technische universität
dortmund

Seminar Data Mining: Item Sets That Compress

Arno Siebes, Jilles Vreeken und Matthijs van Leeuwen 2006

Niels Ackermann
7. August 2009

Fakultät für Informatik
Künstliche Intelligenz (Ls8)
Technische Universität Dortmund
http://www-ai.cs.uni-dortmund.de/

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Closed Item Sets 3

3 Kompression **4**
3.1 Kolmogorov Komplexität 4
3.2 Minimum Description Length 5
3.3 Kodierung 6

4 Algorithmen **9**
4.1 Naive Kompression 9
4.2 Pruning 10
4.3 Noise Reduktion 11

5 Experimente **13**

6 Fazit **15**

A Beispiel: Berechnung Naive + Pruning **16**

Abbildungsverzeichnis **19**

Tabellenverzeichnis **20**

Literaturverzeichnis **21**

1 Einleitung

Ziel des *Frequent Item Set Mining* ist es aus einer Datenbank von Transaktionen - man stelle sich einen Einkauf als Transaktion vor - interessante Resultate zu extrahieren. Diese Resultate sind häufige Mengen von Gegenständen, sogenannte *frequent Item Sets*. Sie liefern begehrte Informationen, welche Produkte zusammen gekauft werden, oder andere Querverbindungen. Die Schwierigkeit besteht nun darin, nur die interessantesten Mengen zu finden, sei es weil es sehr viele frequent Item Sets gibt, oder viele frequent Item Sets dieselbe Menge von Items Sets beschreiben. Der Ansatz in [Arn06] ist es durch eine Komprimierung diese interessanten Item Sets zu finden. Hierbei werden keine frequent Item Sets, sondern Item Sets gesucht, die zu einer guten Kompression und somit zu einer guten Beschreibung der Datenbank führen.

2 Grundlagen

Frequent Item Set Mining besteht aus der Extraktion von häufigen Mengen (*frequent Item Sets*) aus einer Datenbank. Eine Datenbank besteht beispielsweise aus Transaktionen eines Kunden mit einem Geschäft. An der Kasse, beim Erwerb der Produkte, bekommt der Einkauf des Kunden nun eine Transaktionsnummer zugewiesen, die zusammen mit den Produktnummern in der Datenbank gespeichert werden. Das Unternehmen hat nun Interesse daran die gewonnenen Daten zu nutzen, sei es der Marketingfachmann, der über sein Productplacement sinniert, oder der Manager, der Entscheidungen über die Produktpalette fällt. Da es vielerlei Anwendungsgebiete gibt, wird von Items und Item Sets gesprochen. Ein Produkt wäre in diesem Falle ein Item und eine Transaktion, bestehend aus vielen Produkten, ein Item Set. Zu suchen sind nun frequent Item Sets, also Item Sets die besonders häufig auftreten und an denen eine bestimmte Kundenpräferenz festzumachen ist.

Aufgrund der Tatsache, dass jede Transaktion gespeichert wird, häufen sich riesige Datenmengen an. Diese Datenmengen in frequent Item Sets zu überführen ist insoweit gelöst, als das es Algorithmen gibt die, abhängig von einer (Häufigkeits-)Schranke, alle frequent Item Sets ermitteln. Hierzu sei [HK06] empfohlen, welches den Apriori-Algorithmus und die zugehörigen Eigenschaften näher erläutert. Die Schranke bestimmt, welche Item Sets als relevant -also frequent- eingestuft werden. Relevant in diesem Sinne ist eine große Häufigkeit (auch Unterstützung, engl. support) in der Datenbank zu besitzen. Das Ergebnis des Apriori-Algorithmus sind nun alle frequent Item Sets, die dieser Schranke genügen.

Sofern ein Item Set als frequent eingestuft wurde, sind auch alle Teilmengen dieses frequent Item Sets ein frequent Item Set. Dies ist auch als Apriori-Eigenschaft bekannt (vgl. [HK06]). Beispielsweise könnte ein (Teil-)Ergebnis eines Apriori die folgende Form haben:

$$\{\{I_1\}\{I_1, I_2\}\{I_1, I_2, I_3\}\{I_1, I_2, I_3, I_4\}\}$$

Dies ist im Sinne der Aufgabenstellung ein korrektes Ergebnis, jedoch beinhaltet dieses Ergebnis viele Redundanzen. Es ist eine kleine Ausgabe erwünscht, die nur die wichtigsten Item Sets enthält.

Die Idee in [Arn06] ist es anstelle aller frequent Item Sets Item Sets zu suchen, die eine gute Beschreibung der Datenbank *db* darstellen. Eine gute Beschreibung ist dann erreicht, wenn eine gute Kompression der Daten vorliegt. Dabei handelt es sich nicht um eine Kompression der frequent Item Sets, sondern um Item Sets, die alle Item Sets in der Datenbank komprimieren, also eine kompakte Darstellung der Datenbank liefern. Diese Darstellung liefert ein Bündel von Item Sets, die häufig vorkommen, aber kaum Redundanzen enthalten. Dies liegt darin begründet, als das für eine zu komprimierende Textstelle bei einer Kompression mit redundanten Item Sets A und B (o.B.d.A. sei A⊂B), entweder A oder B zur Kompression herangezogen werden kann. Bei Kompression mit A fehlen B Teile der zu komprimierenden Menge und wird somit obsulet. Bei Kompression mit B kann A für die mit B komprimierten Teile keine Kompression mehr durchführen, da B alles aus A abdeckt, somit sinkt die Häufigkeit von A.

Ein anderen Weg Redundanzen zu reduzieren ist es die frequent Item Sets als Closed Item Sets anzugeben. Diese werden nachfolgend beschrieben.

