Überblick und Implementierung von Algorithmen zur Klassifikation mittels Assoziationsregeln

Overview and Implementation of algorithms for classification using association rules

Bachelor-Thesis von Jens Brunn

Tag der Einreichung:

1. Gutachten: Prof. Johannes Fürnkranz
2. Gutachten: Eneldo Loza Mencia

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Überblick und Implementierung von Algorithmen zur Klassifikation mittels Assoziationsregeln
Overview and Implementation of algorithms for classification using association rules

Vorgelegte Bachelor-Thesis von Jens Brunn

1. Gutachten: Prof. Johannes Fürnkranz
2. Gutachten: Eneldo Loza Mencia

Tag der Einreichung:

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. September 2017

_____

(Jens Brunn)

Zusammenfassung

Diese Arbeit beschäftigt sich mit Klassifikationsalgorithmen, die auf Assoziationsregeln basieren. Diese Algorithmen unterscheiden sich von anderen Ansätzen, da Klassifikation mittels Assoziationsregeln im Vergleich zu vorherigen Ansätzen generell eine hohe Klassifikationsgenauigkeit erreicht, obwohl Assoziationsregeln ursprünglich nicht zur Klassifikation gedacht waren. Im Gegenzug müssen diese Algorithmen mit der Schwierigkeit zurecht kommen, dass es grundsätzlich eine sehr große Zahl an Assoziationen in einem Datensatz gibt und es entsprechend schwierig sein kann, alle zu finden.

In dieser Arbeit wird zunächst ein Einblick in diverse grundlegende Konzepte zum Regellernen und Klassifizieren gegeben. Danach werden diverse Algorithmen mit verschiedenen Ansätzen in der Berechnung von Assoziationsregeln, Auswahl von Assoziationsregeln für den Klassifizierer oder der Art der eigentlichen Klassifikation gegeben. Des Weiteren wurde im Rahmen der Arbeit der CBA-Algorithmus in ein Separate and Conquer Rule Learning Framework integriert und mit anderen Ansätzen zur Klassifikation, wie zum Beispiel einem Standardansatz zum Top-Down-Regellernen, verglichen. Dies bietet einen Ansatz, um Klassifikationsalgorithmen auf Basis von Assoziationsregeln sowohl untereinander als auch mit anderen Algorithmen zu vergleichen, ohne dass die Ergebnisse auf Grund von unterschiedlichen Implementierungen verfälscht werden.

Inhaltsverzeichnis

1 Einleitung

Eine grundlegende Problemstellung des maschinellen Lernens besteht darin, Beispiele mit gewissen Attributen in unterschiedliche Klassen zu unterteilen. Ein Ansatz dieses Problem zu lösen besteht darin Regeln zu lernen. Wenn ein Beispiel die Prämisse einer Regel erfüllt, dann kann die Regel zu der Konklusion kommen, dass das Beispiel zu einer bestimmten Klasse gehört. Um solche Regeln zu finden, gibt es wiederum viele verschiedene Ansätze. Der naive Ansatz besteht darin, eine möglichst gute Regel zu lernen und danach die Beispiele, die von dieser Regel klassifiziert werden, aus dem Trainingsdatensatz zu entfernen oder ihren Einfluss zu verringern. Dieser Schritt kann dann solange wiederholt werden, bis die Trainingsbeispiele ausreichend stark abgedeckt werden und die bisher gelernten Regeln zusammen ein Modell bilden.

In dieser Arbeit wird sich hauptsächlich mit einem neueren Ansatz von Algorithmen beschäftigt, bei denen nur sogenannte Assoziationsregeln verwendet werden. Generell handelt es sich bei Assoziationsregeln um Regeln, die dazu gedacht sind häufig auftretende Muster innerhalb eines Datensatzes zu finden. Ziel der Klassifikationsalgorithmen ist es nun, den Teil der Muster zu finden, die mit der jeweiligen Klasse der Beispiele zusammen hängen und diese Muster dann zur Klassifikation zu verwenden. Durch das Verwenden dieser Muster soll erreicht werden, dass die Klassifikation mit einer höheren Genauigkeit stattfinden kann, da alle Regeln auch auf dem gesamten Datensatz betrachtet relevant sind.

1.1 Motivation

In bisherigen individuellen Implementationen von diversen konkreten Algorithmen haben sich Assoziationsregeln generell als vielversprechend für die Nutzung zur Klassifikation gezeigt. Da es bisher allerdings keine einheitliche Implementierung der einzelnen Algorithmen gibt, ist es schwierig, die Algorithmen direkt mit konventionellen Regel-Lern-Algorithmen oder untereinander zu vergleichen.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, einen Überblick über existierende Algorithmen zur Klassifikation mittels Assoziationsregeln zu bieten, sowie die Implementierung des CBA-Algorithmus[1] innerhalb des SeCo-Frameworks[2]. Mithilfe dieser Implementierung soll dann ein Vergleich des CBA-Algorithmus mit anderen Regel-Lern-Algorithmen durchgeführt werden. Damit soll ein Grundstein gelegt werden, um es möglich zu machen, Algorithmen zur Klassifikation mittels Assoziationsregeln untereinander und mit konventionellen Regel-Lern-Algorithmen zu vergleichen.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit werden zunächst Grundlagen zu Regeln und insbesondere Assoziationsregeln, sowie zwei Algorithmen zum Finden von Assoziationsregeln vorgestellt. Des Weiteren werden Klassifizierer und insbesondere der Separate and Conquer Ansatz zum Regellernen vorgestellt.

Danach werden die Grundlagen der Klassifikation mithilfe von Assoziationsregeln erklärt und dadurch entstehende Probleme beleuchtet. Des Weiteren werden mit CBA[1], CMAR[3], CPAR[4] und RMR[5] einige konkrete Algorithmen zur Klassifikation mit Assoziationsregeln vorgestellt und miteinander verglichen.

Darauffolgend wird die Struktur des Separate and Conquer Frameworks erklärt und vorgestellt, welche Änderungen vorgenommen werden mussten, um den CBA-Algorithmus in diesem Framework zu realisieren. Abschließend werden einige Experimente, die mit der Implementation von CBA im SeCo-Framework durchgeführt wurden, vorgestellt, sowie ein Ausblick darauf gegeben, welche weiterführenden Arbeiten damit möglich sind.

2 Grundlagen

2.1 Regeln

Regeln haben generell die Form $A \rightarrow B$, wobei A und B jeweils eine Menge von Attributen sind. A stellt dabei die Menge der Prämissen und B die Menge der Konklusionen dar. Die Regel $x, y \rightarrow z$ sagt also aus, dass ein Beispiel, mit den Attributen x und y, vermutlich auch das Attribut z hat. Außerdem wird eine Regel r1 als allgemeiner im Vergleich zu einer zweiten Regel r2 definiert, wenn die Konklusionen von r1 und r2 gleich sind und die Menge der Prämissen von r1 eine Teilmenge der Prämissen von r2 ist.

2.2 Assoziationsregeln

Assoziationsregeln werden dazu eingesetzt, um Muster in Datenmengen darzustellen. Assoziationsregeln haben die Form $A \rightarrow B$, wobei A und B jeweils ein Set von Attributen sind. Bekannte Maße zur Evaluation sind Support, ein Maß für die Relevanz der Regel, und Konfidenz, ein Maß für die Genauigkeit der Regel. Der Support einer Regel gibt an, wie groß der Anteil der Datenmenge ist, auf den die Regel angewendet werden kann, und berechnet sich als $supp(A \rightarrow B) = supp(A \bigcup B) = \frac{n(A \bigcup B)}{n}$. Die Konfidenz berechnet sich als $conf(A \rightarrow B) = \frac{supp(A \bigcup B)}{supp(A)} = \frac{n(A \bigcup B)}{n(A)}$, wobei n(X) die Anzahl der Elemente der Trainingsmenge, in denen X enthalten ist, angibt.

Ein bekanntes Beispiel für Assoziationsregeln ist die sogenannte Warenkorbanalyse. Hierbei besteht jeder Eintrag aus einem Set von Elementen und es werden Muster zwischen diesen Sets gesucht. Eine beispielhafte Datenmenge und ein paar Assoziationsregeln können den Tabellen 1 und 2 entnommen werden.

Tabelle 1: Assoziationsregellerner Beipsieldatensatz

Liste 1	Bier, Chips, Wein, Brot
Liste 2	Apfel, Wein, Brot, Aufstrich, Milch
Liste 3	Aufstrich, Bier, Chips
Liste 4	Apfel, Milch, Bier, Wein
Liste 5	Brot, Aufstrich, Milch, Wein, Chips

Tabelle 2: Assoziationsregeln

Regel	Support	Konfidenz
Chips → Bier, Wein	0.2	0.33
Aufstrich, Brot → Milch	0.2	1.0

Assoziationsregellerner versuchen normalerweise alle Assoziationsregeln zu finden, die einen minimalen Support und eine minimale Konfidenz erfüllen, allerdings gibt es auch „Greedy“-Ansätze, bei denen die Menge der untersuchten Regeln eingeschränkt wird, um zu einem schnelleren Ergebnis zu kommen. In der Praxis können Assoziationsregeln zum Beispiel bei der Analyse von Einkaufslisten verwendet werden, um den Kunden beim Einkauf gezielt weitere Waren zu zeigen, an denen sie vermutlich interessiert sind.

2.3 Apriori-Algorithmus

Ein bekannter Algorthmus für das Finden von Assoziationsregeln ist der Apriori-Algorithmus[6] von Agrawal und Srikant. Der Apriori-Algorithmus unterteilt den Prozess, Assoziationsregeln zu finden, in zwei Schritte. Im ersten Schritt werden alle Item-Sets gefunden, die den minimalen Support erfüllen. Im zweiten Schritt werden aus diesen häufigen Item-Sets Assoziationsregeln gebildet, die die minimale Konfidenz erfüllen.

2.3.1 Häufige Item-Sets generieren

Um im ersten Schritt alle häufigen Item-Sets zu finden, wird zunächst für alle einelementigen Sets der Support ermittelt und alle häufigen Item-Sets gesammelt. Ausgehend von dieser Menge an kleinstmöglichsten häufigen Item-Sets werden alle neue Kandidaten für häufige Itemsets gebildet, die genau ein Element mehr haben. Dies geschieht, indem zu einem häufigen Item-Set der Größe N ein Element eines anderen häufigen Item-Sets der Größe N hinzugefügt wird, um ein Item-Set der Größe $N + 1$ zu bilden. Danach wird überprüft, ob jede Teilmenge dieses potentiell häufigen Item-Sets bereits als häufiges Itemset identifiziert wurde. Da dies eine Eigenschaft ist, die von allen häufigen Item-Sets erfüllt werden muss und der Algorithmus alle kleineren häufigen Item-Sets bereits vorher gefunden haben muss, ist es ausreichend,

nur Teilsets der Größe N, die im letzten Schritt gefunden wurden, zu überprüfen. Durch diesen Test kann die Anzahl der potentiell häufigen Item-Sets stark reduziert werden, bevor der tatsächliche Support der Item-Sets berechnet werden muss, was deutlich mehr Zeit in Anspruch nimmt. Wenn der tatsächliche Support berechnet wurde und den minimalen Support erfüllt, wird das Item-Set als häufiges Item-Set zusammen mit seinem Support gespeichert. Der Algorithmus hört erst auf Item-Sets der Größe $N+1$ zu finden, wenn es keine häufigen Item-Sets der Größe N gab. In diesem Fall ist dann direkt bekannt, dass alle häufigen Item-Sets im Datensatz gefunden wurden.

2.3.2 Assoziationsregel Generierung

Im zweiten Schritt werden aus diesen häufigen Item-Sets Assoziationsregeln erstellt. Zu diesem Zweck werden für alle häufigen Item-Sets A jeweils alle nicht-leeren Teilmengen B gefunden und je eine Regel der Form $B \rightarrow (A-B)$ erstellt und deren Konfidenz überprüft. Dies kann relativ schnell überprüft werden, da alle häufigen Teilsets bereits in einem früheren Schritt erzeugt wurden und ihr Support berechnet wurde. Dieser Schritt kann weiter gekürzt werden, indem die potentiellen Regeln in Reihenfolge der Größe der Konklusion der Regel untersucht werden, da keine Regel $a, b, c \rightarrow d$ die minimale Konfidenz erfüllen kann, solange nicht auch eine Regel $a, b \rightarrow c, d$ die minimale Konfidenz erfüllt und daher nicht generiert und überprüft werden muss.

