Implementierung eines Data-Mining-Verfahrens für adaptive Datenquellen

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von: Marc-Eric Meier
Matrikelnummer: 211201312
geboren am: 26.09.1989 in Rostock
Gutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: PD Dr.-Ing. habil. Meike Klettke
Betreuer: M.Sc. Hannes Grunert
Abgabedatum: 14. April 2015

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 14. April 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung und Aufgabenstellung

1.1 Assistenzssysteme

Assistenzsysteme sollen dem Menschen im Lebens- und Arbeitsalltag zur Seite stehen, ohne ihn dabei zu behindern. Dazu ist es notwendig, dass das System die Intention des Nutzers erkennt und selbstständig unterstützende Maßnahmen ergreifen kann. Ein System zur automatisierten Dokumentation von Pflegeleistungen muss in der Lage sein, den Pflegenden, den zu Pflegenden sowie die Pflegeleistung selbst zu erkennen. Damit ein Roboterarm einem Mechaniker das richtige Werkzeug reichen kann, muss der Computer den aktuellen Arbeitsschritt identifizieren, um das entsprechende Utensil zu wählen.
Hierfür sind zum einen Daten aus Sensoren, welche die Umgebung wahrnehmen vonnöten, ebenso Wissen über Abläufe und Individuen des Systems. Dieses kann a priori bekannt sein. Für einen Patienten kann bereits eine Krankenakte vorliegen, der Ablauf der Reparatur eines Schiffsmotors kann ebenfalls im System hinterlegt worden sein. Häufig muss neues Wissen allerdings erst aus bekannten Informationen und Beobachtungen gewonnen werden. Bei diesem Prozess werden mit großer Wahrscheinlichkeit auch personenbezogene Daten erhoben und verarbeitet, wodurch besondere datenschutzrechtliche Aspekte in Betracht gezogen werden müssen.

1.2 Datenschutz in Deutschland

In der *Allgemeinen Erklärung der Menschenrechte* der Vereinten Nationen vom 10.12.1948 heißt es im Artikel 12 wörtlich:

> „Niemand darf willkürlichen Eingriffen in sein Privatleben, seine Familie, seine Wohnung und seinen Schriftverkehr oder Beeinträchtigungen seiner Ehre und seines Rufes ausgesetzt werden.“[Gen48]

Weiterhin sind im Grundgesetz der Bundesrepublik Deutschland von 1949 Artikel 1, 2, 10, 13 festgelegt [Bun12]. Aus diesen Grundsätzen folgt die Notwendigkeit des Schutzes persönlicher Daten. Das Bundesverfassungsgericht beschloss im *Volkszählungsurteil* vom 15.12.1983, dass der Schutz des Einzelnen gegen unbegrenzte Erhebung, Speicherung, Verwendung und Weitergabe persönlicher Daten durch Artikel 2 GG eingeschlossen wird, weshalb jeder Bürger ein Grundrecht auf informationelle Selbstbestimmung besitzt.
In Deutschland wird dieses Recht durch das Bundesdatenschutzgesetz [Bun09] geregelt[1]. Es definiert

[1] Für den Standort der Universität Rostock sei noch Artikel 6 der Verfassung des Landes Mecklenburg-Vorpommern [Lan93] (Datenschutz, Informationsrechte) genannt und auf das Landesdatenschutzgesetz [Lan02] verwiesen werden. Aufgrund des (inter-)nationalen Charakters der Forschung soll es dabei belassen werden.

personenbezogene Daten als Einzelangaben über persönliche oder sachliche Verhältnisse einer bestimmten oder bestimmbaren natürlichen Person (§ 3 Abs. 1 BDSG). Solche Daten dürfen grundsätzlich nur erhoben werden, wenn eine informierte Einwilligung des Betroffenen vorliegt oder wenn es durch ein gültiges Gesetz erlaubt oder angeordnet wird. Die Nutzung der Informationen ist nur zum angegebenen Zweck zulässig. Es wird Datenvermeidung und Datensparsamkeit geboten. Das heißt, es sollen nur so wenig personenbezogene Daten wie möglich erhoben werden. Soweit es nach dem Verwendungszweck möglich und verhältnismäßig ist, sind diese Daten zu anonymisieren oder pseudonymisieren.

1.3 Aufgabenstellung und Projektrahmenbedingungen

Es ist deutlich zu sehen, dass diese Aspekte nicht ignoriert werden können, wenn intelligente Umgebungen in den menschlichen Alltag Einzug halten sollen. Für welche Informationen wie vorgegangen werden darf, muss daher mit einer Datenschutzrichtlinie in Form von Sichtkonzepten und Zugriffsrechten erst konfiguriert werden.
Intelligente Umgebungen sind nur selten starre Gebilde, sondern im Zuge ihrer Weiterentwicklung und Wartung erhalten sie neue Sensoren, alte werden ausgetauscht oder ganz entfernt. Liegen für neue Messgeräte seitens der Nutzer oder Entwickler noch keine Voreinstellungen bezüglich des Datenschutzes vor, müssen diese manuell ergänzt werden. Um den Aufwand für Wartung und Administration zu minimieren, ist es daher wünschenswert, entsprechende Voreinstellungen aus bekannten Datenschutzeinstellungen generieren zu können.
Der Lehrstuhl für Datenbank- und Informationssysteme (DBIS) der Universität Rostock möchte für das PArADISE-Framework (Privacy-AwaRe Assistive Distributed Information System Environment [Gru14]) einen derartigen Mechanismus in zwei Schritten realisieren. Dazu werden im ersten Schritt mittels Similarity Flooding [MGMR02] Schema-Mappings zwischen neuen und bekannten Datenquellen erzeugt und die Privatspähreeinstellungen entsprechend übernommen. Für Attribute, denen auf diese Weise keine Voreinstellungen zugeordnet werden können, sollen automatisch entsprechende Werte zugewiesen werden. Ziel dieser Arbeit ist es, Data-Mining-Verfahren für diesen Zweck zu evaluieren und prototypisch zu realisieren.

1.4 Leitfaden durch diese Arbeit

In den folgenden Kapiteln soll zuerst das Thema Data-Mining im Allgemeinen erschossen werden, anschließend werden geeignete Techniken evaluiert und eine Entscheidung festgelegt[2]. Kapitel 4 beschreibt das Schema-Mapping-Verfahren *Similarity Flooding* und beschäftigt sich mit der vorgegebenen Implementierung im PArADISE-Framework. Ihre Ausgaben dienen als Grundlage für das Data-Mining. In Kapitel 5 wird die Assoziationsanalyse noch einmal genauer vorgestellt werden. Es wird auf die Verfahren *Apriori* und *FPGrowth* eingegangen, für letzteres soll eine rudimentäre Implementation vorgestellt werden. Dieses wird genutzt, um in Kapitel 6 die Anwendbarkeit zu überprüfen. Mithilfe der gewonnenen Erkenntnisse wird in Kapitel 7 eine Neuimplementation vorgeschlagen. Abschließend wird ein kleiner Ausblick auf die Zukunft gewagt werden.

[2]Kapitel 1 bis 3 wurden in großen Teilen aus der vorangegangenen Literaturarbeit übernommen.

Kapitel 2

Knowledge Discovery in Databases und Data-Mining

2.1 Knowledge Discovery in Databases

In Forschung und Wirtschaft ist es unabdingbar, Wissen aus Datenbeständen zu gewinnen. Diese Aufarbeitung der Daten bietet zum einen die Möglichkeit, die Datenmenge als Ganzes zu erfassen und zu verstehen, zum anderen können aus gefundenen Mustern und Regeln Vorgehensweisen für zukünftiges Handeln hergeleitet werden. Unternehmen nutzen gesammelte Informationen über ihre Kunden, kategorisieren diese und können so beispielsweise zielgerichtetes Marketing betreiben. Banken werten Erfahrungen vergangener Kreditvergaben aus, um abschätzen zu können, ob ein potentieller Kunde kreditwürdig ist oder nicht. Astronomen sichten riesige Mengen Bilddaten von Teleskopen und versuchen so neue Himmelskörper zu erkennen und bekannten Kategorien zuzuordnen. [FPSS96, ES00]
Die Datenberge wachsen mit den sich stetig weiter entwickelnden Techniken, Daten zu gewinnen und zu speichern. Aus diesem Grund ist es wichtig, Methoden zu finden, die diesen Prozess in großen Teilen, statt vom Menschen, maschinell durchführen zu können. Dieser Prozess wird als Knowledge Discovery in Databases (KDD) bezeichnet und besteht aus sieben iterativen Schritten [HK01].
Die ersten vier Schritte beschreiben die datenaufbereitenden Vorgänge Datenbereinigung, Datenintegration, Attributsauswahl und Datentransformation. Hierbei handelt es sich um manuelle Aufgaben, welche teilweise maschinell unterstützt werden können. Sie werden in Hinblick auf das gewählte Ziel und den genutzten Algorithmus durchgeführt und optimieren damit das Data-Mining.
Die Anwendung eines geeigneten Algorithmus auf die aufbereiteten Daten wird als Data-Mining bezeichnet. Dieser Schritt ist Hauptthema dieser Arbeit. Am Ende des Vorgangs stehen gefundene Muster. Die Bewertung dieser Ergebnisse und die Präsentation des erlangten Wissens sind die letzten Aufgaben im Gesamtprozess und müssen vom Benutzer manuell durchgeführt werden. Auch hier ist maschinelle Unterstützung möglich. Zu jedem Zeitpunkt kann zu einem vorhergehenden Schritt zurückgekehrt und die notwendigen Schritte wiederholt werden.
KDD ist ein stark interdisziplinärer Prozess, der sich unter anderem an Techniken von maschinellem Lernen, Mustererkennung, Statistik, Datenbanken, künstlicher Intelligenz, Datenvisualisierung und Hochleistungsrechnen bedient. Für den Teilschritt *Data-Mining* sind dabei insbesondere die ersten drei Disziplinen von Interesse.

2.2 Data-Mining-Methoden

2.2.1 Clustering

Häufig sollen in Daten Gruppen von ähnlichen Objekten gefunden werden. Bei der Auswertung einer Kundendatenbank können das verschiedene Kundengruppen sein, die ein ähnliches Verhalten aufweisen. Für eine gegebene Menge von Vertretern etwa einer Pflanzenart ist es denkbar, Merkmale und deren Ausprägungen zu untersuchen und so Unterarten festzustellen. In der Geographie können Satellitenaufnahmen auf markante Formationen hin untersucht werden.
Solche Gruppen, Klassen oder Kategorien werden als Cluster bezeichnet. Bei dieser Einteilung der Daten ist es wünschenswert, dass Objekte, die im gleichen Cluster liegen, so ähnlich wie möglich sind und solche, die es nicht tun, möglichst verschieden. Daher ist ein geeignetes Ähnlichkeitsmaß (oder Unterscheidungsmaß) zu wählen. Ein mögliches Maß hierfür ist etwa die euklidische Distanz[1] zweier Punkte mit Anteil verschiedener Elemente von Mengen. Bei der Wahl eines geeigneten Clusteringalgorithmus sind die verschiedenen zu erwartenden Charakteristika von Clustern zu beachten. Neben Form und Dichte sollte in Betracht gezogen werden, dass Cluster nicht nur nebeneinander liegen können, sondern auch hierarchisch verschachtelt sein können. Deshalb müssen geeignete Algorithmen gewählt werden, die Cluster verschiedenster Charakteristika erkennen können, je nach Anwendungsfall kann es auch sinnvoll sein, mehrere Verfahren anzuwenden.
Partitionierende Verfahren ordnen jedem Cluster mindestens ein Objekt und jedes Objekt genau einem Cluster zu. Ein möglicher Ansatz hierfür ist das Suchen zentraler Punkte und die Zuordnung naher Objekte zu diesem Punkt. Bekannte Algorithmen dieser Art sind *k-Means* [M$^+$67], *Partitioning Around Medoids* (PAM) und Varianten [VdLPB03], sowie *Clustering Large Applications based on Randomized Search* (CLARANS) [NH02]. Andere partitionierende Algorithmen basieren auf der Annahme, dass jeder Cluster einem normalverteilten Prozess entspricht. Es wird nun das wahrscheinlichste Modell von Normalverteilungen gesucht, aus dem die vorliegenden Daten entstanden sein können. DENCLUE ist ein Beispiel für einen solchen Algorithmus [HK98].
Die genannten Verfahren sind iterative Prozesse, die beginnend bei einem initialen Modell (zum Beispiel k beliebige Centroide oder Normalverteilungen, wobei $k \in \mathbb{N}$) eine Verbesserung vornehmen. Dabei ist es möglich, dass sie nur ein lokales Optimum finden. Den Algorithmen müssen im Übrigen die Anzahl der zu findenden Cluster als Parameter übergeben werden, wenn diese Zahl nicht bekannt ist, muss entsprechend eine gute Schätzung vorliegen oder es finden mehrere Durchläufe statt, von denen das beste Ergebnis gewählt wird. Außerdem sind beide Verfahren nur dazu in der Lage, annähernd kreisförmige Cluster zu entdecken. Dies ist beispielsweise problematisch, wenn auf einer Karte ein Fluss erkannt werden soll.
Abhilfe schaffen unter anderem die dichtebasierten Verfahren. Stellvertretend für diese Klasse von Algorithmen ist DBSCAN (A Density-Based Clustering Method Based on Cennected Regions with Sufficiently High Density). Nacheinander wird hier die Umgebung aller Objekte auf Dichtekriterien überprüft, was soviel bedeutet, wie 'Liegen im Umkreis ϵ mindestens n weitere Objekte?'. Hierbei müssen ϵ und n als Parameter übergeben werden, der Parameter n erhält in der Literatur oft den Namen *MinPts*. Mithilfe geeigneter Heuristiken lassen sich hierfür leicht gute Werte bestimmen [EKSX96]. Eine Erweiterung von DBSCAN ist OPTICS (Ordering Points To Identify the Clustering Structure). OPTICS macht das Ergebnis der Clustersuche unabhängiger von den eingegebenen Parametern. Diese Verfahren findet keine expliziten Cluster der gegebenen Daten, viel mehr liefert es eine Clusterordnung der Daten. OPTICS ist damit in der Lage, auch hierarchische Cluster zu finden [ABKS99].
Gitterbasierte Verfahren teilen den Datenraum in endlich viele Zellen ein. Sie zeichnen sich häufig durch eine gute Performance aus. Als Beispiele seien STING (Statistical Information Grid) [WYM$^+$97] und CLIQUE (Clustering in Quest) [AGGR98] genannt. Ebenso gibt es Ansätze, die auf neuronalen Netzwerken basieren, etwa Kohonennetze [Koh90]. Dieses nicht-überwachte Verfahren wird häufig zur Visualisierung hochdimensionaler Daten genutzt.

[1] Euklidische Distanz zweier d-dimensionaler Punktvektoren x und y: $dist(x, y) = \sqrt{(x_1 + y_1)^2 + .. + (x_d + y_d)^2}$

2.2.2 Klassifikation

Bei der Klassifikation sollen aus einer Menge von bereits kategorisierten Daten Regeln gelernt werden, mit deren Hilfe neue Datenobjekte in diese Kategorien eingeordnet werden können. Beispielsweise kann es für ein Kreditinstitut interessant sein, ob ein Kreditanwärter aufgrund der über ihn bekannten Informationen in die Kategorie *kreditwürdig* oder *nicht kreditwürdig* fällt. Als Grundlage für diese Entscheidung können Erfahrungen von bereits abgeschlossene Kreditverträgen herangezogen werden.
Zum Lernen der Regeln müssen Trainingsdaten vorhanden sein, für dessen Objekte die Kategorien bereits bekannt sind. Weiterhin sollten Testdaten zur Verfügung stehen, die zur Validierung des gewonnenen Wissens genutzt werden. Um auch mit kleinen Mengen bereits klassifizierter Daten auszukommen, ist es möglich, eine sogenannte Kreuzvalidierung (Cross Validation) durchzuführen. Dabei werden die Objekte in m Teilmengen eingeteilt, jeweils $m-1$ Teilmengen als Trainingsmenge und die übrige als Testmenge verwendet. Die durchschnittliche Klassifikationsgenauigkeit[2] aller möglichen Kombinationen kann als Güte des Klassifikators auf der Gesamtmenge angesehen werden.
Bayes-Klassifikatoren basieren auf den bedingten Wahrscheinlichkeiten von Attributen eines Objektes, wenn dieses in einer bestimmten Kategorie enthalten ist [Mit97]. In der Praxis wird der naive Bayes-Klassifikator etwa beispielsweise der Klassifikation von Texten oder der Interpretation von Rasterbildern genutzt; hierbei können teils gute Quoten im Bereich von 70% - 80% erreicht werden [CDF$^+$00]. Dabei kann der naive Bayes-Klassifikator nicht trivial bei kontinuierlichen Variablen eingesetzt werden, hierzu sind spezielle Verfahren notwendig [Bou05, JL95].
Werden die Trainingsobjekte als d-dimensionale Attributvektoren dargestellt, kann ein neues Objekt durch Überprüfung der Nachbarschaft kategorisiert werden. Dazu werden die k dem Objekt am nächsten liegenden Objekte ausgewählt und anhand derer Kategorien und Abstände zum Ausgangsobjekt die wahrscheinlichste Zugehörigkeit ermittelt. Der Entscheidungsprozess dieses *k-Nächste-Nachbarn-Klassifikator* genannten Klassifizierungsverfahren geschieht immer anhand der Trainingsdaten, ein tatsächlicher Gewinn von Wissen findet nicht statt. Daher ist es nötig, geeignete Datenstrukturen zu wählen, um diese möglichst effizient im Hauptspeicher oder zumindest in einer verfügbaren Datenbank zu halten [Pet09].
Mit Entscheidungsbaumklassifikatoren lässt sich wiederum tatsächliches Wissen aus den Trainingsdaten extrahieren. Ein Entscheidungsbaum ist ein endlicher Baum, dessen innere Knoten Attribute und die Blätter Klassen darstellen. Die Kanten, welche die Elemente des Baumes verbinden, stellen Tests auf dem im Vaterknoten spezifizierten Attribut dar. Hierbei kann beispielsweise getestet werden, ob der Attributwert einer bestimmten Kategorie entspricht oder ob er, bei numerischen Werten, in einem bestimmten Intervall liegt. Soll ein Objekt kategorisiert werden, wird zuerst das in der Wurzel stehende Attribut evaluiert. Je nach Ausprägung der Eigenschaft wird die entsprechende Kante zum nächsten Element des Baumes verfolgt. Da immer genau eine Kante erfolgreich ist, handelt es sich hierbei um einen deterministischen Vorgang. Wird ein Blatt erreicht, kann das Objekt der entsprechenden Kategorie zugeordnet werden. Bei der Generierung eines solchen Entscheidungsbaumes (*Growth Phase*) wird ein Attribut gewählt und anhand der Kategorien der Testdaten eine Aufteilung (*Splitting*) des Wertebereiches vorgenommen. Bei der Wahl von Attribut und Aufteilung sollten Gütewerte wie Informationsgewinn oder Gini-Index[3] beachtet werden [Mur12]. Weiterhin ist es häufig vonnöten, eine Beschneidung (Pruning Phase) des Baumes vorzunehmen und so die Fehlerrate aufgrund von schlechten Trainingsdaten zu minimieren. Dieser Effekt, bei dem der Entscheidungsbaum vor der Beschneidung eine kleinere Fehlerrate auf den Trainingsdaten, als auf der Grundgesamtheit der Daten hat, wird in der Fachliteratur als Overfitting bezeichnet.
Ein anderes Verfahren um Overfitting zu vermeiden und gleichzeitig die Genauigkeit auf den Testdaten zu erhalten, ist das Erstellen eines Zufallswaldes (Random Forest). Hierbei werden mehrere kleine Entscheidungsbäume erstellt und durchlaufen, die am häufigsten erreichte Kategorie wird dabei gewählt [Bre01].
Ein auf den neuronalen Netzen basierender Ansatz ist die Klassifikation durch Fehlerrückführung (Clas-

[2] Die Klassifikationsgenauigkeit $G_{TE}(K)$ eines Klassifikators K auf der Testmenge TE entspricht der Anzahl der richtig kategorisierten Elemente durch die Gesamtzahl der Testelemente: $G_{TE}(K) = \frac{|\{o \in TE | K(o) = C(o)\}|}{|TE|}$

[3] Gini-Index für eine Menge T von Trainingsobjekten $gini(T) = 1 - \sum_{i=1}^{k} p_i^2$, für eine Partitionierung von T: $gini(T_1, T_2, ..., T_n) = \sum_{i=1}^{m} \frac{|T_i|}{|T|} \cdot gini(T_i)$. Gibt die zu erwartende Fehlerrate oder Unreinheit an

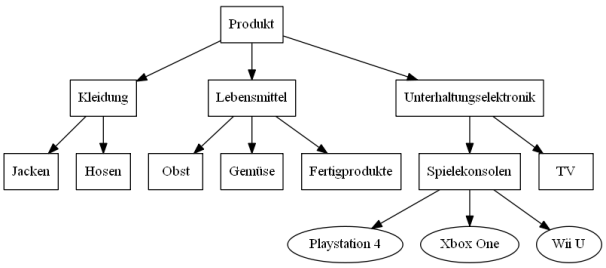


Abbildung 2.1: Beispiel für die Einteilung von Waren (Ellipse) in Warengruppen (Rechteck).

sification by Backpropagation). Bei dieser Technik wird häufig der langsame Lernprozess bemängelt. Ebenso ist das gelernte Wissen aus neuronalen Netzen oft schwer zu interpretieren, da sich die trainierten Gewichtungen dem menschlichen Verstand nur schwer erschließen. Jedoch weisen sie eine sehr hohe Toleranz gegenüber stark verrauschten Trainingsdaten auf.

