Technische Universität Kaiserslautern
Fachbereich Informatik

AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder

AG Heterogene Informationssysteme
Prof. Dr.-Ing. Stefan Deßloch

Integriertes Seminar
Datenbanken und Informationssysteme
Sommersemester 2005
Thema: Data Streams

Data Mining auf Datenströmen

Andreas M. Weiner
a_weiner@informatik.uni-kl.de

Betreuer: Dipl.-Inf. Boris Stumm

Zusammenfassung

In dieser Seminararbeit befasse ich mich mit dem Data Mining auf Datenströmen. Dazu werde ich zunächst erklären was man unter Data Mining allgemein versteht und grundlegende Verfahren des Data Minings (Klassifikation, Cluster-Analyse und Assoziationsregeln) vorstellen. Danach werde ich Anforderungen an Data-Mining-Verfahren formulieren, so dass diese auf Datenströmen angewendet werden können. Zum Schluss werde ich Verfahren zur Cluster-Analyse, Klassifikation und zum Data Mining von Assoziationsregeln auf Datenströmen vorstellen.

Inhaltsverzeichnis

1 Einleitung

Systeme zur Online-Verarbeitung von Sensordaten oder Börsenkursen sind äußerst datenintensive Applikationen, die mit herkömmlichen Datenbankverwaltungssystemen (DBVS) schwer bzw. überhaupt nicht realisiert werden können. Hierzu zählen auch Anwendungen zur sog. *Intrusion Detection*[1] in Netzwerken, Anwendungen zur Analyse von Sensor-Netzwerken und zur Abrechnung von Telekommunikationsverbindungen.

Klassische DBVS sind in der Regel passiv und nur begrenzt in der Lage, riesige Datenmengen zu speichern, wie sie z.B. bei der Überwachung von Sensordaten anfallen. Die ACID-Eigenschaften stellen sicher, dass Zugriffe auf die Datenbanken synchronisiert werden und jede Anfrage eine eindeutige Antwort erzeugt. Aktives Verhalten wird zwar grundsätzlich unterstützt, Trigger und Alerter wurden bisher aber eher stiefmütterlich behandelt. Carney et al. [CCC+02] bezeichnen dieses Modell der Informationsverarbeitung als *Human-Active, DBMS-Passive (HADP)*.

Will man hingegen Daten, wie sie z.B. im Rahmen von Anwendungen zur Trendanalyse von Börsenkursen anfallen, in Echtzeit und in unterschiedlichen Geschwindigkeiten verarbeiten, so stoßen konventionelle DBVS schnell an ihre Grenzen. Sie können die immense Datenflut gar nicht oder nur kurzzeitig speichern. Diese neuen Anforderungen verlangen nach einem grundlegenden Paradigmenwechsel im Sinne von Kuhn [Kuh93]. Bei Carney et al. [CCC+02] heißt dieses neue Verarbeitungsparadigma: *DBMS-Active, Human-Passive (DAHP)*. In der wissenschaftlichen Literatur hat sich die Bezeichnung Datenströme (*engl. Data Streams)* durchgesetzt. Babcock et al. [BBD+02] bieten einen guten Überblick über die Anwendung und Verarbeitungsmodelle von Datenströmen. Aus meiner Sicht lässt sich der Begriff des Datenstroms wie folgt definieren:

Definition 1.1 (Datenstrom) *Ein Datenstrom (engl.* Data Stream*) ist eine geordnete, in der Länge meist unbeschränkte, Sequenz von $x_1, \ldots, x_n$ Datenelementen, die in Echtzeit verarbeitet werden müssen, nicht wahlfrei zugreifbar sind und nur einmal oder in geringer Zahl erneut gelesen werden können.*

Im Zusammenhang mit der Verarbeitung von riesigen Datenmengen spielt *Data Mining* eine große Rolle. Dabei geht es um das Aufspüren von interessanten Mustern und Regeln in großen Datenmengen. Hier finden verschiedene Verfahren aus Informatik und Mathematik ihre Anwendung. Zum Beispiel werden Techniken aus den Bereichen maschinelles Lernen, Statistik, künstliche neuronale Netze und genetische Algorithmen verwendet. Han und Kamber [HK01] bieten einen guten Überblick über die grundlegenden Verfahren und Konzepte des Data Minings.

In Kapitel 2 werde ich klassische Data-Mining-Verfahren (Klassifikation, Cluster-Analyse und Assoziationsregeln) vorstellen. In Kapitel 3 werde ich zunächst zeigen, warum die vorgestellten Algorithmen und Konzepte in dieser Form nicht auf Datenströme anwendbar sind. Anschließend werde ich Verfahren vorstellen, die das Mining von Datenströmen ermöglichen.

[1] Unter *Intrusion Detection* versteht man allgemein das Aufspüren von Eindringlingen in Computer-Netzwerken mit dem Ziel einen möglichen Angriff abzuwehren.

2 Grundlegende Verfahren des Data Minings

In diesem Abschnitt stelle ich allgemeine Verfahren zum Data Mining vor, wie sie heute in betrieblichen Applikationen zur Anwendung kommen. Zunächst definiere ich, was man allgemein unter Data Mining versteht:

Definition 2.1 (Data Mining) *Unter* Data Mining *versteht man Verfahren zum Aufspüren von neuen und interessanten Mustern und Regeln, sowie von Beziehungen zwischen Datensätzen in großen Datenmengen.*

Beim Data Mining können nach Elmasri und Navathe [EN04] vier verschiedene Ziele unterschieden werden: Vorhersage, Identifikation, Klassifikation und Optimierung.

Erst im Zusammenspiel mit weiteren Techniken entfaltet Data Mining seine ganze Mächtigkeit. Nach Adriaans und Zantinge [AZ96] ist Data Mining Teil eines sechsstufigen *Knowledge-Discovery*-Prozesses. Der Prozess besteht aus 1. Datenauswahl, 2. Datenbereinigung, 3. Datenanreicherung, 4. Datentransformation, 5. Data Mining und 6. Berichterstattung und Reportgenerierung. Auf die Schritte 1–4 und 6 wird nachfolgend nicht weiter eingegangen. Es wird davon ausgegangen, dass die Daten bereits derart aufbereitet wurden, dass man Data-Mining-Verfahren anwenden kann.

2.1 Assoziationsregeln

Beim sog. *Association Rule Mining* sollen in großen Datenmengen interessante Abhängigkeiten und Beziehungen zwischen Datensätzen gefunden werden.

2.1.1 Einführendes Beispiel

Eine typische Anwendung ist die sog. *„Warenkorb-Analyse“*. Legt man zum Beispiel bei einem Online-Buchhändler wie Amazon ein Buch in seinen virtuellen Warenkorb, so werden einem verwandte Artikel angezeigt, die aufgrund des eigenen Kaufverhaltens interessant sein könnten. Assoziationsregeln lassen sich in Form von prädikatenlogischen Formeln darstellen:

Beispiel 2.1 (Assoziationregel) $kauft(X, \text{„Buddenbrooks“}) \Rightarrow kauft(X, \text{„Der Zauberberg“})$ *[support=5%, confidence=65%]*

Die Maße *support* und *confidence* im Beispiel 2.1 haben folgende Bedeutung: Das Maß support gibt an, dass 5% der analysierten Datensätze darauf hinweisen, dass die Bücher „Buddenbrooks“ und „Der Zauberberg“ zusammen gekauft wurden. Das Maß confidence zeigt an, dass 65% der Kunden, die das Buch „Buddenbrooks“ gekauft haben, auch „Der Zauberberg“ kauften. Assoziationsregeln sind genau dann interessant, wenn support und confidence über einem benutzerdefinierten minimalen Schwellwert liegen.

2.1.2 Der Apriori-Algorithmus

Der Apriori-Algorithmus ermöglicht das Aufspüren von häufig vorkommenden Mustern (engl. *frequent patterns*) in großen Datenmengen und wurde ursprünglich von Agrawal und Srikant [AS94] entwickelt. Zunächst werde ich einige wichtige Begriffe einführen, die wir für das Verständnis des Apriori-Algorithmus benötigen.

Definition 2.2 (Assoziationsregeln) *Sei* $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ *eine Menge von Items. Sei D, die Menge der zu untersuchenden Daten, eine Menge von Transaktionen, wobei jede Transaktion* $T \in D$ *eine Menge von Items ist, derart, dass* $T \subseteq \mathcal{I}$*. Jede Transaktion wird eindeutig durch einen*

Schlüssel TID identifiziert. Sei A eine Menge von Items. Eine Transaktion T enthält A dann und nur dann, wenn $A \subseteq T$ gilt. Eine Assoziationsregel (engl. Assoziation Rule*) ist eine Implikation der Form $A \Rightarrow B$, wobei gilt: $A \subset \mathcal{I}$, $B \subset \mathcal{I}$ und $A \cap B = \emptyset$.*

Definition 2.3 (support) *Die Regel $A \Rightarrow B$ hat einen* support*-Wert von s, wenn s% der Transaktionen in D $A \cup B$ enthalten.*

$$support(A \Rightarrow B) = \frac{\#\_Transaktionen\_die\_A\_und\_B\_enthalten}{\#\_aller\_Transaktionen}$$

Definition 2.4 (confidence) *Die Regel $A \Rightarrow B$ hat einen* confidence*-Wert von c, wenn c% der Transaktionen in D, die A enthalten auch B enthalten.*

$$confidence(A \Rightarrow B) = \frac{\#\_Transaktionen\_die\_A\_und\_B\_enthalten}{\#\_Transaktionen\_die\_A\_enthalten}$$

Definition 2.5 (starke Assoziationsregel) *Sei s_{min} ein minimaler Schwellwert für support und sei c_{min} ein minimaler Schwellwert für confidence. Eine Assoziationsregel $A \Rightarrow B$ heißt stark (engl.* strong*), wenn gilt: $support(A \Rightarrow B) \geq s_{min}$ und $confidence(A \Rightarrow B) \geq c_{min}$.*

Eine Menge von k Items nennen wir ein k-Itemset. Die Auftrittshäufigkeit (engl. *occurence frequency*) eines Itemsets ist die Anzahl der Transaktionen, die das Itemset enthalten. Ein Itemset erfüllt die *Minimum-Support-Bedingung* genau dann, wenn die Auftrittshäufigkeit des Itemsets größer oder gleich dem Produkt aus s_{min} und der Anzahl der Transaktionen in D ist. Erfüllt ein Itemset diese Bedingung, so kommt dieses Itemset häufig vor (engl. *frequent itemset*). Die Menge der häufig vorkommenden k-Itemsets sei mit L_k bezeichnet.

Die Suche nach Assoziationsregeln findet in einem zweistufigen Prozess statt: Zunächst müssen alle häufig vorkommenden Itemsets gefunden werden. Danach werden daraus starke Assoziationsregeln abgeleitet.

Der Apriori-Algorithmus benutzt früheres Wissen um häufig vorkommende Itemsets zu finden. Er benutzt ein iteratives Verfahren, das auch als *level-wise search* bekannt ist. Dabei werden die k-Itemsets benutzt um die $(k+1)$-Itemsets zu untersuchen. Zuerst wird also die Menge der häufig vorkommenden 1-Itemsets gefunden, die mit L_1 bezeichnet wird. Dann wird die Menge L_2 der häufig vorkommenden 2-Itemsets gefunden usw. bis keine k-Itemsets mehr gefunden werden. Für das Auffinden jeder Menge L_k ist ein vollständiger Scan der Datenbank notwendig. Aus diesem Grund wird zur Effizienzsteigerung die *Apriori-Eigenschaft* ausgenutzt. Die Apriori-Eigenschaft besagt, dass jede nichtleere Teilmenge eines häufig vorkommenden Itemsets ebenfalls häufig vorkommen muss. Liegt der support-Wert des Itemsets I unter s_{min}, so kommt I nicht häufig vor. Wird nun ein Item i in I eingefügt – also $I \cup \{i\}$ –, so kommt $I \cup \{i\}$ nicht häufiger vor als I. Der Apriori-Algorithmus besteht aus 2 Schritten:

1. **Verbund** Um L_k zu finden wird eine Kandidaten-Menge von k-Itemsets erzeugt, indem die Menge L_{k-1} mit sich selbst verbunden wird. Wir bezeichnen diese Kandidaten-Menge mit C_k. Seien l_1 und l_2 Itemsets in L_{k-1}. Die Notation $l_i[j]$ bezeichne das j-te Item in l_i. Der Apriori-Algorithmus setzt voraus, dass die Items innerhalb einer Transaktion oder innerhalb eines Itemsets in lexikographischer Ordnung sortiert sind. Elemente von L_{k-1} können miteinander verbunden werden – $L_{k-1} \bowtie L_{k-1}$ –, wenn ihre ersten $(k-2)$ Items übereinstimmen. Das bedeutet, dass $l_1, l_2 \in L_{k-1}$ verbunden werden können, wenn gilt:
   $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \ldots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$.

