Kombiniertes Data Mining
Effiziente Generierung von Hilfsinformationen während des Clustering

Diplomarbeit

zur Erlangung des akademischen Grades eines Magisters
der Sozial- und Wirtschaftswissenschaften
Diplomstudium Wirtschaftsinformatik

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik –
Data & Knowledge Engineering

Begutachter: o. Univ.-Prof. Dr. Michael Schrefl
Mitbetreuer: Dipl.-Wirtsch.-Inf.. Mathias Goller

Verfasst von: Klaus Stöttinger

Wels, 2004

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wels, 10.5.2004

_____

Klaus Stöttinger

Abstract

Verschiedene Fragestellungen im Data Mining können nur durch eine Kombination der verschiedenen Data Mining Verfahren, wie Clustering, Klassifikation und Assoziation, beantwortet werden. In den bestehenden Ansätzen wird die Kombination der Data Mining Verfahren losgelöst voneinander betrachtet.

Diese Arbeit führt den Begriff des „Kombinierten Data Mining“ ein. Dabei werden die verwendeten Data Mining Verfahren nicht mehr isoliert betrachtet, sondern als Einheit – mit dem Ziel aus Sicht der Qualität, Interpretierbarkeit und Effizienz ein „besseres“ Ergebnis zu erreichen. Eine Möglichkeit besteht darin im ersten Verfahren Hilfsinformationen zu berechnen, die im Nachfolgeverfahren Gewinn bringend verwendet werden können.

Im Rahmen dieser Arbeit werden Clustering und Klassifikation miteinander kombiniert. Dazu wird eine Implementierung der beiden Clusteringalgorithmen K-Means und DBSCAN vorgestellt, die als Vorgängerverfahren so viele Hilfsinformationen wie möglich für eine Klassifikation generieren. Untersucht werden die möglichen Hilfsinformationen, die während des Clustering erzeugt werden können, und der Mehraufwand, der durch diese Ermittlung der Hilfsinformationen, in Form einer längeren Laufzeit der Algorithmen, entsteht.

Some questions in Data Mining can only be solved with a combination of different Data Mining methods, like Clustering, Classification and Association. The combined Data Mining methods are viewed separately in the existing works.

This work introduces the term of “Combined Data Mining”. As a result of the “Combined Data Mining” the combined methods should be seen as a unit. The aim is to end up in a better result, in terms of quality, interpretability and efficiency. One possibility is to generate additional information in the first method, which could be used by the second method.

Clustering and Classification will be combined within this work. Therefore the two Clustering algorithms K-Means and DBSCAN will be implemented. These algorithms are predecessors and will generate as much additional information as possible for a classification. One aim of this work is to investigate the possible further information, which could be generated in the Clustering. Another aim is the investigation of the additive effort that accrues when the additional information will be generated.

Inhaltsverzeichnis

1 Einleitung

Der technologische Fortschritt der letzten Jahrzehnte und dessen Errungenschaften, wie zum Beispiel der Computer oder Satelliten zur Erdbeobachtung, ermöglichen es uns immer größere Mengen von immer komplexeren Daten zu speichern. Diese Daten können potentiell wichtige Informationen enthalten. Zum Beispiel können Daten über bereits getätigte Einkäufe von Kunden eines Sportartikelherstellers interessante Informationen darüber enthalten, welcher Typ von Kunde, welche Art von Artikeln kauft. Diese Information kann der Sportartikelhersteller nutzen um seine Umsätze zu steigern. Dieses Beispiel zeigt nur eines der vielen möglichen Anwendungsgebiete des Data Mining. Weitere mögliche Anwendungsgebiete wären zum Beispiel Marketing, Customer Relationship Management, Satellitenbeobachtungsdaten usw. In all diesen Bereichen werden große Mengen an Daten gespeichert. Diese enthalten in vielen Fällen potentiell wichtige Informationen. Die Extraktion solcher Informationen durch eine manuelle Analyse übersteigt allerdings menschliche Kapazitäten bei weitem. Das ist die Motivation des neuen Gebiets Knowledge Discovery in Databases bzw. Data Mining. Dieses Kapitel führt in den Themenbereich, die Struktur und den Aufbau der Diplomarbeit ein.

1.1 Gegenstand und Motivation

Data Mining und Knowledge Discovery in Databases (KDD) sind häufig verwendete Begriffe für diese junge interdisziplinäre Wissenschaft zum Extrahieren von Informationen aus Datenbanken. Dabei werden vor allem Methoden aus den Bereichen der Mathematik, der Statistik und des Maschinenlernens herangezogen. Es ist allerdings zu beachten, dass die Begriffe Data Mining und Knowledge Discovery in Databases, oder kurz einfach Knowledge Discovery (KD), eine unterschiedliche Bedeutung haben. Der Begriff Data Mining ist als Herzstück eines umfassenderen Prozesses, nämlich des Knowledge Discovery, zu sehen [ES00].

Data Mining ist die Anwendung effizienter Algorithmen, welche die in einer Datenbank enthaltenen gültigen Muster finden [FPSS96]. Ziel des Data Mining ist es Informationen von hoher Qualität, d.h. gültig, bisher unbekannt und potentiell nützlich, möglichst effizient zu extrahieren. Es gibt verschiedene Arten von Data Mining Verfahren. Die wichtigsten dieser Verfahren sind Clustering, Klassifikation, Assoziationsregeln und Generalisierung (siehe Kap. 2). Je nach Ziel der Anwendung

und des Typs der Daten wird entschieden welches Verfahren zielführend ist. Oft ist es nötig verschiedene Data Mining Verfahren miteinander zu kombinieren, um ein „besseres“ Ergebnis aus Sicht der Qualität, Interpretierbarkeit und Effizienz zu erhalten. Das nachfolgende Beispiel der Kombination von Clustering und Klassifikation soll dies verdeutlichen:

Clusteringverfahren gruppieren Daten anhand von Ähnlichkeiten in den untersuchten Merkmalen. Ob das gefundene Ergebnis tatsächlich eine befriedigende Lösung der ursprünglichen Aufgabenstellung darstellt, kann der Algorithmus nicht feststellen, da er die Problemstellung nicht kennt, bzw. nicht begreifen kann. Das bedeutet, dass die Ergebnisse durch den Menschen beurteilt werden müssen. Eine Analyse eines Clusteringergebnis durch den Menschen ist mit Schwierigkeiten verbunden, da in den meisten Fällen eine geeignete Darstellungsform für das Ergebnis fehlt. Graphische Darstellungsformen sind auf Grund der meist mehrdimensionalen Daten nicht möglich. Ebenso ist eine textuelle Darstellung in Form von Regeln auf Grund der meist großen Anzahl von gefundenen Regeln nicht zielführend. Die Klassifikation von den geclusterten Daten liefert dem Anwender die nötigen beschreibenden Daten, um die Güte des Clustering beurteilen zu können. Als Beispiel kann der Entscheidungsbaum angeführt werden, der dem Menschen beschreibende Daten zu seinem Ergebnis liefert. Dies ist nur eines der möglichen Anwendungsgebiete für die Kombination von Data Mining Verfahren.

Diese Arbeit beschäftigt sich mit der Steigerung der Effizienz bei wiederholter kombinierter Anwendung von verschiedenen Data Mining Verfahren. Dabei wird vor allem die Kombination von Clusteringverfahren mit Klassifikationsverfahren sowohl praktisch als auch theoretisch untersucht. Da der Begriff des „Kombinierten Data Mining“ in der Literatur bisher noch nicht definiert wurde, folgt nun die Definition des „Kombinierten Data Mining“:

- **Definition:** Beim „Kombinierten Data Mining“ wird ein Data Mining Verfahren *A* vor einem Data Mining Verfahren *B* ausgeführt, sodass *B* von *A* profitiert. *B* kann dazu das Ergebnis von *A* oder/und eigens ermittelte Hilfsinformationen von *A* für *B* nutzen. *B* profitiert dann von *A* wenn das Ergebnis von *B*, gemäß einem geeigneten Gütemaß, besser ist, oder/und sich die Laufzeit von *B* verringert.

Data Mining befasst sich mit sehr großen Datenbeständen. Demzufolge ist der Ressourcenverbrauch an Rechnerzeit und Speicher nicht zu vernachlässigen. Durch die Anwendung des „Kombinierten Data Mining“ soll eine Reduzierung des

Ressourcenbedarfs an Rechnerzeit und Speicher erreicht werden. Dies soll durch die Berechnung von Hilfsinformationen im ersten Schritt für die folgenden Schritte erreicht werden. Eine weitere Möglichkeit zur Reduktion besteht in der Kenntnis der verwendeten Verfahren. Die Metainformationen über die Vorgängerverfahren erlauben dem nachfolgenden Verfahren von einem vereinfachten Problem auszugehen. Aus diesen Möglichkeiten zur Reduktion des Ressourcenbedarfs lassen sich verschiedene Arten von „Kombinierten Data Mining“ ableiten, die nachfolgend vorgestellt werden:

- **Naives „Kombiniertes Data Mining“**: Beim naiven „Kombinierten Data Mining“ nutzt das Nachfolgerverfahren nur das Ergebnis des Vorgängerverfahren. Eine Optimierung findet nicht statt. Zum Beispiel kann ein Klassifikationsverfahren die Ergebnisse eines Clusteringverfahren als Trainingsdaten verwenden.

- **Vorgängerverfahren kennt Nachfolgerverfahren**: Das Vorgängerverfahren berechnet Hilfsinformationen für das Nachfolgeverfahren. Hilfsinformationen können zum Beispiel arithmetische Werte wie Summen, Minima oder Maxima der verwendeten Daten sein. Dadurch können die Berechnungen des Nachfolgeverfahrens vereinfacht werden (siehe Kap. 5).

- **Nachfolgerverfahren kennt Vorgängerverfahren**: Das Nachfolgeverfahren kann durch die Kenntnis über das Vorgängerverfahren von einem vereinfachten Problem ausgehen. Wenn, zum Beispiel, bei einem Clusteringverfahren nur konvexe Cluster erzeugt werden, muss der nachfolgende Klassifikationsalgorithmus die Attribute nach denen geclustert wurde, nicht mehr berücksichtigen, da das Klassifikationsergebnis sich auf direktem Wege errechnen lässt. In diesem Fall trennt die mittelsenkrechte Ebene zwischen den Cluster-Mittelpunkten die beiden Cluster optimal (vergleiche [HM04]).

Der Ansatz des naiven „Kombinierten Data Mining“ wird bereits, wie im oben erwähnten Beispiel, genutzt (siehe Kap. 3). Dies bedeutet, dass die beiden Ansätze „Vorgängerverfahren kennt Nachfolgerverfahren“ und „Nachfolgerverfahren kennt Vorgängerverfahren“ des „Kombinierten Data Mining“ neue Optimierungsansätze auf diesem Gebiet liefern. Zum Beispiel liefern Clusteringverfahren wie K-Means (siehe Kap. 2.2.2) immer kreisförmige Cluster. Daher lassen sich die Cluster leicht durch eine „Trennlinie“ voneinander trennen. Ein nachfolgendes Klassifikationsverfahren muss „nur“ ermitteln auf welcher Seite der Trennlinie ein Punkt liegt. Je nach Lage wird der Punkt dem entsprechenden Cluster zugeordnet. Abbildung 1 verdeutlicht dies vereinfacht mit zwei Clustern.

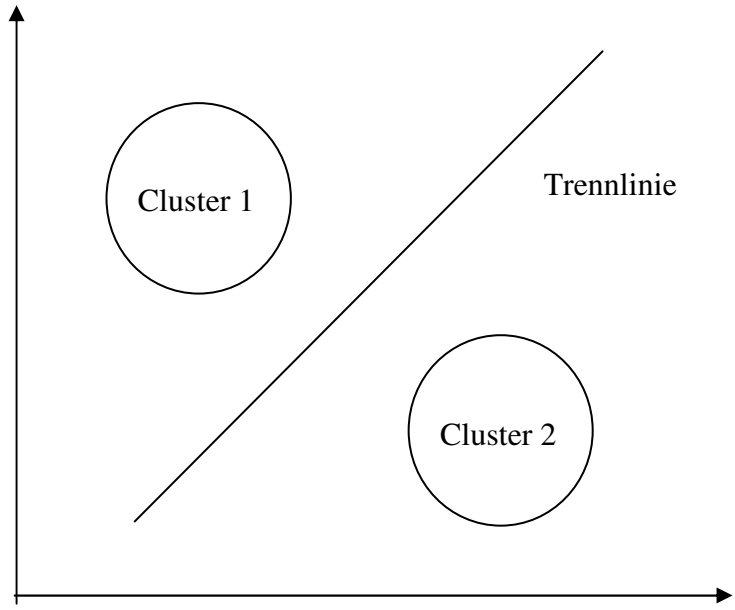


Abbildung 1: Beispiel der Nutzung von Hilfsinformationen und Metainformationen [GM04]

1.2 Aufgabenstellung

Aufgabenstellung der Diplomarbeit ist es festzustellen welche Hilfsinformationen bereits während des ersten Schritts, dem Clustering, gesammelt werden können. Dazu werden zwei Clusteringverfahren als Programm umgesetzt.

Zusätzlich werden aktuelle Ansätze zur Optimierung von Clusteringverfahren, und dem Einsatz von Clustering in Kombination mit anderen Data Mining Verfahren untersucht.

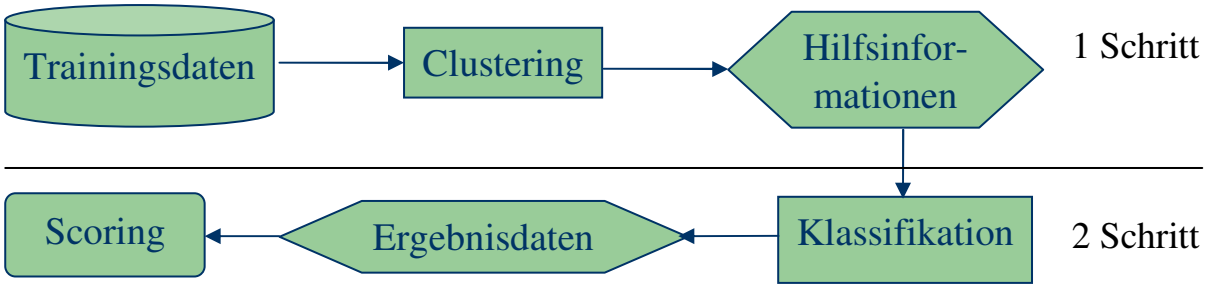


Abbildung 2: Ablaufdiagramm der Realisierung der Aufgabenstellung [GM04]

In einer zweiten Arbeit von Humer [HM04] werden die ermittelten Hilfsinformationen des Clustering für den zweiten Schritt der Klassifikation verwendet. Abbildung 2 stellt die Kooperation der beiden Arbeiten (Schritte) dar.

1.3 Zielsetzung

Gemäß der Aufgabenstellung werden zwei Clusteringverfahren, nämlich K-Means und DBSCAN, als Programm in der Programmiersprache C++ implementiert. Jedes dieser Verfahren wird so erweitert, dass während des Clustering so viele Hilfsinformationen wie möglich generiert werden. Zusätzlich muss die Ermittlung dieser Hilfsinformationen so effizient wie möglich gestaltet werden.

Anhand eines Laufzeit- und eines Genauigkeitstest werden die beiden erweiterten Clusteringverfahren K-Means und DBSCAN mit ihrer jeweiligen Standardimplementierung, d.h. ohne Ermittlung von Hilfsinformationen, verglichen.

Die Analyse der möglichen Verbesserung der Klassifikation durch die ermittelten Hilfsinformationen des Clustering wird von Humer Markus [HM04] durchgeführt.

1.4 Aufbau der Arbeit

Nachdem die ersten drei Abschnitte des Kapitels in die zu behandelnde Problematik eingeführt haben, wird nun ein kurzer Überblick über den inhaltlichen Aufbau der Arbeit gegeben.

Im nächsten Kapitel werden die notwendigen Grundlagen aus den Bereichen Knowledge Discovery in Databases, Data Mining, Clustering und Klassifikation zusammengefasst. Im dritten Kapitel wird eine Definition für das „Kombinierte Data Mining“ gegeben und die möglichen Arten des „Kombinierten Data Mining“ vorgestellt. Des Weiteren werden bestehende Ansätze des „Kombinierten Data Mining“ erläutert. Das vierte Kapitel zeigt Optimierungsmöglichkeiten zu den beiden Clusteringalgorithmen K-Means und DBSCAN und schließt mit Indexstrukturen für Datenbanken zur Verbesserung der Performance von Data Mining Algorithmen.

Das fünfte Kapitel stellt die ermittelten Hilfsinformationen für das „Kombinierte Data Mining“ für die beiden Algorithmen K-Means und DBSCAN vor. Anschließend beschäftigt sich das sechste Kapitel mit der Implementierung der beiden Algorithmen, mit besonderer Berücksichtigung der Ermittlung der Hilfsinformationen. Das siebte Kapitel analysiert die Testergebnisse des Laufzeit- und des Genauigkeitstest. Im letzten Kapitel wird schließlich ein Fazit über die neuen Ansätze des „Kombinierten Data Mining“ und die Testergebnisse abgegeben.

2 Grundlagen

Das letzte Jahrhundert war geprägt vom technologischen Fortschritt und denn damit einhergehenden rasanten Änderungen. Zum Beispiel wurde der Geschwindigkeits-weltrekord der Stanley Zwillinge von 1906 mit 122 Meilen pro Stunde inzwischen durch die Apollo Astronauten um das 223-fache überboten [BL00]. Aber vor allem die Menge der gespeicherten Daten hat sich im Laufe der Zeit stark verändert. Täglich wird eine enorme Menge von Daten erzeugt. Das allgegenwärtige Auftreten von Computern ermöglicht es, jegliche Information zu speichern. Preisgünstige Speicherträger mit immer größerem Speichervolumen erleichtern dies, und verleiten dazu nahezu alles zu speichern. Auch Informationen die früher verworfen worden wären, da sie für unnütz gehalten wurden, werden gespeichert. Das Internet ist ein weiteres Beispiel für die riesigen Mengen an Daten, die einerseits über das Internet verfügbar sind und andererseits beim Surfen im Netz gespeichert werden, in dem alle besuchten Seiten eines Benutzers registriert werden.

Ein kurzes Beispiel soll helfen zu vermitteln, wie groß die Datenmengen sind. Eine der größten Bibliotheken der Welt besitzt rund 17 Millionen Bücher. Würde man diese Bücher als MS Word Text speichern würde dies cirka 17 Terabyte an Daten ausmachen. Zum Vergleich umfasst die Datenbank des Packetzustellunternehmens UPS ebenfalls 17 Terabyte, d.h. ein einziges Unternehmen besitzt genauso viele Daten wie eine der größten Bibliotheken der Welt. Es wird geschätzt, dass sich die Menge der Daten, die in den Datenbanken der gesamten Welt gespeichert sind, sich mit dem derzeitigen raschen technologischen Fortschritt alle 20 Monate verdoppeln [BL00].

Nachfolgend führt dieses Kapitel in die Grundlagen des Knowledge Discovery in Databases und des Data Mining ein. Insbesondere werden Begriffe und Zusammenhänge des Data Mining und des Knowledge Discovery in Databases erläutert. Der erste Abschnitt geht auf die Definitionen der Begriffe Knowledge Discovery und Data Mining ein. Insbesondere wird die Beziehung der beiden Begriffe näher erläutert. Ein Schwerpunkt wird in diesem Teil auf den KDD-Prozess und dessen Schritte gelegt.

Der zweite Abschnitt gibt einen genaueren Einblick in die Clusteranalyse. Dabei wird zuerst vor allem auf den Begriff der „Ähnlichkeit“ näher eingegangen und dieser im Zusammenhang mit Clustering erläutert. Anschließend werden die zwei grund-

legenden Clusteringverfahren „Partitionierendes Clustering“ und „Hierarchisches Clustering“ dargestellt. Vor allem das partitionierende Clustering wird anhand zweier unterschiedlicher Algorithmen, K-Means und DBSCAN, erklärt.

Das Kapitel endet mit einer kurzen Einführung in die Klassifikation. Dabei wird anhand eines kurzen Beispiels die Vorgehensweise der Klassifikation erklärt. Des weiteren wird auf interessante Zusatzliteratur zu diesem Thema verwiesen.

2.1 Data Mining und Knowledge Discovery in Databases

Knowledge Discovery ist der Prozess der Extraktion von Wissen aus Datenbanken, das

- *gültig* (im statistischen Sinne)
- bisher unbekannt und
- *potentiell nützlich* (für eine gegebene Anwendung) ist [FW99].

Die Idee dahinter ist, Algorithmen zu entwickeln, die in den vorhandenen Datenbanken (semi-)automatisch nach Regelmäßigkeiten und Mustern suchen. Häufig auftretende Muster, falls solche gefunden werden können, können genutzt werden, um Vorhersagen über zukünftige Daten zu treffen. Ein solcher Prozess ist auch mit Problemen verbunden. Gefundene Muster können banal und uninteressant sein. Andere wiederum sind zufällig entstanden. Hinzu kommt noch, dass die vorhandenen Daten in den Datenbanken meist nicht vollständig und fehlerfrei sind. Manche Werte können falsch sein und manche können ganz fehlen. Jedes gefundene Muster kann Ausnahmen beinhalten und gewisse Muster werden vielleicht gar nicht gefunden. Mit all diesen Problemen muss der KDD-Prozess fertig werden. Dieser muss sowohl unvollständige Daten verarbeiten können, ohne das Ergebnis zu verfälschen, als auch die gefundenen Muster nach ihrer Nutzbarkeit zu beurteilen. Dazu ist unter Umständen ein Eingriff des Benutzers nötig.

Laut Fayyad, Piatetsky-Shapiro und Smith [FPSS96] ist Data Mining ein Teilprozess des Knowledge Discovery. Data Mining ist das Herzstück eines umfassenderen Prozesses, nämlich den des Knowledge Discovery. Zusätzlich zu dem Data Mining Schritt umfasst der Prozess des Knowledge Discovery noch einige Schritte der Vorverarbeitung der zugrunde liegenden Daten und der Nachbearbeitung der gefundenen Muster (siehe Abbildung 3).

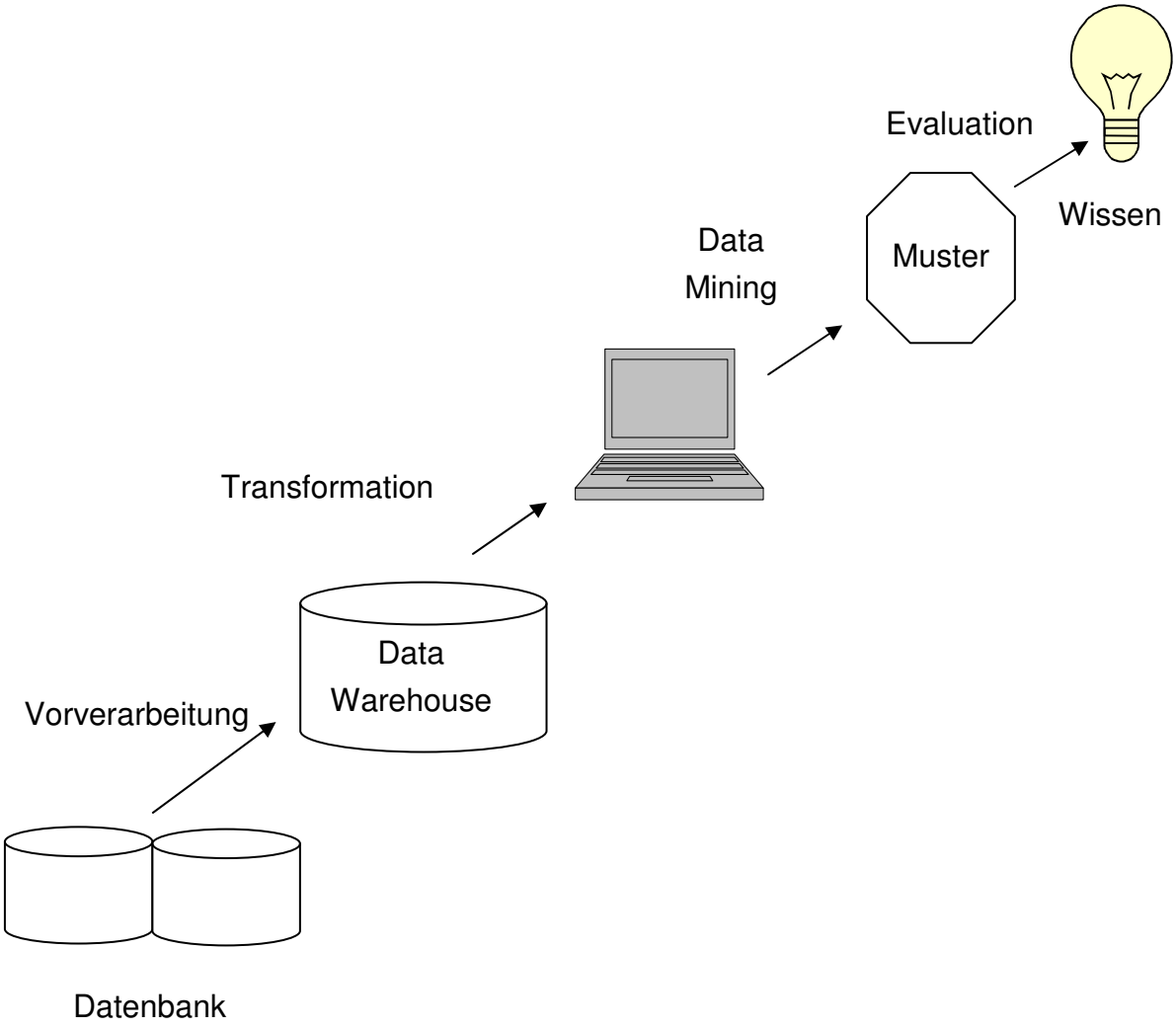


Abbildung 3: Data Mining als Herzstück des Prozesses des Knowledge Discovery

Die einzelnen Schritte werden im Folgenden kurz erläutert. Für eine genauere Behandlung der meisten KDD-Schritte vergleiche Frank und Witten [FW99]. Des Weiteren behandelt Pyle [PY99] sehr detailliert die Schritte der Vorverarbeitung und der Transformation.

Vorverarbeitung

Ziel der Vorverarbeitung ist es, die benötigten Daten zu integrieren, konsistent zu machen und zu vervollständigen. Dieser Schritt kann sehr zeitaufwendig sein und umfasst oft einen großen Teil des Gesamtaufwands eines KDD-Projekts. Um diesen Aufwand zu minimieren kann man auf einem Data Warehouse aufsetzen. Dies enthält die Daten bereits in integrierter und konsistenter Form.

Falls Daten aus verschiedenen Quellen gewonnen werden, müssen diese integriert werden, da verschiedene Quellen meist unterschiedliche Konventionen verwenden. Zum Beispiel können verschieden Werte, wie Umsätze, in der einen Quelle tageweise und in der anderen Quelle wochenweise gespeichert werden. Genauso müssen Inkonsistenzen behoben werden, die auftreten, wenn verschiedene Werte für dasselbe Attribut verwendet werden [HK00].

Transformation

In diesem Schritt der Transformation werden die bereits vorverarbeiteten Daten in eine für das Ziel des KDD geeignete Repräsentation transformiert. Typische Transformationen sind die Attribut-Selektion und die Diskretisierung von Attributen.

Normalerweise sind nicht alle Attribute der vorhandenen Daten für den Data-Mining Schritt relevant. Ein Beispiel für nicht relevante Attribute sind zum Beispiel Namen oder Telefonnummern, da diese in den meisten Anwendungsgebieten einmalig sind, und daher für eine Gruppierung nicht in Frage kommen. Dies ist allerdings stark abhängig von dem Anwendungsbereich. Eine solche Attributselektion erfolgt meist manuell, da oft nur der Anwender im Vorhinein entscheiden kann welches Attribut für die gegebene Anwendung relevant ist und welches nicht. Durch eine solche Attribut-Selektion wird meist die Effizienz des Data-Mining Algorithmus verbessert und die Qualität des Ergebnisses gesteigert. Umgekehrt kann sich die Qualität des Ergebnisses verschlechtern, wenn keine Attributselektion vorgenommen wird.

Für manche Data Mining Algorithmen ist eine Diskretisierung, d.h. eine Transformation von numerischen in kategorische Attribute, nötig, da diese nur kategorische Attribute verarbeiten können. Einfache Verfahren teilen den Wertebereich eines Attributs in Intervalle gleicher Länge oder in Intervalle mit gleicher Häufigkeit von enthaltenen Attributwerten.

Data Mining

Das Herzstück des KDD-Prozess ist der Data Mining Schritt. Dieser findet die gültigen Muster in der zugrunde liegenden Datenbank auf möglichst effiziente Art und Weise. Zuerst muss allerdings festgestellt werden, welches Data Mining Verfahren verwendet wird. Die wichtigsten Data Mining Verfahren werden in Abbildung 4 illustriert und im Folgenden kurz erläutert [ES00]:

- *Clustering / Entdecken von Ausreißern*

  Ziel des Clustering ist die Partitionierung einer Datenbank in Gruppen (Cluster) von Objekten, so dass Objekte eines Clusters möglichst ähnlich, Objekte verschiedener Cluster möglichst unähnlich sind. Ausreißer sind Objekte, die zu keinem der gefundenen Cluster gehören.

- *Klassifikation*

  Bei der Klassifikation sind Trainingsobjekte mit Attributwerten gegeben, die bereits einer Klasse zugeordnet sind. Es soll eine Funktion gelernt werden, die zukünftige Objekte aufgrund ihrer Attributwerte einer der Klassen zuweist.

- *Assoziationsregeln*

  Gegeben ist eine Datenbank von Transaktionen. Assoziationsregeln beschreiben häufig auftretende und starke Zusammenhänge innerhalb der Transaktionen wie z.B. WENN A UND B DANN C. Ein Beispiel für Transaktionen sind Einkäufe in einem Supermarkt. Jeder Einkauf (Transaktion) enthält eine bestimmte Menge von Produkten. Durch die Analyse mehrere Einkäufe kann ein Zusammenhang zwischen den Produkten gefunden werden. Zum Beispiel in der Form, dass bei einem Kauf von Brot und Wurst auch immer Butter gekauft wird.

- *Generalisierung*

  Ziel der Generalisierung ist es, eine Menge von Daten möglichst kompakt zu beschreiben, indem die Attributwerte generalisiert und die Zahl der Datensätze reduziert wird.

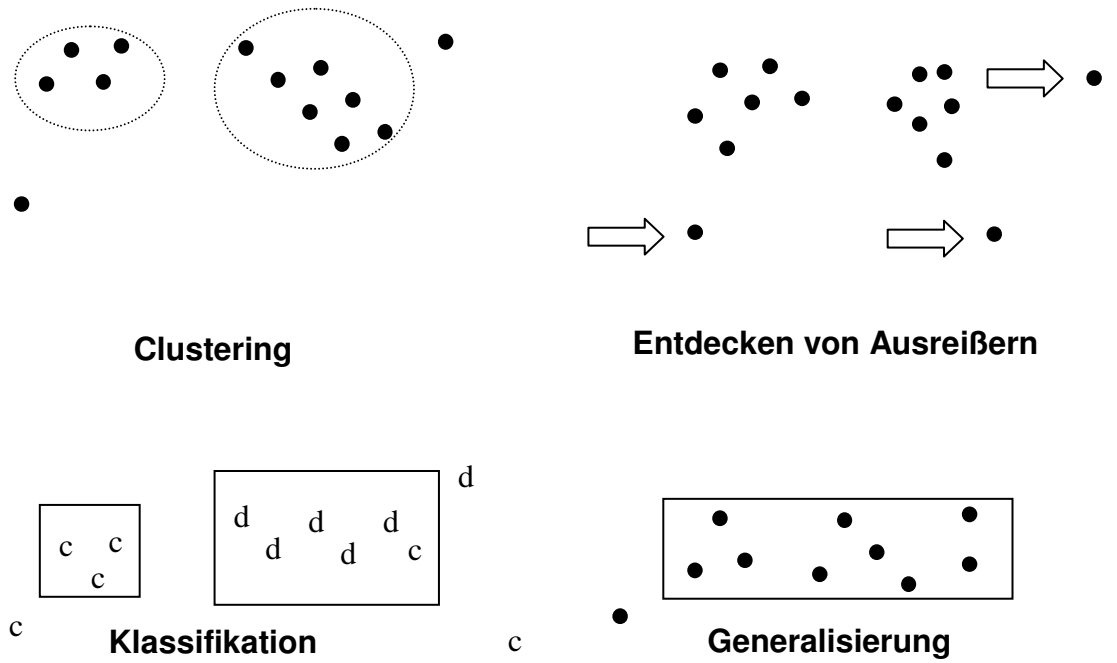


Abbildung 4: Die wichtigsten Data Mining Verfahren

Die Auswahl des Algorithmus eines Data Mining Verfahren erfolgt abhängig vom Ziel der Anwendung und dem Typ der Daten. Manche Algorithmen können nur bestimmte Daten, zum Beispiel nur Daten mit numerischen Attributen, verarbeiten.

Evaluation und Präsentation

Im letzten Schritt des KDD-Prozesses geht es darum, die gefundenen Muster in geeigneter Art zu präsentieren. Ob die Ergebnisse den definierten Zielen entsprechen, muss von einem Experten festgestellt werden. Ist dies nicht der Fall, muss unter Umständen der gesamte KDD-Prozess oder ein Teil des Prozesses, zum Beispiel der Data Mining Schritt, mit veränderten Parametern noch einmal durchgeführt werden. Falls das Ergebnis den definierten Zielen entspricht, wird das gewonnene Wissen in das bestehende System integriert und kann für neue KDD-Prozesse genutzt werden.

Für eine erfolgreiche Evaluierung der Ergebnisse ist die Präsentation der gefundenen Muster durch das System sehr entscheidend. Dies ist oft mit Schwierigkeiten verbunden, da entweder sehr viele Muster gefunden werden, vor allem bei den Assoziationsregeln, oder die Zahl der verwendeten Attribute sehr groß ist. Aus diesen Gründen ist eine Visualisierung der Ergebnisse für den Benutzer oft verständlicher als eine textuelle Darstellung.

In den folgenden Abschnitten werden die Data Mining Verfahren Clustering und Klassifikation beschrieben. Dabei werden vor allem beim Clustering die zwei Algorithmen K-Means und DBSCAN näher erläutert.

2.2 Clustering

Ziel von Clusteringalgorithmen ist es, Daten (semi-)automatisch so in Kategorien, Klassen oder Gruppen (Cluster) einzuteilen, dass Objekte im gleichen Cluster möglichst ähnlich und Objekte aus verschiedenen Cluster möglichst unähnlich zueinander sind [ES00].

Abbildung 5: Beispiele für 2-dimensionale Clusterstrukturen

Daraus ergibt sich, dass die Modellierung der Ähnlichkeit zwischen Datenobjekten erforderlich ist, um einen Clusteringalgorithmus sinnvoll anwenden zu können. Die beiden Hauptziele, Ähnlichkeit innerhalb eines Clusters und Unähnlichkeit zwischen den Clustern, sind allerdings nicht genug, um ein gutes Clustering zu gewährleisten. Denn diese Bedingungen können relativ trivial erfüllt werden, indem jedes Datenobjekt einen Cluster darstellt. Daher ist es von enormer Wichtigkeit, eine relativ geringe Anzahl von Clustern zu ermitteln, sodass möglichst viele Datenobjekte in einem Cluster vorkommen, natürlich unter der Berücksichtigung der Ähnlichkeit bzw. Unähnlichkeit. Eine der größten Herausforderungen des Clustering ist es, diese drei Ziele so zu erfüllen, dass das Ergebnis den gestellten Anforderungen entspricht.

Insbesondere im Kontext des Data Mining, der vor allem das Suchen von bisher unbekannten Mustern in Datenbanken beinhaltet, ist es wünschenswert, dass der Clusteringalgorithmus die Zahl der Cluster und deren Form automatisch ermittelt, basierend auf den zu clusternden Daten [FR02].

Des weiteren sollte bei der Auswahl und Anwendung eines Clusteringalgorithmus berücksichtigt werden, dass Cluster in den Daten sowohl unterschiedliche Größe,

Form und Dichte haben können, als auch hierarchisch ineinander verschachtelt sein können [ES00]. Abbildung 5 zeigt verschiedene Beispiele für Clusterstrukturen im zweidimensionalen Raum.

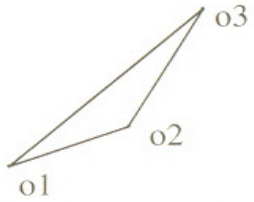
2.2.1 Ähnlichkeit zwischen Datenobjekten - Distanzfunktionen

Die Ähnlichkeit zwischen Datenobjekten spielt beim Clustering eine entscheidende Rolle. Meist wird diese Ähnlichkeit über eine Distanzfunktion *dist* modelliert, die für ein Paar von Datenobjekten definiert ist. Zur Definition einer solchen Distanzfunktion zwischen zwei Datenobjekten werden die Eigenschaften der Datenobjekte herangezogen, oder es werden Eigenschaften abgeleitet. Das Ergebnis einer Distanzfunktion zwischen zwei Datenobjekten wird wie folgt interpretiert:

- Kleine Distanzen ≈ ähnliche Datenobjekte
- Große Distanzen ≈ unähnliche Datenobjekte

Die konkrete Definition der Distanzfunktion *dist* hängt vom Datentyp der Objekte und der Anwendung ab. Unabhängig von der jeweiligen Form der Funktion *dist* müssen aber mindestens die folgenden Bedingungen für alle Objekte o_1, o_2 aus der Menge der betrachteten Objekte O gelten [ES00]:

1. $dist(o_1, o_2) = d \in R^{\geq 0}$,
2. $dist(o_1, o_2) = 0$ genau dann wenn $o_1 = o_2$,
3. $dist(o_1, o_2) = dist(o_2, o_1)$ (Symmetrie).



Die Funktion *dist* ist eine Metrik, wenn zusätzlich die Dreiecksungleichung gilt, d.h. wenn für alle o_1, o_2, $o_3 \in O$ gilt:

4. $dist(o_1, o_3) \leq dist(o_1, o_2) + dist(o_2, o_3)$.

Das Clustering wird auch oft als „Distanzgruppierung“ bezeichnet (vergleiche [ES00]), da die Ähnlichkeit zwischen den Datenobjekten über die Distanz zwischen den Objekten ermittelt wird. Alternativ zu einer Distanzfunktion werden hin und wieder auch so genannte Ähnlichkeitsfunktionen (im Englischen „similarity function) verwendet. Dabei gilt für eine Ähnlichkeitsfunktion $sim(o_1, o_2)$, je ähnlicher die Datenobjekte, desto größer der Wert $sim(o_1, o_2)$. Dies ist also genau umgekehrt zur Distanzfunktion, wo der Wert $dist(o_1, o_2)$ möglichst klein sein soll, damit Objekte ähnlich sind. Für

Clusteringalgorithmen hat eine Änderung von einer Distanzfunktion auf eine Ähnlichkeitsfunktion zur Ermittlung der Ähnlichkeit nur geringe Auswirkungen, da normalerweise nur die Vergleichsoperatoren umgedreht werden müssen.

Distanzfunktionen

Der Datentyp der verwendeten Datenobjekte und die konkrete Anwendung des Data Mining sind entscheidend für die Auswahl einer Distanzfunktion. Nachfolgend werden einige typische Distanzfunktionen für verschiedene Datentypen angeführt, die für die jeweiligen Datentypen häufig angewendet werden.

- Für Datensätze $x = (x_1,...,x_n)$ mit *numerischen Attributwerten* x_i:

  Euklidische Distanz: $dist(x, y) = ((x_1 - y_1)^2 + ... + (x_n - y_n)^2)^{1/2}$,

  Manhatten Distanz: $dist(x, y) = | x_1 - y_1| + ... + | x_n - y_n|$,

  Maximums Metrik: $dist(x, y) = max(| x_1 - y_1|, ..., | x_n - y_n|)$

  Minkowski Distanz: $dist(x, y) = (| x_1 - y_1|^q + ... + | x_n - y_n|^q)^{1/q}$

- Für Datensätze $x = (x_1,...,x_n)$ mit *kategorischen Attributwerten* x_i:

  Anzahl der verschiedenen Komponenten in x und y: $dist(x, y) = \sum_{i=1}^{d} \delta(x_i, y_i)$

  wobei $\delta(x_i, y_i) = \begin{cases} 0 \text{ falls } x_i = y_i \\ 1 \text{ sonst} \end{cases}$

- Für endliche Mengen $x = \{x_1,...,x_n\}$:

  Anteil der verschiedenen Elemente in x und y: $dist(x, y) = \frac{|x \cup y| - |x \cap y|}{|x \cup y|}$

- Für Datensätze $x = (x_1,...,x_n)$ mit *binären Attributwerten* 0 (=falsch) und 1 (=wahr):

  Grundlage für die Definition einer Distanzfunktion für binäre Attribute ist meist eine Unähnlichkeitsmatrix.

		Object j		
		1	0	Sum
Object i	1	q	r	q + r
	0	s	t	s + t
	Sum	q + s	r + t	p

Tabelle 1: Unähnlichkeitsmatrix für 2 binäre Datenobjekte

Aus dieser Matrix lässt sich eine Distanzfunktion ableiten, allerdings muss dabei zwischen symmetrischen und asymmetrischen binären Attributen unterschieden werden. Ein binäres Attribut ist symmetrisch wenn beide Werte, falsch und wahr, dieselbe Gewichtung und denselben Stellenwert haben. Ein Beispiel hierfür wäre das Geschlecht. Bei asymmetrischen binären Attributen hingegen sind die Ergebnisse der Werte nicht gleich wichtig, d.h. einem Wert, falsch oder wahr, wird eine höhere Bedeutung beigemessen. Ein HIV – Test wäre ein Beispiel für ein asymmetrisches Attribut.

- Symmetrisch: Simple matching coefficient: $d(i, j) = (r + s) / (q + r + s + t)$
- Asymmetrisch: Jaccard coefficient: $d(i, j) = (r + s) / (q + r + s)$

Unähnlichkeitsmatrix

Clusteringalgorithmen arbeiten oft nicht direkt mit den oben angeführten Funktionsgleichungen, sondern mit einer Unähnlichkeitsmatrix.

- Unähnlichkeitsmatrix [HK00]:

  Die Unähnlichkeitsmatrix speichert Näherungswerte die für alle Paare von n Objekten verfügbar sind. Diese wird of als n x n Tabelle dargestellt:

$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & & \\ d(n,1) & d(n,2) & \cdots & \cdots & 0 \end{bmatrix}$$

  Die Funktion *d(x,y)* kann eine der oben angeführten Distanzfunktionen oder eine Unähnlichkeitsfunktion sein. Dabei gilt, dass sich zwei Objekte ähnlich

sind wenn der Wert der Funktion *d(x,y)* nahe 0 ist. Je unterschiedlicher die beiden Objekte sind, desto größer wird der Wert für die Funktion *d(x,y)*.

2.2.2 Partitionierende Clusteringverfahren

Ein partitionierendes Clusteringverfahren zerlegt die gegebene Datenmenge von n Datenobjekten in k Cluster, wobei jedes Cluster k eine Punktmenge kleiner gleich n beinhaltet. Dabei müssen folgende Bedingungen erfüllt sein:

1. Jeder Cluster enthält mindestens ein Datenobjekt

2. Jedes Datenobjekt gehört zu maximal einem Cluster

Die Datenobjekte eines Clusters sollen dabei möglichst „ähnlich“ sein. Diese Ähnlichkeit wird meist über eine der oben angeführten Distanzfunktionen ermittelt. Datenobjekte von verschiedenen Clustern sollten hingegen möglichst „unähnlich“ sein.

2.2.2.1 Iterative partitionierende Clusteringverfahren – K-Means

Der K-Means Algorithmus ist einer der bekanntesten und am häufigsten eingesetzten Clusteringalgorithmen. Ziel des K-Means Algorithmus ist es die gegebene Punktmenge in k Klassen zu teilen, und zwar so, dass die oben angeführten Prämissen, bezüglich der Clusterzugehörigkeit und der Ähnlichkeit, erfüllt sind, d.h. Punkte eines Clusters sind „ähnlich“ und Punkte von verschiedenen Clustern „unähnlich“. Der Parameter k ist ein Eingabeparameter und wird vom Benutzer vorgegeben.

Beim K-Means Algorithmus wird die Ähnlichkeit über eine Distanzfunktion ermittelt, wobei immer vom Mittelpunkt des Clusters, dem so genannten Centroid ausgegangen wird. Meist wird die Euklidische Distanz als Distanzfunktion verwendet. Der Centroid eines Clusters wird wie folgt ermittelt:

$$\mu_C = (\bar{x}_1(C), \bar{x}_2(C), \ldots, \bar{x}_d(C)), \text{ wobei } \bar{x}_j(C) = \frac{1}{n_C} \cdot \sum_{p \in C} x^p{}_j$$

der Mittelwert der j-ten Dimension aller Punkte in C ist; n_c ist die Anzahl der Objekte in C.

Vorgehensweise des K-Means Algorithmus

Zuerst werden zufällig k Objekte aus der gesamten Datenmenge ausgewählt. Jeder dieser ausgewählten k Objekte entspricht einem Centroid, d.h. einem Clustermittelpunkt. Anschließend werden die übrigen Objekte dem ähnlichsten Centroid zugeordnet, basierend auf der Distanz zwischen dem Objekt und dem Centroid. Nachdem ein Objekt einem Centroid zugeordnet wurde, wird der Centroid des zugehörigen Clusters neu berechnet. Der Algorithmus wird beendet, wenn sich die Clusterzugehörigkeit von keinem Objekt mehr ändert.

Algorithmus: K-Means. Die gegebene Menge der Objekte wird anhand der Distanz zum Clustermittelpunkt (Centroid) in k Cluster partitioniert.

Input: Die Anzahl der Cluster k und die Datenbank mit den n Objekten.

Output: Eine Menge von k Clustern mit ihren zugehörigen Objekten.

Methode:

(1) Zufällige Auswahl von k Objekten als initiale Centroide

(2) Wiederhole

a. Zuordnung der Objekte zu ihrem ähnlichsten Centroid

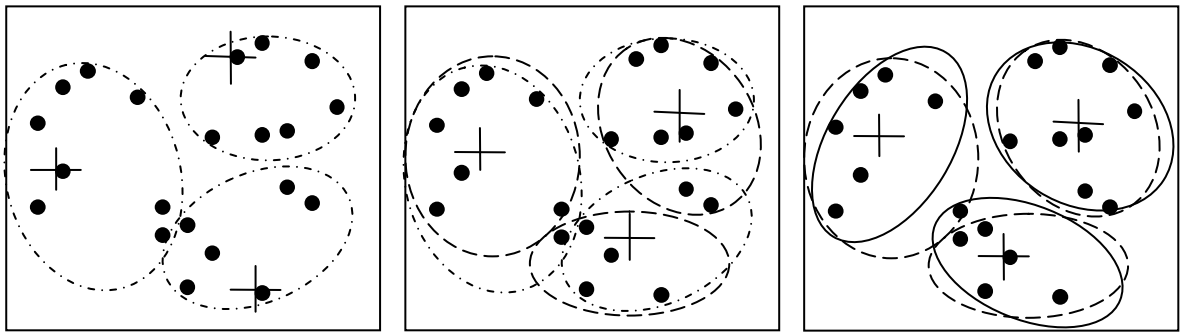
b. Wenn sich die Zuordnung des Objekts geändert hat werden die Centroide der geänderten Cluster neu ermittelt

(3) Bis keine Zuordnung mehr geändert wird

Beispiel K-Means

Abbildung 6 (a) zeigt die gegebene Punktemenge, die geclustert werden soll. Der Parameter k wird vom Benutzer mit 3 angegeben, d.h. die Punktmenge soll in 3 Cluster geteilt werden. Bezugnehmend auf den oben angeführten Algorithmus, werden 3 Objekte als initiale Clustermittelpunkte (Centroide) zufällig ausgewählt. Die ausgewählten Centroide werden in der Abbildung als „+“ dargestellt. Jedes Objekt wird seinem nähesten Centroid zugeordnet, dabei werden nach jeder Zuordnung die Centroide neu ermittelt. Abbildung 6 (a) zeigt die Cluster nach der ersten Iteration.

Abbildung 6 (b) zeigt die Ergebnisse der nächsten beiden Iterationen mit den zugehörigen Centroiden nach der dritten Iteration. In Abbildung 6 (c) wird das Ergebnis des K-Means Algorithmus mit der gegebenen Punktmenge nach fünf Iterationen dargestellt. Die Cluster mit der durchgezogenen Linie bilden dabei das Endergebnis, das der Algorithmus zurückliefert.



(a) (b) (c)

Abbildung 6: Beispiel für K-Means Algorithmus

Eigenschaften des Algorithmus

- Konvergiert gegen ein (möglicherweise nur lokales) Minimum
- Anzahl der Iterationen ist im allgemeinen klein (~5 – 10)
- Ergebnis und Laufzeit hängen stark von der initialen Zerlegung ab
- Aufwand: O(ndkt), wobei *n* die Anzahl der Objekte, *d* die Anzahl der Dimensionen, *k* die Zahl der zu findenden Cluster und *t* die Anzahl der Iterationen ist

Weitere Beispiele für iterativ partitionierende Algorithmen sind PAM und CLARANS (vergleiche [ES00]). Allerdings haben beide Algorithmen, vor allem PAM, eine sehr hohe Laufzeit im Vergleich zu K-Means. Daher wird in den meisten Fällen K-Means den beiden Algorithmen vorgezogen.

2.2.2.2 Dichte-basiertes partitionierendes Clustering – DBSCAN

Der bisher vorgestellte partitionierende Clusteringalgorithmus K-Means arbeitet, so wie viele partitionierende Clusteringalgorithmen, mit der Distanz zwischen Objekten.

Solche Algorithmen finden nur kreisförmige Cluster und haben Schwierigkeiten, damit andersförmige Cluster zu finden (siehe auch Abbildung 5).

Beim dichte-basierten Clustering werden die Cluster als Gebiete im d-dimensionalen Raum angesehen, in denen die Objekte dicht beieinander liegen, getrennt durch Gebiete, in denen die Objekte weniger dicht liegen [ES00]. Die Grundidee eines dichte-basierten Clustering ist, dass ein gegebener Cluster so lange erweitert wird bis die Dichte (Anzahl von Objekten) seiner „Nachbarschaft" einen vorgegebenen Schwellenwert nicht mehr überschreitet, d.h. die Punkte innerhalb des Clusters müssen diesen Schwellenwert überschreiten. Die „Nachbarschaft" eines jeden Objekts muss innerhalb eines vorgegebenen Radius (ε) zumindest eine minimale Anzahl (MinPts) von Punkten besitzen. Eine solche Methode kann verwendet werden, um so genannte Ausreißer, auch als Rauschen bezeichnet, auszuschließen, und vor allem um Cluster jeglicher Form zu finden. DBSCAN ist ein typischer dichte-basierter Algorithmus, der nach der oben angeführten Grundidee vorgeht und daher repräsentativ unter den dichte-basierten Algorithmen ist. Weitere dichte-basierte Algorithmen sind OPTICS und DENCLURE (vergleiche auch [HK00]).

Die Grundidee des dichte-basierten Clustering beinhaltet auch eine Reihe von neuen Definitionen. Diese Definitionen werden präsentiert und anschließend anhand eines Beispiels erklärt.

- **Definition 1:** Die Nachbarschaft eines Objekts innerhalb eines Radius ε wird als **ε-Nachbarschaft** (ε-neighborhood) oder ε-Umgebung des Objekts bezeichnet [HK00]

- **Definition 2:** Wenn die ε -Nachbarschaft eines Objekts zumindest die minimale Anzahl von Punkten, MinPts, enthält, dann ist das Objekt ein **Kernobjekt** [HK00]**.** Die Werte ε und MinPts sind dabei Parameter, die einen minimalen Dichtegrenzwert spezifizieren. Der Abstand zwischen den Objekten wird über eine Distanzfunktion dist(p,q) ermittelt (siehe Kap. 2.2.1)

$$|N_\varepsilon(o)| \geq MinPts, \text{ wobei } N_\varepsilon(o) = \{o' \in O \mid dist(o, o') \leq \varepsilon\}.$$

- **Definition 3:** Bei einer gegebenen Punktemenge O, ist ein Objekt p **direkt dichte-erreichbar** von einem Objekt q, wenn p innerhalb der ε-Nachbarschaft

von q liegt, und q ein Kernobjekt ist. Alle Objekte innerhalb der ε-Nachbarschaft eines Objekts q sind direkt dichte-erreichbar von q [EKSX96].

- **Definition 4:** Ein Objekt p ist **dichte-erreichbar** von einem Objekt q bezüglich ε und MinPts in der Menge von Objekten O, wenn es eine Folge von Objekten $p_1,..., p_n$ in O gibt, so dass $p_1 = q$, $p_n = p$ ist, und es gilt: p_{i+1} ist direkt dichte-erreichbar von p_i bezüglich ε und MinPts in O für $1 \leq i \leq n$ [ES00].

- **Definition 5:** Ein Objekt p ist **dichte-verbunden** mit einem Objekt q bezüglich ε und MinPts in einer Menge von Objekten O, wenn es ein o aus der Menge O gibt, so dass sowohl p als auch q dichte-erreichbar bezüglich ε und MinPts von o sind [ES00]

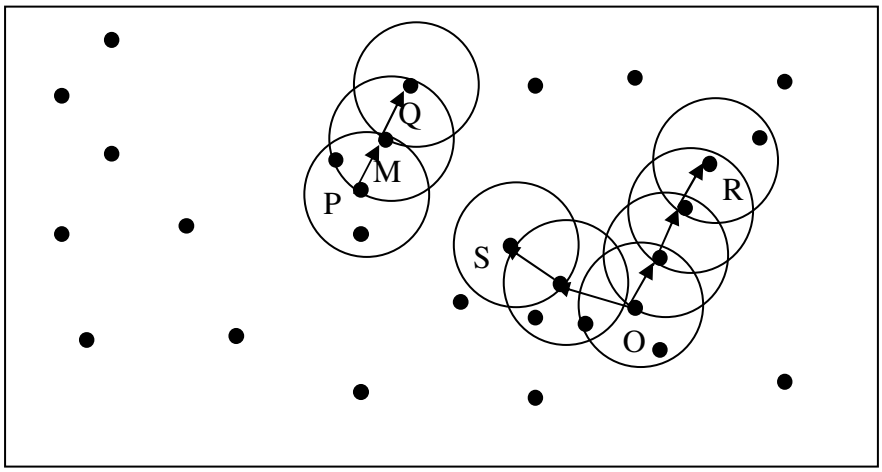


Abbildung 7: Dichte-Erreichbarkeit und Dichte-Verbundenheit beim dichte-basierten Clustering

In Abbildung 7 wird ε durch den Radius der Kreise dargestellt. Der Parameter MinPts wird in den nachfolgenden beispielhaften Erklärungen, der oben angeführten Definitionen, mit 3 angenommen.

1. Von den gekennzeichneten Punkten sind M, P, O und R Kernobjekte, da jeder innerhalb seiner ε-Nachbarschaft zumindest 3 Punkte hat

2. Q ist direkt dichte-erreichbar von M. M ist direkt dichte-erreichbar von P und vice versa.

3. Q ist (indirekt) dichte-erreichbar von P, da Q direkt dichte-erreichbar von M und direkt dichte-erreichbar von P ist. P ist hingegen nicht dichte-erreichbar von Q, da Q kein Kernobjekt ist. Gleichermaßen sind R und S dichte-erreichbar von O und O dichte-erreichbar von R.

4. O, R und S sind alle dichte-verbunden.

- **Definition 6:** Ein **dichte-basierter Cluster** C ist eine Menge von Objekten, die alle miteinander dichte-verbunden sind (Verbundenheit), und alle Objekte die von einem Kernobjekt des Clusters dichte-erreichbar sind (Maximalität) gehören auch zum Cluster.

- **Definition 7:** Jedes der Objekte, dass nicht zu einem der dichte-basierten Cluster gehört, wird als „**Rauschen**“ bezeichnet

- **Definition 8:** Ein dichte-basiertes Clustering CL der Menge O bezüglich ε und MinPts ist eine Menge von dichte-basierten Clustern bezüglich ε und MinPts in O, CL = $\{C_1,..., C_k\}$, so dass für alle C gilt: wenn C ein dichte-basierter Cluster bezüglich ε und MinPts in O ist, dann ist schon C $\in$ CL.

Vorgehensweise des DBSCAN-Algorithmus

Der Algorithmus DBSCAN (Density Based Spatial Clustering of Applications with Noise) liefert, ausgehend von den oben angeführten Definitionen, bei gegebenen Parametern ε und MinPts einen Cluster iterativ in zwei Schritten. Zuerst wird zufällig ein Objekt aus der gesamten Datenmenge ausgewählt, das die Bedingungen eines Kernobjekts erfüllt. Anschließend werden im zweiten Schritt alle Objekte, die von diesem ausgewählten Objekt aus dichte-erreichbar sind, dem Cluster des ausgewählten Objekts zugeordnet. Diese beiden Schritte werden wiederholt bis alle Punkte entweder Clustern oder dem Rauschen zugeordnet sind.

Nachfolgend wird eine Grundversion des DBSCAN-Algorithmus dargestellt, ohne auf genaue Details wie Datentypen usw. einzugehen [EKSX96]:

```
DBSCAN (Objektmenge O, Real ε, Integer MinPts)
    // Zu Beginn sind alle Objekte unklassifiziert, d.h.
    // o.ClId = UNKLASSIFZIERT für alle o aus der Objektmenge O
    ClusterId := nextId(NOISE);
    for i from 1 to |O| do
        Objekt := O.get(i);
        if Objekt.ClId = UNKLASSIFZIERT then
            if ExpandiereCluster(O, Objekt, ClusterId, ε, MinPts)
                then ClusterID := nextId(ClusterId);

ExpandiereCluster(Objektmenge O, Objekt StartObjekt, Integer ClusterId,
            Real ε, Integer MinPts) : Boolean;
    Seeds := ε-Nachbarschaft(StartObjekt);
    if |seeds| < MinPts then    // StartObjekt ist kein Kernobjekt
        StartObjekt.ClId := NOISE;
        return false;
    // sonst: StartObjekt ist ein Kernobjekt
    for each o in seeds do o.ClId := ClusterId;
    entferne StartObjekt aus seed;
    while seeds != {} do
        whäle ein Objekt o aus der Menge seeds;
        Nachbarschaft := ε-Nachbarschaft(o);
        if |Nachbarschaft| >= MinPts then     // o ist ein Kernobjekt
            for i from 1 to |Nachbarschaft| do
                p := Nachbarschaft.get(i);
                if p.ClId in {UNKLASSIFIZIERT, NOISE} then
                    if p.ClId == UNKLASSIFZIERT then
                        füge p zur Menge seeds hinzu;
                    p.ClId := ClusterId;
        entferne o aus der Menge seeds;
    return true;
```

Dem Algorithmus werden die zu clusternde Datenmenge O und die Parameter ε und MinPts, zur Bestimmung der Dichte, übergeben. Zu Beginn sind alle Objekte unklassifziert. In der Methode *DBSCAN* wird die gegebene Datenmenge linear durchgegangen und ausgehend von jedem noch unklassifzierten Objekt wird versucht einen Cluster zu ermitteln. Dies übernimmt die Methode *ExpandiereCluster*.

Die Methode *ExpandiereCluster* ermittelt zuerst die ε-Nachbarschaft des übergebenen Objekts. Anhand der ε-Nachbarschaft wird festgestellt, ob das Objekt ein Kernobjekt ist oder nicht. Falls es kein Kernobjekt ist wird dieses Objekt (vorläufig) dem Rauschen zugeordnet, und die Methode liefert den Wert *false* zurück. Dem Rauschen zugeordnete Objekte können später noch als Randobjekte klassifiziert werden.

Falls das Objekt ein Kernobjekt ist, dann wird ausgehend von diesem Objekt ein Cluster ermittelt, indem zuerst alle von ihm aus dichte-erreichbaren Objekte der Datenmenge gesucht werden. Alle Objekte die innerhalb der ε-Nachbarschaft liegen gehören auf jeden Fall zum Cluster, auf Grund der direkten Dichte-Erreichbarkeit. Die von diesen Objekten wiederum direkt dichte-erreichbaren Objekte gehören auch zum Cluster. Diese Vorgehensweise wird so lange fortgeführt, bis keine dichte-erreichbaren Objekte mehr gefunden werden.

Die Methode ExpandiereCluster ermittelt den gesamten Cluster durch iterative Berechnung der direkten Dichte-Erreichbarkeit zum ausgewählten Startobjekt. Alle gefunden Objekte werden dem Cluster über die Cluster-Id zugeordnet.

Wenn zwei Cluster C1 und C2 sehr nahe beieinander liegen, kann es vorkommen, dass gewisse Objekte o zu beiden Clustern, C1 und C2, gehören. In diesem Fall ist o ein Randpunkt, anderenfalls würden die Cluster C1 und C2, auf Grund der Dichte-Erreichbarkeit, verschmelzen. Das Objekt o wird dem zuerst ermittelten Cluster zugeordnet.

Eigenschaften des Algorithmus

- DBSCAN berechnet ein dichte-basiertes Clustering und die Menge des „Rauschen“ gemäß den Definitionen 1 bis 8 (siehe oben), bis auf mehrfache Zuordnung von Randpunkten, falls diese zu mehreren Clustern gehören. In diesem Fall wird der Randpunkt dem ersten gefundenen Cluster, zu dem er gehört, zugeordnet.

- Bis auf die Zuordnung von Randpunkten, die zu mehreren Clustern gehören können, ist das Ergebnis unabhängig von der Reihenfolge der Daten.

- Aufwand: O(n * Aufwand zur Bestimmung einer ε-Nachbarschaft). Wenn kein räumliches Indexverfahren verwendet wird, dann ist der Aufwand $O(n^2)$. Mit der Verwendung eines R*-Baums (siehe auch Kap. 4.3.2) bei der Ermittlung der ε-Nachbarschaft ist der Aufwand nur noch O(n * log n).

2.2.3 Hierarchische Clusteringverfahren

Im Gegensatz zu partitionierenden Verfahren erzeugen hierarchische Clusteringverfahren keine einfache Zerlegung der Datenmenge, sondern eine

hierarchische Repräsentation der Daten, aus der man eine Clusterstruktur ableiten kann [ES00].

Hierarchische Clusteringverfahren können in „Top-Down"-Verfahren und „Bottom-Up"-Verfahren unterteilt werden. „Bottom-Up"-Verfahren beginnen damit, dass jedes Objekt einem Cluster zugeordnet wird, d.h. jedes Objekt einen Cluster darstellt. Anschließend werden iterativ die zwei ähnlichsten (nähesten) Cluster verschmolzen, bis letztendlich nur noch ein Cluster vorhanden ist, der alle Objekte enthält. „Top-Down"-Verfahren arbeiten genau umgekehrt. Sie beginnen damit, alle Objekte einem Cluster zuzuordnen. Dann wird dieser Cluster iterativ in immer kleiner Cluster unterteilt. Dies wird so lange fortgesetzt, bis entweder alle Objekte einen Cluster präsentieren, oder bis eine gewisse Terminierungsbedingung erfüllt ist.

Zusätzlich zu den paarweisen Distanzen zwischen zwei Objekten (siehe Kap. 2.2.1) benötigen hierarchische Clusteringalgorithmen auch Distanzfunktionen, die den Abstand zwischen zwei Mengen von Objekten ermitteln können. Um diese Distanz zwischen zwei Objektmengen X und Y zu ermitteln, betrachtet man üblicherweise die Distanz zwischen den Objekten aus den beiden Objektmengen X und Y. Dazu gibt es einige Ansätze, wie zum Beispiel die kleinste, die größte oder die durchschnittliche Distanz [ES00], [HK00]:

- **Minimum-Distanz:** $$dist\text{-}sl(X, Y) = \min_{x \in X, y \in Y} dist(x, y).$$
- **Maximum-Distanz:** $$: dist\text{-}cl(X, Y) = \max_{x \in X, y \in Y} dist(x, y).$$
- **Durchschnitts-Distanz:** $$dist\text{-}al(X, Y) = \frac{1}{|X| \cdot |Y|} \cdot \sum_{x \in X, y \in Y} dist(x, y).$$
- **Mittelpunkt-Distanz**: dist-mean(X,Y) = $|m_x - m_y|$

Diese Distanzfunktionen sind so ausgelegt, dass jeder hierarchische Clusteringalgorithmus diese zur Berechnung des Clustering nutzen kann.

Das Ergebnis eines hierarchischen Clusteringalgorithmus wird meist in der Form eines Dendogramms dargestellt. Ein Dendogramm ist ein Baum, der die hierarchische Zerlegung der Datenmenge in immer kleinere Teilmengen darstellt. Die Wurzel des Baums repräsentiert die gesamte Datenmenge in Form eines Clusters. Die Blätter des Baums stellen einzelne Objekte als Cluster dar. Innere Knoten repräsentieren die Vereinigung aller ihrer Kindknoten.

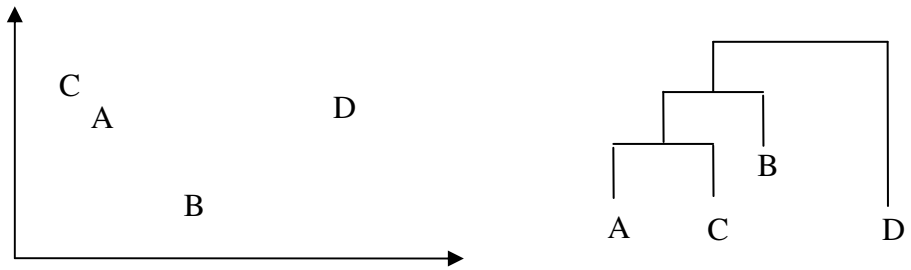


Abbildung 8: Links wird eine zweidimensionale Datenmenge mit 4 Objekten dargestellt; Rechts ein Dendogramm erzeugt durch ein "Bottom-Up" Clustering

Die Kanten zwischen den Knoten sollen die Distanz zwischen den Knoten darstellen. Dendogramme werden üblicherweise in graphischer Form dargestellt. Abbildung 8 zeigt ein Dendogramm als Ergebnis eines hierarchischen Clusteringalgorithmus. Einige bekannte hierarchische Clusteringalgorithmen sind Single-Link, OPTICS, BIRCH oder CURE. Für eine genaue Behandlung der Algorithmen vergleiche auch Ester und Sander [ES00] oder Han und Kamber [HK00].

2.3 Klassifikation

Im Gegensatz zum Clustering sind bei der Klassifikation die Klassen schon a priori in der Datenbank bekannt. Aufgabe der Klassifikation ist es, Objekte aufgrund ihrer Attributwerte einer der vorgegebenen Klassen zuzuordnen. Ausgegangen wird dabei von einer Menge von Trainingsobjekten mit Attributwerten, die bereits einer Klasse zugeordnet sind. Mit Hilfe der Trainingsdaten soll eine Funktion gelernt werden, der so genannte Klassifikator, die andere Objekte mit unbekannter Klassenzugehörigkeit aufgrund ihrer Attributwerte einer der Klassen zuweist [ES00]. Der Klassifikator muss bewertet werden, um seine Leistungsfähigkeit zu ermitteln. Als ein wichtiges Leistungsmaß bietet sich der Klassifikationsfehler an, d.h. der Anteil der Objekte die falsch klassifziert werden. Dazu werden Daten benötigt, um den Klassifikationsfehler schätzen zu können. Bei der Verwendung der Trainingsdaten selbst erhält man im Allgemeinen viel zu kleine Werte für den Klassifikationsfehler. Der gelernte

Klassifikator ist nämlich für die Stichprobe der Trainingsdaten optimiert, liefert aber häufig auf der Grundgesamtheit aller Daten wesentlich schlechtere Ergebnisse. Dieser Effekt wird als *Overfitting* bezeichnet. Stattdessen kann die Menge aller Objekte in Trainings- und Testdaten geteilt werden. Diese beiden Mengen müssen disjunkt sein, um einen „guten" Klassifikationsfehler zu erhalten. Diese Methode wird auch als *Train and Test* bezeichnet. Abbildung 9 zeigt ein Beispiel für eine Klassifikation mit Trainings- und Testdaten.

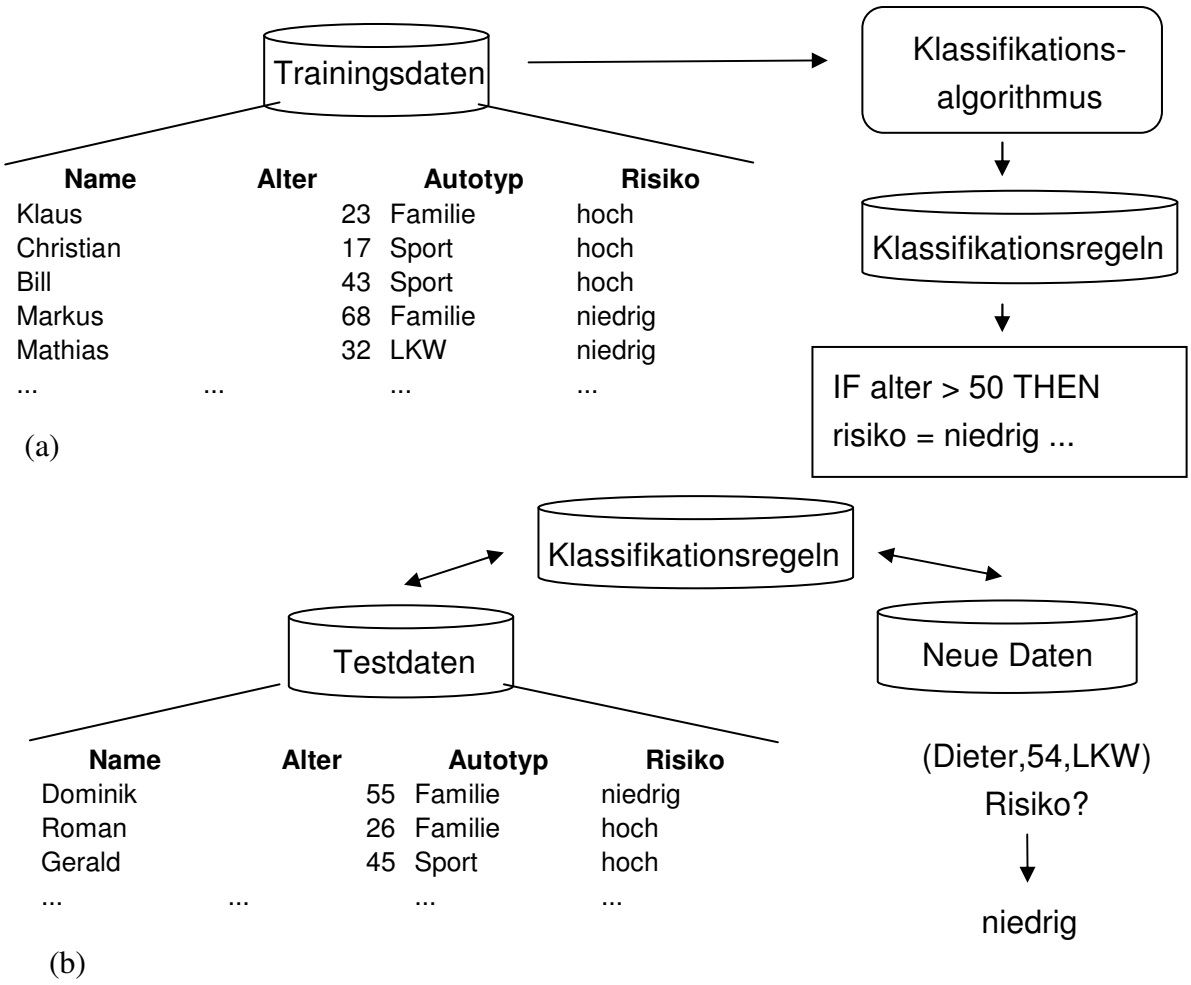


Abbildung 9: Beispiel für eine Klassifikation

In Abbildung 9 (a) werden die Trainingsdaten von einem Klassifikationsalgorithmus analysiert. In diesem Beispiel ist das Attribut Risiko das Klassenattribut. Es enthält die Werte (=Klassen) *niedrig* und *hoch*. Das Ergebnis der Klassifikation wird in Form von Klassifikationsregeln (Klassifikator) dargestellt. In Abbildung 9 (b) wird der gelernte Klassifikator mit Testdaten auf seine Genauigkeit getestet. Wenn die Genauigkeit einen gewünschten Wert erreicht, kann der Klassifikator für neue Datenobjekte

herangezogen werden. Die rechte Seite der Abbildung 9 (b) zeigt die Anwendung des Klassifikators auf neue Daten.