2.1 Closed Item Sets

Closed Item Sets sind Item Sets, die nur die größten Item Sets verwenden, sofern die Teilmengen die gleiche Unterstützung erfahren. Sie zeichnen sich durch folgende Eigenschaft aus:

I ist closed, wenn gilt:
∄ Item Set J:

- $I \subset J$ und
- $supp_{db}(I) = supp_{db}(J)$

Damit reduzieren sich die Item Sets, indem solche gefiltert werden, die eine gleiche Unterstützung wie die Untermenge besitzen. Die Closed Item Sets reduzieren damit redundante Informationen, als das die reduzierten Item Sets aus den Untermengen reproduziert werden können.

Ein Beispiel hierfür wäre die Menge

$$\text{Item Set: } \{\{I_1 : 3\}\underbrace{\{I_1, I_2 : 2\}\{I_1, I_2, I_3 : 2\}}\{I_1, I_2, I_3, I_4 : 1\}\}$$
$$\text{Closed Item Set: } \{\{I_1 : 3\}\{I_1, I_2, I_3 : 2\}\{I_1, I_2, I_3, I_4 : 1\}\}$$

wobei in $\{I_a, I_b : c\}$ das Item durch I und durch die Nummer a bzw. b dargestellt wird. Die Häufigkeit des Item Sets $\{I_a, I_b\}$ wird durch c beschrieben.

Closed Item Sets sind also eine Reduktion der Redundanz und werden im weiteren Verlauf als Vergleich herangezogen.

3 Kompression

Ziel einer Kompression ist es eine Reduktion des Speicherbedarfs zu erwirken. Überlicherweise ist eine möglichst große Reduktion wünschenswert. Dabei existieren Kompressionen die für eine bessere Kompression Informationsverluste in Kauf nehmen und solche die verlustfrei operieren. Hier liegt der Fokus auf verlustfreier Kompression.

Eine Kompression besteht in Bezug auf Item Sets nun aus zwei Schritten:

1. Beschreibung aller Item Sets mit einer Auswahl von Item Sets
2. Kodierung der beschreibenden Item Sets in Bits

Es werden also zunächst alle Item Sets der Datenbank *db* betrachtet. Nun wird ein Teilset aus den Item Sets ausgewählt, um die Datenbank mit diesem Teilset zu kodieren. Wichtig hierbei ist, dass das Teilset die komplette Datenbank erfassen kann. Anschliessend wird das Teilset in Bits kodiert. Um eine Kompression beschreiben zu können und deren Eigenschaften zu kennen, beschäftigen wir uns mit der Kolmogorov Komplexität.

3.1 Kolmogorov Komplexität

Die *Kolmogorov Komplexität* ist eine Definition für eine Kompression. Ziel ist es eine darzustellende Zeichenkette mit einem Programm zu erzeugen. Die Kolmogorov Komplexität ist dann die Länge des kürzesten Programms.

Wichtige Eigenschaft für die Kompression ist die Strukturiertheit der Zeichenkette. Je mehr Strukturiertheit vorhanden ist, desto mehr kann komprimiert werden. Eine Kompression tritt auf, sofern die Kolmogorov Komplexität (also das kürzeste Programm) kleiner ist als die Zeichenkette.

Ein einfaches Beispiel ist die Zeichenkette, die aus tausend 0en besteht. Es lässt sich ein einfaches Programm - eine Schleife - angeben, die diese tausend 0en erzeugt und selbst eine geringere Länge hat.

Zeichenkette:
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000

Programm Nullfolge(int n)

```
1        for(int i=0; i<n; i++){
2                System.out.println('0');
3        }
```

Wie zu sehen ist das Programm eine deutlich kompaktere Variante die Zeichenkette darzustellen. Aufgrund der Strukturiertheit der Zeichenkette ist die Kolmogorov Komplexität, geringer als die Zeichenkette.

Nebst der anschaulichen Definition besteht das Problem, dass die Kolmogorov Komplexität von der gewählten Programmiersprache abhängig ist und zudem nicht berechenbar ist. Wir wenden uns daher einem konkreteren Ansatz zu, der Minimum Description Length.

3.2 Minimum Description Length

Die *Minimum Description Length*, kurz MDL, ist ein praktischer Ansatz für die Kolmogorov Komplexität. Um eine Berechenbarkeit zu erreichen ist die Ausdrucksstärke der Programmiersprachen zu groß. In der MDL werden daher weniger ausdrucksstarke Beschreibungsmethoden genutzt. Es wird sich auf einen Teil der verfügbaren Modelle beschränkt. Diese Auswahl der Modelle muss beschränkt genug sein, so dass für alle zu komprimierende Datensätze ein bestes Modell berechnet werden kann. Diese Menge von Modellen wird Hypothese $\mathcal{H}$ genannt. Der Nachteil dieses praktischen Ansatzes ist es, dass es strukturierte Sequenzen geben kann, die unter der gegebenen Hypothese nicht komprimiert werden können.

Ist ein Modell $H \in \mathcal{H}$ ausgewählt, kodiert es die Menge von Beispielen mit einem eindeutigem Präfix, so dass eine kodierte Nachricht dekodiert werden kann. Hierzu ein kleines Beispiel:

Nachricht	n1	n2	n3	n4
Code	0	10	110	111

10001111100100 $\rightarrow$ 10 0 0 111 110 0 10 0 $\rightarrow$ n2 n1 n1 n4 n3 n1 n2 n1

Zu Beginn der kodierten Daten steht eine „1“, somit kommen n2, n3, n4 in Frage, da sie mit einer „1“ beginnen. Die Nachricht n1 kann nicht gewählt werden. Da keine Klarheit darüber besteht welche der Nachrichten n2-n4 gemeint ist, wird die nächste Ziffer „0“ angeschaut. Wir erhalten „10“. Dies lässt sich eindeutig der Nachricht n2 zuordnen, da n3 und n4 mit „11“ beginnen. Nach der Dekodierung wird sich die restliche (01)-Folge (Folge ohne die führende „10“) angeschaut, diese beginnt mit einer „0“. Da nur Nachricht n1 mit einer „0“ beginnt haben wir somit eine weitere Dekodierung.

