technische universität
dortmund

Bachelorarbeit

Nutzungsmuster in Datenströmen Android-basierter Smartphones

Mohamed Asmi
17. November 2014

Gutachter:
Prof. Dr. Katharina Morik
Dipl.-Inf. Nico Piatkowski

Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS8)
Technische Universität Dortmund

Mohamed Asmi
Matrikelnummer: 131675
Studiengang: Bachelor Informatik

Thema: Nutzungsmuster in Datenströmen Android-basierter Smartphones

Eingereicht: 17. November 2014

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 17. November 2014

Mohamed Asmi

Kurzzusammenfassung

Bei der kontinuierlichen Nutzung von Smartphones werden kontinuierlich Informationen über den gegenwärtigen Systemzustand sowie den Status der Hardware und Sensorik geliefert. Die erzeugten Daten können auf natürliche Weise als Datenstrom interpretiert werden. Beispielsweise kann die Leistungsaufnahme von Smartphones durch die Aggregation des Datenverkehrs einzelner Apps verringert werden. Um zu entscheiden ob Ressourcen eingespart werden können, müssen zuvor Systemkomponenten mit einem hohen Ressourcenverbrauch identifiziert werden. In dieser Arbeit werden Android-basierte Smartphones betrachtet. Es wird untersucht, inwiefern mit Data Mining Methoden ressourcenintensive Situationen in Smartphone-Datenströmen gefunden werden können. Dabei wird die Data-Mining Methode Subgruppenentdeckung verwendet. Da die erzeugten Daten als Datenstrom betrachtet werden können und keine Ansätze für die Subgruppenentdeckung auf dem Datenstrom gibt, wird im Rahmen dieser Bachelorarbeit einen Algorithmus entwickelt, der die Subgruppenentdeckung auf dem Datenstrome ermöglicht. Der Algorithmus wird vorgestellt und implementiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ubiquitäre Systeme wie Smartphones oder Bordcomputer in Kraftfahrzeugen sind überall und zu jeder Zeit verfügbar. Sie unterliegen starken Ressourcenbeschränkungen sowie sich ändernden Nutzungsbedingungen. Daher ist es notwendig, dass sich solche Systeme an die entsprechende Nutzung anpassen und so wenig Ressourcen wie möglich verbrauchen. Die Nutzung solcher Systeme erzeugt eine Fülle von Daten, zum einen über gegenwärtig ausgeführte Applikationen (*Apps*), zum anderen über den gegenwärtigen Systemzustand. Dieser beinhaltet die aktuelle Auslastung des Hauptprozessors, des Speichers und der Netzwerkverbindungen, wie z.B. Wireless Local Area Network (WLAN) oder Universal Mobile Telecommunications System (UMTS). Darüber hinaus wird der Status diverser Zusatzhardware und -sensorik wie Beschleunigungssensoren oder Global Positioning System (GPS) erfasst. Durch die kontinuierliche Nutzung des Smartphones können die erzeugten Daten auf natürliche Weise als Datenstrom interpretiert werden. Beispielsweise kann die Leistungsaufnahme von Smartphones durch die Aggregation des Datenverkehrs einzelner Apps verringert werden oder durch einen automatischen Wechsel der Übertragungstechnik von UMTS auf WLAN. Um zu entscheiden ob Ressourcen eingespart werden können, müssen zuvor Apps mit einem hohen Ressourcenverbrauch identifiziert werden. In dieser Arbeit werden Android-basierte Smartphones betrachtet. Es wird untersucht, inwiefern mit Data Mining Methoden ressourcenintensive Apps und Situationen in Smartphone-Datenströmen gefunden werden können.
Die bekannteste Data-Mining Methode zur Erkennung von Mustern mit vorgegebenen Eigenschaften (z.B. ein hoher Energieverbrauch) ist die *Subgruppenentdeckung* [11, 12, 14]. Die Subgruppenentdeckung dient dazu, dass Regeln in den Daten gesucht werden. Die Regeln werden den Wert eines vorher spezifizierten *Zielattributs* (z.B. Energieverbrauch) mit den Werten anderer *Attribute* der Daten erklären. Durch eine Qualitätsfunktion wird die Güte der entdeckten Regeln bewertet. Nur die besten Regeln werden ausgegeben. Da das Erzeugen aller möglichen Regeln einen sehr hohen Speicher- und Rechenzeitbedarf hat, können zum einen algorithmische Techniken eingesetzt werden um den Suchraum zu

beschränken, zum anderen kann die maximale Länge der Regeln durch eine Konstante beschränkt werden.

Für das Auffinden von Subgruppen in Datenströmen sind zur Zeit keine Ansätze bekannt. Die Subgruppenentdeckung auf statischen Datensätzen arbeitet in zwei Phasen, nämlich 1) dem Auffinden der Regeln sowie 2) dem bewerten der Regeln. Da das Auffinden der Regeln in einem Datenstrom im Prinzip eine unendliche Laufzeit benötigt, können diese Phasen auf Datenströmen nicht definiert werden. Für das verwandte Problem des Auffindens häufiger Mengen auf Datenströmen existieren aktuelle Ansätze [4]. Dort werden mehrere Elemente des Datenstroms blockweise eingelesen und verarbeitet. Allerdings ist der Algorithmus nicht in der Lage, Mengen zu finden, die bezüglich eines vorgegebenen Zielattributes, wie z.B. dem Energieverbrauch, interessant sind. Daher soll im Rahmen dieser Bachelorarbeit ein Datenstromalgorithmus für die Subgruppenentdeckung entwickelt, implementiert und evaluiert werden.

Die Idee des in dieser Arbeit zu entwickelnden Algorithmus für die Subgruppenentdeckung auf Datenströmen *StreamsSD* basiert auf der Idee der statischen Subgruppenentdeckung sowie dem verlustbehafteten Zählalgorithmus *Lossy Counting* [5]. Beide der oben genanten Phasen der klassischen Subgruppenentdeckung werden simultan ausgeführt. Dazu wird *Lossy Counting* verwendet um die Häufigkeit der Kandidaten für neue Regeln zu zählen. Durch *Lossy Counting* werden seltene Kandidaten automatisch verworfen. Die Komplexität der Kandidaten ist durch die Anzahl der betrachteten Attribute bedingt. Dies induziert eine Hierarchie, bei der die Kandidaten entsprechend ihrer Komplexität in *Level* eingestuft werden. Die Kandidaten aller Levels werden nicht auf einmal erzeugt. Anfangs werden Regeln der Komplexität 1 erzeugt und nachdem eine vom Benutzer spezifizierte, oder im Algorithmus bestimmte Anzahl von Paketen verarbeitet wurde, werden Kandidaten mit einer höheren Komplexität bottom-up aus den besten Regeln der darunterliegenden Ebenen erzeugt. Alle Kandidaten, die in der Ausgabemenge von *Lossy Counting* sind, werden Regeln. Aber nur aus den k besten Regeln werden Kandidaten höherer Levels erzeugt, wobei die Güte mit der Qualitätsfunktion bestimmt wird. Somit kann die aktuelle Regelmenge jederzeit vom Benutzer angefragt werden.

Um Datenstromalgorithmen so modular zu implementieren, dass sie sowohl eigenständig in einer App als auch als Teil einer größeren Datenanalyse einsetzbar sind, kann ein *Datenstrom-Framework* verwendet werden. Solche Frameworks erlauben eine einfache Anbindung von Algorithmen an eine Vielzahl strömender Datenquellen. Bekannte Datenstrom-Frameworks wie Twitters Storm sind auf die Ausführung in großen Rechenzentren bzw. der Cloud ausgelegt und sind nicht ohne Weiteres auf aktuellen Smartphones ausführbar. Das Datenstrom-Framework *Streams* [1] erlaubt es, die Ausführungseinheit

auszutauschen. Damit ist es auf einer Vielzahl von Plattformen lauffähig. Es ist Open-Source und ermöglicht sowohl eine modulare Implementierung als auch die direkte Verarbeitung von Datenströmen und statischen Daten auf Rechenzentren, Workstations und Smartphones. Mit Streams lassen sich Datenstromverarbeitungsprozesse flexibel in einem XML-Format definieren, wodurch es sich für eine Vielzahl von Anwendungsfällen verwenden lässt. Zur Zeit ist keine Subgruppenentdeckung für das Streams-Framework verfügbar.

1.1 Ziele der Arbeit

In dieser Bachelorarbeit wird ein Datenstromalgorithmus für die Subgruppenentdeckung entwickelt. Da der Algorithmus auf Smartphones ausgeführt werden soll, werden Speicherverbrauch und Laufzeit auf Android-basierten Smartphones evaluiert. Das Problem wird formal definiert und bestehende Ansätze für die Subgruppenentdeckung auf statischen Daten werden erläutert. Der entwickelte Algorithmus wird innerhalb des Datenstrom-Frameworks Streams implementiert. Das Resultat sowie der Ressourcenverbrauch einer bestehenden, statischen Implementierung werden mit denen des neu entwickelten Algorithmus verglichen. Zur Evaluation werden sowohl Benchmarkdatensätze für die Subgruppenentdeckung als auch echte Smartphone Daten verwendet.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen vorgestellt. Dort werden die in dieser Arbeit verwendeten Algorithmen erklärt. Außerdem wird das *streams-Framework* und die *Android-Platform* kurz eingeführt. In Kapitel 3 wird der im Rahmen dieser Arbeit entwickelte Algorithmus präsentiert. in Kapitel 4 wird der Algorithmus evaluiert und die Ergebnisse der Experimente interpretiert und diskutiert.

Kapitel 2

Grundlagen

In diesem Kapitel werden die klassische Subgruppenentdeckung und der Lossy Counting Algorithmus vorgestellt. Darüber hinaus gibt es eine Beschreibung der Eigenschaften des Streams-Framework sowie für das meist verbreitete mobile Betriebssystem Android.
Die Abschnitte in diesem Kapitel basieren auf der zitierten Literatur.

2.1 Maschinellen Lernen und Data-Mining

Maschinelles Lernen ist das Endecken von Wissen aus vorhandenen Daten. Dabei werden Modelle gesucht. Das nennt man Wissensentdeckung. Die Anwendung von Data-Mining Methoden ist ein Teil von dem maschinellen Lernprozess.
Maschinen lernen aus Beispielen, die durch Erfahrungen gesammelt wurden. Für das Dntdecken des Wissens verwendet man Data-Mining Algorithmen, die in der Lage sind, Daten zu analysieren und Zusammenhänge zu ermitteln. Man kann das Maschinelle Lernen in zwei Bereiche unterteilen nämlich überwachtes und nicht überwachtes Lernen.
Bei dem überwachten Lernen werden Hypothesen gesucht, die möglichst gute Aussagen treffen sollen. Dafür werden schon vorhandene Ergebnisse von Experimenten oder gesammelten Datensätzen verwendet. Unter Hypothese versteht man eine Abbildung, die für jede Eingabe einen Ausgabewert schätzen sollen. Der Fehler der Funktion soll dabei minimiert werden. In diesem Bereich des Maschinellen Lernen kann man *die Klassifikation* [17] zuordnen. Aus einer Trainingsmenge (Menge von gesammelten Beispiele), die alle nötigen Informationen für eine Klassifikation beinhaltet, z.B die verfügbaren Klassen, werden Informationen gesammelt. Anhand dieser Informationen werden Abbildungen gebildet, die in der Lage sind Elemente zu klassifizieren d.h einer Klasse zu zuordnen. Das geschieht in dem über ein Element eine Vorhersage getroffen wird. Die Trainingsmenge oder einen Teil davon könnte als Testmenge verwendet werden. Dadurch wird die gefundene Abbildung getestet. Bekannte Klassifikationverfahren sind z.B Naive Bayes oder Entscheidungsbäume.

Bei dem nicht überwachten Lernen ist das Ziel nicht Hypothesen zu finden, die vorhersagen sollen, sondern werden hier unbekannte Muster in den Daten gesucht. *Clustering* gehört zu dem Bereich des nicht überwachten lernen. Bei *Clustering* wird ein Modell aufgefunden, bei dem Elemente aus den Daten in Cluster unterteilt werden. Dabei wird eine Abbildung gesucht, die Elemente anhand ihrer Merkmale in Gruppen (Cluster) zusammenstellt. Die Elemente in einem Cluster haben ähnliche Merkmale. Hier werden die Beziehungen von allen Elementen der Daten zueinander in Betracht genommen. Es gibt verschieden Clustering Verfahren darunter ist z.B k-means.
Eine ausführliche Einführung in das Maschinelle Lernen, findet man in [9].
In dieser Arbeit wird eine Data-Mining Methode benötigt, die Muster in den Daten sucht bezüglich eines bestimmten Attribute. Deshalb wird die *Subgruppenentdeckung* verwendet.

2.2 Subgruppen Entdeckung

Die bekannteste Methode zur Erkennung von Mustern mit vorgegebenen Eigenschaften ist die *Subgruppenentdeckung*. Zum ersten Mal wurde sie von Kloesgen und Wrobel [11, 12] eingeführt. Die Subgruppenentdeckung [14] liegt zwischen den zwei Bereichen des maschinellen Lernen, da bei der Subgruppenentdeckung vorhergesagt werden soll um eine Beschreibung der Daten zu liefern. Andere Data-Mining Methoden zur Erkennung von Mustern sind in [3] zu finden.

Definition der Subgruppenentdeckung

Sei $\mathcal{D}$ ein Datensatz, der aus Datenitems $\vec{d_i}$ besteht. Ein Datenitem $\vec{d_i} = (\vec{a}, t)$ ist ein Paar aus Attribute $\{a_1, a_2, ..., a_m, t\}$, die mit $\vec{a}$ bezeichnet wird, und einem Zielattribut. In dieser Arbeit werden die Begriffe Datenitem und Transaktion die gleiche Bedeutung haben. Das *Zielattribut* definiert die eingegeben Eingenschaft, für die die Daten erklärt werden sollen. Das Zielattribut muss binär sein. Jedoch hat jedes Attribut a_m einen Wert aus einem $dom(\mathcal{A})$. Die Werte der Attribute können binär, nominal oder numerisch sein. Daher sind die Domäne $dom(\mathcal{A}_m) = \{0, 1\}$, $|dom(\mathcal{A}_m)| \in \mathbb{N}_0$ oder $dom(\mathcal{A}_m) = \mathbb{R}$. $\vec{d_i}$ wird das i^{te} Datenitem genannt. Die Menge $dom(\mathcal{A})$ enthält die Domäne der einzelnen Attribute. Außerdem bezeichnen $\vec{a^i}$ und t^i der i^{te} Vektor der Attribute und das i^{te} Zielattribut. Die Größe der Datenmenge wird mit $N = |\mathcal{D}|$ bezeichnet.
Betrachtet wird der in 2.1 gegebene Beispieldatensatz. Ein Datenitem $\vec{d_i}$ ist eine Zeile aus diesem Datensatz z.B d_3 bezeichnet die dritte Zeile in der Tabelle. Die Menge der Attribute in diesem Beispieldatensatz ist {Alter,Geschlecht,Auto,Haus}. Das Zielattribut ist Einkommen. Es definiert die Eigenschaft ob man hohen Einkommen (Einkommen=1) oder niedrigen Einkommen (Einkommen=0). Ein Domain für die Werte des Attributes z,B Geschlecht wäre $dom(\mathcal{A}_{Auto}) = \{$männlich, weiblich $\}$.

