Fachgebiet Knowledge Engineering

Titel der Diplomarbeit:
Separate and Conquer Framework und disjunktive Regeln

Betreuer:
Prof. Dr. Johannes Fürnkranz

von
Matthias Thiel
Matrikelnr.: 1099230
Sperberweg 14, 64291 Darmstadt
08.07.2005

Zusammenfassung

Diese Arbeit beginnt mit einer Einführung in das maschinelle Lernen. Es wird ein Lernproblem definiert und dargestellt, wie solche Probleme mittels **Separate and Conquer** (SeCo) Algorithmen gelöst werden. Der CN2 und der BEXA Algorithmus sind zwei Vertreter dieser Familie und werden hier genauer beschrieben und schließlich auf eine abstrakte Darstellung der SeCo-Algorithmen abgebildet. Als allgemein verwendbare Konzeption wird das **SeCo-Framework** vorgestellt, das sowohl der Implementation von SeCo-Algorithmen dient, als auch die automatische Erzeugung von Lernalgorithmen anhand einer XML-Beschreibung ermöglicht. Durch den modularen Aufbau ist es möglich, gezielt einzelne Aspekte des Regellernens isoliert zu betrachten, was in einer anschließenden Fallstudie zur Untersuchung der **Hypothesensprachen** des BEXA und des CN2 Algorithmus auch genutzt wird. Dabei wird analysiert in welchen Fällen die Verwendung von konjunktiven bzw. disjunktiven Regeln von Vorteil ist und zwar nicht nur hinsichtlich der Korrektheit, sondern auch unter Betrachtung der Qualität der gelernten Regelmengen. Zu diesem Zweck werden neue Meßkriterien eingeführt. Der Vergleich erfolgt unter Verwendung verschiedener Heuristiken und Grenzwerteinstellungen, um einen Eindruck zu bekommen, mit welchen Parametern die Hypothesensprachen die besten Erfolge erzielen. Schließlich erfolgt eine empirische Untersuchung einer neuen BEXA Variante, welche die Ausdrucksformen beider Hypothesensprachen ausnutzen soll.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quelltexte

1 Einleitung

1.1 Einführung in die Problemstellung

Mittlerweile gibt es eine Vielzahl von Separate and Conquer Algorithmen, die verschiedene Eigenschaften und Vorgehensweisen haben, aber die gleiche Grundstruktur besitzen. Zur Beurteilung der Algorithmen werden Vergleiche durchgeführt indem die gelernten Regelmengen gegenübergestellt und hinsichtlich der Korrektheit und Größe verglichen werden. In der Regel geht ein Algorithmus als Sieger[1] hervor und wird dem anderen vorgezogen. Meist gibt es eine oder mehrere wichtige Eigenschaften, die den speziellen Charakter des jeweiligen Algorithmus ausmachen, wie im Fall des BEXA, dessen spezielle Eigenschaft das Ausschließen von Attributwerten im Verfeinerungsprozeß ist.[2] Neben den Hauptmerkmalen gibt es oft noch weitere Merkmale, die zum Erfolg des Algorithmus beitragen. Ungewiß ist jedoch das Ausmaß, indem diese Merkmale dazu beitragen. Eine genaue Messung solcher Ausmaße wird meist schon dadurch erschwert, daß die Implementationen der Algorithmen völlig unterschiedlich sind, da deren Autoren verschieden sind. Um eine genauere Untersuchung beim Vergleich von Separate and Conquer Algorithmen zu ermöglichen, ist im Rahmen dieser Arbeit ein Framework entstanden, daß es erlaubt, die SeCo-Algorithmen modular zu implementieren, so daß eine gezielte Untersuchung einzelner Merkmale vereinfacht wird. Erreicht wird dies durch die Beschreibung der Algorithmen in XML, was eine Modifikation der Algorithmen ohne Veränderung des Quelltextes bzw. erneutem Kompilieren möglich macht.

In [16] wurde bereits der BEXA mit seiner innovativen Hypothesensprache (Ausschluß von Attributwerten) vorgestellt und mit der klassischen Variante[3] (im CN2-Algorithmus) verglichen. Das Ergebnis ist, daß der BEXA in den meisten Fällen besser ist. Die Fallstudie in dieser Arbeit zeigt jedoch, daß die Hypothesensprache des BEXA nicht das alleinige Merkmal ist, welches für die guten Ergebnisse verantwortlich ist, sondern daß die neue Hypothesensprache nur für eine bestimmte Art von Lernproblemen von Vorteil ist. Interessant dabei ist, daß der Vorteil für diese Fälle sehr groß sein kann.