2.4 FP-Growth

Ein weiterer interessanter Algorithmus zum Finden von häufigen Item-Sets ist der FP-Growth(Frequent Pattern Growth) Algorithmus von J. Han, J. Pei und Y. Yin. Bei diesem Algorithmus wird versucht, die Performance Probleme von Apriori zu lösen, indem die häufigen Item-Sets möglichst direkt aus den Trainingsbeispielen gelernt werden, anstatt zusätzlich noch große Mengen an Kandidaten zu generieren und auszuwerten. Damit kann der Aufwand vorallem für häufige Item-Sets mit vielen Attributen deutlich reduziert werden. Des Weiteren benutzt der Algorithmus eine FP-Tree(Frequent Pattern Tree) Struktur, um seine Ergebnisse zu speichern und ist damit auch bei der Speicherung effizienter als klassische Ansätze wie Apriori.

2.4.1 Frequent Pattern Tree

Die Idee eines FP-Tree baut darauf auf, dass man eine Baumstruktur für häufige Item-Sets bildet, wobei der Baum kompakt Daten für alle einelementigen häufigen Itemsets enthält und man daraus leicht Informationen über größere Item-Sets gewinnen kann.

Zu diesem Zweck werden zunächst alle einelementigen häufigen Item-Sets im Datensatz gesucht und nach ihrer Häufigkeit sortiert. Als Ergebnis des ersten Schrittes erhält man also eine Liste mit einelementigen häufigen Item-Sets und deren Häufigkeit im Datensatz. Aus der Anzahl der Erscheinungen eines häufigen Item-Sets kann man leicht den relativen Support des Item-Sets im gesamten Datensatz berechnen, und aus dem minimalen Support kann man leicht eine untere Grenze an Erscheinungen für häufige Item-Sets gewinnen, daher werden diese Anzahl der Erscheinungen im Datensatz im folgenden als absoluter Support betrachtet.

Danach wird ein zweites Mal jedes Beispiel des Datensatzes betrachtet und als Pfad seiner einelementigen häufigen Item-Sets im Baum gespeichert. Hierbei sind die Knoten für ein Beispiel ausgehend vom Wurzelknoten nach Häufigkeit sortiert. Wenn sich bei zwei Beispielen die Pfade zu Beginn überschneiden, dann teilen sich die beiden Beispiele auch Knoten in dem FP-Tree. Zusätzlich speichert jeder Knoten im FP-Tree auch die Anzahl an Beispielen, deren Pfad im Baum durch diesen Knoten verläuft. Zuletzt hat jeder Knoten noch einen Link zum nächsten Knoten in einem anderen Zweig, der das selbe einelementige Item-Set repräsentiert, sofern ein solcher Knoten existiert.

Eine wichtige Eigenschaft dieser Konstruktion ist, dass der resultierende Baum nicht expontentiell größer sein kann als der Datensatz, der repäsentiert wird, wie es in Apriori bei der Kandidatengenerierung der Fall sein kann. Dies lässt sich leicht zeigen, da jedes Beispiel des Datensatzes nur einen Pfad im Baum erzeugen kann und jeder Pfad durch die Anzahl der häufigen Items limitiert ist.

2.4.2 Häufige Item-Sets aus einem FP-Tree gewinnen

Um häufige Item-Sets aus dem FP-Tree zu gewinnen, werden der Reihe nach für alle einelementigen häufigen Item-Sets alle häufigen Item-Sets gesucht, die dieses Element enthalten. Indem mit den Elementen mit dem geringsten Support begonnen wird, kann garantiert werden, dass immer nur Pfade nach oben weiter verfolgt werden müssen, beziehungsweise untersuchte Elemente aus dem Baum entfernt werden können. Durch dieses Entfernen von Knoten kann es nicht passieren, dass einzelne Item-Sets übersehen werden, da ein häufiges Itemset, dass neben dem aktuellen Element auch ein Element mit niedrigerem Support enthält, bei der Untersuchung von dem Element mit niedrigerem Support bereits gefunden wurde.

Um ein Element zu untersuchen wird ein neuer FP-Tree gebildet, bei dem nur noch das Subset von Pfaden, in denen das entsprechende Element enthalten ist, berücksichtigt werden und entsprechend die Häufigkeit der einzelnen Knoten

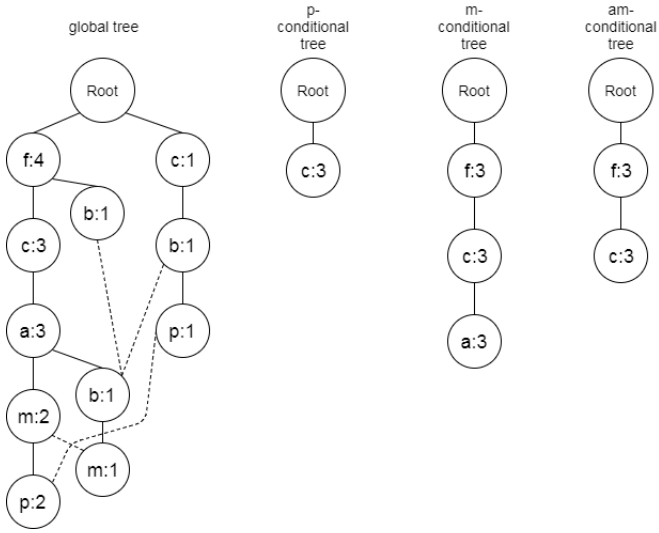

Abbildung 1: Bedingte FP-Trees zu einem globalen FP-Tree mit einem minimalen Support von 3

angepasst wird. Durch diese Anpassung der Häufigkeit kann es sein, dass einige Elemente innerhalb von Pfaden nicht mehr den Grenzwert für den minimalen Support erreichen und daher nicht wieder aufgenommen werden. Dadurch kann es weiterhin möglich sein, zwei verschiedene Pfade komplett oder teilweise zusammen zu legen. Wenn ein solcher bedingter FP-Tree keine Knoten mehr enthält, ist der Algorithmus fertig und nur das betrachtete Element wird als häufiges Item-Set zurückgegeben. Ansonsten muss er erneut auf dem neuen bedingten Baum angewendet werden und das Element wird zu allen Ergebnissen innerhalb des rekursiven Aufrufs auf dem bedingten Baum hinzugefügt. Ein Beispiel, wie bedingte FP-Trees erstellt werden, ist in Abbildung 1 zu sehen.

In diesem Beispiel gibt es im globalen FP-Tree zwei Pfade zu p. Zum einen den Pfad fcamp mit einem Support von 2 und zum anderen den Pfad cbp mit dem Support 1. Insgesamt erreichen f, a, m und b damit jeweils keinen Support von 3 und fallen aus dem p-basierten Baum heraus. Als Resultat bleibt daher nur noch ein Pfad c mit einem Support von 3 in dem neuen Baum übrig. Bei einem Versuch, daraus einen pc-basierten Baum zu bilden, fällt auf, dass es keine Knoten in diesem Baum gibt und daher zusätzlich zu dem Item-Set p nur ein weiteres häufiges Item-Set bestehend aus pc zurück gegeben wird.

Für den m basierten Baum gibt es erneut zwei Pfade, allerdings diesmal mit einem gemeinsamen Anfang. Für den Algorithmus macht das allerdings keinen Unterschied, da für den gesamten Pfad jeweils der Support von dem betrachteten Element entscheidend ist. Das heißt, es gibt diesmal den Pfad fcam mit einem Support von 2 und den fcabm-Pfad mit einem Support von 1. Nur das Element b erreicht damit nicht den minimalen Support von 3 und fällt aus dem m-basierten Baum heraus. Dadurch ist es möglich, die anderen beiden Pfade zusammen zu fassen.

Zuletzt wird noch der am-basierte Baum gezeigt, bei dem es vorher nur einen Pfad gab, der den minimalen Support erreicht hat, und es daher auch als Resultat weiterhin genau diesen einen Pfad gibt. Lediglich der Knoten a taucht in diesem Pfad nicht weiter auf.

Führt man diesen Algorithmus weiter aus und bildet auch cm- und fm-basierte Bäume, sowie sämtliche daraus entstehende sub-Bäume, dann erhält man die häufigen Item-sets fcam, fam und am aus dem am-basierten Baum, fcm und cm aus dem cm basierten Baum und fm aus dem fm-basierten Baum. Zusammen mit dem Item-Set m gibt es also insgesamt 7 häufige Item-Sets, die beim Betrachten des Elements m zurück gegeben werden. Die anderen Elemente des globalen Baums können dann nach demselben Prinzip untersucht werden.

Dieser Prozess kann noch weiter vereinfacht werden, da man die häufigen Item-Sets auch direkt aus diesem Pfad ablesen kann, wenn es in einem FP-Tree nur noch einen Pfad gibt. In diesem Fall sind alle möglichen Kombinationen von Elementen des Pfades auch häufige Item-Sets und haben den Support von dem Item mit dem geringsten Support aus dem jeweiligen Set. Algorithmus 1 zeigt den Pseudocode zum Auswerten eines FP-Trees.

2.5 Klassifikation

Beim Basisproblem der Klassifikation betrachtet man eine Datenmenge mit Elementen, bei denen jedes Element ein Set von Attributen besitzt und zu einer Klasse zugeordnet werden kann. Typischerweise gibt es eine Trainingsmenge von bereits klassifizierten Beispielen, mit deren Hilfe versucht wird, ein Modell zu entwickeln, das neue nicht klassifizierte Beispiele korrekt klassifizieren kann. Dieses Modell bezeichnet man auch als Klassifizierer. Um einen möglichst effizienten

Algorithm 1 FP-growth

```
1: Procedure FP-growth(tree, attribute):
2: frequentItemSets = ∅;
3: if Tree contains a single Path p then
4:     for each Combination c of the nodes in p do
5:         set = c ∪ attribute
6:         set.support = minimum support of nodes in c;
7:         frequentItemSets = frequentItemSets ∪ set;
8:     end for
9: else
10:    for each attribute a_i in the header of tree do
11:        generate Combination c = a_i ∪ attribute with c.support = a_i.support;
12:        construct c's conditional pattern base and conditional FP-Tree t;
13:        if tree ≠ ∅ then
14:            frequentItemSets = frequentItemSets ∪ FP-GROWTH(tree,c);
15:        end if
16:    end for
17: end if
18: return frequentItemSets;
```

Klassifizierer zu bilden, kommt es häufig vor, dass zunächst viele potentielle Regeln gefunden werden und diese danach wieder gekürzt oder entfernt werden müssen. Diesen Prozess bezeichnet man als Pruning.

Ein häufiges Anwendungsgebiet für Klassifizierer ist das Finden von Spam in einem E-Mail Postfach oder allgemein das Vorhersagen von Reaktionen auf bestimmte Situationen. Ein Beispiel für einen Datensatz und ein paar Regeln können den Tabellen 3 und 4 entnommen werden.

Tabelle 3: Klassifikation Beispieldatensatz

outlook	temperature	humidity	windy	surfing
sunny	mild	high	FALSE	no
sunny	mild	high	True	yes
overcast	hot	high	FALSE	no
rainy	hot	high	True	yes
rainy	cool	normal	True	no

Tabelle 4: Klasifikationsregeln

outlook = sunny, windy = TRUE → surfing = yes
windy = FALSE → surfing = no

Ein einzelner Klassifizierer besteht nun aus einer Menge von Regeln und kann diese auf verschiedenen Wegen zur Klassifikation benutzen. Eine Variante besteht darin, eine sogenannte Decision List zu bilden. Das heißt, dass man allen Regeln eine Priorität zuordnet und damit die Regeln in eine eindeutige Reihenfolge sortiert. Bei der Klassifikation eines neuen Beispiels wird dann für die Regeln der Reihe nach überprüft, ob sie auf das Beispiel zutreffen oder nicht. Die erste Regel, die auf das Beispiel zutrifft, gibt dann vor, welche Klasse dem Beispiel zugeordnet wird.

Damit weiterhin sichergestellt werden kann, dass alle möglichen Beispiele klassifiziert werden können, muss die Menge an Regeln entweder direkt erschöpfend sein und jede Kombination von Attributen mit mindestens einer Regel abdecken oder es muss eine zusätzliche sogenannte Default-Regel am Ende der Decision List angehängt werden. Eine Default-Regel ordnet dann allen verbleibenden Beispielen eine Klasse zu. Diese Default-Regel kann zum Beispiel allen Beispielen die Klasse zuordnen, die in den Trainingsdaten am häufigsten aufgetreten ist oder, falls nicht alle Trainingsbeispiele von dem Klassifizierer behandlet werden, die Klasse, die unter den verbleibenden nicht behandelten Trainingsbeispielen am häufigsten auftritt.

Eine weitere Variante besteht darin, dass man die Regeln darüber abstimmen lässt, welche Klasse vorhergesagt wird, indem von allen relevanten Regeln die vorhergesagten Klassen gesammelt und aufsummiert werden und am Ende die Klasse vorhergesagt wird, die von den meisten Regeln angenommen wurde. Ein Problem mit diesem Ansatz liegt allerdings darin, dass man zusätzlich noch den Fall, dass zwei oder mehr Klassen gleich oft vorhergesagt wurden, lösen muss.

