Data Mining - Association Rule Mining

Roger Jäggi
jaeggir@student.ethz.ch
ETH Zürich

30. Juni 2006

Inhaltsverzeichnis

Zusammenfassung

Dies ist eine Ausarbeitung zum Thema 'Data Mining - Association Rules Mining' für das Fachseminar **Algorithmen für Datenbanksysteme** von Prof. Kossmann und Prof. Widmayer.

In der Einleitung werden die zwei Begriffe *Data Mining* sowie *Association Rule Mining* definiert und die Problemstellung konkretisiert. Das Problem kann immer in zwei Teile aufgeteilt werden: das Finden von *large itemsets* und die Extraktion der Regeln aus diesen gefundenen Mengen. In dieser Ausarbeitung wird vorallem auf den ersten Teil eingegangen.

Im zweiten Abschnitt wird ein erster Algorithmus zur Bestimmung von Association Rules vorgestellt und analysiert.

Danach wird mit *Apriori* ein etwas anderer Ansatz betrachtet. Abschliessend folgt eine kurze Zusammenfassung.

1 Einleitung

In den folgenden zwei Unterkapiteln werden kurz die Begriffe Data Mining und Association Rules Mining definiert und erläutert.

1.1 Data Mining

In Unternehmen, Behörden und Forschungsprojekten entstehen immer grössere Datenbanken. Als Beispiele seien hier Datenbanken für Lagerbestände, Auftragsdaten, Verkaufs- und Umsatzdaten, Personendaten oder Daten von Messreihen genannt. Ein Teil der Daten wird nur aus rechtlichen Gründen archiviert (zum Beispiel bei Finanzinstituten), aber oft möchte man aus diesen riesigen Datenbeständen auch Rückschlüsse ziehen, um den täglichen Workflow zu optimieren resp. anzupassen.

> "Unter Data Mining versteht man das systematische (in der Regel automatisierte oder halbautomatische) Entdecken und Extrahieren unbekannter Informationen aus grossen Mengen von Daten[1]."

In den Fokus der Unternehmen und deren Marketingabteilungen kam *Data Mining* erst in den letzten zehn Jahren. Entwicklungen im Bereich von Datenbanken aber vorallem auch bei Mehrprozessorsystemen ermöglichten eine bessere Performance und machten Data Mining erst so richtig interessant.

Wie wichtig Data Mining geworden ist, sieht man an der steigenden Anzahl von Kundenkarten wie Cumulus oder Supercard, um zwei Vertreter zu nennen. Während früher das Hauptaugenmerk bei der Kundenbindung (zB. durch Rabattmarken) lag, möchte man heute vorallem Kundenprofile gewinnen und auswerten.

[1] $http://de.wikipedia.org/wiki/Data\_Mining$

1.2 Association Rule Mining

1.2.1 Informale Definition

Eine typische und weitverbreitete Form ist die Warenkorbanalyse. In einer Datenbank sind alle Einkäufe (= Transaktionen) von Kunden gespeichert. Jeder Einkauf besteht natürlich aus mindestens einem Produkt (= Item). Ziel der Warenkorbanalyse ist es nun, Beziehungen zwischen Produkten zu finden. Man möchte also Regeln folgender Art finden:

> "Kauft ein Kunde Milch und Butter, besteht eine grosse Wahrscheinlichkeit, dass er sich auch noch für Brot entscheidet."

Eine der bekanntesten Regeln sagt übrigens aus, dass Kunden, die Windeln kaufen, auch oft noch Bier auf ihrem Einkaufszettel haben (die Daten, welche dieser Regel zugrunde liegen, wurden allerdings mittels Umfragen erfasst).

Diese Regeln dürfen natürlich auch komplizierter sein. Um eine Aussagekraft zu haben, müssen diese Regeln gewisse Bedingungen erfüllen. Zum einen muss ein bestimmter Prozentsatz aller Transaktionen die Produkte A, B und C enthalten sowie ein gewisser Prozentsatz der Einkäufe, welche A und B enthalten, müssen auch C enthalten. Diese beiden zusätzlichen Bedingungen bestimmen die Stärke der Regel sowie die statistische Signifikanz.

Neben der Warenkorbanalyse gibt es noch weitere Formen von Association Rule Mining (z.B. Fehler Identifikation), welche aber im weiteren Verlauf dieser Ausarbeitung nicht betrachtet werden.

1.2.2 Formale Definition

Gegeben sei eine Menge von Literalen $\mathcal{I} = \{i_1, i_2, ..., i_m\}$ genannt *Items*. Sei $\mathcal{D}$ eine Menge von Transaktionen $\mathcal{T}$, wobei jede Transaktion $t \in \mathcal{T}$ wiederum eine Menge von Items ist, so dass $\mathcal{T} \subseteq \mathcal{I}$ gilt.

Eine Association Rule bezeichnet nun eine Implikation der Form

$$\{X \Rightarrow Y \mid X \subset \mathcal{I},\ Y \subset \mathcal{I}\ und\ X \cap Y = \emptyset\}$$

Neben den trivialen Bedingungen $X \subset \mathcal{I},\ Y \subset \mathcal{I}\ und\ X \cap Y = \emptyset$ müssen für eine gültige Regel noch zwei weitere Nebenbedingungen beachtet und erfüllt werden:

1. **Vertrauen**
   Eine Regel $X \Rightarrow i_k$ hält in der Menge der Transaktionen $\mathcal{D}$ mit einem Vertrauensfaktor $0 \leq c \leq 1$ sofern mindestens $c\%$ der Transaktionen, welche X enthalten, auch i_k enthalten.
   Vertrauen ist somit ein Mass für die Stärke einer Regel.

2. **Support**
   Support für eine Regel $X \Rightarrow i_k$ ist definiert als Prozentsatz der Transaktionen in $\mathcal{D}$, welche die Vereinigung $X \cup i_k$ enthalten.
   Support ist also ein Mass für die statistische Signifikanz einer Regel.

Die dritte Klasse von Einschränkungen ist nicht zwingend und kann zusätzlich angegeben werden. Da diese Einschränkungen aber nichts am Algorithmus ändern, werden sie im weiteren Verlauf dieser Ausarbeitung nicht betrachtet und sind nur der Vollständigkeit halber aufgeführt:

3. **Syntaktische Einschränkungen**
   Es ist möglich, die Menge der Items welche in einer Regel vorkommen, einzuschränken. Somit kann zB. spezifiziert werden, dass jede Regel ein Item I_j im Bedingungsteil (Antezedenz) enthalten muss.

   "Ein Kaufhaus hat ein Lockvogelangebot gestartet und ist nun daran interessiert, inwiefern es sich auf andere Produkte auswirkt. Desshalb sind nur Regeln mit dem Lockvogelangebot im Bedingungsteil von Interesse."

Gegeben sei nun eine Menge von Transaktionen $\mathcal{D}$. Man will alle Regeln finden welche mindestens einen minimalen Support *minsup* sowie einen minimalen Vertrauenswert *minconf* aufweisen. *minsup* und *minconf* sind dabei frei wählbare Werte und müssen je nach Situation gesetzt werden.

2 AIS - Ein erster Algorithmus

Der folgende Algorithmus wurde 1993 von Agrawal, Imielinski und Swami [1] vorgeschlagen. Im weiteren Verlauf dieser Ausarbeitung wird der Algorithmus mit **AIS** referenziert.

Die Ausgangskonfiguration besteht aus einer Datenbank in welcher Kunden-Transaktionen gespeichert wurden. Eine Transaktion entspricht dabei einem Einkauf eines Kundens. Die einzelnen Elemente einer Transaktion sind demzufolge die Produkte die gekauft wurden (und im Folgenden Items genannt werden).

2.1 Idee

Wie bereits weiter oben erwähnt, ist der ***Support*** ein Gradmesser der statistischen Signifikanz. Wir sind also nur an Regeln interessiert, welche auch eine genügend grosse Datengrundlage (*minsup*) aufweisen. Um den Algorithmus zu beschleunigen und Speicherplatz zu sparen, muss also versucht werden, Regeln mit geringem Support so früh wie möglich zu verwerfen, damit sie bei weiteren Schritten nicht mehr berücksichtigt werden müssen.

Der Algorithmus kann in folgende zwei Teilprobleme aufgeteilt werden:

1. Generiere alle Kombinationen von Items mit einem Support grösser als der gewählte minimale Support *minsup*. Diese Kombinationen werden im weiteren Verlauf *large itemsets* genannt. Alle anderen Kombinationen dementsprechend *small itemsets*.

2. Generiere nun für ein gegebenes *large itemset*

$$Y = I_1 I_2 ... I_k, k \geqslant 2$$

alle Regeln mit Items aus der Menge $I_1 I_2 ... I_k$. Die linke Seite der Regel besteht aus $(k-1)$ Items, während die rechte Seite dementsprechend aus einem Item besteht. Dies kann auch dahingehend verallgemeinert werden, dass die rechte Seite der Regel aus mehr als einem Item besteht. Für diese Ausarbeitung wird aber darauf verzichtet. Für die neu generierte, provisorische Regel $X \Rightarrow \mathcal{I}_j$ muss nun das *Vertrauen* geprüft werden. Dazu teilt man den *Support* von Y durch den *Support* von X. Falls der Quotient kleiner ist als *minconf* c, kann die Regel verworfen werden.

```
Beispiel:
Y = {i,j,k}, Support Y = 0.3
X = {i,j}  , Support X = 0.4

Regel: {i,j} => {k}

Vertrauen der Regel = 0.3 / 0.4 = 0.75
```

Die Bestimmung der Regeln in Schritt 2 ist einfach. Jede mögliche Regel $X \Rightarrow I_j$ mit $X,\ I_j \in Y$ erfüllt die geforderten Nebenbedingungen und ist somit eine gültige Regel. In Abschnitt 2.3 wird kurz auf die Extraktion der Regeln eingegangen. Das Hauptaugenmerk liegt aber ganz klar in der Bestimmung der *large itemsets*, welche im folgenden Kapitel 2.2 ausführlich erläutert wird.

2.2 Bestimmung der large itemsets

Um die Berechnung von *large itemsets* zu erläutern, müssen einige zusätzliche Begriffe eingeführt werden.

- Gegeben seien zwei *itemset* X und Y wobei $size(Y) = k$.
  $X+Y$ (kann auch als $X \cup Y$ geschrieben werden) wird dann k-Erweiterung (k-extension) von X genannt falls $X \cap Y = \emptyset$.
- Das *frontier set* besteht aus allen *itemsets* welche während eines Durchlaufes erweitert wurden und im nächsten Schritt erneut getestet werden müssen.
- In jedem Durchlauf wird der Support von bestimmten *itemsets* berechnet. Diese Mengen werden *candidate itemsets* genannt und leiten sich aus den Tupeln in der Datenbank und den *itemsets* im *frontier set* ab.

Zu Beginn besteht das *frontier set* nur aus einem Element, der Leeremenge. Für jede gespeicherte Transaktion in der Datenbank wird getestet, ob ein Element des *frontier sets* darin vorkommt. Beim ersten Durchlauf ist diese Bedingung natürlich immer erfüllt, da das *frontier set* ja nur aus der Leeremenge besteht. Für jeden Treffer werden nun rekursiv mit *getCandidateItemset* eine Reihe von möglichen *itemsets* berechnet und ins *candidate itemset* aufgenommen. Wie und

vorallem welche Erweiterungen genau berechnet werden, wird in Abschnitt 2.2.1 diskutiert.

Danach muss für jedes Itemset im *candidate itemset* verifiziert werden, ob es ein *large itemset* ist (und somit zur Resultatmenge gehört) und ob es nötig ist, das itemset noch zusätzlich ins *frontier set* aufzunehmen, da es irrtümlich als zu klein angenommen und desshalb nicht erweitert wurde (mehr dazu im Abschnitt 2.2.2).

Solange das *frontier set* nicht leer, ist werden die oberen beiden Schritte wiederholt.

Die beiden Funktionen $getCandidateItemsets(t, f)$ und $useAsFrontierSet(c)$ werden in den zwei folgenden Abschnitten erklärt.

Algorithm 1 Berechnung aller large itemsets

```
procedure LargeItemsets
    let Large set L = ∅;
    let Frontier set F = ∅;
    while F ≠ ∅ do
        let Candidate set C = ∅;
        for all database tuples t do
            for all itemsets f in F do
                if (t contains f) then
                    C_f = getCandidateItemsets(f, t);
                    for all itemsets c_f in C_f do
                        if (c_f ∈ C) then
                            c_f.count++;
                        else
                            c_f.count = 0;
                            C = C + c_f;
        let F = ∅;
        for all itemsets c in C do
            if (count(c)/dbsize > minsup) then      ▷ dbsize = #transactions
                L = L + c;
            if (useAsFrontierSet(c)) then
                F = F + c;
    end while
end procedure
```

2.2.1 Bestimmung der candidate itemsets

Die Bestimmung des *candidate itemset* ist ein rekursiver Prozess. Es wird angenommen, dass die Items im gegebenen Itemset X geordnet sind. Somit können Duplikate vermieden werden und die Funktion vereinfacht sich.