1.2 Überblick

Diese Arbeit beginnt zunächst mit einer Einführung in das **maschinelle Regellernen** in Abschnitt 2, indem zunächst ein Lernproblem allgemein formuliert und anhand eines konkreten Beispiels veranschaulicht wird. Danach gibt es einen Überblick über die vorhandenen **Familien von Separate and Conquer Algorithmen** und deren Merkmale, in denen sie sich unterscheiden. Insbesondere der **Kernalgorithmus** in Abschnitt 2.5, der allgemein und halbformal beschrieben ist, soll das Grundprinzip dieser Algorithmen verdeutlichen.

In Abschnitt 3.3 wird dann der einfache **CN2-Algorithmus** vorgestellt und im Abschnitt 3.5 dann der **BEXA** Algorithmus, der im weiteren Verlauf dieser Arbeit untersucht wird. Dabei interessiert vor allem die gegensätzliche

[1] Hierbei gibt es nicht immer einen eindeutigen Sieger. Oft müssen auch die Fälle unterschieden werden, in welchen ein Algorithmus besser ist als der andere.

[2] Daher auch der Name *Basic EXclusion Algorithm*.

[3] Klassisch meint, daß die Hypothesensprache für die Attribute konkrete Werte in Form von Bedingungen fordert.

Modellierung der Hypothesensprache, die im Vergleich zum CN2 nicht den =-Komparator, sondern den ≠-Komparator verwendet. Dazu wird jedoch zunächst das **SeCo-Framework** in Abschnitt 4 vorgestellt, das die Grundlage für die Implementation des BEXA (siehe Abschnitt 5) ist. Dabei zeigt sich, daß das SeCo-Framework als allgemeine Konzeption zur Umsetzung und Untersuchung von SeCo-Algorithmen nützlich ist.

Die **empirischen Untersuchungen** beginnen in Abschnitt 6 mit dem Vergleich des BEXA mit einer unabhängigen Implementation des Ripper [5] aus der Weka-Bibliothek [17], um eine Einschätzung zu bekommen, wie gut der BEXA wirklich ist.

Die Untersuchung der **verschiedenen Hypothesensprachen** erfolgt dann in Abschnitt 7, wo auch die Modifikationen des BEXA vorgestellt werden. In Abschnitt 8 folgt dann schließlich eine Zusammenfassung und Interpretation der Ergebnisse, sowie die offenen Punkte, die im Rahmen dieser Arbeit nicht geklärt werden konnten.

Im Anhang befindet sich eine kurze Beschreibung der XML-Sprache, die im SeCo-Framework zur Konfiguration eines Algorithmus verwendet wird, sowie die Dokumentation der Java-Klassen aus denen das Framework besteht.

2 Separate and Conquer Algorithmen

Zur Erklärung, wie Separate and Conquer Algorithmen funktionieren und durch welche speziellen Eigenschaften sie sich auszeichnen, soll an dieser Stelle zunächst der Begriff Lernproblem definiert werden, sowie die hierfür notwendigen Begriffe. Anschließend wird dann die Funktionsweise von Separate and Conquer Algorithmen anhand seines abstrakten Pseudo-Codes.

2.1 Das Lernproblem

Was gelernt werden soll, sind Beschreibungen für *Lernbegriffe* (auch Klasse genannt) wie z.B. "Es wird morgen nicht regnen.". Um solche Lernbegriffe erlernen zu können, wird eine Menge von Beispielen (auch Instanzen genannt), die sogenannte *Trainingsmenge*, gegeben. Die Besonderheit der Trainingsmenge ist, daß die Information, ob der Begriff zutreffend ist, ebenfalls angegeben ist. Ein *Beispiel* setzt sich wiederum aus Attributen zusammen, denen ein konkreter Wert zugeordnet wird. Wir beginnen also mit folgender Definition:

Definition 2.1 (Attribut) *Das Attribut A besteht aus einem Namen, einem Typ und einer Domäne.*