2.2.3 Assoziationsanalyse

Die Assoziationsanalyse beschäftigt sich mit dem Finden häufig auftretender Teilmengen in einer Transaktionsdatenbank. Ein anschauliches Beispiel hierfür ist die sogenannte *Warenkorbanalyse*. Jeder Einkauf (Transaktion) ist eine Menge von Artikeln (Items). Durch die Untersuchung auf häufig auftretender Teilmengen ist es nun möglich, Zusammenhänge zwischen Artikeln zu finden. So ist es beispielsweise möglich, eine Assoziationsregel zu finden, die besagt, dass Kunden die Kuchen kaufen mit hoher Wahrscheinlichkeit auch Kaffee kaufen. Das auf diese Weise gewonnene Wissen kann etwa in Marketing, Katalogdesign oder bei der Planung von Verkaufsräumen genutzt werden.
Eine einfache Methode der Assoziationsanalyse ist der Apriori-Algorithmus, basierend auf *Support* und *Konfidenz* von Teilmengen. Unter dem Support einer Menge X ist der Anteil von X in der Transaktionsdatenbank zu verstehen. Sind insgesamt 100 Transaktionen in der Datenbank vorhanden und zwei davon enthalten sowohl Kaffee als auch Kuchen, so ist der Support der Menge {Kaffee, Kuchen} in der Transaktionsdatenbank 2%. Hinter der Konfidenz einer Assoziationsregel $X \rightarrow Y$ verbirgt sich der Anteil der Transaktionen, die Y enthalten, aus der Teilmenge der Transaktionsdatenbank, die X enthalten. Gehen wir vom letzten Beispiel aus und sagen, es existieren zwei weitere Mengen die Kaffee enthalten, jedoch keinen Kuchen, so hat die Assoziationsregel *Kaffee* $\rightarrow$ *Kuchen* eine Konfidenz von 50%. Ziel des Apriori-Algorithmus ist es nun, sämtliche Assoziationsregeln zu finden, die mindestens einen gewissen Support[4] und eine gewisse Konfidenz haben. Im ersten Schritt sind sämtliche häufige[5] Teilmengen zu finden. Aufgrund der Monotonieeigenschaft, dass jede Teilmenge eines häufig auftretenden Itemsets ebenfalls häufig sein muss, ist es nicht nötig alle Teilmengen zu untersuchen und das Problem ist so effizient lösbar. Aus allen gefundenen Teilmengen X können nun Assoziationsregeln der Form $A \rightarrow (X - Y)$ gebildet werden, wobei gilt: $Y \subset X$. Von Interesse sind die Assoziationsregeln, die mindestens eine gewisse Konfidenz haben [AS$^+$94].
In der Praxis sind die Ergebnisse des Apriori-Algorithmus in Abhängigkeit vom geforderten, minimalen Support oft wenig nützlich. Während bei einem hohen Minimalsupport meist wenige einfache Assoziationsregeln gefunden werden, ist die Anzahl bei einem geringen Wert oft unüberschaubar. Daher kann es von Vorteil sein, Items hierarchisch zu organisieren, dies wird als Item-Taxonomie oder *is-a Hierarchie* bezeichnet. Artikel eines Händlers können so in Warengruppen zusammengefasst werden, etwa wie in Abbildung 2.1. Eine mögliche Regel wäre beispielsweise Spielekonsole $\rightarrow$ TV, statt Wii U $\rightarrow$ Samsung TV 123. Assoziationsregeln auf dieser Ebene sind im Allgemeinen aussagekräftiger. Der entsprechende

[4] Analog zum Support einer Teilmenge X entspricht der Support einer Assoziationsregel $X \rightarrow Y$ dem Support von $X \cup Y$

[5] Dem Algorithmus werden als Parameter die minimalen Werte für den Support und Konfidenz übergeben. Teilmengen, die unter diesen Werten liegen, sind irrelevant.

Modell	Jahr	Farbe	Verkäufe
Chevy	1990	rot	5
Chevy	1991	weiss	87
...	...	...	...
Ford	1990	blau	63
Ford	1992	blau	39

Tabelle 2.1: *Sales* nach [GCB+97]. Die Dimensionen sind Modell, Jahr und Farbe. Die Anzahl der Verkäufe ist das einzige Maß.

Algorithmus ist eine Erweiterung des Apriori-Algorithmus. Dabei werden für jedes Item noch zusätzlich die übergeordneten Items zur jeweiligen Transaktion hinzugefügt.
Neben dem bloßen Vorhandensein eines Items in einer Transaktion können auch Informationen über Quantitäten von Belang sein. Käufer eines PKW kaufen im Allgemeinen vier Winterreifen. Statt der Anzahl kann ein beliebiges numerisches oder kategorisches Attribut betrachtet werden. Andrew Pole von der amerikanischen Einzelhandelskette *Target Corporation* fand beispielsweise heraus, dass sich durch den häufigen Kauf bestimmter Produkte der Geburtstermin bei einer werdenden Mutter auf ein kleines Zeitfenster einschätzen lässt [Hil12, Duh12]. Auch dieses Problem lässt sich durch einen modifizierten Apriori-Algorithmus lösen [SA96].

2.2.4 Generalisierung

Generalisierungsverfahren ermöglichen es, große Datenmengen in einer kompakten und abstrakten Weise zu beschreiben. Damit sind sie einerseits für den menschlichen Benutzer zugänglicher, auf der anderen Seite kann ein auf diese Weise gekürzter Datenbestand als Grundlage für weitere Data-Mining-Schritte dienen. Häufig werden in Verbindung hiermit *Data Warehouse*[6]-Techniken genannt. Insbesondere Datenwürfel (*Data Cubes*) sind geläufig. Dabei handelt es sich um ein multidimensionales Datenmodell, welches auf relationalen Datenbanken basiert. Als Dimensionen von Daten werden hierbei unabhängige Attribute bezeichnet, während abhängige Attribute Maße genannt werden. Um die Notwendigkeit von Datenwürfeln zu demonstrieren soll Tabelle 2.1 als Grundlage dienen. Intention der Entwicklung von Datenwürfeln waren unter anderem Defizite bei der Aggregation mit dem GROUP BY-Befehl von SQL. So fehlen etwa die Unterstützung von Histogrammen, Roll-Up-Summen und Zwischensummen für Drill-Downs (*Roll-up Totals and Sub-Totals for drill-downs [GCB+97]*).
Histogramme sind die Aggregation von Werten über erzeugten Kategorien. So können beispielsweise geographische Länge und Breite auf Nationen abgebildet werden, welche wiederum als Kategorien dienen. Auf Kategorien wiederum kann eine Hierarchie definiert werden, etwa Region → Nation → Stadt → Straße.
Zur Auswertung von Prozessen müssen häufig Sichten auf verschiedenen Ebenen erstellt werden. Im zuvor beschriebenen PKW-Beispiel ist sicherlich die Gesamtzahl der verkauften Fahrzeuge von Interesse, allerdings kann es auch von Bedeutung sein, wie viele rote Fords insgesamt verkauft wurden, oder etwa nur rote Fahrzeuge. Grundsätzlich lässt sich dieses Problem mit GROUP BY lösen, in der Praxis entstehen bei einem umfassenden Überblick allerdings lange Abfragen. Hinter den zuvor genannten Begriffen *roll-up* und *drill-down* versteht ist das auf- bzw. absteigen zwischen diesen Ebenen zu verstehen.
Datenwürfel sind im SQL-Standard aufgenommen worden [sql06] und werden bereits von einigen Datenbankmanagementsystemen unterstützt [Mic14, Ora14, Gro14].

[6]Data Warehouses sind subjektorientierte, integrierte, zeitvariante und nichtflüchtige Datensammlungen aus verschiedenen Quellen, die der Unterstützung von Entscheidungsprozessen dienen [Pon10, CD97].

2.2.5 Weitere Verfahren

Je nach Quelle werden weitere Kategorien von Verfahren genannt. Häufig kommt es dabei zu Überschneidungen mit den bereits genannten Techniken, teilweise auch untereinander.
Mit *Regression* können Attribute von Datenobjekten vorhergesagt, beziehungsweise abgeschätzt werden. Hierbei handelt es sich nicht, wie bei der Klassifikation, um kategorische Attribute, sondern um numerische. Ein Beispiel von Witten et al. [WF00] ist die Abschätzung der Performance eines Computersystems in Punkten anhand von numerischen Attributen wie der Größe des Hauptspeichers, des Caches, etc. Diese geschieht mit Hilfe einer linearen Regressionsgleichung[7] der Form $x = \sum_{i=0}^{k} w_i a_i$, wobei x dem abzuschätzenden Attribut und a_i[8] den numerischen Attributen des Objekts entsprechen. Bei w_i handelt es sich um Wichtungen, welche errechnet werden müssen. Dazu muss die Differenz zwischen errechnetem und tatsächlichem Attribut x für jede Instanz der Trainingsdaten minimiert werden[9]. Eine weitere Möglichkeit für die Abschätzung der Leistungspunkte ist das Erstellen eines Regressionsbaumes. Im Unterschied zu den bekannten Arten von Entscheidungsbäumen finden sich in den Blättern keine festen Werte, sondern durchschnittliche Werte. Ähnlich dazu ist der Modellbaum (*model tree*); hier enthalten die Blätter wiederum ein lineares Regressionsmodell. Ebenfalls existieren nichtlineare Regressionsmethoden, etwa Stützvektormaschinen (*Support Vector Machines*); diese lassen sich auch zur Kategorisierung einsetzen.
Verfahren, die sich mit Objekten beschäftigen, die sich mit der Zeit verändern, werden der *Evolutionsanalyse* zugeschrieben. So können Trends, Zyklen, saisonale Erscheinungen oder irreguläre Veränderungen erkannt und vorhergesagt werden. Beispiele sind die Trendanalyse auf dem Aktienmarkt oder der saisonale Einfluss[10] auf den Einzelhandel. Verwendete Techniken sind gleitende Mittelwerte, die kleinste-Quadrate-Methoden, Saisonindizes und weitere.
Mithilfe der Außreißeranalyse (Outlier Analysis) werden starke Abweichungen von den „normalen“ Daten gesucht. So können etwa Messfehler in einem Experiment, Unregelmäßigkeiten in Maschinen entdeckt oder gezielt nach Betrügern, etwa in Bezug auf Kreditkarten gesucht werden. Hierfür gibt es statistische Verfahren, welche Annahmen über die Verteilung der Daten benötigen, distanz-basierte Verfahren, welche Ausreißer anhand verhältnismäßig großer Abstände ausmachen [KNT00] und abweichungs-basierte Verfahren. Zum Finden letzterer existieren geeignete Verfahren mithilfe von Datenwürfeln.

[7] Witten et al. erläutern an späterer Stelle, dass die Lösung mit linearer Regression vergleichsweise schlecht ist, da sich die Daten nur bedingt auf ein lineares Modell abbilden lassen.

[8] a_0 ist hierbei immer 1

[9] $\sum_{i=0}^{n}(\mathcal{X}^{(i)} = \sum_{j=0}^{k} w_j a_j^{(i)})^2$ muss minimiert werden. Die Notation $a_j^{(i)}$ besagt, dass das Attribut a_j zur i-ten Instanz der Trainingsdaten gehört.

[10] Zum Beispiel Feiertage, wie Valentienstag oder Weihnachten, haben offensichtlich einen starken Einfluss auf das Kaufverhalten.

Kapitel 3

Eignung von Data-Mining-Methoden

3.1 Auswahlverfahren

Im Folgenden sollen nun einige Verfahren auf Eignung zur automatischen Auswahl der Privatsphäreeinstellungen überprüft werden. Um entsprechende Einstellung für ein gegebenes Attribut zu finden, ist es notwendig, semantisch äquivalente Attribute aus anderen Schemata zu finden, für die Einstellungen bekannt sind.
Die naheliegendste Herangehensweise für das Problem ist die *Klassifikation*. Anhand der Eigenschaften bekannter Attribute können Klassifikatoren gelernt werden, die Objekte in Klassen entsprechend der Datenschutzeinstellungen[1] einordnen. Die Klassifikation kann durch *Clustering* vorbereitet werden. In einem Cluster befinden sich solche Attribute, die semantisch ähnlich sind. Idealerweise beschreiben sie den gleichen Sachverhalt. Jedem Cluster wird nun eine Privatspähreeinstellung zugewiesen, die sich aus den Elementen ergibt. Bei der Klassifikation wird nun ein Objekt in ein Cluster eingeordnet und dessen Einstellung übernommen. Der Vorteil dieses Umwegs liegt darin, dass der Erkenntnisgewinn für einen Nutzer, der das System wartet, größer ist. Zum einen ist ersichtlich, welche Attribute das System als ähnlich zueinander betrachtet. Es ist leicht zu visualisieren und nachzuvollziehen, dass eine bestimmte Einstellung vorgenommen wurde, weil das System ein Attribut einem bestimmten Cluster zugeordnet hat. Anzunehmen ist, dass diese Algorithmen besser funktionieren, wenn sie die Daten in die Cluster einordnen, da diese der natürlichen Struktur der Daten entsprechen. Li und Clifton [LC94] beschreiben ein Verfahren, welches zuerst die Attribute mit Kohonennetzen clustert und danach ein neuronales Netz mit den selben Daten trainiert. Auf diese Weise kann ein vollständiges Schema-Mapping mit minimalem Eingreifen realisiert werden.
Der Nutzen einer *Assoziationsanalyse* ist fragwürdig, höchstwahrscheinlich kann diese nur unterstützend zur Klassifikation eingesetzt werden. Hierbei würden Schemata Transaktionen und einzelne Attribute Items entsprechen. Da nicht davon auszugehen ist, dass gleiche Attribute in unterschiedlichen Schemata intuitiv als gleich erkannt werden, müssen ähnliche Items durch Clustering zusammengefasst werden. Die Assoziationsanalyse kann nun häufig im gleichen Schema auftretende Attribute identifizieren.
Die weiteren genannten Verfahren haben intuitiv keinen Nutzen beim Herleiten der Datenschutzeinstellungen. Weil es am plausibelsten scheint, eine Klassifikation durchzuführen, soll dies in der anschließenden Bachelorarbeit umgesetzt werden. Optional kann untersucht werden, ob ein Clustering der Daten im Vorfeld tatsächlich Vorteile bringt.

[1] Nach aktuellem Stand der Privacy Policies sind das public, confidential, secret und top-secret [Gru13].

3.2 Algorithmen im Vergleich

Es sollen nun passende Klassifikationsverfahren ermittelt werden. Da zum jetzigen Zeitpunkt nicht ausreichend geklärt ist, mit welchen Daten die Attribute charakterisiert werden, kann noch keine konkrete Festlegung getroffen werden. Zu erwarten ist die Darstellung als mehrdimensionaler Vektor, ähnlich wie es in SemInt [LC00] der Fall ist. Hierbei werden normalisierte Metadaten über das Attribut, etwa Datentyp, Schlüsseleigenschaften, Gestattung von Nullwerten und auch Aussagen über den Wertebereich, etwa Minimum, Maximum oder Median genutzt. Aussagen über die Dimension, Abhängigkeiten der Eigenschaften untereinander, Rauschen und Umfang der Trainingsdaten können nicht getätigt werden. Diese sind bei der Auswahl jedoch unabdingbar.
Naive Bayes-Klassifikatoren sind möglich, wenn die Eigenschaften der Attribute unabhängig oder stark abhängig sind, im Bereich zwischen diesen Extremen ist keine überragende Leistung zu erwarten [Ris01]. Weiterhin wird die Gesamtleistung von Naiven Bayes-Klassifikatoren nur gering von der Dimension der Daten beeinflusst, allerdings ist die Leistung durchweg schlechter als bei anderen Verfahren [CKY08].
Die Nächste-Nachbarn-Methode scheint intuitiv ein passendes Verfahren zu sein. Es werden Objekte in der näheren Umgebung überprüft und die häufigste Klasse übernommen. Wenn die Objekte in einer Datenbank mit geeignetem Index vorliegen, kann diese Suche effizient bewältigt werden. Problematisch wird die Nächste-Nachbarn-Methode allerdings, wenn die Dimension der Objekte zu groß wird. Beyer et al. [BGRS99] führen hier maximal 10 bis 15 Dimensionen an, danach ist der Kontrast in den Daten zu gering und das Verfahren wird ungenau. Außerdem ist hierbei eine gewisse Menge an Trainingsdaten vonnöten. Der Vorteil dieser Methode ist, dass im Vorfeld definitiv kein Clustering vonnöten ist und das Verfahren schnell mit neuem Wissen arbeiten kann, ohne dass der Lernprozess wiederholt werden muss. Dies ist etwa der Fall, wenn ein neuer Sensor mitsamt zugehöriger Datenschutzeinstellungen in das System eingepflegt wurde oder eine gelernte Einstellung durch einen Nutzer nachträglich bestätigt oder angepasst wurde.
Die Fehlerrückführung (Back Propagation) bietet eine gute Erkennungsrate nach einer längeren Lernphase und hat keine Probleme mit großen Dimensionen. Es wird bereits erfolgreich von Li und Clifton zum Klassifizieren von Attributen eingesetzt [LC00]. Hierbei handelt es sich um das einzige Verfahren, für das Erfahrungsberichte in einem ausreichend ähnlichen Anwendungsfall zu finden waren.
Entscheidungsbäume, beziehungsweise deren Weiterentwicklung Zufallswälder sind schnell, zuverlässig und durch einen menschlichen Nutzer interpretierbar. Ein großer Vorteil ist, dass keine Parameter erwartet werden, von denen das Ergebnis der Klassifikation abhängig ist.

3.3 Festlegung auf Assoziationsanalyse

In Absprache mit Betreuer und Gutachtern dieser Arbeit wurde sich, entgegen der Ausführungen des letzten Abschnitts, für die Assoziationsanalyse entschieden. Ein angestrebtes Klassifikationsverfahren, welches anhand der Strukturinformationen der Attribute eine Zuordnung vornimmt, verspricht wenig Mehrwert, da diese Informationen bereits beim *Similarity Flooding* genutzt werden. Jedoch kann die Ausgabe in Form von unvollständigen Mappings der Attribute bereits die Grundlage für eine Assoziationsanalyse bieten. Sind sich außerdem Attribute ausreichend ähnlich genug, können diese auf dasselbe Item abgebildet werden.

