technische universität
dortmund

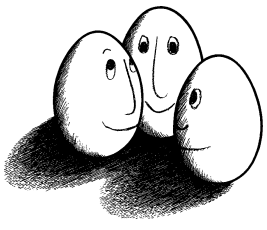
Bachelorarbeit

Aggregation häufiger Mengen in Datenströmen

Adrian Skirzynski
Juni 2013

Gutachter:
Prof. Dr. Katharina Morik
Dipl.-Inf. Christian
Bockermann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS8)
http://www-ai.cs.tu-dortmund.de



INHALTSVERZEICHNIS

1

EINLEITUNG

Die Aggregation von häufigen Mengen innerhalb von Datenströmen ist in der heutigen Zeit, aufgrund der immer weiter steigenden Nutzungsmöglichkeiten, ein breit untersuchtes Thema. Um zu erklären, was häufige Mengen sind und warum die Analyse dieser einen immer größer werdenden Stellenwert bekommt, kann man ein bekanntes Beispiel, die sogenannte Warenkorbanalyse, betrachten. Hierfür nehme man einen normalen Supermarkt. Die einzelnen Produkte, die in diesem Supermarkt verkauft werden, nennen wir Items. Einen Einkauf eines Kunden nennen wir eine Transaktion. Wenn ein Kunde also Milch, Chips und Käse kauft, ist Milch das erste Item, Chips das zweite und der Käse das dritte Item. Betrachten wir diese Produkte nun als ganzes, also in der Form eines Einkaufs {Milch, Chips, Käse}, bilden diese drei Produkte eine Transaktion. Weiter ist diese Transaktion, bestehend aus diesen drei Produkten, auch eine Teilmenge der gesamten Produktpalette des Supermarktes. Nehmen wir jetzt einmal an, dass bei 1000 Transaktionen immer die Items Milch, Chips und Käse vorkommen. Dann kann man sagen, dass diese Teilmenge der gesamten Produktpalette häufig vorkommt. Dabei ist dem Betrachter selbst überlassen, wann er eine Menge als häufig ansieht. Für einige wäre die Teilmenge {Milch, Chips, Käse} vielleicht schon häufig, wenn sie innerhalb von 1000 Transaktionen 50 mal vorkommt, für andere erst bei 500 mal. Häufige Mengen sind interessant für Analysen von Datenmengen, um Zusammenhänge und Korrelationen zu finden. Aufgrund dieser können Entscheidungen gefällt werden.

Ein Teil der Analyse von häufigen Mengen und darauf basierenden Entscheidungen ist die Aggregation. Aggregation kann hierbei zweierlei bedeuten. Zum einen versteht man darunter die Herstellung einer Verbindung, zum Beispiel Assoziationsregeln, zwischen

Daten. Wenn sich bei 1000 Transaktionen herausstellt, dass 500 mal Milch und Käse zusammen gekauft wurden, könnte man die Assoziationsregel Milch $\rightarrow$ Käse herleiten. Der Besitzer eines Supermarktes könnte nämlich daran interessiert sein, welche Gegenstände in Verbindung mit anderen gekauft werden. In diesem Fall könnte sich der Geschäftsleiter dazu entscheiden, beide Produkte nebeneinander aufzustellen.

Zum anderen versteht man bei Aggregation auch die Erstellung von Metadaten. Metadaten sind strukturierte Daten, die Informationen über andere Daten enthalten. Diese sind dann nicht nur zum Beispiel maschinell les- und verwendbar, sondern können auch so aufbereitet werden, dass sie von Menschen gelesen werden können. Als Beispiel eignet sich das Projekt *ViSTA-TV*, an dem der Lehstuhl 8 der Fakultät für Informatik an der technischen Universität Dortmund mitarbeitet. Hierbei handelt es sich um ein Projekt, das anonymisierte Benutzerdaten von einer Online-Plattform für Fernsehen und Rundfunk sammelt und analysiert. Aus diesen Daten kann man Schlüsse über das Fernsehverhalten ziehen. Diese Daten werden zunächst in einer Logdatei per Text gespeichert. Diese Logdatei ist jedoch sehr groß und auf den ersten Blick für einen menschlichen Betrachter unbrauchbar. Die Aufgabe besteht nun darin, diese Daten zu verarbeiten und zu analysieren. Die daraus gezogenen Informationen kann man dann für weitere Zwecke verwenden, wie zum Beipiel Marktanalysen. Um die Ergebnisse dann auch für einen menschlichen Betrachter zugänglich zu machen, kann man aus den extrahierten Daten ein Schaubild zur Verdeutlichung erstellen.

Die eintreffenden Transaktionen eines Supermarktes, sowie die gesammelten Daten von *ViSTA-TV*, müssen natürlich für die Analyse gespeichert werden. Zunächst gehen wir von einer Speicherung in einer statischen Datenbank aus. Statische Datenbanken werden in bestimmten Abständen erweitert um weitere Daten. Auf die Datenbanken können dann Algorithmen angewendet werden, um häufige Mengen zu ermitteln. Wenn man nun von einer Supermarktkette anstelle eines einzelnen Marktes ausgeht, kann man sich vorstellen, dass sehr viele Transaktionen eintreffen. Schnell wird klar, dass der Speicherverbrauch immens zunimmt mit einer wachsenden Zahl von Transaktionen. Immer öfter werden Daten nicht über einen längeren Zeitraum in einer Datenbank gespeichert, sondern werden innerhalb eines Datenstroms verarbeitet. Im Gegensatz zu statischen Datenbanken ergeben sich bei der Verarbeitung eines Datenstroms zusätzliche Probleme. Zum Beispiel müssen Transaktionen in einem Strom schnell verarbeitet werden, da sie nur eine bestimmte Zeit lang gespeichert werden. Verlässt eine Transaktion den Datenstrom ohne verarbeitet worden zu sein, geht die in ihr enthaltene Information verloren. Bei der Betrachtung von Datenströmen, können zudem auch zeitliche Zusammenhänge erkannt werden. Beispielsweise lässt sich während einer Fussball-Weltmeisterschaft sicherlich feststellen, dass der Verkauf von Bier und Chips enorm zunimmt. Die Produkt-

positionierung kann somit auch saisonell erfolgen.

Die vorliegende Bachelorarbeit soll einen Überblick über die Aggregation von häufigen Mengen geben. Hierfür werden wichtige unt interessante Forschungsergebnisse zu diesem Thema vom Anfang der Forschung bis heute zusammengefasst. Die wichtigsten Algorithmen werden zunächst für statische Datenbanken erläutert und untersucht. Daraufhin wird in Bezug auf Datenströme ein Überblick darüber gegeben, was bei der Behandlung von diesen zu beachten ist. Es werden Algorithmen aufgeführt, die versuchen, die zusätzlichen Anforderungen im Gegensatz zu statischen Datenbanken effizient zu behandeln. Schließlich wird der Algorithmus StreamKrimp untersucht. StreamKrimp ist ein Algorithmus, der viele Anforderungen in Bezug auf Datenströme erfüllt. Er wurde für diese Arbeit in einer abgeänderten Form in Java Implementiert, um zu prüfen, ob dadurch die Laufzeit verbessert werden kann ohne dabei an Qualität bei den Ergebnissen zu verlieren. Basierend auf den Ergebnissen könnten weiterführende Überlegungen zum Einsatz auf Datensätze von *ViSTA-TV* getätigt werden.

Struktur

In Kapitel 2 werden zunächst die Grundlagen, die zum Verständnis von der Aggregation von häufigen Mengen notwendig sind, erläutert. Daraufhin folgt eine Übersicht zu Methoden für statische Datenbanken. Die zwei wichtigsten Algorithmen für statische Datenbanken werden im dritten Kapitel vorgestellt. Obwohl sich diese Bachelorarbeit mit Streams beschäftigt, ist es wichtig, diese Algorithmen auch zu kennen. Sie sind nicht nur der Grundstein, der für die Aggregation von häufigen Mengen gelegt wurde, sondern es wird auch schnell ersichtlich, dass ihre Ideen und Datenstrukturen bei den vorgestellten Streamingalgorithmen eingesetzt werden. Kapitel 4 beschäftigt sich mit Datenströmen und welche weiteren Anforderungen im Gegensatz zu statischen Datenbanken entstehen. Es werden Modelle zur Behandlung von Datenströmen vorgestellt. Algorithmen, die verschiedenste Techniken benutzen, werden dafür aufgeführt. Der Streamingalgorithmus StreamKrimp wird in Kapitel 5 untersucht. Anschließend wird der veränderte Algorithmus verwendet, um Datensätze zu untersuchen und zu vergleichen. Kapitel 6 illustriert die Ergebnisse, die mit dem Algorithmus bei Anwendung auf die Datensätze erzielt wurden. Abgeschlossen wird die Bachelorarbeit in Kapitel 7 mit einem Fazit.

2

Grundlagen

2.1 Statische Datenbanken

Als das Studium des Themengebietes der häufigen Mengen begann, wandte man die Algorithmen zunächst an statischen Datenbanken an. Erst später wurden die Forschungen auf Datenströme ausgeweitet. Es ist allerdings wichtig, die Prinzipien der Aggregation von häufigen Mengen in statischen Datenbanken zu verstehen und nicht direkt bei Datenströmen zu beginnen. Zum einen liegt es daran, dass viele Streamingalgorithmen sich aus Algorithmen für statische Datenbanken ableiten und diese als Vorbild haben. Zum anderen können Definitionen für statische Datenbanken übernommen werden, wenn Datenströme betrachtet werden.

In diesem Kapitel werden zunächst grundlegende Begriffe definiert und wichtige Algorithmen vorgestellt, um häufige Mengen in statischen Datenbanken zu finden. Basierend darauf, werden im nächsten Kapitel Algorithmen für Datenströme betrachtet.

2.1.1 Grundlagen

Im Folgenden wird immer davon ausgegangen, sofern nicht anders erwähnt, dass eine Datenbank (im weiteren Verlauf mit $\mathcal{DB}$ bezeichnet) aus einer endlichen Menge von Transaktionen besteht. Eine Transaktion besteht aus einem oder mehreren Items. Wie in der Einleitung erwähnt, sind Items zum Beispiel Produkte in einem Supermarkt wie Milch. Eine Transaktion (ein Einkauf eines Kunden) ist dann eine Teilmenge von allen

TID	Transaktion
1	{Milch, Käse}
2	{Milch, Käse, Cola}
3	{Bier, Käse}
4	{Bier, Chips}
5	{Cola, Käse}
⋮	⋮

Tabelle 2.1: Beispiel für eine Transaktionsdatenbank eines Supermarktes

Produkten. Die Menge aller Items bezeichnen wir mit $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$, wobei n $\in \mathbb{N}$ und jedes i stellt ein einziges Item dar.
Eine Transaktion besteht dann aus einer Menge von Items aus $\mathcal{I}$.

Definition 2.1.1 (Transaktionen). $\mathcal{T} = \{t_1, \ldots, t_n\}$ ist eine Menge von Transaktionen, wobei n $\in \mathbb{N}$ und $t_i \subseteq \mathcal{I}$.

In Tabelle 2.1 ist ein kleines Beispiel einer Datenbank für einen Supermarkt dargestellt. Jede Zeile aus $\mathcal{DB}$ repräsentiert eine Transaktion mit ihrer Transaktions-id.
Um überhaupt mit häufigen Mengen arbeiten zu können, muss definiert werden, wann eine Menge als häufig angesehen wird. Dafür benötigen wir zunächst die Definition des *Supports*.

Definition 2.1.2 (Support einer Itemmenge). Eine Menge o $\subseteq \mathcal{I}$ wird von einer Transaktion t_i unterstützt, wenn o $\subseteq t_i$. Der *Support* s der Menge o ist dann

$$s(o) = |\{t \in T : o \subseteq t\}|.$$

Der *Support* ist also die Anzahl der Transaktionen aus $\mathcal{DB}$, die o enthalten. Damit eine Menge o als häufig angesehen wird, benötigen wir einen Wert, der festlegt, wie oft eine Menge mindestens in $\mathcal{DB}$ vorkommen muss, damit sie für uns relevant wird. Diesen Wert nennen wir *minSupport*.

Definition 2.1.3 (minSupport). Betrachtet man einen Schwellenwert $\delta \in \mathbb{R}_{\geq}0$, gilt eine Menge o als häufig, wenn $s(o) \geq \delta$.

Dieser Wert ist absolut. In der Praxis wird meist ein relativer Schwellenwert benutzt, also $\delta_{rel} \in [0, 1]$. Damit der Wert relativ wird, teilen wir den absoluten Wert durch die Größe der Datenbank

$$\frac{s(o)}{|\mathcal{DB}|} \geq \delta_{rel}$$

Da aber meist der relative Wert gemeint ist, wird immer δ anstelle von δ_{rel} geschrieben. Die zu lösende Aufgabe ist dann leicht zu formulieren:

Problem 1 (Häufige Mengen finden). Gegeben sei eine Datenbank $\mathcal{DB}$ und ein *minSupport* δ. Finde alle Itemmengen o mit $s(o) \geq \delta$.

2.1.2 Häufige Mengen und ihre Teilmengen

Es stellt sich nun die Frage, wie man die häufigen Mengen aus einer Datenbank herausfiltern kann. Ein naives Vorgehen könnte so aussehen: Bilde alle Teilmengen o aus $\mathcal{I}$ und prüfe für jede Transaktion t_i, ob sie o enthält. Dieses Verfahren stößt aber schnell an Grenzen. Es müssen nämlich $2^{|\mathcal{I}|}$ Teilmengen gebildet werden, die dann jeweils geprüft werden. Nehme man mal an, ein Supermarkt hätte 100 Produkte, müssten man schon 2^{100} Teilmengen prüfen. Selbst für so eine geringe Zahl an Produkten ist die zu prüfende Menge schon gewaltig, abgesehen davon, dass viele Märkte tausende von Produkten anbieten und die Zahl 100 somit nicht realistisch ist.
Es sind also Algorithmen erforderlich, die den Suchraum stark einschränken, um nicht alle Teilmengen zu prüfen. Aber nicht nur die Wahl eines intelligenten Algorithmus hilft bei der Einschränkung des Suchraums. Man kann auch die Auswahl an zu suchenden häufigen Mengen einschränken. Bis jetzt wurde davon ausgegangen, dass alle häufigen Mengen gefunden werden sollen. Es gibt aber auch Teilmengen von häufigen Mengen, die den Suchraum weiter einschränken.

Maximal häufige Mengen Maximal häufige Mengen sind Mengen, von denen keine Obermenge auch häufig ist. Man spart sich also die Betrachtung von allen Untermengen. Die Anzahl dieser ist also viel kleiner als die aller häufigen Mengen. Allerdings gehen bei maximal häufigen Mengen auch Informationen verloren. Man kann zwar die häufigen Untermengen ableiten, jedoch hat man keine Information darüber, wie hoch der *Support* für diese Untermengen ist. Die fett markierten Mengen in Abbildung 2.1 zeigen die maximal häufigen Mengen bei $\delta = 3$.