- *Der Name eines Attributs kann beliebig gewählt werden, sollte jedoch eindeutig sein.*
- *Die Domäne $\mathcal{D}_A$ von A ist eine Menge, welche alle möglichen Werte beinhaltet, die das Attribut A annehmen kann.*
- *Die beiden wichtigsten Attributtypen, die in dieser Arbeit betrachtet werden, sind **nominal** und **linear** (auch als numerisch bezeichnet.). Vom nominalen Typ spricht man, wenn es zur Domäne $\mathcal{D}_A$ keine Ordnungsrelation gibt, während die Domäne von linearen Attributen eine Ordnungsrelation besitzt.*

Als Beispiele seien folgende Attribute aufgeführt:

Tabelle 2.1: Beispiele für Attribute

Name	**Typ**	**Domäne**
Aussicht	nominal	{sonnig,bewölkt,regen}
Herbst	nominal	{ja,nein}
Temperatur	linear	{15..35}

Die Menge von Attributen, die in einem Lernproblem definiert sind, bezeichnen wir als $\mathcal{A}$.

Definition 2.2 (Instanz (oder auch Beispiel)) *Eine Instanz I ist eine Menge von Wertzuweisungen für Attribute $A \in \mathcal{A}$ der Form $A = x$, wobei $x \in \mathcal{D}_A$. Dabei wird jedem Attribut A höchstens ein Wert zugewiesen. Bringt man die Attribute $\mathcal{A}$ in eine feste Reihenfolge $A_1, ..., A_n$, so kann man eine Instanz I auch als Kreuzprodukt $I \in \mathcal{D}_{A_1} \times ... \times \mathcal{D}_{A_n}$ darstellen, was einer Liste von Attributwerten entspricht.*

Definition 2.3 (Trainingsmenge) *Die Trainingsmenge $\mathcal{TS}$ ist eine gegebene*

Menge von Instanzen und deren Zuordnung zu einem Begriff: $\mathcal{TS} \in \mathcal{D}_{A_1} \times ... \times \mathcal{D}_{A_n} \times \mathcal{C}$*, wobei* $\mathcal{C}$ *die Menge der Lernbegriffe ist.*

Eine Trainingsmenge könnte also wie folgt aussehen, wobei die Klasse + den Lernbegriff "Es wird morgen nicht regnen." und die Klasse - den Lernbegriff "Es wird morgen regnen." bezeichnet:

Tabelle 2.2: Beispiel für Trainingsmenge

#	**Aussicht**	**Herbst**	**Temperatur**	**Klasse**
1	sonnig	ja	17	-
2	bewölkt	nein	18	-
3	regen	ja	16	-
4	sonnig	ja	22	-
5	sonnig	nein	29	-
6	bewölkt	nein	30	-
7	bewölkt	nein	35	-
8	regen	ja	23	-
9	regen	nein	27	-
10	sonnig	ja	28	+
11	bewölkt	nein	23	+
12	sonnig	nein	27	+
13	regen	nein	23	+

2.2 Charakteristik eines SeCo Algorithmus

Die Abkürzung **SeCo** steht für die Klasse von Separate-and-Conquer Algorithmen (auch bekannt als Covering Algorithmen), die sich dadurch charakterisieren, daß sie nach einer Regel (siehe Definition 2.4) suchen, die einen Teil von Trainingsbeispielen erklärt, die Menge der Beispiele um diese reduziert und rekursiv die übrigen Beispiele durch weitere Regeln abdeckt (siehe [6]). Dadurch soll sichergestellt werden, daß jedes Beispiel der Trainingsmenge durch mindestens eine Regel abgedeckt wird. Die abstrakte Form dieser Algorithmen ist in Abschnitt 2.5 dargestellt. Die Merkmale, in denen sich die SeCo-Algorithmen unterscheiden, lassen sich in drei Dimensionen einteilen:

Hypothesensprache Die Hypothesensprache begrenzt den Suchraum für Hypothesen, die der Algorithmus lernen kann, denn sie ist die Beschreibungssprache für den zu erlernenden Begriff. Die Hypothesensprache setzt sich aus Regeln zusammen, die in 2.4 definiert sind.

