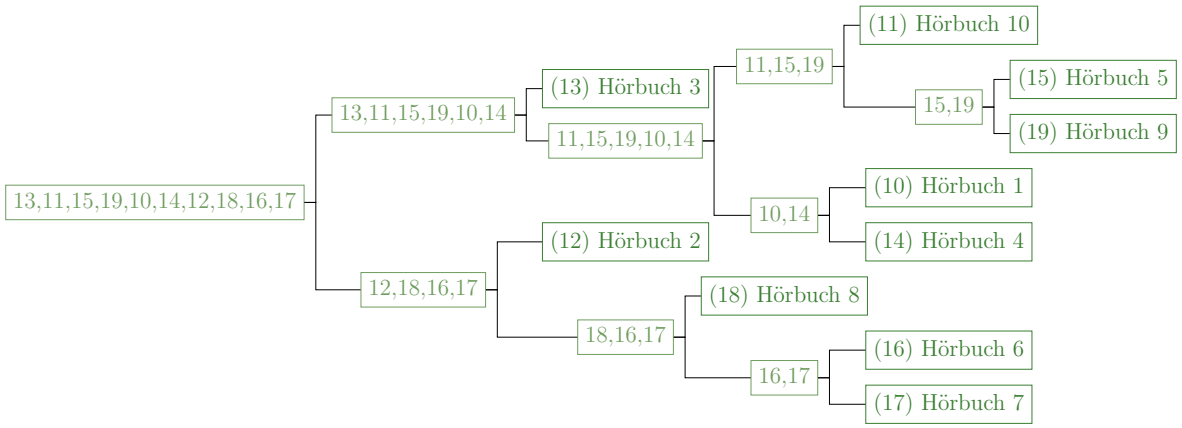

Abbildung B.3: Dendrogramm der Ward Agglomeration für Cluster 2 der Genreklassifizierung

B.2 Dendrogramme der Notenerkennung

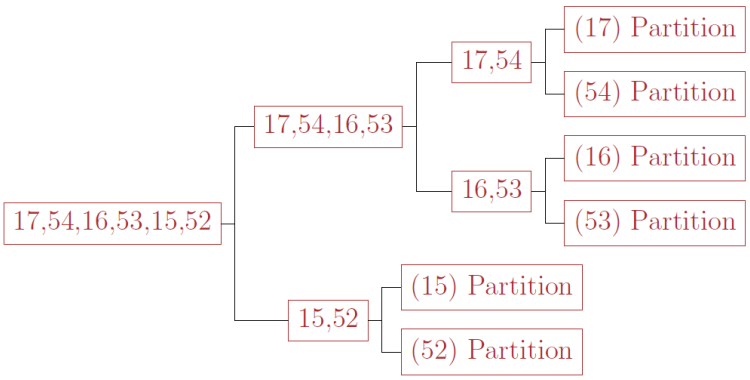

Abbildung B.4: Dendrogramm der Ward Agglomeration für Cluster 0 der Notenerkennung