Dies kann zum Beispiel durch Prioritäten der einzelnen Regeln, wie es sie bei der Decision List auch gibt, gelöst werden. Weiterhin muss es auch hier wieder eine Default Regel geben, um Beispiele, für die es keine relevanten Regeln gibt, klassifizieren zu können. Diese Variante kann noch weiter ausgebaut werden indem man den einzelnen Regeln jeweils unterschiedliche Gewichte zuordnet, je nach dem wie viel Vertrauen man in die einzelnen Regeln hat. In diesem Fall wird dann nicht die Anzahl der Regeln, die eine bestimmte Klasse vorhersagen, aufsummiert, sondern stattdessen die Gewichte der jeweiligen Regeln. Mit diesen Gewichten ist es möglich auch Regeln, die häufig von den restlichen Regeln überstimmt werden sollen und nur auf einer kleinen Gruppe von Beispielen tatsächlich wichtig sind in den Klassifizierer aufzunehmen, ohne dadurch die Ergebnisse für andere Beispiele zu verfälschen.

Des Weiteren gibt es noch eine weitere ausgebaute Variante dieses Ansatzes, bei dem nur die k besten Regeln auf einem Beispiel zur Klassifikation des Beispiels benutzt werden. Im Gegensatz zu dem Ansatz mit Gewichten wird hier mehr Wert darauf gesetzt, dass es immer eine gewisse Anzahl an Regeln gibt, die einen großen Einfluss auf die Klassifikation haben und die niedrig priorisierten Regeln nicht alle von einer Regel mit hoher Priorität überstimmt werden können.

2.5.1 Separate and Conquer

„Separate and Conquer“(dt. trennen und erobern) beschreibt einen allgemeinen Ansatz, wie Regeln zur Klassifikation bestimmt werden können. Die Grundidee hinter dem Ansatz besteht darin, dass der Algorithmus nacheinander Regeln lernt und nach jeder gelernten Regel die durch diese Regel erfassten Beispiele aus dem Datensatz entfernt, bevor eine weitere Regel gelernt wird. Wann genau der Algorithmus aufhört, weitere Regeln zu lernen, kann sich, je nachdem um welchen genauen Algorithmus es sich handelt, unterscheiden. Es ist zum Beispiel möglich aufzuhören, wenn es keine oder nicht mehr genügend weitere positive Beispiele in den Trainingsdaten gibt, die klassifiziert werden könnten, oder wenn keine Regel gefunden werden kann, die einen Grenzwert in einer bestimmten Heuristik erreicht. Des Weiteren ist es möglich, die Theorie, die so gefunden wurde, danach noch weiter mit Hilfe von einem beliebig komplexen Post-Prozessor zu verbessern. Der Pseudocode für diesen Ansatz kann Algorithmus 2 entnommen werden.

Algorithm 2 Separate and Conquer[7]

```
1:  theory = ∅;
2:  while POSITIVE(examples) ≠ ∅ do
3:      rule = FINDBESTRULE(examples);
4:      covered = COVER(rule, examples);
5:      if RULESTOPPINGCRITERION(theory, rule, examples) then
6:          exit while
7:      end if
8:      theory = theory ∪ rule ;
9:      examples = examples \ covered;
10: end while
11: theory = POSTPROCESS(theory);
12: return theory;
```

Wie genau die jeweils nächste Regel in diesem Ansatz gefunden werden kann unterscheidet sich stark je nachdem um welchen Algorithmus es sich genau handelt. Grundsätzlich wird aber zunächst eine Regel initialisiert und danach ausgehend von dieser Regel solange versucht, bessere Regeln zu finden, bis ein stoppingCriterion erfüllt ist. Dabei ist es durchaus möglich, dass in späteren Schritten mehrere potentielle Regeln weiter verfolgt werden. Im Pseudocode Algorithmus 3 spiegelt sich diese Möglichkeit durch die filterRules und selectCandidates Methoden wieder. Bei einem klassischen top-down Ansatz würde initializeRule etwa eine Regel mit der Prämisse true zurück geben, refineRule würde eine weitere Bedingung zu der Prämisse hinzufügen, selectCandidates und filterRules würden zusammen dafür sorgen, dass immer nur die beste Regel aus den potentiellen Regeln ausgewählt wird, und evaluateRule würde die Genauigkeit der Regel bestimmen. Der Pseudocode hierfür findet sich in Algorithmus 3.

Algorithm 3 findBestRule method[7]

```
1:  initRule = initializeRule(examples);
2:  initVal = evaluateRule(initRule);
3:  bestRule = (initVal, initRule);
4:  rules = bestRule;
5:  while rules ≠ ∅ do
6:      candidates = selectCandidates(rules, examples);
7:      rules = rules \ candidates;
8:      for each candidate ∈ candidates do
9:          refinements = refineRule(rules, examples);
10:         for each refinement ∈ refinements do
11:             evaluation = evaluateRule(refinement, examples);
12:             if !stoppingCriterion(refinement, examples) then
13:                 newRule = (evaluation, refinement);
14:                 rules = insertSort(newRule, rules);
15:                 if newRule>bestRule then
16:                     bestRule = newRule;
17:                 end if
18:             end if
19:         end for
20:     end for
21:     rules = filterRules(rules, examples);
22: end while
23: return bestRule;
```

3 Algorithmen zur Klassifikation mittels Assoziationsregeln

Das Ziel der Klassifikation mittels Assoziationsregeln ist es, dass sämtliche Regeln des Klassifizierers nicht nur lokal auf dem Teil der noch zu klassifizierenden Beispiele, sondern auch global auf den gesamten Datensatz gesehen relevant sind. Dazu gibt es allerdings generell einige Probleme zu lösen, da es grundsätzlich sehr viele Assoziationen in einem Datensatz gibt und alle zu finden sehr zeitaufwendig sein kann. Außerdem muss aus diesen Assoziationen dann noch eine Menge an Regeln gebildet werden, die auch unbekannte Beispiele gut klassifizieren kann. Des Weiteren kann ein zu niedriger Support leicht zu sogenanntem „overfitting“ führen, während ein zu hoher Support sehr präzise Regeln ignoriert. Overfitting beschreibt hierbei, dass eine Regel einzelne Beispiele, anstatt von allgemeinen Mustern im Datensatz, beschreibt. Dies ist ein Problem, da Datensätze häufig ein Rauschen, das heißt einzelne fehlerhafte Testbeispiele, enthalten.

Weiterhin gibt es das Problem, dass Datensätze grundsätzlich auch numerische Attribute enthalten können, die von Assoziationsregeln nicht direkt behandelt werden können. Das heißt, dass alle Algorithmen, die Assoziationsregeln verwenden, zunächst alle Attribute diskretisieren müssen. Für diesen preprocessing-Schritt kann ein beliebiger bekannter Diskretisierungsalgorithmus verwendet werden.

3.1 CBA

Der CBA-Algorithmus (Classification Based on Associations)[1] ist einer der ersten erfolgreichen Ansätze zur Klassifikation mittels Assoziationsregeln und lässt sich zusätzlich zur Diskretisierung in zwei Teile unterteilen.

Im ersten Schritt werden alle potentiellen Klassen-Assoziationsregeln ermittelt. Zu diesem Zweck wird im Algorithmus ein abgewandelter Apriori-Algorithmus verwendet. Die erste Änderung zum klassischen Apriori-Algorithmus ist nötig, da zur Klassifikation nur die Untermenge aller Assoziationsregeln mit ausschließlich dem Klassenattribut im Head der Regel interessant ist. Zu diesem Zweck werden in der CBA-Version des Algorithmus Rule-Items anstelle von Condition-Sets betrachtet. Jedes Rule-Item besteht dabei aus einer Menge von Conditions und einer Klasse. Der Support von Rule-Items wird berechnet, indem der Anteil an Beispielen, die alle Conditions erfüllen und in der entsprechenden Klasse liegen, bestimmt wird. Bei der Erweiterung von Rule-Items können einfach die Conditionsets erweitert werden, mit der Einschränkung, dass als Teilmenge nur Rule-Items in Frage kommen, die das selbe Klassenattribut haben.

Algorithm 4 CBA rule generation algorithm (vlg. [1])

```
1: frequentItems = largeRuleitems_1;
2: rules = genRules(frequentItems);
3: while frequentItems ≠ ∅ do
4:     candidates = candidateGen(frequentItems);
5:     for each candidate ∈ candidates do
6:         candidate.support = calculateSupport(candidate, examples);
7:     end for
8:     frequentItems = {candidate ∈ candidates | minsup ≤ candidate.support};
9:     rules = genRules(frequentItems) ∪ rules;
10: end while
11: prunedRules = pruneRules(rules);
12: return prunedRules;
```

Die Erstellung einer Regel ist danach trivial, da jedes Rule-Item direkt in eine Regel umgeformt werden kann, bei der das Conditionset den Body und die Klassenvariable den Head der Regel bilden.

Im letzten Schritt wird aus den einzelnen Klassen-Assoziationsregeln ein Klassifizierer entwickelt. Um einen Klassifizierer zu bilden, werden die Regeln zunächst sortiert. Dabei steht eine Regel r vor einer Regel s, wenn r

- eine höhere Konfidenz hat.
- die gleiche Konfidenz, aber einen höheren Support hat.
- die gleiche Konfidenz und den gleichen Support hat, aber früher generiert wurde.

Der einfachste Ansatz, um aus dieser geordneten Menge von Regeln einen Klassifizierer zu bilden, besteht darin, zu überprüfen, ob die Regel mit der höchsten Priorität ein Beispiel abdeckt. Sofern das der Fall ist, kann die Regel zu dem Modell hinzugefügt werden und alle Beispiele, die von der Regel abgedeckt werden, können entfernt werden. Dieses Modell kann dann weiter verbessert werden, indem der Punkt gesucht wird, bei dem die wenigsten Fehler gemacht werden, wenn alle noch nicht abgedeckten Beispiele einer Default-Klasse zugeordnet werden, und alle Regeln nach diesem Punkt verworfen werden.

Algorithm 5 Basic CBA classifier generation algorithm (vgl. [1])

```
1: classifier = SORT(rules);
2: for each Rule r ∈ classifier do
3:     temp = ∅;
4:     for each example e ∈ examples do
5:         if e satisfies the condition of r then
6:             temp = temp ∪ e.id;
7:             mark r if it correctly classifies e;
8:         end if
9:         if r is marked then
10:            insert r at the end of classifier;
11:            delete all the instances with the ids in temp from examples;
12:            calculate a default class for classifier;
13:            compute the total number of errors of classifier;
14:        end if
15:    end for
16: end for
17: Find the first rule r in classifier with the lowest total number of errors
18: Delete all rules after r in classifier;
19: Add the default class associated with r to end of classifier,
20: return classifier;
```

3.1.1 Alternativer Algorithmus zur Klassifizierer Konstruktion

Bei dem Alternativen Ansatz zur Klassifizierer Konstruktion wird sich die Eigenschaft zu nutze gemacht, dass wir zu jedem Beispiel die Regel mit der höchsten Priorität, die das Beispiel abdeckt und richtig (im folgenden c-rule) beziehungsweise falsch (im folgenden w-rule) klassifiziert, bestimmen können. Danach können anhand dieser Regeln die Beispiele in mehrere Kategorien unterteilt werden. Für alle Beispiele, bei denen die c-rule eine höhere Priorität als die w-rule hat, ist bekannt, dass die c-rule Teil des vorläufigen Modells sein muss, da sie die Regel mit der höchsten Priorität ist, die das Beispiel abdeckt und mit diesem Beispiel auch mindestens ein Beispiel korrekt klassifiziert. Da somit bekannt ist, dass die Regel Teil des Modells sein wird, kann die Regel zusätzlich markiert werden, damit sie bei der Entscheidung von komplizierteren Beispielen benutzt werden kann. Für den Fall, dass die w-rule eine größere Priorität als die c-rule hat, können wir jedoch keine direkte eine Aussage über die entsprechende c- oder w-rule machen.

Wenn bei einem solchen Beispiel zusätzlich die w-rule wegen einem anderen Beispiel im Modell aufgenommen werden muss, dann wird das Beispiel von dem Modell als falsch klassifiziert und c-rule muss nicht direkt zu dem Modell hinzugefügt werden. Für Beispiele, deren w-rule nicht im Modell aufgenommen werden muss, müssen auch noch weitere Regeln, die das Beispiel potentiell falsch klassifizieren, also zwischen w-rule und c-rule liegen, überprüft werden. Da der Alternative Ansatz zur Klassifizierer Konstruktion insgesamt deutlich komplexer ist, wird er in der folgenden detaillierteren Beschreibung in 3 Stufen unterteilt.