Die Aufgabe der Klassifikation lässt sich in zwei Teilaufgaben zerlegen. Die erste Teilaufgabe ist die Zuordnung von Objekten zu einer Klasse. Dies erfolgt aufgrund der Attributwerte eines Objekts und lässt sich allein mit der Hilfe von implizitem Wissen lösen. Die zweite Teilaufgabe besteht darin, Klassifikationswissen zu generieren, d.h. es soll explizites Wissen über die Klassen ermittelt werden.

Es gibt viele verschiedene Ansätze von Klassifikationsverfahren, zum Beispiel Bayes-Klassifiaktoren, Nächste-Nachbarn-Klassifikatoren oder Entscheidungsbaum-Klassifikatoren. Die Bayes-Klassifikatoren beruhen auf der Bestimmung bedingter Wahrscheinlichkeiten der Attributwerte für die verschiedenen Klassen. Nächste-Nachbarn-Klassifikatoren verzichten auf das Finden von explizitem Wissen und arbeiten stattdessen direkt auf den Trainingsdaten. Entscheidungsbaum-Klassifikatoren liefern explizites Wissen zur Klassifikation in Form von Entscheidungsbäumen. Genaueres zu den einzelnen Verfahren und zu den Algorithmen ist unter Ester und Sander [ES00], Han und Kamber [HK00], Freitas [FR02] oder Humer [HM04] zu finden.

Die Klassifikation von Texten ist ein häufiges Anwendungsgebiet der Klassifikation. Dabei sollen Texte automatisch klassifiziert werden, um diese einem gewissen Zweck zuordnen zu können. Zum Beispiel können so Emails in einer Firma klassifiziert werden, um dann dem richtigen Sachbearbeiter zugeordnet zu werden. Eine andere zunehmend wichtige Anwendung ist die Klassifikation von Webseiten für Zwecke der automatischen Indizierung [ES00]. Ein weiteres Anwendungsgebiet ist die Klassifikation von Sternen anhand der gesammelten Daten der Radioteleskope. Die Verwendung von Entscheidungsbäumen zur Klassifikation von Daten ist ebenfalls verbreitet und kann praktisch auf jede Datenmenge angewendet werden.

3 Kombiniertes Data Mining

Nachdem Kapitel 2 zuvor in die Grundlagen des Data Mining und der Data Mining Verfahren Clustering und Klassifikation eingeführt hat, beschäftigt sich dieses Kapitel mit dem „Kombinierten Data Mining“. Dazu wird der Begriff „Kombiniertes Data Mining“ zu allererst durch die nachfolgende Definition definiert:

- **Definition:** Beim „Kombinierten Data Mining“ wird ein Data Mining Verfahren *A* vor einem Data Mining Verfahren *B* ausgeführt, sodass *B* von *A* profitiert. *B* kann dazu das Ergebnis von *A* oder/und eigens ermittelte Hilfsinformationen von *A* für *B* nutzen. *B* profitiert dann von *A* wenn das Ergebnis von *B*, gemäß einem geeigneten Gütemaß, besser ist, oder/und sich die Laufzeit von *B* verringert.

Wie der Name „Kombiniertes Data Mining“ schon verrät, werden hier verschiedene Data Mining Verfahren hintereinander ausgeführt. Dafür kann es viele verschiedene Gründe geben. Zum Beispiel kann ein Clusteringverfahren genutzt werden um Trainingsdaten für ein Klassifikationsverfahren zu generieren. Umgekehrt kann ein Klassifikationsverfahren auf das Ergebnis eines Clusteringverfahrens angewendet werden, um dieses Ergebnis besser darstellen und interpretieren zu können. Dies ist oft der Fall, da für das Ergebnis eines Clustering die geeigneten graphischen und textuellen Darstellungsformen fehlen. Hingegen kann das Klassifikationsergebnis, zum Beispiel an hand eines Entscheidungsbaums, gut dargestellt werden. Ein weiteres Beispiel für die Anwendung des „Kombinierten Data Mining“ ist dann gegeben, wenn eine große Zahl von Klassen für eine Klassifizierung vorhanden ist und diese mit einem Clusteringverfahren zu Gruppen zusammengefasst werden, um die Klassifizierung zu vereinfachen. Bestehende Ansätze des „Kombinierten Data Mining“ und mögliche Anwendungsgebiete werden in den nachfolgenden Abschnitten beschrieben.

Data Mining befasst sich mit sehr großen Datenbeständen. Demzufolge ist der Ressourcenverbrauch an Rechnerzeit und Speicher nicht zu vernachlässigen. Durch die Anwendung des „Kombinierten Data Mining“ soll eine Reduzierung des Ressourcenbedarfs an Rechnerzeit und Speicher erreicht werden. Dies soll durch die Berechnung von Hilfsinformationen im ersten Schritt für die folgenden Schritte erreicht werden. Eine weitere Möglichkeit zur Reduktion besteht in der Kenntnis der verwendeten Verfahren. Die Metainformationen über die Vorgängerverfahren erlauben

dem nachfolgenden Verfahren von einem vereinfachten Problem auszugehen. Aus diesen Möglichkeiten zur Reduktion des Ressourcenbedarfs lassen sich verschiedene Arten von „Kombinierten Data Mining“ ableiten, die nachfolgend vorgestellt werden:

- **Naives „Kombiniertes Data Mining“**: Beim naiven „Kombinierten Data Mining“ nutzt das Nachfolgeverfahren nur das Ergebnis des Vorgängerverfahren. Eine Optimierung findet nicht statt. Zum Beispiel kann ein Klassifikationsverfahren die Ergebnisse eines Clusteringverfahrens als Trainingsdaten verwenden (siehe Tabelle 2).

- **Vorgängerverfahren kennt Nachfolgerverfahren**: Das Vorgängerverfahren berechnet Hilfsinformationen für das Nachfolgeverfahren. Hilfsinformationen können zum Beispiel arithmetische Werte wie Summen, Minima oder Maxima der verwendeten Daten sein. Dadurch können Berechnungen des Nachfolgeverfahrens vereinfacht werden. Mögliche Hilfsinformationen werden in Kapitel 5 vorgestellt.

- **Nachfolgerverfahren kennt Vorgängerverfahren**: Das Nachfolgeverfahren kann durch die Kenntnis über das Vorgängerverfahren von einem vereinfachten Problem ausgehen. Wenn, zum Beispiel, bei einem Clusteringverfahren nur konvexe Cluster erzeugt werden, muss der nachfolgende Klassifikations-algorithmus die Attribute nach denen geclustert wurde, nicht mehr berücksichtigen, da das Klassifikationsergebnis sich auf direktem Wege errechnen lässt. In diesem Fall trennt die mittelsenkrechte Ebene zwischen den Cluster-Mittelpunkten die beiden Cluster optimal. Für genauere Informationen zu diesem Ansatz wird auf Humer [HM04] verwiesen.

Von diesen drei Arten des „Kombinierten Data Mining“ können die zwei Arten, Vorgängerverfahren kennt Nachfolgeverfahren und Nachfolgeverfahren kennt Vorgängerverfahren, gemeinsam auftreten. Dadurch sollte eine Reduktion des Ressourcenverbrauchs an Rechnerzeit und Speicher erreicht werden. Abbildung 10 zeigt den Ansatz des „Kombinierten Data Mining“ für Clustering und Klassifikation, der in dieser Arbeit genauer analysiert wird. Es wird verdeutlich wie die Klassifikation die ermittelten Hilfsinformationen des Clustering nutzt, und selbst als beschreibende Daten für das Ergebnis des Clustering genutzt werden können.

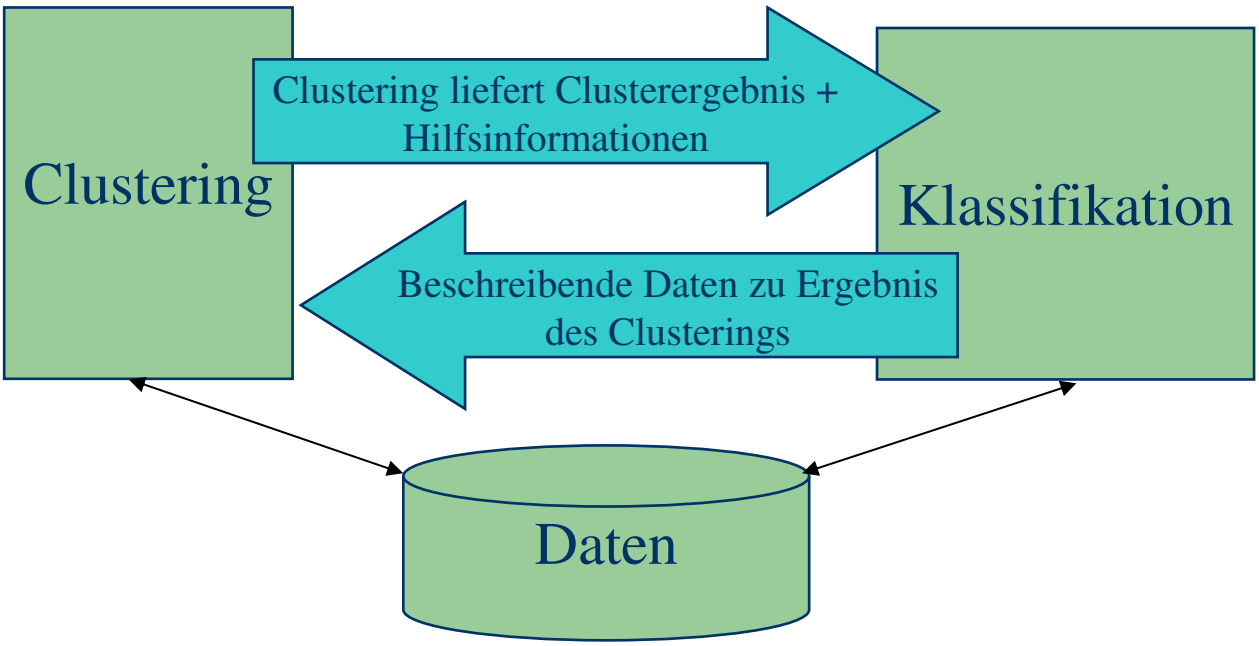


Abbildung 10: "Kombiniertes Data Mining" mit Clustering und Klassifikation [GM04]

„Kombiniertes Data Mining“ wird in den bis dato veröffentlichten Arbeiten zum Themenbereich Data Mining nicht erwähnt. Es gibt allerdings einige Ansätze in denen bereits naives „Kombiniertes Data Mining“ verwendet wird. Dabei wird, wie oben erwähnt, das Ergebnis des Vorgängerverfahrens vom Nachfolgeverfahren genutzt. Nachfolgend wird kurz auf einige dieser bestehenden Ansätze des „Kombinierten Data Mining“ eingegangen. Die dazu verwendeten Verfahren sind einerseits Clustering und Klassifikation, und andererseits Clustering und Assoziation. Die nachfolgende Tabelle 2 zeigt die Eingliederung der bestehenden Ansätze in die drei Arten des „Kombinierten Data Mining“.

	naiv	**Vorg. kennt Nachfolger**	**Nachfolger kennt Vorg.**
Clustering von Wörtern für Textklassifikation	X		
CBC – Clustering Based (Text) Classification	X		
Kombination von Self-Organizing Maps und dem K-Means Clustering zur Online-Klassifikation von Sensordaten	X		

Automatische Generation eines Fuzzy Classification Systems mit Hilfe einer Fuzzy Clustering Methode	X		
Clustering basierend auf Hypergraphs von Assoziationsregeln	X		
Kombination von Clustering und Assoziation für Zielgerichtetes Marketing im Internet	X		
ARCS – Association Rule Clustering System	X		

Tabelle 2: Auflistung der nachfolgend beschriebenen Ansätze des "Kombinierten Data Mining" und Einordnung in die drei Arten des "Kombinierten Data Mining"

3.1 Kombiniertes Data Mining – Clustering und Klassifikation

Beim Clustering ist die Klassenzugehörigkeit der Objekte a priori nicht bekannt, und wird erst durch das Clustering vollzogen. Hingegen muss bei der Klassifikation diese Klassenzughörigkeit bereits im Vorhinein bekannt sein. Daraus lässt sich schon relativ leicht ableiten, dass die Klassifikation oft auf dem Ergebnis des Clustering aufsetzt, da dieses die erforderliche Klassenzugehörigkeit liefert, falls diese a priori nicht bekannt ist. Dies alleine ist schon eine Kombination der beiden Verfahren Clustering und Klassifikation, allerdings wird dabei „nur“ das Ergebnis des Clustering für die Klassifikation genutzt.

Eine weitere Kombinationsmöglichkeit der beiden Verfahren ist dadurch gegeben, dass das Clusterergebnis für den Anwender oft kein befriedigendes Erklärungsmodell liefert. Vor allem wenn die zu clusternden Objekte viele Dimensionen aufweisen, ist eine graphische Darstellung nur sehr schwer möglich. Daher wird zur Erklärung des Clusterergebnis oft eine Klassifikation in Form eines Entscheidungsbaums herangezogen. Der Entscheidungsbaum liefert ein für den Benutzer verständliches Erklärungsmodell. So wird die Klassifikation als Erklärungsmodell für das Clustering herangezogen.

Nachfolgend werden einige Bereiche angeführt, in denen eine Kombination der beiden Verfahren genutzt wird. Dies betrifft vor allem den Bereich der Textklassifikation, und in diesem Zusammenhang den Bereich der Dokumentensuche, aber auch einige

andere Bereiche, die kurz erläutert werden. Dabei wird allerdings nur der Ansatz des naiven „Kombinierten Data Mining“ genutzt.

3.1.1 Clustering von Wörtern für Textklassifikation

Bei einer gegebenen Menge von Dokumenten, in Form von Dokumentenvektoren $\{d_1, d_2, ..., d_n\}$, und deren zugehörigen Klassen $c(d_i)$ e $\{c_1, c_2, ..., c_l\}$, ist es die Aufgabe der Textklassifikation die Klasse eines neuen Dokuments zu ermitteln [DKM02]. Typischerweise werden die Dokumentenvektoren aus den zugehörigen Wörtern des Dokuments erstellt. Daraus ergibt sich das Problem, dass relativ viele Dimensionen bei der Klassifikation berücksichtigt werden müssen, da in der Regel ein Dokument aus sehr vielen unterschiedlichen Wörtern besteht. Schon eine relativ geringe Anzahl von Dokumenten kann mehrere tausend Wörter, im Fall der Klassifikation also Dimensionen, enthalten. Diese große Anzahl von Dimensionen kann bei den bestehenden Klassifikationsalgorithmen wie Support Vector Machines oder k-nearest neighbour zu erheblichen Problemen führen, da die Laufzeit mit der Anzahl der Dimensionen steigt (vergleiche [ES00] und [MI97]).

Um diesem Problem der großen Anzahl von Dimensionen zu begegnen, werden die Wörter der Dokumente, von einem Clusteringalgorithmus in Klassen unterteilt. Die Textklassifikation verwendet nur noch die Klassen des Clusteringalgorithmus als Dimensionen. So wird die Anzahl der Dimensionen vermindert. Als Beispiel kann hier eine Klassifikation von Dokumenten über Sport in gewisse Bereiche wie „Basketball“, „Eishockey“ oder „Tennis“ angeführt werden. In den zugrunde liegenden Dokumenten werden, zum Beispiel, die Wörter „Puck“ und „Tormann“ wahrscheinlich nur in den Dokumenten der Klasse „Eishockey“ vorkommen. Daher ist es nicht nötig zwischen diesen Wörtern zu unterscheiden. Aus diesem Grund werden diese Wörter in einem Cluster zusammengefügt. So wird auch mit den weiteren Wörtern vorgegangen. Das Clustern von Wörtern verringert nicht nur die Anzahl der Dimensionen für die Klassifikation sondern steigert in den meisten Fällen auch die Genauigkeit des Klassifikators [BM98].

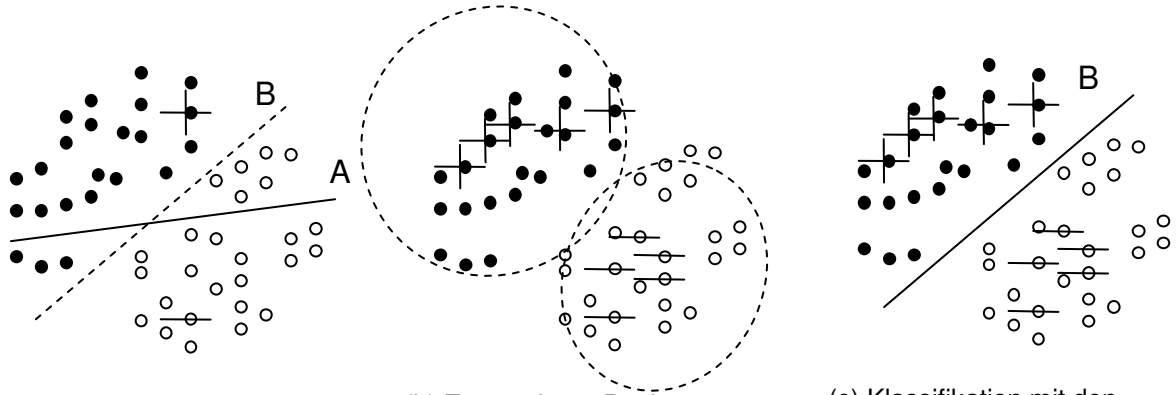
In einem der Ansätze zur Kombination von Clustering von Wörtern und der Textklassifikation wird Distributional Clustering [LPT93] als Clusteringalgorithmus und der Naive Bayes [ES00] als Klassifikationsalgorithmus verwendet. Informationen zum Distributional Clustering in Kombinationen mit Textklassifikation können in Backer und McCallum [BM98] und Dhillon, Kumar und Mallelon [DKM02] gefunden werden.

3.1.2 CBC – Clustering Based (Text) Classification

Wenn die Anzahl der Trainingsdaten pro Klasse, d.h. der Dokumente, die einer Klasse zugeordnet sind, sinkt, dann leidet auch die Genauigkeit des Klassifikators von herkömmlichen Textklassifikationen drastisch. Dies ist allerdings oft der Fall, da eine Zuordnung von Dokumenten zu Klassen oft nur manuell durchgeführt werden kann, und dadurch mit einem sehr hohen Aufwand an Zeit und Kosten verbunden ist [CLMWZ03].

Um diesem Problem zu entgegnen, gibt es den Ansatz des Clustering Based Classification, kurz CBC. Hier werden alle vorhandenen Dokumente, sowohl mit Klassenzugehörigkeit als auch ohne, in einem ersten Schritt mit einem Clusteringverfahren in Klassen unterteilt. Dabei wird von den Dokumenten mit der vorhandenen Klassenzughörigkeit ausgegangen und ein Teil der nicht zugeordneten Dokumente einer Klasse zugeteilt. So wird die Menge der Trainingsdaten erweitert und eine Klassifikation mit diesen Daten kann durchgeführt werden, wobei eine höhere Genauigkeit des Klassifikators wahrscheinlicher ist.

Abbildung 11 zeigt ein Beispiel für die CBC-Technik. In der gegebenen Punktmenge sind die schwarz und grau hinterlegten Punkte noch keiner Klasse zugehörig. Lediglich die Punkte, die mit „+“ und „-„ markiert sind, sind einer Klasse zugeordnet. Abbildung 11 (a) zeigt wie das Klassifikationsergebnis aussehen würde wenn zuvor kein Clustering durchführt wird, wobei die Linie B das Soll-Ergebnis darstellt. In Abbildung 11 (b) zeigt das Clusterergebnis, ausgehend von den Objekten mit einer Klassenzugehörigkeit. Abbildung 11 (c) zeigt das Ergebnis der Klassifikation, nachdem das Clustering ausgeführt wurde.



(a) Klassifikation mit ursprünglich zugeordneten Daten

(b) Zugeordnete Punkte durch Clustering erweitern

(c) Klassifikation mit den neu zugeordneten Daten

Abbildung 11: Beispiel für Clustering Based Classification (CBC)

3.1.3 Kombination von Self-Organizing Maps und dem K-Means Clustering zur Online-Klassifikation von Sensordaten

In der Mensch-Computer Interaktion sollte der Computer alle offensichtlichen Entscheidungen selbst treffen, ohne den Menschen damit zu belasten [VL01]. Somit sollte dem Menschen die Arbeit erleichtert werden. Ein wichtiger Schritt, um eine solche Forderung umsetzen zu können, ist die Erkennung des Kontexts (Aufenthaltsort, Aktivität des Benutzers, Status der Applikation usw.) durch den Computer. Damit eine solche Erkennung des Kontexts möglich ist, werden Sensoren benutzt, welche die eingehenden Daten an einen Algorithmus weiterleiten, der die Daten richtig verarbeiten kann. Beispiele für die Anwendung solcher Sensoren, zur Erkennung des Kontexts, sind Mobiltelefone und Handhelds. Mobiltelefone sollen je nach Umgebung und Situation unterschiedlich reagieren, zum Beispiel laut klingeln bei einer lauten Umgebung oder nur vibrieren in einer Besprechung. Handhelds sollten abhängig vom Kontext gewisse Anwendungen starten.

Damit ein bestimmter Kontext erkannt werden kann, müssen so viele Informationen wie möglich über den Kontext gesammelt werden. Dazu sind viele Sensoren und gute Klassifikationsalgorithmen nötig. Der kritische Teil der Kontexterkennung ist sicher der Klassifikationsalgorithmus, der die eingehenden Sensordaten interpretieren und richtig zuordnen muss. Dazu ist ein Algorithmus nötig der einen Kontext „erlernen“ kann, d.h. nicht der Softwareentwickler gibt die möglichen Kontexte vor, sondern der Benutzer selbst entscheidet welche Kontexte wichtig sind, und teilt dem System mit, welche eingehenden Daten, welchen Kontext beschreiben. Da die eingehenden Sensordaten oft unwichtiges Material, so genanntes Rauschen, enthalten, wird meist ein Neuronales Netz zur Erkennung des Kontexts herangezogen.

Um die eingehenden Sensordaten effektiv zu nutzen, werden diese oft in einem Vorverarbeitungsschritt mit der Standardabweichung oder dem Mittelwert kombiniert. Somit sollte die große Menge an eingehenden Sensordaten gefiltert werden, da nur relevante Daten verarbeitet werden. Aufgabe des Klassifikationsalgorithmus ist es nun, für die vorverarbeiteten Sensordaten den relevanten Kontext zu finden. Als Klassifikationsalgorithmus wird die Kohonen Self-Organizing Map (KSOM) verwendet [KO94]. KSOM ist ein neuronales Netzwerk und erzeugt aus den eingehenden Sensordaten eine 2D-Karte. Der KSOM-Algorithmus neigt dazu bereits erlerntes Wissen zu überschreiben [VL01], vor allem in der Phase der Initialisierung. Damit dieser Nachteil ausgeglichen wird, wird der KSOM-Algorithmus mit dem K-Means Algorithmus kombiniert. Ausgehend von diesem Ergebnis, durch Kontrolle in welchem

Cluster sich die eingehenden Daten befinden, erhält man den Kontext der eingehenden Daten.

3.1.4 Automatische Generation eines Fuzzy Classification Systems mit Hilfe einer Fuzzy Clustering Methode

Im Gegensatz zu Neuronalen Netzen haben Fuzzy Classification System, kurz FCS, den Vorteil, dass sie ein Erklärungsmodell des Ergebnisses liefern. Durch die Regeln des FCS kann der Benutzer nachvollziehen, warum sich das System so verhält, wie es sich verhält. Des Weiteren kann er eigene Regeln hinzufügen, falls dies notwendig ist. Ein großer Nachteil des FCS ist, dass Expertenwissen über die Klassifikationsaufgabe vorhanden sein muss. Normalerweise kann beim FCS nicht von Trainingsdaten gelernt werden [GG94].

Um dieses Problem zu lösen, werden oft Clusteringverfahren eingesetzt. Auf dem Ergebnis des Clustering kann dann die Klassifikation durchgeführt werden. Fuzzy Clustering Methoden sind dazu sehr gut geeignet, da diese Objekte nicht nur einer Klasse zuordnen, sondern angeben mit welcher Wahrscheinlichkeit ein Objekt zu einer Klasse gehört.

Ein FCS besteht im Wesentlichen aus drei Hauptkomponenten: Der „Membership“-Funktion, den Klassifikationsregeln und der Gewichtung der Regeln. Die Regeln werden dabei normalerweise von Experten, die Wissen über das Anwendungsgebiet haben, festgelegt. Über die Gewichtung kann die Wichtigkeit der einzelnen Regeln beeinflusst werden.

Mit Hilfe des Fuzzy Clustering soll es möglich sein, die drei Hauptkomponenten, „Membership“-Funktion, Klassifikationsregeln und die Gewichtung der Regeln, des FCS automatisch zu generieren, d.h. es ist kein Experte mehr nötig. Die „Membership“-Funktion wird dadurch ermittelt, dass jede Dimension separat geclustert wird. Das Ergebnis, d.h. die ermittelten Wahrscheinlichkeiten zu welchem Cluster das Objekt gehört, dieses Clustering kann direkt als „Membership“-Funktion eingesetzt werden. Die Regeln lassen sich direkt aus den ermittelten Clustern und der „Membership“-Funktion ableiten. Die Gewichtung der Regeln wird durch die Berechnung der Beziehung zwischen den Clustern und Klassen ermittelt. Genauere Informationen über das Fuzzy Clustering und das FCS können bei Genther und Glesner [GG94] gefunden werden.

3.2 Kombiniertes Data Mining – Clustering und Assoziation

Assoziationsregeln sind ein Mittel zur so genannten *Warenkorbanalyse*. Gegeben ist dabei eine Datenbank von „Warenkörben“, die gewisse Kundentransaktionen repräsentieren. Ein einzelner Warenkorb (Transaktion) ist im weitesten Sinne eine Menge von zusammen eingekauften Artikeln, Dienstleistungen oder Informationen eines Anbieters [ES00]. Jeder dieser Artikel oder Dienstleistungen einer Transaktion wird Item genannt, d.h. eine Transaktion besteht aus einer Menge von Items.

Eine typische Anwendung einer Transaktionsdatenbank ist die Sammlung von Daten durch die Scannerkassen eines Supermarkts. Dabei werden alle gekauften Artikel pro Kundentransaktion gespeichert. Assoziationsregeln drücken Zusammenhänge innerhalb der Transaktionen aus, die in der gesamten Datenmenge häufig vorkommen [ES00]. Im Supermarkt-Beispiel können über Assoziationsregeln Zusammenhänge zwischen häufig gekauften Artikeln hergestellt werden. Eine solche Regel kann für das Supermarktbeispiel wie folgt aussehen: {Brot, Wurst} → {Butter}. Diese Regel ist dann interessant, wenn Brot, Wurst und Butter oft gemeinsam, d.h. innerhalb einer Transaktion, gekauft werden.

Eine Kombination von Assoziation und Clustering findet oft in dem Sinne statt, dass ausgehend vom Ergebnis der Assoziation ein Clustering durchgeführt wird. Dadurch soll die Anzahl der gefundenen Assoziationen, die meist sehr groß ist, eingeschränkt und auf die wesentlichen begrenzt werden. Nachfolgend werden ein paar Möglichkeiten zur Kombination von Clustering und Assoziation vorgestellt. Auch diese vorgestellte Kombination basiert auf dem naiven „Kombinierten Data Mining“, da immer nur mit den Ergebnissen der Verfahren gearbeitet wird.

3.2.1 Clustering basierend auf Hypergraphs von Assoziationsregeln

Bei dem Ansatz von Han, Karypis, Kumar und Mobasher [HKKM97] wird das Clustering auf die häufig auftretenden Items (Itemsets), auch Frequent Itemsets genannt, die vom Assoziationsalgorithmus entdeckt wurden, ausgeführt. Frequent Itemsets sind alle Mengen von Items, die einen minimalen Support (siehe [ES00]) erfüllen. Ausgehend von diesen Frequent Itemsets wird ein Hypergraph mit den Items erzeugt, und ein partitionierender Clusteringalgorithmus zerlegt diesen in Cluster. Solche Cluster können dazu genützt werden, bestimmte Items zu klassifizieren, Vorhersagen über ähnliche Items zu treffen, oder die Anzahl der Regeln zu minimieren, in dem die weniger Interessanten entfernt werden. Des Weiteren können auch Transaktionen anhand ihrer Items geclustert werden. Dies kann vor allem dann

interessant sein, wenn das Verhalten der Benutzer, wie zum Beispiel das Einkaufsverhalten im Supermarkt, analysiert werden soll.

Ausgehend von den Frequent Itemsets wird ein gewichteter Hypergraph erzeugt. Dieser Hypergraph wird durch eine Ähnlichkeitsfunktion partitioniert, die an der Konfidenzberechnung [siehe ES00] der Assoziationsregeln der Frequent Itemsets angelehnt ist.

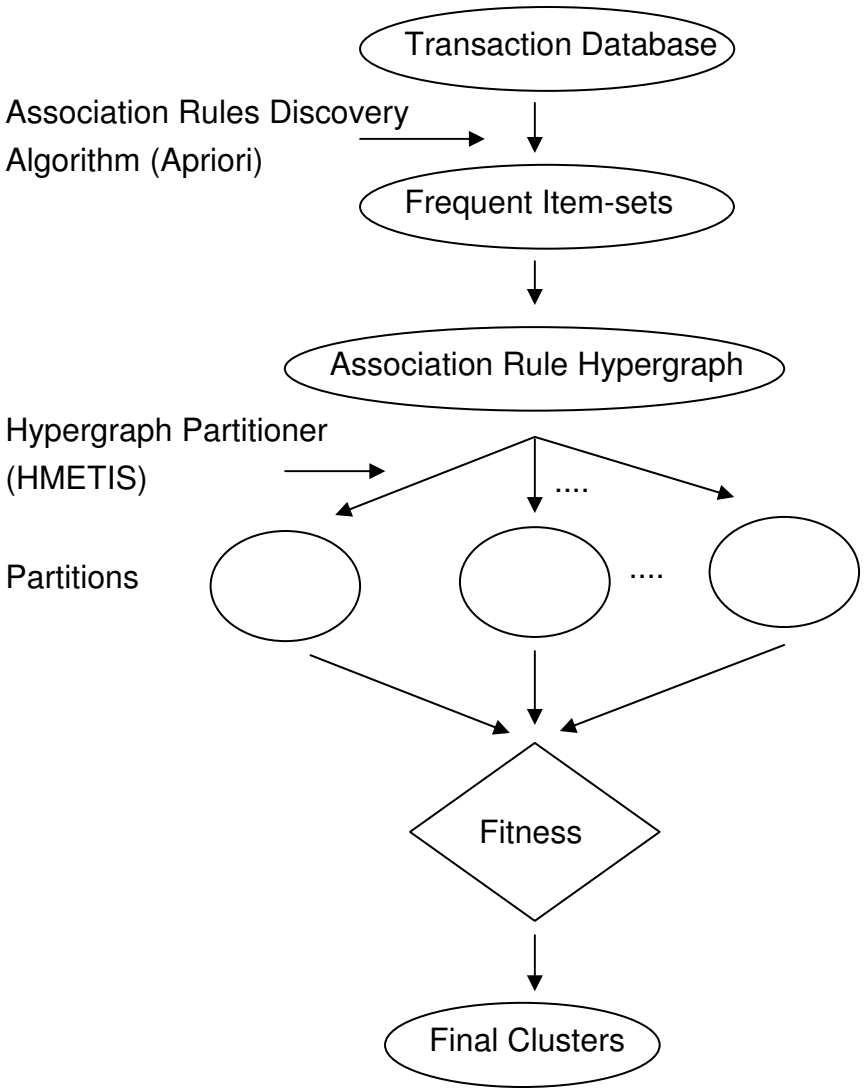


Abbildung 12: Clusteringalgorithmus ausgehend von einem Hypergraph

Abbildung 12 zeigt den Algorithmus für den Clusteringprozess. Ein Hypergraph [BE97] H = (V, E) besteht aus Knoten (vertices) und Kanten (edges). Ein Hypergraph ist eine Erweiterung des normalen Graphen in dem Sinne, dass jede Kante mehr als zwei Knoten verbinden kann. In diesem Modell stellen die Knoten die Items der Datenbank

dar und die Kanten die Frequent Itemsets. Wenn zum Beispiel {A B C} ein Frequent Itemset ist, dann enthält der Hypergraph eine Kante, die A, B und C verbindet. Als partitionierender Clusteringalgorithmus wird HMETIS [KAKS96] verwendet. HMETIS produziert k balancierte Partitionen, wobei k von einem Benutzer vorgegeben wird.

3.2.2 Kombination von Clustering und Assoziation für Zielgerichtetes Marketing im Internet

Zielgerichtetes kundenspezifisches Marketing kann aus Sicht eines Unternehmens die Kundenzufriedenheit der Kunden steigern. Dieses Zielgerichtete Marketing war bisher allerdings meist mit hohen Kosten verbunden, und daher nur eingeschränkt realisierbar [LY00]. Die hohen Kosten begründeten sich vor allem in der Beschaffung der relevanten Daten und der Analyse dieser Daten. Durch das Internet hat sich vor allem die Beschaffung der Daten vereinfacht, vor allem werden nicht mehr nur noch demographische Daten, wie Alter oder Geschlecht, sondern auch Daten zum benutzerspezifischen Verhalten des Benutzers im Internet gespeichert, d.h. der Entscheidungsprozess des Benutzers kann unter Umständen nachvollzogen werden. Die Analyse der Daten wird vor allem durch Clustering und Assoziation erleichtert.

In einem Ansatz von Lai und Yang [LY00] werden in zwei Schritten Clustering und Assoziation eingesetzt, um eine solches Zielgerichtetes Marketing durchzuführen. Im ersten Schritt wird ein Clustering auf die vorhanden Kundendaten durchgeführt, dabei werden demographische, geographische und auch Daten zum Verhalten der Kunden herangezogen. Als Clusteringalgorithmus wird PAM (Partitioning Around Medoids) [KR90] verwendet. Die ermittelten Cluster und Medoide werden als Userprofile gespeichert. In einem zweiten Schritt wird die Assoziation auf jedes Cluster ausgeführt. Als Ergebnis liefert die Assoziation Assoziationsregeln für jedes Cluster. Die Regeln gelten jeweils nur für einen Cluster. Basierend auf diesem Ergebnis wird dann das Marketing ausgeführt.

3.2.3 ARCS – Association Rule Clustering System

In den bisher verwendeten Assoziationsalgorithmen ist meist von Transaktionsdaten, zum Beispiel in Form von Einkäufen in einem Supermarkt, ausgegangen worden. Die Anwendung von Assoziationsregeln ist aber nicht auf diesen einen Bereich beschränkt. Es ist genauso möglich, zum Beispiel, mit demographischen Kundendaten Assoziationsregeln zu erzeugen. Dabei wird jeder Kunde durch einen Tupel repräsentiert, der aus einer gewissen Anzahl von Attributen besteht, die eine gewisse Ausprägung besitzen. Im Vergleich zur Assoziation mit Transaktionsdaten sind in

diesem Fall die einzelnen Attribute mit ihren Ausprägungen die Items, und ein Tupel ist eine Transaktion. Eine solche Assoziationsregel kann zum Beispiel wie folgt aussehen:

(alter = 40) & (einkommen > 40.000 €) → (hausbesitzer = ja)

Falls eine solche Assoziation auf eine große Menge von Kundendaten durchgeführt wird, dann werden in der Regel hunderte bis tausende Assoziationsregeln gefunden. Um diese große Anzahl von Assoziationsregeln einzuschränken, gibt es nun den Ansatz „Clustering Assoziationsregeln“ [LSW00]. Dabei sollen ähnliche Assoziationsregeln kombiniert werden, damit eine geringe Anzahl von allgemeinen Regeln entsteht. Zum Beispiel kann aus den beiden Regeln (alter = 40) → (hausbesitzer = ja) und (alter = 41) → (hausbesitzer = ja) eine Regel in der Form (39 < alter < 42) → (hausbesitzer = ja) erstellt werden. Dieser Ansatz führt nicht nur zu einer Reduktion der gefundenen Assoziationsregeln, sondern auch zu einer besseren Interpretierbarkeit durch den Benutzer. Des Weiteren lassen sich weniger Regeln auch leichter visuell darstellen.

In diesem Ansatz eines „Association Rule Clustering Systems“ (ARCS) von Lend, Swami und Widom [LSW97] werden nur Assoziationsregeln im zwei-dimensionalen Raum berücksichtigt, wobei jede Dimension ein Attribut der linken Regelseite darstellt. Die beiden Attribute werden normalerweise vom Benutzer ausgewählt. Es gibt dazu aber auch statistische Kennzahlen, welche die wichtigsten Attribute aus einer Datenmenge bestimmen. Des Weiteren wählt der Benutzer ein drittes Attribut aus, nach dem anschließend beim Clustering segmentiert wird. In den oben angeführten Beispielen wäre dieses dritte Attribut „hausbesitzer“.

Beim ARCS–Ansatz, dessen Architektur in Abbildung 13 dargestellt ist, werden zuerst die Daten eingelesen und wenn nötig konvertiert. Im nächsten Schritt werden die Assoziationsregeln ermittelt. Dies erfolgt in einem einzigen Durchlauf über die gesamte Datenmenge. Anschließend werden die gefunden Assoziationsregeln mit dem BitOp-Algorithmus [CLR90] geclustert. Dieser liefert dann als Ergebnis die „zusammengefassten“ Assoziationsregeln. Mit Hilfe von Testdaten wird überprüft ob die gefundenen Assoziationsregeln ein gewünschtes Maß an Genauigkeit erfüllen. Falls dies nicht der Fall ist, wird noch einmal mit der Assoziation begonnen, und zwar mit geänderten Parametern, zum Beispiel höherer Support oder höhere Konfidenz.

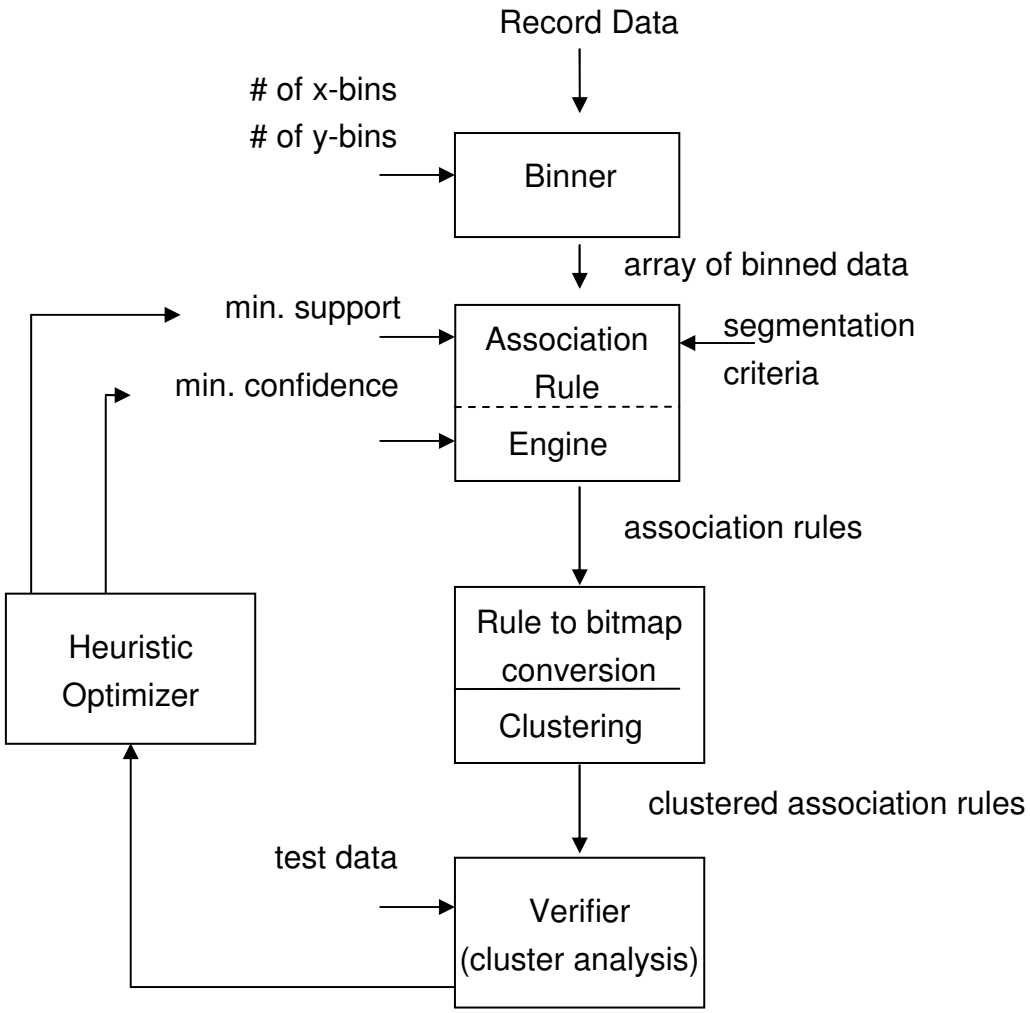


Abbildung 13: Architektur des "Association Rule Clustering System"

4 Klassische Verbesserungsmöglichkeiten durch Optimierung der Algorithmen

Das vorige Kapitel hat die drei Arten des „Kombinierten Data Mining“, *naiv*, *Vorgänger kennt Nachfolger* und *Nachfolger kennt Vorgänger*, und einige bestehende Ansätze zum naiven „Kombinierten Data Mining“ genauer erörtert. Bevor nun im nächsten Kapitel auf die möglichen Verbesserungsmöglichkeiten durch den Einsatz von „Kombinierten Data Mining“ eingegangen wird, werden in den nächsten beiden Abschnitten Optimierungsmöglichkeiten der Algorithmen K-Means und DBSCAN vorgestellt. Der abschließende dritte Abschnitt gibt Aufschlüsse über Optimierungsmöglichkeiten durch Indexstrukturen für Datenbanken, welche die vorhandenen Data Mining Algorithmen beschleunigen können. Die Optimierung der Data Mining Algorithmen und die Nutzung von Indexstrukturen führt zu einer generellen Verbesserung. Auch in Bezug auf alle drei Arten des „Kombinierten Data Mining“, dadurch, dass die Einzelverfahren beschleunigt werden.

4.1 Optimierungsmöglichkeiten des K-Means Algorithmus

Es gibt einige Ansätze der Optimierung für den K-Means Algorithmus, die alle auf den „Schwächen“ des Algorithmus aufsetzen. Wie bei der Beschreibung des Algorithmus (siehe Kap. 2.2.2.1) bereits erwähnt wurde, ist das Ergebnis stark vom ersten Schritt abhängig, d.h. vom initialen Clustering. Daher beschäftigen sich viele Optimierungsmöglichkeiten mit der Initialisierung von iterativ optimierenden Clusteringalgorithmen, wie dem K-Means. Des Weiteren ist das Ergebnis des K-Means Algorithmus natürlich stark von der Wahl des Eingabeparameters k, der die Anzahl der zu findenden Cluster angibt, abhängig. Der Algorithmus liefert für jeden Wert von k ein Ergebnis. Es stellt sich allerdings die Frage welcher Wert für k die gegebene Datenmenge am Besten trennt. Eine weitere „Schwäche“ ergibt sich aus der Verwendung der euklidischen Distanz (siehe Kap. 2.2.1) für die Distanzermittlung zwischen zwei Objekten. Daraus lässt sich nämlich eine kreisförmige Darstellung der Cluster ableiten, d.h. es werden nur kreisförmige Cluster gefunden.

Nachfolgend werden Optimierungsmöglichkeiten vorgestellt, die versuchen diese „Schwächen“ zu beheben bzw. die Auswirkungen der „Schwächen“ zu mindern.

4.1.1 Optimierung der Initialisierung des K-Means Algorithmus

Ein gutes initiales Clustering kennzeichnet sich dadurch, dass die tatsächliche Clusterstruktur schon gut approximiert wird. Durch eine solche gute Annäherung der Clusterstruktur ergibt sich eine Optimierung in zweifacher Weise:

- Je besser das initiale Clustering, desto besser ist die Qualität des Endergebnisses des Algorithmus.

- Je besser das initiale Clustering, desto weniger Iterationen benötigt der Algorithmus bis zur Terminierung, d.h. die Laufzeit des Algorithmus wird minimiert.

4.1.1.1 Initialisierung durch wiederholtes Ziehen von Stichproben

Die im folgenden vorgestellte Heuristik von Fayyad, Reina und Bradley [FRB98] zur Bestimmung eines guten initialen Clustering, kann nicht nur für den K-Means Algorithmus verwendet werden, sondern auch für alle anderen iterativ optimierenden Clusteringalgorithmen.

Die Heuristik basiert darauf, dass bei wiederholtem Ziehen von Stichproben aus der Datenmenge die Punkte der Stichproben auf natürliche Weise zu den Clusterzentren tendieren [ES00]. In der nachfolgenden Abbildung 14 wird eine Datenmenge mit drei Gaußclustern im 2-dimensionalen Raum dargestellt.

Die linke Seite der Abbildung zeigt die gesamte Datenmenge. Auf der rechten Seite wird eine kleine Stichprobe der gesamten Datenmenge gezeigt, die insgesamt das

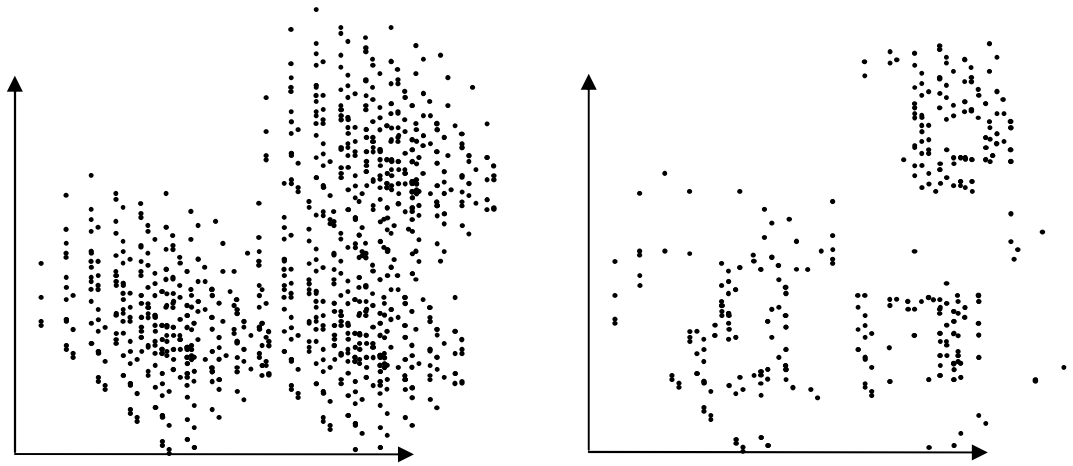


Abbildung 14: Drei Gaußcluster im 2-dimensionalen Raum: Gesamtmenge und Stichprobe

erwartete Verhalten zeigt. Aus Abbildung 14 lässt sich ableiten, dass man durch Clustering einer kleinen Stichprobe schon gute initiale Cluster erhält. Dies wird zwar auf die meisten Stichproben, allerdings nicht auf alle Stichproben, die man zieht, zutreffen. Um das Problem einzelner Stichproben, die keine gute Approximation der Gesamtmenge liefern, zu vermeiden, werden unabhängig voneinander *n* verschiedene Stichproben gezogen. Mit den gezogenen Stichproben wird jeweils ein Clustering mit einer zufälligen Startkonfiguration durchgeführt. Als Ergebnis erhält man *n* verschiedene Schätzungen für *k* Clusterzentren. Ein initiales Clustering für den Algorithmus auf der Gesamtmenge erhält man nun, indem man die *k * n* Ergebnisse für die Stichproben zusammennimmt und mit jedem Ergebnis einer Stichprobe als Startkonfiguration clustert. Von den entstehenden *n* Ergebnissen wird das mit dem besten Wert bezüglich des zugehörigen Maßes für die Güte eines Clustering als initiales Clustering ausgewählt. Ein Beispiel für ein Gütemaß eines Clustering ist der Silhouetten-Koeffizient (siehe Kap. 4.1.2.1).

4.1.1.2 Initialisierung mit Hilfe eines kd-Baums

Ein kd-Baum ist eine Datenstruktur zum Speichern einer endlichen Menge von mehrdimensionalen Punkten [BE80]. Der kd-Baum kann verwendet werden um die Anzahl der Distanzberechnungen eines K-Means Algorithmus zu reduzieren. Dies basiert vor allem darauf dass die Knoten des kd-Baums eine große Anzahl an Punkten enthalten. Durch diese Knoten ist eine Abfrage in der Form „jeder Punkt der mit diesem Knoten assoziiert ist hat X als Centroiden“ für einen Centroiden X effizient möglich. Zusätzlich zu diesen möglichen Abfragen werden auch noch statistische Daten für jeden Knoten des kd-Baums gespeichert. Dies führt dazu, dass die Anzahl der arithmetischen Operationen zu Berechnung der Centroide stark reduziert wird. Der kd-Baum kann sowohl für die Initialisierung des K-Means als auch für den gesamten K-Means Algorithmus eingesetzt werden. Bei der Verwendung des kd-Baums zur Initialisierung wird ein K-Means Clustering mit einer kleinen Stichprobe mittels kd-Baum durchgeführt. Als Ergebnis erhält man einen kd-Baum dessen Knoten als initiale Centroide für den gesamten K-Means Algorithmus verwendet werden können.

Im Ansatz von Pelleg und Moore [PM99] wird eine spezielle Form des kd-Baums namens mrkd-Baum (multi-resolution kd-Baum) verwendet. Für genauere Informationen zum kd-Baum vergleiche Moore [MO91]. Die genauen Eigenschaften des mrkd-Baums werden nachfolgend dargestellt:

- Der mrkd-Baum ist ein Binärbaum.

- Jeder Knoten enthält Informationen über seine Punkte, die in einem so genannten hyper-rectangle *h* enthalten sind. Das hyper-rectangle wird mit zwei M-langen Grenzvektoren h^{max} und h^{min} in seinem Knoten gespeichert. Zusätzlich werden im Knoten die Anzahl der Punkte, der Centroid und die Summen für die Euklidschen Berechnungen innerhalb eines *h* gespeichert. Alle Söhne eines Knoten repräsentieren hyper-rectangles innerhalb von *h*.

- Jeder „Nicht-Blattknoten" hat eine „Splitdimension" *d* und einen „Splitwert" *v*. Dessen Söhne *l* und *r* repräsentieren zwei hyper-rectangles h_l und h_r, beide innerhalb von *h*, sodass alle Punkte in h_l und h_r einen kleineren Wert für ihren *d*-ten Dimensionswert als *v* besitzen.

- Der Wurzelknoten repräsentiert ein hyper-rectangle das alle Punkte enthält.

- Blattknoten speichern die aktuellen Punkte.

Für einen Punkt *x* und ein hyper-rectangle *h* wird *closest(x,h)* als der Punkt, der in *h* am nähesten zu *x* ist, definiert. Der Aufwand zur Berechnung des *closest(x,h)* ist *O(M)*. Die Distanz *d(x,h)* zwischen einen Punkt *x* und einem hyper-rectangle *h* wird als *d(x,closest(x,h)* definiert. Der Vektor h^{max} und h^{min} eine hyper-rectangle *h* wird mit *width(h)* angegeben.

Der Algorithmus von Pelleg und Moore [PM99] zur Verwendung eines kd-Baums beim K-Means Clustering heißt Blacklisting Algorithmus. Für die genaue Vorgehensweise des Algorithmus und experimentelle Testergebnisse vergleiche Pelleg und Moore [PM99].

4.1.2 Wahl des Parameters k für die Anzahl von Clustern

Die Wahl des Parameters k für die Anzahl der Cluster wird beim herkömmlichen K-Means (siehe Kap. 2.2.2.1) vom Benutzer durchgeführt. Diesem ist die „richtige" Anzahl der Cluster aber in den meisten Anwendungen im Vorhinein nicht bekannt. Daher gibt es einige Methoden die den Benutzer bei der Wahl des Parameters unterstützen bzw. ihm diese Wahl abnehmen.

4.1.2.1 Bestimmung des besten k mittels Silhouetten-Koeffizient

Bei der Bestimmung des besten k für den K-Means Algorithmus wird in diesem Fall für alle k = 2, ..., n-1 jeweils ein Clustering gemäß K-Means durchgeführt. Anschließend

wird aus der Menge der Ergebnisse das „beste“ Clustering ausgewählt. Um das „beste“ Clusteringergebnis auswählen zu können ist ein Maß für die Güte des Clustering notwendig, das unabhängig von der Anzahl k der Cluster ist. Ein geeignetes Maß für die Güte des K-Means ist der so genannte *Silhouetten-Koeffizient* eines Clustering [KR90]. Die Silhouette s(o) eines Objekts o lässt sich wie folgt errechnen:

$$s(o) = \frac{b(o) - a(o)}{\max\{a(o), b(o)\}}$$

Dabei ist a(o) der Abstand eines Objekts o zu seinem Repräsentanten des Clusters, d.h. dem Centroiden, und b(o) der Abstand zum Repräsentanten des „zweitnächsten“ Cluster. Der Ausdruck max{a(o), b(o)} steht für die größte Distanz eines Objekts des Clusters zu seinem zweitnähesten Cluster. Für die Silhouette eines Objekts o gilt: $-1 \leq s(o) \leq 1$. Der Silhouetten-Koeffizient ist ein Maß dafür, wie gut die Zuordnung eines Objekts o zu seinem Cluster ist. Die Werte s(o) haben dabei folgende Interpretation:

- $s(o) \approx 0$, d.h. $a(o) \approx b(o)$: o liegt ungefähr zwischen seinem eigenen und dem Nachbarcluster.

- $s(o) \approx 1$, d.h. a(o) ist wesentlich kleiner als b(o): o ist gut klassifiziert.

- $s(o) \approx -1$, d.h. b(o) ist wesentlich kleiner als a(o): o ist schlecht klassifiziert.

Daraus ergibt sich die einfache Regel, dass je größer der Wert s(o) ist, desto besser ist die Zuordnung von o zu seinem Cluster. Der durchschnittliche Wert der Silhouetten s(o) aller Objekte o eines Cluster C kann damit als Maß für die Güte des Clusters aufgefasst werden [ES00]. Dieser Wert für einen Cluster C heißt auch *Silhouettenweite* von C. Dieser kann wie folgt ermittelt werden:

$$s(C) = \left(\sum_{o \in C} s(o) \right) / |C|$$

Ausgehend von der Silhouettenweite ist der Silhouetten-Koeffizient eines Clustering C_M die Silhouettenweite der Gesamtmenge O. Dieser ist definiert als:

$$s(C_M) = \frac{\sum_{C \in C_M} \sum_{p \in C} s(p)}{|O|}$$

Genauso wie für ein Objekt o gilt auch hier die Regel, dass je größer der Wert $s(C_M)$, desto besser ist das Clustering. Nach [KR90] kann der Silhouetten-Koeffizient wie folgt interpretiert werden:

- $0{,}70 < s(C_M) \leq 1{,}00$: starke Struktur,
- $0{,}50 < s(C_M) \leq 0{,}70$: brauchbare Struktur,
- $0{,}25 < s(C_M) \leq 0{,}50$: schwache Struktur,
- $s(C_M) \leq 0{,}25$: keine Struktur.

4.1.2.2 X-Means: Erweiterter K-Means mit effizienter Schätzung der Anzahl der Cluster

Der X-Means Algorithmus von Pelleg und Moore [PM00] ist eine Erweiterung des K-Means Algorithmus, in welcher der Parameter k für die Anzahl der Cluster automatisch ermittelt wird. Dabei wird nach jeder Iteration des K-Means eine Entscheidung darüber getroffen welche der vorhandenen Centroide gesplittet werden sollen. Durch die Splits der Centroide soll ein besseres Ergebnis ermittelt werden. Die Split-Entscheidung wird durch das Berechnen des *Bayesian Information Criterion* (BIC) [KW95] unterstützt.

Beim X-Means Algorithmus wählt der Benutzer nicht einen Wert für den Parameter k aus, sondern einen Bereich für k, in dem der „richtige“ Wert für k vermutlich liegen wird. Der Wert für k der am Besten bezüglich einer Bewertung wie BIC abschneidet, ist die „richtige“ Wahl für die Anzahl der Cluster. Der Algorithmus startet mit dem Wert für k, welcher der unteren Grenze des gewählten Bereichs entspricht. Anschließend werden durch Splits der bestehenden Centroide so lange Centroide hinzugefügt bis die obere Grenze des gewählten Bereichs für k erreicht

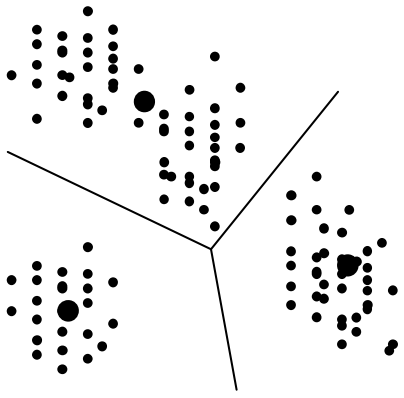


Abbildung 15: K-Means mit 3 Centroiden

wird.

Der Wert für k der das beste Ergebnis gemäß BIC erzielt, wird als Ergebnis ausgegeben. Der X-Means Algorithmus besteht aus den folgenden 2 Schritten:

- Improve-Params
- Improve-Structure

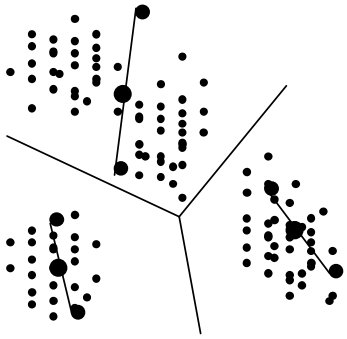


Abbildung 17: Split in 2 Söhne pro Centroid

Wenn k > k_{max} (=obere Grenze des gewählten Bereichs) wird der Algorithmus beendet und liefert den Wert für k der das beste Ergebnis gemäß BIC liefert

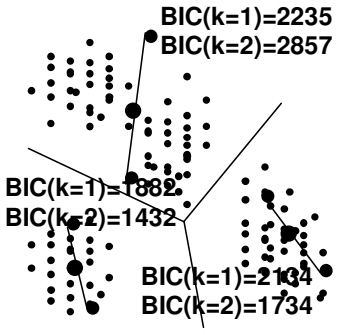


Abbildung 16: Resultat des BIC-Scoring für die Söhne

Der erste Schritt **Improve-Params** entspricht der gewöhnlichen Ausführung des K-Means Algorithmus. Der zweite Schritt **Improve-Structure** ermittelt, welche Centroide gesplittet werden sollen um das Ergebnis zu verbessern. Die Vorgehensweise dieses Schritts wird anhand eines Beispiels erläutert. Abbildung 15 zeigt das Ergebnis des K-Means Algorithmus für 3 Centroide. Zusätzlich werden die Grenzen der einzelnen Bereiche pro Centroid angezeigt. Im nächsten Schritt werden alle Centroide in zwei „Söhne“ gesplittet (siehe Abbildung 17). Diese beiden Söhne werden in einer Distanz proportional zur Größe des Bereichs des ursprünglichen Centroiden, dem Vater, in entgegengesetzter Richtung entlang eines beliebigen Vektors platziert. Anschließend wird in jedem Bereich eines Vaters, d.h. dem ursprünglichen Centroiden, ein lokales K-Means Clustering mit dem Parameter k=2 für jedes Paar von Söhnen durchgeführt. Lokal ist diese Ermittlung insofern, dass die Söhne nur die Punkte ihres Vaters clustern und keine anderen. Abbildung 16 zeigt den ersten Schritt aller lokalen K-Means Läufe und stellt ein mögliches Ergebnis für alle Söhne nach den lokalen K-Means Läufen dar.

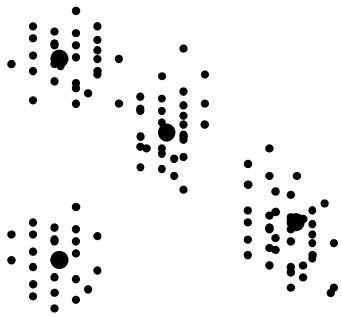


Abbildung 18: Ergebnis nach BIC-Scoring

An dieser Stelle wird für jedes Paar von Söhnen ein „Scoring“ mit BIC durchgeführt, d.h. es wird ermittelt ob die zwei Söhne oder der Vater die Punktmenge besser repräsentieren. Basierend auf diesem Ergebnis werden entweder der Vater, oder die beiden Söhne als Centroide verworfen. Punktmengen, die durch ihren Centroid schlecht repräsentiert werden, werden hier durch die Teilung der Centroide besser dargestellt. Abbildung 18 zeigt das Ergebnis des „Scoring“ mit BIC von Abbildung 16 mit den drei Paaren von Söhnen. Für die genaue Vorgehensweise des „Scoring“ mit BIC wird auf Pelleg und Moore [PM00] verwiesen. Die beiden Schritte **Improve-Params** und **Improve-Structure** des X-Means Algorithmus werden so lange wiederholt bis die obere Grenze des gewählten Bereichs für k erreicht wird.

4.1.3 K-Means Algorithmus mit Berücksichtigung von Hintergrundinformationen

In vielen Anwendungsgebieten besitzen einige Anwender Hintergrundwissen, dass für die Ausführung des Clustering von Bedeutung sein kann. Ein solches Hintergrundwissen kann anwendungsspezifisch oder datenbezogen sein. Der „herkömmliche“ K-Means Algorithmus (siehe Kap. 2.2.2) hat keine Möglichkeiten ein solches Hintergrundwissen in den Algorithmus einzubinden und davon zu profitieren.

In dem folgenden Ansatz von Wagstaff, Cardie, Rogers und Schroedl [WCRS01] wird dargestellt wie Hintergrundwissen bzw. Hintergrundinformationen in einem erweiterten K-Means Algorithmus berücksichtigt werden können. Ein relativ einfaches Beispiel des Nutzens von Hintergrundwissen ist die Wahl des Parameters k für die Anzahl der Cluster. Wenn der optimale Wert für k schon bekannt ist, dann sollte dieser verwendet werden. Damit kann der Aufwand zur Ermittlung des richtigen k vermindert werden (siehe Kap. 4.1.2). Dies stellt allerdings noch keine Erweiterung des K-Means Algorithmus dar. Ein guter Weg um Hintergrundwissen a priori in das Clustering einzubinden sind Bedingungen auf Instanzebene, d.h. zwischen den Datenobjekten, aufzustellen. Dabei kann man zwischen zwei Typen von paarweisen Bedingungen unterscheiden:

- **Must-link** Bedingung: Zwei Datenobjekte müssen im selben Cluster sein
- **Cannot-link** Bedingung: Zwei Datenobjekte dürfen nicht im selben Cluster sein

Diese Bedingungen definieren eine transitive binäre Beziehung zwischen den Datenobjekten. Wenn zum Beispiel d_i durch eine Must-link Bedingung an d_j gebunden ist, und d_j eine Cannot-link Bedingung zu d_k besitzt, dann lässt sich daraus eine Cannot-link Bedingung von d_i und d_k ableiten. Alle bekannten Bedingungen und die

daraus abgeleiteten Bedingungen werden an den erweiterten K-Means Algorithmus übergeben. Diese Bedingungen werden mit Hilfe von Hintergrundwissen von Experten erzeugt (vergleiche [WCRS01]).

Erweiterter K-Means Algorithmus COP-KMEANS

Dem erweiterten K-Means Algorithmus COP-KMEANS werden die gesamte Datenmenge (D), die Must-link Bedingungen ($Bed_{=}$) und die Cannot-link Bedingungen ($Bed_{\neq}$) übergeben. Als Ergebnis liefert der Algorithmus ein Clustering der Datenmenge D, dass die Must-link und die Cannot-link Bedingungen erfüllt. Nachfolgend wird der COP-KMEANS Algorithmus vorgestellt. Die Änderungen gegenüber dem herkömmlichen K-Means sind mit Fettschrift hervorgehoben.

COP-KMEANS(Datenmenge D, Must-link Bedingungen $Bed_{=}$, Cannot-link Bedingungen $Bed_{\neq}$)

1. Ermittlung der initialen Centroide C_1 ... C_k
2. Für jeden Punkt d_i in D wird das näheste Cluster C_j ermittelt **sodass VIOLATE-CONSTRAINTS(d_i, C_j, $Bed_{=}$, $Bed_{\neq}$) gleich false ist. Wenn kein solches Cluster gefunden wird, dann wird dieser Punkt nicht zugeordnet.**
3. Für jedes Cluster C_i werden die Centroide neu ermittelt.
4. Wiederhole Schritte 2 und 3 bis keine Änderung der Zuordnungen der Punkte zu den Clustern mehr auftritt
5. Rückgabe der Cluster {C_1 ... C_k}

VIOLATE-CONSTRAINTS(Punkt d, Cluster C, Must-link Bedingungen $Bed_{=}$, Cannot-link Bedingungen $Bed_{\neq}$)

1. Für jedes Paar ($d,d_{=}$) $\in$ $Bed_{=}$: Wenn $d_{=} \in C$ dann wird true zurückgegeben.
2. Für jedes Paar ($d,d_{\neq}$) $\in$ $Bed_{\neq}$: Wenn $d_{\neq} \in C$ dann wird true zurückgegeben.
3. Andernfalls wird false zurückgegeben.

Die Erweiterung betrifft vor allem die neue Methode VIOLATE-CONSTRAINTS die prüft, dass die aufgestellten Bedingungen auch erfüllt werden. Es wird versucht jeden

Punkt d_i seinem nähesten Cluster C_j zuzuordnen. Falls ein anderer Punkt $d_=$, der im selben Cluster wie d_i sein soll, bereits einem anderen Cluster zugeordnet ist, dann wird d_i nicht C_j zugeordnet. Die Prüfung der Cannot-link Bedingung wird auch durchgeführt, d.h. es darf kein $d_\neq$ im Cluster C_j geben, dass nicht mit d_i in einem Cluster sein darf. Dieser Vorgang wird solange durchgeführt bis das näheste Cluster gefunden wurde, das alle Bedingungen erfüllt.

4.1.4 Verwendung der Dreiecksungleichung zur Verbesserung des K-Means Algorithmus

Die Anzahl der Distanzberechnungen eines K-Means Algorithmus ist *nke*, wobei *n* die Anzahl der Punkte ist, *k* die Anzahl der Cluster und *e* die Anzahl der Iterationen. Die Anzahl der Iterationen *e* steigt mit *k*, *n* und der Anzahl der Dimensionen *d* der Punkte. Der Ansatz von [Elkan 2003] beschäftigt sich damit, die Anzahl der Distanzberechnungen auf nahezu *n* im Vergleich zu *nke* zu reduzieren. Dies führt zu einer Verbesserung der Laufzeit des K-Means Algorithmus. Die Verringerung der Distanzberechnungen beruht darauf, dass ein Teil der Distanzberechnungen des herkömmlichen K-Means (siehe Kap. 2.2.2) redundant sind. Wenn ein Punkt weit entfernt von einem Centroiden liegt, dann ist es nicht notwendig die exakte Distanz zwischen dem Punkt und dem Centroid zu berechnen um festzustellen, dass dieser Punkt nicht zu diesem Centroid zugeordnet wird. Umgekehrt gilt natürlich genauso, wenn ein Punkt näher zu einem Centroiden liegt als zu einem Anderen, dann wird der Punkt dem ersten Centroiden zugeordnet ohne die exakten Distanzen zwischen dem Punkt und den Centroiden zu berechnen. Der verbesserte K-Means Algorithmus soll überall eingesetzt werden können, wo auch der herkömmliche K-Means eingesetzt werden kann. Des Weiteren soll das Ergebnis des verbesserten K-Means, bei gleicher Initialisierung der Centroide, dem Ergebnis des K-Means entsprechen.

Die bereits oben angeführte Verbesserung des K-Means Algorithmus durch Reduktion der Distanzberechnungen beruht auf der Dreiecksungleichung. Die Dreiecksungleichung für drei Punkte *x*, *y* und *z* lautet wie folgt:

$$d(x,z) \leq d(x,y) + d(y,z)$$

Diese Dreiecksungleichung gilt für alle Distanzfunktionen (siehe Kap. 2.2.1). Diese Dreiecksungleichung liefert uns Obergrenzen. Zum Vermeiden von Distanzberechnungen sind allerdings auch Untergrenzen von Nöten. Wenn *x* ein Punkt ist und *b* und *c* Centroide sind, dann muss *d(x,c)* größer als *d(x,b)* sein, um die Berechnung von *d(x,c)* zu vermeiden. Die beiden nachfolgenden Lemma zeigen wie die

Dreiecksungleichung angewandt werden muss, um sinnvolle Untergrenzen zu erhalten:

- **Lemma 1**: Wenn *x* ein Punkt ist, *b* und *c* Centroide sind und $d(b,c) \geq 2d(x,b)$, dann gilt $d(x,c) \geq d(x,b)$.

- **Lemma 2**: Wenn *x* ein Punkt ist und *b* und *c* Centroide sind, dann gilt $d(x,c) \geq max\{0, d(x,b) - d(b,c)\}$

Diese beiden Lemma gelten für drei beliebige Punkte, also nicht nur für einen Punkt und zwei Centroide.

Lemma 1 wird im verbesserten K-Means wie folgt benutzt. Wenn *x* ein beliebiger Punkt ist, *c* der Centroid dem der Punkt zur Zeit zugeordnet ist und *c'* irgendein anderer Centroid ist, dann besagt Lemma 1, dass wenn $\frac{1}{2}d(c,c') \geq d(x,c)$, dann ist $d(x,c') \geq d(x,c)$. In diesem Fall ist es nicht notwendig $d(x,c')$ zu berechnen. Angenommen die exakte Distanz für $d(x,c)$ ist nicht bekannt. Allerdings ist eine Obergrenze *u* bekannt, sodass $u \geq d(x,c)$ ist. Dann müssen die Distanzen $d(x,c')$ und $d(x,c)$ nur berechnet werden, wenn $u > \frac{1}{2}d(c,c')$. Falls $u \leq \frac{1}{2}min\ d(c,c')$ dann bleibt der Punkt *x* seinem Centroiden *c* zugeordnet, und alle Distanzberechnungen des Punkts *x* zu anderen Centroiden können vermieden werden.

Lemma 2 wird wie folgt benutzt. *x* ist ein beliebiger Punkt, *b* ein beliebiger Centroid und *b'* die vorherige Version des selben Centroiden. Angenommen die Centroide sind durchnummeriert von 1 bis *k*, und *b* ist Centroid Nummer *j*, dann ist *b'* Centroid Nummer *j* in der vorherigen Iteration. Falls in der vorherigen Iteration eine Untergrenze *l'* bekannt war, sodass $d(x,b') \geq l'$, dann kann eine Untergrenze *l* für diese Iteration wie folgt abgeleitet werden:

$$d(x,b) \geq \max\{0, d(x,b') - d(b,b')\} \geq \max\{0,l' - d(b,b')\} = l.$$

Wenn *l'* eine gute Annäherung zwischen der vorherigen Distanz von *x* und dem *j*-ten Centroide war, und der Centroid sich nur um eine kleine Distanz verschoben hat, dann ist *l* auch eine gute Annäherung zur neuen Distanz.