Geschlossene häufige Mengen Geschlossene häufige Mengen sind Mengen, von denen keine Obermenge den selben *Support* hat. Auch hier ist die Anzahl geringer als die aller häufigen Mengen. Im Gegensatz zu den maximal häufigen Mengen jedoch kann man den *Support* von den häufigen Untermengen ableiten. Eine häufige Menge muss nämlich eine Untermenge einer geschlossenen Menge sein. Somit entspricht der *Support* dem der gleichen Teilmenge in einer der geschlossenen Mengen. Wenn also zum Beispiel $\{a, b, c\}$ als geschlossene Menge ausgegeben wird, mit einem *Support* von fünf, aber nicht

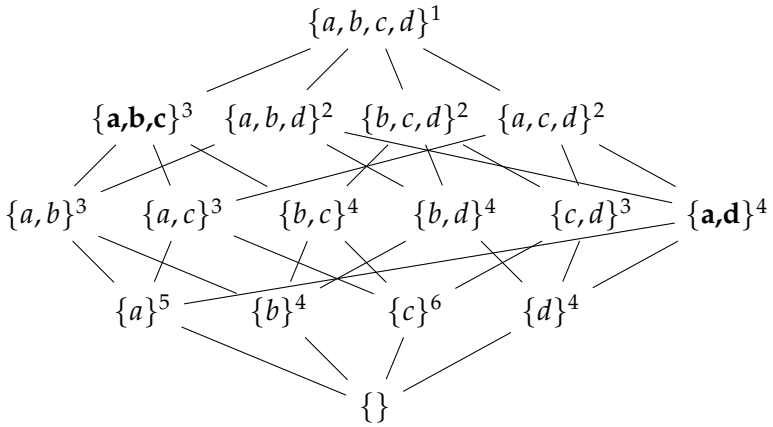


Abbildung 2.1: Beispiel für maximal häufige Mengen (die Häufigkeit ist neben der Menge angegeben)

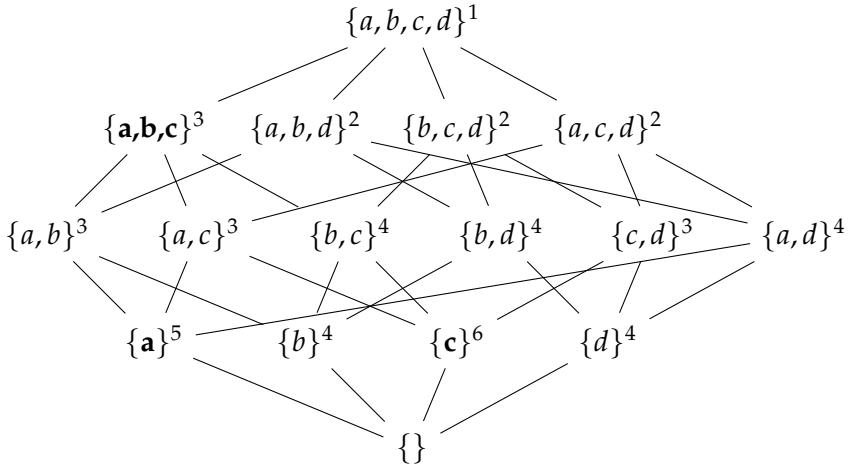


Abbildung 2.2: Beispiel für geschlossene häufige Mengen (die Häufigkeit ist neben der Menge angegeben)

die Menge $\{a, b\}$, weiß man, dass die Menge $\{a, b\}$ ebenfalls einen *Support* von fünf haben muss. In Abbildung 2.2 sind die geschlossenen häufigen Mengen mit $\delta = 3$ fett markiert.

Die verschiedenen häufigen Mengen haben den folgenden Zusammenhang:

Maximal häufige Mengen $\subseteq$ Geschlossene häufige Mengen $\subseteq$ Häufige Mengen

3

Algorithmen für statische Datenbanken

3.1 Häufige Mengen in statischen Datenbanken

In diesem Kapitel werden zwei Algorithmen vorgestellt, die häufige Mengen innerhalb von Datenbanken finden und ausgeben. Bei beiden Algorithmen werden die exakten häufigen Mengen gefunden. Sie sind die wichtigsten und bekanntesten Algorithmen, weshalb auch viele Folgearbeiten auf ihnen basieren. Gerade die Datenstruktur des FPTree aus FPGrowth ist sehr beliebt, da sie sehr kompakt ist.

3.1.1 Apriori

Mit Apriori [4] wurde 1993 der erste Algorithmus vorgestellt, um alle häufigen Mengen zu finden. Er nutzt eine Eigenschaft der Korrelation zwischen der Häufigkeit einer Menge und ihren Teilmengen aus: wenn eine Menge häufig ist, dann müssen auch ihre Teilmengen häufig sein (Monotonie-Eigenschaft bei häufigen Mengen). Aufgrund dieser Eigenschaft müssen nicht alle Teilmengen der Items in der Datenbank erzeugt werden. Falls ein Item oder eine Itemmenge nicht häufig ist, können ihre Obermengen auch nicht häufig sein. Abbildung 3.1 zeigt diesen Sachverhalt.

Der Algorithmus geht dafür wie folgt vor: Zunächst wird über die Datenbank gescannt und die Menge der 1-elementigen häufigen Mengen H_1 ermittelt. Alle Mengen, deren Häufigkeit $< \delta$ ist, werden wieder entfernt. Aus den 1-elementigen Mengen H_1 wer-

den nun die Kandidaten für die 2-elementigen häufigen Mengen K_1 gebildet. Für die Menge K_1 wird die Datenbank wieder gescannt und die Häufigkeiten ermittelt. Alle Mengen, die den *minSupport* erfüllen, kommen in die Menge H_2, die zum Schluss aus den 2-elementigen häufigen Mengen besteht. Aus diesen wird die Menge K_2 aller 3-elementigen Kandidaten gebildet und wieder über die Datenbank gescannt. Dieser Vorgang wiederholt sich, bis keine Kandidaten mehr gefunden oder gebildet werden können. Obwohl der Algorithmus den Suchraum enorm einschränkt, ist leicht zu sehen, dass er auch zwei entscheidene Nachteile hat.

Mehrfacher Datenbankdurchlauf Zum einen muss unter Umständen sehr oft über die Datenbank gescannt werden. Die Anzahl der Scans wird lediglich durch die Länge der längsten Transaktionen $|t_{lang}|$beschränkt. Angenommen t_{lang} besteht aus 40 Items. Im schlimmsten Fall muss dann 40 mal über die Datenbank gescannt werden, da Kandidaten der länge 40 generiert werden müssen. Dann muss für alle Kandidaten der Länge k $\leq$ 40 ein Datenbankscan vorgenommen werden. Gerade bei großen Datenbanken mit sehr vielen Transaktionen dauert dieser Vorgang lange.

Kandidatengenerierung Zum anderen ist die Kandidatengenerierung an sich gerade bei vielen Items und langen Transaktionen sehr aufwändig, da viele Teilmengen gebildet werden müssen. Im Falle von vielen häufigen Mengen (beispielsweise bei einem kleinen *minSupport*) werden viele Kandidaten erzeugt, die überprüft werden müssen. Wenn zum Beispiel im ersten Lauf 10^4 1-elementige häufige Mengen gefunden wurden, werden 10^7 2-elementige Kandidaten generiert, deren Häufigkeit

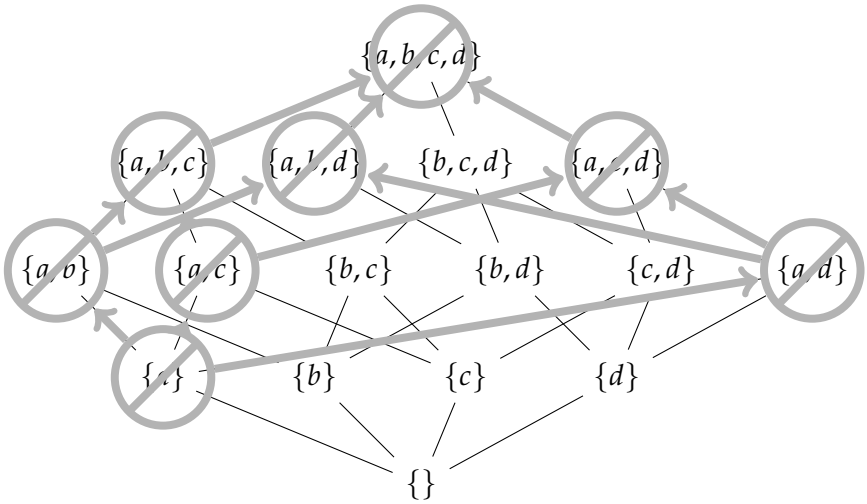


Abbildung 3.1: Da $\{a_1\}$ nicht häufig ist, wird für alle Obermengen von $\{a_1\}$ ausgeschlossen, dass sie häufig sind und somit der Suchraum stark eingeschränkt (Abbildung aus [16].

TID	Items in Transaktion	Sortierte häufige Items in Transaktion
1	$\{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
2	$\{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
3	$\{b, f, h, j, o\}$	$\{f, b\}$
4	$\{b, c, k, s, p\}$	$\{c, b, p\}$
5	$\{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$

Tabelle 3.1: Es wird eine Liste erstellt mit nach Häufigkeit sortierten Transaktionen. Dafür werden zwei Datenbankdurchläufe benötigt. Für dieses Beispiel ist $\delta = 2$ (Beispiel aus [16].

bestimmt werden muss. Auch das Finden von großen häufigen Mengen ist sehr aufwändig. Für eine 100-elementige häufige Menge müssen $2^{100} - 2 \approx 10^{30}$ Kandidaten generiert werden.

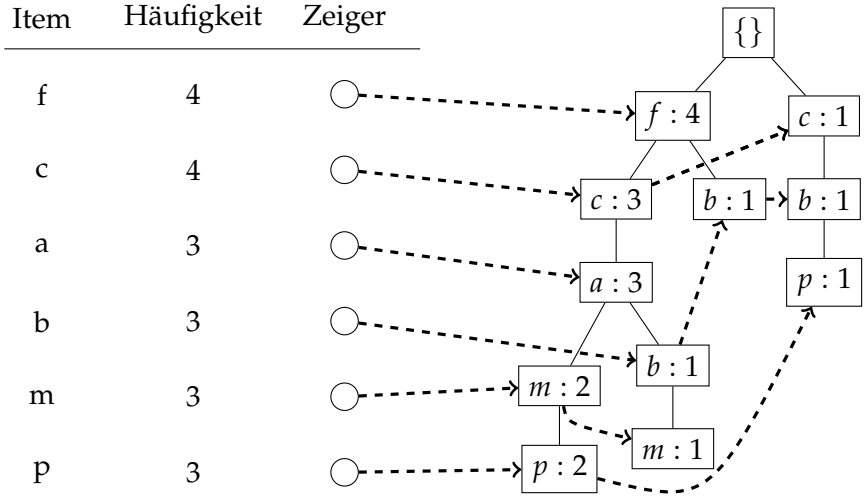
3.1.2 FPGrowth

Ein wichtiger und erfolgreicher Folgealgorithmen nach APRIORI wurde 2000 vorgestellt. Der größte Vorteil von FPGROWTH [3] ist, dass der Algorithmus ohne Kandidatengenerierung auskommt. Zudem ist die benutzte Datenstruktur, der FPTREE, sehr kompakt. Der Algorithmus benötigt lediglich zwei Scans über die Datenbank, um alle häufigen Mengen zu finden. Im ersten Scan werden wie bei dem ersten Schritt von APRIORI alle häufigen 1-elementigen Mengen gesucht. Im zweiten Durchlauf werden die Transaktionen als geordnete Transaktionen dargestellt, in denen die Items je nach Häufigkeit geordnet sind. Ein Beispiel hierfür ist in Tabelle 3.1 gegeben. Anhand dieser geordneten Transaktionen wird die FP-Struktur erstellt. Diese Struktur besteht aus einer Zeiger-Tabelle und dem FPTREE mit einer leeren Wurzel. Die Knoten in einem FPTREE speichern ein Item, die Häufigkeit und möglicherweise einen Zeiger zu einem weiteren Knoten. In der Zeiger-Tabelle wird das Item, die Häufigkeit dieses Items und ein Zeiger auf den ersten Knoten, der dieses Item beinhaltet, gespeichert (Abbildung 3.2a).

Erstellung des FP-Trees

Nachdem die geordneten Transaktionen erstellt wurden, wird bei der ersten Transaktion begonnen. Für jedes Item in der Transaktion, angefangen beim häufigsten, gibt es nun mehrere Möglichkeiten.

1. Fall : Das Item ist in einem Kindknoten der betrachteten Wurzel enthalten. Die Häufigkeit im Knoten wird um 1 erhöht. Ebenso wird der Zähler in der Zeiger-Tabelle für das Item um 1 erhöht. Die neue betrachtete Wurzel ist nun dieser Knoten. Mache weiter mit



(a) Zeiger-Tabelle **(b)** FP-Tree

Abbildung 3.2: Erstellung eines FPTREE: Für jedes häufige Item gibt es einen Eintrag in der Zeiger-Tabelle (sortiert nach der Häufigkeit) und einen Zeiger, der auf das erste Vorkommen des Items im FPTREE zeigt. Jeder Knoten im Baum enthält ein Item, die Häufigkeit des Items und einen Zeiger, der auf die nächste Position des selben Items im Baum zeigt (Abbildung aus [16]).

dem nächsten Item, falls vorhanden.

2. Fall (a) : Das Item ist nicht in einem Kindknoten der betrachteten Wurzel enthalten, aber in der Zeiger-Tabelle. Erstelle für die Wurzel einen neuen Kindknoten mit dem Item und der Häufigkeit 1. Erhöhe ebenso den Zähler des Items in der Zeiger-Tabelle. Folge dem Zeiger für das Item angefangen bei der Zeiger-Tabelle bis zum letzten Knoten. Füge einen Zeiger von diesem letzten Knoten zu dem neu erstellten Knoten hinzu.

2. Fall (b) : Das Item ist nicht in einem Kindknoten der betrachteten Wurzel enthalten und auch nicht in der Zeiger-Tabelle. Erstelle für die Wurzel einen neuen Kindknoten mit dem Item und der Häufigkeit 1. Füge das Item ebenso in die Zeiger-Tabelle hinzu mit einem Zähler bei 1 initialisiert. Füge zum Schluss noch einen Zeiger auf den neu erzeugten Knoten hinzu von der Zeiger-Tabelle aus. Mache weiter mit dem nächsten Item, falls vorhanden.

Sollte die Transaktion kein Item mehr enthalten, wird die nächste Transaktion genommen. Dieser Vorgang wird solange wiederholt, bis jede Transaktion abgearbeitet wurde. Der so entstandene FPTREE für das Beispiel ist in Abbildung 3.2b zu sehen.

$\{f, c, a\}$
$\{f, c, a\}$
$\{f, c, a, b\}$

(a) *Conditional pattern-base* bezüglich m

(b) *Conditional* FPTREE bezüglich m

Abbildung 3.3: Aus Abbildung 3.2 erstellte *condtional pattern-base* und *conditional* FPTREE bezüglich m. 3.3a bildet eine neue Datenbank aus der 3.3b erstellt wird. Dabei wird b nicht hinzugefügt, da die Häufigkeit $< \delta$ ist (Beispiel aus [16].

Ermittlung der häufigen Mengen

Mit dem so entstandenen FPTREE lassen sich die häufigen Mengen bestimmen. Dafür werden die Zeiger der Items benutzt. Für jedes Item in der Zeiger-Tabelle gehen wir an den Zeigern entlang. Für jeden Knoten, der so erreicht wird, wird eine Liste der Elternknoten des Items erstellt. Für jedes Item entstehen so eine oder mehrere Listen, auch *conditional pattern base* genannt (Abbildung 3.3a). Diese kann als eigene Datenbank aufgefasst werden. Wie im Schritt zur Erstellung des FPTREE können diese Listen beziehend auf ein Item genutzt werden, um einen weiteren FPTREE zu erzeugen, in diesem Fall *conditional* FPTREE, da er sich auf ein einziges Item bezieht (Abbildung 3.3b). Dieser Vorgang wird solange wiederholt bis kein neuer Baum mehr erstellt werden kann oder der resultierende Baum aus nur einem Pfad besteht. Besteht der Baum aus nur einem Pfad, bilden alle Teilmengen aus den in dem Pfad befindlichen Items häufige Mengen. Kann kein neuer Baum gebildet werden aus der *conditional pattern base*, bilden alle Teilmengen der Items in dieser häufige Mengen.

Es ist ersichtlich, dass FPGROWTH häufige Mengen schneller findet als APRIORI, da keine Kandidaten generiert werden und alle häufigen Mengen mit Hilfe des FPTREE erstellt werden.