In einem ersten Schritt werden alle 1-Erweiterung ins *candidate itemset* aufgenommen. In einem zweiten Schritt wird für jede 1-Erweiterung geschätzt ob es ein *large itemset* ist . Ist dies der Fall, wird $getCandidateItemsets$ rekursiv für die Erweiterung aufgerufen.

Algorithm 2 Berechnung des candidate itemsets

```
1: procedure getCandidateItemsets(X : itemset, t : tuple)
2:     // Precondition: items in X are ordered
3:     // I_j be such that ⋁ I_l ∈ X, I_j ≥ I_l;
4:     for all items I_k in t such that I_k > I_j do
5:         Output(XI_k);
6:         if (XI_k is expected to be large) then
7:             getCandidateItemsets(XI_k, t);
8: end procedure
```

Die Schätzung ob ein Set XI_k die geforderte Grösse erreicht oder nicht, ist dabei zentral und wird darum an dieser Stelle ausführlich erläutert.

Angenommen $X + Y$ sei eine k-Erweiterung vom *frontier set* X, welches in x Tupel vorhanden ist (dieser Wert ist bekannt, da X im vorhergehenden Durchlauf berechnet wurde). Angenommen $X + Y$ wird zum ersten mal betrachtet nachdem bereits bekannt ist, dass bereits c Tupel im aktuellen Durchlauf X enthielten. Wenn mit $f(I_j)$ die relative Frequenz eines Items in der Datenbank bezeichnet wird, dann berechnet sich der Support $\overline{s}$ von $X + Y$ als

$$\overline{s} = f(I_1) * f(I_2) * ... f(I_k) * (x - c)/dbsize$$

Dabei ist $(x - c)/dbsize$ der aktuelle Support von X im verbleibenden Teil der Tupel. Diese Schätzung gilt nur unter der Annahme, dass die Items statistisch unabhängig sind. Falls $\overline{s}$ grösser als *minsup* ist, wird die k-Erweiterung als *large itemset* bezeichnet.

Die Arbeitsweise des Algorithmus zeigt sich am Besten an einem Beispiel. Dabei wurde jeweils willkürlich bestimmt ob ein itemset nun ein *large itemset* ist oder nicht.

```
Beispiel:
I = {A, B, C, D, E, F}
frontier set F = {AB}
t = ABCDF

Folgende candidate itemsets werden generiert:

ABC     expected large: continue extending
ABCD    expected small: do not extend any further
ABCF    expected large: cannot be extended any further
ABD     expected small: do not extend any further
ABF     expected large: cannot be extended any further
```

Die Erweiterung ABCDF wurde zB. nicht berücksichtigt weil ABCD als klein angenommen wurde. Ähnliches gilt für ABDF. ABCE und ABE wurden nicht berücksichtigt weil E nicht in t vorhanden ist.

2.2.2 Bestimmung der frontier itemsets

Das *frontier itemset* beinhaltet alle *itemsets*, welche bei der Bestimmung des *candidate itemset* nicht als ein *large itemset* identifiziert wurden, es aber dennoch sind. Demzufolge wurden deren Erweiterungen noch nicht betrachtet. Um Vollständigkeit zu gewährleisten, muss dies im nächsten Schritt getan werden. Dies ist durch die Aufnahme ins *frontier itemset* gewährleistet, da dieses dann sicher nicht leer ist und die Datenbank somit nochmals durchlaufen wird.

2.3 Extraktion der Regeln

Die *large itemsets* wurden im vorhergehenden Schritt berechnet, nun müssen daraus die Regeln extrahiert werden.

Eine mögliche Lösung könnte so aussehen:

Finde für jedes *large itemset* l alle nicht leeren Teilmengen a. Für jede Teilmenge a kann nun eine Regel der Form

$$a \Rightarrow (l \ - \ a)$$

generiert werden. Falls Support(l)/Support(a) $\geqslant$ $minconf$ gilt, ist die Regel gültig und kann ausgegeben werden. Sollen nur Regeln der Form $X \ \Rightarrow \ I_j$ gefunden werden, vereinfacht sich der Vorgang natürlich dementsprechend.

Eine schnelle Implementierung dieses Problems kann zum Beispiel in [3] gefunden werden.

3 Apriori Algorithmus

Der Apriori Algorithmus wurde 1994 von Rakesh Agrawal und Ramakrishnan Srikant entwickelt[2].