Um nun für alle Beispiele die relevante Regel im endgültigen Modell zu finden, müssen zunächst Informationen über das Beispiel gespeichert werden und danach erneut überprüft werden, nachdem alle Beispiele mit eindeutiger c-rule gefunden wurden. Zu diesem Zweck werden in der ersten Stufe des Algorithmus alle c-rules mit der Anzahl an Beispielen, die sie klassifizieren, gespeichert. Außerdem werden alle c-rules, die eine höherer Priorität als die entsprechende w-rule auf einem Beispiel haben, als garantierte Regel im Modell markiert und gesondert gespeichert. Zuletzt werden alle Beispiele deren w-rule eine höhere Priorität als die c-rule hat, in einem Set zusammen mit ihrer c- und w-rule gespeichert. (vgl. Algorithmus 6)

In der zweiten Stufe des Algorithmus (vgl. Algorithmus 7) werden jetzt nur noch Beispiele betrachtet, deren c-rule eine niedrigere Priorität als ihre w-rule haben. Da Regeln mit hoher Priorität nach Definition vergleichsweise viele Beispiele richtig klassifizieren, gibt es einen sehr großen Teil an Beispielen, die nur ein einziges Mal in der ersten Phase des Algortihmus betrachtet werden müssen.

Zunächst gibt es, wie bereits zu Anfang erklärt, den einfachen Fall, dass die w-rule des Beispiels in der ersten Stufe als garantierte Regel im Modell markiert wurde, und das Beispiel falsch klassifizieren wird. In diesem Fall muss lediglich die Anzahl der erfassten Beispiele für c-rule und w-rule aktualisiert werden. Für den komplizierteren Fall, dass die w-rule bei einem Beispiel noch nicht markiert wurde, müssen zunächst alle Regeln, die dieses Beispiel als falsch klassifizieren könnten, gefunden werden. Dies betrifft Regeln, die sowohl c-rule eines anderen Beispiels sind, als auch eine höhere Priorität verglichen mit der c-rule des aktuellen Beispiels, haben. Diese Regeln werden nun als potentielle Regel im Klassifizierer betrachtet, daher muss für jede dieser Regeln die c-rule, die sie womöglich ersetzt, gespeichert werden und die Anzahl der Beispiele, die sie möglicherweise abdeckt, angepasst werden. Außerdem werden alle betroffenen Kandidaten Regeln

Algorithm 6 Alternativer CBA classifier generation algorithm Stage 1 (vgl. [1])

```
 1: c-rules = ∅;                                                    ▷ all c-rules
 2: priorityC-rules = ∅;          ▷ c-rules with higher priority than w-rule on at least one example
 3: metadata = ∅;                          ▷ collection of (exampleId, exampleClass, c-rule, w-rule)
 4: for each example e ∈ examples do
 5:     c-rule = maxCoverRule(c, e);                    ▷ c is the set of rules that correctly classify e
 6:     w-rule = maxCoverRule(w, e);                    ▷ w is the set of rules that wrongly classify e
 7:     c-rules = c-rules ∪ c-rule;
 8:     c-Rule.classCasesCovered[e.class]++;
 9:     if c-Rule ≻ w-Rule then
10:         priorityC-rules = priorityC-rules ∪ {c-Rule};
11:         mark c-Rule;
12:     else
13:         metadata = metadata ∪ (e.id, e.class, c-Rule, w-Rule);
14:     end if
15: end for
```

zu der in Stufe 1 des Algorithmus erwähnten Menge an Regeln, die garantiert im endgültigen Klassifizierer auftauchen, hinzugefügt.

Algorithm 7 Alternativer CBA classifier generation algorithm Stage 2 (vgl. [1])

```
 1: for each entry (e.id, e.class, cRule, wRule) ∈ metadata do
 2:     if w-Rule is marked then
 3:         c-Rule.classCasesCovered[e.class]–;
 4:         w-Rule.classCasesCovered[e.class]++;
 5:     else
 6:         potentialW-rules = allCoverRules(c-rules, e.id.case, cRule);
 7:         for each rule w ∈ wSet do
 8:             w.replace = w.replace ∪ {(cRule, e.id, e.class)};
 9:             w.classCasesCovered[y]++;
10:         end for
11:         priorityC-rules = priorityC-rules ∪ potentialW-rules;
12:     end if
13: end for
```

In der letzten Stufe des Algorithmus müssen nun nur noch die Regeln, die den Klassifizierer bilden, ausgewählt werden. Dies gelingt, indem für alle Regeln festgestellt wird, ob sie im Klassifizierer benötigt werden und ab welchem Zeitpunkt die Regeln schlechter als die Default-Regel sind. Zunächst werden die potentiellen Regeln daher wieder nach ihrer Priorität sortiert. Danach wird der Reihe nach für jede Regel überprüft, ob sie weiterhin Beispiele korrekt klassifiziert, und wenn das der Fall ist, wird die Regel in den Klassifizierer aufgenommen. Bei anderen Regeln, die teilweise die gleichen Beispiele erfasst hätten, muss dann die Anzahl der erfassten Beispiele aktualisiert werden. Außerdem wird berechnet, wie viele Fehler der Klassifizierer machen würde, wenn dies die letzte Regel wäre, die in den Klassifizierer aufgenommen wird. Zuletzt muss nur noch die erste Regel ausgewählt werden, bei der der Klassifizierer die wenigsten Fehler macht, und alle darauffolgenden Regeln wieder aus dem Klassifizierer entfernt werden. Zusammen mit einer Default-Regel bilden die restlichen Regeln dann den Klassifizierer. Details können auch noch als Pseudocode in Algorithmus 8 gesehen werden.

3.2 CMAR

Der CMAR(Accurate and Efficient Classification Based on Multiple Class-Association Rules) Algorithmus[3] versucht, den Ansatz der Klassifikation mittels Assoziationsregeln zu verbessern, indem anstatt nur einer Regel mehrere Assoziationsregeln zur eigentlichen Klassifikation benutzt werden. Die Autoren schlagen vor, mehrere Assoziationsregeln zur Klassifikation eines Beispiels zu benutzen, um zu verhindern, dass eine einzelne Regel mit hoher Konfidenz, aber niedrigem Support stärker bewertet wird, als mehrere Regeln mit höherem Support und etwas niedrigerer Konfidenz. Des Weiteren wird bei diesem Ansatz eine andere Datenstruktur verwendet, um die Anforderungen an den Speicher zu reduzieren.

Algorithm 8 Alternativer CBA classifier generation algorithm Stage 3 (vgl. [1])

```
1: classDistr = compClassDistri(examples);
2: ruleErrors = 0;
3: priorityC-rules = sort(priorityC-rules);
4: for each Rule r in priorityC-rules in sequence do
5:     if c.classCasesCovered[r.class] ≠ 0 then
6:         for each Entry (rul, dId, y) in r.replace do
7:             if the dId instance has been covered by a previous r then
8:                 r.classCasesCovered[y]–;
9:             else
10:                rul.classCasesCovered[y]–;
11:            end if
12:            ruleErrors = ruleErrors + errorsOfRule(r);
13:            classDistr = update(r, classDistr);
14:            defaultClass = selectDefault(classDistr);
15:            defaultErrors = defErr(defaultClass, classDistr);
16:            totalErrors = ruleErrors + defaultErrors;
17:            theory = append(theory, r, defaultClass, totalErrors);
18:        end for
19:    end if
20: end for
21: Find the first Rule p in theory with the lowest totalErrors and discard all the rules after p from theory;
22: Add the default class associated with p to the end of theory;
23: return theory;
```

3.2.1 CMAR Suche von Assoziationsregeln

Im ersten Schritt des CMAR Algorithmus werden wieder alle Klassen-Assoziationsregeln gesucht, allerdings wird diesmal eine leicht veränderte Variante des in Abschnitt 1.4 vorgestellten FP-Growth Algorithmus benutzt. Die einzige notwendige Änderung am normalen FP-Growth Algorithmus, um Klassenassoziationsregeln zu erhalten, besteht darin, dass zur Klassifikation jedem Itemset zusätzlich eine Klasse zugeordnet werden muss. Im FP-Tree wird dazu einfach bei den Blattknoten des Baums nicht nur ein Support gespeichert, sondern ein oder mehrere Support Werte in Abhängigkeit von den Klassen, die dem Set zugeordnet werden können. Diese seperaten Support Werte können auch anstelle des globalen Supports benutzt werden, um zu sehen, ob ein Item-Set den minimalen Support erfüllt, da nur dann eine Regel gefunden werden kann, die sowohl das Item-Set als Prämisse, als auch eine Klasse als Konklusion hat, und den minimalen Support erfüllt.

Für die somit gefundenen Regeln muss nun noch die Konfidenz berechnet werden, bevor sie in einer Struktur ähnlich zu einem FP-Tree gespeichert werden kann. Um die Regeln in dieser Struktur zu speichern, werden die Prämissen der Regeln wieder wie häufige Item-Sets betrachtet und die Konklusion, zusammen mit Support und Konfidenz der Regel, im letzten Element des Pfades gespeichert. Diese Struktur wird als CR-Tree bezeichnet.

3.2.2 Pruning in CMAR

Im zweiten Schritt des Algorithmus werden die Regeln gepruned, um die Anzahl an Regeln, die im Klassifizierer verwendet werden, zu limitieren. Im Vergleich zu CBA ist dieser Schritt erforderlich, da nicht nur die Regel mit der höchsten Priorität, sondern prinzipiell alle vorhandenen Assoziationsregeln zur Klassifikation eines Beispiels benutzt werden. Da im ersten Schritt eine sehr große Menge an Regeln gefunden werden kann, müssen diese Regeln noch einmal gefiltert werden, um Regeln, die zu overfitting neigen, oder einfach nur keinen weiteren Fortschritt bringen, zu entfernen. Damit kann ein effizienterer und genauerer Klassifizierer gebildet werden.

Zu diesem Zweck werden die Regeln, ähnlich wie im CBA-Algorithmus, nach ihrer Priorität sortiert. Hierbei steht eine Regel r vor einer Regel s, wenn r

- eine höhere Konfidenz hat.
- die gleiche Konfidenz, aber einen höheren Support hat.
- die gleiche Konfidenz und den gleichen Support hat, aber weniger Attribute als Prämisse hat.
- die gleiche Konfidenz, den gleichen Support und die gleiche Menge an Attributen in der Prämisse hat, aber früher generiert wurde.

Wenn es innerhalb dieser Ordnung von Regeln eine Regel gibt, die allgemeiner ist als eine Regel mit niedrigerer Priorität, dann wird die Regel mit niedrigerer Priorität verworfen und nur die Regel mit höhrerer Priorität abgespeichert. Dieser Schritt lässt sich durchführen, während Regeln in den CR-Tree eingefügt werden, da Regeln, die allgemeiner sind als eine andere Regel, dabei relativ leicht entdeckt werden können.

Des Weiteren wird für jede Regel überpüft, ob ihre Prämisse in positiver Korrelation zu ihrer Konklusion steht. Dazu kann ein statistischer χ^2 Test durchgeführt werden. Dieser Test ist effizient, da die benötigten Klassenverteilungen im Vergleich zu häufigen Item-Sets bei der Regelsuche bereits ermittelt wurden.

In einem letzten Schritt, nachdem alle anderen Pruning Prozesse abgeschlossen sind, werden noch ähnlich zu dem CBA Algorithmus die Regeln entfernt, die entsprechend ihrer Priorität keine richtigen Entscheidungen auf dem Testdatensatz mehr treffen können. Für CMAR heißt das, dass der Reihe nach beginnend mit der Regel mit der höchsten Priorität für jede Regel die Beispiele gesucht werden, die von der Regel klassifiziert werden. Jedes mal wenn eine Regel ein Beispiel klassifizeren würde, wird ein Zähler für dieses Beispiel erhöht. Sobald der Zähler für ein Beispiel einen Grenzwert überschreitet, womit alle für dieses Beispiel relevanten Regeln betrachtet wurden, wird dieses Beispiel aus dem Testdatensatz entfernt. Wenn eine Regel kein Beispiel mehr richtig klassifiziert, wird diese Regel stattdessen aus der Liste gelöscht.

3.2.3 Klassifikation mit CMAR

Um die Klasse von einem neuen Beispiel zu bestimmen, werden zunächst alle Regeln, deren Prämissen auf dieses Beispiel zutreffen, gesucht und in Gruppen nach ihrer Klasse sortiert. Wenn es nur eine Klasse gibt, die vorhergesagt wird, dann ist die Klassifikation trivial und es wird genau diese eine Klasse vorhergesagt. In dem Fall, dass es mehrere Gruppen von Regeln gibt, die jeweils eine unterschiedliche Klasse vorhergesagt wird, dann müssen die einzelnen Gruppen von Regeln evaluiert werden, um eine möglichst aussagekräftige Gruppe für die Klassifikation zu benutzen. Da normale χ^2 Werte generell für kleinere Klassen höher sind, wurde nach einigen Experimenten für diesen Algorithmus zusätzlich eine $max\chi^2$ Heuristik als beste Option zur Normalisierung unterschiedlich großer Klassen benutzt. Der genaue Wert für eine Gruppe von Regeln, wobei eine Regel R definiert ist als $R : P \rightarrow c$, berechnet sich daher folgendermaßen:

$\sum \frac{\chi^2 * \chi^2}{max\chi^2}$

wobei $max\chi^2 = (minsup(P), sup(c) - \frac{sup(P)sup(c)}{|T|})^2|T|e$ und $e = \frac{1}{sup(P)sup(c)} + \frac{1}{sup(P)(|T|-sup(c))} + \frac{1}{(|T|-sup(P))sup(c)} + \frac{1}{(|T|-sup(P))(|T|-sup(c))}$ [3]

3.3 CPAR

Bei dem CPAR-Algorithmus[4] (Classification based on Predictive Association Rules) handelt es sich dabei um einen Algorithmus, der zwar nicht direkt Assoziationsregeln mittels Support und Konfidenz lernt, aber versucht, die Eigenschaft von Assoziationsregellernern zu imitieren. Dazu benutzt der Algorithmus nicht nur die derzeit beste Regel, sondern betrachtet auch weitere, ähnlich gute Regeln. Dadurch kann der Algorithmus das Problem von zu vielen generierten Assoziationsregeln mit einem greedy Ansatz lösen. Im Gegensatz zu den bisher vorgestellten Algorithmen wird bei CPAR also nicht versucht, alle existierenden Assoziationsregeln zu lernen, sondern lediglich eine für die Klassifikation ausreichende Menge an Regeln. Zu diesem Zweck wird ein Predictive Rule Mining (PRM) Ansatz, der eine Modifikation des FOIL Ansatzes ist, verwendet. Des Weiteren benutzt der Algorithmus, im Gegensatz zu CBA, jeweils die k besten Regeln, die auf ein Beispiel zutreffen, um es zu klassifizieren.

3.3.1 CPAR Regelsuche

Bei CPAR werden Regeln gelernt, indem nacheinander die jeweils vielversprechendsten Literale an die Regel angehängt werden. Welche Literale am vielversprechendsten sind, wird durch eine gain-Heuristik bestimmt. Zusätzlich zu dem vielversprechendsten Literal werden allerdings auch noch alle weiteren Literale, die eine sehr ähnlichen Performance haben, an die Regel angehängt und in einer Queue gespeichert, um danach betrachtet zu werden. Des Weiteren wird ein Beispiel, dass von einer Regel abgedeckt wird, nicht direkt aus dem Datensatz gelöscht. Stattdessen wird das Gewicht des Beispiels reduziert und das Beispiel nur gelöscht, wenn durch diese Reduktion das Gewicht unter einen Grenzwert fällt. Eine Regel ist dann fertig, wenn es keinen möglichen Literal gibt, der angehängt werden könnte und den minimalen Grenzwert für die Verbesserung der Regel erreicht.

Um diese Auswertung möglichst effizient zu gestalten, werden zu jeder Regel Informationen in einem sogenannten PN-Array gespeichert. Dieses Array speichert sowohl für die Regel selbst, als auch für jede potentielle Erweiterung um einen Literal, die Anzahl der positiven und die Anzahl der negativen Beispiele, die von der Regel erfasst werden. Diese Datenstruktur kann mit einem Lauf über die Trainingsbeispiele für eine beliebige Regel gefüllt werden. Durch diese Datenstruktur kann danach leicht herausgefunden werden, welche Literale als Erweiterungen der Regel vielversprechend sind. Außerdem kann die Datenstruktur vergleichsweise schnell aktualisiert werden, wenn eine Regel fertig gestellt wurde, und Beispiele aus dem Datensatz entfernt werden.

Der Pseudocode für den angepassten Predictive Rule Mining Algorithmus kann Algorithmus 9 entnommen werden. (vgl. auch Predictive Rule Mining in [4])

Algorithm 9 CPAR Rule generation

P/N mark the sets of all positive/negative examples, D is the set of all examples
set the weight of every example to 1
theory = $\emptyset$;
totalWeight = TotalWeight(positives);
Q = empty Queue;
A =CALCULATEPNARRAY(examples, emptyRule);
while TotalWeight(P)> $\delta * totalWeight$ **do**
    if queue $\neq \emptyset$ **then**
        extract Rule r from queue
        A' = CALCULATEPNARRAY(examples, r);
    else
        Rule r = emptyRule; A'=A;
    end if
    N'=negatives; P'=positives;A'=A;
    while true **do**
        find best literal p according to A'
        if gain(p)<min_gain **then**
            break;
        else
            for each literal p' with ($gain(p) - gain(p') < treshold$) and $gain(p') > min\_gain$ **do**
                r'= ADDCONDITION(r, p')
                q.PUSH(r')
            end for
        end if
        append p to r
        update A', P', N' according to examples not covered by r anymore.
    end while
    add r to Theory
    for each example e in positives satisfying r's body **do**
        t.weight = $\alpha *$ t.weight;
        update A according to the decreased weight;
    end for
end while

3.3.2 Klassifikation mittel CPAR

Da CPAR ein greedy Algorithmus ist, wurden bereits in dem vorherigen Schritt der Regelgeneration nur Regeln gelernt, die auf den verbleibenden Beispielen grundsätzlich nützlich sind. Daher ist es nicht, wie in anderen Algorithmen, nötig beim Bilden eines Klassifizierers erneut Regeln aus dem Modell zu filtern. Der Algorithmus kann also direkt alle bisher gelernten Regeln zur Klassifikation verwenden. (vgl. Algorithmus 10)

Die Klassifikation von Beispielen erfolgt vergleichsweise einfach auf Basis der nach Laplace Genauigkeit k besten Regeln jeder Klasse. Das heißt, es wird beim Bilden des Klassifizierers zunächst für alle Regeln die Laplace Genauigkeit berechnet. Die Regeln werden dann entsprechend ihrer Laplace Genauigkeit in eine Reihenfolge gebracht. Der genaue Pseudocode hierfür findet sich in Algorithmus 10.

Algorithm 10 CPAR classifier generation

```
1: k = classesCount(examples);
2: for each Rule r ∈ theory do
3:     c = correctClassified(r, examples);
4:     w = wrongClassified(r, examples);
5:     r.LaplaceAccuracy = (c + 1)/(c + w + k);
6: end for
7: theory = sortRules(theory);
```

Um nun ein Beispiel zu klassifizieren, werden zunächst alle Regeln, die das Beispiel abdecken, gesucht. Aus dieser Menge werden danach für jede Klasse die k besten Regeln ausgewählt und deren durschnittliche Laplace Genauigkeit berechnet. Das Beispiel bekommt dann die Klasse zugeordnet, bei der die durchschnittliche Laplace Genauigkeit am höchsten war.

3.4 RMR

Bei dem Ranked Multilabel Rule Algorithmus[5] handelt es sich, wie der Name des Algorithmus ahnen lässt, um einen Algorithmus, der mehrere Klassen zu einer einzelnen Assoziationsregel zuordnen kann und damit auch für sogenannte multi-label-Klassifikation benutzt werden kann. Multi-label-Klassifikation heißt, dass die einzelnen Beispiele nicht nur zu einer Klasse zugeordnet werden können, sondern häufig zu mehreren Klassen gehören. Ein Beispiel dafür ist ein Formel 1-Artikel, der sowohl zur Klasse „Sport“ als auch zur Klasse „Automobile“ zugeordnet werden kann. Im Gegenzug dazu erlaubt der Ansatz jedoch nicht, dass zwei Regeln des Klassifizierers sich in der Evaluation bei der Klassifikation von Trainingsbeispielen überschneiden. Durch diesen Schritt soll verhindert werden, dass die Performanz einer Regel davon abhängt, wie eine andere, besser bewertete Regel, aussieht.

3.4.1 Regelsuche in RMR

Der RMR Algorithmus benutzt ein neues Modell, das sehr ähnlich zu sogenannten transaction identifier lists (tid-list) ist, um möglichst effizient Assoziationsregeln zu finden. Eine tid-list enthält zu einem Item-Set jeweils eine Liste mit allen Trainingsbeispielen, in denen dieses Item-Set gefunden werden kann. Um häufige Item-Sets aus dieser Struktur zu gewinnen, muss man lediglich die Größe dieser Liste mit dem minimalen Support vergleichen.

Zusätzlich dazu kann man ein frequent Item-Set leicht erweitern, indem man den Schnitt zweier Listen bildet. Von diesem Schnitt kann man dann wiederum leicht den Support des neuen Item-Sets bilden. Um aus der Menge von häufigen k-großen Item-Sets, mit ihren zugehörigen tid-lists, die k+1-großen häufigen Item-Sets zu bilden, muss also für alle Item-Sets, die jeweils k-1 gemeinsame Elemente haben, der Schnitt ihrer tid-lists gebildet werden. Danach muss überprüft werden, ob die neue tid-list groß genug ist, um den minimalen Support zu erreichen.

Damit diese Struktur auch zur Klassifikation verwendet werden kann, muss lediglich noch zusätzlich der Klassenwert der Beispiele abgespeichert werden. Dieser Klassenwert wird dann benutzt, um alle möglichen klassenspezifischen Supports eines Item-Sets zu ermitteln, sofern das Item-Set, das die Prämisse der potentiellen Regeln bildet, den minimalen Support erreicht hat. Sollte es für ein Item-Set keine Klasse geben, mit der es eine Regel bildet, die den minimalen Support erreicht, dann kann das Item-Set direkt wieder gepruned werden. Um den klassenspezifischen Support zu ermitteln, muss lediglich die Anzahl der Elemente in der tid-list mit der entsprechenden Klasse gezählt werden und durch die Gesamtzahl an Trainingsbeispielen dividiert werden. Für die Konfidenz muss der selbe Wert durch die Größe der tid-list geteilt werden. Dieser Prozess hat den Vorteil, dass die tatsächlichen Trainingsbeispiele nur ein einziges mal überprüft werden müssen, um die einelementigen häufigen Item-Sets zu finden, und danach nur mit der schnelleren tid-list gearbeitet wird. Der Pseudocode hierfür kann in Algorithmus 11 gefunden werden.

Alle Regeln, die mit diesem Prozess gefunden werden, können dann eindeutig nach Priorität sortiert werden, wobei eine Regel r eine höhere Priorität als eine Regel s hat, wenn r

Algorithm 11 RMR rule generation

```
1: initialize empty tid-class-lists;
2: for each Example e ∈ examples do
3:     = updateTidClassList(example);
4: end for
5: add tid-class-lists above support treshhold to newCandidates;
6: add tid-class-lists above support and confidence treshhold to frequentItems.
7: while new tid-clas-lists can be found do
8:     calculate cuts for all tid-class-lists ∈ newCandidates with n-1 shared elements in the Set;
9:     calculate Support for new tid-lists;
10:    add tid-class-lists above support treshold to newCandidates and frequentItems;
11:    add tid-class-lists above support and confidence treshhold to frequentItems.
12: end while
13: return frequentItems;
```

- eine höhere Konfidenz hat.
- die gleiche Konfidenz, aber einen höheren Support hat.
- die gleiche Konfidenz, den gleichen Support und weniger Prämissen hat.
- die gleiche Konfidenz, den gleichen Support, die gleiche Anzahl an Prämissen und eine größere Klasse als Konklusion hat.
- die gleiche Konfidenz, den gleichen Support, die gleiche Anzahl an Prämissen und eine gleich große Klasse als Konklusion hat, aber früher generiert wurde.

Der Algorithmus benutzt wieder eine sehr simple Pruning Methode, bei der nacheinander Regeln in den Klassifizierer aufgenommen werden, und alle Ids von Beispielen, die von der Regel korrekt klassifiziert werden, für alle weiteren Regeln ignoriert werden. Wenn eine Regel dadurch keine Beispiele mehr klassifizieren kann, wird sie gepruned. Des Weiteren wird für die Regeln jeweils ihre vorhergesagte Klasse an die Menge an übrig gebliebenen Beispiel Ids angepasst. (vgl. Algorithmus 12) Da es bis zu diesem Schritt für jede Regel nur eine einzelne Klasse, die vorhergesagt wird, und keine zwei Regeln, die die gleichen Prämissen haben, gab handelt es sich nach diesem Schritt um eine Art singlelabel Klassifizierer. Die einzige Besonderheit in Bezug auf den endgültigen Multi-Label-Klassifizierer besteht darin, dass nur korrekt klassifizierte Beispiele aus der Trainingsmenge entfernt werden, da die anderen Beispiele noch durch ein zweites Label von der selben Regel klassifiziert werden könnten.

Algorithm 12 RMR Pruning

```
1: sort Rules;
2: tempIds = ∅;
3: theory = ∅;
4: for each Rule r in order of priority do
5:     if r then.coveredIds \ tempIds ≠ ∅
6:         calculate new majority class c of r on examples r.coveredIds \ tempIds;
7:         add r with new predicted class to theory
8:         tempIds = tempIds ∪ r.coveredIds[c];
9:     end if
10: end for
11: remainingExamples = examplesIds \ tempIds
```

Sobald alle Regeln einmal betrachtet wurden, also entweder gepruned wurden oder die abgedeckten Beispiele gesammelt wurden, werden die noch nicht abgedeckten Beispiele als neuer Datensatz betrachtet, auf dem ein weiteres Regelset gesucht wird, indem allerdings erneut die häufigen einelementigen Item-Sets aus dem ersten Schritt benutzt werden. Dieser Prozess wird solange wiederholt, bis keine häufigen Item-Sets mehr gefunden werden, die weitere Beispiele auf einem der neuen kleineren Datensätze abdecken. Durch diesen Prozess werden die angesprochenen zusätzlichen Klassenlabel für einen Teil der Regeln mit in den Klassifizierer aufgenommen.

Sobald keine weiteren häufigen Item-Sets zur Klassifikation mehr gefunden werden können, wird mit den restlichen Beispielen eine Defaultklasse gebildet.

3.4.2 Klassifizierer in RMR

Bei der Klassifikation mit RMR wird zunächst die Regel mit der höchsten Priorität, die das Beispiel abdeckt, gesucht. Sollte es eine solche Regel nicht geben, wird stattdessen die Regel mit höchster Priorität, bei der sich ein Attribut mit dem Beispiel deckt, gewählt. Erst wenn auch so eine Regel nicht existiert, wird die Default-Regel zur Klassifikation benutzt. Da es sich um einen Multilabel Klassifizierer handelt, werden dem Beispiel dann alle Klassen zugeordnet, die von der gewählten Regel vorhergesagt werden.

3.5 Vergleich der Algorithmen

Im Vergleich zu CBA hat CMAR eine vergleichbare Genauigkeit, die insgesamt dennoch etwas höher erscheint. Laut den Ergebnissen der Experimente von den CMAR Autoren gewinnt CBA in 8 von 26 Datensätzen, während CMAR auf 17 von 26 Datensätzen bessere Ergebnisse erzielt. Auf dem letzten Datensatz erreichen beide Datensätze die gleiche Genauigkeit. Wie bereits zu erwarten war, haben sich dagegen deutliche Unterschiede in der Performanz auf großen Datensätzen, sowohl was den benötigten Speicher angeht, als auch in der Zeit, die benötigt wird, den Klassifizierer zu bilden, gezeigt. Diese Ergebnisse decken sich mit den Erwartungen an die Algorithmen, da CMAR im Vergleich vor allem Verbesserungen an der Art, wie Regeln gefunden und gespeichert werden, vornimmt und lediglich zusätzliche Regeln in betracht zieht, um Beispiele zu klassifizieren, wobei die entsprechende Regel mit dem höchsten Gewicht in CMAR häufig die entscheidende Regel in CBA ist und CMAR daher sehr häufig zu denselben Klassifikationen als Ergebnis kommen sollte.

CPAR ist im Vergleich zu sowohl CBA als auch CMAR erneut deutlich schneller und benutzt deutlich weniger Beispiele, da es den greedy Ansatz zur Regelsuche benutzt. Was die Genauigkeit angeht ist CPAR trotz des greedy Ansatzes nur sehr knapp hinter CMAR. Für RMR konnte zwar kein direkter Vergleich mit CMAR oder CPAR gefunden werden, allerdings ist die Performanz von RMR im Vergleich zu CBA darauf schließen, dass der Algorithmus auch eine höhrere Genauigkeit als CPAR und CMAR erreicht. Unklar ist allerdings, auf welche Eigenschaft diese deutlich höhere Performanz zurückzuführen ist.

4 Implementierung von CBA im SeCo-Framework

In diesem Abschnitt werden die bereits existierenden Features des SeCo-Frameworks[2] kurz erläutert. Des Weiteren wird erklärt, welche neuen Features im Rahmen dieser Arbeit implementiert wurden.

4.1 SeCo-Framework

Das SeCo-Framework hat das Ziel, verschiedene Algorithmen aus der Familie der Separate-and-Conquer Algorithmen (im folgenden SeCo,) in einer gemeinsamen Umgebung zu implementieren, um es leicht möglich zu machen, diese untereinander zu vergleichen. Zu diesem Zweck wurde der Basis SeCo-Algorithmus (vgl. Algorithmus 2) mithilfe von leicht austauschbaren Modulen implementiert. Bei diesen Modulen handelt es sich hauptsächlich um

- heuristic - Die Heuristik, die zur Bewertung und Ordnung der Regeln verwendet werden soll.
- ruleinitializer - Ein Modul, das angibt, wie eine Regel initialisiert werden soll, bevor irgendwelche Verbesserungen an ihr durchgeführt wurden.
- rulerefiner - Ein Modul, das bestimmt, auf welche Art Verbesserungen an einer Regel ausgeführt werden sollen.
- rulefilter - Ein Filter um nur vielversprechende Regeln weiter zu betrachten.
- rulestoppingcriterion - Ein Kriterium, das bestimmt, wann aufgehört werden soll, weitere Regeln zu suchen.
- candidateselector - Ein Modul, das auswählt, welche Kandidatenregeln weiter verbessert werden sollen.
- stoppingcriterion - Ein Kriterium, das bestimmt, wann der Algorithmus aufhören soll, eine Regel weiter zu verbessern.
- postprocessor - Der Post-Prozessor wird benutzt, um eine vollständige Theorie noch weiter zu optimieren.

Zur Konfiguration der verschiedenen austauschbaren Module werden xml-Dateien verwendet, in denen für jedes Modul eine Implementierung ausgewählt werden kann. Des Weiteren gibt es je eine Default Implementierung, die verwendet wird, wenn keine andere Implementierung in der xml-Datei festgelgt wurde. Eine Beispiel xml-Datei kann in Algorithmus 13 gefunden werden. Das SeCo-Framework benutzt diese xml-Datei, um innerhalb einer Factory-Klasse auszuwählen, welche Implementierungen der einzelnen Komponenten benutzt werden sollen, sowie welche Parameter für die Initialisierung dieser Komponenten verwendet werden sollen. Aus diesen Komponenten wird dann von der Factory-Klasse ein konkreter Lernalgorithmus erstellt.

Um mit Hilfe von diesem Lernalgorithmus einen konkreten Klassifizierer zu bilden, wird dann die WeKa-Bibliothek[8] benutzt, die ein abstraktes Modell für Klassifizierer, sowie Methoden zur Verarbeitung von Trainings- und Beispieldaten, bereitstellt.

Algorithm 13 xml configurationfile for standard top-down classifier

```
 1: <seco growingSetSize="1" minNo="1" weighted="false" seed="0">
 2:     <heuristic classname="MEstimate"/>
 3:     <rulerefiner classname="TopDownRefiner"  nominalCompareMode="equal"/>
 4:     <stoppingcriterion classname="NoNegativesCoveredStop"/>
 5:     <rulestoppingcriterion classname="CoverageRuleStop"/>
 6:     <candidateselector classname="SelectAllCandidatesSelector"/>
 7:     <ruleinitializer classname="TopDownRuleInitializer"/>
 8:     <rulefilter classname="BeamWidthFilter" beamwidth="1"/>
 9:     <postprocessor classname="NoOpPostProcessor"/>
10: </seco>
```

Diese modulare Implementierung ermöglicht es, zwei verschiedene Algorithmen zu vergleichen und dabei das Ausmaß von verschiedenen Elementen, die die Algorithmen unterscheiden, zu bestimmen. Anschaulich könnte man zum Beispiel beim Vergleich von CN-2 mit dem oben erwähnten standard Top-Down-Klassifizierer zunächst testen, wie sich die Performanz des Standard-Klassifizierers ändert, wenn an Stelle von MEstimate Laplace als Heuristik benutzt wird. Danach kann man die Heuristik wieder zurücksetzen und nur den Einfluss von einer Likelihood Ratio als Stoppingcriteron oder eines Multirule Filters anstelle des Beamwidth Filters bestimmen. Nach dem selben Prinzip ist es natürlich auch möglich, den Einfluss von 2 Änderungen zusammen zu betrachten. Damit ist es prinzipiell möglich, nützliche Eigenschaften von Algorithmen deutlich leichter zu identifizieren und diese Eigenschaften direkt in andere Ansätze zu integrieren und testen.

4.2 Änderungen für den CBA Algorithmus

Die größte Änderung am SeCo-Framework, um den CBA-Algorithmus[1] zu unterstützen, ist, dass für den CBA-Algorithmus zunächst ein vereinfachter Apriori-Algorithmus implementiert werden musste (vgl. Algorithmus 4). Um weiterhin eine hohe Flexibilität in Form von Ersetzbarkeit der einzelnen Module des Frameworks zu garantieren, wurde der Apriori Algorithmus dabei als austauschbare Komponente implementiert, die in der xml-Konfigurationdatei festgelegt werden kann. Damit auch ältere bereits implementierte Algorithmen weiterhin eine gleichbleibende Performance erreichen können, ohne ihre Konfigurationsdateien zu verändern, wurde gleichzeitig ein NoAssociationRuleFinder implementiert und als Default-Konfiguration festgelegt. Die Änderungen, die dadurch am Separate and Conquer Algorithmus entstehen, können in Algorithmus 14 beobachtet werden. (vgl. auch mit Algorithmus 2)

Algorithm 14 modified Separate and Conquer

```
1: if associationRuleLearning then
2:     ruleList = associationRuleLearner(examples)
3:     ruleList.sort(heuristik)
4: end if
5: theory = ∅;
6: while positive(examples) ≠ ∅ do
7:     if associationRuleLearning then
8:         rule = pickBestRule(examples);
9:     else
10:        rule = findBestRule(examples);
11:    end if
12:    covered = cover(rule, examples);
13:    if ruleStoppingCriterion(theory, rule, examples) then
14:        exit while
15:    end if
16:    theory = theory ∪ rule ;
17:    examples = examples \ covered;
18: end while
19: theory = postProcess(theory);
20: return theory;
```

Eine weitere wichtige Änderung musste durchgeführt werden, da der CBA-Algorithmus nur einmal zu Beginn des Algorithmus Regeln, erstellt und danach lediglich aus diesem Set von Regeln, nach einem einfachen Covering Prinzip, Regeln auswählt, die für den Klassifizierer verwendet werden sollen. Dies ist ein offensichtlicher Unterschied zur FindBestRule-Methode, die im normalen SeCo Algorithmus verwendet wird. Daher wurde als Alternative zur findBestRule-Methode eine pickNextRule-Methode implementiert, die lediglich aus der gegebenen Liste von Regeln eine Regel auswählt, die für den nächten Schritt des Covering Prozesses benutzt werden soll. Zurzeit ist der Algorithmus von pickNextRule nicht als eigenes Modul implementiert und greift auf keine anderen Module zu. Er lässt sich daher auch nicht direkt mithilfe von einer Konfigurationsdatei verändern. Die pickNextRule Methode wird stattdessen immer benutzt, wenn in der Konfigurationsdatei festgelegt wurde, dass Assoziationsregeln gelernt werden sollen. Damit wird gleichzeitig auch sichergestellt, dass pickNextRule nur benutzt werden kann, wenn der Algorithmus zuvor auch eine Liste möglicher Regeln erstellt hat, von denen eine ausgewählt werden kann. Der Pseudocode für diese neue Methode pickNextRule ist in Algorithmus 15 zu sehen.

Algorithm 15 pickNextRule

```
1: while ruleList ≠ ∅ do
2:     rule = extract first Element from sortedRules
3:     if rule.casesCovered(examples) > 0 then
4:         return rule;
5:     end if
6: end while
7: rule = calculateDefaultRule(examples);
8: return rule;
```

Als letzte große Änderung am Framework wurden die neuen konfigurierbaren Komponenten in die xml Konfigurationsdateien eingebunden, sowie Default Werte festgelgt. Algorithmus 16 zeigt die daraus resultierende xml-Datei für den

CBA Algorithmus. Alle vorher existierenden xml-Dateien können durch die Wahl der Default-Werte weiterhin unverändert benutzt werden.

Algorithm 16 xml configurationfile for CBA

```
1:  <seco growingSetSize="1" minNo="1" weighted="false" seed="0">
2:     <heuristic classname="CombinedConfidenceSupport"/>
3:     <rulerefiner classname="TopDownRefiner"  nominalCompareMode="equal"/>
4:     <stoppingcriterion classname="NoNegativesCoveredStop"/>
5:     <rulestoppingcriterion classname="NoOpRuleStop"/>
6:     <candidateselector classname="SelectAllCandidatesSelector"/>
7:     <ruleinitializer classname="RandomRuleInitializer"/>
8:     <rulefilter classname="BeamWidthFilter" beamwidth="1"/>
9:     <postprocessor classname="PostProcessorCBA"/>
10:    <associationrulefinder classname="Apriori"/>
11: </seco>
```

Zusätzlich zu dem neuen Attribut associationrulefinder sind hierbei auch die Heuristik CombinedConfidenceSupport und der Post-Prozessor PostProcessorCBA interessant, da diese beiden Komponenten für den CBA Algorithmus erforderlich sind, ohne jedoch dass dadurch Änderungen am SeCo-Framework notwendig waren. Der Post-Prozessor wird hier für die letzten Schritte der Klassifizierer Generation aus der Basis Variante benötigt. Der Post-Prozessor sucht also die Regel im Klassifizierer, nach der die kommenden Regeln mehr Fehler machen als die Default Regel und entfernt danach diese Regeln.(Zeile 17-19 in Algorithmus 5)

Die Heuristik CombinedConfidenceSupport wird benutzt, um Regeln entsprechend ihrer Konfidenz und ihrem Support in eine Prioritätsreihenfolge zu bringen. Da Heuristiken im SeCo-Framework als Werte Heuristiken realisiert wurden, kann dies jedoch nicht direkt als Vergleich passieren, sondern die Heuristik muss jeder Regel einen Wert zuordnen können, mit dem alle Regeln entsprechend dieser Priorität sortiert werden können. Für diese Arbeit wurde das durch folgende simple Formel realisiert: HeuristicValue = confidence + $0.001 * \text{support}$.

Theoretisch ist diese Formel zwar nicht perfekt und könnte zwei Regeln mit sehr ähnlicher Konfidenz in eine falsche Reihenfolge bringen, allerdings ist das in der Praxis kein Problem, da es nur Regeln betrifft, die nahezu gleich gut sind und daher die Ordnung, auf alle Regeln betrachtet, nicht signifikant stört.

Zuletzt ist noch interessant, dass einige der anderen Attribute zwar angegeben werden können, allerdings keine Auswirkungen auf den Klassifizierer haben, da die FindBestRule Methode nicht mehr ausgeführt wird, wenn ein associationrulefinder gewählt wird. Konkret betrifft das die Module für ruleinitializer, rulefilter, candidateselector, rulerefiner und stoppingcriterion.

Tabelle 5: Vergleich verschiedener Support und Konfidenz Werte für den tic-tac-toe Datensatz

Support	Konfidenz	Precision	Simulationszeit[ms]
0.05	X	1	196
0.09	0.7	0.7393	77
0.1	0.1	0.7525	65
0.1	0.2	0.7395	66
0.1	0.3	0.7489	68
0.1	0.5	0.7302	67
0.1	0.7	0.7393	66
0.1	0.8	0.6843	50
0.2	0.2	0.7083	14
0.2	0.3	0.7102	14
0.2	0.5	0.7113	15
0.2	0.7	0.7269	17
0.3	0.3	0.7378	10
0.3	0.4	0.7350	10
0.3	0.5	0.6952	10
0.3	0.7	0.7219	10

5 Evaluation

In diesem Abschnitt werden einige Experimente mit der CBA[1]-Implementierung im SeCo-Framework[2] vorgestellt. Zunächst wurden einige verschiedene Werte für Support und Konfidenz des Algorithmus im Vergleich zueinander gesetzt und in Hinblick auf Genauigkeit und Simulationszeit, um den Klassifizierer zu bilden, untersucht. Für dieses Experiment wurde der Datensatz tic-tac-toe benutzt, der nur diskrete Attribute enthält und daher nicht diskretisiert werden musste. Die Genauigkeit wurde mithilfe von 10-fold cross validation ermittelt. Die Ergebnisse des Experiments können in Tabelle 5 gefunden werden.

Wie zu erwarten war, hat der verwendete Support einen sehr starken Einfluss auf die Zeit, die benötigt wird, um ein Simulationsmodell zu erstellen, da die Anzahl der potentiellen Kandidaten für Assoziationsregeln direkt mit dem minimalen Support zusammen hängt. Dies wird besonders deutlich, wenn man berücksichtigt, dass in den extrem Fällen mit einem sehr niedrigen bzw. sehr hohem minimalen Support jede bzw. keine Kombination von einem Attribut und einem Klassenwert als frequent Item-Set berücksichtigt werden muss. Die Genauigkeit des Klassifizieres scheint im Gegenzug bei geringerem minimalen Support größer zu sein, allerdings erhöht sich damit auch das Risiko, dass es zu overfitting kommt.

Zumindest in diesem Datenset lässt sich keine eindeutige Beziehung zwischen der Qualität des Klassifizieres und der minimalen Konfidenz erkennen. Bei einem minimalen Support von 0.1 scheint eine höhere minimale Konfidenz zu besseren Ergebnissen zu führen, während bei einem minimalen Support von 0.2 eine höhere Konfidenz eher zu schlechteren Ergebnissen zu führen scheint.

Dies lässt sich relativ leicht damit erklären, dass eine niedrigere minimale Konfidenz lediglich weitere Regeln bei der Regelgeneration zulässt. Da diese allerdings alle am Ende der Prioritätsliste angehängt werden, spielen sie nur selten eine bedeutende Rolle bei der Klassifikation und werden häufig beim Bilden des Klassifizierers einfach wieder gelöscht. Wenn diese Regeln nicht gelöscht werden, ist es weiterhin relativ wahrscheinlich, dass sie nur zu Overfitting führen, anstatt den Klassifizierer zu verbessern. Dadurch erklärt sich, dass die niedrigere Konfidenz teilweise zu schlechteren Ergebnissen führt, obwohl dadurch mehr Regeln in betracht gezogen werden. Bei größeren Datensätzen ist zu erwarten, dass eine niedrige Konfidenz hauptsächlich Einflüsse auf die Simulationszeit hat, während eine zu hohe Konfidenz stärkere und negative Einflüsse auf die Genauigkeit des Klassifizierers hat.

Der Einfluss der Konfidenz auf die Simulationszeit scheint relativ gering zu sein, sollte sich aber insgesamt ähnlich zum Support verhalten, da auch hier mehr potentielle Kandidaten länger berücksichtigt werden müssen, wenn die minimale Konfidenz niedriger ist. Eine niedrigere Konfidenz führt dabei zu mehr Regeln, die durch die Heuristik zur Regelbewertung am Ende der Decision List angehängt werden und immer ausgewertet werden müssen, da es keine abkürzende Abbruchbedingung gibt. Ein früheres Erreichen der Abbruchkriterium durch das Berücksichtigen von zusätzlichen Regeln ist hier also nicht möglich.

Tabelle 6: CBA Accuracy im Vergleich zu anderen Regellernern auf diskretisierten Datensets

Dataset	WeKa Jrip	SeCo CBA-1	Seco Top-Down	SeCo CBA-2
balance-scale	70.56	74.88	76.96	82.24
breast-w	94.13	87.69	94.70	92.13
cleveland-heart-disease	79.20	75.24	75.57	79.53
contact-lenses	75	66.66	70.83	66.66
credit	86.12	85.30	80.81	84.48
diabetes	72.78	70.05	68.61	73.56
glass	54.20	41.58	53.73	43.45
heart-c	80.19	76.56	73.92	79.20
horse-colic	84.78	83.15	75.27	81.52
iris	93.33	97.33	90.66	96
labor	84.21	82.45	85.96	80.70
monk 2	55.02	66.27	47.92	66.86
monk 3	86.88	83.60	85.24	85.24
primary-tumor	39.23	37.75	30.97	-
tic-tac-toe	97.80	100	97.28	100
titanic	78.32	78.10	78.32	74.01
vote	95.40	94.48	94.25	-
vowel	63.43	09.09	71.71	9.09
waveform-5000	73.22	64.74	-	62.82
waveform-10000	84.04	66.11	-	63.19
wine	83.14	86.51	83.14	-

5.1 Vergleich von CBA zu anderen Regellernern auf diskretisierten Datensets

Für den Vergleich von CBA mit anderen Algorithmen wurden alle Datensätze zunächst mit den Standard Optionen des Diskretisierungsfilters, der von WeKa[8] bereitgestellt wird, diskretisiert. Als Vergleichsalgorithmen wurden Jrip in der Weka eigenen Implementation, sowie ein simpler top-down Regel-Lern-Algorithmus aus SeCo, benutzt. Des Weiteren wurden zwei unterschiedliche Varianten von CBA getestet. Zunächst CBA-1 mit einem minimalen Support von 0.05 und minimaler Konfidenz von 0.6. Des Weiteren CBA-2 mit einem minimalen Support von 0.03 und minimaler Konfidenz von 0.7. Diese Werte wurden gewählt um eine relativ gute Performanz auf möglichst vielen Datensätzen zu gewähren. Einen niedrigeren Support als in CBA-2 zu wählen hätte dazu geführt, dass wegen Speicherlimitationen für weitere Datensätze kein Klassifizierer gebildet werden könnte. Für alle anderen Algorithmen wurden für alle Parameter die von Weka beziehungsweise SeCo vorgegebenen Default Werte benutzt. Ergebnisse der Experimente in Hinblick auf Accuracy können Tabelle 6 entnommen werden.

Zunächst fällt bei der Simulation auf, dass die Apriori Implementation deutlich mehr Heap-Speicher benötigt. CBA war in dieser Implementierung mit den gegebenen ca. 950MB Heap-Speicher nicht in der Lage, Modelle für alle Datensätze zu entwickeln. Datensätze, für die auch CBA-1 kein Modell liefern konnte, wurden nicht zusätzlich in der Tabelle aufgeführt, da keine weiterführenden sinnvollen Vergleiche durchgeführt werden können. Diese hohen Anforderungen an den Speicher sind auf die Assoziationsregelsuche zurück zu führen und können durch strengere Werte für Support und Konfidenz zwar gemindert werden, allerdings wird bereits am Beispiel der Support- und Konfidenzwerte für CBA-1 sichtbar, dass unter den strengeren Werten auch die Genauigkeit des CBA-Klassifizierers im Vergleich zu anderen Klassifizierern deutlich leidet.

Anhand von CBA-2 kann man deutlich erkennen, dass der CBA Algorithmus mit gut gewählten Parametern eine insgesamt ähnliche Genauigkeit erzielt wie der Jrip Algorithmus, der von den Vergleichsalgorithmen am besten abgeschnitten hat.

Eine deutliche Abweichung der Performance lässt sich jedoch bei dem vowel Datensatz erkennen, bei dem CBA lediglich eine Genauigkeit von 9% erreicht, während Jrip und die Standard Konfiguration von SeCo jeweils 60-70% erreichen. Dies lässt sich nach genauerer Betrachtung des Vowel Datensatzes leicht erklären, da der Datensatz aus 11 etwa gleich großen Klassen besteht und es daher schwierig, ist eine Regel zu finden, die den minimalen Support und die minimale Konfidenz erfüllen kann. Im Vergleich dazu hat die Regel mit der höchsten Priorität des Jrip Klassifizierers lediglich 26 Instanzen abgedeckt und erreicht damit nur einen Support von ca 0,026. Um diesen Datensatz korrekt zu klassifizieren, müsste daher ein niedrigerer Wert für den minimalen Support gewählt werden. Als zusätzlicher Test wurden deshalb auf diesem Datensatz zwei weitere CBA Klassifizierer mit einem minimalen Support von 0.01 und 0.006 als Parameter erstellt. Diese

Klassifizierer waren bereits in der Lage, 45% beziehungsweise 62% der Test Instanzen korrekt zu klassifzieren. Eine bessere Wahl der Parameter löst also das Problem für diesen Datensatz.