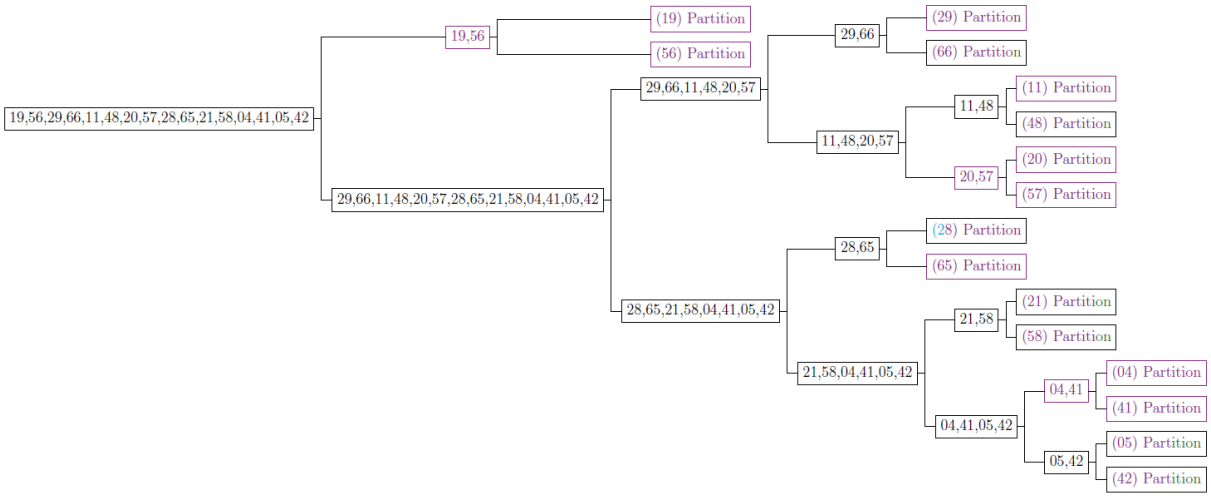

Abbildung B.5: Dendrogramm der Ward Agglomeration für Cluster 1 der Notenerkennung

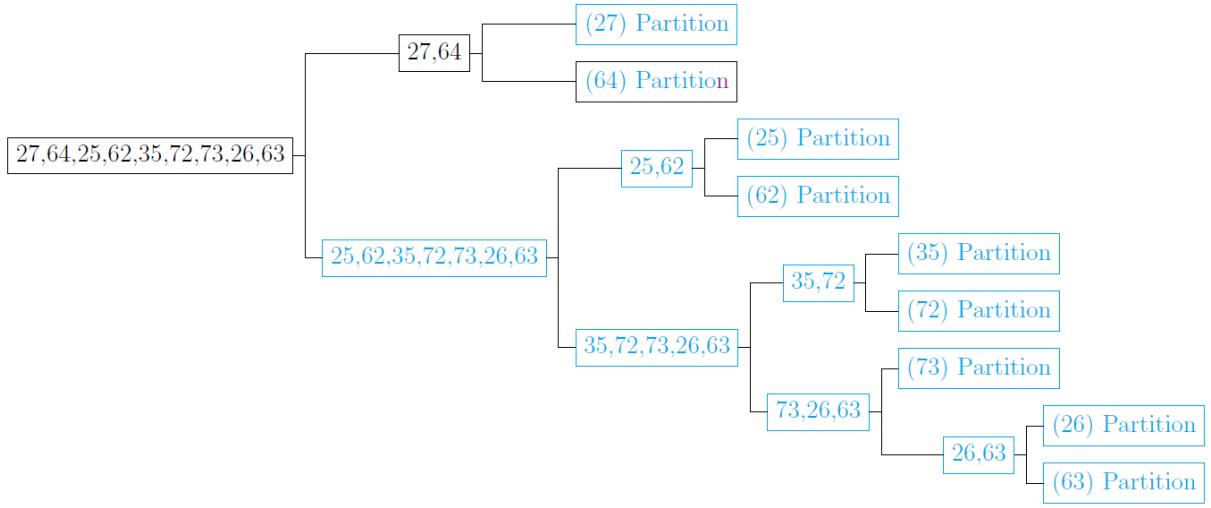

Abbildung B.6: Dendrogramm der Ward Agglomeration für Cluster 2 der Notenerkennung

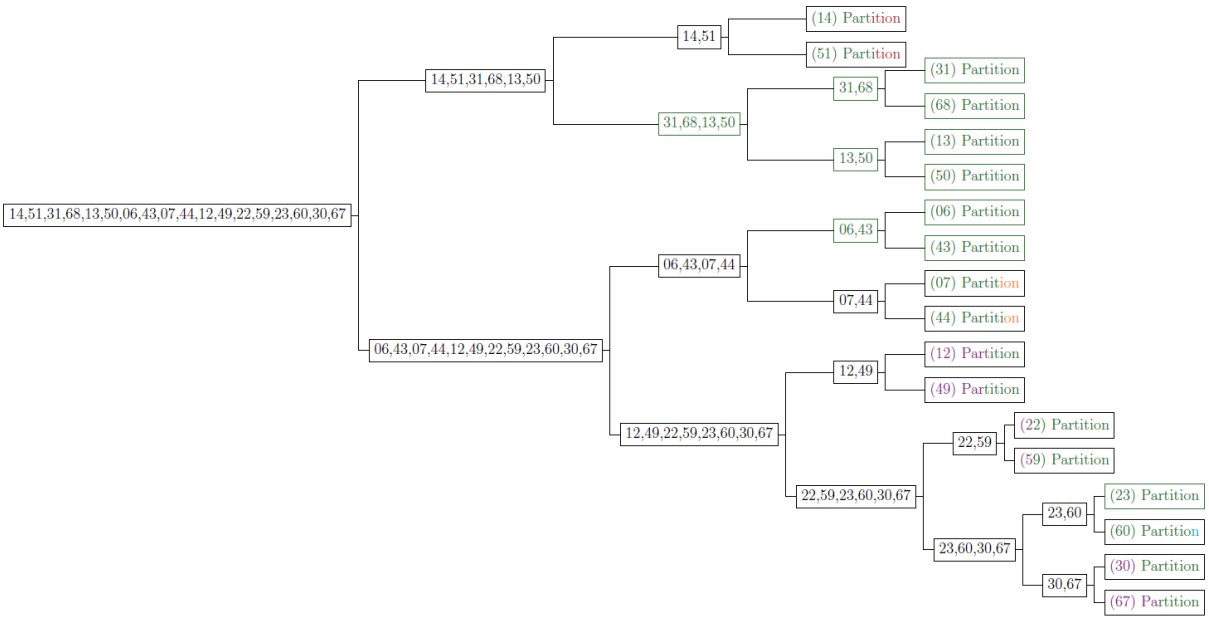

Abbildung B.7: Dendrogramm der Ward Agglomeration für Cluster 3 der Notenerkennung

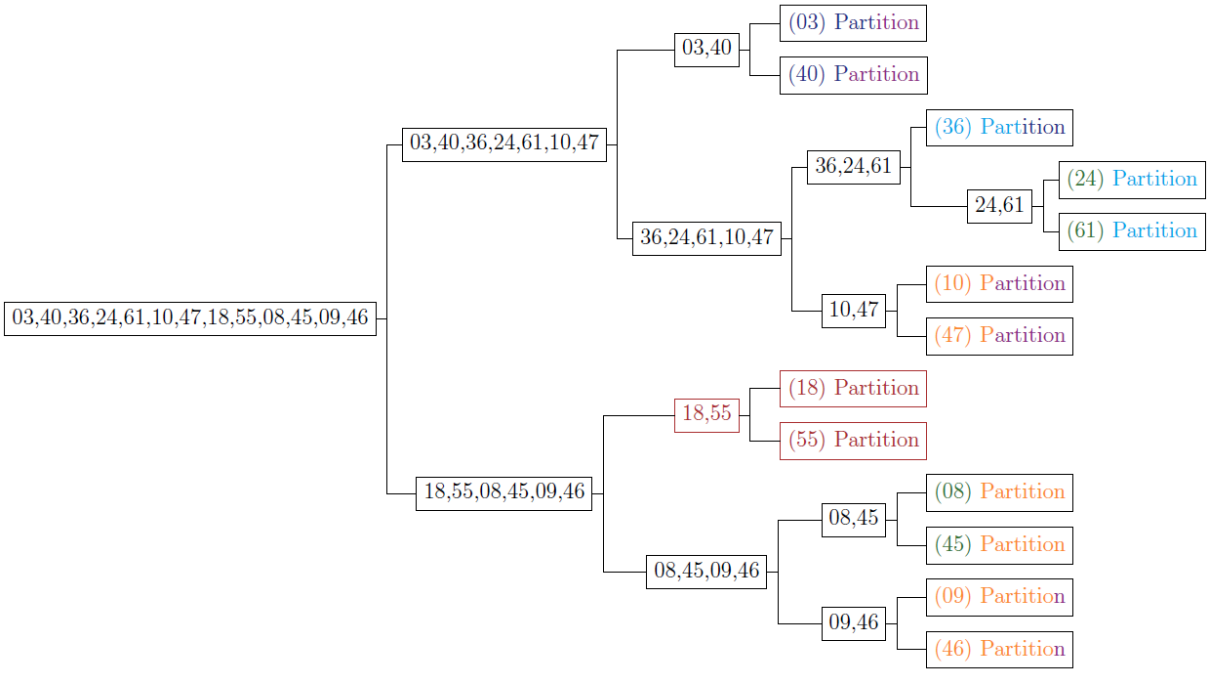

Abbildung B.8: Dendrogramm der Ward Agglomeration für Cluster 24 der Notenerkennung

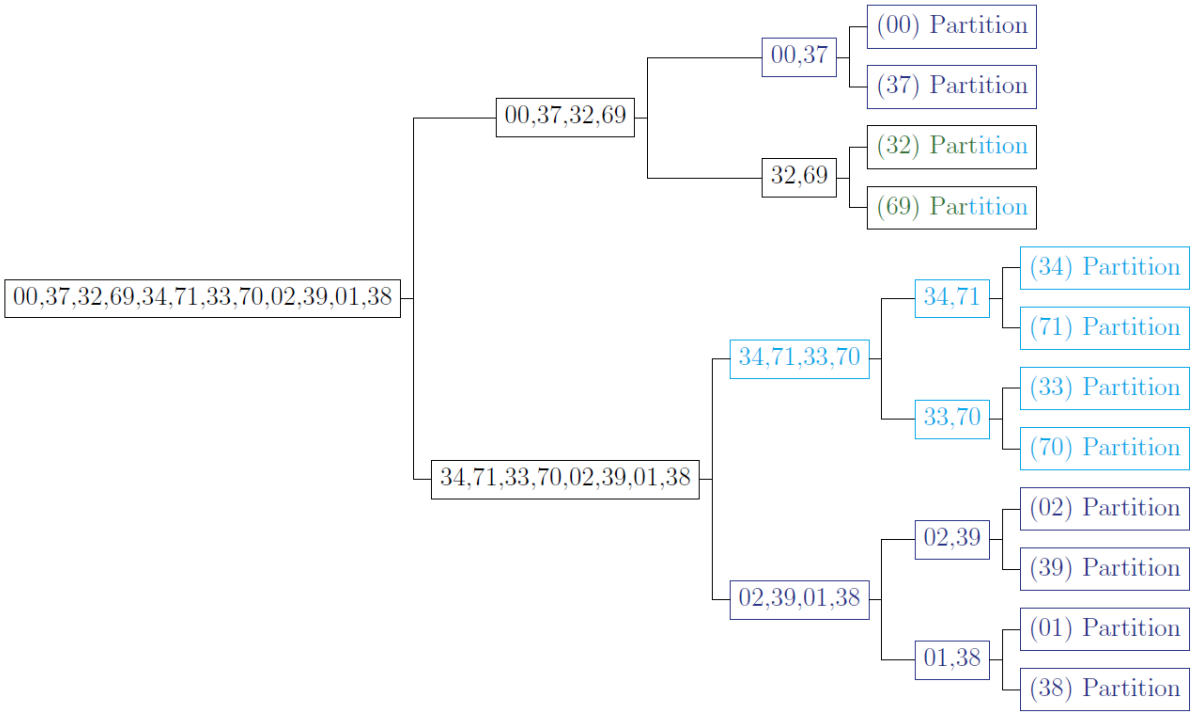

Abbildung B.9: Dendrogramm der Ward Agglomeration für Cluster 5 der Notenerkennung

Algorithmenverzeichnis

Literatur

[1] *About RapidMiner*. https://rapidminer.com/us/. Zugegriffen am 2021-04-23.

[2] Julien Ah-Pine und Xinyu Wang. "Similarity Based Hierarchical Clustering with an Application to Text Collections". In: *Proceedings of the 15th International Conference on Intelligent Data Analysis (IDA)*. Hrsg. von Henrik Boström, Arno J. Knobbe, Carlos Soares und Panagiotis Papapetrou. Springer International Publishing, Okt. 2016, S. 320–331. DOI: 10.1007/978-3-319-46349-0. URL: https://link.springer.com/chapter/10.1007/978-3-319-46349-0_28.

[3] Olatz Arbelaitz, Ibai Gurrutxaga, Javier Muguerza, Jesús M. Pérez und Iñigo Perona. "An Extensive Comparative Study of Cluster Validity Indices". In: *Pattern Recognition* 46.1 (Jan. 2013), S. 243–256. DOI: 10.1016/j.patcog.2012.07.021. URL: https://www.sciencedirect.com/science/article/pii/S003132031200338X.

[4] Mahta Bakhshizadeh, Ali Moeini, Mina Latifi und Maryam T. Mahmoudi. "Automated Mood Based Music Playlist Generation By Clustering The Audio Features". In: *Proceedings of the 09th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, Okt. 2019, S. 231–237. DOI: 10.1109/ICCKE48569.2019.8965190. URL: https://ieeexplore.ieee.org/abstract/document/8965190.

[5] Asa Ben-Hur, David Horn, Hava T. Siegelmann und Vladimir Vapnik. "A Support Vector Method for Clustering". In: *Proceedings of the 13th Conference on Neural Information Processing Systems (NIPS)*. Hrsg. von Todd K. Leen, Thomas G. Dietterich und Volker Tresp. MIT Press, 2000, S. 367–373. URL: https://proceedings.neurips.cc/paper/2000/file/14cfdb59b5bda1fc245aadae15b1984a-Paper.pdf.

[6] Asa Ben-Hur, David Horn, Hava T. Siegelmann und Vladimir Vapnik. "Support Vector Clustering". In: *Journal of Machine Learning Research (JMLR)* 2 (Dez. 2001), S. 125–137. URL: https://www.jmlr.org/papers/volume2/horn01a/horn01a.pdf.

[7] Joseph C. Dunn. "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters". In: *Cybernetics and Systems* 3 (Nov. 1973), S. 32–57. DOI: 10.1080/01969727308546046.

[8] Daniel Ellis und Graham Poliner. "Identifying 'Cover Songs' with Chroma Features and Dynamic Programming Beat Tracking". In: *Proceedings of the 2007 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Bd. 4. IEEE, Apr. 2007, S. 1429–1432. DOI: 10.1109/ICASSP.2007.367348. URL: https://ieeexplore.ieee.org/document/4218379.

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander und Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the 02nd International Conference on Knowledge Discovery and Data Mining (KDD)*. Hrsg. von Evangelos Simoudis, Jiawei Han und Usama M. Fayyad. AAAI Press, Jan. 1996, S. 226–231. URL: https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf.

[10] Martin Ester und Jörg Sander. *Knowledge Discovery in Databases: Techniken und Anwendungen*. Springer, 2000. ISBN: 3-540-67328-8. DOI: 10.1007/978-3-642-58331-5. URL: https://www.springer.com/de/book/9783540673286.

[11] Maria Halkidi, Yannis Batistakis und Michalis Vazirgiannis. "Cluster Validity Methods: Part I". In: *SIGMOD Record* 31.2 (Juni 2002), S. 40–45. DOI: 10.1145/565117.565124. URL: https://dl.acm.org/doi/10.1145/565117.565124.

[12] Trevor Hastie, Jerome H. Friedman und Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001. ISBN: 978-1-4899-0519-2. DOI: 10.1007/978-0-387-21606-5. URL: https://link.springer.com/book/10.1007%2F978-0-387-21606-5.

[13] Jesper Højvang Jensen, Daniel P.W. Ellis, Mads G. Christensen und Søren H. Jensen. "Evaluation of Distance Measures Between Gaussian Mixture Models of MFCCs". In: *Proceedings of the 08th International Conference on Music Information Retrieval (ISMIR)*. Hrsg. von Simon Dixon, David Bainbridge und Rainer Typke. Austrian Computer Society, Sep. 2007, S. 107–108. URL: https://ismir2007.ismir.net/proceedings/ISMIR2007_p107_jensen.pdf.