Kapitel 4

Similarity Flooding

4.1 Schema Mapping durch Similarity Flooding

Wie bereits in Abschnitt 1.3 erläutert, ist das Data-Mining erst der zweite Schritt bei der Herleitung von Datenschutzeinstellungen. Diesem geht bereits ein Schemaintegrationsprozess zuvor. Da dessen Ausgaben als Ausgangspunkt des Data-Minings dienen, soll an dieser Stelle das verwendete Verfahren *Similarity Flooding* erklärt werden.
Similarity Flooding wurde 2002 von Melnik, Garcia-Molina und Rahm veröffentlicht [MGMR02]. Das Verfahren besteht aus vier Teilschritten. Zu Beginn werden die beiden ineinander zu überführenden Schemata jeweils in eine geeignete Darstellung als gerichteter, beschrifteter Graph konvertiert, dieser soll im Folgenden als das Modell des Schemas bezeichnet werden. Dabei wird keine konkrete Überführungsform vorausgesetzt, laut den Autoren bietet sich etwa das Open Information Model an [BBC+99]. Abbildung 4.1 soll demonstrieren, wie das in Listing 4.1 gezeigte Schema in ein solches Modell überführt werden kann.

```
CREATE TABLE Personnel (
        Pno int,
        Pname string,
        Dept string,
        Born date,
        UNIQUE pkey(Prio)
)
```

Listing 4.1: Schema S_1 nach [MGMR02]

Ausgehend von den Modellen der Schemata wird nun ein paarweiser Konnektivitätsgraph (Pairwise Connectivity Graph, PCG) erzeugt. Dieser ist für zwei Modelle A und B wie folgt definiert:

$$((x,y),p,(x',y')) \in PCG(A,B) \leftrightarrow (x,p,x') \in A \wedge (y,p,y') \in B \tag{4.1}$$

Hierbei ist p die Beschriftung der gerichteten Kanten. Aus dem paarweisen Konnektivitätsgraph wird nun ein induzierter Verbreitungsgraph (induced propagation graph) gebildet. Dies geschieht, indem für jede gerichtete Kante des Konnektivitätsgraphen eine Kante in Gegenrichtung hinzugefügt wird. Gleichzeitig werden die Beschriftungen durch einen Verbreitungskoeffizienten (propagation coefficient) $w((x,y),(x',y')) \in [0,1], ((x,y),p,(x',y')) \in PCG(A,B)$ ersetzt. Für die Berechnung des Verbreitungskoeffizienten wird kein konkretes Verfahren vorausgesetzt. Die Autoren verwenden die Anzahl der gleichnamigen ein- beziehungsweise ausgehenden Kanten im Konnektivitätsgraphen und bilden davon das Reziproke. Zusätzlich wird auf weitere Verfahren verwiesen [MGMR01].

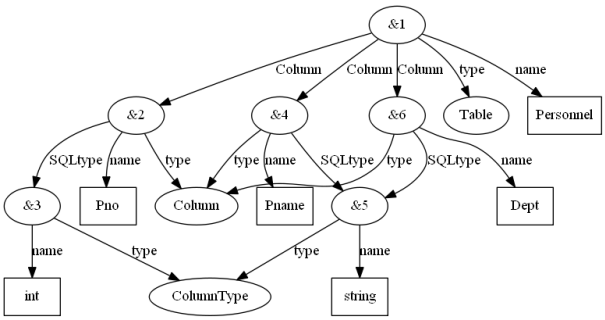


Abbildung 4.1: Konvertierung von S_1 (Listing 4.1) in einen Graphen nach [MGMR02] (unvollständig)

Variation	Fixpunktberechnung
Basic	$\sigma^{i+1} = normalize(\sigma^i + \phi(\sigma^i))$
A	$\sigma^{i+1} = normalize(\sigma^0 + \phi(\sigma^i))$
B	$\sigma^{i+1} = normalize(\phi(\sigma^0 + \sigma^i))$
C	$\sigma^{i+1} = normalize(\sigma^0 + \sigma^i + \phi(\sigma^0 + \sigma^i))$

Tabelle 4.1: Variationen der Fixpunktberechnung

Der dritte Schritt ist die Berechnung der Ähnlichkeit der Elemente zweier Modelle A und B.

$$\sigma : A \times B \to \mathbb{R} \tag{4.2}$$

σ ist eine totale Abbildung, welche in einer iterativen Fixpunktberechnung berechnet wird. Dabei ist σ^i die Abbildung nach der i-ten Iteration. Für die Initialisierung von σ^0 schlagen die Autoren vor, eine Metrik der Namen der Elemente zu wählen, etwa die Levenshtein-Distanz oder der Vergleich von Prä- und Suffixen. Davon ausgehend werden nun mithilfe von Formel 4.3 weitere Abbildungen berechnet, bis die Veränderung unter einen zuvor gewählten Schwellwert ϵ fällt ($\delta(\sigma^n, \sigma^{n+1}) < \epsilon$) oder eine zuvor gewählte Anzahl von Iterationen durchgeführt wurde.

$$\begin{aligned}\sigma^{i+1}(x,y) = \sigma^i(x,y) + & \sum_{(a_u,p,x)\in A,(b_u,p,y)\in B} \sigma^i(a_u,b_u)\cdot w((a_u,b_u),(x,y)) + \\ & \sum_{(x,p,a_v)\in A,(x,p,b_v)\in B} \sigma^i(a_v,b_v)\cdot w((a_v,b_v),(x,y))\end{aligned} \tag{4.3}$$

Für die Konvergenzeigenschaften der Berechnung soll an dieser Stelle auf die Originalquellen verwiesen werden [BBC+99, MGMR01]. Der Berechnung liegt die Annahme zugrunde, dass die Ähnlichkeit zweier Elemente sich auch auf die Ähnlichkeit der Nachbarn auswirkt. Tabelle 4.1 zeigt Varianten der Fixpunktberechnung mit Einbeziehung von Normalisierung. Melnik et al. haben in Versuchen nachgewiesen, dass die Variation C häufig am schnellsten konvergiert.

Nach Abschluss dieses Vorgangs liegt nun eine Abbildung σ vor, deren 2^n Teilmengen potentiell gute Abbildungen zwischen den Modellen der Schemata darstellen. Daher ist der letzte Schritt nun die Filterung einer bestmöglichen Teilmenge. Dies geschieht zum einen durch *Typ- und Kardinalitätsbeschränkungen* (type/cardinality constraints). Weil der Zweck des Verfahrens das Finden von Gegenstücken zu etwa Tabellen und Spalten ist, können andere, uninteressante Typen an dieser Stelle durch eine *Typbeschränkung* herausgefiltert werden. Ebenso kann festgelegt werden, dass zu jedem Element eines gewählten Schemas genau eine Verknüpfung in das andere Schema gewählt werden soll. Auf diese Weise lässt sich die Anzahl der Teilmengen erheblich reduzieren.

Die verbleibenden Abbildungen entsprechen einem Problem der stabilen Hochzeit (stable marriage problem) und mit den entsprechenden Verfahren lässt sich auch hier der passendste Kandidat finden [GS62].

Das Problem der stabilen Hochzeit lässt sich wie folgt erläutern: Für n Frauen und ebenso viele Männer liegen die Präferenzen für potentielle Partner des jeweils anderen Geschlechts vor. Dies lässt sich einfach als gewichteter, bipartiter Graph darstellen. Nun gilt es, eine stabile Heirat von Männern und Frauen zu finden. Eine solche ist als vollständige Paarung aller Männer und Frauen zu verstehen, für die keine zwei Paare (x,y) und (x',y') existieren, sodass x y' gegenüber y bevorzugt und gleichzeitig die Präferenz von y' zu x höher ist als zu x'.

4.2 Implementierung

Eine Java-Implementierung des Similarity Flooding Algorithmus liegt zur Entstehungszeit dieser Arbeit vor und soll genutzt werden. Im Mittelpunkt steht die Klasse `SimilarityFlooding`. Der Konstruktor erwartet bei der Erzeugung eines Objektes zwei Graphen der Klasse `Graph<Node>` als Eingabeparameter. Nach einem Aufruf der Methode `computeSimilarity()` wird der Similarity-Flooding-Algorithmus durchgeführt. Das Ergebnis wird in `Map<NodePair, Double> finalMapping` gespeichert, was mittels `getFinalMapping()` abrufbar ist. Diese Datenstruktur gibt für jedes Knotenpaar (Typ `NodePair`) die Ähnlichkeit als `Double` zurück.
Beim `finalMapping` handelt es sich bereits um eine gefilterte Menge von Ähnlichkeitswerten. Es kann allerdings auch interessant sein, die vollständige Funktion σ nach der letzten Iteration des Algorithmus zu kennen. Die Methode `getSigmaMapping()` schafft hierbei Abhilfe. Sie gibt, genau wie `getFinalMapping()`, eine Datenstruktur vom Typen `Map<NodePair,Double>` zurück.
Zur Konvertierung eines Datenbankschemas in einen Graphen existiert das Interface `IGraphConverter`. Dieses spezifiziert die Methode `getGraphRepresentation()`, welche ein Objekt der Klasse `Graph<Node>` erzeugt, um als Eingabeparameter für `SimilarityFlooding` zu dienen. Es wird durch die Klasse `RelationalGraphConverter` implementiert. Diese ist in der Lage, sich nach Angabe von Treiberinformationen und Zugangsdaten mit einem Datenbankmanagementsystem zu verbinden und entsprechenden Graphen für ein Schema zu erzeugen[1]. Die Methode `setFormula()` ermöglicht die Auswahl einer Berechnungsformel entsprechend Tabelle 4.1.

4.3 Beispielanwendung und Probleme mit der Implementierung

Um sich mit der Implementation vertraut zu machen, wurde ein Beispiel konstruiert. Der Tabelle *lamp* im Datenbankschema *Musama* wurde ein weiteres Attribut mit der Bezeichnung *EventType* hinzugefügt. Gleichnamige Attribute existieren bereits in weiteren Tabellen des Schemas.

```
CREATE TABLE IF NOT EXISTS `lamp` (
  `id` varchar(100) NOT NULL,
  `timestamp` varchar(100) NOT NULL,
  `DimValue` varchar(100) NOT NULL,
  `EventMillis` varchar(100) NOT NULL,
  `EventType` varchar(100) NOT NULL,
  `PowerStatus` varchar(100) NOT NULL,
  PRIMARY KEY (`id`,`timestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Listing 4.2: Die Tabelle *lamp* mit neuem Attribut *EventType*

Im ersten Versuch wurde eine Kopie des Schemas *Musama* angelegt und die Tabelle *lamps* angepasst. Während die Konstruktion beider Graphen auf dem verwendeten System zuverlässig in weniger als einer

[1] Zum Zeitpunkt dieser Arbeit gehören MySQL und PostgreSQL zu den unterstützten Systemen.

Sekunde funktionierte, terminierte der Similarity-Flooding-Algorithmus nicht in annehmbarer Zeit. Die Komplexität von Similarity-Flooding variiert von $\mathcal{O}(m \cdot n)$ im Durchschnitt bis $\mathcal{O}(m^2 \cdot n^2)$ im schlimmsten Fall, wobei m und n der Anzahl der Knoten in den Eingabegraphen entsprechen. Die Graphenrepräsentation des *Musama*-Schemas hat 231 Knoten und ist damit im Verhältnis zu anderen Datenbanken nicht sonderlich groß. Ob die schlechte Laufzeit seine Ursache in der Implementation oder im Verfahren selbst hat, kann an dieser Stelle nicht geklärt werde, da kein Vergleich vorliegt.
Um den Versuch in annehmbarer Zeit durchzuführen wurde die Kopie von *Musama* auf die Tabelle *lamp* (Listing 4.2) reduziert und der Prozess wiederholt. Das Matching der 231 Knoten des *Musama*-Schemas und der 21 Knoten von *lamp* konnte innerhalb weniger Sekunden durchgeführt werden. Tabelle 4.2 zeigt die Ergebnisse; Mappings von Knoten, die Meta-Daten, wie Namen oder Datentypen repräsentieren, wurden dabei ausgelassen. Da selbst absolut identische Strukturen nicht erkannt werden konnten, handelt

musama	lamp	Ähnlichkeit
Tabelle recording_ubisense	Tabelle lamp	0.1783
Spalte recording_roomusage_events.EventType	Spalte lamp.EventType	0.0028
Primärschlüssel recording_roomusage_events.timestamp	Primärschlüssel lamp.timestamp	0.0035

Tabelle 4.2: Erkannte Mappings bei der Integration einer veränderten Tabelle *lamp*

es sich offensichtlich um kein gutes Mapping. Auch hier ist nicht ersichtlich, ob sich dies durch die Implementierung oder das Verfahren an sich begründen lässt. Selbst bei dem Versuch, Mapping zwischen Ausgangsschema und einer einzelnen identischen Tabelle zu finden, liefert kein befriedigendes Ergebnis. Auch in Versuchen mit anderen Tabellen werden *lamp*, *eibgateway* und *metadata* immer fälschlicherweise höchste Ähnlichkeit zur Tabelle *ubisense* zugesprochen. Auch die Zuordnungen der Attribute ist nur in Einzelfällen korrekt. Durch Variation der Formel oder der maximalen Anzahl der Iterationen[2] konnten sich die Ergebnisse nicht verbessern lassen.
Weitere Probleme der Implementierung finden sich in der Klasse *RelationalGraphConverter*. Aus nicht weiter erkenntlichen Gründen wurden bei der Konstruktion eines Graphen aus dem Musama-Schema auf dem Datenbanksystem des Lehrstuhls DBIS Tabellen ausgelassen, was dazu führte, dass der resultierende Graph nicht vollständig und damit unbrauchbar war. Ferner wurden einige Datentypen noch nicht unterstützt, dringend benötigte wurden vom Betreuer dieser Arbeit kurzfristig nachgepflegt. Aufgrund dieser nicht gerade geringen Anzahl von Problemen mit der Implementierung und der Zeit, die zur Bewältigung aufgebracht wurde und noch werden müsste, soll das Verfahren in allen weiteren Schritten ignoriert werden und konstruierte Beispiele als Grundlage weiterer Arbeiten dienen.

[2]Hierfür wurde in den Quelltext der Implementierung eingegriffen.

Kapitel 5

Assoziationsanalyse

5.1 Finden von häufigen Itemsets

Die Vielzahl der Verfahren zur Suche von Assoziationsregeln lässt sich in zwei Teilschritte zerlegen: Dem Finden von häufig zusammen auftretenden Objekten (Items) und dem anschließenden Ableiten von Assoziationsregeln aus dieser Menge.
Im Vorfeld der Erläuterungen sollen erst einige Begrifflichkeiten vereinbart werden. Sei $I = \{i_1, i_2, ..., i_n\}$ die Menge der Items, außerdem soll auf I eine lexikographische Ordnung $i_1 < 1_2 < ... < i_n$ definiert sein. Eine Transaktionsdatenbank $D = \{< TID_1, T_1 >, < TID_2, T_2 >, ..., < TID_m, T_m >\}$ ist die Menge von Tupeln aus Transaktionsidentifikatoren und den dazugehörigen Transaktionen. Dabei gilt außerdem $\forall i \in [1, m] : T_i \subseteq I$. Außerdem stellt TID_i einen Identifikator der jeweiligen Transaktion dar. Mit L_k wird die Menge der k-elementige Itemmengen bezeichnet, die den Supportbedingungen genügen. Dies ist der Fall, wenn die Anzahl der Elemente der Itemmenge mindestens dem zuvor festgelegten Mindestsupport *minsup* entspricht. Daher wird hierbei von *häufigen k-Itemsets* gesprochen.

5.1.1 Der Apriori-Algorithmus und seine Varianten

In Abschnitt 2.2.3 wurde der Apriori-Algorithmus bereits grob umrissen. Agrawal et al. stellten die Verfahren *Apriori*, *AprioriTid* und *AprioriHybrid* als Verbesserungen der zuvor bekannten Verfahren vor, etwa *AIS* [AIS93] und *SETM* [HS95] vor [AS$^+$94, AMS$^+$96]. Sie ermöglichen eine effiziente Suche nach häufigen Itemsets in einer Transaktionsdatenbank. Da es sich beim Algorithmus Apriori und seinen Varianten um die bekanntesten Verfahren ihrer Art handeln, sollen diese hier nun kurz vorgestellt werden. Dabei werden insbesondere auch die Nachteile angesprochen, die dazu führen, dass statt einer Apriori-Variante die Wahl auf das anschließend vorgestellte Verfahren *FPGrowth* gefallen ist.

```
L_1 = {Large 1-itemsets};
for (k = 2; L_{k-1} ≠ ∅; k++) do begin
        C_k = apriori_gen(L_{k-1});
        forall transactions t ∈ D do begin
                C_t = subset(C_k, t);
                forall candidates c ∈ C_t do
                        c.count++;
                end
                L_k = {c ∈ C_k | c.count ≥ minsup}
end
Answer = ⋃_k L_k
```

Listing 5.1: Der Apriori-Algorithmus nach [AS+94]

Der Apriori-Algorithmus (Listing 5.1) fasst zu Beginn alle mindestens *minsup*-mal auftretenden Items zu L_1 zusammen. Solange $L_k \neq \emptyset$ werden nun die häufigen k-elementigen Itemsets wie folgt gebildet: Aus den häufigen k-Itemsets in L_{k-1} wird die Kandidatenmenge C_k generiert. Alle Itemsets $p, q \in L_{k-1}$, die sich nur in ihrem letzten Element unterscheiden, werden zu einem k-elementigen Itemset vereinigt. Die resultierenden Mengen werden C_k hinzugefügt (Listing 5.2). Dieser Schritt wird in der Literatur als *Join Step* bezeichnet.

```
insert into C_k
select   p.item_1, p.item_2,
         ...
         p.item_{k-1}, q.item_{k-1}
from L_{k-1}p, L_{k-1}q
where p.item_1=q.item_1,
      p.item_2=q.item_2,
      p.item_{k-1} < q.item_{k-1}
```

Listing 5.2: Generierung der Kandidatenmenge $C_k = apriori\_gen(L_k)$ (Join Step) nach [AS+94]

Aus der Kandidatenmenge werden anschließend alle Itemsets entfernt, die nicht-häufige k-1-Itemsets enthalten. Dieser Schritt wird als *Prune Step* bezeichnet (Listing 5.3).

```
forall itemsets c ∈ C_k do
   forall (k-1)-subsets s of c do
      if(s ∉ L_{k-1} then)
         delete c from C_k;
```

Listing 5.3: Generierung der Kandidatenmenge $C_k = apriori\_gen(L_k)$ (Prune Step) nach [AS+94]

Nachdem nun die Kandidatenmenge C_k erzeugt wurde, werden aus dieser solche Teilmengen entfernt, die in den Transaktionen der Transaktionsdatenbank nicht häufig vorkommen[1]. Übrig bleibt die Menge L_k. Die Menge der häufigen Itemsets und damit die Ausgabe von *Apriori* ist die Vereinigung aller häufiger k-Itemsets $\bigcup_k L_k$. Problematisch bei diesem Verfahren ist es, dass die Transaktionsdatenbank immer dann durchlaufen werden muss, wenn der Support eines k-Itemsets überprüft werden soll. Dies kann bei großen Datenbanken oder leistungsschwachen Systemen dazu führen, dass die Leistung nicht mehr tragbar ist.

[1] Um effizent überprüfen zu können, ob eine C_k eine Teilmenge in einer Transaktion T_i enthält, empfehlen die Agrawal et al. die Speicherung der Kandidatenmenge in einem Hash-Baum.