4

STREAMS

4.1 Grundlagen

Mit der weiter voranschreitenden Technologie sah man sich sehr schnell mit einem Problem konfrontiert: Daten wurden so schnell und in solchen Mengen gesammelt, dass eine Speicherung in statischen Datenbanken entweder untragbar, oder sogar unmöglich wurde. Die Transaktionen eines einzigen Supermarktes ließen sich noch in einer Datenbank speichern, doch eine Supermarktkette, bei der die einzelnen Märkte vernetzt sind und die Transaktionen zusammengetragen werden, bringen sehr viele Transaktionen zusammen. Würde man diese Transaktionen sammeln, bräuchte man sehr viel Speicherplatz. Zudem muss die Überlegung gemacht werden, wie lange die Daten gespeichert werden sollten. Das hängt natürlich davon ab, wann sie verarbeitet werden und die daraus gezogenen Informationen analysiert. Man könnte natürlich für jeden Markt dieser Kette eine eigene Datenbank anlegen, jedoch müsste man dann bei der Analyse der Daten auf zeitliche Hintergründe achten. Die Transaktionen jeder Datenbank müssten zum Beispiel beim zusammentragen zunächst sortiert werden. Deshalb wurde klar, dass die Idee der Verarbeitung dieser Daten überarbeitet werden muss. Um zusätzlichen Aufwand zu sparen, sollten die Daten sofort zusammengetragen und so schnell wie möglich bearbeitet werden, am besten zur Echtzeit während sie eintreffen. Bearbeitete Daten müssten somit nicht mehr beachtet und vor allem gespeichert werden. Eine Sortierung wäre dann auch nichtmehr notwendig. Dies führte zu Forschungen im Bereich Datenströme.

4.1.1 Datenströme vs statische Transaktionsdatenbanken

Datenströme sind im Gegensatz zu statischen Datenbanken eine geordnete Sequenz von Items oder Transaktionen, die mit einer schnellen Rate eintreffen. Datenströme sind kontinuierlich, unbegrenzt und es könnte vorkommen, dass sich die Verteilung der Items mit der Zeit ändert. Aufgrund dieser Eigenschaften gibt es bei Datenströmen im Gegensatz zu statischen Datenbanken weitere Herausforderungen bei der Bearbeitung.

Definition 4.1.1 (Datenstrom). Ein Datenstrom $\mathcal{S}$ ist eine Sequenz bestehend aus Transaktionen $<t_1, t_2, t_3, \ldots>$, wobei jede Transaktion aus Items aus $\mathcal{I}$ besteht.

Da die Transaktionen mit hoher Geschwindigkeit eintreffen und nicht genug Speicherplatz zur Verfügung steht, um große Teile des Datenstromes im Hauptspeicher zu halten, sollten die eintreffenden Items so schnell wie möglich bearbeitet werden. Deshalb ist es unter anderem nicht möglich oder effizient, mehrmals durch den Datenstrom zu gehen wie bei APRIORI. Dies ist aber nicht das einzige Problem. Die generellen Probleme werden im nächsten Abschnitt etwas näher betrachtet.

4.1.2 Generelle Probleme

Datenverarbeitungsmodell

Wie schon erwähnt, ist es kaum möglich oder effizient, den gesamten Datenstrom zu speichern. Deshalb muss die Überlegung angestellt werden, welche Teile des Datenstromes ausgewählt werden sollten, um die Algorithmen darauf anzuwenden. Betrachtet man die bisher getätigten Forschungen, lassen sich drei Modelle rausfiltern. Prinzipiell werden bestimmte Teilsequenzen eines Datenstromes betrachtet. Diese Sequenzen werden Fenster genannt. Bei den Fenstermodellen werden nur die Items/Transaktionen innerhalb des Fensters betrachtet. Bei den drei Modellen handelt es sich um das sliding-window, damped-window und das landmark-window Modell.

sliding-window Modell Beim sliding-window Modell gibt es ein Fenster einer bestimmten Grösse, das über den Datenstrom geschoben wird. Es ist wichtig, die Informationen in dem aktuellen Fenster zu bearbeiten, da sie, nachdem das Fenster weiter über den Datenstrom geschoben wird, also die ältesten Transaktionen rausfallen, verloren gehen. Dieses Modell kann benutzt werden, wenn nur die aktuellsten Transaktionen interessant sind für eine Anwendung.

damped-window Modell Das damped-window Modell führt eine Gewichtung der Transaktionen ein. Neue Transaktionen bekommen eine höhere Gewichtung als ältere. Dies geschieht dadurch, dass die Gewichtung von Transaktionen mit der Zeit immer kleiner wird. Dieses Modell kann benutzt werden, wenn ältere Transaktionen

einen Einfluss auf neuere Transaktionen haben, aber dieser Einfluss mit der Zeit abnimmt. Die Grösse des Fensters ist adaptiv. Es wird solange vergrössert, bis ein definiertes Zwischenziel oder das Ende des Datenstroms erreicht ist.

landmark-window Modell Das landmark-window Modell ist dem damped-window Modell sehr ähnlich. Der Unterschied ist lediglich, dass keine Gewichtung der Transaktionen stattfindet. Es werden alle Transaktionen von einem bestimmten Zeitpunkt (dem landmark) bis zum aktuellen Zeitpunkt betrachtet. Auch diese Version ist sichtlich adaptiv. Dieses Modell lässt sich in das damped-window Modell umwandeln, indem man eine Gewichtungs- und Zerfallsfunktion einführt.

Alle drei Modelle finden in den aktuellen Forschungen Verwendung. Die Wahl eines Modells hängt von der Anwendung und den Anforderungen ab.

Speicher-Management

Nachdem entschieden wurde, wie und wo die Informationen gesammelt werden, muss eine Datenstruktur erzeugt werden, die es nicht nur erlaubt, die gesammelten Informationen effizient abzuspeichern, sondern auch sie zu aktualisieren und abzurufen. Es ist nicht sehr effizient, einfach alle Mengen mit ihrer jeweiligen Häufigkeit zu speichern. Wie schon beschrieben würde dafür zu viel Speicherplatz benötigt werden. Eine weitere wichtige Frage ist, was für Mengen gespeichert werden sollen. Wie in Kapitel 2 beschrieben gibt es häufige Mengen sowie maximal häufige Mengen und geschlossene häufige Mengen. Zudem kann man Algorithmen auch unterscheiden in exakte und approximierte Algorithmen. Exakte Algorithmen speichern alle häufigen Mengen mit einer Häufigkeit über dem definierten *minSupport*. Approximierte Algorithmen geben keine genauen Resultate wider, sondern Mengen mit einer geschätzten Häufigkeit. Dabei kann, da geschätzte Ergebnisse nicht immer fehlerfrei sind, eine Fehlertoleranz gegeben sein. Diese ist aber kein Muss. Bei approximierten Algorithmen unterscheidet man auch zwischen *false negativ* und *false positiv*. Bei *false negativ* werden tatsächliche häufige Mengen nicht erkannt. Bei *false positiv* werden nicht-häufige Mengen fälschlicherweise als solche erkannt.

Veränderung der Verteilung

Bei Datenströmen ist es nicht auszuschliessen, dass sich die Verteilung der Items mit der Zeit ändert. Als Beispiel lassen sich hier die in der Einleitung erwähnten saisonellen Einkäufe nennen. Solche Veränderungen in der Verteilung sollten bei Bedarf von dem Algorithmus auch erkannt werden. Dabei gibt es zwei Arten von Veränderungen zu unterscheiden.

concept drift Hierbei ändert sich die Verteilung über einen gewissen Zeitraum langsam aber stetig.

concept shift Im Gegensatz dazu gibt es den *concept shift*. Hierbei ändert sich die Verteilung schlagartig.

Zusammenfassung

In 4.1 wurden die grundlegensten Unterschiede von Datenströmen zu statischen Datenbanken erläutert. Aufgrund der hohen Rate, mit der Transaktionen eintreffen können, müssen diese so schnell wie möglich verarbeitet werden. Damit wird sichergestellt, dass der von ihnen belegte Speicher freigegeben wird für neue Transaktionen. Die gewählte Datenstruktur sollte leicht zu verwalten sein und bei Bedarf Änderungen in der Verteilung erkennen lassen. Vorallem bei der Wahl der Datenstruktur sollte vorher festgelegt werden, welches Modell benutzt werden soll, um über den Datenstrom zu gehen.

In diesem Forschungsgebiet wurden nun viele Algorithmen vorgeschlagen, um diese Anforderungen zu erfüllen. Dabei gibt es nicht „den“ Algorithmus, der alles aufeinmal erfüllt. Es gibt eher viele Algorithmen die sich auf einen Teilaspekt spezialisieren, wie das Extrahieren von maximal häufigen Mengen. Im Rest dieses Kapitels werden einige Algorithmen vorgestellt, die auf verschiedene Arten versuchen Teilprobleme zu behandeln.

4.2 Apriori für Streams?

Bei APRIORI für statische Datenbanken sah man sich mit zwei Problemen konfrontiert: einerseits die Kandidatengenerierung und zum anderen die mehreren Datenbankdurchläufe. Diese beiden Vorgänge sind sehr zeitintensiv. In Kapitel 4.1.2 wurde festgestellt, dass die Bearbeitung eines Stroms vorallem schnell ablaufen muss, da immer neue Items eintreffen. Da stellt sich die Frage, ob es überhaupt Sinn macht, APRIORI für Streams zu benutzen. Ein Vorschlag wurde mit dem Algorithmus WINEPI gemacht.

4.2.1 WINEPI

Ursprünglich wurde WINEPI [8] entwickelt, um häufige Sequenzen in einem Telekommunikationsnetzwerk zu finden. Er lässt sich aber durch kleine Veränderungen auf andere sequenzielle Daten erweitern. Im ursprünglichen Algorithmus werden parallele Sequenzen, serielle Sequenzen oder Sequenzen, die eine Kombination aus diesen darstellen, betrachtet (Abbildungen 4.1, 4.2, 4.3,). Sequenzen bestehen hier aus sogenannten Events.

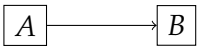


Abbildung 4.1: Serielle Sequenz mit Events A, B und C

Abbildung 4.2: Parallele Sequenz mit Events A und B

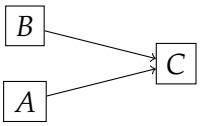


Abbildung 4.3: Kombinierte Sequenz mit Events A, B und C

In den Abbildungen sind A,B und C die Events. Betrachtet wird der Datenstrom mithilfe eines Fensters fester Länge, das über den Stream geschoben wird. Nur Sequenzen innerhalb dieses Fensters werden betrachtet. Die Häufigkeit einer Sequenz wird dann dargestellt mit der Anzahl der Fenster in denen diese Sequenz vorkommt. Wie bei APRIORI werden zunächst die häufigen 1-elementigen Sequenzen über die Fenster ermittelt, dann Kandidaten erzeugt und geprüft, bis keine Kandidatengenerierung mehr möglich ist.

Ermittlung der Häufigkeiten

Der Algorithmus vollzieht inkrementelle Aktualisierungen auf die erstellten Datenstrukturen. Er fängt mit dem leeren Fenster vor den ersten Sequenzen im Strom an und endet mit dem leeren Fenster nach den letzten Sequenzen.

Parallele Sequenzen Für jede parallele Sequenz *ps* gibt es einen Zähler *ps*.counter. Dieser Zähler sagt aus, wieviele Events der Sequenz in dem aktuellen Fenster vorhanden sind. Wenn die Anzahl Events in dem Zähler gleich der Größe der Sequenz wird, wird angenommen, dass die komplette Sequenz in dem Fenster vorhanden ist. Wenn der Zähler für *ps* sich wieder verringert, die Sequenz also nichtmehr vollständig im Fenster liegt, wird die Häufigkeit von *ps* um 1 erhöht. Zum Schluss bekommt man die Anzahl der Fenster, in denen die parallele Sequenz komplett drin lag.

Serielle Sequenzen Um serielle Sequenzen zu erkennen, werden endliche Automaten benutzt. Die einzelnen Zustände des Automaten stellen die Präfixe der Sequenzen da. Wenn ein Automat in den akzeptierenden Zustand kommt, ist die Sequenz vollständig in dem gerade betrachteten Fenster enthalten. Für jede Sequenz gibt es

einen Automaten. Es kann auch mehrere Automaten gleichzeitig geben für eine Sequenz. Sobald das erste Event einer Sequenz das Fenster betritt, wird ein Automat initialisiert. Verlässt das erste Event das Fenster wieder, wird der Automat gelöscht. Wird nun ein Automat, der sich im akzeptierenden Zustand befindet, entfernt, und befindet sich kein anderer Automat in diesem Zustand, wird die Häufigkeit für diese Sequenz erhöht.

Kombinierte Sequenzen Bei kombinierten Sequenzen ist der einfachste Weg, alle Events in der Sequenz zunächst wie bei parallelen Sequenzen zu verarbeiten. Findet man in einem Fenster die komplette Sequenz wieder, kann man überprüfen, ob die benötigte partielle Ordnung vorhanden ist.

Wie schon erwähnt ist die Kandidatengenerierung von APRIORI problematisch. Bei seriellen Sequenzen kann es je nach Fenstergröße vorkommen, dass viele Automaten generiert und verwaltet werden müssen. Dadurch könnte viel Speicherplatz gebraucht werden. Zudem erfordert die Überprüfung vieler Automaten als Kandidaten zusätzlichen Aufwand.

4.3 Algorithmen über die gesamte Stromgeschichte

Jetzt werden zwei sich ähnelnde Algorithmen betrachtet, die häufige Mengen über die gesamte Geschichte des Datenstroms ausgeben. Sie betrachten die Häufigkeiten der Items nicht über bestimmte Abschnitte, sondern über den gesamten Strom. Es werden zwar Fenster einer bestimmten Größe benutzt, doch wird zum Beispiel nicht gespeichert, dass ein Item zu Anfang häufig war und ab der Mitte des Stroms nicht mehr. Der *Support* bezieht sich also auf den gesamten Strom. Somit gibt die benutzte Datenstruktur keine Hinweise auf zeitabhängige Häufigkeiten. Die Algorithmen können also keine *Concept shifts* oder *Concept drifts* erkennen.
Ein naives Vorgehen, um häufige Mengen über den Verlauf des gesamten Stroms zu extrahieren, lässt sich einfach implementieren. Man geht den Datenstrom Item für Item durch und je nach Fall gibt es eine Vorgehensweise:

1. Das Item wurde schon einmal betrachtet. Hierbei wird für das gespeicherte Item einfach der Zähler um eins erhöht.
2. Das Item wurde noch nicht betrachtet. Hierbei wird ein neuer Eintrag mit einem Zähler mit Eins initialisiert.

Nachdem der Teilstrom durchlaufen wurde, kann man alle Einträge nochmal durchgehen, und die Items löschen, die den *minSupport* nicht erfüllen. Alternativ kann man dieses pruning auch zwischendurch durchführen, um den Speicher zu entlasten.

Dieses Vorgehen ist zwar einfach, jedoch ist auch schnell zu sehen, dass es Schwachstellen hat. Der Speicherbedarf ist unter Umständen groß. Als Beispiel nimmt man einen Datenstrom aus 1 Million Items, die sich alle unterscheiden. Findet kein Pruning statt, werden bis zum Ende des Stroms 1 Million Items mit ihren Häufigkeiten, in dem Falle eins, gespeichert. Ein *minSupport* von zwei reicht schon, damit die Speicherung dieser Items unnötig wird. Solch ein naiver Algorithmus erkennt das aber erst am Ende.

Effizienteres Vorgehen

Wie man sieht, ist ein naives Vorgehen nicht sehr effizient. Jetzt werden zwei Algorithmen vorgestellt, die speichereffizienter häufige Mengen über den ganzen Strom ermitteln. Die erzeugten Ausgaben sind zwar approximiert, jedoch ist sichergestellt, dass die Fehlertoleranz einen vom Benutzer spezifizierten Fehlerparameter nicht übersteigt. Es ist möglich, einzelne Items sowie Itemmengen über einen Strom zu ermitteln, die einen voreingestellten *minSupport* erfüllen. Nicht- häufige Mengen werden rechtzeitig gelöscht und nicht mehr gespeichert.