Verbesserter K-Means Algorithmus mit Dreiecksungleichung

Basierend auf den beiden oben angeführten Lemma zur Dreiecksungleichung wird nachfolgend der verbesserte K-Means Algorithmus vorgestellt [EL03]:

Zuerst werden die initialen Centroide ausgewählt. Anschließend wird die Untergrenze *l(x,c)* für jeden Punkt und jeden Centroid gleich 0 gesetzt. Jeder Punkt *x* wird zu seinem nähesten initialen Centroiden $c(x) = min_c d(x,c)$ gemäß Lemma 1 zugeteilt. Bei jeder Berechnung von *d(x,c)* wird die Untergrenze *l(x,c) = d(x,c)* gesetzt. Die Obergrenze ist gleich $u(x) = min_c d(x,c)$. Die nachfolgenden Schritte werden so lange wiederholt, bis sich die Zuordnung der Punkte nicht mehr ändert:

1. Für alle Centroide *c* und *c'* wird d(c,c') berechnet und für alle Centroide c wird $s(c) = {}^1/_2 min_{c \neq c} d(c,c')$ berechnet.

2. Ermittlung aller Punkte *x* die die Bedingung $u(x) \leq s(c(x))$ erfüllen.

3. Für alle übrigen Punkte *x* und Centroide *c* die folgenden Bedingungen erfüllen

   I. $c \neq c(x)$ und

   II. $u(x) > l(x,c)$ und

   III. $u(x) > {}^1/_2 d(c(x),c)$ gelten folgende Bedingungen:

      a. Wenn *r(x) = true* dann wird *d(x,c(x))* berechnet und *r(x) = false* gesetzt. Andernfalls ist *d(x,c(x)) = u(x)*.

      b. Wenn $d(x,c(x)) > l(x,c)$ oder $d(x,c(x)) > {}^1/_2 d(c(x),c)$ dann wird d(x,c) berechnet. Falls $d(x,c) < d(x,c(x))$ dann wird $c(x) = c$ zugeordnet.

4. Für jeden Centroiden *c* ist *m(c)* der Mittelwert der Punkte die *c* zugeordnet sind

5. Setze für jeden Punkt *x* und jeden Centroiden *c* $l(x,c) = max\{l(x,c) - d(c,m(c)),0\}$

6. Setze für jeden Punkt x $u(x) = u(x) + d(m(c(x)),c(x))$ und $r(x) = true$

7. Ersetze jeden Centroiden *c* durch *m(c)*.

In Schritt 3 wird bei jeder Berechnung von *d(x,c)* für jedes *x* und *c* die Untergrenze aktualisiert mit *l(x,c) = d(x,c)*. Genauso wird *u(x)* neu ermittelt, wenn sich *c(x)* ändert oder *d(x,c(x))* neu berechnet wird.

Der Grund für die Reduzierung der Anzahl der Distanzberechnungen ist, dass die Obergrenzen *u(x)* und die Untergrenzen *l(x,c)* zu Beginn jeder Iteration sehr gut für die

meisten Punkte *x* und die Centroide *c* angepasst sind. Wenn die gewählten Grenzen zu Beginn der Iteration „gut“ sind, dann sind sie es meist auch am Ende der Iteration und somit auch am Beginn der nächsten Iteration. Dies beruht darauf, dass sich die Centroide und somit auch die Grenzen nur geringfügig verschieben. Für experimentelle Ergebnisse vergleiche [EL03].

4.2 Optimierungsmöglichkeiten des DBSCAN Algorithmus

Der Aufwand des DBSCAN-Algorithmus beim Ermitteln des Clustering ist O(n * Aufwand zur Bestimmung der ε-Nachbarschaft), d.h. einen Ansatz für Optimierungen liefert die Ermittlung der ε -Nachbarschaft (siehe auch Kap. 2.2.2.2). Da bei dieser Ermittlung der ε -Nachbarschaft vor allem die Datenbank involviert ist, da der Abstand eines Objekts zu allen anderen ermittelt werden muss, ist vor allem hier das Potential für Optimierungen gegeben. Näheres zu den möglichen Verbesserungen durch Indexstrukturen für eine Leistungssteigerung siehe bei Kap. 4.3.

Parameterbestimmung für ε und MinPts

Das Ergebnis des DBSCAN-Clustering hängt stark von seinen Parametern ε und MinPts ab. Wird zum Beispiel der Parameter MinPts zu niedrig gewählt, d.h. mit 1 oder 2, dann kann es zum so genannten „Single-Link-Effekt“ führen. Falls verschiedene Cluster durch eine Linie von Punkten verbunden sind, und der Abstand kleiner als ε ist, dann werden diese Cluster verschmolzen. Bei einem größeren Wert für MinPts kann dieser Effekt vermieden werden. Wird der Wert allerdings zu groß angenommen, kann dies dazu führen, dass nur ein einziger, bzw. nur sehr wenige Cluster gefunden werden, obwohl mehrere Cluster vorhanden sind. Gute Werte für die beiden Parameter ε und MinPts wären die Werte, die den am wenigsten dichten Cluster charakterisieren, da das DBSCAN Verfahren alle Cluster findet, deren Dichte größer ist als die Dichte, die durch die beiden Parameter gegeben ist. Diese Werte sind aber meist nicht bekannt und können daher nur heuristisch ermittelt werden.

Eine solche Heuristik basiert auf dem Diagramm der sortierten *knn*-Distanzen. Dazu wird für ein gegebenes k ≥ 1 eine Funktion *k-Distanz* definiert, die jedem Objekt die Distanz zu seinem k-nächsten Nachbarn zuordnet. Diese Distanzen werden absteigend sortiert und graphisch angezeigt [ES00]. Der Graph dieser sortierten k-Distanzen lässt einige Rückschlüsse auf die Dichteverteilung in der Datenmenge zu. Die ist relativ einfach dadurch zu begründen, dass Objekte in dichten Gebieten kleinere k-Distanzen haben, als Objekte in weniger dichten Gebieten. Abbildung 19 zeigt beispielhaft ein solches k-Distanz Diagramm.

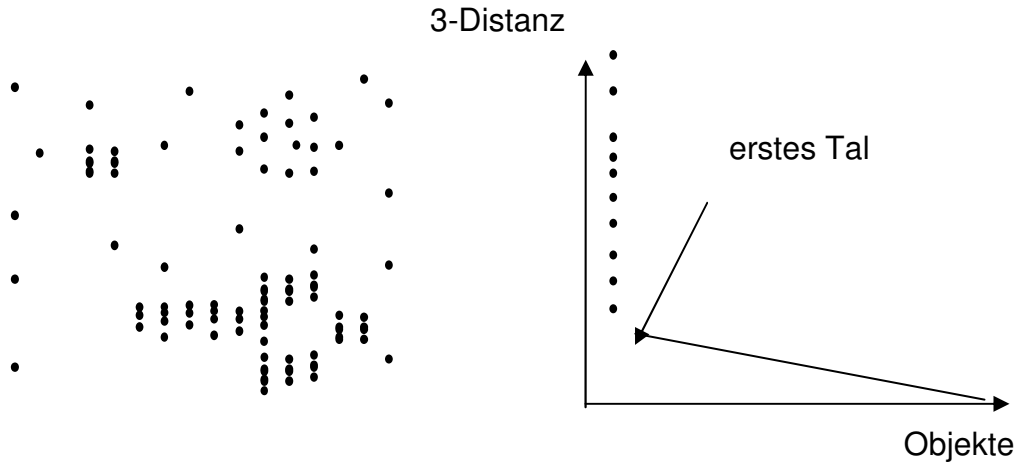


Abbildung 19: Beispiel für ein k-Distanz Diagramm (k = 3) für die dargestellte Punktmenge

Bei der Auswahl der Parameter ε und MinPts ist folgender Zusammenhang zwischen den k-Distanzen und den dichte-basierten Clustern zu beachten: wenn man ein beliebiges Objekt *p* aus der Datenmenge auswählt, den Parameter ε gleich k-Distanz(p) und den Parameter MinPts gleich k+1 setzt, dann werden alle Objekte mit gleicher oder kleinerer k-Distanz zu Kernobjekten [ES00]. Das Objekt *p* soll, wenn möglich, ein Grenzobjekt mit maximaler k-Distanz im „dünnsten", d.h. am wenigsten dichten, Cluster sein. Dadurch ergeben sich für die gegebene Heuristik die Fragen, welcher Wert für *k* angenommen wird, und wie man ein Grenzobjekt im k-Distanz Diagramm erkennt. Als guter Default-Wert für den Parameter *k* hat sich die Regel $k = 2 * d - 1$ [ES00] erwiesen, wobei d die Anzahl der Dimensionen der zu clusternden Menge ist. Abgeleitet von diesem k kann der Parameter MinPts als $MinPts = k + 1$ oder $MinPts = 2 * d$ angenommen werden. Nun stellt sich nur noch die Frage, wie ein Grenzobjekt aus dem k-Distanz Diagramm ermittelt werden kann. Gewöhnlich reicht es einen Punkt in der Nähe des ersten „Tals" des k-Distanz Diagramms auszuwählen, vor allem wenn Rauschen und Cluster gut unterscheidbar sind. Die Auswahl des genauen Grenzobjekts erfolgt allerdings vom Benutzer. Für die Erstellung des k-Distanz Diagramms reicht es eine kleine Stichprobe der gesamten Punktmenge heranzuziehen, da dieses meist schon die gleiche Form hat wie das Diagramm für die gesamte Datenmenge.

Heuristik zur Bestimmung der Parameter ε und MinPts:

1. Auswahl eines Wertes für *k* (Default ist $k = 2*d - 1$) und des Parameters MinPts mit $MinPts = k + 1$.

2. k-Distanz Diagramm für eine kleine Stichprobe der Datenmenge ermitteln und graphisch darstellen.

3. Auswahl eines Grenzobjekts *o* durch den Benutzer in der Nähe des ersten „Tals“. Der Parameter ε entspricht ε = k-Distanz(o).

Ohne eine Heuristik zur Ermittlung der Parameter ε und MinPts erfolgt die Bestimmung dieser Parameter meist willkürlich bzw. zufällig. Dies kann dann zu einem unerwarteten Ergebnis führen, dass zum Beispiel nur ein Cluster liefert, oder viele kleine Cluster. In diesem Fall ist es dann oft nötig, den Algorithmus noch einmal mit geänderten Parametern laufen zu lassen. Bis das „gewünschte“ oder „erwartete“ Ergebnis vorliegt, sind dann meist mehrere Durchläufe des Algorithmus notwendig. Um dies zu vermeiden, kann die oben vorgestellte Heuristik zur Bestimmung der Parameter herangezogen werden. Dadurch „spart“ man sich die mehrfachen Durchläufe, und eine Optimierung des Verfahrens ist indirekt erreicht worden.

4.3 Indexstrukturen für Datenbanken zur Leistungssteigerung

Wie bereits kurz in der Einleitung zu Kapitel 4.2 erwähnt, gibt es Möglichkeiten mit Hilfe von Indexstrukturen für Datenbanken die Ausführung eines Clusteringalgorithmus zu beschleunigen. Des Weiteren kann durch die Anwendung von solchen Indexstrukturen eine größere Datenmenge für das Clustering verwendet werden. Das Grundprinzip zur Beschleunigung eines Clusteringalgorithmus ist die Verwendung von räumlichen Indexstrukturen oder speziell entwickelten Datenstrukturen, die räumlichen Indexstrukturen sehr ähnlich sind. Folgende Eigenschaften von räumlichen Indexstrukturen lassen sich ausnutzen, um Clusteringalgorithmen zu beschleunigen (siehe auch [EKSX98]):

- Indexstrukturen können auch als einfache Clusteringverfahren interpretiert werden, da sie versuchen, ähnliche Objekte, d.h. räumlich benachbarte Objekte, möglichst auf einer gemeinsamen Datenseite abzuspeichern.

- Indexstrukturen liefern ein grobes Vor-Clustering sehr schnell, da sie in der Regel nur einfache Heuristiken zum Clustering verwenden, um die Aufbauzeit für den Index möglichst klein zu halten.

- Da die Daten in räumlichen Indexstrukturen, wie bereits erwähnt, einem groben Vor-Clustering entsprechen, ermöglichen diese schnelle Zugriffsmethoden für verschiedene Ähnlichkeitsanfragen, wie Bereichsanfragen. Ein Beispiel für eine

solche Bereichsanfrage wäre die Bestimmung der ε-Nachbarschaft beim DBSCAN-Algorithmus.

4.3.1 Stichprobenverfahren – Indexbasiertes Sampling

Bei einem Stichprobenverfahren wird, wie der Name schon sagt, eine Stichprobe, basierend auf einem Datenbankindex, gezogen, auf die das Clustering ausgeführt wird. Ausgehend von dem Ergebnis der geclusterten Stichprobe kann anschließend das Clustering auf die Gesamtmenge ermittelt werden. Nachfolgend wird die Vorgehensweise eines solchen Verfahrens geschildert (siehe auch [EKX95a] und [EXK95b]):

1. Erstelle mit den zu clusternden Daten einen R-Baum, falls noch nicht vorhanden.

2. Wähle pro Datenseite des R-Baums einen oder mehrere Repräsentanten für die Stichprobe aus.

3. Ausführen des Clusteringverfahren nur auf die ausgewählte Repräsentantenmenge.

Die nachfolgende Abbildung 20 zeigt die Datenseitenstruktur eines R*-Baums (das ist eine bestimmte Variante eines R-Baums) für die gegebene Punktmenge. Jede Datenseite des Baums enthält die gleiche Anzahl von Punkten.

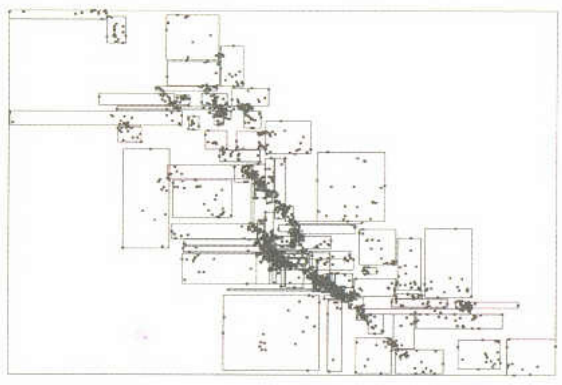


Abbildung 20: Datenseitenstruktur eines R*-Baums [ES00]

Dadurch sind Datenseiten in weniger dichten Gebieten größer als Datenseiten in sehr dichten Gebieten. Dadurch ergibt sich aus den repräsentativen Objekten der

Datenseiten eine sehr gut verteilte Stichprobe. Die Anzahl der auszuwählenden repräsentativen Punkte hängt vom verwendeten Clusteringverfahren ab. Beim Ermitteln des Clustering für die Gesamtmenge, ausgehend vom Clustering der Stichprobe, gibt es je nach dem verwendeten Verfahren verschiedene Möglichkeiten. Beim K-Means Verfahren, zum Beispiel, können die ermittelten Centroide der Stichprobe für die Gesamtmenge herangezogen werden. Bei dichte-basierten Verfahren muss zuerst eine Repräsentation der Cluster gebildet werden, zum Beispiel in Form von repräsentativen Punkten. Ausgehend von diesen Repräsentanten werden die Objekte dem „besten“, zum Beispiel dem nächstgelegenen, der gefunden Cluster zugeordnet.

4.3.2 Indexunterstützte Bereichsanfragen für dichte-basiertes Clustering (DBSCAN)

Die Ermittlung der ε-Nachbarschaft beansprucht einen großen Teil des Aufwands beim Durchführen eines Clustering mit dem DBSCAN-Algorithmus. Vor allem dadurch, dass die Ermittlung der ε-Nachbarschaft für jedes Objekt der Datenmenge durchgeführt werden muss. Ohne Verwendung einer Indexstruktur kann diese ε-Nachbarschaft einfach dadurch berechnet werden, dass die Distanz eines Objekts o zu allen anderen Objekten p ermittelt wird, und mit diesen ermittelten Distanzen die Bedingung $dist(o, p) \leq \varepsilon$ geprüft wird. Solche Bereichsanfragen mit einem Zentrum, dies entspricht dem Objekt o, und einem Radius ε werden allerdings auch durch räumliche Indexstrukturen wie den R*-Baum unterstützt. Bei Verwendung eines solchen R*-Baums kann zumindest in niedrig dimensionalen Räumen eine solche Bereichsanfrage in logarithmischer Laufzeit durchgeführt werden. Bei einem direkten Zugriff auf die ε-Nachbarschaft der Objekte, wie etwa beim Gridfile [NHS84], ist der Zugriff auf die Nachbarschaft in konstanter Zeit möglich. In der nachfolgenden Tabelle 3 sind die Laufzeitkomplexitäten mit und ohne Index dargestellt.

Laufzeitkomplexität	**einer einzelnen Bereichsanfrage**	**DBSCAN-Algorithmus**
ohne Index	O(n)	$O(n^2)$
mit räumlichen Index	O(log n)	O(n * log n)
mit direktem Zugriff	O(1)	O(n)

Tabelle 3: Laufzeitkomplexitäten mit und ohne Indexunterstützung

4.3.3 Indexstruktur als Ausgangsbasis für Clustering (GRID-Clustering)

Bei diesem Verfahren wird davon ausgegangen, dass die räumliche Indexstruktur bereits ein grobes Clustering der Daten liefert, da ähnliche Objekte auf einer Datenseite gespeichert werden. Ausgehend von diesem Grob-Clustering wird das Clustering durchgeführt, indem das Grob-Clustering durch Verschmelzen von Seitenregionen nur noch nachbearbeitet wird. Seitenregionen mit hoher Punktdichte werden als Clusterzentren angesehen und rekursiv mit benachbarten weniger dichten Seitenregionen verschmolzen. Bezeichnet wird ein solches Verfahren als Grid-Clustering.

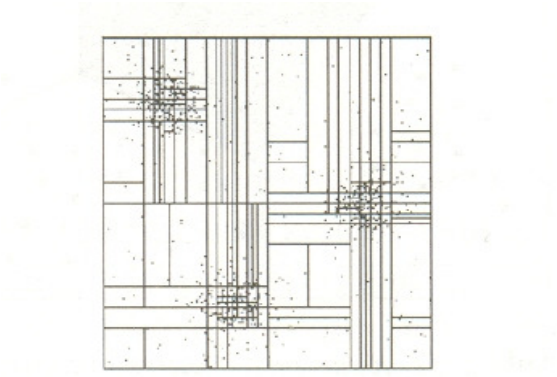


Abbildung 21: Aufteilung einer 2-dimensionalen Datenmenge durch die Datenseite eines Gridfiles [ES00]

Bei einem von Schikuta [SCH96] vorgestellten Gird-Clustering-Verfahren wird das Gridfile als Indexstruktur verwendet. Ähnlich wie für die Datenseiten eines R-Baums gilt auch für die Datenseiten des Gridfile, dass das Volumen des durch die Seite repräsentierten Datenraums umso kleiner ist, je dichter die Punkte in diesem Gebiet liegen. Das Prinzip des Grid-Clustering-Verfahrens lässt sich analog auf die Datenseiten aller Indexstrukturen übertragen, für die diese Eigenschaft gilt [ES00]. Abbildung 21 zeigt die Aufteilung einer Punktemenge durch die Datenseiten des Gridfile.

5 Verbesserungsmöglichkeiten durch „Kombiniertes Data Mining“ - Vorgänger kennt Nachfolger

Das vorige Kapitel stellte Verbesserungsmöglichkeiten der Algorithmen K-Means und DBSCAN und mögliche Indexstrukturen für Datenbanken vor. Die Optimierung der Data Mining Algorithmen und die Nutzung von Indexstrukturen führt zu einer Verbesserung aller drei Arten des „Kombinierten Data Mining“, dadurch, dass die Einzelverfahren beschleunigt werden. Anschließend werden Verbesserungsmöglichkeiten aufgezeigt, die durch den Einsatz des „Kombinierten Data Mining“ möglich sind. Dabei wird ausschließlich auf die Art „Vorgänger kennt Nachfolger“ des „Kombinierten Data Mining“ näher eingegangen. Die Art „Nachfolger kennt Vorgänger“ des „Kombinierten Data Mining“ wird von Humer [HM04] näher behandelt.

Die „Kombinierte Data Mining“ Art *Vorgänger kennt Nachfolger* ist dadurch gekennzeichnet, dass das Vorgängerverfahren Hilfsinformationen für das Nachfolgeverfahren ermittelt (siehe Kap. 3). Als Nachfolgeverfahren kann im Prinzip jeder Klassifikationsalgorithmus herangezogen werden. Dieser muss lediglich so erweitert werden, dass dieser die Hilfsinformationen nutzen und verarbeiten kann. Nachfolgend werden die ermittelten Hilfsinformationen für die beiden implementierten Data Mining Algorithmen, K-Means und DBSCAN, erläutert. Ziel war es, so viele Hilfsinformationen so effizient wie möglich zu ermitteln. Beide Algorithmen ermitteln die selben Hilfsinformationen. Allerdings werden sie auf unterschiedliche Art ermittelt. Die Realisierung der Ermittlung der Hilfsinformationen in der Implementierung wird im nachfolgenden Kapitel 6 erläutert.

Die ermittelten Hilfsinformationen können in zwei Arten geteilt werden. Einerseits können Hilfsinformationen zur gesamten Datenbasis ermittelt werden, und andererseits können Hilfsinformationen zu den gefundenen Clustern gespeichert werden. Durch diese ermittelten Hilfsinformationen soll das Nachfolgeverfahren (vergleiche Humer [HM04]) so profitieren, dass das Ergebnis, gemessen an einem ausgewählten Gütemaß (z.B. tatsächlicher Klassifikationsfehler [ES00]), besser ist, als ohne diese Hilfsinformationen. Unter Umständen kann auch ein Performancegewinn erzielt werden, da gewisse Berechnungen des Nachfolgeverfahrens vereinfacht werden. Bei bestehenden naiven „Kombinierten Data Mining“ Verfahren (siehe Kap. 3.1 und 3.2) werden die beiden Verfahren oft mehrmals ausgeführt bis das „gewünschte“ Ergebnis erreicht wird. Durch die Hilfsinformationen sollte die Zahl der nötigen Wiederholungen verringert werden (vergleiche Humer [HM04]).

5.1 Hilfsinformationen zur Datenbasis

Zu jeder Dimension, der für die beiden Algorithmen herangezogenen Datenbasis, werden folgende Hilfsinformationen ermittelt:

- **Summe der Werte pro Dimension:**

  Die Summe der Werte pro Dimension wird für die Berechnung von weiteren Hilfsinformationen, wie Mittelpunkt und Standardabweichung benötigt. Außerdem kann dieser Wert für mathematische Berechnungen verwendet werden.

- **Quadratsumme der Werte pro Dimension:**

  Die Quadratsumme der Werte pro Dimension wird für die Berechnung der Standardabweichung, die auch eine Hilfsinformation darstellt, benötigt. Des Weiteren kann diese für mathematische Berechnungen verwendet werden.

- **Mittelwert der Werte pro Dimension (Mittelpunkt):**

  Der Mittelwert wird mit Hilfe der Summe pro Dimension und der Anzahl der Punkte berechnet, d.h. dieser muss nicht während der Iterationen über die Punkte des Datenbestands ermittelt werden, sondern erst nachdem alle verwendeten Punkte einmal aus der Datenbank gelesen wurden. Daraus ergibt sich, dass die Berechnung des Mittelwerts nur einmal erfolgt. Der Mittelwert kann mit den Hilfsinformationen Minimum, Maximum, 25%-Quantil, Median, 75%-Quantil und Standardabweichung für die Ermittlung einer ungefähren Verteilung der Punkte herangezogen werden. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden.

- **Standardabweichung der Werte pro Dimension:**

  Die Standardabweichung wird mit Hilfe der Summe, Quadratsumme und der Anzahl der Punkte berechnet. Genauso wie beim Mittelwert wird die Standardabweichung erst berechnet, wenn die nötigen Hilfsinformationen bereits berechnet sind. Daraus ergibt sich, dass die Berechnung der Standard-abweichung nur einmal erfolgt. Wie der Mittelwert kann die Standardabweichung mit den anderen Hilfsinformationen zur Ermittlung einer

Verteilung verwendet werden. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden.

- **Minimum pro Dimension:**

  Das Minimum wird wie der Mittelwert und die Standardabweichung für die Ermittlung einer Verteilung benötigt. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden.

- **Maximum pro Dimension:**

  Das Maximum wird wie der Mittelwert, die Standardabweichung und das Minimum für die Ermittlung einer Verteilung benötigt. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden.

- **Ungefähre Median pro Dimension (Annäherung):**

  Der Median wird näherungsweise berechnet. Wie der Mittelwert, die Standardabweichung, das Minimum und das Maximum kann dieser für die Ermittlung einer Verteilung benötigt werden. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden. Details zur Annäherung siehe Kap. 6.3.2.1.

- **Ungefähre 25 % Quantil der Dimension (Annäherung):**

  Das 25%-Quantil wird näherungsweise berechnet. Wie der Mittelwert, die Standardabweichung, das Minimum, das Maximum und der Median kann dieser für die Ermittlung einer Verteilung benötigt werden. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden. Details zur Annäherung siehe Kap. 6.3.2.1.

- **Ungefähre 75 % Quantil der Dimension (Annäherung):**

  Das 75%-Quantil wird näherungsweise berechnet. Wie der Mittelwert, die Standardabweichung, das Minimum, das Maximum, der Median und das 25%-

Quantil kann dieser für die Ermittlung einer Verteilung benötigt werden. Die Verteilung der Punkte kann für die Splitstrategie eines Klassifikationsalgorithmus zur Ermittlung von Entscheidungsbäumen verwendet werden. Details zur Annäherung siehe Kap. 6.3.2.1.

- **Anzahl der Punkte pro Dimension:**

  Dieser Wert ist natürlich für alle Dimensionen gleich und wird zur Berechnung des Mittelwerts und der Standardabweichung benötigt. Des Weiteren kann dieser Wert für mathematische Berechnungen verwendet werden.

Die angeführten Hilfsinformationen zur Datenbasis werden, bis auf die Quantile, beim K-Means Algorithmus in der ersten Iteration berechnet, da in dieser bereits die komplette Datenbasis durchlaufen wird. In den weiteren Iterationen ist die Berechnung dieser Hilfsinformationen daher auch nicht mehr nötig. Der Mittelwert und die Standardabweichung werden einmal am Ende der ersten Iteration berechnet. Die genaue Umsetzung der Implementierung der Ermittlung der Hilfsinformationen zur Datenbasis kann unter 6.2.2.1 gefunden werden.

Die angeführten Hilfsinformationen zur Datenbasis werden, bis auf die Quantile, beim DBSCAN-Algorithmus bei der ersten Ermittlung der ε-Nachbarschaft berechnet, da in dieser bereits die komplette Datenbasis durchlaufen wird. Bei den weiteren Ermittlungen der ε-Nachbarschaften ist eine Berechnung der Hilfsinformationen nicht mehr nötig. Der Mittelwert und die Standardabweichung werden ähnlich wie beim K-Means Algorithmus erst nach der ersten Ermittlung der ε-Nachbarschaft berechnet. Nähere Informationen zur Implementierung der Hilfsinformationen können unter 6.3.2.1 gefunden werden.

5.2 Hilfsinformationen zu den gefundenen Clustern

Zu jedem der gefundenen Cluster werden folgende Hilfsinformationen für jede Dimension des Clusters berechnet:

- **Summe der Werte pro Dimension:**

  Die Summe der Werte pro Dimension wird für die Berechnung von weiteren Hilfsinformationen, wie Mittelpunkt (Centroid) und Standardabweichung des Clusters, benötigt. Außerdem kann dieser Wert für mathematische Berechnungen verwendet werden.

- **Quadratsumme der Werte pro Dimension:**

  Die Quadratsumme der Werte pro Dimension wird für die Berechnung der Standardabweichung des Clusters, die auch eine Hilfsinformation darstellt, benötigt. Des Weiteren kann diese für mathematische Berechnungen verwendet werden.

- **Mittelwert pro Dimension (Mittelpunkt, Centroid):**

  Der Mittelwert (Centroid) wird mit Hilfe der Summe pro Dimension und der Anzahl der Punkte berechnet. Dieser wird immer dann berechnet, wenn dem Cluster ein Punkt hinzugefügt oder entfernt wird. Der Mittelpunkt wird für die Berechnung der Distanz eines Punktes zu diesem Cluster benötigt. Der Mittelwert kann mit den Hilfsinformationen Minimum, Maximum und Standardabweichung für die Ermittlung einer ungefähren Verteilung der Punkte im Cluster herangezogen werden. Durch diese Verteilungen pro Cluster können eventuelle Überlappungen der Cluster festgestellt werden.

- **Standardabweichung pro Dimension:**

  Die Standardabweichung wird mit Hilfe der Summe, Quadratsumme und der Anzahl der Punkte berechnet. Die Standardabweichung für einen Cluster wird erst am Ende der Algorithmen berechnet, da erst da alle Punkte einem Cluster zugeordnet sind. Daraus ergibt sich, dass die Berechnung der Standardabweichung nur einmal erfolgt. Wie der Mittelwert kann die Standardabweichung mit den anderen Hilfsinformationen zur Ermittlung einer Verteilung im Cluster verwendet werden. Durch diese Verteilungen pro Cluster können eventuelle Überlappungen der Cluster festgestellt werden.

- **Anzahl der Punkte pro Dimension:**

  Dieser Wert ist natürlich für alle Dimensionen gleich und wird zur Berechnung des Mittelwerts (Centroiden) und der Standardabweichung benötigt. Des Weiteren kann dieser Wert für mathematische Berechnungen verwendet werden. Dieser Wert gibt Auskunft wie viele Punkte dem Cluster zugeordnet wurden.

- **Wurzel über die Anzahl der Punkte pro Dimension:**

  Dieser Wert wird mit Hilfe der Anzahl der Punkte berechnet. Genauso wie bei der Standardabweichung erfolgt die Berechnung des Werts erst am Ende der Algorithmen, da erst da alle Punkte einem Cluster zugeordnet sind.

- **Minimum pro Dimension:**

  Das Minimum wird wie der Mittelwert und die Standardabweichung für die Ermittlung einer Verteilung benötigt. Durch diese Verteilungen pro Cluster können eventuelle Überlappungen der Cluster festgestellt werden.

- **Maximum pro Dimension:**

  Das Maximum wird wie der Mittelwert, die Standardabweichung und das Minimum für die Ermittlung einer Verteilung benötigt. Durch diese Verteilungen pro Cluster können eventuelle Überlappungen der Cluster festgestellt werden.

- **Nähesten Punkte eines Clusters:**

  Pro Cluster wird eine vom Benutzer vorgegeben Zahl von nähesten Punkten ermittelt. Diese entsprechen den repräsentativsten Punkten des Clusters.

- **Entferntesten Punkte eines Clusters:**

  Pro Cluster wird eine vom Benutzer vorgegeben Zahl von entferntesten Punkten ermittelt. Diese Punkte entsprechen den am wenigsten repräsentativen Punkten des Clusters.

Zu jedem der gefundenen Cluster wird eine bestimmte Anzahl von Punkten der nähesten und entferntesten Punkte des Clusters ermittelt. Die Anzahl der nähesten bzw. entferntesten Punkte wird jeweils über einen Parameter beim Aufruf der Algorithmen übergeben, d.h. der Benutzer steuert diesen Wert. Beim K-Means Algorithmus wird bei der Ermittlung dieser Punkte die Distanz der Punkte zu ihrem Centroiden als Maß für die Nähe und die Entfernung herangezogen. Je geringer die Distanz eines Punktes zu seinem Centroiden, desto näher ist dieser Punkt, und vice versa. Beim DBSCAN Algorithmus gilt bei der Ermittlung dieser Punkte ein Kernobjekt (siehe Kap. 2.2.2.2) als naher Punkt, und ein Randobjekt, d.h. kein Kernobjekt, als entfernter Punkt.

Genauere Informationen bezüglich der Implementierung und der näheren Realisierung der Hilfsinformationen können für den K-Means unter Kap. 6.2.2.2 und für den DBSCAN unter Kap. 6.3.2.2 gefunden werden.

Die angeführten Hilfsinformationen erheben keinen Anspruch auf Vollständigkeit, d.h. es kann durchaus noch weitere Hilfsinformationen geben, die genutzt werden können. Allerdings wurde bei diesen Hilfsinformationen darauf geachtet, dass diese möglichst effizient ermittelt werden. Ein Beispiel für eine weitere mögliche Hilfsinformation wäre der Silhouetten-Koeffizient (siehe Kap. 4.1.2.1). Dieser ist ein Gütemaß für die Zuordnung eines Punkts zu seinem Cluster. Zur Berechnung des Silhouetten-Koeffizienten benötigt man den zugeordneten Cluster und den „zweitnähesten“ Cluster des Punkts. Um diese Hilfsinformation speichern zu können, müsste man diese für jeden Punkt einzeln speichern. Dies bedeutet nicht nur eine zusätzliche Spalte (Dimension) in der Datenbank sondern auch, dass ein Datenbank-Update für jeden Punkt nötig ist. Vor allem sind diese Updates bei jeder Iteration des K-Means nötig. Dadurch wäre in jeder Iteration ein Datenbank-Update auf jeden Punkt nötig. Da gerade die Datenbankoperationen im Sinne der Laufzeit sehr teure Operationen sind, würden diese eine deutliche Steigerung der Laufzeit der Algorithmen verursachen. Aus Effizienzsicht ist daher die Ermittlung des Silhouetten-Koeffizienten nicht vertretbar.

6 „Kombiniertes Data Mining“ – Dokumentation der Implementierung des ersten Schritts mit K-Means und DBSCAN

Nachdem in den bisherigen Kapiteln ausschließlich das Prinzip des „Kombinierten Data Mining“ vorgestellt wurde, zeigt dieses Kapitel eine praktische Realisierung des ersten Schritts des „Kombinierten Data Mining“ in Form einer Implementierung der beiden Clustering Algorithmen K-Means und DBSCAN. In den beiden Algorithmen K-Means und DBSCAN werden jeweils die in Kapitel 5 angegebenen Hilfsinformationen ermittelt. Durch die ermittelten Hilfsinformationen soll in einem zweiten Schritt, nämlich der Klassifikation, ein „besseres“ Ergebnis erzielt werden [HM04].

Nachfolgend werden die Implementierungen der beiden Algorithmen im Detail erläutert, wobei der Hauptaugenmerk auf die Ermittlung der Hilfsinformationen gelegt werden. Die wichtigen Programmpassagen werden anhand von Auszügen aus dem Quellcode erläutert. Des Weiteren wird erklärt warum die Ermittlung gerade an den ausgewählten Stellen implementiert wurde. Bevor nun auf die Details der Implementierung eingegangen wird, wird zuerst die Architektur für die Umsetzung der beiden Algorithmen erklärt.

6.1 Architektur

Dieser Abschnitt soll die Architektur zur Umsetzung der beiden Algorithmen kurz erläutern. Die nötigen Daten für das Clustering werden aus einer Datenbank gelesen. Das Ergebnis des Clustering wird anschließend mit den Hilfsinformationen in der Datenbank gespeichert. Die Implementierung der Algorithmen erfolgte mittels der Programmiersprache C++. und wurde mit der Unterstützung der Entwicklungsumgebung Visual Studio C++ 6.0 realisiert. Die Verbindung der Oracle Datenbank mit C++ wurde mit Embedded SQL verwirklicht. Für die Kompilierung des eingebundenen Embedded SQL Quellcode muss zuerst der Precompiler ProC/C++ verwendet werden, um den Embedded SQL Quellcode in C++ - Quellcode umzuwandeln. Nachfolgend ist die Architektur der Umsetzung graphisch dargestellt:

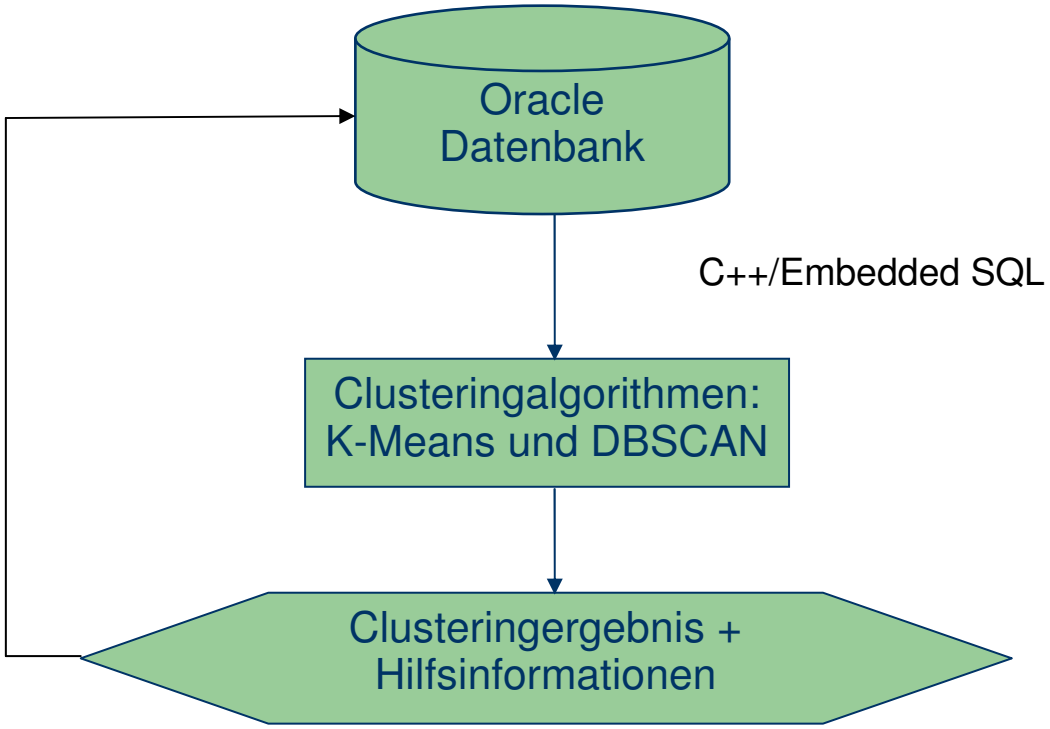


Abbildung 22: Architektur der Umsetzung

Abbildung 22 zeigt die beschriebene Architektur. Die Implementierung der beiden Algorithmen K-Means und DBSCAN werden nachfolgend beschrieben. Dabei wird zuerst jeweils die Klassentopologie mittels UML-Diagramm vorgestellt. Im Anschluss daran werden die einzelnen Klassen mit ihren Attributen und Methoden beschrieben. Darauf folgt die detaillierte Beschreibung der Implementierung des Algorithmus und vor allem der Ermittlung der Hilfsinformationen.

6.2 Erweiterter K-Means Algorithmus

Der nachfolgend vorgestellte K-Means Algorithmus basiert in den Grundzügen auf dem „Standard“ K-Means Algorithmus (siehe Kap. 2.2.2.1). Zusätzlich werden noch die Hilfsinformationen ermittelt. Anschließend werden die Klassen des implementierten K-Means Algorithmus mittels UML-Diagramm dargestellt. Das UML-Diagramm enthält alle verwendeten Klassen, inklusive der Klassen für die Hilfsinformationen und die Datenbankanbindung. Pro Klasse werden die wichtigsten Attribute und die wichtigsten Methoden angeführt. Der Quellcode der Klassen und deren Attribute und Methoden kann im Anhang (siehe Kap. 12.1) gefunden werden. Im Anschluss an das UML-Diagramm werden die Klassen und deren wichtige Attribute und Methoden näher erläutert.

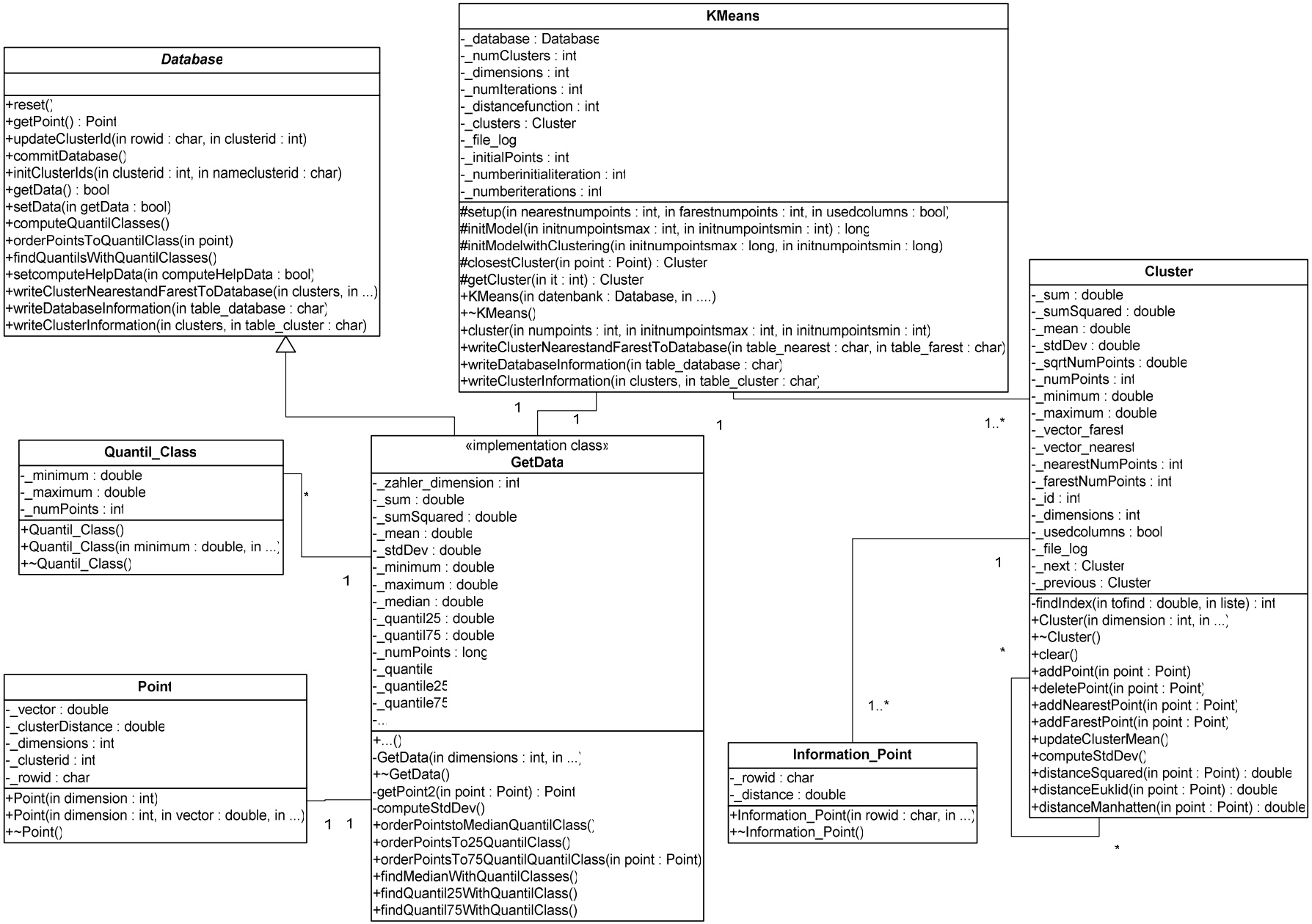


Abbildung 23: UML-Diagramm des erweiterten K-Means Algorithmus

Database

Die Klasse *Database* ist eine abstrakte Klasse und enthält alle Methoden die eine Klasse für die Datenbankanbindung des K-Means Algorithmus benötigt. Eine Klasse die die Datenbankanbindung für den K-Means Algorithmus realisieren soll, muss von der Klasse *Database* erben und dessen Methoden implementieren. Nachfolgend werden die Methoden der Klasse beschrieben:

- **reset():**

  Die Methode *reset* setzt den Datenbestand (Datenbankcursor) zurück und liefert beim nächsten Aufruf der Methode *getPoint* wieder den ersten Punkt.

- **getPoint():**

  Die Methode *getPoint* liefert als Rückgabewert den nächsten Punkt des Datenbestands. Zusätzlich werden in dieser Methode ein Teil der Hilfsinformationen der Datenbasis berechnet (siehe Kap. 6.2.2.1).

- **updateClusterId(char *rowid, int clusterid):**

  Die Methode *updateClusterId* führt ein Datenbankupdate der Cluster-Id eines Punkts durch. Der Punkt wird über seine Rowid identifiziert und die neue Cluster-Id wird übergeben.

- **commitDatabase():**

  Die Methode *commitDatabase* setzt ein Datenbank-Commit ab. Dies ist nach jeder Iteration des K-Means Algorithmus notwendig.

- **initClusterIds(char *clusterid, char *nameclusterid):**

  Die Methode *initClusterIds* initialisiert alle Punkte mit der übergebenen Cluster-Id. Welche Spalte (Dimension) der Datenbank die Cluster-Id repräsentiert, wird mit dem Attribut *nameclusterid* übergeben.

- **getData():**

  Die Methode *getData* liefert einen Boolean-Wert der angibt ob die Ausführung der Methode *getPoint* (Ermittlung des nächsten Punkts) erfolgreich war.

- **setgetData(bool getData):**

Die Methode *setgetData* setzt Boolean-Attribut *getData*, das angibt ob das Lesen des Punkts aus der Datenbasis erfolgreich war (siehe *getData*).
• **writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char *table_farest)** Die Methode *writeClusterNearestandFarestToDatabase* schreibt die ermittelten Hilfsinformationen zu den nähesten und entferntesten Punkten der übergebenen Cluster in die übergebenen Datenbanktabellen. Die Datenbanktabellen werden dabei neu angelegt.
• **writeDatabaseInformation(char *table_database)** Die Methode *writeDatabaseInformation* schreibt die ermittelten Hilfsinformationen zur Datenbasis in die übergebene Datenbanktabelle. Die Datenbanktabelle wird dabei neu angelegt.
• **writeClusterInformation(Cluster *clusters, char *table_cluster)** Die Methode *writeClusterInformation* schreibt die ermittelten Hilfsinformationen zu den gefundenen Clustern in die übergebene Datenbanktabelle. Die Datenbanktabelle wird neu angelegt.
• **setcomputeHelpData(bool computeHelpData)**: Setzt Boolean-Attribut der angibt ob Hilfsinformationen zur Datenbasis berechnet werden sollen oder nicht.
• **computeQuantilClasses()**: Ermittlung von Klassen die zur näherungsweisen Berechnung der Quantile benötigt werden (siehe Kap. 6.2.2)
• **orderPointsToQuantilClass(Point *point):** Die Punkte der Datenbasis werden zur Berechnung der Quantile ihrer jeweiligen Klasse zugeordnet (siehe Kap. 6.2.2)
• **findQuantilsWithQuantilClasses()**: Methode ermittelt ungefähre Quantile (25%, Median, 75%) für Datenbasis

Tabelle 4: K-Means: Methoden der abstrakten Klasse Database

GetData

Die Klasse *GetData* erbt von der abstrakten Klasse Database und implementiert die oben angeführten Methoden dieser Klasse. Diese Klasse realisiert die tatsächliche Datenbankanbindung des K-Means Algorithmus und die Ermittlung der Hilfsinformationen zur Datenbasis. Nachfolgend werden die im UML-Diagramm angeführten Attribute der Klasse *GetData* zur Speicherung der Hilfsinformationen beschrieben:

• **_sum:** Summe über die Werte der Dimensionen aller Punkte
• **_sumSquared:** Quadratsumme über die Werte der Dimensionen aller Punkte
• **_mean:** Mittelwert über die Werte der Dimensionen aller Punkte
• **_stdDev:** Standardabweichung pro Dimension aller Punkte
• **_minimum:** Minimum pro Dimension aller Punkte
• **_maximum:** Maximum pro Dimension aller Punkte
• **_median:** Median pro Dimension aller Punkte
• **_quantil25:** 25%-Quantil pro Dimension aller Punkte
• **_quantil75:** 75%-Quantil pro Dimension aller Punkte
• **_numPoints:** Anzahl der Punkte

Tabelle 5: K-Means: Attribute zur Speicherung der Hilfsinformationen der Klasse GetData

Zusätzlich werden auch die weiteren Attribute, die ausschließlich intern verwendet werden, erläutert:

• **_data:** Ausgelesene Werte der Dimensionen des Punkts aus Datenbank
• **_zaehler_dimension:** Gibt an welche Tabellenspalte welcher Dimension entspricht
• **_quantile:** Hilfsvektor zur Berechnung des Median
• **_quantile25:** Hilfsvektor zur Berechnung des 25%-Quantil

• **_quantile75:** Hilfsvektor zur Berechung des 75%-Quantil
• **_clusterid:** Dimension die Cluster-Id entspricht
• **_dimensions:** Anzahl der Dimensionen
• **_getData:** Gibt an ob Lesen aus Datenbank erfolgreich war
• **_computeHelpData:** Gibt an ob Hilfsinformationen zur Datenbank ermittelt werden sollen
• **_point:** Punkt der aus Datenbank gelesen wurde
• **_table:** Tabellenname der Datenbasis
• **_columns:** Spalte der Datenbasis die für Algorithmus verwendet werden
• **_file_log:** Log-File für Protokollierung
• **_countiteration:** Anzahl der Iterationen
• **_countnumPoints:** Anzahl der Punkte die geclustert werden sollen

Tabelle 6: K-Means: Restliche Attribute der Klasse GetData

Die weiteren wichtigen Methoden der Klasse *GetData* werden nachfolgend beschrieben. Die Methoden zur Kapselung der Attribute werden hier nicht näher erläutert. Diese sind im Anhang angeführt (siehe Kap. 12.1)

• **GetData(int dimensions, int clusterid, char *table, char *columns, FILE *file_log, int countnumpoints):** Konstruktor der Klasse *GetData,* der die nötigen Attribute der Klasse *GetData* initialisiert.
• **~GetData():** Destruktor der Klasse *GetData*
• **getPoint2():** Die Methode *getPoint2* liefert den nächsten Punkt der Datenbasis ab der 2 Iteration. Dies ist nötig da der Datenbank-Cursor bei wiederholter Anwendung

in der Methode *getPoint* nicht die gleichen Punkte geliefert hat. Daher liefert die Methode *getPoint2* alle Punkte deren Cluster-Id nicht initial ist.
• **orderPointsToMedianQuantilClass(Point *point):** Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des Median zugeordnet (siehe Kap. 6.2.2)
• **orderPointsTo25QuantilQuantilClass(Point *point):** Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des 25%-Quantils zugeordnet (siehe Kap. 6.2.2)
• **orderPointsTo75QuantilQuantilClass(Point *point):** Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des 75%-Quantils zugeordnet (siehe Kap. 6.2.2)
• **findMedianWithQuantilClasses():** Näherungsweise Ermittlung des Median über die Klassen (siehe Kap. 6.2.2)
• **findQuantil25WithQuantilClasses():** Näherungsweise Ermittlung des 25%-Quantils über die Klassen (siehe Kap. 6.2.2)
• **findQuantil75WithQuantilClasses():** Näherungsweise Ermittlung des 75%-Quantils über die Klassen (siehe Kap. 6.2.2)

Tabelle 7: K-Means: Methoden der Klasse GetData

Point

Die Klasse Point repräsentiert einen Punkt, der aus der Datenbank gelesen wird. Die Klasse enthält folgende Attribute:

• **_vector:** Werte der Dimensionen des Punkts
• **_rowid:** Rowid des Punkts
• **_clusterid:** Cluster-Id des Punkts
• **_clusterDistance:** Distanz des Punkts zu dem Centroiden seines Clusters
• **_dimensions:** Anzahl der Dimensionen des Punkts

Tabelle 8: K-Means: Attribute der Klasse Point

Nachfolgend werden die Methoden der Klasse Point beschrieben. Die weiteren Methoden zur Kapselung der Attribute werden hier nicht näher erläutert.

• **Point(int dimension):** Konstruktor der Klasse *Point* zur Initialisierung
• **Point(int dimension, double *vector, char *rowid, int clusterid):** Zweiter Konstruktor der Klasse *Point* zur Initialisierung
• **~Point():** Destruktor der Klasse *Point*

Tabelle 9: K-Means: Methoden der Klasse Point

Cluster

Die Klasse *Cluster* repräsentiert einen ermittelten Cluster des K-Means Algorithmus. Darin werden auch die ermittelten Hilfsinformationen pro Cluster gespeichert. Die Attribute zur Speicherung der Hilfsinformationen sehen wie folgt aus:

• **_sum:** Summe der Werte der Punkte des Clusters pro Dimension
• **_sumSquared:** Quadratsumme der Werte der Punkte des Clusters pro Dimension
• **_mean:** Mittelpunkt des Clusters (Centroid)
• **_stdDev:** Standardabweichung der Punkte des Clusters pro Dimension
• **_minimum:** Minimum pro Dimension
• **_maximum:** Maximum pro Dimension
• **_sqrtNumPoints:** Wurzel über die Anzahl der Punkte des Clusters
• **_numPoints:** Anzahl der Punkte des Clusters
• **_vector_farest:** Vektor mit entferntesten Punkten des Clusters (sortiert nach Distanz)
• **_vector_nearest:** Vektor mit nähesten Punkten des Clusters (sortiert nach Distanz)

Tabelle 10: K-Means: Attribute zur Speicherung der Hilfsinformationen der Klasse Cluster

Zusätzlich werden auch die weiteren Attribute, die ausschließlich intern verwendet werden, erläutert:

• **_nearestNumPoints:** Anzahl der nähesten Punkte, die gespeichert werden sollen
• **_farestNumPoints:** Anzahl der entferntesten Punkte die gespeichert werden sollen
• **_id:** Cluster-Id

• **_dimensions:** Anzahl der Dimensionen pro Punkt
• **_usedcolumns:** Gibt an welche Spalten für Clustering relevant sind (nur numerische Spalten)
• **_file_log:** Log-File für Protokollierung
• **_next, _previous:** Zeiger auf nächsten bzw. vorherigen Cluster

Tabelle 11: K-Means: Restliche Attribute der Klasse Cluster

Nachfolgend werden die Methoden der Klasse *Cluster* beschrieben. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert. Diese sind im Anhang (siehe Kap. 12.1) angeführt.

• **Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, bool *usedcolumns, FILE *file_log):** Konstruktor der Klasse *Cluster* zur Initialisierung der Attribute
• **~Cluster():** Destruktor der Klasse *Cluster*
• **clear():** Initialisiert Vektoren für näheste und entfernteste Punkte, und die Vektoren für die Berechnung des Minimum und des Maximum
• **addPoint(Point *point):** Die Methode *addPoint* fügt dem Cluster einen Punkt hinzu und berechnet dabei auch die Hilfsinformationen wie Summe, Quadratsumme und Anzahl der Punkte
• **deletePoint(Point *point):** Die Methode *deletePoint* entfernt einen Punkt aus dem Cluster und korrigiert die Hilfsinformationen wie Summe, Quadratsumme und Anzahl der Punkte
• **updateClusterMean():**

Mittelpunkt des Clusters wird ermittelt (Centroid)
• **distanceSquared(Point *point):** Liefert die Quadratische Distanz zwischen Cluster-Mittelpunkt (Centroid) und dem übergebenen Punkt
• **distanceEuklid(Point *point):** Liefert die Euklidische Distanz zwischen Cluster-Mittelpunkt (Centroid) und dem übergebenen Punkt
• **distanceManhatten(Point *point):** Liefert die Manhatten Distanz zwischen Cluster-Mittelpunkt (Centroid) und dem übergebenen Punkt
• **computeStdDev():** Standardabweichung der Punkte wird pro Dimension ermittelt. Zusätzlich wird die Wurzel der Anzahl der Punkte berechnet.
• **addNearestPoint(Point *point):** Der übergebene Punkt wird dem sortierten Vektor mit den nähesten Punkten hinzugefügt, wenn der Punkt näher ist als der bisher am weitesten entfernte näheste Punkt des Vektors (siehe Kap. 6.2.2).
• **addFarestPoint(Point *point):** Der übergebene Punkt wird dem sortierten Vektor mit den entferntesten Punkten hinzugefügt, wenn der Punkt entfernter ist als der bisher am nähesten entfernte Punkt des Vektors (siehe Kap. 6.2.2).
• **findIndex(double tofind, vector<Information_Point> liste):** Falls ein Punkt in einem der Vektoren für die nähesten oder entferntesten Punkte eingefügt werden soll, ermittelt diese Methode über binäres Suchen den Index an welcher Stelle des Vektors eingefügt wird, damit die Sortierung erhalten bleibt (siehe Kap. 6.2.2)

Tabelle 12: K-Means: Methoden der Klasse Cluster

Quantil_Class

Hilfsklasse zur näherungsweisen Berechnung des 25%-Quantil, Median und des 75%-Quantil. Eine genaue Erläuterung der Ermittlung der Hilfsinformationen ist in Kapitel 6.2.2 zu finden. Nachfolgend werden die Attribute der Hilfsklasse dargestellt:

• **_minimum:** Untere Grenze der Klasse
• **_maximum:** Obere Grenze der Klasse
• **_numPoints:** Anzahl der Punkte in Klasse

Tabelle 13: K-Means: Attribute der Klasse Quantil_Class

Methode der Klasse *Quantil_Class*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **Quantil_Class():** Konstruktor der Klasse
• **Quantil_Class(double minimum, double maximum):** Zweiter Konstruktor der Klasse
• **~Quantil_Class():** Destruktor der Klasse
• **addNumPoints():** Anzahl der Punkte in Klasse wird um 1 erhöht

Tabelle 14: K-Means: Methoden der Klasse Quantil_Class

Information_Point

Hilfsklasse für die Vektoren der nähesten und entferntesten Punkte (siehe Kap. 6.2.2). Attribute der Klasse *Informatione_Point*:

• **_rowid:** Rowid des Punkts
• **_distance:** Distanz des Punkts zu seinem Centroide

Tabelle 15: K-Means: Attribute der Klasse Information_Point

Methoden der Klasse *Informatione_Point*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **Information_Point(char *rowid, double distance):** Konstruktor der Klasse
• **~Informatione_Point():** Destruktor der Klasse

Tabelle 16: K-Means: Methoden der Klasse Information_Point

KMeans

Die Klasse *KMeans* ist sozusagen die „Hauptklasse“ und führt auch den K-Means Algorithmus aus. Eine genaue Beschreibung der Implementierung des Algorithmus kann in Kapitel 6.2.1 gefunden werden. Nachfolgend werden die Attribute der Klasse *KMeans* näher erläutert:

• **_database:** Zugrunde liegende Datenbasis des Algorithmus
• **_numClusters:** Anzahl der zu ermittelten Cluster (Parameter k)
• **_clusters:** Cluster die der Algorithmus erzeugt
• **_dimensions:** Anzahl der Dimensionen pro Punkt
• **_numIterations:** Anzahl der Iterationen
• **_file_log:** Log-File für Protokollierung
• **_distancefunction:** Steuerung welche Distanzfunktion verwendet wird (1 = Quadratische Distanz, 2 = Euklidische Distanz, 3 = Manhatten – Distanz)
• **_initialPoints:** Anzahl der Punkte pro Cluster nach der Initialisierung
• **_numberinitialiterations:** Maximale Anzahl der Iterationen bei Initialisierung
• **_numberiterations:** Maximale Anzahl der Iterationen des K-Means Algorithmus

Tabelle 17: K-Means: Attribute der Klasse KMeans

Methoden der Klasse *KMeans*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **KMeans(Database *datenbank, int anz_cluster, int dimension, int nearestnumpoints, int farestnumpoints, int distancefunction, bool *usedcolumns, FILE *file_log, int *initialpoints, int numberinitialiterations, int numberiterations):** Konstruktor der Klasse K-Means
• **~KMeans():**

Destruktor der Klasse K-Means
• **setup(int nearestnumpoints, int farestnumpoints, bool *usedcolumns):** Erzeugen der nötigen *k* Cluster für den K-Means Algorithmus
• **initModel(long initnumpointsmax, long initnumpointsmin):** Jedem der erzeugten Cluster wird per Zufallszahl ein Punkt als Centroid zugeordnet.
• **initModelwithClustering(long initnumpointsmax, long initnumpointsmin):** Mit den zufällig ermittelten Centroiden wird ein Clustering auf eine kleine Stichprobe (Summe der Zufallszahlen von *initModel*) der Daten ausgeführt. Die dadurch ermittelten Centroide sind die Centroide zu Beginn des K-Means Algorithmus
• **cluster(long numpoints,long initnumpointsmax, long initnumpointsmin):** Die Methode *cluster* veranlasst die Initialisierung und führt den K-Means Algorithmus aus (siehe Kap. 6.2.1).
• **closestCluster(Point *point):** Ermittlung des nähesten Cluster zum übergebenen Punkt und Berechnung des Minimum und des Maximum pro Cluster
• **getCluster(int id):** Liefert das Cluster zur übergebenen Cluster-Id
• **writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char *table_farest)** Die Methode *writeClusterNearestandFarestToDatabase* schreibt die ermittelten Hilfsinformationen zu den nähesten und entferntesten Punkte der übergebenen Cluster in die übergebenen Datenbanktabellen.
• **writeDatabaseInformation(char *table_database)** Die Methode *writeDatabaseInformation* schreibt die ermittelten Hilfsinformationen zur Datenbasis in die übergebene Datenbanktabelle.

• **writeClusterInformation(Cluster *clusters, char *table_cluster)** Die Methode *writeClusterInformation* schreibt die ermittelten Hilfsinformationen zu den gefundenen Clustern in die übergebene Datenbanktabelle.

Tabelle 18: K-Means: Methoden der Klasse KMeans

6.2.1 K-Means Algorithmus

Der implementierte K-Means Algorithmus entspricht, bis auf die Ermittlung der Hilfsinformationen, zu einem Großteil dem „Standard“ K-Means Algorithmus (siehe Kap. 2.2.2.1). Lediglich die Initialisierung und die Abbruchbedingung wurden etwas modifiziert. Daher werden anschließend die geänderte Initialisierung und die geänderte Abbruchbedingung näher erklärt. Abschließend wird dann kurz die Implementierung des tatsächlichen K-Means Algorithmus dargestellt.

Initialisierung

Die „Standard“-Initialisierung des K-Means Algorithmus entspricht einer zufälligen Auswahl der Centroide zu Beginn des Clustering. Kapitel 4.1.1 zeigt bereits die Nachteile dieser Initialisierung auf. Basierend auf der vorhandenen Optimierungsmöglichkeit der Initialisierung (siehe Kap. 4.1.1.1) von Ester und Sander [ES00] wurde hier die Initialisierung gestaltet. Folgende Vorgehensweise wird bei der implementierten Initialisierung verwendet:

1. Zufällige Auswahl von Centroiden für jeden Cluster
2. Clustering einer kleinen Stichprobe mit den zufällig ausgewählten Centroiden
3. Die Centroide des Clusteringergebnis der Stichprobe werden als initiale Centroide für das Clustering herangezogen

Im Gegensatz zum Ansatz von Ester und Sander [ES00] wird hier nur eine Stichprobe und nicht mehrere gezogen werden. Allerdings zeigen die Testergebnisse (siehe Kap. 7.2), dass der angeführte Ansatz der Initialisierung einen Großteil der Schwächen der „Standard“-Initialisierung beseitigt.

Abbruchbedingung

Die „Standard“-Abbruchbedingung des K-Means Algorithmus (siehe Kap. 2.2.2.1) beendet den Algorithmus dann, wenn sich die Clusterzuordnung keines Punkts innerhalb einer Iteration mehr ändert. Diese Abbruchbedingung wurde so weit modifiziert, dass der Algorithmus beendet wird, sobald sich die Clusterzuordnung von weniger als einem Promil der Punkte der verwendeten Datenmenge innerhalb einer Iteration geändert hat. Wie die Testergebnisse zeigen (siehe Kap. 7.2.3) wirkt sich dies nicht negativ auf das Ergebnis des Clustering aus.

Nachfolgend wird die Implementierung des Algorithmus kurz erläutert. Die Ermittlung der Hilfsinformationen wird dabei aber noch nicht berücksichtigt (siehe Kap. 6.2.2).

```
/**
 * Methode cluster führt Kmeans-Algorithmus aus
 * @param numpoints Anzahl der Punkte die für den Algorithmus herangezogen
 *                  werden (optional)
 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
 */
void KMeans::cluster(long numpoints, long initnumpointsmax, long
initnumpointsmin){
        Point *point;
        Cluster *cluster, *oldcluster;
        long count, changecount;
        bool changeCluster;
        long promilegrenze;

        if (numpoints)
                promilegrenze = numpoints / 1000;

        // Initialisierung durchführen
        initModelwithClustering(initnumpointsmax, initnumpointsmin);

        _numIterations = 0;        // Variable mit Anzahl der Iterationen initialisieren

        do{
                changeCluster = false;
                changecount = 0;
                _numIterations++;

                _database->reset();                    // Datenquelle zurücksetzen

                count = 0;        // Zählvariable (Anzahl der Punkte)
                // Solange noch ein Punkt vorhanden ist und die Anzahl der Punkte
                // unter der max. Anzahl der Punkte liegt wird Schleife durchgeführt
                point = _database->getPoint();
                while((_database->getData()) && ((count++ < numpoints) ||
                        (numpoints == 0))){
```

```
                    // Liefert "nähestes" Cluster zu Punkt point
                    cluster = closestCluster(point);

                    if (!cluster){
                        fprintf(_file_log,"No Cluster found. \n");
                    }else{
                        if (cluster->getId() != point->getClusterId()){
                            changeCluster = true;
                            changecount++;

                            if (point->getClusterId() > 0){
                                oldcluster = getCluster(point->getClusterId());
                                oldcluster->deletePoint(point);
                                oldcluster->updateClusterMean();
                            }
                            // Fügt Punkt zu nähestem Cluster hinzu
                            cluster->addPoint(point);

                            _database->updateClusterId(
                                        point->getRowID(),cluster->getId());

                            // Berechnet Mittelpunkt des Clusters neu
                            cluster->updateClusterMean();
                        }
                    }
                    if (count < numpoints)
                        point = _database->getPoint();
                }

                // Wenn Anzahl der Änderungen unter Promilegrenze, dann ist
                // Clustering fertig
                if (changecount <= promilegrenze)
                    changeCluster = false;

                if (_numIterations >= _numberiterations){
                    changeCluster = false;
                    fprintf(_file_log, "Manuelles Beenden des Clustering. \n");
                }
            }while(changeCluster);
        }
```

6.2.2 Ermittlung der Hilfsinformationen

Bisher wurden die verwendeten Klassen und der K-Means Algorithmus näher erläutert. Anschließend wird detailliert auf die Implementierung zur Ermittlung der Hilfsinformationen eingegangen. Dabei wird unterschieden zwischen den Hilfsinformationen zur Datenbasis (siehe Kap. 5.1) und den Hilfsinformationen zu den gefundenen Clustern (siehe Kap. 5.2).

6.2.2.1 Hilfsinformationen zur Datenbasis

Die Hilfsinformationen zur Datenbasis werden mit Ausnahme der Quantile in der ersten Iteration des K-Means Algorithmus ermittelt. Dies wird über das Attribut *_computeHelpData* der Klasse *GetData* gesteuert. Die Ermittlung der Hilfsinformationen erfolgt nur, wenn dieses Attribut *true* ist. Nach der ersten Iteration wird das Attribut daher auf *false* gesetzt. Das Auslesen der Punkte aus der Datenbank erfolgt in der Methode *getPoint* der Klasse *getData*. Daher wird auch in dieser Methode die Berechnung der Hilfsinformationen für die Datenbasis durchgeführt. Lediglich die näherungsweise Berechnung der Quantile erfolgt nicht in der Methode *getPoint*. Nachfolgend wird der Quellcode der Methode *getPoint* dargestellt:

```
/**
 * Methode getPoint liefert nächsten Punkt der Datenbasis
 * @return Liefert nächsten Punkt der Datenbasis
 */
Point *GetData::getPoint(void){
        int i=0,c=0,j=0, clusterid=0;
        char *rowid;

        if (_countiteration > 1){
                return getPoint2();
        }else{
                EXEC SQL WHENEVER SQLWARNING CONTINUE;
                EXEC SQL WHENEVER NOTFOUND CONTINUE;
                EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

                EXEC SQL FETCH cursor USING DESCRIPTOR select_des;

                _numPoints++;
                for (i=0; i < select_des->F; i++){
                        if (select_des->T[i] == 3){
                                _data[j] = (int) *select_des->V[i];                //INT
                                if (_computeHelpData){
                                        _sum[j]+= (int) *select_des->V[i];
                                        _sumSquared[j]+= _data[j] * _data[j];
                                        _zaehler_dimension[j] = i;

                                        // Minimum bestimmen
                                        if (_numPoints == 0)
                                                _minimum[j] = _data[j];
                                        else if (_data[j] < _minimum[j])
                                                _minimum[j] = _data[j];

                                        // Maximum bestimmen
                                        if (_numPoints == 0)
                                                _maximum[j] = _data[j];
                                        else if (_data[j] > _maximum[j])
                                                _maximum[j] = _data[j];
```

```
                        }

                        j++;
                    }
                    if (select_des->T[i] == 4){
                        _data[j] = *(float *)select_des->V[i];              //FLOAT
                        if (_computeHelpData){
                            _sum[j]+= *(float *)select_des->V[i];
                            _sumSquared[j]+= _data[j] * _data[j];
                            _zaehler_dimension[j] = i;

                            // Minimum bestimmen
                            if (_numPoints == 0)
                                _minimum[j] = _data[j];
                            else if (_data[j] < _minimum[j])
                                _minimum[j] = _data[j];

                            // Maximum bestimmen
                            if (_numPoints == 0)
                                _maximum[j] = _data[j];
                            else if (_data[j] > _maximum[j])
                                _maximum[j] = _data[j];
                        }
                        j++;
                    }
                    if (i == 0)
                        rowid = (char *)select_des->V[i];                   //ROWID
                    if (i == _clusterid)
                        clusterid = (int) *select_des->V[i];                //Cluster ID
                }
                if (_computeHelpData)
                    computeStdDev();                            // Standardabweichung

                _point->set(_data, rowid, clusterid);
                return _point;
        }
sqlerror:
        if (sqlca.sqlcode == -1002) {
            if (_numPoints < _countnumPoints){
                fprintf(_file_log,"Fetch-Fehler, obwohl noch Punkte vorhanden!
                                \n");
                fflush(_file_log);
            }else
                setgetData(false);          //Fetch fertig, kein Datensatz mehr
            return 0;
        } else {
            fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
            fflush(_file_log);
            exit(1);
        }
}
```

Die Stellen des Quellcode, die die Ermittlung der Hilfsinformationen veranlassen, sind blau markiert. Die arithmetischen Werte Summe, Quadratsumme, Minimum, Maximum und Anzahl der Punkte werden direkt in dieser Methode ermittelt. Die Standardabweichung und der Mittelwert werden durch den Aufruf der Methode *computeStdDev* berechnet. Diese wird erst beim letzten Aufruf der Methode *getPoint* in der ersten Iteration aufgerufen.

```
/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des
 * Clusters
 */
void GetData::computeStdDev(void){
        double *sum = _sum, *sumSquared = _sumSquared, *stdDev = _stdDev,
                *mean = _mean, h_stddev;

        for(int i = 0; i < _dimensions; i++){
                h_stddev = *sum++ / _numPoints;
                *stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints –
                                h_stddev * h_stddev));
                *mean++ = h_stddev;
    }
}
```

Berechnung der Quantile

Bei einer exakten Berechnung der Quantile (25%-Quantil, Median, 75%-Quantil) ist es nötig, dass die Werte jeder Dimension sortiert vorliegen. Nur dann können die Quantile genau ermittelt werden. Dies ist beim K-Means Algorithmus allerdings nicht der Fall, und vor allem bei mehreren Dimensionen nicht sinnvoll. Um dies zu realisieren, müsste pro Dimension eine Iteration über die Datenbank gemacht werden, damit die Werte jeweils nach einer Dimension sortiert werden können. Dies ist aber mit einem vertretbaren Aufwand nicht möglich. Daher werden die Quantile bei diesem Algorithmus näherungsweise bestimmt. Dabei wird wie folgt vorgegangen:

- In der ersten Iteration wird das Minimum und das Maximum pro Dimension ermittelt (ist bereits eine Hilfsinformation). Der Bereich zwischen Minimum und Maximum wird in 100 gleich große Klassen geteilt.