```
L_1 = {Large 1-itemsets};
C̄_1 = database D
for (k = 2; L_{k-1} ≠ ∅; k++) do begin
        C_k = apriori_gen(L_{k-1});
        C̄_k = ∅
        forall entries t ∈ C̄_{k-1} do begin
                C_t = {c ∈ C_k|(c - c[k]) ∈ t.setOfItems};
                forall candidates c ∈ C_t do
                        c.count++;
                if (C_t ≠ ∅) then C̄_k+ =< t.TID, C_t >;
        end
                L_k = {c ∈ C_k|c.count ≥ minsup}
end
Answer = ⋃_k L_k
```

Listing 5.4: Algorithmus AprioriTID nach [AS+94]

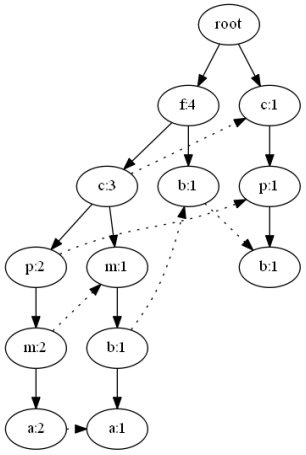
Aus diesem Grund haben Agrawal et al. *AprioriTID* (Listing 5.4) eingeführt, welches die zusätzliche Menge $\overline{C}_k = \{< TID_i, \{t_{ik}\} >\}$ einführt. Dabei entsprechen TID_i den Transaktionsidentifikatoren in D und t_{ik} den Kandidaten für ein k-Itemset, die in der mit TID_i identifizierten Transaktion vorkommen. $\overline{C}_1$ wird dabei beim ersten vollständigen Lesen von D generiert. Intuitiv wird durch Zählen der Items in $\overline{C}_1$ L_1 berechnet, analog zum *Apriori* danach C_2. Für $k \geq 2$ wird nun jeweils $\overline{C}_k$ erzeugt. Dazu wird für jeden Eintrag $< TID_i, t_{ik} > \in \overline{C}_{k-1}$ eine eigene Kandidatenliste $C_{TID_i} = \{c \in C_k | (c - c[k]) \in t_{ik} \wedge (c - c[k-1]) \in t_{ik}$ erzeugt und gemeinsam mit TID_i zu $\overline{C}_k$ hinzugefügt. Weiterhin wird L_k erzeugt und so weiter, bis dieses der leeren Menge entspricht und der Algorithmus terminiert.
Im Gegensatz zu *Apriori* muss die Transaktionsdatenbank D nur ein einziges mal durchlaufen werden, in den nachfolgenden Durchgängen wird immer $\overline{C}_k$ zur Generierung der Kandidaten genutzt. Mit wachsendem k schrumpft offensichtlich die Größe von $\overline{C}_k$. Dennoch ist diese Menge gerade für kleine k mitunter sehr groß, für $k = 1$ wird die gesamte Transaktionsdatenbank in $\overline{C}_1$ gehalten. Deshalb empfehlen Agrawal et al. für kleine k die Nutzung von *Apriori*. Eine Heuristik schätzt ab, ob eine zu generierende Menge $\overline{C}_k$ in den Speicher passen würde. Ist dies der Fall und die Anzahl der Kandidaten für häufige Itemsets ist rückläufig, findet eine Wechsel zu *AprioriTID* statt. Dieses Verfahren wird als *AprioriHybrid* bezeichnet.

5.1.2 Frequent-Pattern-Trees

Die Generierung von häufigen Itemsets mit den im vorangegangenen Abschnitt beschriebenen Verfahren birgt einen großen Nachteil: Erzeugung und Speicherung der Kandidatenmenge erfordert einen hohen Speicheraufwand. Daher ist es erstrebenswert, Verfahren zu entwickeln, die auf diesen Schritt verzichten können.
Han et al. [HPY00] entwickelten hierzu die Frequent-Pattern-Trees als besondere Datenstruktur, welche eine kompakte Darstellung einer Transaktionsdatenbank ermöglicht. Diese Baumstruktur besteht aus einer Wurzel, dessen Kindknoten als Präfix-Teilbäume (item prefix subtree) bezeichnet werden. Jeder Knoten im Präfix-Teilbaum enthält eine Referenz zu einem Item, eine Zählvariable, welche die Anzahl der zu repräsentierenden Transaktionen darstellt und einem Verweis (node-link) zum nächsten Knoten, welcher dasselbe Item enthält. Außerdem sind Verweise zu Vorgänger- und Kindknoten notwendig. In einer Tabelle (frequent-item header table) werden Verweise von jedem häufigen Item auf den zuerst im Frequent-Pattern-Tree eingefügten Knoten mit entsprechendem Item gespeichert. Bei der Erzeugung eines Frequent-Pattern-Trees wird die Transaktionsdatenbank genau zweimal durchlaufen. Im ersten Durchlauf werden alle Items gezählt. Erfüllt ein Item nicht den minimalen Support, so wird es in sämtlichen nachfolgenden Schritten verworfen. Auf den übrigen Items wird eine Ordnung entsprechend ihrer Häufigkeit definiert. Im zweiten Durchlauf werden nacheinander die einzelnen Transaktionen zur Datenstruktur hinzugefügt. Ausgehend von der Wurzel wird überprüft, ob für das häufigste Item bereits

Transaktionsnummer	Enthaltene Items
1	f,a,c,d,g,i,m,p
2	a,b,c,f,l,m,o
3	b,f,h,j,o
4	b,c,k,s,p
5	a,f,c,e,l,p,m,n

(a) Transaktionsdatenbank



(b) Frequent-Pattern-Tree

Abbildung 5.1: Beispiel eines Frequent-Pattern-Tree mit dazugehöriger Transaktionsdatenbank

ein direkter Kindknoten existiert. Ist dies der Fall, wird die Zählvariable des entsprechenden Knotens inkrementiert und von diesem Knoten aus mit dem nächst-häufigeren Item gleich verfahren. Existiert noch kein solcher Knoten, wird für das Item ein neuer Knoten mit Startwert 1 erzeugt und eingefügt. In der *frequent-item header table* wird nach einem Eintrag für das aktuelle Item gesucht. Existiert dieser, wird die Kette von Knoten, die beim Verfolgen der *node-links* entsteht um den neu erzeugen Knoten am Ende ergänzt. Ansonsten wird der aktuelle Knoten direkt in die Tabelle eingetragen. Abbildung 5.1a zeigt ein Beispiel für eine Transaktionsdatenbank mit entsprechendem Frequent-Pattern-Tree in Abbildung 5.1b. *Node-links* werden hier durch gestrichelte Linien dargestellt. Die *freqent-item header table* ist in der Abbildung nicht vorhanden.

```
Procedure FP–growth(Tree, α)
  IF Tree contains a single path link P
  THEN FORALL combinations (denoted as β) of the nodes in
      the Path P DO
      generate pattern β ∪ α with support = minimum support
          of nodes in β
  ELSE FORALL α_i in the header of Tree DO
      generate pattern β = α_i ∪ α with support = a_i.support
      construct β's conditional pattern base and then β's
          conditional FP–tree Tree_β
      IF Tree_β ≠ ∅
      THEN call FP–growth(Tree_β, β)
```

Listing 5.5: FP-growth nach [HPY00]

Die häufigen Itemsets werden ermittelt, indem für den vollständig konstruierten Frequent-Pattern-Tree die Funkion FP-growth (Listing 5.5) aufgerufen wird, wobei $\alpha = \emptyset$. Diese prüft, ob es sich bei der Eingabe um einen Baum mit genau einem Pfad P handelt. Ist dies der Fall, wird jede Kombination der Elemente des Pfades vereinigt mit der Eingabe α in die Menge der häufigen Itemsets aufgenommen. Andernfalls

wird für jedes Element a_i in der *frequent-item header table* mit der Eingabe α vereinigt. Die daraus resultierende Menge, genannt β wird der Menge der häufigen Itemsets hinzugefügt. Weiterhin werden die konditionale Basis (conditional pattern base) und darauf aufbauend der konditionale Frequent-Pattern-Tree $Tree_\beta$ konstruiert. Handelt es sich bei letzterem um einen nicht-leeren Baum, wird die Funktion *FP-growth*($Tree_\beta, \beta$) rekursiv aufgerufen.
Die konditionale Basis für ein Item α_i enthält alle Pfade von Knoten, die α_i enthalten, bis zur Wurzel. Dabei wird dem jeweiligen Pfad der Support von α_i im Ausgangsknoten zugewiesen. Die konditionale Basis für das Item p in Abbildung 5.1b ist {[f,c]:2,[c]:1}, der entsprechende Frequent-Pattern-Tree lässt sich auf Basis der Transaktionen [{f,c},{f,c},{c}] konstruieren.

5.2 Gewinnen von Assoziationsregeln aus häufigen Itemsets

Sind die häufigen Itemsets bekannt, können die Assoziationsregeln aus ihnen gewonnen werden. Die Verfahren dazu sind allgemein nicht davon abhängig, mit welcher Methode die Itemsets zuvor gesucht wurden, solange für jedes häufige Itemset auch der Support bekannt ist. Generell können Assoziationsregeln gefunden werden, indem jedes häufige k-Itemset l mit $k > 1$ und alle Teilmengen $a \subset l$ Assoziationsregeln der Form $a \rightarrow (l - a)$ erzeugt und deren Konfidenz ermittelt wird. Diese soll dabei mindestens einer geforderten Minimalkonfidenz entsprechen, um in die Menge der gefundenen Regeln aufgenommen zu werden. Weil die Teilmengen von l immer auch häufige Itemsets sind und deren Support bereits bekannt ist, lässt sich die Konfidenz wie folgt einfach ermitteln:

$$conf(a \rightarrow (l - a)) = \frac{support(l)}{support((l - a))}) \tag{5.1}$$

Dabei müssen nicht zwangsläufig alle möglichen Regeln überprüft werden. Offensichtlich ist der Support für jede Menge $\tilde{a} \subset a$ mindestens so groß, wie der von a. Demzufolge kann die Konfidenz einer Regel $\tilde{a} \rightarrow (l - \tilde{a}))$ nicht größer sein, als die der Regel $a \rightarrow (l - a)$, wodurch entsprechende Regeln nicht überprüft werden müssen. Agrawal et al. schlagen daher den einfachen Algorithmus 5.6 vor, der sich dieser Eigenschaften bedient.

```
forall large itemsets l_k, k ≥ 2 do
    call genrules(l_k, l_k);