Item Nr.	Alter	Geschlecht	Auto	Haus	Einkommen
1	25	männlich	nein	nein	0
2	28	weiblich	nein	nein	0
3	32	weiblich	ja	nein	1
4	34	weiblich	ja	nein	1
5	40	männlich	ja	nein	1
6	44	weiblich	nein	ja	1
7	47	männlich	nein	ja	0
8	50	weiblich	ja	ja	0
9	55	männlich	ja	ja	1
10	60	weiblich	nein	nein	1

Tabelle 2.1: Beispieldatensatz

Nun benötigt man die Definition einer Regel um eine Subgruppe definieren zu können. Eine Regel ist eine Funktion $p : P(\mathcal{A}) \times dom(\mathcal{A})) \rightarrow \{0, 1\}$, wobei $P(\mathcal{A})$ die Potenzmenge der Attribute darstellt. Mit $\mathcal{P}$ bezeichnet man die Menge aller Regeln. Man sagt eine Regel p überdeckt einen Datenitem $\vec{d^i}$ genau dann, wenn $p(\vec{a^i}) = 1$ ist. Die Attribute werden mit einander konkateniert um $\vec{a}$ zu konstruieren. Eine Regel hat die Form:

$$\text{Bedingung} \rightarrow \text{Wert der Regel.}$$

Die Bedingung einer Regel ist die Konkatenation von Paaren (Attribut,Wert). Der Wert der Regel wird das Zielattribut darstellen.

Definition (Subgruppe) Eine *Subgruppe* G_p ist die Menge aller Datenitems, die von der Regel p überdeckt werden.

$$G_p = \{\vec{d_i} \in \mathcal{D} | p(\vec{a}^i) = 1\}$$

Das Komplement einer Subgruppe G ist $\bar{G}$ und enthält alle $\vec{d_i} \notin G$ d.h alle Datenitems

die von p nicht überdeckt werden. Mit n und $\bar{n}$ wird die Anzahl der Elemente in G und $\bar{G}$ gekennzeichnet, wobei $n = N - \bar{n}$.

Wenn man eine Subgruppenentdeckung über den Datensatz 2.1, wäre für die gegebene Eigenschaft Einkommen interessant zu wissen, z.B ob die Männer oder die Frauen ein höheres Einkommen haben oder ob Frauen, die ein hohes Einkommen haben, ein Haus besitzen oder nicht.

Die Subgruppenentdeckung arbeitet in zwei Phasen, nämlich dem Auffinden der Kandidaten der Regeln sowie dem Bewerten der Regeln. Es werden zuerst Regeln mit einer kleineren Komplexität (allgemeine Regeln) aufgefunden z.B $R1$:$Geschlecht = Frau \rightarrow Einkommen = 1$, von denen im Laufe des Subgruppenentdeckung Prozesses immer komplexere (konkretere) Regeln generiert werden z.B $R2$:$Geschlecht = Frau \wedge Haus = Ja \rightarrow$

Einkommen = 1. Die Komplexität der Regeln ist durch die Anzahl der betrachteten Attribute bedingt. Die Regel $R1$ hat die Komplexität 1 und $R2$ hat die Komplexität 2.
Zuerst werden Kandidaten mit der Komplexität 1 aufgefunden. Danach werden Kandidaten mit höher Komplexität bottom-up generiert. Mit Hilfe einer *Qualitätsfunktion* werden die Regeln bewertet.
Die Überdeckung der beiden Regeln $R1$ und $R2$ wird in Tabelle 2.2 veranschaulicht. Die in grün markierten Datenitems beschreiben die Subgruppe, die von $R1$ überdeckt sind, und die in blau markierten Datenitems beschreiben die Subgruppe, die von $R2$ überdeckt sind.

Item Nr.	Alter	Geschlecht	Auto	Haus	Einkommen
1	25	männlich	nein	nein	0
2	28	weiblich	nein	nein	0
3	32	weiblich	ja	nein	1
4	34	weiblich	ja	nein	1
5	40	männlich	ja	nein	1
6	44	weiblich	nein	ja	1
7	47	männlich	nein	ja	0
8	50	weiblich	ja	ja	0
9	55	männlich	ja	ja	1
10	60	weiblich	nein	nein	1
Item Nr.	Alter	Geschlecht	Auto	Haus	Einkommen
1	25	männlich	nein	nein	0
2	28	weiblich	nein	nein	0
3	32	weiblich	ja	nein	1
4	34	weiblich	ja	nein	1
5	40	männlich	ja	nein	1
6	44	weiblich	nein	ja	1
7	47	männlich	nein	ja	0
8	50	weiblich	ja	ja	0
9	55	männlich	ja	ja	1
10	60	weiblich	nein	nein	1

Tabelle 2.2: Subgruppen für R1 und R2

Qualitätsfunktion

Die *Qualitätsfunktion* [10, 13] spielt eine wichtige Rolle bei der Subgruppenentdeckung. Sie bestimmt die Güte der Regeln. Damit kann man die besten Regeln ausgeben.

Definition (Qualitätsfunktion) Eine *Qualitätsfunktion* ist ist eine Funktion φ: $\mathcal{P} \to \mathbb{R}$, die jeder Regel einen Wert (die Güte) zuweist.

Man kann die Auswahl der besten Regeln nach verschiedenen Kriterien treffen. Entweder werden die Regeln nach ihrer Güte sortiert und dann die besten k Regeln ausgegeben oder wird die Ausgabe durch einen minimalen Wert der Qualitätsfunktion beschränkt. Außerdem kann man eine minimale Menge von Regeln mit eine maximale Qualität suchen. Diese Verfahren für die Auswahl der besten Regeln wird in diese Arbeit nicht betrachtet. Es gibt viele *Qualitätsfunktionen* und es ist immer schwer zu sagen welche wichtig sind. Die Wahl der *Qualitätsfunktionen* wird von den Datenanalytikern getroffen. Entscheiden ist die aktuelle Aufgabe. In diesem Abschnitt wird eine Auswahl von Qualitätsfunktionen präsentiert.

- Coverage: liefert den Prozentanteil der Elemente der Datenmenge, die von einer Regel überdeckt sind.
  $$Cov(R) = \frac{TP+FP}{N}$$
  mit TP bezeichnet man, wie oft war eine Regel wahr war und sie richtig vorhergesagt wurde. Dagegen gibt FP eine Aussage über, wie oft war eine Regel wahr war aber sie falsch vorhergesagt wurde.

- Precision: liefert der Anteil der tatsächlichen richtig vorhergesagten Regeln, wenn die Regel wahr war.
  $$Pr(R) = \frac{TP}{FP+TP}$$

- Recall: liefert der Anteil alle wahren Regeln, die richtig vorhergesagt wurden, von allen wahr vorhergesagten Regeln.
  $$Re(R) = \frac{TP}{TP+FN}$$
  wobei FN die Anzahl der falschen Regeln ist, die falsch vorhergesagt wurden.

- Accuracy: liefert der Anteil der richtigen vorhergesagten Regeln von allen Regeln
  $$Acc(R) = \frac{TP+TN}{N}$$

- Weigthed Relative Accuracy(WRAcc) [20]: Diese Gütefunktion gibt eine Aussage über die Ausgewogenheit zwischen der Überdeckung und der Genauigkeit einer Regel. WRacc ist die am meisten verwendete Qualitätsfunktion bei der Subgruppenentdeckung.

$$WRAcc(R) = Cov(R)\left(\frac{TP+FN}{N} - \frac{TP+TN}{N}\right)$$

mit TN ist die Anzahl der falschen Regeln, die richtig vorhergesagt wurden. Er wird verwendet, da die einzelne Betrachtung von Accuracy zu falschen Schlüssen führen könnte.

- F1-Score [18]: das harmonische Mittel von Precision und Recall.

$$Fsr(R) = \frac{2*Pr(R)*Re(R)}{Pr(R)+Re(R)}$$

Suchstrategien

Die Anzahl der aufgefundenen Kandidaten bei der Subgruppenentdeckung kann exponentiell wachsen. Das kann beim Generieren der Regeln mit hoher Komplexität einen sehr hohen Speicher- und Rechenbedarf benötigen. Deshalb können algorithmische Techniken eingesetzt werden, die den Suchraum verkleinern. Hierbei kann eine heuristische Suche durchgeführt werden z.B Beam-search [23]. Darüber hinaus kann man zwei Parameter einstellen um den Suchraum zu beschränken. Die maximale Komplexität einer Regel kann beschränkt werden. Weiterhin kann man nur bestimmte Kandidaten betrachten z.B die von einer Qualitätsfunktion besten bewerteten Regeln.

Ähnliche algorithmische Probleme

Es gibt verschiedene Data-Mining Methoden zum Auffinden von Mustern. Jedoch haben diese Methoden verschiedene Zielergebnisse. So soll z.B eine Methode Muster mit der besten Qualität, die die Daten beschreiben, entdecken wohingegen andere versuchen, eine Menge von Mustern zu finden, die die Daten am besten komprimieren.
Ähnlich zu der Subgruppenentdeckung ist das KRIMP Problem. Der *KRIMP*-Algorithmus ist ein guter Einsatz für die Beschränkung der Anzahl der aufgefundenen Mustern. Der *KRIMP*-Algorithmus [22] dient dazu, dass eine minimale Anzahl von Mustern aufgefunden werden, die die Daten am besten komprimieren. Er basiert auf dem MDL (Minimum Description Length) [7] Prinzip. Als Datenstruktur verwendet *KRIMP* eine Codetabelle. Codetabellen bestehen aus zwei Spalten. In der ersten Spalte stehen Attributen oder Mengen von Attribute und in der zweiten Spalte die dazugehörige Codierung. Am Anfang sind in der Tabelle nur Einzelattribute. Danach kommen Kandidaten aus den Daten, die mehr Attribute enthalten. Wenn ein Kandidat hinzugefügt wird, wird immer überprüft, ob die Codetabelle eine bessere Codierung liefert. Wenn dies nicht der Fall ist, wird der Kandidat nicht in die Codetabelle eingefügt.
Für diesen Algorithmus wurde ein Datenstromalgorithmus entwickelt. Er ist der *STREAMKRIMP* [21]. Er entdeckt die Änderung der Verteilungen in einem Datenstrom. *STREAMKRIMP* basiert auch auf dem MDL Prinzip und verwendet auch Codetabellen. Im Gegensatz zu der Codetabelle von *Krimp* enthält hier jede Zeile den Zeitpunkt, zu der sie hinzugefügt wurde. Der Datenstrom wird in kleinere endliche Sequenzen unterteilt. Auf

jeder Menge wird *KRIMP* eingesetzt um eine Codetabelle für diese Sequenz des Datenstromes zu erstellen. Am Anfang wird die Codetabelle der ersten Sequenz erstellt und wird als die aktuelle Codetabelle verwendet. Danach werden Codetabellen für die anderen Sequenzen ermittelt. Wenn eine neue Codetabelle berechnet wird, wird überprüft, sie den Datenstrom besser kodiert. Wenn dies der Fall ist, dann ist eine Änderung in der Verteilung entdeckt worden und die neue Codetabelle wird als aktuelle Codetabelle verwendet.

2.3 Der Lossy Counting Algorithmus

Bei der Subgruppenentdeckung spielt das Zählen der Häufigkeit eine wichtige Rolle. In dieser Arbeit wird einen verlustbehafteten Zählalgorithmus verwendet, der nur die häufigsten Elemente aufzählt und zurückgibt. Der *Lossy Counting (LC)* löst approximativ das Problem der häufigsten Mengen auf einem Datenstrom [5].

Grundidee der Lossy Counting Algorithmus

Ein Datenstrom ist eine Sequenz von Elementen $(e_1, e_2, e_3..)$. N ist die Länge des Datenstromes. Zusätzlich zu einem Datenstrom bekommt der Algorithmus zwei Parameter als Eingabe, nämlich ein Support Parameter $s \in [0, 1]$ und ein Fehlerparameter ϵ. LC findet alle Elemente in einem Datenstrom, deren Häufigkeit größer als sN ist. Mit dem Fehlerparameter ϵ garantiert der Algorithmus, dass alle ausgegebenen Elemente mindesten die Häufigkeit $(s - \epsilon)N$ haben. Angenommen, es wäre von Interesse alle Elemente deren Häufigkeit größer als 0.1% der Länge des Datenstroms auszugeben. Dann ist s=0,01 und die zulässige Fehlertoleranz wird durch die Eingabe des Fehlerparameters ϵ eingestellt. Der Fehlerparameter ϵ kann z.B ein Zehntel kleiner als s sein. Wenn $\epsilon = 0,001$ ist, dann stehen in der Ausgabemenge des Algorithmus garantiert alle Elemente deren Häufigkeit mindestens $0,009N$ ist.
Der eingehende Datenstrom wird in Puffern der festen Länge $w = \lceil 1/\epsilon \rceil$ unterteilt. Die Puffer werden mit einem Index i markiert. Immer, wenn ein Puffer voll ist, wird der Index erhöht. $b_{current}\lceil N/w \rceil$ bezeichnet den aktuellen Index. Mit f_e wird die konkrete Häufigkeit eines Elements e bezeichnet.
Die Lossy Counting Datenstruktur ist eine Menge $\mathcal{D}$ von Zählern der Form (e, f, Δ). Mit f bezeichnet man die von LC berechnete approximative Häufigkeit. Ferner ist Δ die maximale Fehlergrenze bei der Ermittlung von f.

Beschreibung des Lossy Counting Algorithmus

Am Anfang ist $\mathcal{D}$ leer. Sobald ein Element e aus dem Datenstrom kommt, wird es überprüft ob es in $\mathcal{D}$ existiert. Wenn e schon in $\mathcal{D}$ ist, wird f erhöht. Sonst wird ein Zähler

$(e, 1b_{[}current0 - 1)$ erzeugt und hinzugefügt. Wenn ein Puffer voll ist d.h $0 \equiv$ N mod w wird $\mathcal{D}$ bereinigt. Alle Elemente, die die Bedingung $f + \Delta \leq b_{current}$ nicht erfüllen, werden gelöscht. Ausgegeben werden alle Elemente, deren approximative -Häufigkeit f kleiner gleich $(s - \epsilon)N$ ist. Der Pseudocode des Algorithmus ist in 2.3.1 zu finden.

Der Fehler Δ sagt wie oft ein Element e in den ersten $b_{current} - 1$ Puffer maximal vorge-

Algorithmus 2.3.1 Lossy Counting

Input: Datensrom von Elementen e

Output: Elemente mit $f \geq (s - \epsilon)N$

```
1:  D =
2:  N = 0
3:  b_current = 1
4:  while Datenstromnichtbeendet do
5:    e = nächste Element vom Datenstrom
6:    N++
7:    if e ∈ D then
8:      f++ [
9:    else
10:     insert(e, 1, b_current − 1)intoD ]
11:   end if
12:   if Nmodw == w then
13:     for all elementinD do
14:       if f + Δ ≤ b_current then
15:         delete (e, f, Δ from D
16:       end if
17:     end for
18:   end if
19: end while
```

kommen sein könnte. Wir haben N ist die Länge des Datenstromes und $w = \lceil 1/\epsilon \rceil$ ist der Größe eines Puffers. Deshalb ist $\Delta = b_{current} - 1$.

Für die Berechnung der am häufigsten vorkommenden Elementen braucht LC höchstens $\frac{1}{\epsilon} \log(\epsilon N)$ Zähler.