2. **Beschneidung** C_k ist eine Obermenge von L_k, d.h. Elemente von C_k kommen nicht notwendigerweise häufig vor, aber alle k-Itemsets die häufig vorkommen, sind in L_k enthalten. Um die Größe von C_k zu reduzieren, wird die Apriori-Eigenschaft benutzt. Jedes $(k-1)$-Itemset, das nicht häufig vorkommt, kann nicht Teilmenge eines häufig vorkommenden k-Itemsets sein. Wenn es also eine $(k-1)$-elementige Teilmenge von Kandidaten-k-Itemsets gibt, die nicht in L_{k-1} liegt, so kann diese nicht häufig vorkommen und kann aus C_k entfernt werden.

Im Anhang auf Seite 25 ist der Apriori-Algorithmus im Pseudocode dargestellt.

2.1.3 Anwendung

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Abbildung 1: Beispieldatenbank mit 9 Transaktionen ([HK01] S. 232)

Abbildung 1 zeigt eine Datenbank mit 9 Transaktionen, anhand derer die Anwendung des Apriori-Algorithmus demonstriert werden soll. Der Ablauf des Algorithmus lässt sich mit Hilfe der Abbildung 2 nachvollziehen.

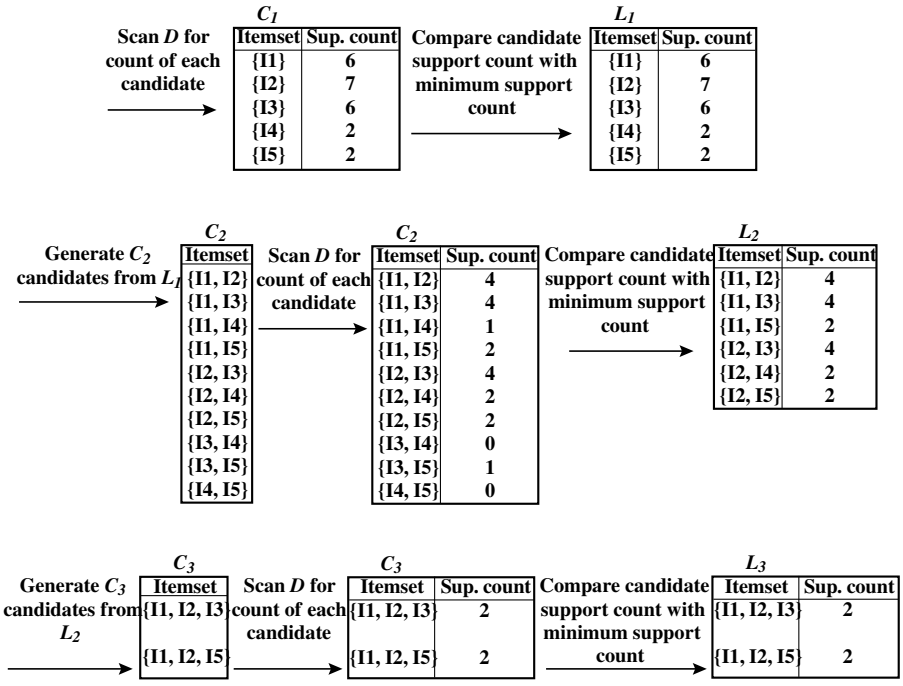

Abbildung 2: Erzeugung der Kandidaten- und häufig vorkommenden Itemsets ([HK01] S. 233)

1. In der ersten Iteration ist jedes Item Element von C_1 – der Kandidaten-Menge von 1-Itemsets. Der Algorithmus führt einen Scan der Transaktionen durch, um die Auftrittshäufigkeit jedes Items zu bestimmen.

2. Es gelte: $min\_sup = 2$. Dann kann die Menge der häufig vorkommenden 1-Itemsets bestimmt werden. Sie besteht aus genau den Kandidaten 1-Itemsets, welche die Minimum-Support-Bedingung erfüllen.

3. Um die Menge der häufig vorkommenden 2-Itemsets L_2 zu bestimmen, benutzt der Algorithmus $L_1 \bowtie L_1$, um die Kandidaten 2-Itemsets C_2 zu erzeugen.

4. Jetzt wird erneut ein Scan der Transaktionen durchgeführt und der support-Wert berechnet.

5. Danach wird die Menge der häufig vorkommenden 2-Itemsets bestimmt. Sie besteht aus genau den Elementen von C_2, welche die Minimum-Support Bedingung erfüllen. Zum Beispiel ist die Menge $\{I3, I5\}$ nicht in L_2 enthalten, da ihr support-Wert gleich 1 ist.

6. Bei der Erzeugung von C_3 wird erstmals der Suchraum beschnitten:

   - Der Join-Step des Algorithmus liefert:
     $C_3 = L_2 \bowtie L_2 = \big\{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\big\} \bowtie$
     $\big\{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\big\} =$
     $\big\{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\big\}$
   - Der Prune-Step löscht folgende Itemsets aus C_3:
     - {I1, I3, I5}: da $\{I3, I5\} \subset \{I1, I3, I5\}$, aber $\{I3, I5\} \notin L_2$
     - {I3, I3, I4}: da $\{I3, I4\} \subset \{I1, I3, I4\}$, aber $\{I3, I4\} \notin L_2$
     - {I2, I3, I5}: da $\{I3, I5\} \subset \{I2, I3, I5\}$, aber $\{I3, I5\} \notin L_2$
     - {I2, I4, I5}: da $\{I4, I5\} \subset \{I2, I4, I5\}$, aber $\{I4, I5\} \notin L_2$

     $\Longrightarrow C_3 = \big\{\{I1, I2, I3\}, \{I1, I2, I5\}\big\}$

7. Nun wird wieder ein Scan der Datenbank D durchgeführt, um L_3 zu bestimmen.

8. Der Join-Step liefert nun: $C_4 = L_3 \bowtie L_3 = \big\{\{I1, I2, I3, I5\}\big\}$. Da aber $\{I2, I3, I5\} \subset \{I1, I2, I3, I5\}$ und $\{I2, I3, I5\} \notin L_3 \Longrightarrow C_4 = \emptyset \Longrightarrow L_3$ enthält alle häufig vorkommenden Itemsets mit $min\_sup \geq 2 \Longrightarrow$ Terminierung des Algorithmus.

Nachdem wir nun die häufig vorkommenden Itemsets bestimmt haben, können wir daraus starke Assoziationsregeln ableiten. Der confidence-Wert lässt sich mit Hilfe von Definition 2.4 wie folgt bestimmen: $confidence(A \Rightarrow B) = P(B|A) = \frac{support\_count(A \cup B)}{support\_count(A)}$. Der Wert $support\_count(A \cup B)$ ist hierbei die Anzahl der Transaktionen, die das Itemset $A \cup B$ enthalten und $support\_count(A)$ ist die Anzahl der Transaktionen die das Itemset A enthalten. Assoziationsregeln können folgendermaßen erzeugt werden:

1. Erzeuge für jedes Itemset l alle nichtleeren Teilmengen von l.

2. Für jede nichtleere Teilmenge s von l gib die Regel „$s \Rightarrow (l - s)$“ aus, wenn $\frac{support\_count(l)}{support\_count(s)} \geq c\_min$.

Sei $c\_min = 70\%$ und sei $s\_min = 2/9 = 22\%$. Die nichtleeren Teilmengen von $l = \{I1, I2, I5\}$ sind: $\{I1, I2\}, \{I1, I5\}, \{I2, I5\}, \{I1\}, \{I2\}, \{I5\}$.
$confidence(I1 \wedge I2 \Rightarrow I5) = 2/4 = 50\%$
$confidence(I1 \wedge I5 \Rightarrow I2) = 2/2 = 100\%$
Die erste Assoziationsregel ist nicht stark, da ihr confidence-Wert kleiner als 70% ist; die zweite hingegen schon.

2.2 Cluster-Analyse

Bei der Cluster-Analyse werden physische oder abstrakte Objekte Clustern zugeordnet. Ein Cluster bezeichnet eine Menge von Objekten, die eine gewisse Ähnlichkeit zu Elementen des gleichen Clusters aufweisen, aber verschieden von den Elementen anderer Cluster sind. Ein Cluster wird durch ein kanonisches Element – dem Cluster-Zentrum – bestimmt. Die Cluster-Analyse wird in einer Vielzahl von Anwendungen wie z.B. der Marktanalyse verwendet.

Definition 2.6 (Clustering-Problem) *Gegeben sei eine Datenbank mit n Datensätzen. Desweiteren sei k die Anzahl der Cluster ($k \leq n$) die gebildet werden sollen. Ziel ist es, die Cluster so zu bilden, dass die Ähnlichkeitsfunktion (z.B. ein Distanzmaß) optimiert wird, so dass die Ähnlichkeit innerhalb der Cluster maximal und die Ähnlichkeit zwischen den Clustern minimal wird.*

2.2.1 Der k-Means-Algorithmus

Der *k-Means-Algorithmus* ist ein bekanntes Verfahren zur Cluster-Analyse und wurde 1967 von MacQueen [Mac67] vorgestellt. Der k-Means-Algorithmus nimmt als Eingabe k – die Anzahl der zu erstellenden Cluster – und n Objekte die den k Clustern zugeordnet werden sollen. Die Cluster-Ähnlichkeit wird mit Hilfe einer Distanzfunktion bestimmt. Im Anhang auf Seite 26 ist der Algorithmus im Pseudocode nachvollziehbar.

Der Algorithmus arbeitet folgendermaßen: Zuerst werden zufällig k Objekte ausgewählt, die zu Beginn einelementige Cluster bilden. Danach werden die verbleibenden Objekte demjenigen Cluster zugeordnet, zu dem sie die größte Ähnlichkeit aufweisen. Anschließend wird auf jedes Objekt die Abstandsfunktion angewendet. Nun wird jedes Objekt dem Cluster zugewiesen, zu dem die größte Ähnlichkeit besteht. Dieses Verfahren wird solange fortgesetzt, bis die Zielfunktion konvergiert.

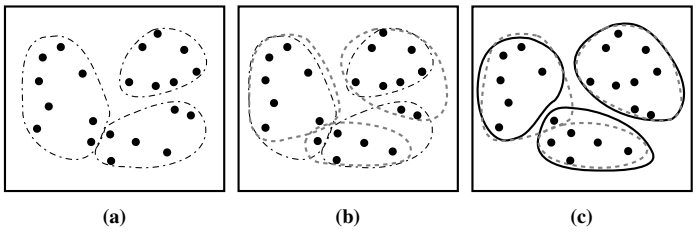

Abbildung 3: Cluster-Analyse mit dem k-Means-Algorithmus ([HK01] S. 350)

2.2.2 Der BIRCH-Algorithmus

Der 1996 von Zhang et al. [ZRL96] vorgestellte *BIRCH*-Algorithmus (***B**alanced **I**terative **R**educing and **C**lustering using **H**ierarchies*) gehört zur Klasse der hierarchischen Clustering-Verfahren. Hierarchische Methoden zur Cluster-Analyse fügen Datenobjekte in einen Baum von Clustern ein.

Zhang et al. [ZRL96] führen in ihrem Aufsatz den sog. *CF-Baum* (engl. *clustering feature tree*) ein. Diese Datenstruktur dient der „Komprimierung“ der Cluster-Repräsentation. Ein Clustering Feature (CF) ist ein Tripel, das Daten eines Subclusters von Objekten zusammenfasst. Seien N d-dimensionale Punkte oder Objekte $\{o_i\}$ mit $1 \leq i \leq N$ in einem Subcluster gegeben, dann ist ein CF wie folgt definiert: $CF = (N, \vec{LS}, SQ)$. Dabei ist N die Anzahl der Punkte im Subcluster, $\vec{LS}$ ist die lineare Summe (engl. *linear sum*) $\sum_{i=1}^{N} \vec{o_i}$, und SQ die quadrierte Summe (engl. *squared sum*) der n Punkte $\sum_{i=1}^{N} \vec{o_i}^2$. Ein CF-Baum ist ein höhen-balancierter Baum, der die *Clustering*

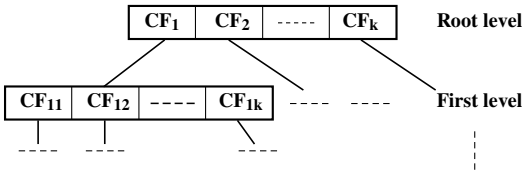

Abbildung 4: Clustering-Feature Baum ([HK01] S. 357)

Features verwaltet. Jeder Knoten, mit Ausnahme des Blattes, speichert die Summe der CFs seiner Söhne und fasst damit Clustering-Informationen über diese zusammen. Ein CF-Baum hat zwei Parameter: den Verzweigungsfaktor B (engl. *branching factor*) und den Schwellwert T (engl. *threshold*). Der Verzweigungsfaktor beschränkt die Anzahl der Söhne, die ein Knoten maximal haben kann, nach oben. Der Schwellwert gibt den maximalen Durchmesser für Subcluster an, die an den Blätter gespeichert werden. Der BIRCH-Algorithmus besteht im Wesentlichen aus zwei Phasen:

- **Phase 1** Der Algorithmus führt einen Scan der Datenbank durch und baut den initialen CF-Baum auf.
- **Phase 2** BIRCH wendet einen (beliebigen) Clustering-Algorithmus auf die Blätter des CF-Baums an.