- In der nächsten Iteration wird gezählt, wie viele Punkte in jede Klasse fallen. Summiert man die Anzahl der Punkte in den Klassen auf erhält man Quantile. Diese werden zwar beliebig sein, z.B. kann Klasse 20 das 47 % Quantil darstellen und Klasse 21 das 51 % Quantil. Dadurch weiß man, dass sich der Median zwischen den Werten von Klasse 21 befinden muss. In jeder Folge-

Iteration kann die entsprechende Klasse weiter verfeinert werden. Je mehr Iterationen erfolgen, desto genauer wird die Annäherung der Quantile.

- Der Mittelwert der Klasse in der sich nach der letzten Iteration zum Beispiel der Median befindet wird als näherungsweiser Median herangezogen.

- Für jedes Quantil, d.h. 25%-Quantil, 75%-Quantil und Median, werden eigens 100 Klassen in Form eines Vektors angelegt und wie bereits beschrieben verfeinert. Dadurch wird für jedes Quantil ein Näherungswert erzielt.

Die Methoden zur Ermittlung der Quantile gehören zur Klasse *GetData*. Die Methode *computeQuantilClasses* ermittelt zu Beginn der zweiten Iteration, d.h. nach der ersten Iteration, die 100 Klassen für jedes Quantil. Ab der zweiten Iteration des K-Means wird über die Methode *orderPointsToQuantilClass* ein Punkt zu seiner jeweiligen Klasse pro Quantil addiert. Am Ende einer jeden Iteration, außer der ersten, ermittelt die Methode *findQuantilsWithQuantilClasses* die näherungsweise ermittelten Quantile. Anschließend werden die Erweiterungen zur Ermittlung der Hilfsinformationen der Datenbasis während des K-Means Algorithmus mit Hilfe von Auszügen des Quellcodes der Methode *cluster* der Klasse *Kmeans* gezeigt (siehe Kap. 6.2.1). Die Ergänzungen des Quellcodes zur Ermittlung der Hilfsinformationen der Datenbasis sind blau markiert. Der Quellcode der einzelnen Methoden befindet sich im Anhang (siehe Kap. 12.1):

```
……..
        _numIterations = 0;        // Variable mit Anzahl der Iterationen initialisieren
        // In erstem Schleifendurchlauf sollen Hilfsinformationen für Datenbasis
        // berechnet werden
        _database->setcomputeHelpData(true);

        do{
                changeCluster = false;
                changecount = 0;
                _numIterations++;

                _database->reset();                    // Datenquelle zurücksetzen

                // Für die Berechnung der Quantile werden die einzelnen Dimensionen
                // pro Quantil in 100 Klassen geteilt
                if (_numIterations == 2)
                        _database->computeQuantilClasses();

                // Vektoren mit Hilfsinformationen werden initialisiert
                for(cluster = _clusters; cluster; cluster = cluster->getNext()){
                        cluster->clear();
                }
```

```
        count = 0;        //Zählvariable (Anzahl der Punkte)
        // Solange noch ein Punkt vorhanden ist und die Anzahl der Punkte
        // unter der max. Anzahl der Punkte liegt wird Schleife durchgeführt
        point = _database->getPoint();
        while((_database->getData()) && ((count++ < numpoints) ||
                (numpoints == 0))){
                // Liefert "nähestes" Cluster zu Punkt point
                cluster = closestCluster(point);

                if (!cluster){
                        fprintf(_file_log,"No Cluster found. \n");
                        fflush(_file_log);
                }else{
                        if (cluster->getId() != point->getClusterId()){
                                changeCluster = true;
                                changecount++;

                                if (point->getClusterId() > 0){
                                        oldcluster = getCluster(point->getClusterId());
                                        oldcluster->deletePoint(point);
                                        oldcluster->updateClusterMean();
                                }
                                // Fügt Punkt zu nähestem Cluster hinzu
                                cluster->addPoint(point);
                                _database->updateClusterId(point-
                                                >getRowID(),cluster->getId());

                                // Berechnet Mittelpunkt des Clusters neu
                                cluster->updateClusterMean();
                        }
                        // Hilfsinformationen werden berechnet
                        if (_numIterations > 1){
                                // Klassen zur Quantil-Berechnung füllen
                                _database->orderPointsToQuantilClass(point);
                        }
                }
                if (count < numpoints)
                        point = _database->getPoint();
        }
        // Quantile (25%, 50% und 75%) werden näherungsweise ermittelt
        if (_numIterations > 1)
                _database->findQuantilsWithQuantilClasses();

        // Nur in 1 Schleifendurchlauf HIlfsinformationen berechnen
        _database->setcomputeHelpData(false);
        // Wenn Anzahl der Änderungen unter Promilegrenze, dann ist
        // Clustering fertig
        if (changecount <= promilegrenze)
                changeCluster = false;
}while(changeCluster);

……..
```

6.2.2.2 Hilfsinformationen zu den gefundenen Clustern

Die Hilfsinformationen zu den Clustern werden während der Iterationen des K-Means Algorithmus berechnet. Je nach Hilfsinformation werden verschiedene Methoden verwendet. Nachfolgend werden Auszüge aus dem Quellcode dieser Methoden pro Hilfsinformation dargestellt.

Summe, Quadratsumme und Anzahl der Punkte

Diese Hilfsinformationen werden bei einer Änderung der Zuordnung eines Punkts zu einem Cluster ermittelt, d.h. wenn ein Punkt einem Cluster hinzugefügt wird oder wenn der Punkt vom Cluster entfernt wird. Die im Folgenden vorgestellten Methoden *addPoint* und *deletePoint* der Klasse *cluster* berechnen diese Hilfsinformationen.

```
/**
 * Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die
 * Werte des Clusters neu
 * @param point Punkt der zu Cluster hinzugefügt wird
 */
void Cluster::addPoint(Point *point){
        // Werte des Clusters werden an lokale Zeiger übergeben
        double *vector = point->_vector, *sum = _sum,
                *sumSquared  = _sumSquared;

        // Summen und Quadratsummen für jede Dimension werden berechnet
        for(int i = 0; i < _dimensions; i++){
                *sumSquared++ += *vector * *vector;
                *sum++ += *vector++;
        }
        _numPoints++;                   // Anzahl der Punkte
}

/**
 * Methode deletePoint löscht einen Punkt des Clusters und berechnet die Werte
 * des Clusters neu
 * @param point Punkt der aus Cluster gelöscht wird
 */
void Cluster::deletePoint(Point *point){
        // Werte des Clusters werden an lokale Zeiger übergeben
        double *vector = point->_vector, *sum = _sum,
                *sumSquared = _sumSquared;

        // Summen und Quadratsummen für jede Dimension werden neu berechnet
        for(int i = 0; i < _dimensions; i++){
                *sumSquared++ -= *vector * *vector;
                *sum++ -= *vector++;
        }
        _numPoints--;                   // Anzahl der Punkte
}
```

Die Methode *addPoint* fügt einen Punkt zu einem Cluster hinzu, und die Methode *deletePoint* entfernt einen Punkt von einem Cluster.

Mittelpunkt (Centroid)

Der Mittelpunkt eines Clusters wird immer dann neu berechnet, wenn entweder ein Punkt dem Cluster hinzugefügt wurde oder ein Punkt aus dem Cluster entfernt wurde. Das Aktualisieren des Mittelpunkts erfolgt mit der Methode *updateClusterMean* der Klasse *cluster*.

```
/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void Cluster::updateClusterMean(void){
        double *sum = _sum, *mean = _mean, h_mean;

        // Mittelpunkt des Clusters wird neu berechnet
        if(_numPoints)
                for(int i = 0; i < _dimensions; i++){
                        h_mean = *sum++ / _numPoints;
                        *mean++ = h_mean;
                }
}
```

Standardabweichung und Wurzel der Anzahl der Punkte

Die Standardabweichung und die Wurzel der Anzahl der Punkte wird nach der letzten Iteration des K-Means Algorithmus berechnet, in dem für jeden Cluster die Methode *computeStdDev* der Klasse *cluster* aufgerufen wird.

```
/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des
 * Clusters
 */
void Cluster::computeStdDev(){
        double *sum = _sum, *sumSquared = _sumSquared, *l_stdDev = _stdDev,
                h_stddev;

        for(int i = 0; i < _dimensions; i++){
                h_stddev = *sum++ / _numPoints;
                *l_stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints –
                                        h_stddev * h_stddev));
    }
    _sqrtNumPoints = (double) sqrt(_numPoints);
}
```

Minimum und Maximum

Das Minimum und das Maximum jeder Dimension eines Clusters werden in jeder Iteration des K-Means Algorithmus neu berechnet, d.h. die letzte Iteration bestimmt das Ergebnis des Minimum und des Maximum. Zu Beginn einer jeden Iteration werden die Werte für das Minimum und das Maximum mit der Methode *clear* der Klasse *cluster* pro Cluster auf 0 gesetzt.

```
/**
 * Methode clear initialisiert Hilfsinformationen des Clusters
 * für neue Iteration des Kmeans
 */
void Cluster::clear(void){
        for (int i=0;i < _dimensions; i++){
                _minimum[i] = 0;
                _maximum[i] = 0;
        }
        _vector_farest.clear();
        _vector_nearest.clear();
}
```

Bei der Ermittlung des nähesten Cluster zu einem eingelesenen Punkt werden zusätzlich das Minimum und das Maximum ermittelt. Dies geschieht in der Methode *closestCluster* der Klasse *KMeans*. Der Quellcode zur Ermittlung dieser Hilfsinformationen wird in der Methode nachfolgend blau markiert.

```
/**
 * Methode closestCluster ermittelt nähesten Cluster zu einem Punkt
 * @param point Punkt zu dem nähestes Cluster ermittelt wird
 * @return Nähestes Cluster zu Punkt point
 */
Cluster* KMeans::closestCluster(Point *point){
        double distance = 1e20, h_distance;
        Cluster *m, *cluster;
        bool found = false;
        double *minimum, *maximum, *vector;

        // Ermittelt nähesten Cluster zu einem Punkt
        for(m = _clusters; m; m = m->getNext()){
                switch( _distancefunction){
                        case 1:                         // Quadratische Distanz
                                h_distance = m->distanceSquared(point);
                                break;
                        case 2:                         // Euklidische Distanz
                                h_distance = m->distanceEuklid(point);
                                break;
                        case 3:                         // Manhatten Distanz
                                h_distance = m->distanceManhatten(point);
                                break;
                }
```

```
                if(h_distance < distance){
                        cluster = m;
                        distance = h_distance;
                        found = true;
                }
        }

        if (found){
                minimum = cluster->getMinimum();
                maximum = cluster->getMaximum();
                vector  = point->get();
                // Minimum und Maximum pro Cluster ermitteln
                for (int i=0;i<_dimensions;i++){
                        // Minimum
                        if (minimum[i] == 0)
                                minimum[i] = vector[i];
                        else if (vector[i] < minimum[i])
                                minimum[i] = vector[i];
                        // Maximum
                        if (maximum[i] == 0)
                                maximum[i] = vector[i];
                        else if (vector[i] > maximum[i])
                                maximum[i] = vector[i];
                }

                point->setClusterDistance(distance);
                return cluster;
        } else
                return 0;
}
```

Nähesten und entferntesten Punkte pro Cluster

Ebenso wie der Ermittlung des Minimum und des Maximum werden die nähesten und entferntesten Punkte pro Cluster in jeder Iteration, außer der ersten, neu ermittelt. In der ersten Iteration ist eine Ermittlung dieser Hilfsinformation noch nicht nötig, da der Algorithmus mindestens zwei Iterationen benötigt. Die Initialisierung der verwendeten Vektoren für diese Punkte erfolgt am Beginn jeder Iteration mit der Methode *clear* der Klasse *Cluster* (siehe oben). Dies ist nötig, da sich nur in der letzten Iteration (fast) keine Zuordnung eines Punkts zu seinem Clustern mehr ändert. Die Ermittlung der nähesten und entferntesten Punkte erfolgt über die Distanz des Punkts zum Centroiden des Clusters. (siehe Kap. 5.2). Die Methode *addNearestPoint* kontrolliert ob ein Punkt zu den nähesten gehört, und wenn ja, dann wird dieser dem Vektor hinzugefügt. Gleiches gilt für die Methode *addFarestPoint* der Klasse *Cluster*, nur mit den entferntesten Punkten. Nachfolgend werden diese Methoden dargestellt.

```
/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des
```

```
 * Clusters gehört. Die _farestNumPoints werden in einer eigenen Liste
 * gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten
 *                  Punkten gehört
 */
void Cluster::addFarestPoint(Point *point){
        Information_Point info_point(point->getRowID(),
                                        point->getClusterDistance());
        if (_vector_farest.empty())
                _vector_farest.push_back(info_point);
        else
                if (_vector_farest.size() < _farestNumPoints)
                        _vector_farest.insert(_vector_farest.begin() +
                           findIndex(info_point.getDistance(), _vector_farest), info_point);
                else
                        if (_vector_farest.front().getDistance() <
                                                        info_point.getDistance()){
                                _vector_farest.erase(_vector_farest.begin());
                                _vector_farest.insert(_vector_farest.begin() +
                                        findIndex(info_point.getDistance(), _vector_farest),
                                                        info_point);
                        }
}

/**
 * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des
 * Clusters gehört
 * Die _nearestNumPoints werden in einer eigenen Liste gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten
 *                  gehört
 */
void Cluster::addNearestPoint(Point *point){
        Information_Point info_point(point->getRowID(), point-
                                                >getClusterDistance());

        if (_vector_nearest.empty())
                _vector_nearest.push_back(info_point);
        else
                if (_vector_nearest.size() < _nearestNumPoints)
                        _vector_nearest.insert(_vector_nearest.begin() +
                         findIndex(info_point.getDistance(), _vector_nearest), info_point);
                else
                        if (_vector_nearest.back().getDistance() >
                                                                info_point.getDistance()){
                                _vector_nearest.pop_back();
                                _vector_nearest.insert(_vector_nearest.begin() +
                                        findIndex(info_point.getDistance(), _vector_nearest),
                                                        info_point);
                        }
}
```

Die beiden angeführten Methoden benutzen die Methode *findIndex* zur Ermittlung an welcher Stelle des sortierten Vektors ein Punkt eingefügt werden soll, sofern dies nötig ist. Die Methode *findIndex* arbeitet mit binärem Suchen. Der Quellcode hierzu ist im Anhang angeführt (siehe Kap. 12.1).

Bisher wurden die einzelnen Methoden zur Ermittlung der Hilfsinformationen pro Cluster vorgestellt. Nachfolgend wird der Aufruf der Methoden im K-Means Algorithmus gezeigt. Die Methoden zur Ermittlung dieser Hilfsinformation sind blau markiert.

```
……..
do{
        changeCluster = false;
        changecount = 0;
        _numIterations++;
        …….
        _database->reset();                    // Datenquelle zurücksetzen

        // Für die Berechnung der Quantile werden
        // die einzelnen Dimensionen in 100 Klassen geteilt
        if (_numIterations == 2)
                _database->computeQuantilClasses();

        // Listen mit Hilfsinformationen werden initialisiert
        for(cluster = _clusters; cluster; cluster = cluster->getNext()){
                cluster->clear();
        }

        count = 0;        //Zählvariable (Anzahl der Punkte)
        // Solange noch ein Punkt vorhanden ist und die Anzahl der Punkte
        // unter der max. Anzahl der Punkte liegt wird Schleife durchgeführt
        point = _database->getPoint();
        while((_database->getData()) && ((count++ < numpoints) ||
                (numpoints == 0))){
                // Liefert "nähestes" Cluster zu Punkt point
                cluster = closestCluster(point);

                if (!cluster){
                        fprintf(_file_log,"No Cluster found. \n");
                        fflush(_file_log);
                }else{
                        if (cluster->getId() != point->getClusterId()){
                                changeCluster = true;
                                changecount++;

                                if (point->getClusterId() > 0){
                                        oldcluster = getCluster(point->getClusterId());
                                        oldcluster->deletePoint(point);
                                        oldcluster->updateClusterMean();
                                }
```

```
                        // Fügt Punkt zu nähestem Cluster hinzu
                        cluster->addPoint(point);

                        _database->updateClusterId(point-
                                >getRowID(),cluster->getId());

                        // Berechnet Mittelpunkt des Clusters neu
                        cluster->updateClusterMean();
                    }
                    // Hilfsinformationen pro Cluster werden berechnet
                    if (_numIterations > 1){
                        cluster->addNearestPoint(point);
                        cluster->addFarestPoint(point);
                    }
                }
                if (count < numpoints)
                    point = _database->getPoint();
            }
            .....
            // Wenn Anzahl der Änderungen unter Promilegrenze, dann ist
            // Clustering fertig
            if (changecount <= promilegrenze)
                changeCluster = false;

            if (_numIterations >= _numberiterations){
                changeCluster = false;
                fprintf(_file_log, "Manuelles Beenden des Clustering. \n");
            }
            …
        }while(changeCluster);

        _database->commitDatabase();
        fprintf(_file_log, "Clustering fertig. \n");
        fflush(_file_log);
        // Berechnung der Standardabweichung für jedes Cluster
        for(cluster = _clusters; cluster; cluster = cluster->getNext()){
            cluster->computeStdDev();
        }
```

Der gesamte Quellcode des K-Means Algorithmus befindet sich im Anhang (siehe Kap. 12.1).

6.2.3 Parameter des K-Means Algorithmus

Beim Aufruf des beschriebenen K-Means Algorithmus zur Ermittlung der Hilfsinformationen müssen bestimmte Parameter, wie die Anzahl der Cluster, die Anzahl der nähesten Punkte usw., befüllt werden. Nachfolgend werden die Parameter des K-Means Algorithmus angeführt. Als zusätzliche Information werden auch der Typ und eine mögliche Musseingabe pro Parameter angegeben:

Parameter	Typ	Muss	Beschreibung	Beispiel
ÚSER	STRING	X	User für Datenbank	stoett
PASSWORD	STRING	X	Passwort für Datenbank	klaus
SERVICE	STRING	X	Datenbankservice	dke3
TABLE	STRING	X	Tabelle mit zu clusternden Daten	clusteringtestdaten
COLUMNS	STRING	X	Dimensionen der Tabelle	"t.X2, t.X3, t.K1, t. CLUSTER_ID"
NUMPOINTS	INTEGER	X	Anzahl der zu clusternden Punkte	100000
NUMCLUSTERS	INTEGER	X	Anzahl Cluster (Parameter K)	6
MAXINITIALNUMPOINTS	INTEGER	X	Max. Anzahl initialer Punkte bei Initialisierung pro Cluster	2000
MININITIALNUMPOINTS	INTEGER	X	Min. Anzahl initialer Punkte bei Initialisierung pro Cluster	1000
CLUSTERID	INTEGER	X	Position der Cluster ID in Parameter COLUMNS	4
NAMECLUSTERID	STRING	X	Name der Cluster ID	CLUSTER_ID
DISTANCEFUNCTION	INTEGER	X	Distanzfunktion (1 = Quadrat, 2 = Euklid, 3 = Manhatten)	2
NEARESTNUMPOINTS	INTEGER	X	Anzahl der nähesten Punkte pro Cluster	2000
FARESTNUMPOINTS	INTEGER	X	Anzahl der entferntesten Punkte pro Cluster	2000
FILEKMEANS	STRING	X	File in das Ergebnis geschrieben wird	kmeans.txt
FILELOG	STRING	X	Log-File für Algorithmus	kmeans_log.txt
IDUNCLASSIFIED	STRING	X	ID für unklassifizierte Punkte	-1
TABLENEAREST	STRING	X	Tabelle mit nähesten Punkten	kmeans_ nearest_2_0_1
TABLEFAREST	STRING	X	Tabelle mit entferntesten Punkten	kmeans_ farest_2_0_1
TABLEDATABASE	STRING	X	Tabelle mit Hilfsinformationen zur Datenbasis	kmeans_ database_2_0_1
TABLECLUSTER	STRING	X	Tabelle mit Hilfsinformationen zu Clustern	kmeans_ cluster_2_0_1
NUMBER-	INTEGER	X	Maximale Anzahl der	40

INITIALITERATIONS			Iterationen bei Initialisierung	
NUMBERITERATIONS	INTEGER	X	Maximale Anzahl der Iterationen des Algorithmus	30
INITIALPOINTS	STRING	X	Initiale Punkte für Cluster Mittelpunkte	"0;0;0;0;0;0"
CATEGORICCOLUMNS	STRING		Kategorische Dimensionen	"3"

Tabelle 19: Parameter des erweiterten K-Means Algorithmus

Anmerkungen zu den Parametern:

- **COLUMNS:** Die einzelnen Dimensionen müssen alle mit **t.** beginnen
- **INITIALPOINTS:** Hier können die initialen Mittelpunkte (Centroide) der Cluster vorgegeben werden; Punkte werden durch Semikolon getrennt; Wenn der Wert 0 ist dann wird Mittelpunkt des Clusters mit Zufallsgenerator ermittelt; Es müssen Werte mitgegeben werden für jedes Cluster
- **CATEGORICCOLUMNS:** Trennzeichen ist Semikolon

6.3 Erweiterter DBSCAN Algorithmus

Der nachfolgend vorgestellte implementierte DBSCAN Algorithmus basiert auf dem DBSCAN-Algorithmus von Ester und Sander [ES00]. Zusätzlich werden lediglich noch die beschriebenen Hilfsinformationen (siehe Kap. 5) ermittelt. Anschließend werden die Klassen des implementierten DBSCAN Algorithmus mittels UML-Diagramm dargestellt. Das UML-Diagramm enthält alle verwendeten Klassen, inklusive der zusätzlichen Klassen für die Ermittlung der Hilfsinformationen und die Datenbankanbindung. Pro Klasse werden die wichtigsten Attribute und die wichtigsten Methoden angeführt. Der Quellcode der Klassen und deren Attribute und Methoden kann im Anhang gefunden werden (siehe Kap. 12.2). Anschließend an das UML-Diagramm werden die Klassen und deren wichtige Attribute und Methoden näher erläutert.

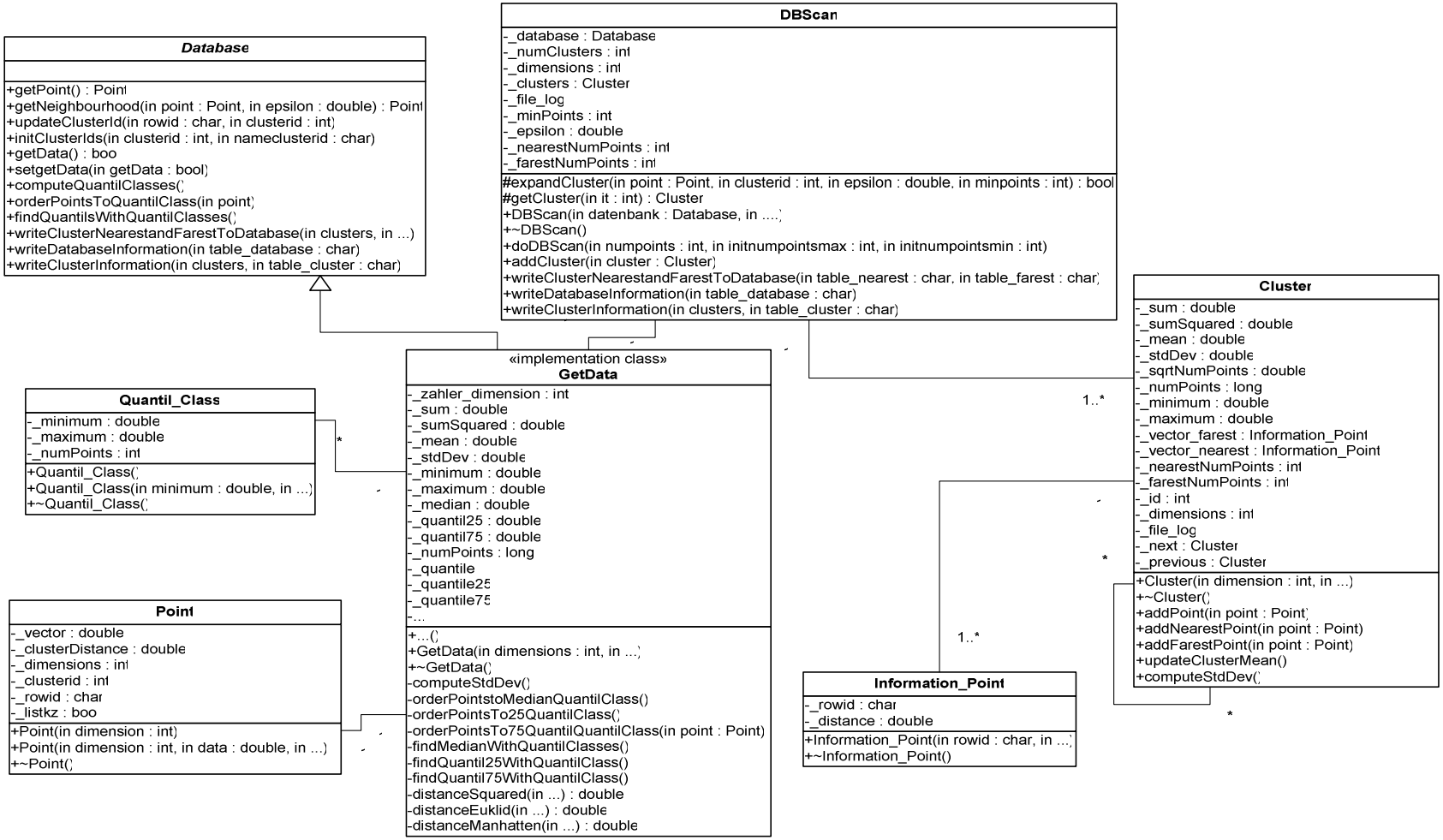


Abbildung 24: UML-Diagramm des erweiterten DBSCAN-Algorithmus

Database

Die Klasse *Database* ist eine abstrakte Klasse und enthält alle Methoden die eine Klasse für die Datenbankanbindung des DBSCAN Algorithmus benötigt. Eine Klasse die die Datenbankanbindung für den DBSCAN Algorithmus realisieren soll, muss von der Klasse *Database* erben und dessen Methoden implementieren. Nachfolgend werden die Methoden der Klasse beschrieben:

- **getPoint():**

  Die Methode *getPoint* liefert als Rückgabewert den nächsten Punkt des Datenbestands. Zusätzlich werden in dieser Methode ein Teil der Hilfsinformationen der Datenbasis berechnet (siehe Kap. 6.3.2.1).

- **getNeighbourhood (Point *point, double epsilon)**

  Die Methode *getNeighbourhood* ermittelt mit dem übergebenen *epsilon* die ε-Umgebung zu dem übergebenen Punkt *point*. Als Ergebnis liefert die Methode einen Vektor mit allen Punkten der ε -Umgebung

- **updateClusterId(char *rowid, int clusterid):**

  Die Methode *updateClusterId* führt ein Datenbankupdate der Cluster-Id eines Punkts durch. Der Punkt wird über seine Rowid identifiziert und die neue Cluster-Id wird übergeben.

- **initClusterIds(char *clusterid, char *nameclusterid):**

  Die Methode *initClusterIds* initialisiert alle Punkte mit der übergebenen Cluster-Id. Welche Spalte (Dimension) der Datenbank die Cluster-Id repräsentiert wird mit dem Attribut *nameclusterid* übergeben.

- **getData():**

  Die Methode *getData* liefert einen Boolean-Wert der angibt ob die Ausführung der Methode *getPoint* (Ermittlung des nächsten Punkts) erfolgreich war.

- **setgetData(bool getData):**

  Die Methode *setgetData* setzt Boolean-Attribut *getData*, das angibt ob das Lesen des Punkts aus der Datenbasis erfolgreich war (siehe *getData*).

• **writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char *table_farest)** Die Methode *writeClusterNearestandFarestToDatabase* schreibt die ermittelten Hilfsinformationen zu den nähesten und entferntesten Punkten der übergebenen Cluster in die übergebenen Datenbanktabellen. Die Datenbanktabellen werden dabei neu angelegt.
• **writeDatabaseInformation(char *table_database)** Die Methode *writeDatabaseInformation* schreibt die ermittelten Hilfsinformationen zur Datenbasis in die übergebene Datenbanktabelle. Die Datenbanktabelle wird dabei neu angelegt.
• **writeClusterInformation(Cluster *clusters, char *table_cluster)** Die Methode *writeClusterInformation* schreibt die ermittelten Hilfsinformationen zu den gefundenen Clustern in die übergebene Datenbanktabelle. Die Datenbanktabelle wird neu angelegt.
• **computeQuantilClasses()**: Ermittlung von Klassen die zur näherungsweisen Berechnung der Quantile benötigt werden (siehe Kap. 6.3.2)
• **orderPointsToQuantilClass(Point *point):** Die Punkte der Datenbasis werden zur Berechnung der Quantile ihrer jeweiligen Klasse zugeordnet (siehe Kap. 6.3.2)
• **findQuantilsWithQuantilClasses()**: Methode ermittelt ungefähre Quantile (25%, Median, 75%) für Datenbasis

Tabelle 20: DBSCAN: Methoden der abstrakten Klasse Database

GetData

Die Klasse *GetData* erbt von der abstrakten Klasse Database und implementiert die oben angeführten Methoden dieser Klasse. Diese Klasse realisiert die tatsächliche Datenbankanbindung des DBSCAN Algorithmus und die Ermittlung der Hilfsinformationen zur Datenbasis. Nachfolgend werden die im UML-Diagramm angeführten Attribute der Klasse *GetData* zur Speicherung der Hilfsinformationen beschrieben:

• **_sum:** Summe über die Werte der Dimensionen aller Punkte
• **_sumSquared:** Quadratsumme über die Werte der Dimensionen aller Punkte
• **_mean:** Mittelwert über die Werte der Dimensionen aller Punkte
• **_stdDev:** Standardabweichung pro Dimension aller Punkte
• **_minimum:** Minimum pro Dimension aller Punkte
• **_maximum:** Maximum pro Dimension aller Punkte
• **_median:** Median pro Dimension aller Punkte
• **_quantil25:** 25%-Quantil pro Dimension aller Punkte
• **_quantil75:** 75%-Quantil pro Dimension aller Punkte
• **_numPoints:** Anzahl der Punkte

Tabelle 21: DBSCAN: Attribute zur Berechnung der Hilfsinformationen der Klasse GetData

Zusätzlich werden auch die weiteren Attribute, die ausschließlich intern verwendet werden, erläutert:

• **_clusterPoints:** Anzahl der zu clusternden Punkte
• **_zaehler_dimension:** Gibt an welche Tabellenspalte welcher Dimension entspricht
• **_quantile:** Hilfsvektor zur Berechnung des Medians
• **_quantile25:** Hilfsvektor zur Berechnung des 25%-Quantils

• **_quantile75:** Hilfsvektor zur Berechung des 75%-Quantils
• **_clusterid:** Dimension die Cluster-Id entspricht
• **_dimensions:** Anzahl der Dimensionen
• **_distancefunction:** Steuerung welche Distanzfunktion verwendet wird (1 = Quadratische Distanz, 2 = Euklidische Distanz, 3 = Manhatten – Distanz)
• **_getData:** Gibt an ob Lesen aus Datenbank erfolgreich war
• **_point:** Punkt der aus Datenbank gelesen wurde
• **_table:** Tabellenname der Datenbasis
• **_columns:** Spalten der Datenbasis die für Algorithmus verwendet werden
• **_usedcolumns:** Gibt an welche Spalten für Clustering relevant sind
• **_file_log:** Log-File für Protokollierung
• **_compute_database_information:** Boolean Wert der angibt ob die Hilfsinformationen der Datenbasis berechnet werden, während der Ermittlung der ε-Umgebung, oder nicht
• **_points:** Anzahl der Punkte die bisher vom DBSCAN berücksichtigt wurden
• **_countneighbourhood:** Zähler für ε -Nachbarschaftsermittlung zur Ermittlung der Quantile

Tabelle 22: DBSCAN: Restliche Attribute der Klasse GetData

Die weiteren wichtigen Methoden der Klasse *GetData* werden nachfolgend beschrieben. Die Methoden zur Kapselung der Attribute werden hier nicht näher erläutert. Diese sind im Anhang angeführt (siehe Kap. 12.2)

• **GetData(int dimensions, int clusterid, int distancefunction, char *table, char *columns, bool *usedcolumns, FILE *file_log, long clusterpoints):** Konstruktor der Klasse *GetData,* der die nötigen Attribute der Klasse *GetData* initialisiert.

- **~GetData():**

  Destruktor der Klasse *GetData*

- **distanceSquared(double *vector_start, double *vector_2):**

  Die quadratische Distanz zwischen den beiden übergebenen Vektoren wird berechnet und als Ergebnis zurück gegeben

- **distanceEuklid(double *vector_start, double *vector_2):**

  Die euklidsche Distanz zwischen den beiden übergebenen Vektoren wird berechnet und als Ergebnis zurück gegeben

- **distanceManhatten(double *vector_start, double *vector_2):**

  Die Manhatten-Distanz zwischen den beiden übergebenen Vektoren wird berechnet und als Ergebnis zurück gegeben

- **computeStdDev():**

  Die Standardabweichung pro Dimension aller verwendeten Punkte wird berechnet

- **orderPointsToMedianQuantilClass(Point *point):**

  Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des Median zugeordnet (siehe Kap. 6.3.2)

- **orderPointsTo25QuantilQuantilClass(Point *point):**

  Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des 25%-Quantils zugeordnet (siehe Kap. 6.3.2)

- **orderPointsTo75QuantilQuantilClass(Point *point):**

  Der übergebene Punkt wird der entsprechenden Klasse zur näherungsweisen Berechnung des 75%-Quantils zugeordnet (siehe Kap. 6.3.2)

- **findMedianWithQuantilClasses():**

  Näherungsweise Ermittlung des Median über die Klassen (siehe Kap. 6.3.2)

- **findQuantil25WithQuantilClasses():**

Näherungsweise Ermittlung des 25%-Quantils über die Klassen (siehe Kap. 6.3.2)
• **findQuantil75WithQuantilClasses():** Näherungsweise Ermittlung des 75%-Quantils über die Klassen (siehe Kap. 6.3.2)

Tabelle 23: DBSCAN: Methoden der Klasse GetData

Point

Die Klasse Point repräsentiert einen Punkt, der aus der Datenbank gelesen wird. Die Klasse enthält folgende Attribute:

• **_vector:** Werte der Dimensionen des Punkts
• **_rowid:** Rowid des Punkts
• **_clusterid:** Cluster-Id des Punkts
• **_clusterDistance:** Distanz des Punkts zu dem Centroiden seines Clusters
• **_dimensions:** Anzahl der Dimensionen des Punkts
• **_listkz:** Boolean-Wert der angibt ob Destruktor aufgerufen werden soll oder nicht

Tabelle 24: DBSCAN: Attribute der Klasse Point

Nachfolgend werden die Methoden der Klasse Point beschrieben. Die weiteren Methoden zur Kapselung der Attribute werden hier nicht näher erläutert (siehe Kap. 12).

• **Point(int dimension):** Konstruktor der Klasse *Point* zur Initialisierung
• **Point(int dimension, double *vector, char *rowid, int clusterid):** Zweiter Konstruktor der Klasse *Point* zur Initialisierung
• **~Point():** Destruktor der Klasse *Point*

Tabelle 25: DBSCAN: Methoden der Klasse Point

Cluster

Die Klasse *Cluster* repräsentiert einen ermittelten Cluster des DBSCAN Algorithmus. Darin werden auch die ermittelten Hilfsinformationen pro Cluster gespeichert. Die Attribute zur Speicherung der Hilfsinformationen sehen wie folgt aus:

• **_sum:** Summe der Werte der Punkte des Clusters pro Dimension
• **_sumSquared:** Quadratsumme der Werte der Punkte des Clusters pro Dimension
• **_mean:** Mittelpunkt des Clusters (Centroid)
• **_stdDev:** Standardabweichung der Punkte des Clusters pro Dimension
• **_minimum:** Minimum pro Dimension
• **_maximum:** Maximum pro Dimension
• **_sqrtNumPoints:** Wurzel über die Anzahl der Punkte des Clusters
• **_numPoints:** Anzahl der Punkte des Clusters
• **_vector_farest:** Vektor mit entferntesten Punkten des Clusters (sortiert nach Distanz)
• **_vector_nearest:** Vektor mit nähesten Punkten des Clusters (sortiert nach Distanz)

Tabelle 26: DBSCAN: Attribute zur Berechnung der Hilfsinformationen der Klasse Cluster

Zusätzlich werden auch die weiteren Attribute, die ausschließlich intern verwendet werden, erläutert:

• **_nearestNumPoints:** Anzahl der nähesten Punkte, die gespeichert werden sollen
• **_farestNumPoints:** Anzahl der entferntesten Punkte die gespeichert werden sollen
• **_id:** Cluster-Id

• **_dimensions:** Anzahl der Dimensionen pro Punkt
• **_file_log:** Log-File für Protokollierung
• **_next, _previous:** Zeiger auf nächsten bzw. vorherigen Cluster

Tabelle 27: Restliche Attribute der Klasse Cluster

Nachfolgend werden die Methoden der Klasse *Cluster* beschrieben. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert. Diese sind im Anhang (siehe Kap. 12.2) angeführt.

• **Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, FILE *file_log):** Konstruktor der Klasse *Cluster* zur Initialisierung der Attribute
• **~Cluster():** Destruktor der Klasse *Cluster*
• **addPoint(Point *point):** Die Methode *addPoint* fügt dem Cluster einen Punkt hinzu und berechnet dabei auch die Hilfsinformationen wie Summe, Quadratsumme und Anzahl der Punkte
• **updateClusterMean():** Mittelpunkt des Clusters wird ermittelt (Centroid)
• **computeStdDev():** Standardabweichung der Punkte wird pro Dimension ermittelt. Zusätzlich wird die Wurzel der Anzahl der Punkte berechnet.
• **addNearestPoint(Point *point):** Wenn der Punkt ein Kernobjekt ist wird er nach einer gewissen Logik den nähesten Punkten zugeordnet (siehe Kap. 6.3.2).
• **addFarestPoint(Point *point):**

Wenn der Punkt kein Kernobjekt ist, also ein Randobjekt, dann wird dieser nach einer bestimmten Logik den entferntesten Punkten zugeordnet (siehe Kap. 6.3.2).

Tabelle 28: DBSCAN: Methoden der Klasse Cluster

Quantil_Class

Hilfsklasse zur näherungsweisen Berechnung des 25%-Quantil, Median und des 75%-Quantil. Eine genaue Erläuterung der Ermittlung der Hilfsinformationen ist in Kapitel 6.3.2 zu finden. Nachfolgend werden die Attribute der Hilfsklasse dargestellt:

• **_minimum:** Untere Grenze der Klasse
• **_maximum:** Obere Grenze der Klasse
• **_numPoints:** Anzahl der Punkte in Klasse

Tabelle 29: DBSCAN: Attribute der Klasse Quantil_Class

Methode der Klasse *Quantil_Class*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **Quantil_Class():** Konstruktor der Klasse
• **Quantil_Class(double minimum, double maximum):** Zweiter Konstruktor der Klasse
• **~Quantil_Class():** Destruktor der Klasse
• **addNumPoints():** Anzahl der Punkte in Klasse wird um 1 erhöht

Tabelle 30: DBSCAN: Methoden der Klasse Quantil_Class

Information_Point

Hilfsklasse für die Vektoren der nähesten und entferntesten Punkte (siehe Kap. 6.3.2). Attribute der Klasse *Information_Point*:

• **_rowid:** Rowid des Punkts
• **_distance:** Distanz des Punkts zu seinem Centroide

Tabelle 31: DBSCAN: Attribute der Klasse Information_Point

Methoden der Klasse *Informatione_Point*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **Information_Point(char *rowid, double distance):** Konstruktor der Klasse
• **~Informatione_Point():** Destruktor der Klasse

Tabelle 32: DBSCAN: Methoden der Klasse Information_Point

DBScan

Die Klasse *DBScan* ist sozusagen die „Hauptklasse“ und führt auch den DBSCAN Algorithmus aus. Eine genaue Beschreibung der Implementierung des Algorithmus kann in Kapitel 6.3.1 gefunden werden. Nachfolgend werden die Attribute der Klasse *DBScan* näher erläutert:

• **_database:** Zugrunde liegende Datenbasis des Algorithmus
• **_numClusters:** Anzahl der ermittelten Cluster
• **_clusters:** Cluster die der Algorithmus erzeugt
• **_dimensions:** Anzahl der Dimensionen pro Punkt
• **_file_log:** Log-File für Protokollierung
• **_minpoints:** Minimale Anzahl von Punkten in Epsilon-Umgebung
• **_epsilon:** Epsilon-Wert für Epsilon-Umgebung
• **_nearestNumPoints:** Anzahl der nähesten Punkte pro Cluster, die gespeichert werden sollen
• **_farestNumPoints:** Anzahl der am weitesten entfernten Punkte pro Cluster, die gespeichert werden sollen

Tabelle 33: DBSCAN: Attribute der Klasse DBScan

Methoden der Klasse *DBScan*. Die weiteren Methoden zur Kapselung der Attribute (set und get-Methoden) werden hier nicht näher erläutert.

• **DBScan(Database *datenbank, int dimension, double epsilon, int minpoints, int nearestnumpoints, int farestnumpoints, FILE *file_log):** Konstruktor der Klasse DBScan
• **~DBScan():** Destruktor der Klasse DBScan
• **doDBScan():**

Aufruf des DBSCAN-Algorithmus
• **expandCluster(Point *point, int clusterid, double epsilon, int minpoints):** Methode expandCluster liefert zu übergebenen Punkt ein Cluster über die Epsilon-Umgebung des Punkts (siehe Kap. 6.3.1)
• **addCluster(Cluster *cluster):** Methode addCluster fügt erzeugtes Cluster zu Liste der Cluster hinzu
• **getNextClusterId(int oldclusterid):** Methode getNextClusterId liefert nächste Cluster Id
• **getCluster(int id):** Liefert das Cluster zur übergebenen Cluster-Id
• **writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char *table_farest)** Die Methode *writeClusterNearestandFarestToDatabase* schreibt die ermittelten Hilfsinformationen zu den nähesten und entferntesten Punkte der übergebenen Cluster in die übergebenen Datenbanktabellen.
• **writeDatabaseInformation(char *table_database)** Die Methode *writeDatabaseInformation* schreibt die ermittelten Hilfs-informationen zur Datenbasis in die übergebene Datenbanktabelle.
• **writeClusterInformation(Cluster *clusters, char *table_cluster)** Die Methode *writeClusterInformation* schreibt die ermittelten Hilfsinformationen zu den gefundenen Clustern in die übergebene Datenbanktabelle.

Tabelle 34: DBSCAN: Methoden der Klasse DBScan

6.3.1 DBSCAN Algorithmus

Der implementierte DBSCAN Algorithmus entspricht, bis auf die Ermittlung der Hilfsinformationen, dem vorgestellten DBSCAN Algorithmus von Ester und Sander (siehe Kap. 2.2.2.2). Da die Implementierung nicht vom Vorschlag von Ester und Sander [ES00] abweicht, wird nachfolgend nur kurz auf die Implementierung eingegangen. Dabei werden Auszüge des Quellcodes der wichtigsten Methoden *doDBScan* und *expandCluster* gezeigt. Die Ermittlung der Hilfsinformationen wird dabei aber noch nicht berücksichtigt (siehe Kap. 6.3.2).

```
/**
 * Methode doDBScan führt DBScan-Algorithmus aus
 * @see expandCluster()
 */
void DBScan::doDBScan(){
        Point *point;

        // Erster unklassifizierte Punkt wird aus Datenbank gelesen
        point = _database->getPoint();

        // Ermittlung der ersten Cluster ID
        int clusterId = getNextClusterId(ID_NOISE);

        // Solange ein unklassifzierter Punkt gefunden wird wird versucht zu diesem
        // ein Cluster zu finden
        while (_database->getData()){
                if (point->getClusterId() == ID_UNCLASSIFIED)
                        if (expandCluster(point, clusterId, _epsilon, _minPoints))
                                clusterId = getNextClusterId(clusterId);
                point = _database->getPoint();
        }
}

/**
 * Methode expandCluster liefert zu übergebenen Punkt ein Cluster über die
 * Epsilon-Umgebung des Punkts
 * @param point Punkt zu dem Cluster über Epsilon-Umgebung ermittelt wird
 * @param clusterid Cluster Id des zu erzeugenden Clusters
 * @param epsilon Epsilon-Wert für Epsilon-Umgebung
 * @param minpoints Minimale Anzahl von Punkten in Epsilon-Umgebung
 * @return Boolean-Wert der angibt ob Cluster zu Punkt ermittelt wurde
 */
bool DBScan::expandCluster(Point *point, int clusterid, double epsilon, int
minpoints){
        // Epsilon-Umgebung zu Punkt wird ermittelt
        list<Point> seeds = _database->getNeighbourhood(point,epsilon);
        list<Point>::iterator i,j;

        // Wenn Anzahl der Punkte die Anzahl der minimal geforderten Punkte nicht
```

```
        // erreicht wird der übergebene Punkt als Noise identifiziert
        if (seeds.size() < minpoints){
                _database->updateClusterId(point,ID_NOISE);
                return false;
        }

        Cluster *cluster = new Cluster(_dimensions, _nearestNumPoints,
                                        _farestNumPoints, clusterid, _file_log);

        // Punkte der Epsilon-Umgebung werden zu Cluster hinzugefügt und
        // Cluster ID der Punkte wird gesetzt
        for (i = seeds.begin(); i != seeds.end(); i++){
                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i->getClusterId() ==
                        ID_NOISE))
                        _database->updateClusterId(i->getPoint(),clusterid);
                i->setClusterId(clusterid);
                cluster->addPoint(i->getPoint());
                cluster->addNearestPoint(i->getPoint());
        }
        _database->updateClusterId(point,clusterid);
        point->setClusterId(clusterid);
        cluster->addPoint(point);

        // Zu den ermittelten Punkte wird wieder die Epsilon-Umgebung erzeugt, bis
        // es keine mehr gibt. Die gefundenen Punkte werden zum Cluster
        // hinzugefügt und die Cluster Id der Punkte wird gesetzt
        while (!seeds.empty()){
                j = seeds.begin();
                list<Point> neighbourhood = _database->getNeighbourhood(j-
                                                        >getPoint(),epsilon);

                if (neighbourhood.size() >= minpoints)
                        for (i = neighbourhood.begin(); i != neighbourhood.end(); i++)
                                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i-
                                                        >getClusterId() == ID_NOISE)){
                                        if (i->getClusterId() == ID_UNCLASSIFIED)
                                                seeds.push_back(*i);      // Hinzufügen zu seeds
                                        _database->updateClusterId(i->getPoint(), clusterid);
                                        cluster->addPoint(i->getPoint());
                                }
                seeds.pop_front();        //        entferne Punkt aus seeds
        }
        addCluster(cluster);
        return true;
}
```

Die Implementierung entspricht der vorgestellten Implementierung von Ester und Sander [ES00]. Die Ermittlung der ε-Umgebung erfolgt ohne Nutzung einer Indexstruktur. Der gesamte Quellcode aller Klassen ist im Anhang zu finden (siehe Kap. 12.2).

6.3.2 Ermittlung der Hilfsinformationen

Bisher wurden die verwendeten Klassen und der DBSCAN Algorithmus näher erläutert. Anschließend wird detailliert auf die Implementierung zur Ermittlung der Hilfsinformationen eingegangen. Dabei wird unterschieden zwischen den Hilfsinformationen zur Datenbasis (siehe Kap. 5.1) und den Hilfsinformationen zu den gefundenen Clustern (siehe Kap. 5.2).

6.3.2.1 Hilfsinformationen zur Datenbasis

Die Hilfsinformationen zur Datenbasis werden bei der ersten Ermittlung der ε-Umgebung berechnet. Damit die Ermittlung nur beim ersten Mal durchgeführt wird, wird das Attribut *_compute_database_information* der Klasse *GetData* verwendet. Ist dieses *true* wird die Ermittlung durchgeführt, sonst nicht. Die Methode *getNeighbourhood* der Klasse *GetData* führt die Ermittlung der ε-Umgebung aus. Nachfolgend wird der Quellcode dieser Methode dargestellt:

```
/**
 * Methode getNeighbourhood liefert alle Punkte der Epsilon-Umgebung des
 * übergebenen Punkts mit dem übergebenen Epsilon-Wert
 * @param point Punkt zu dem Epsilon-Umgebung berechnet wird
 * @param epsilon Wert des Epsilon
 * @return Punkte in Epsilon-Umgebung des übergebenen Punkts
 */
list<Point> GetData::getNeighbourhood(Point *point, double epsilon){
        list<Point> l_neighbourhood;
        Point *h_point;
        int i=0, j=0, clusterid=0, anzahl=0;
        double distance = 0;
        char *rowid;

        l_neighbourhood.clear();
        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND  GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        EXEC SQL OPEN neighbourhood_cursor USING DESCRIPTOR
        bind_des_neighbour;
        anzahl = 0;
        _countneighbourhood++;
        if (_countneighbourhood == 2)
                this->computeQuantilClasses();

        while(_clusterpoints > anzahl){
                EXEC SQL FETCH neighbourhood_cursor USING DESCRIPTOR
                select_des_neighbour;
                anzahl++;
                if (_compute_database_information)
```

```
        _numPoints++;
strncpy((char *)rowid1.arr,point->getRowId(), 18);
rowid1.len = 18;
strncpy((char *)rowid2.arr,(char *)select_des_neighbour->V[0], 18);
rowid2.len = 18;
// Startobjekt nicht noch einmal hinzufügen
if (strcmp((char *)rowid1.arr,(char *)rowid2.arr) != 0){
        j = 0;
        for (i=0; i < select_des_neighbour->F; i++){
                if (select_des_neighbour->T[i] == 3){
                        _data[j] = (int) *select_des_neighbour->V[i];   // INT
                        if (_compute_database_information){
                                _sum[j]+= _data[j];
                                _sumSquared[j]+= _data[j] * _data[j];
                                _zaehler_dimension[j] = i;

                                // Minimum bestimmen
                                if (_data[j] < _minimum[j])
                                    _minimum[j] = _data[j];
                                // Maximum bestimmen
                                if (_data[j] > _maximum[j])
                                    _maximum[j] = _data[j];
                        }
                        j++;
                }
                if (select_des_neighbour->T[i] == 4){         // FLOAT
                        _data[j] = *(float *)select_des_neighbour->V[i];
                        if (_compute_database_information){
                                _sum[j]+= _data[j];
                                _sumSquared[j]+= _data[j] * _data[j];
                                _zaehler_dimension[j] = i;

                                // Minimum bestimmen
                                if (_data[j] < _minimum[j])
                                    _minimum[j] = _data[j];
                                // Maximum bestimmen
                                if (_data[j] > _maximum[j])
                                    _maximum[j] = _data[j];
                        }
                        j++;
                }
                if (i == 0)                                     // ROWID
                        rowid = (char *)select_des_neighbour->V[i];
                if (i == _clusterid)                            // CLUSTER ID
                        clusterid = (int) *select_des_neighbour->V[i];
        }
        // Distanz wird berechnet
        switch( _distancefunction){
                case 1:                                 // Quadratische Distanz
                        distance = distanceSquared(point->getVector(),
                                                                _data);
                        break;
```

```
                case 2:                                 // Euklidische Distanz
                    distance = distanceEuklid(point->getVector(), _data);
                    break;
                case 3:                                 // Manhatten Distanz
                    distance = distanceManhatten(point->getVector(),
                                                                _data);
                    break;
            }
            if ((_countneighbourhood > 1) &&
                (_countneighbourhood <= 10)){
                h_point = new Point(_dimensions);
                h_point->set(_data,rowid,clusterid,true);
                // Klassen zur Quantil-Berechnung füllen
                this->orderPointsToQuantilClass(h_point);
                delete h_point;
            }
            // Prüfen ob Punkt in Epsilon-Umgebung ist
            if (distance <= epsilon){
                h_point = new Point(_dimensions);
                h_point->set(_data,rowid,clusterid,true);
                l_neighbourhood.push_front(*h_point);
                delete h_point;
            }
        } else {
            // Übergebener Punkt muss auch bei Berechnung der
            // Hilfsinformationen berücksichtigt werden
            if (_compute_database_information){
                j = 0;
                for (i=0; i < select_des_neighbour->F; i++){
                    if (select_des_neighbour->T[i] == 3){
                        _sum[j]+= _data[j];
                        _sumSquared[j]+= _data[j] * _data[j];
                        _zaehler_dimension[j] = i;
                        _minimum[j] = _data[j];
                        _maximum[j] = _data[j];
                        j++;
                    }
                    if (select_des_neighbour->T[i] == 4){
                        _sum[j]+= _data[j];
                        _sumSquared[j]+= _data[j] * _data[j];
                        _zaehler_dimension[j] = i;
                        _minimum[j] = _data[j];
                        _maximum[j] = _data[j];
                        j++;
                    }
                }
            }
        }
    }
    if ((_compute_database_information) && (anzahl == _clusterpoints)){
        computeStdDev();
        for (int i = 0; i < _dimensions; i++)
```

```
                    _mean[i] = _sum[i] / _numPoints;          // Mittelwerte berechnen
        }
        // Quantile (25%, 50% und 75%) werden näherungsweise ermittelt
        // (nach jeder Iteration genauer)
        if ((_countneighbourhood > 1) && (_countneighbourhood <= 10))
            this->findQuantilsWithQuantilClasses();
        EXEC SQL CLOSE neighbourhood_cursor;
        _compute_database_information = false;
        return l_neighbourhood;
sqlerror:
        if ((sqlca.sqlcode == -1002) || (sqlca.sqlcode == 1403)) {
            EXEC SQL CLOSE neighbourhood_cursor;
            if ((_compute_database_information) && (anzahl == _clusterpoints)){
                computeStdDev();
                for (int i = 0; i < _dimensions; i++)
                  _mean[i] = _sum[i] / _numPoints;       // Mittelwerte berechnen
            }
            _compute_database_information = false;
            return l_neighbourhood;
        } else {
            fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
            EXEC SQL ROLLBACK RELEASE;
            fflush(_file_log);
            exit(1);
            return 0;
        }
}
```

Die Stellen des Quellcode, die die Ermittlung der Hilfsinformationen veranlassen, sind blau markiert. Die arithmetischen Werte Summe, Quadratsumme, Minimum, Maximum, Mittelwert und Anzahl der Punkte werden direkt in dieser Methode ermittelt. Der Mittelwert wird dabei, genauso wie die Standardabweichung, erst nach dem Einlesen des letzten Punktes berechnet. Die Standardabweichung wird durch den Aufruf der Methode *computeStdDev* berechnet.

```
/**
 * Methode computeStdDev ermittelt die Standardabweichung der
 */
void GetData::computeStdDev(){
        double *sum = _sum, *sumSquared = _sumSquared, *stdDev = _stdDev,
                h_stddev;

        for(int i = 0; i < _dimensions; i++){
            if (_usedcolumns[i]){
                h_stddev = *sum++ / _numPoints;
                *stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints –
                                h_stddev * h_stddev));
            }
         }
}
```

Berechnung der Quantile

Bei einer exakten Berechnung der Quantile (25%-Quantil, Median, 75%-Quantil) ist es nötig, dass die Werte jeder Dimension sortiert vorliegen. Nur dann können die Quantile genau ermittelt werden. Dies ist beim Ermitteln der ε-Umgebung allerdings nicht der Fall. Die Quantile werden bei diesem DBSCAN-Algorithmus näherungsweise bestimmt. Dabei wird wie folgt vorgegangen:

- Bei der ersten Bestimmung der ε-Umgebung wird das Minimum und das Maximum pro Dimension ermittelt (ist bereits eine Hilfsinformation). Der Bereich zwischen Minimum und Maximum wird in 100 gleich große Klassen geteilt.
- Wird die ε-Umgebung erneut bestimmt, wird gezählt wie viele Punkte in jede Klasse fallen. Summiert man die Anzahl der Punkte in den Klassen auf, erhält man Quantile. Diese werden zwar beliebig sein, z.B. kann Klasse 20 das 47 % Quantil darstellen und Klasse 21 das 51 % Quantil. Aber dadurch weiß man, dass sich der Median zwischen den Werten von Klasse 21 befinden muss. In jeder Folge-Iteration kann die entsprechende Klasse weiter verfeinert werden. Je mehr Iterationen erfolgen, desto genauer wird die Annäherung der Quantile.
- Der Mittelwert der Klasse in der sich nach der letzten Iteration zum Beispiel der Median befindet wird als näherungsweiser Median herangezogen.
- Für jedes Quantil, d.h. 25%-Quantil, 75%-Quantil und Median, werden eigens 100 Klassen in Form eines Vektors angelegt und wie bereits beschrieben verfeinert. Dadurch wird für jedes Quantil eine Näherungswert erzielt.

Die Methoden zur Ermittlung der Quantile gehören zur Klasse *GetData*. Die Methode *computeQuantilClasses* ermittelt zu Beginn des zweiten Aufrufs der ε-Umgebung die 100 Klassen für jedes Quantil. Ab dem zweiten Aufruf der ε -Umgebung bis zum zehnten Aufruf der ε -Umgebung wird über die Methode *orderPointsToQuantilClass* ein Punkt zu seiner jeweiligen Klasse pro Quantil addiert. Am Ende der Ermittlung der ε–Umgebung ermittelt die Methode *findQuantilsWithQuantilClasses* die näherungsweise ermittelten Quantile. Der Aufruf der Methoden zur Berechnung der Quantile wurde zuvor in der Methode getNeighbourhood dargestellt und blau markiert. Der Quellcode der einzelnen Methoden befindet sich im Anhang (siehe Kap. 12.2).

6.3.2.2 Hilfsinformationen zu den gefundenen Clustern

Die Hilfsinformationen zu den Clustern werden während der Ausführung des DBSCAN Algorithmus berechnet. Je nach Hilfsinformation werden verschiedene Methoden

verwendet. Nachfolgend werden Auszüge aus dem Quellcode dieser Methoden pro Hilfsinformation dargestellt.

Summe, Quadratsumme, Minimum, Maximum und Anzahl der Punkte

Diese Hilfsinformationen werden beim Hinzufügen eines Punkts zu einem Cluster berechnet. Die im Folgenden vorgestellte Methode *addPoint* der Klasse *Cluster* berechnet diese Hilfsinformationen:

```
/**
 * Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die
 * Werte des Clusters neu
 * @param point Punkt der zu Cluster hinzugefügt wird
 */
void Cluster::addPoint(Point *point){
	// Werte des Clusters werden an lokale Zeiger übergeben
	double *vector = point->_vector, *sum = _sum, *sumSquared =
		_sumSquared, *minimum = _minimum, *maximum = _maximum;

	// Summen und Quadratsummen für jede Dimension werden berechnet
	for(int i = 0; i < _dimensions; i++){
		*sumSquared++ += *vector**vector;
		*sum++ += *vector++;
		// Minimum
		if (minimum[i] == 0)
			minimum[i] = vector[i];
		else if (minimum[i] > vector[i])
			minimum[i] = vector[i];
		// Maximum
		if (maximum[i] == 0)
			maximum[i] = vector[i];
		else if (maximum[i] < vector[i])
			maximum[i] = vector[i];
	}
	_numPoints++;		// Anzahl der Punkte
}
```

Mittelpunkt (Centroid)

Der Mittelpunkt eines Clusters wird am Ende der Methode expandCluster berechnet, nachdem alle Punkte dem Cluster zugewiesen wurden. Das Aktualisieren des Mittelpunkts erfolgt mit der Methode *updateClusterMean* der Klasse *Cluster*.

```
/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void Cluster::updateClusterMean(void){
	double *sum = _sum, *mean = _mean;
```

```
        // Mittelpunkt des Clusters wird neu berechnet
        if(_numPoints)
                for(int i = 0; i < _dimensions; i++)
                        *mean++ = *sum++ / _numPoints;
}
```

Standardabweichung und Wurzel der Anzahl der Punkte

Die Standardabweichung und die Wurzel der Anzahl der Punkte werden wie der Mittelpunkt am Ende der Methode *expandCluster* ermittelt, in dem die Methode *computeStdDev* der Klasse *Cluster* aufgerufen wird.

```
/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des
 * Clusters
 */
void Cluster::computeStdDev(){
        double *sum = _sum, *sumSquared = _sumSquared, *l_stdDev = _stdDev,
                h_stddev;

        for(int i = 0; i < _dimensions; i++){
                h_stddev = *sum++ / _numPoints;
                *l_stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints –
                                        h_stddev * h_stddev));
    }
    _sqrtNumPoints = (double) sqrt(_numPoints);
}
```

Nähesten und entferntesten Punkte pro Cluster

Wenn ein Punkt ein Kernobjekt ist, dann ist es ein potentieller Kandidat für die nähesten Punkte eines Clusters. Jeder Punkt der ein Kernobjekt ist wird abhängig von einer Zufallszahl den nähesten Punkten zugewiesen. Die Zufallszahl liegt zwischen 0 und der Anzahl der Punkte im Cluster. Falls diese Zufallszahl kleiner ist als die Menge der zu ermittelnden nähesten Punkte wird dieser Punkt den nähesten Punkten hinzugefügt. Gleiches gilt für die entferntesten Punkte wenn der Punkt ein Randobjekt ist. Nachfolgend werden die Methoden addNearestPoint und addFarestPoint zur Ermittlung der nähesten und entferntesten Punkte dargestellt:

```
/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des
 * Clusters gehört. Die _farestNumPoints werden in einer eigenen Liste
 * gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten
 *                      Punkten gehört
 */
void Cluster::addFarestPoint(Point *point){
        long zufall;
```

```
        Information_Point info_point(point->getRowId(),point-
                                                    >getClusterDistance());

        srand((unsigned)time( NULL ));
        zufall = rand() % _numPoints;

        if (zufall < _farestNumPoints)
            if (_vector_farest.size() < _farestNumPoints)
                _vector_farest.push_back(info_point);
            else {
                _vector_farest.erase(_vector_farest.begin() + zufall);
                _vector_farest.push_back(info_point);
            }
    }

    /**
     * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des
     * Clusters gehört. Die _nearestNumPoints werden in einer eigenen Liste
     * gespeichert
     * @see Cluster()
     * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten
     *                  gehört
     */
    void Cluster::addNearestPoint(Point *point){
        long zufall;
        Information_Point info_point(point->getRowId(), point-
                                                        >getClusterDistance());

        srand((unsigned)time( NULL ));
        zufall = rand() % _numPoints;

        if (zufall < _nearestNumPoints)
            if (_vector_nearest.size() < _nearestNumPoints)
                _vector_nearest.push_back(info_point);
            else {
                _vector_nearest.erase(_vector_nearest.begin() + zufall);
                _vector_nearest.push_back(info_point);
            }
    }
```

Bisher wurden die einzelnen Methoden zur Ermittlung der Hilfsinformationen pro Cluster vorgestellt. Nachfolgend wird der Aufruf der Methoden im DBSCAN Algorithmus (Methode *expandCluster*) gezeigt. Die Methoden zur Ermittlung dieser Hilfsinformation sind blau markiert.

```
    /**
     * Methode expandCluster liefert zu übergebenen Punkt ein Cluster über die
     * Epsilon-Umgebung des Punkts
     * @param point Punkt zu dem Cluster über Epsilon-Umgebung ermittelt wird
     * @param clusterid Cluster Id des zu erzeugenden Clusters
     * @param epsilon Epsilon-Wert für Epsilon-Umgebung
```

```
 * @param minpoints Minimale Anzahl von Punkten in Epsilon-Umgebung
 * @return Boolean-Wert der angibt ob Cluster zu Punkt ermittelt wurde (wenn
 *              nein --> Noise)
 */
bool DBScan::expandCluster(Point *point, int clusterid, double epsilon, int
minpoints){
        // Epsilon-Umgebung zu Punkt wird ermittelt
        list<Point> seeds = _database->getNeighbourhood(point,epsilon);
        list<Point>::iterator i,j;

        // Wenn Anzahl der Punkte die Anzahl der minimal geforderten Punkte nicht
        // erreicht wird der übergebene Punkt als Noise identifiziert
        if (seeds.size() < minpoints){
                _database->updateClusterId(point,ID_NOISE);
                return false;
        }

        Cluster *cluster = new Cluster(_dimensions, _nearestNumPoints,
                                        _farestNumPoints, clusterid, _file_log);
        // Punkte der Epsilon-Umgebung werden zu Cluster hinzugefügt und
        // Cluster ID der Punkte wird gesetzt
        for (i = seeds.begin(); i != seeds.end(); i++){
                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i->getClusterId() ==
                                                                        ID_NOISE))
                        _database->updateClusterId(i->getPoint(),clusterid);
                i->setClusterId(clusterid);
                cluster->addPoint(i->getPoint());
                cluster->addNearestPoint(i->getPoint());
        }
        _database->updateClusterId(point,clusterid);
        point->setClusterId(clusterid);
        cluster->addPoint(point);

        // Zu den ermittelten Punkte wird wieder die Epsilon-Umgebung erzeugt, bis
        // es keine mehr gibt. Die gefundenen Punkte werden wieder zum Cluster
        // hinzugefügt und die Cluster Id der Punkte wird gesetzt
        while (!seeds.empty()){
                j = seeds.begin();
                list<Point> neighbourhood = _database->getNeighbourhood(j-
                                                        >getPoint(),epsilon);
                if (neighbourhood.size() >= minpoints)
                        for (i = neighbourhood.begin(); i != neighbourhood.end(); i++)
                                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i-
                                                        >getClusterId() == ID_NOISE)){
                                        if (i->getClusterId() == ID_UNCLASSIFIED)
                                                seeds.push_back(*i);        // Hinzufügen zu seeds
                                        _database->updateClusterId(i->getPoint(), clusterid);

                                        cluster->addPoint(i->getPoint());
                                        cluster->addNearestPoint(i->getPoint());
                                }
                else
```

```
                cluster->addFarestPoint(j->getPoint());
            seeds.pop_front();        //        entferne Punkt aus seeds
        }
        cluster->updateClusterMean();
        cluster->computeStdDev();
        addCluster(cluster);
        return true;
    }
```

Der gesamte Quellcode des DBSCAN Algorithmus befindet sich im Anhang (siehe Kap. 12.2).

6.3.3 Parameter des DBSCAN Algorithmus

Beim Aufruf des beschriebenen DBSCAN Algorithmus zur Ermittlung der Hilfsinformationen müssen bestimmte Parameter, wie die Anzahl der Cluster, die Anzahl der nähesten Punkte usw., befüllt werden. Nachfolgend werden die Parameter des DBSCAN Algorithmus angeführt. Als zusätzliche Information werden auch der Typ und eine mögliche Musseingabe pro Parameter angegeben:

Parameter	Typ	Muss	Beschreibung	Beispiel
ÚSER	STRING	X	User für Datenbank	stoett
PASSWORD	STRING	X	Passwort für Datenbank	klaus
SERVICE	STRING	X	Datenbankservice	dke3
TABLE	STRING	X	Tabelle mit zu clusternden Daten	clusteringtestdaten
COLUMNS	STRING	X	Dimensionen der Tabelle	"t.X2, t.X3, t. CLUSTER_ID"
CLUSTERPOINTS	INTEGER	X	Anzahl der zu clusternden Punkte	100000
MINPOINTS	INTEGER	X	Anzahl der minimalen Punkte pro ε-Umgebung	6
EPSILON	DOUBLE	X	Wert des ε zur Ermittlung der ε-Umgebung	2000
CLUSTERID	INTEGER	X	Position der Cluster ID in Parameter COLUMNS	4
NAMECLUSTERID	STRING	X	Name der Cluster ID	CLUSTER_ID
DISTANCEFUNCTION	INTEGER	X	Distanzfunktion (1 = Quadrat, 2 = Euklid, 3 = Manhatten)	2
NEARESTNUMPOINTS	INTEGER	X	Anzahl der nähesten Punkte pro Cluster	2000

FARESTNUMPOINTS	INTEGER	X	Anzahl der entferntesten Punkte pro Cluster	2000
FILE_DBSCAN	STRING	X	File in das Ergebnis geschrieben wird	kmeans.txt
FILE_LOG	STRING	X	Log-File für Algorithmus	kmeans_log.txt
IDUNCLASSIFIED	STRING	X	ID für unklassifizierte Punkte	-1
TABLE_NEAREST	STRING	X	Tabelle mit nähesten Punkten	kmeans_ nearest_2_0_1
TABLE_FAREST	STRING	X	Tabelle mit entferntesten Punkten	kmeans_ farest_2_0_1
TABLE_DATABASE	STRING	X	Tabelle mit Hilfsinformationen zur Datenbasis	kmeans_ database_2_0_1
TABLE_CLUSTER	STRING	X	Tabelle mit Hilfsinformationen zu Clustern	kmeans_ cluster_2_0_1
CATEGORICCOLUMNS	STRING		Kategorische Dimensionen	"3"

Tabelle 35: Parameter des erweiterten DBSCAN-Algorithmus

Anmerkungen zu den Parametern:

- **COLUMNS:** Die einzelnen Dimensionen müssen alle mit **t.** beginnen
- **CATEGORICCOLUMNS:** Trennzeichen ist Semikolon

7 Testergebnisse

Dieses Kapitel beschäftigt sich mit dem Test und den daraus resultierenden Testergebnissen der beiden implementierten Algorithmen K-Means und DBSCAN. Nachfolgend werden die Testdaten, die Testergebnisse des K-Means Algorithmus und die Testergebnisse des DBSCAN Algorithmus näher erläutert.

7.1 Testdaten

Die Tests der beiden Algorithmen erfolgten mit synthetischen Testdaten. Die Testdaten wurden mit einem eigens dafür implementierten Testgenerator von Mathias Goller [GM04] erzeugt. Dieser Testgenerator erzeugt Testdaten, die dem Ergebnis eines K-Means Algorithmus gleichen, d.h. er erzeugt eine gewisse Anzahl von Clustern mit einer gewissen Anzahl von Punkten. Beim Aufruf des Testgenerators werden die Anzahl der zu generierenden Cluster, die gesamte Anzahl der zu erzeugenden Punkte und die Clustermittelpunkte (Centroide) übergeben. Des Weiteren kann die Anzahl der Punkte pro Cluster prozentuell und die Abweichung der Punkte vom Mittelpunkt bestimmt werden. Als Ergebnis erhält man ein Clusteringergebnis mit der gewünschten Anzahl von Clustern und Punkten, mit den übergebenen Centroiden. Das Ergebnis ist für den K-Means Algorithmus optimiert, d.h. die Cluster tendieren zu einer kreisförmigen Form. Zusätzlich zu den bereits erwähnten Parametern werden auch der Name der zu erzeugenden Tabelle und die Anzahl der Dimensionen angegeben. Es können numerische und kategorische Dimensionen erzeugt werden. Für die kategorischen Dimensionen muss allerdings ein eigenes XML-Konfigurationsfile übergeben werden, das steuert mit welcher Wahrscheinlichkeit ein bestimmter Wert eines Attributs der kategorischen Dimension angenommen wird. Die Punkte, die der Testgenerator erzeugt, sind normalverteilt.