Beweis: Sei $B = b_{current}$ der aktuelle Pufferindex. Für alle $i \in [1, B]$, d_i ist die Anzahl der Zähler in $\mathcal{D}$ für die $b - i + 1$ Puffer. Ein Element im Puffer $B - i + 1$ muss mindestens $i - mal$ vorgekommen sein. Sonst wurde es von LC gelöscht. Das kann man mit Hilfe der Größe w eines Puffers in der folgenden Formel ausdrucken:

$$\sum_{i=1}^{j} id_i \leqslant jw \forall j = 1, 2, ..., B \tag{2.1}$$

Daraus Folgt:

$$\sum_{i=1}^{j} d_i \leqslant \sum_{i=1}^{j} \frac{w}{i} \forall j = 1, 2, ..., B \tag{2.2}$$

Mit Induktion wird 2.2 gezeigt. Für $j = 1$ gilt aus 2.1. Angenommen wird, dass 2.2 für alle $j = 1, 2, 3..., p-1$ gilt. Zu zeigen ist, dass 2.2 für $j = p$ auch gilt.
Aus der Induktionsvoraussetzung gilt 2.2 für alle $j = 1, 2, 3..., p-1$. Für $p-1$ Termen der 2.2 wird einmal für $j = p$ 2.1 addiert. Das ergibt 2.3.

$$\sum_{i=1}^{p} id_i + \sum_{i=1}^{1} d_i + \sum_{i=1}^{2} d_i + ... + \sum_{i=1}^{p-1} d_i \leqslant pw + \sum_{i=1}^{1} \frac{w}{i} + \sum_{i=1}^{2} \frac{w}{i} + ... + \sum_{i=1}^{p-1} \frac{w}{i} \tag{2.3}$$

Das kann man in 2.4 zusammenfassen.

$$p \sum_{i=1}^{p} d_i \leqslant pw + \sum_{i=1}^{p-1} \frac{(p-1)w}{i} \tag{2.4}$$

Wenn man beide Terme der Ungleichung durch p teilt, bekommt man die Gleichung in 2.2 für j=p. Insofern ist 2.2 für $j = p$ gültig
Da $|\mathcal{D}| = \sum_{i=1}^{B} d_i$ ist, folgt aus 2.2 dass, $|\mathcal{D}| \leqslant \sum_{i=1}^{B} \frac{w}{i} \leqslant \frac{1}{\epsilon} \log(B) = \frac{1}{\epsilon} \log(\epsilon N)$ ist. Somit wächst die Anzahl der gebrauchten Zählern logarithmisch.

2.4 Das Datenstrom-Framework Streams

Um eine Datenanalyse auf Datenströmen durchführen zu können, benötigt man ein Datenstrom-Framewok. In dieser Arbeit wird das *Streams-Framework* [1] vorgestellt. *streams* ist aus einer Projekt-Gruppe am LS8 der Fakultät für Informatik der TU Dortmund entstanden. Das Ziel des Frameworks ist das Erleichtern der Modellierung der Datenstromverarbeitung. *streams* erlaubt es, Ausführungseinheit auszutauschen. Damit ist es auf Vielzahl von Plattformen lauffähig. Darüber hinaus ist es möglich andere Bibliotheken, wie z.B das RapidMiner Tool, in das Framework einzubinden. Weiterhin ist das *streams-Framework* in der Lage, verschiedene Prozesse auf Datenströmen unterschiedlichen Quellen auszuführen. *streams* stellt auch ein API bereit, das die Erweiterung der strams-Bibliothek ermöglicht. Mit Hilfe der schon vorhandenen und dazu entwickelten Prozessoren kann man kontinuierliche Prozesse modellieren. Das geschieht einfach durch die Definition eines Flow-Graphes in einer XML-Datei. Dafür stellt *strems* vier wichtige Elemente bereit.

Datenstrom

Der Datenstrom spielt eine wichtige Rolle bei der Datenstromverarbeitung. Er ist die Datenquelle.
Ein Datenstrom ist eine Sequenz von Datenitems. Ein Datenitem ist eine Menge von Paaren der Form (k, v). Der Schlüssel k ist von Typ String und bezeichnet des Attribute. v ist

der Wert des Attributes und hat einen Typ, der das Java-Interface `Serializable` implementierte.
streams stellt schon verschiedene Datenströme Implementierungen z.B aus einem CSV-Datei oder SQL-Datenbanken bereit. Aber auch eine eigene Implementierung ist durch das implementieren der abstrakten Klasse `AbstractStream` möglich. *stream* beschreibt einen Datenstrom im *streams-Framework*.

Prozesse und Prozessoren

Bei einer Datenstromanalyse müssen die Daten verarbeitet werden. Dafür sind die Prozesse zuständig. In *streams* ist ein Prozess das aktive Element bei der Datenverarbeitung. Er wird nicht nur wie ein Datenstrom gelesen, sondern führt er auch eine Menge von Prozessoren auf dem Datenstrom aus.
Ein Prozessor ist das kleinste Element in dem *streams-Framework*. Jedoch spielt er die wichtige Rolle bei der Datenstromverarbeitung. Nämlich beinhaltet er Funktionen, die auf dem Datenstrom durchgeführt werden sollen, um den zu extrahieren oder manipulieren. Ein Prozess kann aus verschiedenen Prozessoren bestehen.
In *streams* sind schon einige Prozessoren implementiert, wie z.B für die Ausgabe des Datenstroms auf dem Bildschirm. Außerdem man ist in der Lage seine die *streams-Bibliothek* mit eigenen Prozessoren zu erweitern, die die gewünschten Verarbeitung durchzuführen. Das ermöglicht die Implementierung von Datenstromalgorithmen. Dafür ist es notwendig das Interfaces `streams.prozessor` zu implementieren.

Services

Um leichte die durch die Prozessoren entstehenden Funktionalitäten überall und jede Zeit zu verwenden, kann man Services benutzen. Sie können z.B eine Klassifikationverfahren darstellen, das auf einer Datenbank durchgeführt wird.
In *streams* sind Services durch Java Interfaces definiert. Durch Implementierung des Interfaces `stream.service.service` wird ein Service hergestellt. Um innerhalb einem Container Services benutzbar zu machen, soll die Klasse, die dieses Service implementiert, mit einer *id* versehen werden.

Container

Der Container ist das Hauptelement für die Modellierung von Datenstromverarbeitung Prozesse. Der Container beinhaltet alle für eine Datenstromverarbeitung benötigten Elemente. Die Quelle des Datenstroms wird deklariert sowie die zu verwendeten Prozesse. Außerdem werden die Services definiert, die man benutzen will. In 2.1 ist eine XML-Datei, die die Struktur eines Containers veranschaulicht.
Hier wird einen CSV-Datenstrom aus der Datei `data.csv` erzeugt. Dieser Datenstrom hat

```
<container>
    <stream id="in" class="stream.io.CsvStream"  url="file:/Users/data.csv" />

    <process input="in">

         <PrintData />

    </process>
</container>
```

Abbildung 2.1: Container als XML-Datei

den Id=in und wird als für den Prozessor übergeben. `PrintData` ist ein vom *streams* zu Verfügung gestellter Prozessor, der die Daten aus dem Datenstrom auf dem Bildschirm zurückgibt.
Bei der Ausführung des Containers werden alle in dem Container definierten Elemente aufgebaut. Datenströme, Services und Prozesse werden erzeugt und die Prozessoren werden auf die Datenströme ausgeführt.

2.5 Android-Plattform

Android ist ein Betriebssystem für Mobile Geräte. Da der in dieser Arbeit zu entwickelnde Algorithmus auf Android Geräten lauffähig sein soll, wird die Plattform kurz vorgestellt. *Android* [6] ist eine Open-source Plattform geeignet für mobile Geräte. Sie stellt alle benötigten Werkzeugen für die Entwicklung einer App zur Verfügung. *Android* wurde im Jahr 2005 von Google gekauft. Bis zu diesem Jahr wurden verschiedene Versionen veröffentlicht. Die aktuellste Version ist *Android 4.4*.

Android Aufbau und Entwicklung

Das für mobile Geräte entwickelte Betriebssystem ist in vier Schichten aufgebaut. 2.2 veranschaulicht die Architektur von *Android*.
Die*Android-Plattform* basiert auf einem Linux-Kernel. Auf diesem Grund ist *Android* auf verschiedenen Geräten mit unterschiedlichen Architekturen lauffähig. Außerdem profitiert *Android* von dem Eigenschaften der Linux Systeme z.B der Sicherheit.
Die nächste Schicht Bibliothek enthält alle Komponenten die für das Ausführen und Kompilieren einer Applikation notwendig sind. Ein Teil davon ist *DALVIK*. Sie ist die virtuelle Maschine die für *Android* entwickelt wurde.
Die Applikation-Framework-Schicht stellt alle Bibliotheken und Services, die bei der Entwicklung einer App hilfreich sind, bereit. Das ist die am bestens dokumentierte Schicht,

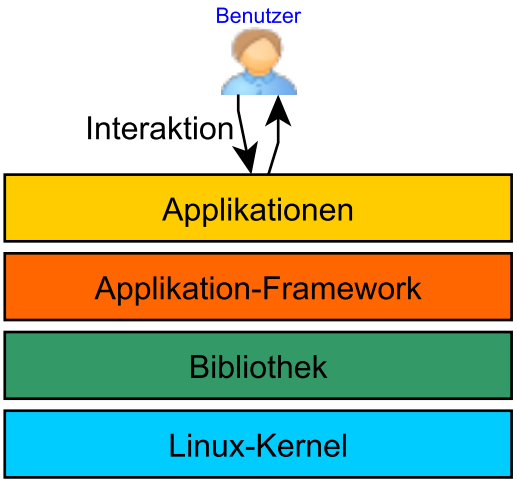


Abbildung 2.2: Schichten von der Android-Platform

da sie der Baustein für die Entwicklung einer gut aufgebauten und effizienten App ist.

Die Applikation-Schicht ist für die Interaktion zwischen den Benutzern und dem Gerät zuständig ist. Sie ist dafür, dass die die App auf dem Gerät installiert und lauffähig sind. Sie beinhaltet alle Komponenten, die für den Benutzer für die Bedienung des Gerätes notwendig sind.

Für die Entwicklung von *Android*-App benötigt man den *Android* SDK. Die Hauptprogrammiersprache ist Java. Deshalb kann man das *streams-Framework* in einer *Android*-App einbinden.
Eine Hauptrolle bei der Programmierung spielt die Klasse Activity. Sie ist dafür da, den gesamten Programmablauf zu organisieren und steuern. In der XML-Datei Manifest werden alle Informationen über die App vorgestellt z.B welche Activity benutzt werden darf, welche Android Version verwendet wird oder welche Zugriffsrechte diese App hat. Ein ganz wichtige Punkt ist, dass *Android* bei der App Entwicklung für eine Trennung zwischen den Ressourcen und dem Quellcode sorgt. Außerdem kann man das Muster Modell-View-Controller(MVC) bei der Implementierung einer App gut erkennen, da *Android* sorgt für eine Trennung zwischen den Daten, die Implementierung der Funktionalitäten und das graphische Benutzer Schnitstelle sorgt.
Mit Hilfe der Entwicklungsumgebung Eclipse kann man zu einem *Android* Projekt Maven-Abhängigkeiten einbinden. Daher kann man auch das streams-Framework auch unter *Android* verwenden.

Android und Maschinelles Lernen

Für eine Verbesserung eines Systems sollte man das System analysieren und schlussfolgern. Das Anwenden von maschinellen Lernverfahren auf einem *Android-System* kann hilfreich sein für das Untersuchen von bestimmten Eigenschaften z.B Energieverbrauch.
Zu Zeit ist es nicht bekannt, ob auf *Android* eine APP, die ein maschinelles Lernverfahren implementiert um Wissen zu sammeln über bestimmten Eingenschaften des Systems. Unter *Android* sind zwei App nämlich *Power Tutor* und *Battery* die den Akkuverbrauch von einzelnen Apps untersuchen. Sie basieren auf linearen Modellen, bei denen viele Annahmen bei der Wahl der Parameter getroffen sind. Jedoch hat Apple dabei schon eine App entwickelt für die Untersuchung des Akkuverbrauches anderer App, die auf maschinellen Lernverfahren basiert.

Kapitel 3

Subgruppenentdeckung auf Datenströmen

Im Rahmen dieser Arbeit wird eine Datenstromalgorithmus für die Subgruppenentdeckung entwickelt und implementiert. In diesem Kapitel wird der entwickelte Algorithmus erläutert. Außerdem wird eine Implementierung des Algorithmus vorgestellt.

3.1 Problemstellung

Für das Auffinden von Subgruppen in Datenströmen sind zur Zeit keine Ansätze bekannt. Für das verwandte Problem des Auffindens häufiger Mengen auf Datenströmen existieren aktuelle Ansätze [4]. Dort werden mehrere Elemente des Datenstroms blockweise eingelesen und verarbeitet. Allerdings ist der Algorithmus nicht in der Lage, Mengen zu finden, die bezüglich einer vorgegebenen Eigenschaft interessant sind. Daher wurde im Rahmen dieser Bachelorarbeit ein Datenstromalgorithmus für die Subgruppenentdeckung entwickelt(*streamSD*).
Gegeben sei einen Datenstrom. Es sollen Subgruppen aufgefunden so dass, die Qualität aller Subgruppen maximiert wird. Dabei wird die Qualitätsfunktion verwendet um die Güte der Subgruppe zu ermitteln.
Die Subgruppen werden nach ihrer Güte sortiert und die beste k Subgruppen werden ausgegeben. Außerdem man kann ein anderes Kriterium anwenden für die Ermittlung der besten Subgruppen. Dazu werden alle Subgruppen ausgegeben ausgegeben deren Qualität größer als ein bestimmter vorgegebener Funktionswert ist.