Des Weiteren ist der Waveform-10000 Datensatz interessant, da es auch hier eine deutliche Abweichung der Performance zwischen Jrip und CBA gibt. Die SeCo Top-Down Konfiguration ist dabei nicht in der Lage überhaupt einen Klassifizierer in angemessener Zeit zu erstellen und hatte auch nach 15 Minuten keinen einzigen Klassifizierer gebildet, während CBA Varianten nach 12 bzw. 22 Sekunden und Jrip nach 77 Sekunden jeweils einen Klassifizerer erstellen konnten. Bei genauerer Betrachtung des Datensatzes und des resultierenden Klassifizierers fällt auf, dass es sich mit 10000 Instanzen um einen vergleichsweise sehr großen Datensatz handelt. Der Jrip klassifizierer besteht hierbei aus 220 Regeln, die im einzelnen zwar nur wenige Fehler machen, aber häufig auch nur ca. 10-20 Instanzen abdecken und daher nur einem initialen Support von 0.001-0.002 entsprechen. Dies ist deutlich unter der gewählten Grenze für den CBA Algorithmus. Der CBA-2 Klassifizierer nutzt mit 63 Regeln auch nur etwas mehr als ein Viertel der von Jrip verwendeten Regeln, wobei die einzelnen Regeln auch innerhalb der Decision List deutlich mehr Beispiele abdecken als die Regeln des Jrip Algorithmus. Als zusätzliches Experiment wurde daher noch ein weiterer CBA Klassifizierer mit einem minimalen Support von 0.02 gebildet, der mit 233 Regeln eine zu dem Jrip Klassifizierer sehr ähnliche Anzahl an Regeln hat. Dieser Klassifizierer erreicht immerhin eine Genauigkeit von 76% und ist damit schon deutlich näher an der Performance des Jrip Klassifizierers. Im Gegenzug dauert allerdings auch das erstellen des neuen CBA Klassifizierers nun etwas länger als der Jrip Klassifizierer. Als letzten Ausreißer unter den untersuchten Datensätzen gibt es noch den glass Datensatz, allerdings kann man auch hier schnell erkennen, dass ein leichtes anpassen des Supports den CBA Klassifizierer auf ein ähnliches Niveau mit dem Jrip Klassifizierer bringt.

Weiterhin fällt bei genauerer Untersuchung der gebildeten Klassifizierer auf, dass die Regeln die am Ende ausgewählt werden, zu dem Zeitpunkt ihrer Auswahl auf den verbleibenden Instanzen teilweise eine sehr schlechte Performance zeigen. Dieses Verhalten lässt sich durch zwei Eigenschaften des Algorithmus erklären. Zunächst werden Regeln einmalig am Anfang bewertet und entsprechend dieser Bewertung sortiert. Das heißt, dass zwei Regeln, die größtenteils die gleichen positiven Beispiele und unterschiedliche negative Beispiele abdecken, beide eine gute Bewertung erhalten können, obwohl die Regel mit niedrigerer Priorität im endgültigen Klassifizierer einen großen Teil der Instanzen, die von der Regel richtig klassifiziert werden würden, nicht mehr sieht.

Überraschend ist zudem, dass der monk 2 Datensatz von CBA deutlich besser klassifizert werden konnte als von Jrip, obwohl der Datensatz von beiden Algorithmen sehr schnell klassifiziert wurde. Eine konkrete Erklärung konnte hierfür nicht direkt gefunden werden, daher ist die Annahme, dass es in dem Datensatz einige Assoziationen gibt, die von CBA gefunden werden, während Jrip sie allerdings nicht finden kann.

Des Weiteren werden die betroffenen Regeln dennoch im endgültigen Klassifizierer aufgenommen, da der Trennpunkt, welche Regeln aufgenommen werden sollen, durch die Performance aller Regeln bis zu dem potentiellen Trennpunkt bestimmt wird und eine einzelne schlechte Regel durch eine oder mehrere darauf folgende gute Regeln ausgeglichen werden kann. Abgesehen von dem bestimmen des Trennpunkts wird nur überprüft, ob eine Regel überhaupt noch ein verbleibendes Beispiel korrekt klassifiziert. Das heißt, ein einziges der verbleibenden Beispiele richtig zu klassifizeren reicht aus, damit die Regel im endgültigen Klassifizerer berücksichtigt wird, solange sich der Trennpunkt nicht vor der Regel befindet.

Theoretisch könnte man die betroffenen Regeln mithilfe einer erneuten genaueren Bewertung der Regel herausfiltern, allerdings würde dieser zusätzliche Schritt auch dafür sorgen, dass es insgesamt länger dauert, den Klassifizerer zu bilden. Des Weiteren wäre es unklar, ob der Klassifizierer dadurch überhaupt verbessert wird, da die darauffolgenden Regeln wieder die selben Fehler machen könnten. Außerdem sollte man berücksichtigen, dass die betroffenen Regeln auf den gesamten Datensatz betrachtet dennoch eine relativ gute Bewertung hatten, daher könnten sie weiterhin relevant für neue, bisher ungesehene Instanzen sein.

Insgesamt wird durch die zusätzlichen Experimente deutlich, dass durch anpassen der Parameter auch für ungewöhnliche Datensätze weiterhin eine vergleichbare Genauigkeit erreicht werden kann. Gleichzeitig wird aber auch ein Nachteil des CBA-Algorithmus deutlich, da es nicht möglich ist standard Parameter zu wählen, die auf allen möglichen Datensätzen zu zufriedenstellenden Ergebnissen, also guter Genauigkeit und schneller Konstruktion, führen. Der Jrip Klassifizierer zeigt sich als deutlich wiederstandsfähiger gegen ungewöhnliche Datensätze.

Für den Vergleich der Zeit, die die unterschiedlichen Algorithmen benötigen, um einen Klassifizierer zu bilden, können die Ergebnisse der Experimente Tabelle 7 entnommen werden. Es fällt zunächst auf, dass der Jrip Klassifizierer von WeKa deutlich schneller erstellt wird, als alle getesteten Varianten des SeCo-Frameworks. Die einzige Ausnahme hierbei ist der waveform-10000 Datensatz, der durch Jrip jedoch deutlich präziser klassifiziert wird. Wie bereits zuvor, beim Vergleich der Genauigkeit, gezeigt, benötigt ein zu dem Jrip vergleichbarer CBA Klassifizierer allerdings auch für diesen Datensatz mehr Zeit zur Konstruktion als der Jrip Klassifizierer. Im Vergleich von CBA zur Top-Down standard Konfiguration von SeCo gibt es keinen klaren Favoriten, da sich je nach Datensatz die Konstruktionszeiten für die beiden Klassifizierer deutlich unterscheiden. Insgesamt scheint der CBA-Algorithmus allerdings weniger starke Ausreißer zu haben.

Es lässt sich außerdem eine Tendenz erkennen, da die CBA-Klassifizierer vergleichsweise schneller auf Datensätzen, die viele Instanzen haben, konstruiert werden können, während der Top-Down Algorithmus effizienter Klassifizierer auf

Tabelle 7: CBA model building time im Vergleich zu anderen Regellernern auf diskretisierten Datensets in Sekunden

Dataset	WeKa Jrip	SeCo CBA-1	SeCo Top-Down	SeCo CBA-2
balance-scale	0.07	0.03	3.6	0.05
breast-w	0.06	0.37	1.93	0.51
cleveland-heart-disease	0.02	0.69	0.6	2.24
contact-lenses	<0.01	0.01	<0.01	0.01
credit	0.01	8.73	2.49	27.72
diabetes	0.04	0.2	11.69	0.17
glass	0.01	0.16	0.46	0.29
heart-c	0.02	0.64	0.6	2.03
horse-colic	0.02	1.27	2.42	5.91
iris	0.01	0.01	0.03	0.01
labor	0.01	0.41	0.01	4.08
monk 2	<0.01	0.03	0.07	0.04
monk 3	0.01	0.02	0.02	0.05
primary-tumor	0.05	72.44	1.14	-
tic-tac-toe	0.04	0.2	1.08	0.46
titanic	0.01	0.03	0.58	0.02
vote	0.01	57.6	0.18	-
vowel	0.23	0.04	24.39	0.07
waveform-5000	2.9	5.32	>600	10.03
waveform-10000	77.58	12.23	>600	21.58
wine	0.02	0.27	0.13	-

Datensätzen mit vielen Attributen erstellt. Dies wird vor allem bei den beiden Waveform Datensätzen, die deutlich mehr Instanzen als die restlichen Datensätze haben, deutlich. Top-Down Klassifizierer auf diesen Datensätzen zu bilden war dabei auch nach zehn Minuten nicht möglich. Eine mögliche Erklärung für diese Erscheinung besteht darin, dass der CBA Algorithmus hauptsächlich mit der Anzahl an Assoziationen, die es in dem Datensatz gibt, skaliert, und diese vor allem mit der Anzahl an Attributen ansteigt, während zusätzliche Instanzen nicht zwangsläufig auch zu weiteren Assoziationen im Datensatz führen und daher von CBA leichter zu verarbeiten sind.

Weiterhin sind die beiden Waveform Datensätze sehr interessant, da es sich bei Waveform-10000 um eine Erweiterung des Waveform-5000 Datensatzes mit exakt doppelt so vielen Instanzen handelt. Im direkten Vergleich fällt auf, dass CBA mit denselben Parametern den Datensatz mit mehr Instanzen nur minimal besser klassifiziert, während sich bei Jrip ein deutlicher Unterschied in der Qualität des Klassifizierers beobachten lässt. Dies lässt sich dadurch erklären, dass etwa dieselben Muster gefunden werden, da bei der doppelten Menge an Instanzen im Datenset auch die doppelte Menge an Instanzen benötigt wird, um den minimalen Support zu erreichen. Bei der Konstruktionszeit der Klassifizierer fällt dagegen auf, dass sich die Zeit, den Klassifizierer bei gleichen Parametern zu bilden, nur etwa verdoppelt hat, während sie bei Jrip um den Faktor 26 gestiegen ist. Dies legt nahe, dass der CBA-Algorithmus linear mit der Anzahl an Instanzen skaliert, solange dadurch nicht viele zusätzliche Assoziationen eingeführt werden.

5.2 Zusammenfassung

Insgesamt gesehen scheint diese Implementierung des CBA Algorithmus bei gut gewählten Parametern eine ähnliche Genauigkeit, wie andere bekannte Klassifizierer Algorithmen, zu erreichen. Hierbei ist insbesondere das Wählen eines guten Supports wichtig. In der derzeitigen Implementierung zeigt der Algorithmus allerdings einige Probleme mit Datensätzen, in denen es sehr viele mögliche Assoziationen gibt, was häufiger bei Datensätzen mit vielen Attributen der Fall ist. Dadurch ist der Algorithmus generell für viele Datensätze deutlich langsamer als Jrip und teilweise nicht in der Lage, überhaupt einen Klassifizierer für besonders schwierige Datensätze zu bilden. In diesen Fällen entstehen auch sehr große Anforderungen an den Speicher. Im Gegensatz dazu hat der Algorithmus in den Experimenten vergleichsweise sehr gut mit der Anzahl an Instanzen skaliert. Es lässt sich daher festhalten, dass der CBA Algorithmus vorallem auf Datensätzen mit hoher Anzahl an Trainingsinstanzen verwendet werden sollte. Für andere Arten von Datensätzen kann der Algorithmus zwar auch verwendet werden, allerdings gibt es mit Jrip andere Optionen, die deutlich zeiteffizienter erscheinen.

6 Ausblick

Da die Experimente gezeigt haben, dass die CBA[1]-Implementierung vergleichsweise langsam ist und viel Speicherplatz benötigt, ist es offensichtlich möglich, als nächstes einen weiteren Algorithmus, um Klassenassoziationsregeln zu finden, zu implementieren oder den existierenden Apriori Ansatz zu optimieren. Besonders auffällig ist hierbei die Möglichkeit den FP-Growth Algorithmus[9], der auch erfolgreich von CMAR[3] verwendet wird, zu implementieren.

Des Weiteren gibt es noch die Möglichkeit, mit bereits implementierten Modulen des SeCo-Frameworks[2] zu experimentieren. Unter anderem könnte die Auswirkung der gewählten Heuristik im Vergleich zu anderen Heuristiken sowohl für CBA als auch für andere Regel-Lern-Algorithmen überprüft werden. Weiterhin könnte es auch interessant sein, einen komplexeren Post-Prozessor anstelle des CBA Post-Prozessors zu benutzen oder ein rulestoppingcriterion zu verwenden.

Zuletzt gibt es noch die Option, die pickNextRule Methode weiter zu parametrisieren und zum Beispiel die Eigenschaft des RMR-Algorithmus[5], bei dem beim Bilden des Klassifizierers noch einmal verändert wird, welche Klasse von der Regel am besten vorhergesagt werden sollte, einzuführen.

Literatur

[1] Bing Liu Wynne Hsu Yiming Ma and Bing Liu. Integrating classification and association rule mining. In *Proceedings of the fourth international conference on knowledge discovery and data mining*, 1998.

[2] Frederik Janssen and Johannes Fürnkranz. The seco-framework for rule learning. Technical Report TUD-KE-2010-02, TU Darmstadt, Knowledge Engineering Group, 2010.

[3] Wenmin Li, Jiawei Han, and Jian Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 369–376. IEEE, 2001.

[4] Xiaoxin Yin and Jiawei Han. Cpar: Classification based on predictive association rules. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 331–335. SIAM, 2003.

[5] Fadi A Thabtah and Peter I Cowling. A greedy classification algorithm based on association rule. *Applied Soft Computing*, 7(3):1102–1111, 2007.

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

[7] Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

[8] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

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