Für die Tests der beiden Algorithmen wurde eine Tabelle mit 10 numerischen und 5 kategorischen Dimensionen erzeugt. Zusätzlich enthält die Tabelle noch eine Dimension für die Cluster-Id des zugehörigen Clusters, d.h. ursprünglich hat die Tabelle 16 Dimensionen. Die 5 kategorischen Attribute werden von den beiden Clusteringalgorithmen nicht verwendet, da beide nur mit numerischen Attributen arbeiten. Allerdings werden zu den kategorischen Attributen auch Hilfsinformationen berechnet falls diese vom Datentyp her numerisch sind. Die kategorischen Attribute werden aber hauptsächlich vom zweiten Schritt des „Kombinierten Data Mining“ der

Klassifikation verwendet (vergleiche [HM04]). Nachfolgend werden die Parameter für den beschriebenen Testgenerator zum Erzeugen der Testdaten angegeben:

- **Tabellenname:** CLUSTERINGTESTDATEN
- **Anzahl Cluster:** 6
- **Anzahl Punkte:** 5 Millionen (= 500 MB)
- **Anzahl Dimensionen:** 16 (10 numerische + 5 kategorische + Cluster-Id)
- **Mittelpunkte der Cluster (Centroide) + Abweichung vom Clustermittelpunkt + Anzahl der Punkte in Cluster:**

Cluster	Mittelpunkt	Abweichung	Anzahl Punkte
1	0,0,0,0,0,0,0,0,0,0	500	12,5 %
2	-1000,2500,-1500,0,0,1000,500,0,1500,-1000	400	25 %
3	-2500,0,3000,0,0,1700,-500,0,0,-500	400	12,5 %
4	1000,-1000,0,0,0,0,0,0,1200,0	500	12,5%
5	1500,-1000,-1200,0,0,-1000,0,0,-200,0	400	12,5%
6	1000,0,200,0,0,-300,0,0,0,0	100	25 %

Tabelle 36: Parameter für Aufruf des Testgenerators

Die erzeugte Testtabelle CLUSTERINGTESTDATEN ist in Oracle dann wie folgt aufgebaut:

Name	Typ	Beschreibung
X1	FLOAT(126)	Numerische Dimension 1
X2	FLOAT(126)	Numerische Dimension 2
X3	FLOAT(126)	Numerische Dimension 3

X4	FLOAT(126)	Numerische Dimension 4
X5	FLOAT(126)	Numerische Dimension 5
X6	FLOAT(126)	Numerische Dimension 6
X7	FLOAT(126)	Numerische Dimension 7
X8	FLOAT(126)	Numerische Dimension 8
X9	FLOAT(126)	Numerische Dimension 9
X10	FLOAT(126)	Numerische Dimension 10
K1	NUMBER(2)	Kategorische Dimension 1
K2	NUMBER(2)	Kategorische Dimension 2
K3	NUMBER(2)	Kategorische Dimension 3
K4	NUMBER(2)	Kategorische Dimension 4
K5	NUMBER(2)	Kategorische Dimension 5
REAL_CLUSTER_ID	NUMBER(2)	Ermittelte Cluster ID des Testgenerators

Tabelle 37: Aufbau der Testtabelle CLUSTERINGTESTDATEN

Die durch den Testgenerator ermittelten Testdaten sind für den K-Means Algorithmus optimiert, da kreisförmige Cluster geliefert werden. Daher wird ein Großteil der Tests auch mit dem K-Means Algorithmus durchgeführt. Die nachfolgenden Kapitel zeigen die Vorgehensweise beim Testen der beiden Algorithmen und die Ergebnisse der durchgeführten Tests.

7.2 Testen des K-Means Algorithmus

Für den K-Means Algorithmus werden ein Laufzeittest und ein Genauigkeitstest mit einer festgelegten Anzahl von Testreihen durchgeführt. Die verwendeten Testreihen, und die dafür nötigen Parameter beim Aufruf des Algorithmus, werden im

nachfolgenden Abschnitt erklärt. Für den Laufzeittest gibt es zwei Varianten des K-Means Algorithmus, eine Variante mit Ermittlung der Hilfsinformationen und eine Variante ohne Ermittlung der Hilfsinformationen. Details zur Implementierung werden in Kapitel 6 erläutert, und der gesamte Quellcode ist im Anhang (siehe Kap. 12.1) zu finden. Durch den Vergleich der beiden Varianten kann festgestellt werden, um wie viel sich die Laufzeit durch die Ermittlung der Hilfsinformationen verlängert. Beim Genauigkeitstest wird das Ergebnis des Clustering des K-Means Algorithmus mit dem erzeugten Clustering des Testgenerators verglichen. Genaue Erläuterungen und Ergebnisse der Tests werden in den jeweiligen nachfolgenden Abschnitten zum Laufzeittest und zum Genauigkeitstest geliefert.

7.2.1 Testreihen

Der K-Means Algorithmus wird mit den erzeugten Testdaten der Tabelle CLUSTERINGTESTDATEN getestet. Zum Testen werden 5 Testreihen mit jeweils einer unterschiedlichen Anzahl von Dimensionen verwendet. Zu jeder Testreihe gibt es jeweils 14 Testläufe mit einer unterschiedlich großen Datenmenge. Die ersten 10 Testreihen werden in 100.000er Schritten beginnend mit 100.000 (=10 MB) Punkten bis zu 1 Million Punkte (=100 MB) durchgeführt. Die letzten vier Testreihen verwenden 2, 3, 4 und 5 Millionen Punkte. Eine Unterscheidung der Anzahl der Dimensionen erfolgt, da der Algorithmus mit wenigen, ein paar und vielen Dimensionen getestet werden soll. Zwei numerische Dimensionen entsprechen in diesem Fall wenigen Dimensionen. Diese Anzahl wurde vor allem deswegen gewählt, da zwei Dimensionen auch graphisch leicht darstellbar sind. Ein paar Dimensionen entsprechen hier fünf numerischen Dimensionen. Dazu gibt es drei Testreihen, die jeweils mit einer unterschiedlichen Zahl von kategorischen Dimensionen kombiniert werden. Diese werden vor allem deswegen verwendet, weil sie im zweiten Schritt des „Kombinierten Data Mining“, der Klassifikation, benötigt werden. Die fünfte Testreihe verwendet alle zehn numerischen Dimensionen der Testdaten, damit auch ein Test mit einer großen Anzahl von Dimensionen durchgeführt wird. Nachfolgend wird der Aufbau der Testreihen kurz erläutert:

- **Testreihe 1:** 2 numerische Dimensionen (X2, X3)
- **Testreihe 2:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 1 kategorische Dimension (K1)
- **Testreihe 3:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 2 kategorische Dimension (K1, K2)

- **Testreihe 4:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 5 kategorische Dimensionen (K1, K2, K3, K4, K5)
- **Testreihe 5:** 10 numerische Dimensionen (X1, X2, X3, X4, X5, X6, X7, X8, X9, X10)

Für jeden Testlauf einer Testreihe müssen die Parameter des K-Means Algorithmus (siehe Kap. 6.2.3) unterschiedlich befüllt werden. Nachfolgend werden die Parameter des K-Means Algorithmus angeführt die pro Testreihe verwendet werden. Auf die Unterschiede der Testläufe innerhalb einer Testreihe wird anschließend eingegangen.

Parameter	Testreihe 1	Testreihe 2	Testreihe 3	Testreihe 4	Testreihe 5
USER	stoett	stoett	stoett	stoett	Stoett
PASSWORD	*****	*****	*****	*****	*****
SERVICE	dke3	dke3	dke3	dke3	dke3
TABLE	clustering-testdaten	clustering-testdaten	clustering-testdaten	clustering-testdaten	Clustering-testdaten
COLUMNS	„t.X2, t.X3, t.estimate_ cluster_id“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.estimate clusterid“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.K2, t.estimate_ clusterid“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.K2, t.K3, t.K4, t.K5, t.estimate_ clusterid_**xxx**“	“t.X1, t.X2, t.X3, t.X4, t.X5, t.X6, t.X7, T.X8, t.X9, t.X10, t.estimate_ cluster_id”
NUMPOINTS	siehe oben	siehe oben	siehe oben	siehe oben	siehe oben
NUMCLUSTERS	6	3	3	6	6
MAXINITIALNUMPOINTS	2000	2000	2000	2000	2000
MININITIALNUMPOINTS	1000	1000	1000	1000	1000
CLUSTERID	3	7	8	11	11
NAMECLUSTERID	estimate_ cluster_id	estimate_ cluster_id	estimate_ cluster_id	estimate_ cluster_id_**xxx**	estimate_ cluster_id
DISTANCEFUNCTION	2	2	2	2	2
NEARESTNUMPOINTS	1000	1000	1000	1000	1000
FARESTNUMPOINTS	1000	1000	1000	1000	1000
FILEKMEANS	kmeans.txt	kmeans.txt	kmeans.txt	kmeans.txt	kmeans.txt
FILELOG	kmeans_log.txt	kmeans_log.txt	kmeans_log.txt	kmeans_log.txt	kmeans_log.txt
IDUNCLASSIFIED	-1	-1	-1	-1	-1
TABLENEAREST	kmeans_ nearest_2_0_**x**	kmeans_ nearest_5_1_**x**	kmeans_ nearest_5_2_**x**	kmeans_ nearest_5_5_**x**	kmeans_ nearest_10_0_**x**
TABLEFAREST	kmeans_ farest_2_0_**x**	kmeans_ farest_5_1_**x**	kmeans_ farest_5_2_**x**	kmeans_ farest_5_5_**x**	kmeans_ farest_10_0_**x**

TABLEDATABASE	kmeans_ database_2_0_**x**	kmeans_ database_5_1_**x**	kmeans_ database_5_2_**x**	kmeans_ database_5_5_**x**	kmeans_ database_10_0_**x**
TABLECLUSTER	kmeans_ cluster_2_0_**x**	kmeans_ cluster_5_1_**x**	kmeans_ cluster_5_2_**x**	kmeans_ cluster_5_5_**x**	kmeans_ cluster_10_0_**x**
NUMBERINITIAL-ITERATIONS	40	40	40	40	40
NUMBERITERATIONS	30	30	30	30	30
INITIALPOINTS	„0;0;0;0;0;0“	„0;0;0“	„0;0;0“	„0;0;0;0;0;0“	„0;0;0;0;0;0“
CATEGORICCOLUMNS	„“	„6“	„6;7“	„6;7;8;9;10“	„“

Tabelle 38: K-Means: Parameter der Testläufe der 5 Testreihen

Eine genaue Beschreibung der Parameter liefert Kapitel 6.2.3. Nachfolgend werden zu gewissen Parametern noch Anmerkungen getätigt:

- **NUMPOINTS:** Die Anzahl der Punkte ist pro Testlauf einer Testreihe verschieden (siehe oben)
- **TABLENEAREST:** Das fett markierte x im Tabellenname steht für den Testlauf der Testreihe, d.h. der erste Testlauf hat Tabellenname kmeans_nearest_2_0_1 und der letzte Testlauf hat kmeans_nearest_2_0_14
- **TABLEFAREST, TABLEDATABASE, TABLECLUSTER:** siehe TABLENEAREST

Für Testreihe 4 wird für jeden Testlauf eine eigene Dimension (Spalte) für die Cluster-Id in der Tabelle CLUSTERINGTESTDATEN angelegt. Somit wird das gesamte Clusterergebnis für alle Testläufe der Testreihe 4 gespeichert. Dies wird für den Genauigkeitstest benötigt. Des Weiteren wird der 2 Schritt des „Kombinierten Data Mining“, nämlich die Klassifikation, basierend auf diesen Daten testen und evaluieren. Daher sind in Testreihe 4 die Parameter COLUMNS und NAMECLUSTERID für jeden Testlauf unterschiedlich.

7.2.2 Laufzeittest

Der Laufzeittest soll zeigen wie sehr sich die Ermittlung der Hilfsinformationen während des K-Means Algorithmus auf die Laufzeit auswirkt. Dazu gibt es zwei Varianten des K-Means Algorithmus, eine mit Ermittlung der Hilfsinformationen und eine ohne Ermittlung der Hilfsinformationen. Pro Testlauf einer jeden Testreihe wird einmal der K-Means mit Hilfsinformationen und einmal der K-Means ohne

Hilfsinformationen ausgeführt. Zuerst wird immer der K-Means Algorithmus mit Hilfsinformationen ausgeführt. Dieser schreibt die zufällig ermittelten Centroide der Initialisierung in das Protokollfile (siehe Parameter FILELOG). Beim Aufruf des K-Means Algorithmus ohne Hilfsinformationen werden diese initialen Centroide über den Parameter INITIALPOINTS an den Algorithmus übergeben. Dadurch ist gewährleistet, dass die beiden Algorithmen das selbe Clusterergebnis liefern, nur einmal mit und einmal ohne Hilfsinformationen. Somit kann die Laufzeit verglichen werden, und man sieht den zusätzlichen Aufwand für die Ermittlung der Hilfsinformationen. Dieser zusätzliche Aufwand wird in den nachfolgenden Testergebnissen als *Mehraufwand* bezeichnet. Dieser Mehraufwand gibt prozentuell die mehr benötigte Laufzeit des K-Means Algorithmus mit Hilfsinformationen zum K-Means Algorithmus ohne Hilfsinformationen an. Die Implementierung des K-Means Algorithmus befindet sich im Anhang (siehe Kap. 12.1). Für den K-Means Algorithmus ohne Hilfsinformationen wird die Ermittlung der Hilfsinformationen aus der Implementierung herausgenommen. Die Implementierung der Hilfsinformationen ist in Kapitel 6.2.2 beschrieben. Nachfolgend werden die Ergebnisse des Laufzeittests pro Testreihe in Form einer Tabelle und graphisch aufbereitet dargestellt:

Testreihe 1: 6 Cluster – 2 numerische Dimensionen

Die im Folgenden angeführte Tabelle 39 zeigt das Ergebnis der 14 Testläufe der Testreihe 1. Dabei werden die Anzahl der Iterationen und die Laufzeiten der beiden Algorithmen (mit und ohne Hilfsinformationen) angegeben. Zusätzlich wird der Mehraufwand des Algorithmus mit Hilfsinformationen im Vergleich zum Algorithmus ohne Hilfsinformationen prozentuell angegeben.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Mit	Mehraufwand
1	100.000	4	655 s	710 s	8 %
2	200.000	3	676 s	780 s	15 %
3	300.000	7	2123 s	2144 s	1 %
4	400.000	3	1266 s	1500 s	18 %
5	500.000	4	1654 s	2064 s	25 %
6	600.000	6	2636 s	3296 s	25 %
7	700.000	6	2968 s	3740 s	26 %
8	800.000	5	3137 s	3720 s	19 %
9	900.000	5	3506 s	4284 s	22 %
10	1.000.000	2	2297 s	2928 s	27 %
11	2.000.000	6	8492 s	10352 s	22 %
12	3.000.000	2	6209 s	6999 s	13 %
13	4.000.000	5	13745 s	16824 s	22 %
14	5.000.000	5	18925 s	22165 s	17 %
			Durchschnittlicher Mehraufwand:		**19** %

Tabelle 39: K-Means: Testergebnisse der 14 Testläufe der Testreihe 1

Durchschnittlich beträgt die Laufzeit des K-Means Algorithmus mit Hilfsinformationen um 19 % mehr als die Laufzeit des K-Means Algorithmus ohne Hilfsinformationen. Abbildung 25 stellt die Laufzeiten der Testläufe graphisch gegenüber:

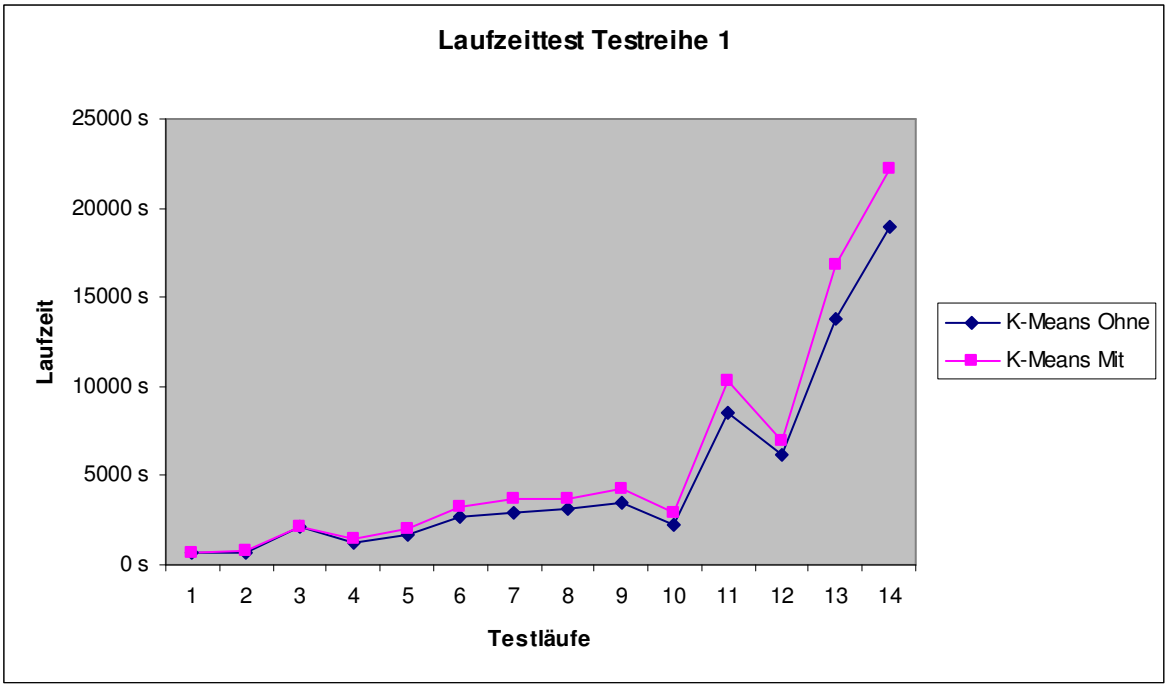


Abbildung 25: Laufzeitvergleich der beiden Algorithmen für Testreihe 1

Testreihe 2: 3 Cluster – 5 numerische Dimensionen + 1 kategorische Dimension

Diese Testreihe wird nicht mit 6 Clustern sondern mit 3 Clustern ausgeführt. Dies begründet sich darin, dass der Vollständigkeit halber beim Testen auch mit dem Parameter k, der Anzahl der Cluster, variiert wurde. Dabei stellte sich heraus, dass der Algorithmus mit 3 Clustern ein gutes Ergebnis liefert. Dies ist damit zu erklären, dass 4 der erzeugten 6 Cluster relativ dicht beieinander liegen und diese dadurch leicht zu einem verschmelzen. Die im Folgenden angeführte Tabelle 40 zeigt das Ergebnis der 14 Testläufe der Testreihe 2. Dabei werden die Anzahl der Iterationen und die Laufzeiten der beiden Algorithmen (mit und ohne Hilfsinformationen) angegeben. Zusätzlich wird der Mehraufwand des Algorithmus mit Hilfsinformationen im Vergleich zum Algorithmus ohne Hilfsinformationen prozentuell angegeben.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Mit	Mehraufwand
1	100.000	2	251 s	328 s	31 %
2	200.000	2	457 s	650 s	42 %
3	300.000	2	716 s	911 s	27 %
4	400.000	2	911 s	1166 s	28 %
5	500.000	4	1768 s	2490 s	41 %
6	600.000	2	1408 s	1706 s	21 %
7	700.000	2	1610 s	1935 s	20 %
8	800.000	2	1776 s	2187 s	23 %
9	900.000	2	2013 s	2444 s	21 %
10	1.000.000	2	2209 s	2691 s	22 %
11	2.000.000	2	4320 s	5174 s	20 %
12	3.000.000	2	6445 s	7752 s	20 %
13	4.000.000	2	8672 s	10453 s	21 %
14	5.000.000	2	12675 s	14750 s	16 %
			Durchschnittlicher Mehraufwand:		**25** %

Tabelle 40: K-Means: Testergebnisse der 14 Testläufe für Testreihe 2

Die Laufzeit des K-Means Algorithmus mit Hilfsinformationen beträgt für Testreihe 2 im Durchschnitt um 25 % mehr, als für den K-Means Algorithmus ohne Hilfsinformationen. Abbildung 26 stellt die Laufzeiten der Testläufe graphisch gegenüber:

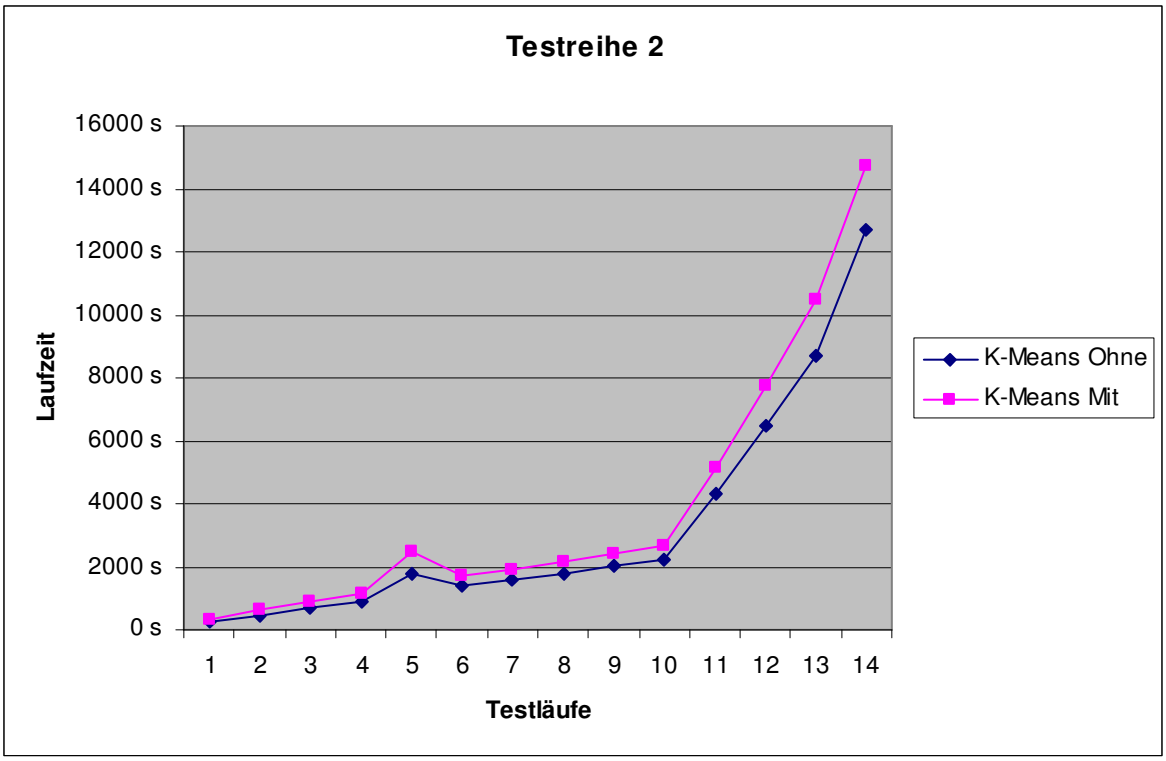


Abbildung 26: Laufzeitvergleich der beiden Algorithmen für Testreihe 2

Testreihe 3: 3 Cluster – 5 numerische + 2 kategorische Dimensionen

Diese Testreihe wird wie Testreihe 2 mit 3 Clustern ausgeführt. Die im Folgenden angeführte Tabelle 41 zeigt das Ergebnis der 14 Testläufe der Testreihe 2. Dabei werden die Anzahl der Iterationen und die Laufzeiten der beiden Algorithmen (mit und ohne Hilfsinformationen) angegeben. Zusätzlich wird der Mehraufwand des Algorithmus mit Hilfsinformationen im Vergleich zum Algorithmus ohne Hilfsinformationen prozentuell angegeben.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Mit	Mehraufwand
1	100.000	2	558 s	687 s	23 %
2	200.000	2	462 s	607 s	31 %
3	300.000	2	977 s	1056 s	8 %
4	400.000	2	897 s	1303 s	45 %
5	500.000	2	1155 s	1451 s	26 %
6	600.000	2	1359 s	1747 s	29 %
7	700.000	2	1576 s	1996 s	27 %
8	800.000	2	1802 s	2313 s	28 %
9	900.000	2	1997 s	2586 s	29 %
10	1.000.000	2	2213 s	2820 s	27 %
11	2.000.000	2	4340 s	5462 s	26 %
12	3.000.000	2	6505 s	8040 s	24 %
13	4.000.000	2	8737 s	10896 s	25 %
14	5.000.000	2	12677 s	15091 s	19 %
			Durchschnittlicher Mehraufwand:		**26** %

Tabelle 41: K-Means: Testergebnisse der 14 Testläufe für Testreihe 3

Durchschnittlich beträgt die Laufzeit des K-Means Algorithmus mit Hilfsinformationen um 26 % mehr als die des K-Means Algorithmus ohne Hilfsinformationen. Abbildung 27 stellt die Laufzeiten der Testläufe graphisch gegenüber:

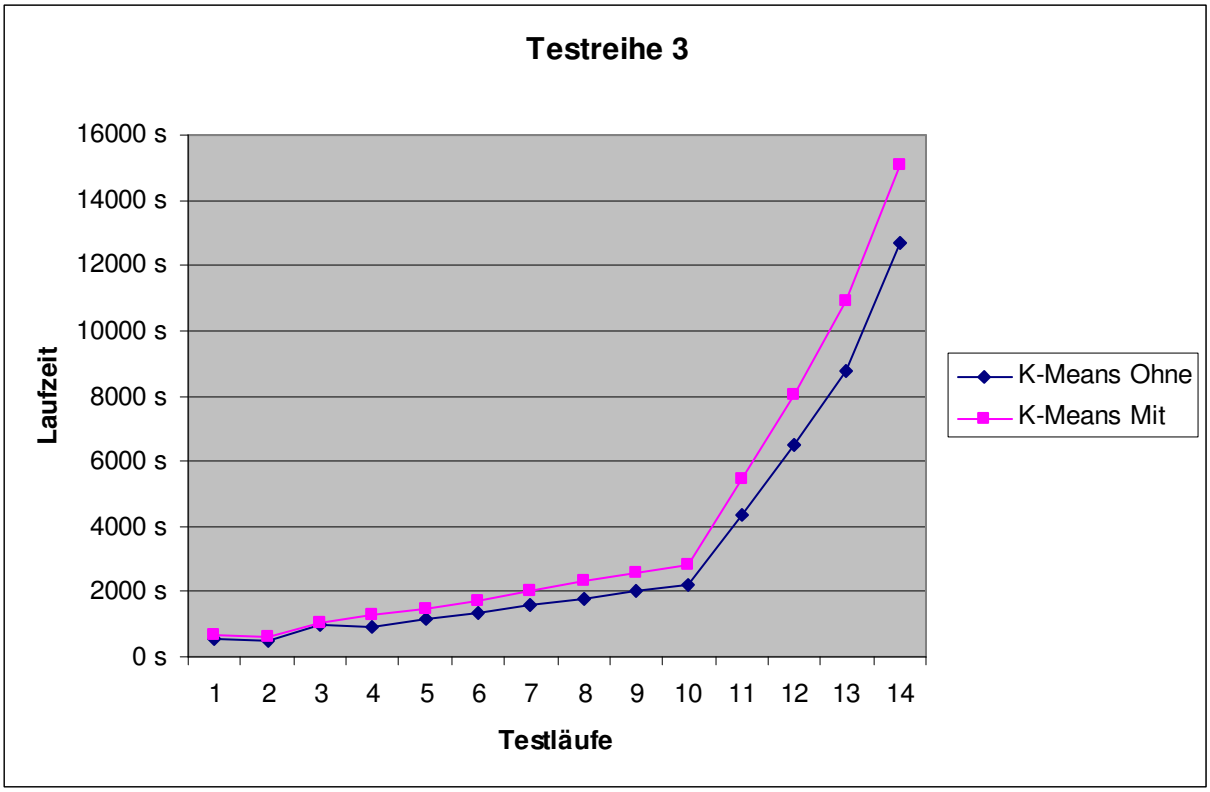


Abbildung 27: Laufzeitvergleich der beiden Algorithmen für Testreihe 3

Testreihe 4: 6 Cluster – 5 numerische + 5 kategorische Attribute

Nachfolgend werden die Ergebnisse der Testreihe 4 dargestellt. Tabelle 42 gibt die wichtigen Informationen, wie Anzahl der Iterationen, Laufzeiten und den Mehraufwand an:

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Mit	Mehraufwand
1	100.000	2	348 s	418 s	20 %
2	200.000	3	777 s	1170 s	51 %
3	300.000	2	698 s	974 s	40 %
4	400.000	3	1296 s	2021 s	56 %
5	500.000	2	1226 s	1751 s	43 %
6	600.000	2	1406 s	1971 s	40 %
7	700.000	2	1621 s	2252 s	39 %
8	800.000	19	9868 s	20095 s	104 %
9	900.000	3	2625 s	4170 s	59 %
10	1.000.000	2	2267 s	3132 s	38 %
11	2.000.000	2	4392 s	5978 s	36 %
12	3.000.000	16	33168 s	59514 s	79 %
13	4.000.000	3	10945 s	16698 s	53 %
14	5.000.000	2	12841 s	16470 s	28 %
			Durchschnittlicher Mehraufwand:		**1,49**

Tabelle 42: K-Means: Testergebnisse der 14 Testläufe für Testreihe 4

Die Laufzeit des K-Means Algorithmus mit Hilfsinformationen beträgt im Durchschnitt um 25 % mehr, als die Laufzeit des K-Means Algorithmus ohne Hilfsinformationen. Abbildung 28 stellt die Laufzeiten der Testläufe graphisch gegenüber:

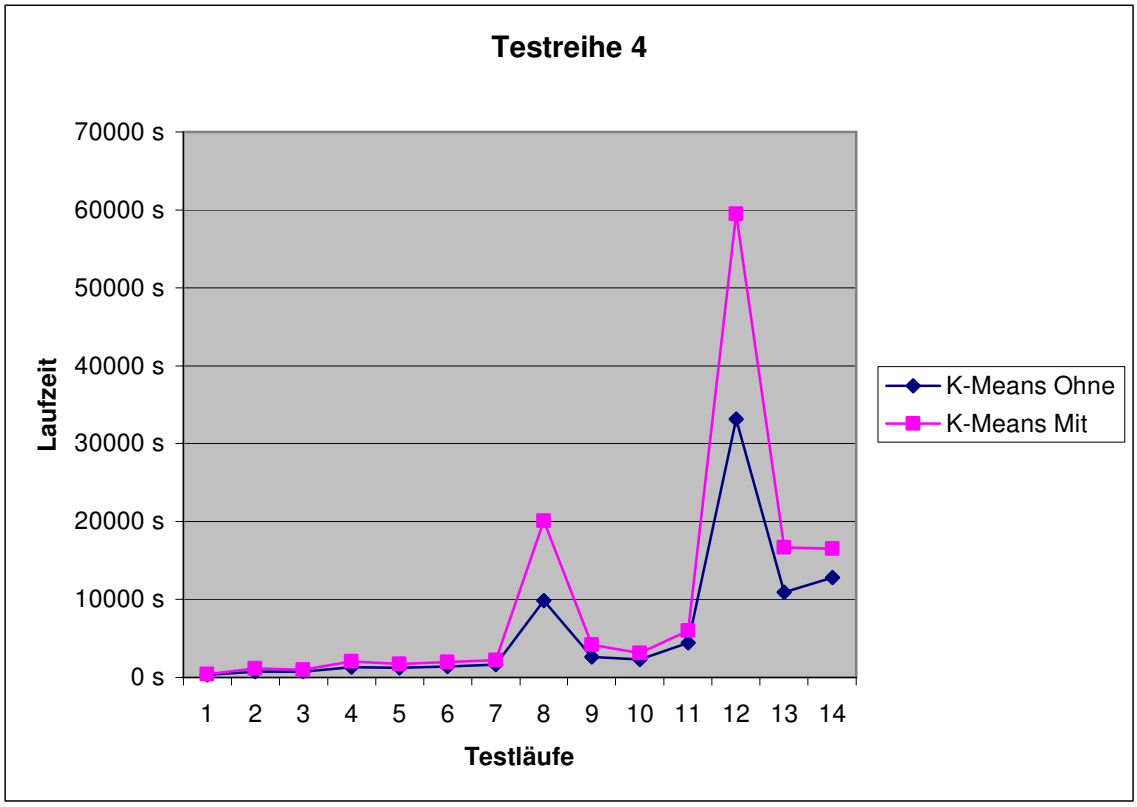


Abbildung 28: Laufzeitvergleich der beiden Algorithmen für Testreihe 4

Testreihe 5: 6 Cluster – 10 numerische Attribute

Testreihe 5 wird mit 6 Clustern und 10 numerischen Attributen ausgeführt. Nachfolgend zeigt Tabelle 43 die Anzahl der Iterationen und die Laufzeiten der beiden Algorithmen (mit und ohne Hilfsinformationen) angegeben. Zusätzlich wird der Mehraufwand des Algorithmus mit Hilfsinformationen im Vergleich zum Algorithmus ohne Hilfsinformationen prozentuell angegeben.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Mit	Mehraufwand
1	100.000	2	633 s	798 s	26 %
2	200.000	16	2833 s	5137 s	81 %
3	300.000	30	6549 s	12965 s	98 %
4	400.000	5	1856 s	2098 s	13 %
5	500.000	19	6672 s	12863 s	93 %
6	600.000	2	1427 s	1993 s	40 %
7	700.000	2	1691 s	2306 s	36 %
8	800.000	2	1925 s	2637 s	37 %
9	900.000	2	2090 s	2939 s	41 %
10	1.000.000	26	21300 s	33897 s	59 %
11	2.000.000	2	4500 s	6016 s	34 %
12	3.000.000	2	6650 s	8893 s	34 %
13	4.000.000	2	9123 s	11951 s	31 %
14	5.000.000	3	15790 s	22498 s	42 %
			Durchschnittlicher Mehraufwand:		**47** %

Tabelle 43: K-Means: Testergebnisse der 14 Testläufe für Testreihe 5

Durchschnittlich beträgt die Laufzeit des K-Means Algorithmus mit Hilfsinformationen für Testreihe um 47 % mehr als die Laufzeit des K-Means Algorithmus ohne Hilfsinformationen. Abbildung 29 stellt die Laufzeiten der Testläufe graphisch gegenüber:

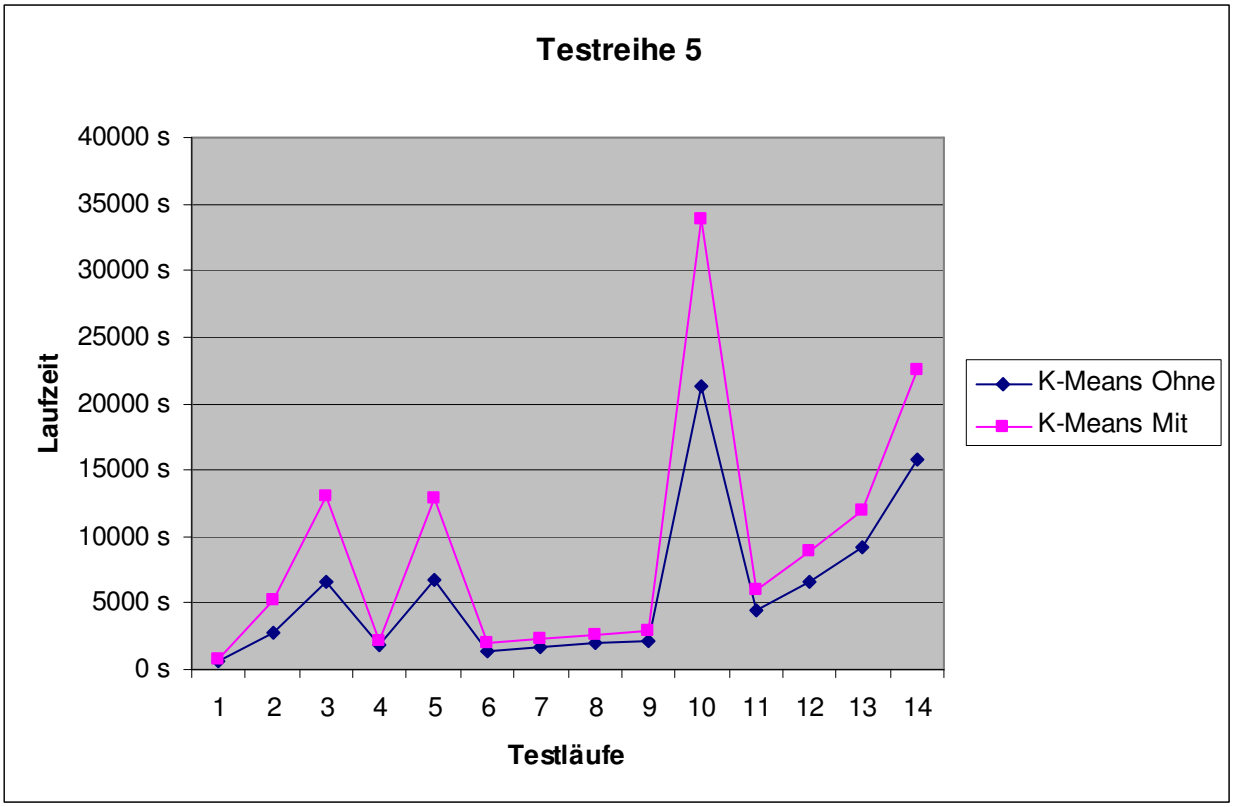


Abbildung 29: Laufzeitvergleich der beiden Algorithmen für Testreihe 5

Diskussion der Testergebnisse

Wenn man die Ergebnisse der 5 Testreihen zusammenfasst bedeutet dies, dass der K-Means Algorithmus mit Hilfsinformationen durchschnittlich einen Mehraufwand von 19 % bis 47 % im Vergleich zum K-Means Algorithmus ohne Hilfsinformationen hat. Dies zeigt, dass der Mehraufwand für die Ermittlung der Hilfsinformationen relativ konstant ist. Die Schwankungen lassen sich durch die unterschiedliche Anzahl der Dimensionen und der Iterationen erklären. Dieser Mehraufwand ist auf alle Fälle vertretbar. Das Ergebnis der Laufzeitvergleiche der 5 Testreihen wird im Balkendiagramm der Abbildung 30 noch einmal graphisch dargestellt.

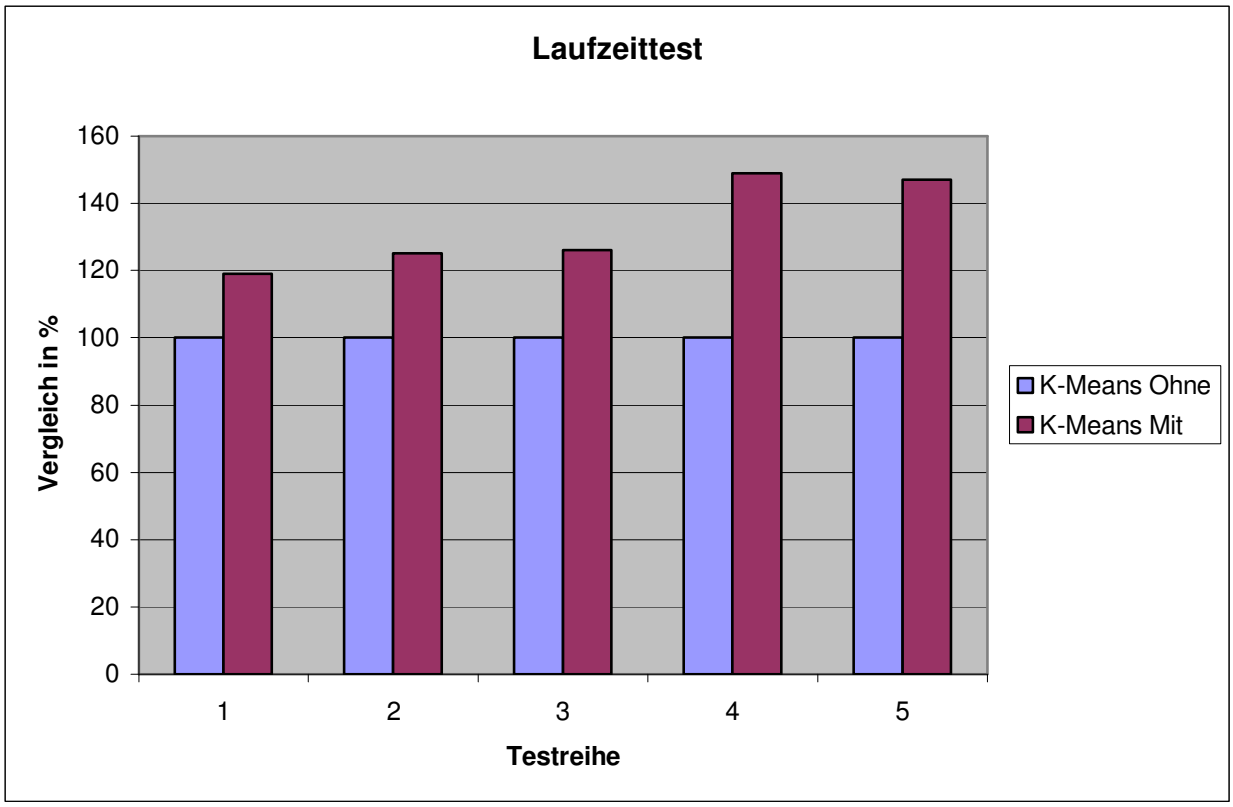


Abbildung 30: Ergebnis der Laufzeittest der 5 Testreihen

Nachdem die Ermittlung der Hilfsinformationen zusätzlich zum „normalen“ K-Means Algorithmus ausgeführt wurde, war zu erwarten, dass der K-Means Algorithmus mit Hilfsinformationen mehr Laufzeit benötigt als der K-Means Algorithmus ohne Hilfsinformationen. Dass der Mehraufwand des K-Means Algorithmus mit Hilfsinformationen variiert, von 19 % bis 47 %, ist vor allem von der Anzahl der Iterationen des K-Means Algorithmus und der Anzahl der verwendeten Dimensionen abhängig. Da ein Teil der Hilfsinformation in jeder Iteration neu berechnet wird (siehe Kap. 6.2.2) steigt der Aufwand mit der Anzahl der Iterationen. Ähnliches gilt bei einer

großen Anzahl von Dimensionen, da für jede Dimension die arithmetischen Hilfsinformationen berechnet werden. Dies gilt sowohl für die Hilfsinformationen zur Datenbasis als auch für die Hilfsinformationen zu den gefundenen Clustern.

Ein Großteil des Mehraufwands durch die Berechnung der Hilfsinformationen entfällt auf die Ermittlung der nähesten und entferntesten Punkte und deren zugehörige Containeroperationen. Diese Punkte werden mit Hilfe der Standardcontainerklasse *Vector* in C++ abgebildet. Da die Punkte innerhalb dieser Vektoren sortiert sein müssen, und immer nur eine gewisse Anzahl von nähesten und entferntesten Punkten gespeichert werden, sind pro Iteration mehrere Einfüge- und Löschoperationen auf die Vektoren nötig. Diese benötigen einen Großteil des Mehraufwands. Die Programmiersprache C++ stellt verschiedene Containerklassen zur Verfügung. Den Anforderungen der nähesten und entferntesten Punkte entsprachen die Containerklassen *Deque*, *Vector* und *List*. Um festzustellen welche der drei Containerklassen am Besten für die Ermittlung der nähesten und entferntesten Punkte geeignet ist, wurde für jede dieser Klassen eine eigene Lösung implementiert. Alle drei Implementierungen wurden mit der Testreihe 1 getestet. Nachfolgend werden die Tests der Laufzeiten der unterschiedlichen Implementierungen gegenübergestellt.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Vector	K-Means Deque	K-Means List
1	100.000	4	655 s	710 s	918 s	720 s
2	200.000	3	676 s	780 s	1089 s	840 s
3	300.000	7	2123 s	2144 s	3132 s	2516 s
4	400.000	3	1266 s	1500 s	1804 s	1611 s
5	500.000	4	1654 s	2064 s	2707 s	2527 s
6	600.000	6	2636 s	3296 s	4530 s	4082 s
7	700.000	6	2968 s	3740 s	5105 s	3895 s
8	800.000	5	3137 s	3720 s	4764 s	4449 s
9	900.000	5	3506 s	4284 s	4863 s	4942 s
10	1.000.000	2	2297 s	2928 s	2669 s	3598 s
11	2.000.000	6	8492 s	10352 s	10556 s	12271 s
12	3.000.000	2	6209 s	6999 s	7060 s	10200 s
13	4.000.000	5	13745 s	16824 s	16660 s	20572 s
14	5.000.000	5	18925 s	22165 s	22677 s	27626 s
		Summe:	**68289 s**	**81506 s**	**88534 s**	**99773 s**

Tabelle 44: K-Means: Laufzeitvergleich der Implementierungen mit verschiedenen Containerklassen

Das Ergebnis des Laufzeittests zeigt, dass der K-Means Algorithmus mit der Containerklasse *Vector* in allen 14 Testläufen schneller ist, als die Implementierungen mit den Containerklassen *List* und *Deque*. Aus diesem Grund wurde die endgültige Implementierung des K-Means Algorithmus mit Hilfsinformationen mit der

Containerklasse *Vector* durchgeführt, da diese im Vergleich zu den anderen Implementierungen eindeutig schneller ist.

Bei der Ermittlung der Hilfsinformationen wurden auch zwei unterschiedliche Möglichkeiten implementiert. Die erste Möglichkeit ist unter 6.2.2 beschrieben und genau erläutert. Dabei werden die Hilfsinformationen während den Iterationen des K-Means Algorithmus berechnet. Eine zweite Möglichkeit besteht darin sämtliche Hilfsinformationen in einer zusätzlichen Iteration nach dem K-Means Algorithmus zu ermitteln, d.h. alle Hilfsinformationen, sowohl zur Datenbasis als auch zu den gefundenen Clustern, werden in einer eigenen Iteration ermittelt. Beide Möglichkeiten wurden implementiert und mit der Testreihe 1 getestet. Das Ergebnis dieses Laufzeittest kann nachfolgend betrachtet werden.

Testlauf	Punkte	Iterationen	K-Means Ohne	K-Means Vector	K-Means + Iteration
1	100.000	4	655 s	710 s	793 s
2	200.000	3	676 s	780 s	940 s
3	300.000	7	2123 s	2144 s	2227 s
4	400.000	3	1266 s	1500 s	1604 s
5	500.000	4	1654 s	2064 s	2266 s
6	600.000	6	2636 s	3296 s	3535 s
7	700.000	6	2968 s	3740 s	3961 s
8	800.000	5	3137 s	3720 s	3975 s
9	900.000	5	3506 s	4284 s	4607 s
10	1.000.000	2	2297 s	2928 s	3084 s
11	2.000.000	6	8492 s	10352 s	10713 s
12	3.000.000	2	6209 s	6999 s	8441 s
13	4.000.000	5	13745 s	16824 s	17900 s
14	5.000.000	5	18925 s	22165 s	25387 s
		Summe:	**68289 s**	**81506 s**	**89433 s**

Tabelle 45: Laufzeitvergleich für 2 unterschiedliche Implementierungen zur Ermittlung der Hilfsinformationen

Der K-Means Algorithmus mit der 1 Möglichkeit entspricht dem K-Means Vector und der K-Means Algorithmus mit der 2 Möglichkeit entspricht dem K-Means + Iteration. Der Laufzeitvergleich zeigt, dass die 1 Möglichkeit in jedem Testlauf schneller war als die 2 Möglichkeit. Daher wurde auch der K-Means Algorithmus mit der Ermittlung der Hilfsinformationen wie unter 6.2.2 beschrieben implementiert.

Der Laufzeittest hat den Mehraufwand des K-Means Algorithmus mit Hilfsinformationen gegenüber dem K-Means Algorithmus ohne Hilfsinformationen gezeigt. Anschließend soll der Genauigkeitstest zeigen, in wie weit das Ergebnis des K-Means dem erzeugten Clustering des Testgenerators entspricht.

7.2.3 Genauigkeitstest

Der Genauigkeitstest vergleicht das Ergebnis des K-Means Algorithmus mit dem erzeugten Clustering des Testgenerators. Dieser Genauigkeitstest ist unabhängig von der Ermittlung der Hilfsinformationen und vergleicht nur das Ergebnis des implementierten K-Means Algorithmus mit dem Clustering des Testgenerators. Dieser Vergleich ist möglich, da beim Erzeugen der Testdaten durch den Testgenerator jeder Punkt einem Cluster zugeteilt wurde, und die zugehörige Cluster-Id in den Testdaten gespeichert wurde (Dimension: real_cluster_id). Bei den 14 Testläufen der Testreihe 4 wurde das Ergebnis für jeden Testlauf in eine eigene Dimension (=Tabellenspalte) der Tabelle CLUSTERINGTESTDATEN geschrieben (siehe auch Kap. 7.2.1). Dadurch können die Ergebnisse relativ einfach mit einem SELECT-Statment in Oracle anhand ihres Mittelpunkts und der Anzahl der Punkte pro Cluster verglichen werden. Beim Vergleich der Mittelpunkte werden nur die numerischen Dimensionen herangezogen, dies entspricht bei der Testreihe 4 einer Anzahl von 5 Dimensionen (siehe Kap. 7.2.1).

Die Mittelpunkte werden nachfolgend pro Cluster gegenübergestellt. Zusätzlich zu den Mittelpunkten wird die Differenz in den einzelnen Dimensionen der Mittelpunkte gezeigt. Diese dienen auch als Maß für die Genauigkeit. Folgende Kategorien für die Genauigkeit werden unterschieden:

- **identisch:** entspricht einer Abweichung von 0 bis 10 Längeneinheiten
- **sehr genau:** entspricht einer Abweichung von 10 bis 50 Längeneinheiten
- **genau:** entspricht einer Abweichung von 50 bis 100 Längeneinheiten
- **ungenau:** Abweichung größer als 100 Längeneinheiten

Diese Kategorien wurden vom Autor dieser Arbeit erstellt und lassen sich nur im Zusammenhang mit den erzeugten Testdaten anwenden. Diese gebildeten Kategorien werden nachfolgend farblich in Form einer Hintergrundfarbe dargestellt. Folgende Hintergrundfarbe entspricht folgender Kategorie:

- identisch
- sehr genau
- genau
- ungenau

Nachfolgend werden die Mittelpunkte der Cluster anhand von 4 Testläufen aus der Testreihe 4 verglichen.

Testlauf 2: 200.000 Punkte

Die Mittelpunkte der Cluster werden nachfolgend pro Cluster verglichen. Dabei wird angegeben welcher Cluster des Testgenerators welchem Cluster des K-Means entspricht. Die Genauigkeit wird anhand der Hintergrundfarben der Differenz bestimmt (siehe oben):

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	1	5,22	7,52	-0,75	9,88	-4,19	24560
K-Means	3	-159,47	7,73	-31,31	59,52	-7,35	19789
	Differenz:	164,69	-0,22	30,56	-49,64	3,16	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	2	-999,82	2499,85	-1496,81	1000,52	1498,47	50484
K-Means	1	-999,82	2499,76	-1496,77	1000,53	1498,42	50487
	Differenz:	0,01	0,08	-0,05	-0,01	0,05	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	3	-2493,15	-1,18	2995,96	1702,58	2,42	24911
K-Means	5	-2493,15	-1,18	2995,96	1702,58	2,42	24911
	Differenz:	0,00	0,00	0,00	0,00	0,00	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	4	1001,54	-1000,31	5,16	-6,98	1199,96	25171
K-Means	6	995,36	-1034,61	7,52	10,89	1252,67	23666
	Differenz:	6,19	34,30	-2,36	-17,87	-52,71	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	5	1503,24	-999,62	-1202,14	-999,61	-204,83	25143
K-Means	2	1499,51	-1003,13	-1201,21	-995,87	-197,57	25394
	Differenz:	3,73	3,51	-0,93	-3,74	-7,26	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	6	998,92	0,27	200,54	-300,01	0,40	49731
K-Means	4	974,58	-5,60	193,82	-289,41	8,72	55753
	Differenz:	24,34	5,86	6,72	-10,59	-8,32	

Testlauf 8: 800.000 Punkte

Nachfolgend werden die Mittelpunkte dieses Testlaufs mit den Mittelpunkten des Testgenerators in den einzelnen Dimensionen verglichen. Die Genauigkeit kann anhand der Hintergrundfarben der Differenz erkannt werden (siehe oben):

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	1	7,32	2,91	-4,52	1,92	-1,34	84344
K-Means	3	-156,39	3,03	-32,91	52,02	-0,47	68168
		163,70	-0,12	28,39	-50,11	-0,87	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	2	-996,53	2500,53	-1500,50	999,24	1499,08	170213
K-Means	2	-996,53	2500,43	-1500,46	999,25	1499,03	170224
		0,00	0,09	-0,05	-0,01	0,05	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	3	-2495,24	-1,55	2997,80	1702,00	1,24	89231
K-Means	5	-2495,24	-1,55	2997,80	1702,00	1,24	89231
	Differenz:	0,00	0,00	0,00	0,00	0,00	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	4	1007,82	-999,97	-5,13	-8,31	1200,48	88463
K-Means	1	1004,46	-1034,95	-4,12	9,71	1253,42	83082
		3,36	34,98	-1,00	-18,02	-52,94	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	5	1500,49	-998,93	-1200,66	-1001,73	-203,37	90365
K-Means	4	1496,63	-1002,42	-1200,84	-998,42	-197,22	91159
		3,86	3,49	0,18	-3,30	-6,15	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	6	1000,19	0,14	199,70	-300,50	0,51	177384
K-Means	6	976,95	-6,62	192,56	-291,46	8,40	198136
		23,24	6,76	7,14	-9,04	-7,89	

Testlauf 11: 2.000.000 Punkte

Auch für diesen Testlauf werden die Ergebnisse des K-Means mit denen des Testgenerators mit Hilfe der Mittelpunkte verglichen. Die Genauigkeit wird anhand der Hintergrundfarben der Differenz in den Dimensionen bestimmt (siehe oben):

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	1	6,24	2,68	-3,29	3,39	-1,76	243596
K-Means	6	-157,45	2,11	-32,00	52,98	-1,32	196892
	Differenz:	163,68	0,57	28,71	-49,59	-0,44	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	2	-996,96	2500,52	-1499,37	999,41	1500,05	495771
K-Means	4	-996,96	2500,43	-1499,32	999,42	1500,00	495803
	Differenz:	0,00	0,09	-0,05	-0,01	0,05	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	3	-2493,49	-1,65	2998,05	1701,82	1,15	251913
K-Means	2	-2493,49	-1,65	2998,05	1701,82	1,15	251913
	Differenz:	0,00	0,00	0,00	0,00	0,00	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	4	1004,64	-997,75	-2,34	-7,66	1200,53	251807
K-Means	1	1000,81	-1032,50	-0,73	11,19	1254,15	236395
	Differenz:	3,84	34,75	-1,61	-18,85	-53,62	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	5	1500,71	-999,49	-1201,84	-1000,83	-203,18	255616
K-Means	5	1496,74	-1002,96	-1201,64	-997,21	-196,55	258056
	Differenz:	3,97	3,48	-0,20	-3,62	-6,63	

	Cluster-Id	X1	X2	X3	X6	X9	Anzahl Punkte
Testgenerator	6	999,75	0,07	199,95	-300,47	0,65	501297
K-Means	3	976,00	-6,40	192,93	-291,18	8,45	560941
	Differenz:	23,76	6,47	7,02	-9,29	-7,81	

Testlauf 14: 5.000.000 Punkte

Die Mittelpunkte der Cluster werden nachfolgend pro Cluster in den einzelnen Dimensionen verglichen. Die Genauigkeit wird anhand der Hintergrundfarben der Differenz in den einzelnen Dimensionen bestimmt (siehe oben):

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	1	3,85	2,24	-1,67	4,35	-1,54	617011
K-Means	6	-159,12	1,91	-30,64	53,22	-1,36	498928
	Differenz:	162,97	0,33	28,97	-48,87	-0,19	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	2	-997,79	2500,46	-1498,85	1000,07	1500,16	1258341
K-Means	1	-997,79	2500,35	-1498,78	1000,08	1500,11	1258437
	Differenz:	0,00	0,11	-0,07	-0,01	0,05	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	3	-2493,17	-1,57	2998,01	1701,88	0,63	622749
K-Means	5	-2493,17	-1,57	2998,01	1701,88	0,63	622749
	Differenz:	0,00	0,00	0,00	0,00	0,00	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	4	1001,14	-997,43	-0,25	-5,67	1200,43	626843
K-Means	2	996,92	-1032,10	1,76	13,92	1254,54	588201
	Differenz:	4,23	34,67	-2,01	-19,60	-54,11	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	5	1500,36	-999,26	-1202,11	-1000,79	-203,20	632024
K-Means	4	1496,28	-1002,82	-1201,72	-997,03	-196,21	638349
	Differenz:	4,08	3,56	-0,39	-3,76	-7,00	

	Cluster-Id	**X1**	**X2**	**X3**	**X6**	**X9**	**Anzahl Punkte**
Testgenerator	6	999,34	0,12	200,21	-300,37	0,69	1243032
K-Means	3	974,90	-6,51	193,30	-290,79	8,52	1393336
	Differenz:	24,44	6,64	6,90	-9,59	-7,83	

Diskussion der Ergebnisse

Der Vergleich der Mittelpunkte der 4 Testläufe zeigt, dass ein Großteil der Dimensionen der Mittelpunkte identisch oder sehr genau übereinstimmt. Die geringen Abweichungen der Mittelpunkte und vor allem die Abweichung der Anzahl der Punkte lassen sich durch die Überschneidungen der erzeugten Cluster des Testgenerators erklären. Dadurch werden manche Punkte anders zugeordnet. Im Großen und Ganzen ist das Ergebnis des Genauigkeitstest zufrieden stellend, da die ermittelten Mittelpunkte fast genau den Mittelpunkten des Clusterings des Testgenerators entsprechen.

7.3 Testen des DBSCAN – Algorithmus

Für den DBSCAN Algorithmus wird lediglich ein Laufzeittest mit einer festgelegten Anzahl von Testläufen durchgeführt. Die verwendeten Testläufe und die dafür nötigen Parameter beim Aufruf des Algorithmus werden im nachfolgenden Abschnitt erklärt. Es gibt zwei Varianten des DBSCAN Algorithmus, eine mit Ermittlung der Hilfsinformationen und eine ohne Ermittlung der Hilfsinformationen. Jeder Testlauf wird mit beiden Varianten durchgeführt. Dadurch kann die Laufzeit verglichen werden und ermittelt werden, um wie viel sich die Laufzeit durch die Ermittlung der Hilfsinformationen verlängert. Details zur Implementierung werden in Kapitel 6 erläutert, und der gesamte Quellcode ist im Anhang (siehe Kap. 12.2) zu finden. Genaue Erläuterungen und Ergebnisse der Tests werden im nachfolgenden Abschnitt zum Laufzeittest geliefert.

7.3.1 Testläufe

Wie der K-Means Algorithmus wird auch der DBSCAN Algorithmus mit den Testdaten der Tabelle CLUSTERINGTESTDATEN getestet. Es wurde bereits erwähnt, dass die Testdaten vor allem für den Test des K-Means Algorithmus geeignet sind. Daher werden für den DBSCAN-Algorithmus im Vergleich zum K-Means Algorithmus nur 5 Testläufe durchgeführt. Jeder Testlauf wird mit 10.000 Punkten ausgeführt. Die einzelnen Testläufe unterscheiden sich in der Anzahl der Dimensionen. Diese Unterscheidung in der Zahl der Dimensionen erfolgt aus dem selben Grund wie beim K-Means Algorithmus. Es soll mit wenigen, ein paar und vielen Dimensionen getestet werden (siehe auch 7.1). Nachfolgend wird der Aufbau der Testläufe kurz erläutert:

- **Testlauf 1:** 2 numerische Dimensionen (X2, X3)

- **Testlauf 2:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 1 kategorische Dimension (K1)
- **Testlauf 3:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 2 kategorische Dimension (K1, K2)
- **Testlauf 4:** 5 numerische Dimensionen (X1, X2, X3, X6, X9) + 5 kategorische Dimensionen (K1, K2, K3, K4, K5)
- **Testlauf 5:** 10 numerische Dimensionen (X1, X2, X3, X4, X5, X6, X7, X8, X9, X10)

Für jeden Testlauf müssen die Parameter des DBSCAN Algorithmus (siehe Kap. 6.3.3) unterschiedlich befüllt werden. Nachfolgend werden die Parameter des DBSCAN Algorithmus angeführt die pro Testlauf verwendet werden.

Parameter	Testlauf 1	Testlauf 2	Testlauf 3	Testlauf 4	Testlauf 5
USER	stoett	stoett	stoett	stoett	stoett
PASSWORD	*****	*****	*****	*****	*****
SERVICE	dke3	dke3	dke3	dke3	dke3
TABLE	clustering-testdaten	clustering-testdaten	clustering-testdaten	clustering-testdaten	clustering-testdaten
COLUMNS	„t.X2, t.X3, t.dbscan_ estimate_ cluster_id_1“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.dbscan_ estimate_ clusterid_3“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.K2, t.dbscan_ estimate_ cluster_id_2“	„t.X1, t.X2, t.X3, t.X6, t.X9, t.K1, t.K2, t.K3, t.K4, t.K5, t.estimate_ cluster_id“	“t.X1, t.X2, t.X3, t.X4, t.X5, t.X6, t.X7, T.X8, t.X9, t.X10, t.estimate_ cluster_id”
CLUSTERPOINTS	10.000	10.000	10.000	10.000	10.000
MINPOINTS	4	10	10	10	20
EPSILON	300	500	450	600	900
CLUSTERID	3	7	8	11	11
NAMECLUSTERID	dbscan_ estimate_ cluster_id_1	dbscan_ estimate_ cluster_id_3	dbscan_ estimate_ cluster_id_2	estimate_ cluster_id	estimate_ cluster_id
DISTANCEFUNCTION	2	2	2	2	2
NEARESTNUMPOINTS	1000	1000	1000	1000	1000
FARESTNUMPOINTS	1000	1000	1000	1000	1000
FILEDBSCAN	dbscan.txt	dbscan.txt	dbscan.txt	dbscan.txt	dbscan.txt
FILELOG	dbscan_log.txt	dbscan_log.txt	dbscan_log.txt	dbscan_log.txt	dbscan_log.txt
IDUNCLASSIFIED	-1	-1	-1	-1	-1

TABLENEAREST	dbscan_ nearest_2_0_1	dbscan_ nearest_5_1_1	dbscan_ nearest_5_2_1	dbscan_ nearest_5_5_2	dbscan_ nearest_10_0_1
TABLEFAREST	dbscan_ farest_2_0_1	dbscan_ farest_5_1_1	dbscan_ farest_5_2_1	dbscan_ farest_5_5_2	dbscan_ farest_10_0_1
TABLEDATABASE	dbscan_ database_2_0_1	dbscan_ database_5_1_1	dbscan_ database_5_2_1	dbscan_ database_5_5_2	dbscan_ database_10_0_1
TABLECLUSTER	dbscan_ cluster_2_0_1	dbscan_ cluster_5_1_1	dbscan_ cluster_5_2_1	dbscan_ cluster_5_5_2	dbscan_ cluster_10_0_1
CATEGORICCOLUMNS	„“	„6“	„6;7“	„6;7;8;9;10“	„“

Tabelle 46: DBSCAN: Parameter der 5 Testläufe

7.3.2 Laufzeittest

Der Laufzeittest soll die Auswirkung der Ermittlung der Hilfsinformationen während des DBSCAN Algorithmus auf die Laufzeit zeigen. Dies soll mit Hilfe der zwei Varianten des DBSCN Algorithmus, mit und ohne Hilfsinformationen, erfolgen. Für jeden Testlauf werden beide Varianten mit den selben Parametern aufgerufen, wobei die Parameter für die Hilfsinformationen nur von der Variante mit Ermittlung der Hilfsinformationen benötigt werden. Mit den Ergebnissen der beiden Algorithmen kann die Laufzeit verglichen werden, und man sieht den zusätzlichen Aufwand für die Ermittlung der Hilfsinformationen. Dieser zusätzliche Aufwand wird in den nachfolgenden Testergebnissen als *Mehraufwand* bezeichnet. Dieser Mehraufwand gibt prozentuell die mehr benötigte Laufzeit des DBSCAN Algorithmus mit Hilfsinformationen zum DBSCAN Algorithmus ohne Hilfsinformationen an. Die Implementierung des DBSCAN Algorithmus befindet sich im Anhang (siehe Kap. 12). Für den DBSCAN Algorithmus ohne Hilfsinformationen wurde die Ermittlung der Hilfsinformationen aus der Implementierung herausgenommen. Die Implementierung der Hilfsinformationen ist in Kapitel 6.2.2 beschrieben. Nachfolgend werden die Ergebnisse des Laufzeittests in Form einer Tabelle und graphisch aufbereitet dargestellt:

Testlauf	MinPts	Epsilon	Anzahl Cluster	Punkte mit Rauschen	DBSCAN Ohne	DBSCAN Mit	Mehraufwand
1	4	300	3	6	43577 s	44110 s	1 %
2	10	500	4	985	46891 s	47974 s	2 %
3	10	450	3	1729	48789 s	50485 s	3 %
4	10	600	3	302	51606 s	51940 s	1 %
5	20	900	4	3464	53311 s	53576 s	0 %
			Durchschnittlicher Mehraufwand:				**2 %**

Tabelle 47: DBSCAN: Laufzeittest der 5 Testläufe

Tabelle 47 zeigt das Ergebnis der 5 Testläufe des DBSCAN Algorithmus. Pro Testlauf werden zusätzlich zu den Laufzeiten der beiden Algorithmen, die Parameter ε und MinPts angegeben (siehe Kap. 2.2.2.2), die beim Aufruf des Algorithmus übergeben werden. Diese wurden mit Hilfe von empirischen Tests ermittelt, indem der DBSCAN Algorithmus mit verschiedenen Werten für diese Parameter ausgeführt wurde. Des Weiteren werden die Anzahl der ermittelten Cluster und die Zahl der Punkte die dem Rauschen zugeordnet wurden angegeben. Der Laufzeitvergleich zeigt, dass der Algorithmus mit Ermittlung der Hilfsinformationen nur geringfügig mehr Zeit in Anspruch nimmt als der Algorithmus ohne Hilfsinformationen. Der Mehraufwand beträgt durchschnittlich 2 %. Nachfolgend wird der Laufzeitvergleich der beiden Algorithmen in der Abbildung 31 dargestellt:

Abbildung 31: DBSCAN: Laufzeitvergleich der beiden Algorithmen für alle 5 Testläufe

Diskussion der Ergebnisse

Die Ergebnisse des Laufzeitvergleichs zeigen, dass der DBSCAN Algorithmus mit Ermittlung der Hilfsinformationen nur um einen Bruchteil mehr an Laufzeit benötigt als der DBSCAN Algorithmus ohne Hilfsinformationen. Dies begründet sich vor allem darin, dass die absolute Laufzeit des DBSCAN Algorithmus im Vergleich zum K-Means Algorithmus auf Grund der Laufzeitkomplexität um einiges höher ist. Daraus

ergibt sich, dass die Ermittlung der Hilfsinformationen beim DBSCAN Algorithmus praktisch keine Auswirkung auf die Laufzeit hat, und daher auf jeden Fall durchgeführt werden kann.

Wie beim K-Means Algorithmus verwendet auch der DBSCAN-Algorithmus für die Ermittlung der nähesten und entferntesten Punkte die vorhandene Containerklasse *Vector* von C++. Tests mit verschiedenen Containerklassen (Vector, Deque, List) haben gezeigt, dass die Containerklasse Vector im Sinne der Laufzeit am Besten abschneidet (siehe Tabelle 44).

8 Fazit

Diese Arbeit hat den Begriff des „Kombinierten Data Mining“ definiert und in die drei Arten naiv, Vorgänger kennt Nachfolger und Nachfolger kennt Vorgänger unterteilt (siehe Kap. 3). Bei der Recherche nach bestehenden „Kombinierten Data Mining“ Ansätzen hat sich ergeben, dass es nur naive Ansätze des „Kombinierten Data Mining“ gibt. Diese bestehenden naiven Ansätze benutzen die Data Mining Verfahren Clustering und Klassifikation (siehe Kap. 3.1) oder Assoziation und Clustering (siehe Kap. 3.2).

Es wurde die Art Vorgänger kennt Nachfolger des „Kombinierten Data Mining“ vorgestellt. Ziel der Arbeit war eine Verbesserung der Qualität des Ergebnisses nach Ausführung der beiden Data Mining Verfahren bei möglichst geringem Mehraufwand. Der Mehraufwand war hier gleichzusetzen mit einer längeren Dauer der Laufzeit der Algorithmen. Dazu wurden für den ersten Schritt des „Kombinierten Data Mining“, in diesem Fall dem Clustering, die zwei Algorithmen K-Means und DBSCAN implementiert (siehe Kap. 6). Diese wurden so erweitert, dass so viele Hilfsinformationen wie möglich so effizient wie möglich ermittelt werden (siehe Kap. 5).

Der Laufzeittest des K-Means Algorithmus hat gezeigt, dass der Mehraufwand des K-Means Algorithmus mit Hilfsinformationen im Vergleich zum K-Means Algorithmus ohne Hilfsinformationen durchschnittlich relativ konstant zwischen 20 % und 50 % beträgt (siehe Kap. 7.2.2). Die Schwankungen zwischen diesem Bereich lassen sich durch die unterschiedliche Anzahl der Dimensionen und der Iterationen erklären. Auf Grund dieses relativ konstanten und geringen Mehraufwands, ist dieser auf alle Fälle vertretbar.

Der Genauigkeitstest des K-Means Algorithmus liefert ein sehr zufrieden stellendes Ergebnis, d.h. die ermittelten Mittelpunkte der Cluster des K-Means Algorithmus entsprechen zum Großteil den Mittelpunkten der generierten Testdaten des Testgenerators (siehe Kap. 7.2.3). Kleine Abweichungen der Mittelpunkte und vor allem der Anzahl der Punkte pro Cluster, lassen sich durch Überlappungen der erzeugten Cluster des Testgenerators erklären.

Die Ergebnisse des Laufzeittest des DBSCAN Algorithmus zeigen, dass der Mehraufwand des DBSCAN Algorithmus mit Ermittlung der Hilfsinformationen nur einen sehr geringen Anteil von durchschnittlich 2 % im Vergleich zum DBSCAN

Algorithmus ohne Hilfsinformationen beträgt (siehe Kap. 7.3.2). Dies begründet sich vor allem darin, dass die absolute Laufzeit des DBSCAN Algorithmus im Vergleich zum K-Means Algorithmus auf Grund der Laufzeitkomplexität um einiges höher ist. Daraus ergibt sich, dass die Ermittlung der Hilfsinformationen beim DBSCAN Algorithmus praktisch keine Auswirkung auf die Laufzeit hat, und daher auf jeden Fall durchgeführt werden kann.

Die erzeugten Hilfsinformationen werden im zweiten Schritt des „Kombinierten Data Mining“, in diesem Fall der Klassifikation, genutzt. Dieser Nutzen sollte vor allem in einem qualitativ besseren Ergebnis, gemäß einem adäquaten Gütemaß, liegen. Des Weiteren könnte eine Verbesserung der Effizienz durch das Nutzen der Hilfsinformationen erreicht werden. Genaue Informationen zu dem zweiten Schritt und dessen Testergebnisse können bei Humer [HM04] nachgelesen werden.

9 Abbildungsverzeichnis

10 Tabellenverzeichnis

11 Literaturverzeichnis

[BE76] BERGE, C. *Graphs and Hypergraphs*. American Elsevier. 1976

[BE80] BENTLEY, J. L. *Multidimensional Divide and Conquer*. ACM 1980: 214-229

[BL00] BERRY, J. A. Michael; LINOFF, S. Gordon. *Mastering Data Mining – The Art and Science of Customer Relationship Management*. Wiley. 2000

[BM98] BAKER, L. Douglas; MCCALLUM, Andrew. *Distributional Clustering of Words for Text Classification*. SIGIR 1998: 96-103

[BW98] BLAXTON, Teresa; WESTPHAL, Christopher. *Data Mining Solutions – Methods and Tools for Solving Real World Problems*. Wiley. 1998

[CLMWZ03] CHEN, Zheng; MA, Wei-Ying; LU, Hongjun; WANG, Xuan-Hui; ZENG, Hua-Jun. *CBC: Clustering Based Text Classification Requiring Minimal Labeld Data*. ICDM 2003: 443-450

[CLR90] CORMEN, T.; LIESERSON, C.; RIVEST, R. *Introduction to Algorithms*. 1990

[DKM02] DHILLON, S. Inderjit; KUMAR, Rahul; MALLELA, Subramanyam. *Enhancded word clustering for hierarchical text classification*. KDD 2002: 191–200.