3.2 Grundidee des Algorithmus

Der *streamSD* Algorithmus basiert auf der Idee der statischen Subgruppen Entdeckung. Er verwendet auch den Zählalgorithmus Lossy Counting für die Ermittlung der Häufigkeit

der Elemente.
Die grundsätzliche Aufgabe des *streamSD* ist die Ausführung einer Subgruppenentdeckung auf einem Datenstrom. Dabei müssen, wie beim statischen Einsatz, Regeln aufgefunden und evaluiert werden. Es ist nicht möglich die zwei Phasen zu trennen, da man nicht bis zum Ende eines Datenstroms warten soll um Regeln zu generieren und evaluieren. Das wäre auf einem unendlichen Datenstrom nicht möglich. Außerdem kann das Speichern aller Elemente eines Datenstromes um am Ende Regeln aufzufinden sehr aufwendig sein. Insofern werden vom *streamSD* entlang der Datenstromverarbeitung Kandidaten für die Regeln erzeugt und Regeln generiert.
Ein Datenstrom ist eine Sequenz von Datenitems. Jeder Datenitem ist eine Menge von Attributen und ihrer Werte.Die Kandidaten sind nicht evaluierte Regeln. Die Anzahl der betrachteten Attribute bestimmt die Komplexität eines Kandidaten. Diese induziert eine Hierarchie, bei der die Kandidaten entsprechend ihrer Komplexität in *Level* eingestuft werden. Nicht alle Kandidaten für alle Levels werden gleichzeitig erzeugt, sondern das geschieht schrittweise. Die Kandidaten für das Level 1 werden direkt aus dem Datenstrom erzeugt. Nachdem eine bestimmte Anzahl von Datenitems verarbeitet wurde, werden Kandidaten für die höheren Levels bottom-up generiert.
Die Häufigkeit der generierten Kandidaten muss ermittelt werden. Dafür wird der *LC* Algorithmus verwendet. Der verlustbehaftete Zählalgorithmus *LC* wurde in Kapitel 2 erklärt. Er ermittelt die häufigsten Elemente in einem Datenstrom und gibt die am Häufigsten vorkommenden Elemente zurück. Dabei wird eine erwünschte untere Schranke der Häufigkeit der ausgegebenen Elementen durch einen vom Benutzer eingegebenen Parametern berücksichtigt. Darüber hinaus kann man den Berechnungsfehler von *LC* steuern. Aus selten vorkommenden Kandidaten können keine gute Regeln erzeugt werden. Deshalb wird in diesem Algorithmus der *LC* verwendet um die Häufigkeit der Kandidaten zu ermitteln. Regeln, die seltene Elemente beschreiben, können keine richtige Aussage über die gesamte Datenmenge liefern. In einem Level wird für die Auswahl der Kandidaten, die Regeln werden sollen, im *streamSD* keine Suchstrategie benötigt. Alle Elemente, die in der Ausgabemenge des *LC* sind, werden evaluiert.
Für die Evaluation der Regeln werden die Qualitätsfunktionen benötigt. Sie ermitteln die Güte der Regeln. Einige wichtige Qualitätsfunktionen sind schon in Kapitel 2 beschrieben worden. Mit einer vom Benutzer vorgegebenen Qualitätsfunktion (auch Gütefunktion genannt) werden die Regeln evaluiert. Die Regeln werden nicht nach jedem Datenitem evaluiert sondern nur bei der Erzeugung von Kandidaten höherer Levels, also immer nachdem eine bestimmte Anzahl von Datenitems verarbeitet wurde.
Die Kandidaten für die Regeln für das Level $n+1$ werden aus den k besten Regeln des Levels n und aus den k besten Regeln des Levels 1 erzeugt. Nur aus den wichtigen Regeln können wichtigere Regeln erstellt werden. Die maximale Komplexität der Kandidaten kann mit einem vom Benutzer eingegebenen Parameter beschränkt werden.

Der Algorithmus ist in der Lage, zu jedem Zeitpunkt die besten Regeln, nach den oben schon vorgestellten Kriterien, auszugeben.

3.3 Beschreibung des Algorithmus

In diesem Abschnitt wird der entwickelte Datenstromalgorithmus für die Subgruppenentdeckung (*streamSD*) genauer beschrieben. Der Algorithmus braucht eine Datenstromquelle als Eingabe. Weiterhin soll der Benutzer die Zeitpunkte, in den Kandidaten höherer Levels erzeugt werden, mit einem Zeitparameter t definieren. Außerdem werden die LC Parameter Fehler- und Supportparameter eingestellt. Darüber hinaus wird noch ein Parameter k für die Anzahl zu verfeinerten Regeln (Regeln aus denen komplexerer Regeln erzeugt werden soll) übergeben.
Man kann die Arbeitweise des Algorithmus in drei großen Phasen unterteilen nämlich 1) die Initialisierungsphase 2) die Generierung und Bewertung der Regeln 3) das Update der Kandidaten. In der Initialisierungsphase werden sowohl alle Parametern gesetzt als auch die ersten Kandidaten für Level 1 aus dem Datenstrom generiert. In der zweiten Phase werden die Regeln generiert und evaluiert. Außerdem könnten nach dieser Phase die besten Regeln ausgegeben werden. Bei der Update Phase werden sowohl neue Kandidaten für alle Levels generiert als auch die Häufigkeit der existierenden Kandidaten hochgezählt. Da für das Zählen der Kandidaten LC verwendet wird, werden die nicht häufigen Kandidaten gelöscht.

Initialisierung

Bei der Initialisierung werden sämtliche eingegebene Parameter gesetzt. Der Benutzer definiert das Zielattribut, anhand dessen werden die Subgruppen in den Daten gesucht. Außerdem wird sowohl das maximale Level als auch die Anzahl der besten Regeln, aus denen Kandidaten für die nächsten Levels generiert werden, eingegeben. Da der *streamSD* auch Lossy Counting für das zählen der Häufigkeit verwendet, müssen auch Support- und Fehlerparameter gesetzt werden. Der Zeitpunkt, zu dem höhere Levels generiert werden sollen, muss auch eingegeben werden. Dieser Zeitpunkt ist die Anzahl der verarbeiteten Datenitems. Er wird durch den Zeitparameter t festgesetzt.
Die Initialisierungsphase endet, nachdem dieser Zeitpunkt zum ersten Mal erreicht wurde. Bis dahin werden die ersten Kandidaten aus dem Datenstrom generiert und ihre Häufigkeit gezählt.

Update der Kandidaten

Das Update der Kandidaten kann man in zwei Kategorien einteilen. Die eine Kategorie ist das Update der Häufigkeit der vorhandenen Kandidaten. Die andere ist das Update der vorhandenen Levels. Unter dem Update vorhandener Levels versteht man die Überprüfung ob neue Kandidaten vorhanden sind, und das Generieren der Kandidaten für das aktuell erhöhtes Level. In dieser Kategorie werden neue Kandidaten generiert.
Die Initialisierungsphase dient dazu, dass die ersten Kandidaten für Level 1 erzeugt werden. Nicht nach der Initialisierungsphase werden alle Kandidaten für die verschiedenen Levels erzeugt, sondern das geschieht schrittweise, weil es einen großen Rechenzeit und einen unnötigen Speicherbedarf braucht. Nachdem eine bestimmte Anzahl t von Datenitems verarbeitet wurde, werden Kandidaten für höherer Levels bottom-up erzeugt. Außerdem ist die Zahl zu generierender Levels mit einem vom Benutzer übergebenen Parameter beschränkt. Wenn t erreicht wird, wird auch überprüft ob neue Kandidaten für die vorhandenen Levels erzeugt werden können. Wenn das maximale Level nicht erreicht wurde, werden Kandidaten aus den top-k Regeln des aktuellen Levels in Kombination mit den Top-k Regeln des Levels 1 für das aktuell erhöhte Level generiert. Die Kandidaten für Level 1 werden sofort aus dem Datenstrom erzeugt. Nur aus den besten Regeln werden Kandidaten höherer Levels erzeugt. Das veranschaulicht das Aprori Prinzip [2]. Bei dem *Apriori-Algorithmus* werden Obermengen nur aus den Häufigsten Mengen erzeugt.
Die Häufigkeit der Kandidaten wird mit Hilfe des verlustbehafteten Zählalgorithmus LC gezählt. Er unterteilt die aus dem Datenstrom ankommenden Elementen in Puffer. Wenn ein Puffer voll ist, fängt LC mit einer Speicherbereinigung an. Er löscht alle seltenen Kandidaten aus dem Speicher. Nach jeder Verarbeitung eines Datenitems wird die Häufigkeit der vorhandenen Kandidaten erhöhrt.

Generieren und Evaluieren von Regeln

Bei der Subgruppenentdeckung beschreiben die Regeln Datenmengen bezüglich einem Zielattributes. Man will die besten Regeln sehen. Deshalb werden die generierten Regeln bewertet. Auf diesem Grund werden aus den Kandidaten Regeln generiert und bewertet.
Zum Zeitpunkt t wird nicht nur ein Update der Kandidaten durchgeführt, sondern es werden auch Regeln erzeugt und evaluiert. Alle Kandidaten, die in der Ausgabemenge des LC sind, werden Regeln, d.h alle Kandidaten, deren Häufigkeit größer als sN ist, wobei s der Support Parameter von LC ist und N ist die Länge des Datenstromes. Die für alle Levels generierten Regeln werden mit Hilfe einer Qualitätsfunktion (siehe Kapitel 2) evaluiert. Die Regeln werden nach ihrer Güte sortiert. Außerdem kann der Benutzer zu jeder Zeit die aktuellen besten Regeln aller Levels abfragen.

Algorithmus 3.3.1 Subgruppen Entdeckung auf den Datenströmen *streamSD*

Input: Datenstrom D, Trigger t, LCParameter, Gütefunktion ϕ, MacLEvel m

Output: Jeder Zeit die besten Subgruppen

```
1: id = 0
2: initialisiereLC(LCParameter
3: for all Dataitem x ∈ D do
4:   for all Level i do
5:     for all Candidate c in Level i do
6:       if c ∈ x then
7:         count[i][c]++
8:       end if
9:     end for
10:  end for
11:  if id MOD t == 0 then
12:    for all Level i do
13:      for all Candidate c in Level i do
14:        evaluate(c,φ)
15:      end for
16:      if i+1 < maxLevel then
17:        for all Top-k rule r in Level i do
18:          Level[i+1]= Level[i+1] ∪ c × Level[1]
19:        end for
20:      end if
21:      updateAllLevel()
22:    end for
23:  end if
24:  id++
25:  print(top-k rule)
26: end for
```

Ablauf des Algorithmus

Die drei Phasen werden nicht simultan ausgeführt. Die Initialisierungsphase ist nach der Verarbeitung von t Datenitems beendet. Während des Update der Häufigkeit der Kandidaten entlang der Verarbeitung des Datenstroms durchgeführt wird, werden die Generierung und Evaluierung der Regeln und das Update der Kandidaten nur immer , wenn t Datenitems verarbeitet wurden, durchgeführt. Diese drei Phasen wiederholen sich in Zyklisch entlang der Datenstromverarbeitung. Der Ablauf des Algorithmus wird in Abbildung 3.1 veranschaulicht. Eine Beschreibung des Ablaufes des *streamSD*-Algorithmus ist in 3.3.1 als

Pseudocode gegeben.

Die Initialisierung von Parametern sowie der Lossy Couting Datenstruktur für alle Levels ist nach der dritten Zeile beendet. Von Zeile 4 bis Zeile 9 wird die Häufigkeit der Kandidaten aller vorhandenen Levels gezählt. Die Initialisierungsphase endet, nachdem Zeile 11 zum ersten Mal erreicht wird. Von Zeile 1 bis Zeile 23 wird die Phase der Generierung und Evaluierung von Regeln simuliert. Immer wenn t Datenitems verarbeitet wurden, wird Zeile 11 erreicht. Von der Zeile 12 bis 15 werden die Regeln generiert und evaluiert. Wenn das maximale Level nicht erreicht ist, werden Kandidaten höherer Level erzeugt(Zeile 16 bis 20). Mit `updateAllLevel()` in Zeile 21 wird überprüft, ob neue Kandidaten für die vorhandenen Levels erzeugt werden können.
Der Datenstrom wird nicht in Subsequenzen zerlegt und gespeichert. Die Zeitpunktpara-

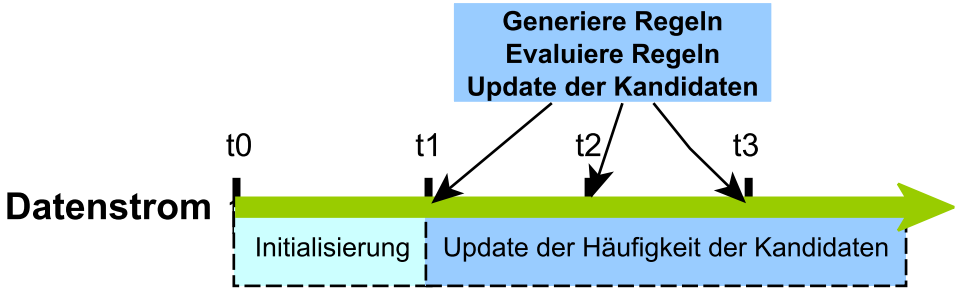


Abbildung 3.1: Arbeitsweise des Algorithmus

met t ist dafür da, dass bestimmte Operationen des Algorithmus zu bestimmten Zeitpunkten durchgeführt werden sollen.

3.4 Implementierung

Damit der Algorithmus sowohl eigenständig in einer APP als auch als Teil einer Datenanalyse einsetzbar ist , wird das *streams-Framework* verwendet. Dabei wird von den Eingenschaften von *streams*, die in Kapitel 2 beschrieben wurden, profitiert. Die benutzte Programmiersprache ist Java. Darüber hinaus wird eine Android-APP entwickelt. Damit wird *streamSD* auf mobilen Geräten lauffähig sein.
In diesem Abschnitt wird die Implementierung des *StreamSD* vorgestellt.

Der Datenstrom

Im *strems* sind verschiedene Datenstrom-Implementierungen vorhanden. Man kann zum Beispiel einen Datenstrom aus einer CSV-Datei erzeugen. Der von *streams* gelieferte Datenstrom ist eine Sequenz von Datenitems. Ein Datenitem ist vom Typ `Map`. Die Schlüssel

der `Map` sind vom Typ String und die Werte sind vom Typ `Serializeable`. Die Strings stellen die Attributen dar. Die Werte der Attribute sind die serialisierten Typen, d.h Typen, die das Java-Interface `Serializeable` implementieren, z.B String oder Integer.

Kodierung der Daten und Datenstruktur

Für eine möglichst schnelle Verarbeitung der aus dem Datenstrom entnommene Daten werden die Daten kodiert. Das übernimmt die Klasse `Crypt`. Sie ist dafür da, um die Daten zu kodieren sowie zu dekodieren.
Ein aus dem Datenstrom entnommenes Element besteht aus einem Attribute und einem Wert. Die Attribute und die Werte der aus dem Datenstrom entnommenen Elemente werden jeweils in einem Vektor gespeichert. Sowohl ein Attribute als auch ein Wert wird nur einmal gespeichert und bekommen eine Id. Die Id ist der Index des Attributes bzw. des Wertes in dem Vektor und ist vom Typ `Integer`. Wenn ein Element aus dem Datenstrom kommt wird überprüft, ob sein Attribute und dessen Wert in dem Vektor gespeichert sind. Wenn der Fall ist, dann bekommt das Attribute und der Wert jeweils deren Id. Wenn nicht der Fall ist, dann werden sie in dem Vektor gespeichert und dann bekommen sie einer Id. Die Attribut- und Wert-Id werden in einem Objekt gekapselt, die von der Klasse `Element` modelliert wird. Alle erzeugten `Element`-Objekte werden auch in einem anderen Vektor gespeichert. Der Index jedes Elements in dem Vektor bestimmt die Id. Wenn später bei der Datenstromverarbeitung das gleiche Element vorkommt, bekommt sofort seine Id. Damit arbeitet Der Algorithmus nur mit 32 Bit `Integer` weiter. Daher wird z.B für tausend Kandidaten des Levels 5 knapp $25KB$ Speicher gebraucht.
Es wurde auf diese Datenstruktur entschieden, weil man nicht auf die Reihenfolge der aus dem Datenstrom entnommene Elemente achten muss.