procedure genrules(l_k: large k–itemset, a_m: large m–itemset)
    A = {(m − 1) − itemsets a_{m−1} | a_{m−1} ⊂ a_m};
    forall a_{m−1} ∈ A do begin
        conf = support(l_k)/support(a_{m−1});
        if(conf \geq minconf) then begin
            output the rule a_{m−1} → (l_k − a_{m−1}) with confidence = conf
                and support = support(l_k);
            if(m − 1 > 1) then
                call genrules(l_k, a_{m−1})
    end
end
```

Listing 5.6: Einfacher Algorithmus zum finden von Assoziationsregeln in häufigen Itemsets nach [AS+94]

5.3 Implementierung eines Verfahrens zur Assoziationsanalyse

Im Rahmen dieser Arbeit wurde eine prototypische Java-Implementierung vorgenommen. Dabei wurde auf Anraten der Gutachter das Verfahren Frequent-Pattern-Trees umgesetzt. Die aktuelle Implementie-

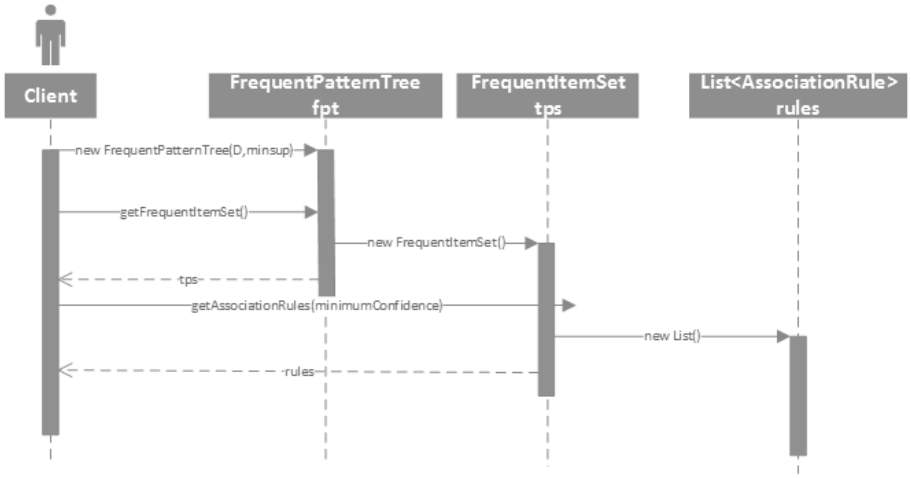


Abbildung 5.2: Sequenzdiagramm zur Veranschaulichung der vorläufigen Implementierung

rung besteht im Wesentlichen aus den generischen Klassen `FrequentPatternTree`, `FrequentItemSet` und `AssociationRule`.
Beim Erzeugen eines Objekts der Klasse `FrequentPatternTree` wird ein Frequent-Pattern-Tree konstruiert, dabei benötigt der Konstruktor zwei Argumente. Beim ersten Parameter vom Typ `List<Set<T>>` handelt es sich um die Transaktionsdatenbank. Der zweite Parameter stellt den minimalen Support dar. Dieser kann entweder absolut als `int` oder relativ zur Gesamtzahl der Transaktionen in der Datenbank als `double` angegeben werden. Dabei muss der relative Support im Intervall [0.0,1.0] liegen. Um die Einhaltung der Reihenfolge zu garantieren wurde für die *frequent-item header table* die Datenstruktur `LinkedHashMap` gewählt. Knoten werden durch die innere Klasse `FrequentPatternNode` dargestellt und entsprechen dabei den Knoten des Präfix-Teilbaumes (Abschnitt 5.1.2). Durch den Aufruf von `getFrequentItemSet()` wird der FP-growth-Algorithmus (Listing 5.5) durchgeführt und die Menge der häufigen Itemsets zurückgegeben
Die Klasse `FrequentItemSet` speichert die häufigen Itemsets zusammen mit dem jeweiligen Support als `Map<Set<T>,Integer>`. Gleichzeitig bildet eine Variable vom Typ `Map<Integer,Set<Set<T>>>` von k auf die k-Itemsets ab.
Eine Liste der Assoziationsregeln mit minimaler Konfidenz c gibt die Methode `getAssociationRules(c)` zurück. Objekte vom Typ `AssociationRule` speichern eine Menge von Objekten als Antezedens[2], eine weitere als Konsequenz. Weiterhin werden die Konfidenz und der Support der Regel gespeichert.
Die Implementierung verwendet einige Fremdbibliotheken. Dazu gehören etwa die *Commons Lang 3.3.2* aufgrund der `Pair`-Klasse. Die Speicherung des Frequent-Pattern-Tree als Bilddatei wird mithilfe der Bibliothek *Wintersleep Graphviz 0.1* realisiert. Schlusendlich ermöglichen die *Guava Google Core Libraries 18.0* die Nutzung von MultiSets und eine Funktion zur Generierung von Potenzmengen.

5.3.1 Testdurchlauf der Implementierung

Zur Überprüfung der Lauffähigkeit wurden aufbereitete Logdateien der Website `www.microsoft.com` aus dem Jahre 1998 verwendet, bereitgestellt durch das Center for Machine Learning and Intelligent Systems der University of California, Irvine[3]. Diese repräsentieren das anonymisierte Besucherverhalten innerhalb einer Woche. Der Umfang des Datensatzes beträgt circa 32000 Einträge, jeder davon enthält die Inhalte, die der Benutzer in diesem Zeitfenster abgerufen hat. Die Anzahl der verschiedenen Items beläuft sich

[2]Unter dem Antezedens ist der linke Teil der Assoziationsregel oder einer Implikation im Allgemeinen zu verstehen.
[3]Die Daten sind unter `http://archive.ics.uci.edu/ml/datasets/Anonymous+Microsoft+Web+Data` abrufbar.

Support	Häufige Items	Häufige Itemsets	Knoten im Baum	Zeit zur Baum-konstruktion	Zeit zum Finden häufiger Itemsets
50	135	2611	21726	165ms	167ms
328 (0.01)	47	197	12183	115ms	19ms
1636 (0.05)	11	19	895	18ms	4ms
3272 (0.1)	7	8	127	15ms	3ms

Tabelle 5.1: Gemessene Werte bei der Erzeugung von Frequent-Pattern-Trees und häufiger Itemsets.

Support	Konfidenz	Zeit	Gefundene Regeln	Maximale Abweichung (absolut)	Durchschnittliche Abweichung (absolut)
50	0.5	474ms	2927	0.4356	0.0846
	0.75	249ms	692	0.4356	0.0672
	0.9	227ms	256	0.2675	0.053
328 (0.01)	0.5	24ms	91	0.1276	0.0331
	0.75	39ms	23	0.0759	0.0295
	0.9	18ms	6	0.0633	0.0286
982 (0.03)	0.5	19ms	14	0.0658	0.021
	0.75	17ms	2	0.0158	0.0146
	0.9	12ms	1	0.0158	0.0158
1636 (0.05)	0.5	4ms	3	0.0221	0.0163
	0.75	4ms	0	-	-
	0.9	3ms	0	-	-
3272 (0.1)	0.5	4ms	1	0.0168	0.0168
	0.75	2ms	0	-	-
	0.9	1ms	0	-	

Tabelle 5.2: Gemessene Werte beim Finden von Assoziationsregeln.

auf 294. Weitere 5000 Transaktionen stehen als Testdaten zur Verfügung. Sie liegen im ascii-basierten DST-Format vor und wurden mithilfe eines selbstimplementierten, rudimentären Parsers eingelesen
Für verschiedene Support- und Konfidenzwerte wurden die Assoziationsregeln bestimmt. Anschließend wurde jede Transaktion der Testdaten auf passende Regeln überprüft. Dazu wurde für jede Regel ermittelt, ob ihr Antezedens Teilmenge der Transaktion ist. War dies der Fall, wurde auch die Konsequenz überprüft. Nach Durchlauf aller Transaktionen wurden die Konfidenzen der Regeln im Testdatensatz mit den zuvor ermittelten verglichen. Alle Zeitwerte ergeben sich dabei aus genau einem Durchlauf und können selbstverständlich auf unterschiedlichen Systemen oder in weiteren Durchläufen abweichen. Bei dem Testsystem handelt es sich um einen handelsüblichen Personal Computer mit einem Intel Core i5-2500k (3.3GHz), 4.00 GB DDR3-Arbeitsspeicher und einem 64 Bit-Windows 7 Betriebssystem. Die verwendete Java-Version ist 1.8. Tabelle 5.1 zeigt die Auswirkung des Supports und damit der zu verarbeitenden häufigen Itemsets auf die Größe des Baumes und die Laufzeit der Konstruktion und der Ermittlung der häufigen Itemsets. Die Ergebnisse sind wenig überraschend. Ein hoher Support bewirkt eine geringe Anzahl von häufigen Items und Itemsets. Ebenso sinken die Anforderungen an Speicher und Zeit.
Interessanter sind die Ergebnisse in Tabelle 5.2. Sie zeigt die Auswirkungen von Support und Konfidenz auf die Anzahl der Ergebnisse und deren Güte. Wie zu erwarten verringert sich die Anzahl der ermittelten Regeln mit größeren Werten für Support und Konfidenz. Gleichzeitig steigt die Güte, was sich durch geringere Abweichungen bemerkbar macht. Daher sollte ein brauchbarer Kompromiss zwischen Regelgüte und -anzahl für den späteren Anwendungsfall ermittelt werden.

5.3.2 Nachträgliche Veränderungen der Grundimplementierung

Im Verlauf der Arbeit unterlag die Implementierung des *Frequent-Pattern-Trees* ständigen Überarbeitungen. Diese sollen in diesem Abschnitt ergänzt werden. In allen anderen Abschnitten, in der von der Implementierung gesprochen wird, soll immer die ursprüngliche Version angenommen werden, solange nicht explizit ein anderer Stand erwähnt wird.
Im Kontext dieser Arbeit sollen Regeln gesucht werden, die es ermöglichen, von den Attributen in unvollständigen Tabellen auf die unbekannten Eigenschaften weiterer, in derselben Tabelle vorkommenden Attribute zu schließen. Offenbar sind nur solche Regeln von Interesse, die in ihrem Antezedens Items enthalten, die potentiell in den unvollständigen Tabellen vorkommen. Eine Regel der Form $\{A\} \rightarrow \{B, C\}$ ist demnach nur in dem Fall interessant, wenn es mindestens eine unvollständige Tabelle gibt, aus der sich das Item A herleiten lässt. Aus diesem Grund wurden neue Konstruktoren[4] eingeführt, die es erlauben, ein Prädikat `Predicate<T>` zu übergeben. Items, die diesem nicht entsprechen, werden aus der Frequent-Item-Header-Table entfernt, was direkten Einfluss auf die erzeugten häufigen Itemsets und damit auch auf die resultierenden Regeln hat. Im Besonderen wurde eine Prädikatklasse `ContainsPolicyItemForAttribute` für `PolicyItems` implementiert. Objekte dieser Klasse verwalten eine Menge von Attributnamen, die über die Methoden `add` und `addAll`, angelehnt an gleichnamige Funktionen für Mengen in Java, hinzugefügt werden können. Ein `PolicyItem` entspricht nur genau dann diesem Prädikat, wenn seine Attributbezeichung in dieser Menge vorkommt. So ist es möglich, schon vor der Regelgenerierung alle Items auszuschließen, die nur in vollständigen Tabellen vorkommen. Beispielhaft kann ein Fall genannt werden, in dem die Regelmenge einer bestimmten, kleinen Policy von insgesamt 1876 auf 186 potentiell interessante Regeln verringert werden konnte. Bei größeren Policies wurde eine Regelextraktion nur so überhaupt möglich, da die Implementierung mit Speicherproblemen zu kämpfen hat. Weitere Informationen zu dieser Problematik folgen im anschließenden Abschnitt.

5.3.3 Weitere, mögliche Verbesserungen der Implementierung

Wird die aktuelle Implementierung und deren Entstehungsprozess reflektiert, sind einige Verbesserungen denkbar. Diese werden eventuell umgesetzt, wenn es sich im Zeitrahmen dieser Arbeit unterbringen lässt[5].

- Um Austauschbarkeit der Algorithmen zu gewährleisten, ist es denkbar, die Schritte der Assoziationsanalyse einzeln zu betrachten und entsprechende Schnittstellen zu definieren. So kann etwa die Suche nach häufigen Itemsets durch eine Schnittstelle `FrequentItemSetGenerator` geschehen, während sich die Extraktion der Regeln hinter einer Schnittstelle `RuleMiner` verbirgt. Weiterhin kann nach dem Vorbild des Strukturmusters *Fassade* [GHJV] der gesamte Prozess nach außen über eine einzige Schnittstelle verfügbar gemacht werden.

- Die Transaktionsdatenbank wird als Struktur `List<Set<T>>` während der gesamten Laufzeit vollständig im Speicher gehalten. Für große Datenbanken und Systeme mit geringem Arbeitsspeicher ist es daher erstrebenswert, eine andere Darstellungsart zu wählen. Weiterhin können die Daten aus verschiedensten Quellen stammen, etwa aus Datenbanksystemen, die via JDBC ausgelesen werden oder aus Dateien im XML- oder CSV-Format. Ein Vorschlag, der beide Probleme behandelt wäre hier die Einführung einer Schnittstelle `TransactionDatabase`. Diese ermöglicht, ähnlich einem `Iterator` [GHJV], das sequentielle Abfragen der Transaktionen. Je nach Quelle können Implementierungen vorgenommen und untereinander ausgetauscht werden.

- Um die Fertigstellung des Prototypen zu beschleunigen, wurde auf Fremdbibliotheken zurückgegriffen. Diese wurden im Abschnitt 5.3 bereits genannt. Keine der genutzten Funktio-

[4] Denkbar wäre auch die Implementierung einer Methode `filterItems(Predicate<T> p)`. Allerdings wird bereits beider Erzeugung der Frequent-Pattern-Tree aufgebaut, was zur Folge hätte, dass dies zum Filtern ein weiteres mal geschehen müsste. Dies zu umgehen, hätte weitere Änderungen der Spezifikation zur Folge, was im Zeitrahmen dieser Arbeit allerdings nur schwer möglich gewesen wäre.

[5] Eine Neuimplementation wird in Kapitel 7 vorgeschlagen

nen ist so umfangreich, dass sie nicht mit annehmbarem Aufwand selbst implementiert oder durch andere Verfahren ersetzt werden könnte. Insbesondere ist hier die Nutzung der Funktion *com.google.common.collect.Sets.powerSet* problematisch, da Sie nur eine Eingabe von Mengen mit maximal 30 Elementen erlaubt[6]. Dieser Umstand ist bei Transaktionsdatenbanken mit vielen Items nicht tragbar. Weiterhin kann die Nutzung von Fremdbibliotheken in größeren Softwareprodukten zu schwerwiegenden Versionskonflikten führen, was zu vermeiden erstrebenswert ist.

- Einige verwendete Datenstrukturen sind nicht optimal in Bezug auf Speicherbedarf und Laufzeit. Besonders bei geringen Werten für den Minimalsupport besteht das Problem, dass das Programm mit einer Exception *java.lang.OutOfMemoryError: GC overhead limit exceeded* endet. Diese tritt auf, wenn der Garbage Collector der Java Virtual Machine viel Zeit in Anspruch nimmt und gleichzeitig nur geringe Speichermengen freigibt. Vermutlich wird dies durch die häufigen rekursiven Aufruf der Funktion `fpGrowth()` verursacht, welche viele weitere, kurzlebige Frequent-Pattern-Trees mitsamt Hilfsstrukturen konstruiert. Eine Deaktivierung des Limits schafft nur bedingt Abhilfe, zum Teil reicht hierbei der Heap nicht aus und das Programm terminiert ebenfalls. Der Versuch, den Speicherbedarf mithilfe des Design Pattern *Flyweight* [GHJV] für Items zu verringern, brachte in dieser Sache noch keinen Durchbruch. Ein alternativer Ansatz wäre es etwa, statt der Items lediglich Referenzen oder weniger speicherintensive Darstellungen in Transaktionen zu speichern. In der Implementierung von Eibe Frank und Stefan Mutter [WF00, WFM05] wird die Transaktionsdatenbank als Array von Integern (Bitmap) codiert. Weiterhin kann die Speicherung von häufigen Itemsets und Assoziationsregeln insofern verbessert werden, dass darin vorkommende Teilmengen effizienter gefunden werden können. Dazu bieten sich Baumstrukturen an, ähnlich dem in [AIS93] vorgeschlagenen Hash-Baum.

[6] http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Sets.html

Kapitel 6

Anwendung der Assoziationsanalyse

6.1 Datenquellen und -aufbereitung

Die Datenschutzeinstellungen für das PArADISE-Framework werden von Nutzern individuell konfiguriert und in Form von XML-Dateien gespeichert. Im Groben definiert eine solche Policy eine oder mehrere Anwendungen, die aus Modulen bestehen. Für jedes Modul können Einstellungen für mehrere Attribute definiert werden. Neben dem Namen gehören hierzu allgemeine Freigaberechte (allow) und Weitergabe von Informationen an Dritt-Anwendungen (third_party_access). Weiterhin können minimale Anfrageintervalle (interval), allgemeine Bedingungen (condition), sowie erlaubte Aggregationsstufen (aggregation) definiert werden. Alternativ ist es auch möglich, eine Datenschutzstufe (privacyLevel) anzugeben. Hierbei sind die Stufen *public*, *confidential*, *secret* und *top-secret* möglich. Weitere Informationen finden sich in der zugehörigen Dokumentation [Gru13]. Im Framework enthalten ist ein Parser um besagte Dateien lesen zu können. Dieser gibt ein Objekt der Klasse `Policy` zurück, welches Anwendungen, Module und Attribute intuitiv als Objekte zur Verfügung stellt. Die Funktionalität zum Schreiben eines Policy-Objekts als XML-Datei wurde im Verlauf der Arbeit durch den Betreuer nachgereicht.
Im Erstellungszeitraum dieser Arbeit existierte noch keine größere Sammlung von Policies, welche als Grundlage für Versuche dienen konnte. Daher war es notwendig, auf manuellem oder automatisiertem Wege Datenschutzeinstellungen zu erzeugen.

6.1.1 Automatisierte Generierung von Datenschutzeinstellungen

Um große Datenmengen zu erstellen, fiel die Wahl auf die letztere Option. Dazu wurde die Klasse `PolicyGenerator` implementiert, welche entsprechende `Policy`-Objekte erzeugt. Diese verbindet sich via JDBC mit einer zuvor definierten Datenbank und liest für ein spezifiziertes Schema die Tabellen- und Attributnamen aus. Anhand dieser Daten wird eine Datenschutzeinstellung generiert.
Dabei muss nicht für das vollständige Schema eine Datenschutzeinstellung erzeugt werden, zum einen kann der Anteil an vollständig zu betrachtenden Tabellen, zum anderen die Quote an Attributen in unvollständigen Tabellen angegeben werden. So kann beispielsweise gefordert werden, dass in neun von zehn Tabellen Datenschutzeinstellungen für alle enthaltenen Attribute erzeugt und in den verbleibenden drei von vier Attributen beachtet werden. Zur Auswahl der Attribute und bei der nachfolgenden Vergabe der Werte wird die Klasse `java.util.Random` genutzt, der dazugehörige Anfangswert kann vom Nutzer eingegeben werden, wodurch reproduzierbare Policies möglich werden.
Da weder das XML-Schema, noch die Ausgabe des Parsers ein Tabellenkonzept kennen, die Zusammengehörigkeit von Attributen berücksichtigen, nutzt der *PolicyGenerator* die Namenskonvention *tabellenname.attributname*, um diese Eigenschaft zu erhalten. Weiterhin ist es nicht ohne Weiteres möglich, ein nicht definiertes Attribut darzustellen. Dies ist der Fall da ein Großteil der Eigenschaften als primitive Datentypen deklariert ist. So ist für den Typen `boolean`, welcher etwa für `allow` genutzt wird nur der

privacyLevel	allow	third_party_access
public	0.75	0.75
confidential	0.75	0.25
secret	0.25	0.25
top-secret	0.1	0.1

Tabelle 6.1: Wahrscheinlichkeiten, dass die angegebenen Eigenschaften den Wert *true* zugewiesen werden, in Abhängigkeit vom zuvor gewählten privacyLevel

Wert *true* oder *false* möglich, die Angabe null ist hier nicht zulässig. Aus diesem Grund wird vereinbart, dass sämtliche Datenschutzeinstellungen für ein Objekt nicht definiert sind, genau dann, wenn die Eigenschaft *privacyLevel* gleich null ist.
Bei der Erstellung der Datenschutzeinstellungen wird sich vorerst auf die Eigenschaften *allow*, *third_party_access* und *privacyLevel* beschränkt. Statt einer Gleichverteilung der Werte wurde eine Abhängigkeit *allow* und *third_party_access* von *privacyLevel* entsprechend Tabelle 6.1 angenommen.
In den Versuchen, welche in Abschnitt 6.2 beschrieben werden, zeigte sich allerdings schnell, dass sich aus auf diese Weise erzeugte Daten nur sehr schlecht Regeln ableiten lassen. Dies erscheint nur logisch, da zwar Abhängigkeiten zwischen den Eigenschaften eines einzelnen Attributes abgebildet werden, die Attribute untereinander allerdings vollkommen unabhängig behandelt werden. Weil jedoch für gleiche Attribute mit hoher Wahrscheinlichkeit die selben Datenschutzeinstellungen gewählt werden, wurde die Generierung der Policies entsprechend angepasst.
Im ersten Schritt wird dem hypothetischen Ersteller der Policy eine bevorzugte Datenschutzstufe in Form des *PrivacyLevels* zufällig zugewiesen. Auf diese Weise sollen etwa Nutzer dargestellt werden, die entweder besonders fahrlässig oder umsichtig mit ihren Daten umgehen. Für jedes Attribut wird daraufhin eine bevorzugte Datenschutzstufe gewählt. So werden etwa besonders schützenswerte Attribute, beispielsweise ein Name, gekennzeichnet. Dabei wird mit 25-prozentiger Wahrscheinlichkeit die bevorzugte Stufe des Autors gewählt, ansonsten wird zufällig eine Stufe festgelegt. Bei der Vergabe der tatsächlichen Datenschutzrichtlinie für ein konkretes Attribut wird mit einer 50-prozentigen Wahrscheinlichkeit die bevorzugte Stufe gewählt, ansonsten wird wieder aus allen möglichen Stufen gezogen. Daraus ergibt sich, dass die bevorzugte Sicherheitsstufe für ein Attribut mit einer Wahrscheinlichkeit von $0,25 + 0,75 \cdot 0,25 = 0,4375$ mit der Stufe des Nutzers übereinstimmt. Weiterhin stimmt die Stufe eines konkreten Attributs mit einer Wahrscheinlichkeit von $0,4375 + 0,5625 \cdot 0,5 \approx 0,72$ der Nutzerstufe entspricht. Die Vergabe der restlichen Eigenschaften geschieht danach wie in 6.1 angegeben.