Zwar ist es mit diesen Algorithmen auch nicht möglich, *Concept shifts* oder *Concept drifts* zu erkennen, jedoch ist dies 1. nicht immer erforderlich und 2. sind sie speichereffizient, wenn man auch mit approximierten Ergebnissen zufrieden ist. Bei den beiden Algorithmen handelt es sich um STICKYSAMPLING und um LOSSYCOUNTING [13].
Die Algorithmen arbeiten mit zwei Parametern: einem *minSupport* und einem Fehlerparameter $\epsilon \in (0,1)$. Dabei ist $\epsilon \leq \delta$. N sei die aktuelle Länge des betrachteten Datenstroms. Für die Ausgaben der Algorithmen ist Folgendes garantiert:

1. Es gibt keine Itemmengen, die als *false negative* in der Ausgabe stehen und es werden alle Itemsets ausgegeben, deren Häufigkeit $\geq \delta$ * N ist.
2. Es werden keine Itemmengen ausgegeben deren Häufigkeit < (δ - ϵ) * N ist.
3. Die Häufigkeiten der ausgegebenen Itemmengen sind zwar approximiert, doch die geschätzten Häufigkeiten sind nicht weiter als ϵ * N vom richtigen Wert entfernt.

4.3.1 Sticky sampling

STICKYSAMPLING [13] benötigt noch einen weiteren Parameter als die zuvor beschriebenen: die Versagenswahrscheinlichkeit λ.
Die benutzte Datenstruktur ist eine Menge aus Einträgen der Form $(i, s(o))$, wobei i ein Item und $s(o)$ die geschätzte Häufigkeit dieses Items ist. Für jedes eintreffende Item gibt es 2 Möglichkeiten: falls es schon in der Datentruktur vorhanden ist, wird $s(o)$ erhöht. Andererseits wird das Item mit einer Samplingrate r gesampled. Falls dadurch das Item ausgewählt wird, wird ein neuer Eintrag für das Item in die Datenstruktur hinzugefügt.

Die benutzte Samplingrate r nimmt logarithmisch mit wachsendem Datenstrom in gewissen Abständen zu.

Bei jeder Veränderung der Samplingrate und somit der Veränderung der Wahrscheinlichkeit des Auswählens eines Items, müssen die vorhandenen Einträge in der Datenstruktur aktualisiert werden. Dabei wird für jeden Eintrag eine faire Münze geworfen bis das erste mal Kopf erscheint. Bis dahin wird für jedes Erscheinen von Zahl $s(o)$ dekrementiert. Auf diese Weise wird die Datenstruktur so angepasst, als ob sie von Anfang an mit der neuen Samplingrate gesampled wurde. Die Items mit $s(o) \geq (\delta$ - $\lambda)$ * N können jederzeit ausgegeben werden, indem einfach die Einträge ausgegeben werden.

4.3.2 Lossy counting

LossyCounting [13] benötigt im Gegensatz zu StickySampling nur die zwei Parameter ϵ und δ. Der Strom wird in gleich große Behälter der Größe $\lceil \frac{1}{\epsilon} \rceil$ unterteilt. Diese Behälter bekommen auch eine ID zugewiesen, die angibt, um welchen Behälter es sich handelt. Der aktuellste Behälter ist dann $b_{aktuell}$. In der benutzten Datenstruktur werden Einträge der Form (i, $s(o)$, μ) gespeichert. i ist hierbei das Item, $s(o)$ die geschätzte Häufigkeit und μ der maximal mögliche Fehler in $s(o)$.

Der Algorithmus geht wie folgt vor: Wir schauen uns den Strom Behälterweise an. Für jedes Item in dem gerade betrachteten Behälter schauen wir zunächst, ob es einen Eintrag in der Datenstruktur gibt. Falls dies der Fall ist, wird die Häufigkeit um 1 erhöht. Falls der Eintrag nicht vorhanden ist, wird ein neuer erstellt. Der neue Eintrag sieht dann so aus : (i, 1, $b_{aktuell}$ - 1).
Mit fortlaufendem Datenstrom muss auch möglichst sichergestellt werden, dass ein Item, falls es nicht mehr häufig ist, gelöscht wird. Dafür wird in bestimmten Abständen, die man selber wählen kann, gepruned. Es werden Einträge gelöscht, die folgende Bedingung erfüllen: $s(o) + \mu \leq b_{aktuell}$

4.4 Algorithmen für maximal häufige Mengen

Die Suche nach maximal häufigen Mengen schränkt die Anzahl der ausgegebenen häufigen Mengen stark ein. Diese Untermenge aller häufigen Mengen ist zwar nicht redundant, jedoch kann man die Häufigkeiten der Untermengen nicht mehr ableiten. Falls die Häufigkeiten nicht relevant sind für den Benutzer, sind maximal häufige Mengen eine gute Wahl.

4.4.1 Max-FISM

Ein Algorithmus zur Suche von maximal häufigen Mengen ist MAX-FISM(**Max***imal* − **F***requent***I***temset***M***ining*) [15]. Es wird ein sliding-window Modell benutzt, bei dem je immer eine Transaktion in das Fenster rutscht und die älteste verworfen wird. Somit behandelt dieser Algorithmus aktuelle häufige maximale Mengen. Benutzt wird eine präfix-Baum basierte Datenstruktur namens *Max-Set*. Die häufigen Mengen können jederzeit vom Benutzer abgefragt werden.

Der Algorithmus beginnt zunächst damit, das Fenster mit der gewünschten Anzahl an Transaktionen zu füllen (vom Benutzer festgelegte Fenstergröße w). Ist das Fenster voll, wird es eine Transaktion nach rechts verschoben. Dabei entfällt die älteste Transaktion. Der Algorithmus führt die folgenden 5 Schritte aus:

1. Lese und sortiere neue Transaktion (Sortierung dient der leichteren Verarbeitung)
2. Füge neues maximales itemset in *Max-Set* ein (Bei t = $\{a, c, d\}$ also $\{acd\}$)
3. Eliminiere die Auswirkungen der ältesten gelöschten Transaktion
4. Finde die maximal häufige Itemmenge im aktuellen *Max-Set*
5. (optional) Pruning

Max-Set

Bei der benutzten Datenstruktur *Max-Set* handelt es sich um eine Baumstruktur. Die Wurzel ist stets *Null*. Alle anderen Knoten speichern das item und die Häufigkeit bezogen auf das gerade betrachtete Fenster. Die Blätter im Baum haben jeweils einen Zeiger auf das nächste linksliegende Blatt.

Falls nun eine neue Transaktion das Fenster betritt, wird aus den items in der Transaktion die maximale Menge gebildet. Falls diese Menge schon in *Max-Set* vorhanden ist, wird die Häufigkeit inkrementiert. Falls es noch nicht vorhanden ist, wird es mit der Häufigkeit 1 hinzugefügt. Verlässt eine Transaktion das Fenster, wird die Häufigkeit beim dazugehörigen Knoten dekrementiert. Fällt die Häufigkeit auf 0, wird der Knoten gelöscht.

Maximale Häufige Mengen extrahieren

Wenn nun vom Benutzer die häufigen maximalen Mengen angefordert werden, müssen diese aus *Max-Set* extrahiert werden. Hierfür werden zwei Listen benutzt.

MFI-Liste beinhaltet alle maximal häufigen Mengen, die bis jetzt gefunden wurden.

MIFI-Liste beinhaltet alle minimal nichthäufigen Mengen. Dabei handelt es sich um Mengen, die selber zwar nicht häufig sind, aber bei denen alle Untermengen häufig sind.

Der Algorithmus extrahiert die maximal häufigen Mengen, indem er von links nach rechts alle Blätter durchgeht. Dabei sind mehrere Fälle möglich:

1. Das betrachtete Item hat eine Häufigkeit > δ und ist nicht in der MFI-Liste. Dann wird das Item in die Liste hinzugefügt.

2. Das betrachtete Item hat eine Häufigkeit > δ und ist in der MFI-Liste (auch als Untermenge einer Obermenge). Dann wird das Item nicht weiter betrachtet.

3. Das betrachtete Item hat eine Häufigkeit < δ und ist nicht in der MIFI-Liste. Dann wird das Item in die Liste hinzugefügt. Dann werden alle (k-1)-elementigen Untermengen untersucht. Die Häufigkeit für die gebildeten Untermengen wird summiert aus den Vorkomnissen der Menge in den Obermengen. Wenn das Item $\{c, d\}$ untersucht wird, und es gibt $\{a, c, d\}$ mit der Häufigkeit 3 und $\{b, c, d\}$ mit der Häufigkeit 1, dann bekommt $\{c, d\}$ die Häufigkeit 4. Mit dieser neuen Häufigkeit wird es dann weiter behandelt, je nachdem welcher Punkt nun zutrifft.

4. Das betrachtete Item hat eine Häufigkeit < δ und ist in der MIFI-Liste. Dann wird das Item nicht weiter betrachtet, sondern seine (k-1)-elementigen Untermengen.

4.4.2 INSTANT

Beim vorherigen Algorithmus MAX-FISM sieht man sehr gut das sogenannte 2-Phasen-Prinzip. Zunächst wird eine interne Daternstruktur verwendet, um ausgewählte Mengen aus dem Datenstrom zu speichern (in dem Fall das *Max-Set*). Wenn der Benutzer nun die häufigen Mengen anfragen möchte, wird diese Datenstruktur einmal durchgegangen, um diese zu extrahieren. Man muss also jedes mal, wenn eine Transaktion das Fenster betritt oder verlässt, die Datenstruktur aktualisieren, sowie diese nochmal zum Extrahieren durchgehen. Der Algorithmus INSTANT (max**I**mal freque**N**t So-far Itemset m**A**i**NT**ainer) [14] muss nur den ersten Schritt durchführen. Die Mengen werden in Listen L_i gespeichert, die Mengen mit bestimmten Häufigkeiten beinhalten. Hat man also einen *minSupport* von drei, gibt es drei Listen (Häufigkeit 1 (L_1), 2 (L_2) und 3++ (L_h)). Sobald eine Menge in die Liste L_h gelangt, gilt sie als häufig und wird ausgegeben. Mengen werden also sofort ausgegeben, sobald sie als häufig eingestuft werden.

Transaktion t	L_h	L_2	L_1	Erklärung
	{}	{}	{}	initialisieren
abc	{}	{}	{abc}	t in L_1 einfügen
acd	{}	{ac}	{abc,acd}	t in L_1 einfügen; ac in L_2 einfügen
abcd	{ac}	{abc,acd}	{abcd}	{ac} in L_h einfügen; lösche {ac} aus L_2; {abc},{acd} in L_2 einfügen; lösche {abc},{acd} aus L_1;füge t in L_1 ein
abd	{ac,ab,ad}	{abc,acd,abd}	{abcd}	{ab},{ad} in L_1 einfügen; {abd} in L_2 einfügen
abcd	{abc,acd,abd}	{abcd}	{}	{ac},{ab},{ad} aus L_h löschen; füge {abc},{acd},{abd} in L_h ein; lösche {abc}, {acd}, {abd} aus L_2; füge {abcd} in L_2 ein; lösche {abcd} aus L_1
acd	keine Veränderung	keine Veränderung	keine Veränderung	keine Aktion

Tabelle 4.1: Beispiel für den Ablauf von INSTANT (Beispiel aus [14])

Algorithmus

Das Vorgehen des Algorithmus ist relativ simpel. Eine im Fenster eintreffende Transaktion t wird zunächst lexikografisch geordnet. Danach werden, angefangen bei L_h, die Listen durchgegangen. Falls die Transaktion t, oder eine Obermenge dieser, in L_h vorkommt, wird nichts gemacht. Wenn t nicht in L_h vorkommt, wird zur nächsten Liste übergegangen. Falls dort t oder eine Untermenge von t vorkommt, wird die Häufigkeit dieser Menge dahingehend erhöht, dass die Menge in die „nächsthöhere“ Liste hinzugefügt wird. Wenn der Algorithmus bei der letzten Liste L_1 angelangt ist und t nicht dort vorkommt, wird t in diese Liste eingefügt. Sobald eine Menge in L_h hochgestuft wird, ist zu beachten, dass alle Untermengen von t gelöscht werden, da uns nur die maximal häufigen Mengen interessieren. In der Tabelle 4.1 ist ein Beispiel aus [14] mit $\delta = 3$ und sechs Transaktionen dargestellt.

4.5 Algorithmen für geschlossene häufige Mengen

Geschlossene häufige Mengen geben Informationen über eine Datenbank oder einen Datenstrom in zusammengefasster Form. Es werden nur Mengen ausgegeben, dessen Häufigkeit nicht gleich dem einer seiner Obermengen ist. Somit sind die ausgegebenen Mengen auch nicht redundant. Die Informationen sind zudem auch komplett, da man aus den ausgegebenen Mengen die Häufigkeiten der Untermengen ableiten kann. Deshalb sind geschlossene Mengen eine gute Wahl, wenn man nicht redundante Informationen sammeln möchte und dennoch die Häufigkeiten benötigt.

Der Algorithmus MOMENT [5] war der erste Versuch einer Umsetzung für dieses Problem. Er benutzt eine Baumstruktur namens *Closed Enumerate Tree (CET Tree)*. Jeder Knoten besteht aus einem Itemset und einem Knotentypen. Es gibt insgesamt vier Knotentypen. Dabei werden nur in einem Knotentypen geschlossene Mengen gespeichert. Die anderen drei Typen speichern Mengen, die eine Art Grenze zwischen den geschlossenen und nicht geschlossenen häufigen Mengen bilden. Diese Grenze wird dann genutzt, um zu entscheiden, ob eine häufige Menge geschlossen ist. Das bedeutet, dass es auch viele redundante Knoten gibt. Für eine geschlossene Menge können unter Umständen bis zu 20 weitere Knoten existieren. Jede Transaktion die eintrifft, muss nach ihrem Typ überprüft werden und dafür der Baum durchlaufen werden. Dieser Vorgang kann bei vielen geschlossenen Mengen sehr lange dauern. Da viele Knoten gespeichert werden müssen, wird auch viel Speicherplatz konsumiert.

In diesem Kapitel werden nun zwei Algorithmen vorgestellt, die versuchen diesen Makel zu beseitigen. CFI-STREAM [6] benutzt auch eine Baumstruktur, die allerdings nur geschlossene Mengen speichert. Der Algorithmus CLOSTREAM [7] hingegen verfolgt einen Ansatz ohne Baumerstellung.

4.5.1 CFI-Stream

Bei CFI-STREAM [6] wird ein sliding-window der Größe w benutzt und eine Baumstruktur namens Direct Update Tree (DIU-tree). Der Baum besteht aus k Ebenen, wobei k gleich der Länge der größten geschlossenen Menge ist. Wenn also die größte geschlossene Menge $\{a, b, c, d\}$ ist, hat der Baum vier Ebenen. In jedem Knoten werden die Items, die Häufigkeiten und Zeiger auf die Kind- und Elternknoten gespeichert. Da bei CFI-STREAM die aktuellen geschlossenen häufigen Mengen gesucht werden, besteht der Baum aus Knoten, die Items aus dem aktuellen Fenster repräsentieren. Das Fenster wird jeweils um eine Transaktion verschoben.

Eintreffen einer neuen Itemmenge

Trifft eine neue Itemmenge X ein, werden mehrere Fälle betrachtet:

1. X ist im DIU-tree enthalten: Die Häufigkeit von X und allen Untermengen wird erhöht.

2. X ist nicht im DIU-tree enthalten, kam aber im aktuellen Fenster schon mindestens einmal vor, zum Beispiel als Untermenge (wird geprüft, indem Obermengen von X im Baum gesucht werden): Es wird geprüft, ob X oder eine Untermenge nun eine geschlossene Menge bildet. Ist dies der Fall werden sie in den Baum hinzugefügt. Falls eine Untermenge von X schon im DIU-tree ist, wird die Häufigkeit erhöht.

3. X ist nicht im DIU-tree und kam bisher auch nicht vor: X wird im DIU tree hinzugefügt. Für die Untermengen von X wird geprüft, ob sie nun geschlossen sind.

Die Prüfung, ob eine Menge geschlossen ist, passiert hierbei durch Prüfung aller Obermengen im Baum.

Vorgehen bei Entlassen einer Itemmenge

Verlässt eine Itemmenge X das Fenster, ist die Aktualisierung der Struktur etwas einfacher:

1. X ist im DIU-tree und die Häufigkeit beträgt eins: Lösche den dazugehörigen Knoten.

2. X ist im DIU-tree und die Häufigkeit ist ≥ 2: Die Häufigkeit von X und allen Untermengen wird aktualisiert.

3. X ist nicht im DIU-tree: Es wird überprüft, ob X oder eine Untermenge nun geschlossen ist.

CFI-STREAM speichert weniger Informationen als MOMENT, jedoch muss die Datenstruktur unter Umständen oft durchlaufen werden. Wenn eine Itemmenge X eintrifft, müssen oft alle Untermengen Y darauf überprüft werden, ob sie nun selber geschlossen sind. Dafür müssen im DIU-tree alle Obermengen von Y geprüft werden. Je nach Fenstergröße kann das viel Zeit beanspruchen.

4.5.2 CloStream

Der Algorithmus CLOSTREAM [7] hingegen umgeht die Prüfung aller Untermengen. Als Datenstruktur werden hierbei eine Tabelle (*Closed Table*) und eine Liste (*Cid List*) benutzt.

In der *Closed Table* wird die *Cid* (Closed id), ein Item und die Häufigkeit des Items gespeichert. Die *Cid List* speichert alle einzelnen Items und die *Cids* aller Itemmengen in der *Closed Table*, die dieses Item enthalten.

Cid	Item	s
0	$\{\}$	0
1	$\{c, d\}$	2
2	$\{a, b\}$	3
3	$\{a, b, c\}$	2
4	$\{c\}$	4
5	$\{a, c, d\}$	1
6	$\{a\}$	4
7	$\{a, c\}$	3

Tabelle 4.2: Beispiel einer *Closed Table* aus [7]

Item	Cidset
a	$\{2, 3, 5, 6, 7\}$
b	$\{2, 3\}$
c	$\{1, 3, 4, 5, 7\}$
d	$\{1, 5\}$

Tabelle 4.3: Beispiel einer dazugehörigen *Cid List* aus [7]

Vorgehen

Die Tabellen 4.2 und 4.3 stellen Beispiele dar, nachdem fünf Transaktionen eingegangen sind. Das Vorgehen vom Algorithmus wird nun anhand des Eintreffens einer weiteren Transaktion $\{b, c\}$ dargestellt (Beispiel aus [7]). Es werden zwei Phasen durchlaufen:

1. Phase In der ersten Phase werden alle Items gesucht, die aktualisiert werden müssen. Um die Items abzuspeichern, wird eine temporäre Tabelle *tempTabelle* mit zwei Spalten benutzt. In der linken Spalte werden Items und in der rechten IDs zu diesen gespeichert. Trifft nun die Transaktion $\{b, c\}$ ein, wird $\{b, c\}$ in *tempTabelle* eingefügt. Die ID wird zunächst auf Null gesetzt. Dann wird SET($\{b, c\}$) berechnet. Das Ergebnis von SET ist die Vereinigung der *Cid List* von b und der von c. SET($\{b, c\}$) wäre somit $\{2, 3\} \bigcup \{1, 3, 4, 5, 7\} = \{1, 2, 3, 4, 5, 7\}$. Für jede *Cid* aus $\{1, 2, 3, 4, 5, 7\}$ wird nun der Durchschnitt mit $\{b, c\}$ erzeugt. Für *Cid*(1) wäre das Ergebnis $\{c, d\} \bigcap \{b, c\} = \{c\}$. $\{c\}$ wird in *tempTabelle* mit der ID 1 eingefügt. Dieser Vorgang wird

für die restlichen *Cid*(2) - *Cid*(7) wiederholt und die zu vergebene ID inkrementiert. Eine Ausnahmebehandlung findet statt, wenn das Ergebniss aus SET schon in *tempTabelle* enthalten ist. In diesem Beispiel wäre das bei *Cid*(3) = $\{a, b, c\}$. Hierfür wäre die Rechnung $\{a, b, c\} \bigcap \{b, c\} = \{b, c\}$. $\{b, c\}$ ist aber schon in *tempTabelle* enthalten. Nun wird die aktuelle ID von $\{b, c\}$ betrachtet, in diesem Falle 0. Es wird die Häufigkeit aus *Closed Table* des Items mit der *Cid*(0) mit der des Items mit der *Cid*(3) verglichen. Da s(*Cid*(0)) geringer ist als s(*Cid*(3)), bekommt $\{b, c\}$ in *tempTabelle* die neue ID 3 (SC von *Cid*(3) + 1). Die resultierende *tempTabelle* ist in Tabelle 4.4 dargestellt.

2. Phase In der zweiten Phase werden die Häufigkeiten der Items aus *tempTabelle* in *Closed Table* aktualisiert. Der erste Eintrag ist $\{c\}$ mit der ID 4. CloStream findet nun in *Closed Table* den Eintrag $\{c\}$ bei Cid(4). Die Häufigkeit für $\{c\}$ wird dann inkrementiert. Das nächste Item wäre $\{b\}$ mit der ID 2. Dafür exisitert aber kein Eintrag in *Closed Table*. Es wird ein neuer Eintrag für $\{b\}$ mit der *Cid* 8 und s = (s(*Cid*(2) + 1) erstellt. Mit dem letzten Eintrag in *tempTabelle* wird genauso verfahren. Die Ergebnistabellen sind in Tabelle 4.5 und 4.6 dargestellt.

Item	ID
$\{c\}$	4
$\{b\}$	2
$\{b, c\}$	3

Tabelle 4.4: *tempTabelle* nach der 1. Phase für die Transaktion $\{b, c\}$ aus [7]

Cid	Item	s
0	$\{\}$	0
1	$\{c, d\}$	2
2	$\{a, b\}$	3
3	$\{a, b, c\}$	2
4	$\{c\}$	5
5	$\{a, c, d\}$	1
6	$\{a\}$	4
7	$\{a, c\}$	3
8	$\{b\}$	4
9	$\{b, c\}$	3

Tabelle 4.5: *Closed Table* nachdem Transaktion $\{b, c\}$ verarbeitet wurde aus [7]

Item	Cidset
a	$\{2,3,5,6,7\}$
b	$\{2,3,8,9\}$
c	$\{1,3,4,5,7,9\}$
d	$\{1,5\}$

Tabelle 4.6: *Cid List* nachdem Transaktion $\{b,c\}$ verarbeitet wurde aus [7]

Die häufigen geschlossenen Mengen können nun einfach ausgegeben werden, indem *Closed Table* durchgegangen wird.

Nun werden noch zwei Algorithmen vorgestellt, um alle häufigen Mengen zu finden.

4.6 Algorithmus basierend auf FPGrowth

4.6.1 FPStream

Viele Algorithmen nutzen als Vorbild den Algorithmus FPGROWTH. Einer von ihnen ist FPSTREAM [11]. Dieser Algorithmus benutzt eine Struktur ähnlich dem FPTREE. Durch eine simple Erweiterung von diesem ist es möglich, die häufigen Mengen über den gesamten Datenstrom abzurufen, und zwar so, dass die Abfrage auch zeitabhängig sein kann.

Es wird ein sliding-window der Größe w genutzt, um über den Strom zu gehen. Um die Transaktionen im aktuellen Fenster zu speichern, wird zunächst der bekannte FPTREE verwendet. Um die Transaktionen aus den vergangenen Fenstern zu speichern, wird eine neue Datenstruktur, nämlich der *pattern-tree*, genutzt.

Im Gegensatz zu anderen Algorithmen werden hier auch Mengen betrachtet, die gerade nicht häufig sind. Der Hintergrund ist einfach: falls nichthäufige Mengen später häufig werden, kann man ihre korrekte Häufigkeit über den gesamten Datenstrom noch nachverfolgen. Deshalb werden Mengen hier in drei Kategorien unterteilt: häufige Mengen, nicht-häufige Mengen und subhäufige Mengen.

Definition 4.6.1 (häufige, subhäufige und nicht-häufige Mengen). Gegeben sei ein *min-Support* δ, ein Fehlerparameter ϵ und eine Lockerungsrate $\sigma = \frac{\epsilon}{\delta}$.
Eine Menge ist häufig, wenn die Häufigkeit $\geq \delta$, subhäufig wenn sie $< \delta$ aber nicht $< \epsilon$ und sonst nicht-häufig.

Es ist natürlich nicht effizient, die nicht-häufigen Mengen zu speichern. Daher interessieren wir uns hier nur für die subhäufigen Mengen.
Wie schon erwähnt ähnelt der pattern-tree stark dem FPTREE. Der Unterschied besteht lediglich darin, dass in den Knoten nicht nur Mengen mit ihren Häufigkeiten, sondern für jeden Knoten, für das jeweilige Item, zusätzlich eine Tabelle mit den Häufigkeiten für jedes bisher betrachtete Fenster gespeichert wird. Somit ist es möglich die Häufigkeiten der Items für jeden Zeitpunkt exakt zu bestimmen.

Das erste Fenster wird etwas anders behandelt als die restlichen. Nachdem es mit der benötigten Anzahl an Transaktionen gefüllt wurde, werden die Häufigkeiten der Items ermittelt und dann wird eine Liste *freqList* erstellt. In dieser Liste sind die Items absteigend nach ihrer Häufigkeit sortiert. Diese Ordnung gilt dann für alle weiteren Fenster. Aus diesem Fenster werden dann mit FPGROWTH die Mengen mit einer Häufigkeit von (ϵ * w) gesammelt. Aus dem FPTREE werden die subhäufigen und häufigen Mengen extrahiert und der pattern-tree damit initialisiert. Danach wird das Fenster $|w|$ Transaktionen weitergeschoben.

Die Transaktionen in den folgenden Fenstern werden nach der Ordnung in der initialen *freqList* sortiert. Es wird wieder FPGROWTH benutzt zur Ermittlung der subhäufigen und häufigen Mengen, um dann eine Fallunterscheidung durchzuführen.

1. Das Item ist schon im *pattern-tree*: Füge die Häufigkeit des Items in die Tabelle des zugehörigen Knotens ein. Danach findet ein pruning statt. Wenn dadurch die Einträge in der Tabelle gelöscht werden, ermittelt FPGROWTH keine Obermengen mehr für dieses Item. Dies kann der Fall sein, wenn die Gesamthäufigkeit des Items, über alle Fenster, nicht mehr die Größe hat,so dass das Item als subhäufig gilt. Wenn die Einträge nicht gelöscht werden, dann werden weiterhin Obermengen gesammelt.

2. Falls das Item nicht in der Struktur ist, wird ein Eintrag angelegt.

Danach wird der *pattern-tree* nochmal durchgegangen. Wenn für ein Item keine Aktualisierung vorgenommen wurde, wird für das Fenster in der Tabelle eine Null eingetragen. Wenn beim Durchgehen ein Blatt erreicht wird mit einer leeren Tabelle, wird es einfach gelöscht.

Hier können häufige Mengen über den gesamten Strom verfolgt werden, jedoch muss dafür der Baum wie bei MOMENT ebenfalls oft durchlaufen werden. Es werden vielleicht nicht soviele Knoten gespeichert, allerdings darf nicht vernachlässigt werden, dass für jeden Knoten zusätzlich eine Tabelle gespeichert werden muss. Bei vielen häufigen Mengen kann das zu erhöhtem Speicherverbrauch und erhöhter Laufzeit führen. Ein Vorteil

der Tabellen ist allerdings, dass es damit möglich ist, leichte *concept shifts* zu erkennen. Solange eine Tabelle nicht gelöscht wird, weil der *minSupport* über den gesamten Strom nichtmehr gegeben ist, sieht man daran, zu welchen Zeiten ein Item nicht häufig sondern subhäufig war.

4.7 Weiterer Ansatz

4.7.1 Algorithmus von Toon und Calders

Der Algorithmus von Toon, Calders und Goethals [2] ist ein weiteres Besipiel für einen Algorithmus, der nicht auf APRIORI oder FPGROWTH basiert. Der Strom wird mit dem landmark-window Modell durchlaufen und für alle häufigen Mengen wird eine Zusammenfassung (*Summary*) erstellt. Dabei handelt es sich um Listen deren Einträge aus einer Häufigkeit und einer Grenze die eine Stelle im Strom markiert. Diese Summarys verbrauchen 1. sehr wenig Speicherplatz und 2. ist es damit möglich, zu jeder Zeit die Häufigkeiten von Mengen zu erstellen. Hier wird der Ansatz verfolgt, dass das Fenster betrachtet wird, in dem die Häufigkeit eines Items am größten ist. Dabei handelt es sich immer um das „letzte“ Fenster. Somit sind die aktuell häufigsten Mengen immer im aktuellen Fenster enthalten.

Sei $\mathcal{S} < t_1, t_2, \dots >$ ein Strom aus Transaktionen. Dabei ist t_1 die älteste Transaktion. Dann ist $\mathcal{S}[s,e]$ der Teilstrom $< t_s, i_{s+1}, ..., i_e >$. $h(\mathcal{I}, \mathcal{S})$ ist die Anzahl der Häufigkeit eines Items in $\mathcal{S}$. Wie schon erwähnt, betrachten wir das Fenster im Strom, in dem die Häufigkeit eines Items am größten ist. Dieses Fenster wird maximales Fenster genannt.

Algorithmus

Der Algorithmus wird an einem einfachen Beispiel mit der Fensterlänge $w = 1$ und für nur ein Item A präsentiert. Er kann für Itemmengen und ein größeres Fenster angepasst werden. Doch zum Verständnis von dem Algorithmus ist dies nicht erforderlich.
In dem Summarys werden sogennante Grenzen und die Häufigkeiten der Zielmenge zwischen Grenzen gespeichert. Eine Grenze, in dem Falle q, muss stets folgende Bedingung erfüllen:

$$\forall\, p \text{ und } r \text{ mit } p < q \leq r \text{ gilt:}$$
$$h(A, \mathcal{S}[p, q-1]) < h(A, \mathcal{S}[q, r]).$$

Mit Worten ausgedrückt, muss die Häufigkeit eines Items vor der Grenze geringer sein als die Häufigkeit danach. In dem unten abgebildeten Beispiel sind zwei Slashes und drei vertikale Linien zu sehen. Die Slashes bilden keine Grenzen, die vertikalen Linien hinge-

gen schon.

$$< \mid \overbrace{\text{aaabbbabb}}^{4/9} / \overbrace{\text{abababababb}}^{4/10} \text{b} \mid \overbrace{\text{aab}}^{2/3} / \overbrace{\text{ab}\mid}^{1/2} \text{a} >$$

Die Summarys:

Seien $p_1 < p_2 < ... < p_r$ Grenzen für ein Item A in $\mathcal{S}$.

$$a_i = count(A, \mathcal{S}[p_i, p_{i+1} - 1])$$

ist dann die Häufigkeit eines Items zwischen zwei Grenzen p_i und p_{i+1}. Damit kann mit einer *Summary* ganz leicht die Häufigkeit eines Items zu jeder beliebigen Grenze ermittelt werden:

$$h(\text{A}, \mathcal{S}_t[p_i, t]) = \sum_{j=i}^{r} \frac{a_j}{t - p_{i+1}}$$

Schaut man sich nun die *Summary* aus dem Beispiel an und erinnert sich an die vorrausgesetzte Bedingung, ist leicht zu sehen, dass der letzte Eintrag der *Summary* die maximale Häufigkeit des Items angibt.

Algorithmus

Die erste *Summary* wird initialisiert, sobald das Zielitem A das erste mal im Strom vorkommt. Für jedes weitere Item kommen zwei Fälle in Frage (für $w = 1$):

1. Eine Obermenge des Items wird als nächstes eingelesen,
a) wenn die vorherige Transaktion auch das Zielitem A beinhaltete, müssen wir den letzten Eintrag in der *Summary* inkrementieren. Ansonsten müssen wir eine neue Grenze bilden, da die Häufigkeit nun höher ist als die vorherige (nämlich $\frac{1}{1}$), und
b) keine vorhandenen Grenzen können gelöscht werden.