Lossy Counting Implementierung

Bei einer Subgruppen Entdeckung ist es wichtig, dass die Häufigkeit der Kandidaten gezählt wird. Im *streamSD*-Algorithmus wird der verlustbehaftete Zählalgorithmus LC verwendet. Er löscht in regelmäßigen Abständen alle seltenen Kandidaten. Damit benötigt der *streamSD*-Algorithmus keine Suchstrategie , um zu wissen welche Kandidaten für die Erzeugung der Regeln betrachtet werden sollen.
Die Lossy Counting Datenstruktur ist ein Suchbaum von Kandidaten, die jeder ein Zähler zugewiesen wird. Das ist in der Klasse `LossyCounting` modelliert. Ein Zähler ist von Typ `Counter`. Er speichert die Häufigkeit eines Kandidaten und die Fehlergrenze. `LossyCounting` verwendet die von Java bereitgestellte generische Klasse `TreeMap`. Die Schlüssel sind die Kandidaten und der dazugehörige Zähler als Wert gespeichert. In der `LossyCounting`-Klasse wird der LC-Algorithmus implementiert.
Der LC unterteilt den Datenstrom in Puffer gleiche Größe. Die Elemente werden nach-

einander in den Puffern geschrieben. Mit einem Element wird in dieser Implementierung einen Datenitem bezeichnet, da ein Kandidat nur einmal in einem Datenitem vorkommen kann. Wenn ein Puffer voll ist wird der Löschvorgang gestartet und ein neuer Puffer wird geöffnet. Die Größe w des Puffers ist durch ϵ bedingt ($w = \lceil 1/\epsilon \rceil$).
In der aktuellen Implementierung werden nicht alle Datenitems gespeichert sondern wird es die Anzahl der verarbeiteten Datenitems gezählt. Wenn sie gleich oder ein Vielfaches der Puffergröße ist, werden die Löschvorgänge gestartet. Zum Beispiel, wenn $\epsilon = 0,05$ wäre, wäre $w = 20$ d.h immer nachdem 20 Datenitems verarbeitet wurden, wird der Speicher durch das Löschen seltener Kandidaten aufgeräumt. Das Zählen der Häufigkeit aller Kandidaten wird bei der Verarbeitung jedes Datenitems durchgeführt.

Modellierung der Kandidaten und Regeln

Die Kandidaten der Regeln werden mit den kodierten Daten erzeugt. Das übernimmt die Klasse `CandidatesController`. Sie ist dafür zuständig, dass die Kandidaten erzeugt werden und die Regeln generiert werden.
Ein Kandidat wird in der Klasse `Candidate` modelliert. Er besteht aus einem als `Integer` codierten Zielattribut und einer Menge von `Integer`, die die Bedingung in einer kodierten Form darstellt. Die verschiedenen Levels für die Kandidaten werden von der Klasse `CandidatesTables` modelliert. Jedes Level besitzt sein eigenes `LossyCounting`-Objekt. Der Grund dafür ist die Ermittlung der Anzahl der verarbeiteten Datenitems für jedes Level, da nicht alle Levels gleichzeitig erzeugt werden.
Eine Regel ist in der Klasse `Rule` modelliert (siehe 3.2). Sie hat zusätzlich zum Kandidaten

```
public class Rule implements Comparable<Rule>{


    private Candidate candidate;
    private Integer tp;
    private Integer fp;
    private Integer tn;
    private Integer fn;
    private Double quality;
    private Rule inverseRule;

    public Rule(Candidate candidate){
        this.setCandidate(candidate);
        inverseRule = null;
        quality = 0.0;
        tp=0;
        fp=0;
        tn=0;
        fn=0;
    }
```

Abbildung 3.2: die Klasse Rule

andere Attributen nämlich die ermittelten FP,FN,TP,TN Werte sowie die Güte der Regel. Die Regeln haben auch verschieden Levels. Sie werden nur temporär in einem Suchbaum gespeichert.

Generieren und Update der Kandidaten

Die zwei Phasen Generieren von Regeln und das Update der Kandidaten sind nicht voneinander trennbar. Der Ablauf der zwei Phasen wird in 3.3 veranschaulicht. Das Diagramm

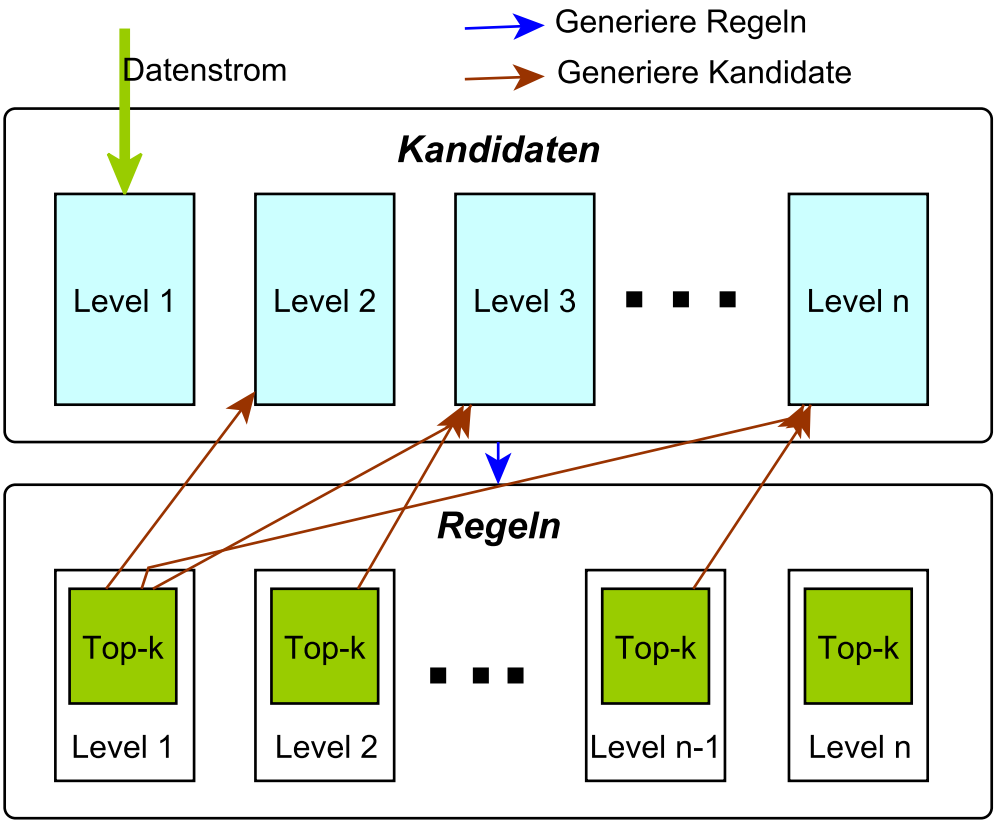


Abbildung 3.3: Ablauf der Generierung und Updaten Phasen

zeigt den Zusammenhang zwischen dem Generieren der Kandidaten und der Regeln. Die Klasse `CandidatesController` ist dafür zuständig, dass Kandidaten generiert werden sowie das Update alle Kandidaten Levels durchgeführt wird. Die Kandidaten für Level 1 werden sofort aus dem Datenstrom generiert. Jedoch nur aus den besten Regeln aus Level 1 und Level $n-1$ werden Kandidaten für Level n generiert. Man soll nur aus den guten Regeln Kandidaten für höhere Levels erzeugen. Die neu erzeugten Kandidaten für Level n werden zuerst die Häufigkeit 0 bekommen. Ab der Verarbeitung des aktuellen Datenitems wird überprüft, ob der Kandidat in dem Datenstrom vorkommt und dabei wird deren Häufigkeit mit LC ermittelt.
Regeln werden alle Kandidaten, die in der Ausgabemenge von Lossy Counting stehen würden. Also alle Kandidaten deren Häufigkeit größer als sn, wobei s der Supportparameter von Lossy Counting und N die aktuelle Länge des Datenstromes ist. Die anderen Kandidaten werden nicht extra gelöscht. Sie werden nur nicht betrachtet bei der Generierung von

Regeln. Wenn die verbliebenen Kandidaten seltener werden, werden sie bei den Löschvorgänge von Lossy Conting später gelöscht.

Regeln evaluieren und Ausgabe

Aus den besten Regeln jedes Levels werden Kandidaten für das nächste Level generiert. Deshalb müssen die Regeln evaluiert werden.
Für die Bewertung der Regeln wird, wie in Kapitel 2 beschrieben wurde, eine Qualitätsfunktion verwendet. Verschiedene Qualitätsfunktionen sind in der Klasse `QualityMeasure` implementiert. Auf diesem Grund müssen die Werte von FP,FN,TP,TN ermittelt werden. Das übernimmt die Methode `setContengencyTable`. Danach berechnet die Methode `evaluate` die Güte jeder Regel. Weil die Berechnung von Qualitätsfunktionen die Anzahl der negativen und positiven Datenitems(auch Beispiele genannt) benötigt, werden sie durch der Klasse `LossyCounting` ermittelt. `nItem` bezeichnet die Anzahl der negativen Datenitems und `pItem` bezeichnet die Anzahl der positiven Datenitems. In einem positiven Datenitems hat das Zielattribut den Wert 1 und in einem negativen hat es den Wert "0". Nach der Bewertung der Regeln, werden die besten positiven Regeln gespeichert gespeichert und für die GUI bereitgestellt. Positiven Regeln sind die Regeln die das Zielattribute entsprechen.

Der Subgruppenentdeckung Prozessor

Für die Subgruppenentdeckung wurde ein Prozessor implementiert. `SubgroupDiscoveryProc` erweitert die Klasse `AbstractProcessor` von dem *streams-Framwork*. Der Prozessor implementiert den in Rahmen dieser Arbeit entwickelten *streamSD*-Algorithmus. Mit Hilfe dieses Prozessors kann jetzt im *streams* einen Subgruppenentdeckung auf Datenströmen durchführen.

Die Android-App

Die in diese Arbeit entwickelte Android-APP ermöglicht die Ausführung des *streamSD* Algorithmus auf einem Smartphone. Jedoch ist die APP noch nicht in der Lage, die aus dem Smartphone kontinuierlichen produzierten Daten zu verarbeiten um die als Datenstrom zu betrachten. Dafür benötigt man eine andere APP, die als Hintergrundprozess immer läuft und für die andere APP einen Datenstrom bereitstellt.
Für das testen des Algorithmus auf dem Smartphone bekommt die APP einen Datenstrom aus einem schon vorher gesammelten Daten.
Die APP ist als Test-APP auf dem CD-ROM im Anhang B zu finden.

3.5 Vor- Und Nachteile der Implementierung

Die Modellierung des Algorithmus und die Auswahl der Datenstrukturen hat Vor- und Nachteile.

Erzeugung der Kandidaten

Die Kandidaten für das Level 1 werden aus dem Datenitem generiert. Dagegen werden Kandidaten höherer Levels aus den besten Regeln der darunter liegenden Levels erzeugt. Das hat den Vorteil, dass nicht bei der Verarbeitung jedes Datenitems verschiedene Teilmengen von Attributen, die Kandidaten seien könnten, für alle Levels erzeugt werden. Damit kann man Rechenzeit sparen. Außerdem sollten die nicht vorkommenden und seltene Kandidaten nicht betrachtet werden. Deshalb werden Kandidaten für die höheren Levels nur aus den wichtigen Regeln erzeugt.

Änderung der Anzahl der Attribute

Der *StreamSD*-Algorithmus ist so implementiert, dass er eine Änderung in der Anzahl der Attribute berücksichtigt. Die Anzahl der betrachteten Attribute soll nicht, während der Datenstromverbreitung konstant sein. Es könnten Attribute entfernt oder hinzugefügt werden.
Das ist sinnvoll, wenn man z.b Datenströme Android-Systeme betrachten soll und man als Attribute die Apps-Namen hat. Der Benutzer kann bestimmte Apps löschen oder neue Apps installieren. Das soll kein Problem sein wenn man eine Subgruppenentdeckung durchführt.

Kapitel 4

Evaluation

In diesem Kapitel werden die auf dem *streamSD*-Algorithmus durchgeführten Experimente präsentiert und beschrieben. Die Experimente werden auf einen Windows-Rechner sowie für die entwickelte Android-App auf einem Android-Smartphone durchgeführt. Außerdem werden die Ergebnisse mit den Ergebnissen des Cortana-tools [15] verglichen.
Die zu durchführende Experimente sollen nicht zeigen, dass mit diesem Algorithmus die besten Subgruppen entdeckt werden sondern, dass:

- die Durchführung eine Subgruppenentdeckung aus den Datenströmen möglich ist.
- eine Übersicht über den Speicherverbrauch und die Laufzeit des Algorithmus verschaffen.
- das ausführen des Algorithmus auf einen Android-Smartphone möglich ist.

Erste Versuche, die während der Implementierung durchgeführt worden sind, haben gezeigt, dass der Algorithmus einen akzeptablen Speicherbedarf und Laufzeit auf einem Windows Rechner hat.
In den folgenden Abschnitten wird sowohl die verwendete Hardware als auch die die verwendeten Testdaten vorgestellt. Darüber hinaus werden die Vorgehensweise für die Durchführung der Experimente erklärt und die Ergebnisse der Versuche analysiert und diskutiert werden.

4.1 Hardware

Die Experimente werden auf einem Laptop mit 64-bit Windows Betriebssystem durchgeführt. Der Laptop hat 8GB Arbeitsspeicher und besitzt einen Intel Prozessor Intel(R) Core(TM) i7-3630QM 2,4GHz. Das Android Smartphone auf dem die App zum Testen laufen soll, hat 3GB Arbeitsspeicher und einen 2,3 GHz-Quad-Core-Prozessor der Marke Qualcomm MSM8974AB. Die verwendete Android-Version ist Android 4.4 (kitkat).

4.2 Datensätze

In diesem Abschnitt werden die Testdateien vorgestellt. Jede Datei wird eine Datenstromquelle darstellen. Auf jedem Datenstrom wird der Algorithmus mit verschiedenen Parametereinstellungen durchgeführt. Es werden drei Datensätze unterschiedlicher Größe verwendet. Die Datensätze enthalten reale, aus Android-Smartphone gesammelte, Daten. Die Datensätze sind auf der CD-ROM in Anhang B zu finden.

Vorbereitung der Datensätze

Die drei für die Experimente ausgesuchten Datensätze enthalten reale Android-Smartphone Daten. Die Daten wurden von Smartphone-Nutzern während einer normalen Verwendung des Handys gesammelt [19]. Die originalen gesammelten Daten wurden verarbeitet und daraus sind die in dieser Arbeit zu verwendeten Datensätze entstanden.
Die Datensätze sind CSV-Dateien. Als Attribute haben sie die Apps-Namen sowie Bezeichnungen von Zusatzhardware und -sensorik z.B WLAN. Die Werte der Attribute liefern Informationen über den Status der einzelnen Systemkomponenten. Für die Apps bezeichnet "1", dass die App gerade in Ausführung ist und die "0"bedeutet, dass die App momentan nicht verwendet wird. Für die Systemeingenschaften wird es das gleiche bedeuten. Zum Beispiel, wenn der WLAN eingeschaltet ist, wird das Attribute WLAN den Wert "1"haben. Außerdem gibt das Attribut `batlevel` Informationen über den Batteriezustand. Seine Werte sind ganze Zahlen.
In dieser Arbeit wird der Energieverbrauch des Android-Smartphones mit Hilfe des *streamSD*-Algorithmus untersucht. Das Zielattribut ist `batlevel`. Mit dem *streamSD*-Algorithmus werden Regeln aufgefunden, die das Zielattribut `batlevel` mit den Werten anderer Attribute erklären. Da das Zielattribute binär sein muss, werden die originale Werte des Attributs `batlevel` diskretisiert. Das Skript dafür ist im Workspace des Projektes auf dem CD-ROM in Anhang B zu finden.
Die Datensätze haben die gleiche Form. Jedoch haben sie nicht alle die gleichen Attribute und die gleiche Länge. Damit liefern sie unterschiedliches Wissen über verschiedene Benutzer. In den folgenden Unterabschnitten werden die einzelnen Datensätze konkret beschrieben.