Im Gegensatz zum ersten Ansatz wird nicht nur ein einzelnes Modul mit allen im Schema vorkommenden Tabellen (von denen einige unvollständig sind) erzeugt. Stattdessen werden 1 - 15 kleinere Module erzeugt, die nicht alle Tabellen enthalten. Außerdem werden nicht alle Attribute einer Tabelle angegeben, dafür werden allerdings Einstellungen für alle angegebenen Attribute erzeugt. Der Anwender erhält so durch den Aufruf der Methode `getPolicy()` eine Policy ohne unbekannte Einstellungen. Dabei besteht absichtlich die Möglichkeit, dass gleiche Tabellen und Attribute in unterschiedlichen Modulen mehrfach beschrieben werden. Eine unvollständige Datenschutzeinstellung kann mittels der Methode `getIncomplete()` angefordert werden. Mit einer Anzahl von einer bis drei Tabellen in genau einem Modul sind diese vergleichsweise klein, die Menge der unbekannten Attribute in den Tabellen ist ebenfalls gering. Sie sollen die Tabellen darstellen, die in eine vollständige Policy eingefügt werden soll und deren Großteil der Attribute bereits im vorhergehenden Schema-Mapping-Schritt bestimmt werden konnte.

6.1.2 Überführung der Policy in eine Transaktionsdatenbank

Die selbstgewählte Eingabe für die Implementierung der Assoziationsanalyse ist eine Transaktionsdatenbank der Form `List<Set<T>>`, in welche die Policy in geeigneter Weise zu überführen ist. Intuitiv entsprechen hierbei die Tabellen des Schemas den Transaktionen, die Attribute, in geeigneter Form mit

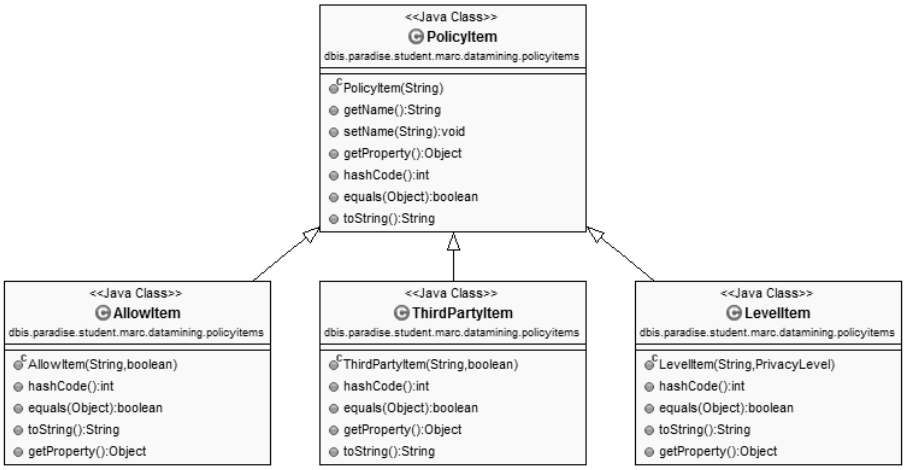


Abbildung 6.1: Vorschlag für eine geeignete Datenstruktur zum Speichern von Attributeigenschaften in Transaktionen

ihren Eigenschaften kombiniert, stellen die Items dar. Es wird davon ausgegangen, dass gleichnamige[1] Attribute als gleich, beziehungsweise ähnlich genug angesehen werden können. Aus diesen lassen sich dieselben Items ableiten.
Dass dieser Ansatz nicht optimal ist, ist dabei ganz offensichtlich. So kann beispielsweise dasselbe Attribut in einer Tabelle die Bezeichnung *Familienname*, an anderer Stelle allerdings *Nachname* heißen. Denkbar wäre es, dass das Similarity Flooding Verfahren (Kapitel 4) mit dem ursprünglichen und dem um neue Tabellen und Attribute ergänzten Schema durchgeführt wird. Liegt das σ-Mapping für zwei Attribute über einem zuvor gewählten Schwellwert, könnten diese als ausreichend ähnlich betrachtet werden und so zu gleichen Items führen. Aufgrund der zuvor erwähnten Probleme mit der Implementierung des Verfahrens konnten dazu allerdings keine Versuche durchgeführt werden.
Alternativ ist die manuelle Definition einer *ist-Ähnlich-zu*-Relation denkbar. Für kleinere Schemata sollte der Aufwand vertretbar sein und bei großen Nutzerzahlen einen Mehrwert bieten. Bei neuen Tabellen und Attributen, beziehungsweise Sensoren im Kontext von intelligenten Umgebungen, muss diese Relation stets mitgepflegt werden. Offensichtlich eignet sich dieser Ansatz nicht für große oder schnell wachsende Umgebungen, weshalb langfristig Versuche mit Schema Mapping Verfahren in Erwägung gezogen werden sollten.
Ist die Problematik der ähnlichen Attribute geklärt, sollte sich der Darstellung der Items gewidmet werden. Die Items entsprechen dabei einem Tupel aus Attributidentifikator, im einfachsten Falle ist dies die Bezeichnung, und dem Wert der Eigenschaft. In den ersten Versuchen wurde hierfür die Klasse `Pair<L,R>`[2] genutzt. Weil diese allerdings tatsächlich nur den Attributidentifikator und einen Wert speichern konnte, nicht aber, um welche Eigenschaft es sich konkret handelte, wurde für jede der Eigenschaften eine eigene Transaktionsdatenbank erstellt und damit auch eine separate Assoziationsanalyse durchgeführt.
Da es jedoch wünschenswert ist, bei Möglichkeit nur eine einzige Analyse durchführen zu müssen, gingen die Bestrebungen dahin, alle Eigenschaften in einer einzigen Transaktionsdatenbank unterzubringen. Dazu wurden die Tupel als `PolicyItem` entsprechend Abblidung 6.1 implementiert. Wird die Transaktionsdatenbank als `List<Set<PolicyItem>>` definiert, können die Items als Objekte der Kindklassen von `PolicyItem` eingefügt werden. So kann auch die Art des Tupels unterschieden werden und die mehrfache Existenz desselben Wertes für unterschiedliche Eigenschaften eines gleichen Attributs einer Transaktion stellt kein Problem mehr dar. Zusätzlich kann pro Attribut ein `PolicyItem` (keine Kindklasse) hinzuge-

[1] Gemeint ist hierbei der tatsächliche Attributname, unabhängig vom in der Policy zusätzlich angegebenen Tabellennamen. Weiterhin wird von Gleichheit im Sinne von String.equals() gesprochen.
[2] Enthalten in der Bibliothek *Apache Commons Lang* ab Version 3.0

fügt werden. Dieses enthält ausschließlich den Attributidentifikator. Auf diese Weise könnten Schlüsse aus dem reinen Vorhandensein eines Attributs gezogen werden, unabhängig von den Eigenschaften. Die einzigen Schnittstellen nach außen sind neben Gettern und Settern für die Attribute und dem Konstruktor lediglich die von `java.lang.Object` geerbten Methoden. Ebenfalls wurden die Methoden `hashCode()` und `equals()` überschrieben[3]. So soll sichergestellt werden, dass Items genau dann als gleich erkannt werden, wenn sie sowohl derselben Klasse angehören, als auch in allen Attributen übereinstimmen. Die automatisierte Implementierung der Methoden durch die Eclipse IDE erwies sich hierbei als völlig ausreichend. Diese Darstellungsform der Items funktioniert auch problemlos bei der Durchführung einer Assoziationsanalyse pro Eigenschaft, weshalb dabei die Klasse `Pair` durch `PolicyItem`, beziehungsweise der entsprechenden Kindklasse ersetzt wurde. Auch hier ist das Hinzufügen des Attributidentifikators als eigenständiges Item denkbar.

6.2 Assoziationsanalyse und Regelanwendung

Um die Anwendbarkeit von Assoziationsregeln zu testen, wurden Analysen für eine Reihe zufällig generierter Policies durchgeführt und die Anzahl der herleitbaren Eigenschaften überprüft. Dabei wurde vorerst stets eine minimale Konfidenz von 0.75 gewählt. Dieser Wert stellt nach Ansicht des Autors einen guten Anfangswert dar, welcher eventuell nach oben hin angepasst werden kann. Aufgrund der verschiedenen Versionen des `PolicyGenerator`, wurden diese Versuche in zwei Anläufen durchgeführt, welche im Folgenden getrennt voneinander besprochen werden sollen.

6.2.1 Versuche mit attributunabhängiger Policy-Generierung

Im ersten Anlauf wurde mit Polcies gearbeitet, in denen die Ähnlichkeit der Attribute untereinander bei der Generierung nicht beachtet wurde (Abschnitt 6.1.1). Da in dieser ersten Version des `PolicyGenerators` vollständige und unvollständige Tabellen in einer einzigen `Policy` gemischt auftraten, musste zuerst eine entsprechende Trennung durchgeführt werden. Aus den vollständigen Tabellen wurde, wie zuvor beschrieben, die Transaktionsdatenbank, beziehungsweise je nach Methode die Transaktionsdatenbanken erzeugt und entsprechende Regelextraktion mittels *FPGrowth* durchgeführt. Nach dem Prinzip von Versuch und Irrtum wurde dabei mit einem Minimalsupport von 1 begonnen. Da dies, je nach Größe der `Policy`, zu Problemen mit Garbage Collector und Heap Size führte, wurde dieser Wert manuell Schritt für Schritt nach oben korrigiert, bis ein Wert gefunden war, der in annehmbarer Zeit zur Terminierung des Algorithmus führte.
Es wurde schnell ersichtlich, dass die Variante, in der alle Eigenschaften der Attribute in einer einzigen Transaktionsdatenbank gespeichert wurden, aufgrund der deutlich höheren Anzahl verschiedener Items schnell zu Problemen führte. Supportbedingungen, die eine Durchführung der Assoziationsanalyse zuließen waren oft so hoch, dass bereits die Menge der häufgen Items leer war, wodurch auch keine Regeln gefunden werden konnten. Auch die Reduktion der häufigen Itemsets durch Filterung (Abschnitt 5.3.2) konnte hier keine Abhilfe schaffen, weshalb ab diesem Zeitpunkt nur noch die Methode mit verschiedenen Transaktionsdatenbanken pro Eignschaft Anwendung fand.
Waren die Regeln extrahiert, mussten sie noch angewendet werden. Der Einfachheit halber wurden die einzelnen Regelmengen für die Eigenschaften *privacyLevel*, *allow* und *thirdPartyAccess* in einer Menge vereint. Für jede Regel und jede unvollständige Tabelle wurde nun überprüft, ob das Antezendens eine Teilmenge der Transaktionsdarstellung der Tabelle war. War dies der Fall, galt auch die Konsequenz der Regel, dementsprechend wurde überprüft, ob deren Schnittmenge mit den unbestimmten Attributen der Tabelle Elemente enthielt. Dies würde bedeuten, dass zumindest einige Eigenschaften bestimmt werden könnten.

[3]Der Vollständigkeit halber soll zusätzlich noch erwähnt sein, dass toString() ebenfalls verändert wurde, dies trägt allerdings weniger zur Funktionalität bei als die zuvor genannten Methoden.

In den Versuchen zeigte sich jedoch, dass dies nur sehr selten der Fall war. Offensichtlicher Grund dafür waren die verwendeten Policies. Die Verteilung der Eigenschaften war, bis auf die Abhängigkeit von *allow* und *thirdPartyAcces* zu *privacyLevel*, vollkommen zufällig, wodurch der Datensatz eher zu einem Rauschen verkam. Weiterhin trat jede Tabelle nur genau ein mal pro Policy auf, sodass für eine Assoziationsanalyse nicht genügend Transaktionen zusammenkamen, beziehungsweise einzelne Items nicht häufig genug in mehreren Transaktionen auftraten.. Hiermit lässt sich auch erklären, dass bereits bei niedrigen Supportwerten keine Regeln zu finden waren.

6.2.2 Versuche mit verbesserter Policy-Generierung

Um dieser Problematik zu entgehen wurde der *PolicyGenerator* wie in Abschnitt 6.1.1 beschrieben angepasst. Weiterhin wurden die zuvor erläuterten Schritte in einer Klasse `PolicyDataMiner` zusammengefasst. Deren Konstruktor nimmt eine vollständige sowie eine unvollständige Policy entgegen und soll, sofern möglich, letztere ergänzen und in die vollständige einfügen. Diese sollen dabei aus der Ausgabe des zuvor stattfindenden *Similarity-Flooding-Prozesses* entstehen[4]. Mit der Methode `getPolicy()` kann die vollständige und weitgehend ergänzte Policy abgerufen werden.
Schnell wurde erkennbar, dass auf diese Weise generierte Daten deutlich bessere Ergebnisse ermöglichen. So konnten bereits für mit einem Minimalsupport von drei in über der Hälfte der Versuche Regeln für mehr als 75% der unbekannten Eigenschaften gefunden werden. Bei niedrigeren Werten bestand weiterhin die Gefahr, dass das Verfahren nicht in annehmbarer Zeit terminiert und ein Timeout auftritt.

6.2.3 Regelkonflikte und Unsicherheit

Kommt es zu einem Konflikt zwischen Regeln, was soviel bedeutet, dass zwei oder mehr anwendbare Regeln zu unterschiedlichen Eigenschaften eines Attributs führen würden, muss die beste Regel ermittelt und angewendet werden. Dabei handelt es sich um eben jene, welche die höchste Konfidenz, bei Gleichheit den höchsten Support besitzt. Sind beide Werte gleich, bestehen verschiedene Möglichkeiten. So kann die Wahl von der Reihenfolge der Regelanwendung abhängig sein oder etwa der restriktivste oder toleranteste Wert gewählt werden. Letztere Optionen dürften, je nach Vorlieben des Nutzers, die zu bevorzugenden Wege darstellen[5].
Ist für eine Eigenschaft gar keine anwendbare Regel zu finden muss trotzdem ein entsprechender Wert gefunden werden. Die naheliegendste Methode wäre ein Standardwert, der zuvor von einem Nutzer oder Administrator definiert wurde. Ebenfalls ist auch hier der restriktivste oder toleranteste Wert gewählt werden. Eventuell kann es auch wünschenswert sein, dass der Nutzer interaktiv aufgefordert wird, nicht ermittelte Einstellungen selbst einzutragen.

[4] Eventuell muss hierbei zustätzlich noch eine Ähnlichkeitsrelation, wie in Abschnitt 6.1.2 beschrieben, übergeben werden
[5] Jedoch ist die Definition einer Ordnung auf den Werten aller Eigenschaften vonnöten, in Java wird dies durch die Implementierung der Schnittstelle *Comparable* oder *Comparator* erreicht.

Kapitel 7

Vorschlag einer Neuimplementation

In den vergangenen Kapiteln wurde die Realisierbarkeit der Ermittlung von Datenschutzeinstellungen mithilfe von Methoden der Assoziationsanalyse untersucht. Dazu wurde eine, stetigen Veränderungen unterliegende, Implementation des FP-Growth-Algorithmus, sowie verschiedenste Methoden zur Gewinnung der Policies und Anwendung der Regeln genutzt. Aufgrund dieses ständigen Wandels und den bis dato bescheidenen Erfahrungen des Autors, was die Konzeption und Programmierung objektorientierter Software, insbesondere auch in der Sprache Java, betrifft, war eine gewissenhafte Planung und Realisierung nur schwer möglich. An dieser Stelle soll daher ein Vorschlag einer Neuimplementation vorgestellt werden.
Dabei soll allerdings eine Einschränkung im Voraus genannt werden: Während für den gesamten Prozess der Gewinnung von Datenschutzeinstellungen ein Gesamtkonzept umrissen werden soll, können konkrete Implementierungsfragen nur für die Assoziationsanalyse, im gegebenen Fall also für das Verfahren *FPGrowth* beantwortet werden. Das ist damit zu begründen, dass für die Anwendung von gewonnenen Regeln auch immer zumindest einige Attribute pro Tabelle bereits mit bekannten Einstellungen versehen sein müssen. Dazu war ursprünglich das Schema-Mapping mittels *Similarity Flooding* vorgesehen, dessen Implementierung aus genannten Gründen nicht anwendbar war und daher keine Erfahrungen diesbezüglich vorliegen.

7.1 Schnittstellen

Schnittstellen spezifizieren die Interaktionsmöglichkeiten eines Nutzers mit Klassen. Sie ermöglichen eine einheitliche Verwendung von Softwarekomponenten und erlauben Austauschbarkeit von Softwarebestandteilen, solange diese die jeweilige Schnittstelle korrekt implementieren.

7.1.1 PolicyAdvisor

Zuerst soll eine Schnittstelle zur Verfügung gestellt werden, welche einem Anwender ermöglicht, eine Policy zu übergeben, der eine Menge von Attributen hinzugefügt werden soll. Für diese Attribute sollen Datenschutzeinstellungen hergeleitet und in einer aktualisierten Policy zurückgegeben werden. Diese Aufgabe übernimmt das Java Interface `PolicyAdvisor`, welches seinen Namen der Tatsache verdankt, dass es dem Nutzer beim Ergänzen von Datenschutzeinstellungen beratend zur Seite steht. Eine Implementation dieser Schnittstelle kann als Umsetzung des Design Pattern *Facade* angesehen werden, da es, bis auf `PolicyItem` sämtliche weiteren Klassen vor dem Anwender verbirgt.
Notwendige Angaben sind die zu ergänzende Policy, welche mittels `setPolicy(Policy)` definiert wird und die Menge der hinzuzufügenden Attributnamen via `setUnknownAttributes(Set<String>)` oder `addUnknownAttribute(String)`. Hierbei muss für sämtliche Bezeichnungen von Attributen, wie schon in Abschnitt 6.1.1 erläutert, die Namenskonvention *tabellenname.attributname* eingehalten werden, da die

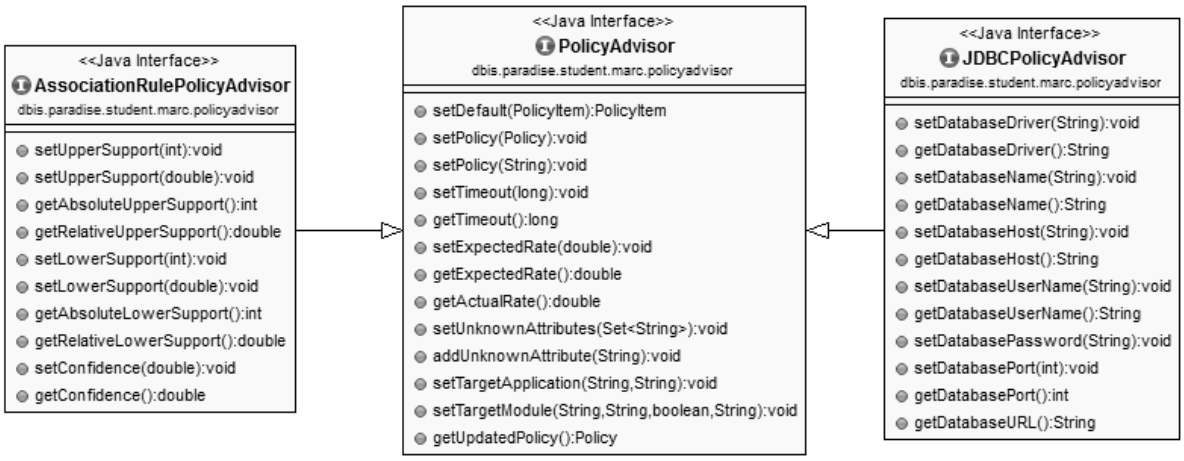


Abbildung 7.1: Interface *PolicyAdvisor* sowie die Erweiterungen *JDBCPolicyAdvisor* und *AssociationRulePolicyAdvisor*.

Klasse `Policy` kein Tabellenkonzept kennt.
Kann das Verfahren keinen passenden Wert für eine Datenschutzeinstellung herleiten, muss ein Standardwert gewählt werden. Dieser kann, für jede Einstellung einzeln (z.B. *PrivacyLevel*, *Allow* etc.), durch den Nutzer im Voraus festgelegt werden, indem ein entsprechendes `PolicyItem` mit gewünschtem Wert an die Methode `setDefault(PolicyItem)` übergeben wird. Ein eventuell im `PolicyItem` hinterlegter Attributname wird dabei ignoriert. Weiterhin kann der Nutzer Forderungen an den minimalen Anteil der vom Verfahren hergeleiteten Eigenschaftswerte festlegen[1]. Dies erlaubt es Implementierungen, die iterativ immer bessere Lösungen suchen, eine geeignete Abbruchbedingung zur Verfügung zu stellen. Im Falle der Herleitung von Werten mittels einer Assoziationsanalyse kann beispielsweise mit einem hohen Minimalsupport begonnen werden.
Genügt das Ergebnis nicht den eben beschriebenen Bedingungen, weil nicht ausreichend Regeln gefunden werden können, um eine entsprechende Anzahl von Eigenschaften zu bestimmen, kann eine weitere Regelgenerierung mit vermindertem Support durchgeführt werden. Das tatsächliche Verhältnis des zuletzt unternommenen Versuch, Datenschutzeinstellungen zu finden, kann mit der Methode `getActualRate()` erfragt werden. Als weiteres Abbruchkriterium für iterativ verbessernde Methoden sowie als Schutz des Nutzers vor langsam oder gar nicht terminierenden Verfahren kann außerdem ein Zeitlimit in mittels `setTimeout(long)` angegeben werden. Wird dieses beim Herleiten überschritten, bricht das Verfahren mit einer `TimeLimitExceededException` ab.
Angaben zu Standardwerten, gefordertem Anteil an tatsächlich hergeleiteten Eigenschaften und Zeitlimit sollen dabei vollkommen optional bleiben. Das bedeutet, dass es nicht als Fehlverhalten angesehen werden darf, wenn genannte Konfigurationen nicht getätigt werden. Entsprechend muss die Implementierung Standardwerte zur Verfügung stellen.

Erweiterungen der PolicyAdvisor-Schnittstelle

Die Schnittstelle *PolicyAdvisor* wurde bewusst minimal gehalten, um die Art der Herleitung der Eigenschaften nicht unnötig zu erwzingen. So wäre es denkbar, andere Data-Mining-Verfahren als die Assoziationsanalyse zu verwenden. Jedoch kann beispielsweise ein Klassifikationsverfahren mit der Angabe eines Minimalsupports nicht viel anfangen und benötigt stattdessen, je nach konkreter Technik, andere Parameter.

[1] Andersherum kann gesagt werden, es wird ein maximaler Anteil von mit Standardwerten belegten Eigenschaften erlaubt.

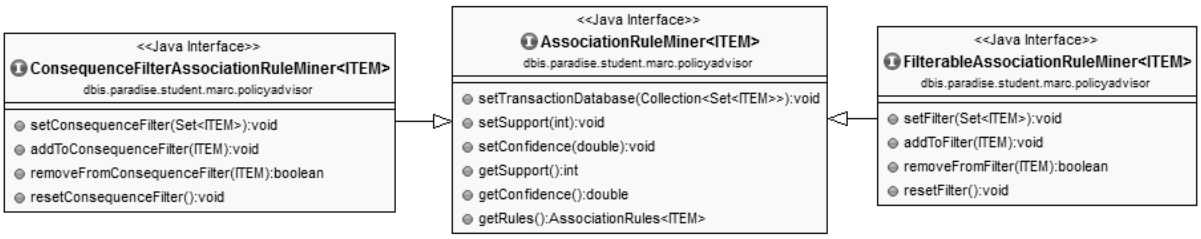


Abbildung 7.2: Interface `AssociationRuleMine` sowie die Erweiterungen `FilterableAssociationRuleMiner` und `ConsequenceFilterAssociationRuleMiner`.

Aus diesem Grund soll die Schnittstelle durch weitere Schnittstellen erweiterbar sein. So ermöglicht `JDBCPolicyAdvisor` etwa die Angabe von Parametern, die zum Verbinden mit einem Datenbankmanagementsystem via JDBC notwendig sind. Dies ist im ursprünglich angedachten Szenario der Fall, wenn vor der der Assoziationsanalyse ein Schema Mapping mithilfe von Similarity Flooding durchgeführt werden muss. Als verschärfende Bedingung sei hier gefordert, dass Attribute und Tabellen, sowohl aus der Policy, als auch die hinzuzufügenden Elemente, in selbiger Form auch im spezifizierten Datenbankschema vorkommen müssen. Es ist offensichtlich, dass das Verfahren nicht sinnvoll durchgeführt werden kann, wenn diese Forderung nicht eingehalten wird.
Schließlich sollen noch Schnittstellen zur Konfiguration der Assoziationsanalyse definiert werden. Konkret erfolgt dies im Interface `AssociationRulePolicyAdvisor`. Sie stellt Methoden zur Eingabe von oberer und unterer Grenze des Minimalsupports[2] sowie die gewünschte Konfidenz. Dabei lassen sich alle Angaben sowohl relativ von der Anzahl der Transaktionen, als auch absolut angeben. Notwendige Einschränkungen sind hierbei, dass zum einen die obere Grenze des Minimalsupports nicht geringer sein darf, als die Untere, sämtliche Angaben sind größer als Null und insbesondere relative Werte befinden sich im Intervall $(0; 1]$.