3.1 Idee

Das Problem wird wiederum in die gleichen zwei Teilprobleme aufgeteilt welche dann separat gelöst werden können. In der ersten Phase werden demzufolge die *large itemsets* gesucht und in der zweiten Phase die Regeln daraus gebildet. Apriori ist ein Algorithmus für Phase eins, Phase zwei kann wiederum mit einem beliebigen Algorithmus gelöst werden.

3.2 Bestimmung der large itemsets

In einem ersten Schritt werden jene *Items* bestimmt, welche mindestens den minimalen Support *minsup* geniessen. Dazu müssen alle Transaktionen einmal durchlaufen werden und die *Items* gezählt werden. *Items* die in *minsup* % der Transaktionen vorkommen, sind für die weiteren Schritte von Interesse.

Danach beginnt der eigentliche Algorithmus, welcher auch wieder in zwei Phasen abläuft. In jedem Durchlauf wird als Datengrundlage die gefundenen *large*

itemsets des letzten Durchganges genommen. Damit werden dann jeweils die neuen, potentiellen *large itemsets* bestimmt, welche wiederum *candidate itemsets* genannt werden.

Um zu prüfen wie oft die einzelnen Kandidaten in den Tupeln vorkommen, wird die Datenbank gescannt und jedes Tupel mit dem *candidate itemset* verglichen. Dafür wird eine spezielle subset-Funktion angewandt um die Effizienz zu steigern. Danach werden die Statistiken aktualisiert und nicht benötigte Kandidaten aussortiert (jene mit einem Support kleiner als *minsup*).

Diese beiden Schritte werden solange wiederholt bis keine neuen *large itemsets* mehr gefunden werden.

Algorithm 3 Apriori Algorithmus

```
1: L_1 = {large 1 − itemsets};
2: for (k = 2; L_{k−1} ≠ ∅; k++) do
3:     C_k = apriori-gen(L_{k−1}) //New candidates
4:     for all transactions t ∈ D do
5:         C_t = subset(C_k, t);
6:         for all candidates c ∈ C_t do
7:             c.count++;
8:     L_k = {c ∈ C_k | c.count ≥ minsup}
9: Answer = ⋃_k L_k;
```

Der Unterschied zum vorhergehenden Algorithmus liegt darin, welche *candidate itemsets* in einem Durchlauf gezählt werden und wie diese überhaupt generiert werden.

Die Grundidee dahinter ist, dass jede Teilmenge eines *large itemsets* auch wieder gross sein muss. Darum werden die Kandidaten generiert indem sets mit k-1 Items gejoint werden und dabei *small itemsets* aussortiert werden. Dies führt zu einer kleinen Anzahl von Kandidaten. Die *apriori-gen* Funktion, welche die Kandidaten generiert, wird im nächsten Abschnitt vorgestellt.

3.3 Bestimmung der candidate itemsets

Die *apriori-gen* Funktion aus dem letzen Abschnitt nimmt als Input L_{k-1}, die Menge aller *itemsets* der Grösse $k-1$. In zwei Schritten wird L_{k-1} zuerst mit sich selbst gejoint, danach werden alle *itemsets* gelöscht welche eine Untermenge der Grösse (k-1) besitzen, die nicht im Input L_{k-1} vorkommt:

Algorithm 4 apriori-gen Funktion

```
1: procedure APRIORI-GEN(L_k : Itemset)
2:     C_k = Join(L_k, L_k);
3:     for all itemsets c ∈ C_k do
4:         for (k-1)-subsets s of c do
5:             if (s ∉ L_{k−1}) then
6:                 delete c from C_k
7: end procedure
```

Die Arbeitsweise des Algorithmus lässt sich wiederum am Besten an einem Beispiel zeigen.

L ist das gegebene $(k-1)$-*Itemset*, C1 ist die Menge der Itemsets vor dem Löschen und C2 nach dem Löschen von unerwünschten *Itemsets*.

```
Beispiel:
L  = {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}
C1 = {{1 2 3 4}, {1 3 4 5}}
C2 = {1 2 3 4}
```

Korrektheit: Wir müssen zeigen, dass $C_k \supseteq L_k$ gilt.
Wenn wir L_{k-1} mit allen möglichen Items erweitern und dann alle löschen die eine Teilmenge der Grösse (k-1) besitzen, welche nicht in L_{k-1} sind, bleibt ein Superset von allen *itemsets* in L_k.

Der Join entspricht nun dem Erweitern von L_{k-1} mit jedem Item in der Datenbank. Nach dem Join-Schritt gilt also bereits $C_k \supseteq L_k$. Im Löschen-Schritt werden nun nur *itemsets* gelöscht, die Teilmengen der Grösse (k-1) besitzen, welche nicht in L_{k-1} vorkommen. Da die *itemsets* von L_k 1-Erweiterungen von *itemsets* aus L_{k-1} sind, kann einfach eingesehen werden, dass somit auch keine *itemsets* aus L_k gelöscht werden können und L_k damit vollständig ist.

3.4 Die Subset Funktion

Sobald alle möglichen Kandidaten gefunden wurden, muss getestet werden, ob diese auch den nötigen Support besitzen. Da diese Operation für alle Transaktionen einer Datenbank in jedem Durchlauf ausgeführt wird, ist eine effiziente Implementierung von Bedeutung.

Dazu werden die *candidte itemsets* C_k in einem Hash-Tree gespeichert (genauer gesagt in den Blättern). Ein Knoten beinhaltet nun entweder eine Liste von *itemsets* (ein Blatt) oder eine Hash-Tabelle (ein innerer Knoten). In einem inneren Knoten der Tiefe d zeigt jeder Eintrag in der Hash-Tabelle auf einen Knoten der Tiefe d+1.

Ein *itemset* wird eingefügt, indem bei jedem inneren Knoten mit Tiefe d eine Hash-Funktion auf den d-ten Knoten des *itemset* angewandt wird und dem entsprechenden Zeiger gefolgt wird. Alle Knoten werden als Blätter initialisiert und bei Überschreiten eines Grenzwertes in einen inneren Knoten umgewandelt.

Die Subset-Funktion findet nun die im Tupel vorhanden Kandidaten wie folgt. Zuerst wird das Tupel in der Datenstruktur gesucht. Endet die Suche bei einem Blatt, kann einfach geprüft werden, welche Kandiaten tatsächlich im Tupel vorkommen (all jene, die auch einen Eintrag im Blatt haben). Endet die Suche bei einem inneren Knoten, wird die Prozedur rekursiv für jedes Item angewandt, welches nach dem letzten Item i im Tupel kommt.

Von grundlegender Wichtigkeit ist die Tatsache, dass die *itemsets* und die Tupel lexographisch geordnet sind. Ansonsten funktioniert dieses Verfahren natürlich nicht.

4 Zusammenfassung und Ausblick

Der Aufbau der beiden vorgestellten Algorithmen ist ähnlich. Beide bestimmen in einem ersten Schritt die *large itemsets* und extrahieren daraus dann in einem zweiten Schritt alle gültigen Regeln, welche unter den gegebenen Nebenbedingungen möglich sind.

Der Unterschied zwischen den zwei Algorithmen liegt in der Anzahl generierter *candidate itemsets*, welche evaluiert werden müssen.

Der **AIS** Algorithmus erweitert ein *candidate itemset*, wenn mit der Schätzung angenommen werden kann, dass es auch ein *large itemset* ist. Ist die Schätzung allerdings falsch und das *itemset* fälschlicherweise als gross angenommen wird, werden viele zusätzliche *itemsets* berechnet, welche dann alle gespeichert und geprüft werden müssen. Dies führt dazu, dass **AIS** bei kleinen Werten für *minsup* sehr langsam wird.

Der **Apriori** Algorithmus hingegen erstellt zuerst alle möglichen *itemsets* unabhängig von den gespeicherten Transaktionen und entscheidet dann aufgrund der subset-Funktion welche neu erstellten *itemsets* überhaupt minimalen Support erreichen können. Dies führt zu deutlich weniger *itemsets* die betrachtet werden müssen.

Auch der Apriori ist noch nicht optimal. Eine Erweiterung, AprioriTid, reduziert die Zugriffe auf die Datenbank nochmals drastisch indem er bei der Bestimmung des Supports nicht auf die Transaktionen der Datenbank zugreifen muss. Messungen haben ergeben, dass AprioriTid in der Startphase langsamer ist als Apriori aber danach drastisch schneller wird. Diese Tatsache wird im Algorithmus AprioriHybrid ausgenutzt. AprioriHybrid ist ein Kombination von Apriori und AprioriTid. In der Startphase wird Apriori genutzt, danach AprioriTid.

Literatur

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *In Proc. of the ACM SIGMOD conference on Management of Data, Washington, D.C.*, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *Research Report RJ 9839, IBM Almaden Research Center, San Jose, California, USA*, June 1994.