2. Die eintreffende Transaktion beinhaltet nicht unser Zielitem A, also

a) kann keine neue Grenze gesetzt werden, aber
b) hier kann es vorkommen, dass ein Grenze gelöscht werden muss. aufgrund der Definition, müssen nur die zwei Häufigkeiten adjazent zur letzten Grenze betrachtet werden. Damit wir eine Grenze löschen können, müssen wir zunächst einen Block vor einer Grenze finden, dessen Häufigkeit höher ist als die nach der Grenze. Der Block mit der höchsten Häufigkeit ist nach Definition gerade der Block, vor der letzten Grenze q. Also

müssen wir nur die Häufigkeit von dem Block vor q mit der Häufigkeit des Blocks nach q bis zum aktuellen Zeitpunkt vergleichen. Falls die vorherige Häufigkeit höher ist als die aktuelle, weil wir eine Transaktion gelesen haben die nicht unser Zielitem A enthält und dadurch die Gesamthäufigkeit gesunken ist, müssen wir die Grenze q löschen um die Bedingungen wieder zu erfüllen.

Man sieht, dass die Aktualisierungsfunktion für die *Summary* sehr effizient ist, da nicht die gesamte *Summary*, sondern nur die letzten zwei Blöcke betrachtet werden müssen, also der Block vor und nach der letzten Grenze.

Dieser Algorithmus bietet eine Datenstruktur, die alle Mengen speichert, die den *minSupport* erfüllen. Die Summarys lassen sich sehr leicht aktualisieren und die Mengen sich über den ganzen Strom verfolgen. Es lassen sich allerdings nur *concept shifts* und *concept drifts* für den aktuellen Zeitraum betrachten.

Zusammenfassung

Dies war natürlich nur ein kleiner Einblick in ausgewählte Algorithmen. Es wurden interessante Ansätze gezeigt und eventuelle Probleme mit diesen erwähnt. Es werden verschiedene Datenstrukturen und Verfahren genutzt um Teilprobleme zu lösen. Allerdings haben alle Algorithmen eines gemeinsam. Es ist entweder nur schwer oder gar nicht möglich *concept shifts* und *concept drifts* zu erkennen. Der folgende Algorithmus bietet mit seiner Datenstruktur und seinem Vorgehen eine Lösung.

5

StreamKrimp

5.1 Krimp

Krimp [1] ist zunächst ein Algorithmus, der für statische Datenbanken ausgelegt ist. StreamKrimp basiert auf diesem und erweitert ihn. Die in Krimp vorgestellten Verfahren und Datenstrukturen lassen sich durch einfache Erweiterungen auch für Datenströme einsetzen. Bisher wurden unter anderem Algorithmen vorgestellt, die alle häufigen Mengen aus einem Datenstrom extrahieren. Aber auch die Anzahl der häufigen Mengen kann mitunter sehr groß sein. Vorallem sind diese Mengen oft redundant. Das liegt an der in Kapitel 2 erwähnten Monotonie-Eigenschaft von häufigen Mengen: Untermengen von häufigen Mengen sind ebenfalls häufig.

Die Idee bei Krimp ist nun, die Anzahl der resultierenden Mengen zu verringern, indem nur diejenigen genommen werden, die die Datenbank am besten beschreiben. Somit hat man die interessantesten Mengen rausgefiltert. Dafür wird das *Minimal Description Length Principle* angewendet. Grob kann man das MDL-Prinzip so beschreiben: Aus einer Menge von Modellen $\mathcal{H}$, gilt als bestes Modell $H \in \mathcal{H}$ dasjenige, welches den Datensatz am besten beschreibt. Im Fall von Krimp suchen wir jenes Modell H, welches

$$L(H) + L(\mathcal{DB} \mid H)$$

minimiert. Hierbei ist L eine Funktion, die angibt, wieviele Bits benötigt werden, um ein Objekt zu beschreiben. L(H) ist dann die Länge der Beschreibung in Bits des Modells und $L(\mathcal{DB} \mid H)$ ist die Länge der Beschreibung der Datenbank, wenn sie mit dem Modell

H kodiert wird.

Codetabellen

Das Kernstück von KRIMP sind die Codetabellen. Sie sind auch zugleich die betrachteten Modelle für die Beschreibung der Datenbank. Diese Tabellen bestehen aus zwei Spalten. In der Linken Spalte befinden sich Items oder auch eine Menge aus Items. In der rechten Spalte gibt es zu jedem Item einen Code.

Definition 5.1.1 (Codetabellen). Sei $\mathcal{I}$ die Menge der Items und $\mathcal{C}$ eine Menge von Codewörtern. Eine Codetabelle CT ist eine Tabelle mit twei Spalten, so dass

1. die Linke Spalte Itemmengen über $\mathcal{I}$ beinhaltet und
2. die rechte Spalte Elemente aus $\mathcal{C}$ beinhaltet. Dabei kommt jedes Element nur einmal vor.

Weiter beschreibt $\mathcal{CT}$ die Menge aller Codetabellen und $\mathcal{CS}$ die Menge aller Items in der Codetabelle.

Itemmenge	Code
{a}	$code_1$
{a,b}	$code_2$
{a,b,c}	$code_3$

Tabelle 5.1: Beispiel für eine Codetabelle

Mit den jeweiligen Codes zu den Items lässt sich nun die Datenbank kodieren. Damit man die gesamte Datenbank kodieren kann, müssen auch alle einzelnen Items aus $\mathcal{I}$ in der Codetabelle vorhanden sein. Um zu wissen, mit welchen Itemmengen man eine Transaktion t aus der Datenbank kodieren kann, benötigen wir die Funktion *cover*. Das Resultat aus *cover*(CT,t) ist eine disjunkte Menge von Itemmengen aus der Codetabelle CT, die die Transaktion t abdecken.

Definition 5.1.2 (Coverfunktion). Sei $\mathcal{DB}$ eine Datenbank über $\mathcal{I}$, t eine Transaktion aus $\mathcal{DB}$ und CT eine Codetabelle. Dann ist cover: $\mathcal{CT}$ x $\mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{I}))$ eine Coverfunktion, wenn sie Mengen ausgibt sodass:

1. *cover*(CT,t) ist eine Teilmenge von $\mathcal{CS}$
2. falls X,Y $\in$ *cover*(CT,t), dann ist entweder X = Y oder X $\bigcap$ Y = $\emptyset$

3. die Vereinigung aller X $\in$ *cover*(CT,t) ist gleich der Transaktion t

Es gibt für jede Codetabelle mindestens eine wohldefinierte Coverfunktion, da in den Codetabellen mindestens alle einzelnen Items vorkommen.

Um nun eine Transaktion t zu kodieren, müssen wir sie mit den Codes der Itemmengen aus der Coverfunktion für t ersetzen. Damit die Dekodierung eindeutig sein kann, muss es sich bei den Codes um *prefix Codes* handeln. Das bedeutet, dass kein Code das Präfix eines anderen Codes sein darf. Da wir daran interessiert sind, die Datenbank zu komprimieren, sollten die längsten und am häufigsten benutzten Items aus der Codetabelle die kürzesten Codes bekommen. Es gibt nun einen Zusammenhang zwischen der Wahrschenlichkeitsverteilung und der optimalen Codelänge, die wir mit der Shannon-Entropie berechnen können.

$$\text{L(X)} = \text{-log(P(X))}.$$

Um die Wahrscheinlichkeit des Auftretens eines Items benennen zu können, müssen wir bestimmen, wie oft eine Menge X genutzt wird, um alle Transaktionen zu decken.

Definition 5.1.3 (Usage). Sei $\mathcal{DB}$ eine Datenbank aus Transaktionen über $\mathcal{I}$, $\mathcal{C}$ Präfixcodes, *cover* eine Coverfunktion und CT eine Codetabelle über $\mathcal{I}$ und $\mathcal{C}$. *Usage* einer Itemmenge X $\in$ CT ist definiert als

$$\textit{usage}\ (\text{X}) = \mid \{\text{t} \in \mathcal{DB} \mid \text{X} \in \textit{cover}(\text{CT,t})\} \mid$$

Die Wahrscheinlichkeit, dass X benutzt wird, um eine Transaktion t $\in \mathcal{DB}$ zu decken, ist

$$\text{P(X} \mid \mathcal{DB}) = \frac{\textit{usage}(X)}{\sum_{Y \in CT} \textit{usage}(Y)}.$$

$code_{CT}(X)$ ist optimal für $\mathcal{DB}$, wenn

$$L(code_{CT}(X)) = |code_{CT}(X)| = \text{-log(P(X} \mid \mathcal{DB})).$$

Eine Codetabelle ist optimal für $\mathcal{DB}$, falls alle Codes optimal sind.

Nun kann man L($\mathcal{DB} \mid$CT) berechnen. Es handelt sich hierbei um die Summe der kodierten Transaktionen in der Datenbank. Die Grösse einer Transaktion ist wiederrum die Summe der Länge der Codes aller Items in *cover*(CT,t). Die Länge der Kodierung des Modells (Codetabelle) lässt sich so auch angeben. Da die rechte Spalte aus Codes besteht werden diese aufsummiert. Jede Itemmenge in der Codetabelle wird mit der Standardcodetabelle bestehend aus den einzelnen Items kodiert. Jedes Item in einer Itemmenge bekommt den dazugehörigen Code aus dieser zugewiesen. Da aber nur die Komplexität der Itemmengen in der Codetabelle interessant ist, wird die Kodierung des Modells vernachlässigt. Die zu lösende Aufgabe lässt sich dann so formulieren:

Problem 2 (minimales $\mathcal{CS}$). Gegeben sei eine Menge von Items $\mathcal{I}$, eine Datenbank $\mathcal{DB}$ über $\mathcal{I}$, eine Coverfunktion und eine Menge von Kanditaten $\mathcal{F}$. Finde das kleinste $\mathcal{CS} \subseteq \mathcal{F}$, so dass die daraus bestehende Codetabelle L($\mathcal{DB}$,CT) minimiert. Bei den Kandidaten handelt es sich um alle Teilmengen aus $\mathcal{I}$ mit einem *minSupport* von 1.

Vorgehen

Der Algorithmus 5.1 veranschaulicht das Vorgehen. Zuerst wird eine Standardcodetabelle aus $\mathcal{DB}$ erzeugt. Diese besteht aus allen einzelnen Items aus $\mathcal{I}$. Diese werden mit Codes versehen und dann $\mathcal{DB}$ mit dieser Tabelle kodiert. Danach werden die Kandidaten nach und nach in die Tabelle eingefügt und überprüft, ob die neue Tabelle $\mathcal{DB}$ besser kodiert. Ist dies der Fall, wird das eingefügte Item aus $\mathcal{F}$ beibehalten, ansonsten wird es verworfen. Die Kandidaten werden absteigend nach Häufigkeit und Größe (*Standard Candidate Order*) eingefügt. Die Items in der Codetabelle werden dann absteigend nach Größe und Häufigkeit (*Standard Cover Order*) sortiert. Wenn nun für die Transaktionen *cover* aufgerufen wird, wird die Codetabelle von oben nach unten durchlaufen. Falls die Transaktion das gerade betrachtete Item enthält, wird es aus der Transaktion gelöscht und das Item in das Resultat von *cover(CT,t)* eingefügt. Sobald die Transaktion leer ist wird die nächste genommen. Wurden alle Transaktionen behandelt ersetzt man alle Items aus den Coverfunktionen mit ihren Codes. Es wird geprüft ob eine bessere Kompression dadurch erreicht wird.

Algorithmus 5.1 Krimp nach [1]

Eingabe: $\mathcal{DB}, \mathcal{F}$
Ausgabe: CT

```
CT ← Standardcodetabelle(DB)
F_0 ← F in Standard Candidate Order
for ∀F ∈ F_0 \ I do
    CT_c ← (CT ⋃ F) in Standard Candidate Order
    if L(DB, CT_c) < L(DB, CT) then
        CT ← CT_c
    end if
end for
return CT
```

Als Ausgabe hat man zum Schluss die CT welche $\mathcal{DB}$ am besten kodiert.

5.2 StreamKrimp

Der im vorherigen Unterkapitel vorgestellte Algorithmus KRIMP lässt sich nun nach [1] relativ einfach für Datenströme erweitern. Die schon eingeführten Codetabellen können wir übernehmen und müssen lediglich dazu speichern, in welchen Zeitpunkten des Stroms sie gelten.

Datenströme und Verteilungen

Schauen wir uns einen Datenstrom $\mathcal{S}$ an, der aus aufeinanderfolgenden Transaktionen besteht, also $\mathcal{S} = \{t_1, t_2, t_3 \ldots\}$. Betrachtet man die Vorkommnisse eines Items im Strom, zum Beispiel in einem Supermarkt, fällt einem auf, dass sich die Häufigkeit in einigen Teilen des Stroms (saisonelle Abhängigkeit) unterscheidet. Im Sommer wird zum Beispiel mehr Grillkohle als im Winter verkauft. Man kann also sagen, dass die Verteilung der Items im Strom variiert. Um also Veränderungen im Strom zu entdecken, muss man die Teile eines Stroms finden, die verschiedene Verteilungen aufweisen. Das Problem lässt sich also zusammenfassen:

Problem 3. Gegeben sei ein Datenstrom $\mathcal{S}$, der aus aufeinanderfolgenden Subströmen $\{S_1, S_2, S_3 \ldots\}$ besteht. S(i,j) bezeichnet den Anfangs- und Endpunkt eines Substroms. Nun wollen wir alle Subströme finden, sodass:

- S_i besteht aus der Verteilung P_i aus $\mathcal{P}(\mathcal{I})$
- $\forall i \in \mathbb{N}$:P$_i$ $\neq$P$_{i+1}$

Die Frage ist nun, wie man eine Verteilung P_i entdeckt und wie man einen Wechsel von P_i zu P_{i+1} entdeckt. Die Darstellung von einem Wechsel findet mithilfe der Codetabellen statt. Für jeden Substrom wird eine Codetabelle kreiert. Die Erstellung einer neuen Codetabelle deutet einen Wechsel der Verteilung an.

Da nur endliche Subströme betrachtet werden, können wir diese wie im vorherigen Kapitel als statische Datenbank behandeln.

Definition 5.2.1 (Bitgröße eines Stroms). Sei S ein endlicher Datenstrom über $\mathcal{I}$ und CT eine code-optimale Codetable für $\mathcal{S}$. Dann ist die totale Größe des kodierten Stroms

$$\text{L(CT,S)} = \text{L(S} \mid \text{CT)} + \text{L(CT} \mid \text{S)}.$$

Codetabelle im Strom finden

Es stellt sich die Frage, wie groß der Substrom sein muss, um mit Sicherheit eine passende Codetabelle zu finden. Dafür nutzt man eine wichtige Eigenschaft von Verteilungen

aus. Falls der Datenstrom $\mathcal{S}$ von nur einer Verteilung Q abstammt, nähern sich die Werte P(X | $\mathcal{S}$(1,n)) ihrem wahren Wert an.