[EKSX96] ESTER, Martin; KRIEGEL, Hans-Peter; SANDER, Jörg; XU, Xiaowei. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise*. München. 1996

[EKSX98] ESTER, Martin; KRIEGEL, Hans-Peter; SANDER, Jörg; XU, Xiaowei. *Clustering for Mining in Large Spatial Databases*. 1998

[EKX95a] ESTER, Martin; KRIEGEL, Hans-Peter; XU, Xiaowei. *Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification*. SSD 1995: 67–82.

[EKX95b] ESTER, Martin; KRIEGEL, Hans-Peter; XU, Xiaowei. *A Database Interface for Clustering in Large Spatial Databases*. KDD 1995: 94-99.

[EL03] ELKAN, Charles. *Using the Triangle Inequality to Accelerate K-Means*. ICML 2003: 147-153.

[ES00] ESTER, Martin; SANDER Jörg: *Knowledge Discovery in Databases*. Springer-Verlag Berlin Heidelberg New York. 2000

[FKR02] FERRÀ, L. Herman; KOWALCZYK, Adam; RASKUTTI, Bhavani. *Combining clustering and co-training to enhance text classification using unlabelled data*. KDD 2002: 620-625

[FPSS96] FAYYAD, U. M.; PIATETSKY-SHAPIRO, G.; SMYTH, P. *Knowledge Discovery and Data Mining*: *Towards a Unifying Framework*. Proceedings 2nd International Conference on Knowledge Discovery and Data Mining, pp. 82-88. 1996

[FR02] FREITAS, A. Alex. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer. 2002

[FRB98] FAYYAD, U.; REINA, C.; BRADLEY, P.S. *Initialising of Iterative Refinement Clustering Algorithms*. KDD 1998: 194-198

[FW99] FRANK, Eibe; WITTER, H. Jan. *Data Mining – Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann. 1999

[GG94] GENTHER, H.; GLESNER, Manfred. *Automatic generation of a fuzzy classification system using fuzzy clustering methods*. SAC 1994: 180-183

[GM04] GOLLER, Mathias; Private Kommunikation. 2004

[HGMZ03] HAN, Hui; MANAVOGLU, Eren; GILES, C. Lee; ZHA, Hongyuan. *Rule-based word clustering for text classification*. SIGIR 2003: 445-446

[HK00] HAN, Jiawei; KAMBER, Micheline. *Data Mining – Concepts and Techniques*. Morgan Kaufmann. 2000

[HKKM97] HAN, Eui-Hong; KARYPIS, George; KUMAR, Vipin; MOBASHER, Bamshad. *Clustering Based On Association Rule Hypergraphs*. DMKD 1997

[HM04] HUMER, Markus. *Kombiniertes Data Mining – Effizientes Klassifizieren und Clustern*. Diplomarbeit. 2004

[KAKS96] KARYPIS, G.; AGGARWAL, R.; KUMAR, V.; SHEKHAR, S. *Hypergraph partitioning: Applications in VLSI domain*. University of Minnesota. 1996

[KO94] KOHONEN, T. *Self-Organizing Maps*. Springer. 1994

[KR90] KAUFMANN, L.; ROUSSEEUW, P. J. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons. New York 1990

[KW95] KASS, R.; WASSERMAN, L. *A reference Bayesian test for nested hypotheses and ist relationship to the Schwarz criterion*. 1995

[LPT93] LEE, Lilian; PEREIRA, Fernando; TISHBY, Naftali. *Distributional clustering of English words*. 1993

[LSW97] LENT, Brian; SWAMI, N. Arun; WIDOM, Jennifer. *Clustering Association Rules*. ICDE 1997: 220–231

[LY00] LAI, Hsiangchu; YANG, Tzyy-Ching. *A Group-based Inference Approach to Customized Marketing on the Web – Integrating Clustering and Association Rules Techniques*. HICSS 2000

[MO91] MOORE, W Andrew. *Efficient Memory-based Learning for Robot Control*. University of Cambridge 1991. Technical report 209

[MI97] MITCHELL, T. M. *Machine Learning*. McGraw-Hill. 1997

[NHS84] NIEVERGELT, J.; HINTERBERGER, H.; SEVCIK, K. C. *The Grid file: An Adaptable, Symmetric Multikey File Structure*. ACM 1984: 38-71

[PM99] PELLEG, Dan; MOORE, W. Andrew. *Accelerating Exact k-means Algorihtms with Geometric Reasoning*. KDD 1999: 277-281

[PM00] PELLEG, Dan; MOORE, W. Andrew. *X-means: Extending K-means with Efficient Estimation of the Number of Clusters*. ICML 2000: 727-734

[PY99] PYLE, D. *Data Preparation for Data Mining*. Morgan Kaufmann. 1999

[SCH96] SCHIKUTA, E. *Grid clustering: An efficient hierarchical clustering method for very large data sets*. 1998

[VL01] VAN LAERHOVEN, Kristof. *Combining the Self-Organizing Map and K-Means Clustering for On-line Classification of Sensor Data*. ICANN 2001: 464–469

[WCRS01] WAGSTAFF, Kiri; CARDIE, Claire; ROGERS, Seth; SCHROEDL, Stefan. *Constrained K-Means Clustering with Background Knowledge*. ICML 2001: 577-584

12 Anhang

Nachfolgend wird der Quellcode der Implementierungen der beiden Algorithmen K-Means und DBSCAN angeführt. Für jede Klasse wird zwischen Header-File und Source-File unterschieden. Das Header-File enthält den Klassenrumpf mit den Attributen und Methoden der Klasse. Das Source-File enthält die Implementierung der Methoden. Eine Beschreibung zu den Methoden und Attributen der Klassen kann in Kapitel 7 gefunden werden. Die Zusammenhänge der einzelnen Klassen werden ebenfalls in Kapitel 7 mittels UML-Diagramm erläutert.

12.1 K-Means Algorithmus

12.1.1 Abstrakte Klasse Database

```
/**
 * Überschrift: Allgemeine Klasse Database mit Interface für zu clusternde Datenbestände
 * Beschreibung: Allgemeine Klasse für die zu clusternden Datenbestände. Für jeden Daten-
 * bestand muss eine eigene Klasse angelget werden die von Database erbt und dessen virtuelle
 * Methoden implementiert
 */
class Database{
	public:
		/**
		 * Methode reset setzt Datenbestand zurück, d.h. beim nächsten Aufruf von
		 * getPoint wird erster Punkt geliefert
		 */
		virtual void reset(void) = 0;

		/**
		 * Methode getPoint liefert nächsten Punkt der Datenbasis
		 * @return Liefert nächsten Punkt der Datenbasis
		 */
		virtual Point *getPoint(void) = 0;

		/**
		 * Methode updateClusterId führt Update der Clusterid eines Punkts auf
		 * Datenbank durch. Der Punkt wird über seine Rowid identifiziert und sein
		 * zugehöriges Cluster wird übergeben
		 * @param rowid Rowid der Datenbank des Punkts
		 * @param clusterid Cluster dem Punkt hinzugefügt wurde
		 */
		virtual void updateClusterId(char *rowid, int clusterid) = 0;

		/**
		 * Commit auf Datenbank wird durchgeführt (nach jedem
		 * Schleifendurchlauf)
		 */
		virtual void commitDatabase(void) = 0;

		/**
		 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen
```

```
 * ClusterId
 * @param clusterid Initiale Clusterid
 * @param nameclusterid Name der Spalte mit Cluster ID
 */
virtual void initClusterIds(char *clusterid, char *nameclusterid) = 0;

/**
 * Methode getData gibt an ob Ausführung der Methode getPoint
 * (Ermittlung der Werte des nächsten Punkts) erfolgreich war
 * @return Boolean-Wert der angibt ob Ermittlung der Werte der
 *             Dimensionen des Punkts erfolgreich war
 */
virtual bool getData(void) = 0;

/**
 * Methode setgetData setzt Boolean-Variable getData, die angibt ob das
 * Lesen des nächsten Punkts aus der Datenbasis erfolgreich war
 * @param getData Gibt an ob Lesen des nächsten Punkts erfolgreich war
 *                          oder nicht
 */
virtual void setgetData(bool getData) = 0;

/**
 * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die
 * Quantil-Berechnung
 */
virtual void computeQuantilClasses(void) = 0;

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen
 * zu den Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
virtual void orderPointsToQuantilClass(Point *point) = 0;

/**
 * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile
 * (25%, 50% und 75%) und ermittelt pro Iteration neue Klassen -->
 * genauere Quantile nach jeder Iteration
 */
virtual void findQuantilsWithQuantilClasses(void) = 0;

/**
 * Methode setComputeHelpData setzt Boolean-Wert der angibt ob
 * Hilfsinformationen für Datenbasis berechnet werden soll
 * @param computeHelpData Boolean-Wert ob Hilfsinformationen
 *                                  berechnet werden sollen
 */
virtual void setcomputeHelpData(bool computeHelpData) = 0;

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und
 * entfernteste Punkte in Datenbank
 * @param clusters Cluster deren näheste und entfernteste Punkte in
 *                          Datenbank geschrieben werden sollen
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste
 *                              Punkte der Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste
 *                          Punkte
 */
virtual void writeClusterNearestandFarestToDatabase(Cluster
                              *clusters,char *table_nearest,char *table_farest) = 0;
```

```
        /**
         * Methode writeDatabaseInformation schreibt Hilfsinformationen zu
         * Datenbasis in Datenbank
         * @param table_database Name der zu erzeugenden Tabelle für
         *                       Hilfsinformationen zu Datenbasis
         */
        virtual void writeDatabaseInformation(char *table_database) = 0;

        /**
         * Methode writeClusterInformation schreibt Hilfsinformationen zu
         * ermittelten Clustern in Datenbank
         * @param clusters Cluster deren Hilfsinformationen in die Datenbank
         *                 geschrieben werden sollen
         * @param table_cluster Name der zu erzeugenden Tabelle für
         *                      Hilfsinformationen zu Clustern
         */
        virtual void writeClusterInformation(Cluster *clusters, char *table_cluster)=0;
};
```

12.1.2 Klasse GetData

Das Source-File der Klasse GetData enthält zusätzlich zu den Implementierungen der Methoden der Klasse GetData auch die Methode *main*, die das Programm startet. Der Quellcode des Source-Files ist der Quellcode vor dem PreCompiler Schritt.

Header-File Klasse GetData

```
/**
 * Überschrift: Klasse GetData zum Lesen der Daten aus Datenbank
 * Beschreibung: Klasse GetData liefert Daten aus Datenbank um
 * Cluster-Algorithmus (Kmeans) auszuführen
 */
class GetData : public Database{
    private:
        // Hilfsinformationen
        double *_sum;          /**< Summe über alle Werte jeder Dimension */
        double *_sumSquared; /**< Vektor mit Quadratsummen pro Dimension */
        double *_mean;       /**< Mittelwert der Werte jeder Dimension */
        double *_stdDev;     /**< Standardabweichung der Werte jeder Dimension */
        double *_minimum;    /**< Minimum jeder Dimension */
        double *_maximum;  /**< Maximum jeder Dimension */
        double *_median;     /**< Ungefährer Median (Schätzverfahren) */
        double *_quantil25;  /**< Ungefähres 25% - Quantil (Schätzverfahren) */
        double *_quantil75;  /**< Ungefähres 75% - Quantil (Schätzverfahren) */
        long _numPoints;     /**< Anzahl der gelesenen Punkte aus Datenbank */

        /**< Hilfsvektor zur Berechnung des Medians (50% - Quantil) */
        vector<vector<Quantil_Class> > _quantile;
        /**< Hilfsvektor zur Berechnung des 25% - Quantils */
        vector<vector<Quantil_Class> > _quantile25;
        /**< Hilfsvektor zur Berechnung des 75% - Quantil2 */
        vector<vector<Quantil_Class> > _quantile75;
        int *_zaehler_dimension;  /**< Gibt an welcher Tabellenspalte die Dimension entspricht */
        double *_data;               /**< Vektor mit Werten pro Dimension */
        /**< Boolean Wert der angibt ob letztes Fetch auf Datenbank erfolgreich war */
        bool _getData;
        /**< Boolean Wert der angibt ob Hilfsinformationen berechnet werden sollen */
        bool _computeHelpData;
```

```
        int _clusterid;          /**< Position der Dimension der Cluster Id */
        int _dimensions;         /**< Anzahl der Dimensionen pro Punkt des Datenbestands */
        Point *_point;           /**< Punkt der über Methode getData zurück gegeben wird */
        char *_table;            /**< Tabelle der Datenbasis aus der Daten gelesen werden */
        char *_columns;          /**< Spalten der Tabelle die für Algorithmus berücksichtigt werden */
        FILE *_file_log;         /**< Log-File des Algorithmus */
        int _countiteration;     /**< Anzahl der Iterationen wird gezählt */
        int _countnumPoints;     /**< Anzahl der Punkte für Algorithmus */

        /**
         * Methode computeStdDev ermittelt die Standardabweichung der Punkte
         */
        void computeStdDev(void);

        /**
         * Methode orderPointsToMedianQuantilClass ermittelt zugehörige Quantil-Klassen für
         * Median zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsToMedianQuantilClass(Point *point);

        /**
         * Methode orderPointsTo25QuantilQuantilClass ermittelt zugehörige Quantil-Klassen für
         * 25%-Quantil zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsTo25QuantilQuantilClass(Point *point);

        /**
         * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen für 75%-
         * Quantil zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsTo75QuantilQuantilClass(Point *point);

        /**
         * Methode findMedianWithQuantilClasses ermittelt ungefähren Median und ermittelt pro
         * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findMedianWithQuantilClasses(void);

        /**
         * Methode findQuantil25WithQuantilClasses ermittelt ungefähres 25%-Quantil und
         * ermittelt pro Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findQuantil25WithQuantilClasses(void);

        /**
         * Methode findQuantil75WithQuantilClasses ermittelt ungefähres 75%-Quantil und
         * ermittelt pro Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findQuantil75WithQuantilClasses(void);

        /**
         * Methode getPoint2 liefert nächsten Punkt der Datenbasis (ab 2 Iteration)
         * @return Liefert nächsten Punkt der Datenbasis
         */
        Point *getPoint2(void);

    public:
        /**
         * Konstruktor der Klasse GetData
```

```
 * @param dimension Anzahl der Dimensionen pro Punkt
 * @param clusterid Position der Dimension der Cluster Id
 * @param table Tabelle aus der Daten gelesen werden
 * @param columns Spalten aus Tabellen die relevant sind
 * @param file_log Log-File des Algorithmus
 * @param countnumpoints Anzahl der Punkte für Algorithmus
 */
GetData(int dimensions, int clusterid, char *table, char *columns, FILE *file_log, int
                                                                        countnumpoints);

/**
 * Destruktor der Klasse GetData
 */
~GetData();

/**
 * Methode reset setzt Datenbestand zurück, d.h. beim nächsten Aufruf von getPoint wird
 * erster Punkt geliefert
 */
void reset(void);

/**
 * Methode getPoint liefert nächsten Punkt der Datenbasis
 * @return Liefert nächsten Punkt der Datenbasis
 */
Point *getPoint(void);

/**
 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen ClusterId
 * @param clusterid Initiale Clusterid
 * @param nameclusterid Name der Spalte mit Cluster ID
 */
void initClusterIds(char *clusterid, char *nameclusterid);

/**
 * Commit auf Datenbank wird durchgeführt (nach jedem Schleifendurchlauf)
 */
void commitDatabase(void);

/**
 * Methode getClusterId liefert Position der Dimension der Cluster ID
 * @return Position der Dimension der Cluster Id
 */
int getClusterId(void){
        return _clusterid;
}

/**
 * Methode updateClusterId führt Update der Clusterid eins Punkts auf Datenbank durch.
 * Der Punkt wird über seine Rowid identifiziert und sein zugehöriges Cluster wird mit
 * übergeben
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster dem Punkt hinzugefügt wurde
 */
void updateClusterId(char *rowid, int clusterid);

/**
 * Methode setgetData setzt Boolean-Wert getData der Klasse getData
 * @param getData Boolean-Wert der angibt ob letztes Fetch auf Datenbank erfolgreich
 *                        war
 */
void setgetData(bool getData){
```

```
        _getData = getData;
}

/**
 * Methode setComputeHelpData setzt Boolean-Wert der angibt ob Hilfsinformationen für
 * Datenbasis berechnet werden soll
 * @param computeHelpData Boolean-Wert ob Hilfsinformationen berechnet werden
 *                                  sollen
 */
void setcomputeHelpData(bool computeHelpData){
        _computeHelpData = computeHelpData;
}

/**
 * Methode getData gibt an ob Ausführung der Methode getPoint (Ermittlung der Werte des
 * nächsten Punkts) erfolgreich war
 * @return Boolean-Wert der angibt ob Ermittlung der Werte der Dimensionen des Punkts
 *              erfolgreich war
 */
bool getData(void){
        return _getData;
}

/**
 * Anzahl der Punkte der Datenbasis die für den Algorithmus verwendet wurden
 * @return Anzahl der verwendeten Punkte der Datenbasis
 */
long getNumPoints(){
        return _numPoints;
}

/**
 * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die Quantil-
 * Berechnung nach erster Iteration des Algorithmus (für 25% - Quantil, Median und 75% -
 * Quantil
 */
void computeQuantilClasses(void);

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen (für alle 3
 * Quantile) zu den Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void orderPointsToQuantilClass(Point *point);

/**
 * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile (25%, 50% und
 * 75%) und ermittelt pro Iteration neue Klassen --> genauere Quantile nach jeder Iteration
 */
void findQuantilsWithQuantilClasses(void);

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste
 * Punkte in Datenbank
 * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank
 *                          geschrieben werden sollen
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der
 *                                  Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
 */
void writeClusterNearestandFarestToDatabase(Cluster *clusters,char *table_nearest, char
                                                                        *table_farest);
```

```
        /**
         * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in
         * Datenbank
         * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                                  Datenbasis
         */
        void writeDatabaseInformation(char *table_database);

        /**
         * Methode writeClusterInformation schreibt Hilfsinformationen der ermittelten Cluster in
         * Datenbank
         * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden
         *                          sollen
         * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                                  Clustern
         */
        void writeClusterInformation(Cluster *clusters, char *table_cluster);
};
```

Source File Klasse GetData

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Quantil_Class.h"
#include "Information_Point.h"
#include "cluster.h"
#include "database.h"
#include "point.h"
#include "kmeans.h"
#include "getdata.h"
#include <string.h>
#include <time.h>
#include <malloc.h>
#include "sqlcpr.h"

/**
 * Variablen für SQL-Anweisungen (Daten lesen)
 */
EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR username[30];
        VARCHAR password[30];
        VARCHAR _categoriccolumns[50]; /**< Spalten (kategorische Werte) die nicht berücksichtigt */
        VARCHAR _initialpoints[50];          /**< Initiale Punkte für Cluster-Mittelpunkte */
        VARCHAR statement[240];
        VARCHAR statement2[240];
        VARCHAR sid[10];
        int init_clusterid;
        int up_clusterid;
        VARCHAR up_rowid[18];
        VARCHAR ins_rowid[18];
        VARCHAR update_statement[240];
        VARCHAR init_statement[240];
        VARCHAR create_statement_nearest[240];
```

```
        VARCHAR create_statement_farest[240];
        VARCHAR insert_statement_nearest[240];
        VARCHAR insert_statement_farest[240];
        VARCHAR create_statement_database[240];
        VARCHAR insert_statement_database[240];
        VARCHAR create_statement_cluster[300];
        VARCHAR insert_statement_cluster[300];
        int dimension;
        double sum;
        double sumsquared;
        double mean;
        double stddev;
        double minimum;
        double maximum;
        double median;
        double quantil25;
        double quantil75;
        double sqrtnumpoints;
        long number_points;
        int clusterid;
EXEC SQL END DECLARE SECTION;

SQLDA *select_des;
SQLDA *bind_des;
SQLDA *select_des2;
SQLDA *bind_des_second;
extern SQLDA *sqlald();
extern void sqlnul();
EXEC SQL INCLUDE SQLDA;
EXEC SQL INCLUDE SQLCA;

long _NumPoints;    /**< Anzahl der Punkte aus Datenbasis die für Clustering herangezogen werden */
int _NumClusters;   /**< Anzahl der zu ermittelnden Cluster (= Parameter k) */
int _MaxInitialNumPoints;/**< Maximale Anzahl der initalen Punkte pro Cluster */
int _MinInitialNumPoints;/**< Minimale Anzahl der initalen Punkte pro Cluster */
int _NearestNumPoints;  /**< Anzahl der nähesten Punkte pro Cluster, die berücksichtigt werden */
int _FarestNumPoints;   /**< Anzahl der entferntesten Punkte pro Cluster, die berücksichtigt werden */
int _clusterid;         /**< Dimension mit Cluster-Id in Tabelle */
char *_nameclusterid;   /**< Name der Spalte mit Cluster ID */
/**< Distanzfunktion angeben: 1 = Quadratische, 2 = Euklidische, 3 = Manhatten */
int _distancefunction;
char *_table;           /**< Tabelle aus der Daten gelesen werden */
char *_columns;         /**< Spalten die für Algorithmus relevant sind */
char *_table_nearest;   /**< Tabelle in die näheste Punkte der Cluster gespeichert werden */
char *_table_farest;    /**< Tabelle in die entfernteste Punkte der Cluster gespeichert werden */
char *_table_database;  /**< Tabelle in die Hilfsinformationen zu Datenbasis gespeichert werden */
char *_table_cluster;   /**< Tabelle in die Hilfsinformationen zu Clustern gespeichert werden */
char *_file_kmeans;     /**< File in das Ergebnis (Dauer) des Algorithmus geschrieben wird */
char *_file_log;        /**< Log-File bei Ausführung des Algorithmus */
char *ID_UNCLASSIFIED;      /**< Cluster ID für unklassifzierte Punkte */
int  _numberinitialiterations;  /**< Anzahl der maximalen Iterationen bei Initialisierung */
int  _numberiterations;         /**< Anzahl der maximalen Iterationen */
char *_user;                    /**< User für Datenbankanmeldung */
char *_password;                /**< Passwort für Datenbankanmeldung */
char *_service;                 /**< Dienst für Datenbankanmeldung */

/**
 * Hauptprogramm
 */
int main(int argc, char *argv[]){
        long time_start, time_end, time_erg_start;
        int i=0,j=0,numdimensions=0, prec, scal, nullok, k, countcategoric = 0;
```

```
FILE *file_ergebnis, *file_log;
GetData *datenbank;                    // Datenbasis mit Daten für Kmeans
KMeans *kmeans;                        // KMeans - Algorithmus
char *bind_var=0, *token, seps[] = ";";
bool *usedcolumns;
int  *initialpoints;
time_t ltime;

try {
        // Programmparameter berücksichtigen
        if (!((argc == 25) || (argc == 26))) {
                printf("Fehlende Programmparameter!!!");
                return 1;
        }
        _user              = argv[1];
        _password                  = argv[2];
        _service           = argv[3];
        _table                     = argv[4];
        _columns                   = argv[5];
        _NumPoints                 = atol(argv[6]);
        _NumClusters       = atoi(argv[7]);
        _MaxInitialNumPoints = atoi(argv[8]);
        _MinInitialNumPoints = atol(argv[9]);
        _clusterid         = atoi(argv[10]);
        _nameclusterid     = argv[11];
        _distancefunction  = atoi(argv[12]);
        _NearestNumPoints  = atoi(argv[13]);
        _FarestNumPoints   = atoi(argv[14]);
        _file_kmeans       = argv[15];
        _file_log          = argv[16];
        ID_UNCLASSIFIED    = argv[17];
        _table_nearest     = argv[18];
        _table_farest      = argv[19];
        _table_database    = argv[20];
        _table_cluster     = argv[21];
        _numberinitialiterations = atoi(argv[22]);
        _numberiterations  = atoi(argv[23]);
        strcpy((char *)_initialpoints.arr, argv[24]);
        _initialpoints.len = strlen((char *)_initialpoints.arr);
        if (argc == 26){
                strcpy((char *)_categoriccolumns.arr, argv[25]);
                _categoriccolumns.len = strlen((char *)_categoriccolumns.arr);
        }
        file_log    = fopen(_file_log, "a");          // Log-File öffnen

        if (_MinInitialNumPoints >= _MaxInitialNumPoints){
                _MinInitialNumPoints = _MaxInitialNumPoints / 2;
                fprintf(file_log,"%s %i %s \n", "Anzahl der minimalen initialen Punkte auf",
                                                        _MinInitialNumPoints, "gesetzt!");
        }

        // Datenbankverbindung wird erstellt
        strcpy((char *)username.arr,_user);
        username.len = strlen((char *)username.arr);
        strcpy((char *)password.arr,_password);
        password.len = strlen((char *)password.arr);
        strcpy((char *)sid.arr,_service);
        sid.len = strlen((char *)sid.arr);

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND CONTINUE;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
```

```
EXEC SQL CONNECT :username IDENTIFIED BY :password USING :sid;
/*****************************************************************************
 * Dynamische SELECT-Anweisung für lesen aus Datenbank
 *****************************************************************************/
// Deskriptoren für dynamisches Select-Statement werden erzeugt
select_des = sqlald(_clusterid + 1,_clusterid + 1,0);
bind_des = sqlald(_clusterid + 1,_clusterid + 1,4);
select_des->N = _clusterid + 1;
bind_des->N = 0;

strcpy((char *)statement.arr,"SELECT t.ROWID, ");
strcat((char *)statement.arr, _columns);
strcat((char *)statement.arr," FROM ");
strcat((char *)statement.arr, _table);
strcat((char *)statement.arr, " t");
statement.len = strlen((char *)statement.arr);

EXEC SQL PREPARE sql_stmt FROM :statement;
EXEC SQL DECLARE cursor CURSOR FOR sql_stmt;
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
bind_des->N = bind_des->F;

for (i=0; i < bind_des->F; i++){
        printf("Eingabe des Wertes für Variable %*.*S:\n?", (int) bind_des->C[i], bind_des-
                                                                                >S[i]);
        gets(bind_var);
        bind_des->L[i] = strlen(bind_var);                      /* Länge des Werts */
        /* Reservieren von Speicher für Wert und Indikatorvariable */
        bind_des->V[i] = (char *)malloc(bind_des->L[i] + 1);
        bind_des->I[i] = (short *)malloc(sizeof(short));

        strcpy(bind_des->V[i], bind_var);       /* Speichern des Wertes */
        bind_des->I[i] = 0;                     /* Indikator-Wert setzen */
        bind_des->T[i] = 1;                     /* Datentyp = CHAR */
}
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
if (select_des->F < 0) select_des->F*= -1;
select_des->N = select_des->F;

for (i=0; i<select_des->F; i++) {
        /* lösche NULL-Bit für NULLable Datentypen */
        sqlnul((unsigned short*)&(select_des->T[i]), (unsigned short*)&(select_des->T[i]),
                                                                        &nullok);

        /* Korrigieren der Länge, falls notwendig */
        switch (select_des->T[i]) {
                case 1: break;                                  /* CHAR */
                case 2:
                        sqlpr2((unsigned int *)&select_des->L[i], &prec, &scal);/* NUMBER */
                        if (prec == 0) prec = 40;
                        if ((scal > 0) || (scal == -127))
                                select_des->L[i] = sizeof(float);       /* FLOAT */
                        else
                                select_des->L[i] = sizeof(int);         /* INT */
                        if (i != _clusterid)
                                numdimensions++;
                        break;
                case 8:  select_des->L[i] = 240; break;         /* LONG */
                case 24: select_des->L[i] = 240; break;         /* LONG RAW */
                case 11: select_des->L[i] = 18; break;          /* ROWID */
                case 12: select_des->L[i] = 9; break;           /* DATE */
                case 23: break;                                 /* RAW */
```

```
                case 104: select_des->L[i] = 18; break;             /* ROWID */
        }
        /* Speicherallokation für Attributwerte sowie Indikatorvar. */
        select_des->V[i] = (char *)malloc(select_des->L[i]);
        select_des->I[i] = (short *)malloc(sizeof(short));

        /* alle Datentypen außer LONG RAW zu CHAR konvertieren */
        if ((select_des->T[i] != 24) && (select_des->T[i] != 2)) select_des->T[i] = 1;

        if (select_des->T[i] == 2)
                if ((scal > 0) || (scal == -127))
                        select_des->T[i] = 4;        //Float
                else
                        select_des->T[i] = 3;        //INT
}

/*****************************************************************************
 * Zweite dynamische SELECT-Anweisung für lesen aus Datenbank
 *****************************************************************************/
// Deskriptoren für dynamisches Select-Statement werden erzeugt
select_des2 = sqlald(_clusterid + 1,_clusterid + 1,0);
bind_des_second = sqlald(_clusterid + 1,_clusterid + 1,4);
select_des2->N = _clusterid + 1;
bind_des_second->N = 1;

strcpy((char *)statement2.arr,"SELECT t.ROWID, ");
strcat((char *)statement2.arr, _columns);
strcat((char *)statement2.arr," FROM ");
strcat((char *)statement2.arr, _table);
strcat((char *)statement2.arr, " t WHERE ");
strcat((char *)statement2.arr, _nameclusterid);
strcat((char *)statement2.arr, " <> ");
strcat((char *)statement2.arr, ID_UNCLASSIFIED);
statement2.len = strlen((char *)statement2.arr);

EXEC SQL PREPARE sql_stmt2 FROM :statement2;
EXEC SQL DECLARE cursor2 CURSOR FOR sql_stmt2;

EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt2 INTO bind_des_second;
bind_des_second->N = bind_des_second->F;

for (i=0; i < bind_des_second->F; i++){
        bind_var = new char;
        strcpy(bind_var, ID_UNCLASSIFIED);
        bind_des_second->L[i] = strlen(bind_var);                    /* Länge des Werts */
        /* Reservieren von Speicher für Wert und Indikatorvariable */
        bind_des_second->V[i] = (char *)malloc(bind_des_second->L[i] + 1);
        bind_des_second->I[i] = (short *)malloc(sizeof(short));

        strcpy(bind_des_second->V[i], bind_var);          /* Speichern des Wertes */
        *(bind_des_second->I[i]) = 0;                     /* Indikator-Wert setzen */
        bind_des_second->T[i] = 5;                        /* Datentyp = CHAR */
}


EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt2 INTO select_des2;
if (select_des2->F < 0) select_des2->F*= -1;
select_des2->N = select_des2->F;

for (i=0; i<select_des2->F; i++) {
        /* lösche NULL-Bit für NULLable Datentypen */
        sqlnul((unsigned short*)&(select_des2->T[i]), (unsigned short*)&(select_des2->T[i]),
```

```
                                                                    &nullok);

        /* Korrigieren der Länge, falls notwendig */
        switch (select_des2->T[i]) {
                case 1: break;                                              /* CHAR */
                case 2:                                                 /* NUMBER */
                        sqlpr2((unsigned int *)&select_des2->L[i], &prec, &scal);
                        if (prec == 0) prec = 40;
                        if ((scal > 0) || (scal == -127))
                                select_des2->L[i] = sizeof(float);          /* FLOAT */
                        else
                                select_des2->L[i] = sizeof(int);            /* INT */
                        break;
                case 8:  select_des2->L[i] = 240; break;                    /* LONG */
                case 24: select_des2->L[i] = 240; break;                /* LONG RAW */
                case 11: select_des2->L[i] = 18; break;                      /* ROWID */
                case 12: select_des2->L[i] = 9; break;                      /* DATE */
                case 23: break;                                             /* RAW */
                case 104: select_des2->L[i] = 18; break;                    /* ROWID */
        }
        /* Speicherallokation für Attributwerte sowie Indikatorvar. */
        select_des2->V[i] = (char *)malloc(select_des2->L[i]);
        select_des2->I[i] = (short *)malloc(sizeof(short));

        /* alle Datentypen außer LONG RAW zu CHAR konvertieren */
        if ((select_des2->T[i] != 24) && (select_des2->T[i] != 2)) select_des2->T[i] = 1;

        if (select_des2->T[i] == 2)
                if ((scal > 0) || (scal == -127))
                        select_des2->T[i] = 4;        //Float
                else
                        select_des2->T[i] = 3;        //INT
}

/*****************************************************************************
 * Update-Statement zum Setzen der Clusterid eines Punkts
 *****************************************************************************/
// Update-Statement wird ermittelt und ausgeführt
strcpy((char *)update_statement.arr, "UPDATE ");
strcat((char *)update_statement.arr, _table);
strcat((char *)update_statement.arr, " SET ");
strcat((char *)update_statement.arr, _nameclusterid);
strcat((char *)update_statement.arr, " = :c WHERE ROWID = :r");
update_statement.len = strlen((char *)update_statement.arr);

EXEC SQL PREPARE up_sql_stmt FROM :update_statement;

/*****************************************************************************
 * Beginn der Main - Routine
 *****************************************************************************/
datenbank = new GetData(numdimensions, _clusterid, (char *)_table, (char *)_columns,
                                                                    file_log, _NumPoints);
datenbank->initClusterIds((char *)ID_UNCLASSIFIED, _nameclusterid);

// Spalten die für Clustering verwendet werden
usedcolumns = new bool[numdimensions];
for(k = 0; k < numdimensions; k++)
        usedcolumns[k] = true;

token = strtok( (char *)_categoriccolumns.arr, seps );
while( token != NULL){
        k = atoi(token);
```

```
        usedcolumns[k] = false;
        countcategoric++;
        token = strtok( NULL, seps);
}

// Initiale Punkte für Cluster - Mittelpunkte
initialpoints = new int[_NumClusters];
for (k = 0; k < _NumClusters; k++)
        initialpoints[k] = 0;
k = 0;
token = strtok( (char *)_initialpoints.arr, seps );
while( token != NULL){
        initialpoints[k] = atoi(token);
        k++;
        token = strtok( NULL, seps);
}
// Kmeans - Algorithmus initialisieren (Anzahl Cluster und Dimensionen)
kmeans = new KMeans(datenbank, _NumClusters, numdimensions, _NearestNumPoints,
                        _FarestNumPoints, _distancefunction, usedcolumns, file_log,
                        initialpoints, _numberinitialiterations, _numberiterations);

// Kmeans starten und Clustering ausführen
time( &ltime );
fprintf(file_log,"%s %s", "Beginn K-means:", ctime( &ltime ) );
fprintf(file_log, "%s %i %s %i %s %i %s %i \n", "Anzahl zu clusternde Punkte:",
        _NumPoints, "Anzahl Cluster:", _NumClusters, "Spalten:", numdimensions,
        "Kategorische Spalten:", countcategoric);
fflush(file_log);
time_start = clock();                   // Hilfsvariable für Zeitmessung
kmeans->cluster(_NumPoints, _MaxInitialNumPoints, _MinInitialNumPoints);

// Hilfsinformationen in Datenbank schreiben
time_erg_start = clock();
fprintf(file_log,"Ergebnisse in Tabellen schreiben. \n");
fflush(file_log);
kmeans->writeClusterNearestandFarestToDatabase((char *)_table_nearest,(char
                                                        *)_table_farest);
kmeans->writeDatabaseInformation((char *)_table_database);
kmeans->writeClusterInformation((char *)_table_cluster);
time_end = clock();                     // Hilfsvariable für Zeitmessung

fprintf(file_log, "%s %f \n", "Dauer für Speichern der Hilfsinformationen:", (time_end –
                                        time_erg_start)/(float)CLOCKS_PER_SEC);

// Ergebnis des Kmeans wird in File geschrieben
time( &ltime );
file_ergebnis = fopen(_file_kmeans, "a");
fprintf(file_ergebnis, "%s %i %s %i %s %i %s %i %s %f %s %d %s", "Punkte:", datenbank-
        >getNumPoints(),"Cluster:", _NumClusters, "Spalten:", numdimensions,
        "Kategorische Spalten:", countcategoric, "Dauer:",
        (time_end - time_start)/(float)CLOCKS_PER_SEC, "Iterationen:", kmeans-
        >getNumIterations(),ctime( &ltime ));
fclose(file_ergebnis);

// Cluster wieder löschen
delete kmeans;

/* Freigeben der Attribute und zugehöriger Indikatorvariablen */
for (i=0; i < select_des->F; i++) {
        free(select_des->V[i]);
        free(select_des->I[i]);
}
```

```
                for (i=0; i < select_des2->F; i++) {
                        free(select_des2->V[i]);
                        free(select_des2->I[i]);
                }
                /* Freigeben der Deskriptoren */
                sqlclu(select_des);
                sqlclu(select_des2);

                EXEC SQL COMMIT WORK RELEASE;

                fprintf(file_log,"Kmeans fertig. \n \n");
                fclose(file_log);
                return 0;
        } catch (exception& e){
                fprintf(file_log,"%s \n", e.what());
                fclose(file_log);
                return 1;
        }
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fclose(file_log);
        }
        return 1;
}

/*****************************************************************
 * Klasse GetData
 *****************************************************************/
/**
 * Konstruktor der Klasse GetData
 * @param dimension Anzahl der Dimensionen pro Punkt
 * @param clusterid Position der Dimension der Cluster Id
 * @param table Tabelle aus der Daten gelesen werden
 * @param columns Spalten aus Tabellen die relevant sind
 * @param file_log Log-File des Algorithmus
 * @param countnumpoints Anzahl der Punkte für Algorithmus
 */
GetData::GetData(int dimensions, int clusterid, char *table, char *columns, FILE *file_log, int
countnumpoints):_quantile(dimensions,vector<Quantil_Class>(100)),
        _quantile25(dimensions,vector<Quantil_Class>(100)),
        _quantile75(dimensions,vector<Quantil_Class>(100)){
        _numPoints = 0;
        _dimensions = dimensions;
        _zaehler_dimension = new int[_dimensions];
        _data       = new double[_dimensions];
        _mean       = new double[_dimensions];
        _stdDev     = new double[_dimensions];
        _sum        = new double[_dimensions];
        _sumSquared = new double[_dimensions];
        _maximum    = new double[_dimensions];
        _minimum    = new double[_dimensions];
        _median     = new double[_dimensions];
        _quantil25  = new double[_dimensions];
        _quantil75  = new double[_dimensions];
        _point    = new Point(_dimensions);
        _table    = (char *)table;
        _columns  = (char *)columns;
        _clusterid = clusterid;
        _file_log  = file_log;
        _quantile.reserve(100);
        _quantile25.reserve(100);
```

```
	_quantile75.reserve(100);
	_computeHelpData = false; // Damit bei Initialisierung keine HIlfsinformationen berechnet werden
	_countiteration = 0;
	_countnumPoints = countnumpoints;

	for (int i = 0; i < _dimensions; i++){
		_sum[i]     = 0;
		_sumSquared[i] = 0;
		_mean[i]      = 0;
		_stdDev[i]    = 0;
		_minimum[i]   = 0;
		_maximum[i]   = 0;
		_median[i]    = 0;
		_quantil25[i] = 0;
		_quantil75[i] = 0;
	}
}

/**
 * Destruktor der Klasse GetData
 */
GetData::~GetData(void){
	delete []_zaehler_dimension;
	delete []_data;
	delete []_mean;
	delete []_sum;
	delete []_sumSquared;
	delete []_stdDev;
	delete []_minimum;
	delete []_maximum;
	delete []_median;
	delete []_quantil25;
	delete []_quantil75;
	delete _point;

	_quantile.clear();
	_quantile25.clear();
	_quantile75.clear();
}

/**
 * Methode reset setzt Datenbestand zurück, d.h. beim nächsten Aufruf von getPoint wird erster Punkt
 * geliefert
 */
void GetData::reset(void){
	_numPoints = 0;
	setgetData(true);
	_countiteration++;

	EXEC SQL WHENEVER SQLWARNING CONTINUE;
	EXEC SQL WHENEVER NOTFOUND CONTINUE;
	EXEC SQL WHENEVER SQLERROR goto sqlerror;

	if (_countiteration > 2)
		EXEC SQL CLOSE cursor2;
	else
		EXEC SQL CLOSE cursor;
	EXEC SQL COMMIT;

	if (_countiteration > 1)
		EXEC SQL OPEN cursor2 USING DESCRIPTOR bind_des_second;
	else
```

```
                EXEC SQL OPEN cursor USING DESCRIPTOR bind_des;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fprintf(_file_log,"%s %i \n","Fehler bei Reset!", sqlca.sqlcode);
                fflush(_file_log);
                exit(1);
        }
}

/**
 * Methode getPoint liefert nächsten Punkt der Datenbasis
 * @return Liefert nächsten Punkt der Datenbasis
 */
Point *GetData::getPoint(void){
        int i=0,c=0,j=0, clusterid=0;
        char *rowid;

        if (_countiteration > 1){
                return getPoint2();
        }else{
                EXEC SQL WHENEVER SQLWARNING CONTINUE;
                EXEC SQL WHENEVER NOTFOUND CONTINUE;
                EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
                EXEC SQL FETCH cursor USING DESCRIPTOR select_des;

                _numPoints++;
                for (i=0; i < select_des->F; i++){
                        if (select_des->T[i] == 3){
                                _data[j] = (int) *select_des->V[i];                     //INT
                                if (_computeHelpData){
                                        _sum[j]+= (int) *select_des->V[i];
                                        _sumSquared[j]+= _data[j] * _data[j];
                                        _zaehler_dimension[j] = i;

                                        // Minimum bestimmen
                                        if (_numPoints == 0)
                                                _minimum[j] = _data[j];
                                        else if (_data[j] < _minimum[j])
                                                _minimum[j] = _data[j];
                                        // Maximum bestimmen
                                        if (_numPoints == 0)
                                                _maximum[j] = _data[j];
                                        else if (_data[j] > _maximum[j])
                                                _maximum[j] = _data[j];
                                }
                                j++;
                        }
                        if (select_des->T[i] == 4){
                                _data[j] = *(float *)select_des->V[i];                  //FLOAT
                                if (_computeHelpData){
                                        _sum[j]+= *(float *)select_des->V[i];
                                        _sumSquared[j]+= _data[j] * _data[j];
                                        _zaehler_dimension[j] = i;

                                        // Minimum bestimmen
                                        if (_numPoints == 0)
                                                _minimum[j] = _data[j];
                                        else if (_data[j] < _minimum[j])
                                                _minimum[j] = _data[j];
                                        // Maximum bestimmen
                                        if (_numPoints == 0)
```

```
                                        _maximum[j] = _data[j];
                                else if (_data[j] > _maximum[j])
                                        _maximum[j] = _data[j];
                                }
                                j++;
                        }
                        if (i == 0)
                                rowid = (char *)select_des->V[i];                       //ROWID
                        if (i == _clusterid)
                                clusterid = (int) *select_des->V[i];                    //Cluster ID ermitteln
                }
                if ((_computeHelpData) && (_numPoints == _countnumPoints))
                        computeStdDev();                                        // Standardabweichung berechnen
                _point->set(_data, rowid, clusterid);
                return _point;
        }
sqlerror:
        if (sqlca.sqlcode == -1002) {
                if (_numPoints < _countnumPoints){
                        fprintf(_file_log,"Fetch-Fehler, obwohl noch Punkte vorhanden! \n");
                        fflush(_file_log);
                }else
                        setgetData(false);          //Fetch fertig, kein Datensatz mehr
                return 0;
        } else {
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
                exit(1);
        }
}

/**
 * Methode getPoint2 liefert nächsten Punkt der Datenbasis
 * @return Liefert nächsten Punkt der Datenbasis
 */
Point *GetData::getPoint2(void){
        int i=0,c=0,j=0, clusterid=0;
        char *rowid;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND CONTINUE;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
        EXEC SQL FETCH cursor2 USING DESCRIPTOR select_des2;

        _numPoints++;
        for (i=0; i < select_des2->F; i++){
                if (select_des2->T[i] == 3)
                        _data[j++] = (int) *select_des2->V[i];                  //INT
                if (select_des2->T[i] == 4)
                        _data[j++] = *(float *)select_des2->V[i];               //FLOAT
                if (i == 0)
                        rowid = (char *)select_des2->V[i];                      //ROWID
                if (i == _clusterid)
                        clusterid = (int) *select_des2->V[i];                   //Cluster ID ermitteln
        }
        _point->set(_data, rowid, clusterid);
        return _point;
sqlerror:
        if (sqlca.sqlcode == -1002) {
                if (_numPoints < _countnumPoints){
                        fprintf(_file_log,"Fetch-Fehler, obwohl noch Punkte vorhanden! \n");
                        fflush(_file_log);
```

```
                }else
                        setgetData(false);          //Fetch fertig, kein Datensatz mehr
                return 0;
        } else {
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
                exit(1);
        }
}

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void GetData::computeStdDev(void){
        double *sum = _sum, *sumSquared = _sumSquared, *stdDev = _stdDev,
                *mean = _mean, h_stddev;

        for(int i = 0; i < _dimensions; i++){
                h_stddev = *sum++ / _numPoints;
                *stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints - h_stddev * h_stddev));
                *mean++ = h_stddev;
        }
}

/**
 * Methode updateClusterId führt Update der Clusterid eins Punkts auf Datenbank durch. Der Punkt
 * wird über seine Rowid identifiziert und sein zugehöriges Cluster wird mit übergeben
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster dem Punkt hinzugefügt wurde
 */
void GetData::updateClusterId(char *rowid, int clusterid){
        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        up_clusterid = clusterid;
        strcpy((char *)up_rowid.arr,rowid);
        up_rowid.len = 18;

        EXEC SQL EXECUTE up_sql_stmt USING :up_clusterid, :up_rowid;
sqlerror:
        if (sqlca.sqlcode != 0)
                if (!((sqlca.sqlcode == -1002) || (sqlca.sqlcode == 1403))){
                        fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                        fflush(_file_log);
                }
}

/**
 * Commit auf Datenbank wird durchgeführt (nach jedem Schleifendurchlauf)
 */
void GetData::commitDatabase(){
        _countiteration = 0;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fprintf(_file_log,"%s %i \n","Fehler bei Commit!", sqlca.sqlcode);
```

```
                fflush(_file_log);
                exit(1);
        }
}

/**
 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen ClusterId
 * @param clusterid Initiale Clusterid
 * @param nameclusterid Name der Spalte mit Cluster ID
 */
void GetData::initClusterIds(char *clusterId, char *nameclusterid){
        // Update-Statement wird erzeugt und ausgeführt
        strcpy((char *)init_statement.arr, "UPDATE ");
        strcat((char *)init_statement.arr, _table);
        strcat((char *)init_statement.arr, " set ");
        strcat((char *)init_statement.arr, nameclusterid);
        strcat((char *)init_statement.arr, " = ");
        strcat((char *)init_statement.arr, clusterId);
        strcat((char *)init_statement.arr, " where ");
        strcat((char *)init_statement.arr, nameclusterid);
        strcat((char *)init_statement.arr, " <> ");
        strcat((char *)init_statement.arr, clusterId);
        init_statement.len = strlen((char *)init_statement.arr);

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
        EXEC SQL prepare init_sql_stmt from :init_statement;
        EXEC SQL execute init_sql_stmt;
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
        }
}

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste Punkte in
 * Datenbank
 * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank geschrieben werden
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
 */
void GetData::writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char
                                                                                        *table_farest){
        Cluster *cluster;
        bool create_nearest = true;
        bool create_farest  = true;
        vector<Information_Point> vector_nearest;
        vector<Information_Point> vector_farest;
        vector<Information_Point>::iterator i;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        // Create-Statment für näheste Punkte der Cluster
        strcpy((char *)create_statement_nearest.arr, "CREATE TABLE ");
        strcat((char *)create_statement_nearest.arr, table_nearest);
        strcat((char *)create_statement_nearest.arr, " AS SELECT ");
        strcat((char *)create_statement_nearest.arr, _columns );
```

```
strcat((char *)create_statement_nearest.arr, " FROM ");
strcat((char *)create_statement_nearest.arr, _table);
strcat((char *)create_statement_nearest.arr, " t WHERE rowid = '");

// Create-Statment für entfernteste Punkte der Cluster
strcpy((char *)create_statement_farest.arr, "CREATE TABLE ");
strcat((char *)create_statement_farest.arr, table_farest);
strcat((char *)create_statement_farest.arr, " AS SELECT ");
strcat((char *)create_statement_farest.arr, _columns );
strcat((char *)create_statement_farest.arr, " FROM ");
strcat((char *)create_statement_farest.arr, _table);
strcat((char *)create_statement_farest.arr, " t WHERE rowid = '");

for(cluster = clusters; cluster; cluster = cluster->getNext()){
        // Nähesten Punkte in Tabelle schreiben
        vector_nearest = cluster->getVectorNearest();
        for(i = vector_nearest.begin(); i != vector_nearest.end();i++){
                strncpy((char *)ins_rowid.arr,i->getRowId(), 18);
                ins_rowid.len = 18;

                if (create_nearest){
                        strcat((char *)create_statement_nearest.arr, (char *)ins_rowid.arr);
                        strcat((char *)create_statement_nearest.arr,"'");
                        create_statement_nearest.len = strlen((char *)create_statement_nearest.arr);
                        EXEC SQL PREPARE create_sql_stmt_nearest FROM
                                                        :create_statement_nearest;
                        EXEC SQL EXECUTE create_sql_stmt_nearest;
                        create_nearest = false;
                } else {
                        //INSERT mit Rowid
                        strcpy((char *)insert_statement_nearest.arr, "INSERT INTO ");
                        strcat((char *)insert_statement_nearest.arr, table_nearest);
                        strcat((char *)insert_statement_nearest.arr, " SELECT ");
                        strcat((char *)insert_statement_nearest.arr, _columns );
                        strcat((char *)insert_statement_nearest.arr, " FROM ");
                        strcat((char *)insert_statement_nearest.arr, _table);
                        strcat((char *)insert_statement_nearest.arr, " t WHERE rowid = '");
                        strcat((char *)insert_statement_nearest.arr, (char *)ins_rowid.arr);
                        strcat((char *)insert_statement_nearest.arr,"'");
                        insert_statement_nearest.len = strlen((char *)insert_statement_nearest.arr);
                        EXEC SQL PREPARE insert_sql_stmt_nearest FROM
                                                                :insert_statement_nearest;
                        EXEC SQL EXECUTE insert_sql_stmt_nearest;
                }
        }

        // Entferntesten Punkte in Tabelle schreiben
        vector_farest = cluster->getVectorFarest();
        for(i = vector_farest.begin(); i != vector_farest.end();i++){
                strncpy((char *)ins_rowid.arr,i->getRowId(), 18);
                ins_rowid.len = 18;

                if (create_farest){
                        strcat((char *)create_statement_farest.arr, (char *)ins_rowid.arr);
                        strcat((char *)create_statement_farest.arr,"'");
                        create_statement_farest.len = strlen((char *)create_statement_farest.arr);
                        EXEC SQL PREPARE create_sql_stmt_farest FROM
                                                                :create_statement_farest;
                        EXEC SQL EXECUTE create_sql_stmt_farest;
                        create_farest = false;
                } else {
                        //INSERT mit Rowid
```

```
                        strcpy((char *)insert_statement_farest.arr, "INSERT INTO ");
                        strcat((char *)insert_statement_farest.arr, table_farest);
                        strcat((char *)insert_statement_farest.arr, " SELECT ");
                        strcat((char *)insert_statement_farest.arr, _columns );
                        strcat((char *)insert_statement_farest.arr, " FROM ");
                        strcat((char *)insert_statement_farest.arr, _table);
                        strcat((char *)insert_statement_farest.arr, " t WHERE rowid = '");
                        strcat((char *)insert_statement_farest.arr, (char *)ins_rowid.arr);
                        strcat((char *)insert_statement_farest.arr,"'");
                        insert_statement_farest.len = strlen((char *)insert_statement_farest.arr);
                        EXEC SQL PREPARE insert_sql_stmt_farest FROM
                                                                :insert_statement_farest;
                        EXEC SQL EXECUTE insert_sql_stmt_farest;
                    }
                }
        }
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
        }
}

/**
 * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu Datenbasis
 */
void GetData::writeDatabaseInformation(char *table_database){
        int i = 0;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        // Create-Statment für Informationen zu jeder Dimension der zugrunde liegenden Tabelle
        strcpy((char *)create_statement_database.arr, "CREATE TABLE ");
        strcat((char *)create_statement_database.arr, table_database);
        strcat((char *)create_statement_database.arr, " (dimension INTEGER, sum FLOAT(126),
        sumsquared FLOAT(126), mean FLOAT(126), stddev FLOAT(126), minimum FLOAT(126),
        maximum FLOAT(126), median FLOAT(126), quantil25 FLOAT(126), quantil75 FLOAT(126),
        number_points LONG, PRIMARY KEY(dimension))");
        create_statement_database.len = strlen((char *)create_statement_database.arr);

        EXEC SQL PREPARE create_sql_stmt_database FROM :create_statement_database;
        EXEC SQL EXECUTE create_sql_stmt_database;

        // Insert-Statment für Informationen jeder Dimension
        strcpy((char *)insert_statement_database.arr, "INSERT INTO ");
        strcat((char *)insert_statement_database.arr, table_database);
        strcat((char *)insert_statement_database.arr, " (dimension, sum, sumsquared, mean, stddev,
        minimum, maximum, median, quantil25, quantil75, number_points) VALUES (:v1, :v2, :v3, :v4,
        :v5, :v6, :v7, :v8, :v9, :v10, :v11)");
        insert_statement_database.len = strlen((char *)insert_statement_database.arr);

        EXEC SQL PREPARE insert_sql_stmt_database FROM :insert_statement_database;

        for (i=0;i < _dimensions;i++){
                dimension     = _zaehler_dimension[i];
                sum           = _sum[i];
                sumsquared    = _sumSquared[i];
                mean          = _mean[i];
```

```
                stddev       = _stdDev[i];
                minimum      = _minimum[i];
                maximum      = _maximum[i];
                median       = _median[i];
                quantil25    = _quantil25[i];
                quantil75    = _quantil75[i];
                number_points = _numPoints;

                EXEC SQL EXECUTE insert_sql_stmt_database USING :dimension, :sum, :sumsquared,
                :mean, :stddev, :minimum, :maximum, :median, :quantil25, :quantil75, :number_points;
        }
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
        }
}

/**
 * Methode writeClusterInformation schreibt Hilfsinformationen zu ermittelten Clustern in Datenbank
 * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden sollen
 * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu Clustern
 */
void GetData::writeClusterInformation(Cluster *clusters, char *table_cluster){
        Cluster *cluster;
        int i = 0;
        double *h_sum, *h_sumsquared, *h_mean, *h_stddev, *h_minimum, *h_maximum;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        // Create-Statment für Informationen zu Clustern
        strcpy((char *)create_statement_cluster.arr, "CREATE TABLE ");
        strcat((char *)create_statement_cluster.arr, table_cluster);
        strcat((char *)create_statement_cluster.arr, " (clusterid INTEGER, dimension INTEGER, sum
        FLOAT(126), sumsquared FLOAT(126), mean FLOAT(126), stddev FLOAT(126), number_points
        LONG, sqrtnumpoints FLOAT(126), minimum FLOAT(126), maximum FLOAT(126), PRIMARY
        KEY(clusterid, dimension))");
        create_statement_cluster.len = strlen((char *)create_statement_cluster.arr);

        EXEC SQL PREPARE create_sql_stmt_cluster FROM :create_statement_cluster;
        EXEC SQL EXECUTE create_sql_stmt_cluster;

        // Insert-Statment für Informationen zu den Clustern
        strcpy((char *)insert_statement_cluster.arr, "INSERT INTO ");
        strcat((char *)insert_statement_cluster.arr, table_cluster);
        strcat((char *)insert_statement_cluster.arr, " (clusterid, dimension, sum, sumsquared, mean,
        stddev, number_points, sqrtnumpoints, minimum, maximum) VALUES (:v1, :v2, :v3, :v4, :v5, :v6,
        :v7, :v8, :v9, :v10)");
        insert_statement_cluster.len = strlen((char *)insert_statement_cluster.arr);

        EXEC SQL PREPARE insert_sql_stmt_cluster FROM :insert_statement_cluster;

        for(cluster = clusters; cluster; cluster = cluster->getNext()){
                h_sum       = cluster->getSum();
                h_sumsquared = cluster->getSumSquared();
                h_mean      = cluster->getMean();
                h_stddev    = cluster->getStdDev();
                h_minimum   = cluster->getMinimum();
                h_maximum   = cluster->getMaximum();
```

```
                for (i=0;i < _dimensions;i++){
                        dimension     = _zaehler_dimension[i];
                        clusterid     = cluster->getId();
                        sum           = h_sum[i];
                        sumsquared    = h_sumsquared[i];
                        mean          = h_mean[i];
                        stddev        = h_stddev[i];
                        minimum       = h_minimum[i];
                        maximum       = h_maximum[i];
                        number_points = cluster->getNumPoints();
                        sqrtnumpoints = cluster->getSqrtNumPoints();

                        EXEC SQL EXECUTE insert_sql_stmt_cluster USING :clusterid, :dimension, :sum,
                        :sumsquared, :mean, :stddev, :number_points, :sqrtnumpoints, :minimum,
                        :maximum;
                }
        }
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
        }
}

/**
 * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die Quantil-Berechnung
 * nach erster Iteration des Algorithmus (für 25% - Quantil, Median und 75% - Quantil
 */
void GetData::computeQuantilClasses(void){
        double difference, minimum;
        Quantil_Class quantil_class;

        for(int i = 0; i < _dimensions; i++){
                difference =  ( _maximum[i] - _minimum[i] ) / 100;
                minimum = _minimum[i];

                for(int j = 0; j < 100; j++){
                        quantil_class.setMinimum(minimum);
                        minimum+=difference;
                        if (j == 99)
                                quantil_class.setMaximum(_maximum[i]);
                        else
                                quantil_class.setMaximum(minimum);
                        _quantile[i][j] = quantil_class;
                }
        }
        _quantile25 = _quantile;
        _quantile75 = _quantile;
}

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen zu den Dimensionen des
 * Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsToQuantilClass(Point *point){
        this->orderPointsToMedianQuantilClass(point);
        this->orderPointsTo25QuantilQuantilClass(point);
        this->orderPointsTo75QuantilQuantilClass(point);
}
```

```
/**
* Methode orderPointsToMedianQuantilClass ermittelt zugehörige Quantil-Klassen für Median zu den
 * Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsToMedianQuantilClass(Point *point){
	double *vector = point->get(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile[i][j] = quantil;
				found = true;
			}
			j++;
		}
	}
}

/**
 * Methode orderPointsTo25QuantilQuantilClass ermittelt zugehörige Quantil-Klassen für 25%-Quantil
 * zu den Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsTo25QuantilQuantilClass(Point *point){
	double *vector = point->get(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile25[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile25[i][j] = quantil;
				found = true;
			}
			j++;
		}
	}
}

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen für 75%-Quantil zu den
 * Dimensionen des Punkts
```

```
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsTo75QuantilQuantilClass(Point *point){
	double *vector = point->get(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile75[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile75[i][j] = quantil;
				found = true;
			}
			j++;
		}
	}
}

/**
 * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile (25%, 50% und 75%) und
 * ermittelt pro Iteration
 * neue Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantilsWithQuantilClasses(void){
	this->findMedianWithQuantilClasses();
	this->findQuantil25WithQuantilClasses();
	this->findQuantil75WithQuantilClasses();
}

/**
 * Methode findMedianWithQuantilClasses ermittelt ungefähren Median und ermittelt pro Iteration
 * neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findMedianWithQuantilClasses(void){
	long median = _numPoints / 2;
	bool found;
	int j = 0;
	Quantil_Class quantil, quantil_class;
	vector<vector<Quantil_Class> > quantile(_dimensions,vector<Quantil_Class>(100));
	double difference, minimum;
	long allnumpoints;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		allnumpoints = 0;

		// Klasse mit Median ermitteln
		while(!found){
			quantil = _quantile[i][j];
			allnumpoints+= quantil.getNumPoints();		// kumulierten Werte

			if (allnumpoints >= median)			// Klasse mit Median gefunden
				found = true;
```

```
                j++;
            }
            _median[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;       // Ungefähre Median

            // Neue genauere Klassen für Median-Berechnung ermitteln
            difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
            minimum = quantil.getMinimum();

            // Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
            quantil_class.setMinimum(_quantile[i][0].getMinimum());
            quantil_class.setMaximum(minimum);              // Minimum der ermittelten Klasse
            quantile[i][0] = quantil_class;

            for(j = 1; j < 99; j++){
                quantil_class.setMinimum(minimum);
                minimum+=difference;
                quantil_class.setMaximum(minimum);
                quantile[i][j] = quantil_class;
            }

            // Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
            quantil_class.setMinimum(quantil.getMaximum());
            quantil_class.setMaximum(_quantile[i][99].getMaximum());
            quantile[i][99] = quantil_class;
        }
        _quantile.clear();
        _quantile = quantile;
}

/**
 * Methode findQuantil25WithQuantilClasses ermittelt ungefähres 25%-Quantil und ermittelt pro
 * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantil25WithQuantilClasses(void){
        long quantil25 = _numPoints / 4;
        bool found;
        int j = 0;
        Quantil_Class quantil, quantil_class;
        vector<vector<Quantil_Class> > quantile25(_dimensions,vector<Quantil_Class>(100));
        double difference, minimum;
        long allnumpoints;

        for(int i = 0; i < _dimensions; i++){
            found = false;
            j = 0;
            allnumpoints = 0;

            // Klasse mit 25%-Quantil ermitteln
            while(!found){
                quantil = _quantile25[i][j];
                allnumpoints+= quantil.getNumPoints();          // kumulierten Werte

                if (allnumpoints >= quantil25)                  // Klasse mit 25%-Quantil gefunden
                    found = true;
                j++;
            }
            // Ungefähres 25% - Quantil
            _quantil25[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;

            // Neue genauere Klassen für Median-Berechnung ermitteln
            difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
            minimum = quantil.getMinimum();
```

```
                // Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
                quantil_class.setMinimum(_quantile25[i][0].getMinimum());
                quantil_class.setMaximum(minimum);          // Minimum der ermittelten Klasse
                quantile25[i][0] = quantil_class;

                for(j = 1; j < 99; j++){
                        quantil_class.setMinimum(minimum);
                        minimum+=difference;
                        quantil_class.setMaximum(minimum);
                        quantile25[i][j] = quantil_class;
                }
                // Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
                quantil_class.setMinimum(quantil.getMaximum());
                quantil_class.setMaximum(_quantile25[i][99].getMaximum());
                quantile25[i][99] = quantil_class;
        }
        _quantile25.clear();
        _quantile25 = quantile25;
}

/**
 * Methode findQuantil75WithQuantilClasses ermittelt ungefähres 75%-Quantil und ermittelt pro
 * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantil75WithQuantilClasses(void){
        long quantil75 = ( _numPoints * 3 ) / 4;
        bool found;
        int j = 0;
        Quantil_Class quantil, quantil_class;
        vector<vector<Quantil_Class> > quantile75(_dimensions,vector<Quantil_Class>(100));
        double difference, minimum;
        long allnumpoints;

        for(int i = 0; i < _dimensions; i++){
                found = false;
                j = 0;
                allnumpoints = 0;

                // Klasse mit 75%-Quantil ermitteln
                while(!found){
                        quantil = _quantile75[i][j];
                        allnumpoints+= quantil.getNumPoints();          // kumulierten Werte

                        if (allnumpoints >= quantil75)                  // Klasse mit 75%-Quantil gefunden
                                found = true;
                        j++;
                }
                // Ungefähres 75 % - Quantil
                _quantil75[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;

                // Neue genauere Klassen für Median-Berechnung ermitteln
                difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
                minimum = quantil.getMinimum();

                // Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
                quantil_class.setMinimum(_quantile75[i][0].getMinimum());
                quantil_class.setMaximum(minimum);          // Minimum der ermittelten Klasse
                quantile75[i][0] = quantil_class;

                for(j = 1; j < 99; j++){
                        quantil_class.setMinimum(minimum);
                        minimum+=difference;
```

```
                quantil_class.setMaximum(minimum);
                quantile75[i][j] = quantil_class;
            }
            // Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
            quantil_class.setMinimum(quantil.getMaximum());
            quantil_class.setMaximum(_quantile75[i][99].getMaximum());
            quantile75[i][99] = quantil_class;
        }
        _quantile75.clear();
        _quantile75 = quantile75;
}
```

12.1.3 Klasse Point

Header-File der Klasse Point

```
/**
 * Klasse Point ist Friend von Cluster
 */
class Cluster;

/**
 * Überschrift: Klasse Point repräsentiert einen Punkt
 * Beschreibung: Die Klasse Point repräsentiert einen Punkt
 * mit den zugehörigen Information, wie Anzahl der Dimensionen, Werte der Dimensionen
 */
class Point{
    private:
        friend class Cluster;       /**< Klasse Cluster ist ein Freund von Point */

        double *_vector;            /**< Vektor mit Werten zu den Dimensionen */
        double _clusterDistance;/**< Distanz des Punkts zu seinem zugeordneten Cluster */
        int _dimensions;            /**< Anzahl der Dimensionen des Punkts */
        int _clusterid;             /**< Cluster ID des Punkts */
        char _rowid[18];            /**< RowId eines Punkts */

    public:
        /**
         * Konstruktor der Klasse Point
         * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
         */
        Point(int dimension);

        /**
         * 2 Konstruktor der Klasse Point
         * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
         * @param vector Vektor mit Werten der Dimensionen des Punkts
         * @param rowid Rowid der Datenbank des Punkts
         * @param clusterid Cluster ID des Punkts
         */
        Point(int dimension, double *vector, char *rowid, int clusterid) ;

        /**
         * Destruktor der Klasse Cluster
         */
        ~Point();

        /**
         * Methode set setzt die Werte der Dimensionen, die RowId und die Position der Cluster-Id
         * des Punkts
```

```
         * @param vector Vektor mit Werten der Dimensionen des Punkts
         * @param rowid Rowid der Datenbank des Punkts
         * @param clusterid Cluster ID des Punkts
         */
        void set(double *vector, char *rowid, int clusterid);

        /**
         * Methode get liefert Vektor mit Werten der Dimensionen des Punkts
         * @return Vektor mit Werten der Dimensionen
         */
        double *get(void){
                return _vector;
        }

        /**
         * Methode setClusterDistance setzt die Distanz des Punkts zu seinem Cluster
         * @param distance Distanz des Punkts zum Cluster
         */
        void setClusterDistance(double distance){
                _clusterDistance = distance;
        }

        /**
         * Methode getClusterDistance liefert Distanz des Punkts zu seinem zugeordneten Cluster
         * @return Distanz des Punkts zu seinem Cluster
         */
        double getClusterDistance(){
                return _clusterDistance;
        }

        /**
         * Methode getRowID liefert RowId des Punkts
         * @return RowId des Punkts
         */
        char *getRowID(){
                return _rowid;
        }

        /**
         * Methode getClusterId liefert Position der Dimension der Cluster Id
         * @return Position der Dimension der Cluster Id
         */
        int getClusterId(){
                return _clusterid;
        }

        /**
         * Methode getMemUsed liefert benötigten Speicher des Objekts in Bytes
         * @return Benötigter Speicher in Bytes
         */
        long getMemUsed(void){
                return sizeof(*this) + _dimensions*sizeof(double);
        }

        /**
         * Methode print gibt Punkt (mit allen Werten seiner Dimensionen) in das
         * übergebene File aus
         * @param file File in das der Punkt geschrieben wird
         */
        void print(FILE *file);
};
```

Source-File der Klasse Point

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "Point.h"

/**
 * Konstruktor der Klasse Point
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 */
Point::Point(int dimension) : _dimensions(dimension){
	_vector = new double[dimension];
	_clusterDistance = 0;
}

/**
 * 2 Konstruktor der Klasse Point
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 * @param vector Vektor mit Werten der Dimensionen des Punkts
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster ID des Punkts
 */
Point::Point(int dimension, double *vector, char *rowid, int clusterid) : _dimensions(dimension){
	_vector = new double[dimension];
	for(int i = 0; i < _dimensions; i++)
		_vector[i] = vector[i];
	strcpy(_rowid, rowid);
	_clusterid = clusterid;
	_clusterDistance = 0;
}

/**
 * Destruktor der Klasse Cluster
 */
Point::~Point(){
	delete []_vector;
}

/**
 * Methode set setzt die Werte der Dimensionen des Punkts
 * @param vector Vektor mit Werten der Dimensionen des Punkts
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster ID des Punkts
 */
void Point::set(double *vector, char *rowid, int clusterid){
	delete []_vector;
	_vector = new double[_dimensions];
	for(int i = 0; i < _dimensions; i++)
		_vector[i] = vector[i];
	strcpy(_rowid,rowid);
	_clusterDistance = 0;
	_clusterid = clusterid;
}

/**
 * Methode print gibt Punkt (mit allen Werten seiner Dimensionen) in das übergebene File aus
 * @param file File in das der Punkt geschrieben wird
 */
```

```
void Point::print(FILE *file){
	for(int i = 0; i < _dimensions; i++)
		fprintf(file, "%f ", _vector[i]);
	fprintf(file, "\n");
}
```