Definition 2.4 (Regel) *Eine Regel besteht aus einem Regelkörper und einem Regelkopf, wobei sich der Regelkörper aus der Konjunktion von* ***Bedingungen*** *(siehe Definition 2.5) zusammensetzt, während der Regelkopf eine Klassenzuweisung für das jeweilige Beispiel beinhaltet.*

Definition 2.5 (Bedingung oder Attributtest) *Ein Attributtest ist eine Bedingung, die fordert, daß ein Attribut* A *einen bestimmten Wert* $x \in \mathcal{D}_A$ *hat oder dessen Wert* x *auf einen Wertebereich* W *mit* $W \subseteq \mathcal{D}_A$ *beschränkt ist, also* $x \in W$ *gilt.*

Zur Formulierung von Bedingungen werden Komparatoren (Definition 2.6) verwendet.

Definition 2.6 (Komparator) *In dieser Arbeit werden die Komparatoren* $=$ *und* $\neq$ *für nominale Attribute und* $\leq$ *bzw.* $>$ *für lineare Attribute verwendet. Zusätzlich gibt es hier auch den Komparator* $\in$*, der den Attributwert mit allen Werten einer Menge* M *vergleicht. Die Bedingung gilt dann als erfüllt, wenn*

- *Fall* $=$*: der Attributwert dem angegebenen Wert gleicht.*
- *Fall* $\neq$*: der Attributwert und der angegebene Wert verschieden sind.*
- *Fall* $\leq$*: der Attributwert kleiner oder gleich dem angegebenen Wert ist.*
- *Fall* $>$*: der Attributwert größer als der angegebene Wert ist.*
- *Fall* $\in$*: der Attributwert in* M *enthalten ist.*

Als Beispiel für die Definitionen betrachte man folgende Regel[4]

$$Aussicht = sonnig \wedge Herbst = nein \rightarrow class = + \tag{2.1}$$

Hier ist $Aussicht = sonnig \wedge Herbst = nein$ der Regelkörper bestehend aus zwei Bedingungen mit jeweils dem Komparator $=$. Der Regelkopf enthält die Klassenzuweisung $class = +$, die dann vorgenommen wird, wenn die Bedingungen des Regelkörpers erfüllt sind.

An dieser Stelle soll noch kurz der Begriff der Extension definiert werden, welcher später noch gebraucht wird.

Definition 2.7 (Extension oder Umfang) *Die Extension* $X_{\mathcal{TS}}(cond)$ *beinhaltet alle Beispiele der Trainingsmenge* $\mathcal{TS}$ *für welche die Bedingung cond erfüllt ist. Dabei kann cond, sowohl eine einzelne sowie eine Menge von Bedingungen sein. Zusätzlich gibt es die Mengen* $X_P(cond)$ *und* $X_N(cond)$*, die wie folgt definiert sind:*

$$X_P(cond) := \{x \in X_{\mathcal{TS}} | x \textit{ ist ein positives Beispiel}\} \tag{2.2}$$

$$X_N(cond) := \{x \in X_{\mathcal{TS}} | x \textit{ ist ein negatives Beispiel}\} \tag{2.3}$$

Als Beispiel zum Lernproblem aus Tabelle 2.2 gilt

$$X_{\mathcal{TS}}(Aussicht = sonnig) = \{1, 4, 5, 10, 12\}, \tag{2.4}$$

$$X_P(Aussicht = sonnig) = \{10, 12\} \text{und} \tag{2.5}$$

$$X_N(Aussicht = sonnig) = \{1, 4, 5\}, \tag{2.6}$$

wenn $+$ die positive Klasse ist.

[4] Mit Hilfe von Tabelle 2.2 kann man erkennen, daß es sich hier um eine Regel handelt, die nicht in allen Fällen korrekt ist.

Suchverfahren Das Suchverfahren soll neue Hypothesen finden und setzt sich zusammen aus Suchalgorithmus, Suchstrategie und Suchheuristik. Die Hauptaufgabe des *Suchalgorithmus* ist das Auffinden neuer Hypothesen innerhalb des Suchraums. In der Regel beginnt man mit einer Ausgangshypothese[5] und verfeinert diese in jeder Iteration. Man erhält daraufhin sehr viele Kandidaten[6] unter denen man die besten auswählen muß, da man nicht alle berechnen kann. Diese Aufgabe übernimmt die *Suchheuristik*, die, meist mit statistischen Verfahren, versucht, den Kandidaten Bewertungen zuzuordnen, so daß die besten Kandidaten die höchste Bewertung erhalten. Die *Suchstrategie* gibt die Richtlinien vor, nach denen Kandidaten verfeinert werden (mehr dazu in Abschnitt 2.5).