[14] William Karush. "Minima of Functions of Several Variables with Inequalities as Side Conditions". In: *Traces and Emergence of Nonlinear Programming*. Hrsg. von Giorgio Giorgi und Tinne H. Kjeldsen. Springer Basel, 2014, S. 217–245. ISBN: 978-3-0348-0439-4. DOI: 10.1007/978-3-0348-0439-4_10. URL: https://doi.org/10.1007/978-3-0348-0439-4_10.

[15] Kamran Khan, Saif ur Rehman, Kamran Aziz, Simon Fong, Sababady Sarasvady und Amrita Vishwa. "DBSCAN: Past, present and future". In: *Proceedings of the 05th International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*. IEEE, Feb. 2014, S. 232–238. DOI: 10.1109/ICADIWT.2014.6814687. URL: https://ieeexplore.ieee.org/document/6814687.

[16] Godfrey N. Lance und William T. Williams. "A General Theory of Classificatory Sorting Strategies: II. Clustering Systems". In: *Computer Journal* 10.3 (Jan. 1967), S. 271–277. DOI: 10.1093/comjnl/10.3.271. URL: https://doi.org/10.1093/comjnl/10.3.271.

[17] Mark Levy und Mark B. Sandler. "Structural Segmentation of Musical Audio by Constrained Clustering". In: *IEEE Transactions on Audio, Speech, and Language Processing* 16.2 (Feb. 2008), S. 318–326. DOI: 10.1109/TASL.2007.910781. URL: https://ieeexplore.ieee.org/document/4432648.

[18] Qing Li und Byeong M. Kim. "Clustering Approach for Hybrid Recommender System". In: *Proceedings of the 03rd IEEE/WIC International Conference on Web Intelligence (WI)*. IEEE Computer Society, Okt. 2003, S. 33–38. DOI: 10.1109/WI.2003.1241167. URL: https://ieeexplore.ieee.org/document/1241167.

[19] Qing Li, Byeong Man Kim, Dong Hai Guan und Duk whan Oh. "A Music Recommender Based on Audio Features". In: *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, Juli 2004, S. 532–533. DOI: 10.1145/1008992.1009106. URL: https://dl.acm.org/doi/10.1145/1008992.1009106.

[20] T. Soni Madhulatha. “An Overview on Clustering Methods”. In: *International Organization of Scientific Research Journal of Engineering (IOSR-JEN)* 2.4 (Apr. 2012), S. 719–725. DOI: 10.9790/3021-0204719725. URL: http://iosrjen.org/Papers/vol2_issue4/AI24719725.pdf.

[21] Glenn W. Milligan und Martha C. Cooper. “An Examination of Procedures for Determining the Number of Clusters in a Data Set”. In: *Psychometrika* 50.2 (Juni 1985), S. 159–179. DOI: 10.1007/BF02294245. URL: https://link.springer.com/article/10.1007/BF02294245.

[22] Juul Mulder, Tom F.M. Bogt, Quinten A.W. Raaijmakers und Wilma A.M. Vollebergh. “Music Taste Groups and Problem Behavior”. In: *Journal of Youth and Adolescence* 36.3 (Apr. 2007), S. 313–324. DOI: 10.1007/s10964-006-9090-1.

[23] Fionn Murtagh und Pedro Contreras. “Algorithms for Hierarchical Clustering: An Overview, II”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7.6 (Dez. 2017), e1219. DOI: 10.1002/widm.1219. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1219.

[24] Fionn Murtagh und Pierre Legendre. “Ward’s Hierarchical Agglomerative Clustering Method: Which Algorithms Implement Ward’s Criterion?” In: *Journal of Classification* 31.3 (Okt. 2014), S. 274–295. DOI: 10.1007/s00357-014-9161-z. URL: https://doi.org/10.1007/s00357-014-9161-z.