Die Wahl eines Modells ist gut, sofern häufig genutzte Nachrichten eine geringe Kodierungslänge erhalten. Das obige Modell ist optimal für:

- W'keit(n1)= 1/2
- W'keit(n2)= 1/4
- W'keit(n3)= 1/8
- W'keit(n4)= 1/8

Dies erklärt sich aus der Shannon-Entropie, die besagt:
Wähle Länge L(i) von n_i:

$$L(i) = -log_2 W'keit(n_i)$$

so ist

$$\text{Länge} \geq -\sum W'keit(n_i) log_2(W'keit(n_i))$$

Somit wählen wir das Modell mit der kürzesten Länge.

Hierbei sei erwähnt, dass die Nachrichten ni in unserem Fall Item Sets sind. Je mehr Items ein Item Set mit einer geringen Kodierungslänge ((01)-Folge) hat, desto größer ist das Einsparungspotential. Die Länge beinhaltet dabei nicht nur die komprimierten Daten, sondern auch die Übersetzungstafel, damit die Daten in ihren Originalzustand dekomprimiert werden können.

Wir erhalten somit eine Güte der Kompression:
L(H) + L(D|H) → min

- L(H): Länge der Beschreibung
- L(D|H): Länge der Daten D kodiert mit H

Um nun eine Kompression durchzuführen muss eine Beschreibung der Kompression, eine sogenannte Kodierung, vorliegen. Dies soll im nächsten Abschnitt erarbeitet werden.

3.3 Kodierung

Um eine gültige Kodierung beschreiben zu können, muss zuerst eine Definition für ein *Coding Item Set* und zugehörige Eigenschaften behandelt werden. Ein Coding Item Set

- besteht aus Item Sets,
- jedes Item ist als Item Set enthalten und
- die Reihenfolge bestimmt die Wahl beim Kodieren.

Aufgrund der Tatsache, dass unsere Datenbank *db* aus Item Sets besteht und wir interessante häufige Item Sets suchen, wählen wir die Coding Item Sets gerade so, dass sie interessante häufige Item Sets bilden. Wir erhalten somit eine Kompression und zusätzlich nicht redundante Item Sets die häufig vorkommen.

Um sicherzustellen, die komplette Datenbank *db* kodieren zu können, muss jedes Item als einelementige Menge im Coding Set vorhanden sein.

Während der Komprimierung muss ein Item mindestens von einem Coding Item abgedeckt sein, damit keine Informationen verloren gehen. Zusätzlich muss ein Item genau von einem Coding Item abgedeckt sein, da sonst nicht klar wäre, welches Coding Item zu wählen wäre. Somit erhalten wir eine eindeutige Überdeckung. Um sicherzustellen, dass jede Kodierung deterministisch abläuft muss eine Reihenfolge der Coding Item Sets festgelegt werden. Diese Reihenfolge bestimmt die Wahl der Coding Items während der Kompression und stellt somit sicher, dass ein Text für gegebene Coding Items immer das gleiche Ergebnis erzeugt. Sie liefert somit die Überdeckung für gegebene Coding Item Sets. Die Überdeckung ist eine Ersetzung, die alle Stellen abdeckt, jedoch keine Stelle doppelt.

Eine gültiger Kodierungalgorithmus C(t) muss für jede Transaktion t daher zwei Eigenschaften erfüllen:

- Eine gültige Kodierung beschreibt jede Transaktion.

- Die Überdeckung ist überschneidungsfrei.

Etwas formaler, mit c_i als Coding Item und C als Menge der Coding Item Sets, liest sich dies folgendermaßen:

C(t) covers t:
for each t $\in db$ $\exists$ subset C(t) $\subseteq$ C:

1. t $= \bigcup_{c_i \in C(t)} c_i$
2. $\forall c_i, c_j \in C(t) : c_i \neq c_j \rightarrow c_i \cap c_j = \emptyset$

Um nun eine umfassende Beschreibung zu erstellen benötigen wir ein Kodierungschema. Hierfür wird eine Menge C benötigt, die die Datenbank *db* umfasst und eine Abbildung S, die *db* in eine mögliche Kombinationen der Menge C abbildet.

Kodierungsschema CS
Paar (C,S):

- C: item set cover von *db*
- S: $db \rightarrow \mathcal{P}(C)$ mit S(t) covers t

In [Arn06] wird hierzu ein Algorithmus zur eindeutigen Überdeckung vorgestellt. Er übersetzt also eine Transaktion t in den zugehörigen Code.

$$\text{Transaktion} \xrightarrow{Cover(C,t)} \text{Code}$$

Wir wählen hierfür ein Coding Item c (bzw. S) welches in die Transaktion t hineinpasst, anhand der Reihenfolge. Anschließend kodieren wir die Fundstelle in t mit c und rufen uns rekursiv auf.

Algorithmus Cover(C, t)

```
1        S := smallest element c of C in coding order for which c ⊆
             t
2        if t \ S = ∅
3                then Res := {S}
4                else Res:= {S} ∪ Cover(C, t \ S)
5        return Res
```

Nachdem beschrieben wurde, wie eine Kompression auszusehen hat, kann die Länge einer kodierten Datenbank angegeben werden.