Heuristik zur Vermeidung von Überbestimmtheit Mit Überbestimmtheit ist gemeint, daß sich die Hypothesen zu sehr an einzelnen Beispielen der Trainingsmenge orientieren und dadurch einerseits wenig interessant sind, da sie nur auf wenige Beispiele anwendbar sind und andererseits eine hohe Wahrscheinlichkeit besitzen, falsch zu sein, da die einzelnen Beispiele, auf der die Hypothese beruht, falsch sein können. Viele SeCo-Algorithmen verwenden daher eine Heuristik, die entweder das Erzeugen solcher Hypothesen von vornherein verhindern soll, oder im nachhinein solche Hypothesen aus der Theorie (siehe Definition 2.8) entfernen soll.

Definition 2.8 (Theorie) *Mit Theorie ist die Menge der Regeln gemeint, die als Ergebnis eines Lernverfahrens erzeugt wird.*

Im allgemeinen werden also einfache Regeln, die viele Beispiele abdecken, denen Regeln vorgezogen, die komplexer sind bzw. nur wenige Beispiele abdecken.

2.3 Familien von SeCo-Algorithmen

Das Ergebnis jedes SeCo-Algorithmus ist eine Beschreibung des zu erlernenden Konzepts. Die Konzeptbeschreibung kann dabei sehr unterschiedlich dargestellt werden und ist meist abhängig von der Hypothesensprache. Anhand der Darstellungsform von Konzeptbeschreibung lassen sich verschiedene Familien von SeCo-Algorithmen unterscheiden.

2.3.1 DNF

Eine weitverbreitete Form der Konzeptbeschreibung ist die **D**isjunktive **N**ormal**F**orm auch DNF genannt. Hierbei wird für jede Klasse eine Disjunktion von Konjunktionen erzeugt, die von den zugehörigen Beispielen erfüllt sein muß. Eine DNF setzt sich aus einzelnen Literalen $L_{i,j}$ zusammen und hat folgende Form:

$$(L_{1,1} \wedge L_{1,2} \wedge ... \wedge L_{1,j}) \vee ... \vee (L_{i,1} \wedge L_{i,2} \wedge ... \wedge L_{i,j}). \tag{2.7}$$

Als Literale werden in der Regel Attributwertvergleiche wie $Aussicht = sonnig$ eingesetzt. Bekannte Vertreter dieser Familie sind die AQ-Algorithmen (siehe [11]), sowie BEXA, der in Abschnitt 3.5 genauer beschrieben wird.

[5] Wie diese Ausgangshypothese bestimmt wird, ist von der verwendeten Suchstrategie abhängig.

[6] Mit Kandidat ist hier eine Regel gemeint, die evaluiert wird, um zu prüfen, ob sie für die weitere Betrachtung interessant ist.

2.3.2 KNF

KNF bedeutet **K**onjunktive **N**ormal**F**orm[7], was eine Konjunktion von Disjunktionen ist. Sie ähnelt damit nicht nur äußerlich der DNF, sondern läßt sich durch Negation auch aus der DNF herleiten. Wenn $\neg L_{i,j}$ die Negation von $L_{i,j}$ ist, dann sieht obiges DNF-Beispiel in KNF wie folgt aus

$$\neg\Big((\neg L_{1,1} \vee \neg L_{1,2} \vee ... \vee \neg L_{1,j}) \wedge ... \wedge (\neg L_{i,1} \vee \neg L_{i,2} \vee ... \vee \neg L_{i,j})\Big). \quad (2.8)$$

In [15] wird ein Algorithmus für diese Form beschrieben.

2.3.3 Entscheidungslisten

Entscheidungslisten bestehen aus Einträgen, die jeweils eine Bedingung und eine Klasse beinhalten. Jeder Eintrag formuliert also eine Regel wie

$$L_1 \wedge L_2 \wedge L_3 \rightarrow class(I) = c \quad (2.9)$$

mit I als Instanz und $c \in \mathcal{C}$ (die Menge der Begriffe/Klassen). Möchte man nun prüfen, zu welcher Klasse ein Beispiel I gehört, dann werden die Regeln der Liste der Reihenfolge nach auf I angewendet bis die erste Regel trifft bzw. deren Bedingung erfüllt ist. Diese Regel bestimmt dann die Klasse von I, die restlichen Regeln bleiben unberücksichtigt.