12.1.4 Klasse Cluster

Header-File der Klasse Cluster

```
/**
 * Klasse Point ist Friend von Cluster
 */
class Point;

/**
 * Überschrift: Klasse Cluster repräsentiert einen Cluster
 * Beschreibung: Die Klasse Cluster repräsentiert einen Cluster
 * mit den zugehörigen Information, wie Mittelpunkt, Anzahl der Punkte
 * Anzahl der Dimensionen usw.
 */
class Cluster{
	private:
		//  Hilfsinformationen
		double *_sum;		/**< Vektor mit Summe der Dimensionen der Punkte des Clusters */
		/**< Vektor mit Quadratsummen der Dimensionen der Punkte des Clusters */
		double *_sumSquared;
		double *_mean;			/**< Mittelpunkt des Clusters (Vektor mit Dimensionen) */
		double *_minimum;		/**< Minimum jeder Dimension */
		double *_maximum;		/**< Maximum jeder Dimension */
		/**< Standardabweichung der Punkte des Clusters (Vektor mit Dimensionen) */
		double *_stdDev;
		double _sqrtNumPoints;	/**< Wurzel über die Anzahl der Punkte */
		long _numPoints;		/**< Anzahl der Punkte des Clusters */
		/**< Vektor mit den _nearestNumPoints nähesten Punkten */
		vector<Information_Point> _vector_farest;
		/**< Vektor mit den _farestNumPoints entferntesten Punkten */
		vector<Information_Point> _vector_nearest;
		/**< Anzahl der nähesten Punkte des Clusters, die gespeicher werden sollen */
		int _nearestNumPoints;
		/**< Anzahl der am weitesten entfernten Punkte des Clusters, die gespeichert werden */
		int _farestNumPoints;
		int _id;			/**< ID des Clusters */
		int _dimensions;		/**< Anzahl der Dimensionen pro Punkt des Clusters */
		bool *_usedcolumns;		/**< Gibt an welche Spalten für Clustering relevant sind */
		FILE *_file_log;		/**< Log-File des Algorithmus */
		Cluster *_next, *_previous; /**< Zeiger auf die Vorgänger und Nachfolger Cluster */

		/**
		 * Methode findIndex ermittelt Index (über binäres Suchen) an dem der übergebene Punkt
		 * (Distanz) in die sortierte Liste eingefügt werden soll
		 * @param tofind Distanz des Punkts der in Liste eingefügt werden soll
		 * @param liste Liste in die Punkt eingefügt werden soll
		 * @return Index an dem Punkt eingefügt werden soll
		 */
		int findIndex(double tofind, vector<Information_Point> liste);

	public:
		/**
		 * Konstruktor der Klasse Cluster
		 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
```

```
 * @param nearestnumpoints Anzahl der nähesten Punkte des Clusters, die gespeichert
 *                                  werden sollen
 * @param farestnumpoints Anzahl der entferntesten Punkte des Clusters, die gespeichert
 *                                 werden sollen
 * @param id ID des Clusters
 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
 * @param file_log Log-File des Algorithmus
 */
Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, bool
          *usedcolumns,    FILE *file_log);

/**
 * Destruktor der Klasse Cluster
 */
~Cluster();

/**
 * Methode clear initialisiert Hilfsinformationen des Clusters für neue Iteration des Kmeans
 */
void clear(void);

/**
 * Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die Werte
 * des Clusters neu
 * @param point Punkt der zu Cluster hinzugefügt wird
 */
void addPoint(Point *point);

/**
 * Methode deletePoint löscht einen Punkt des Clusters und berechnet die Werte
 * des Clusters neu
 * @param point Punkt der aus Cluster gelöscht wird
 */
void deletePoint(Point *point);

/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void updateClusterMean(void);

/**
 * Die Methode distanceSquared liefert die Quadratische-Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
double distanceSquared(Point *point);

/**
 * Die Methode distanceEuklid liefert die Euklidische Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
double distanceEuklid(Point *point);

/**
 * Die Methode distanceManhatten liefert die Manhatten Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
```

```
double distanceManhatten(Point *point);

/**
 * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des Clusters
 * gehört. Die _nearestNumPoints werden in einer eigenen Liste gespeichert
 * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten gehört
 */
void addNearestPoint(Point *point);

/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des Clusters
 * gehört. Die _farestNumPoints werden in einer eigenen Liste gespeichert
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten Punkten gehört
 */
void addFarestPoint(Point *point);

/**
 * Methode print schreibt wichtige Werte des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void print(FILE *file = stdin);

/**
 * Methode printConfidence schreib Konfidenz des Clusters in das übergebene
 * File
 * @param file File in das Geschrieben wird
 */
void printConfidence(FILE *file = stdin);

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void computeStdDev(void);

/**
 * Methode getAllocated liefert die Anzahl der allokierten Cluster
 * @return Anzahl der allokierten Cluster
 */
static long getAllocated(void);

/**
 * Methode setPrevious setzt Vorgänger des Clusters
 * @param cluster Vorgängercluster
 */
void setPrevious(Cluster *cluster){
	_previous = cluster;
}

/**
 * Methode setNext setzt Nachfolger des Clusters
 * @param cluster Nachfolgercluster
 */
void setNext(Cluster *cluster){
	_next = cluster;
}

/**
 * Methode setNumPoints setzt die Anzahl der Punkte des Clusters
 * @param numpoints Anzahl der Punkte des Clusters
 */
void setNumPoints(int numpoints){
	_numPoints = numpoints;
```

```
}

/**
 * Methode setMean setzt Mittelpunkt des Clusters
 * @param mean Mittelpunkt des Clusters
 */
void setMean(double *mean){
	for(int i = 0; i < _dimensions; i++)
		_mean[i] = mean[i];
}

/**
 * Methode setSum setzt Summe der Werte der Dimensionen der zugeordneten Punkte
 * @param Summe der Werte der zugeordneten Punkte
 */
void setSum(double *sum){
	for(int i = 0; i < _dimensions; i++)
		_sum[i] = sum[i];
}

/**
 * Methode getPrevious liefert Vorgänger-Cluster des Clusters
 * @return Vorgänger-Cluster wird zurück gegeben
 */
Cluster *getPrevious(void){
	return _previous;
}

/**
 * Methode getNext liefert Nachfolger-Cluster des Clusters
 * @return Nachfolger-Cluster wird zurück gegeben
 */
Cluster *getNext(void){
	return _next;
}

/**
 * Methode getId liefert ID des Clusters
 * @return ID des Clusters
 */
int getId(void){
	return _id;
}

/**
 * Methode getMemUsed liefert benötigten Speicher des Objekts in Bytes
 * @return Benötigter Speicher in Bytes
 */
long getMemUsed(void){
	return sizeof(*this) + 4*_dimensions*sizeof(double);
}

/**
 * Methode getNumPoints liefert Anzahl der zugeordneten Punkte des Clusters
 * @return Anzahl der zugeordneten Punkte
 */
long getNumPoints(void){
	return _numPoints;
}

/**
 * Methode getSqrtNumPoints liefert Wurzel der Anzahl der zugeordneten Punkte des
```

```
 * Clusters
 * @return Wurzel der Anzahl der zugeordneten Punkte
 */
double getSqrtNumPoints(void){
	return _sqrtNumPoints;
}

/**
 * Methode getMean liefert Mittelpunkt des Clusters
 * @return Mittelpunkt des Clusters (Vektor mit Werten)
 */
double *getMean(void){
	return _mean;
}

/**
 * Methode getSum liefert Summe der Werte der Dimensionen der zugeordneten Punkte
 * @return Summe der Werte der zugeordneten Punkte
 */
double *getSum(void){
	return _sum;
}

/*
 * Methode getSumSquared liefert Quadratsumme der Dimensionen der Punkte des
 * Clusters
 * @return Quadratsumme der Dimensionen der Punkte
 */
double *getSumSquared(void){
	return _sumSquared;
}

/*
 * Methode getStdDev liefert Standardabweichung der Dimensionen der Punkte des
 * Clusters
 * @return Standardabweichung der Dimensionen der Punkte
 */
double *getStdDev(void){
	return _stdDev;
}

/*
 * Methode getMinimum liefert Minimum der Dimensionen der Punkte des Clusters
 * @return Minimum der Dimensionen der Punkte
 */
double *getMinimum(void){
	return _minimum;
}

/*
 * Methode getMaximum liefert Maximum der Dimensionen der Punkte des Clusters
 * @return Maximum der Dimensionen der Punkte
 */
double *getMaximum(void){
	return _maximum;
}

/*
 * Methode getVectorFarest liefert Vektor mit entferntesten Punkten zu Cluster
 * @return Vektor mit entferntesten Punkten des Clusters
 */
vector<Information_Point> getVectorFarest(){
```

```
                return _vector_farest;
        }

        /*
         * Methode getVectorNearest liefert Vektor mit nähesten Punkten des Clusters
         * @return Vektor mit nähesten Punkten des Clusters
         */
        vector<Information_Point> getVectorNearest(){
                return _vector_nearest;
        }
};
```

Source-File der Klasse Cluster

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include "Information_Point.h"
#include "Point.h"
#include "Cluster.h"

static int Allocated = 0;              /**< Anzahl der allokierten Cluster */

/**
 * Konstruktor der Klasse Cluster
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 * @param nearestnumpoints Anzahl der nähesten Punkte des Clusters, die gespeicher werden sollen
 * @param farestnumpoints Anzahl der entferntesten Punkte des Clusters, die gespeicher werden
 * @param id ID des Clusters
 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
 * @param file_log Log-File des Algorithmus
 */
Cluster::Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, bool *usedcolumns,
FILE *file_log) : _dimensions(dimension), _next(0), _nearestNumPoints(nearestnumpoints),
_farestNumPoints(farestnumpoints){
        // Erzeuge Double-Vektoren für die zu berechnenden Werte des Clusters
        _sum        = new double[dimension];
        _sumSquared = new double[dimension];
        _mean       = new double[dimension];
        _stdDev     = new double[dimension];
        _minimum    = new double[dimension];
        _maximum    = new double[dimension];
        _usedcolumns = new bool[_dimensions];
        _id         = id;
        _file_log   = file_log;

        for(int i = 0; i < dimension; i++){
                _sum[i] = 0;
                _sumSquared[i] = 0;
                _mean[i]    = 0;
                _stdDev[i]  = 0;
                _minimum[i] = 0;
                _maximum[i] = 0;
                _usedcolumns[i] = usedcolumns[i];
        }
```

```
	// Initialisiere Variablen des Clusters
	_numPoints     = 0;
	_sqrtNumPoints = 0;
	_vector_nearest.reserve(nearestnumpoints);
	_vector_farest.reserve(farestnumpoints);
	Allocated++;		// Anzahl der allokierten Cluster
}

/**
 * Destruktor der Klasse Cluster
 */
Cluster::~Cluster(){
	delete []_mean;
	delete []_sum;
	delete []_sumSquared;
	delete []_minimum;
	delete []_maximum;
	delete []_stdDev;
	delete []_usedcolumns;
	_vector_nearest.clear();
	_vector_farest.clear();
	Allocated--;
	if(Allocated < 0)
		fprintf(_file_log, "Too many calls to ~Cluster().\n");
	else if(Allocated == 0)
		fprintf(_file_log,"Destruktor ~Cluster() OK.\n");
}

/**
 * Methode getAllocated liefert die Anzahl der allokierten Cluster
 * @return Anzahl der allokierten Cluster
 */
long Cluster::getAllocated(void){
	return Allocated;
}

/**
 * Methode clear initialisiert Hilfsinformationen des Clusters für neue Iteration des Kmeans
 */
void Cluster::clear(void){
	for (int i=0;i < _dimensions; i++){
		_minimum[i] = 0;
		_maximum[i] = 0;
	}
	_vector_farest.clear();
	_vector_nearest.clear();
}

/**
 * Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die Werte
 * des Clusters neu
 * @param point Punkt der zu Cluster hinzugefügt wird
 */
void Cluster::addPoint(Point *point){
	// Werte des Clusters werden an lokale Zeiger übergeben
	double *vector = point->_vector, *sum = _sum, *sumSquared = _sumSquared;

	// Summen und Quadratsummen für jede Dimension werden berechnet
	for(int i = 0; i < _dimensions; i++){
		*sumSquared++ += *vector * *vector;
		*sum++ += *vector++;
	}
```

```
        _numPoints++;                // Anzahl der Punkte
}

/**
 * Methode deletePoint löscht einen Punkt des Clusters und berechnet die Werte des Clusters neu
 * @param point Punkt der aus Cluster gelöscht wird
 */
void Cluster::deletePoint(Point *point){
        // Werte des Clusters werden an lokale Zeiger übergeben
        double *vector = point->_vector, *sum = _sum, *sumSquared = _sumSquared;

        // Summen und Quadratsummen für jede Dimension werden neu berechnet
        for(int i = 0; i < _dimensions; i++){
                *sumSquared++ -= *vector * *vector;
                *sum++ -= *vector++;
        }
        _numPoints--;                // Anzahl der Punkte
}

/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void Cluster::updateClusterMean(void){
        double *sum = _sum, *mean = _mean, h_mean;

        // Mittelpunkt des Clusters wird neu berechnet
        if(_numPoints)
                for(int i = 0; i < _dimensions; i++){
                        h_mean = *sum++ / _numPoints;
                        *mean++ = h_mean;
                }
}

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void Cluster::computeStdDev(){
        double *sum = _sum, *sumSquared = _sumSquared, *l_stdDev = _stdDev, h_stddev;

        for(int i = 0; i < _dimensions; i++){
                h_stddev = *sum++ / _numPoints;
                *l_stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints - h_stddev * h_stddev));
        }
        _sqrtNumPoints = (double) sqrt(_numPoints);
}

/**
 * Methode print schreibt wichtige Werte des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void Cluster::print(FILE *file){
        double *mean = _mean;

        // Ausgabe des Mittelpunkts des Clusters
        fprintf(file,"%s", "Mittelpunkt des Clusters:");
        for(int i = 0; i < _dimensions; i++)
                fprintf(file, "%f", mean[i]);
        fprintf(file, "\n");
        fprintf(file, "%s %d\n","Anzahl der Punkte im Cluster:", _numPoints);
        fprintf(file, "%s %i\n","Anzahl der nähesten Punkte des Clusters:", _vector_nearest.size());
        if (!_vector_nearest.empty()){
                fprintf(file, "%s %i\n", "Anzahl der nähesten Punkte:", _vector_nearest.size());
```

```
                Information_Point info_point = _vector_nearest.front();
                fprintf(file, "%s %f\n","Distanz des nähesten Punkts des Cluster:",
                                                                info_point.getDistance());
                info_point = _vector_nearest.back();
                fprintf(file, "%s %f\n","Distanz des letzten nähesten Punkts des Cluster:",
                                                                info_point.getDistance());
        }
        if (!_vector_farest.empty()){
                fprintf(file, "%s %i\n", "Anzahl der entferntesten Punkte:", _vector_farest.size());
                Information_Point info_point = _vector_farest.front();
                fprintf(file, "%s %f\n","Distanz des letzten entferntesten Punkts des Cluster:",
                                                                info_point.getDistance());
                info_point = _vector_farest.back();
                fprintf(file, "%s %f\n","Distanz des entferntesten Punkts des Cluster:",
                                                                info_point.getDistance());
        }
        fprintf(file,"\n");
}

/**
 * Methode printConfidence schreib Konfidenz des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void Cluster::printConfidence(FILE *file){
        double *stdDev = _stdDev;

        for(int i = 0; i < _dimensions; i++)
                fprintf(file, "%f ", stdDev[i]/_sqrtNumPoints);
        fprintf(file, "\n");
}

/**
 * Die Methode distanceSquared liefert die Quadratische-Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Liefert die Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
double Cluster::distanceSquared(Point *point){
        double h_dist, distance = 0, *vector = point->_vector, *mean = _mean;

        for(int i = 0; i < _dimensions; i++){
                if (_usedcolumns[i]){
                        h_dist = *mean++ - *vector++;
                        distance += h_dist * h_dist;
                }
        }
        return distance;
}

/**
 * Die Methode distanceEuklid liefert die Euklidische Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Liefert die Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
double Cluster::distanceEuklid(Point *point){
        double h_dist, distance = 0, *vector = point->_vector, *mean = _mean;

        for(int i = 0; i < _dimensions; i++){
                if (_usedcolumns[i]){
                        h_dist = *mean++ - *vector++;
                        distance += h_dist * h_dist;
```

```
            }
        }
        return sqrt(distance);
}

/**
 * Die Methode distanceManhatten liefert die Manhatten Distanz zwischen Cluster-
 * Mittelpunkt und dem übergebenen Punkt
 * @param point Punkt zu dem Distanz berechnet wird
 * @return Liefert die Distanz zwischen Punkt und Mittelpunkt des Cluster
 */
double Cluster::distanceManhatten(Point *point){
        double h_dist, distance = 0, *vector = point->_vector, *mean = _mean;

        for(int i = 0; i < _dimensions; i++){
                if (_usedcolumns[i]){
                        h_dist = *mean++ - *vector++;
                        distance += fabs(h_dist);
                }
        }
        return distance;
}

/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des Clusters gehört
 * Die _farestNumPoints werden in einer eigenen Liste gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten Punkten gehört
 */
void Cluster::addFarestPoint(Point *point){
        Information_Point info_point(point->getRowID(),point->getClusterDistance());

        if (_vector_farest.empty())
                _vector_farest.push_back(info_point);
        else
                if (_vector_farest.size() < _farestNumPoints)
                        _vector_farest.insert(_vector_farest.begin() + findIndex(info_point.getDistance(),
                                                                        _vector_farest), info_point);
                else
                        if (_vector_farest.front().getDistance() < info_point.getDistance()){
                                _vector_farest.erase(_vector_farest.begin());
                                _vector_farest.insert(_vector_farest.begin() +
                                findIndex(info_point.getDistance(), _vector_farest), info_point);
                        }
}

/**
 * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des Clusters gehört
 * Die _nearestNumPoints werden in einer eigenen Liste gespeichert
 * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten gehört
 */
void Cluster::addNearestPoint(Point *point){
        Information_Point info_point(point->getRowID(), point->getClusterDistance());

        if (_vector_nearest.empty())
                _vector_nearest.push_back(info_point);
        else
                if (_vector_nearest.size() < _nearestNumPoints)
                        _vector_nearest.insert(_vector_nearest.begin() + findIndex(info_point.getDistance(),
                                                                        _vector_nearest), info_point);
                else
                        if (_vector_nearest.back().getDistance() > info_point.getDistance()){
```

```
                        _vector_nearest.pop_back();
                        _vector_nearest.insert(_vector_nearest.begin() +
                                findIndex(info_point.getDistance(), _vector_nearest), info_point);
                }
}

/**
 * Methode findIndex ermittelt Index (über binäres Suchen) an dem der übergebene Punkt (Distanz)
 * in die sortierte Liste eingefügt werden soll
 * @param tofind Distanz des Punkts der in Liste eingefügt werden soll
 * @param liste Liste in die Punkt eingefügt werden soll
 * @return Index an dem Punkt eingefügt werden soll
 */
int Cluster::findIndex(double tofind, vector<Information_Point> liste){
        int top = liste.size() - 1, bottom = 0, middle = int(liste.size() / 2);

        // Wenn die Liste einen oder zwei Punkte enthält wird die Sortierung ohne binäres Suchen
        // ermittelt

        // Liste enthält einen Punkt (Sortierung selbst durchführen)
        if (liste.size() == 1){
                if (liste.front().getDistance() < tofind)
                        return 1;
                else
                        return 0;
        }
        // Liste enthält zwei Punkte (Sortierung selbst durchführen)
        if (liste.size() == 2){
                if (liste.back().getDistance() < tofind)
                        return 2;
                else {
                        if (liste.front().getDistance() < tofind)
                                return 1;
                        else
                                return 0;
                }
        }
        // Algorithmus zum Auffinden des Index sodass Element sortiert eingefügt wird
        // Binäres Suchen
        do{
                Information_Point vector_middle = liste.at(middle);
                if (liste.at(middle).getDistance() <= tofind){
                        bottom = middle;
                        middle = int(middle + (top - middle) / 2);
                } else {
                        top = middle;
                        middle = int(middle - (middle - bottom) / 2);
                }
        } while(top - bottom > 1);

        // Rückgabe des Index an dem Wert eingefügt werden soll
        if (liste.at(bottom).getDistance() >= tofind)
                return bottom;
        else if (liste.at(top).getDistance() <= tofind)
                return top + 1;
        else
                return bottom + 1;
}
```

12.1.5 Klasse Quantil_Class

Header-File der Klasse Quantil_Class

```
/**
 * Überschrift: Klasse Quantil_Class repräsentiert eine Klasse zur Berechnung der Quantile
 * Beschreibung: Die Klasse Quantil_Class repräsentiert eine Klasse zur Berechnung der Quantile
 * (25%, 50% und 75%) verwendete wird
 */
class Quantil_Class{
	private:
		double _minimum;		/**< Untere Grenze der Klasse zur Ermittlung der Quantile */
		double _maximum;		/**< Obere Grenze der Klasse zur Ermittlung der Quantile */
		long _numPoints;		/**< Anzahl der Punkte in Klasse */

	public:
		Quantil_Class();

		/**
		 * Konstruktor der Klasse Quantil_Point
		 * @param minimum Untere Grenze der Klasse zur Ermittlung der Quantile
		 * @param maximum Obere Grenze der Klasse zur Ermittlung der Qunatile
		 */
		Quantil_Class(double minimum, double maximum);

		/**
		 * Destruktor der Klasse Information_Point
		 */
		~Quantil_Class();

		/**
		 * Methode addNumPoints erhöht die Anzahl der Punkte der Klasse um 1
		 */
		void addNumPoints(void){
			_numPoints++;
		}

		/**
		 * Methode setMinimum setzt untere Grenze der Klasse
		 * @param minimum Untere Grenze der Klasse
		 */
		void setMinimum(double minimum){
			_minimum = minimum;
		}

		/**
		 * Methode setMaximum setzt obere Grenze der Klasse
		 * @param maximum Obere Grenze der Klasse
		 */
		void setMaximum(double maximum){
			_maximum = maximum;
		}

		/**
		 * Methode getMinimum liefert die untere Grenze der Klasse
		 * @return Untere Grenze der Klasse
		 */
		double getMinimum(void){
			return _minimum;
		}

		/**
```

```
		 * Methode getMaximum liefert obere Grenze der Klasse
		 * @return Obere Grenze der Klasse
		 */
		double getMaximum(void){
			return _maximum;
		}

		/**
		 * Methode getNumPoints liefert die Anzahl der Punkte pro Klasse
		 * @return Anzahl der Punkte in Klasse
		 */
		long getNumPoints(void){
			return _numPoints;
		}
};
```

Source-File der Klasse Quantil_Class

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "quantil_class.h"

Quantil_Class::Quantil_Class(){
	_numPoints = 0;
}

/**
 * Konstruktor der Klasse Quantil_Class
 * @param minimum Untere Grenze der Klasse
 * @param maximum Obere Grenze der Klasse
 */
Quantil_Class::Quantil_Class(double minimum, double maximum){
	_minimum = minimum;
	_maximum = maximum;
	_numPoints = 0;
}

/**
 * Destruktor der Klasse Quantil_Class
 */
Quantil_Class::~Quantil_Class(){}
```

12.1.6 Klasse Information_Point

Header-File der Klasse Information_Point

```
/**
 * Überschrift: Klasse Information_Point repräsentiert einen Punkt zur Berechnung der
 * Hilfsinformationen
 * Beschreibung: Die Klasse Information_Point repräsentiert einen Punkt der zur Berechnung der
 * Hilfsinformationen (näheste Punkte, weit entfernteste Punkte) verwendet wird
 */
class Information_Point{
	private:
		char _rowid[18];			/**< RowID eine Punkts */
		double _distance;			/**< Distanz des Punkts zu seinem Cluster */
```

```
        public:
                /**
                 * Konstruktor der Klasse Information_Point
                 * @param rowid RowId eines Punkts
                 * @param distance Distanz eines Punkts zu seinem Cluster
                 */
                Information_Point(char *rowid, double distance);

                /**
                 * Destruktor der Klasse Information_Point
                 */
                ~Information_Point();

                /**
                 * Methode getRowid liefert die RowID des Punkts
                 * @return RowId des Punkts
                 */
                char *getRowId(void);

                /**
                 * Methode getDistance liefert Distanz des Punkts zu seinem Cluster
                 * @return Distanz zum Cluster
                 */
                double getDistance(void);
};
```

Source-File der Klasse Information_Point

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "information_point.h"

/**
 * Konstruktor der Klasse Information_Point
 * @param rowid Rowid des Punkts aus Datenbank
 * @param distance Distanz des Punkts zu seinem Centroiden
 */
Information_Point::Information_Point(char *rowid, double distance){
        strcpy(_rowid,rowid);
        _distance = distance;
}

/**
 * Destruktor der Klasse Information_Point
 */
Information_Point::~Information_Point(){}

/**
 * Methode getRowid liefert die RowID des Punkts
 * @return RowId des Punkts
 */
char *Information_Point::getRowId(){
        return _rowid;
}

/**
```

```
 * Methode getDistance liefert Distanz des Punkts zu seinem Centroiden
 * @return Distanz zum Cluster
 */
double Information_Point::getDistance(){
        return _distance;
}
```

12.1.7 Klasse KMeans

Header-File der Klasse KMeans

```
/**
 * Die Klassen Cluster, Database und Point sind Friend von Kmeans
 */
class Cluster;
class Database;
class Point;
/**
 * Überschrift: Klasse Kmeans die Kmeans-Algorithmus ausführt
 * Beschreibung: Klasse Kmeans führt Kmeans-Algorithmus aus. Das Ergebnis
 * des Algorithmus wird in den Clustern gespeichert
 */
class KMeans{
        private:
                Database *_database;      /**< Datenbasis des Algorithmus */
                int _numClusters;         /**< Anzahl der Cluster = Parameter k */
                int _dimensions;          /**< Anzahl der Dimensionen pro Punkt */
                int _numIterations;       /**< Anzahl der ausgeführten Iterationen */
                int _distancefunction;/**< Distanzfunktion (1=Quadratisce,2=Euklidische,3=Manhatten) */
                Cluster *_clusters;   /**< Enthält numClusters Cluster in denen Ergebnis gespeichert wird */
                FILE *_file_log;          /**< Log-File des Algorithmus */
                int *_initialPoints;      /**< Initiale Punkte für Cluster-Mittelpunkte */
                int _numberinitialiterations; /**< Anzahl der maximalen Iterationen bei Initialisierung */
                int _numberiterations;    /**< Anzahl der maximalen Iterationen */

        protected:
                /**
                 * Methode setup erzeugt numCluster Clusters für den Algorithmus
                 * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer
                 *                         eigenen Liste gespeichert werden
                 * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer
                 *                        eigenen Liste gespeichert werden
                 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
                 */
                void setup(int nearestnumpoints, int farestnumpoints, bool *usedcolumns);

                /**
                 * Methode initModel initialisiert alle Cluster mit einer per Zufallsgenerator ermittelten
                 * Anzahl von Punkten
                 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
                 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
                 * @return Gesamte Anzahl der initialen Punkte aller Cluster
                 */
                long initModel(long initnumpointsmax, long initnumpointsmin);

                /**
                 * Methode initModelwithClustering führt initiales Clustering mit einer zufällig ermittelten
                 * Anzahl von Punkten durch. Das initiale Clusteringergebnis wird als Ausgangsbasis für
                 * den Kmeans-Algorithmus herangezogen
                 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
                 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
```

```
        */
        void initModelwithClustering(long initnumpointsmax, long initnumpointsmin);

        /**
         * Methode closestCluster ermittelt nähesten Cluster zu einem Punkt
         * @param point Punkt zu dem nähestes Cluster ermittelt wird
         * @return Nähestes Cluster zu Punkt point
         */
        Cluster *closestCluster(Point *point);

        /**
         * Methode getCluster liefert Cluster zu einer Id
         * @return Cluster zu übergebener Id
         */
        Cluster *getCluster(int id);

public:
        /**
         * Konstruktor der Klasse Kmeans
         * @param datenbank Datenbasis die Daten für Kmeans-Algorithmus liefert
         * @param anz_cluster Anzahl der zu ermittelten k Cluster
         * @param dimension Anzahl der Dimensionen pro Punkt
         * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer
         *                                      eigenen Liste gespeichert werden
         * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer
         *                                    eigenen Liste gespeichert werden
         * @param distancefunction Gibt an welche Distanzfunktion verwendet werden soll
         *                                    (1=Quadratisce,2=Euklidische,3=Manhatten)
         * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
         * @param file_log Log-File des Algorithmus
         * @param initialpoints Initiale Punkte für Cluster-Mittelpunkte
         * @param numberinitialiterations Anzahl der maximalen Iterationen bei Initialisierung
         * @param numberiterations Anzahl der maximalen Iterationen
         */
        KMeans(Database *datenbank, int anz_cluster, int dimension, int nearestnumpoints, int
        farestnumpoints, int distancefunction, bool *usedcolumns, FILE *file_log, int *initialpoints,
        int numberinitialiterations, int numberiterations);

        /**
         * Destruktor der Klasse Kmeans
         */
        ~KMeans();

        /**
         * Methode cluster führt Kmeans-Algorithmus aus
         * @param numpoints Anzahl der Punkte die für den Algorithmus herangezogen werden
         *                                (optional)
         * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
         * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
         */
        void cluster(long numpoints = 0,long initnumpointsmax, long initnumpointsmin);

        /**
         * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste
         * Punkte in Datenbank
         * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der
         *                              Cluster
         * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
         */
        void writeClusterNearestandFarestToDatabase(char *table_nearest, char *table_farest);

        /**
```

```
        * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in
        * Datenbank
        * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu
        *                       Datenbasis
        */
        void writeDatabaseInformation(char *table_database);

        /**
        * Methode writeClusterInformation schreibt Hilfsinformationen zu ermittelten Clustern in
        * Datenbank
        * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden
        *                 sollen
        * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu
        *                      Clustern
        */
        void writeClusterInformation(char *table_cluster);

        /**
        * Methode getNumIterations liefert die Anzahl der ausgeführten Iterationen des
        * Algorithmus
        * @return Anzahl der ausgeführten Iterationen
        */
        int getNumIterations(void){
                return _numIterations;
        }

        /**
        * Ergebnis des Clusters (Mittelpunkt, Standardabweichung, Anzahl der Punkte)
        * wird in File geschrieben
        * @param filename File in das Ergebnisdaten des Clusters geschrieben werden
        */
        void writeClusterToFile(char *filename);
};
```

Source-File der Klasse KMeans

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Point.h"
#include "Information_Point.h"
#include "kmeans.h"
#include "Cluster.h"
#include "database.h"
#include <time.h>

const ID_UNCLASSIFIED = -1;       /**< Initiale Cluster Id für unklassifizierte Objekte */

/**
 * Konstruktor der Klasse Kmeans
 * @param datenbank Datenbasis die Daten für Kmeans-Algorithmus liefert
 * @param anz_cluster Anzahl der zu ermittelten k Cluster
 * @param dimension Anzahl der Dimensionen pro Punkt
 * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer eigenen Liste
 *                         gespeichert werden
 * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer eigenen Liste
```

```
 *                                  gespeichert werden
 * @param distancefunction Gibt an welche Distanzfunktion verwendet werden soll (1=Quadratisce,
 *                                  2=Euklidische, 3=Manhatten)
 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
 * @param file_log Log-File des Algorihtmus
 * @param initialpoints Initiale Punkte für Cluster-Mittelpunkte
 * @param numberinitialiterations Anzahl der maximalen Iterationen bei Initialisierung
 * @param numberiterations Anzahl der maximalen Iterationen
 */
KMeans::KMeans(Database *datenbank, int anz_cluster, int dimension, int nearestnumpoints, int
farestnumpoints, int distancefunction, bool *usedcolumns, FILE *file_log, int *initialpoints, int
numberinitialiterations, int numberiterations) : _database(datenbank), _numClusters(anz_cluster),
                    _dimensions(dimension), _distancefunction(distancefunction), _file_log(file_log),
                _numberinitialiterations(numberinitialiterations), _numberiterations(numberiterations){
        setup(nearestnumpoints, farestnumpoints, usedcolumns);        // Erzeuge _numClusters Cluster
        _initialPoints = new int[anz_cluster];

        for (int i=0; i < anz_cluster; i++)
                _initialPoints[i] = initialpoints[i];
}

/**
 * Destruktor der Klasse Kmeans
 */
KMeans::~KMeans(){
        Cluster *cluster, *nextcluster;
        for(cluster = _clusters; cluster; cluster = nextcluster){
                nextcluster = cluster->getNext();
                delete cluster;
        }
        delete _database;
}

/**
 * Methode setup erzeugt numCluster Clusters für den Algorithmus
 * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer eigenen Liste
 *                                  gespeichert werden
 * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer eigenen Liste
 *                                  gespeichert werden
 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
 */
void KMeans::setup(int nearestnumpoints, int farestnumpoints, bool *usedcolumns){
        Cluster *newCluster;              // Hilfsvariable für einen Cluster
        _clusters = 0;              // Verkettete Liste mit Clustern initialisieren

        // Schleife über Anzahl der Cluster
        for(int i = 0; i < _numClusters; i++){
                // Erzeuge neuen Cluster (mit Anzahl der Dimensionen)
                newCluster = new Cluster(_dimensions, nearestnumpoints, farestnumpoints, i + 1,
                                                                                usedcolumns, _file_log);
                // Setzt vorherigen Cluster (falls schon vorhanden)
                if(_clusters)
                        _clusters->setPrevious(newCluster);
                // Setzt nächsten Cluster
                newCluster->setNext(_clusters);
                _clusters = newCluster;
        }
}

/**
 * Methode cluster führt Kmeans-Algorithmus aus
 * @param numpoints Anzahl der Punkte die für den Algorithmus herangezogen werden (optional)
```

```
 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
 */
void KMeans::cluster(long numpoints, long initnumpointsmax, long initnumpointsmin){
	long time_it_start, time_it_end, time_init_start, time_init_end;
	Point *point;
	Cluster *cluster, *oldcluster;
	long count, changecount;
	bool changeCluster;
	long promilegrenze;

	if (numpoints)
		promilegrenze = numpoints / 1000;

	// Cluster mit beliebigen Punkten initialisieren
	fprintf(_file_log,"Cluster initialisieren. \n");
	fflush(_file_log);
	time_init_start = clock();
	initModelwithClustering(initnumpointsmax, initnumpointsmin);
	time_init_end = clock();
	fprintf(_file_log, "%s %s %f \n","Initialisierung fertig.", "Dauer:", (time_init_end -
	time_init_start)/(float)CLOCKS_PER_SEC);
	fflush(_file_log);

	_numIterations = 0;		// Variable mit Anzahl der Iterationen initialisieren
	// In erstem Schleifendurchlauf sollen Hilfsinformationen für Datenbasis berechnet werden
	_database->setcomputeHelpData(true);

	do{
		time_it_start = clock();
		changeCluster = false;
		changecount = 0;
		_numIterations++;

		fprintf(_file_log,"%s %i \n", "Iteration: ", _numIterations);
		fflush(_file_log);
		_database->reset();			// Datenquelle zurücksetzen

		// Für die Berechnung der Quantile werden die einzelnen Dimensionen in 100 Klassen
		// geteilt
		if (_numIterations == 2)
			_database->computeQuantilClasses();

		// Listen mit Hilfsinformationen werden initialisiert
		for(cluster = _clusters; cluster; cluster = cluster->getNext()){
			cluster->clear();
		}

		count = 0;		//Zählvariable (Anzahl der Punkte)
		// Solange noch ein Punkt vorhanden ist und die Anzahl der Punkte
		// unter der max. Anzahl der Punkte liegt wird Schleife durchgeführt
		point = _database->getPoint();
		while((_database->getData()) && ((count++ < numpoints) || (numpoints == 0))){
			// Liefert "nähestes" Cluster zu Punkt point
			cluster = closestCluster(point);

			if (!cluster){
				fprintf(_file_log,"No Cluster found. \n");
				fflush(_file_log);
			}else{
				if (cluster->getId() != point->getClusterId()){
					changeCluster = true;
```

```
                                changecount++;

                                if (point->getClusterId() > 0){
                                        oldcluster = getCluster(point->getClusterId());
                                        oldcluster->deletePoint(point);
                                        oldcluster->updateClusterMean();
                                }
                                // Fügt Punkt zu nähestem Cluster hinzu
                                cluster->addPoint(point);

                                _database->updateClusterId(point->getRowID(),cluster->getId());

                                // Berechnet Mittelpunkt des Clusters neu
                                cluster->updateClusterMean();
                        }

                        // Hilfsinformationen pro Cluster werden berechnet
                        if (_numIterations > 1){
                                cluster->addNearestPoint(point);
                                cluster->addFarestPoint(point);
                                // Klassen zur Quantil-Berechnung füllen
                                _database->orderPointsToQuantilClass(point);
                        }
                }
                if (count < numpoints)
                        point = _database->getPoint();
        }

        // Quantile (25%, 50% und 75%) werden näherungsweise ermittelt (nach jeder Iteration
        // genauer)
        if (_numIterations > 1)
                _database->findQuantilsWithQuantilClasses();

        fprintf(_file_log,"%s %i \n", "Anzahl Änderungen der Zuordnungen:", changecount);
        fflush(_file_log);
        // Nur in 1 Schleifendurchlauf HIlfsinformationen berechnen
        _database->setcomputeHelpData(false);
        // Wenn Anzahl der Änderungen unter Promilegrenze, dann ist Clustering fertig
        if (changecount <= promilegrenze)
                        changeCluster = false;

        if (_numIterations >= _numberiterations){
                changeCluster = false;
                fprintf(_file_log, "Manuelles Beenden des Clustering. \n");
        }
        time_it_end = clock();
        fprintf(_file_log,"%s %f \n","Dauer:",(time_it_end –
                                                time_it_start)/(float)CLOCKS_PER_SEC);
    }while(changeCluster);

    _database->commitDatabase();
    fprintf(_file_log, "Clustering fertig. \n");
    fflush(_file_log);

    // Berechnung der Standardabweichung für jedes Cluster
    for(cluster = _clusters; cluster; cluster = cluster->getNext()){
        cluster->computeStdDev();
    }
}

/**
 * Methode initModel initialisiert alle Cluster mit einer per Zufallsgenerator ermittelten Anzahl
```

```
 * von Punkten
 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
 * @return Gesamte Anzahl der initialen Punkte aller Cluster
 */
long KMeans::initModel(long initnumpointsmax, long initnumpointsmin){
	Cluster *cluster;
	Point *point;
	int i, j, k;
	long initnumPoints = 0;

	_database->reset();

	/* Seed the random-number generator with current time so that
	 * the numbers will be different every time we run.
	 */
	srand( (unsigned)time( NULL ) );

	k = 0;
	// Schleife über alle Cluster
	for(cluster = _clusters; cluster; cluster = cluster->getNext()){
		// Zufallszahl ermitteln (abhängig von Parameter initnumpointsmax)
		if (_initialPoints[k] > 0)
			j = _initialPoints[k];
		else {
			j = 0;
			while (j < initnumpointsmin){
				j = rand() % initnumpointsmax;
			}
		}
		initnumPoints+= j;

		// Lest Punkt aus Datenbasis
		for(i = 0; i <= j; i++){
			point = _database->getPoint();
		}

		// 1 Punkt wird zufällig als Mittelpunkt des Clusters ausgewählt
		cluster->addPoint(point);
		_database->updateClusterId(point->getRowID(),cluster->getId());
		fprintf(_file_log,"%s %i %s %i \n", "Cluster:", cluster->getId(), "Punkt:", j);
		// Mittelpunkt des Clusters berechnen
		cluster->updateClusterMean();
		k++;
	}
	_database->commitDatabase();
	return initnumPoints;
}

/**
 * Methode initModelwithClustering führt initiales Clustering mit einer zufällig ermittelten
 * Anzahl von Punkten durch. Das optimale initiale Clustering wird als Ausgangsbasis für den
 * Kmeans-Algorithmus herangezogen
 * @param initnumpointsmax Max. Anzahl der initialen Punkte pro Cluster
 * @param initnumpointsmin Min. Anzahl der initialen Punkte pro Cluster
 */
void KMeans::initModelwithClustering(long initnumpointsmax, long initnumpointsmin){
	Point *point;
	Cluster *cluster, *oldcluster;
	long count, changecount = 0;
	bool changeCluster;
	int numIterations = 0;
```

```
        // Cluster mit beliebigen Punkten initialisieren
        long initnumPoints = initModel(initnumpointsmax, initnumpointsmin);

        do{
                changeCluster = false;
                // Datenquelle zurücksetzen
                _database->reset();
                changecount = 0;

                numIterations++;
                fprintf(_file_log,"%s %i", "It.:", numIterations);
                fflush(_file_log);

                count = 0;        //Zählvariable (Anzahl der Punkte)
                // Solange noch ein Punkt vorhanden ist und die Anzahl der Punkte
                // unter der max. Anzahl der Punkte liegt wird Schleife durchgeführt
                point = _database->getPoint();
                while((_database->getData()) && (count++ < initnumPoints)){
                        // Liefert "nähestes" Cluster zu Punkt
                        cluster = closestCluster(point);

                        if (!cluster)
                                fprintf(_file_log,"No Cluster found");
                        else
                                if (cluster->getId() != point->getClusterId()){
                                        changeCluster = true;
                                        changecount++;
                                        if (point->getClusterId() > 0){
                                                oldcluster = getCluster(point->getClusterId());
                                                oldcluster->deletePoint(point);
                                                oldcluster->updateClusterMean();
                                        }
                                        // Fügt Punkt zu nähestem Cluster hinzu
                                        cluster->addPoint(point);

                                        _database->updateClusterId(point->getRowID(),cluster->getId());

                                        // Berechnet Mittelpunkt des Clusters neu
                                        cluster->updateClusterMean();
                                }
                        if (count < initnumPoints)
                                point = _database->getPoint();
                }
                fprintf(_file_log,"%s %i \n", " Änd.:", changecount);
                fflush(_file_log);

                if (numIterations >= _numberinitialiterations){
                        changeCluster = false;
                        fprintf(_file_log, "Manuelles Beenden der Initialisierung nach 41 Iterationen. \n");
                }
        } while(changeCluster);

        _database->commitDatabase();
        fprintf(_file_log,"%s %i \n","Anzahl der Iterationen bei Initialisierung:", numIterations);
}

/**
 * Methode closestCluster ermittelt nähesten Cluster zu einem Punkt
 * @param point Punkt zu dem nähestes Cluster ermittelt wird
 * @return Nähestes Cluster zu Punkt point
 */
```

```
Cluster* KMeans::closestCluster(Point *point){
	double distance = 1e20, h_distance;
	Cluster *m, *cluster;
	bool found = false;
	double *minimum, *maximum, *vector;

	// Ermittelt nähesten Cluster zu einem Punkt
	for(m = _clusters; m; m = m->getNext()){
		switch( _distancefunction){
			case 1:                                  // Quadratische Distanz
				h_distance = m->distanceSquared(point);
				break;
			case 2:                                  // Euklidische Distanz
				h_distance = m->distanceEuklid(point);
				break;
			case 3:                                  // Manhatten Distanz
				h_distance = m->distanceManhatten(point);
				break;
		}
		if(h_distance < distance){
			cluster = m;
			distance = h_distance;
			found = true;
		}
	}

	if (found){
		minimum = cluster->getMinimum();
		maximum = cluster->getMaximum();
		vector  = point->get();
		// Minimum und Maximum pro Cluster ermitteln
		for (int i=0;i<_dimensions;i++){
			// Minimum
			if (minimum[i] == 0)
				minimum[i] = vector[i];
			else if (vector[i] < minimum[i])
				minimum[i] = vector[i];
			// Maximum
			if (maximum[i] == 0)
				maximum[i] = vector[i];
			else if (vector[i] > maximum[i])
				maximum[i] = vector[i];
		}

		point->setClusterDistance(distance);
		return cluster;
	} else
		return 0;
}

/**
 * Methode getCluster liefert Cluster zu einer Id
 * @return Cluster zu übergebener Id
 */
Cluster* KMeans::getCluster(int id){
	Cluster *m, *cluster;

	for(m = _clusters; m; m = m->getNext())
		if(m->getId() == id)
			cluster = m;

	return cluster;
```

```
}

/**
 * Ergebnis des Clusters (Mittelpunkt, Standardabweichung, Anzahl der Punkte)
 * wird in File geschrieben
 * @param filename File in das Ergebnisdaten des Clusters geschrieben werden
 */
void KMeans::writeClusterToFile (char *filename){
        FILE *clusterfile;
        Cluster *cluster;
        int i = 1;

        clusterfile = fopen(filename, "a");

        for(cluster = _clusters; cluster; cluster = cluster->getNext())         {
                fprintf(clusterfile, "%s %d\n", "Cluster:", cluster->getId());
                cluster->print(clusterfile);
        }
        fclose(clusterfile);
}

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste Punkte in
 * Datenbank
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
 */
void KMeans::writeClusterNearestandFarestToDatabase(char *table_nearest, char *table_farest){
        _database->writeClusterNearestandFarestToDatabase(_clusters,table_nearest,table_farest);
}

/**
 * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu Datenbasis
 */
void KMeans::writeDatabaseInformation(char *table_database){
        _database->writeDatabaseInformation(table_database);
}

/**
 * Methode writeClusterInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden sollen
 * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu Clustern
 */
void KMeans::writeClusterInformation(char *table_cluster){
        _database->writeClusterInformation(_clusters, table_cluster);
}
```

12.2 DBSCAN Algorithmus

12.2.1 Abstrakte Klasse Database

Header-File der Klasse Database

```
/**
 * Überschrift: Allgemeine Klasse Database mit Interface für zu clusternde Datenbestände
 * Beschreibung: Allgemeine Klasse für die zu clusternden Datenbestände. Für jeden Daten-
 * bestand muss eine eigene Klasse angelget werden die von Database erbt und dessen virtuelle
 * Methoden implementiert
```

```
 */
class Database{
	public:
		/**
		 * Methode getPoint liefert nächsten Punkt der Datenbasis
		 * @return Liefert nächsten Punkt der Datenbasis
		 */
		virtual Point *getPoint(void) = 0;

		/**
		 * Methode getNeighbourhood liefert alle Punkte der Epsilon-Umgebung des übergebenen
		 * Punkts mit dem übergebenen Epsilon-Wert
		 * @param point Punkt zu dem Epsilon-Umgebung berechnet wird
		 * @param epsilon Wert des Epsilon
		 * @return Punkte in Epsilon-Umgebung des übergebenen Punkts
		 */
		virtual list<Point> getNeighbourhood(Point *point, double epsilon) = 0;

		/**
		 * Methode updateClusterId führt Update der Clusterid eins Punkts auf Datenbank durch.
		 * Der Punkt wird über seine Rowid identifiziert und sein zugehöriges Cluster wird mit
		 * übergeben
		 * @param rowid Rowid der Datenbank des Punkts
		 * @param clusterid Cluster dem Punkt hinzugefügt wurde
		 */
		virtual void updateClusterId(Point *point, int clusterid) = 0;

		/**
		 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen ClusterId
		 * @param clusterid Initiale Clusterid
		 * @param nameclusterid Name der Spalte mit Cluster Id
		 */
		virtual void initClusterIds(char *clusterid, char *nameclusterid) = 0;

		/**
		 * Methode getData gibt an ob Ausführung der Methode getPoint (Ermittlung der Werte des
		 * nächsten Punkts) erfolgreich war
		 * @return Boolean-Wert der angibt ob Ermittlung der Werte der Dimensionen des Punkts
		 *		erfolgreich war
		 */
		virtual bool getData(void) = 0;

		/**
		 * Methode setgetData setzt Boolean-Variable getData, die angibt ob das Lesen des
		 * nächsten Punkts aus der Datenbasis erfolgreich war
		 * @param getData Gibt an ob Lesen des nächsten Punkts erfolgreich war oder nicht
		 */
		virtual void setgetData(bool getData) = 0;

		/**
		 * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die Quantil-
		 * Berechnung
		 */
		virtual void computeQuantilClasses(void) = 0;

		/**
		 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen zu den
		 * Dimensionen des Punkts
		 * @param point Punkt der Quantil-Klassen zugeordnet wird
		 */
		virtual void orderPointsToQuantilClass(Point *point) = 0;
```

```
        /**
         * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile (25%, 50% und
         * 75%) und ermittelt pro Iteration neue Klassen --> genauere Quantile nach jeder Iteration
         */
        virtual void findQuantilsWithQuantilClasses(void) = 0;

        /**
         * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste
         * Punkte in Datenbank
         * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank
         *                              geschrieben werden sollen
         * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der
         *                               Cluster
         * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
         */
        virtual void writeClusterNearestandFarestToDatabase(Cluster *clusters,char
                                                        *table_nearest,char *table_farest) = 0;

        /**
         * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in
         * Datenbank
         * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                                Datenbasis
         */
        virtual void writeDatabaseInformation(char *table_database) = 0;

        /**
         * Methode writeClusterInformation schreibt Hilfsinformationen zu ermittelten Cluster in
         * Datenbank
         * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden
         * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                              Clustern
         */
        virtual void writeClusterInformation(Cluster *clusters, char *table_cluster) = 0;
};
```

12.2.2 Klasse GetData

Das Source-File der Klasse GetData enthält zusätzlich zu den Implementierungen der Methoden der Klasse GetData auch die Methode *main*, die das Programm startet. Der Quellcode des Source-Files ist der Quellcode vor dem PreCompiler Schritt.

Header-File Klasse GetData

```
/**
 * Überschrift: Klasse GetData zum Lesen der Daten aus Datenbank
 * Beschreibung: Klasse GetData liefert Daten aus Datenbank um
 * Cluster-Algorithmus (Kmeans) auszuführen
 */
class GetData : public Database{
    private:
        double *_data;      /**< Vektor mit Werten zu einer Dimension */

        //  Hilfsinformationen
        double *_sum;           /**< Summe über alle Werte jeder Dimension */
        double *_sumSquared;    /**< Vektor mit Quadratsummen jeder Dimensionen */
        double *_mean;          /**< Mittelwert der Werte jeder Dimension */
        double *_stdDev;        /**< Standardabweichung der Werte jeder Dimension */
        double *_minimum;       /**< Minimum jeder Dimension */
```

```
double *_maximum;       /**< Maximum jeder Dimension */
double *_median;        /**< Ungefährer Median (Schätzverfahren) */
double *_quantil25;     /**< Ungefähres 25% - Quantil (Schätzverfahren) */
double *_quantil75;     /**< Ungefähres 75% - Quantil (Schätzverfahren) */
long _numPoints;        /**< Anzahl der gelesenen Punkte aus Datenbank */

/**< Hilfsvektor zum Berechnung des Medians (50% - Quantil) */
vector<vector<Quantil_Class> > _quantile;
/**< Hilfsvektor zum Berechnung des 25% - Quantils */

vector<vector<Quantil_Class> > _quantile25;
/**< Hilfsvektor zum Berechnung des 75% - Quantil2 */
vector<vector<Quantil_Class> > _quantile75;
int *_zaehler_dimension;  /**< Gibt an welcher Tabellenspalte die Dimension entspricht */
long _clusterPoints;      /**< Anzahl der zu clusternden Punkte */
int _dimensions;          /**< Anzahl der Dimensionen pro Punkt des Datenbestands */
int _clusterid;           /**< Position der Dimension der Cluster Id */
int _distancefunction;/**< Distanzfunktion (1=Quadratisce,2=Euklidische,3=Manhatten) */
/**< Boolean Wert der angibt ob letztes Fetch auf Datenbank erfolgreich war */
bool _getData;
Point *_point;            /**< Hilfspunkt für Funktion getPoint */
char *_table;             /**< Tabelle der Datenbasis aus der Daten gelesen werden */
/**< Spalten der Tabelle die für Algorithmus berücksichtigt werden */
char *_columns;
bool *_usedcolumns;       /**< Gibt an welche Spalten für Clustering relevant sind */
FILE *_file_log;          /**< Log-File für Algorithmus */
/**< Boolean Wert der angibt ob die Hilfsinformationen der Datenbasis berechnet werden */
bool _compute_database_information;
int _points;                        /**< Zähler für Punkte des DBSCAN */
/**< Zähler für e-Nachbarschaftsermittlung zur Ermittlung der Quantile */
int _countneighbourhood;

/**
 * Methode distanceSquared ermittelt Quadratische Distanz zwischen den Werten der
 * Dimension von 2 Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector_2 Werte der Dimensionen des zweiten Punkt
 * @return Distanz zwischen zwei Punkte
 */
double distanceSquared(double *vector_start, double *vector_2);

/**
 * Methode distanceEuklid ermittelt Euklidische Distanz zwischen den Werten der
 * Dimension von 2 Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector_2 Werte der Dimensionen des zweiten Punkt
 * @return Distanz zwischen zwei Punkte
 */
double distanceEuklid(double *vector_start, double *vector_2);

/**
 * Methode distanceManhatten ermittelt Manhatten Distanz zwischen den Werten der
 * Dimension von 2 Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector_2 Werte der Dimensionen des zweiten Punkt
 * @return Distanz zwischen zwei Punkte
 */
double distanceManhatten(double *vector_start, double *vector_2);

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
```

```
        void computeStdDev();

        /**
         * Methode orderPointsToMedianQuantilClass ermittelt zugehörige Quantil-Klassen für
         * Median zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsToMedianQuantilClass(Point *point);

        /**
         * Methode orderPointsTo25QuantilQuantilClass ermittelt zugehörige Quantil-Klassen für
         * 25%-Quantil zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsTo25QuantilQuantilClass(Point *point);

        /**
         * Methode orderPointsTo75QuantilClass ermittelt zugehörige Quantil-Klassen für 75%-
         * Quantil zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsTo75QuantilQuantilClass(Point *point);

        /**
         * Methode findMedianWithQuantilClasses ermittelt ungefähren Median und ermittelt pro
         * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findMedianWithQuantilClasses(void);

        /**
         * Methode findQuantil25WithQuantilClasses ermittelt ungefähres 25%-Quantil und
         * ermittelt pro Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findQuantil25WithQuantilClasses(void);

        /**
         * Methode findQuantil75WithQuantilClasses ermittelt ungefähres 75%-Quantil und
         * ermittelt pro Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
         */
        void findQuantil75WithQuantilClasses(void);

public:
        /**
         * Konstruktor der Klasse GetData
         * @param dimensions Anzahl der Dimensionen pro Punkt
         * @param clusterid Position der Dimension der Cluster Id
         * @param distancefunction Gibt an welche Distanzfunktion verwendet werden soll
         *                         (1=Quadratisce,2=Euklidische,3=Manhatten)
         * @param table Tabelle aus der Daten gelesen werden
         * @param columns Spalten aus Tabellen die relevant sind
         * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
         * @param file_log Log-File für Algorithmus
         * @param clusterpoints Anzahl der zu clusternden Punkte
         */
        GetData(int dimensions, int clusterid, int distancefunction, char *table, char *columns,
                                        bool *usedcolumns, FILE *file_log, long clusterpoints);

        /**
         * Destruktor der Klasse GetData
         */
        ~GetData();
```

```
/**
 * Methode getPoint liefert nächsten Punkt der Datenbasis
 * @return Liefert nächsten Punkt der Datenbasis
 */
Point *getPoint(void);

/**
 * Methode getNeighbourhood liefert alle Punkte der Epsilon-Umgebung des übergebenen
 * Punkts mit dem übergebenen Epsilon-Wert
 * @param point Punkt zu dem Epsilon-Umgebung berechnet wird
 * @param epsilon Wert des Epsilon
 * @return Punkte in Epsilon-Umgebung des übergebenen Punkts
 */
list<Point> getNeighbourhood(Point *point, double epsilon);

/**
 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen ClusterId
 * @param clusterid Initiale Clusterid
 * @param nameclusterid Name der Spalte mit Cluster Id
 */
void initClusterIds(char *clusterid, char *nameclusterid);

/**
 * Methode getClusterId liefert Position der Dimension der Cluster ID
 * @return Position der Dimension der Cluster Id
 */
int getClusterId(){
        return _clusterid;
}

/**
 * Methode updateClusterId führt Update der Clusterid eins Punkts auf Datenbank durch.
 * Der Punkt wird über seine Rowid identifiziert und sein zugehöriges Cluster wird mit
 * übergeben
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster dem Punkt hinzugefügt wurde
 */
void updateClusterId(Point *point, int clusterid);

/**
 * Methode setgetData setzt Boolean-Wert getData der Klasse getData
 * @param getData Boolean-Wert der angibt ob letztes Fetch auf Datenbank erfolgreich
 *                        war
 */
void setgetData(bool getData){
        _getData = getData;
}

/**
 * Methode getData gibt an ob Ausführung der Methode getPoint (Ermittlung der Werte des
 * nächsten Punkts) erfolgreich war
 * @return Boolean-Wert der angibt ob Ermittlung der Werte der Dimensionen des Punkts
 *              erfolgreich war
 */
bool getData(){
        return _getData;
}

/**
 * Anzahl der Punkte der Datenbasis die für ´den Algorithmus verwendet wurden
 * @return Anzahl der verwendeten Punkte der Datenbasis
 */
```

```
        long getNumPoints(){
                return _numPoints;
        }

        /**
         * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste
         * Punkte in Datenbank
         * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank
         *                          geschrieben werden sollen
         * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der
         *                                  Cluster
         * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
         */
        void writeClusterNearestandFarestToDatabase(Cluster *clusters,char *table_nearest, char
                                                                                        *table_farest);

        /**
         * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in
         * Datenbank
         * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                                  Datenbasis
         */
        void writeDatabaseInformation(char *table_database);

        /**
         * Methode writeClusterInformation schreibt Hilfsinformationen zu den ermittelten Clustern
         * in Datenbank
         * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden
         * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         *                                  Clustern
         */
        void writeClusterInformation(Cluster *clusters, char *table_cluster);

        /**
         * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die Quantil-
         * Berechnung nach erster Iteration des Algorithmus (für 25% - Quantil, Median und 75% -
         * Quantil
         */
        void computeQuantilClasses(void);

        /**
         * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen (für alle 3
         * Quantile) zu den Dimensionen des Punkts
         * @param point Punkt der Quantil-Klassen zugeordnet wird
         */
        void orderPointsToQuantilClass(Point *point);

        /**
         * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile (25%, 50% und
         * 75%) und ermittelt pro Iteration neue Klassen --> genauere Quantile nach jeder Iteration
         */
        void findQuantilsWithQuantilClasses(void);
};
```

Source-File der Klasse GetData

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <list>
#include <vector>
#include <iostream>
```

```
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Quantil_Class.h"
#include "Information_Point.h"
#include "cluster.h"
#include "database.h"
#include "point.h"
#include "dbscan.h"
#include "getdata.h"
#include <string.h>
#include <time.h>
#include <malloc.h>
#include "sqlcpr.h"

/**
 * Variablen für SQL-Anweisungen (Daten lesen)
 */
EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR username[30];
        VARCHAR password[30];
        VARCHAR sid[10];
        VARCHAR _categoriccolumns[50];/**< Spalten (kategorische Werte) die nicht berücksichtigt */
        short val1, val2;
        VARCHAR statement[300];
        VARCHAR neighbourhood_statement[240];
        int init_clusterid;
        int up_clusterid;
        VARCHAR up_rowid[18];
        VARCHAR ins_rowid[18];
        VARCHAR rowid1[18];
        VARCHAR rowid2[18];
        VARCHAR update_statement[240];
        VARCHAR init_statement[240];
        VARCHAR create_statement_nearest[240];
        VARCHAR create_statement_farest[240];
        VARCHAR insert_statement_nearest[240];
        VARCHAR insert_statement_farest[240];
        VARCHAR create_statement_database[240];
        VARCHAR insert_statement_database[240];
        VARCHAR create_statement_cluster[300];
        VARCHAR insert_statement_cluster[300];
        int dimension;
        double sum;
        double sumsquared;
        double mean;
        double stddev;
        double minimum;
        double maximum;
        double median;
        double quantil25;
        double quantil75;
        double sqrtnumpoints;
        long number_points;
        int clusterid;
EXEC SQL END DECLARE SECTION;

SQLDA *select_des;
SQLDA *bind_des;
SQLDA *select_des_neighbour;
SQLDA *bind_des_neighbour;
```

```
extern SQLDA *sqlald();
extern void sqlnul();
EXEC SQL INCLUDE SQLDA;
EXEC SQL INCLUDE SQLCA;
int _clusterid;                 /**< Position der Dimension der Cluster Id */
double _epsilon;                /**< Epsilon-Wert für Epsilon-Umgebung */
int _minpoints;                 /**< Minimale Anzahl von Punkten in Epsilon-Umgebung */
long _clusterpoints;            /**< Anzahl der zu clusternden Punkte */
int _NearestNumPoints;   /**< Anzahl der nähesten Punkte pro Cluster, die berücksichtigt werden */
int _FarestNumPoints;    /**< Anzahl der entferntesten Punkte pro Cluster, die berücksichtigt werden */
int _distancefunction;/**< Distanzfunktion angeben: 1 = Quadratische, 2 = Euklidische, 3 = Manhatten */
char *ID_UNCLASSIFIED;    /**< Cluster ID für unklassifzierte Punkte */
char *_nameclusterid;       /**< Name der Spalte mit Cluster Id */
char *_table;               /**< Tabelle mit deren Daten geclustert wird */
char *_columns;             /**< Spalten die für Algorithmus relevant sind */
char *_table_nearest;       /**< Tabelle in die näheste Punkte der Cluster gespeichert werden */
char *_table_farest;        /**< Tabelle in die entfernteste Punkte der Cluster gespeichert werden */
char *_table_database;   /**< Tabelle in die Hilfsinformationen zu Datenbasis gespeichert werden */
char *_table_cluster;       /**< Tabelle in die Hilfsinformationen zu Clustern gespeichert werden */
char *_file_dbscan;         /**< File in das Ergebnis (Dauer) des Algorithmus geschrieben wird */
char *_file_log;            /**< Log-File bei Ausführung des Algorithmus */
char *_user;                /**< User für Datenbankanmeldung */
char *_password;            /**< Passwort für Datenbankanmeldung */
char *_service;             /**< Dienst für Datenbankanmeldung */

/**
 * Hauptprogramm
 */
int main(int argc, char *argv[]){
        long time_start, time_end;
        int i=0, j=0, numdimensions=0, prec, scal, nullok, k, countcategoric = 0;
        FILE *file_ergebnis, *file_log;
        GetData *datenbank;
        DBScan *dbscan;
        char *bind_var=0, *token, seps[] = ";", *stopstring;
        bool *usedcolumns;
        time_t ltime;

        try{
                // Programmparameter berücksichtigen
                if (!((argc == 21) || (argc == 22))) {
                        printf("Fehlende Programmparameter!!!\n");
                        return 1;
                }
                _user               = argv[1];
                _password                       = argv[2];
                _service            = argv[3];
                _table                          = argv[4];
                _columns                        = argv[5];
                _clusterpoints      = atol(argv[6]);
                _minpoints                      = atoi(argv[7]);
                _epsilon                        = strtod(argv[8], &stopstring);
                _clusterid          = atoi(argv[9]);
                _nameclusterid      = argv[10];
                _distancefunction   = atoi(argv[11]);
                _NearestNumPoints   = atoi(argv[12]);
                _FarestNumPoints    = atoi(argv[13]);
                _file_dbscan        = argv[14];
                _file_log           = argv[15];
                ID_UNCLASSIFIED     = argv[16];
                _table_nearest      = argv[17];
                _table_farest       = argv[18];
```

```
_table_database      = argv[19];
_table_cluster       = argv[20];
if (argc == 22){
        strcpy((char *)_categoriccolumns.arr, argv[21]);
        _categoriccolumns.len = strlen((char *)_categoriccolumns.arr);
}
file_log      = fopen(_file_log, "a");                  // Log-File öffnen

// Datenbankverbindung wird erstellt
strcpy((char *)username.arr,"stoett");
username.len = strlen((char *)username.arr);
strcpy((char *)password.arr,"klaus");
password.len = strlen((char *)password.arr);
strcpy((char *)sid.arr,"dke3");
sid.len = strlen((char *)sid.arr);

EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER NOTFOUND CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
EXEC SQL CONNECT :username IDENTIFIED BY :password USING :sid;

/*****************************************************************************
 * Erste dynamische SELECT-Anweisung für äußere Schleife des Algorithmus
 * Nur über unklassifzierte Punkte
 *****************************************************************************/
select_des    = sqlald(_clusterid + 1,_clusterid + 1,4);
bind_des      = sqlald(_clusterid + 1,_clusterid + 1,4);
select_des->N = _clusterid + 1;
bind_des->N   = 0;

strcpy((char *)statement.arr,"SELECT t.ROWID, ");
strcat((char *)statement.arr, _columns);
strcat((char *)statement.arr," FROM ");
strcat((char *)statement.arr, _table);
strcat((char *)statement.arr, " t WHERE ");
strcat((char *)statement.arr, _nameclusterid);
strcat((char *)statement.arr, " = ");
strcat((char *)statement.arr, ID_UNCLASSIFIED);
statement.len = strlen((char *)statement.arr);

EXEC SQL PREPARE sql_stmt FROM :statement;
EXEC SQL DECLARE cursor CURSOR FOR sql_stmt;
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
bind_des->N = bind_des->F;

for (i=0; i < bind_des->F; i++){
        printf("Eingabe des Wertes für Variable %*.*S:\n?", (int) bind_des->C[i], bind_des-
                                                                                >S[i]);
        gets(bind_var);
        bind_des->L[i] = strlen(bind_var);                          /* Länge des Werts */
        /* Reservieren von Speicher für Wert und Indikatorvariable */
        bind_des->V[i] = (char *)malloc(bind_des->L[i] + 1);
        bind_des->I[i] = (short *)malloc(sizeof(short));
        strcpy(bind_des->V[i], bind_var);       /* Speichern des Wertes */
        bind_des->I[i] = 0;                     /* Indikator-Wert setzen */
        bind_des->T[i] = 1;                     /* Datentyp = CHAR */
}
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
if (select_des->F < 0) select_des->F*= -1;
select_des->N = select_des->F;

for (i=0; i<select_des->F; i++) {
```

```
        /* lösche NULL-Bit für NULLable Datentypen */
        sqlnul((unsigned short*)&(select_des->T[i]), (unsigned short*)&(select_des->T[i]),
                                                                        &nullok);

        /* Korrigieren der Länge, falls notwendig */
        switch (select_des->T[i]) {
                case 1: break;                                          /* CHAR */
                case 2:                                                 /* NUMBER */
                        sqlpr2((unsigned int *)&select_des->L[i], &prec, &scal);
                        if (prec==0) prec = 40;
                        if ((scal > 0) || (scal == -127))
                                select_des->L[i] = sizeof(float);       /* FLOAT */
                        else
                                select_des->L[i] = sizeof(int);         /* INT */
                        if (i != _clusterid)
                                numdimensions++;
                        break;
                case 8:  select_des->L[i] = 240; break;                 /* LONG */
                case 11: select_des->L[i] = 18; break;                  /* ROWID */
                case 12: select_des->L[i] = 9; break;                   /* DATE */
                case 23: break;                                         /* RAW */
                case 24: select_des->L[i] = 240; break;                 /* LONG RAW */
                case 104: select_des->L[i] = 18; select_des->T[i] = 11; break;  /* ROWID */
        }
        /* Speicherallokation für Attributwerte sowie Indikatorvar. */
        select_des->V[i] = (char *)malloc(select_des->L[i]);
        select_des->I[i] = (short *)malloc(sizeof(short));

        /* alle Datentypen außer LONG RAW zu CHAR konvertieren */
        if ((select_des->T[i] != 24) && (select_des->T[i] != 2)) select_des->T[i] = 1;
        if (select_des->T[i] == 2)
                if ((scal > 0) || (scal == -127))
                        select_des->T[i] = 4;        //Float
                else
                        select_des->T[i] = 3;                //int
}
// Spalten die für Clustering verwendet werden
usedcolumns = new bool[numdimensions];
for(k = 0; k < numdimensions; k++)
        usedcolumns[k] = true;
token = strtok((char *)_categoriccolumns.arr, seps );
while( token != NULL){
        k = atoi(token);
        usedcolumns[k] = false;
        countcategoric++;
        token = strtok( NULL, ";");
}
/*****************************************************************************
 * Zweite dynamische Select-Anweisung zum Ermitteln der Epsilon-Umgebung zu einem
 * Punkt
 *****************************************************************************/
select_des_neighbour    = sqlald(_clusterid + 1,_clusterid + 1,4);
bind_des_neighbour      = sqlald(_clusterid + 1,_clusterid + 1,4);
select_des_neighbour->N = _clusterid + 1;
bind_des_neighbour->N   = 0;

strcpy((char *)neighbourhood_statement.arr,"SELECT t.ROWID, ");
strcat((char *)neighbourhood_statement.arr, _columns);
strcat((char *)neighbourhood_statement.arr," FROM ");
strcat((char *)neighbourhood_statement.arr, _table);
strcat((char *)neighbourhood_statement.arr, " t");
neighbourhood_statement.len = strlen((char *)neighbourhood_statement.arr);
```

```
EXEC SQL PREPARE sql_stmt_neighbour FROM :neighbourhood_statement;
EXEC SQL DECLARE neighbourhood_cursor CURSOR FOR sql_stmt_neighbour;

EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt_neighbour INTO
                                                    bind_des_neighbour;
bind_des_neighbour->N = bind_des_neighbour->F;

for (i=0; i < bind_des_neighbour->F; i++){
        printf("Eingabe des Wertes für Variable %*.*S:\n?", (int) bind_des_neighbour->C[i],
                                                    bind_des_neighbour->S[i]);
        gets(bind_var);
        bind_des_neighbour->L[i] = strlen(bind_var);        /* Länge des Werts */
        /* Reservieren von Speicher für Wert und Indikatorvariable */
        bind_des_neighbour->V[i] = (char *)malloc(bind_des_neighbour->L[i] + 1);
        bind_des_neighbour->I[i] = (short *)malloc(sizeof(short));

        strcpy(bind_des_neighbour->V[i], bind_var);        /* Speichern des Wertes */
        bind_des_neighbour->I[i] = 0;                      /* Indikator-Wert setzen */
        bind_des_neighbour->T[i] = 1;                      /* Datentyp = CHAR */
}
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt_neighbour INTO
                                                    select_des_neighbour;
if (select_des_neighbour->F < 0) select_des_neighbour->F*= -1;
select_des_neighbour->N = select_des_neighbour->F;

for (i=0; i<select_des_neighbour->F; i++) {
        /* lösche NULL-Bit für NULLable Datentypen */
        sqlnul((unsigned short*)&(select_des_neighbour->T[i]), (unsigned
                                    short*)&(select_des_neighbour->T[i]), &nullok);
        /* Korrigieren der Länge, falls notwendig */
        switch (select_des_neighbour->T[i]) {
                case 1: break;                                          /* CHAR */
                case 2:                                                 /* NUMBER */
                        sqlpr2((unsigned int *)&select_des_neighbour->L[i], &prec, &scal);
                        if (prec == 0) prec = 40;
                        if ((scal > 0) || (scal == -127))
                                select_des_neighbour->L[i] = sizeof(float);
                        else
                                select_des_neighbour->L[i] = sizeof(int);
                        break;
                case 8:  select_des_neighbour->L[i] = 240; break;       /* LONG */
                case 11: select_des_neighbour->L[i] = 18; break;        /* ROWID */
                case 12: select_des_neighbour->L[i] = 9; break;         /* DATE */
                case 23: break;                                         /* RAW */
                case 24: select_des_neighbour->L[i] = 240; break;       /* LONG RAW */
                case 104:                                               /* ROWID */
                        select_des_neighbour->L[i] = 18; select_des_neighbour->T[i] = 11;
                        break;
        }
        /* Speicherallokation für Attributwerte sowie Indikatorvar. */
        select_des_neighbour->V[i] = (char *)malloc(select_des_neighbour->L[i]);
        select_des_neighbour->I[i] = (short *)malloc(sizeof(short));

        /* alle Datentypen außer LONG RAW zu CHAR konvertieren */
        if ((select_des_neighbour->T[i] != 24) && (select_des_neighbour->T[i] != 2))
                select_des_neighbour->T[i] = 1;

        if (select_des_neighbour->T[i] == 2)
                if ((scal > 0) || (scal == -127))
                        select_des_neighbour->T[i] = 4;        //Float
                else
```

```
                        select_des_neighbour->T[i] = 3;             //int
}

// Update-Statement wird ermittelt und ausgeführt
strcpy((char *)update_statement.arr, "UPDATE ");
strcat((char *)update_statement.arr, _table);
strcat((char *)update_statement.arr, " SET ");
strcat((char *)update_statement.arr, _nameclusterid);
strcat((char *)update_statement.arr, " = :c WHERE ROWID = :r");
update_statement.len = strlen((char *)update_statement.arr);

EXEC SQL PREPARE up_sql_stmt FROM :update_statement;

/*****************************************************************************
 * Beginn der Main - Routine
 *****************************************************************************/
datenbank = new GetData(numdimensions, _clusterid, _distancefunction, (char *)_table,
                                    (char *)_columns, usedcolumns, file_log, _clusterpoints);
datenbank->initClusterIds((char *)ID_UNCLASSIFIED, _nameclusterid);

dbscan = new DBScan(datenbank, numdimensions, _epsilon, _minpoints,
                                    _NearestNumPoints, _FarestNumPoints, file_log);

time( &ltime );
fprintf(file_log,"%s %s", "Beginn DBScan:", ctime( &ltime ) );
fprintf(file_log, "%s %i %s %i %s %i \n", "Anzahl zu clusternde Punkte:", _clusterpoints,
        "Spalten:", numdimensions, "Davon kategorische Spalten:", countcategoric);
fflush(file_log);
time_start = clock();                   // Hilfsvariable für Zeitmessung
dbscan->doDBScan();                                             // DBScan ausführen

fprintf(file_log, "Ergebnisse in Tabellen schreiben. \n");
fflush(file_log);
dbscan->writeClusterNearestandFarestToDatabase((char *)_table_nearest,(char
                                                                    *)_table_farest);
dbscan->writeDatabaseInformation((char *)_table_database);
dbscan->writeClusterInformation((char *)_table_cluster);

time_end = clock();                     // Hilfsvariable für Zeitmessung
// Ergebnis des DBScan wird in File geschrieben
_file_dbscan = argv[14];
file_ergebnis = fopen(_file_dbscan, "a");
fprintf(file_ergebnis, "%s %i %s %i %s %i %s %f %s", "Punkte:", datenbank-
        >getNumPoints(), "Spalten:", numdimensions, "Davon kategorische Spalten:",
        countcategoric, "Dauer:", (time_end - time_start)/(float)CLOCKS_PER_SEC,ctime(
        &ltime ));
fclose(file_ergebnis);

// Cluster wieder löschen
delete dbscan;

/* Freigeben der Attribute und zugehöriger Indikatorvariablen */
for (i=0; i < select_des->F; i++) {
        free(select_des->V[i]);
        free(select_des->I[i]);
}
/* Freigeben der Deskriptoren */
sqlclu(select_des);

for (i=0; i < select_des_neighbour->F;i++){
        free(select_des_neighbour->V[i]);
        free(select_des_neighbour->I[i]);
```