Definition 5.2.2 (Konvergenz von Codetabellen). Sei der Strom $\mathcal{S}$ abstammend von der Verteilung Q aus $\mathcal{P}(\mathcal{I})$, dann gilt

$$\forall\, X \in \mathcal{I}\text{: } \lim_{n\to\infty} P(X|S(1,n)) = Q(X).$$

Daraus kann man eine wichtige Eigenschaft von Codetabellen ableiten: Codetabellen konvergieren. Weiter kann man daraus schliessen:

$$\forall\, k \in \mathbb{N} : \lim_{n\to\infty} |\mathrm{CT}_n\ (\mathrm{S}(1,n))\ \text{-}\ \mathrm{CT}_{n+k}(S(1,n))| = 0$$

Wenn der Strom also aus einer Verteilung besteht, ist es egal, ob die Codetabelle aus den ersten 1000 oder den ersten 100000 Transaktionen gebildet wird. Wir benötigen also nur einen Teil, der groß genug ist, also einen genügend großen „Kopf“ des Stroms.
Wir können nun mit diesen Erkenntnissen einen Algorithmus angeben, um eine Codetabelle im Strom zu finden. Die Idee dahinter ist relativ simpel. Da einzelne Transaktionen die Verteilung im Strom nicht beeinflussen, werden Blöcke von Transaktionen betrachtet. Meist wird hier eine Blocklänge der Größe $|\mathcal{I}|$, also der Anzahl der unterschiedlichen items, gewählt. Mithilfe des ersten Blocks wird eine Codetabelle erstellt und gespeichert. Um sagen zu können, dass diese Codetabelle gut genug ist, wird ein weiterer Block von Transaktionen angehängt und eine neue Codetabelle erzeugt. Mithilfe dieser beiden Codetabellen wird dann die *Improvement Rate* ausgerechnet.

Definition 5.2.3 (Improvement Rate). Zur Prüfung einer Codetabelle auf Konvergenz wird die *Improvement Rate* benutzt:

$$\mathrm{IR} = \frac{|L(S(1,n),CT_n) - L(S(1,n),CT_{n+k})|}{L(S(1,n),CT_n)}$$

Falls IR kleiner dem voreingestellten Wert ist, also die zweite Codetabelle den Kopf des Stroms nicht besser kodiert, wird die erste Codetabelle akzeptiert. Falls nicht wird die zweite Codetabelle als aktuelle genommen und der Vorgang solange wiederholt, bis IR kleiner dem voreingestellten Wert wird. Dazu wird für die Codetabellen eine Laplacekorrektur vorgenomen damit jede Transaktion kodiert werden kann.

Algorithmus 5.2 stellt den Vorgang zur Erstellung einer Codetabelle im Strom dar. Mit der Variable *offset* ist es möglich an jedem Punkt im Strom anzufangen.

Algorithmus 5.2 FindCodetableOnStream nach [1]

Eingabe: $\mathcal{S}, offset, blockSize, minSupport, maxIR$
Ausgabe: CT

```
numTransactions ← blockSize
CT ← KRIMP(S(offset, offset + numTransactions), minsup)
ir ← ∞
while ir > maxIR do
    numTransactions ← numTransactions + blockSize
    newCT ← KRIMP(S(offset, offset + numTransactions), minsup))
    ir ← ImprovementRate(CT, newCT)
    CT ← newCT
end while
return CT
```

Veränderungen in der Verteilung entdecken

Nachdem nun eine Codetabelle für den Kopf des Stroms gefunden wurde, ist es interessant zu erfahren, wann ein Wechsel der Verteilung stattfindet. Nachdem eine Codetabelle für den Anfang des Stroms generiert wurde, wird eine weitere Codetabelle erzeugt. Zwischen den beiden Codetabellen wird die *Code table Difference* CTD berechnet, die angibt, um wieviel Prozent die neue Codetabelle den Strom besser kodiert als die vorherige. Wenn die CTD den gewählten Wert überschreitet, also die neue Codetabelle den gerade betrachteten Teilstrom zum Beispiel um 10% besser kodiert als die vorherige, haben wir einen Wechsel in der Verteilung gefunden. Die neu erstellte Codetabelle wird als die aktuelle betrachtet und dieser Vorgang wiederholt sich bis zum Ende des Stroms.

Definition 5.2.4 (Code Table Difference CTD). Gegeben sei eine Codetabelle CT_1 und eine Codetabelle CT_2, wobei CT_2 für den gerade betrachteten Teilstrom S_2 generiert wurde. Die Code Table Difference

$$\text{CTD} = \frac{|L(S_2,CT_1) - L(S_2,CT_2)|}{L(S_2,CT_2)}$$

gibt genau wie die *improvement Rate* an, um wieviel Prozent CT_2 den neuen Kopf des Stroms besser kodiert als die letzte Codetabelle CT_1.

Wenn man alles zusammenfügt, ergibt sich folgender Algorithmus 5.3:

Es wird zunächst eine Codetabelle für den Kopf des Stroms erstellt. Danach wird versucht, so viele Blöcke wie möglich zu überspringen. Dafür wird ein statistischer Test verwendet. Bevor der Kopf, der für eine neue CT benutzt wurde, entlassen wird, werden aus diesem Block zufällige Blöcke der Größe $|\mathcal{I}|$ ausgewählt. Es wird dieser Block

Algorithmus 5.3 StreamKrimp nach [1]

Eingabe: $\mathcal{S}, minSupport, blocksize, maxIR, leaveOut, minCTD$

Ausgabe: CTS

```
i ← 1
CTS_i ← FindCodeTableOnStream(S, 0, blockSize, minSupport, maxIR)
pos ← CTS_i.endPosition
while pos < sizeOf(S) do
    pos ← SkipBlocks(S, CTS_i, pos, blockSize, leaveOut)
    candidateCT ← FindCodeTableOnStream(S, pos, blockSize, minSupport, maxIR))
    if CTD(S, CTS_i, candidateCT) >= minCTD then
        i ← i + 1
        CTS_i ← candidateCT
        pos ← candidateCT.endPosition
    elsepos ← blockSize
    end if
end while
return CTS
```

kodiert und ein prozentualler Anteil (hier der Parameter *leaveOut*) abgeschnitten. Wenn nun der nächste betrachtete kodierte Block im Strom nicht siginifikant abweicht, wird angenommen, dass er zu der zuletzt erstellten Codetabelle gehört. Wenn ein Block nicht übersprungen werden kann, wird eine neue Codetabelle erstellt und danach CDT ausgerechnet. Wird der voreingestellte Wert überschritten haben wir einen Wechsel in der Verteilung entdeckt.

5.3 Eigenimplementation

Für diese Bachelorarbeit wurde der Algorithmus in Java implementiert. Hierfür wurde KRIMP an einigen Stellen verändert, um zu prüfen, ob dadurch eine schnellere Laufzeit erreicht wird ohne an Qualität zu verlieren. Zunächst einmal werden weniger Kandidaten in Betracht gezogen als im Original. Im ursprünglichen KRIMP werden alle möglichen Teilmengen, die einen *minSupport* von Eins erfüllen, als Kandidaten genommen. Im Falle des Datensatzes *Adult* sind das zum Beispiel 58461763 Mengen, die überprüft werden. Damit wird eine Kompression von 24,4% erreicht. Hier wurden mit FPGROWTH verschiedene Anzahlen an Kandidaten mit einem absteigenden *minSupport* gesammelt und die Kompression überprüft. Die Kandidaten wurden auch nur bezogen auf den gerade betrachteten Teilstrom generiert. Im Original werden alle möglichen Kandidaten sofort aus $\mathcal{I}$ generiert. Dass der *minSupport* absteigend ist, erklärt sich durch die Implementation von FPGROWTH. Hierfür wurde die Implementation von RapidMiner [17] be-

nutzt. Gibt man an, dass 1000 häufige Mengen mit einem *minSupport* von 0,95 ausgegeben werden sollen, die jedoch nicht gefunden werden, wird der *minSupport* automatisch verkleinert und weitere Mengen gesucht. Dieser Vorgang wird solange wiederholt, bis die gewünscht Mindestanzahl an Mengen gefunden wurde. Die Idee dahinter ist folgende: Die häufigen Mengen werden in der *Standard Candidate Order* nach und nach eingefügt, wenn eine Codetabelle erstellt werden soll. Dabei werden die Mengen mit einem hohen *Support* von zum Beispiel 0,9 zuerst betrachtet. Daran ändert sich nichts, wenn man sich mit FPGROWTH alle Mengen mit einem *minSupport* von beispielsweise 0,7 ausgeben lässt. Wenn nun eine Codetabelle mit KRIMP erstellt wird, könnte es sein, dass die maximale Kompression schon nach 5000 getesteten Mengen erreicht ist. Im Falle von *Adult* werden dann weitere 58456763 Mengen überprüft, obwohl dies nicht notwendig ist. Lässt man sich mit FPGROWTH nun die häufigsten 8000 Mengen ausgeben, werden weniger Mengen unnötig betrachtet, wodurch viel Zeit gespart wird. Es kommt die Frage auf, was denn nun passiert, wenn die maximale Kompression erst nach 10000 betrachteten Mengen erreicht wird. Dies könnte natürlich vorkommen, jedoch werden mit Betrachtung der ersten übereinstimmenden 8000 Mengen die „wichtigsten“, also die Mengen die die Datenbank am besten beschreiben auch betrachtet. Es ist nicht relevant für den Betrachter, ob die nichthäufigen Mengen, die überbleiben nachdem man die Transaktionen mit den Kandidaten mit dem größten *Support* gedeckt hat, aus einzelnen Items oder aus Itemmengen der Größe zwei bestehen. Natürlich könnte die Kompression darunter leiden, jedoch zeigen die Experimente, dass dies nicht der Fall ist.

Einhergehend mit dieser Änderung wurde die Erstellung der Codetabellen etwas verändert. Im Original werden die Kandidaten nach der *Standard Candidate Order* sortiert. Darauf wird die Standartcodetabelle genommen und der erste Kandidat wird eingefügt. Es wird geprüft, ob die neue Codetabelle die Datenbank besser kodiert als die vorherige, in diesem Fall die Standardcodetabelle. Ist dies der Fall, wird die neue Codetabelle mit dem hinzugefügten Kandidaten als aktuelle Tabelle genommen. Darauf wird der nächste Kandidat eingefügt und wieder die Kompression verglichen. Dieser Vorgang wird für jeden Kandidaten durchgeführt. Für den Datensatz *Adult* wird die Datenbank 58456763 mal durchgegangen. In der Implementation für diese Arbeit werden die Kandidaten mit den einzelnen Items in einer Codetabelle zusammengelegt und dann nach der *Standard Candidate Order* sortiert. Nun wird für jede Transaktion die Codetabelle angefangen beim obersten Item durchgegangen und geschaut, ob das Item die Transaktion deckt. Ist dies der Fall, wird das Item aus der Transaktion gelöscht, die Häufigkeit des Items um eins erhöht und mit der restlichen Transaktion fortgefahren, bis sie leer ist. Somit wird die Datenbank nur noch einmal durchgegangen. Die Codetabelle kann unter Umständen auch groß sein (je nachdem wieviele Kandidaten man benutzt), jedoch muss

sie nicht zwangsläufig komplett durchlaufen werden für jede Transaktion.

Beide Änderungen wurden mit dem Ziel vorgenommen, die Zeit für die Auswahl der Codetabelle drastisch zu senken. In den Experimenten ist dies für Datensätze mit sehr vielen Kandidaten auch der Fall. Und dennoch wird eine vergleichbare Kompression erreicht. Da im Original für den Datensatz *Mushroom* fast vier Stunden benötigt werden bei der Erstellung der Codetabelle, ist es offensichtlich, dass dieser Vorgang nicht unbedingt für Stöme geeignet ist. Wenn auch nur fünf Codetabellen erstellt werden müssen, könnte die Abarbeitung des Stroms bis zu 20 Stunden dauern. Ausserdem ist zu beachten, dass die Kandidaten aus einer Itemmenge aus 119 verschiedenen Items erstellt wurden. Daraus resultierten 5574930437 Kandidaten. Nimmt man einen Supermarkt mit tausenden von Items muss man die Größe der Kandidaten nicht einmal ausrechnen, um zu sehen, dass die Anzahl um ein vielfaches überstiegen wird.

Zuletzt wurde der Algorithmus mit einer anderen Kodierungsart für die Items implementiert. Im Original wird das *minimum description length principle* angewandt. Dies ist allerdings ein theoretischer Ansatz. Hier wurde ein praktischer Ansatz, nämlich die Huffman-Kodierung, implementiert. Dieser Ansatz wird durch Experimente für KRIMP getestet. Der praktische Ansatz wäre nützlich um die Datenbank komplett zu rekonstruieren. Bei dem theoretischen Ansatz ist dies nicht möglich. Die Huffman-Kodierung ist gerade für die Kompression interessant, da er nicht ganz so effizient ist wie der theoretische Ansatz. Die Huffman-Kodierung ist an das *minimum description length principle* angelehnt, indem die häufigsten Items den kürzesten Code erhalten. Jedoch gibt es einen wichtigen Unterschied. Im theoretischen Ansatz bekommen Items mit der selben Häufigkeit einen Code der selben Länge. Dies ist bei Huffman nicht der Fall. Da es sich um eine binäre Baumstruktur handelt, ist es nicht möglich für drei oder mehr Items mit derselben Häufigkeit Codes der gleichen Länge zu erzeugen. Man bekommt also verschieden Lange Codes. Da bei KRIMP nur die Länge der Codes betrachtet wird ist es interessant zu sehen wie sich eine Kodierung mit Huffman auf die Kompression auswirkt. Zudem soll der praktische Ansatz auf die Anwendbarkeit auf STREAMKRIMP überprüft werden. Dafür muss zuerst eine Entscheidung bezüglich Items in einer Codetabelle mit einer Häufigkeit von 0 getroffen werden. Eine Laplacekorrektur ist bei der Huffman-Kodierung ein grober Ansatz. Dadurch werden auch alle Items mit der Häufigkeit 0 in die Baumstruktur der Huffman-Kodierung eingepflegt. Die Länge aller Bitcodes der Items wird dadurch mit jedem neu eingefügten Item erhöht. Da sich dadurch der Speicherverbrauch erhöht, ist dieser Ansatz nicht ideal für eine praktische Anwendung. Stattdessen wird der Ansatz gewählt, den Items die keinen Code haben den Code des Items mit dem niedrigsten *usage* zu vergeben. Die Implementierung für die Huffman-Kodierung wurde von

www.developer.com [1] entnommen und für die Implementation angepasst. Die Experimente für StreamKrimp wurden mit dem praktischen Ansatz ausgewertet.

[1] http://www.developer.com/java/other/article.php/3603066/Understanding-the-Huffman-Data-Compression-Algorithm-in-Java.htm

6

DATENSÄTZE

6.1 Evaluierung

STREAMKRIMP ist ein Algorithmus, der Datenströme und die damit einhergehenden Probleme effizient verarbeitet. Die resultierenden Codetabellen geben einen übersichtlichen Einblick über die häufigen Mengen im Datenstrom. Es werden vor allem nicht alle häufigen Mengen damit ausgegeben, sondern nur eine Auswahl der interessantesten, die den Teilstrom am besten beschreiben. Wie man im späteren Kapitel an den Experimenten sieht, ist der Speicherverbrauch von STREAMKRIMP sehr gering. Dank der entdeckten Konvergenzeigenschaft müssen viele Teile des Stroms nicht gespeichert werden. Es werden einfach Blöcke die zu einer Codetabelle gehören übersprungen. Die Bearbeitungszeit ist dann sehr gering, da die Prüfung, ob ein Teil eines Stroms zur letzten Codetabelle gehört, schneller zu bewältigen ist als die Erstellung einer neuen Codetabelle. Erst wenn ein Wechsel der Verteilung stattfindet, wird eine neue Codetabelle erzeugt. Der Wechsel der Verteilung ist durch die Codetabellen auch sehr übersichtlich. Man muss lediglich den Inhalt einer Codetabelle mit der vorherigen vergleichen, um die Unterschiede zu erkennen. Da die Blockgröße relativ gering ist und eine Codetabelle im besten Fall aus einem Block erstellt wird, ist die Übersicht garantiert.

6.2 Datensätze

Für die Experimente wurden Datensätze der UCI-library benutzt. Bei den Datensätzen handelt es sich um folgende:

Mushroom [1] Dieser Datensatz besteht aus 8124 Transaktionen und 119 verschiedenen einzelnen Items.

Adult [2] Dieser Datensatz besteht aus 48842 Transaktionen und 97 verschiedenen einzelnen Items.

Led7 [3] Dieser Datensatz besteht aus 3200 Transaktionen und 24 verschiedenen einzelnen Items.