In der ersten Phase wird der CF-Baum dynamisch durch das Einfügen der Datenobjekte erzeugt. Ein Datenobjekt wird demjenigen Clustering Feature hinzugefügt, dem es am nächsten ist. Übersteigt der Durchmesser des Subclusters, in welchen das Datenobjekt eingefügt wurde, den Schwellwert T, so wird der Baum rebalanciert. Nachdem ein neues Objekt in den Baum eingefügt wurde, werden die Daten in der Wurzel des CF-Baums aktualisiert. Der Speicherbedarf des Baums lässt sich über den Schwellwert T steuern. Übersteigt der Speicherbedarf, der benötigt wird um einen CF-Baum im Speicher zu halten, den vorhanden Speicher, so kann der Schwellwert verringert werden und der Baum neu aufgebaut werden. Es ist kein kompletter Neuaufbau des Baums notwendig, vielmehr wird hier ein Verfahren – ähnlich dem Restrukturierungs- und Splitting-Verfahren in $B^\star$-Bäumen – angewendet. Nachdem der Baum vollständig aufgebaut wurde, kann ein beliebiger Clustering-Algorithmus (z.B. der k-Means-Algorithmus) für die zweite Phase verwendet werden.

2.3 Klassifikation

Bei der Klassifikation geht es um die Partitionierung der Daten in disjunkte Klassen oder Kategorien. Um diese Aufgabe zu lösen, greift ein Algorithmus zur Klassifikation auf vorhandenes Wissen zurück. Dieses Wissen ist durch eine Menge von disjunkten Klassen gegeben, in die bereits Datensätze partitioniert wurden. Die bereits klassifizierten Daten werden benutzt, um einen Klassifikator zu lernen. Der Klassifikator kann dann verwendet werden, um neue Datensätze einer bestimmten Klasse zuzuordnen.

So kann ein eMail-Client mit Hilfe der Klassifikation unerwünschte Nachrichten erkennen und diese in ein Spam-Verzeichnis verschieben oder sofort löschen.

2.3.1 Entscheidungsbaum-Induktion

Ein bekanntes Verfahren zur Klassifikation ist die Entscheidungsbaum-Induktion (engl. *Induction of Decision Trees*). Quinlan [Qui86] hat 1986 den ID3-Algorithmus zur Entscheidungsbaum-Induktion vorgestellt, den ich in diesem Abschnitt erläutern möchte.

Ein Entscheidungsbaum (engl. *decision tree*) nimmt als Eingabe ein Objekt oder eine Situation, die durch eine Menge von Attributen beschrieben wird, und liefert eine „Entscheidung“ zurück (vgl. [RN03] S. 653 und [Mit97] S. 52). Die Attribute können nummerisch oder nominal sein; ich werde hier aber nur den nominalen Fall behandeln. Jeder innere Knoten des Baums entspricht einem Test des Attributs und jeder Zweig zeigt den Ausgang des Tests an. Die Blätter des Baums stehen für die verschiedenen Klassen bzw. Kategorien. Der Entscheidungsbaum in Abbildung 5 zeigt einen typischen Entscheidungsbaum, der feststellt ob eine Person einen Computer kaufen wird oder nicht. Der im Anhang auf Seite 27 im Pseudocode vorgestellte Algorithmus zur

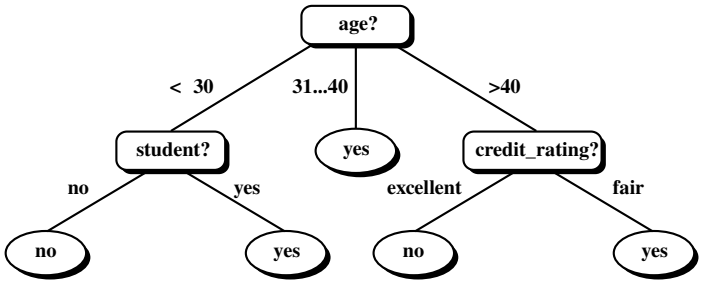

Abbildung 5: Entscheidungsbaum ([HK01] S. 284)

Entscheidungsbaum-Induktion ist ein Greedy-Algorithmus, der top-down vorgeht und ein rekursives divide-and-conquer-Verfahren verwendet. Der Algorithmus arbeitet folgendermaßen:

- Im ersten Schritt wird der Wurzelknoten des Baums erzeugt.
- Liegen die Datensätze alle in der gleichen Klasse, so wird aus dem Knoten ein Blatt, das dann mit der Klassenbezeichnung versehen wird.
- Im anderen Fall wird der unten vorgestellte entropie-basierte Informationsgewinn berechnet. Das Attribut mit dem höchsten Informationsgewinn wird zum „Test“-Attribut des Knotens und wird verwendet, um die Klassen zu partitionieren.
- Für jeden bekannten Wert des Attributs wird nun ein Zweig erzeugt und die Datensätze entsprechend partitioniert.
- Dasselbe Verfahren wird nun rekursiv angewendet um den Entscheidungsbaum zu erzeugen.
- Die Rekursion stoppt genau dann, wenn eine der folgenden Bedingungen nicht mehr erfüllt ist:
  (a) Alle Datensätze eines Knotens liegen in der gleichen Klasse.
  (b) Es gibt keine weiteren Attribute mit deren Hilfe eine Partitionierung möglich wäre. In diesem Fall wird der Knoten in ein Blatt umgewandelt und wird mit der Klasse ausgezeichnet, die am häufigsten in den Datensätzen auftritt (engl. *majority vote*).
  (c) Die Menge s_i ist leer. In diesem Fall wird ein Blatt erzeugt und mit dem Klassenlabel versehen, welches sich nach dem Mehrheitsprinzip ergibt.

Definition 2.7 (Informationsgewinn) *Sei S eine Menge von s Datensätzen. Man nehme an, dass das Klassifizierungsattribut m unterschiedliche Werte hat und damit m verschiedene Klassen*

C_i (für $i = 1, \ldots, m$) definiert. Sei s_i die Anzahl der Datensätze aus S, die in C_i liegen. Der erwartete Informationsgehalt ist gegeben durch:

$$I(s_1, s_2, \ldots, s_m) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{1}$$

wobei $p_i = s_i/s$ der Wahrscheinlichkeit entspricht, dass ein beliebiger Datensatz zur Klasse C_i gehört. Das Attribut A habe v verschiedene Werte $\{a_1, a_2, \ldots, a_v\}$. A kann dazu verwendet werden, um S in v Teilmengen $\{S_1, S_2, \ldots, S_v\}$ zu partitionieren, wobei S_j genau die Datensätze aus S enthält, die den Wert $a_j \in A$ haben. Sei s_{ij} die Anzahl der Datensätze der Klasse C_i in einer Teilmenge S_j von S. Dann ist die Entropie gegeben durch:

$$E(A) = \sum_{j=1}^{v} \frac{s_{1j} + \ldots + s_{mj}}{s} I(s_{1j}, \ldots, s_{mj}). \tag{2}$$

Für eine Teilmenge S_j von S gilt dann:

$$I(s_{1j}, \ldots, s_{mj}) = -\sum_{i=1}^{m} p_{ij} \log_2(p_{ij}). \tag{3}$$

wobei $p_{ij} = \frac{s_{ij}}{|S_j|}$ die Wahrscheinlichkeit bezeichne, dass ein Datensatz aus S_j zur Klasse C_i gehört. Zusammen mit (1) erhält man nun den Informationsgewinn

$$IG(A) = I(s_1, s_2, \ldots, s_m) - E(A). \tag{4}$$

2.3.2 Beispiel

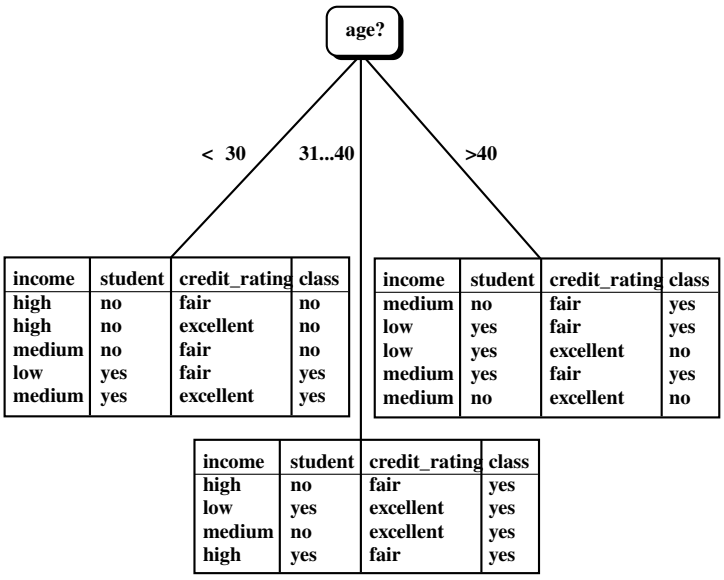

Abbildung 6: Entscheidungsbaum nach der Wahl von *age* als Wurzelknoten ([HK01] S. 289)

Die nachfolgende Tabelle ([HK01] S. 288) zeigt eine Trainingsdatenbank mit deren Hilfe ein Entscheidungsbaum aufgebaut werden soll:

RID	age	income	student	credit_rating	Class: buys_computer
1	≤ 30	high	no	fair	no
2	≤ 30	high	no	excellent	no
3	$31 \ldots 40$	high	no	fair	yes
4	> 40	medium	no	fair	yes
5	> 40	low	yes	fair	yes
6	> 40	low	yes	excellent	no
7	$31 \ldots 40$	low	yes	excellent	yes
8	≤ 30	medium	no	fair	no
9	≤ 30	low	yes	fair	yes
10	> 40	medium	yes	fair	yes
11	≤ 30	medium	yes	excellent	yes
12	$31 \ldots 40$	medium	no	excellent	yes
13	$31 \ldots 40$	high	yes	fair	yes
14	> 40	medium	no	excellent	no

Das Attribut *buys_computer* hat zwei verschiedene Werte $\{yes, no\}$; also können daraus zwei Klassen gebildet werden. ($m = 2$). Die Klasse C_1 entspreche *yes* und C_2 entspreche *no*. Es gibt neun Datensäte mit *buys_computer=yes* und fünf Datensätze mit *buys_computer=no*. Um den Informationsgewinn jedes Attributs zu berechnen, verwenden wir zunächst Gleichung (a) aus Definition 2.7. Man erhält: $I(s_1, s_2) = I(9, 5) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0,940$. Als Nächstes müssen wir mit Gleichung (2) aus Definition 2.7 für jedes Attribut die Entropie bestimmen: Das Attribut *age* kann drei Werte annehmen: $\{\leq 30, 31 \ldots 40, > 40\}$.

- Für *age*=≤ 30 erhält man:
  $s_{11} = 2$, $s_{21} = 3$ und somit $I(s_{11}, s_{21}) = 0,971$
- Für *age*=$31 \ldots 40$ erhält man:
  $s_{12} = 4$, $s_{22} = 0$ und somit $I(s_{12}, s_{22}) = 0$
- Für *age*=> 40 erhält man:
  $s_{13} = 3$, $s_{23} = 2$ und somit $I(s_{13}, s_{23}) = 0,971$

Nun kann man die Entropie von *age* berechnen:
$E(age) = \frac{5}{14}I(s_{11}, s_{21}) + \frac{4}{14}I(s_{12}, s_{22}) + \frac{5}{14}I(s_{13}, s_{23}) = 0,694$.
Der Informationsgewinn einer Partitionierung nach *age* würde somit:
$IG(age) = I(s_1, s_2) - E(age) = 0,246$ betragen.

Analog kann man den Informationsgewinn für die restlichen Attribute berechnen. Da *age* den höchsten Informationsgewinn hat, wird dieses Attribut zum Wurzelknoten des Entscheidungsbaums. Abbildung 6 zeigt den Entscheidungsbaum nachdem *age* als Wurzel gewählt wurde. Setzt man dieses Verfahren rekursiv fort, so erhält man den in Abbildung 5 dargestellten Entscheidungsbaum. Nachdem wir nun einen Klassifikator aus den Trainingsdaten gelernt haben, können wir den Entscheidungsbaum benutzen um Kunden in zwei disjunkte Klassen – Käufer und Nichtkäufer – zu partitionieren. Haben wir es beispielsweise mit einem höchstens 30 Jahre alten Studenten mit geringem Einkommen und mittelmäßiger Kreditwürdigkeit zu tun, so kann dieser sofort der Klasse der Käufer zugeordnet werden. Diesem Kunden könnten wir dann den neuesten Prospekt mit Computerzubehör zuschicken.