Normalerweise wird bei diesem Verfahren eine Defaultregel, die keine Bedingung beinhaltet, als letztes in die Liste eingefügt wird. Sie soll zutreffen, wenn keine andere Regel zutrifft, um in jedem Fall eine Klasse zuzuordnen. Als Klasse wählt man dann diejenige die in den Trainingsmengen am häufigsten auftaucht.

Grundsätzlich kann man Entscheidungslisten auch in DNF formulieren und umgekehrt, jedoch ist zu beachten, daß man bei Konvertierung von Entscheidungslisten zu DNF nicht einfach alle Regeln für eine Klasse mit einer $\vee$-Verknüpfung zusammenführen kann, da jede Regel für sich keine allgemeingültige Aussage darstellt. Sie basiert nämlich auf der Annahme, daß die vorhergehenden Regeln (innerhalb der Liste) nicht zutreffend sind. Die Regeln gelten also nur für einen bestimmten Fall, der durch die vorhergehenden Regeln bestimmt wird. Dadurch, daß dieser Fall nicht mehr explizit in der jeweiligen Regel formuliert ist, sind die Regeln oftmals kürzer. Sogar die Anzahl der Regeln verringert sich. Dieser Sachverhalt muß beachtet werden, wenn man die Komplexität von Konzeptbeschreibungen in DNF mit solchen vergleicht, die in Entscheidungslisten formuliert sind.
Zu den populärsten Algorithmen, die mit Entscheidungslisten arbeiten, gehört der CN2, der in Abschnitt 3.3 genauer beschrieben wird. Weitere Konzeptbeschreibungen sind **Logische Programme**, wie sie beispielsweise mit FOIL (siehe [13]) erzeugt werden und **Funktionale Relationen**, siehe [14].

2.4 Vergleich von SeCo-Algorithmen

Um verschiedene SeCo-Varianten miteinander vergleichen zu können, werden die Varianten auf die selben Lernprobleme angewendet und die Ergebnisse betrachtet. Dabei sind folgende Aspekte von Bedeutung:

[7] Im englischen bekannt als CNF für Conjunctive Normalized Form.

2.4.1 Korrektheit

Die Korrektheit kann sich sowohl auf die Trainings- sowie auf die Testmenge beziehen und sagt aus, wieviel Prozent der Beispiele, anhand der gelernten Theorie, korrekt klassifiert worden sind. Im Fall der Korrektheit bzgl. der Trainingsmenge wird die gelernte Theorie bzw. der Klassifizierer auf die Trainingsmenge angewendet, mit welcher die Regeln gelernt worden sind. Hier ist o.B.d.A. ein hoher Prozentsatz von etwa 100% zu erwarten, weshalb dieser Wert meist wenig interessant ist. Im zweiten Fall werden die Regeln auf die Testmenge angewendet, deren Beispiele während des Lernverfahrens nicht bekannt sind. Um dies zu erreichen wird eine *10-fold stratified cross validation* eingesetzt [10]. Dabei wird die gegebene Beispielmenge bereits vor Anwendung des Lernalgorithmus in 10 gleich große Mengen aufgeteilt. Anschließend wendet man den Algorithmus auf neun der zehn Mengen an, um eine Theorie zu lernen. Die zehnte Menge wird dann als Testmenge verwendet, um die Regeln zu überprüfen. Dieser Vorgang wird so oft wiederholt bis jede dieser 10 Mengen einmal als Testmenge verwendet worden ist. Von diesen 10 Messungen wird dann das arithmetische Mittel der Korrektheitswerte gebildet. Die Korrektheit gibt also eine Schätzung für die Wahrscheinlichkeit an, mit der die Prognosen des Klassifizierers, auf bisher unbekannten Beispielen, richtig sind.