Beschreibung der Datensätze

Die originale verwendeten Daten sind $APPS\_BAT\_A\_1800s$, $APPS\_BAT\_G\_1800s$ und $APPS\_BAT\_J\_1800s$. Sie sind im Anhang B auf dem CD-ROM zu finden. Sie enthalten auch Informationen über den Zeitpunkt, in dem die Daten gesammelt worden sind.
Für die Experimente werden die diskretisierten Dateien `dataA`, `dataB` und `dataC` verwendet.

	dataA	dataG	dataJ
Diskretisiert aus $APPS\_BAT\_$	A_1800s	G_1800s	J_1800s
Anzahl der Attribute	108	66	111
Anzahl der Datenitems	991	2235	3984

Tabelle 4.1: Datensätze Informationen

Man kann sie auch auf dem CD-ROM im Anhang B finden.
Informationen über die Datensätze sind in Tabelle 4.1 eingetragen.
Darüber hinaus um zu überprüfen dass, der Algorithmus nicht nur binäre Attributen Werte akzeptiert, wird der Datensatz `mushroom.csv` benutzt, Der Datensatz besteht aus verschiedenen Attributen, die die Eigenschaften von Pilzen darstellen. Das Zielattribute in diesem Datensatz ist target". Er hat die Werte ëünd pünd liefert die Information ob ein Pilz essbar(ë") oder nicht(p"). Informationen über diesen Datensatz ist in 4.2 zu finden. Da das Zielattribute bei der Subgruppenentdeckung binär sein muss, wurden die Werte von targetïnvertiert(e→1, p→0). Die Datei mit den konvertierte ist `dataMushroom.csv`. Beide Dateien sind auch auf dem CD-ROM im Anhang B zu finden.

	Anzahl der Attribute	Anzahl der Datenitems
`dataMushroom.csv`	22	5643

Tabelle 4.2: Informationen für dataMushroom.csv

4.3 Experimente

In diesem Abschnitt werden die durchzuführenden Experimente sowie ihre Ergebnisse beschrieben. Wie schon am Anfang des Abschnitts erwähnt wurde, wird der SSD-Algorithmus auf verschiedenen Datensätzen durchgeführt. Für die Experimente werden verschiedene Datensätze verwendet. Aus einem Datensatz wird mit dem *streams-Framework* ein CSV-Datenstrom hergestellt. Auf jedem Datenstrom soll der Algorithmus mit verschiedenen Parametern gestartet werden.
Zum Evaluieren werden für die Bewertung der Subgruppen die zwei Qualitätsfunktionen F1-Score und WRAcc benutzt. Deren Berechnungen basieren auf den einfachen Gütefunktionen wie z.B Precison, Accuracy oder Recall und liefern bessere und präziserere Bewertungen als die anderen. Der Algorithmus wird auch mit verschiedenen Parameterkombinationen gestartet. Dabei wird der Speicherverbrauch im Fokus stehen. Außerdem werden die Ergebnisse des in dieser Arbeit entwickelten Datenstromalgorithmus mit den Ergebnissen der Subgruppenentdeckung auf statischen Daten verglichen. Dabei wird der gleiche Datensatz verwendet aus dem der Datenstrom für den *streamSD*-Algorithmus erzeugt wurde.

Wenn bei den Experimenten keine Änderung eines Parameters erwähnt wurde, dann hat er die Standardeinstellung. Die Standardeinstellung ist $\epsilon = 0,002$, $maxLevel = 5$, $t = 100$, $s = 0,01$ und $k = 50$.

4.3.1 Laufzeituntersuchung

In diesem Unterabschnitt wird die Laufzeit des Algorithmus untersucht. Dabei wird untersucht welche der drei Parameter t, ϵ und k den größten Einfluss auf der Laufzeit hat. Für diese Versuche werden die zwei größten Datensätze, die in `DataG.csv` und `DataH.csv` sind, verwendet.
Da das Ziel des folgenden Versuches ist, festzustellen, welcher der drei Parameter ϵ, t und k den größten Einfluss auf der Laufzeit des Algorithmus hat. In dem Versuch werden bei jeder Durchführung zwei Parameter festgesetzt und der andere wird variiert. Während dieser Versuche werden die Werte des Supportparameters auf $s = 0,01$ und des maximalen Levels auf $maxLevel = 5$ festgelegt. Die gemessene Laufzeit wird in Sekunden gegeben. Die Ergebnisse sind in den Tabellen 4.3 und 4.4 eingetragen.

t=100 $\epsilon = 0,002$	Laufzeit	t=100 k=50	Laufzeit	k=50 $\epsilon = 0,002$	Laufzeit
k=10	5,907	ϵ =0,005	160,653	t=100	155,461
k=25	30,245	ϵ =0,001	161,921	t=200	71,871
k=50	155,461	ϵ =0,0005	161,510	t=300	20,003

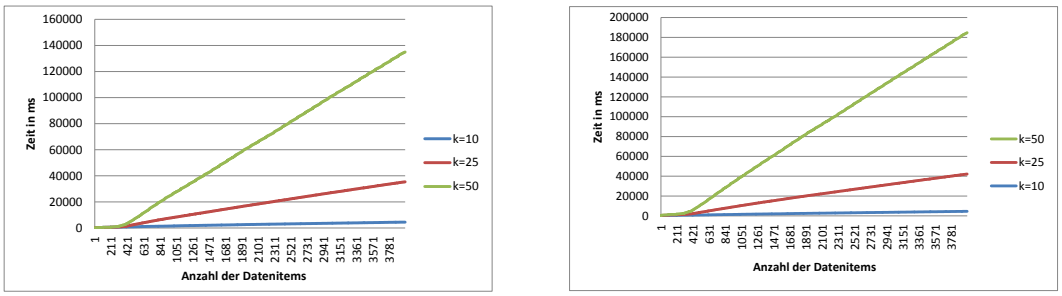
Tabelle 4.3: Laufzeit bei Anwendung auf DataG

t=100 $\epsilon = 0,002$	Laufzeit	t=100 k=50	Laufzeit	k=50 $\epsilon = 0,002$	Laufzeit
k=10	8,871	ϵ =0,005	118,604	t=100	142,859
k=25	40,155	ϵ =0,001	115.369	t=200	108,190
k=50	142,859	ϵ =0,0005	153,767	t=300	146,738

Tabelle 4.4: Laufzeit bei Anwendung auf DataJ

Wenn man den Fehlerparameter ϵ verkleinert, ist nur eine leichte Steigerung der Laufzeit zu sehen, z.B. von 118, 604 auf 115.369, wenn man den Fehler ϵ bei `dataG` von 0,001 auf 0,0005 verkleinert. Wenn man jedoch t verändert, kann man eine Änderung in der Laufzeit betrachten. Außerdem ist eine höherere Steigerung der Laufzeit in den beiden Tabellen zu sehen, wenn man den Parametern k einen größeren Wert zuweist. In 4.4, als der Wert von k von 25 auf 50 verdoppelt wurde, hat sich die Laufzeit fast vervierfacht von 40,155 auf 142,859 Sekunden. Damit hat der Parameter k den größten Einfluss auf die Laufzeit des Algorithmus.

Für die Laufzeitanalyse wäre auch interessant zu wissen, wie sich die Laufzeit entlang des Datenstroms verhält. Dafür wird die Zeit für verschiedene k nach der Verarbeitung jedes Datenitems gemessen. Für dieses Experiment wird der Datenstrom aus der Datei `dataG.csv` verwendet und für $\epsilon = 0,002$ und $\epsilon = 0,001$ durchgeführt. Die Werte von ϵ sind so gewählt, das die Größe des *Lossy Counting* Puffers sich verdoppelt. Die Ergebnisse sind in 4.1 zu sehen.



(a) Laufzeit für $\epsilon = 0,002$ (b) Laufzeit für $\epsilon = 0,001$

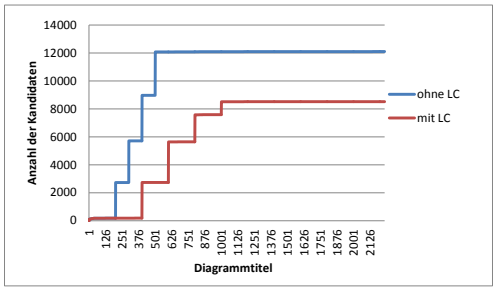
Abbildung 4.1: Laufzeit des Algorithmus

In 4.1 kann man beobachten, dass die Laufzeit linear ist. Je größer k ist desto höher ist die Steigung. Darüber hinaus kann man sehen, dass die Änderung des Fehlerparameters ϵ keinen starken Einfluss auf die Laufzeit hat.

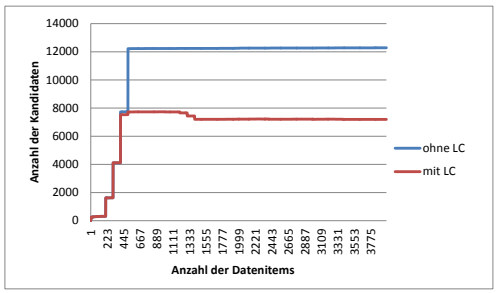
4.3.2 Speicherverbrauchanalyse

In diesem Unterabschnitt wird der Speicherverbrauch des Algorithmus untersucht. Dabei wird die Anzahl der gespeicherten Kandidaten während der Laufzeit gemessen.
Das Hauptziel für die Verwendung des *Lossy Counting* Algorithmus in dieser Arbeit war die Minimierung des Speicherverbrauchs. *Lossy Counting* entfernt in regelmäßigen Abständen, nachdem jeder Puffer voll ist, seltene Elemente. Im nächsten Experiment wird der *streamSD*-Algorithmus einmal mit und einmal ohne *Lossy Counting* gestartet. Dieses Experiment wird auf den Datenströmen aus `DataG.csv` und `DataJ.csv` durchgeführt. Der Versuch wird auf den zwei Datenströmen mit verschiedenen Fehlerparametern ϵ und und Zeitparameter t durchgeführt, weil die zwei Datenströme unterschiedliche Längen haben. In beiden Fällen ist $k = 50$. Die Ergebnisse sind in 4.2 zu sehen.

In den beiden Diagrammen in 4.2 ist klar zu beobachten, dass die Anzahl der gespeicherten Kandidaten bei der Verwendung von *Lossy Counting* weniger wächst ist als ohne



(a) Ausführung auf DataG.csv $\epsilon = 0,002$ und $t = 200$

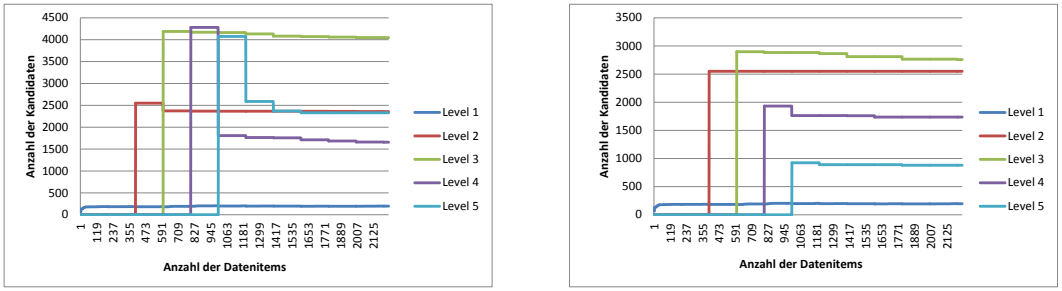


(b) Ausführung auf DataJ.csv $\epsilon = 0,001$ und $t = 100$

Abbildung 4.2: Vergleich der Anzahl der gespeicherten Kandidaten mit und ohne Lossy Counting

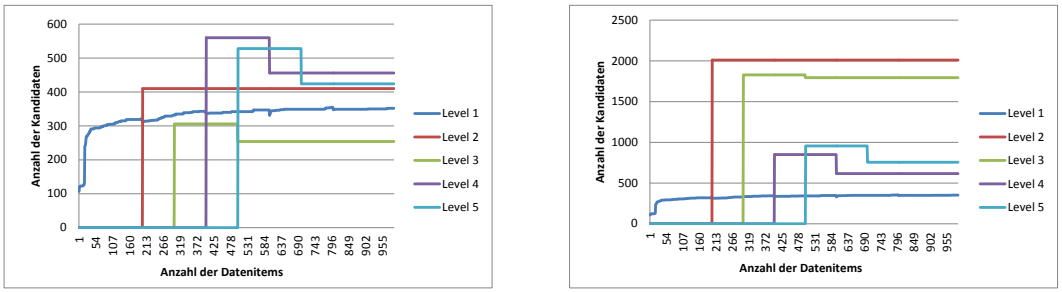
Lossy Counting. Daraus folgt, dass der *streamSD*-Algorithmus bei der Anwendung von *Lossy Counting* weniger Speicher verbraucht. Jedoch werden die gleichen Top-10 Subgruppen ausgegeben.
Jetzt wird untersucht, ob die Auswahl der Qualitätsfunktion eine Wirkung auf den Speicherbedarf hat.
Die Kandidaten für Level n werden aus den besten Regeln von Level $n-1$ und Level 1 erzeugt. Die Regeln werden nach ihrer Güte sortiert. Für die Ermittlung der Güte einer Regel wird eine Qualitätsfunktion verwendet. In diesem Experiment werden die zwei Qualitätsfunktionen WRAcc und F1-Score verwendet. Deren Beschreibung ist in Kapitel 1 zu finden. Die Datenstromquelle ist `DataG.csv`. Für die zwei Experimente ist $k = 50$. Gemessen wird dabei die Anzahl der Kandidaten in jedem Level. Die Ergebnisse dieses Experiments sind in 4.3 zu sehen.
In 4.3 ist zu sehen, dass bei Änderung der Qualitätsfunktion sich die Anzahl der gesamten sowie auf den einzelnen Levels gespeicherten Kandidaten ändert.
Der Parameter k ist ein entscheidender Faktor in dem *SSD*-Algorithmus. Er bestimmt die Anzahl der Regeln aus denen Kandidaten höheren Levels erzeugt werden. Bei der Laufzeitanalyse wurde festgestellt, dass die Änderung von k einen starken Einfluss auf die Laufzeit des Algorithmus hat. Bei dem nächsten Experiment wird untersucht, wie sich der Speicherverbrauch bei der Änderung von k verhält. Dabei werden alle anderen Parameter festgesetzt und nur k geändert. Für dieses Experiment werden Datenströme aus `DataA.csv`, `DataG.csv` und `DataJ.csv` verwendet. Die Ergebnisse sind in 4.4, 4.5 und 4.6 zu sehen.