Grösse der Datenbank

$$L_{(C,S)}(db) = -\sum_{c \in C} freq(c) log(\frac{freq(c)}{\sum_{d \in C} freq(d)})$$

mit

- Häufigkeit eines Elements: $freq(c) = |\{t \in db \mid c \in S(t)\}|$
- Codelänge: $L(d) = -log(P(d))$
- Wahrscheinlichkeitsverteilung des Codes: $\forall c \in C : P(c) = \frac{freq(c)}{\sum_{d \in C} freq(d)}$

Hierbei ist die Häufigkeit eine reine Zählung, die Codelänge nach der Shannon-Entropie (vgl 3.2 auf S. 5) und die Wahrscheinlichkeitsverteilung auf Basis der Häufigkeiten definiert.

Eine Kompression könnte nun also, wie im folgenden Beispiel aussehen:

db :

Transaktion	A	B	C	D	Item Set	Komprimiert
T_{100}	1	1	1	1	{A,B,C,D}	{X}
T_{200}	1	1	0	1	{A,B,D}	{Y}
T_{300}	1	0	1	0	{A,C}	{A,C}
T_{400}	1	0	0	1	{A,D}	{A,D}
T_{500}	1	0	0	0	{A}	{A}
T_{600}	0	0	0	1	{D}	{D}

Tabelle 3.1: Beispiel einer Kompression

- X:={A,B,C,D},Y:={A,B,D}
- $L_{(C,S)}(db) = 14,5$

In der Beispieltabelle sind 4 Items A,B,C,D -man stelle sich Ananas, Bananen, Cashewnüsse und Datteln vor- gegeben. Es existieren 6 Transaktionen (respektive Einkäufe) T_{100} bis T_{600} über diese Items. Die Transaktion T_{100} enthält nun also A,B,C und D -Ananas, Bananen, Cashewnüsse und Datteln-, während T_{400} nur A und D -Ananas und Datteln- enthält. Das Ziel der Kompression wurde vom Algorithmus mit der Kompression von {A,B,C,D} und {A,B,D} -man stelle sich Obstkörbe aus Ananas, Bananen, (mit bzw. ohne Cashewnüsse) und Datteln vor-, hier der Einfachhalt halber mit X und Y beschrieben, erfüllt. Das restliche Obst wird mit sich selber kodiert -eine Ananas ist nun ein Obstkorb mit einer Ananas- und wurde der Einfachheit halber nicht durch andere Variablen ersetzt.

Das Beispiel enthält das gültige Coding Item Set {{A,B,C,D},{A,B,D},{A},{D},{B},{C}}. Dies sind die komprimierenden Item Sets plus die einelementigen Mengen. Das bedeutet, dass {A,B,C,D} und {A,B,D} vom Algorithmus zur Komprimierung auserwählt wurden. Hierbei ist die Reihenfolge der Item Sets entscheidend, als dass T_{100} bei Kodierung durch {A,B,D} das Ergebnis {C,X} erhalten würde und somit {A,B,C,D} obsulet wäre, da es in keiner Transition mehr zum Einsatz zu bringen wäre. Wie genau die Kompression, das Coding Item Set und dessen Reihenfolge zu berechnen ist, wird im folgenden Kapitel behandelt.

4 Algorithmen

Um einen Komprimierungsalgorithmus definieren zu können, ist es essentiell die Reihenfolge der Item Sets festzulegen. Dies dient dazu eine stets gleiche Interpretation der Item Sets zu bekommen, um eine deterministische Kompression zu ermöglichen. Im folgenden werden informal zwei Algorithmen vorgestellt, die genau eine Reihenfolge festlegen. Diese Reihenfolge wurde derart gewählt, als dass sie eine intuitive Art der Auflistung darstellt.

- $\mathcal{I}$: einelementige Item Sets
- J: alle Item Sets

Algorithmus Standard($\mathcal{I}$, db)

- absteigend sortiert nach Häufigkeit

Algorithmus Cover-Order(J,db)

1. sortiere nach Länge
   - Längere nach vorne
2. sortiere nach Häufigkeit
   - Häufige nach vorne

Der Algorithmus Standard sortiert einelementige Item Sets nach ihrer Häufigkeit in der Datenbank *db*, wobei Cover-Order jegliche Item Sets (auch die einelementigen) zuerst nach Länge und innerhalb der Gleichlangen nach Häufigkeit in *db* sortiert.

Nachdem nun alle Vorraussetzungen für einen Komprimierungsalgorithmus bestehen, beginnen wir mit einem naiven Greedy-Ansatz.

4.1 Naive Kompression

Die Naive Kompression nimmt das Item Set mit der höchsten Frequenz. Sofern dies nicht eindeutig ist, wird aus der Menge der Item Sets mit der höchsten Frequenz dasjenige Item Set ausgewählt, welches die größte Menge besitzt. Anschließend wird dieses Item Set zu dem bestehenden Code Set (es enthält zur Initialisierung alle einelementigen Mengen nach Standard sortiert) hinzugefügt. Sofern die Länge der Daten unter dem neuen Code Set besser, also geringer, als dass die Länge des Alten Code Sets, wird das neu eingefügte Item Set behalten, sonst wird es verworfen. Der entstehende Algorithmus wurde Naive-Compression genannt.

Algorithmus Naive-Compression($\mathcal{I}$, J, db)

```
1     CodeSet := Standard(𝓘, db)
2     J := J \ 𝓘
3     CanItems := Cover−Order(J, db)
4     while CanItems ≠ ∅ do
5           cand := maximal element of CanItems
6           CanItems := CanItems \ {cand}
7           CanCodeSet := CodeSet ⊕ {cand}
8           if L_CanCodeSet(db) < L_CodeSet(db)
9                 then CodeSet := CanCodeSet
10    return CodeSet
```

Im Algorithmus Naive-Compression ist *CodeSet*, das beste bisher gefundene Coding Set und *CanItems* die Kandidaten die zur Hinzufügung zur Verfügung stehen. *Cand* ist hierbei ein aktueller Kandidat, welcher in *CanCodeSet* als zu evaluierendes Coding Set zum *CodeSet* hinzugefügt wird. L ist die Länge der Kodierung der Datenbank db mit einem Coding Set nach Kapitel 3.3, S. 6. Das Zeichen $\oplus$ ist die Hinzufügung eines Item Sets mit Hilfe der Sortierung Cover-Order.

Sofern wir diesen Algorithmus auf das Beispiel in Tabelle 3.1 auf Seite 8 anwenden, ergibt sich folgender Verlauf:

- $CodeSet_{Start}$ = {{A},{D},{B},{C}}
- CanItems := {{A,B,C,D},{A,B,D},...{A,D},{A,B},...,{C,D}}
- cand = {A,D} (höchste Frequenz: 3)
- prüfe ob CodeSet $\oplus$ {A,D} geringere Länge hat
  $L_{CanCodeSet}(db) = 22,3 < L_{CodeSet}(db) = 24,5$
  $\rightarrow$ Behalten von {A,D}
- nächster Kandidat: {A,B,D} (größte Menge mit Frequenz 2)

Aufgrund der Naivität des Ansatzes lässt sich ein Beispiel für die Schwächen des Algorithmus finden. Dies beinhaltet die drei Item Sets

- $CS_1 = \{\{I_1, I_2\}, \{I_1\}, \{I_2\}, \{I_3\}\}$
- $CS_2 = \{\{I_1, I_2, I_3\}, \{I_1, I_2\}, \{I_1\}, \{I_2\}, \{I_3\}\}$
- $CS_3 = \{\{I_1, I_2, I_3\}, \{I_1\}, \{I_2\}, \{I_3\}\}$

mit der Annahme, dass

$$supp(\{I_1, I_2, I_3\}) = supp(\{I_1, I_2\}) - 1.$$

Sofern

$$L_{CS_2}(db) > L_{CS_1}(db) > L_{CS_3}(db)$$

ist, ist die Lösung des Algorithmus nicht optimal. Die Naive-Compression startet bei CS_1 und überprüft zunächst CS_2 und befindet dies für schlechter. CS_3 wird somit, da CS_2 verworfen wird, nie vom Algorithmus geprüft, obwohl es hier das beste Ergebnis liefert. Um solche Fälle dennoch zu prüfen wurde ein weiterer Algorithmus entworfen, das Pruning.

4.2 Pruning

Das Pruning überprüft für ein gegebenes Coding Item Set, ob durch Entfernen von Item Sets eine bessere Kompression erfolgen kann. Da das Pruning nach der naiven Kompression ausgeführt wird, ist es für den Algorithmus nicht lohnenswert das zuletzt hinzugefügte Item Set zu prüfen. Das Entfernen würde die gewonnene Datenverkleinerung rückgängig machen. Auch das Entfernen der einelementigen Item Sets, als Basisbestandteil, darf nicht durchgeführt werden. Es sind also Coding Item Sets gefragt, die durch das neu hinzugefügte Coding Item Set eine Reduzierung ihrer Überdeckung erfahren haben. Diese Coding Item Sets decken nun einen kleineren Teil der Daten ab, als für den sie ausgewählt wurden. Es sollte also geprüft werden, ob solche Coding Item Sets noch lohnenswert sind. Das Pruning kann nun nach jedem Schritt des naiven Ansatzes, siehe 4.1 auf Seite 9, ausgeführt werden und wird daher Prune-on-the-fly genannt.

Algorithmus Prune-on-the-fly(CanCodeSet,CodeSet, db)

```
1          PruneSet :=
2                  {J ∈ CodeSet | cover_CanCodeSet(J) < cover_CodeSet(J)}
3          PruneSet := Standard(PruneSet, db)
4          while PruneSet ≠ ∅ do
5                  cand := element of PruneSet with minimal cover
6                  PruneSet := PruneSet \ {cand}
7                  PosCodeSet := CodeSet ⊖ {cand}
8                  if L_PosCodeSet(db) < L_CanCodeSet(db)
9                          then CanCodeSet := PosCodeSet
10         return CanCodeSet
```

Das *PruneSet* ist hierbei die Menge der Item Sets die zum Pruning zur Verfügung stehen. Dabei werden die Item Sets eingefügt, deren Anzahl überdeckter Transitionsstellen (*cover*) sich verringert hat. $\ominus$ ist das Entfernen anhand der Sortierung Cover-Order. Die restlichen Variablen und Zeichen finden ihre Erklärung in Abschnitt 4.1.

Als Beispiel sei hier erneut Tabelle 3.1 S. 8 herangezogen, mit den Ergebnissen aus Abschnitt 4.1:

- Menge nach zweitem Naive-Compression-Schritt:
  $CanCodeSet = \{\{A, B, D\}, \{A, D\}, \{A\}, \{D\}, \{B\}, \{C\}\}$
  $CodeSet = \{\{A, D\}, \{A\}, \{D\}, \{B\}, \{C\}\}$
- $Prunset := \{\{A, D\}\}$
- $cand = \{A, D\}$
- $PosCodeSet = \{\{A, B, D\}, \{A\}, \{D\}, \{B\}, \{C\}\}$
- $L_{PosCodeSet}(db) = 17,7 < L_{CanCodeSet}(db) = 18$
  → Entfernen von {A,D}

Nach dem ersten naiven Kompressionsschritt gibt es noch kein Item Set zum prunen, weil das neu hinzugefügte Item Set nicht vom Pruning getestet werden braucht und sonst nur einelementige Item Sets im Code Set beinhaltet sind.