Zudem sind noch weitere Erweiterungen denkbar. So kann es etwa notwendig sein, dass besonders unsichere Zuweisungen von Werten erst durch den Nutzer bestätigt werden müssen. Andernfalls soll die Angabe einer gewünschten Einstellung möglich sein. Konkretisierungen einer solchen interaktiven Schnittstelle existieren derzeitig nicht.

7.1.2 RuleAssociationMiner

Weiterhin soll eine allgemeine Spezifikation zur Durchführung einer Assoziationsanalyse definiert werden. Diese ist generisch bezüglich des Datentyps der zu betrachtenden Items. Es ermöglicht die Angaben der Transaktionsdatenbank (`setTransactionDatabase(TransactionDatabase)`), der (absolute) Minimalsupport und -konfidenz (`setSupport(int)`, `setConfidence(double)`). Schlussendlich muss die Analyse durchgeführt und Regeln zurückgegeben werden (`getRules()`). Transaktionsdatenbank und Regelmenge liegen dabei wiederum in Objekten vor, die entsprechende Schnittstellen implementieren müssen, diese werden in den nachfolgenden Abschnitten erläutert.
Auch dieses Interface soll erweiterbar sein. Zum einen kann es gewünscht sein, dass, wie in Abschnitt 5.3.2 vorgeschlagen, die Menge der Items, aus denen Regeln gebildet werden, einzuschränken. Hierfür bietet die generische Schnittstelle `FilterableAssociationRuleMiner` Methoden um diese Menge zu definieren. Zudem existiert eine weitere Möglichkeit die Anzahl der erzeugten Regeln maßgeblich zu reduzieren. Dazu wird eine Menge von Zielitems definiert. Regeln, deren Konsequenz nicht mindestens ein Item enthalten, welches auch in dieser Zielmenge vorkommt, sollen gar nicht erst erzeugt werden. Wenngleich diese Methode in der ersten Implementation nicht verwendet wurde und dementsprechend keine Erfahrungs-

[2] Eine obere und eine untere Grenze des Supports werden benötigt, wenn das Verfahren versucht, iterativ eine bessere Lösung zu finden.

werte vorliegen, ist sie sehr vielversprechend und sollte die Leistung positiv verändern. Das entsprechende Interface heißt `ConsequenceFilterAssociationRuleMiner`.

7.1.3 TransactionDatabase

Der Wahl einer geeigneten Darstellungsform der Transaktionsdatenbank sind zahlreiche Überlegungen vorangegangen. Die bisher gewählte Form `List<Set<ITEM>>` hat zwei entscheidende Nachteile. Zum einen muss die gesamte Transaktionsdatenbank zumindest zu Beginn vollständig im Speicher gehalten werden, was bei großen Datensätzen und Systemen mit geringem Hauptspeicher schnell problematisch werden kann. Weniger offensichtlich ist die zweite Problematik. Da Listen in Java nicht zwangsweise vor Veränderung geschützt werden können, ist es in Multi-Thread-Programmen zumindest theoretisch möglich, Transaktionen während der Assoziationsanalyse hinzuzufügen, oder zu löschen. So kann, am Beispiel von *FPGrowth* etwa, nach der Ermittlung der häufigen Items eine Transaktion gelöscht werden. Die nachfolgende Generierung der häufigen Itemsets ist nun offensichtlich nicht mehr korrekt.
Abhilfe schaffen soll die generische Schnittstelle `TransactionDatabase`. Sie implementiert das Interface `Iterable` der Java Standard Library und ermöglicht damit Zugriff auf eine unveränderbare Transaktionsdatenbank. Das bedeutet, dass jeder Aufruf der Methode `iterator()` einen Iterator liefert, der dieselben Transaktionen durchläuft[3]. Dabei dürfen gleiche Transaktionen explizit mehrfach auftauchen. Entsprechend liefert die Methode `size()` stets dieselbe Anzahl aller Transaktionen in der Datenbank. Eine Reduktion auf einzigartige Elemente findet nicht statt.
Wird die Transaktionsdatenbank als Objekt einer Klasse dargestellt, die diese Schnittstelle implementiert, kann zumindest das erste Problem gelöst werden. So ist zum Beispiel auch möglich Daten via JDBC aus einer Datenbank oder als Datenstrom aus einer Datei zu gewinnen. Quellen dieser Art erlauben es, die geforderten Informationen sequentiell abzurufen, wodurch die jeweilige Implementation entscheiden kann, ob die gesamte Transaktionsdatenbank tatsächlich im Speicher gehalten wird.
Leider sichert auch diese Schnittstelle nicht zwangsläufig Threadsicherheit zu; diese muss durch die konkrete Implementation zugesichert werden. Existieren Referenzen auf die *Set*-Objekte oder deren Elemente ist eine Manipulation weiterhin möglich. Genaue Details zu dieser Thematik entziehen sich derzeit der Kenntnis des Autors dieser Arbeit, weshalb es ruhen gelassen werden soll.

7.1.4 AssociationRules und RuleIterator

Als letzte Schnittstelle soll *AssociationRules* vorgestellt werden. Sie soll in erster Linie die Suche nach anwendbaren Regeln und deren Konsequenzen ermöglichen. Die tatsächliche Herausgabe von Regelobjekten, wie es im ersten Implementationsversuch der Fall war, ist eher uninteressant und es hat sich gezeigt, dass das häufige und unnötige Erzeugen von Objekten kontraproduktiv ist. `getSupport()` und `getConfidence()` liefern den minimalen Support- beziehungsweise Konfidenzwert aller gespeicherten Regeln zurück. `AssociationRules` implementiert ebenfalls die Schnittstelle `Iterable`. Dementsprechend kann mithilfe der Methode `iterator()` ein besonderer Iterator über alle Konsequenzen der gespeicherten Regeln angefordert werden. Wird hingegen `iterator(Set<ITEM>, int, double)` aufgerufen, läuft der entsprechende Iterator nur über die Konsequenzen der Regeln, die auf die übergebene Transaktion anwendbar sind und gleichzeitig den angegebenen Werten für Support und Konfidenz genügen.
Allerdings ist das Iterieren über die vollständige Menge der Regelkonsequenzen ohne zusätzliche Informationen über die Regel an sich nicht zweckmäßig. Daher wurde eine Erweiterung der Schnittstelle `Iterator` vorgenommen. `RuleIterator` bietet zusätzlich Methoden zur Abfrage von Support, Konfidenz und Antezedens der zuletzt abgefragten Regel.

[3] Jedoch nicht zwangsläufig immer in derselben Reihenfolge.

7.1.5 Sonstige Schnittstellen

Transaktionen und Regelkomponenten sollen in der Regel nicht veränderlich sein. Allerdings bietet die verwendete Schnittstelle *Set* eben solche Methoden zur Manipulation an. Aus diesem Grund sollen, in Anlehnung an das Design Pattern *Proxy*, entsprechende Mengen in Stellvertreterobjekten gekapselt werden. Dazu erweitert die Schnittstelle *LockableSet* die verwendeten Sets um die Methoden `lock()` und `isLocked()`. Klassen, die dieses Interface implementieren funktionieren wie gewöhnliche Mengen, bis sie durch den Aufruf von `lock()` gesperrt werden. Danach sind alle manipulierenden Methoden verboten und führen zu einer *IllegalStateException*.
Im ersten Implementationsversuch stellte es sich schnell heraus, dass die häufige Erzeugung von kurzlebigen Objekten die Achillisferse darstellte. Insbesondere bei der Suche von Assoziationsregeln in den häufigen Itemsets sorgte die übermäßige Anforderung neuer Mengenobjekte häufig dazu, dass der Garbage Collector nicht mit dem bereinigen des Speichers nachkam. Neben der Reduzierung der zu untersuchenden Kandidaten für Regeln kann Wiederverwendung von Regelobjekten Abhilfe schaffen[4]. Eine erzeugte und nicht mehr benötigte Menge kann geleert und an anderer Stelle wiederverwendet werden, was dazu führt, dass sie nicht als verwaistes Objekt im Speicher liegt, diesen relativ lange belegt und erst spät vom Garbage Collector entfernt wird. Das generische Interface `ObjectPool<T>` nimmt sich dieser Problematik an. Die Methode `getObject()` liefert ein gewünschtes Objekt, welches zuvor in einem Vorrat lag, zurück. Ist dieser leer, wird eine neue Instanz erzeugt. Nicht mehr benötigte Objekte können mit `restoreObject(T)` wieder zum Vorrat hinzugefügt werden. Die Implementierung sollte dafür sorgen, dass es in einen Ausgangszustand versetzt wird, bevor es aufs Neue in Umlauf gebracht wird. Bei einer Menge funktioniert dies mit der Methode `clear()`[5].
Wie es auch in der ersten Implementation der Fall war, soll auch im zweiten Versuch ein geeigneter Container für häufige Itemsets, die das Produkt der Algorithmen *Apriori* und *FPGrowth* darstellen, zur Verfügung gestellt werden. Die Schnittstelle `FrequentItemSet` beschränkt sich dabei diesmal auf Methoden zum Hinzufügen von Itemsets (`update(Set<T>, int)`), zum Überprüfen auf Vorhandensein, beziehungsweise zum Erfragen des Supports (`contains(Set<T>)`), sowie die Anforderung eines Iterators auf alle gespeicherten Teilmengen eines Itemsets. Auf die Erzeugung der Regeln mithilfe von Objekten, die diese Schnittstelle implementieren, wird verzichtet.

7.2 Konkrete Implementationsvorschläge

7.2.1 Transaktionsdatenbanken aus Policies

An dieser Stelle soll ein simpler Vorschlag zur Implementierung einer Transaktionsdatenbank aus einer Policy vorgestellt werden. Hierbei wird, vergleichbar mit der ersten Implementation, die gesamte Policy durchlaufen und daraus eine interne Repräsentation der Transaktionsdatenbank erstellt. Diese entspricht dem Datentypen `ArrayList<LockableHashSet<PolicyItem>>`. Um zu verhindern, dass der durch `Arraylist.iterator()` erzeugte Iterator die optionale Methode `remove()` implementiert, wird hierfür eine Proxyklasse genutzt. Diese fängt entsprechende Nachrichten ab und erschwert somit eine Veränderung der Transaktionsdatenbank deutlich.
In Abschnitt 6.1.2 wurde erläutert, dass es angeraten ist, für jede Eigenschaft der Datenschutzeinstellungen eine eigene Assoziationsanalyse durchzuführen. Das bedeutet, dass jeweils eine eigene

[4] Diese Behauptung stellte sich im späteren Verlauf als unhaltbar heraus. Der tatsächliche Grund konnte bis zum Ende der nicht gefunden werden.

[5] Wird ein Objekt in den Vorrat zurückgeführt, verschwindet nicht notwendigerweise die Referenz, die der Nutzer zuvor hatte. Auch weitere Referenzen können ohne Weiteres existieren. So ist es möglich, dass das Objekt, während es im Vorrat liegt oder bereits einem neuen Anwender zugewiesen wurde, weiterhin manipuliert werden kann, was zu schwerwiegenden Konsistenzproblemen oder gar Sicherheitsrisiken führen kann. Diese Thematik liegt allerdings nicht im Fokus dieser Arbeit, die Nutzung solcher Objekte in einer kontrollierten Umgebung, beispielsweise in einer Klasse gekapselt, sollte problemlos sein.

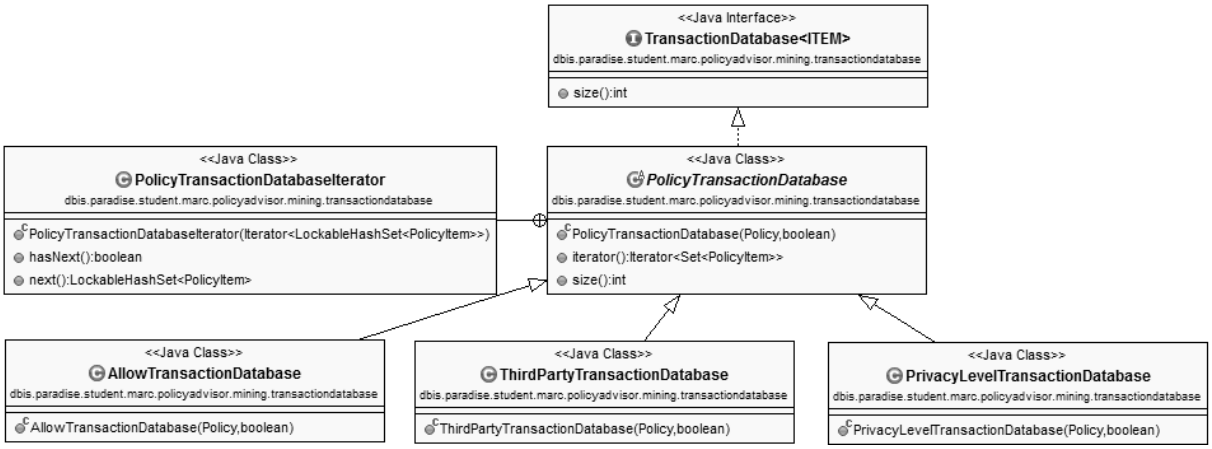


Abbildung 7.3: Klassendiagramm der Implementation einer Transaktionsdatenbank aus einer Policy

Transaktionsdatenbank vonnöten ist. Aus diesem Grund ist die Klasse *PolicyTransactionDatabase* als abstrakt deklariert, wodurch keine Instanz von ihr erzeugt werden kann. Zwar implementiert sie bereits den Großteil der Erstellung der Datenbank, jedoch ist die dazu benötigte Methode `getPolicyItem(Attribute)` ebenfalls abstrakt und muss einer Kindklassen implementiert werden[6]. Dies wird durch die Klassen `AllowTransactionDatabase`, `PrivacyLevelTransactionDatabase` und `ThirdPartyTransactionDatabase` bewerkstelligt. Es kann gewünscht oder auch unerwünscht sein, dass neben den Tupeln aus Attributname und Wert zusätzlich auch nur die Namen der Attribute als Item auftauchen. Dazu kann der Schalter `addName` des Konstruktors von `PolicyTransactionDatabase` entsprechend belegt werden.
Zu Testzwecken wurden außerdem die Klassen `TrivialTransactionDatabase` und `DSTTransactionDatabase` implementiert. Erste erwartet die Eingabe einer `List<Set<ITEM>>` und stellt diese als Transaktionsdatenbank zur Verfügung. `DSTTransactionDatabase` hingegen parst DST-Dateien (Abschnitt 5.3.1) und ermöglicht Assoziationsanalysen mit auf dieser Weise gespeicherten Daten. Beide Klassen sind als experimentell und unvollständig zu betrachten und sollten daher nicht genutzt werden.

7.2.2 Ein einfacher Container für Assoziationsregeln

Aufgrund der fortgeschrittenen Zeit dieser Arbeit wird zur Speicherung der eine recht einfache, allerdings auch weniger effiziente Methode implementiert. Da jedoch im gegebenen Fall die Anzahl der auf anwendbare Regeln zu überprüfenden Transaktionen relativ gering ist, sollte diese Tatsache weniger ins Gewicht fallen. Anschließend wird noch ein Vorschlag für eine effizientere Methode vorgestellt, dessen tatsächliche Implementierung nicht mehr geplant ist[7].
Die Klasse `SimpleAssociationRules` kapselt Assoziationsregeln als 4-Tupel und speichert diese in einer Menge. Hinzugefügt werden können Regeln mithilfe der Methode `addRule(LockableSet<ITEM>, LockableSet<ITEM>, int, double)`. Beide `iterator()`-Methoden liefern einen gekapselten Iterator der Regelmenge. Wird eine Filterung gewünscht, werden von diesem entsprechende Regeln übersprungen. Offensichtlicher Nachteil ist hierbei, dass dabei sämtliche im Container gespeicherte Regeln auf Anwendbarkeit überprüft werden müssen. Die Prüfung, ob eine Menge Teilmenge einer anderen ist, hat eine lineare Zeitkomplexität. Wird angenommen, dass im Container n Regeln gespeichert sind und insgesamt m Items existieren, kann im schlimmsten Fall von einer Komplexität $\Theta(m \cdot n)$ ausgegangen werden.

[6] Gamma et al beschreiben dieses Design Pattern als *Template Method* [GHJV].
[7] Bis zum Abschluss dieser Arbeit gelang es, eine prototypische Implementierung vorzunehmen.

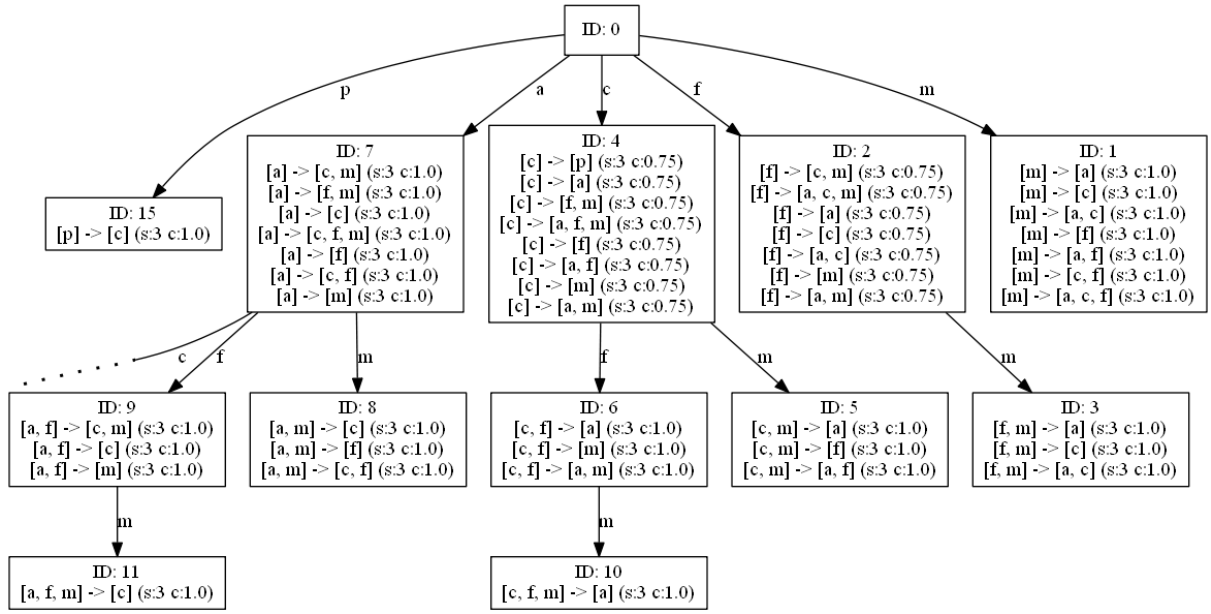


Abbildung 7.4: Gekürzte Darstellung einer Trie-basierten Datenstruktur zur Speicherung der aus der Tabelle 5.1a dargestellten Transaktionsdatenbank gewonnenen Regeln.

Kurze Vorstellung effizienterer Datenstrukturen

Eine Möglichkeit zur effizienteren Suche nach passenden Regeln, oder allgemeiner die Suche nach Teilmengen, stellen digitale Bäume dar, beispielsweise *Tries*. Diese werden unter anderem in [ES11, SSH11] erläutert und werden beispielsweise zur Indizierung von Texten verwendet.
Regeln sollen entsprechend ihres Antezedens in einen Trie eingefügt, was erst möglich ist, wenn dessen Elemente entsprechend einer festgelegten Ordnung sortiert werden. Dazu wird der in Abschnitt 7.2.6 erwähnte `GenericComparator` verwendet. An jedem Knoten existiert für jedes Item maximal eine ausgehende Kante die auf den nächsten Knoten verweist. Jeder Knoten kann dabei mehrere Regeln enthalten. Dabei werden die Konsequenzen als Schlüssel der Datenstrukturen `Map<Set<ITEM>,Integer>>` und `Map<Set<ITEM>,Double>>` dargestellt, die jeweiligen Werte entsprechen Support und Konfidenz.
Die Suche nach anwendbaren Assoziationsregeln für eine gegebene Transaktion gestaltet sich als denkbar einfach. Anders formuliert werden alle Regeln gesucht, für deren Antezedenzien die eingegebene Menge eine Teilmenge ist. Dazu sollen Pfade von der Wurzel zu den Blättern gefunden werden, auf denen alle in der Transaktion vorhandenen Items vorkommen. Wird auf einem Pfad das höchstwertigste Item der Eingabe gefunden, wird in den Nachfolgern das nächsthöhere Element gesucht. Wird dabei ein Knoten über einen Pfad erreicht, der alle Elemente der Transaktion enthält, so können alle darin gespeicherten Regeln, sowie die sämtlicher Nachfolger, der Zielmenge hinzugefügt werden. Aufgrund der Trie-Struktur kann dabei die Suche an Knoten abgebrochen werden, deren repräsentiertes Item geringer ist, als jenes gesuchte. Das Suchmuster entspricht einer beschnittenen Tiefensuche.

7.2.3 Implementierung des Algorithmus FPGrowth

Entsprechend der Spezifikation der Schnittstellen, welche in Abschnitt 7.1.2 vorgestellt wurden, soll die Implementation des Algorithmus *FPGrowth* neu konzipiert werden. Dabei wird ausschließlich die Klasse `FPGrowth` für den Anwender sichtbar sein, alle weiteren Klassen, insbesondere die Komponenten des Frequent-Pattern-Trees, werden in versteckten Klassen implementiert und sind so für den Nutzer nicht

von Belang[8]. Außerdem soll sich dabei nicht nur auf die Erzeugung der häufigen Itemsets beschränkt, sondern anschließend auch die Regelgenerierung vorgenommen werden.

Konstruktion von Frequent-Pattern-Trees

Für die Darstellung des Frequent-Pattern-Trees wird nicht, wie es in der ersten Implementierung noch der Fall war, eine eigene Klasse definiert. Stattdessen besteht die Datenstruktur lediglich aus der Wurzel `FPTreeRoot` und mehreren Knoten `FPTreeNode`. Zur Vereinfachung der Handhabung der Komponenten wurden diese unter `AbstractFPTreeNode` zusammengefasst. Weiterhin wurden die Klassen `ItemComparator` zum Definieren einer Ordnung auf Items anhand ihrer Häufigkeit und `FPTreeNodeTool` als Implementierung von `AbstractObjectPool` implementiert. Alle genannten Klassen sind generisch bezüglich des Item-Typs.
Da die Konstruktion des Frequent-Pattern-Trees sich im Wesentlichen nicht von der ersten Implementierung unterscheidet, soll hier nur auf die Unterschiede eingegangen werden. Die Datenstruktur wird erstellt, indem zuerst ein Wurzelobjekt erzeugt wird. Dessen Konstruktor erwartet die Übergabe einer Transaktionsdatenbank, welche als Quelle dient. Beim Zählen der häufigen Items besteht nun keine Abhängigkeit zur *Guava*-Bibliothek mehr, statt des `MultiSet` findet nun eine einfache `Map<ITEM,Integer>` Verwendung. Die Verwendung des oben bereits genannten `ItemComparator`, welcher für den Datentyp `ITEM`, statt wie zuvor `Pair<Integer,ITEM>` definiert ist, ist als eleganter anzusehen. Knoten mit gleichen Items waren untereinander nach der Art von verketteten Listen miteinander Verbunden. Dazu verfügte jeder Knoten über ein Verweis `nodeLink` auf den nächsten. Stattdessen definiert die *Frequent-Item Header Table* nun nicht mehr die jeweiligen Einstiegspunkte dieser Ketten, sondern referenziert jetzt eine Liste von entsprechenden Knoten.
Auch die Konstruktion des Frequent-Pattern-Trees entspricht im Wesentlichen dem ersten Ansatz. Neue Knotenobjekte werden nicht mehr bei Bedarf erzeugt, sondern im `FPTreeNodeTool` angefordert, in den sie bei Abbau des Baumes zurückgeführt werden. Dazu sollte im Anschluss die Methode `FPTreeRoot.tearDown()` aktiviert werden. Diese Entscheidung ist damit begründet, dass die Knoten häufig erzeugt, verworfen und später erneut erzeugt werden, was zu Lasten der Leistung geht. Bei der Erzeugung einer Repräsentation des Baumes im Dot-Format, welche zu Informations- und Testzwecken generiert werden kann, wird nun auf die Verwendung der *Wintersleep-Graphviz* Bibliothek verzichtet.

Extraktion der häufigen Itemsets

Bei der Extraktion der häufigen Itemsets mittels des Algorithmus *FPGrowth* halten sich die Unterschiede zur ersten Implementation ebenfalls in Grenzen. Diese beschränken sich im Wesentlichen auf Anpassungen an veränderte Datenstrukturen im Baum selbst. Entgegen der ursprünglichen Bestrebungen wurde die Abhängigkeit von der Guava-Library zur Erzeugung von Potenzmengen vorerst nicht aufgehoben. Zur Konstruktion der konditionalen Frequent-Pattern-Trees aus den *conditional patterns* existiert die Klasse `ConditionalPatternSet` als versteckte Implementation von `TransactionDatabas`. Die Menge der häufigen Itemsets wird durch eine Implementation des Interfaces `FrequentItemSet` gespeichert.