In 4.4, 4.5 und 4.6 kann man beobachten, dass es bei der Änderung von k eine Änderung in der Anzahl der Kandidaten gibt. Wenn der Wert von k verdoppelt wurde, ist die



(a) Qualitätsfunktion WRACC (b) Qualitätsfunktion F1SCORE

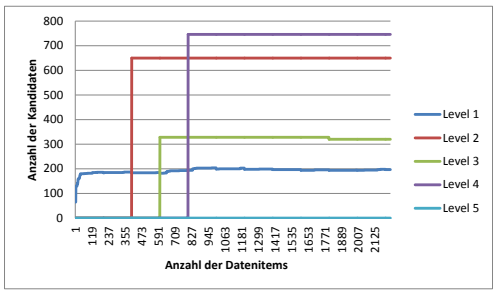
Abbildung 4.3: Vergleich der Anzahl der gespeicherten Kandidaten für verschiedene Qualitätsfunktionen(dataG.csv)



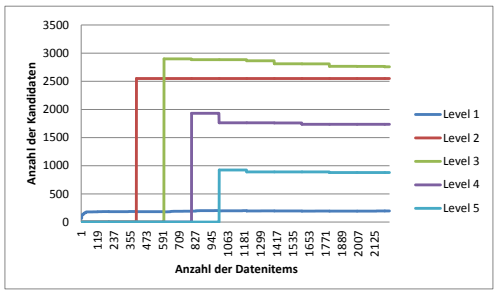
(a) k=25 (b) k=50

Abbildung 4.4: Vergleich der Anzahl der gespeicherten Kandidaten für verschiedene k (dataA.csv)

Anzahl der Kandidaten in allen Levels gestiegen, außer in Level 1. Zum Beispiel hat sich die Anzahl der Kandidaten von Level 2 in 4.5 fast verzehnfacht. Damit steigt auch der Speicherbedarf des Algorithmus.
Zudem wird noch die Wirkung des Fehlerparameters ϵ auf den Speicherbedarf untersucht. Der Fehlerparameter ϵ regelt indirekt die Löschvorgänge von *Lossy Counting*. Er bestimmt die Größe der Puffer ($w = \lceil 1/\epsilon \rceil$). Für das Experiment werden Datenströme aus `dataG.csv` und `dataJ.csv` verwendet. In diesem Experiment wird $k = 25$ verwendet. Gemessen wird die gesamte Anzahl der Kandidaten während der Laufzeit. Die Ergebnisse sind in 4.7.
Die Durchführung des Experiments auf `DataG.csv` zeigt keine Änderung in der Anzahl von Kandidaten bei der Änderung von von ϵ. Ein Grund dafür wäre, dass keine seltenen Elemente in diesem Datenstrom vorkommen. Jedoch zeigt das rechte Diagramm in 4.7, dass

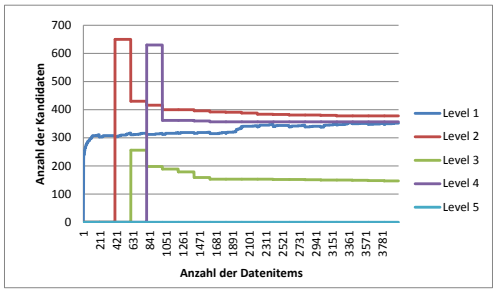


(a) k=25

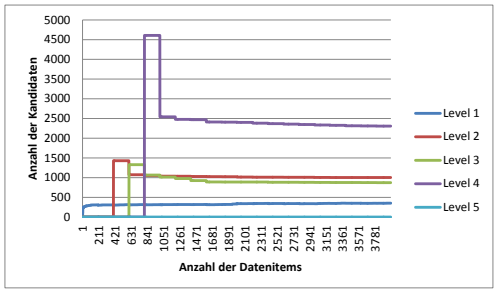


(b) k=50

Abbildung 4.5: Vergleich der Anzahl der gespeicherten Kandidaten für verschiedene k (dataG.csv)



(a) k=25



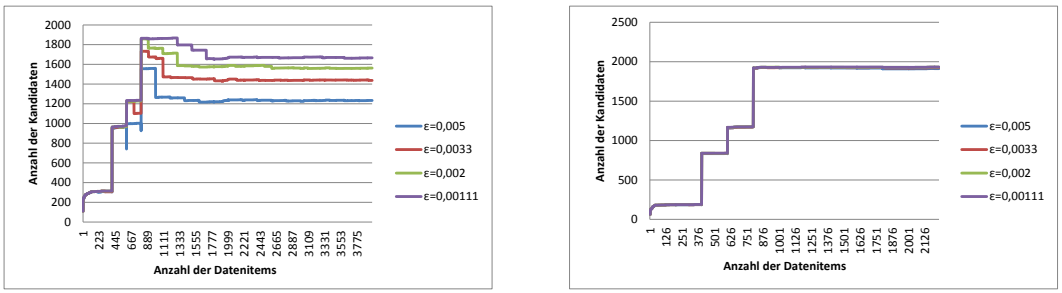
(b) k=50

Abbildung 4.6: Vergleich der Anzahl der gespeicherten Kandidaten für verschiedene k (dataJ.csv)

das Vergrößern von ϵ den Speicherbedarf reduziert. Außerdem kann man die Überlappung der Löschvorgänge und das Update bzw. das Generieren eines höheren Levels (der Fall, in dem $t = \epsilon$ ist) durch die Reduzierung und die sofortige Steigerung der Anzahl der Kandidaten erkennen (z.B. in Datenitem zwischen 445 und 667). Also wenn die Löschoperation und das Erhöhen des aktuellen Level während der Verarbeitung der gleichen Datenitems durchgeführt werden.

Laufzeit auf dem Smartphone

Der *streamSD* wurde auf dem Rechner und auf dem Smartphone startet. Die dafür entwickelte APP ist auf der CD-ROM im Anhang B zu finden. Gemessen wurde die Laufzeit. Die Datenstromquelle war die `mushroom.csv` Datei. Die eingestellten Parameter waren:



(a) `dataJ.csv` (b) `dataG.csv`

Abbildung 4.7: Vergleich der Anzahl der gespeicherten Kandidaten für verschiedene ϵ

$\epsilon = 0,001$, $maxLevel = 3$, $t = 500$, $s = 0,01$ und $k = 25$. Die Ergebnisse des Experimentes sind in 4.5 zu finden.

	Laufzeit in (s)
streamSD auf dem Rechner	22,40
streamSD auf dem Smartphone	1540,35

Tabelle 4.5: Vergleich der Laufzeit

Das Ausfüren des *streamSD* Algorithmus auf dem Smartphone hat fast 10 mal länger gedauert als auf dem Rechner. Jedoch zeigt die App bei der Bearbeitung von 5642 Datenitems eine akzeptable Laufzeit.

Jeder Zeit Ausgabe

Der *streamSD*-Algorithmus ist in der Lage, jederzeit die gefundenen Subgruppen anzuzeigen. Man kann auch die besten Regeln in einer `.txt`-Datei abspeichern. Da das Datum der Anfrage bei einer Datenstromverarbeitung wichtig ist, wird es mit angezeigt bzw. gespeichert. Die aufgefundenen Subgruppen aus dem Datenstrom aus $APPS\_BAT\_G\_1800s$ sind im Anhang A in A.6 zu sehen.

Vergleich mit einem Frequent Itemset Algorithmus

Für dieses Experiment wird *Rapidminer* [16] (Version 6.0) verwendet. Der *FP-Growth* Algorithmus [8] wird verwendet für das Auffinden von den häufigsten Mengen.
In diesem Experiment wird die extensionale Redundanz der einzelnen und der gesamten

∪	ext(R1)	ext(R2)	ext(R3)	∩	ext(R1)	ext(R2)	ext(R3)
ext(M1)	3984	3984	3984	ext(M1)	1959	1960	1960
ext(M2)	3984	3984	3984	ext(M2)	1959	1960	1960
ext(M3)	3984	3984	3984	ext(M3)	1959	1960	1960

Tabelle 4.6: Anzahl der Elemente der Schnitt und Vereinigungsmenge (dataj)

aufgefundenen Regeln bzw. häufigsten Mengen untersucht. Dabei werden die Extensionen der drei besten Regeln von *streamSD* und Mengen von *FP-Growth* betrachtet. Eine Extension besteht aus den Instanzen (Datenitems), die von der Regel überdeckt sind. Die gefundenen Top-3 Regeln bei der Anwendung von *FP-Growth* und *streamSD* ($\epsilon = 0,001$, $maxLevel = 3$) auf `dataJ.csv` sind im Anhang A (A.4,A.1) zu sehen.
Bei der Durchführung der Experimente wurde festgestellt, dass die Extensionen von den drei besten häufigen Mengen, die von FP-Growth gefunden wurden, aus allen Instanzen bestehen. Jedoch überdecken R2 und R3 jeweils 1960 Instanzen und R1 1959 Instanzen. Die Größe der Vereinigungs- und Schnittmengen der Extensionen sind in 4.6 dargestellt.

	Redundanz
|ext(R1)∩ext(M1)| / |ext(R1)∪ext(M1)|	0,491
|ext(R1)∩ext(M2)| / |ext(R1)∪ext(M2)|	0,491
|ext(R1)∩ext(M3)| / |ext(R1)∪ext(M3)|	0,491
|ext(R2)∩ext(M1)| / |ext(R2)∪ext(M1)|	0,492
|ext(R2)∩ext(M2)| / |ext(R2)∪ext(M2)|	0,492
|ext(R2)∩ext(M3)| / |ext(R2)∪ext(M3)|	0,492
|ext(R3)∩ext(M1)| / |ext(R3)∪ext(M1)|	0,492
|ext(R3)∩ext(M2)| / |ext(R3)∪ext(M2)|	0,492
|ext(R3)∩ext(M3)| / |ext(R3)∪ext(M3)|	0,492

Tabelle 4.7: Extensionale Redundanz der einzelnen Regeln/Mengen (dataJ.csv)

	Redundanz
|ext(R1)∩ext(R2)∩ext(R3)| / |ext(R1)∪ext(R2)∪ext(R3)|	0,99
|ext(M1)∩ext(M2)∩ext(M3)| / |ext(M1)∪ext(M2)∪ext(M3)|	1

Tabelle 4.8: Extensionale Redundanz der gesamten Regeln/Mengen (dataJ.csv)

Die Ergebnisse der Untersuchung der Redundanz werden in 4.7 und 4.8 eingetragen.

Da die Extensionen der drei häufigsten Mengen M1, M2, M3 jeweils den gesamten Datensatz und die Extensionen der besten drei Regel R1, R2, R3 nur fast die Hälfte der Instanzen überdecken, können die Werte der Redundanz in 4.7 die Eins nicht annähern. Daher sind die Extensionen der einzelnen Regeln nicht gleich der Extensionen der einzelnen häufigsten Mengen, sondern sind sie eine echte Teilmenge der Extensionen der einzelnen häufigsten Mengen.
Die Regeln R2 und R3 haben die selben Extensionen. Die Extension von R1 hat eine Instanz mehr als die anderen. Deshalb ist die totale Redundanz gleich 0,99 (siehe 4.8). Die häufigsten Mengen M1, M2, M3 haben die gleiche Extension. Auf diesem Grund ist deren gesamte Redundanz 1 (siehe 4.8).

Das gleiche Experiment wird auf dem Datensatz `dataMushroom.csv` nochmals durchgeführt. Jedoch soll das Experiment hier nicht nur die Redundanz untersuchen, sondern es soll auch zeigen, dass *streamSD* nicht nur binäre Attribute verarbeiten kann.
Die gefundenen Top-3 Regeln bei der Anwendung von *FP-Growth* und *streamSD* ($\epsilon = 0,001$, $maxLevel = 3$) auf `mushroom.csv` sind im Anhang A (A.2,A.3) zu sehen. Die Anzahl der Elemente der einzelnen Extensionen sowie die Größe derer Schnitt- und Vereinigungsmengen sind in 4.9 und 4.10 zu finden.

	ext(Rp1)	ext(Rp2)	ext(Rp3)	ext(Mp1)	ext(Mp2)	ext(Mp3)
Anzahl der Elemente	4940	4940	4922	5644	5636	5626

Tabelle 4.9: Anzahl der Elemente in der Extensionen(mushromm.csv)

$\cup$	ext(Rp1)	ext(Rp2)	ext(Rp3)
ext(Mp1)	5644	5644	5644
ext(Mp2)	5636	5636	5636
ext(Mp3)	5644	5644	5626

$\cap$	ext(R1)	ext(R2)	ext(R3)
ext(Mp1)	4940	4940	4922
ext(Mp2)	4940	4940	4922
ext(Mp3)	4922	4922	4922

Tabelle 4.10: Anzahl der Elemente der Schnitt und Vereinigungsmenge(mushromm.csv)

Die Ergebnisse der Untersuchung der Redundanz werden in 4.11 und 4.12 eingetragen.

Anhand von 4.10 und 4.9 kann man feststellen, dass die Schnittmenge der Regeln Extensionen und der einzelnen Extensionen der häufigsten Mengen nicht leer ist. Teilweise ist

	Redundanz
|ext(Rp1)∩ext(Mp1)| / |ext(Rp1)∪ext(Mp1)|	0,88
|ext(Rp1)∩ext(Mp2)| / |ext(Rp1)∪ext(Mp2)|	0,88
|ext(Rp1)∩ext(Mp3)| / |ext(Rp1)∪ext(Mp3)|	0,87
|ext(Rp2)∩ext(Mp1)| / |ext(Rp2)∪ext(Mp1)|	0,88
|ext(Rp2)∩ext(Mp2)| / |ext(Rp2)∪ext(Mp2)|	0,88
|ext(Rp2)∩ext(Mp3)| / |ext(Rp2)∪ext(Mp3)|	0,87
|ext(Rp3)∩ext(Mp1)| / |ext(Rp3)∪ext(Mp1)|	0,87
|ext(Rp3)∩ext(Mp2)| / |ext(Rp3)∪ext(Mp2)|	0,87
|ext(Rp3)∩ext(Mp3)| / |ext(Rp3)∪ext(Mp3)|	0,88

Tabelle 4.11: Extensionale Redundanz der einzelnen Regeln/Mengen(mushroom.csv.csv)

	Redundanz
|ext(Rp1)∩ext(Rp2)∩ext(Rp3)| / |ext(Rp1)∪ext(Rp2)∪ext(Rp3)|	0,99
|ext(Mp1)∩ext(Mp2)∩ext(Mp3)| / |ext(Mp1)∪ext(Mp2)∪ext(Mp3)|	0,99

Tabelle 4.12: Extensionale Redundanz der gesamten Regeln/Mengen(mushroom.csv)

die Extension einer Regel eine echte Teilmenge der Extension der einzelnen aufgefundenen häufigsten Mengen. So kann man z.B. beobachten, dass die Extension von R2 eine echte Teilmenge der Extension von Mp2 ist. Aus diesem Grund sind die Redundanz-Werte in 4.11 hoch. Daher sind die Extensionen von den einzelnen Regeln bzw. Mengen ungefähr gleich.
Die Tabelle 4.12 zeigt, dass die gesamte Redundanz gleich ist. Daraus folgt, dass die Ergebnisse von *streamSD* und *FP-Growth* insgesamt genauso gut sind.
Man kann von den gesamten Beobachtungen bei der Durchführung dieses Experiment auf dem Mushroom-Datensatz feststellen, dass die Extension der Top-3 von *streamSD* gefundenen Regeln und die Extensionen der gefundenen häufigsten Mengen von *FP-Growth* eine Schnittmenge haben.

Vergleich mit der statischen Subgruppenentdeckung

Jetzt wird ein Vergleich zwischen der statischen Subgruppenentdeckung und der Subgruppenentdeckung auf dem Datenstrom durchgeführt. Die verwendete Qualitätsfunktion ist `WRAcc`.
Für die statische Subgruppenentdeckung wird Cortana (Version 2.0) verwendet. Dabei wird maximale Länge der Bedingung einer Regel auf drei gesetzt. Die verwendete Suchstrategie ist Beam-Search (beam width = 100). Die anderen Parameter haben die Standardeinstellung von Cortana.