3 Data Mining auf Data Streams

Domingos und Hulten [DH01] haben bei einem Workshop im Jahre 2001 Forschungsziele für das Data Mining auf Datenströmen formuliert. Nach ihren Angaben fallen beim Verkauf von Produkten des Einzelhandelskonzerns WalMart täglich 20 Millionen Transaktionen an. Bei der Suchmaschine Google werden täglich 70 Millionen Suchanfragen verarbeitet. Diese wahre Datenflut hat erhebliche Auswirkungen darauf, wie in Zukunft Data-Mining-Aufgaben gelöst werden. Die bisher bekannten Data-Mining-Algorithmen können nur einen Bruchteil dieser Daten verarbeiten. Domingos und Hulten [DH01] schlagen folgende Designkriterien für Data-Mining-Algorithmen auf Datenströmen vor:

- Die Verarbeitung eines Datensatzes muss in konstanter Zeit möglich sein, damit die Daten schneller verarbeitet werden können, als sie das Verarbeitungssystem erreichen.
- Der Algorithmus darf nur einen beschränkten Teil des Arbeitsspeichers in Anspruch nehmen, unabhängig davon, wieviele Datensätze bereits verarbeitet wurden.
- Das Verfahren muss mit einem einzigen Scan der Datensätze in der Lage sein, ein vernünftiges Modell zu erstellen, da es aufgrund des immensen Datenaufkommens nicht in der Lage ist, ältere Datensätze erneut zu verarbeiten oder diese aus dem beschränkten Sekundärspeicher zu laden.
- Zu jedem Zeitpunkt muss ein vernünftiges Modell vorliegen, da Datenströme potentiell unendlich lang sind. Bei klassischen Verfahren, die eine endliche Menge von Datensätzen verarbeiten, ist es hingegen ausreichend, dass am Ende des Verarbeitungsprozesses ein korrektes Modell vorliegt.
- Wünschenswert ist die Erzeugung eines Modells, das äquivalent zu einem Modell ist, welches von einem klassischen Data-Mining-Algorithmus ohne die beschriebenen Einschränkungen generiert wurde.
- Zu jeder Zeit sollte das Modell den aktuellsten Stand widerspiegeln und ältere und möglicherweise noch relevante Daten aufbewahren.

Drei Fragestellungen für die zukünftige Entwicklung von Data-Mining-Verfahren heben Domingos und Hulten [DH01] hervor:

1. Obwohl wir potentiell unendlich viele Datensätze zur Verfügung haben, stellt sich die Frage, wieviele Datensätze wir benötigen um ein vernünftiges Modell zu erstellen.
2. Wie können wir den Konflikt zwischen einer möglichst hohen Aktualität des Modells und der Aufbewahrung von alten Datensätzen lösen? Benutzt man sog. *Sliding Windows*, so führt ein zu großes Fenster zu einer schlechten Anpassung an neue Gegebenheiten und ein zu kleines Fenster zum Verlust von Informationen und zu einem trivialen Modell. Gibt es eine Möglichkeit dieses Spannungsverhältnis aufzulösen?
3. Welche Algorithmen sind am besten geeignet, um Data Mining auf sehr schnellen Datenströmen durchzuführen? Gibt es allgemeine Eigenschaften, die eine einfache Transformation einer Klasse von Algorithmen gestatten?

3.1 Cluster-Analyse

Die oben beschriebenen Algorithmus-Anforderungen haben gezeigt, dass der k-Means-Algorithmus unter diesen Einschränkungen nicht auf Datenströmen angewendet werden kann. O'Callagahn et al. [OMM+02] haben den *STREAM*-Algorithmus zur Cluster-Analyse auf Datenströmen entwickelt. Der Algorithmus benutzt den später vorgestellten *LSEARCH*-Algorithmus von Guha et al. [GMM+03] als Unterprogramm.

3.1.1 Der STREAM-Algorithmus

Der STREAM-Algorithmus nimmt an, dass die Daten das Verarbeitungssystem immer in $X_1, \ldots, X_n$ Haufen (engl. *chunks*) des Datenstroms erreichen. Die Chunks sind gerade noch so groß, dass sie in den Hauptspeicher passen. Der Algorithmus prüft zunächst, ob der Chunk aus weniger als k verschiedenen Elementen besteht, die nur mehrfach wiederholt werden. Ist dies der Fall, so werden die einzelnen Elemente entsprechend ihrer Häufigkeit gewichtet. Im anderen Fall wird der LSEARCH-Algorithmus auf den Chunk X_i angewendet. Wir erhalten dann für X_i k gewichtete Cluster-Zentren. Das Gewicht des Cluster-Zentrums ist die Summe der Gewichte seiner Elemente. Zum Schluss wenden wir den LSEARCH-Algorithmus auf alle gewichteten Cluster Zentren an, die wir zuvor für die Chunks $X_1, \ldots, X_i$ erstellt haben, um die gewichteten Zentren für den gesamten Datenstrom $X_1 \cup \cdots \cup X_i$ zu erhalten. im Anhang auf Seite 27 findet sich der Algorithmus im Pseudocode wieder.

Betrachtet man Datenströme die von Intrusion-Detection-Systemen untersucht werden, so fällt auf, dass die Datenströme aus wenigen, sich oft wiederholenden Daten bestehen. In diesem Fall ist der Algorithmus wesentlich effizienter, wenn er auf der gewichteten Darstellung arbeiten kann. Da die Erzeugung der gewichteten Darstellung $O(nq)$ Schritte benötigt, ist es wünschenswert, dass es sich fast nur um redundante Daten handelt.[2] Gäbe es nämlich n verschiedene Werte, so würde man $\Omega(n^2)$ Schritte für die Erzeugung benötigen.

3.1.2 Der LSEARCH-Algorithmus

Wir haben bisher gesehen, wie der STREAM-Algorithmus prinzipiell funktioniert. Nun soll der LSEARCH-Algorithmus, welcher vom STREAM-Algorithmus als Unterprogramm aufgerufen wird, genauer betrachtet werden. Der LSEARCH-Algorithmus verwendet das Prinzip der lokalen Suche. Dabei wird eine Startlösung gewählt und diese dann fortschreitend verbessert, bis eine ausreichende Annäherung an die Optimallösung erreicht wurde.

Der LSEARCH-Algorithmus macht sich die Tatsache zunutze, dass die Anzahl der zu erstellenden Cluster k in den Zwischenschritten des Algorithmus nicht konstant gehalten werden muss. Vielmehr arbeitet der Algorithmus mit mehr als k Clustern und verringert die Anzahl der Cluster nach und nach auf *genau* k. Im Gegensatz dazu ist beim k-Means-Algorithmus die Anzahl der zu erstellenden Cluster k konstant. Guha et al. [GMM+03] merken an, dass eine Reduzierung auf weniger als k Zentren zu vermeiden ist, da die beste Lösung für $k-1$ Zentren viel teurer sein kann, als die beste Lösung für k Zentren. Der LSEARCH-Algorithmus arbeitet folgendermaßen: Zunächst wird eine triviale obere Schranke z_{max} bestimmt. Diese Schranke gibt die Kosten an, die im schlechtesten Fall für die Erstellung von neuen Cluster-Zentren anfallen. Die mittleren Kosten seien gegeben durch: $z = \frac{z_{min}+z_{max}}{2}$. Nun wird mit Hilfe des *InitialSolution*-Algorithmus eine gute Startlösung für den LSEARCH-Algorithmus bestimmt. Dadurch wird die Anzahl der Iterationen

[2] q sei hier die Anzahl der unterschiedlichen Werte

im LSEARCH-Algorithmus von $\Theta(\log n)$ auf $O(1)$ reduziert. O'Callagahn et al. [OMM+02] haben gezeigt, dass es nicht notwendig ist den gesamten Suchraum nach möglichen Teilmengen von Cluster-Zentren zu durchsuchen, sondern es ausreichend ist in einem beschränkten Suchraum zu suchen, welcher potentiell günstige Cluster-Zentren enthält. Sie haben sogar gezeigt, dass es genügt $\Theta(\frac{1}{p}\log k)$ beliebige Elemente als Cluster-Zentren zu wählen, um eine hinreichend optimale Lösung zu erhalten. Danach wird ein binäres Suchverfahren für $z_{min} \leq z \leq z_{max}$ angewendet, das genau dann stoppt, wenn genau k Cluster-Zentren bestimmt wurden. Innerhalb dieses Suchverfahrens wird aus Effizienzgründen der *FL*-Algorithmus verwendet, der folgende Optimierung vornimmt: Mit $gain(x)$ für $x \in N$ bezeichne man die Kosten, die man spart, wenn man einen neuen Cluster mit dem Zentrum x erstellt. Die Kosten berücksichtigen insbesondere alle Neuzuweisungen von Elementen anderer Cluster und das Schließen von obsoleten Clustern. Dabei sind folgende Beschränkungen zu beachten:

1. Elemente anderer Cluster können nur x zugewiesen werden.
2. Ein Cluster kann dann und nur dann geschlossen werden, wenn seine Elemente zuvor x zugewiesen wurden.

Jede Anwendung der $gain()$-Operation bringt uns einer bestmöglichen Lösung näher. Der Gesamtgewinn nach Durchführung von $gain()$-Operationen sollte deshalb konstant abnehmen. Betrachtet man den Fall, dass für den aktuellen Wert z die Kosten sich nur geringfügig verändern, aber die aktuelle Lösung weit weniger als k Zentren aufweist, so kann man das Verfahren getrost abbrechen, da man davon ausgehen kann, dass für diesen Wert von z die bestmögliche Lösung mit k Zentren nicht erreicht werden kann. Auf eine Verbesserung der Lösung kann ebenfalls verzichtet werden, wenn wir bereits k Zentren haben und sich nach einer erneuten Durchführung der $gain()$-Operation nur eine geringfügige Änderung der Kosten abzeichnet.

3.1.3 Leistungsvergleich und Komplexitätsbetrachtungen

Guha et al. [GMM+03] haben im Rahmen ihrer empirischen Untersuchungen die Überlegenheit des STREAM-Algorithmus gegenüber dem k-Means Algorithmus gezeigt. Abbildung 7 zeigt die signifikanten Qualitätsunterschiede beim Clustering einer Menge von 2-dimensionalen Datensätzen durch den k-Means-Algorithmus bzw. durch den LSEARCH-Algorithmus. Für den Leistungsver-

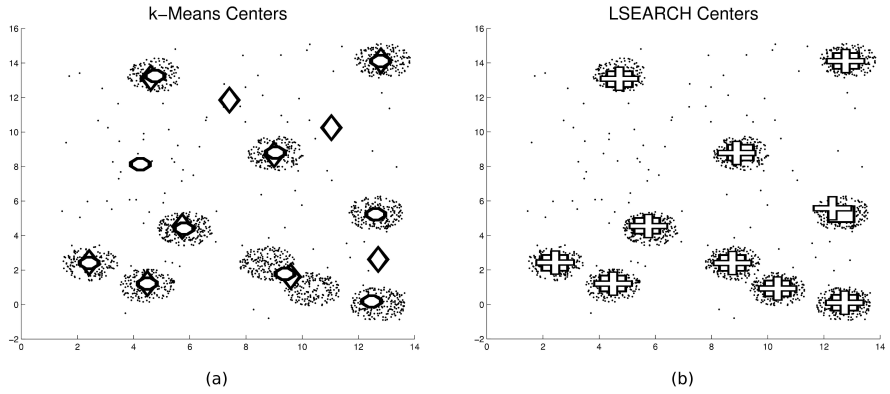

Abbildung 7: Beste und schlechteste k-Means- und LSEARCH-Ergebnisse für 2D-Datensätze [GMM+03]

gleich wurden die Testdaten einerseits so aufbereitet, dass ihre Elemente besonders gut den Clustern zugeordnet werden können und andererseits so angeordnet, dass eine Zuordnung zu den einzelnen Clustern besonders schwierig wird. Es zeigte sich, dass der k-Means-Algorithmus besonders empfindlich auf asymmetrische Daten reagiert. Abbildung 7 (a) zeigt die billigsten – durch Achtecke dargestellt – und die teuersten Lösungen – durch Rauten dargestellt – die der k-Means-Algorithmus für das Clustering-Problem gefunden hat. Im Gegensatz dazu zeigt sich bei der Verwendung des LSEARCH-Algorithmus, dass dieser nahezu worstcase-optimal ist. Man sieht dies deutlich in Abbildung 7 (b), wo die schlechtesten Lösungen mit Rechtecken und die besten Lösungen mit „+“-Zeichen markiert sind. Die Unterlegenheit des k-Means-Algorithmus ist keine

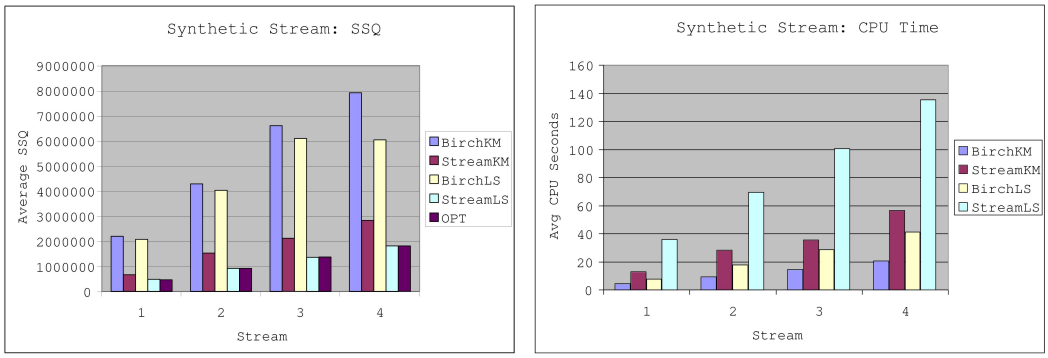

Abbildung 8: Messergebnisse für den synthetischen Datenstrom [GMM+03]

große Überraschung, da es bereits einige Algorithmen gibt, die diesem überlegen sind.