Suche nach Regeln

Die Regelextraktion unterscheidet sich nicht von der ersten Implementation. Aus Zeitgründen wurde auf eine Filterung der erzeugten Regeln nach Items in der Konsequenz verzichtet. Daher kann die Klasse `FPGrowth` das Interface `ConsequenceFilterAssociationRuleMiner` nicht implementieren.

7.2.4 Weitere Anmerkungen

Zur einfachen Nachvollziehbarkeit des Verfahrens wurden Methoden zur Erzeugung einer grafischen Darstellung geschaffen. Dazu muss die Instanz von `FPGrowth` in den Debug-Modus versetzt werden, was

[8] Es soll nicht unerwähnt bleiben, dass diese Struktur durch die Weka-Implementation inspiriert ist [WFM05].

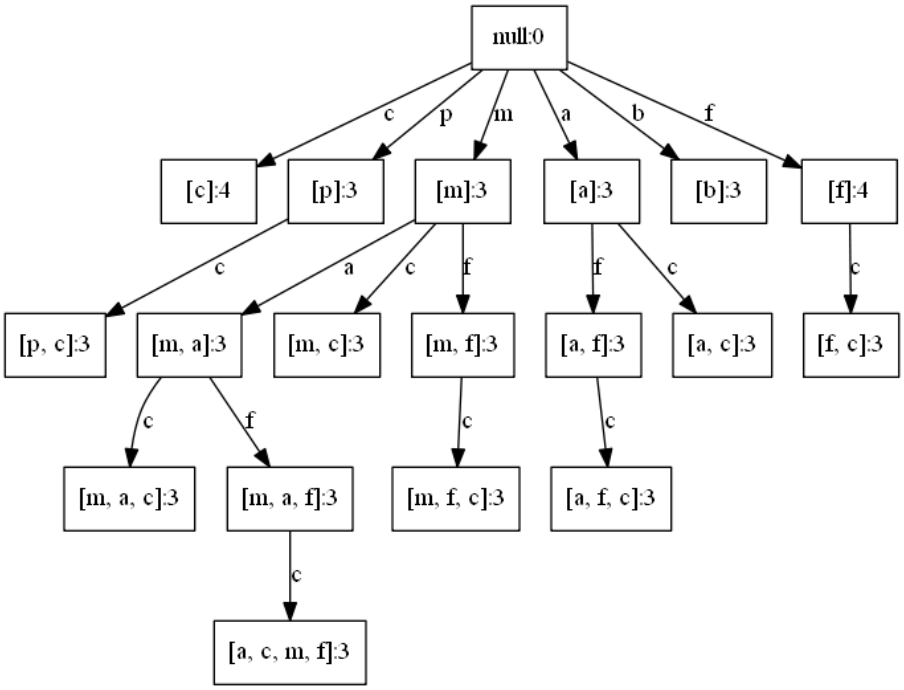


Abbildung 7.5: Darstellung der aus Tabelle 5.1a gewonnenen häufigen Itemsets in einem `FrequentItemsetTrie`. Die Beschriftung der Knoten zeigt die jeweils gespeicherte Menge, sowie den dazugehörigen Support.

mit `setDebug(true)` möglich ist. Wird mittels `getRules()` nun die Suche häufiger Itemsets gestartet, wird außerdem eine *Dot*-Repräsentation jedes einzelnen Frequent-Pattern-Trees erzeugt. Diese werden im Verzeichnis *target/debug*[9] abgelegt. Wird das ebenfalls im genannten Verzeichnis abgelegte Bash-Skript *build.sh* ausgeführt, erstellt es eine hierarchische Darstellung aller Bäume.

7.2.5 Datenstruktur zur Speicherung häufiger Itemsets

Zur Speicherung der häufigen Itemsets wurd die Klasse `FrequentItemSetTrie` als Implementierung der Schnittstelle `FrequentItemSets` realisiert. Statt, wie es im ersten Versuch der Fall war, Mengen und Supportwerte Tabellenstrukturen zu speichern, sollte eine Möglichkeit gewählt werden, die das effizientere Suchen von Teilmengen ermöglicht. Diese wird gerade bei der Regelgenerierung häufig durchgeführt. Werden entsprechende Mengen in Form eines Tries organisiert, ähnlich des Vorschlags einer Datenstruktur für Regelmengen in Abschnitt 7.2.2, ist dies problemlos möglich. Im Gegensatz zum `AssociationRulesTrie` wird pro Knoten jedoch nur ein einzelnes Element gespeichert. Außerdem gestaltet sich die Suche anders. Während bei der Suche nach Regeln, die auf die eingegebene Menge anwendbar sind, nur solche ausgegeben werden, welche die Eingabe vollständig in ihrem Antezedens beinhalten, genügt es, wenn hier mindestens ein Element vorhanden ist. Daher können bereits alle Nachfolger eines Knotens ausgegeben werden, wenn dieser ein solches beinhaltet. Die Implementation bietet eine Ausgabe in Dot-Notation an.

[9] Aufgrund persönlicher Präferenzen des Autors wurde für die Implementation das Tool *Maven* gewählt, welches gewisse Konventionen voraussetzt.

7.2.6 Sonstige Implementierungsvorschläge

Zur Darstellung der Items der Transaktionsdatenbank wird weiterhin die in Abschnitt 6.1.2 eingeführte Klasse *PolicyItem* mitsamt ihren Kindklassen genutzt. Bei genauerer Betrachtung ist festzustellen, dass es sich bei Instanzen der beschriebenen Klassen um Objekte handelt, die zum einen während der Laufzeit nicht mehr verändert werden und zum anderen potentiell mehrfach auftreten. Daher ist in diesem Fall die Anwendung des Design Pattern *Flyweight* durchaus sinnvoll, um unnötige Objekterzeugung und Speicherbelegung zu vermeiden. Dies wurde realisiert, indem zuerst die Sichtbarkeit sämtlicher Konstruktoren von *public* auf *private* gesetzt wurde. Dadurch ist es anderen Objekten nicht mehr möglich, neue Instanzen von *PolicyItem* zu erzeugen. Stattdessen wird dies nun an eine statische Methode `get(Attribute)` pro Klasse delegiert, welche überprüft, ob ein entsprechendes Objekt bereits existiert und dieses an den Aufrufer zurückgibt. Andernfalls wird eine neu erzeugte Instanz übergeben und für den nächsten Aufruf festgehalten. Die Speicherung der erzeugten Instanzen erfolgt in *MultiKeyMap*s, welche in den *Apache Commons Collections*[10] enthalten sind. Sie stellt die Funktionen von *Map*s, jedoch mit mehreren Schlüsseln zur Verfügung.
Die Schnittstelle `ObjectPool` wurde in drei Hierarchiestufen implementiert. An oberster Stelle steht die abstrakte Klasse `AbstractObjectPool`, welche die Speicherung von Objekten in einer Queue (`LinkedList`) speichert. Erbende Klassen müssen die unsichtbaren Methoden `create()` und `reset(T)` implementieren, welche festlegen, wie ein Objekt erzeugt, beziehungsweise in einen Zustand versetzt wird, der dem eines neu erzeugten Objektes ausreichend ähnlich ist. `AbstractSetPool` ist ebenfalls eine abstrakte Klasse, sie erbt von `AbstractObjectPool` und implementiert `reset(Set)`, indem es lediglich die Menge mit `clear()` leert. Tatsächlich instanziierbar ist die Klasse `HashSetPool`, welche zusätzlich die `create()` Methode umsetzt. Bis auf `release(T)` laufen alle Methoden in linearer Zeit. Dies ist der Fall, weil Sets, welche zur Implementierung die erste Wahl gewesen wären, Duplikate im Allgemeinen mittels `equals()` erkennen, anstatt der Referenz. Daher muss bei der Überprüfung, ob ein Element bereits im Pool ist, die Queue vollständig durchlaufen werden.
Weiterhin steht mit der Klasse *LockableHashSet* eine Implementierung von *LockableSet* bereit. Diese kapselt ein *HashSet*, verwaltet den Zustand der Menge und delegiert entsprechende Methodenaufrufe weiter.
Einige genutzte Verfahren benötigen eine beliebige und dennoch festgelegte totale Ordnung auf bestimmte Objekte. Dazu wurde die Klasse `GenericComparator` implementiert. Sie verwaltet intern eine Reihenfolge von Objekten einer Klasse. Wird ein Vergleich mit einem bisher unbekannten Objekt angefordert, wird dieses der Reihenfolge am Ende hinzugefügt.

7.3 Stand der Implementation am Ende dieser Arbeit

Die aktuellste Version der Implementation liegt im SVN des PArADISE-Frameworks in der Revisionsnummer 221 vor. Dabei ist sie im Paket `dbis.paradise.student.marc.policyadvisor` organisiert. Alle beschriebenen Interfaces wurden definiert und nahezu vollständig in *JavaDoc* dokumentiert.
`FPGrowth` ist implementiert und die Extraktion von Regeln funktioniert bereits. Allerdings fehlen die Funktionalitäten des Interfaces `ConsequenceFilterAssociationRuleMiner`. Für die Schnittstellen `FrequentItemSets` und `AssociationRules` wurden die angesprochenen Trie-Datenstrukturen implementiert und angewandt. Ebenso sind die `PolicyTransactionDatabase`-Implementationen einsatzbereit.
Alles in allem sind sämtliche, zur Regelextraktion in Policies notwendigen Klassen implementiert worden. Dabei wurden bisher keine Tests geplant und durchgeführt, welche die Korrektheit zeigen können.

[10] Aktuell ist Version 4.0. Da allerdings im PArADISE-Framework bereits die Version 3.2.1 in Verwendung ist, wird diese genutzt.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Diese Arbeit beschreibt die Anwendung von Data-Mining-Verfahren zur Ermittlung von Datenschutzeinstellungen. Dabei fiel die Wahl auf die Assoziationsanalyse, insbesondere auf das Verfahren *FPGrowth*. Mithilfe einer rudimentären Implementierung gelang es für künstlich erzeugte Datenschutzeinstellungen entsprechende Eigenschaften von hinzugefügten Attributen herzuleiten. Dazu wurden Tabellen auf Transaktionen und Attribute mitsamt ihren Einstellungen auf Items abgebildet. Für die Eigenschaften *allow*, *third_party_access* und *privacyLevel* wurde jeweils eine eigene Transaktionsdatenbank erstellt und aus diesen Regeln gewonnen. Wurden diese auf unvollständige Datenschutzeinstellungen zu hinzuzufügenden Tabellen angewendet, konnten häufig viele bis alle Eigenschaften ermittelt werden.
Dabei wurden in Versuchen relativ niedrige Werte für den minimalen Support benötigt, um ausreichend Regeln zu finden. Dies wirkte sich allerdings negativ auf die Laufzeit des Verfahrens aus, mitunter stürzte die Implementierung ab.
Aus diesem Grund wurde eine Neuimplementation vorgeschlagen, welche zum einen ein sauberes Konzept bietet und zum anderen mit niedrigen Supportwerten umgehen können soll. Diese umfasst neben Schnittstellen und Klassen für die Assoziationsanalyse auch Konzepte für den Gesamtprozess der Herleitung von Datenschutzeinstellungen.
Weil die Anwendung von Regeln auf Tabellen, deren Einstellungen komplett unbekannt sind, nicht möglich ist, eignet sich diese Technik nur als Unterstützung von Verfahren, die zumindest einen Teil dieser Informationen herleiten können. Dazu war ursprünglich ein Schema-Mapping-Prozess angedacht, dessen Ergebnis durch Data-Mining vervollständigt werden sollte. Aufgrund von Problemen mit der *Similarity Flooding*-Implementation des PArADISE-Frameworks konnten hierzu keine Erkenntnisse gewonnen werden.

8.2 Ausblick

- An erster Stelle sollte die Komplettierung der Implementation der Assoziationsanalyse stehen. Zum Ende dieser Arbeit ist bereits der Großteil fertig. Jedoch muss die Korrektheit und Funktionstüchtigkeit noch festgestellt werden. Dies sollte im Wesentlichen durch jUnit[1]-Testklassen realisierbar sein. Insbesondere muss weitere Ursachenforschung für die Ursache der Instabilität der ursprünglichen Implementation betrieben werden. Diese trat vornehmlich bei größeren Anzahlen von Items, bedingt durch geringe Supportwerte auf. Mögliche Anhaltspunkte sind optimierte Verfahren zur Suche von Regeln in häufigen Itemsets, die Einschränkung des Suchraumes, wie in Abschnitt 7.1.1 vorgeschlagen oder etwa die verstärkte Wiederverwendung von Objekten.

[1] Selbstverständlich kann hier jedes beliebige Test-Framework gewählt werden

- Weil im gegebenen Zeitraum keine verbesserte Implementierung der *Similarity Flooding*-Implementation zur Verfügung gestellt werden konnte, sollten entsprechende Versuche nachgeholt werden, sobald die Probleme diesbezüglich behoben worden sind. Hierbei gilt es mehrere Fragen zu klären. Wie groß ist der Anteil der erkannten Attribute, ist er ausreichend um eine Anwendung der Regeln zu ermöglichen? Auf welche Weise lässt sich die finale σ-Funktion nutzen, um Attribute zu finden, die sich auf dasselbe Item abbilden lassen? Anschließend soll eine konkrete Implementation der Schnittstelle `PolicyAdvisor` vorgenommen werden.

- Die genaue Einbindung in das PArADISE-Framework, sowie die Schnittstelle zum Benutzer wurde bisher ebenfalls außen vor gelassen. So sollte geklärt werden, wie viele Informationen über die Vorgänge der Nutzer bekommt und welche Interaktionsmöglichkeiten er hat. Interessant ist auch die Akzeptanz der hergeleiteten Werte. Sind die Ergebnisse korrekt in dem Sinne, dass der Anwender sie annimmt und auch selbst so gewählt hätte?

- Statt der manuellen Vorgabe von Standardwerten könnte versucht werden, auch diese durch Data-Mining-Verfahren herzuleiten. Die einfachste Methode wäre die Nutzung von Policies mehrerer Nutzer zur Generierung von Regeln. Der Betreuer dieser Arbeit hat diese Möglichkeit jedoch gleich zu Beginn ausgeschlossen, begründet sei dies mit der Befürchtung, dass ein Großteil der Nutzer, insbesondere in bekannten sozialen Netzwerken, nur ein geringes Interesse an Datenschutzeinstellungen haben. Stattdessen könnte jedoch ein Clustering der Nutzer anhand ihrer Datenschutzeinstellungen vorgenommen werden. So würden Benutzerprofile, beispielsweise *sicherheitsbedacht* oder *gleichgültig*, gefunden werden, für die jeweils eigene Regeln zu Standardwerten gesucht werden können. Der konkrete Anwender sollte bezüglich seiner Nutzergruppe klassifiziert werden und entsprechende Regeln Anwendung finden.

- Natürlich können auch weitere Data-Mining-Verfahren auf Anwendbarkeit überprüft werden. So konzentrierten sich die Bemühungen dieser Arbeit über lange Zeit in Richtung der Klassifikation. Attribute sollten dabei bezüglich ihrer Schema- und Werteinformationen untersucht werden und entsprechend klassifiziert werden. Auch das Schema-Mapping-Verfahren kann möglicherweise substituiert werden.

Literaturverzeichnis

[ABKS99] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *ACM Sigmod Record*, volume 28, pages 49–60. ACM, 1999.

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. *Automatic subspace clustering of high dimensional data for data mining applications*, volume 27. ACM, 1998.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[AMS+96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[AS+94] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[BBC+99] Philip A Bernstein, Thomas Bergstraesser, Jason Carlson, Shankar Pal, Paul Sanders, and David Shutt. Microsoft repository version 2 and the open information model. *Information Systems*, 24(2):71–98, 1999.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Database Theory—ICDT'99*, pages 217–235. Springer, 1999.

[Bou05] Remco R Bouckaert. Naive bayes classifiers that perform well with continuous variables. In *AI 2004: Advances in Artificial Intelligence*, pages 1089–1094. Springer, 2005.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bun09] Bundesrepublik Deutschland. Bundesdatenschutzgesetz vom 20. Dezember 1990 (BGBl. I S. 2954), neugefasst durch Bekanntmachung vom 14. Januar 2003 (BGBl. I S. 66), zuletzt geändert durch Gesetz vom 29.07.2009 (BGBl. I, S. 2254), durch Artikel 5 des Gesetzes vom 29.07.2009 (BGBl. I, S. 2355 [2384] und durch Gesetz vom 14.08.2009 (BGBl. I, S. 2814), 2009.

[Bun12] Bundesrepublik Deutschland. Grundgesetz für die Bundesrepublik Deutschland vom 20.05.1949 zuletzt geändert durch Art. 1 G v. 11.7.2012 I 1478, 2012.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[CDF+00] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Seán Slattery. Learning to construct knowledge bases from the world wide web. *Artif. Intell.*, 118(1-2):69–113, April 2000.

[CKY08] Rich Caruana, Nikos Karampatziakis, and Ainur Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 96–103. ACM, 2008.

[Duh12] Charles Duhigg. How companies learn your secrets, February 2012.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[ES00] Martin Ester and Jörg Sander. *Knowledge Discovery in Databases: Techniken und Anwendungen*. Springer, 2000.

[ES11] S. Edelkamp and S. Schroedl. *Heuristic Search: Theory and Applications*. Elsevier Science, 2011.

[FPSS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17 No 3:37–54, 1996.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[Gen48] Generalversammlung der Vereinten Nationen. Allgemeine Erklärung der Menschenrechte vom 10.12.1948, 1948.

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. 1995. *Reading, Massachusetts: Addison-W esley. ISBN 0-201-63361-2*.

[Gro14] PostgreSQL Global Development Group. Postgresql: Documentation: 9.1: cube, 2014.

[Gru13] Hannes Grunert. XSD Schema for Privacy Policy, 2013. `http://ls-dbis.de/pp4se` zuletzt abgerufen am 03.01.2015.

[Gru14] Hannes Grunert. Privacy-aware adaptive query processing in dynamic networks. In *Proceedings of the 8th Joint Workshop of the German Research Training Groups in Computer Science*, 2014.

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, pages 9–15, 1962.

[Hil12] Kashmir Hill. How target figured out a teen girl was pregnant before her father did, February 2012. `http://www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did/` zuletzt abgerufen am 12.02.2015.

[HK98] Alexander Hinneburg and Daniel A Keim. An efficient approach to clustering in large multimedia databases with noise. In *KDD*, volume 98, pages 58–65, 1998.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2001.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[HS95] Maurice Houtsma and Arun Swami. Set-oriented mining for association rules in relational databases. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 25–33. IEEE, 1995.

[JL95] George H John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[KNT00] Edwin M Knorr, Raymond T Ng, and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal—The International Journal on Very Large Data Bases*, 8(3-4):237–253, 2000.

[Koh90] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[Lan93] Land Mecklenburg-Vorpommern. Verfassung des Landes Mecklenburg-Vorpommern vom 23. Mai 1993, 1993.

[Lan02] Land Mecklenburg-Vorpommern. vom 28. März 2002 (GVOBl. M-V S. 154) letzte berücksichtigte Änderung: mehrfach geändert durch Artikel 2 des Gesetzes vom 20. Mai 2011 (GVOBl. M-V S. 277, 278), 2002.

[LC94] Wen-Syan Li and Chris Clifton. Semantic integration in heterogeneous databases using neural networks. In *VLDB*, volume 94, pages 12–15, 1994.

[LC00] Wen-Syan Li and Chris Clifton. Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data & Knowledge Engineering*, 33(1):49–84, 2000.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. California, USA, 1967.

[MGMR01] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm (extended technical report). Technical Report 2001-25, Stanford InfoLab, June 2001.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128. IEEE, 2002.

[Mic14] Microsoft Corporation. Definieren und Bereitstellen eines Cubes (SQL Server-Video), 2014. `http://technet.microsoft.com/de-de/library/cc952924%28v=sql.100%29.aspx` zuletzt abgerufen am 27.12.2014.

[Mit97] Tom M Mitchell. *Machine learning*. McGraw-Hill, 1997.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[NH02] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 14(5):1003–1016, 2002.

[Ora14] Oracle Corporation. Building olap cubes, 2014. `http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/olap/olap_cube/buildicubes.htm` zuletzt abgerufen am 27.12.2014.

[Pet09] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[Pon10] Paulraj Ponniah. *Data Warehousing Fundamentals for it Professionals*. John Wiley & Sons, Inc, Hoboken, NJ, USA, 2010.

[Ris01] Irina Rish. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *ACM SIGMOD Record*, volume 25, pages 1–12. ACM, 1996.

[sql06] *Information technology — Database languages — SQL —*, 2006. Committee Draft (CD) under Consideration.

[SSH11] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken - Implementierungstechniken (3. Aufl.)*. MITP, 2011.

[VdLPB03] Mark Van der Laan, Katherine Pollard, and Jennifer Bryan. A new partitioning around medoids algorithm. *Journal of Statistical Computation and Simulation*, 73(8):575–584, 2003.

[WF00] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques with java implementations*. Morgan Kaufmann, 2000.

[WFM05] Ian H Witten, Eibe Frank, and Stefan Mutter. Javadoc: weka.associations, 2005. `http://www.cs.waikato.ac.nz/~ml/weka/` zuletzt abgerufen am 01.04.2015.

[WYM+97] Wei Wang, Jiong Yang, Richard Muntz, et al. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, volume 97, pages 186–195, 1997.