Für die Subgruppenentdeckung auf dem Datenstrom wird der im Rahmen dieser Arbeit entwickelten Algorithmus mit der Standardeinstellungen (außer $\epsilon = 0,001$, $maxLevel = 3$) verwendet.
Für die beiden Einsätze wird `dataJ.csv` verwendet. Hier wird, wie beim Vergleich mit dem *FP-Growth*-Algorithmus, die Redundanz untersucht. Die gefundenen besten Regeln von Cortana sind in A.5 im Anhang A zu finden.
Die Anzahl der Elemente der einzelnen Extensionen sowie deren Schnitt- und Vereinigungsmenge sind in 4.13 und 4.14 dargestellt.

	ext(R1)	ext(R2)	ext(R3)	ext(Rc1)	ext(Rc2)	ext(Rc3)
Anzahl der Elemente	1959	1960	1960	662	662	730

Tabelle 4.13: Anzahl der Elemente in der Extensionen(mushromm.csv)

Die Ergebnisse der Untersuchung der Redundanz werden in 4.15 und 4.16 eingetragen.

Bei der Betrachtung von 4.13 und 4.14 wird festgestellt, dass die Extensionen der besten drei Regeln von Cortana und der besten drei Regeln von *streamSD* nicht die gleichen Extensionen haben. Jedoch ist deren Schnittmenge nicht leer. Dies bestätigen die Redundanz-Werte in 4.15, deren Werte die Null annähern.
Die Ergebnisse der Untersuchung der gesamten Redundanz von Cortana und *streamSD* in 4.16 bedeuten, dass die besten Cortana Regeln mehr verschiedene Instanzen als *streamSD* überdecken.

4.4 Diskussion

In diesem Abschnitt werden die Ergebnisse der Experimente interpretiert und diskutiert.

Laufzeit

Die Experimente haben gezeigt, dass der Algorithmus eine akzeptable Laufzeit hat. Die zwei Parameter, die die Laufzeit stark beeinflussten sind t und k.

∪	ext(R1)	ext(R2)	ext(R3)	∩	ext(R1)	ext(R2)	ext(R3)
ext(Rc1)	2150	2150	2149	ext(Rc1)	472	472	472
ext(Rc2)	2149	2150	2150	ext(Rc2)	472	472	472
ext(Rc3)	2171	2172	2172	ext(Rc3)	518	518	518

Tabelle 4.14: Anzahl der Elemente der Schnitt und Vereinigungsmenge(dataj)

	Redundanz
|ext(R1)∩ext(Rc1)| / |ext(R1)∪ext(Rc1)|	0,21
|ext(R1)∩ext(Rc2)| / |ext(R1)∪ext(Rc2)|	0,21
|ext(R1)∩ext(Rc3)| / |ext(R1)∪ext(Rc3)|	0,23
|ext(R2)∩ext(Rc1)| / |ext(R2)∪ext(Rc1)|	0,21
|ext(R2)∩ext(Rc2)| / |ext(R2)∪ext(Rc2)|	0,21
|ext(R2)∩ext(Rc3)| / |ext(R2)∪ext(Rc3)|	0,23
|ext(R3)∩ext(Rc1)| / |ext(R3)∪ext(Rc1)|	0,21
|ext(R3)∩ext(Rc2)| / |ext(R3)∪ext(Rc2)|	0,21
|ext(R3)∩ext(Rc3)| / |ext(R3)∪ext(Rc3)|	0,23

Tabelle 4.15: Extensionale Redundanz der einzelnen Regeln(dataJ.csv)

	Redundanz
|ext(R1)∩ext(R2)∩ext(R3)| / |ext(R1)∪ext(R2)∪ext(R3)|	0,99
|ext(Rc1)∩ext(Rc2)∩ext(Rc3)| / |ext(Rc1)∪ext(Rc2)∪ext(Rc3)|	0.3

Tabelle 4.16: Extensionale Redundanz der gesamten Regeln(dataJ.csv)

Da die Zeitintervalle definiert werden, nachdem neue Regeln erzeugt werden und ein Update der alten Regeln durchgeführt wird, sollte der Algorithmus auf dem gleichen Datenstrom bei größeren Zeitintervallen diese Operationen nicht so oft durchführen. Auf einem Datenstrom, der z.B. aus 3000 Datenitems besteht, werden diese Operationen 30 Mal durchgeführt, wenn $t = 300$ ist. Dagegen werden sie auf dem gleichen Datenstrom nur 10 Mal durchgeführt, wenn $t = 300$ ist. Außerdem bezeichnet k die Anzahl der besten Regeln, aus den Kandidaten für höhere Levels erzeugt werden. Je größer k ist, desto mehr Kandidaten werden für höhere Levels erzeugt. Auf diesem Grund wird auch mehr Zeit benötigt. Jedoch haben die Experimente nicht gezeigt, dass der Fehlerparameter ϵ von *Lossy Counting* keinen starken Einfluss auf die Laufzeit hat, obwohl ϵ die Größe der Puffer w von *Lossy Counting* bestimmt($w = \lceil 1/\epsilon \rceil$). Wenn ein Puffer voll ist, werden die seltenen Elemente von *Lossy Counting* entfernt. Auf einem Datenstrom, der z.B. aus 3000 Datenitems besteht, wird nur dann ein Löschvorgang gestartet, wenn $\epsilon = 0,0005$ ist. Deshalb kann der Grund sein, warum ϵ die Laufzeit nicht stark beeinträchtigt, dass in dem Datenstrom wenig seltene Elemente vorgekommen sind.

Speicherverbrauch

Der andere Aspekt, der in den Experimenten betrachtet wurde, ist der Speicherverbrauch. In dem *streamSD*-Algorithmus wurde *Lossy Counting* verwendet. Dieser löscht seltene in dem Datenstrom vorkommende Elemente. Bei mancher Wertauswahl für den Fehlerpa-

rameter ϵ können die Löschoperationen auf einem endlichen Datenstrom nicht gestartet werden, da ϵ die Größe des Puffers bestimmt und die Löschoperationen nur nach einem vollen Puffer verlaufen. Auf einem Datenstrom der Länge 500 werden die Löschoperationen bei $\epsilon = 0,001$ nicht gestartet, da die Größe des Puffers $w = 1000$ ist.
Die Experimente haben auch gezeigt, dass auch die Auswahl der Qualitätsfunktion eine Auswirkung auf den Speicherbedarf hat. Durch die unterschiedliche Bewertung der verschiedenen Qualitätsfunktionen kann die Anzahl der gefundenen Regeln, deren Zielattribute den vom Benutzer angegebenen Zielattribut entsprechen, kleiner als k sein. Daher werden weniger Kandidaten für das nächste Level erzeugt. Außerdem könnte sein, dass manchmal Qualitätsfunktionen seltene Regeln gut bewerten und sie später von *Lossy Counting* entfernt werden. Diesen Effekt kann man in 4.3 betrachten. Auf dem Diagramm, in dem der F1-Score verwendet wurde, sind Kandidaten kurz nach ihrer Erzeugung von *Lossy Counting* gelöscht worden.
Bei der Untersuchung von k wurde festgestellt, dass die Anzahl der Kandidaten auf Level 1 sich nicht geändert hat als k verdoppelt wurde. Das hat den Grund, dass die Generierung von den Kandidaten in Level 1 nicht von k abhängig ist. Sie werden sofort aus dem Datenstrom erzeugt und nicht aus den darunterliegenden Levels.
Bei allen Versuchen für die Untersuchung des Speicherverhaltens während der Laufzeit ist zu beobachten, dass die Anzahl der Kandidaten sich nach einer bestimmten Zeit stabilisiert. Das bedeutet, dass während der weiteren Verarbeitung des Algorithmus ein konstanter Speicherbedarf besteht. Das hat einen Vorteil bei der Anwendung des SDD-Algorithmus auf einen unendlichen Datenstrom. Nach der Verarbeitung einer bestimmten Anzahl von Datenpaketen bleibt der Speicherverbrauch konstant.
Es hat verschiedene Gründe, warum die besten Regeln von Cortana nicht die gleichen sind wie beim *streamSD*-Algorithmus. Einer davon ist, dass Cortana die gesamten Daten betrachtet und für das Generieren der Regeln alle Levels betrachtet. Dagegen werden im *streamSD*-Algorithmus Regeln höherer Levels schrittweise erzeugt. Das hat zum Beispiel zur Folge, dass die Daten für die Ermittlung der Häufigkeit der Kandidaten aus Level 3 bei $t = 200$ erst nach Verarbeitung von 400 Datenitems betrachtet werden.

Kapitel 5

Zusammenfassung

In dieser Arbeit wurde der Datenstroms-Algorithmus für die Subgruppenentdeckung *StreamSD* eingeführt. Der Algorithmus wurde erklärt, implementiert und evaluiert. *StreamSD* funktioniert ähnlich wie die statische Subgruppenentdeckung. Kandidaten werden aufgefunden, Regeln werden erzeugt und bewertet. Die Kandidaten werden entsprechend ihrer Komplexität in Levels eingestuft. Nicht alle Kandidaten für alle Levels werden auf einmal erzeugt. Kandidaten für Level 1 werden sofort aus dem Datenstrom erzeugt. Immer, wenn eine bestimmte Anzahl von Datenitems verarbeitet wurde, werden Kandidaten höherer Levels bottom-up erzeugt. Die Häufigkeit der Kandidaten wird entlang der Datenstromverarbeitung mit Hilfe des *Lossy Counting*-Algorithmus ermittelt. Aus allen Kandidaten, die in der Ausgabemenge von *Lossy Counting* sind, werden Regeln erzeugt. Die Erzeugten Regeln werden mit eine Qualitätsfunktion bewerte. Aus den besten Regeln von Level 1 und Level n werden Kandidaten für Level $n + 1$ erzeugt. Der Benutzer ist in der Lage in jeder Zeit dies besten Regeln anzeigen zu lassen.
Der *StreamSD*-Algorithmus wurde innerhalb des Datenstrom Frameworks *streams* implementiert. Damit kann der Algorithmus, sowohl als Teil einer Datenstromanalyse als auch eigenständig in einer App einsetzbar.

Anhang A

Regeln Tabellen

	Regel
M1	com.googlecode.android_scripting=1
M2	com.google.android.talk=1
M3	com.google.android.syncadapters.contacts=1

Tabelle A.1: Die drei häufigsten Mengen in(dataJ.csv)

	Regel
Rp1	veil-type=p AND gill-size=b
Rp2	veil-color=w AND gill-size=b
Rp3	gill-attachment=f AND gill-size=b

Tabelle A.2: Top-3 Regeln von streamSD(dataMushroom.csv)

	Regel
Mp1	veil-type = p
Mp2	veil-color = w
Mp3	gill-attachment = f

Tabelle A.3: Die drei häufigsten Mengen in (dataMushroom.csv)

	Regel
R1	com.android.keychain=0 AND com.google.android.inputmethod.latin=1 AND org.leo.android.dict=0
R2	android=1 AND com.android.keychain=0 AND com.google.android.inputmethod.latin=1
R3	com.android.keychain=0 AND com.android.nfc=1 AND com.google.android.inputmethod.latin=1

Tabelle A.4: Top-3 Regeln von streamSD(dataJ.csv)

	Regel
Rc1	com.esys.satfinder = '0' AND com.ideashower.readitlater.pro = '0' AND com.socialnmobile.dictapps.notepad.color.note = '0'
Rc2	com.ideashower.readitlater.pro = '0' AND com.esys.satfinder = '0' AND com.socialnmobile.dictapps.notepad.color.note = '0'
Rc3	com.esys.satfinder = '0' AND com.ideashower.readitlater.pro = '0' AND de.barcoo.android = '0'

Tabelle A.5: Top-3 Regeln von Cortana(dataJ.csv)

Regel	Güte der Regel
android=1 > batlevel=1	0.06117
com.android.nfc=1 > batlevel=1	0.06117
com.android.noisefield=1 > batlevel=1	0.06117
com.android.phone=1 > batlevel=1	0.06117
com.android.providers.applications=1 > batlevel=1	0.06117
com.android.providers.contacts=1 > batlevel=1	0.06117
com.android.providers.settings=1 > batlevel=1	0.06117
com.android.providers.telephony=1 > batlevel=1	0.06117
com.android.providers.userdictionary=1 > batlevel=1	0.06117
com.android.stk=1 > batlevel=1	0.06117

Tabelle A.6: Top-10 Regeln von *streamSD*(dataG.csv)

Anhang B

Datenträger

Dieser Ausarbeitung ist eine CD-ROM beigelegt, auf der sich zusätzliches Material befindet. Die CD-ROM beinhaltet:

- Diese Ausarbeitung im PDF-Format
- Das Workspace des Projektes
- Die verwendeten Datensätze
- Die entwickelte Android-APP

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] C. Bockermann and H. Blom. The streams framework. Technical Report SFB876 (5), TU Dortmund, 2012.

[2] T. Bollinger. Assoziationsregeln–analyse eines data mining verfahrens. *Informatik-Spektrum*, 19(5):257–261, 1996.

[3] J.-F. Boulicaut, K. Morik, and A. Siebes. *Local Pattern Detection - International Seminar Dagstuhl Castle, Germany, April 12-16, 2004, Revised Selected Papers*. Springer, Berlin, Heidelberg, 2005. aufl. edition, 2005.

[4] T. Calders, N. Dexters, J. J. Gillis, and B. Goethals. Mining frequent itemsets in a stream. *Inf Syst*, 39:233–255, 2014.

[5] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *SIGCOMM Comput. Commun. Rev.*, 38(1):5–5, Jan. 2008.

[6] M. Gargenta and M. Nakamura. *Learning Android: Develop Mobile Apps Using Java and Eclipse*. O'Reilly Media, Incorporated, 2014.

[7] P. D. Grünwald. *The minimum description length principle*. MIT press, 2007.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[9] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[10] F. Herrera, C. J. Carmona, P. González, and M. J. del Jesus. An overview on subgroup discovery: foundations and applications. *Knowledge and information systems*, 29(3):495–525, 2011.

[11] W. Kloesgen. Explora: a multipattern and multistrategy discovery assistant. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in*

Knowledge Discovery and Data Mining, chapter Explora: A Multipattern and Multistrategy Discovery Assistant, pages 249–271. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[12] W. Klösgen. Applications and research problems of subgroup mining. In Z. Raś and A. Skowron, editors, *Foundations of Intelligent Systems*, volume 1609 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 1999.

[13] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski. Subgroup discovery with cn2-sd. *The Journal of Machine Learning Research*, 5:153–188, 2004.

[14] M. Mampaey, S. Nijssen, A. Feelders, and A. Knobbe. Efficient algorithms for finding richer subgroup descriptions in numeric and nominal data. In *IEEE International Conference on Data Mining*, pages 499–508, 2012.

[15] M. Meeng and A. Knobbe. Flexible enrichment with cortana–software demo. In *Proceedings of BeneLearn*, pages 117–119, 2011.

[16] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940. ACM, 2006.

[17] P. Mills. Efficient statistical classification of satellite measurements. *International Journal of Remote Sensing*, 32(21):6109–6132, 2011.

[18] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[19] J. Streicher, N. Piatkowski, K. Morik, and O. Spinczyk. Open smartphone data for mobility and utilization analysis in ubiquitous environments. In M. Atzmüller and C. Scholz, editors, *Proceedings of the 4th International Workshop on Mining Ubiquitous and Social Environments (MUSE)*, 2013.

[20] L. Todorovski, P. Flach, and N. Lavrač. *Predictive performance of weighted relative accuracy*. Springer, 2000.

[21] M. Van Leeuwen and A. Siebes. Streamkrimp: Detecting change in data streams. In *Machine Learning and Knowledge Discovery in Databases*, pages 672–687. Springer, 2008.

[22] J. Vreeken, M. Van Leeuwen, and A. Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.

[23] W. Zhang. Complete anytime beam search. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 425–430, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift