Realisierung eines vertikalen Speichers für das SeCo Regellern-Framework

Realization of a vertical memory for the SeCo rule learning framework

Bachelor-Thesis von Andrej Felde aus Fergana

Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Frederik Janssen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Realisierung eines vertikalen Speichers für das SeCo Regellern-Framework
Realization of a vertical memory for the SeCo rule learning framework

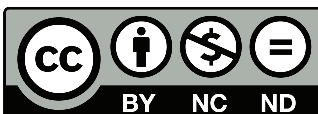
Vorgelegte Bachelor-Thesis von Andrej Felde aus Fergana

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Frederik Janssen

Tag der Einreichung:

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-12345
URL: http://tuprints.ulb.tu-darmstadt.de/1234

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 2.0 Deutschland
http://creativecommons.org/licenses/by-nc-nd/2.0/de/

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 4. August 2016

____

(A. Felde)

Inhaltsverzeichnis

Abbildungsverzeichnis II

Tabellenverzeichnis III

1. Einleitung 2
- 1.1. Motivation 2
- 1.2. Ziel der Arbeit 2
- 1.3. Aufbau der Arbeit 2

2. Grundlagen 3
- 2.1. Maschinelles Lernen 3
  - 2.1.1. Geschichte 3
  - 2.1.2. Definition 3
  - 2.1.3. Ansätze 5
  - 2.1.4. Anwendungsgebiete 6
- 2.2. Der *Separate and Conquer* - Ansatz 6
  - 2.2.1. SeCo - Framework 7
- 2.3. Assoziationsanalyse 8
  - 2.3.1. FP-Growth 10
    - 2.3.1.1. FP-Tree 10
  - 2.3.2. SD-Map 17
- 2.4. Zusammenfassung 18

3. Implementation 19
- 3.1. SeCo - EVALUATERULE 19
- 3.2. Evaluierung mit Hilfe eines SD-Tree 20
  - 3.2.1. Erweiterter *SD-Tree* 23
  - 3.2.2. Evaluierung mit einem erweiterten SD-Tree 27
- 3.3. Überischt zur SeCo-Vorgehensweise mit einem erweiterten SD-Tree 28
- 3.4. Zusammenfassung 29

4. Evaluation 30
- 4.1. Datensätze 30
- 4.2. Geschwindigkeitsanalyse 30
  - 4.2.1. Geschwindigkeitsvergleich bei EVALUATERULE 31
  - 4.2.2. Vergleich einer einzelnen Regelevaluation 33
- 4.3. Baumeigenschaften 34
- 4.4. Overhead-Anteil 36
- 4.5. Gesamtzeit bei einer anderen Heuristik 37
- 4.6. Zusammenfassung 38

5. Ausblick 39
- 5.1. Vollständige Einbindung des Baumes in *SeCo* 39
- 5.2. Ansatz einer *horizontalen* Baumevaluation 39
- 5.3. Kombination mit bestehenden Verfahren 41
- 5.4. Aktuelle Forschung 41
- 5.5. Abschluss 42

Literatur IV

A. Anhang A
- A.1. SeCoMemory; eine Implementierung des Algorithmus mit zugehöriger Baumstruktur A
- A.2. SeCoID; Erstellung und Bearbeitung eines Baumes mit Pfad-IDs G

Abbildungsverzeichnis

2.1. Möglicher Entscheidungsbaum für den Datensatz aus Tabelle 2.1 5
2.2. Visualisierung einer Decision Boundry (rot) durch eine Support Vector Machine [6] 5
2.3. Lernproblem nach [7] 7
2.4. Aufbau eines allgemeinen FP-Tree Knotens 10
2.5. Abstrakte Visualisierung eines FP-Trees (rechts) und einer Header Table (links) 11
2.6. Visualisierung des Baumes nach den ersten beiden Schritten 12
2.7. Änderungen des FP-Tree nachdem jeweils die 3., 4. und 5. Transaktion eingefügt wurden. 13
2.8. FP-Tree und *Header Table* nachdem alle Instanzen in den Baum eingefügt wurden. 13
2.9. Beispiel eines nicht minimalen FP-Tree (ohne *Node-Link*s und *Header Table*) 14
2.10. Conditional Pattern Base und Conditional FP-Tree mit Header Table von i_2 (links) und i_4 (rechts) 15
2.11. Ein *Einzelpfad* - Baum mit seinen jeweiligen Anteilen dargestellt. 16
2.12. Vergleich eines FP-Tree - Knotens zu einem SD-Tree - Knoten 18
2.13. *SD-Tree* und *Header Table* zu Datensatz 2.1 ($\xi_{abs} = 3$) 18

3.1. Vollständiger *SD-Tree* zu Datensatz 3.3 ($\xi_{abs} = 0$) 20
3.2. Visualisierung der einzelnen Schritte während der Evaluierung. Die roten Kanten stehen für die einzelnen Schritte. 21
3.3. *SD-Trees* zu allen möglichen Klassenwerten in Datensatz 3.5 (ohne *Header Table* und *Node-Links*) 23
3.4. Vergleich eines SD-Tree - Knotens zu der Erweiterung 24
3.5. Die Bäume nach den ersten zwei Schritte sowie der fertige Baum in (c) 26
3.6. Der angepasste Baum für die Regel $a_1 = 0 \Rightarrow a_4 = +$ 27

4.1. Benötigte Gesamtzeit in Sekunden (logarithmisch) 31
4.2. Dauer einer Regelevaluation anhand eines Baumes mit verschiedenen Regellängen 32
4.3. Anzahl der Pfade zwischen *Kr Vs. Kp* und *Waveform* 5 000 zur jeweiligen Tiefe des Baumes 32
4.4. Geschwindigkeitsvergleich einer Regelevaluation zwischen der originalen und baumnutzenden Variante . . 33
4.5. Baumkompression: Instanzen Vs. Pfadanzahl 34
4.6. Baumkompression: Einträge im Datensatz Vs. Knotenanzahl 35
4.7. Gesamtzeit aller Baumerstellungen und Anpassungen im Baum 36
4.8. Prozentualer Anteil des Overheads an der Gesamtzeit 36
4.9. Gesamtzeiten zur Heuristik Accuracy 37

5.1. Schema der *horizontalen* Verarbeitung zur Regel: $a_1 = 1 \land a_3 = 1 \Rightarrow a_4 = \Diamond$ 40
5.2. Regelrepräsentation in einem Chromosom nach [15] 41
5.3. Assoziationsanalyse als Service nach [9] 42

Tabellenverzeichnis

2.1. Aufbau einer Datenmenge 4
2.2. Beispiel einer Transaktionsmenge 9
2.3. Datensatz aus Kapitel 2.3 und rechts die *Frequent Items* der jeweiligen Transaktion 12

3.1. Allgemeiner Aufbau einer Konfusionsmatrix 19
3.2. Beispielhafte Konfusionsmatrix einer Regel 19
3.3. Datensatz aus Tabelle 2.1 und rechts die Zuordnung unter der Regel $(a_2 = 1 \wedge a_3 = 1) \Rightarrow a_4 = +$ 19
3.4. Konfusionsmatrix zu Regel $(a_2 = 0 \wedge a_3 = 1) \Rightarrow a_4 = +$ unter Datensatz 3.3 22
3.5. Erweiterte Datenmenge 2.1 mit 3 Klassen $(+, -, \diamondsuit)$ 22

4.1. Datenübersicht und Lernergebnisse 30
4.2. Evaluationsergebnisse zur Heuristik Accuracy 37

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Problem der Regelevaluation innerhalb des *SeCo*-Frameworks, welches eine Implementierung eines *Separate and Conquer*-Ansatzes ist. Das neue Verfahren nutzt eine erweiterte Form eines FP-Tree, welcher innerhalb des *SD-Map* - Algorithmus eingeführt wurde. Zur Verwendung innerhalb des Frameworks wird die Baumstruktur soweit verfeinert, dass für alle möglichen Klassenwerte die Werte eines Attributes direkt abgelesen werden können. Zudem komprimiert die Baumstruktur den Datensatz und soll so ein redundantes Traversieren vermeiden. Der entwickelte Baum wird soweit verändert, dass dieser nur ein einziges Mal erstellt werden muss. Des Weiteren wird eine Anpassung des Baumes konzipiert, die nicht nur effizient ist, sondern die einmalige Generierung des Baumes erst ermöglicht. Das zur Evaluation des Baumes entwickelte Verfahren, passt sich der Vorgehensweise des *SeCo*-Framework an und kann so mit verschiedenen Heuristiken und Vorgehensweisen umgehen. Abschließend wird aufgezeigt, dass ohne eine *Threshold* der Ansatz des Baumes nur für große Datensätze einen Vorteil bringen könnte. Eine Effizienzsteigerung konnte hingegen auch ohne eine *Threshold* besonders bei sehr vielen kurzen Regeln beobachtet werden.

1 Einleitung

Im Bereich der künstlichen Intelligenz gibt es verschiedene Ansätze diese zu erreichen. Ein in den letzten Jahrzehnten immer populärerer Ansatz ist im Bereich des Maschinellen Lernens zu finden. In diesem wird versucht den Computer dazu zu bringen eine Problemstellung selber zu lösen oder für den Menschen noch unbekannte Strukturen finden zu lassen. Der Ansatz geht davon aus, dass die Maschine Daten bekommt mit denen eine Theorie / Konzept entwickelt werden kann. Allgemein kann der Prozess des Lernens als eine Entwicklung verschiedener Kandidatenhypothesen auf Basis von Beispieldaten angesehen werden. Die dabei erzeugten Hypothesen werden ausgewertet und miteinander verglichen. Die entwickelte Theorie kann dabei eine Regressionsgerade wie in der Statistik sein, die für alle Eingaben einen entsprechenden Punkt auf der Geraden als Vorhersage tätigt oder eine Menge von Regeln, welche die Datenmenge beschreibt und so auch für unbekannte Fälle Vorhersagen ermöglicht.
Im Bereich des Regellernens müssen zur Erstellung einer Regelmenge viele Kandidatenregeln ausgewertet werden. Der naive Ansatz innerhalb des genutzten Frameworks geht für jede Regel durch die Trainingsdaten und zählt für jede Klasse wie viele der Beispiele abgedeckt werden. In dieser Arbeit wird eine Baumstruktur vorgestellt, die diesen Vorgang beschleunigen soll. Die Baumstruktur bedient sich dabei dem *FP-Tree* zu finden im Algorithmus *FP-Growth* [11] sowie dessen Erweiterung in *SD-Map* [5]. Der Baum wird auf das genutzte Framework *SeCo* angepasst sowie an die Problemstellung optimiert. Zudem wird ein Konzept entwickelt den erweiterten Baum zu bearbeiten und anhand dessen eine Regel zu evaluieren.

1.1 Motivation

Die Evaluation einer Regel soll durch die Nutzung einer erweiterten Form des *FP-Tree* beschleunigt werden. Die dabei entwickelte Baumstruktur soll zudem innerhalb des *SeCo*-Frameworks nutzbar sein.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer Baumstruktur, auf Basis des erweiterten FP-Tree in SD-Map, welche innerhalb des *SeCo*-Frameworks zur Regelevaluation genutzt werden kann. Die Struktur soll anschließend auf Geschwindigkeit und somit Effizienz mit der bereits vorhandenen Variante der Regelevaluation verglichen werden. Dabei soll festgestellt werden inwieweit eine Verbesserung der Zeit erreicht werden kann und welche Dateneigenschaften sich auf die Geschwindigkeit auswirken bzw. diese beeinflussen.

1.3 Aufbau der Arbeit

Die Arbeit führt zunächst alle grundlegenden Bereiche und Ansätze ein. Zu Beginn wird der Bereich des Maschinellen Lernens vorgestellt, wie sich dieser entwickelt hat und welche Ansätze sowie Anwendungen genutzt werden. Anschließend wird der *Separate and Conquer*-Ansatz von [7] vorgestellt. Es wird erläutert wie dieser mit der vorliegenden Arbeit zusammenhängt und was angepasst werden muss damit dieser einen Baum zur Regelevaluation nutzt. Abschließend wird in der theoretischen Einleitung die Assoziationsanalyse vorgestellt. Diese wird formal definiert sowie durch den Algorithmus *FP-Growth* [11] verdeutlicht. Die Veranschaulichung anhand von *FP-Growth* dient dabei gleichzeitig dem Verständnis wie der nachfolgend entwickelte Algorithmus funktioniert und welche Problemstellungen weiter betrachtet werden müssen.
Darauffolgend wird die genutzte Struktur innerhalb der Implementation schrittweise vorgestellt und gezeigt wie diese an die Problemstellung angepasst wurde. Es wird ein Konzept entwickelt und anhand eines Beispieles vorgestellt eine Regel mit Hilfe des Baumes zu evaluieren. Das dabei gezeigte Verfahren wird abschließend auf der optimierten Version des Baumes in Algorithmus 6 formal festgehalten.
Abschließend wird die Evaluation anhand des Baumes mit dem im *SeCo* genutzten Ansatz zur Regelevaluation verglichen. Es wird geschaut ob ein Geschwindigkeitsvorteil erreicht werden kann und ob anhand der Daten ersichtlich wird wie die Eigenschaften des jeweiligen Datensatzes die Evaluation beeinflussen.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe und Bereiche erläutert. Zunächst wird der Bereich des *"Maschinellen Lernens"* eingeführt. Es wird kurz gezeigt wie dieser entstanden ist, in welche Bereiche *"Maschinelles Lernen"* aufgeteilt wird und wie es konkret angewendet werden kann. Anschließend wird der *Separate and Conquer*-Ansatz vorgestellt. Es wird aufgezeigt wie dieser in den Grundzügen funktioniert und welche Verbindung zur Arbeit besteht. Darauf aufbauend wird dies abschließend in der Assoziationsanalyse vertieft, welches eine Methode darstellt Relationen in einer Datenbank zu finden und anhand von Regeln darzustellen.

2.1 Maschinelles Lernen

Bei „Machinelles Lernen“ handelt es sich um ein Forschungsgebiet, welches Computern die Fähigkeit geben soll eine Aufgabe zu erledigen ohne explizit dafür programmiert worden zu sein. Computer sollen also die notwendigen Fähigkeiten *erlernen* eine Aufgabe zu bewältigen [20]. Dabei reichen die Aufgaben von automatischer Spam-Mail-Erkennung bis hin zum autonomen Fahrzeug, welches selbstständig durch den Straßenverkehr manövriert. Dabei lernt der Computer auf Basis einer Datenmenge, welche bestehende Beispiele beinhaltet, zukünftige Fälle vorherzusagen oder Muster zu erkennen. Verschiedene Methoden wie *Support Vector Machines* oder *Neuronale Netzwerke* sind daraus u. a. entstanden und haben verschiedene Gebiete erweitert sowie in der Entwicklung vorangebracht.

2.1.1 Geschichte

Maschinelles Lernen ist aus dem Gebiet der künstlichen Intelligenz entstanden. Dabei ging es vor allem darum Computern anhand von Daten die Fähigkeiten erlernen zu lassen eine Aufgabe erfolgreich zu bewältigen, anstatt die Vorgehensweise des Computers, bezogen auf die Problemstellung, explizit zu programmieren bzw. vorzugeben. Die daraus entstanden Modelle und Vorgehensweisen, entstammen dabei auch aus anderen Gebieten wie der Statistik, Psychologie oder Biologie und haben diese Gebiete ebenfalls erweitert und in der Entwicklung vorangebracht [4, 13, 10, 21]. Bekannte Modelle sind dabei u. a. Neuronale Netzwerke, die auf dem biologischen Vorbild des Gehirns basieren oder auch Support Vector Machines, die bei einer Klassifizierung oder der Regression genutzt werden können. Neuronale Netzwerke wurden dabei bis Ende der 1990er Jahre vernachlässigt und erst durch die gesteigerte Leistung von Computern und der Entwicklung von *Backpropagation* immer stärker eingebunden und genutzt. In den 1990er Jahren entwickelte sich der Bereich des Maschinellen Lernens zu einem komplett eigenen Forschungsgebiet und begann durch die Verschiebung zur Statistik sowie der Wahrscheinlichkeitstheorie aufzublühen. Ein weiterer Faktor, der die Entwicklung beschleunigte, war die Verbreitung des Internets und die daraus hervorgegangenen Datenmengen, die sowohl genutzt als auch zunächst verarbeitet werden mussten [12].

2.1.2 Definition

Eine formale Definition nach [16] lautet:

„A computer program is said to learn from *experience E* with respect to some class of *tasks T* and *performance measure P*, if its performance at tasks in T, as measured by P, improves with experience E.“

Auf die oben genannten Beispiele eines autonomen Fahrzeuges oder der Spam-Mail-Erkennung angewendet, kann eine mögliche Einteilung wie folgt aussehen.

Spam-Mail-Erkennung:

- *Task T:* Klassifizierung von Mail in Spam- und nicht Spam-Mail.
- *Performance Measure P:* Prozentzahl richtig klassifizierter Mail.
- *Training Experience E:* Eine Datenmenge mit bereits klassifizierten E-Mails.

Autonomes Fahrzeug:

- *Task T:* Fahren auf der Straße durch Nutzung von Sensoren.
- *Performance Measure P:* Messung bewältigter Distanzen ohne einen Fehler.
- *Training Experience E:* Sensoraufzeichnungen eines menschlichen Fahrers.

Diese Beispiele haben unterschiedliche Eigenschaften und können je nach Beschaffenheit der Daten verschieden erlernt werden. Daher wird Maschinelles Lernen in drei Gebiete eingeteilt.

- *Überwachtes Lernen (Supervised Learning)*

  Es werden Daten genutzt, welche zu gegebenen Eingaben die gewünschten Ausgaben vorgegeben. Das Ziel ist eine Funktion zu erlernen, bei der die Eingaben auf die Ausgaben abgebildet werden.

- *Unüberwachtes Lernen (Unsupervised Learning)*

  Die genutzten Daten geben zu entsprechenden Eingaben keine gewünschten Ausgaben vor. Das Ziel ist dabei (verdeckte) Strukturen oder Eigenschaften zu finden.

- *Bestärkendes Lernen (Reinforcement Learning)*

  Der Lernalgorithmus *"interagiert"* in diesem Fall mit der Umgebung. Dabei werden indirekt Züge belohnt, die das Programm dem Ziel näher bringen. Vorstellen kann man sich dies wie bei dem Menschen, der eine bestimmte Tätigkeit erlernt.

Neben den genannten drei Gebieten gibt es noch Zwischengebiete wie das Teilüberwachte Lernen (Semi-Supervised Learning) oder das Gebiet des Aktiven Lernens (Achtive Learning). Nach dieser Einteilung könnte das Problem der Spam-Mail-Erkennung, je nach Datenaufbau und gewünschtem Ergebnis, sowohl zum Überwachten als auch Unüberwachten Lernen gehören. Daher kann ebenfalls basierend auf dem gewünschtem Output eingeteilt werden. Die Bekanntesten sind dabei folgende:

- Klassifikation: Die Eingaben sind in zwei oder mehr Klassen eingeteilt. Der Lernalgorithmus lernt dabei Eingaben auf die Klassen abzubilden und so bei neuen Eingaben die Klasse vorherzusagen.
- Regression: In Gegensatz zur Klassifikation sind die Ausgaben bei der Regression kontinuierlich. So können zu Eingaben beliebig viele Ausgaben vorhergesagt werden. Ein Beispiel dafür wäre die Preisbestimmung eines Hauses anhand gegebener Eigenschaften.
- Clusteranalyse: Es sollen Gruppen bzw. *Cluster* bestimmt werden. Diese sind dabei anders als bei der Klassifikation vorher nicht bekannt. Es sollen also verborgene bzw. unbekannte Strukturen entdeckt werden.

Auf das zuvor genannte Lernproblem der Spam-Mail-Erkennung angewendet, würde dies einem Klassifikationsproblem entsprechen. Die Mails sollen auf zwei Klassen abgebildet werden. Es wird nur zwischen Spam und Nicht-Spam unterschieden.

a_1	a_2	a_3	a_4
0	1	1	+
0	0	1	+
0	1	1	−
0	1	0	−
0	0	1	+
1	0	0	−
0	1	0	−
0	1	1	+
0	0	1	+
1	0	1	+

Tabelle 2.1.: Aufbau einer Datenmenge

In dieser Arbeit ist die *Training Experience e* eine Datenmenge bereits existierender Beispiele. Diese kann eine unterschiedliche Anzahl an Beispielen, auch Instanzen genannt, beinhalten. Eine Instanz besteht aus $n \in \mathbb{N}$ Attributen, wobei ein Attribut a_i eine Eigenschaft des vorliegenden Problems ist. In der Instanz ist dem jeweiligen Attribut ein Wert zugeordnet. In Tabelle 2.1 ist eine Datenmenge mit zehn Instanzen und vier Attributen a_i dargestellt. Die Zeilen der Tabelle stehen für die Instanzen und die Reihen für Attribute. Für die erste Instanz ist dem Attribut $a_1 = 0$ zugeordnet. Das letzte Attribut (in diesem Fall a_4) ist im Normalfall das sogenannte Klassenattribut, welches der Computer erlernt vorherzusagen. Es ist zu beachten, dass jedes Attribut a_i das Klassenattribut sein kann aber in den verwendeten Beispielen und in den meisten Fällen dies das letzte Attribut ist. Nachdem die Vorhersage erlernt wurde, kann diese auf unterschiedliche Art und Weise ausgedrückt werden. Bekannte Ansätze sind dabei Regelmengen, Neuronale Netzwerke oder ein Entscheidungsbaum.

2.1.3 Ansätze

Bekannte Ansätze sind künstliche Neuronale Netzwerke, Entscheidungsbäume, *Support Vector Machines* oder die Assoziationsanalyse. Bei Entscheidungsbäumen wird ein Baum erstellt, bei dem Blätter für die Klassen stehen, Knoten für die Attribute und die Kanten für die jeweilige Entscheidung welcher Wert für das Attribut gilt. In Abbildung 2.1 ist ein möglicher Entscheidungsbaum für den Datensatz aus Tabelle 2.1 gegeben. Im Baum kann ablesen werden, dass bei a_1 die Entscheidung zwischen 0 und 1 möglich ist. Es ist zu beachten, dass bei anderen Datenmengen auch Wertebereiche bei den Kanten stehen können. Wenn für $a_1 = 0$ gilt, wird daraus folgend geschaut, ob für das Attribut $a_2 = 0$ oder $a_2 = 1$ gilt. Ist $a_2 = 0$, wird für das Klassenattribut direkt $a_4 = 1$ vorhergesagt. Ist dies nicht der Fall, wird der Vorgang mit dem restlichen Baum wiederholt, bis ein Blatt erreicht wird. Der Entscheidungsbaum kann dabei sowohl für Klassifikationen als auch für Regression genutzt werden. Bei Regression müssen die Blätter entsprechend alle möglichen Wertebereiche abdecken. Ein anderer Ansatz sind *Support Vector Machines*. Diese versuchen bei einer Klassifikation die Distanz der Trennlinie zwischen den Klassen zu maximieren. Ein mögliches Beispiel sieht man in Abbildung 2.2. In dieser Abbildung sieht man die zwei Klassen 0 und 1. Das Klassenattribut ist in diesem Fall durch y gegeben und die Kreise markieren die nächstmöglichen Datenpunkte zwischen den Klassen. Durch die Maximierung des Abstand wird versucht eine möglichst genaue Vorhersage zu treffen und die Klassen so optimal wie möglich von einander zu trennen.

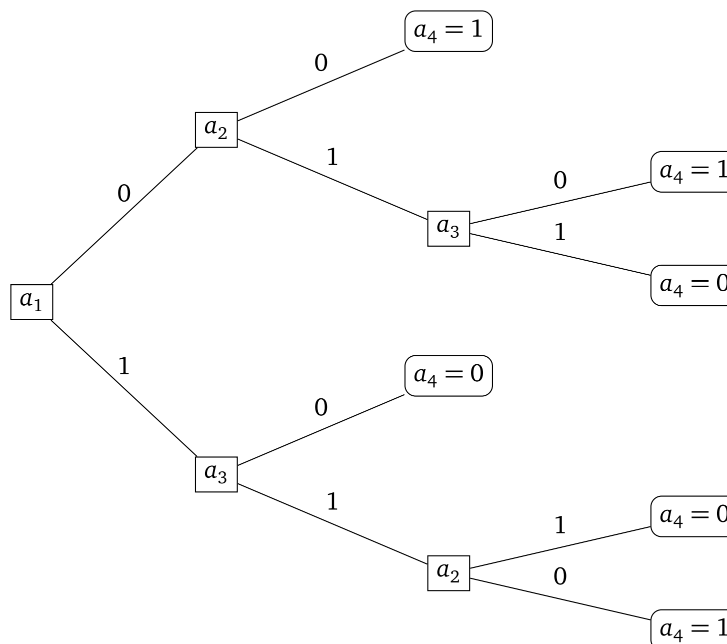


Abbildung 2.1.: Möglicher Entscheidungsbaum für den Datensatz aus Tabelle 2.1

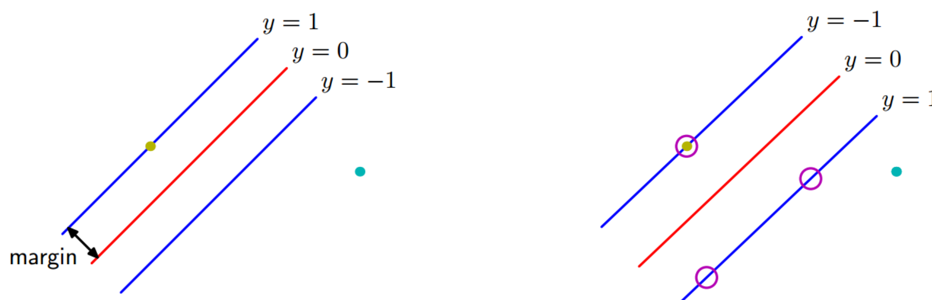


Abbildung 2.2.: Visualisierung einer Decision Boundry (rot) durch eine Support Vector Machine [6]

Ein weiterer Ansatz ist die Assoziationsanalyse. Bei dieser werden Zusammenhänge durch Regeln dargestellt. Dabei wird versucht möglichst *gute* Regeln zu erstellen. Das Ergebnis ist eine Regelmenge, die sowohl für Regression als auch in einer Klassifikation genutzt werden kann. Allgemein gilt, dass die entstandenen Regeln sowohl beim überwachten als auch unüberwachten Lernen genutzt werden können. In Kapitel 2.3 wird die Assoziationsanalyse näher erläutert.

2.1.4 Anwendungsgebiete

Für Maschinelles Lernen gibt es unzählige Anwendungsgebiete. Bekannte Beispiele sind Anwendungen in der Bioinformatik, bei der Vorhersagen zu Proteinen getroffen werden können oder Anwendungen in der Medizin. In der Medizin könnte versucht werden automatisiert festzustellen, inwieweit ein Tumor, anhand von Größe, Farbe, Gewicht, Struktur oder anderer Eigenschaften, gutartig oder bösartig ist. Weitere bekannte Anwendungsgebiete sind die Gesichts- oder Schrifterkennung. In diesem Fall werden besonders Neuronale Netzwerke eingesetzt, da sie in der Vergangenheit gute Ergebnisse geliefert haben.
Inzwischen existieren unzählige Open-Source Software als auch kommerzielle Software. Einige bekannte Software-Suites sind dabei:

- MATLAB
- GNU Octave
- R
- Weka
- RapidMiner
- Google Prediction API

In dieser Arbeit wird *SeCo* genutzt. Dies ist eine Software Suite der *Knowledge Engineering Group* der TU Darmstadt. Diese baut auf *Weka* auf und nutzt daher die von *Weka* vorgegebenen Strukturen und Dateneigenschaften.

2.2 Der *Separate and Conquer* - Ansatz

Nachfolgend wird der *Separate and Conquer* - Ansatz eingeführt, sowie Abhängigkeiten zur Arbeit aufgezeigt. Für eine vollständige Einführung wird auf *"Separate-and-Conquer Rule Learning"* [7] verwiesen. Beim *Separate and Conquer* - Ansatz, auch als *Covering* bekannt, handelt es sich um eine Familie von Algorithmen, die im Allgemeinen nach zwei Schritten vorgehen. Im ersten Schritt (Separate-Schritt) wird zunächst eine Regel gesucht, die einen Teil der Datenmenge erklärt. Dieser Teil wird dann von der restlichen Datenmenge *"separiert"* / abgetrennt. Dies wird anschließend im zweiten Schritt (Conquer-Schritt) solange auf der restlichen Datenmenge wiederholt, bis keine Beispiele übrig bleiben. Dies sorgt dafür, dass jede Instanz mindestens von einer Regel abgedeckt wird.
Zum einfacheren Verständnis wird zunächst die Problemstellung in Abbildung 2.3 nach [7] vorgestellt. Wie bereits beim Maschinellen Lernen eingeführt, sind für die hier genutzten Ansätze eine Datenmenge mit positiven und negativen Beispielen notwendig. Die Beispiele bestehen aus mehreren Attributen (den Eigenschaften eines Beispieles). Ein vorgegebenes *Zielkonzept* besagt was vorhergesagt bzw. beschrieben werden soll. Das optionale Hintergrundwissen, kann u.a. durch einen Experten beigesteuert werden. Diese Beschreibung kann in Tabelle 2.1 beispielhaft entnommen werden. Das gesuchte Ergebnis ist eine Regelmenge zur Vorhersage. Die Ermittelung der Regelmenge kann auf unterschiedliche Art und Weise erreicht werden. Eine bekannte Alternative zu dem hier vorgestellten Ansatz sind die zuvor eingeführten Entscheidungsbäume, deren Popularität auf ihrer Effizienz beruht. Nachteile der Entscheidungsbäume sind hingegen u.a. ihre Komplexität und daraus folgende Schwierigkeiten diese zu verstehen [7, S.6].
Da das verwendete Framework auf dem *Separate and Conquer* - Ansatz von [7] aufbaut, wird nachfolgend die Funktionsweise angedeutet. Außerdem werden die Stellen aufgezeigt, die diese Arbeit anpasst. Der am Anfang des Kapitels vorgestellte naive Ansatz von Separate-and-Conquer wird in Algorithmus 1 erweitert. Dieser liefert zwar bereits eine vollständige und konsistente Lösung, beachtet aber nicht mögliche Störquellen *(Noise)* innerhalb der Daten. Dies kann ohne Anpassung dazu führen, dass sich die Regelmenge zu stark an diese Störquellen anpasst und so unnötig komplexe und weniger vorhersagende Regeln erzeugt werden. Daher werden *"Stop-Kriterien"* sowie Post-Processing genutzt um simplere Regelmengen zu entwickeln, die zwar nicht mehr vollständig sind aber dafür bessere Vorhersagen auf unbekannten Daten liefern [7, S.8].
Der erweiterte Algorithmus 1 ermöglicht das Nutzen verschiedener bewährter Ansätze und Algorithmen. So kann u.a. sowohl ein *Top-Down*-Ansatz als auch ein *Bottom-Up*-Ansatz verwendet werden. Dies wird durch die verschiedenen Funktionen / Prozeduren innerhalb des Algorithmus ermöglicht. Der Algorithmus 1 SEPARATEANDCONQUER startet mit einer leeren Theorie (einer leeren Regelmenge). Solange die Regelmenge positive Beispiele hat, wird mit FINDBESTRULE die bestmögliche Regel herausgesucht, die einen Teil der Datenmenge abdeckt. Zur Bestimmung wie gut eine Regel ist, wird eine Heuristik verwendet. In der Regel ist der heuristische Wert höher / besser, wenn eine Regel mehr positive und weniger negative Beispiele abdeckt. Stoppt das RULESTOPPINGCRITERION nicht die Schleife, werden alle abgedeckten Beispiele

Gegeben:

- Ein Zielkonzept
- negative und positive Beispiele
- Die Beispiele sind durch verschiedene Eigenschaften beschrieben
- Optionales Hintergrundwissen

Gesucht:

- Eine simple Menge von Regeln, die zwischen positiven und negativen Beispielen unterscheidet. Die Regeln sollen dabei auch auf vorher unbekannte Beispiele anwendbar sein.