2.4.2 Größe der Regelmenge

Ein weiteres Qualitätsmerkmal eines SeCo-Algorithmus ist die Anzahl der Regeln, die er erlernen muß, um alle Beispiele der Trainingsmenge abzudecken. Obwohl eine große Regelmenge nicht schädlich für die Korrekheit sein muß, so sind kleine Regelmengen zu bevorzugen, da sie verständlicher sind und weniger an Überbestimmtheit leiden. Hierbei ist natürlich nicht nur die Anzahl der Regeln, sondern auch die Anzahl ihrer Bedingungen zu betrachten, da es sehr wohl möglich ist, eine Regelmenge so umzuformen, daß mehrere Regeln zu einer zusammengefaßt werden können und umgekehrt. Desweiteren ist darauf zu achten, inwiefern die Hypothesensprachen vergleichbar sind. Eine mächtigere Hypothesensprache benötigt oftmals weniger Regeln, um die selben Hypothesen darzustellen, wie eine weniger mächtige Hypothesensprache.

2.4.3 Komplexität

Die Komplexität gibt das Laufzeitverhalten des Algorithmus an. Je komplexer ein Algorithmus ist, desto stärker steigt der Bedarf an Rechenzeit an, wenn die Datenmengen größer werden. Dieser Aspekt ist insbesondere bei der Optimierung anderer Gütekriterien interessant, da eine Verbesserung der Algorithmen oftmals zu Lasten der Komplexität geht.

2.5 Der Kernalgorithmus

In [6] wurden die Gemeinsamkeiten von Separate-and-Conquer Algorithmen zusammengefaßt und eine Grundstruktur des Algorithmus formuliert. Diese Grundstruktur bezeichnen wir hier als Kernalgorithmus, der sich in allen SeCo-Algorithmen wiederfinden läßt.

Wie in Quelltext 2.1 zu erkennen ist, gibt es zunächst zwei Prozeduren, *SeparateAndConquer* und *FindBestRule*. Erstere beinhaltet eine Schleife, welche

Quelltext 2.1 Allgemeiner SeCo-Algorithmus

```
procedure SeparateAndConquer(examples)
{
   theory := ∅
   while positive(examples)≠∅
   {
      rule := FindBestRule(examples)
      covered := cover(rule, examples)
      if RuleStoppingCriterion(theory, rule, examples)
         exit while
      examples := examples \ covered
      theory := theory ∪ rule
   }
   theory := PostProcess(theory)
   return (theory)
}

procedure FindBestRule(examples)
{
   initRule := InitializeRule(examples)
   initVal := EvaluateRule(initRule)
   bestRule := <initVal, initRule>
   rules := {bestRule}
   while rules≠∅
   {
      candidates := SelectCandidates(rules, examples)
      rules := rules \ candidates
      for candidate∈candidates
      {
         refinements := RefineRule(candidate, examples)
         for refinement∈refinements
         {
            evaluation := EvaluateRule(refinement, examples)
            while ( !StoppingCriterion(refinement, evaluation,
                                       examples) )
            {
               newRule := <evaluation, refinement>
               rules := InsertSort(newRule, rules)
               if ( newRule > bestRule )
                  bestRule := newRule
            }
         }
      }
      rules := FilterRules(rules, examples)
   }
   return (bestRule)
}
```

die Theorie bzgl. einer zu lernenden Klasse zu finden versucht. Beginnend mit einer leeren Theorie, wird wiederholt eine "beste Regel" ermittelt. Solange das *RuleStoppingCriterion* nicht zutrifft, wird diese Regel der Theorie hinzugefügt und die abgedeckten Beispiele werden entfernt[8]. Desweiteren wird die "beste Regel" in die Theorie aufgenommen.

Die **FindBestRule** Prozedur beginnt zunächst mit einer initialen Regel, die gleichzeitig als beste Regel markiert wird und in die Regelmenge *Rules* aufgenommen wird. Solange in *Rules* noch Regeln enthalten sind, werden aus dieser Menge Kandidatenregeln ausgewählt und in eine eigene Menge der Kandidaten (*Candidates*) verschoben. Für jede dieser Kandidatenregeln wird eine Verfeinerung (*RefineRule*) durchgeführt. Die Varianten von Verfeinerungsprozessen kann man anhand der **Suchstrategie** unterteilen:

- **Top-Down-Strategie** Ausgehend von einer allgemeineren Regel, möchte man hier eine speziellere Regel erzeugen. Da die Spezialisierung nur eine Teilmenge der Beispiele abdecken kann, die von der Ausgangsregel abgedeckt worden sind, versucht man, die zusätzliche(n) Bedingung(en) so zu wählen, daß möglichst viele negativ-Beispiele ausgeschlossen bzw. nicht mehr von der verfeinerten Regel abgedeckt werden und gleichzeitig, möglichst viele positiv-Beispiele weiterhin abzudecken.