Weit wichtiger für die Anwendung auf Datenströmen ist der Vergleich mit dem BIRCH-Algorithmus. Der STREAM-Algorithmus benutzt den LSEARCH-Algorithmus als Unterprogramm. Guha et al. [GMM+03] haben bei ihren Untersuchungen den LSEARCH-Algorithmus durch den BIRCH- bzw. den k-Means-Algorithmus ersetzt. Der k-Means-Algorithmus wurde dahingehend abgeändert, dass er nun auch mit gewichteten Daten umgehen kann.

Zuerst wurde ein synthetischer Datenstrom mit einer Länge von ungefähr 16 MB und rund 50.000 Punkten in einem 40-dimensionalen euklidischen Raum untersucht. Die Ergebnisse sind in Abbildung 8 veranschaulicht. Der SSQ-Wert[3] des STREAM-Algorithmus war um 2–3 mal kleiner als der Wert, der mit dem BIRCH-Algorithmus unter Verwendung desselben Clustering-Algorithmus erzielt wurde. Der STREAM-Algorithmus war 2–3 mal langsamer als der BIRCH-Algorithmus. Der STREAM-Algorithmus in Verbindung mit dem LSEARCH-Algorithmus erzeugte fast optimale Lösungen.

Für den zweiten Leistungsvergleich wählten Guha et al. [GMM+03] Intrusion-Detection-Datensätze des KDD-Cup 1999[4]. Auch hier zeigte sich ein ähnliches Bild. Der STREAM-Algorithmus ist wesentlich schneller, hat eine höhere Laufzeit und liefert bessere Ergebnisse als der BIRCH-Algorithmus. Beim Clustering von Web-Suchanfragen ist Geschwindigkeit der wichtigste Faktor. Für dieses Anwendungsgebiet bietet sich die Verwendung des BIRCH-Algorithmus an. Für Anwendungsgebiete bei denen Fehler äußerst kostspielig und sicherheitskritisch sein können, ist die Verwendung des STREAM-Algorithmus zu empfehlen.

Der LSEARCH-Algorithmus hat eine Laufzeit von $O(nm + nk \log k)$. Die Variable m ist hierbei die Anzahl der durch den InitialSolution-Algorithmus erstellten Cluster-Zentren. Obwohl m von

[3] Der SSQ-Wert (engl. *sum of squares*) ist die Summe der Abstände der Punkte eines Clusters zu seinem Zentrum
[4] `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

N abhängig ist, führt dieser Algorithmus zu einer bedeutenden Verbesserung gegenüber anderen Algorithmen, da m normalerweise relativ klein ist.

3.2 Klassifikation

Moderne Verfahren zum Data Mining auf Datenströmen sind einerseits durch die Laufzeit und andererseits durch den Speicherbedarf beschränkt. Das Hauptproblem stellt heute aber die Fülle an vorhandenen Trainingsdaten dar. Die Verfügbarkeit von Trainingsdaten steigt mit der Zeit unbegrenzt an. Da die meisten Data-Mining-Algorithmen damit nicht umgehen können und eine Vielzahl der Trainingsbeispiele verwerfen, führt dies oft zu einem suboptimalen Aufbau des Entscheidungsbaums aufgrund fehlender Informationen (engl. *underfitting*).

Ein anderes Problem stellt das neue Verarbeitungsparadigma der Datenströme dar. Die Klassifikation beschränkt sich nicht mehr auf eine endliche Menge von gespeicherten Datensätzen, bei denen das mehrfache Lesen möglich ist. Stattdessen müssen die Daten in Realzeit verarbeitet werden und können nur in geringem Umfang gespeichert werden.

3.2.1 Der VFDT-Algorithmus

Im Jahre 2000 wurde mit dem VFDT-Algorithmus *(**V**ery **F**ast **D**ecision **T**ree learner)* von Domingos und Hulten [DH00] eines der ersten Verfahren zur Entscheidungsbaum-Induktion auf Datenströmen vorgestellt. Algorithmen wie ID3 oder der zum späteren Leistungsvergleich herangezogene C4.5-Algorithmus von Quinlan [Qui93] – eine Weiterentwicklung des ID3-Algorithmus, der auch mit nicht-nummerischen und unvollständigen Daten umgehen kann – haben den entscheidenden Nachteil, dass die Anzahl der Trainingsdaten, die als Entscheidungsgrundlage dienen, durch den Hauptspeicher begrenzt werden. Aus diesem Grund sind diese Verfahren nicht für die Klassifikation von potentiell unendlichen Datenströmen geeignet. Zur Lösung dieses Problems benutzen Domingos und Hulten [DH00] sog. *Hoeffding-Bäume*, die im schlechtesten Fall in einer Zeit die proportional zur Anzahl der Attribute ist, gelernt werden können.

Um das entsprechende Attribut zu finden, das am jeweiligen inneren Knoten des Entscheidungsbaums zur Entscheidung herangezogen wird, genügen nach Domingos und Hulten [DH00] einige wenige Trainingsdaten aus, wenn man unterstellt, dass die Trainingsdaten in zufälliger Reihenfolge das Verarbeitungssystem erreichen und die Wahrscheinlichkeitsverteilung sich nicht ändert. Betrachtet man einen Datenstrom von Trainingsdaten, so wählt man die ersten Datensätze als Test des Wurzelknotens aus. Nachdem der Wurzelknoten bestimmten wurde, wird dieses Verfahren mit den nachfolgenden Trainingsdaten rekursiv auf die Kinder des Wurzelknotens angewendet.

Die Frage wieviele Testdaten für den jeweiligen Knoten benötigt werden, kann man mit Hilfe der sog. *Hoeffding-Schranke* [Hoe63, MM94] beantworten. Man betrachte eine reelle Zufallsvariable r mit einer oberen Schranke R (in unserem Fall gilt: $R = 1$). Führt man nun n unabhängige Zufallsexperimente durch und berechnet man deren Mittelwert $\overline{r}$, dann folgt mit der Hoeffding-Schranke, dass mit einer Wahrscheinlichkeit von $1 - \delta$ der echte Mittelwert der Zufallsvariablen mindestens $\overline{r} - \epsilon$ ist, mit

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}. \tag{5}$$

Die Hoeffding-Schranke ist unabhängig von der Wahrscheinlichkeitsverteilung, die durch die Beobachtungen erzeugt wurde. Sei $G(X_i)$ das heuristische Maß (z.B. der Informationsgewinn), das zur Auswahl des Testattributs dient. Ziel ist es nun zu garantieren, dass das Attribut, welches unter

Verwendung von n Trainingsdaten als Testattribut ausgewählt wurde, mit hoher Wahrscheinlichkeit dasselbe Attribut ist, welches unter Verwendung einer unendlichen Anzahl von Trainingsdaten ausgewählt werden würde. Man nehme an, dass G maximiert werden soll. Sei nun X_a das Attribut, welches nachdem n Trainingsdatensätze verarbeitet wurden, den höchsten Wert $\overline{G}$ hat und sei X_b das Attribut mit dem zweit-höchsten Wert. Dann definieren wir die Differenz zwischen den beiden heuristischen Werten durch: $\Delta\overline{G} = \overline{G}(X_a) - \overline{G}(X_b) \geq 0$. Für ein gegebenes δ garantiert die Hoeffding-Schranke, dass X_a das beste Testattribut für den Knoten mit einer Wahrscheinlichkeit von $1 - \delta$ ist, nachdem n Trainingsdatensätze verarbeitet wurden und $\Delta\overline{G} > \epsilon$ gilt. Das bedeutet, dass wenn der beobachtete Wert $\Delta\overline{G} > \epsilon$ ist, dann garantiert die Hoeffding-Schranke, dass $\Delta G \geq \Delta\overline{G} - \epsilon > 0$ gilt mit einer Wahrscheinlichkeit von $1 - \delta$.

Domingos und Hulten [DH00] haben gezeigt, dass der VFDT-Algorithmus in der Lage ist unter realistischen Annahmen zu garantieren, dass der von ihm erzeugte Entscheidungsbaum sich asymptotisch nicht sonderlich von einem durch einen stapelverarbeitenden Algorithmus (z.B. den ID3-Algorithmus) erzeugten Entscheidungsbaum unterscheidet.

Im Anhang auf Seite 29 ist der VFDT-Algorithmus in der Fassung von Hulten et al. [HSD01] wiedergegeben. Sie haben einige Verbesserungen gegenüber der Fassung von Domingos und Hulten [DH00] eingebracht. Ein Schwellwert im VFDT-Algorithmus verhindert, dass der Vergleich von Attributen zu viel Zeit in Anspruch nimmt, bei denen die Differenz verschwindend gering ist. Dazu wird X_a genau dann als Splitting-Attribut gewählt, wenn $\Delta\overline{G} < \epsilon < \tau$ gilt (τ ist hier ein benutzerdefinierter Schwellwert). Das sog. *„pre-pruning“* – das Beschneiden des Entscheidungsbaums – führt zu einer erheblichen Effizienzsteigerung. Dazu wird für jeden Knoten ein „null“ Attribut $X_\emptyset$ geführt. Ein Splitting wird genau dann durchgeführt, wenn mit einer Konfidenzwahrscheinlichkeit von $1 - \delta$ das bestmögliche Splitting bzgl. G günstiger ist, als kein Splitting durchzuführen.

Die ständige Neuberechnung der Funktion G kostet sehr viel Zeit. Da es nicht sehr effizient ist, für jeden Knoten zu berechnen, ob bei ihm ein Splitting vorgenommen werden soll, kann mit der Variablen n_{min} festgelegt werden, wieviele Attribute sich in einem Knoten befinden müssen, bevor G erneut berechnet wird.

Solange der VFDT-Algorithmus Daten schneller verarbeiten kann, als diese ankommen, stellt der beschränkte Arbeitsspeicher das wesentliche Hindernis beim Lernen von komplexen Modellen dar. Ist nicht mehr genug Arbeitsspeicher vorhanden, so kann der VFDT-Algorithmus bestimmte Blätter deaktivieren und somit neuen Platz schaffen. Der Speicherbedarf wird zusätzlich dadurch verringert, dass Daten, die schon vor langer Zeit verarbeitet wurden und für die zukünftige Entscheidung nicht mehr relevant sind, entfernt werden.

Der VFDT-Algorithmus hat einen Speicherbedarf von $O(ldvc)$, wobei die Variablen folgende Bedeutung haben: d bezeichnet die Anzahl der Attribute, v ist die maximale Anzahl von Werten pro Attribut, c steht für die Anzahl der Klassen und l ist die Anzahl der Blätter des Baums.