```
                }
                sqlclu(select_des_neighbour);
                EXEC SQL COMMIT WORK RELEASE;

                fprintf(file_log,"%s \n \n","DBScan fertig!");
                fclose(file_log);

                return 0;
        } catch (exception& e){
                fprintf(file_log,"%s \n", e.what());
                fclose(file_log);
                return 1;
        }
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fclose(file_log);
                EXEC SQL ROLLBACK RELEASE;
                exit(1);
        }
        return 1;
}

/*****************************************************************
 * Klasse GetData
 *****************************************************************/
/**
 * Konstruktor der Klasse GetData
 * @param dimensions Anzahl der Dimensionen pro Punkt
 * @param clusterid Position der Dimension der Cluster Id
 * @param distancefunction Gibt an welche Distanzfunktion verwendet werden soll
 *                         (1=Quadratisce,2=Euklidische,3=Manhatten)
 * @param table Tabelle aus der Daten gelesen werden
 * @param columns Spalten aus Tabellen die relevant sind
 * @param usedcolumns Gibt an welche Spalten für Clustering relevant sind
 * @param file_log Log-File des Algorithmus
 * @param clusterpoints Anzahl der zu clusternden Punkte
 */
GetData::GetData(int dimensions, int clusterid, int distancefunction, char *table, char *columns, bool
*usedcolumns, FILE *file_log, long clusterpoints):_quantile(dimensions,vector<Quantil_Class>(100)),
        _quantile25(dimensions,vector<Quantil_Class>(100)),
        _quantile75(dimensions,vector<Quantil_Class>(100)){
        setgetData(true);
        _numPoints = 0;
        _dimensions       = dimensions;
        _distancefunction  = distancefunction;
        _zaehler_dimension = new int[_dimensions];
        _data      = new double[_dimensions];
        _mean      = new double[_dimensions];
        _stdDev    = new double[_dimensions];
        _sum       = new double[_dimensions];
        _sumSquared = new double[_dimensions];
        _minimum    = new double[_dimensions];
        _maximum    = new double[_dimensions];
        _median    = new double[_dimensions];
        _quantil25  = new double[_dimensions];
        _quantil75  = new double[_dimensions];
        _point     = new Point(_dimensions);
        _quantile.reserve(100);
        _quantile25.reserve(100);
        _quantile75.reserve(100);
        _table     = (char *)table;
```

```
        _columns       = (char *)columns;
        _usedcolumns   = new bool[_dimensions];
        _file_log   = file_log;
        _clusterpoints = clusterpoints;
        _compute_database_information = true;
        _points = 0;
        _countneighbourhood = 0;

        for (int i = 0; i < _dimensions; i++){
                _zaehler_dimension[i] = 0;
                _sum[i]     = 0;
                _sumSquared[i] = 0;
                _mean[i]    = 0;
                _stdDev[i]  = 0;
                _minimum[i] = 0;
                _maximum[i] = 0;
                _usedcolumns[i] = usedcolumns[i];
                _median[i]    = 0;
                _quantil25[i] = 0;
                _quantil75[i] = 0;
        }
        _clusterid  = clusterid;
}

/**
 * Destruktor der Klasse GetData
 */
GetData::~GetData(void){
        delete []_zaehler_dimension;
        delete []_data;
        delete []_mean;
        delete []_stdDev;
        delete []_sum;
        delete []_sumSquared;
        delete []_minimum;
        delete []_maximum;
        delete []_median;
        delete []_quantil25;
        delete []_quantil75;
        delete _point;

        _quantile.clear();
        _quantile25.clear();
        _quantile75.clear();
        fprintf(_file_log,"Destruktor Database erfolgreich ausgeführt! \n");
        fflush(_file_log);
}

/**
 * Methode updateClusterId führt Update der Clusterid eins Punkts auf Datenbank durch. Der Punkt
 * wird über seine Rowid identifiziert und sein zugehöriges Cluster wird mit übergeben
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster dem Punkt hinzugefügt wurde
 */
void GetData::updateClusterId(Point *point, int clusterid){
        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

        if (point->getClusterId() == atoi(ID_UNCLASSIFIED))
                _points++;
        up_clusterid = clusterid;
```

```
        strcpy((char *)up_rowid.arr,point->getRowId());
        up_rowid.len = 18;

        EXEC SQL EXECUTE up_sql_stmt USING :up_clusterid, :up_rowid;
        EXEC SQL COMMIT WORK;
sqlerror:
        if (sqlca.sqlcode != 0)
                if (!((sqlca.sqlcode == -1002) || (sqlca.sqlcode == 1403))){
                        fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                        fflush(_file_log);
                        exit(1);
                }
}

/**
 * Methode getNeighbourhood liefert alle Punkte der Epsilon-Umgebung des übergebenen Punkts
 * mit dem übergebenen Epsilon-Wert
 * @param point Punkt zu dem Epsilon-Umgebung berechnet wird
 * @param epsilon Wert des Epsilon
 * @return Punkte in Epsilon-Umgebung des übergebenen Punkts
 */
list<Point> GetData::getNeighbourhood(Point *point, double epsilon){
        list<Point> l_neighbourhood;
        Point *h_point;
        int i=0, j=0, clusterid=0, anzahl=0;
        double distance = 0;
        char *rowid;

        l_neighbourhood.clear();
        EXEC SQL WHENEVER SQLWARNING CONTINUE;
        EXEC SQL WHENEVER NOTFOUND  GOTO sqlerror;
        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
        EXEC SQL OPEN neighbourhood_cursor USING DESCRIPTOR bind_des_neighbour;

        anzahl = 0;
        _countneighbourhood++;
        if (_countneighbourhood == 2)
                this->computeQuantilClasses();

        while(_clusterpoints > anzahl){
                EXEC SQL FETCH neighbourhood_cursor USING DESCRIPTOR select_des_neighbour;
                anzahl++;

                if (_compute_database_information)
                        _numPoints++;

                strncpy((char *)rowid1.arr,point->getRowId(), 18);
                rowid1.len = 18;
                strncpy((char *)rowid2.arr,(char *)select_des_neighbour->V[0], 18);
                rowid2.len = 18;

                // Startobjekt nicht noch einmal hinzufügen
                if (strcmp((char *)rowid1.arr,(char *)rowid2.arr) != 0){
                        j = 0;
                        for (i=0; i < select_des_neighbour->F; i++){
                                if (select_des_neighbour->T[i] == 3){
                                        _data[j] = (int) *select_des_neighbour->V[i];          // INTEGER
                                        if (_compute_database_information){
                                                _sum[j]+= _data[j];
                                                _sumSquared[j]+= _data[j] * _data[j];
                                                _zaehler_dimension[j] = i;
```

```
                        // Minimum bestimmen
                        if (_data[j] < _minimum[j])
                                _minimum[j] = _data[j];
                        // Maximum bestimmen
                        if (_data[j] > _maximum[j])
                                _maximum[j] = _data[j];
                }
                j++;
        }
        if (select_des_neighbour->T[i] == 4){
                _data[j] = *(float *)select_des_neighbour->V[i];          // FLOAT
                if (_compute_database_information){
                        _sum[j]+= _data[j];
                        _sumSquared[j]+= _data[j] * _data[j];
                        _zaehler_dimension[j] = i;

                        // Minimum bestimmen
                        if (_data[j] < _minimum[j])
                                _minimum[j] = _data[j];
                        // Maximum bestimmen
                        if (_data[j] > _maximum[j])
                                _maximum[j] = _data[j];
                }
                j++;
        }
        if (i == 0)
                rowid = (char *)select_des_neighbour->V[i];          // ROWID
        if (i == _clusterid)
                clusterid = (int) *select_des_neighbour->V[i];       // CLUSTER ID
}
// Distanz zwischen Ausgangspunkt und ermittelten Punkt wird berechnet
switch( _distancefunction){
        case 1:                         // Quadratische Distanz
                distance = distanceSquared(point->getVector(), _data);
                break;
        case 2:                         // Euklidische Distanz
                distance = distanceEuklid(point->getVector(), _data);
                break;
        case 3:                         // Manhatten Distanz
                distance = distanceManhatten(point->getVector(), _data);
                break;
}
if ((_countneighbourhood > 1) && (_countneighbourhood <= 10)){
        h_point = new Point(_dimensions);
        h_point->set(_data,rowid,clusterid,true);
        // Klassen zur Quantil-Berechnung füllen
        this->orderPointsToQuantilClass(h_point);
        delete h_point;
}
// Prüfen ob Punkt in Epsilon-Umgebung ist
if (distance <= epsilon){
        h_point = new Point(_dimensions);
        h_point->set(_data,rowid,clusterid,true);
        l_neighbourhood.push_front(*h_point);
        delete h_point;
}
} else {
// Übergebener Punkt muss auch bei Berechnung der Hilfsinformationen
// berücksichtigt werden
if (_compute_database_information){
        j = 0;
        for (i=0; i < select_des_neighbour->F; i++){
```

```
                                        if (select_des_neighbour->T[i] == 3){
                                                _sum[j]+= _data[j];
                                                _sumSquared[j]+= _data[j] * _data[j];
                                                _zaehler_dimension[j] = i;
                                                _minimum[j] = _data[j];
                                                _maximum[j] = _data[j];
                                                j++;
                                        }
                                        if (select_des_neighbour->T[i] == 4){
                                                _sum[j]+= _data[j];
                                                _sumSquared[j]+= _data[j] * _data[j];
                                                _zaehler_dimension[j] = i;
                                                _minimum[j] = _data[j];
                                                _maximum[j] = _data[j];
                                                j++;
                                        }
                                }
                        }
                }
        }
        if ((_compute_database_information) && (anzahl == _clusterpoints)){
                computeStdDev();
                for (int i = 0; i < _dimensions; i++)
                        _mean[i] = _sum[i] / _numPoints;              // Mittelwerte berechnen
        }
        // Quantile (25%, 50% und 75%) werden näherungsweise ermittelt (nach jeder Iteration genauer)
        if ((_countneighbourhood > 1) && (_countneighbourhood <= 10))
                this->findQuantilsWithQuantilClasses();

        EXEC SQL CLOSE neighbourhood_cursor;
        _compute_database_information = false;
        return l_neighbourhood;
sqlerror:
        if ((sqlca.sqlcode == -1002) || (sqlca.sqlcode == 1403)) {
                EXEC SQL CLOSE neighbourhood_cursor;
                if ((_compute_database_information) && (anzahl == _clusterpoints)){
                        computeStdDev();
                        for (int i = 0; i < _dimensions; i++)
                          _mean[i] = _sum[i] / _numPoints;              // Mittelwerte berechnen
                }
                _compute_database_information = false;
                return l_neighbourhood;
        } else {
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                EXEC SQL ROLLBACK RELEASE;
                fflush(_file_log);
                exit(1);
                return 0;
        }
}

/**
 * Methode getPoint liefert Werte zu den Dimensionen eines Punkts. Beim Aufruf werden
 * die Werte des nächsten unklassifzierten Punkts geliefert
 * @return Liefert Vektor mit Werten zu den Dimensionen eines zu clusternden Punkts
 */
Point *GetData::getPoint(void){
        int i=0, j=0, clusterid=0;
        char *rowid;
        int anzahl;

        EXEC SQL WHENEVER SQLWARNING CONTINUE;
```

```
	EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
	EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
	EXEC SQL OPEN cursor USING DESCRIPTOR bind_des;
	EXEC SQL FETCH cursor USING DESCRIPTOR select_des;

	anzahl = _points;
	anzahl++;

	if (anzahl > _clusterpoints){
		setgetData(false);
		return 0;
	}
	for (i=0; i < select_des->F; i++){
		if (select_des->T[i] == 3)
			_data[j++] = (int) *select_des->V[i];			//INT
		if (select_des->T[i] == 4)
			_data[j++] = *(float *)select_des->V[i];		//FLOAT
		if (i == 0)
			rowid = (char *)select_des->V[i];			//ROWID
		if (i == _clusterid)
			clusterid = (int) *select_des->V[i];			//Cluster ID ermitteln
	}
	EXEC SQL CLOSE cursor;
	_point->set(_data, rowid, clusterid, false);
	return _point;
sqlerror:
	if ((sqlca.sqlcode == -1002) || (sqlca.sqlcode == 1403)) {
		fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
		setgetData(false);
		return 0;
	} else {
		fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
		EXEC SQL ROLLBACK RELEASE;
		fflush(_file_log);
		exit(1);
	}
}

/**
 * Methode initClusterId initialisiert alle Punkte mit der übergebenen initialen ClusterId
 * @param clusterid Initiale Clusterid
 * @param nameclusterid Name der Spalte mit Cluster Id
 */
void GetData::initClusterIds(char *clusterId, char *nameclusterid){
	// Update-Statement wird erzeugt und ausgeführt
	strcpy((char *)init_statement.arr, "UPDATE ");
	strcat((char *)init_statement.arr, _table);
	strcat((char *)init_statement.arr, " set ");
	strcat((char *)init_statement.arr, nameclusterid);
	strcat((char *)init_statement.arr, " = ");
	strcat((char *)init_statement.arr, clusterId);
	strcat((char *)init_statement.arr, " where ");
	strcat((char *)init_statement.arr, nameclusterid);
	strcat((char *)init_statement.arr, " <> ");
	strcat((char *)init_statement.arr, clusterId);
	init_statement.len = strlen((char *)init_statement.arr);

	EXEC SQL WHENEVER SQLWARNING CONTINUE;
	EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
	EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
	EXEC SQL prepare init_sql_stmt from :init_statement;
	EXEC SQL execute init_sql_stmt; // using :init_clusterid;
```

```
	EXEC SQL COMMIT;
sqlerror:
	if (sqlca.sqlcode < 0)
		fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
}

/**
 * Methode distanceSquared ermittelt Quadratische Distanz zwischen den Werten der Dimension von 2
 * Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector Werte der Dimensionen des zweiten Punkt
 * @return Quadratische Distanz zwischen zwei Punkte
 */
double GetData::distanceSquared(double *vector_start, double *vector_2){
	double h_distance, distance = 0;

	for(int i = 0; i < _dimensions; i++){
		if (_usedcolumns[i]){
			h_distance = *vector_start++ - *vector_2++;
			distance += h_distance * h_distance;
		}
	}
	return distance;
}

/**
 * Methode distanceEuklid ermittelt Distanz zwischen den Werten der Dimension von 2 Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector Werte der Dimensionen des zweiten Punkt
 * @return Distanz zwischen zwei Punkte
 */
double GetData::distanceEuklid(double *vector_start, double *vector_2){
	double h_distance, distance = 0;

	for(int i = 0; i < _dimensions; i++){
		if (_usedcolumns[i]){
			h_distance = *vector_start++ - *vector_2++;
			distance += h_distance * h_distance;
		}
	}
	return sqrt(distance);
}

/**
 * Methode distanceManhatten ermittelt Distanz zwischen den Werten der Dimension von 2 Punkten
 * @param vector_start Werte der Dimensionen von einem Punkt
 * @param vector Werte der Dimensionen des zweiten Punkt
 * @return Distanz zwischen zwei Punkte
 */
double GetData::distanceManhatten(double *vector_start, double *vector_2){
	double h_distance, distance = 0;

	for(int i = 0; i < _dimensions; i++){
		if (_usedcolumns[i]){
			h_distance = *vector_start++ - *vector_2++;
			distance += fabs(h_distance);
		}
	}
	return distance;
}

/**
```

```
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void GetData::computeStdDev(){
	double *sum = _sum, *sumSquared = _sumSquared, *stdDev = _stdDev, h_stddev;

	for(int i = 0; i < _dimensions; i++){
		if (_usedcolumns[i]){
			h_stddev = *sum++ / _numPoints;
			*stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints - h_stddev * h_stddev));
		}
	}
}

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste Punkte in
 * Datenbank
 * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank geschrieben werden
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
 */
void GetData::writeClusterNearestandFarestToDatabase(Cluster *clusters, char *table_nearest, char
                                                                                    *table_farest){
	Cluster *cluster;
	bool create_nearest = true;
	bool create_farest  = true;
	vector<Information_Point> vector_nearest;
	vector<Information_Point> vector_farest;
	vector<Information_Point>::iterator i;

	EXEC SQL WHENEVER SQLWARNING CONTINUE;
	EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
	EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

	// Create-Statment für näheste Punkte der Cluster
	strcpy((char *)create_statement_nearest.arr, "CREATE TABLE ");
	strcat((char *)create_statement_nearest.arr, table_nearest);
	strcat((char *)create_statement_nearest.arr, " AS SELECT ");
	strcat((char *)create_statement_nearest.arr, _columns );
	strcat((char *)create_statement_nearest.arr, " FROM ");
	strcat((char *)create_statement_nearest.arr, _table);
	strcat((char *)create_statement_nearest.arr, " t WHERE rowid = '");

	// Create-Statment für entfernteste Punkte der Cluster
	strcpy((char *)create_statement_farest.arr, "CREATE TABLE ");
	strcat((char *)create_statement_farest.arr, table_farest);
	strcat((char *)create_statement_farest.arr, " AS SELECT ");
	strcat((char *)create_statement_farest.arr, _columns );
	strcat((char *)create_statement_farest.arr, " FROM ");
	strcat((char *)create_statement_farest.arr, _table);
	strcat((char *)create_statement_farest.arr, " t WHERE rowid = '");

	for(cluster = clusters; cluster; cluster = cluster->getNext()){
		// Nähesten Punkte in Tabelle schreiben
		vector_nearest = cluster->getVectorNearest();
		for(i = vector_nearest.begin(); i != vector_nearest.end();i++){
			strncpy((char *)ins_rowid.arr,i->getRowId(), 18);
			ins_rowid.len = 18;

			if (create_nearest){
				strcat((char *)create_statement_nearest.arr, (char *)ins_rowid.arr);
				strcat((char *)create_statement_nearest.arr,"'");
				create_statement_nearest.len = strlen((char *)create_statement_nearest.arr);
```

```
                        EXEC SQL PREPARE create_sql_stmt_nearest FROM
                                                                :create_statement_nearest;
                        EXEC SQL EXECUTE create_sql_stmt_nearest;
                        create_nearest = false;
                    } else {
                        //INSERT mit Rowid
                        strcpy((char *)insert_statement_nearest.arr, "INSERT INTO ");
                        strcat((char *)insert_statement_nearest.arr, table_nearest);
                        strcat((char *)insert_statement_nearest.arr, " SELECT ");
                        strcat((char *)insert_statement_nearest.arr, _columns );
                        strcat((char *)insert_statement_nearest.arr, " FROM ");
                        strcat((char *)insert_statement_nearest.arr, _table);
                        strcat((char *)insert_statement_nearest.arr, " t WHERE rowid = '");
                        strcat((char *)insert_statement_nearest.arr, (char *)ins_rowid.arr);
                        strcat((char *)insert_statement_nearest.arr,"'");
                        insert_statement_nearest.len = strlen((char *)insert_statement_nearest.arr);
                        EXEC SQL PREPARE insert_sql_stmt_nearest FROM
                                                                :insert_statement_nearest;
                        EXEC SQL EXECUTE insert_sql_stmt_nearest;
                    }
                }
                // Entferntesten Punkte in Tabelle schreiben
                vector_farest = cluster->getVectorFarest();
                for(i = vector_farest.begin(); i != vector_farest.end();i++){
                    strncpy((char *)ins_rowid.arr,i->getRowId(), 18);
                    ins_rowid.len = 18;

                    if (create_farest){
                        strcat((char *)create_statement_farest.arr, (char *)ins_rowid.arr);
                        strcat((char *)create_statement_farest.arr,"'");
                        create_statement_farest.len = strlen((char *)create_statement_farest.arr);
                        EXEC SQL PREPARE create_sql_stmt_farest FROM
                                                                :create_statement_farest;
                        EXEC SQL EXECUTE create_sql_stmt_farest;
                        create_farest = false;
                    } else {
                        //INSERT mit Rowid
                        strcpy((char *)insert_statement_farest.arr, "INSERT INTO ");
                        strcat((char *)insert_statement_farest.arr, table_farest);
                        strcat((char *)insert_statement_farest.arr, " SELECT ");
                        strcat((char *)insert_statement_farest.arr, _columns );
                        strcat((char *)insert_statement_farest.arr, " FROM ");
                        strcat((char *)insert_statement_farest.arr, _table);
                        strcat((char *)insert_statement_farest.arr, " t WHERE rowid = '");
                        strcat((char *)insert_statement_farest.arr, (char *)ins_rowid.arr);
                        strcat((char *)insert_statement_farest.arr,"'");
                        insert_statement_farest.len = strlen((char *)insert_statement_farest.arr);
                        EXEC SQL PREPARE insert_sql_stmt_farest FROM
                                                                :insert_statement_farest;
                        EXEC SQL EXECUTE insert_sql_stmt_farest;
                    }
                }
        }
        EXEC SQL COMMIT;
sqlerror:
        if (sqlca.sqlcode < 0){
                fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
                fflush(_file_log);
        }
}

/**
```

```
 * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu Datenbasis
 */
void GetData::writeDatabaseInformation(char *table_database){
	int i = 0;

	EXEC SQL WHENEVER SQLWARNING CONTINUE;
	EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
	EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

	// Create-Statment für Informationen zu jeder Dimension der zugrunde liegenden Tabelle
	strcpy((char *)create_statement_database.arr, "CREATE TABLE ");
	strcat((char *)create_statement_database.arr, table_database);
	strcat((char *)create_statement_database.arr, " (dimension INTEGER, sum FLOAT(126),
	sumsquared FLOAT(126), mean FLOAT(126), stddev FLOAT(126), minimum FLOAT(126),
	maximum FLOAT(126), median FLOAT(126), quantil25 FLOAT(126), quantil75 FLOAT(126),
	number_points LONG, PRIMARY KEY(dimension))");
	create_statement_database.len = strlen((char *)create_statement_database.arr);

	EXEC SQL PREPARE create_sql_stmt_database FROM :create_statement_database;
	EXEC SQL EXECUTE create_sql_stmt_database;

	// Insert-Statment für Informationen jeder Dimension
	strcpy((char *)insert_statement_database.arr, "INSERT INTO ");
	strcat((char *)insert_statement_database.arr, table_database);
	strcat((char *)insert_statement_database.arr, " (dimension, sum, sumsquared, mean, stddev,
	minimum, maximum, median, quantil25, quantil75, number_points) VALUES (:v1, :v2, :v3, :v4,
	:v5, :v6, :v7, :v8, :v9, :v10, :v11)");
	insert_statement_database.len = strlen((char *)insert_statement_database.arr);

	EXEC SQL PREPARE insert_sql_stmt_database FROM :insert_statement_database;

	for (i=0;i < _dimensions;i++){
		dimension     = _zaehler_dimension[i];
		sum           = _sum[i];
		sumsquared    = _sumSquared[i];
		mean          = _mean[i];
		stddev        = _stdDev[i];
		minimum       = _minimum[i];
		maximum       = _maximum[i];
		median        = _median[i];
		quantil25     = _quantil25[i];
		quantil75     = _quantil75[i];
		number_points = _numPoints;
		EXEC SQL EXECUTE insert_sql_stmt_database USING :dimension, :sum, :sumsquared,
		:mean, :stddev, :minimum, :maximum, :median, :quantil25, :quantil75, :number_points;
	}
	EXEC SQL COMMIT;
sqlerror:
	if (sqlca.sqlcode < 0){
		fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
		fflush(_file_log);
	}
}

/**
 * Methode writeClusterInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden sollen
 * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu Clustern
 */
void GetData::writeClusterInformation(Cluster *clusters, char *table_cluster){
	Cluster *cluster;
```

```
	int i = 0;
	double *h_sum, *h_sumsquared, *h_mean, *h_stddev, *h_minimum, *h_maximum;

	EXEC SQL WHENEVER SQLWARNING CONTINUE;
	EXEC SQL WHENEVER NOTFOUND GOTO sqlerror;
	EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

	// Create-Statment für Informationen zu Clustern
	strcpy((char *)create_statement_cluster.arr, "CREATE TABLE ");
	strcat((char *)create_statement_cluster.arr, table_cluster);
	strcat((char *)create_statement_cluster.arr, " (clusterid INTEGER, dimension INTEGER, sum
	FLOAT(126), sumsquared FLOAT(126), mean FLOAT(126), stddev FLOAT(126), number_points
	LONG, sqrtnumpoints FLOAT(126), minimum FLOAT(126), maximum FLOAT(126), PRIMARY
	KEY(clusterid, dimension))");
	create_statement_cluster.len = strlen((char *)create_statement_cluster.arr);

	EXEC SQL PREPARE create_sql_stmt_cluster FROM :create_statement_cluster;
	EXEC SQL EXECUTE create_sql_stmt_cluster;

	// Insert-Statment für Informationen zu den Clustern
	strcpy((char *)insert_statement_cluster.arr, "INSERT INTO ");
	strcat((char *)insert_statement_cluster.arr, table_cluster);
	strcat((char *)insert_statement_cluster.arr, " (clusterid, dimension, sum, sumsquared, mean,
	stddev, number_points, sqrtnumpoints, minimum, maximum) VALUES (:v1, :v2, :v3, :v4, :v5, :v6,
	:v7, :v8, :v9, :v10)");
	insert_statement_cluster.len = strlen((char *)insert_statement_cluster.arr);

	EXEC SQL PREPARE insert_sql_stmt_cluster FROM :insert_statement_cluster;

	for(cluster = clusters; cluster; cluster = cluster->getNext()){
		h_sum        = cluster->getSum();
		h_sumsquared = cluster->getSumSquared();
		h_mean       = cluster->getMean();
		h_stddev     = cluster->getStdDev();
		h_minimum    = cluster->getMinimum();
		h_maximum    = cluster->getMaximum();

		for (i=0;i < _dimensions;i++){
			dimension     = _zaehler_dimension[i];
			clusterid     = cluster->getId();
			sum           = h_sum[i];
			sumsquared    = h_sumsquared[i];
			mean          = h_mean[i];
			stddev        = h_stddev[i];
			number_points = cluster->getNumPoints();
			sqrtnumpoints = cluster->getSqrtNumPoints();
			minimum       = h_minimum[i];
			maximum       = h_maximum[i];

			EXEC SQL EXECUTE insert_sql_stmt_cluster USING :clusterid, :dimension, :sum,
			:sumsquared, :mean, :stddev, :number_points, :sqrtnumpoints, :minimum,
			:maximum;
		}
	}
	EXEC SQL COMMIT;
sqlerror:
	if (sqlca.sqlcode < 0){
		fprintf(_file_log,"\n\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
		fflush(_file_log);
	}
}
```

```
/**
 * Methode computeQuantilClasses füllt pro Dimension 100 Klassen für die Quantil-Berechnung
 * nach erster Iteration des Algorithmus (für 25% - Quantil, Median und 75% - Quantil
 */
void GetData::computeQuantilClasses(void){
	double difference, minimum;
	Quantil_Class quantil_class;

	for(int i = 0; i < _dimensions; i++){
		difference =  ( _maximum[i] - _minimum[i] ) / 100;
		minimum = _minimum[i];

		for(int j = 0; j < 100; j++){
			quantil_class.setMinimum(minimum);
			minimum+=difference;
			if (j == 99)
				quantil_class.setMaximum(_maximum[i]);
			else
				quantil_class.setMaximum(minimum);
			_quantile[i][j] = quantil_class;
		}
	}
	_quantile25 = _quantile;
	_quantile75 = _quantile;
}

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen zu den Dimensionen des
 * Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsToQuantilClass(Point *point){
	this->orderPointsToMedianQuantilClass(point);
	this->orderPointsTo25QuantilQuantilClass(point);
	this->orderPointsTo75QuantilQuantilClass(point);
}

/**
 * Methode orderPointsToMedianQuantilClass ermittelt zugehörige Quantil-Klassen für Median zu den
 * Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsToMedianQuantilClass(Point *point){
	double *vector = point->getVector(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile[i][j] = quantil;
				found = true;
			}
			j++;
```

```
		}
	}
}

/**
 * Methode orderPointsTo25QuantilQuantilClass ermittelt zugehörige Quantil-Klassen für 25%-Quantil
 * zu den Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsTo25QuantilQuantilClass(Point *point){
	double *vector = point->getVector(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile25[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile25[i][j] = quantil;
				found = true;
			}
			j++;
		}
	}
}

/**
 * Methode orderPointsToQuantilClass ermittelt zugehörige Quantil-Klassen für 75%-Quantil zu den
 * Dimensionen des Punkts
 * @param point Punkt der Quantil-Klassen zugeordnet wird
 */
void GetData::orderPointsTo75QuantilQuantilClass(Point *point){
	double *vector = point->getVector(), wert;
	bool found;
	int j = 0;
	Quantil_Class quantil;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		wert = vector[i];

		while(!found){
			quantil = _quantile75[i][j];

			if (((quantil.getMinimum() <= wert) && (quantil.getMaximum() >= wert)) || (j == 99)){
				quantil.addNumPoints();
				_quantile75[i][j] = quantil;
				found = true;
			}
			j++;
		}
	}
}
```

```
/**
 * Methode findQuantilsWithQuantilClasses ermittelt ungefähre Quantile (25%, 50% und 75%) und
 * ermittelt pro Iteration neue Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantilsWithQuantilClasses(void){
	this->findMedianWithQuantilClasses();
	this->findQuantil25WithQuantilClasses();
	this->findQuantil75WithQuantilClasses();
}

/**
 * Methode findMedianWithQuantilClasses ermittelt ungefähren Median und ermittelt pro Iteration
 * neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findMedianWithQuantilClasses(void){
	long median = _numPoints / 2;
	bool found;
	int j = 0;
	Quantil_Class quantil, quantil_class;
	vector<vector<Quantil_Class> > quantile(_dimensions,vector<Quantil_Class>(100));
	double difference, minimum;
	long allnumpoints;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		allnumpoints = 0;

		// Klasse mit Median ermitteln
		while(!found){
			quantil = _quantile[i][j];
			allnumpoints+= quantil.getNumPoints();		// kumulierten Werte

			if (allnumpoints >= median)			// Klasse mit Median gefunden
				found = true;
			j++;
		}
		_median[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;	// Ungefähre Median

		// Neue genauere Klassen für Median-Berechnung ermitteln
		difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
		minimum = quantil.getMinimum();

		// Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
		quantil_class.setMinimum(_quantile[i][0].getMinimum());
		quantil_class.setMaximum(minimum);		// Minimum der ermittelten Klasse
		quantile[i][0] = quantil_class;

		for(j = 1; j < 99; j++){
			quantil_class.setMinimum(minimum);
			minimum+=difference;
			quantil_class.setMaximum(minimum);
			quantile[i][j] = quantil_class;
		}

		// Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
		quantil_class.setMinimum(quantil.getMaximum());
		quantil_class.setMaximum(_quantile[i][99].getMaximum());
		quantile[i][99] = quantil_class;
	}
	_quantile.clear();
	_quantile = quantile;
```

```
}

/**
 * Methode findQuantil25WithQuantilClasses ermittelt ungefähres 25%-Quantil und ermittelt pro
 * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantil25WithQuantilClasses(void){
	long quantil25 = _numPoints / 4;
	bool found;
	int j = 0;
	Quantil_Class quantil, quantil_class;
	vector<vector<Quantil_Class> > quantile25(_dimensions,vector<Quantil_Class>(100));
	double difference, minimum;
	long allnumpoints;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		allnumpoints = 0;

		// Klasse mit 25%-Quantil ermitteln
		while(!found){
			quantil = _quantile25[i][j];
			allnumpoints+= quantil.getNumPoints();		// kumulierten Werte

			if (allnumpoints >= quantil25)			// Klasse mit 25%-Quantil gefunden
				found = true;
			j++;
		}
		// Ungefähres 25% - Quantil
		_quantil25[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;

		// Neue genauere Klassen für Median-Berechnung ermitteln
		difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
		minimum = quantil.getMinimum();

		// Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
		quantil_class.setMinimum(_quantile25[i][0].getMinimum());
		quantil_class.setMaximum(minimum);		// Minimum der ermittelten Klasse
		quantile25[i][0] = quantil_class;

		for(j = 1; j < 99; j++){
			quantil_class.setMinimum(minimum);
			minimum+=difference;
			quantil_class.setMaximum(minimum);
			quantile25[i][j] = quantil_class;
		}
		// Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
		quantil_class.setMinimum(quantil.getMaximum());
		quantil_class.setMaximum(_quantile25[i][99].getMaximum());
		quantile25[i][99] = quantil_class;
	}
	_quantile25.clear();
	_quantile25 = quantile25;
}

/**
 * Methode findQuantil75WithQuantilClasses ermittelt ungefähres 75%-Quantil und ermittelt pro
 * Iteration neue Quantil-Klassen --> genauere Quantile nach jeder Iteration
 */
void GetData::findQuantil75WithQuantilClasses(void){
	long quantil75 = ( _numPoints * 3 ) / 4;
```

```
	bool found;
	int j = 0;
	Quantil_Class quantil, quantil_class;
	vector<vector<Quantil_Class> > quantile75(_dimensions,vector<Quantil_Class>(100));
	double difference, minimum;
	long allnumpoints;

	for(int i = 0; i < _dimensions; i++){
		found = false;
		j = 0;
		allnumpoints = 0;

		// Klasse mit 75%-Quantil ermitteln
		while(!found){
			quantil = _quantile75[i][j];
			allnumpoints+= quantil.getNumPoints();		// kumulierten Werte

			if (allnumpoints >= quantil75)			// Klasse mit 75%-Quantil gefunden
				found = true;
			j++;
		}
		// Ungefähre 75% - Quantil
		_quantil75[i] = ( quantil.getMaximum() + quantil.getMinimum() ) / 2;

		// Neue genauere Klassen für Median-Berechnung ermitteln
		difference =  ( quantil.getMaximum() - quantil.getMinimum() ) / 98;
		minimum = quantil.getMinimum();

		// Erste Klasse von Minimum (tatsächlich) bis zu Minimum der errechnten Klasse
		quantil_class.setMinimum(_quantile75[i][0].getMinimum());
		quantil_class.setMaximum(minimum);		// Minimum der ermittelten Klasse
		quantile75[i][0] = quantil_class;

		for(j = 1; j < 99; j++){
			quantil_class.setMinimum(minimum);
			minimum+=difference;
			quantil_class.setMaximum(minimum);
			quantile75[i][j] = quantil_class;
		}
		// Letzte Klasse von Maximum der errechneten Klasse bis zu tatsächlichem Maximum
		quantil_class.setMinimum(quantil.getMaximum());
		quantil_class.setMaximum(_quantile75[i][99].getMaximum());
		quantile75[i][99] = quantil_class;
	}
	_quantile75.clear();
	_quantile75 = quantile75;
}
```

12.2.3 Klasse Point

Header-File der Klasse Point

```
/**
 * Klasse Point ist Friend von Cluster
 */
class Cluster;
/**
 * Überschrift: Klasse Point repräsentiert einen Punkt
 * Beschreibung: Die Klasse Point repräsentiert einen Punkt
 * mit den zugehörigen Information, wie Anzahl der Dimensionen, Werte der Dimensionen
 */
```

```
class Point{
	private:
		friend class Cluster;		/**< Klasse Cluster ist ein Freund von Point */

		double *_vector;		/**< Vektor mit Werten zu den Dimensionen */
		double _clusterDistance; /**< Distanz des Punkts zu seinem zugeordneten Cluster */
		int _dimensions;		/**< Anzahl der Dimensionen des Punkts */
		int _clusterid;			/**< Cluster ID des Punkts */
		char _rowid[18];	/**< Rowid (eindeutiger Key) des Punkts aus Datenbank */
		bool _listkz;		/**< Boolean-Wert der angibt ob Destruktor aufgerufen werden soll */

	public:
		/**
		 * 1 Konstruktur der Klasse Points
		 */
		Point();

		/**
		 * 2 Konstruktor der Klasse Point
		 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
		 */
		Point(int dimension);

		/**
		 * 3 Konstruktor der Klasse Point
		 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
		 * @param data Vektor mit Werten der Dimensionen des Punkts
		 * @param rowid Rowid der Datenbank des Punkts
		 * @param clusterid Cluster ID des Punkts
		 */
		Point(int dimension, double *data, char *rowid, int clusterid) ;

		/**
		 * Destruktor der Klasse Cluster
		 */
		~Point();

		/**
		 * Methode getPoint liefert Zeiger auf sich selbst
		 * @return Zeiger auf aktuelles Objekt
		 */
		Point *getPoint(){return this;}

		/**
		 * Methode set setzt die Werte der Dimensionen, die RowId und die Position der Cluster-Id
		 * des Punkts
		 * @param data Vektor mit Werten der Dimensionen des Punkts
		 * @param rowid Rowid der Datenbank des Punkts
		 * @param clusterid Cluster ID des Punkts
		 * @param listkz Kennzeichen ob Destruktor aufgerufen werden soll oder nicht
		 */
		void set(double *data, char *rowid, int clusterid, bool listkz);

		/**
		 * Methode setClusterDistance setzt die Distanz des Punkts zu seinem Cluster
		 * @param distance Distanz des Punkts zum Cluster
		 */
		void setClusterDistance(double distance){
			_clusterDistance = distance;
		}

		/**
```

```
        * Methode getClusterDistance liefert Distanz des Punkts zu seinem zugeordneten Cluster
        * @return Distanz des Punkts zu seinem Cluster
        */
       double getClusterDistance(){
              return _clusterDistance;
       }

       /**
        * Methode getVector liefert Vektor mit Werten der Dimensionen des Punkts
        * @return Liefert Vektor mit Werten der Dimensionen
        */
       double *getVector(void){
              return _vector;
       }

       /**
        * Methode getMemUsed liefert benötigten Speicher des Objekts in Bytes
        * @return Benötigter Speicher in Bytes
        */
       long getMemUsed(void){
              return sizeof(*this) + _dimensions*sizeof(double);
       }

       /**
        * Methode getRowId liefert RowId der Datenbank des Punkts (=Key)
        * @return RowId der Datenbank
        */
       char *getRowId(){
              return _rowid;
       }

       /**
        * Methode getClusterId liefert Position der Dimension der Cluster Id
        * @return Position der Dimension der Cluster Id
        */
       int getClusterId(){
              return _clusterid;
       }

       /**
        * Methode getDimensions liefert Anzahl der Dimensionen des Punkts
        * @return Anzahl Dimensionen pro Punkt
        */
       int getDimensions(){
              return _dimensions;
       }

       /**
        * Methode setClusterid setzt RowId der Datenbank des Punkts (=Key)
        * @param clusterid Position der Dimension der Cluster Id
        */
       void setClusterId(int clusterid){
              _clusterid = clusterid;
       }

       /**
        * Methode print gibt Punkt (mit allen Werten seiner Dimensionen) in das
        * übergebene File aus
        * @param file File in das der Punkt geschrieben wird
        */
       void print(FILE *file);
};
```

Source-File der Klasse Point

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "Point.h"

/**
 * Konstruktor der Klasse Point
 */
Point::Point(){}

/**
 * 2 Konstruktor der Klasse Point
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 */
Point::Point(int dimension) : _dimensions(dimension){
	_vector = new double[dimension];
	_clusterid = 0;
}

/**
 * 3 Konstruktor der Klasse Point
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 * @param data Vektor mit Werten der Dimensionen des Punkts
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster ID des Punkts
 */
Point::Point(int dimension, double *data, char *rowid, int clusterid) : _dimensions(dimension){
	_vector = new double[dimension];
	for(int i = 0; i < _dimensions; i++)
		_vector[i] = data[i];
	strcpy(_rowid,rowid);
	_clusterid = clusterid;
}

/**
 * Destruktor der Klasse Cluster
 */
Point::~Point(){
	if (!_listkz){
		delete []_vector;
	}
}

/**
 * Methode set setzt die Werte der Dimensionen, die RowId und die Position der Cluster-Id des Punkts
 * @param data Vektor mit Werten der Dimensionen des Punkts
 * @param rowid Rowid der Datenbank des Punkts
 * @param clusterid Cluster ID des Punkts
 * @param listkz Kennzeichen ob Destruktor aufgerufen werden soll oder nicht
 */
void Point::set(double *data, char *rowid, int clusterid, bool listkz){
	delete []_vector;
	_vector = new double[_dimensions];
	for(int i = 0; i < _dimensions; i++)
```

```
            _vector[i] = data[i];
        strcpy(_rowid, rowid);
        _clusterid = clusterid;
        _listkz = listkz;
}

/**
 * Methode print gibt Punkt (mit allen Werten seiner Dimensionen) in das
 * übergebene File aus
 * @param file File in das der Punkt geschrieben wird
 */
void Point::print(FILE *file){
        for(int i = 0; i < _dimensions; i++)
            fprintf(file, "%f ", _vector[i]);
        fprintf(file, "\n");
}
```

12.2.4 Klasse Cluster

Header-File der Klasse Cluster

```
/**
 * Klasse Point ist Friend von Cluster
 */
class Point;
/**
 * Überschrift: Klasse Cluster repräsentiert einen Cluster
 * Beschreibung: Die Klasse Cluster repräsentiert einen Cluster mit den zugehörigen Information, wie
 * Mittelpunkt, Anzahl der Punkte Anzahl der Dimensionen usw.
 */
class Cluster{
        private:
            // Hilfsinformationen
            double *_sum;       /**< Vektor mit Summe der Dimensionen der Punkte des Clusters */
            /**< Vektor mit Quadratsummen der Dimensionen der Punkte des Clusters */
            double *_sumSquared;
            double *_mean;    /**< Mittelpunkt des Clusters (Vektor mit Dimensionen) */
            /**< Standardabweichung der Punkte des Clusters (Vektor mit Dimensionen) */
            double *_stdDev;
            double *_minimum;         /**< Minimum jeder Dimension */
            double *_maximum;         /**< Maximum jeder Dimension */
            double _sqrtNumPoints; /**< Wurzel über die Anzahl der Punkte */
            long _numPoints;          /**< Anzahl der Punkte des Clusters */
            /**< Vektor mit den _nearestNumPoints nähesten Punkten */
            vector<Information_Point> _vector_farest;
            /**< Vektor mit den _farestNumPoints entferntesten Punkten */
            vector<Information_Point> _vector_nearest;

            /**< Anzahl der nähesten Punkte des Clusters, die gespeicher werden sollen */
            int _nearestNumPoints;
            /**< Anzahl der am weitesten entfernten Punkte des Clusters, die gespeichert werden */
            int _farestNumPoints;
            int _id;                        /**< ID des Clusters */
            int _dimensions;                /**< Anzahl der Dimensionen pro Punkt des Clusters */
            FILE *_file_log;                /**< Log-File des Algorithmus */
            Cluster *_next, *_previous;        /**< Zeiger auf die Vorgänger und Nachfolger Cluster */

        public:
            /**
             * Konstruktor der Klasse Cluster
             * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
```

```
 * @param nearestnumpoints Anzahl der nähesten Punkte des Clusters, die gespeichert
 * werden sollen
 * @param farestnumpoints Anzahl der entferntesten Punkte des Clusters, die gespeichert
 * werden sollen
 * @param id ID des Clusters
 * @param file_log Log-File des Algorithmus
 */
Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, FILE *file_log);

/**
 * Destruktor der Klasse Cluster
 */
~Cluster();

/**
* Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die Werte
* des Clusters neu
* @param point Punkt der zu Cluster hinzugefügt wird
*/
void addPoint(Point *point);

/**
 * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des Clusters
 * gehört. Die _nearestNumPoints werden in einer eigenen Liste gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten gehört
 */
void addNearestPoint(Point *point);

/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des Clusters
 * gehört. Die _farestNumPoints werden in einer eigenen Liste gespeichert
 * @see Cluster()
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten Punkten gehört
 */
void addFarestPoint(Point *point);

/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void updateClusterMean(void);

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void computeStdDev(void);

/**
 * Methode print schreibt wichtige Werte des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void print(FILE *file = stdin);

/**
 * Methode printConfidence schreib Konfidenz des Clusters in das übergebene
 * File
 * @param file File in das Geschrieben wird
 */
void printConfidence(FILE *file = stdin);

/**
 * Methode setPrevious setzt Vorgänger des Clusters
```

```
 * @param cluster Vorgängercluster
 */
void setPrevious(Cluster *cluster){
	_previous = cluster;
}

/**
 * Methode setNext setzt Nachfolger des Clusters
 * @param cluster Nachfolgercluster
 */
void setNext(Cluster *cluster){
	_next = cluster;
}

/**
 * Methode getPrevious liefert Vorgänger-Cluster des Clusters
 * @return Vorgänger-Cluster wird zurück gegeben
 */
Cluster *getPrevious(void){
	return _previous;
}

/**
 * Methode getNext liefert Nachfolger-Cluster des Clusters
 * @return Nachfolger-Cluster wird zurück gegeben
 */
Cluster *getNext(void){
	return _next;
}

/**
 * Methode getId liefert ID des Clusters
 * @return ID des Clusters
 */
int getId(void){
	return _id;
}

/**
 * Methode getMemUsed liefert benötigten Speicher des Objekts in Bytes
 * @return Benötigter Speicher in Bytes
 */
long getMemUsed(void){
	return sizeof(*this) + 4*_dimensions*sizeof(double);
}

/**
* Methode getAllocated liefert die Anzahl der allokierten Cluster
* @return Anzahl der allokierten Cluster
*/
static long getAllocated(void);

/**
 * Methode getNumPoints liefert Anzahl der zugeordneten Punkte des Clusters
 * @return Anzahl der zugeordneten Punkte
 */
long getNumPoints(void){
	return _numPoints;
}

/**
 * Methode getSqrtNumPoints liefert Wurzel der Anzahl der zugeordneten Punkte des
```

```
 * Clusters
 * @return Wurzel der Anzahl der zugeordneten Punkte
 */
double getSqrtNumPoints(void){
	return _sqrtNumPoints;
}

/**
 * Methode getMean liefert Mittelpunkt der Punkte des Clusters
 * @return Mittelpunkt des Clusters
 */
double *getMean(void){
	return _mean;
}

/*
 * Methode getSum liefert Summe der Dimensionen der Punkte des Clusters
 * @return Summe der Dimensionen der Punkte
 */
double *getSum(void){
	return _sum;
}

/*
 * Methode getSumSquared liefert Quadratsumme der Dimensionen der Punkte des
 * Clusters
 * @return Quadratsumme der Dimensionen der Punkte
 */
double *getSumSquared(void){
	return _sumSquared;
}

/*
 * Methode getStdDev liefert Standardabweichung der Dimensionen der Punkte des
 * Clusters
 * @return Standardabweichung der Dimensionen der Punkte
 */
double *getStdDev(void){
	return _stdDev;
}

/**
 * Methode getMinimum liefert Minimum der Punkte des Clusters
 * @return Minimum des Clusters
 */
double *getMinimum(void){
	return _minimum;
}

/*
 * Methode getMaximum liefert Maximum der Dimensionen der Punkte des Clusters
 * @return Maximum der Dimensionen der Punkte
 */
double *getMaximum(void){
	return _maximum;
}

/*
 * Methode getVectorFarest liefert Vektor mit entferntesten Punkten zu Cluster
 * @return Vektor mit entferntesten Punkten des Clusters
 */
vector<Information_Point> getVectorFarest(){
```

```
                return _vector_farest;
        }

        /*
         * Methode getVectorNearest liefert Vektor mit nähesten Punkten des Clusters
         * @return Vektor mit nähesten Punkten des Clusters
         */
        vector<Information_Point> getVectorNearest(){
                return _vector_nearest;
        }
};
```

Source-File der Klasse Cluster

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "Information_Point.h"
#include "Point.h"
#include "Cluster.h"

static int Allocated = 0;              // Anzahl der allokierten Cluster

/**
 * Konstruktor der Klasse Cluster
 * @param dimension Gibt die Anzahl der Dimensionen pro Punkt an
 * @param nearestnumpoints Anzahl der nähesten Punkte des Clusters, die gespeichert werden
 * @param farestnumpoints Anzahl der entferntesten Punkte des Clusters, die gespeicher werden
 * @param id ID des Clusters
 * @param file_log Log-File des Algorithmus
 */
Cluster::Cluster(int dimension, int nearestnumpoints, int farestnumpoints, int id, FILE *file_log) :
_dimensions(dimension), _next(0), _nearestNumPoints(nearestnumpoints),
_farestNumPoints(farestnumpoints){
        // Erzeuge Double-Vektoren für die zu berechnenden Werte des Clusters
        _sum       = new double[dimension];
        _sumSquared = new double[dimension];
        _mean      = new double[dimension];
        _stdDev    = new double[dimension];
        _minimum   = new double[dimension];
        _maximum   = new double[dimension];
        _id        = id;
        _file_log  = file_log;

        for(int i = 0; i < dimension; i++){
                _sum[i]       = 0;
                _sumSquared[i] = 0;
                _mean[i]      = 0;
                _stdDev[i]    = 0;
                _minimum[i]   = 0;
                _maximum[i]   = 0;
        }

        // Initialisiere Variablen des Clusters
        _numPoints    = 0;
```

```
	_sqrtNumPoints = 0;
	_vector_nearest.reserve(nearestnumpoints);
	_vector_farest.reserve(farestnumpoints);

	Allocated++;		// Anzahl der allokierten Cluster
}

/**
 * Destruktor der Klasse Cluster
 */
Cluster::~Cluster(){
	delete []_sum;
	delete []_sumSquared;
	delete []_mean;
	delete []_stdDev;
	delete []_minimum;
	delete []_maximum;

	Allocated--;
	if(Allocated < 0)
		fprintf(_file_log,"Too many calls to ~Cluster().\n");
	else if(Allocated == 0)
		fprintf(_file_log,"Destruktor ~Cluster() OK.\n");
}

/**
 * Methode getAllocated liefert die Anzahl der allokierten Cluster
 * @return Anzahl der allokierten Cluster
 */
long Cluster::getAllocated(void){
	return Allocated;
}

/**
 * Methode addPoint fügt einen Punkt zu einem Cluster hinzu und berechnet die Werte
 * des Clusters neu
 * @param point Punkt der zu Cluster hinzugefügt wird
 */
void Cluster::addPoint(Point *point){
	// Werte des Clusters werden an lokale Zeiger übergeben
	double *vector = point->_vector, *sum = _sum, *sumSquared = _sumSquared,
		*minimum = _minimum, *maximum = _maximum;

	// Summen und Quadratsummen für jede Dimension werden berechnet
	for(int i = 0; i < _dimensions; i++){
		// Minimum
		if (*minimum == 0)
			*minimum = *vector;
		else if (*minimum > *vector)
			*minimum = *vector;
		// Maximum
		if (*maximum == 0)
			*maximum = *vector;
		else if (*maximum < *vector)
			*maximum = *vector;
		*minimum++;
		*maximum++;
		*sumSquared++ += *vector**vector;
		*sum++ += *vector++;
	}
	_numPoints++;		// Anzahl der Punkte
}
```

```
/**
 * Methode updateClusterMean berechnet den Mittelpunkt des Clusters neu
 */
void Cluster::updateClusterMean(void){
	double *sum = _sum, *mean = _mean;

	// Mittelpunkt des Clusters wird neu berechnet
	if(_numPoints)
		for(int i = 0; i < _dimensions; i++)
			*mean++ = *sum++ / _numPoints;
}

/**
 * Methode computeStdDev ermittelt die Standardabweichung der Punkte des Clusters
 */
void Cluster::computeStdDev(){
	double *sum = _sum, *sumSquared = _sumSquared, *l_stdDev = _stdDev;
	double h_stddev;

	for(int i = 0; i < _dimensions; i++){
		h_stddev = *sum++ / _numPoints;
		*l_stdDev++ = (double) (sqrt(*sumSquared++ / _numPoints - h_stddev * h_stddev));
	}
	_sqrtNumPoints = (double) sqrt(_numPoints);
}

/**
 * Methode print schreibt wichtige Werte des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void Cluster::print(FILE *file){
	// Ausgabe des Mittelpunkts des Clusters
	fprintf(file,"%s", "Mittelpunkt des Clusters:");
	for(int i = 0; i < _dimensions; i++)
		fprintf(file, "%f ", _mean[i]);
	fprintf(file, "\n");
	fprintf(file, "%s %d\n","Anzahl der Punkte im Cluster:", _numPoints);
}

/**
 * Methode printConfidence schreib Konfidenz des Clusters in das übergebene File
 * @param file File in das Geschrieben wird
 */
void Cluster::printConfidence(FILE *file){
	for(int i = 0; i < _dimensions; i++)
		fprintf(file, "%f ", _stdDev[i]/_sqrtNumPoints);
	fprintf(file, "\n");
}

/**
 * Methode addFarestPoint überprüft ob Punkt zu den entferntesten Punkten des Clusters gehört
 * Die _farestNumPoints werden in einer eigenen Liste gespeichert
 * @param point Punkt bei dem überprüft wird ob er zu den entferntesten Punkten gehört
 */
void Cluster::addFarestPoint(Point *point){
	long zufall;
	Information_Point info_point(point->getRowId(),point->getClusterDistance());

	srand((unsigned)time( NULL ));
	zufall = rand() % _numPoints;
```

```
        if (zufall < _farestNumPoints)
                if (_vector_farest.size() < _farestNumPoints)
                        _vector_farest.push_back(info_point);
                else {
                        _vector_farest.erase(_vector_farest.begin() + zufall);
                        _vector_farest.push_back(info_point);
                }
}

/**
 * Methode addNearestPoint überprüft ob Punkt zu den nähesten Punkten des Clusters gehört
 * Die _nearestNumPoints werden in einer eigenen Liste gespeichert
 * @param point Punkt bei dem überprüft wird ob er zu den nähesten Punkten gehört
 */
void Cluster::addNearestPoint(Point *point){
        long zufall;
        Information_Point info_point(point->getRowId(), point->getClusterDistance());

        srand((unsigned)time( NULL ));
        zufall = rand() % _numPoints;

        if (zufall < _nearestNumPoints)
                if (_vector_nearest.size() < _nearestNumPoints)
                        _vector_nearest.push_back(info_point);
                else {
                        _vector_nearest.erase(_vector_nearest.begin() + zufall);
                        _vector_nearest.push_back(info_point);
                }
}
```

12.2.5 Klasse Quantil_Class

Header-File der Klasse Quantil_Class

```
/**
 * Überschrift: Klasse Quantil_Class repräsentiert eine Klasse zur Berechnung der Quantile
 * Beschreibung: Die Klasse Quantil_Class repräsentiert eine Klasse zur Berechnung der Quantile
 * (25%, 50% und 75%) verwendete wird
 */
class Quantil_Class{
        private:
                double _minimum;        /**< Untere Grenze der Klasse zur Ermittlung der Quantile */
                double _maximum;        /**< Obere Grenze der Klasse zur Ermittlung der Quantile */
                long _numPoints;        /**< Anzahl der Punkte in Klasse */

        public:
                Quantil_Class();

                /**
                 * Konstruktor der Klasse Quantil_Point
                 * @param minimum Untere Grenze der Klasse zur Ermittlung der Quantile
                 * @param maximum Obere Grenze der Klasse zur Ermittlung der Qunatile
                 */
                Quantil_Class(double minimum, double maximum);

                /**
                 * Destruktor der Klasse Information_Point
                 */
                ~Quantil_Class();
```

```
        /**
         * Methode addNumPoints erhöht die Anzahl der Punkte der Klasse um 1
         */
        void addNumPoints(void){
                _numPoints++;
        }

        /**
         * Methode setMinimum setzt untere Grenze der Klasse
         * @param minimum Untere Grenze der Klasse
         */
        void setMinimum(double minimum){
                _minimum = minimum;
        }

        /**
         * Methode setMaximum setzt obere Grenze der Klasse
         * @param maximum Obere Grenze der Klasse
         */
        void setMaximum(double maximum){
                _maximum = maximum;
        }

        /**
         * Methode getMinimum liefert die untere Grenze der Klasse
         * @return Untere Grenze der Klasse
         */
        double getMinimum(void){
                return _minimum;
        }

        /**
         * Methode getMaximum liefert obere Grenze der Klasse
         * @return Obere Grenze der Klasse
         */
        double getMaximum(void){
                return _maximum;
        }

        /**
         * Methode getNumPoints liefert die Anzahl der Punkte pro Klasse
         * @return Anzahl der Punkte in Klasse
         */
        long getNumPoints(void){
                return _numPoints;
        }
};
```

Source-File der Klasse Quantil_Class

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "quantil_class.h"

Quantil_Class::Quantil_Class(){
        _numPoints = 0;
}
```

```
/**
 * Konstruktor der Klasse Quantil_Class
 * @param minimum Untere Grenze der Klasse
 * @param maximum Obere Grenze der Klasse
 */
Quantil_Class::Quantil_Class(double minimum, double maximum){
	_minimum = minimum;
	_maximum = maximum;
	_numPoints = 0;
}

/**
 * Destruktor der Klasse Quantil_Class
 */
Quantil_Class::~Quantil_Class(){}
```

12.2.6 Klasse Information_Point

Header-File der Klasse Information_Point

```
/**
 * Überschrift: Klasse Information_Point repräsentiert einen Punkt zur Berechnung der
 * Hilfsinformationen
 * Beschreibung: Die Klasse Information_Point repräsentiert einen Punkt der zur Berechnung der
 * Hilfsinformationen (näheste Punkte, weit entfernteste Punkte) verwendet wird
 */
class Information_Point{
	private:
		char _rowid[18];			/**< RowID eine Punkts */
		double _distance;			/**< Distanz des Punkts zu seinem Cluster */

	public:
		/**
		 * Konstruktor der Klasse Information_Point
		 * @param rowid RowId eines Punkts
		 * @param distance Distanz eines Punkts zu seinem Cluster
		 */
		Information_Point(char *rowid, double distance);

		/**
		 * Destruktor der Klasse Information_Point
		 */
		~Information_Point();

		/**
		 * Methode getRowid liefert die RowID des Punkts
		 * @return RowId des Punkts
		 */
		char *getRowId(void);

		/**
		 * Methode getDistance liefert Distanz des Punkts zu seinem Cluster
		 * @return Distanz zum Cluster
		 */
		double getDistance(void);
};
```

Source-File der Klasse Information_Point

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "information_point.h"

/**
 * Konstruktor der Klasse Information_Point
 * @param rowid Rowid des Punkts aus Datenbank
 * @param distance Distanz des Punkts zu seinem Centroiden
 */
Information_Point::Information_Point(char *rowid, double distance){
        strcpy(_rowid,rowid);
        _distance = distance;
}

/**
 * Destruktor der Klasse Information_Point
 */
Information_Point::~Information_Point(){}

/**
 * Methode getRowid liefert die RowID des Punkts
 * @return RowId des Punkts
 */
char *Information_Point::getRowId(){
        return _rowid;
}

/**
 * Methode getDistance liefert Distanz des Punkts zu seinem Centroiden
 * @return Distanz zum Cluster
 */
double Information_Point::getDistance(){
        return _distance;
}
```

12.2.7 Klasse DBScan

Header-File der Klasse DBScan

```
/**
 * Die Klassen Cluster und Database sind Friends von DBScan
 */
class Cluster;
class Database;
/**
 * Überschrift: Klasse DBScan die DBScan-Algorithmus ausführt
 * Beschreibung: Klasse DBScan führt DBScan-Algorithmus aus. Das Ergebnis
 * des Algorithmus wird in den Clustern gespeichert
 */
class DBScan{
        private:
                Database *_database;    /**< Datenbasis des Algorithmus */
                int _numClusters;       /**< Anzahl der ermittelten Cluster */
                int _dimensions;        /**< Anzahl der Dimensionen pro Punkt */
                Cluster *_clusters;  /**< Enthält numClusters Cluster in denen Ergebnis gespeichert wird */
                int  _minPoints;        /**< Minimale Anzahl von Punkten in Epsilon-Umgebung */
                double _epsilon;        /**< Epsilon-Wert für Epsilon-Umgebung */
```

```
        /**< Anzahl der nähesten Punkte pro Cluster, die gespeicher werden sollen */
        int _nearestNumPoints;
        /**< Anzahl der am weitesten entfernten Punkte pro Cluster, die gespeichert werden */
        int _farestNumPoints;
        FILE *_file_log;                /**< Log-File für Algorithmus */

protected:
        /**
         * Methode getNextClusterId liefert nächste Cluster Id
         * @param oldclusterid Letzte Cluster Id
         * @return Liefert nächste Cluster Id
         */
        int getNextClusterId(int oldclusterid);

        /**
         * Methode expandCluster liefert zu übergebenen Punkt ein Cluster über die Epsilon-
         * Umgebung des Punkts
         * @param point Punkt zu dem Cluster über Epsilon-Umgebung ermittelt wird
         * @param clusterid Cluster Id des zu erzeugenden Clusters
         * @param epsilon Epsilon-Wert für Epsilon-Umgebung
         * @param minpoints Minimale Anzahl von Punkten in Epsilon-Umgebung
         * @return Boolean-Wert der angibt ob Cluster zu Punkt ermittelt wurde (wenn nein -->
         *              Noise)
         */
        bool expandCluster(Point *point, int clusterid, double epsilon, int minpoints);

        /**
         * Methode getCluster liefert Cluster zu einer Id
         * @return Cluster zu übergebener Id
         */
        Cluster *getCluster(int id);

public:
        /**
         * Konstruktor der Klasse DBScan
         * @param datenbank Datenbasis des Algorithmus
         * @param dimension Anzahl der Dimensionen pro Punkt
         * @param epsilon Epsilon-Wert für Epsilon-Umgebung
         * @param minpoint Minimale Anzahl von Punkten in Epsilon-Umgebung
         * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer
         *                                      eigenen Liste gespeichert werden
         * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer
         *                                    eigenen Liste gespeichert werden
         * @param file_log Log-File für Algorithmus
         */
        DBScan(Database *datenbank, int dimension, double epsilon, int minpoints, int
                                        nearestnumpoints, int farestnumpoints, FILE *file_log);

        /**
         * Destruktor der Klasse DBScan
         */
        virtual ~DBScan();

        /**
         * Methode doDBScan führt DBScan-Algorithmus aus
         * @see expandCluster()
         */
        void doDBScan();

        /**
         * Methode addCluster fügt erzeugtes Cluster zu Liste der Cluster hinzu
         * @param cluster Cluster der hinzgefügt werden soll
```

```
        */
        void addCluster(Cluster *cluster);

        /**
         * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste
         * Punkte in Datenbank
         * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank
         *                 geschrieben werden sollen
         * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der
         *                      Cluster
         * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
         */
        void writeClusterNearestandFarestToDatabase(char *table_nearest, char *table_farest);

        /**
         * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in
         * Datenbank
         * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         * Datenbasis
         */
        void writeDatabaseInformation(char *table_database);

        /**
         * Methode writeClusterInformation schreibt Hilfsinformationen zu ermittelten Clustern in
         * Datenbank
         * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden
         * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu
         * Clustern
         */
        void writeClusterInformation(char *table_cluster);

        /**
         * Ergebnis des Clusters (Mittelpunkt, Standardabweichung, Anzahl der Punkte)
         * wird in File geschrieben
         * @param filename File in das Ergebnisdaten des Clusters geschrieben werden
         */
        void writeClusterToFile(char *filename);
};
```

Source-File der Klasse DBScan

```
/**
 * Benötigte Include-Files (Klassen)
 */
#include <list>
#include <vector>
#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Information_Point.h"
#include "Point.h"
#include "Cluster.h"
#include "dbscan.h"
#include "database.h"
#include <time.h>

const ID_NOISE = 0;                 /**< Cluster ID für Noise-Objekte (Rauschen) */
const ID_UNCLASSIFIED = -1;         /**< Initiale Cluster Id für unklassifizierte Objekte */

/**
```

```
 * Konstruktor der Klasse DBScan
 * @param datenbank Datenbasis des Algorithmus
 * @param dimension Anzahl der Dimensionen pro Punkt
 * @param epsilon Epsilon-Wert für Epsilon-Umgebung
 * @param minpoint Minimale Anzahl von Punkten in Epsilon-Umgebung
 * @param nearestnumpoints Anzahl der nähesten Punkte pro Cluster, die in einer eigenen Liste
 *                                  gespeichert werden
 * @param farestnumpoints Anzahl der entferntesten Punkte pro Cluster, die in einer eigenen Liste
 *                                 gespeichert werden
 * @param file_log Log-File für Algorithmus
 */
DBScan::DBScan(Database *datenbank, int dimension, double epsilon, int minpoints, int
nearestnumpoints, int farestnumpoints, FILE *file_log) : _database(datenbank),
          _dimensions(dimension), _epsilon(epsilon), _minPoints(minpoints){
        _clusters = 0;
        _nearestNumPoints = nearestnumpoints;
        _farestNumPoints  = farestnumpoints;
        _file_log = file_log;
}

/**
 * Destruktor der Klasse DBScan
 */
DBScan::~DBScan(){
        // Alle erzeugten Cluster des Algorithmus werden gelöscht
        Cluster *cluster, *nextcluster;
        for(cluster = _clusters; cluster; cluster = nextcluster){
                nextcluster = cluster->getNext();
                delete cluster;
        }
        delete _database;
}

/**
 * Methode getNextClusterId liefert nächste Cluster Id
 * @param oldclusterid Letzte Cluster Id
 * @return Liefert nächste Cluster Id
 */
int DBScan::getNextClusterId(int oldclusterid){
        if (++oldclusterid > 99)
                exit(1);
        return oldclusterid;
}

/**
 * Methode doDBScan führt DBScan-Algorithmus aus
 * @see expandCluster()
 */
void DBScan::doDBScan(){
        Point *point;

        // Erster unklassifizierte Punkt wird aus Datenbank gelesen
        point = _database->getPoint();

        // Ermittlung der ersten Cluster ID
        int clusterId = getNextClusterId(ID_NOISE);

        // Solange ein unklassifzierter Punkt gefunden wird wird versucht zu diesem ein Cluster zu finden
        while (_database->getData()){
                if (point->getClusterId() == ID_UNCLASSIFIED)
                        if (expandCluster(point, clusterId, _epsilon, _minPoints))
                                clusterId = getNextClusterId(clusterId);
```

```
                point = _database->getPoint();
        }
}

/**
 * Methode expandCluster liefert zu übergebenen Punkt ein Cluster über die Epsilon-Umgebung des
 * Punkts
 * @param point Punkt zu dem Cluster über Epsilon-Umgebung ermittelt wird
 * @param clusterid Cluster Id des zu erzeugenden Clusters
 * @param epsilon Epsilon-Wert für Epsilon-Umgebung
 * @param minpoints Minimale Anzahl von Punkten in Epsilon-Umgebung
 * @return Boolean-Wert der angibt ob Cluster zu Punkt ermittelt wurde (wenn nein --> Noise)
 */
bool DBScan::expandCluster(Point *point, int clusterid, double epsilon, int minpoints){
        // Epsilon-Umgebung zu Punkt wird ermittelt
        long time_it_start, time_it_end;
        time_it_start = clock();
        list<Point> seeds = _database->getNeighbourhood(point,epsilon);
        list<Point>::iterator i,j;

        // Wenn Anzahl der Punkte die Anzahl der minimal geforderten Punkte nicht erreicht wird
        // der übergebene Punkt als Noise identifiziert
        if (seeds.size() < minpoints){
                _database->updateClusterId(point,ID_NOISE);
                return false;
        }

        Cluster *cluster = new Cluster(_dimensions, _nearestNumPoints, _farestNumPoints, clusterid,
                                                                                        _file_log);
        fprintf(_file_log,"%s %i \n","Neuer Cluster:",cluster->getId());
        fprintf(_file_log,"%s %i \n","e-Nachbarschaft:",seeds.size());
        fflush(_file_log);

        // Punkte der Epsilon-Umgebung werden zu Cluster hinzugefügt und Cluster ID der
        // Punkte wird gesetzt
        for (i = seeds.begin(); i != seeds.end(); i++){
                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i->getClusterId() == ID_NOISE))
                        _database->updateClusterId(i->getPoint(),clusterid);
                i->setClusterId(clusterid);
                cluster->addPoint(i->getPoint());
                cluster->addNearestPoint(i->getPoint());
        }
        _database->updateClusterId(point,clusterid);
        point->setClusterId(clusterid);
        cluster->addPoint(point);

        // Zu den ermittelten Punkte wird wieder die Epsilon-Umgebung erzeugt, bis es keine mehr gibt
        // Die gefundenen Punkte werden wieder zum Cluster hinzugefügt und die Cluster Id der Punkte
        // wird gesetzt
        while (!seeds.empty()){
                j = seeds.begin();
                list<Point> neighbourhood = _database->getNeighbourhood(j->getPoint(),epsilon);

                if (neighbourhood.size() >= minpoints)
                        for (i = neighbourhood.begin(); i != neighbourhood.end(); i++)
                                if ((i->getClusterId() == ID_UNCLASSIFIED) || (i->getClusterId() ==
                                                                                        ID_NOISE)){
                                        if (i->getClusterId() == ID_UNCLASSIFIED)
                                                seeds.push_back(*i);          // Hinzufügen zu seeds
                                        _database->updateClusterId(i->getPoint(), clusterid);

                                        cluster->addPoint(i->getPoint());
```

```
                                        cluster->addNearestPoint(i->getPoint());
                                }
                        else
                                cluster->addFarestPoint(j->getPoint());
                        seeds.pop_front();          //      entferne Punkt aus seeds
                        neighbourhood.clear();
                }
                seeds.clear();
                cluster->updateClusterMean();
                cluster->computeStdDev();
                time_it_end = clock();
                fprintf(_file_log, "%s %d %s %d \n", "Cluster: ", clusterid, "Anzahl Punkte: ", cluster-
                                                                                >getNumPoints());
                fprintf(_file_log,"%s %f \n","Dauer:",(time_it_end - time_it_start)/(float)CLOCKS_PER_SEC);
                fflush(_file_log);
                addCluster(cluster);
                return true;
}

/**
 * Methode getCluster liefert Cluster zu einer Id
 * @return Cluster zu übergebener Id
 */
Cluster* DBScan::getCluster(int id){
        Cluster *m, *cluster;

        for(m = _clusters; m; m = m->getNext())
                if(m->getId() == id)
                        cluster = m;

        return cluster;
}

/**
 * Ergebnis des Clusters (Mittelpunkt, Standardabweichung, Anzahl der Punkte)
 * wird in File geschrieben
 * @param filename File in das Ergebnisdaten des Clusters geschrieben werden
 */
void DBScan::writeClusterToFile (char *filename){
        FILE *clusterfile;
        Cluster *c;
        int i = 1;

        clusterfile = fopen(filename, "a");

        for(c = _clusters; c; c = c->getNext())  {
                fprintf(clusterfile, "%s %d\n", "Cluster:", i++);
                c->print(clusterfile);
        }

        fclose(clusterfile);
}

/**
 * Methode addCluster fügt erzeugtes Cluster zu Liste der Cluster hinzu
 * @param cluster Cluster der hinzgefügt werden soll
 */
void DBScan::addCluster(Cluster *cluster){
        if(_clusters)
                _clusters->setPrevious(cluster);
        // Setzt nächsten Cluster
        cluster->setNext(_clusters);
```

```
	_clusters = cluster;
}

/**
 * Methode writeClusterNearestandFarestToDatabase schreibt näheste und entfernteste Punkte in
 * Datenbank
 * @param clusters Cluster deren näheste und entfernteste Punkte in Datenbank geschrieben werden
 *                          sollen
 * @param table_nearest Name der zu erzeugenden Tabelle für näheste Punkte der Cluster
 * @param table_farest Name der zu erzeugenden Tabelle für entfernteste Punkte
 */
void DBScan::writeClusterNearestandFarestToDatabase(char *table_nearest, char *table_farest){
	_database->writeClusterNearestandFarestToDatabase(_clusters,table_nearest,table_farest);
}

/**
 * Methode writeDatabaseInformation schreibt Hilfsinformationen zu Datenbasis in Datenbank
 * @param table_database Name der zu erzeugenden Tabelle für Hilfsinformationen zu Datenbasis
 */
void DBScan::writeDatabaseInformation(char *table_database){
	_database->writeDatabaseInformation(table_database);
}

/**
 * Methode writeClusterInformation schreibt Hilfsinformationen zu den ermittelten Clustern in Datenbank
 * @param clusters Cluster deren Hilfsinformationen in die Datenbank geschrieben werden sollen
 * @param table_cluster Name der zu erzeugenden Tabelle für Hilfsinformationen zu Clustern
 */
void DBScan::writeClusterInformation(char *table_cluster){
	_database->writeClusterInformation(_clusters, table_cluster);
}
```