Nursery [4] Dieser Datensatz besteht aus 12960 Transaktionen und 32 verschiedenen einzelnen Items.

Die Datensätze mussten für die Umsetzung angepasst werden. Hierbei handelt es sich um CSV-Dateien in der jede Zeile die Form $(item_1, item_2, item_3, \ldots, item_n)$ hat. FPGROWTH nimmt als Eingabe eine Tabelle. Die Anzahl der Zeilen ist gleich der Anzahl der Transaktionen. Es gibt für jedes Item eine Spalte. Wenn ein Item in einer Transaktion vorkommt muss in der jeweiligen Spalte der Wert True gesetzt werden. Dafür mussten die einzelnen Items in den Transaktion in die Form (Item, true) oder (Item, false) umgeformt werden, um dann in eine Tabelle für FPGROWTH übergeben zu werden.

6.3 Experimente

Prüfung der Eigenimplementation

Die Experimente wurden durchgeführt mit folgenden Spezifikationen:
Intel(R) Core(TM)2 Quad CPU Q9550 @ 2,83GHz 2,83GHz und 4,00 GB RAM.

Die Versuche wurden jeweils für verschiedene *minSupports* durchgeführt. Es wurden bei dem Datensatz *Adult* die ersten 10000 Transaktionen ausgewertet. Da in den Experimenten für STREAMKRIMP Blöcke von höchstens 119 Items gewählt wurden, ist es nicht nötig die kompletten 46000 Transaktionen auszuwerten. Deshalb wurde die Größe auf 10000 beschränkt. Bei den Datensätzen *Mushroom* und *Led7* wurden alle Transaktionen genommen (8124 und 3200 Transaktionen).

[1] http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.data
[2] http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data
[3] http://www.sgi.com/tech/mlc/db/led7.all
[4] http://archive.ics.uci.edu/ml/machine-learning-databases/nursery/nursery.data

Die folgenden Tabellen illustrieren die Ergebnisse bei Benutzung der Huffman-Kodierung (die Werte wurden jeweils gerundet):

minSupport	Kompressionsrate	Zeit
0,9	31%	3s
0,8	31%	3s
0,7	31%	6s
0,6	31%	5s
0,5	31%	5s
0,4	31%	5s
0,3	36%	7s
0,2	51%	2m40s
0,1	60%	39m27s

Tabelle 6.1: Ergebnisse für den Datensatz *Mushroom*

In [1] wurden für diesen Datensatz 3 Stunden und 40 Minuten benötigt. Es wurde eine Kompressionsrate von 20,6% erreicht. Somit ist eine deutliche Verbesserung in der Laufzeit und Kompression zu beobachten.

minSupport	Kompressionsrate	Zeit
0,9	27%	2m20s
0,8	31%	2m30s
0,7	31%	2m30s
0,6	31%	2m30s
0,5	31%	2m30s
0,4	31%	2m30s
0,3	31%	2m40s
0,2	31%	2m40s
0,1	31%	2m50s

Tabelle 6.2: Ergebnisse für den Datensatz *Adult*

In [1] wurden für diesen Datensatz 2 Minuten und 25 Sekunden benötigt. Es wurde eine Kompressionsrate von 24,4%erreicht. Die Kompressionsrate ist zwar ein wenig besser, jedoch kann man diesen Schluss nicht über die Laufzeit ziehen. Es wurden hier im Schnitt 2 Minuten und 30 Sekunden benötigt, obwohl nur 10000 Transaktionen genommen wurden.

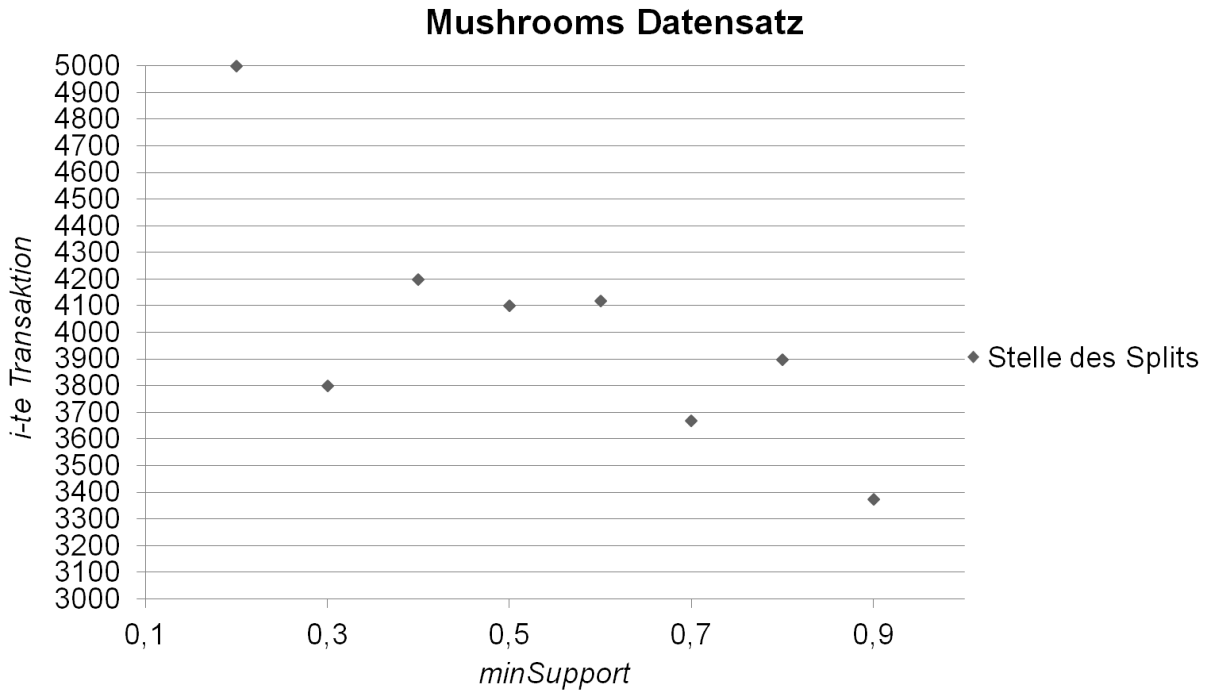
minSupport	Kompressionsrate	Zeit
0,9	31%	1s
0,8	36%	1s
0,7	38%	1s
0,6	41%	1s
0,5	44%	1s
0,4	50%	1s
0,3	56%	2s
0,2	56%	2s
0,1	55%	2s

Tabelle 6.3: Ergebnisse für den Datensatz *Led7*

In [1] wurden für diesen Datensatz 5 Sekunden benötigt. Es wurde eine Kompressionsrate von 28,6% erreicht. Die Laufzeit und die Kompressionsrate wurden verbessert.
Die Unterschiede in den Kompressionsraten sind teilweise sehr hoch. Um diesen Sachverhalt aufklären zu können, wäre es von Vorteil die Codetabellen mit denen aus [1] zu vergleichen. Dies wäre jedoch ein Ansatz für weitere Analysen.

Experimente für StreamKrimp

Der veränderte Teilalgorithmus KRIMP wurde nun auch in Form von STREAMKRIMP verwendet. Wie in [1] wurden folgende Parameter für den Datensatz *Mushroom* verwendet: minCTD = 10%, minIR = 2%, leaveOut = 0,01 und blockSize = 119.



minSupport	Zeit
0,9	1m4s
0,8	1m6s
0,7	1m10s
0,6	1m30s
0,5	2m30s
0,4	6m10
0,3	17m40s
0,2	1h23m11s

Tabelle 6.4: Benötige durchschnittliche Zeiten für den Datensatz *Mushroom*

In[1] wurde der *concept drift* immer in der Mitte des Streams, also ungefähr an der 4000. Transaktion erkannt. Dort findet auch der tatsächliche Wechsel der Verteilung statt. Wie man an dem Graphen erkennt, wurde der *concept drift* in der veränderten Version nicht

an der selben Stelle erkannt. Jedoch ist die Abweichung bei einem *minSupport* zwischen 0,3 und 0,8 nicht sehr stark. Allerdings wurde die Laufzeit stark verbessert. Man muss beachten, dass die Erstellung einer Codetabelle für den Datensatz nach [1] fast 4 Stunden benötigt. Da mindestens zwei Codetabellen erstellt werden, ist zu sehen, dass die veränderte Version deutlich schneller ist. Es wurden allerdings nur 15 Blöcke im Schnitt übersprungen. In [1] waren es 44. Somit wurden mehr Codetabellen erstellt und überprüft. Bei einer weiteren Überprüfung und Verbesserung der Anzahl an übersprungenen Blöcken könnte die Laufzeit weiter minimiert werden (siehe Fazit).

Die Datensätze *Adult* und *Nursery* dagegen ergaben nicht die erwünschten Resultate. Es konnte kein Zusammenhang zwischen den Parametern und der Ausgabe erstellt werden. Die *concept drifts* wurden nicht an den richtigen Stellen erkannt und es wurden viel mehr Codetabellen gebildet als nötig. Um hier weitere Schlüsse und Lösungsansätze ziehen zu können, müssten diese Datensätze weiter untersucht werden. Mögliche Ansätze werden im Fazit erläutert.

7

FAZIT

7.1 Fazit

Die Aggregation häufiger Mengen ist ein wichtiges und breit untersuchtes Themengebiet. Die Analyse der gesammelten Informationen unterstützt Anwender bei der Auswahl weiterer Vorgehensweisen. Marktleiter zum Beispiel können mit entsprechenden Informationen ihre Produktpositionierung anpassen. Marktanalysen sind ohne die passenden Informationen unvorstellbar.

Durch die weiter voranschreitende Technologie werden Daten immer schneller und in größeren Mengen gesammelt. Es ist daher wichtig, diese Daten so schnell wie möglich zu verarbeiten, um sie nicht unnötig lange speichern zu müssen. Viele Algorithmen wurden bisher vorgeschlagen, um die Probleme, die bei der Verarbeitung von Datenströmen entstehen, zu lösen. Dabei lösen die meisten Algorithmen nur Teilprobleme. Eine Auswahl an Algorithmen wurde in dieser Arbeit vorgestellt. Gerade für Märkte und Marktanalysen ist es wichtig *concept shifts* und *concept drifts* rechtzeitig zu erkennen, um darauf reagieren zu können. Ein Algorithmus der dies effizient tut ist STREAMKRIMP.

Er wurde für diese Arbeit in veränderter Form implementiert. Ziel war es zu prüfen, ob die Laufzeit verringert werden kann ohne an Qualität zu verlieren. Die Experimente zeigten, dass die Laufzeit des ersten Teils des Algorithmus verbessert werden konnte, ohne an Qualität zu verlieren. Die Kompressionsraten waren bei einer niedrigeren Laufzeit vergleichbar. Die Anwendung auf einen Datenstrom hingegen waren nicht so erfolgreich. Lediglich für einen Datensatz wurden vergleichbare Ergebnisse erzielt wie in [1]. Bei den

anderen Datensätzen gab es keinen sichtlichen Zusammenhang zwischen den Eingabeparametern und der Ausgabe.

Für eine praktische Anwendung müssten die Implementierte Version und die Datensätze weiter untersucht werden. Da bei dem Datensatz *Mushroom* vergleichbare Ergebnisse erzielt wurden, müsste die Struktur des Datensatzes mit der Struktur der Datensätze *Nursery* und *Adult* verglichen und analysiert werden. Die daraus resultierenden Informationen müssen dann mit dem Aufbau des Algorithmus in Verbindung gebracht werden.

Möchte man den praktischen Kodierungsansatz beibehalten, müsste die Wahl der Codes bei Items mit der Häufigkeit 0 überdacht werden. Im Gegensatz zum Original wurden weniger Blöcke übersprungen. Die Konsequenz daraus war, dass mehr Codetabellen erstellt werden mussten. Dieser Vorgang hat zusätzlich Zeit beansprucht. Die Auswirkungen einer anderen Kodierungsart müssten dann auch auf die Erstellung der Codetabellen und den daraus resultierenden Einfluss auf die Experimente untersucht werden.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ALGORITHMENVERZEICHNIS

LITERATURVERZEICHNIS

[1] MATTHIJS VAN LEEUWEN, *Patterns that Matter*.
SIKS Dissertation Series No. 2010-01 2010.

[2] T. CALDERS, N. DEXTERS, B. GOETHALS, *Mining Frequent Itemsets in a Stream*.
2007 in: ICDM

[3] HAN, J. IAWEI, J. IAN PEI, YIWEN YIN, *Mining frequent patterns without candidate generation*.
In: Proceedings of the 2000 ACM SIGMOD international conference on Mana-gement of data, SIGMOD '00, Seiten 1–12, New York, NY, USA, 2000. ACM.

[4] AGRAWL, R. AKESH ,R. SRIKANT, *Patterns that Matter*.
In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, Seiten 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[5] CHI, Y., WANG, H., YU, P.S., MUNTZ, R.R., *Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window.*.
In: Proceedings of 2004 IEEE International Conference on Data Mining, Brighton, Seiten 59–66 (2004)

[6] JIANG, N., GRUENWALD, *CFI-Stream: Mining Closed Frequent Itemsets in Data Streams.*.
In: Proceedings of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, Seiten 592–597 (2006)

[7] SHOW-JANE YEN, YUE-SHI LEE, CHENG-WEI WU, AND CHIN-LIN LIN,
An Efficient Algorithm for Maintaining Frequent Closed Itemsets over Data Stream.
B.-C. Chien et al. (Eds.): IEA/AIE 2009, LNAI 5579, pp. 767–776, 2009.

[8] H. MANNILA, H. TOIVONEN, A. I. VERKAMO, *Discovery of Frequent Episodes in Event Sequences*.
Data Mining and Knowledge Discovery 1, Seiten 259–289 (1997)

[9] RAJESH RAWAT, ASST. PROF. NIDHI JAIN, *A Survey on Frequent ItemSet Mining Over Data Stream*.
International Journal of Electronics Communication and Computer Engineering Volume 4, Issue 1, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

[10] M. M. GABER, A. ZASLAVSKY, S. KRISHNASWAMY, *Mining Data Streams: A Review*.
SIGMOD Record, Vol. 34, No. 2, June 2005

[11] CHRIS GIANNELLA , JIAWEI HAN, JIAN P EI, XIFENG Y AN, PHILIP S. Y U,
Mining Frequent Patterns in Data Streams at Multiple Time Granularities.
suchen

[12] NAN JIANG, LE GRUENWALD, *Research Issues in Data Stream Association Rule Mining* .
SIGMOD Record, Vol. 35, No. 1, Mar. 2006

[13] GURMEET SINGH MANKU, RAJEEV MOTWANI, *Approximate Frequency Counts over Data Streams*.
Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

[14] G. MAO, X. WU, X. ZHU, G. CHEN, C. LIU, *Mining maximal frequent itemsets from data streams*.
Journal of Information Science 2007 ;33: 251, originally published online Mar 23, 2007.

[15] G. MAO, X. WU, X. ZHU, G. CHEN, C. LIU, *Max-FISM: Mining (recently) maximal frequent itemsets over data streams using the sliding window model*.

[16] MARCIN SKIRZYNSKI, *Subspace-Clustering mit parallelen häufigen Mengen*.
Technische Universität Dortmund, Lehrstuhl 8 der Fakultät für Informatik, März 2012

[17] MIERSWA, INGO, M. WURST, R. KLINKENBERG, M. SCHOLZ, T. EULER,
YALE: Rapid Prototyping for Complex Data Mining Tasks. In: U NGAR, LYLE, M ARK CRAVEN, D IMITRIOS GUNOPULOS und TINA ELIASSI-R AD (Herausge-ber): KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Know-ledge discovery and data mining, Seiten 935–940, New York, NY, USA, August 2006. ACM.

Hiermit bestätige ich, die vorliegende Bachelorarbeit selbständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst zu haben.

Ich bin damit einverstanden, dass Exemplare dieser Arbeit in den Bibliotheken der Universität Dortmund ausgestellt werden.

Dortmund, den 17. Juni 2013

Adrian Skirzynski