Abbildung 2.3.: Lernproblem nach [7]

separiert und die Regel zur Regelmenge (Theorie) hinzugefügt. Abschließend ergibt sich die Möglichkeit die Regelmenge / Theorie mit Post-Processing zu bearbeiten und so u.U. zu vereinfachen bzw. zu verbessern.
Zur Bestimmung der *besten* Regel wird FINDBESTRULE in Algorithmus 1 genutzt, welches anhand einer gegebenen Heuristik nach der bestmöglichen Regel sucht. Die Funktion INITIALIZERULE dient dazu erste Regeln nach einem vorgegebenen Prinzip zu initialisieren. Anschließend werden die Regeln durch die Funktion EVALUATERULE evaluiert. Die Regel mit dem besten heuristischen Wert wird als *BestRule* gespeichert und in die Menge der Kandidatenregeln *Rules* eingefügt. Solange die Menge *Rules* nicht leer ist, werden innerhalb der Schleife Kandidatenregeln mit SELECTCANDIDATES erzeugt. Diese werden aus der Regelmenge *Rules* entfernt und mit REFINERULES verfeinert. Daraus werden alle möglichen Verfeinerungen der Regel erzeugt und evaluiert. Wenn die Funktion STOPPINGCRITERION dies nicht unterbindet, wird die neue Regel *NewRule* in die Regelmenge *Rules* einsortiert und bei einem höheren heuristischen Wert zur neuen *BestRule*. Abschließend wird in der Schleife mit FILTERRULES eine Teilmenge der Regeln ausgewählt, die in den nächsten Iterationen genutzt wird. Am Ende wird *BestRule* als beste gefundene Regel zurückgegeben.
Durch verschiedene Wahl der Funktionen INITIALIZERULE, REFINERULE, SELECTCANDIDATES, FILTERRULES und EVALUATERULE, können Probleme (Search Bias, Language Bias) bei der Erstellung der Regel vermieden werden. Die Funktionen RULESTOPPINGCRITERION, STOPPINGCRITERION und POSTPROCESS können abschließend genutzt werden, um *Overfitting* zu vermeiden [7].
In dieser Arbeit soll in die Funktion EVALUATERULE eingegriffen werden. Der heuristische Wert wird auf der genutzten Version von *SeCo* durch wiederholtes iterieren über die Datenmenge bestimmt. Dies soll in dem hier genutzten Ansatz durch einen angepassten *FP-Tree* vermieden werden. Eine detaillierte Beschreibung der aktuellen Vorgehensweise von *SeCo* in EVALUATERULE und einem Vergleich zum neuen Ansatz wird in Kapitel 3.1 beschrieben.

2.2.1 SeCo - Framework

Beim *SeCo* - Framework (***Se**parate and **Co**nquer*) handelt es sich um eine Implementation des Algorithmus 1 durch die Knowledge Engineering Group der TU Darmstadt. Die hier zum Vergleich verwendete Version geht bei der Evaluation jeder Regel wiederholt durch den Datensatz.

Algorithm 1 Allgemeiner *Separate and Conquer* - Algorithmus nach [7, S.10]

```
 1: procedure SEPARATEANDCONQUER(Examples)
 2:     Theory = ∅
 3:     while POSITIVE(Examples) ≠ ∅ do
 4:         Rule = FINDBESTRULE(Examples)
 5:         Covered = COVER(Rule, Examples)
 6:         if RULESTOPPINGCRITERION(Theory, Rule, Examples) then
 7:             exit while loop
 8:         Examples = Examples \ Covered
 9:         Theory = Theory ∪ Rule
10:     Theory = POSTPROCESS(Theory)
11:     return Theory

 1: procedure FINDBESTRULE(Examples)
 2:     InitRule = INITIALIZERULE(Examples)
 3:     InitVal = EVALUATERULE(InitRule)
 4:     BestRule = <InitVal, InitRule>
 5:     Rules = {BestRule}
 6:     while Rules ≠ ∅ do
 7:         Candidates = SELECTCANDIDATES(Rules, Examples)
 8:         Rules = Rules \ Candidates
 9:         for all Candidate ∈ Candidates do
10:             Refinements = REFINERULE(Candidate, Examples)
11:             for all Refinement ∈ Refinements do
12:                 Evaluation = EVALUATERULE(Refinements, Examples)
13:                 unless STOPPINGCRITERION(Refinements, Examples)
14:                     NewRule = <Evaluation, Refinement>
15:                     Rule = INSERTSORT(NewRule, Rules)
16:                     if NewRule > BestRule
17:                         BestRule = NewRule
18:         Rules = FILTERRULES(Rules, Examples)
19:     return BestRule
```

2.3 Assoziationsanalyse

Die Assoziationsanalyse bezeichnet den Vorgang bei dem ein Zusammenhang zwischen Daten oder Datenmengen gesucht bzw. aufgedeckt wird. Dieser wird anschließend in entsprechenden Regeln festgehalten. Die dabei entstandenen Regeln sollen möglichst aussagekräftig sein. Zur Bestimmung der Aussagekraft werden verschiedene Maßeinheiten genutzt, da nicht jedes Maß gleich gut geeignet ist eine Aussage zu treffen. Ein weiteres Problem ist die Interpretation der Werte zum jeweiligen Maß [14]. Für die Maße der Aussagekraft gibt es bisher keine entsprechend weitverbreitete Einigung über deren formale Definition. Die bisher vorliegenden Definitionen können in Konzepte wie Prägnanz, Abdeckung, Zuverlässigkeit, Eigenheit, Diversität, Neuheit und Andere vereinigt werden. Diese werden bei [8, S.2] im Detail beschrieben. Die bekanntesten und meist verwendeten Maßstäbe zur Auswahl einer Regel sind der *Support* und die *Confidence*.

Die formale Definition der Assoziationsanalyse nach [1] besagt:

> Sei $I = \{i_1, i_2, \dots, i_m\}$ eine Menge von m binären Attributen, welche *Items* genannt werden.
> Sei $T = \{t_1, t_2, \dots, t_n\}$ eine Menge von Transaktionen, welche als *Database* bezeichnet wird.

Jede Transaktion $t_j \in T$ mit $j \leq n$ wird als binärer Vektor repräsentiert. Es gilt $t_j[k] = 1$, wenn die Transaktion das *Item* i_k mit $k \leq m$ beinhaltet und $t_j[k] = 0$, wenn dies nicht der Fall ist. Sei des Weiteren X eine Menge von *Items* in I. Eine Transaktion t_j befriedigt (*statisfies*) X, wenn für alle *Items* $i_k \in X$, $t_j[k] = 1$ gilt. Darauf aufbauend sind Assoziationsregeln nach [1] eine Implikation der Form $X \Rightarrow i_j$, mit $i_j \in I$ und $i_j \notin X$. D. h., wenn die *Items* in der Menge X gegeben sind, dann ist auch das *Item* i_j vorhanden. Somit folgt auch warum i_j nicht in der Menge X und gleichzeitig auf der rechten Seite der Implikation stehen kann bzw. sein sollte. Eine Aussage der Form *"Wenn Item i_j vorhanden ist, dann ist i_j vorhanden."* hat keine Aussagekraft hat.

Transaktions-ID	i_1	i_2	i_3	i_4
0	0	1	1	1
1	0	0	1	1
2	0	1	1	0
3	0	1	0	0
4	0	0	1	1
5	1	0	0	0
6	0	1	0	0
7	0	1	1	1
8	0	0	1	1
9	1	0	1	1

Tabelle 2.2.: Beispiel einer Transaktionsmenge

In Tabelle 2.2 ist eine Datenbank mit zehn Transaktionen T_j, wobei t_1 für die erste Transaktion mit Transaktions-ID 0 steht. Jeder Transaktionsvektor t_j besteht aus vier Einträgen für das jeweilige *Item*. Die Menge der Attribute wäre in diesem Beispiel $I = \{i_1, i_2, i_3, i_4\}$. Für die Transaktionsmenge T und die darin beinhalteten Vektoren ergeben sich folgende Mengen:

$$T = \begin{Bmatrix} [0 & 1 & 1 & 1], \\ [0 & 0 & 1 & 1], \\ \vdots & \vdots & \vdots & \vdots \\ [0 & 0 & 1 & 1], \end{Bmatrix}, \qquad T_2 = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}, \qquad T_2[3] = 1$$

Mögliche Regeln der Form nach [1] für das Beispiel aus Tabelle 2.2 wären:

$$\{i_2 = 1, i_3 = 0\} \Rightarrow i_4 = 1 \text{ oder} \tag{2.1}$$
$$\{i_1 = 0, i_2 = 1, i_4 = 0\} \Rightarrow i_3 = 1 \tag{2.2}$$

Dabei ist die Menge X für Regel (2.1) als $X = \{i_2 = 1, i_3 = 0\}$ gegeben. Die Regel (2.1) sagt dabei aus, dass wenn *Item* i_2 vorhanden ist und *Item* i_3 nicht, *Item* i_4 vorhanden sein soll bzw. ist. Für die restlichen *Items* i_1, i_3 trifft diese Regel keine Aussage, d. h. diese können sowohl vorhanden sein als auch nicht.
Zur Erzeugung der entsprechenden Regeln werden verschiedene Algorithmen verwendet. Der bekannteste Algorithmus ist der *Apriori*-Algorithmus, welcher auf Basis einer Breitensuche den *Support* der Itemmengen bestimmt und daraus, durch Ausnutzung der Anti-Monotonie Apriori Heuristik, entsprechende Kandidaten erzeugt. Die Anti-Monotonie Apriori Heuristik besagt, dass wenn ein *Pattern* der Länge k nicht häufig (*frequent*) vorkommt, dann ist das *Super-Pattern* mit der Länge $k+1$ ebenfalls nicht häufig (*frequent*) [2]. Die Funktionsweise des Algorithmus geht dabei auf die Idee zurück, dass iterativ *Items* der Länge $k+1$ aus *frequent Items* der Länge k erzeugt werden, wobei $k \geq 1$ gilt. Die neu erzeugten *Items* der Länge $k+1$ werden dann auf ihre Häufigkeit überprüft und nur die, die eine Mindesthäufigkeit vorweisen werden genutzt um wiederum größere Super-Pattern zu erzeugen, bis alle möglichen (*frequent*) Pattern erzeugt wurden. Dabei leiden alle *Apriori*-ähnlichen Algorithmen unter zwei nicht trivialen Kosten [11]:

- Die Erzeugung möglicher frequent Pattern ist äußerst kostspielig. Bei Datensätzen mit 10^4 frequent Items der Länge 1, muss der Apriori-Algorithmus mehr als $10^7 \cdot \text{Patternlänge} - 2$ mögliche Pattern erzeugen und auf die Häufigkeit testen. Besonders auffällig werden die Kosten bei größeren Pattern. Um ein Pattern der Länge 100 zu entdecken, müssen $2^{100} - 2 \approx 10^{30}$ erzeugt und geprüft werden.

- Zudem ist es sehr mühsam die Datenmenge wiederholt durchzugehen um festzustellen, ob ein Pattern frequent ist.

Daher wurde bei [11] versucht eine Methode zu entwickeln, die diese Kosten eliminiert bzw. verringert. Daraus ist die Datenstruktur des FP-Tree (***F****requent* ***P****attern* - ***T****ree*) entstanden, welche im zudem entwickelten Algorithmus *FP-Growth* genutzt wird. Daher wird in dieser Arbeit eine Abwandlung des FP-Growth-Algorithmus genutzt, um entsprechende Regeln zu erzeugen. Die genaue Funktionsweise von FP-Growth wird im Folgenden Kapitel 2.3.1 erläutert. Man beachte, dass beide Algorithmen nur einen Teil der Arbeit erledigen, da sie nur die entsprechenden Itemmengen erzeugen. Daraus folgend müssen noch die Regeln generiert werden.

2.3.1 FP-Growth

Bei FP-Growth (kurz für ***F**requent **P**attern*-Growth) handelt es sich um ein Algorithmus, der von Han et al. entwickelt und in [11] veröffentlicht wurde. Darin wird auch der dazu korrespondierende FP-Tree eingeführt, welcher eine Erweiterung eines Präfix-Baumes ist. Das besondere an FP-Growth ist die Kostenverringerung, die in erster Linie durch drei Methoden erreicht wird:

1. Große Datenmengen werden in einer Datenstruktur, dem FP-Tree, komprimiert. So kann ein wiederholtes durchgehen durch die Daten vermieden werden.
2. Die vorgestellte Methode den Baum durchzugehen vermeidet die kostspielige Generierung großer Mengen von Kandidatenmengen.
3. Der *Divide-and-Conquer* Ansatz von FP-Growth, verkleinert den Suchraum.

Folgend wird zunächst die Datenstruktur des FP-Tree vorgestellt. Anschließend wird die Funktionsweise von FP-Growth erläutert und anhand eines Beispiels verdeutlicht.

2.3.1.1 FP-Tree

Beim *FP-Tree* handelt es sich um eine Baumstruktur, bestehend aus einem Wurzelknoten, den FP-Knoten und einer *Header Table*. Der Wurzelknoten ist immer ein leerer Knoten. Dieser dient nur als Einstiegspunkt in den Baum und speichert sonst keine weiteren Informationen. Die restlichen FP-Knoten sind die konkreten Konten des Baumes, welche die Informationen des Datensatz verarbeitet darstellen. Ein FP-Knoten besteht aus vier Felder.

FP-Knoten Felder:

- *item-name*: Speichert welches Item durch den Knoten repräsentiert wird.
- *count*: Speichert die Anzahl an Transaktionen in der Datenmenge, welche durch den Pfad vom Wurzelknoten bis zum Knoten selbst in der Datenmenge vorhanden sind.
- *node-link*: Ein Verweis innerhalb des Baumes auf den nächsten Knoten, der das gleiche Item repräsentiert.
- *children*: Verweise auf die *Kind-Knoten*. Die Anzahl der Kind-Knoten ist durch die Anzahl der Items, die noch nicht im Pfad durch einen Knoten repräsentiert werden, beschränkt.

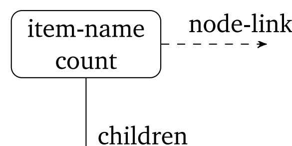


Abbildung 2.4.: Aufbau eines allgemeinen FP-Tree Knotens

In Abbildung 2.4 ist eine Visualisierung eines allgemeinen FP-Tree-Knotens dargestellt. Beim *node-link* ist zu beachten, dass dieser entweder auf einen weiteren Knoten zeigt, der das gleiche Item repräsentiert oder *null* ist, also keinen Verweis hat! Das Feld *count* wird mit 1 initialisiert, wächst monoton und ist stets positiv.
Neben dem FP-Tree selbst wird eine *Header Table* geführt. Diese ist ein Verweis zu allen Items, die im Baum vorhanden sind. Die darin gespeicherten Verweise zu den Knoten im Baum können als Startpunkt einer Linked-List interpretiert werden, die alle Knoten beinhaltet die das gleiche Item repräsentieren. Die Verweise im Feld *node-link* der einzelnen FP-Tree-Knoten sind die Referenzen auf die restlichen Elemente der Liste. Die Header Table wird im späteren Verlauf genutzt alle Counts eines Items schneller wiederzufinden.
In Abbildung 2.5 ist links eine Header Table und rechts ein FP-Tree zu sehen. Die Header Table verlinkt zu drei Knoten, welche in einem konkreten Fall alle verschiedene Items repräsentieren würden. In diesem Fall sind die Items als auch der Count in den Knoten nicht näher genannt. Es wird nur verdeutlicht, dass es drei verschiedene Items V_1, V_2 und V_3 existieren. Von der Header Table aus können alle Knoten mit dem jeweiligen Item erreicht werden. Dies wird ersichtlich wenn man bei der Header Table von V_3 aus startet und dem *node-link* im jeweiligen Knoten folgt. So erreicht man zwangsläufig den Knoten ganz rechts mit der Beschriftung V_3. Da keine weiteren Knoten wie V_1 und V_2 existieren, sind die *node-link*-Felder in diesen beiden Knoten *null*.

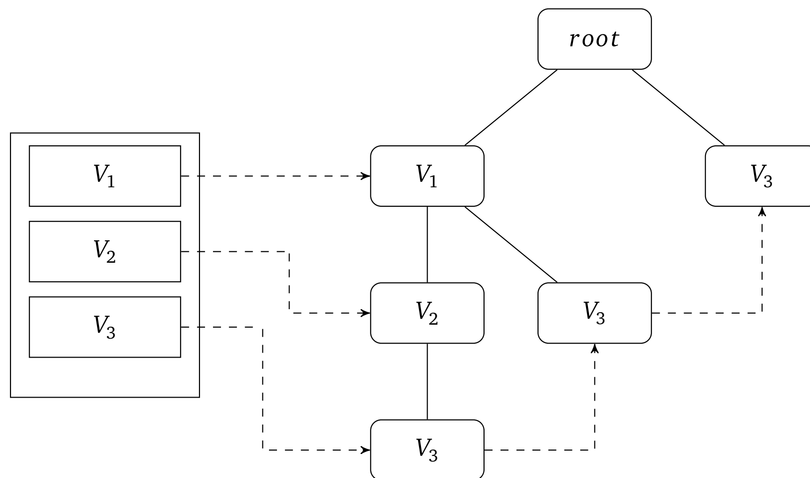


Abbildung 2.5.: Abstrakte Visualisierung eines FP-Trees (rechts) und einer Header Table (links)

Der Algorithmus FP-Growth besteht aus zwei Schritten. Zunächst muss der Baum anhand einer Datenmenge erzeugt werden. Anschließend werden die *Frequent Pattern* durch traversieren des Baumes generiert bzw. gefunden. Die Erstellung des Baumes erfolgt dabei nach [11] wie folgt:

Algorithm 2 FP-tree construction

```
 1: procedure CREATEFPTREE(transaction database D, minimum support ξ)
 2:     Scan database D once and collect F, the set of frequent items, and the support of each item in D
 3:     Sort F in support-descending order as FList, the list of frequent items (support ≥ ξ)
 4:
 5:     Create the empty root of FP-Tree T
 6:     for each transaction Trans in D do
 7:         Select the frequent items (support ≥ ξ) in Trans and sort them according to the order of FList
 8:         Let the sorted frequent-item list in Trans be [p|P], where p is the first element and P is the remaining list
 9:
10:         N = root of the FP-Tree T
11:         do
12:             if N has a child node C and C.item-name == p.item-name then
13:                 incerement C's count by 1
14:             else
15:                 create a new node C, with its count initilized to 1,
16:                 set its parent link to N,
17:                 link it to the nodes with the same item-name via node-link
18:             N = C
19:             remove the first element p from the list [p|P]
20:         while P is nonempty
```

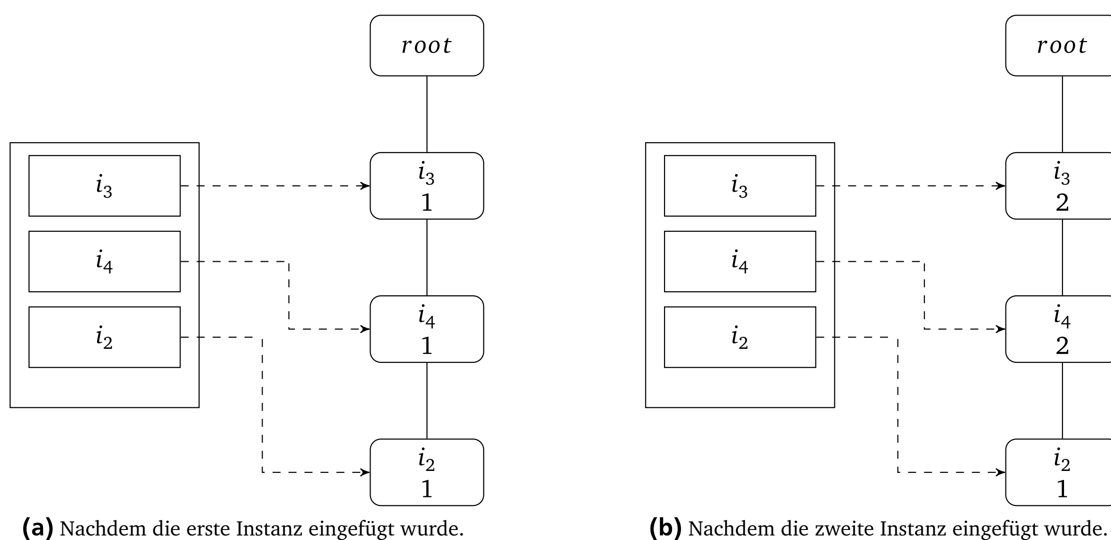
Zur Veranschaulichung wird ein FP-Tree anhand des gegebenen Algorithmus 2 `createFPTree` erzeugt. Als Datenmenge wird Tabelle 2.2 aus der Einführung zur Assoziationsanalyse und ein *Support-Threshold* ξ von 30% genutzt.

Zunächst wird das Auftreten von jedem Item gezählt. Durch traversieren der Datenmenge erhält man $i_1 = 2, i_2 = 5, i_3 = 7$ und $i_4 = 6$. Daraus wird die sortierte Liste *FList* erstellt. Dazu muss der *Support* des Items $\geq$ 30% sein. Bei einer Datenmenge mit 10 Transaktionen bzw. Instanzen ergibt sich ein absoluter *Support* von 3. Somit ist die sortierte Liste *FList* gegeben durch $[i_3, i_4, i_2]$. Das Item i_1 entfällt, da es nur einen Support von 20% hat bzw. nur zweimal in der Datenmenge vorkommt. Anschließend muss der Datensatz erneut durchgegangen werden, um die *Frequent Items* in der Reihenfolge der sortierten Liste *FList* in den Baum einzufügen.

In Tabelle 2.3 steht für jede Transaktion in der Spalte „sortierte *Frequent Items*“ welche Items und in welcher Reihenfolge diese in dem Baum eingefügt werden. Für die erste Transaktion folgt daher, dass alle Items bis auf i_1 eingefügt werden. Zudem ist die Reihenfolge für die erste Transaktion $i_3 \rightarrow i_4 \rightarrow i_2$. Da der Baum zu dem Zeitpunkt noch komplett leer ist, werden der Reihe nach die Knoten mit dem jeweiligen Item eingefügt.

i_1	i_2	i_3	i_4	sortierte *Frequent Items*
0	1	1	1	i_3, i_4, i_2
0	0	1	1	i_3, i_4
0	1	1	0	i_3, i_2
0	1	0	0	i_2
0	0	1	1	i_3, i_4
1	0	0	0	-
0	1	0	0	i_2
0	1	1	1	i_3, i_4, i_2
0	0	1	1	i_3, i_4
1	0	1	1	i_3, i_4

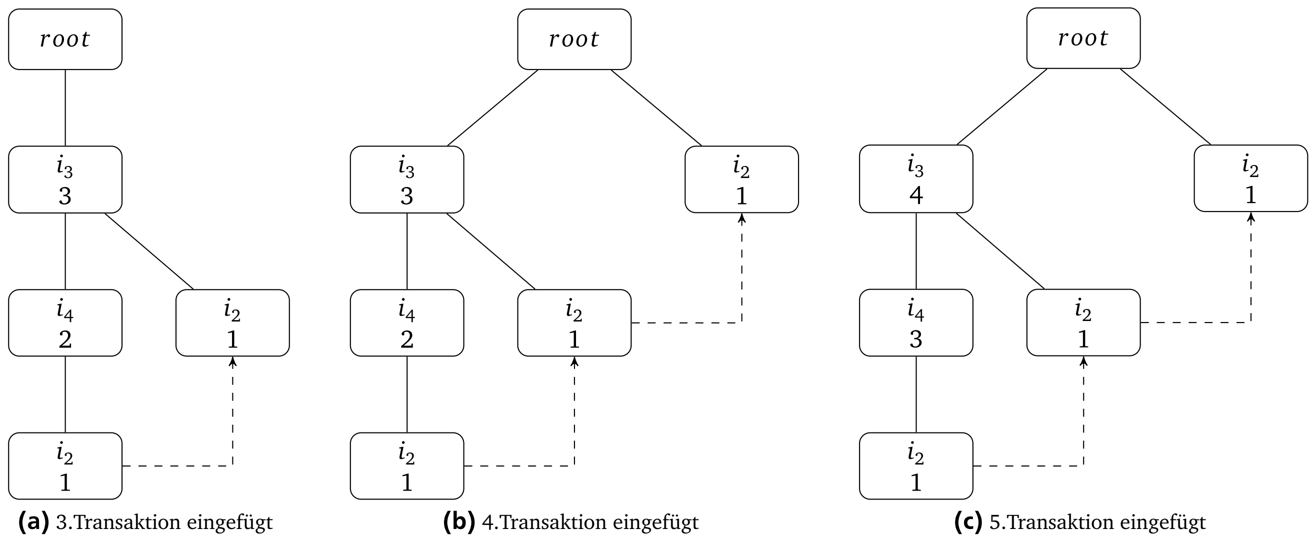
Tabelle 2.3.: Datensatz aus Kapitel 2.3 und rechts die *Frequent Items* der jeweiligen Transaktion



(a) Nachdem die erste Instanz eingefügt wurde. **(b)** Nachdem die zweite Instanz eingefügt wurde.

Abbildung 2.6.: Visualisierung des Baumes nach den ersten beiden Schritten

In Abbildung 2.6a ist das Ergebnis nachdem die erste Transaktion eingefügt wurde. Da jeder dieser Knoten neu war, wurde auch die *Header Table* mit den entsprechenden Referenzen gefüllt. Analog wird dies für die restlichen Transaktionen wiederholt.
Als nächstes muss die zweite Transaktion in den Baum eingefügt werden. Da keine neuen Items in dem Baum eingefügt werden, ändert sich an der *Header Table* nichts. Der Wurzelknoten besitzt bereits einen Kind-Knoten mit dem Item i_3. Daher muss dieser nicht neu erstellt werden und es wird lediglich der *Count* des Knoten mit dem Item i_3 um 1 erhöht. Für Item i_4 folgt analog, dass ebenfalls nur der *Count* erhöht werden muss. In Abbildung 2.6b ist der Baum nachdem die zweite Instanz eingefügt wurde zu sehen.
In Abbildung 2.7 sind die Bäume, nachdem jeweils die 3., 4. und 5. Transaktion eingefügt wurden. Da sich die *Header Table* nicht mehr ändert, wurde diese zur Vereinfachung weggelassen. Alle Items, die vorkommen können, sind bereits im Baum vertreten. Bei der 3. Transaktion ist eine Besonderheit zu beobachten, da der bereits existente Pfad im Baum und der neu einzufügende Pfad einen gemeinsamen Präfix i_3 haben. In Abbildung 2.7a ist der Baum nachdem die 3. Transaktion eingefügt wurde zu sehen. Es wurden die Items i_3 und i_2 eingefügt. Da der Pfad mit dem Item i_3 beginnt, muss zunächst nur der *Count* im entsprechenden Knoten erhöht werden. Der Knoten, der das Item i_3 repräsentiert, hat zu diesem Zeitpunkt allerdings nur ein Kind-Knoten und dieses repräsentiert das Item i_4. Daher wird ein neuer Knoten mit dem Item i_2 erstellt und eine Verlinkung zu dem bereits vorhandenen Knoten mit dem Item i_2 geknüpft.
Nachfolgend werden entsprechend die restlichen Transaktionen eingefügt. In Abbildung 2.7c ist noch der Zustand des Baumes nach dem die 5. Transaktion eingefügt wurde zu sehen. In diesem Fall musste nur das Item i_2 eingefügt werden. Da es vom Wurzelknoten aus keine Pfade gibt, die mit i_2 beginnen, wird ein neuer Knoten erstellt. Dieser wird dann ebenfalls mit den anderen Knoten, die i_2 repräsentieren, verlinkt.
Abschließend kann festgestellt werden, dass die Reihenfolge der Items entscheidend dafür ist wie gut die Items in einem Knoten zusammengefasst werden können. Wären die Items nicht in absteigender Reihenfolge sortiert, hätte man die



(a) 3.Transaktion eingefügt **(b)** 4.Transaktion eingefügt **(c)** 5.Transaktion eingefügt

Abbildung 2.7.: Änderungen des FP-Tree nachdem jeweils die 3., 4. und 5. Transaktion eingefügt wurden.

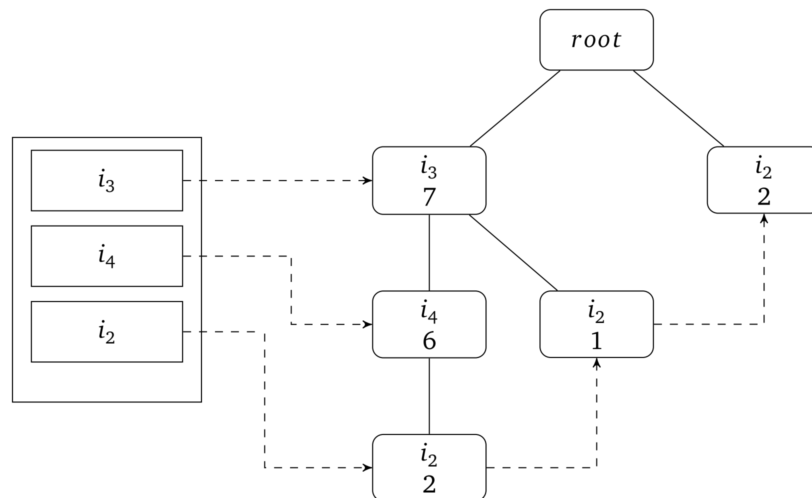


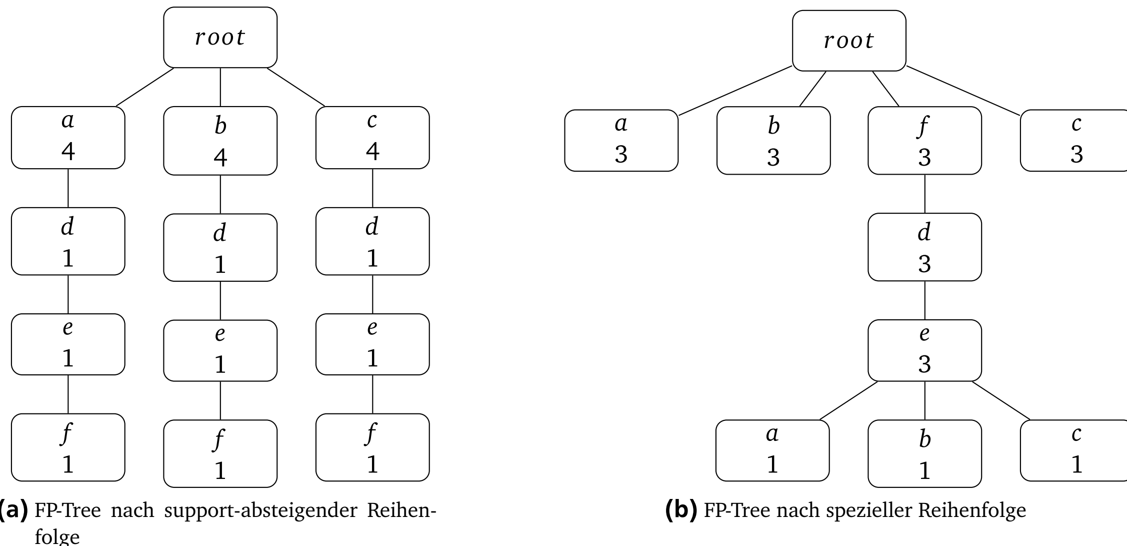
Abbildung 2.8.: FP-Tree und *Header Table* nachdem alle Instanzen in den Baum eingefügt wurden.

Items, die sehr häufig vorkommen, nicht so gut in einem Knoten zusammenfassen können. Würden im Baum zunächst die Items stehen, die am wenigsten in der Datenmenge auftreten, würden diese zwar auch zusammengefasst, doch da diese bereits weniger häufig vorkommen, ist auch der mögliche Vorteil eines Zusammenfassens nicht so gut wie bei absteigender Reihenfolge. Allerdings ist zu beachten, dass durch die absteigende Reihenfolge der Baum trotzdem nicht immer minimal oder besonders kompakt wird.

In Abbildung 2.8 ist der Baum nachdem alle Transaktionen eingefügt wurden dargestellt. Die Items i_3 und i_4 werden hierbei durch einen einzigen Knoten repräsentiert. Item i_2 wird hingegen durch drei via *node-link* verknüpfte Knoten dargestellt. Das Gesamtvorkommen eines Items wird anhand des Baumes durch die Summe aller *Counts* von Knoten mit gleichem Item bestimmt. Da es keine weiteren Knoten mit dem Item i_3 und i_4 gibt, geben diese bereits das Gesamtvorkommen des jeweiligen Items in der Datenmenge wieder. Zur Bestimmung des Gesamtvorkommen von i_2, wird die Summe mit Hilfe der *node-link*s bestimmt.

Des Weiteren soll verdeutlicht werden, dass die Pfade vom Wurzelknoten bis zu einem beliebigen Knoten, zum Teil, den Instanzen in der Datenmenge entsprechen. Der *Count* im Knoten besagt wie häufig die Items bis zu diesem Knoten gemeinsam vorkommen. Der Pfad vom Wurzelknoten bis zum Knoten mit Item i_4 besagt, dass es sechs Transaktionen gibt bei denen die Items i_3 und i_4 beide vorkommen. Für Abbildung 2.8 folgt:

- Die Items i_2, i_3 und i_4 treten zweimal in der Datenmenge gemeinsam auf.
- Die Items i_2 und i_3 treten ohne i_4 nur einmal gemeinsam auf.
- Item i_2 kommt insgesamt $\sum count(item\text{-}name == i_2) = 2 + 1 + 2 = 5$ mal in der Datenmenge vor.



(a) FP-Tree nach support-absteigender Reihenfolge

(b) FP-Tree nach spezieller Reihenfolge

Abbildung 2.9.: Beispiel eines nicht minimalen FP-Tree (ohne *Node-Link*s und *Header Table*)

Zusammenfassend hat ein FP-Tree nach [11] folgende Eigenschaften:

- Ein FP-Tree ist ein Präfix-Baum.
- Durch die Support-absteigend sortierte Reihenfolge der Items ist die Wahrscheinlichkeit höher, dass mehr Präfixe zusammengefasst werden können.
- Die Kompaktheit des Baumes wird durch die gegebene Reihenfolge der Items erhöht.
- Die Erstellung des Baumes benötigt exakt zwei Durchgänge durch die Datenmenge.
- Die Größe des Baumes ist durch die Größe der Datenbank beschränkt. Jede Instanz wird maximal nur einen neuen Pfad erzeugen.
- Die Tiefe des Baumes ist durch die maximale Anzahl von *frequent items* in einer Transaktion beschränkt.

Die Kompaktheit des Baumes wird durch die Support-absteigende Reihenfolge erhöht. Dies führt aber nicht zu einem maximal kompakten Baum. Durch betrachten der Dateneigenschaften kann eine bessere Kompaktheit erreicht werden. Betrachtet man die Datenmenge $\{adef, bdef, cdef, a, a, a, b, b, b, c, c, c\}$, gegeben in [11], mit einem Support von 25%, erhält man mit Support-absteigender Reihenfolge den Baum in Abbildung 2.9a. Werden die Knoten manuell angeordnet, kann der Baum wie in Abbildung 2.9b weiter minimiert werden. Der Baum hat in diesem Fall drei Knoten weniger als der FP-Tree nach Items mit Support-absteigender Reihenfolge.

Nachdem der Baum erstellt wurde, werden anhand des Baumes alle *Frequent Pattern* generiert, die den Minimal-*Support* ξ erfüllen. Auf Basis der Datenmenge aus Tabelle 2.2 und dem daraus erzeugten FP-Tree aus Abbildung 2.8 sowie einer *Threshold* $\xi = 30\%$, wird das Vorgehen des Algorithmus erläutert.
Im ersten Schritt wird das Item i, welches am wenigsten häufig im FP-Tree vorkommt ausgewählt. Da die Reihenfolge der Items in der Liste *FList* aus Algorithmus 2 bereits gesammelt wurde, wird dort dementsprechend das letzte Item der Liste genutzt. Mit Hilfe der *Header Table* und den *Node-Links* werden alle Pfade im Baum vom Wurzelknoten bis zum Knoten mit Item i gesammelt. Auf dem gegebenen Datensatz angewendet, ist das gesuchte Item $i = i_2$. Die daraus resultierenden Pfade sind $\{(i_3 : 7 \rightarrow i_4 : 6 \rightarrow i_2 : 2), (i_3 : 7 \rightarrow i_2 : 1), (i_2 : 2)\}$. Der Wurzelknoten wird dabei weggelassen, da dieser nur als Startpunkt des Baumes dient. Es ist zu beachten, dass die Kombination aller Items im Pfad nur so häufig vorkommt, wie der *Count* des tiefsten Knoten dies angibt. Daraus folgt:

- Itemmenge $\{i_3, i_4, i_2\}$ kommt gemeinsam nur zweimal in der Datenmenge vor.
- Itemmenge $\{i_3, i_2\}$ kommt dreimal in der Datenmenge vor. Es gibt mehrere Pfade, die diese Items beinhalten.
- Beim letzten Pfad handelt es sich nur um einen Knoten. Daher ist das Gesamtvorkommen, wie bereits eingeführt die Summe der *Counts* von Knoten mit Item i_2. Daher kommt die Itemmenge $\{i_2\}$ fünfmal in der Datenmenge vor.

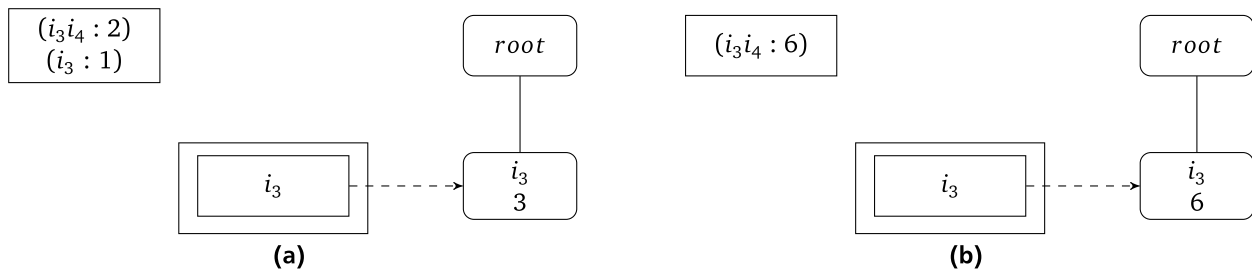


Abbildung 2.10.: Conditional Pattern Base und Conditional FP-Tree mit Header Table von i_2 (links) und i_4 (rechts)

Als erstes wird das zuvor ermittelte Item als *frequent pattern* $(i_2 : 5)$ in die Menge aller gefundenen *Frequent Pattern* hinzugefügt. Anschließend werden die resultierenden Itemmengen verarbeitet. Zur einfacheren Handhabung wird in [11] die vereinfachte Notation $\{(i_3i_4 : 2), (i_3 : 1)\}$ eingeführt. Da die Itemmengen unter der Kondition erstellt wurden, dass Item i_2 vorkommt, wird dieses bei der Notation weggelassen. Des Weiteren ist die *Count*-Zahl der Itemmenge gegeben durch den *Count* des Knoten mit Item i_2. Die resultierende Menge $\{(i_3i_4 : 2), (i_3 : 1)\}$ ist die *Sub-Pattern Base*, welche auch *Conditional Pattern Base* von Item i_2, genannt wird. Eine andere Schreibweise wäre $\{$*Sub-Pattern Base* $|$ *Kondition*$\}$. Auf die gegebene Itemmenge wäre dies $\{((i_3i_4 : 2), (i_3 : 1)) | i_2\}$. Die *Conditional Pattern Base* wird anschließend genutzt um einen FP-Tree, wie in Algorithmus 2 beschrieben, zu erstellen. Der resultierende Baum wird als *Conditional FP-Tree* von i_2 bezeichnet. Die Erstellung des Baumes führt im vorliegenden Beispiel zu einem Baum mit nur einem Pfad, welcher auch nur einen Knoten mit Item i_3 beinhaltet. Dieser ist in Abbildung 2.10a veranschaulicht. Dies tritt ein, da wieder nur Items genutzt werden, die die absolute *Threshold* $\xi_{abs} = 3$ erfüllen.
Da dieser Baum unter der Kondition erstellt wurde, dass das Item i_2 vorhanden ist, werden den neu gefundenen *Frequent Pattern* das Item i_2 angefügt. Die so entstandenen Bäume werden solange rekursiv bearbeitet, bis der Baum leer ist bzw. nur noch aus dem Wurzelknoten besteht. In diesem Fall wird daher noch das Frequent Pattern $(i_3i_2 : 3)$ in die Menge aller gefundenen Frequent Pattern hinzugefügt.
Dies wird für die restlichen Items in Support-aufsteigender Reihenfolge wiederholt. Somit würde als nächstes die Conditional Pattern Base für i_4 erstellt werden. Dazu werden erneut alle Pfade vom Wurzelknoten bis zu einem Knoten mit dem Item i_4 gesammelt. Es werden erneut über die *Header Table* und die *Node-Links* Knoten mit dem gesuchten Item herausgesucht. Von dort aus werden dann die Pfade zum Wurzelknoten ermittelt. In diesem Fall gibt es nur einen Pfad mit $\{(i_3 : 7 \rightarrow i_4 : 6)\}$. Das direkt gefundene *Frequent Pattern* $(i_4 : 6)$ wird daraufhin eingefügt und anschließend die Conditional Pattern Base $\{(i_3 : 6)\}$ von i_4 erstellt. Als nächstes wird wieder der Conditional FP-Tree von i_4 generiert. In diesem Fall wäre der Baum wieder bestehend aus einem Pfad, welcher auch nur aus einem Knoten i_3 mit dem *Count* 6 besteht, veranschaulicht in Abbildung 2.10b. Aus der Kombination des bisher gefundenen Pattern und dem neu erstellten Knoten wird erneut ein neues Frequent Pattern $(i_3i_4 : 6)$ in die Menge aller gefundenen Frequent Pattern hinzugefügt. Da im nächsten Schritt der Baum leer wäre, endet die Rekursion und das nächste Item wird bearbeitet.
Dabei fällt auf, dass das erste bearbeitete Item i_2 im zweiten Schritt nicht mehr beachtet wurde. Es werden lediglich Pattern erzeugt, die das bearbeitete Items selbst und die Items, die oberhalb des Knoten im Baum angeordnet sind, beinhalten. Jegliche Kombinationen, die das Item i_2 beinhalten, wurden bereits im ersten Schritt gefunden. Gleiches gilt, wenn das letzte Item i_3 bearbeitet wird. Zusammenfassend werden alle Items ignoriert, die bereits verarbeitet wurden. Für die gegebene Datenmenge ergeben sich damit folgende *Frequent Pattern*.

$$\{(i_2 : 5), (i_2i_3 : 3), (i_4 : 6), (i_4i_3 : 6), (i_3 : 7)\}$$

Damit wäre die Vorgehensweise zusammengefasst wie folgt:

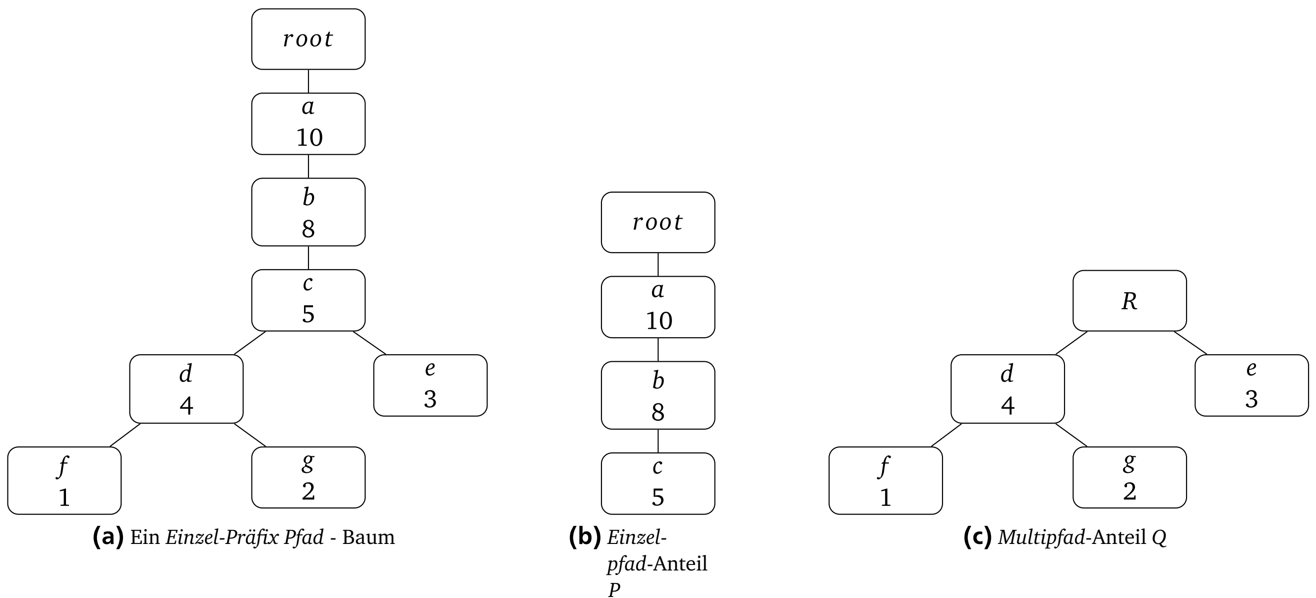
1. Conditional Pattern Base für alle Items in der *Header Table* erstellen.

2. Conditional FP-Tree bauen.

3. Das Item selbst in Kombination mit dem vorherigen Pattern in die Menge aller *Frequent Pattern* hinzufügen. (Beim ersten Aufruf ist das vorherige Pattern leer.)

4. Schritt 1-3 für den Conditional FP-Tree wiederholen, solange bis keine Conditonal FP-Trees erstellt werden können.

Die gezeigte Vorgehensweise würde bereits für alle möglichen Bäume funktionieren. Bevor dies in einem Algorithmus festgehalten wird, muss noch ein Spezialfall erläutert werden. Bei diesem handelt es sich um einen *"Einzel-Präfix Pfad'* - Baum. Ein solcher Baum besteht aus nur *einem* Pfad vom Wurzelknoten bis zu einem inneren Knoten, der den Baum erst dann weiter unten in mehrere Pfade aufteilt. Ein solcher Baum ist in Abbildung 2.11a dargestellt. Der Baum wurde



(a) Ein *Einzel-Präfix Pfad* - Baum **(b)** *Einzelpfad*-Anteil P **(c)** *Multipfad*-Anteil Q

Abbildung 2.11.: Ein *Einzelpfad* - Baum mit seinen jeweiligen Anteilen dargestellt.

dabei auf einer unspezifizierten Datenmenge mit einer absoluten Threshold $\xi_{abs} = 1$ erstellt. Der Einzelpfad-Anteil P ist dabei gegeben durch $(a : 10) \rightarrow (b : 8) \rightarrow (c : 5)$ und der Multipfad-Anteil Q ist gegeben durch den Baum beginnend bei Knoten $(c : 5)$, dargestellt in Abbildung 2.11b und Abbildung 2.11c.
Zur Bestimmung der *Frequent Pattern* kann in diesem Fall zunächst der Einzelpfad-Anteil P durch Kombination der Knoten miteinander verarbeitet werden. Dabei bilden alle möglichen Teilpfade ein neues *Frequent Pattern*. Der *Count* des Pattern entspricht dabei dem Minimum der beinhalteten Items. Auf das Beispiel angewendet wäre die Menge der *Frequent Pattern* für P gegeben durch $\{(a : 10), (b : 8), (c : 5), (ab : 8), (ac : 5), (bc : 5), (abc : 5)\}$. Der Multipfad-Anteil wird wie zuvor beschrieben bearbeitet. Dabei werden die Schritte 1-4 aus der Zusammenfassung solange wiederholt bis die *Frequent Pattern* - Menge von Q generiert wurde. Diese ist gegeben durch $\{(d : 4), (e : 3), (f : 1), (g : 2), (df : 1), (dg : 2)\}$. Zur Bestimmung der *Frequent Pattern* - Menge des gesamten Baumes, wird eine Kreuzprodukt-ähnliche Operation ausgeführt. Dabei wird jedes Element mit jedem vereinigt und der *Count* des neuen Pattern ist dabei das Minimum von den einzelnen Elementen. Nachfolgend ist die Frequent Pattern - Menge für das gegebene Beispiel zu sehen. Die einzelnen Pattern sind dabei so angeordnet, dass jede Zeile für ein Element aus P steht und jede Spalte für ein Element aus Q.

	$(d : 4)$,	$(e : 3)$,	$(f : 1)$,	$(g : 2)$,	$(df : 1)$,	$(dg : 2)$,
$(a : 10)$,	$(ad : 4)$,	$(ae : 3)$,	$(af : 1)$,	$(ag : 2)$,	$(adf : 1)$,	$(adg : 2)$,
$(b : 8)$,	$(bd : 4)$,	$(be : 3)$,	$(bf : 1)$,	$(bg : 2)$,	$(bdf : 1)$,	$(bdg : 2)$,
$(c : 5)$,	$(cd : 4)$,	$(ce : 3)$,	$(cf : 1)$,	$(cg : 2)$,	$(cdf : 1)$,	$(cdg : 2)$,
$(ab : 8)$,	$(abd : 4)$,	$(abe : 3)$,	$(abf : 1)$,	$(abg : 2)$,	$(abdf : 1)$,	$(abdg : 2)$,
$(ac : 5)$,	$(acd : 4)$,	$(ace : 3)$,	$(acf : 1)$,	$(acg : 2)$,	$(acdf : 1)$,	$(acdg : 2)$,
$(bc : 5)$,	$(bcd : 4)$,	$(bce : 3)$,	$(bcf : 1)$,	$(bcg : 2)$,	$(bcdf : 1)$,	$(bcdg : 2)$,
$(abc : 5)$,	$(abcd : 4)$,	$(abce : 3)$,	$(abcf : 1)$,	$(abcg : 2)$,	$(abcdf : 1)$,	$(abcdg : 2)$

Da jedes Item in einem Pfad maximal einmal vorkommt, besitzen die Teilbäume keine gemeinsamen Knoten. Dies erst ermöglicht die Aufteilung der Prozedur in zwei verschiedene Bäume. Zusammen ergeben diese den FP-Growth-Algorithmus von Han et al. [11], wiedergegeben in Algorithmus 3.
In Zeile 1-10 wird der Spezialfall eines *Einzelpfad*-Baumes verarbeitet. Dabei wird der Baum in zwei Teile P und Q aufgeteilt. Auf dem Teilbaum P werden dann durch Kombination der Knoten die Frequent Pattern erstellt. Ist der gegebene Baum kein Einzelpfad - Baum, ist Q für die nachfolgenden Schritte der gesamte vorliegende Baum. In Zeile 12-17 wird die zuvor gezeigte allgemeine und rekursive Methode ausgeführt, die für jede Art von Baum funktioniert. Dazu werden für jedes Item das Pattern in Kombination mit dem jeweiligen *Konditionspattern* erzeugt. Danach wird die Conditional-Base sowie der Conditional FP-Tree rekursiv aufgerufen.
Abschließend wird in Zeile 19 die Vereinigung, der Frequent Pattern - Mengen von P, Q und dem *"Kreuzprodukt"* $P \times Q$ zurückgegeben.
Im vorgestellten Algorithmus wird durch den rekursiven Ansatz auch ein *Einzelpfad* - Baum erkannt, der sich erst weiter unten im Baum befindet, was die Effizienz des Algorithmus weiter erhöht. Ein weiterer Vorteil ist, dass selbst im Falle,

wenn die Datenmenge eine exponentielle Anzahl an Frequent Pattern ermöglicht, der FP-Tree selbst im Regelfall sehr klein ist und niemals exponentiell wächst [11, S.69]. Die Effizienz des Algorithmus sowie des FP-Trees wird in [11, S.73 ff.] aufgezeigt.
Die Problemstellung in dieser Arbeit ist allerdings kein Data Mining Problem. Das Ziel ist einen Ansatz basierend auf dem FP-Tree zu entwickeln eine Regel zu evaluieren und mit dem bisherigen Ansatz in Effizienz zu vergleichen. Dementsprechend wird in dieser Arbeit ein angepasster Baum sowie ein speziell auf das gegebene Framework angepasster Algorithmus entwickelt und vorgestellt. Dabei werden Elemente einer Erweiterung des FP-Tree, welche in [5] unter dem Algorithmus *SD-Map* vorgestellt werden, genutzt.

Algorithm 3 FP-Growth

```
1:  procedure FPGrowth(FPTree T, Itemset α)
2:      if Tree T contains a single prefix path then
3:          FPTree P = single prefix-path part of T
4:          FPTree Q = multipath part of T, with the top branching node replaced by a null root
5:
6:          for each combination (denoted as β) of the nodes in the path P do
7:              generate pattern β ∪ α with support = minimum support of nodes in β
8:              freqPatternSetP = add that Pattern to the frequent pattern set of P
9:      else
10:         Q = T
11:
12:     for each item a_i in Q do
13:         generate pattern β = a_i ∪ α with support = a_i.support
14:         construct β's conditional pattern-base and then conditional fp-tree Tree_β
15:         if Tree_β ≠ ∅ then
16:             call FPGrowth(Tree_β, β)
17:             freqPatternSetQ = the set of pattern so generated
18:
19:     return freqPatternSetP ∪ freqPatternSetQ ∪ (freqPatternSetP × freqPatternSetQ)
```

2.3.2 SD-Map

Bei *SD-Map* (**S**ubgroup **D**iscovery - Map) handelt es sich um einen von Atzmueller und Puppe entwickelten Algorithmus zur Bestimmung von Untergruppen bzw. Teilgruppen [5]. Der Algorithmus baut dabei auf FP-Growth auf, erweitert diesen aber um die Fähigkeit fehlende Daten in der Datenmenge zu verarbeiten. Die Vorgehensweise des Algorithmus selbst ist für diese Arbeit zwar nicht relevant aber die erweiterte Fassung des FP-Tree (nachfolgend *SD-Tree* genannt), wird genutzt.

Bei einem *SD-Tree* handelt es sich um einem FP-Tree, der in den Knoten nicht das Auftreten des jeweiligen Items in der Datenmenge speichert, sondern vermerkt wie häufig ein Attribut-Wert-Paar mit einem bestimmten Klassenwert vorkommt bzw. nicht vorkommt. Zusammengefasst speichert ein Knoten die *positiven* und *negativen* Fälle eines Attribut-Wert-Paares ab. Der positive Fall ist dabei, wenn das Attribut-Wert-Paar mit dem gewünschtem Wert der Klassenvariable vorkommt. Analog ist der negative Fall, wenn die Klassenvariable nicht den gewünschten Wert hat. Abbildung 2.12 zeigt den Unterschied zwischen einem SD-Tree-Knoten und einem FP-Tree-Knoten nochmals auf.
Zur Veranschaulichung wird die Datenmenge aus Tabelle 2.1 genutzt. Dabei ist zu beachten, dass anders als die genutzte Datenmenge in Tabelle 2.2, die Einträge in der Tabelle 2.1 für den konkreten Wert des Attributes stehen und nicht ob ein Item vorkommt. Die positiven und negativen Fälle sind hier mit einem + bzw. − markiert. In diesem Fall gibt es nur zwei Klassen (+ und −). Für mehr als zwei Klassen wird eine Klasse als positiver Fall und die restlichen als Negativer genutzt. So kann aus Tabelle 2.1 entnommen werden, dass das Attribut-Wert-Paar $a_1 = 1$ sowohl in einem positiven als auch in einem negativen Fall auftritt. In einem entsprechenden SD-Tree-Knoten würde daher für beide Werte eine 1 gespeichert. Das Attribut-Wert-Paar $a_2 = 1$ kommt in zwei positiven und drei negativen Fällen vor.
Der Aufbau eines *SD-Trees* funktioniert ähnlich wie der eines FP-Trees. Zur Ermittlung der Reihenfolge in der die Attribut-Wert-Paare in den Baum eingefügt werden sollen, wird das Auftreten mit der Klassenvariablen gezählt. Dies ist wichtig für den Fall, dass das Klassenattribut selbst in der Datenmenge fehlt. Ist die Datenmenge vollständig, entspricht dies einem einfachen Zählen wie häufig ein bestimmtes Attribut-Wert-Paar vorkommt. Für das genutzte Beispiel ergibt sich daraus die folgende Reihenfolge:

$$[(a_1 = 0) : 8 \rightarrow (a_3 = 1) : 7 \rightarrow (a_2 = 1) : 5 \rightarrow (a_2 = 0) : 5 \rightarrow (a_3 = 0) : 3 \rightarrow (a_1 = 1) : 2]$$

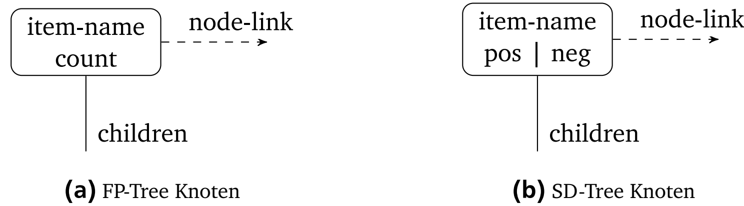


Abbildung 2.12.: Vergleich eines FP-Tree - Knotens zu einem SD-Tree - Knoten

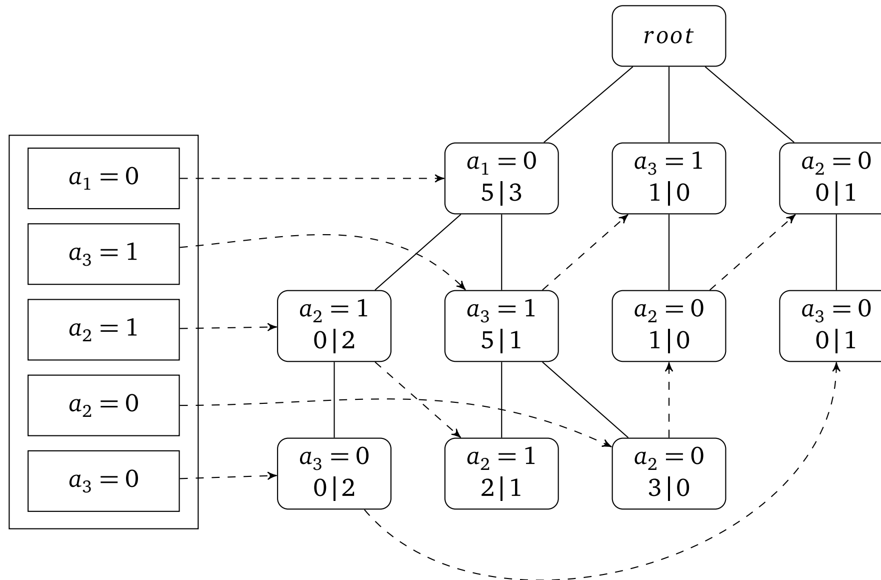


Abbildung 2.13.: *SD-Tree* und *Header Table* zu Datensatz 2.1 ($\xi_{abs} = 3$)

Bei einer absoluten *Threshold* $\xi_{abs} = 3$, entfällt nur ($a_1 = 1$). Anschließend wird wie bei *FP-Growth* jede Instanz nach Reihenfolge der Liste in den Baum eingefügt. Der daraus resultierende Baum ist in Abbildung 2.13 dargestellt. Die benötigten Werte können dem Baum wie aus einem FP-Tree entnommen werden. Dabei muss für das Vorkommen eines Attribut-Wert-Paares nur der *Count* (die Summe aus positiven und negativen Fällen) bestimmt werden. Die restliche Vorgehensweise ist analog zu der in einem FP-Tree. So kann dem Baum entnommen werden, dass das Attribut-Wert-Paar $a_2 = 1$ insgesamt fünfmal vorkommt. Zudem aber kann entnommen werden, dass es zweimal in positiven und dreimal im negativen Fall auftritt. Dazu muss wie bei einem FP-Tree nur den *Node-Links*, beginnend mit aus der *Header Table*, gefolgt werden.

2.4 Zusammenfassung

In diesem Kapitel wurde zunächst der Bereich des Maschinellen Lernens eingeführt. Es wurde erläutert wie Dieser entstanden ist, welche Ansätze es gibt und wo dieser Bereich alles eingesetzt werden kann. Erweitert wurde dies durch das Kapitel *Separate and Conquer*, welches den genutzten Ansatz des verwendeten Frameworks erläuterte und wo die Verbindung zu der hier vorgestellten Arbeit besteht. Abschließend wurde in der Assoziationsanalyse das Gebiet selbst definiert und mit FP-Growth und SD-Map entsprechende Ansätze dafür vorgestellt. Dabei wurde insbesondere FP-Growth eingeführt, da die dort verwendete Struktur des FP-Tree und eine abgewandelte Vorgehensweise des Ansatzes in dieser Arbeit genutzt wird.

In diesem Kapitel wird zunächst die Funktionsweise der Funktion EVALUATERULE in der genutzten Version von *SeCo* erläutert. Anschließend wird der Ansatz mit einem FP-Tree eingeführt. Es wird Schritt für Schritt aufgezeigt wie dieser auf das Framework angepasst und in verschiedenen Bereichen optimiert wurde.

3.1 SeCo - EVALUATERULE

In der genutzten Version von *SeCo* ist die Funktion EVALUATERULE einfach gehalten. Sie hat die Aufgabe für eine Regel die Werte einer Konfusionsmatrix zu bestimmen und mit Hilfe dieser, durch eine gegebene Heuristik, den heuristischen Wert der Regel zu berechnen. Die vorliegende Vorgehensweise beruht dabei auf einem einfachen iterativen Ansatz. In diesem geht die Funktion in jedem Aufruf durch die Datenmenge und überprüft für jede Instanz ob diese von der Regel abgedeckt wird oder nicht. Dabei wird zusätzlich unterschieden, ob die Regel für die jeweilige Instanz korrekt ist. Das bedeutet der Regelkopf bzw. die Vorhersage entspricht dem Klassenwert dieser Instanz. Zusammengefasst entspricht dies den *true positive*, *false positive*, *true negative* und *false negative* einer Kofusionsmatrix.

		actual	
		positve	negative
predicted	positive	true positive	false positive
	negative	false negative	true negative

Tabelle 3.1.: Allgemeiner Aufbau einer Konfusionsmatrix

2	1
4	3

Tabelle 3.2.: Beispielhafte Konfusionsmatrix einer Regel

In Tabelle 3.1 ist der Aufbau einer allgemeinen Konfusionsmatrix gegeben. Bei *predicted* sind die Werte der Vorhersage gemeint und unter *actual* die tatsächlich vorliegende Verteilung in den Daten. Folglich beschreiben *true positive* und *true negative*, die korrekten Vorhersagen und *false positive* sowie *false negative* die inkorrekten Vorhersagen. Auf den gegebenen Datensatz aus Tabelle 2.1 und der Regel $(a_2 = 1 \wedge a_3 = 1) \Rightarrow a_4 = +$ ergibt sich die beispielhafte Konfusionsmatrix in Tabelle 3.2.
Zur Bestimmung dieser Werte verarbeitet die Funktion EVALUATERULE iterativ jede Instanz einzeln. So zählt die erste Instanz als *true positive*, da die Attribute die entsprechenden Werte aufweisen und das Attribut $a_4 = +$ mit dem Regelkopf übereinstimmt. Für die 2. Instanz folgt, dass diese als *false negative* gewertet wird, da die Instanz selbst positiv ist aber von der Regel nicht als solche erkannt wird, da das Attribute $a_2 = 0$ nicht mit der Regel übereinstimmt. Für die 3. Instanz folgt analog, dass dies als *false positive* gewertet wird, da die Attribute a_2 und a_3 zwar mit der Regel übereinstimmen aber die Instanz selbst den Klassenwert $a_4 = -$ hat. Abschließend ist die 4. Instanz ein *true negative*, da das Attribute $a_3 = 0$ ist. Somit wird die Instanz korrekterweise nicht abgedeckt, da die Instanz selbst den Klassenwert $a_4 = -$ hat. In Tabelle 3.3 ist eine Übersicht zu allen Instanzen und deren Zuordnung unter der gegebenen Regel.
Die genutzte Variante von EVALUATERULE würde die Datenmenge zur Evaluierung einer jeden Regel jedes mal erneut durchgehen. Dieses Vorgehen ist redundant und damit ineffizient. Daher soll eine Struktur genutzt werden, die den

a_1	a_2	a_3	a_4	Zuordnung
0	1	1	+	*true positive*
0	0	1	+	*false negative*
0	1	1	−	*false positive*
0	1	0	−	*true negative*
0	0	1	+	*false negative*
1	0	0	−	*true negative*
0	1	0	−	*true negative*
0	1	1	+	*true positive*
0	0	1	+	*false negative*
1	0	1	+	*false negative*

Tabelle 3.3.: Datensatz aus Tabelle 2.1 und rechts die Zuordnung unter der Regel $(a_2 = 1 \wedge a_3 = 1) \Rightarrow a_4 = +$

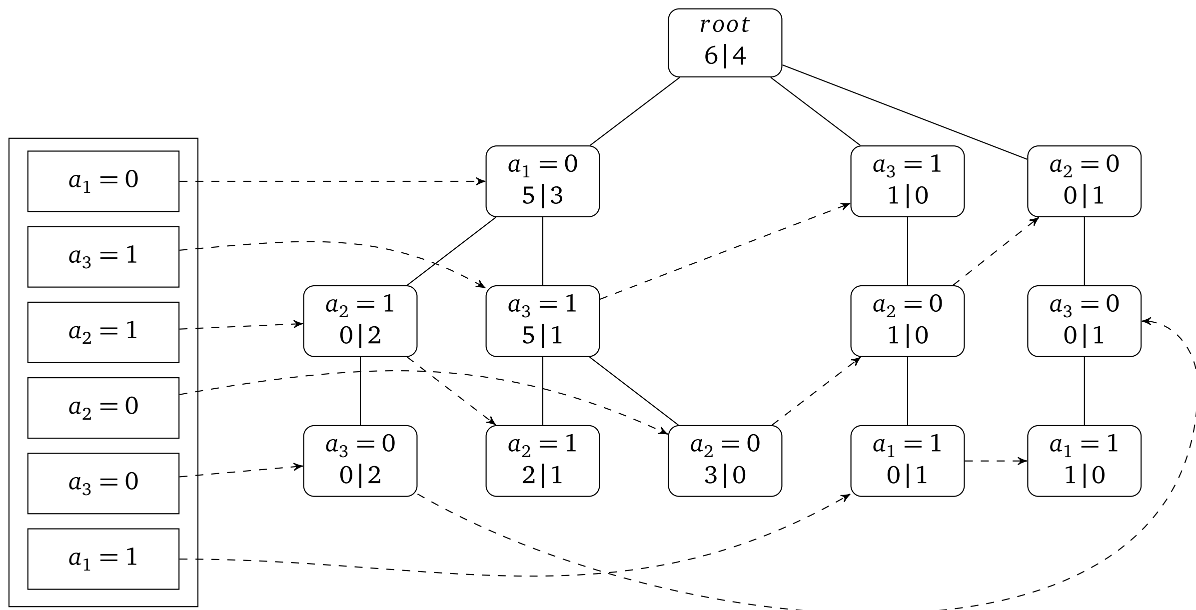


Abbildung 3.1.: Vollständiger *SD-Tree* zu Datensatz 3.3 ($\xi_{abs} = 0$)

Datensatz komprimiert und gleichzeitig das Ermitteln der Konfusionmatrix verbessert. Eine Effizienzsteigerung kann durch Beachtung verschiedener Besonderheiten erreicht werden.

- Das Iterieren über die Datenmengen sollte minimiert werden. Dazu wird eine Struktur geschaffen, welche die relevanten Instanzen in der Datenmenge direkt auswählt. Unter relevant wird hierbei verstanden, dass die Instanzen wahrscheinlicher durch die Regel abgedeckt werden.

- In einer Datenmenge können Instanzen mehrfach vorkommen. Daher sollte eine Struktur geschaffen werden, die diese zusammenfasst und damit den Datensatz komprimiert.

Eine Struktur, die diese Anforderungen erfüllt, ist der *SD-Tree*. Dieser besitzt, genauso wie der *FP-Tree*, die Eigenschaft die Daten zu komprimieren. Gleiche Instanzen erzeugen im *SD-Tree* nur einen einzigen Pfad. Durch die *Header Table* können auch direkt bestimmte Werte angesprungen werden. Dadurch muss zusätzlich neben der Komprimierung auch nicht mehr jeder Pfad bzw. jede Instanz überprüft werden.

3.2 Evaluierung mit Hilfe eines SD-Tree

Der SD-Tree wurde bereits in Kapitel 2.3.2 eingeführt. Die Erstellung des SD-Trees verläuft dabei analog zu der eines FP-Trees und wird hier nicht weiter vertieft. Die einzige Änderung bezogen auf die Erstellung des Baumes, ist das Hinzufügen der *tp* und *fp* über alle Daten in den Wurzelknoten. Nachfolgend wird ein Ansatz vorgestellt den Baum zu bearbeiten, der auch auf den später erweiterten Bäumen funktioniert und speziell für die Evaluierung einer Regel konzipiert wurde. Das Verfahren soll anhand eines Beispieles vorgestellt werden. Dazu wird der Datensatz aus Tabelle 3.3 genutzt. Anders als bei *FP-Growth* oder *SD-Map* müssen für die Evaluierung alle Daten genutzt werden. Daher ist eine *Threshold* ξ nicht mehr notwendig. Diese kann somit komplett weggelassen oder auf $\xi_{abs} = 0$ gesetzt werden. Der daraus resultierende Baum ist in Abbildung 3.1 zu sehen. Dabei fällt auf, dass ein *SD-Tree* mit $\xi_{abs} = 0$ immer „vollständige' Pfade besitzt. In jedem Pfad ist jedes Attribut-Wert-Paar (außer die Klassenvariable) genau einmal vorhanden. Daraus ergeben sich folgende Eigenschaften:

1. Die Tiefe des Baumes ist durch die Anzahl der Attribute - 1 beschränkt bzw. gegeben. Zusätzlich hat jeder Pfad im Baum diese / die gleiche Tiefe.

2. Die *"positive"* und *"negative"* Werte der inneren Knoten sind immer die Summe der *"positive"* und *"negative"* Werte der Blattknoten, die vom dem Knoten ausgehen.

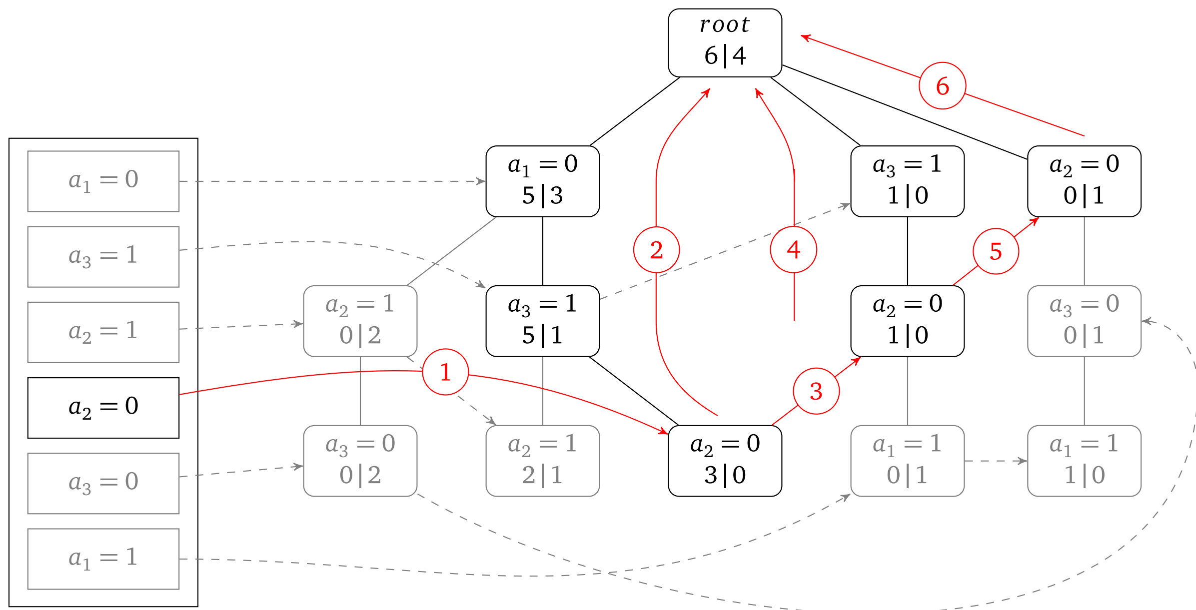


Abbildung 3.2.: Visualisierung der einzelnen Schritte während der Evaluierung. Die roten Kanten stehen für die einzelnen Schritte.

Bezogen auf das gegebene Beispiel, hat der Datensatz vier Attribute a_1 - a_4 und daraus folgend die Tiefe 3. Die zweite Eigenschaft lässt sich ebenfalls in Abbildung 3.1 beobachten. Der Wurzelknoten beschreibt das Vorkommen des Klassenwertes über die gesamte Datenmenge. Werden die *“positive“* und *“negative“* Werte aller Blattknoten summiert, erhält man die Werte im Wurzelknoten. Analog sind die Werte im Knoten $(a_1 = 0 : 5|3) = (a_3 = 0 : 0|2) + (a_2 = 1 : 2|1) + (a_2 = 0 : 3|0)$ durch seine Blattknoten gegeben. Es ist zu beachten, dass diese Eigenschaften für FP-Trees oder SD-Trees im Allgemeinen nicht gelten. Nur bei einer *Threshold* $\xi_{abs} = 0$ ist der Baum vollständig und erfüllt die beschriebenen Eigenschaften. Ersichtlich wird dies, wenn Abbildung 2.13 mit $\xi_{abs} = 3$ und Abbildung 3.1 mit $\xi_{abs} = 0$ verglichen werden.

Zur Evaluation einer Regel r müssen die Konditionen, die in der Regel vorkommen, im Baum gefunden werden. Gleichzeitig muss sichergestellt sein, dass die gesuchten Konditionen auf einem gemeinsamen Pfad liegen und es damit mindestens eine Instanz im Datensatz gibt, die beide Konditionen aufweist. Der Vorgang soll daher zunächst beispielhaft für die Regel $(a_2 = 0 \land a_3 = 1) \Rightarrow a_4 = +$ besprochen werden.

Als erstes wird im Regelkörper die Kondition ermittelt, die am tiefsten im Baum aufzufinden ist. In diesem Fall entspricht dies $a_2 = 0$. Darauffolgend wird mit Hilfe der *Header Table* der erste Knoten *linkNode* mit der Kondition $a_2 = 0$ im Baum herausgesucht (1.Schritt in Abbildung 3.2). Vom *linkNode* bis zum Wurzelknoten wird geprüft, ob der jeweilige Knoten von der Regel abgedeckt wird. Ein Knoten gilt als abgedeckt, wenn dieser bei gleichem Attribut den gleichen Wert aufweist. Für den ersten Knoten ist dies der Fall, da $a_2 = 0$ in der Regel vorkommt. Anschließend wird der darüber liegende Knoten genauso geprüft. In diesem Fall ist der Knoten $(a_3 = 1 : 5|1)$ ebenfalls abgedeckt. Da bereits alle Konditionen der Regel abgedeckt sind und jede Kondition genau einmal in einem Pfad vorkommt, muss der letzte Knoten im Pfad $(a_1 = 0 : 5|3)$ nicht mehr überprüft werden (2.Schritt in Abbildung 3.2). Über die Kondition in diesem Knoten wird von der Regel $(a_2 = 0 \land a_3 = 1) \Rightarrow a_4 = +$ keine Aussage getätigt. Abschließend für diesen Pfad werden die *“positive“* und *“negative“* Werte des *linkNode*-Knotens zu den *true positive* und *false positive* Werten der Regel hinzugefügt, wenn der Pfad von der Regel abgedeckt wurde. Ein Pfad gilt dabei als abgedeckt, wenn nur Knoten vorkommen, die abgedeckt oder ignoriert wurden. Ist im Pfad ein Knoten mit einem Attribut, welches in der Regel vorkommt aber einen anderen Wert aufweist als die Regel, ist der Pfad nicht abgedeckt. Als nächstes wird mit Hilfe der *Link-Node* - Referenz der nächste Pfad herausgesucht.

Zu den *true positive (tp)* und *false positive (fp)* einer Regel werden nur die *“positive“* und *“negative“* Werte des tiefsten betrachteten Knoten hinzugefügt (dem *linkNode*). Die Werte der restlichen Knoten spielen für die Auswertung des Pfades keine Rolle. Diese decken Instanzen ab, die auch andere Konditionen beinhalten und somit für andere Instanzen in der Datenmenge stehen. Für das gegebene Beispiel deckt der Elternknoten vom *linkNode* auch die Instanzen mit der Kondition $a_2 = 1$ ab.

4	0
2	4

Tabelle 3.4.: Konfusionsmatrix zu Regel $(a_2 = 0 \wedge a_3 = 1) \Rightarrow a_4 = +$ unter Datensatz 3.3

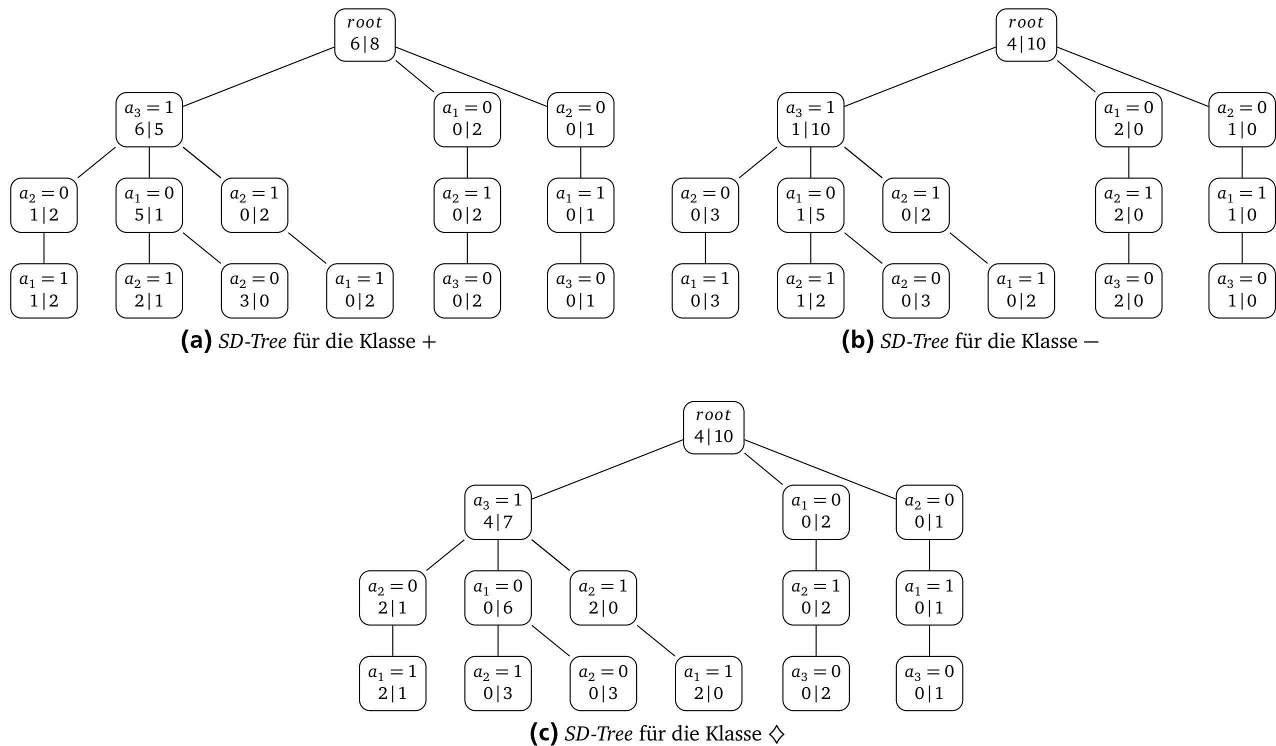
a_1	a_2	a_3	a_4
0	1	1	+
0	0	1	+
0	1	1	−
0	1	0	−
0	0	1	+
1	0	0	−
0	1	0	−
0	1	1	+
0	0	1	+
1	0	1	+
1	0	1	◇
1	1	1	◇
1	0	1	◇
1	1	1	◇

Tabelle 3.5.: Erweiterte Datenmenge 2.1 mit 3 Klassen $(+, -, \diamondsuit)$

Die beschriebene Vorgehensweise wird nach dem ersten Pfad für die restlichen Pfade wiederholt. Für den zweiten Pfad im Baum, beginnend beim neuen *linkNode*-Knoten $(a_2 = 0 : 1|0)$, wird wieder zunächst geprüft ob jeder Knoten von der Regel abgedeckt wird (3. und 4. Schritt in Abbildung 3.2). Da der Pfad zum Wurzelknoten nur aus zwei Knoten besteht und diese der Regel entsprechen, werden die *"positive"* und *"negative"* Werte des *linkNode* zu den *tp* und *fp* der Regel hinzugefügt. Damit ist der aktuelle Stand der Regel, dass diese vier *true positives* und null *false positives* vorweist.
Zum Schluss wird der letzte Pfad im Baum überprüft. Dieser beginnt beim Knoten $(a_2 = 0 : 0|1)$ zu sehen im 5.Schritt in Abbildung 3.1. Von dort wird wieder geprüft, ob jeder Knoten von der Regel abgedeckt wird. Zwar ist hier dies der Fall aber es konnten nicht alle Konditionen der Regel verarbeitet werden. So konnte auf dem Pfad keine Aussage zur benötigten Kondition $a_3 = 1$ in der Regel getätigt werden (6. Schritt in Abbildung 3.2). Das Problem kann gelöst werden, wenn beachtet wird, dass die Knoten im Baum nach einer speziellen Reihenfolge angeordnet wurden. Die Reihenfolge besagt, dass die Knoten mit der Kondition $a_2 = 0$ nach einem Knoten mit der Kondition $a_3 = 1$ kommen. Da in diesem Pfad kein Knoten mit der benötigten Kondition gefunden wurde, kann gefolgert werden, dass dieser im Pfad nicht existiert. Da aber alle Pfade beginnend vom Wurzelknoten bis zu einem Blattknoten jedes Attribut genau einmal beinhalten, muss für alle Knoten unterhalb des *linkNode* gelten, dass der Wert für das Attribut $a_3 \neq 1$ ist. In Abbildung 3.1 ist zu erkennen, dass dies auch der Fall ist.
Nachdem alle Pfade im Baum bearbeitet wurden, stehen die *tp* und *fp* für die gegebene Regel fest. Zum Schluss müssen nur noch die *false negative (fn)* und *true negative (tn)* bestimmt werden. Diese werden indirekt durch den Wurzelknoten abgelesen.

$$fn = root.positive - rule.tp$$
$$tn = root.negative - rule.fp$$

Dabei steht *root.positive* und *root.negative* für die Werte im Wurzelknoten und *rule.tp* sowie *rule.fp* für die zuvor ermittelten *tp* und *fp* der Regel. Für das gegebene Beipiel ist $fn = 6 - 4 = 2$ und $tn = 4 - 0 = 4$. Der gesamte zuvor beschriebene Ablauf ist in Abbildung 3.2 dargestellt. Insgesamt ergibt sich die Konfusionsmatrix in Tabelle 3.4. Wird die Erstellung und Traversierung des Baumes innerhalb der EVALUATERULE-Mehtode durchgeführt, ergibt sich das Problem, dass keine Leistungssteigerung erreicht werden kann. Allein für die Erstellung des Baumes muss der Algorithmus zwei mal durch die Datenmenge und anschließend noch durch den Baum, um die Konfusionsmatrix zu bestimmen. Die von *SeCo* verwendete Version hingegen muss nur einmal durch die Daten iterieren. Daher muss die Anzahl wie oft ein Baum erstellt wird vermindert werden. Dabei stößt man auf das Problem, dass der Baum den Regelkopf, also den Klassenwert, der vorhergesagt werden soll, benötigt. Ohne diesen kann der Baum die Werte innerhalb des Knoten nicht richtig setzen. Ein zwei Klassenproblem stellt da noch kein Problem dar, da die *"positive"* und *"negative"* Werte einfach vertauscht werden können. Bei einem Mehrklassen-Problem muss hingegen jeder Fall einzeln betrachtet werden. Ein naiver Ansatz wäre es



(a) *SD-Tree* für die Klasse +

(b) *SD-Tree* für die Klasse −

(c) *SD-Tree* für die Klasse ♢

Abbildung 3.3.: *SD-Trees* zu allen möglichen Klassenwerten in Datensatz 3.5 (ohne *Header Table* und *Node-Links*)

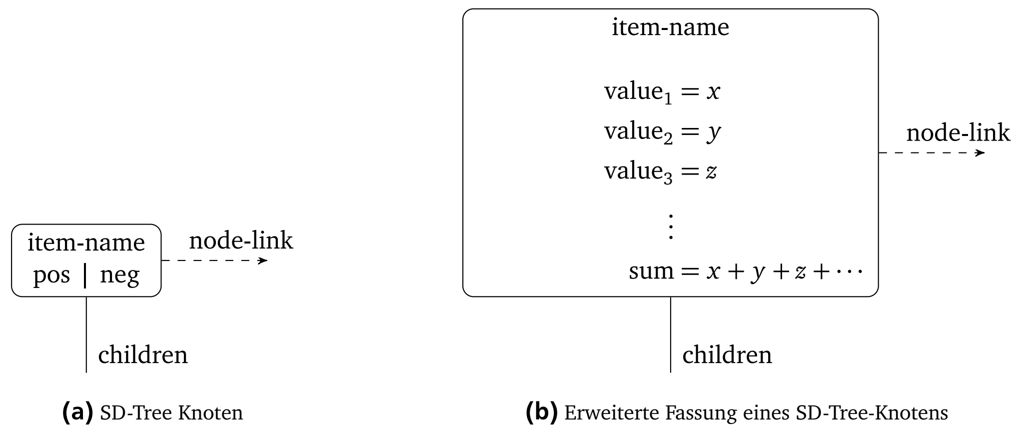
am Anfang für jeden Klassenwert im Datensatz einen eigenen Baum zu erstellen und diese passend zur jeweiligen Regel in EVALUATERULE zu nutzen. Dazu betrachte man den erweiterten Datensatz 2.1 in Tabelle 3.5. Der Datensatz besteht aus 15 Instanzen und hat drei Klassenwerte (+, −, ♢). Zur Erstellung der Bäume wird eine Klasse als positiver Fall betrachtet und die beiden restlichen als negativer Fall. Die daraus resultierenden *SD-Trees* sind in Abbildung 3.3 dargestellt. Zur Vereinfachung wurden die *Header Table* sowie die *Node-Links* zwischen den Knoten weggelassen. Diese sind für alle drei Bäume identisch, da diese nur von der Struktur des Baumes abhängen. Somit ist der Aufbau für jeden Klassenwert gleich. Die einzige Änderung sind die *"positive"* und *"negative"* Werte in den Knoten.

3.2.1 Erweiterter *SD-Tree*

Aus dem Schluss des vorherigen Abschnitts ergibt sich eine Erweiterung, die die Anzahl der Erstellungen eines Baumes innerhalb von SEPARATEANDCONQUER weiter verringert. In den Knoten werden nicht die jeweiligen *"positive"* und *"negative"* Werte gespeichert, sondern wie häufig die Konditionen (in Kombination mit den überliegenden Knoten) für den jeweiligen Klassenwert auftreten. Aus diesem Schluss wird die Struktur des *SD-Tree* ein letztes Mal erweitert. Der Knoten beinhaltet nun eine Tabelle für jeden Klassenwert und wie häufig dieser im jeweiligen Pfad vorkommt. In Abbildung 3.4 sind der *SD-Tree* und die erweiterte Fassung veranschaulicht. In Abbildung 3.4b steht *item-name*, wie auch zuvor, für eine beliebige Kondition. Die Einträge $value_i$ für einen konkreten Klassenwert. Für die genutzten Beispieldaten entspricht dies +, − und ♢. Die Zuordnungen x, y und z zu den einzelnen Werten besagen wie häufig dieser Eintrag in diesem Knoten vorkommt. Ein Knoten kann so beliebig viele Klassenwerte speichern. Abschließend beinhaltet der Knoten ein weiteres neues Feld *sum*. Dieses speichert wie häufig die Klassenwerte insgesamt in diesem Knoten vorkommen. Dies wird bereits bei der Erstellung des Baumes festgestellt und verhindert so ein weiteres Durchgehen durch die Tabelle während der Evaluation einer Regel.
Das Feld *sum* selbst ist dazu notwendig um die $negative_i$ Werte zu bestimmen. Für das Klassenattribut value_1 wird von der Summe im Knoten der zugehörige x-Wert abgezogen. Für die restlichen Klassenwerte folgt dies analog.

$$negative_1 = \text{sum} - x, \qquad negative_2 = \text{sum} - y, \qquad negative_3 = \text{sum} - z, \quad \cdots$$

Nachdem die abschließende Erweiterung des *SD-Tree* eingeführt wurde, werden die Erstellung und Evaluation dieses Baumes formal festgehalten. Die Erstellung des Baumes verläuft analog zu dem eines *FP-Tree* bzw. *SD-Tree*. Beim Hinzufügen eines Knoten wird lediglich zwischen allen Klassenwerten unterschieden, sowie die Summe aller Klassenwerte im Knoten um eins erhöht.



(a) SD-Tree Knoten

(b) Erweiterte Fassung eines SD-Tree-Knotens

Abbildung 3.4.: Vergleich eines SD-Tree - Knotens zu der Erweiterung

Algorithm 4 Erweiterte Baumerstellung

```
 1: procedure CREATETREE(transaction database D)
 2:     Scan database D once and collect F, the set of frequent conditions, and the occurence of each condition in D
 3:     Sort F in occurence-descending order as FList, the list of frequent conditions
 4:
 5:     Create the empty root of the tree T
 6:     for each instance inst in D do
 7:         Select and sort all conditions in inst according to the order of FList
 8:         Let the sorted condition list in inst be [p|P], where p is the first element and P is the remaining list
 9:
10:         N = root of the FP-Tree T
11:         do
12:             if N has a child node C and C.item-name == p.item-name then
13:                 incerement C's count for the class value by 1
14:                 increment C's sum by 1
15:             else
16:                 create a new node C, with its counts for all class values initilized to 0
17:                 incerement C's count for the class value by 1
18:                 initialize C's sum with 1 and set its parent link to N,
19:                 link it to the nodes with the same item-name via node-link
20:             N = C
21:             remove the first element p from the list [p|P]
22:         while P is nonempty
```

Algorithmus 4 erzeugt einen erweiterter *SD-Tree*, der nachfolgend zur Evaluierung einer Regel genutzt wird. In Zeile 2 - 5 werden die Konditionen und deren Auftreten in der Datenmenge gesammelt und gezählt. Es ist zu beachten, dass anders als bei der Erstellung des *FP-Tree* oder des *SD-Tree* alle Konditionen genutzt werden. Beginnend in Zeile 6 werden alle Instanzen der Datenmenge erneut verarbeitet und als Pfad in den Baum eingefügt. Die aufgetretenen Attribut-Wert-Paare werden anhand der sortierten Liste *FList* angeordnet und in den Baum als Knoten in einem Pfad eingefügt. In Zeile 12 - 19 wird geprüft, ob auf dem Pfad bereits ein Knoten mit dem jeweiligen Attribut-Wert-Paar existiert. Ist dies der Fall, wird der *Count* in der Tabelle des Knotens mit dem Klassenwert der Instanz um eins erhöht. Weiter wird das Feld *sum* inkrementiert. Ab Zeile 20 wird der bearbeitete Knoten zum neuen betrachteten Knoten und die entsprechende Kondition wird aus der List der einzufügenden Konditionen für den Pfad entfernt. Dies wird wiederholt bis die Liste leer ist.
Dies soll nun beispielhaft für die ersten beiden Instanzen im Datensatz 3.5 illustriert werden. Zunächst wird gezählt wie häufig jedes Attribut-Wert-Paar vorkommt und in einer absteigend sortierten Liste gespeichert. Für das gegebene Beispiel ergibt sich *FList* durch:

$$FList = [(a_3 = 1) : 11, (a_1 = 0) : 8, (a_2 = 0) : 7, (a_2 = 1) : 7, (a_1 = 1) : 6, (a_3 = 0) : 3]$$

Mit dieser Liste werden die einzelnen Instanzen als Pfade in den Baum eingefügt. Der Vorgang wie einzelne Knoten eingefügt und verlinkt werden ist bereits in Algorithmus 2 vorgestellt worden und wird hier nicht weiter vertieft. In der ersten Instanz sind die Konditionen $[(a_1 = 0), (a_2 = 1), (a_3 = 1)]$. Nach *FList* ist die Reihenfolge für den Pfad gegeben

durch $[(a_3 = 1) \rightarrow (a_1 = 0) \rightarrow (a_2 = 1)]$. Zum Zeitpunkt der ersten Instanz ist der Baum noch leer. Daher werden die Knoten neu erstellt und die Einträge für die neu hinzugekommenen Konditionen in die *Header Table* eingefügt. Der resultierende Baum nach dem die erste Instanz verarbeitet wurde ist in Abbildung 3.5a dargestellt. Anschließend werden diese Schritte für die zweite Instanz wiederholt. In der zweiten Instanz sind die Konditionen $[(a_1 = 0), (a_2 = 0), (a_3 = 1)]$. Nach *FList* ist die Reihenfolge des Pfades $[(a_3 = 1) \rightarrow (a_1 = 0) \rightarrow (a_2 = 0)]$. Beim Einfügen werden die Zähler für die jeweilige Klasse der Instanz in den Knoten jeweils um eins erhöht und neu hinzugefügte Knoten mit der *Header Table* verlinkt. Das Ergebnis ist in Abbildung 3.5b dargestellt. Wird dies für alle Instanzen im Baum wiederholt, entsteht der dargestellte Baum in Abbildung 3.5c.
Der erstellte Baum soll in der *Separate and Conquer* Implementierung *SeCo* zur Regelevaluation genutzt werden. Wird eine Regel zur Theorie aufgenommen, werden alle abgedeckten Beispiele aus der Datenmenge entfernt. Dementsprechend müsste der Baum ebenfalls angepasst werden, um dem aktualisierten Datensatz zu entsprechen. Dazu könnte der Baum auf der angepassten Datenmenge neu erstellt oder die Werte in den Knoten angepasst werden. Dabei gilt zu beachten, dass ein neu erstellter Baum kompakter ist als ein Baum bei dem nur die Werte in den Knoten angepasst werden, da u. a. Konditionen in der Datenmenge wegfallen könnten und damit auch ein kompletter Eintrag im Baum wegfällt. Ein weiterer Punkt ist die Tatsache, dass die Knoten neu zusammengefasst werden und damit der Baum potenziell weniger Knoten benötigt. Argumente für die Anpassung der Werte sind hingegen der geringere Aufwand. Zur Erstellung eines neuen Baumes muss der komplette Datensatz erneut zweimal verarbeitet werden. Bei der Anpassung werden nur ein Bruchteil der Pfade bearbeitet. Wie bereits zuvor angemerkt sollte zur Effizienzsteigerung die Erstellung des Baumes vermindert werden. Daher wird nachfolgend ein Algorithmus vorgestellt, der die Werte im Baum effizient anpasst und so die Möglichkeit bietet den Baum weiter effektiv zu verarbeiten.
Zur Anpassung der Datenmenge werden nur die Instanzen entfernt, die von einer Regel abgedeckt werden. Diese Instanzen wurden bei der Regelevaluierung bereits festgestellt. Daher könnten die abgedeckten Pfade bei der Regelevaluierung gespeichert und so später direkt herausgesucht und verändert werden. Zur Speicherung des kompletten Pfades reicht dabei ein Vermerk auf den tiefsten betrachteten Knoten aus. Alle Knoten oberhalb des gespeicherten Knotens liegen automatisch auf dem Pfad. Dabei wurde aber noch nicht beachtet, dass die herausgesuchten Pfade bei einer Regelevaluierung nicht immer einem kompletten Pfad vom Wurzelknoten bis zu einem Blattknoten entsprechen. Bei der Evaluation einer Regel mit weniger Konditionen als es Attribute gibt, könnten potenziell nur Knoten verarbeitet werden, die sehr weit oben im Baum vertreten sind.
Betrachtet man Datensatz 3.5 mit dem vollständigen Baum in Abbildung 3.5c sowie der Regel $a_1 = 0 \Rightarrow a_4 = +$, werden nach dem zuvor vorgestellten Verfahren zur Regelevaluierung nur zwei Pfade bearbeitet:

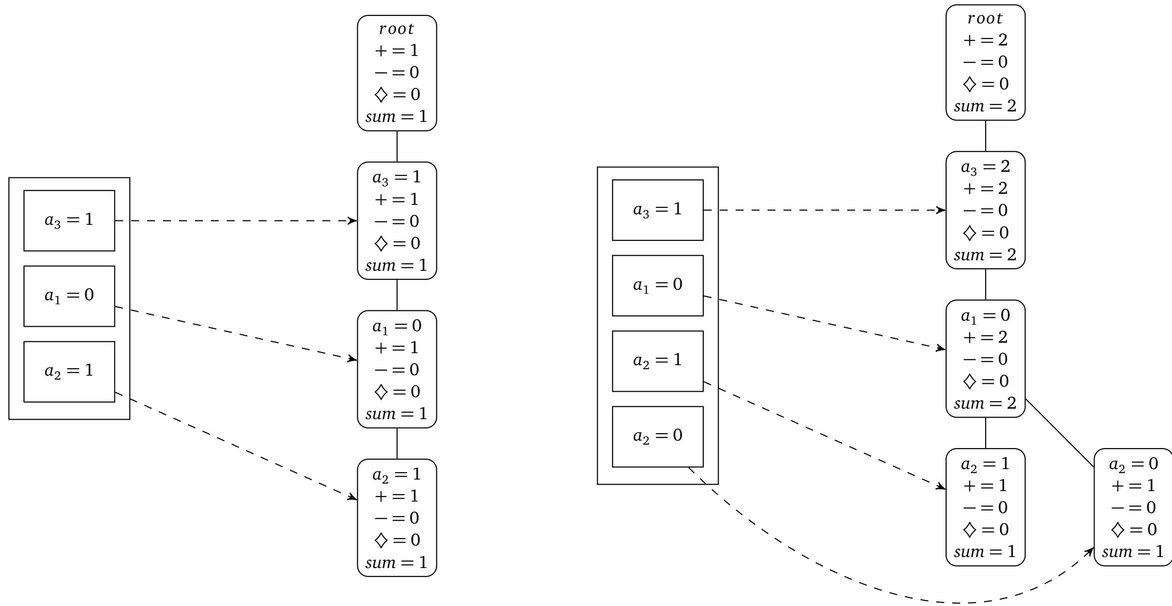
$$[(a_1 = 0) : + = 5 \rightarrow (a_3 = 1) : + = 6] \qquad \text{und} \qquad [(a_1 = 0) : + = 0]$$

Die Knoten unterhalb von a_1 werden bei der Regelevaluierung ignoriert und müssen bei der Anpassung der Knotenwerte gesondert behandelt werden. Hierbei nutzen wir die zuvor festgestellten Eigenschaften des Baumes. Die Knotenwerte sind die Summe der Blattknotenwerte. Alle Knoten unterhalb des gespeicherten Knoten werden daher ebenfalls automatisch abgedeckt. Diese könnten daher mit dem gespeichert Knoten selbst komplett gelöscht werden. Diese Schlussfolgerung gilt weder für einen normalen SD-Tree noch für den FP-Tree. Die Löschung der Knoten würde aber zu einem weiteren Problem führen. Die *Node-Links* müssten dann ebenfalls angepasst werden. Daher werden nur die Werte auf 0 gesetzt und so vermerkt, dass der Knoten nicht mehr genutzt wird und bei der nächsten Regelevaluierung übersprungen werden kann. Die formal festgehaltene Vorgehensweise ist dabei in Algorithmus 5 festgehalten.

Algorithm 5 Baumanpassung

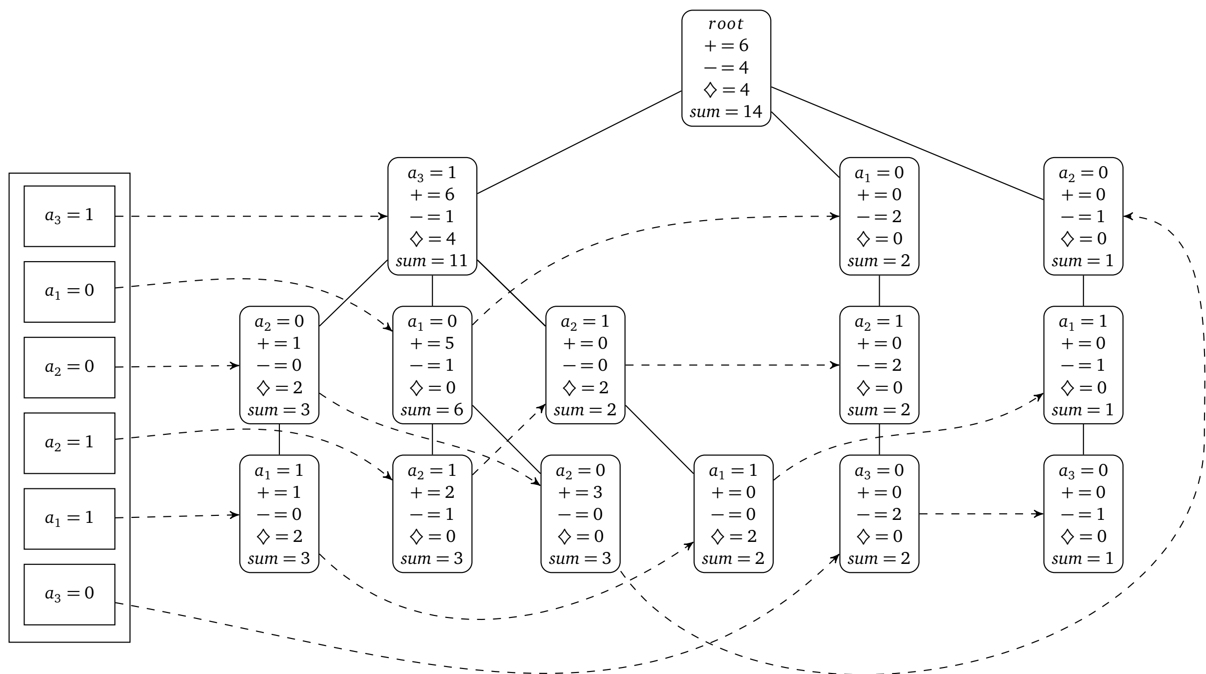
```
1: procedure ADJUSTTREE(tree t, rule r, nodeList covered)
2:     if r has no conditions then
3:         remove all nodes from the tree;
4:         return ;
5:
6:     for each baseNode in covered do
7:         reset all values in the nodes below baseNode (all counts to 0);
8:         climb the path upwards and subtract the corresponding baseNode values from the node;
9:         reset baseNode;
```

In Zeile 1 - 3 wird der Fall abgefangen, wenn eine Regel keine Bedingungen hat und somit alle Instanzen von der Regel abgedeckt werden. Daher wird ein komplett neuer Baum ohne Knoten erstellt, was für die Entfernung aller Instanzen steht. In Zeile 6 - 9 werden einerseits alle Knoten unterhalb des gespeicherten Knotens zurückgesetzt, andererseits werden die abgedeckten Instanzen aus den oberen Pfadknoten entfernt. Dazu werden die Zähler der jeweiligen Klassen im Knoten durch die Werte im gespeicherten Knoten verringert. Für das zuvor genutzte Beispiel würden für den ersten Pfad die Kind-Knoten zurückgesetzt und für den einzigen Pfadknoten oberhalb würde folgen, dass für die Klasse “ + “ $= 6 - 5 = 1$ der



(a) 1.Instanz wurde eingefügt

(b) 2.Instanz wurde eingefügt



(c) Baum nachdem alle Instanzen eingefügt wurden

Abbildung 3.5.: Die Bäume nach den ersten zwei Schritte sowie der fertige Baum in (c)

neue Wert ist und für die Klasse “−“ $= 1 - 1 = 0$. Die Klasse “◇“ bleibt im Pfadknoten unverändert. Abschließend werden noch die Werte im Wurzelknoten angepasst, damit dieser weiterhin die Gesamtverteilung in Baum speichert. Für den zweiten Pfad werden alle Knoten nur zurückgesetzt, da es bereits der oberste Knoten ist. Der daraus resultierende Baum ist in Abbildung 3.6 zu sehen. Die *Header Table* sowie die *Node-Links* wurden dabei zur Übersicht weggelassen, da sich diese nicht ändern.

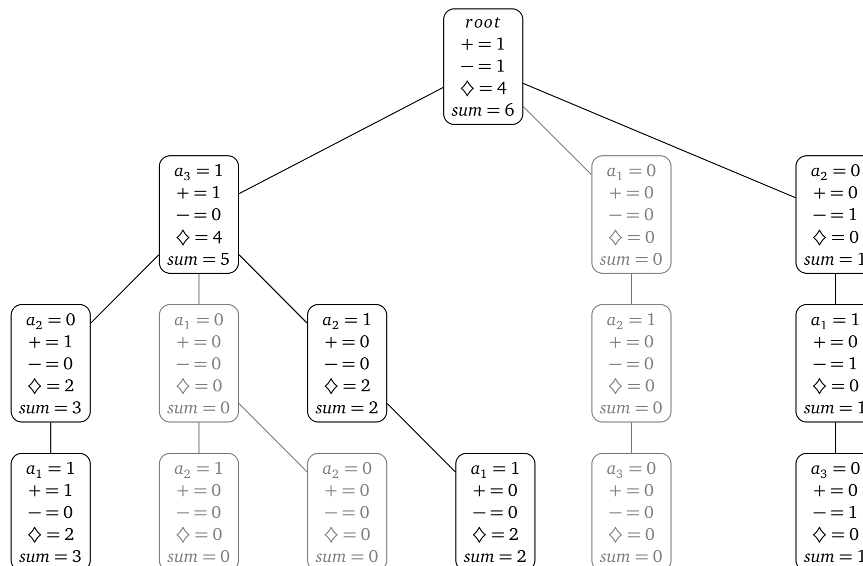


Abbildung 3.6.: Der angepasste Baum für die Regel $a_1 = 0 \Rightarrow a_4 = +$

3.2.2 Evaluierung mit einem erweiterten SD-Tree

Nachdem alle Besonderheiten des vorliegenden Problems besprochen, die Struktur erweitert und die Baumanpassung vorgestellt wurde, wird nachfolgend die zuvor vorgestellte Vorgehensweise zur Evaluierung einer Regel auf dem *SD-Tree* auf die erweiterte Fassung angepasst und formal in Algorithmus 6 festgehalten. Dieser geht dabei von der erweiterten Variante des *SD-Tree* aus und speichert direkt die abgedeckten Knoten ab, damit diese zur Baumanpassung genutzt werden können.

Algorithm 6 Regelevaluierung durch den erweiterten SD-Tree

```
 1: procedure EVALRULE(tree t, rule r)
 2:     p = 0; n = 0;
 3:     if r has no conditions then
 4:         return p and n from the root node;
 5:
 6:     condition = least frequent rule condition based on the sorted list FList;
 7:     node = get node from the header table based on condition;
 8:     while node != null do
 9:         if node is used (sum > 0) traverse upwards and check whether every parent node is covered by the rule
10:         if covered and used then
11:             incerement p by the count in the node for the rule head condition;
12:             incerement n by the subtraction of sum and the count for the rule head condition;
13:             add node to the covered base nodes of rule r
14:         node = next node from the node-link reference;
15:     true positive = p; false positive = n;
16:     false negative = root.pos(rule head condition) - p;
17:     true negative = root.sum - root.pos(rule head condition) - n;
18:     return HEURISTIC(tp, fp, fn, tn);
```

Die Variablen p und n stehen für die *positiven* und *negativen* Abdeckungen der Regel und werden am Anfang mit 0 initialisiert. In den Zeilen 3 - 4 wird der Fall einer leeren Regel abgefangen. Hat eine Regel keine Konditionen, muss der Baum nicht traversiert werden, da alle Instanzen abgedeckt werden. Hat eine Regel mindestens eine Kondition, müssen alle Instanzen und damit Pfade des Baumes ermittelt werden, die durch die Regel abgedeckt werden. Dazu wird die Kondition der Regel ermittelt, die am tiefsten im Baum auftritt. Dies geschieht auf Basis der Liste *FList*, die die Reihenfolge der Konditionen bei der Erstellung des Baumes gespeichert hat. Anschließend werden alle Knoten, die diese Kondition beinhalten mit Hilfe der *Header Table* und den *Node-Links* in den Knoten herausgesucht. Handelt es sich bereits um einen angepassten Baum, wird mit der Abfrage, dass das Feld $sum > 0$ ist abgedeckt, dass nicht mehr verwendete Knoten übersprungen werden. In Zeile 10 - 13 werden anschließend wie Werte der Kondition im Regelkopf

dem Knoten entnommen und auf die bisherige Summe von p addiert. Für das Feld n wird der *negative*-Wert, wie in Abschnitt 3.2.1 gezeigt, ermittelt und anschließend ebenfalls zur Variable n addiert. Da zudem der Pfad von der Regel abgedeckt wird, wird der tiefste Knoten des besuchten Pfades für die Regel gespeichert, damit dieser, wenn die Regel zur Theorie aufgenommen wird, bei der Anpassung des Baumes verwendet werden kann. Abschließend werden in den Zeilen 15 - 18 die benötigen tp, fp, fn, tn berechnet und innerhalb der Heuristik verwendet.
Eine Veranschaulichung anhand eines Beispieles wird hier ausgelassen, da die Vorgehensweise und ein Beispiel bereits in Abschnitt 3.2 vorgestellt wurden. Die einzigen Änderungen zum vorherigen Vorgehen sind:

- Das Feld *sum* wird auf $sum > 0$ überprüft, damit nicht mehr verwendete Knoten übersprungen werden.
- Jeder abgedeckte Pfad wird als solcher (indirekt) für jede Regel gespeichert.
- Die *"positive"* und *"negative"* Werte sind in einem erweiterten SD-Tree gegeben durch:
  - $positive$ = Knoteneintrag in der Tabelle zum Klassenwert des Regelkopfes
  - $negative = sum - positive$

3.3 Überischt zur SeCo-Vorgehensweise mit einem erweiterten SD-Tree

Algorithm 7 Angepasster *Separate and Conquer* - Algorithmus von [7, S.10]

```
1:  procedure separateAndConquerTree(Examples)
2:      Theory = ∅
3:      Tree = createTree(Examples)
4:      while positive(Examples) ≠ ∅ do
5:          Rule = findBestRuleTree(Examples, Tree)
6:          Covered = cover(Rule, Examples)
7:          if ruleStoppingCriterion(Theory, Rule, Examples) then
8:              exit while loop
9:          Examples = Examples \ Covered
10:         Tree = adjustTree(Rule)
11:         Theory = Theory ∪ Rule
12:     Theory = postProcess(Theory)
13:     return Theory

1:  procedure findBestRuleTree(Examples, Tree)
2:      InitRule = initializeRule(Examples)
3:      InitVal = evaluateRuleTree(InitRule, Tree)
4:      BestRule = <InitVal, InitRule>
5:      Rules = {BestRule}
6:      while Rules ≠ ∅ do
7:          Candidates = selectCandidates(Rules, Examples)
8:          Rules = Rules \ Candidates
9:          for each Candidate ∈ Candidates do
10:             Refinements = refineRule(Candidate, Examples)
11:             for each Refinement ∈ Refinements do
12:                 Evaluation = evaluateRuleTree(Refinements, Tree)
13:                 unless stoppingCriterion(Refinements, Examples)
14:                     NewRule = <Evaluation, Refinement>
15:                     Rule = insertSort(NewRule, Rules)
16:                     if NewRule > BestRule
17:                         BestRule = NewRule
18:         Rules = filterRules(Rules, Examples)
19:     return BestRule
```

Wie bereits beschrieben, soll der erweiterte Ansatz einer Regelevaluierung anhand eines Baumes auf der *Separate and Conquer* - Implementierung *SeCo* getestet werden. In Algorithmus 7 sind die Zeilen 3,5 und 10 in separateAndConquer-Tree sowie 3 und 12 in findBestRuleTree hervorgehoben, da diese die einzigen Änderungen im Vergleich zum originalen

Ansatz beinhalten. In separateAndConquerTree Zeile 3 wird der Baum wie in Algorithmus 4 beschrieben erstellt und in Zeile 10 wie in Algorithmus 5 beschrieben angepasst. Die Funktion findBestRuleTree nutzt neben der Datenmenge zusätzlich einen erweiterten SD-Tree. Dieser kann in den Zeilen 3 und 12, wie in Algorithmus 6 beschrieben, zur Evaluation einer Regel verwendet werden.
Es ist anzumerken, dass die *SeCo*-Implementierung nicht mehr vollständig dem gezeigten Ansatz in Algorithmus 1 entspricht und daher auch minimale Änderungen in der konkreten Implementierung aufzufinden sind. Diese beeinflussen aber weder den vorgestellten Ansatz noch die Vorgehensweisen zur Erstellung eines Baumes sowie dessen Nutzung. Im Anhang ist eine konkrete Implementierung in Java innerhalb des *SeCo*-Frameworks zu finden.

3.4 Zusammenfassung

In diesem Kapitel wurde zunächst vorgestellt wie die originale Version von *SeCo* eine Regel evaluiert. Dazu geht die entsprechende Methode jedes Mal durch den Datensatz und sammelt so die Werte einer Konfusionmatrix. Diese beinhaltet die *true positives*, *false positives*, *false negatives* und *true negatives* einer Regel. Damit die Regeln anschließend miteinander verglichen werden können, wird eine Heuristik wie *Precision* angewendet.
Darauffolgend wird der Ansatz anhand eines SD-Trees vorgestellt. Da die Urversion auf *SeCo* dazu führen würde, dass keine Effizienzsteigerung erreicht werden kann, wird dieser so erweitert, dass die Erstellung des Baumes so selten wie möglich notwendig ist. Dazu wird in jedem Knoten statt der positiven und negativen Werte eines einzigen Klassenwerten, das Auftreten eines jeden Klassenwertes gespeichert. Zudem wurde gezeigt, dass dies möglich ist, da die Struktur des Baumes unabhängig vom Klassenwert selbst ist.
Des Weiteren wurde der Fall besprochen, dass die betrachtete Datenmenge während der Erstellung der Regelmenge kleiner wird und dies innerhalb des Baumes wiedergegeben werden muss. Dazu wurde Algorithmus 5 konzipiert und vorgestellt.
Abschließend wurde die Evaluation einer Regel anhand eines Baumes formal in Algorithmus 6 festgehalten und in Abschnitt 3.3 aufgezeigt wie die angepasste Version von *Separate and Conquer* den Baum nutzt.

4 Evaluation

In diesem Kapitel wird ein Vergleich zwischen der originalen Version von *SeCo* und der baumnutzenden Version durchgeführt. Der Vergleich beschränkt sich dabei auf Zeitmessungen anhand von neun Datensätzen. Die Datensätzen haben dabei verschiedene Größen sowie Eigenschaften. Zunächst werden die Datensätze eingeführt. Anschließend wird ein Vergleich der Gesamtzeit durchgeführt. Die dabei auffallenden Besonderheiten werden abschließend weiter untersucht. Es wird festgehalten inwieweit Dateneigenschaften dazu führen können, dass die Evaluation anhand des Baumes langsamer wird als die redundante Traversierung der Daten in der originalen Version von *SeCo*.

4.1 Datensätze

Beim Vergleich der Geschwindigkeiten zwischen den beiden Ansätzen zur Regelevaluation werden Datensätze mit 450 — 10 000 Instanzen genutzt. Diese haben eine unterschiedliche Anzahl an Attributen, welche aber alle nominal sind. Datensätze, die nicht nominale (numerische) Attribute besitzen, wurden mit Hilfe von *Weka* zuvor diskretisiert. In Tabelle 4.1 ist eine Übersicht aller genutzter Datensätze zu sehen. Diese bietet einen Überblick über die Anzahl der Instanzen, die Anzahl der Attribute in einer Instanz, ob der Datensatz zur Evaluation diskretisiert wurde sowie eine Übersicht über die Regelanzahl und die durchschnittliche Regellänge, die durch einen *TopDown-Refiner* und der Heuristik *Precision* erstellt wurden. In den nachfolgenden Geschwindigkeitsanalysen wurden die Regeln stets mit einem *TopDown-Refiner* sowie der Heurisitik *Precision* erstellt. Wurden die Messungen mit anderen Verfahren oder Heuristiken durchgeführt wird dies explizit erwähnt.

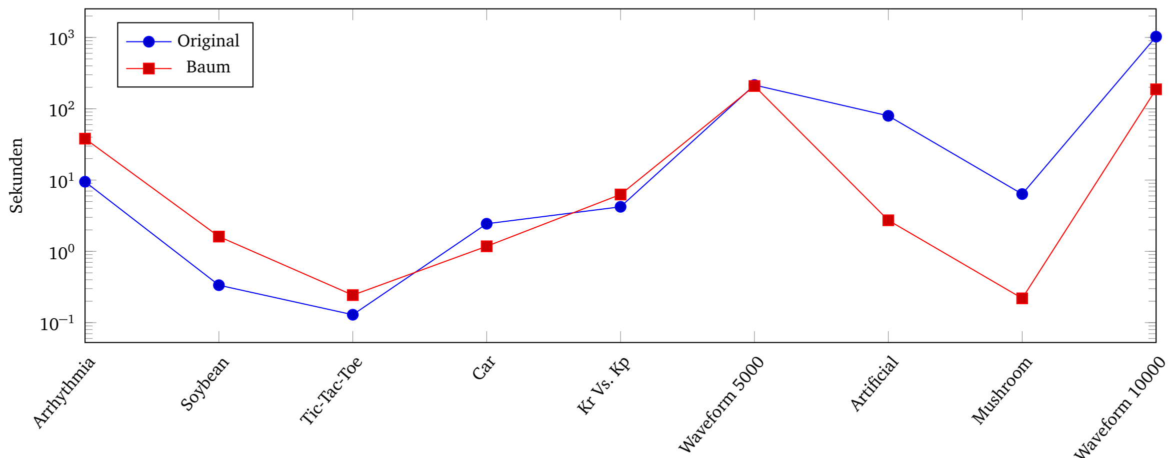
	Arrhythmia	Soybean	Tic-Tac-Toe	Car	Kr Vs. Kp	Waveform 5 000	Artificial	Mushroom	Waveform 10 000
Instanzen	452	683	958	1728	3196	5000	5109	8124	10000
Attribute	280	36	10	7	37	41	8	23	41
Diskretisiert	×					×	×		×
Regeln	128	69	24	161	50	688	259	11	690
∅ Regellänge	1.46	2.43	3.75	5.00	3.34	3.28	3.22	1.18	2.97

Tabelle 4.1.: Datenübersicht und Lernergebnisse

Durch Betrachtung der Dateneigenschaften in Tabelle 4.1 sollte direkt auffallen, dass Datensatz *Arrhythmia* mit 280 Attributen absolut und relativ gesehen die meisten Attribute aufweist. Das Gegenteil dazu ist der Datensatz *Artificial*, welcher relativ gesehen, auf jede Instanz, die wenigsten Attribute hat. Wie sich dies auf die Zeiten auswirkt kann, wird nachfolgend weiter untersucht.

4.2 Geschwindigkeitsanalyse

In Abbildung 4.1 sind die benötigten Gesamtzeiten zur Erstellung einer Theorie / Regelmenge dargestellt. In der Tabelle darunter sind die einzelnen Werte nochmal explizit vermerkt. Man beachte, dass zur Übersichtlichkeit eine logarithmische Skalierung gewählt wurde. Im Diagramm ist zu sehen, dass für die Datensätze *Arrhythmia, Soybean, Tic-Tac-Toe, Car* sowie *Kr Vs. Kp* keine wirkliche Geschwindigkeitsverbesserung erreicht werden kann. Für die Datensätze *Arrhythmia, Soybean, Tic-Tac-Toe* und *Kr Vs. Kp* ist die *SeCo*-Variante, die den Baum nutzt, sogar langsamer. Ab Datensatz *Waveform 5000* ist die *SeCo*-Variante, die den Baum nutzt, konsistent schneller.
Ein wichtiger Punkt, der zunächst beachtet werden muss, ist die Tatsache, dass das *SeCo*-Framework nicht komplett auf den Baum umgestellt wurde. *SeCo* nutzt weiterhin den Datensatz und verändert diesen entsprechend. Der Baum wird wie bereits in den vorherigen Kapiteln erwähnt nur in der Methode EVALUATERULE genutzt. Dadurch werden bestimmte Operationen mehrfach durchgeführt. So müssen die abgedeckten Instanzen sowohl aus der Datenmenge als auch aus dem Baum entfernt werden. Zudem kommen neben den restlichen Operationen in *SeCo* die Erstellung sowie Anpassung des Baumes nur zusätzlich hinzu. Es werden, bis auf EVALUATERULE, keine Operationen ersetzt.
Beim Zeitvergleich in Abbildung 4.1 fällt auf, dass bei kleineren Datensätzen, in diesem Kontext $< 5\,000$ Instanzen, die Baumvariante nicht mehr zuverlässig schneller ist als ein normales Durchgehen durch den Datensatz. Mögliche Gründe



	Arrhythmia	Soybean	Tic-Tac-Toe	Car	Kr Vs. Kp	Waveform 5 000	Artificial	Mushroom	Waveform 10 000
Original	9.47	0.34	0.13	2.44	4.21	215.20	79.59	6.36	1026.93
Baum	38.08	1.61	0.24	1.18	6.27	208.40	2.73	0.22	187.46

Abbildung 4.1.: Benötigte Gesamtzeit in Sekunden (logarithmisch)

dafür könnte der zusätzliche Aufwand zur Erstellung und Anpassung des Baumes sein. Eine andere Möglichkeit könnte ein geringerer bzw. nicht vorhandener Geschwindigkeitsvorteil bei der Evaluation einer Regel sein. Für die Datensätze *Arrhythmia* und *Soybean* braucht die Baumvariante mehr als das vierfache der Zeit der originalen *SeCo*-Version. Ein weiterer Punkt, der dabei auffällt, ist die Anzahl der Attribute in den Datensätzen. Die Datensätze *Tic-Tac-Toe* bis *Kr Vs. Kp* sind zwar unterschiedlich groß aber der Zeitunterschied zwischen den beiden *SeCo*-Varianten ist minimal. Selbst bei *Tic-Tac-Toe*, welches nur minimal größer ist, ist kaum ein Zeitunterschied zu beobachten.
Ein weiterer Aspekt, der die Evaluation zu beeinflussen scheint, ist die Attributanzahl. In Tabelle 4.1 ist zu sehen, dass die Datensätze *Arrhythmia* und *Soybean*, auf die Anzahl der Instanzen betrachtet, sehr viele Attribute haben. Hat jedes Attribut nur zwei verschiedene Werte müssten allein schon bei *Arrhythmia* für jede Instanz ein eigener Pfad erzeugt werden. Dementsprechend wird nur wenig bis gar nichts zusammengefasst, was einen elementaren Vorteil des Baumes eliminiert und damit dazu führt, dass bei der Regelevaluation mehr Knoten bearbeitet werden müssen. Wie wichtig diese Eigenschaft der Komprimierung und damit die Anzahl der Attribute ist, sieht man in der benötigten Gesamtzeit der Datensätze *Waveform 5000* sowie *Waveform 10000*. Obwohl Datensatz *Waveform 10000* das Doppelte an Instanzen aufweist, ist die Erstellung der Regelmenge in der baumnutzenden *SeCo*-Variante schneller, als bei *Waveform 5000*. Eine mögliche Erklärung ist die Tatsache, dass zwar mehr Instanzen vorhanden sind, die Anzahl der Attribute aber gleich bleibt. Daraus folgen mehr ähnliche bis identische Instanzen. Da im Baum nach Häufigkeit sortiert wird, können die öfter auftretenden Attribut-Wert-Paare klarer und somit besser zusammengefasst werden. Dies führt, relativ gesehen, zu weniger Knoten, die während der Regelevaluation verarbeitet werden müssen. Dies ist aber nur ein Indiz für die Wichtigkeit der Komprimierung und erklärt noch nicht wieso die benötigte Gesamtzeit nicht nur konstant bleibt sondern sogar weniger wird.
Zusammengefasst ergeben sich folgende Punkte, die weiter untersucht werden:

1. Geschwindigkeitsvorteil der baumnutzenden *SeCo*-Variante in EVALUATERULE.
2. Aufwand zur Erstellung und Anpassung des Baumes und wie sich dies auf die Gesamtzeit auswirkt.
3. Auswirkung der Dateneigenschaften (#Attribut und #Instanzen) auf den Baum.

4.2.1 Geschwindigkeitsvergleich bei EVALUATERULE

In Abbildung 4.2 ist eine Übersicht zur Evaluationszeit verschiedener Regeln in der baumnutzenden *SeCo* Variante aufgeführt. Die dafür genutzten Regeln wurden zufällig erzeugt und waren nach keinem Muster, wie bei der Erstellung einer Regelmenge in *SeCo*. Die erzeugten Regeln haben dabei maximal 10 Konditionen. Bei 0 Konditionen besitzt die Regel nur einen Regelkopf und keine Bedingungen. Die Datensätze *Tic-Tac-Toe, Car* und *Artificial* haben jeweils nur 10, 7 und 8 Attribute (inklusive des Klassenattributes). Daher wurden für diese Datensätze nur Regeln mit jeweils 9, 6 und 7 Konditionen erstellt und evaluiert. Weiter ist zu entnehmen, dass für den Datensatz *Kr Vs. Kp* die Evaluation einer Regel

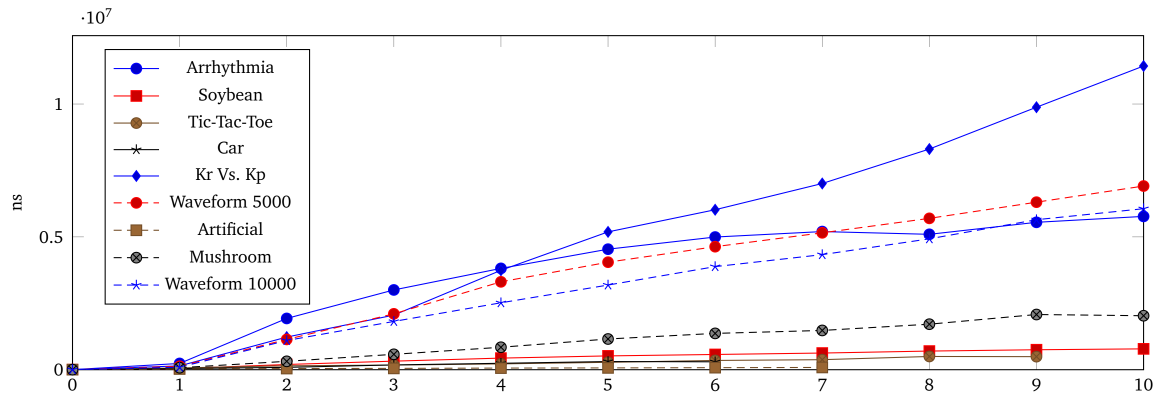


Abbildung 4.2.: Dauer einer Regelevaluation anhand eines Baumes mit verschiedenen Regellängen

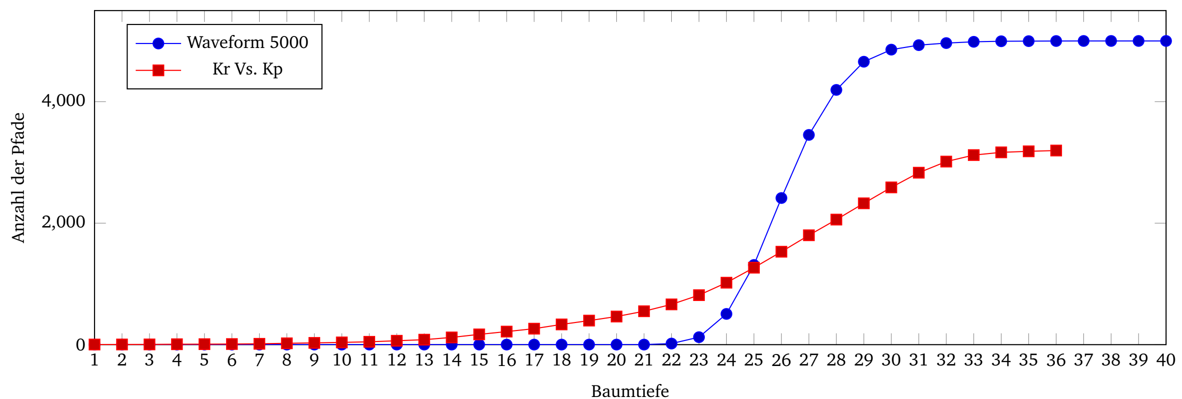


Abbildung 4.3.: Anzahl der Pfade zwischen *Kr Vs. Kp* und *Waveform* 5 000 zur jeweiligen Tiefe des Baumes

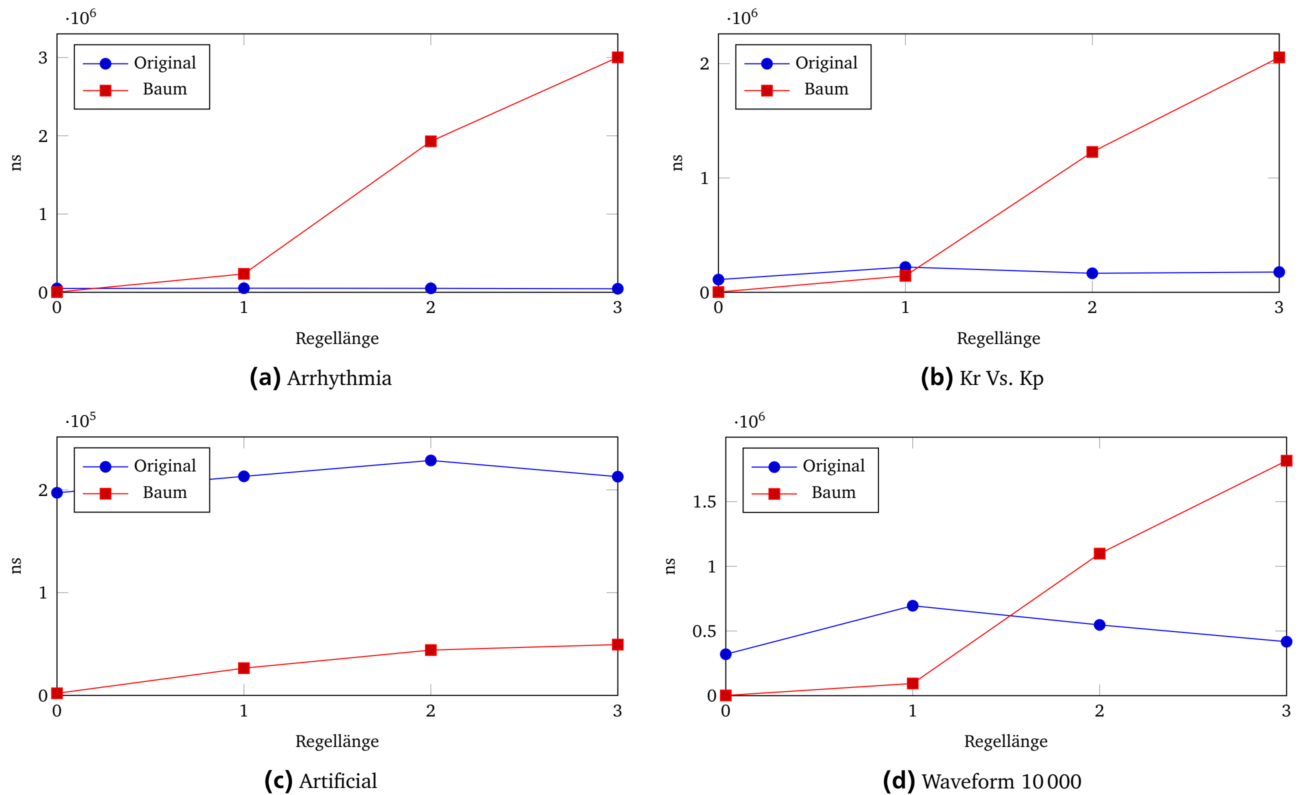
mit steigender Konditionszahl am längsten dauert und die Evaluation bei Datensatz Waveform 10 000 immer schneller ist, als der korrespondierende Datensatz *Waveform* 5 000, der aber nur die Hälfte der Instanzen aufweist.

Für Regeln mit mindestens 2 Konditionen ist ein lineares Wachstum zu beobachten. Desto mehr Konditionen eine Regel hat, desto mehr Zeit wird zur Evaluation einer Regel benötigt. Dies ergibt Sinn, wenn man das Verfahren zur Evaluation einer Regel betrachtet. Zur Evaluation müssen die Pfade von unten nach oben verarbeitet werden. Desto länger eine Regel ist, desto mehr Konditionen müssen auf dem Pfad verarbeitet werden. Des Weiteren ist bei einer längeren Regel die Wahrscheinlichkeit höher, dass zur Evaluation der Pfad von ganz unten und damit komplett verarbeitet werden muss.

Für Regeln mit nur einer Kondition benötigt das Verfahren kaum mehr Zeit, als für Regeln ohne Bedingungen. Dies kann dadurch erklärt werden, dass die entsprechenden Konditionen in Regelkörper durch die *Header Table* direkt referenziert werden können. Dadurch müssen, je nachdem wie häufig die Kondition im Datensatz vorkommen, nur wenige Knoten insgesamt bearbeitet werden und nicht ein gesamter Pfad.

Eine weitere Beobachtung ist, dass die benötigte Zeit nicht direkt von der Anzahl der Instanzen abzuhängen scheint. Zur Evaluation von Regeln im Datensatz *Kr Vs. Kp* wird mehr Zeit benötigt als bei Datensatz *Waveform* 5 000, welches mehr Instanzen aufweist aber die relative Anzahl von Attributen ähnlich ist. Außerdem hat ein Attribut bei *Kr. Vs. Kp* im Schnitt zwei mögliche Werte. Attribute in Datensatz *Waveform* 5 000 haben hingegen mindestens drei mögliche Werte. Daher kann auch ausgeschlossen werden, dass durch eine geringere Anzahl an Konditionen ein kompakterer Baum generiert wird. Eine mögliche Erklärung für den Zeitunterschied ist hingegen die Verteilung der Werte innerhalb der Attribute. Es könnten viele Pfade entstehen, die schlecht zusammengefasst werden. Diese Annahme wird, wenn man die benötigte Zeit für kleinere Regeln (≤ 3 Konditionen) betrachtet, verstärkt. Die benötigte Zeit ist dort für beide Datensätze sehr ähnlich. Wie bereits festgestellt ist die Wahrscheinlichkeit dafür, dass der komplette Pfad verarbeitet werden muss, für längere Regeln höher.

In Abbildung 4.3 ist für die Datensätze *Kr Vs. Kp* und *Waveform* 5 000 die Anzahl der Pfade zu einer gegebenen Tiefe dargestellt. Die Datensätze haben ohne das Klassenattribut jeweils 36 bzw. 40 Attribute. Die Tiefe des Baumes ist durch die Anzahl der Attribute beschränkt und endet bei *Kr Vs. Kp* bereits bei einer Tiefe von 36. Die Anzahl der Pfade steigt bei *Kr Vs. Kp* direkt stetig an. Bei *Waveform* 5 000 hingegen, ist bis zu einer Tiefe von 21 nur ein einziger Pfad vorhanden. Ab einer Tiefe von 22 beginnt auch der Baum für *Waveform* 5 000 in die Breite zu wachsen und hat bereits ab einer Tiefe



(a) Arrhythmia

(b) Kr Vs. Kp

(c) Artificial

(d) Waveform 10 000

Abbildung 4.4.: Geschwindigkeitsvergleich einer Regelevaluation zwischen der originalen und baumnutzenden Variante

von 25 genauso viele Pfade wie der Baum für *Kr Vs. Kp*, bei gleicher Tiefe. Die Bäume erreichen eine Maximalbreite mit jeweils 5000 bzw. 3196 Pfaden.
Aus dem Verlauf kann somit entnommen werden, dass *Waveform 5000* aufgrund der Verteilung der Daten stärker zusammengefasst werden konnte. Erst nachdem mehr als die Hälfte der möglichen Attribute bei *Waveform* 5 000 verarbeitet wurde, haben beide Bäume die gleiche Breite (Anzahl der Pfade). Es verwundert auch nicht, dass der Baum zu *Waveform* 5 000 mehr Pfade hat, da der zugrundeliegende Datensatz zum einen mehr Instanzen, als auch mehr Attribute insgesamt aufweist. Weiter kann aus der Maximalbreite gefolgert werden, dass jede Instanz im Datensatz einzigartig ist, da es für jede Instanz ein eigenen *kompletten* Pfad gibt.
Zudem kann aus Abbildung 4.3 entnommen werden wie wichtig eine *Threshold* ξ für die Effizienz der Baumtraversierung ist. Dadurch würde nur ein Bruchteil der Pfade entstehen und es ergebe sich ein relativ schmaler Baum. Bei den verwendeten Bäumen in Abbildung 4.3 könnten so, für einen Baum mit einer Tiefe von 25, ein Baum mit nur in etwa 1300 Pfaden entstehen. Dies würde jeweils etwa einem drittel bzw. etwa einem fünftel des ursprünglichen Datensatzes entsprechen.

4.2.2 Vergleich einer einzelnen Regelevaluation

In Abbildung 4.4 sind exemplarisch vier Datensätze gezeigt, bei denen Regeln einer bestimmten Länge auf beiden Varianten evaluiert wurden. Zu sehen ist dabei, dass für alle Visualisierungen die Evaluation einer Regel in der originalen *SeCo*-Variante in etwa konstant bleibt. Beim Baum hingegen steigt die benötigte Zeit linear an. Einzig bei *Artificial* ist die Evaluation anhand des Baumes immer schneller als auf der originalen Version. Eine weitere Beobachtung ist, dass die Zeit erst für Regeln mit mindestens zwei Konditionen stark ansteigt.
Führt man sich erneut die Vorgehensweise zur Evaluation einer Regel im Baum vor Augen, ergeben die Messungen Sinn. Für Regeln mit keiner oder nur einer Kondition, können direkt die betroffenen Knoten referenziert und bearbeitet werden. Es ist nicht notwendig den Baum von unten nach oben abzuarbeiten, da bereits der eine Knoten auf dem Pfad alle Konditionen der Regel abdeckt. Für Regeln mit mindestens zwei Konditionen hingegen, müssen auch die darüber liegenden Knoten evaluiert werden, um festzustellen ob der Pfad von der Regel abgedeckt wird. Dabei ist aber nicht sichergestellt, dass die Konditionen in der Regel im Baum nah beieinander liegen. Es kann so durchaus vorkommen, dass eine Regel aus Konditionen besteht, bei der sich eine Kondition sehr tief im Baum und die Andere sehr nah an der Wurzel befindet. So müsste zur Evaluation trotzdem fast der gesamte Pfad verarbeitet werden. Dies widerspricht aber auch nicht den vorherigen Interpretationen der Messungen. Zu Abbildung 4.2 wurde festgestellt, dass kürzere Regeln dazu führen,

dass weniger vom Pfad verarbeitet werden muss. Die Wahrscheinlichkeit zur Evaluation einer Regel weiter oben im Baum zu beginnen, ist bei weniger Konditionen sehr viel höher, als bei einer Regel mit mehr Konditionen. Dadurch ergeben sich zwangsweise kürzere Pfade, die evaluiert werden müssen und damit geringere Evaluationszeiten. Abschließend kann aus Abbildung 4.4 entnommen werden, dass bei der originalen *SeCo*-Variante die Evaluation einer Regel unabhängig von der Regellänge ist, da diese in etwa konstant bleibt. Diese Schlussfolgerung ergibt Sinn, da unabhängig von der Regellänge immer der komplette Datensatz verarbeitet werden muss.
Es ist noch nicht geklärt warum die Evaluation über den Baum bei längeren Regeln langsamer ist, statt über den Datensatz zu iterieren. Dazu muss sich vor Augen geführt werden, dass die Verarbeitung über den Datensatz um einiges einfacher gehalten ist. Zur Evaluation anhand des Baumes müssen die Konditionen in den Regeln jedes mal sortiert werden. Je länger eine Regel ist, desto mehr Aufwand bereitet dieser Schritt. Würden die Konditionen bereits bei der Erstellung in der richtigen Reihenfolge eingefügt werden, könnten die zusätzlichen Kosten einer Sortierung vermieden werden. Dabei ist aber zu beachten, dass sich jeder *RuleRefiner* in *SeCo* dynamisch an die Reihenfolge des Baumes anpassen müsste. Des Weiteren würde darunter die Lesbarkeit der Regeln leiden. Diese müssten am Ende wieder in die ursprüngliche Reihenfolge umgestellt werden. Ein weiterer Punkt ist die Bearbeitung der Knoten. Eine Instanz im Datensatz zu verarbeiten, ist schneller und einfacher als einen Pfad im Baum. Die Konditionen in der Regel greifen direkt auf die entsprechenden Attribute im Datensatz der jeweiligen Instanz zu. Für den Baum kann nicht gesagt werden wo sich die Attribute befinden. Dies führt dazu, dass auch Knoten mit anderen Konditionen evaluiert werden müssen. Dieses Merkmal ist ein Indiz zur Ursache des Problems. Damit das Verfahren anhand des Baumes schneller ist, müssen die Instanzen zusammengefasst werden. Können die Instanzen nur schlecht bis gar nicht zu einem einzigen Pfad zusammengefasst werden, ergibt sich das Problem, dass genauso viele Pfade im Baum existieren wie es Instanzen gibt. Das würde dazu führen, dass nicht nur mehr Attribute für einen einzelnen Pfad / Instanz geprüft werden müssen, sondern die maximal mögliche Anzahl aller in Frage kommender Pfade entsteht und diese geprüft werden müssen. Würde der Baum ab einer gewissen Tiefe abgeschnitten werden, könnten diese Fälle umgangen werden. Dies würde dazu führen, dass die erstellten Regelmengen nicht mehr mit den Ergebnissen der ursprünglichen Version von *SeCo* übereinstimmen.

4.3 Baumeigenschaften

Neben der Anzahl der Pfade soll nachfolgend auch die Anzahl der Knoten im Baum untersucht werden. Diese Eigenschaften können eine Aussage darüber treffen wie stark ein Datensatz komprimiert wird. Baumeigenschaften, welche die Geschwindigkeit der Evaluation essentiell zu beeinflussen scheinen.

1. ***Pfadanzahl***: Damit ist die Anzahl der Pfade gemeint, die vom Wurzelknoten bis zum Blattknoten gehen. Ein solcher (vollständiger) Pfad steht mindestens für eine Instanz. Ist im Blattknoten, des jeweiligen Pfades, das Feld $sum = 1$, ist diese Instanz, die durch den Pfad dargestellt wird, einzigartig.

2. ***Knotenanzahl***: Ein weiteres Maß zur Kompaktheit. Jede Datensatz hat eine bestimmte Anzahl an Konditionen. Da in den Knoten die Konditionen gespeichert und zusammengefasst werden, ist dies wichtig wie häufig zur Evaluation zwischen den Knoten gesprungen werden muss. Die Anzahl der Konditionen eines Datensatzes wird nachfolgend mit $\#Konditionen = \#Instanzen \cdot \#Attribute$ bestimmt.

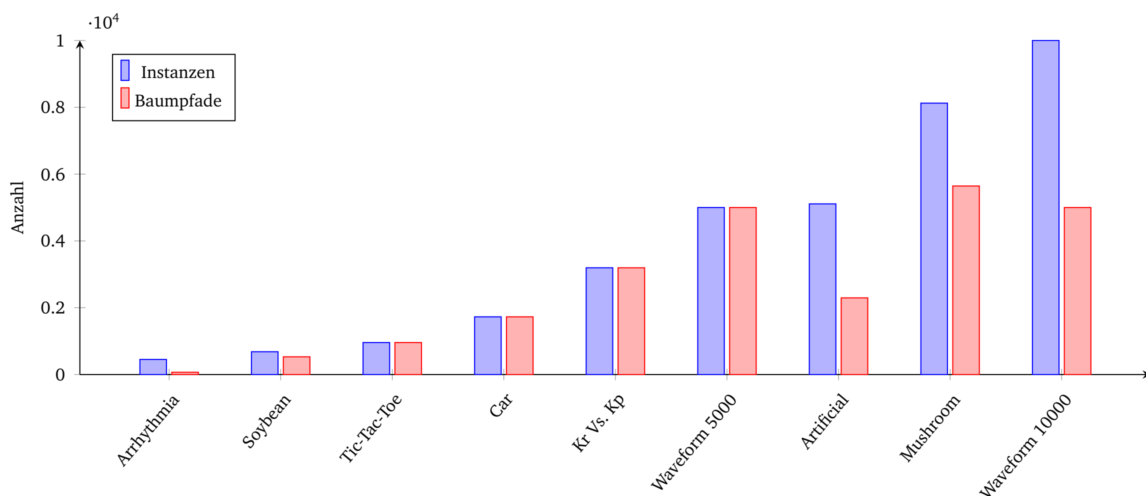


Abbildung 4.5.: Baumkompression: Instanzen Vs. Pfadanzahl

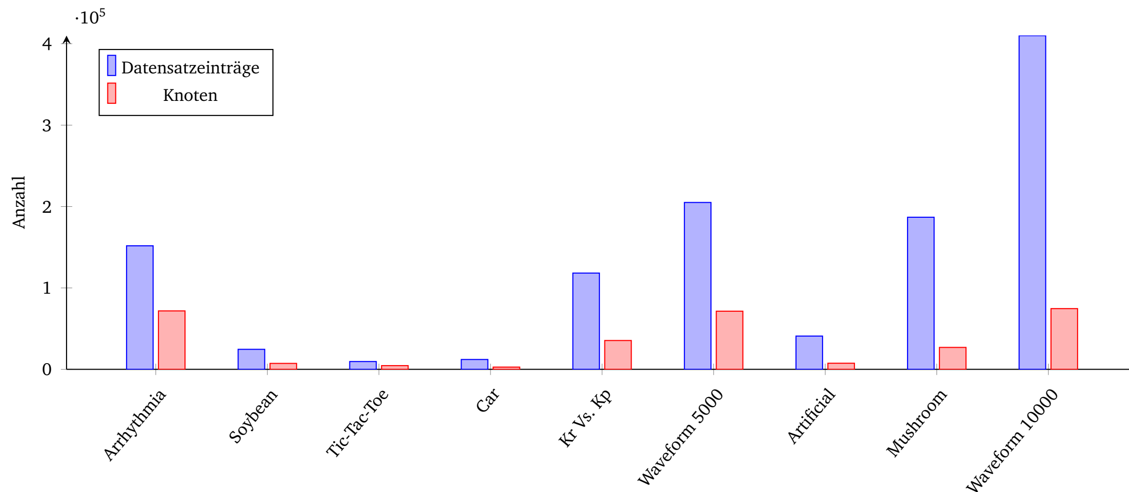


Abbildung 4.6.: Baumkompression: Einträge im Datensatz Vs. Knotenanzahl

In Abbildung 4.5 ist die Anzahl der Pfade im Baum dargestellt. Diese werden mit der Anzahl der Instanzen, des jeweiligen Datensatzes, verglichen. Es ist zu sehen, dass für die Datensätze *Arrhythmia, Soybean, Artificial, Mushroom* und *Waveform* 10 000 die Anzahl der Pfade geringer ist, als die Anzahl der Instanzen im Datensatz. Bei den restlichen Datensätzen entspricht die Anzahl der Pfade der Anzahl der Instanzen.

Daraus kann gefolgert werden, dass in den Datensätzen, zu denen Bäume mit weniger Pfaden erzeugt wurden, die Instanzen nicht einzigartig sind und somit mehrfach vorkommen. Wie bereits erläutert, können nur dann weniger (vollständige) Pfade entstehen, als es Instanzen im Datensatz gibt. Zudem kann ebenso gefolgert werden, dass jede Instanz in den Datensätzen *Tic-Tac-Toe, Car, Kr Vs. Kp* und *Waveform 5000* einzigartig ist. Werden die Zeiten zur Evaluation einer Regel sowie die Gesamtzeiten zur Erstellung einer Regelmenge hinzugezogen, ergibt sich das Bild, dass die Evaluation anhand des Baumes nur Sinnvoll ist, wenn der Baum weniger Pfade hat als es Instanzen gibt. Vereinfacht gesagt müssen dazu im Datensatz identische Instanzen vorkommen. Diese werden dann im Baum zu einem einzigen Pfad zusammengefasst. So kann aus Abbildung 4.5 weiter entnommen werden, dass Datensatz *Waveform* 10 000 zwar im Vergleich zu *Waveform 5000* das Doppelte an Instanzen aufweist, aber im Endeffekt alle Instanzen gleich sind und keine neuen einzigartigen hinzukommen. Daher kann der Datensatz genauso auf 5000 Pfade zusammengefasst werden. Kombiniert man nun dieses Wissen mit der Tatsache, dass Regeln zu Datensatz *Waveform* 10 000 nach Tabelle 4.1 kürzer sind und die Evaluation auf kürzeren Regeln schneller geht, ergibt auch die Messung zur Erstellung der Regelmenge Sinn. Durch die bessere Kompression und die kürzeren Regeln wird zur Erstellung der Regelmenge weniger Zeit benötigt als bei *Waveform* 5000.

Es könnte nun versucht werden zu argumentieren, dass wenn im Datensatz Instanzen durch eine entsprechende Gewichtung zusammengefasst werden, der Baum überflüssig wäre. Dabei wird aber nicht beachtet, dass zum Einen für kürzere Regeln die Referenzen zu den benötigten Attributen einer Regel vorhanden sind und damit direkt gefunden werden und zum Anderen auch „Teilinstanzen" zusammengefasst werden. Besonders gut kann dies aus Abbildung 4.3 entnommen werden. Die zugrundeliegenden Datensätze haben keine doppelten Instanzen. Dadurch entstehen Bäume mit der maximalen Anzahl an (vollständigen) Pfaden. Allerdings können Konditionen, trotzdem zusammengefasst werden und verringern somit den Aufwand eine Regel zu evaluieren. Bis zur Tiefe von 20 hat der Baum nur einen Bruchteil der maximalen Anzahl der Pfade. Regeln, die auf diesen Konditionen arbeiten, müssen nur eine sehr geringe Anzahl an Knoten verarbeiten.

Des Weiteren wird bei der Erstellung versucht die Konditionen in den Knoten zusammenzufassen. In Abbildung 4.6 ist die Anzahl der Konditionen im Datensatz sowie die Anzahl der Knoten im Baum zu sehen. Unter der Anzahl der Konditionen im Datensatz ist das Produkt aus der Anzahl der Instanzen und der Anzahl der Attribute gemeint. Anders ausgedrückt wird die Anzahl der Einträge im Datensatz mit der Anzahl der Knoten verglichen. Beim Vergleich werden fehlende Werte im Datensatz ignoriert. In Abbildung 4.6 ist zu sehen, dass für jeden Datensatz die Anzahl der Knoten geringer ist, als die Anzahl der Konditionen im Datensatz. Außerdem weisen die Bäume zu den Datensätzen *Arrhythmia* und *Waveform* fast die gleiche Anzahl an Knoten auf und es kann entnommen werden, dass die Bäume zu den beiden *Waveform*-Datensätzen fast gleichgroße Bäume generieren. Datensatz *Arrhythmia* hat insgesamt zwar nur 452 Instanzen aber gleichzeitig auch 280 Attribute. Dadurch haben *Arrhythmia* und *Waveform 5000* eine ähnliche Anzahl an Konditionen. Zudem kann für *Arrhythmia* durch die Erstellung des Baumes aber auch nicht so viel zusammengefasst werden (im Gegensatz zu den *Waveform*-Datensätzen). Dies führt dazu, dass auch die Bäume von der Anzahl der Knoten sehr ähnlich sind. Die Regelevaluation anhand des Baumes kann als Durchgehen durch die Konditionen vorgestellt werden. Die Eva-

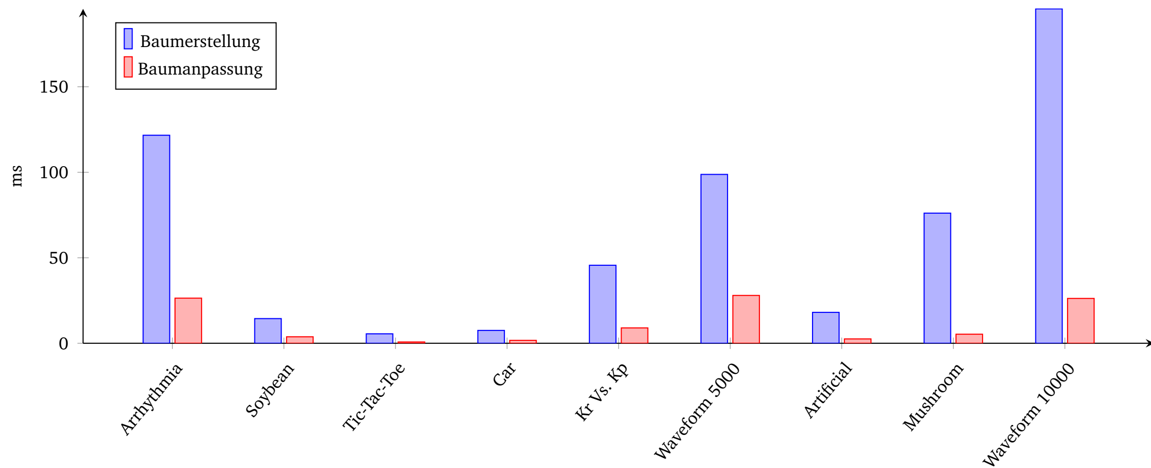


Abbildung 4.7.: Gesamtzeit aller Baumerstellungen und Anpassungen im Baum

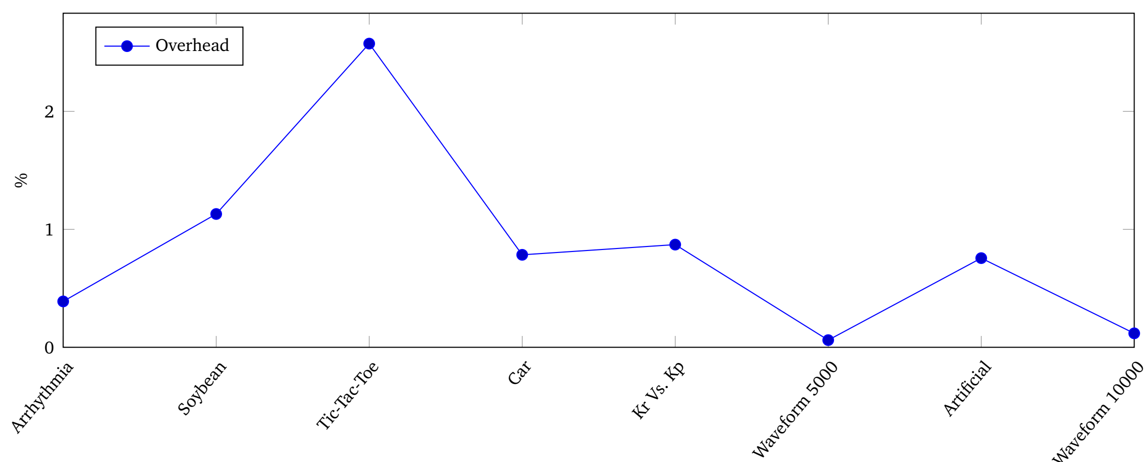
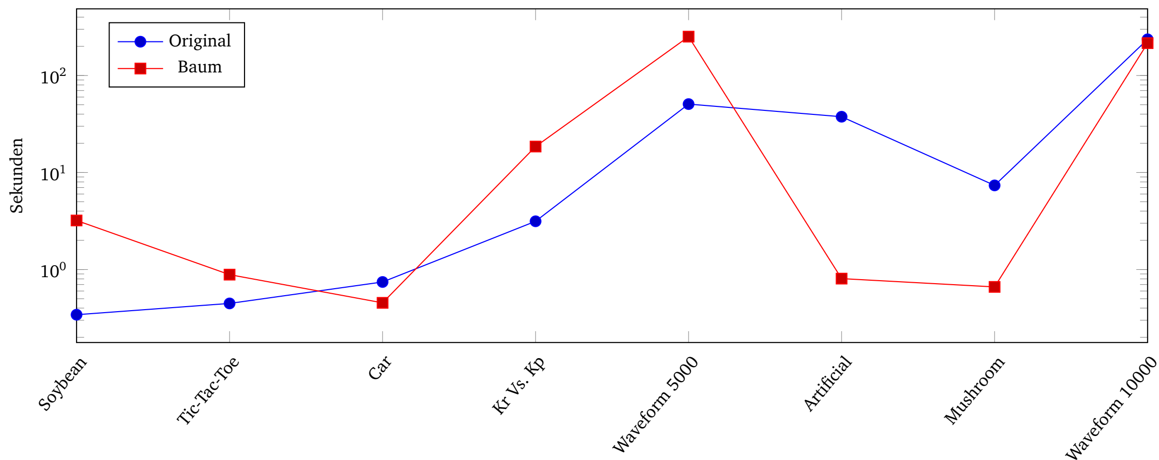


Abbildung 4.8.: Prozentualer Anteil des Overheads an der Gesamtzeit

luation anhand des Datensatzes ist hingegen ein Durchgehen durch die Instanzen. Dementsprechend ist die Evaluation durch dem Baum langsamer, da hier beim Baum 71 787 Knoten im Vergleich zu 452 Instanzen stehen. Die Bäume zu den *Waveform*-Datensätzen sind nicht exakt gleich, da die Verteilung der Konditionen innerhalb des Datensatzes nicht gleich geblieben ist. So entsteht ein leicht anderer Baum, der minimal mehr Knoten besitzt. Dies zeigt einen weiteren Grund auf warum die Erstellung der Regelmenge nicht nur gegenüber der Originalen *SeCo*-Variante um einiges schneller ist, sondern auch gegenüber der Evaluation anhand des Baumes zu *Waveform 5000*.
Abschließend ist es selbsterklärend warum die Anzahl der Knoten niemals größer als die Anzahl der Konditionen im Baum sein kann. Dies würde nicht nur bedeuten, dass jede Kondition einzigartig wäre und damit nichts zusammengefasst werden kann, sondern im Baum andere und mehr Konditionen zu finden sind als in der ursprünglichen Datenmenge.

4.4 Overhead-Anteil

In Abbildung 4.7 ist zu sehen wie viel Zeit innerhalb von *SeCo* die Erstellung sowie Anpassung des Baumes eingenommen hat. Dabei ist zu beachten, dass hier die Summe aller Anpassungen aufgeführt ist. Der Baum wird in *SeCo* nur einmal erstellt. Daher steht die Zeit bei der Erstellung nur für das Erstellen eines einzigen Baumes. Der Grafik kann entnommen werden, dass die Anpassung des Baumes immer, nicht nur schneller ist als die Erstellung sondern, dass alle Anpassungen, die zur Generierung der Regelmenge benötigt wurden, schneller sind als ein einziges Erstellen des Baumes. Dies bestätigt, dass das zur Anpassung des Baumes entworfene Verfahren sinnvoll ist und die Gesamtzeit so um ein vielfaches verringert wird.
In Abbildung 4.8 ist der prozentuale Anteil des Overheads (Baumerstellung + Anpassung) zur Gesamtzeit visualisiert. Dabei ist zu sehen, dass der Anteil immer bei ca. 1% liegt. Würde der Baum jedes Mal, wenn die Datenmenge angepasst wird, neu generiert werden, wäre der Anteil um einiges größer und würde dazu führen, dass die Evaluation bei kleineren



	Soybean	Tic-Tac-Toe	Car	Kr Vs. Kp	Waveform 5 000	Artificial	Mushroom	Waveform 10 000
Original	0.342	0.447	0.744	3.142	50.761	37.607	7.367	236.196
Baum	3.203	0.885	0.453	18.529	251.743	0.806	0.662	215.719

Abbildung 4.9.: Gesamtzeiten zur Heuristik Accuracy

Datensätzen noch langsamer wäre als die originale *SeCo*-Version und es wäre bei größeren Datensätzen ein geringerer Vorteil zu sehen. Hat ein Baum einige wenige *Single Path Trees*, wie hier die *Waveform*-Datensätze wird nochmal weniger Zeit benötigt, da nicht ständig neue Knoten erstellt werden müssen.
Abschließend kann so festgehalten werden, dass der zusätzliche Overhead, besonders bei größeren Datensätzen, und der Heuristik *Precision* durch die Vermeidung der Erstellung des Baumes, kaum ins Gewicht fällt. Viel wichtiger ist der Vorteil, der durch die Evaluierung der Regel sowie der Baumgröße erreicht wird.

4.5 Gesamtzeit bei einer anderen Heuristik

Abschließend wird noch ein kurzer Überblick zur Effizienz bei anderen Heuristiken geliefert, bzw. wie sich diese auf die Evaluationszeit auswirkt. Dazu wird exemplarisch die Heuristik *"Accuracy"* gewählt. In Abbildung 4.9 sind die Gesamtzeiten zur Erstellung der Regelmenge unter dieser Heuristik dargestellt. Es ist zu sehen, dass die Evaluation anhand des Baumes nur bei den Datensätzen *Car, Artificial, Mushroom* sowie *Waveform* 10 000 dazu führt, dass die Erstellung der Regelmenge schneller geht. Des Weiteren ist zu sehen, dass die Evaluation mit dem Baum für die *Waveform*-Datensätze mit der Accuracy Heuristik länger dauert, als unter der Heuristik Precision. Der Trend, dass die Erstellung der Regelmenge für den Größeren der beiden Datensätze schneller geht, ist hingegen erhalten geblieben.
Die Ergebnisse aus Abbildung 4.9 können ohne eine Betrachtung der erstellten Theorien nicht sauber interpretiert werden. Daher ist die Anzahl der Regeln sowie die durchschnittliche Regellänge zum jeweiligen Datensatz in Tabelle 4.2 aufgeführt. Dabei ist zu sehen, dass im Vergleich zu der Heuristik *Precision* weniger Regeln erstellt werden. Diese haben im Vergleich aber eine ähnliche Anzahl an Konditionen.

	Soybean	Tic-Tac-Toe	Car	Kr Vs. Kp	Waveform 5 000	Artificial	Mushroom	Waveform 10 000
#Regeln	50	92	52	42	195	108	9	192
∅ Regellänge	2.24	4.17	4.6	4.76	3.08	2.7	1.22	2.81

Tabelle 4.2.: Evaluationsergebnisse zur Heuristik Accuracy

Zusammen können die Werte so interpretiert werden, dass die Evaluation anhand des Baumes für verschiedene Heuristiken nur wirklich sinnvoll ist, wenn die Anzahl der Instanzen im Datensatz >> 10 000 beträgt und dabei auch Attribute zusammengefasst werden können. Es ist zu beachten, dass die Anzahl der Attribute, wie zuvor beschrieben, relativ klein sein muss. Ein weiterer Punkt ist die Anzahl der Regeln. Aus Abbildung 4.4 konnte entnommen werden, dass die Evaluation des Baumes vor allem für kurze Regel gegenüber der originalen Version Zeit gutmacht. Werden weniger Regeln erstellt, verbringt die Evaluation weniger Zeit mit der Evaluation kurzer Regeln. Die Ursache für den Zeitunterschied

zwischen den *Waveform*-Datensätzen wird dadurch nicht beeinflusst und bleibt daher erhalten. Werden weniger aber dafür längere Regeln evaluiert steigt die benötigte Zeit zur Erstellung der Regelmenge noch stärker an und es entsteht der Verlauf in Abbildung 4.9.

4.6 Zusammenfassung

In diesem Abschnitt wurden zunächst die Gesamtzeiten zur Erstellung einer Regel verglichen und es wurde dabei festgestellt, dass die Zeit zur Erstellung einer Regelmenge besonders von der Anzahl der Attribute und der Baumkompression abhängt. Es wurde nachfolgend erörtert, dass durch die Vorgehensweise zur Regelevaluation längere Regeln langsamer verarbeitet werden, als kürzere Regeln. Außerdem wurde dabei nochmal vor Augen geführt wie wichtig die *Threshold* ξ ist. Weiter wurde gezeigt wie gut ein Datensatz zusammengefasst werden kann, wovon die Komprimierung abhängt und wie wichtig die Knoten und insbesondere die Pfadanzahl, wie auch die Entwicklung der Pfadanzahl, bei der Evaluation ist. Zum Schluss wurde nochmal aufgezeigt wie viel zusätzliche Arbeit die Erstellung und Anpassung des Baumes kostet und wie sich die Evaluation bei anderen Heuristiken verhält. Dabei konnte auch gezeigt werden wie wichtig die Vermeidung einer Baumerstellung ist und wie viel Zeit durch die Anpassung gespart wird.

5 Ausblick

Abschließend sollen, basierend auf den Ergebnissen der Evaluation, Vorgehensweisen und Ansätze vorgestellt werden, die das Verfahren verbessern könnten oder die Baumstruktur auf eine andere Weise nutzen. Außerdem werden die Mängel der Implementation betrachtet und es wird diskutiert wie diese ausgebessert werden könnten. Zum Schluss wird ein Überblick zu Arbeiten und Studien gegeben, die einen ähnlichen Ansatz haben oder sich mit aktuellen Problemstellungen bzgl. der Assoziationsanalyse befassen.

5.1 Vollständige Einbindung des Baumes in *SeCo*

Wie bei der Vorstellung des *FP-Tree* in Abschnitt 2.3, sowie bei der Evaluation gezeigt wurde, arbeitet der *FP-Tree* und seine Erweiterungen nur auf nominalen Attributen. Dadurch können verschiedene Ansätze und damit insbesondere Heuristiken oder Datensätze ohne Diskretisierung nicht mehr genutzt werden. Ein weiteres Problem sind die redundanten Operation, die durch parallele Nutzung des Datensatzes und des Baumes entstehen. Bei der Evaluation wurde aufgeführt wie wichtig die Datenkomprimierung anhand des Baumes ist. Gibt es numerische Attribute, kann mit einer sehr hohen Wahrscheinlichkeit angenommen werden, dass diese kaum bis gar nicht zusammengefasst werden können und so der Baum immer eine maximale Anzahl an Pfaden aufweisen würde. Ein Vorschlag zur Behebung dieses Problems wäre eine Vorverarbeitung numerischer Attribute auf Intervalle, während der Erstellung des Baumes, die dann im Baum statt der konkreten Werte oder inklusive dieser in einem Knoten gespeichert werden. So könnten numerische Werte, wie bei einer vorher durchgeführten Diskretisierung, genutzt werden. Dabei müssten aber zusätzliche Optionen ermöglicht werden, damit die Attribute auf verschiedene Art und Weise diskretisiert werden können. Der Vorteil dieser Vorgehensweise innerhalb des Baumes wäre die dynamische Anpassung der Intervalle. Diese könnten während der Erstellung der Theorie weiterhin angepasst werden und wären somit besser als eine vorher durchgeführte Diskretisierung, bei der der Baum dann von nominalen Attributen ausgeht. Da die Nutzung numerischer Attribute immer noch eine Herausforderung im Bereich der Assoziationsanalyse ist, wird eine allgemeine Variante zur Nutzung von numerischen Attributen in Abschnitt 5.4 vorgestellt.
Zudem könnten die benötigten Gesamtzeiten weiter verringert werden, wenn das restliche Framework nur den Baum nutzen würde. Dadurch müssten nicht zwei Strukturen bearbeitet werden, die im Grunde beide nur eine Repräsentation des Datensatzes sind. Dazu müsste aber das gesamte Framework umgeschrieben werden, was über den Rahmen dieser Arbeit hinausgehen würde. Dabei sollte aber auch angezweifelt werden, inwieweit eine solche Umstellung sinnvoll ist und wirklich effektiver wäre. Es hat sich gezeigt, dass die Evaluation des Baumes für längere Regeln langsamer ist als ein Durchgehen durch den Datensatz. Zudem konnte beobachtet werden, dass der *Overhead*, der durch die Erstellung und Anpassung des Baumes generiert wird, keinen großen Anteil der Gesamtzeit einnimmt. Würde die Evaluation anhand des Baumes verbessert werden, sodass diese nicht mehr von der Länge der Regel abhängt und damit auch für längere Regeln schneller wäre, als die Traversierung des Datensatzes, erst dann wird die Umstellung auf den Baum als sinnvoll erachtet. Zwei mögliche Ansätze werden in den nachfolgenden Abschnitten vorgestellt.

5.2 Ansatz einer *horizontalen* Baumevaluation

Ein Problem, welches bei der Evaluation eines Pfades beobachtet werden konnte, sind Knoten mit Konditionen, die nicht in der Regel vorkommen. Der Pfad muss, bei einer Regel mit mehr als zwei Konditionen, immer von unten nach oben verarbeitet werden. Könnte dieser Schritt eliminiert werden und die Konditionen im Baum, wie bei einer Regel mit einer Kondition, direkt referenziert werden, könnte eine Geschwindigkeitssteigerung erreicht werden. Der Baum würde dann nicht mehr von unten nach oben bearbeitet, sondern durch die *Header Table* von links nach rechts und damit *"horizontal"*. Dazu müssten die Knoten innerhalb des Baumes wissen auf welchen Pfaden diese liegen, bzw. wissen welche Knoten sich unterhalb oder oberhalb befinden, ohne diese explizit zu speichern. Dazu könnte jeder Pfad eine *ID* bekommen, welche in die Knoten eingefügt wird. Ein Knoten nah an der Wurzel müsste dadurch viele Pfad-IDs speichern, ein Blattknoten nur eine Einzige. So könnten die einzelnen Konditionsknoten einer Regel direkt angesprungen werden. Es ist dabei wahrscheinlich zu empfehlen, dass die Konditionen einer Regel zuvor anhand der *frequentList* sortiert werden, damit möglichst wenig Pfade in Betracht gezogen werden. Die Vorgehensweise könnte so in den nachfolgenden Schritten zusammengefasst werden:

1. Seltenste noch nicht verwendete Kondition nehmen und die Knoten durch die *Header Table* anspringen.
2. Beim ersten Mal die dort enthaltenen Pfad-IDs ohne Überprüfung in einer Liste speichern und mit Schritt 1 fortfahren.

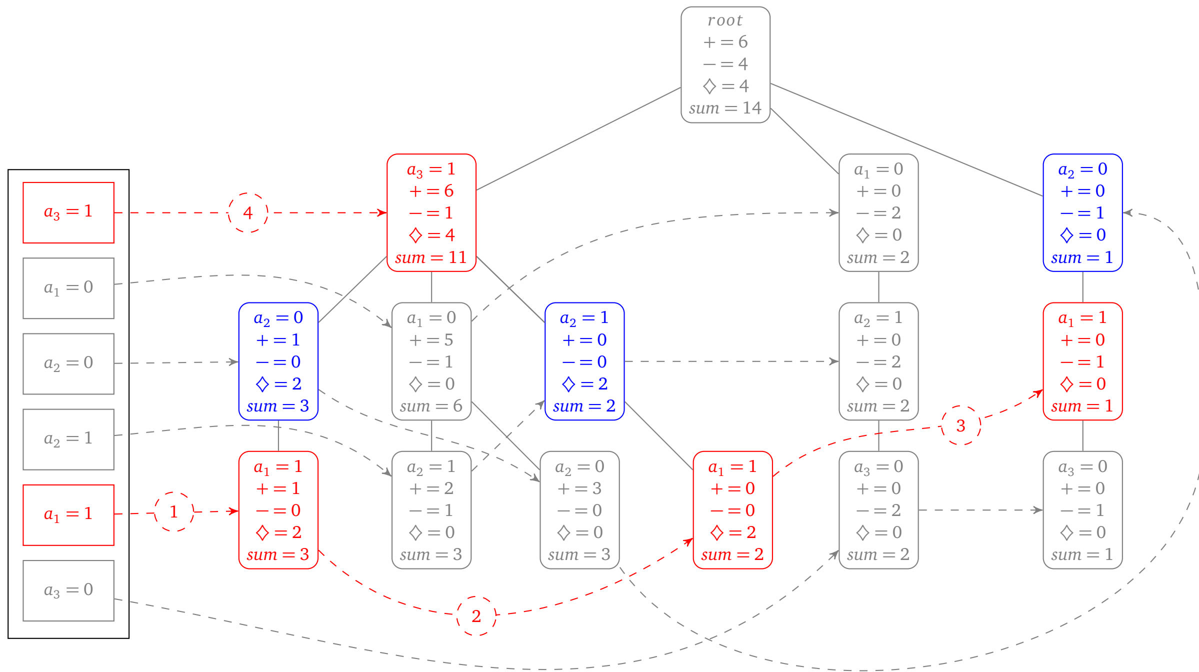


Abbildung 5.1.: Schema der *horizontalen* Verarbeitung zur Regel: $a_1 = 1 \wedge a_3 = 1 \Rightarrow a_4 = \diamondsuit$

3. Überprüfen, ob eine (einzige) der Pfad-IDs im Knoten auch innerhalb der gespeicherten Liste enthalten ist. Ist eine ID enthalten, müssen auch alle andere IDs enthalten sein, da der Knoten oberhalb liegt.

4. Ist eine Pfad-ID enthalten, werden alle Pfad-IDs des dazu betrachteten Knoten in einer neuen Liste übernommen. Nachdem alle Knoten der Kondition verarbeitet wurden, wird die neue Liste als Referenzliste gewählt.

5. Mit Schritt 2 fortfahren, wenn noch nicht alle Konditionen der Regel verarbeitet wurden.

6. Die Knoten, die zum Schluss mit den Pfad-IDs übrigbleiben, werden für die Regel als abgedeckte Pfade betrachtet und auf die *tp* und *fp* verrechnet.

Vorteile dieser Vorgehensweise sind, dass nur Knoten mit den verwendeten Konditionen angesprungen werden und damit nicht mehr überflüssige Knoten auf dem Pfad von unten nach oben betrachtet werden müssen. Ein weiterer Vorteil ist die Tatsache, dass der Baum wie bisher angepasst werden kann und die angepasste Version ebenfalls durch dieses Vorgehen verarbeitet werden kann. In Abbildung 5.1 ist die Vorgehensweise für den Datensatz aus Tabelle 3.5 und dem dazugehörigen Baum aus Abbildung 3.5 nochmal visuell aufgearbeitet. Die evaluierte Regel soll dabei $a_1 = 1 \wedge a_3 = 1 \Rightarrow a_4 = \diamondsuit$ sein. Die roten Markierungen stehen für die bearbeiteten Strukturen nach dem eben vorgestellten Verfahren. Die blauen Knoten wären die noch zusätzlichen Knoten, welche bei einer Bearbeitung der Pfade von unten nach oben zusätzlich hinzukommen würden.
Die seltenste nicht verwendete Kondition ist $a_1 = 1$. Dementsprechend werden alle Knoten dazu verarbeitet und die Pfad-IDs dieser Knoten in einer Liste gespeichert. Da dies Blattknoten sind oder nur einen einzigen Pfad (ganz rechts) haben, sind am Ende drei Pfad-IDs in der Liste vermerkt. Darauf hin wird die nächste Kondition bearbeitet. In diesem Fall wäre dies $a_3 = 1$. Dieser Knoten enthält die Pfad-IDs der untersten markierten Blattknoten, welche durch die *Link-Nodes* mit der 1 und 2 angesprungen werden. Der markierte Knoten, welcher durch den *Node-Link* mit der 3 angesprungen wird, wird nicht mehr übernommen. Da bereits alle Knoten und Konditionen verarbeitet wurden, werden die Werte aus den zwei übriggebliebenen Knoten auf die *tp* und *fp* der Regel verrechnet. Durch diese Vorgehensweise müsste in diesem Fall nur in etwa die Hälfte der vorher bearbeiteten Knoten evaluiert werden. Ein entsprechender Ansatz wurde bereits begonnen und kann im Anhang A.2 entnommen werden. Diese Vorgehensweise ist noch nicht ausgereift und sollte weiter verfeinert werden. Erste Evaluationen konnten keinen Signifikaten Vorteil aufzeigen. Zur Erstellung der Pfade wurde dabei die Java-Klasse `UUID` genutzt. Die Generierung des Baumes hat dementsprechend länger gedauert und es sollte bei einer möglichen weiteren Bearbeitung darauf geachtet werden inwieweit sich dies negativ auf den Overhead auswirkt.

5.3 Kombination mit bestehenden Verfahren

Bei der Evaluation konnte weiter festgestellt werden, dass für längere Regeln eine Traversierung des Datensatzes schneller ist, als das vorgestellte Verfahren zur Evaluation anhand des Baumes. Eine Möglichkeit wäre es für längere Regeln nicht mehr den Baum zu nutzen, sondern das bestehende Verfahren oder ein anderes neues Vorgehen zu entwickeln. Dabei sollte aber beachtet werden, dass die Anpassung des Baumes darauf beruht, dass die Regeln vorher evaluiert wurden. Nur dann kann der Baum wie beschrieben angepasst werden. Werden längere Regeln nicht mehr mit dem Baum evaluiert, müsste die Anpassung des Baumes verändert werden oder der Baum jedes Mal, wenn die Datenmenge angepasst wird, neu erstellt werden. Der Evaluation konnte weiter entnommen werden wie wichtig die Vermeidung der Erstellung ist und aus Abbildung 4.7 wie viel Zeit dadurch eingespart wird. Dementsprechend wäre eine neue Art der Anpassung ein notwendige Bedingung zur Kombination mit anderen Verfahren.
Eine weitere Möglichkeit ist eine Vorverarbeitung der Daten. Dabei könnte ähnlich einer *Threshold* ξ versucht werden, Attribute und Konditionen zu streichen, die selten vorkommen. Dadurch würde sich ein viel schmalerer Baum bilden (wie in Abschnitt 4.2.1 festgestellt) und das Problem der Evaluation langer Regeln würde damit aufgehoben werden. Die Art der Vorverarbeitung sollte sich dabei daran orientieren, dass möglichst wenig Pfade im Baum entstehen! Die Knotenanzahl hängt vor allem von der Anzahl möglicher Konditionen oder der Pfadanzahl ab. Dementsprechend empfiehlt sich eine Vorverarbeitung, die die Pfadanzahl minimiert.
Es könnte aber auch, wie in Abschnitt 2.3.1 beschrieben, versucht werden die Evaluation aller möglichen Regeln zuvor durchzuführen. Das hätte aber den Nachteil, dass zum Einen extra Speicher dazu angelegt werden muss, in dem die Ergebnisse gespeichert werden. Zum Anderen wäre das Verfahren sehr starr. Damit ist gemeint, dass die zuvor vorgestellte Vorgehensweise sich an *SeCo* anpasst und nur Regeln evaluiert und verarbeitet werden, die auch in Betracht gezogen wurden. Wird die Evaluation vorher durchgeführt, werden alle möglichen Kombination erstellt und evaluiert. Ohne eine *Threshold* ξ erscheint dieser Ansatz wenig sinnvoll. Weiter ist zu beachten, dass eine einfache Anpassung der Baumstruktur nicht mehr sinnvoll wäre und somit der Baum bei jeder Änderung der Datenmenge komplett neu erstellt und erneut evaluiert werden sollte. Dementsprechend wurde diese Variante im Rahmen dieser Arbeit ausgelassen.
Abschließend wäre eine Parallelisierung des Vorgehens eine mögliche Alternative. In [24] wird ein solches Verfahren vorgestellt. Dabei wird ein Datensatz in p gleichgroße Datensätze aufgeteilt, wobei p für die Anzahl der Prozessoren steht. Anschließend wird das Auftreten, jedes Items, lokal auf dem jeweiligen Datensatz bestimmt. Die Ergebnisse werden global zusammengetragen und in einer *Header Table* gespeichert. Diese besitzt dann nicht mehr eine Referenz pro Kondition, sondern für jede Kondition werden p Referenzen erstellt, da p Bäume generiert wurden. Abschließend werden die Bäume, wie in Abschnitt 2.3.1 vorgestellt bearbeitet. Durch die Aufteilung ist nicht sichergestellt, dass nicht die gleichen *Frequent Pattern* im Baum erstellt werden. Diese müssen daher, bevor die *Conditional Pattern Base* erstellt wird, zusammengefasst werden. Bei der Evaluation konnte eine Geschwindigkeitssteigerung beobachtet werden und es hat sich auch gezeigt, dass bei einer relativen Anpassung der *I/O*-Zugriffe eine mehrfache Verbesserung, abhängig von der Prozessorzahl, erreicht wird [24, S. 668].

5.4 Aktuelle Forschung

Im Bereich der Assoziationsanalyse können aktuell Arbeiten gefunden werden, die einen stochastischen Ansatz verfolgen [9, 23], sich mit dem Problem numerischer Attribute beschäftigen [3, 15] oder sich der Problemstellung einer Netzwerk basierenden Lösung befassen [19, 18].

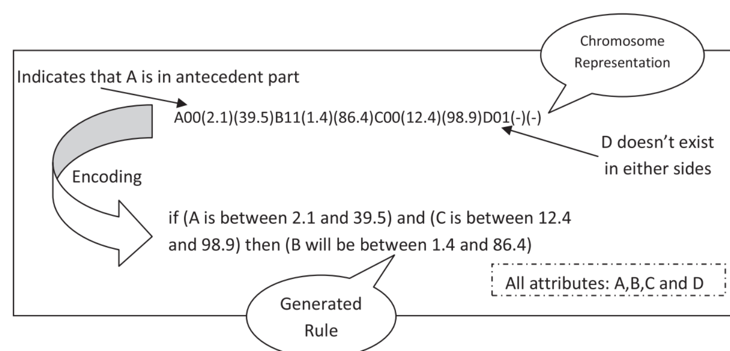


Abbildung 5.2.: Regelrepräsentation in einem Chromosom nach [15]

In [15] wird anhand eines genetischen Algorithmus versucht Regeln zu erstellen, die numerische Attribute nutzen. Dabei ist aber zu beachten, dass die Vorgehensweise nicht allgemein nutzbar ist, da durch den Algorithmus nur die *"besten"* Re-

geln gefunden werden. Anders als bei FP-Growth, welches durch eine Anpassung der *Support Threshold* ξ alle möglichen Kombinationen finden kann. Zur Erstellung der Regeln werden in [15, S. 18] der *Confidence*, die *Comprehensibility* und die *Interestingness* genutzt. Zur Repräsentation der *Chromosome* des genetischen Algorithmus wird der *Michigan*-Ansatz genutzt. Dieser besagt, dass jedes Chromosom nur eine einzige Regel darstellt. In Abbildung 5.2 ist ein Beispiel dargestellt. Die Buchstaben stehen für die verschiedenen Attribute. Die *Bits* / Zahlen darauffolgend sagen aus, ob das Attribut im Regelkopf oder im Regelkörper steht. Abschließend sind zwischen den Klammern die Grenzen des Intervalls. Die Bits eines Attributes werden während der Mutation und der Kreuzung zufällig verändert. Die Intervalle sind durch die Maxima und Minima des jeweiligen Attributes gebunden, werden aber während des Algorithmus ebenfalls zufällig erstellt. Der Algorithmus terminiert nach einer festen Anzahl an Iterationen oder wenn ein *Pareto*-Optimum erreicht wurde. Das *Pareto*-Optimum beschreibt dabei einen Zustand bei dem keine Regeln mehr hinzugefügt werden können, ohne negative Folgen. Es ist zu beachten, dass zur Vermeidung minimaler Intervalle eine Mindestgröße festgelegt wird. Weiter werden Regeln mit einem geringen *Support* ebenfalls entfernt, selbst wenn diese zum *Pareto*-Optimum beigetragen hätten. Bei Experimenten konnte festgestellt werden, dass der vorgestellte Ansatz sinnvoll ist und nützliche numerische Werte bestimmt werden können.

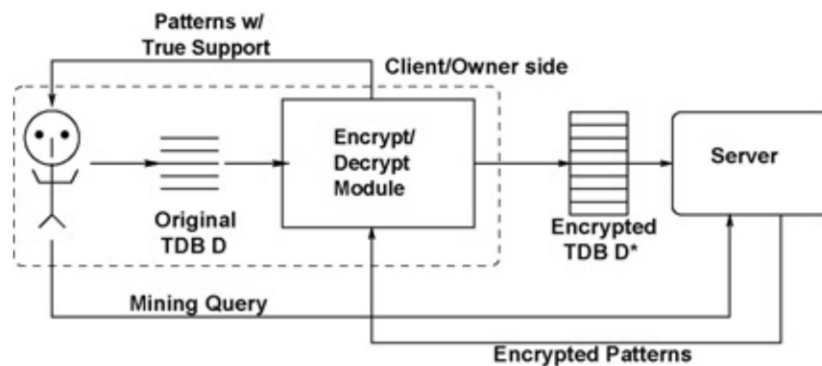


Abbildung 5.3.: Assoziationsanalyse als Service nach [9]

Ein anderer Ansatz beschäftigt sich mit der Option die Assoziationsanalyse auszulagern. Die dabei genutzten Algorithmen können die Gleichen sein wie zuvor beschrieben. Die Daten müssen aber entsprechend verschlüsselt werden, da diese ansonsten nicht nur verändert, sondern auch gestohlen werden könnten. Zudem soll die Partei, die die Assoziationsanalyse durchführt keine sensiblen Daten einsehen können. Der Grund zur Auslagerung ist vor allem der Trend zum *Cloud-Computing* und den fehlenden Kompetenzen innerhalb eines Unternehmens die Daten zu verarbeiten. In [9] wird die Assoziationsanalyse als Service betrachtet. Der dabei vorgestellte Ansatz versucht die Daten so zu verschlüsseln, dass ein *Item* nicht von $k-1$ anderen Items unterschieden werden kann. Das Vorgehen soll dabei Effektiv und Skalierbar sein, sowie die Datensicherheit gewährleisten. Der *Support* und die *Frequent Item Pattern* werden, bevor der *Mining-Service* darauf zugreifen kann, verschlüsselt. Die Werte und Pattern, die zurückgegeben werden, entsprechen dabei nicht den konkreten Ergebnissen und müssen daher entschlüsselt werden. In Abbildung 5.3 ist dabei das eben beschriebene Konzept dargestellt. Zur Verschlüsselung werden dabei anders als bei [22] keine *Fake-Items* erstellt. Wie in [17] gezeigt wurde, ist das einfache Einbinden von *Fake-Items* wenig Sinnvoll.
Beide Ansätze können gemeinsam genutzt werden um die Problemstellung der numerischen Attribute innerhalb des vorgestellten Verfahren zu bestimmen und die Regeln dann anhand von entsprechenden Services auszulagern und dabei massiv zu parallelisieren. Dabei müssten die Daten wie zuvor beschrieben verschlüsselt werden.

5.5 Abschluss

Abschließend kann zusammengefasst werden, dass das vorgestellte Verfahren, welches einen FP-Tree nutzt, keine überzeugenden Ergebnisse geliefert hat. In der Evaluation konnte gezeigt werden, dass ohne eine *Threshold* nur bei sehr kleinen Regeln ein Geschwindigkeitsvorteil beobachtet werden konnte. Für längere Regeln ist es bereits langsamer, als ein redundantes Durchgehen durch die Daten. Zudem kann es nicht mit numerischen Attributen umgehen und erzeugt ein zusätzlichen Overhead, der aber auch nur einen geringen Anteil ausmacht. Die vorgeschlagenen Verbesserungen könnten einen entscheidenden Vorteil liefern und sollten daher in ausgereifter Form evaluiert werden. Außerhalb des *SeCo*-Frameworks könnten, durch die Nutzung von *FP-Growth* trotzdem enorme Geschwindigkeitssteigerungen erreicht werden. Aktuelle Arbeiten beschäftigen sich mehr mit stochastischen Ansätzen oder mit praxisrelevanten Problemen zur Evaluation von Daten innerhalb eines Netzwerkes. Diese könnten aber ebenso in eine Evaluation anhand eines FP-Tree eingebaut werden.

Literatur

[1] R. Agrawal, T. Imielinski und A. N. Swami. „Mining Association Rules between Sets of Items in Large Databases". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.* Hrsg. von P. Buneman und S. Jajodia. Bd. 22. Association for Computing Machinery (ACM), Special Interest Group on Management of Data (SIGMOD). ACM Press, 1993, S. 207–216.

[2] R. Agrawal und R. Srikant. „Fast Algorithms for Mining Association Rules in Large Databases". In: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Hrsg. von J. B. Bocca, M. Jarke und C. Zaniolo. Bd. 1215. Morgan Kaufmann, 1994, S. 487–499.

[3] B. Alatas, E. Akin und A. Karci. „MODENAR: Multi-objective differential evolution algorithm for mining numeric association rules". In: *Applied Soft Computing* 8.1 (2008), S. 646–656.

[4] T. W. Anderson. *An Introduction to Multivariate Statistical Analysis*. 3. Aufl. Wiley-Interscience, Juli 2003. ISBN: 9780471360919.

[5] M. Atzmüller und F. Puppe. „SD-Map - A Fast Algorithm for Exhaustive Subgroup Discovery". In: *Knowledge Discovery in Databases: PKDD 2006, 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, Berlin, Germany, September 18-22, 2006, Proceedings*. Hrsg. von J. Fürnkranz, T. Scheffer und M. Spiliopoulou. Lecture Notes in Computer Science. Springer, 2006, S. 6–17.

[6] C. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 1st ed. 2006. Corr. 2nd printing 2011. Springer, 2006. ISBN: 0387310738.

[7] J. Fürnkranz. „Separate-and-Conquer Rule Learning". In: *Artificial Intelligence Review* 13.1 (1999), S. 3–54.

[8] L. Geng und H. J. Hamilton. „Interestingness Measures for Data Mining: A Survey". In: *ACM Computing Surveys (CSUR)* 38.3 (2006), S. 9.

[9] F. Giannotti u. a. „Privacy-Preserving Mining of Association Rules From Outsourced Transaction Databases". In: *IEEE Systems Journal* 7.3 (2013), S. 385–395. ISSN: 1932-8184.

[10] M. A. Gluck und D. E. Rumelhart, Hrsg. *Neuroscience and Connectionist Theory (Developments in Connectionist Theory Series)*. 1. Aufl. Psychology Press, Feb. 1990. ISBN: 0805806199.

[11] J. Han u. a. „Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach". In: *Data Mining and Knowledge Discovery* 8.1 (2004), S. 53–87.

[12] P. Langley. „The changing science of machine learning". In: *Machine Learning* 82.3 (2011), S. 275–279.

[13] W. S. McCulloch und W. Pitts. „A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), S. 115–133.

[14] A. Merceron und K. Yacef. „Interestingness measures for association rules in educational data". In: *Educational Data Mining 2008, The 1st International Conference on Educational Data Mining, Montreal, Québec, Canada, June 20-21, 2008. Proceedings*. Hrsg. von R. S. J. de Baker, T. Barnes und J. E. Beck. www.educationaldatamining.org, 2008, S. 57–66.

[15] B. Minaei-Bidgoli, R. Barmaki und M. Nasiri. „Mining numerical association rules via multi-objective genetic algorithms". In: *Information Sciences* 233 (2013), S. 15–24.

[16] T. M. Mitchell. *Machine Learning (McGraw-Hill International Editions Computer Science Series)*. 1st. McGraw-Hill, Okt. 1997. ISBN: 0071154671.

[17] I. Molloy, N. Li und T. Li. „On the (In)Security and (Im)Practicality of Outsourcing Precise Association Rule Mining". In: *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*. Hrsg. von W. Wang u. a. IEEE. IEEE Computer Society, 2009, S. 872–877.

[18] G. Qian u. a. „Boosting association rule mining in large datasets via Gibbs sampling". In: *Proceedings of the National Academy of Sciences* 113.18 (2016), S. 4958–4963.

[19] K. N. V. D. Sarath und V. Ravi. „Association rule mining using binary particle swarm optimization". In: *Engineering Applications of Artificial Intelligence* 26.8 (2013), S. 1832–1840.

[20] P. Simon. *Too Big to Ignore: The Business Case for Big Data (Wiley and SAS Business Series)*. 1. Aufl. Wiley, Nov. 2015. ISBN: 9781119217848.

[21] R. S. Sutton und A. G. Barto. „A temporal-difference model of classical conditioning“. In: *Proceedings of the ninth annual conference of the cognitive science society*. Seattle, WA. 1987, S. 355–378.

[22] W. K. Wong u. a. „Security in Outsourcing of Association Rule Mining“. In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Hrsg. von C. Koch u. a. Very Large Data Bases Endowment. Association for Computing Machinery (ACM), 2007, S. 111–122.

[23] X. Yi u. a. „Privacy-Preserving Association Rule Mining in Cloud Computing“. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. Hrsg. von F. Bao u. a. Association for Computing Machinery (ACM). 2015, S. 439–450.

[24] O. R. Zaïane, M. El-Hajj und P. Lu. „Fast Parallel Association Rule Mining without Candidacy Generation“. In: *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA*. Hrsg. von N. Cercone, T. Y. Lin und X. Wu. IEEE. IEEE Computer Society, 2001, S. 665–668.

A Anhang

A.1 SeCoMemory; eine Implementierung des Algorithmus mit zugehöriger Baumstruktur

```
1  package de.tu_darmstadt.ke.seco.models.vertical;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Comparator;
7  import java.util.HashMap;
8  import java.util.Map;
9  import java.util.Map.Entry;
10
11 import weka.core.Attribute;
12 import weka.core.Instance;
13 import weka.core.Instances;
14 import de.tu_darmstadt.ke.seco.models.Condition;
15 import de.tu_darmstadt.ke.seco.models.NominalCondition;
16 import de.tu_darmstadt.ke.seco.models.NumericCondition;
17 import de.tu_darmstadt.ke.seco.models.Rule;
18 import de.tu_darmstadt.ke.seco.stats.TwoClassConfusionMatrix;
19
20 public class SeCoMemory implements Serializable, SeCoVerticalMemory {
21
22     private static final long serialVersionUID = -3816503988906729946L;
23     public SeCoTree root;
24     private Map<String, SeCoTree> headerTable;
25     private Map<String, ArrayList<SeCoTree>> coveredPaths;
26     private ArrayList<Condition> frequentConditionList;
27
28     public SeCoMemory(Instances examples) {
29         this.root = new SeCoTree();
30         this.headerTable = new HashMap<String, SeCoTree>();
31         this.frequentConditionList = new ArrayList<Condition>();
32         this.coveredPaths = new HashMap<String, ArrayList<SeCoTree>>();
33
34         this.createFrequentItemList(this.frequentConditionList, examples);
35         this.buildTree(examples, root, headerTable, this.frequentConditionList);
36     }
37
38     private void createFrequentItemList(ArrayList<Condition> frequentConditionList, Instances examples) {
39         Map<Condition, Double> frequency = new HashMap<Condition, Double>();
40         for(Instance instance : examples) {
41             for(int index = 0; index < instance.numAttributes(); index++) {
42                 if(index == instance.classIndex()) {
43                     continue;
44                 }
45
46                 Condition cond = createCondition(instance.attribute(index), instance.value(index));
47                 if(frequency.containsKey(cond)) {
48                     frequency.put(cond, frequency.get(cond) + instance.weight());
49                 } else {
50                     frequency.put(cond, instance.weight());
51                 }
52             }
53         }
54
55         ArrayList<Map.Entry<Condition, Double>> frequencyList = new ArrayList<Map.Entry<Condition,Double
               >>(frequency.entrySet());
56         frequencyList.sort(new Comparator<Map.Entry<Condition, Double>>() {
57             @Override
58             public int compare(Entry<Condition, Double> arg0, Entry<Condition, Double> arg1) {
59                 return (int) (arg1.getValue() - arg0.getValue()); // descending order
60             }
```

```
61          });
62          for(Map.Entry<Condition, Double> entry : frequencyList) {
63              frequentConditionList.add(entry.getKey());
64          }
65      }
66
67      @Override
68      public TwoClassConfusionMatrix getTwoClassConfusionMatrix(Rule rule) {
69          TwoClassConfusionMatrix evaluationResult = new TwoClassConfusionMatrix();
70
71          if(rule.getBody().isEmpty()) {
72              evaluationResult.addTruePositives(this.root.getPos(rule.getHead().toString()));
73              evaluationResult.addFalsePositives(this.root.getNeg(rule.getHead().toString()));
74              return evaluationResult;
75          }
76
77          ArrayList<SeCoTree> coveredNodes = new ArrayList<SeCoTree>();
78          // find lowest condition in the tree and save all used attributes (in the rule) for later use.
79          ArrayList<Condition> conditionList = new ArrayList<Condition>(rule.getBody());
80          Condition lowestCondition = conditionList.get(0);
81          for(int freqCondIndex = this.frequentConditionList.size() - 1; freqCondIndex >= 0; freqCondIndex
                --) {
82              if(conditionList.contains(this.frequentConditionList.get(freqCondIndex))) {
83                  lowestCondition = this.frequentConditionList.get(freqCondIndex);
84                  break;
85              }
86          }
87
88          SeCoTree linkNode = this.headerTable.get(lowestCondition.toString());
89          while(linkNode != null) {
90              if(linkNode.isUsed() && isPathCoveredByRule(conditionList, linkNode)) {
91                  coveredNodes.add(linkNode);
92                  evaluationResult.addTruePositives(linkNode.getPos(rule.getHead().toString()));
93                  evaluationResult.addFalsePositives(linkNode.getNeg(rule.getHead().toString()));
94              }
95              linkNode = linkNode.getLink();
96          }
97
98          coveredPaths.put(ruleLabel(rule), coveredNodes);
99          evaluationResult.addFalseNegatives(this.root.getPos(rule.getHead().toString()) - evaluationResult
                .getNumberOfTruePositives());
100         evaluationResult.addTrueNegatives(this.root.getNeg(rule.getHead().toString()) - evaluationResult.
                getNumberOfFalsePositives());
101         return evaluationResult;
102     }
103
104     /**
105      * Removes the covered instances from the tree and therefore adjusts the
106      * counts in the tree for every covered node for the given rule.
107      *
108      * @param rule Rule for which the paths are checked and the counts adjusted.
109      */
110     public void adjustCoveredPathCounts(Rule rule) {
111         if(rule.getBody().isEmpty()) {
112             this.root = new SeCoTree();
113             return;
114         }
115
116         for (SeCoTree node : coveredPaths.get(ruleLabel(rule))) {
117             resetChildren(node.getChildren().values());
118
119             SeCoTree parentNode = node.getParent();
120             while (parentNode != null) {
121                 for (Map.Entry<String, Double> nodeEntry : node.getPosTable().entrySet()) {
122                     parentNode.adjustPosNeg(nodeEntry.getKey(), -nodeEntry.getValue());
123                 }
124                 parentNode = parentNode.getParent();
125             }
126             node.resetCounts();
127         }
```

```
128     }
129
130     private void resetChildren(Collection<SeCoTree> children) {
131         for (SeCoTree child : children) {
132             child.resetCounts();
133             resetChildren(child.getChildren().values());
134         }
135     }
136
137     private String ruleLabel(Rule rule) {
138         return rule.getHead().toString() + ":-" + rule.getBody();
139     }
140
141     /**
142      * Checks whether the node is covered by the given rule.
143      * @param rule Rule to be applied to that node
144      * @param node Node whose attribute is to be checked
145      * @return Returns only false then the attribute is used in the node and the rule but its values are
            different; else true
146      */
147     public boolean ruleCoversNode(Rule rule, SeCoTree node) {
148         Condition nodeCondition = node.getCondition();
149
150         for(Condition ruleCondition : rule.getBody()) {
151             if(ruleCondition.getAttr().index() == nodeCondition.getAttr().index())
152                 return (ruleCondition.compareTo(nodeCondition) == 0)? true : false;
153         }
154         return true;
155     }
156
157     /**
158      * @return The stored SeCoTree
159      */
160     public SeCoTree getTree() {
161         return root;
162     }
163
164     /**
165      * Creates a SeCoTree for every class value present in the given data set
166      * and sets the links in the header table. The created tree can be retrieved trough
167      * {@link #getTree()}.
168      *
169      * @param examples
170      *            The data set on which the tree is created.
171      *            Only the class values present in this set are considered!
172      * @param root
173      *            The root of the tree which new paths are added.
174      * @param headerTable
175      *            The header table which will hold the link nodes to every possible item label.
176      */
177     private void buildTree(Instances examples, SeCoTree root, Map<String, SeCoTree> headerTable,
178                            ArrayList<Condition> frequentConditionList) {
179
180         for (Instance instance : examples) {
181             SeCoTree parentNode = root;
182             Condition classCondition = createCondition(instance.classAttribute(), instance.classValue());
183             root.adjustPosNeg(classCondition.toString(), instance.weight());
184
185             for (Condition condition : frequentConditionList) {
186                 if (condition.covers(instance)) {
187                     SeCoTree childNode = parentNode.getChild(condition.toString());
188
189                     if (childNode == null) {
190                         SeCoTree newChild = new SeCoTree(condition, classCondition.toString(),
191                                 instance.weight(), parentNode);
192                         parentNode.addChild(newChild);
193                         updateHeaderTable(newChild, headerTable);
194                         parentNode = newChild;
195                     } else {
196                         childNode.adjustPosNeg(classCondition.toString(), instance.weight());
```

```
197                             parentNode = childNode;
198                         }
199                     }
200                 }
201             }
202         }
203 
204         /**
205          * Updates the links in the given header table (sets the links in the necessary nodes).
206          *
207          * @param node
208          *            The node which shall be added to the header table. The node
209          *            should currently not be present in the header table!
210          * @param headerTable
211          *            The header table which will be updated and to which the node is added.
212          */
213         private void updateHeaderTable(SeCoTree node, Map<String, SeCoTree> headerTable) {
214             SeCoTree linkNode = headerTable.get(node.getItemLabel());
215             if (linkNode != null) {
216                 node.setLink(linkNode);
217                 headerTable.put(node.getItemLabel(), node);
218             } else {
219                 headerTable.put(node.getItemLabel(), node);
220             }
221         }
222 
223         /**
224          * Creates a nominal or numeric condition according to the given attribute.
225          * Creates a nominal condition if the given attribute is neither.
226          *
227          * @param attr
228          *            An nominal or numeric Attribute.
229          * @param value
230          *            The value in internal representation.
231          * @return A nominal or numeric condition.
232          */
233         private Condition createCondition(Attribute attr, double value) {
234             return attr.isNumeric() ? new NumericCondition(attr, value) : new NominalCondition(attr, value);
235         }
236 
237         /**
238          * Checks whether one single Path is covered by the given condition list,
239          * which represents the rule body. The given SeCoTree node is the deepest node for the given rule.
240          *
241          * @param conditionList
242          *            A List with the conditions of the rule body in descending order.
243          * @param linkNode
244          *            Deepest node for the rule. Function traverses upwards from that node.
245          * @return Returns true if for every condition (of the condition list) the
246          *         value equals the value in the path; else false.
247          */
248         private boolean isPathCoveredByRule(ArrayList<Condition> conditionList, SeCoTree linkNode) {
249             SeCoTree pathNode = linkNode;
250             int uncoveredConditions = conditionList.size();
251 
252             while(!pathNode.isRoot()) {
253                 if(conditionList.contains(pathNode.getCondition())) {
254                     uncoveredConditions--;
255                     if(uncoveredConditions == 0) {
256                         return true;
257                     }
258                 }
259                 pathNode = pathNode.getParent();
260             }
261             return false;
262         }
263     }
```

```
1  package de.tu_darmstadt.ke.seco.models.vertical;
2
3  import java.io.Serializable;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  import de.tu_darmstadt.ke.seco.models.Condition;
8
9  public class SeCoTree implements Serializable {
10
11     private static final long serialVersionUID = -7675411382447693031L;
12     private Condition condition;
13     private SeCoTree link;
14     private Map<String, Double> pos;
15     private double sum;
16
17     private SeCoTree parent;
18     private Map<String, SeCoTree> children;
19
20     public SeCoTree(Condition condition, SeCoTree link, String classLabel, double pos, double neg,
           SeCoTree parent, Map<String, SeCoTree> children) {
21         this.condition = condition;
22         this.link = link;
23         this.pos = new HashMap<String, Double>();
24         this.pos.put(classLabel, pos);
25         this.sum = pos + neg;
26         this.parent = parent;
27         this.children = children;
28     }
29
30     public SeCoTree(Condition condition, SeCoTree link, SeCoTree parent, Map<String, SeCoTree> children)
           {
31         this.condition = condition;
32         this.link = link;
33         this.pos = new HashMap<String, Double>();
34         this.sum = 0;
35         this.parent = parent;
36         this.children = children;
37     }
38
39     public SeCoTree(Condition condition, String classLabel, double pos, SeCoTree parent) {
40         this(condition, null, classLabel, pos, 0, parent, new HashMap<String, SeCoTree>());
41     }
42
43     public SeCoTree() {
44         this(null, null, null, new HashMap<String, SeCoTree>());
45     }
46
47     public Condition getCondition() {
48         return this.condition;
49     }
50
51     public SeCoTree getLink() {
52         return link;
53     }
54
55     public double getPos(String label) {
56         return this.pos.get(label) == null? 0 : this.pos.get(label);
57     }
58
59     public double getNeg(String label) {
60         return this.pos.get(label) == null? this.sum : this.sum - this.pos.get(label);
61     }
62
63     public double getSum() {
64         return this.sum;
65     }
66
67     public Map<String, SeCoTree> getChildren() {
```

```
68          return this.children;
69      }
70
71      public SeCoTree getChild(String item) {
72          return this.children.get(item);
73      }
74
75      public SeCoTree getParent() {
76          return parent;
77      }
78
79      public String getItemLabel() {
80          return this.condition == null ? "ROOT" : this.condition.toString();
81      }
82
83      public boolean coversCondition(Condition condition) {
84          return this.pos.containsKey(condition.toString());
85      }
86
87      public boolean isRoot() {
88          return this.parent == null;
89      }
90
91      public boolean isUsed() {
92          return this.sum > 0;
93      }
94
95      public void adjustPosNeg(String label, double deltaPos) {
96          this.sum += deltaPos;
97          if (this.pos.get(label) != null) {
98              this.pos.put(label, this.pos.get(label) + deltaPos);
99          } else {
100             this.pos.put(label, deltaPos);
101         }
102     }
103
104     public void setParent(SeCoTree parent) {
105         this.parent = parent;
106     }
107
108     public void setLink(SeCoTree link) {
109         this.link = link;
110     }
111
112     public void addChild(SeCoTree child) {
113         this.children.put(child.getItemLabel(), child);
114     }
115
116     @Override
117     public String toString() {
118         return "(" + this.getItemLabel() + ":␣" + this.pos + "|␣SUM=␣" + this.sum + "␣-->␣" + this.
                children.values() + ")";
119     }
120
121     public boolean isLeaf() {
122         return this.children.isEmpty();
123     }
124
125     public Map<String, Double> getPosTable() {
126         return this.pos;
127     }
128
129     public void resetCounts() {
130         this.pos = new HashMap<String, Double>();
131         this.sum = 0;
132     }
133 }
```

A.2 SeCoID; Erstellung und Bearbeitung eines Baumes mit Pfad-IDs

```
1  private void buildTree(Instances examples, SeCoIDTree root, Map<String, SeCoIDTree> headerTable,
       ArrayList<Condition> frequentConditionList) {
2
3          for (Instance instance : examples) {
4                  SeCoIDTree parentNode = root;
5                  Condition classCondition = createCondition(instance.classAttribute(), instance.classValue
                       ());
6                  root.adjustPosNeg(classCondition.toString(), instance.weight());
7
8                  // inserting path
9                  boolean isNewPath = false;
10                 ArrayList<SeCoIDTree> actualPath = new ArrayList<SeCoIDTree>();
11                 for (Condition condition : frequentConditionList) {
12                         if (condition.covers(instance)) {
13                                 SeCoIDTree childNode = parentNode.getChild(condition.toString());
14
15                                 if (childNode == null) {
16                                         isNewPath = true;
17                                         SeCoIDTree newChild = new SeCoIDTree(condition, classCondition.
                                               toString(), instance.weight(), parentNode);
18                                         parentNode.addChild(newChild);
19                                         updateHeaderTable(newChild, headerTable);
20                                         parentNode = newChild;
21                                         actualPath.add(newChild);
22                                 } else {
23                                         childNode.adjustPosNeg(classCondition.toString(), instance.weight
                                               ());
24                                         parentNode = childNode;
25                                         actualPath.add(childNode);
26                                 }
27                         }
28                 }
29
30                 // set IDs for every node
31                 if(isNewPath) {
32                         UUID id = UUID.randomUUID();
33                         for(SeCoIDTree node : actualPath) {
34                                 node.addPathID(id);
35                         }
36                 }
37         }
38 }
39
40 public TwoClassConfusionMatrix getTwoClassConfusionMatrix(Rule rule) {
41         TwoClassConfusionMatrix evaluationResult = new TwoClassConfusionMatrix();
42
43         if(rule.getBody().isEmpty()) {
44                 evaluationResult.addTruePositives(this.root.getPos(rule.getHead().toString()));
45                 evaluationResult.addFalsePositives(this.root.getNeg(rule.getHead().toString()));
46                 return evaluationResult;
47         }
48
49         ArrayList<SeCoIDTree> coveredNodes = new ArrayList<SeCoIDTree>();
50         ArrayList<Condition> conditionList = new ArrayList<Condition>(rule.getBody());
51         ArrayList<Condition> sortedConditionList = new ArrayList<Condition>();
52         for(int freqCondIndex = this.frequentConditionList.size() - 1; freqCondIndex >= 0; freqCondIndex
               --) {
53                 if(conditionList.contains(this.frequentConditionList.get(freqCondIndex))) {
54                         sortedConditionList.add(this.frequentConditionList.get(freqCondIndex));
55                 }
56         }
57
58         Condition lowestCondition = sortedConditionList.get(0);
59         sortedConditionList.remove(0);
60         ArrayList<SeCoIDTree> lowestPathNodes = new ArrayList<SeCoIDTree>();
61         SeCoIDTree initLinkNode = this.headerTable.get(lowestCondition.toString());
62
```

```
63          while(initLinkNode != null) {
64                  lowestPathNodes.add(initLinkNode);
65                  initLinkNode = initLinkNode.getLink();
66          }
67
68          for(Condition condition : sortedConditionList) {
69                  SeCoIDTree linkNode = this.headerTable.get(condition.toString());
70                  ArrayList<SeCoIDTree> coveredLowestPathNodes = new ArrayList<SeCoIDTree>();
71
72                  while(linkNode != null) {
73                          if(!linkNode.isUsed())
74                                  continue;
75
76                          for(SeCoIDTree lowestPathNode : lowestPathNodes) {
77                                  // only one ID must be covered! b/c then all IDs are covered!
78                                  boolean covered = linkNode.getPathIDs().contains(lowestPathNode.
                                        getPathIDs().iterator().next());
79                                  // each node will be added just once! no need to check whether the lowest
                                         node is already included!
80                                  if(covered) {
81                                          coveredLowestPathNodes.add(lowestPathNode);
82                                  }
83                          }
84                          linkNode = linkNode.getLink();
85                  }
86                  lowestPathNodes = coveredLowestPathNodes;
87          }
88
89          for(SeCoIDTree coveredDeepestNode : lowestPathNodes) {
90                  evaluationResult.addTruePositives(coveredDeepestNode.getPos(rule.getHead().toString()));
91                  evaluationResult.addFalsePositives(coveredDeepestNode.getNeg(rule.getHead().toString()));
92                  coveredNodes.add(coveredDeepestNode); // for tree adjustments
93          }
94
95          coveredPaths.put(ruleLabel(rule), coveredNodes);
96          evaluationResult.addFalseNegatives(this.root.getPos(rule.getHead().toString()) - evaluationResult
                .getNumberOfTruePositives());
97          evaluationResult.addTrueNegatives(this.root.getNeg(rule.getHead().toString()) - evaluationResult.
                getNumberOfFalsePositives());
98          return evaluationResult;
99  }
```