3.2.2 Leistungsvergleich

Domingos und Hulten [DH00] haben ihren VFDT-Algorithmus mit dem C4.5-Algorithmus von Quinlan [Qui93] – einer Weiterentwicklung des Verfahrens von Seite 9 – verglichen. Der erste Testlauf wurde auf einem synthetischen Datenstrom durchgeführt. Abbildung 9 (a) zeigt die durchschnittliche Genauigkeit der beiden Verfahren. Der C4.5-Algorithmus ist bis ca. 25.000 Beispiele genauer als der VFDT-Algorithmus. Diese anfängliche Überlegenheit ist darin begründet, dass der C4.5-Algorithmus Beispiele auf verschiedenen Ebenen des Entscheidungsbaums mehrfach verwendet. Der VFDT-Algorithmus benutzt hingegen ein Beispiel nur ein einziges Mal. Im Bereich von 25.000 bis 100.000 Beispiele sind die beiden Verfahren ungefähr gleich gut. Bei mehr als 100.000

Beispielen kann der C4.5-Algorithmus nicht mehr angewendet werden. Der VFDT-Algorithmus kann mehr als 100.000 Beispiele ohne Probleme verarbeiten. In Abbildung 9 (b) zeigt sich ein ähnliches Bild. Die Anzahl der Knoten des Entscheidungsbaums, welcher vom C4.5-Algorithmus erstellt wurde, steigt exponentiell an. Im Vergleich dazu steigt die Anzahl der Knoten beim VFDT-Algorithmus sehr langsam an. Insgesamt erreicht der VFDT-Algorithmus mit weniger Knoten eine höhere Genauigkeit als der C4.5-Algorithmus. Es zeigte sich beim Training mit 160 Millionen Da-

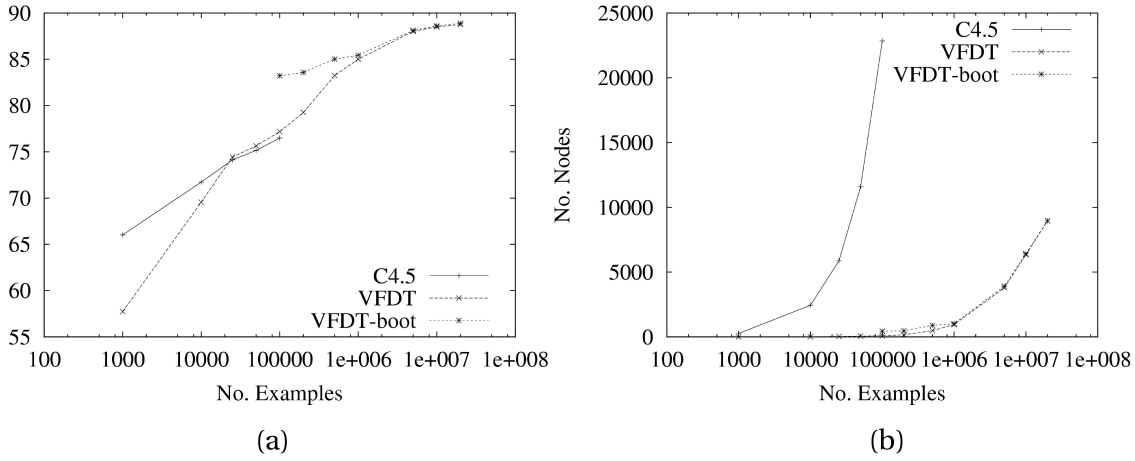

Abbildung 9: Leistungsvergleich zwischen dem C4.5- und VFDT-Algorithmus [DH00]

tensätzen, dass die Genauigkeit des VFDT-Algorithmus ab ca. 10 Millionen Beispielen gegen 90% konvergiert. Für die Verarbeitung der Datensätze benötigte der VFDT-Algorithmus auf einem handelsüblichen PC ca. 160 Minuten.

Für den zweiten Testlauf wählten Domingos und Hulten [DH00] einen Datenstrom mit Webpage-Requests ihrer Universität aus. Im Laufe einer Woche ergab sich nach ihren Angaben ein Datenstrom mit 82,8 Millionen Datensätzen. Zu Spitzenzeiten erreichten das Verarbeitungssystem 17.400 Datensätze pro Minute. Für ihre Untersuchungen benutzten sie die Datensätze eines Tages (ca. 276.230 Beispiele). Für die Wahl des Testattributs des Wurzelknotens benötigte der VFDT-Algorithmus rund 21 Minuten und erreichte eine Genauigkeit von 64,2%. Für die vollständige Verarbeitung der 276.230 Beispiele wurden rund 24 Minuten benötigt; davon waren rund 16 Minuten für das Einlesen der Datensätze vom Externspeicher notwendig. Der C4.5-Algorithmus wurde auf 75.000 Datensätze angewendet. Er konnte nicht auf die gesamten Trainingsdaten angewendet werden, da sein verfügbarer Speicher auf 40 MB beschränkt wurde (dem VFDT-Algorithmus standen ebenfalls nur 40 MB zur Verfügung). Für die Verarbeitung dieser Teilmenge der Trainingsdaten benötigte der C4.5-Algorithmus rund 50 Minuten und erreichte eine Genauigkeit von 73,3%. Insgesamt zeigte sich aber, dass der VFDT-Algorithmus in der Lage war 1,61 Million Beispiele zu lernen. In derselben Zeiten gelang es dem C4.5-Algorithmus nur 75.000 Beispiele zu lernen.

3.3 Assoziationsregeln

Im ersten Teil dieser Ausarbeitung haben wir den Apriori-Algorithmus als Verfahren zum Aufspüren von Assoziationsregeln kennengelernt. Dieser Algorithmus führt Join-Operationen zur Vereinigung der Itemsets durch. Da dieser Operator blockierend ist, d.h. die Verarbeitung anderer Itemsets wird so lange verzögert, bis die Join-Operation alle eingetroffenen und zukünftig eintreffenden Datensätze verarbeitet hat, kann der Apriori-Algorithmus nicht auf Datenströme angewen-

det werden. Giannella et al. [GHP+04] haben ein Verfahren – *FP-Stream* – entwickelt, welches das Mining nach häufig auftretenden Mustern bei unterschiedlichen Zeit-Granularitäten ermöglicht.

Wie können wir häufig vorkommende Muster in einem potentiell unendlichen Datenstrom bestimmen? Bei den klassischen Verfahren wie dem Apriori-Algorithmus wurden selten vorkommende Muster ignoriert oder verworfen. Für die klassischen Verfahren war dieses Vorgehen effizient, im Datenstrom-Paradigma würde man Informationen verlieren, wenn man selten vorkommende Muster ignorieren würde. Muster die im Moment selten auftreten, können wenige Augenblicke später häufig vorkommen und umgekehrt. Deshalb lassen sich Muster nach Giannella et al. [GHP+04] in 3 Klassen unterteilen: häufig vorkommende Muster (engl. *frequent patterns*), gelegentlich vorkommende Muster (engl. *subfrequent patterns*) und selten vorkommende Muster (engl. *infrequent patterns*).

Definition 3.1 (Auftrittshäufigkeit und Maß an Zustimmung) *Die Auftrittshäufigkeit (engl.* frequency*) eines Itemsets I in einem Zeitraum T ist die Anzahl der Transaktionen in denen I innerhalb T auftritt. Das Maß an Zustimmung (engl.* support*) von I ist die quotiale Verknüpfung der Auftrittshäufigkeit von I und der Anzahl aller Transaktionen innerhalb von T.*

Oft ist es wünschenswert alte und neue Daten in unterschiedlicher Granularität zu speichern. Dabei wird die gerade vergangene Zeit so feingranular wie möglich erfasst; wohingegen alte Daten so grobgranular wie möglich verwaltet werden. Chen et al. [CDH+02] haben gekippte Zeitfenster (engl. *tilted-time window*) eingeführt, die diese unterschiedlichen Granularitäten verwalten können. Abbildung 10 zeigt ein gekipptes Zeitfenster. Die letzte Stunde wird durch vier Zeiteinheiten à einer

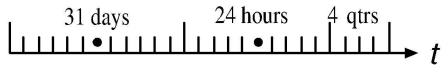

Abbildung 10: Beispiel für ein tilted-time window [GHP+04]

Viertelstunde dargestellt. Die letzten 24 Stunden werden durch 24 Zeiteinheiten repräsentiert. Für die letzten 31 Tage gilt dasselbe. Insgesamt werden nur $4 + 24 + 31 = 59$ Zeiteinheiten benötigt. Abbildung 11 zeigt die Zuordnung von Mengen von häufig vorkommden Mustern zu den gekippten Zeitfenstern. Folgende Fragen sollen beantwortet werden:

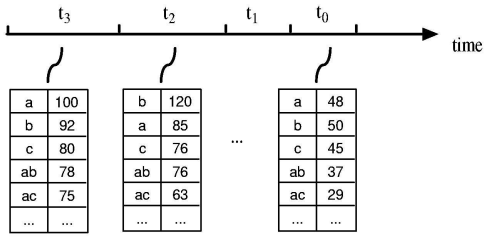

Abbildung 11: Zuordnung von häufig auftretenden Mustern zu den gekippten Zeitfenstern [GHP+04]

1. Wie sieht die Menge der häufig vorkommenden Muster innerhalb des Zeitabschnitts zwischen t_0 und t_2 aus?
2. In welchem Zeitabschnitt kommt das Muster (a, b) häufig vor?
3. Wie verändert sich der support-Wert für das Muster (c, d) zwischen t_2 und t_1?

Für jedes gekippte Zeitfenster können wir für jedes Muster die Auftrittshäufigkeit festhalten. Wir greifen dazu auf eine Datenstruktur von Han et al. [HPY00] zurück – den sog. *Frequent-Pattern-Baum* (kurz FP-Baum).

3.3.1 Der FP-Baum

Der FP-Baum von Han et al. [HPY00] hat sich als Datenstruktur zur Verwaltung von häufig-auftretenden Mustern bewährt. Folgendes Beispiel aus Han et al. [HPY00] dient zur Illustration, wie ein solcher Baum erzeugt werden kann. Gegeben sei folgende Menge von Datensätzen:

TID	gekaufte Produkte	häufig auftretende Muster (geordnet)
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Führt man nun einen Scan dieser Datenbank durch, so kann man den support-Wert (Wert hinter dem „:“) für die einzelnen Items bestimmen: $\langle (f:4), (a:3), (b:3), (m:3), (p:3) \rangle$. Nun können wir den FP-Baum erstellen. Zuerst erstellen wir die Wurzel (engl. *root*) des FP-Baums und markieren sie mit „null“. Danach führen wir erneut einen Scan der Transaktionen durch. Der Scan der ersten Transaktion liefert den ersten Zweig des FP-Baums: $\langle (f:1), (c:1), (a:1), (m:1), (p:1) \rangle$. Der Knoten $(f:1)$ ist mit der Wurzel verbunden, $(c:1)$ mit $(f:1)$ und so weiter. Die zweite

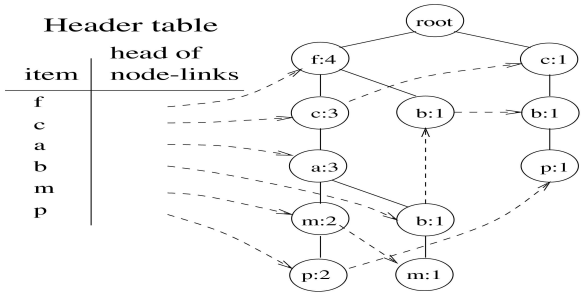

Abbildung 12: vollständiger FP-Baum [HPY00]

Transaktion $\langle f, c, a, b, m \rangle$ hat mit der ersten Transaktion folgenden Präfix gemeinsam: $\langle f, c, a \rangle$. Die support-Werte entlang des Präfixes werden um eins erhöht und der neue Knoten $(b:1)$ wird an den Knoten $(a:2)$ angehängt. Verfährt man mit den verbleibenden Transaktionen genauso, so erhält man den FP-Baum aus Abbildung 12.

3.3.2 FP-Stream

Bei der Konstruktion des FP-Baums haben wir gesehen, dass wir zwei vollständige Scans der Daten benötigen, um den FP-Baum zu erzeugen. Da das Datenstrom-Paradigma aber nur einen einmaligen Scan erlaubt, ist der FP-Baum in dieser Form nicht verwendbar. Wir modifizieren den FP-Baum derart, dass er nicht mehr Muster an Blättern speichert, sondern Teile des Datenstroms. Für gewöhnlich ändern sich häufig auftretende Muster im Zeitablauf eher selten. Deshalb überlappen sich die Baumstrukturen für verschiedene gekippte Zeitfenster oft. Um Speicherplatz einzusparen, werden die gekippten Zeitfenster in die einzelnen Knoten des FP-Baums eingebettet.

Wir verwenden einen einzigen FP-Baum, bei dem an jedem Knoten die Auftrittshäufigkeit für jedes gekippte Zeitfenster in einer Tabelle verwaltet wird. Abbildung 13 zeigt eine solche Tabelle.

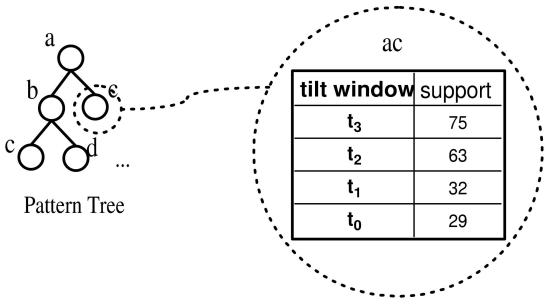

Abbildung 13: Pattern-Baum mit einem gekippten Zeitfenster [HPY00]

3.3.3 Verwaltung von gekippten Zeitfenstern

Die Verwaltung von gekippten Zeitfenstern wie in Abbildung 10 gezeigt ist relativ einfach. Sind die vier Viertelstunden verstrichen, so werden sie zu einer Stunde zusammengefasst. Verwendet man gekippte Zeitfenster mit einer logarithmischen Partitionierung, so kann die Zahl der benötigten Zeitfenster weiter reduziert werden. Abbildung 14 zeigt ein gekipptes Zeitfenster, das logarithmisch

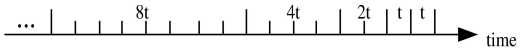

Abbildung 14: gekipptes Zeitfenster mit logarithmischer Partitionierung [HPY00]

partitioniert ist. Der jüngsten Viertelstunde wird eine Zeiteinheit zugewiesen. Der vorletzten Viertelstunde wird auch eine Zeiteinheit zugewiesen. Den vorvorletzten zwei Viertelstunden wird eine Zeiteinheit zugewiesen und so weiter. So können wir ein Jahr mit $\log_2(365\times24\times4)+1 \approx 17$ Zeiteinheiten verwalten, wenn wir als feinstes Granulat eine Viertelstunde wählen. Ohne diese Partitionierung würden wir $365\times24\times4 = 35.040$ Zeiteinheiten benötigen. Etwas formaler ausgedrückt bedeutet das, dass wir den Datenstrom in gleichgroße Stapel (engl. *batches*) $B_1, B_2, \ldots, B_n$ unterteilen, wobei B_1 der älteste und B_n der jüngste Stapel ist. Wir definieren für alle $i \geq j$ $B(i,j) := \bigcup_{k=j}^{i} B_k$. Für ein gegebenes Itemset I sei $f_I(i,j)$ die Auftrittshäufigkeit von I in $B(i,j)$. Wir betrachten nur folgende Auftrittshäufigkeiten: $f(n,n); f(n-1,n-1); f(n-2,n-3); f(n-4,n-7); \ldots$

3.3.4 Aktualisierung von gekippten Zeitfenstern

Für einen neuen Stapel von Datensätzen B kann man das gekippte Zeitfenster für ein Itemset I wie folgt aktualisieren: Zuerst ersetzen wir $f(n,n)$, die Auftrittshäufigkeit in der feingranularsten Zeiteinheit (Ebene 0) mit $f(B)$ und verschieben $f(n,n)$ in die Ebene 1. Dort wird $f(n-1,n-1)$ durch $f(n,n)$ ersetzt. Bevor $f(n-1,n-1)$ in die Ebene 2 verschoben wird, wird geprüft, ob das Pufferfenster (engl. *intermediate buffer window*) leer ist. Für jede Ebene gibt es ein solches Pufferfenster. Ist das Pufferfenster leer, so wird $f(n-1,n-1)$ in dieses verschoben und der Algorithmus stoppt. Ist das Pufferfenster hingegen voll, so werden so lange Verschiebungen vorgenommen, bis das Verfahren stoppt. Giannella et al. [GHP+04] weisen darauf hin, dass ein gekipptes Zeitfenster nicht mehr als $2\lceil\log_2(N)\rceil + 2$ Elemente enthält, wenn N die Anzahl der bisher verarbeiteten Stapel des Datenstroms ist.

3.3.5 Approximation der Auftrittshäufigkeiten

Seien $t_0, \ldots, t_n$ gekippte Zeitfenster, die die Stapel des bisher verarbeiteten Datenstroms gruppieren, wobei t_0 das jüngste und t_n das älteste Zeitfenster ist. Die Größe des Fensters t_i bezeichnen wir mit w_i. Unser Ziel ist das Aufspüren aller häufig vorkommenden Muster, deren support-Wert innerhalb des Zeitraums $T = t_k \cup t_{k+1} \cup \ldots \cup t_{k'}$ (mit $0 \leq k \leq k' \leq n$) größer als σ ist. Die Größe von T sei $W = \sum_{i=k}^{k'} w_i$. Es genügt nicht nur die häufig vorkommenden Muster zu speichern, da ein Muster das momentan selten vorkommt, wenig später häufig vorkommen kann. Da wir dafür aber sehr viel Speicher benötigen würden, werden wir nur einen Teil der Stapel – $f_I(t_0), \ldots, f_I(t_{m-1})$ für ein beliebiges m mit $0 \leq m \leq n$ – aufbewahren. Gilt die Bedingung (6), so können die nachfolgenden Elemente – der sog. *tail* – verworfen werden.

$$\exists l, \forall i \text{ mit } l \leq i \leq n, f_I(t_i) < \sigma w_i \text{ und } \forall\, l' \text{ mit } l \leq m \leq l', \sum_{i=l}^{l'} f_I(t_i) < \epsilon \sum_{i=l}^{l'} w_i. \tag{6}$$

Daraus folgt, dass wir nun keine genaue Auftrittshäufigkeit im Zeitraum T mehr haben, sondern nur noch eine ungefähre Auftrittshäufigkeit (engl. *approximate frequency*)
$\hat{f}_I(T) = \sum_{i=k}^{\min\{m-1,k'\}} f_I(t_i)$, falls $m > k$ und $\hat{f}_I(T) \cong \epsilon W$, falls $m \leq k$. Die Beziehung zwischen der Näherung und der eigentlichen Auftrittshäufigkeit lässt sich dann durch folgende Ungleichung beschreiben:

$$f_I(T) - \epsilon W \leq \hat{f}_I(T) \leq f_I(T) \tag{7}$$

Sei m die kleinste Zahl die die Bedingung (6) erfüllt. Giannella et al. [GHP+04] haben gezeigt, dass wenn das gekippte Zeitfenster eines Itemsets I beschnitten wird, so dass $t_0, \ldots, t_{m-1}$ übrig bleiben, so können wir trotzdem alle häufig vorkommenden Muster im Zeitraum T finden, wenn man den Fehler ϵ berücksichtigt. Diese Art der Beschneidung des Baums nennen wir *tail pruning*.

Die Datenstruktur FP-Stream verwaltet die gekippten Zeitfenster. Sobald ein neuer Stapel B das Verarbeitungssystem erreicht, werden die Itemsets untersucht und der FP-Baum aktualisiert. Kommt ein Itemset I aus B nicht im FP-Stream vor, dann wird es ihm hinzugefügt, wenn $f_I(B) \geq \epsilon|B|$ gilt. Wird I gefunden, dann wird $f_I(B)$ zu I hinzugefügt und anschließend das sog. *tail pruning* durchgeführt. Werden alle Fenster verworfen, dann wird I aus FP-Stream entfernt.

Zur weiteren Effizienzsteigerung der Verwaltung der Datenstruktur FP-Stream nutzen wir die Eigenschaft der Anti-Monotonie aus. Giannella et al. [GHP+04] haben folgendes gezeigt: Die Eigenschaft, dass die Auftrittshäufigkeit eines Itemsets größer oder gleich dem support-Wert seiner Obermengen ist, gilt auch für den Fall das gekippte Zeitfenster und eine approximierte Auftrittshäufigkeit verwendet werden. Außerdem sollte die Größe der Tabelle der gekippten Zeitfenster größer oder gleich der Tabelle seiner Obermengen sein. Auf Grundlage dieser Überlegung führen wir einen weiteren Beschneidungstyp (*Type I Pruning*) ein: Kommt I in B vor, aber nicht in FP-Stream, dann ist auch keine Obermenge von I in FP-Stream. Gilt $f_I(B) < \epsilon|B|$, dann muss keine Obermenge von I weiter untersucht werden.

Desweiteren gilt: Wenn alle Einträge der Tabelle der gekippten Zeitfenster von I gelöscht wurden, dann kann jede Obermenge von I verworfen werden (*Type II Pruning*).

Die Suche nach starken Assoziationsregeln, wie sie beim Apriori-Algorithmus beschrieben wurde, kann hier in analoger Weise verwendet werden.

4 Fazit und Ausblick

Wir haben gesehen was man unter Data Mining versteht und wie klassische Probleme wie z.B. die Klassifikation oder die Cluster-Analyse einer endlichen Menge von Datensätzen gelöst werden können. Danach habe ich gezeigt, dass diese klassischen Verfahren nicht ohne Abänderung auf das neue Verarbeitungsparadigma der Datenströmen übertragen werden können. Ich habe dargestellt, welche Anforderungen an einen Data-Mining-Algorithmus, der auf Datenströmen angewendet werden soll, gestellt werden müssen. Danach habe ich verschiedene Verfahren zum Data Mining auf Datenströmen vorgestellt.

Der VFDT-Algorithmus von Domingos und Hulten [DH00] zur Entscheidungsbaum-Induktion kann nur mit nominalen Daten umgehen. Jin und Agrawal [JA03] zeigen, wie auch nummerische Daten verarbeitet werden können. Außerdem zeigen sie, dass man ebenfalls gute Ergebnisse mit einer kleineren Stichprobe erzielen kann. Aggarwal et al. [AHWY04] zeigen wie Datenströme bedarfsgetrieben (engl. *on demand*) verarbeitet werden können. Dieses Verfahren ermöglicht die simultane Verarbeitung von Trainings- und Testdaten.

An der Universität von Illinois wird zur Zeit von Cai et al. [CCP+04] ein System entwickelt, das verschiedene Verfahren zum Data Mining auf Datenströmen vereint (Cluster-Analyse, Klassifikation, Aufspüren von Assoziationsregeln und Mustern sowie Visualisierung). Das System trägt den Namen *MAIDS* (***M**ining **A**larming **I**ncidents from **D**ata **S**treams*)[5].

Insgesamt lässt sich konstatieren, dass das Forschungsgebiet der Datenströme hohe Relevanz für die Lösung praktischer Probleme hat. Die Nachfrage nach effizienten Data-Mining-Algorithmen zur Beherrschung des stark zunehmenden Datenaufkommens im Web und anderen Anwendungsgebieten wird in absehbarer Zeit nicht abflachen, da Data-Mining-Systeme als Entscheidungsunterstützungssysteme in den Unternehmen zunehmend an Bedeutung gewinnen.

[5] Auf folgender Website findet der geneigte Leser nähere Informationen zu MAIDS: `http://maids.ncsa.uiuc.edu/`

A Algorithmen

A.1 Der Apriori-Algorithmus

Algorithmus 1 : *Apriori-Algorithmus* ([HK01] S. 235)

Input : Database, D, of transactions; minimum support threshold, *min_sup*.
Output : L, frequent itemsets in D.

```
1  L_1 = find_frequent_1-itemsets(D);
2  for k = 2; L_{k-1} ≠ ∅; k++ do
3      C_k = apriori_gen (L_{k-1}, min_sup);
       /* scan D for counts                                          */
4      foreach transaction t ∈ D do
           /* get the subsets of t that are candidates               */
5          C_t = subset(C_k, t);
6          foreach candidate c ∈ C_t do
7              c.count++;
8          end
9      end
10     L_k = {c ∈ C_k | c.count ≥ min_sup};
11 end
12 return L = ∪_k L_k;
```

Prozedur `apriori_gen`(L_{k-1}: frequent$(k-1)$-itemsets; *min_sup*: minimum support threshold)

```
1  foreach itemset l_1 ∈ L_{k-1} do
2      foreach itemset l_2 ∈ L_{k-1} do
3          if (l_1[1] = l_2[1]) ∧ (l_1[2] = l_2[2]) ∧ ... ∧ (l_1[k-2] = l_2[k-2]) ∧ (l_1[k-1] < l_2[k-1])
           then
               /* join step: generate candidates                     */
4              c = l_1 ⋈ l_2;
5              if has_infrequent_subset (c, L_{k-1}) then
                   /* prune step: remove unfruitful candidate        */
6                  delete c;
7              else
8                  add c to C_k;
9              end
10         end
11     end
12 end
13 return C_k;
```

Prozedur `has_infrequent_subset`(c : candidate$k-$itemset; L_{k-1}:frequent$(k-1)$-itemsets)

```
  /* use prior knowledge                                                    */
1 foreach (k − 1)-subset s of c do
2     if s ∉ L_{k−1} then
3         return TRUE;
4     end
5 end
6 return FALSE;
```

A.2 Der k-Means-Algorithmus

Algorithmus 4 : *k-Means.* The k-Means-Algorithmus for partitioning based on the mean of the objects in the cluster. ([HK01] S. 349)

Input : The number of clusters k and a database containing n objects.
Output : A set of k clusters that minimize the squared-error criterion.

```
1 arbitrarily choose k objects as the initial cluster centers;
2 repeat
3     (re)assign each object to the cluster to which the object is the most similar, based on
      the mean value of the objects in the cluster;
4     update the cluster means, i.e., calculate the mean value of the objects for each cluster;
5 until no change;
```

A.3 Der ID3-Algorithmus

Algorithmus 5 : *Generate_decision_tree* ([HK01] S. 285)

Input : The training samples, *samples*, represented by discrete-valued attributes; the set of candidate attributes, *attribute-list*.
Output : A decision tree.