Nach dem zweiten Schritt ist also erst ein Pruning möglich, wobei PruneSet nur dann mit eben jenen gefüllt wird, sofern es Item Sets gibt, deren cover verringert wurde. Dies ist bei {A,D} der Fall, da {A,B,D} in T_{100} und T_{200} Stellen beansprucht, die von {A,D} im Schritt davor komprimiert wurden (beachte: Reihenfolge der Coding Item Sets nach Cover-Order). Nun wird {A,D} als Kandidat ausgewählt und mit Erfolg aus der Menge entfernt (geprunt). Die komplette Berechnung von naiver Kompression mit Pruning befindet sich im Anhang A.

4.3 Noise Reduktion

Noise, im Sinne eines Störungsrauschen, sind ungewöhnliche Transaktionen, also Transaktionen, mit einer Item-Kombinationen die sehr selten vorkommt. Solche Transaktionen sind für die Suche nach frequent Item Sets nicht erforderlich, erfordern aber, dass sie in einer Kompression beachtet werden. Aufgrund der Tatsache, dass sie für uns für die Kompression relevant, aber für die Suche nach frequent Item Sets hinderlich sind, möchten wir solche Transaktionen gerne entfernen.

Eine ungewöhnliche Transaktion kennzeichnet sich durch folgende Eigenschaft:

- $L_S(t) < L_{CS}(t)$

Ihre Codelänge ist mit Komprimierung größer als ohne Komprimierung (mit Standardkodierung).

Im Paper werden hierzu drei Algorithmen vorgestellt:

1. zum Finden solcher Transaktionen (Noise),
2. zum Entfernen solcher Transaktionen aus der Datenbank (Denoise) und
3. zum Entfernen der Transaktionen aus der Komprimierung (Sanitize).

Algorithmus Noise($\mathcal{I}$,CodeSet, db)

```
1    Noise := ∅
2    foreach t ∈ db do
3        if L_S(t) < L_CS(t) then
4            Noise := Noise∪{t}
5    return Noise
```

Algorithmus Denoise($\mathcal{I}$,CodeSet, db)

```
1    Noise := Noise(I,CodeSet, db)
2    db := db \ Noise
3    return db
```

Algorithmus Sanitize($\mathcal{I}$,CodeSet, db)

```
1    db := Denoise(I,CodeSet, db)
2    foreach J ∈ CodeSet do
3        if cover_db(J) = ∅ then
4            CodeSet := CodeSet ⊖ J
5    return CodeSet
```

Noise ist hierbei die Menge der ungewöhnlichen Transaktionen und $cover_{db}(J) = \emptyset$ der Test, ob ein Coding Item Set J keine Überdeckung in der Komprimierung mehr birgt, also durch Entfernung von Noise obsulet wird. $\ominus$ ist das Entfernen anhand der Sortierung Cover-Order.

5 Experimente

Um die entwickelten Algorithmen zu prüfen wurden diese auf verschiedenen Datenbanken geprüft. Es wurden zwei Datenbanken, eine Schach und eine Pilzdatenbank, herangezogen. Ziel der Experimente war es zu zeigen, dass eine gute Komprimierung vorliegt und dass eine geringe Zahl von Coding-Items ausgewählt wird. Aufgrund der Art der Kompressionsalgorithmen ist anzunehmen, dass die Coding-Items verschieden sind und somit verschiedene frequent Item Sets vorliegen.

Die Schachdatenbank hat (bei minsup=1500) 2 Millionen frequent Item Sets zu 550.000 Closed Item Sets. Die Pilzdatenbank hat (bei minsup=724) 945.000 frequent Item Sets zu 7800 Closed Item Sets.

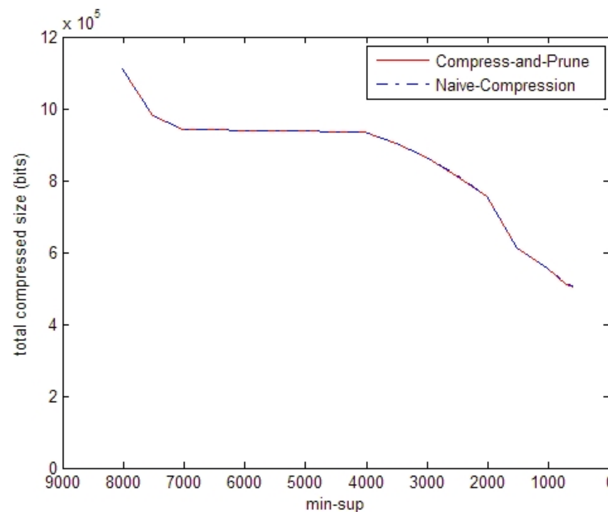

Abbildung 1: Naive-Compression mit und ohne Pruning, aus [Arn06]

Wie in Abbildung 1 zu sehen, ist der Unterschied zwischem reinem Naive-Compression und Naive-Compression mit Prune-On-The-Fly (hier: Compress-and-Prune) für die Pilzdatenbank marginal. Die Abbildung stellt hier die komprimierte Größe zu der gegebenen (Häufigkeits-)Schranke (min-sup) gegenüber.

Abbildung 2 faßt die Informationen zur Schachdatenbank zusammen. Auf der linken Seite wird die Kompression dargestellt, auf der rechten Seite die Items der Codetabelle. Hier werden als Startpunkt die Closed Items Sets und All Item Sets gegenübergestellt. Für die unterschiedlichen Supports ergibt sich ein vergleichbares Niveau.

In Tabelle 5.1 sind die Ergebnisse der Pilzdatenbank zusammengefasst. In der Datenbank konnte keinerlei Noise gefunden werden. Die nach der Reduktion vorhandenen Item Sets in der Pilzdatenbank belaufen sich auf 2% (von 7800) für die Closed Item Sets. Für All Item Sets beläuft sich dies auf 0,03% (von 945.000). Trotz der größeren Reduktion sind die absoluten Zahlen für Closed Item Sets mit 147 geringer als die für All Item Sets mit 282. Mit All Item Sets konnte durch die größere Auswahl an Item Sets eine bessere Kompression erreicht werden. Die höhere Anzahl an All Item Sets nach der Kompression weißt auf eine Struktur hin, die von den Closed Item Sets nicht erfasst wurde.

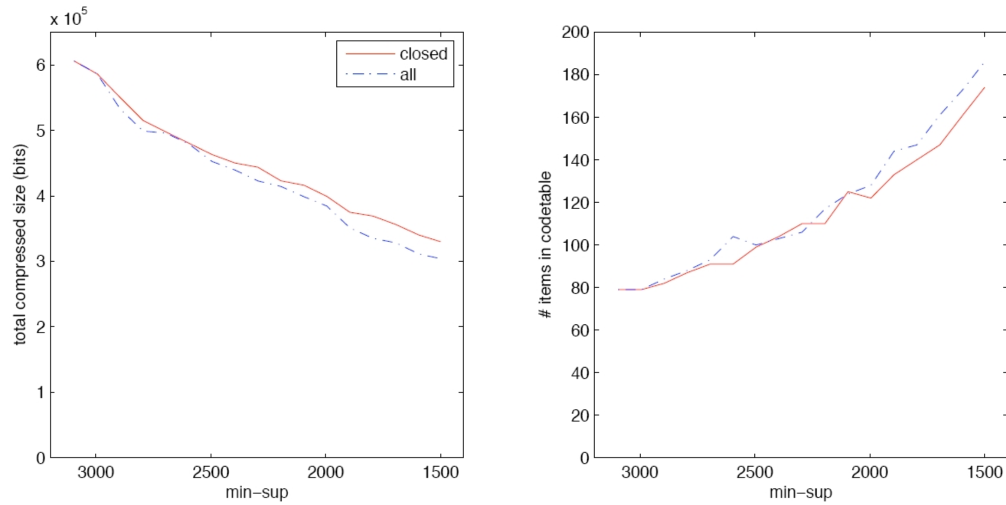

Abbildung 2: Closed Item Sets VS All Item Sets, aus [Arn06]

Source	Algo	C-Items	C-Table	Db	Noise	Compress
All	Naive	338	9901	425977	0	435878
All	Naive+Sanit	338	9901	425977	0	435878
All	Naive&Prune	282	8252	423487	0	431739
All	All	282	8252	423487	0	431739
Closed	Naive	149	4754	507691	0	512445
Closed	Naive+Sanit	149	4754	507691	0	512445
Closed	Naive&Prune	147	4631	507633	0	512246
Closed	All	147	4631	507633	0	512246

Tabelle 5.1: Ergebnisse, aus [Arn06]

6 Fazit

Die vorgestellten Algorithmen sind Algorithmen, die erfolgreich nicht-redundante frequent Item Sets finden. Dabei wurde sowohl mit allen Item Sets, als auch mit Closed Item Sets eine gute Kompression der Datenbank erreicht. Sie liefern somit eine gute Beschreibung der Datenbank. Bezüglich Closed und All Item Sets liefern die Algorithmen unterschiedliche Ergebnisse. Einige wohl interessante frequent Item Sets kommen in den Closed Item Sets nicht vor. Insgesamt hat der Naive-Compression- und der Prune-on-the-fly-Algorithmus mit einem Bruchteil der Item Sets mehr als die Hälfte der Datenbank eingespart, Noise hingegen hatte keinen Effekt. Für Noise ist anzunehmen, dass die Testbeispiele keine Noise enthielten, wozu weitere Tests notwendig wären. Die Algorithmen sind daher eine gute Möglichkeit frequent Item Sets ohne Redundanzen zu finden.

A Beispiel: Berechnung Naive + Pruning

db :

Transaktion	A	B	C	D	Item Set	Komprimiert
T_{100}	1	1	1	1	{A,B,C,D}	{X}
T_{200}	1	1	0	1	{A,B,D}	{Y}
T_{300}	1	0	1	0	{A,C}	{A,C}
T_{400}	1	0	0	1	{A,D}	{A,D}
T_{500}	1	0	0	0	{A}	{A}
T_{600}	0	0	0	1	{D}	{D}

Tabelle A.1: Beispiel einer Kompression, siehe Tabelle 3.1

Formel:

$$L_{(C,S)}(db) = -\sum_{c \in C} freq(c) log_2(\frac{freq(c)}{\sum_{d \in C} freq(d)})$$

mit freq(c) als Häufigkeit des Item Set c

Schritt 1: Kompression nur mit einelementigen Mengen

A	5	$-5 * log(\frac{5}{13}) =$	6,9
D	4	$-4 * log(\frac{4}{13}) =$	6,8
B	2	$-2 * log(\frac{2}{13}) =$	5,4
C	2	$-2 * log(\frac{2}{13}) =$	5,4
	13		24,5

Schritt 2: Naive Kompression, füge {A,D} hinzu

A,D	3	$-3 * log(\frac{3}{10}) =$	5,2
A	2	$-2 * log(\frac{2}{10}) =$	4,6
D	1	$-1 * log(\frac{1}{10}) =$	3,3
B	2	$-2 * log(\frac{2}{10}) =$	4,6
C	2	$-2 * log(\frac{2}{10}) =$	4,6
	10		22,3

→ Wähle dies als neue Kompression!

Schritt 3: Naive Kompression, füge {A,B,D} hinzu

A,B,D	2	$-2 * log(\frac{2}{8}) =$	4
A,D	1	$-1 * log(\frac{1}{8}) =$	3
A	2	$-2 * log(\frac{2}{8}) =$	4
D	1	$-1 * log(\frac{1}{8}) =$	3
B	0	$-0 * log(\frac{0}{8}) =$	0
C	2	$-2 * log(\frac{2}{8}) =$	4
	8		18

→ Wähle dies als neue Kompression!

Schritt 4: Pruning von Schritt 3, prune {A,D}

A,B,D	2	$-2 * log(\frac{2}{9}) =$	4,3
A	3	$-3 * log(\frac{3}{9}) =$	4,8
D	2	$-2 * log(\frac{2}{9}) =$	4,3
B	0	$-0 * log(\frac{0}{9}) =$	0
C	2	$-2 * log(\frac{2}{9}) =$	4,3
	9		17,7

→ Wähle dies als neue Kompression!

Schritt 5: Naive Kompression, füge {A,B,C,D} hinzu

A,B,C,D	1	$-1 * log(\frac{1}{8}) =$	2,1
A,B,D	1	$-1 * log(\frac{1}{8}) =$	2,1
A	3	$-3 * log(\frac{3}{8}) =$	4,2
D	2	$-2 * log(\frac{2}{8}) =$	4
B	0	$-0 * log(\frac{0}{8}) =$	0
C	1	$-1 * log(\frac{1}{8}) =$	2,1
	8		14,5

→ Wähle dies als neue Kompression!

Schritt 6: Pruning von Schritt 5, prune {A,B,D}

A,B,C,D	1	$-1 * log(\frac{1}{10}) =$	3,3
A	4	$-4 * log(\frac{4}{10}) =$	5,3
D	3	$-3 * log(\frac{3}{10}) =$	5,2
B	1	$-1 * log(\frac{1}{10}) =$	3,3
C	1	$-1 * log(\frac{1}{10}) =$	3,3
	10		20,4

→ Keine Verbesserung zu Schritt 5, belasse {A,B,D} in der Menge.

Schritt 7: Naive Kompression, füge {A,C} hinzu

A,B,C,D	1	$-1 * log(\frac{1}{7}) =$	2,8
A,B,D	1	$-1 * log(\frac{1}{7}) =$	2,8
A,C	1	$-1 * log(\frac{1}{7}) =$	2,8
A	2	$-2 * log(\frac{2}{7}) =$	3,6
D	2	$-2 * log(\frac{2}{7}) =$	3,6
B	0	$-0 * log(\frac{0}{7}) =$	0
C	0	$-0 * log(\frac{0}{7}) =$	0
	7		15,6

→ Keine Verbesserung zu Schritt 5!

Anmerkungen zu den Schritten:

Schritt 1: Dient als Vergleich, Kodierung ohne Komprimierung durch Item Sets.

Schritt 2: Wähle {A,D}, da dies den höchsten support (von 3) hat. Anschließend kein Pruning, da {A},{D},{B},{C} einelementig und {A,D} gerade erst hinzugefügt.

Schritt 3: Wähle {A,B,D}, da dies den höchsten support (von 2) hat. {A,D} ist von support 3 auf 1 gefallen, steht somit zum Pruning zur Verfügung. Der Rest ist gerade hinzugefügt, oder einelementig. Somit besteht die Pruningmenge nur aus {{A,D}}.

Schritt 5: Wähle {A,B,C,D}, da dies den höchsten support (von 1) und eine größere Länge als {A,C} hat. {A,B,D} ist von support 2 auf 1 gefallen, steht somit zum Pruning zur Verfügung. Der Rest ist gerade hinzugefügt, oder einelementig. Somit besteht die Pruningmenge nur aus {{A,B,D}}.

Schritt 7: Verbleibende Menge {A,C}. Kein anschließendes Pruning, weil {A,B,D} gleichen Support wie vorher hat und {A,B,C,D} ebenso. {A,C} wurde gerade hinzugefügt,

der Rest ist einelementig.

Abbildungsverzeichnis

1 Naive-Compression mit und ohne Pruning, aus [Arn06] 13
2 Closed Item Sets VS All Item Sets, aus [Arn06] 14

Tabellenverzeichnis

3.1 Beispiel einer Kompression 8
5.1 Ergebnisse, aus [Arn06] 14
A.1 Beispiel einer Kompression, siehe Tabelle 3.1 16

Literaturverzeichnis

[Arn06] Arno Siebes, Jilles Vreeken and Matthijs van Leeuwen. Item sets that compress. In *Procs. SIAM Int. Conference on Data Mining*, 2006.

[Com09] Wikipedia Community. Kolmogorow-Komplexität. `http://de.wikipedia.org/wiki/Kolmogorov-Komplexit%C3%A4t`, 2009. (Verfügbar 21.04.09).

[HK06] Jiawei Han und Micheline Kamber. *Data Mining - Concepts and Techniques.* Morgan Kaufmann, 2 edition, 2006.

[Mor08] Katharina Morik. Vorlesung Maschinelles Lernen: Klassifikation und Regression: nächste Nachbarn. Folien zur Vorlesung Maschinelles Lernen, LS 8 Künstliche Intelligenz Fakultät für Informatik Technische Universität Dortmund, 10 2008.

[P.G05] P.Grünwald. A tutorial introduction to the minimum description length principle. *MIT Press*, 2005.