[25] William S. Noble. “What is a Support Vector Machine?” In: *Nature Biotechnology* 24 (Dez. 2006), S. 1565–1567. DOI: 10.1038/nbt1206-1565. URL: https://www.nature.com/articles/nbt1206-1565.

[26] Shraddha Pandit und Suchita Gupta. “A Comparative Study on Distance Measuring Approaches for Clustering”. In: *International Journal of Research in Computer Science (IJORCS)* 2.1 (Dez. 2011), S. 29–31. DOI: 10.7815/ijorcs.21.2011.011.

[27] Yogita Rani und Harish Rohil. “A Study of Hierarchical Clustering Algorithm”. In: *International Journal of Information and Computation Technology (IJICT)* 3.11 (2013), S. 1225–1232. URL: http://www.irphouse.com/ijict_spl/14_ijictv3n11spl.pdf.

[28] *RapidMiner Studio Documentation*. https://docs.rapidminer.com/9.7/studio/. Zugegriffen am 2021-03-26.

[29] Chandan K. Reddy und Bhanukiran Vinzamuri. “A Survey of Partitional and Hierarchical Clustering Algorithms”. In: *Data Clustering: Algorithms and Applications*. Hrsg. von Charu C. Aggarwal und Chandan K. Reddy. CRC Press, 2013, S. 87–110. URL: https://www.routledge.com/Data-Clustering-Algorithms-and-Applications/Aggarwal-Reddy/p/book/9781466558212.

[30] Oliver Rittho, Ralf Klinkenberg, Simon Fischer und Ingo Mierswa. *Yale: Yet Another Learning Environment*. 2001. DOI: 10.17877/DE290R-15309. URL: http://dx.doi.org/10.17877/DE290R-15309.

[31] Archana Singh, Avantika Yadav und Ajay Rana. “K-means with Three different Distance Metrics”. In: *International Journal of Computer Applications (IJCA)* 67.10 (Apr. 2013), S. 13–17. DOI: 10.5120/11430-6785. URL: https://research.ijcaonline.org/volume67/number10/pxc3886785.pdf.

[32] Wei-Ho Tsai und Duo-Fu Bao. “Clustering Music Recordings Based on Genres”. In: *Proceedings of the 2010 International Conference on Information Science and Applications*. IEEE, Apr. 2010, S. 1–5. DOI: 10.1109/ICISA.2010.5480365. URL: https://ieeexplore.ieee.org/document/5480365.

[33] Igor Vatolkin, Wolfgang M. Theimer und Martin Botteck. “AMUSE (Advanced MUSic Explorer) - A Multitool Framework for Music Data Analysis”. In: *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR)*. Hrsg. von J. Stephen Downie und Remco C. Veltkamp. International Society for Music Information Retrieval, Aug. 2010, S. 33–38. URL: https://ismir2010.ismir.net/proceedings/ismir2010-8.pdf.

[34] Joe H. Ward Jr. “Hierarchical Grouping to Optimize an Objective Function”. In: *Journal of the American Statistical Association* 58.301 (März 1963), S. 236–244. DOI: 10.1080/01621459.1963.10500845. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500845.

[35] Claus Weihs, Dietmar Jannach, Igor Vatolkin und Guenter Rudolph, Hrsg. *Music Data Analysis: Foundations and Applications*. Computer Science and Data Analysis Series. CRC Press, 2016. ISBN: 978-1-4987-1956-8. DOI: 10.1201/9781315370996.

[36] Junjie Wu. “Advances in K-means Clustering: A Data Mining Thinking”. Diss. Tsinghua University, 2012. ISBN: 978-3-642-29806-6. URL: http://d-nb.info/1021362204.

[37] Sašo Živanović. *Forest: a PGF/TikZ-based package for drawing linguistic trees*. LaTeX-Paketdokumentation. Online erhältlich unter https://ctan.org/pkg/forest?lang=de, zugegriffen am 2021-04-23. 2017.

Eidesstattliche Versicherung
(Affidavit)

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.