create a node N;
if samples *are all of the same class,* C **then**
    return N *as a leaf node labeled with the class* C;
end
if attribute-list *is empty* **then**
    `/* majority voting */`
    return N *as a leaf node labeled with the most common class in* samples;
end
select *test-attribute*, the attribute among *attribute-list* with the highest i information gain;
label node N with *test-attribute*;
`/* partition the samples */`
foreach *known value* $a_i \in$ test-attribute **do**
    grow a branch from node N for the condition *test-attribute*$= a_i$;
    `/* a partition */`
    let s_i be the set of samples in *samples* for which *test-attribute*$=a_i$;
    if s_i *is empty* **then**
        attach a leaf labeled with the most common class in *samples*;
    end
    else attach the node returned by *Generate_decision_tree*(s_i, *attribute-list-test-attribute*);
end

A.4 Der STREAM-Algorithmus

Algorithmus 6 : STREAM

foreach *chunk* X_i *in the stream* **do**
    if *a sample of size* $\geq \frac{1}{\epsilon} \log \frac{k}{\delta}$ *contains fewer than* k *distinct points* **then**
        $X_i \leftarrow$ weighted representation;
    end
    Cluster X_i using `LSEARCH`;
    $X' \leftarrow ik$ centers obtained from chunks 1 through i iterations of the stream, where each center c obtained by clustering X_i is weighted by the number of points in X_i assigned to c;
    Output the k centers obtained by clustering X' using `LSEARCH`;
end

Algorithmus 7 : InitialSolution(N, z)

Input : a data set N, the facility cost z for opening a new facility
Output : an initial solution

Reorder data points randomly;
Create a cluster center at the first point;
forall *points after the first* **do**
    Let d be the distance from the current point to the nearest existing cluster center;
    With probability d/z create a new cluster center at the current point; otherwise add the current point to the best current cluster
end

Algorithmus 8 : FL(N, $d(\cdot,\cdot)$, z, ϵ, (I,a))

Input : $\epsilon \in \mathbb{R}$ controls how soon the algorithm stops to improve its solution; N is a data set of size n; the metric $d(\cdot,\cdot)$; the facility cost z; an initial solution (I,a) where $I \subseteq N$ is a set of facilities and $a : N \to I$ is an assignment function.

Begin with (I,a) as the current solution;
Let C be the cost of the current solution on N. Consider the feasible centers in random order, and for each feasible center y, if $gain(y) > 0$, perform all advantagegous closures and reassignments (as per *gain* description), to obtain a new solution (I',a');
Let C' be the cost of the new solution;
if $C' \leq (1-\epsilon)\cdot C$ **then return** *to 2*

Algorithmus 9 : LSEARCH(N, $d(\cdot,\cdot)$, k, ϵ, ϵ', ϵ'')

Input : N is a data set of size n; the metric $d(\cdot,\cdot)$; k the number of medians to create; $\epsilon, \epsilon', \epsilon'' \in \mathbb{R}$
Output : a solution (F',g')

$z_{min} = 0$;
`/* for` x_0 `an arbitrary point in` N `*/`
$z_{max} = \sum_{x \in N} d(x, x_0)$;
$z = \frac{z_{max}+z_{min}}{2}$;
(I,a) =`InitialSolution(`N`, `z`);`
Randomly pick $\Theta(\frac{1}{p}\log k)$ points as feasible medians;
while *# medians* $\neq k \wedge z_{min} < (1-\epsilon'')\cdot z_{max}$ **do**
    Let (F,g) be the current solution;
    Run `FL(`N`, `d`,`z`, `ϵ`, `(F,g)`)` to obtain a new solution (F',g');
    if $k \leq |F'| \leq 2k$ **then** then exit loop;
    if $|F'| > 2k$ **then**
        $z_{min} = z$;
        $z = \frac{z_{max}+z_{min}}{2}$;
    end
    if $|F'| < k$ **then**
        $z_{max} = z$;
        $z = \frac{z_{max}+z_{min}}{2}$;
    end
end
return *our solution* (F',g');

A.5 Der VFDT-Algorithmus

Algorithmus 10 : VFDT(S, X, G, δ, τ)

Input : S is a stream of examples, X is a set of symbolic attributes, $G(\cdot)$ is a split evaluation function, δ is one minus the desired probability of choosing the correct attribute at any given node, τ is a user-supplied tie threshold, n_{min} is the # of examples between checks for growth.

Output : HT a decision tree

```
1  Let HT be a tree with a single leaf l_1 (the root);
2  Let X_1 = X ∪ {X_∅};
3  Let G_1(X_∅) be the G obtained by predicting the most frequent class in S;
4  foreach class y_k do
5      foreach value x_ij of each attribute X_i ∈ X do Let n_ijk(l_1) = 0;
6      foreach example (x, y) in S do
7          Sort (x, y) into a leaf l using HT;
8          foreach x_ij in x such that X_i ∈ X_l do Increment n_ijy(l);
9          Label l with the majority class among the examples seen so far at l;
10         Let n_l be the number of examples seen at l;
11         if the examples seen so far at l are not of the same class and n_l mod n_min is 0 then
12             Compute G_l(X_i) for each attribute X_i ∈ X_l − {X_∅} using the counts n_ijk(l);
13             Let X_a be the attribute with the highest G_l;
14             Let X_b be the attribute with the second-highest G_l;
15             Compute ε using Equation (5);
16             Let ΔG_l = G_l(X_a) − G_l(X_b);
17             if ((ΔG_l > ε) ∨ (ΔG_l ≤ ε < τ)) ∧ X_a ≠ X_∅ then
18                 Replace l by an internal node that splits on X_a;
19                 foreach branch of the split do
20                     Add a new leaf l_m, and let X_m = X − {X_a};
21                     Let G_m(X_∅) be the G obtained by predicting the most frequent class at
                       l_m;
22                     foreach class y_k and each value x_ij of each attribute X_i ∈ X_m − {X_∅} do
                       Let n_ijk(l_m) = 0;
23                 end
24             end
25         end
26     end
27 end
28 return HT;
```

Literaturverzeichnis

[AHWY04] Aggarwal, Charu C. ; Han, Jiawei ; Wang, Jianyong ; Yu, Philip S.: On Demand Classification of Data Streams. In: *KDD '04, Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–888–1, S. 503–508

[AS94] Agrawal, Rakesh ; Srikant, Ramakrishnan: Fast Algorithms for Mining Association Rules in Large Databases. In: *VLDB 1994, Proceedings of the 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Morgan Kaufmann Publishers, 1994. – ISBN 1–55860–153–8, S. 487–499

[AZ96] Adriaans, Pieter ; Zantinge, Dolf: *Data Mining*. Reading, Massachusetts : Addison-Wesley, 1996. – 176 S. – ISBN 0–201–40380–3

[BBD+02] Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Motwani, Rajeev ; Widom, Jennifer: Models and Issues in Data Stream Systems. In: *PODS '02: Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–507–6, S. 1–16

[CCC+02] Carney, Donald ; Cetintemel, Uğur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Seidman, Greg ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stanley B.: Monitoring Streams - A New Class of Data Management Applications. In: *VLDB 2002, Proceedings of the 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, Morgan Kaufmann Publishers, 2002. – ISBN 1–55860–869–9, S. 215–226

[CCP+04] Cai, Y. D. ; Clutter, David ; Pape, Greg ; Han, Jiawei ; Welge, Michael ; Auvil, Loretta: MAIDS: Mining Alarming Incidents from Data Streams. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–859–8, S. 919–920

[CDH+02] Chen, Yixin ; Dong, Guozhu ; Han, Jiawei ; Wah, Benjamin W. ; Wang, Jianyong: Multi-Dimensional Regression Analysis of Time-Series Data Streams. In: *VLDB 2002, Proceedings of the 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, Morgan Kaufmann Publishers, 2002. – ISBN 1–55860–869–9, S. 323–334

[DH00] Domingos, Pedro ; Hulten, Geoff: Mining High-Speed Data Streams. In: *KDD '00, Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 20-23, 2000, Boston, MA, USA*. New York, NY, USA : ACM Press, 2000. – ISBN 1–58113–233–6, S. 71–80

[DH01] Domingos, Pedro ; Hulten, Geoff: Catching Up with the Data: Research Issues in Mining Data Streams. In: *Workshop on Research Issues in Data Mining and Knowledge Discovery DMKD 2001, Santa Barbara, CA – May 20th 2001* (2001). `http://www.cs.washington.edu/homes/ghulten/papers/hsds-dmkd01.pdf`

[EN04] Elmasri, Ramez ; Navathe, Shamkant B.: *Fundamentals of Database Systems*. Fourth Edition. Boston, San Francisco [u.a.] : Addison-Wesley, 2004. – 1030 S. – ISBN 0–321–12226–7

[GHP+04] Giannella, Chris ; Han, Jiawei ; Pei, Jian ; Yan, Xifeng ; Yu, Philip S.: 6 Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In: Kargupta, Hillol (Hrsg.) ; Joshi, Anupam (Hrsg.) ; Sivakumar, Krishnamoorthy (Hrsg.) ; Yesha, Yelena (Hrsg.): *Data Mining: Next Generation Challenges and Future Directions*. MIT/AAAI Press, 2004, S. 105–124. – ISBN 0–262–61203–8

[GMM+03] Guha, Sudipto ; Meyerson, Adam ; Mishra, Nina ; Motwani, Rajeev ; O'Callaghan, Liadan: Clustering Data Streams: Theory and Practice. In: *IEEE Transactions on Knowledge and Data Engineering* 15 (2003), Nr. 3, S. 515–528

[HK01] Han, Jiawei ; Kamber, Micheline: *Data Mining: Concepts and Techniques*. San Francisco, San Diego [u.a.] : Morgan Kaufmann Publishers, 2001 (The Morgan Kaufmann Series in Data Management Systems). – 550 S. – ISBN 1–55860–489–8

[Hoe63] Hoeffding, Wassily: Probability inequalities for sums of bounded random variables. In: *Journal of the American Statistical Association* 58 (1963), S. 13–30

[HPY00] Han, Jiawei ; Pei, Jian ; Yin, Yiwen: Mining Frequent Patterns without Candidate Generation. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. New York, NY, USA : ACM Press, 2000. – ISBN 1–58113–217–4, S. 1–12

[HSD01] Hulten, Geoff ; Spencer, Laurie ; Domingos, Pedro: Mining Time-Changing Data Streams. In: *KDD '01, Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 26-29, 2001, San Francisco, CA, USA*. New York, NY, USA : ACM Press, 2001. – ISBN 1–58113–391–X, S. 97–106

[JA03] Jin, Rouming ; Agrawal, Gagan: Efficient Decision Tree Construction on Streaming Data. In: *KDD '03, Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–737–0, S. 571–576

[Kuh93] Kuhn, Thomas S.: *Die Struktur wissenschaftlicher Revolutionen*. 2. rev. Aufl. [Nachdr.]. Frankfurt am Main : Suhrkamp Taschenbuch, 1993 (Suhrkamp Taschenbuch Wissenschaft 25). – 238 S. – Titel der Originalausgabe: The Structure of Scientific Revolutions. – ISBN 3–518–27625–5

[Mac67] MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: *5th Berkeley Symp. Math. Statist. Prob.* 1 (1967), S. 281–297

[Mit97] Mitchell, Tom M.: *Machine Learning*. International Edition. New York, St. Louis [u.a.] : McGraw-Hill, 1997 (McGraw-Hill Series in Computer Science). – 414 S. – ISBN 0–07–115467–1

[MM94] Maron, Oded ; Moore, Andrew: Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. In: *Advances in Neural Information Processing Systems* Bd. 6. San Mateo, CA : Morgan Kaufmann Publishers, April 1994. – ISBN 1–55860–360–3, S. 59–66

[OMM+02] O'Callaghan, Liadan ; Meyerson, Adam ; Motwani, Rajeev ; Mishra, Nina ; Guha, Sudipto: Streaming-Data Algorithms for High-Quality Clustering. In: *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA, USA*, IEEE Computer Society, 2002. – ISBN 0–7695–1531–2, S. 685–696

[Qui86] Quinlan, J. R.: Induction of Decision Trees. In: *Machine Learning* 1 (1986), S. 81–106

[Qui93] Quinlan, J. R.: *C4.5: Programs for Machine Learning*. San Francisco, CA, USA : Morgan Kaufmann, 1993 (Morgan Kaufmann Series in Machine Learning). – 302 S. – ISBN 1–55860–238–0

[RN03] Russel, Stuart J. ; Norvig, Peter: *Artificial Intelligence –A Modern Approach*. Second Edition (International Edition). London, UK : Prentice Hall International, 2003 (Prentice Hall Series in Artificial Intelligence). – 1081 S. – ISBN 0–13–080302–2

[ZRL96] Zhang, Tian ; Ramakrishnan, Raghu ; Livny, Miron: BIRCH: An Efficient Data Clustering Method for Very Large Databases. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. New York, NY, USA : ACM Press, 1996, S. 103–114