- **Bottom-Up-Strategie** Diese beginnt mit einer speziellen Regel und verallgemeinert diese durch Entfernen von Bedingungen. Bei diesem Schritt möchte man möglichst viele weitere positiv-Beispiele abdecken und nur wenige bzw. gar keine negativ-Beispiele.

Die verfeinerten Regeln werden anhand der Evaluierungsfunktion bewertet und anschließend mit dem *StoppingCriterion* überprüft, um festzustellen, ob eine weitere Verfeinerung dieser Regel "Sinn"[9] macht.

Jede Verfeinerung, auf die das StoppingCriterion nicht zutrifft, wird in *Rules* aufgenommen. Wenn sie besser als die bisher beste Regel ist, wird sie als beste Regel markiert. Die beste Regel wird dann als Ergebnis zurückgegeben.

2.6 Probleme beim Vergleich der Algorithmen

Es wurden bereits viele Varianten von SeCo-Algorithmen entwickelt und verbessert. Immer wieder werden diese miteinander verglichen, um Vor- und Nachteile gegenüber zu stellen, aber auch, um den Effekt von neuen Ideen meßbar zu machen. Obwohl den Algorithmen der SeCo-Familie die gleiche Konzeption zu Grunde liegt und die Art und Weise, wie die Versuche durchgeführt werden, übereinstimmt, so besteht oftmals dennoch das Problem, die Unterschiede der Meßergebnisse auf ihre eigentlichen Ursachen zurückzuführen. Dieses Zuordnungsproblem entsteht dadurch, daß sich die Implementationen der zu vergleichenden Varianten nicht nur in dem interessierenden Punkt unterscheiden,

[8] Abhängig vom konkreten Algorithmus betrachtet man, beim Entfernen abgdeckter Regeln, entweder *alle* oder nur die *positiven* Regeln.

[9] Wie das StoppingCriterion zu wählen ist, soll an dieser Stelle nicht vertieft werden. Es gibt verschiedene Möglichkeiten, wobei die einfachste die ist, kein Kriterium zu definieren. Dadurch kann man sicher stellen, daß man das Auffinden guter Regeln durch dieses Kriterium nicht verhindert. Es geht jedoch zu Lasten der Performanz, weil mehr Verfeinerungen generiert werden.

den man untersuchen möchte, sondern die Implementationen in der Regel völlig unterschiedlich ist. Im Gegensatz dazu könnte man sich vorstellen, daß insbesondere der Kernalgorithmus, sowie Heuristiken und Hypothesensprache in einer gemeinsam verwendeten Bibliothek einmal implementiert sind und von allen zu vergleichenden SeCo-Algorithmen eingebunden werden. Dieser Ansatz wird mit dem SeCo-Framework in Abschnitt 4 verfolgt.

Mit Hilfe des SeCo-Framework wird dann der BEXA Algorithmus untersucht. Zuvor soll jedoch im Abschnitt 3.3 die Funktionsweise des CN2 Algorithmus vorgestellt werden, welcher sich aufgrund seiner Bekanntheit und Einfachheit als Referenzalgorithmus zum Vergleich eignet.

3 Instanziierung eines SeCo

Im Abschnitt 2.5 wurde die Klasse der Separate and Conquer Algorithmen allgemein beschrieben. In diesem Abschnitt soll die konkrete Erzeugung eines solchen Algorithmus erklärt werden, anhand der existierenden Algorithmen CN2 und BEXA. Der CN2-Algorithmus (siehe [4] und [3]) ist einer der bekanntesten Vertreter der Separate and Conquer Algorithmen, der gerne zum Vergleich mit anderen Varianten herangezogen wird. Er soll hier als einführendes Beispiel eines einfachen SeCo-Algorithmus vorgestellt werden. Desweiteren dient seine Hypothesensprache zum Vergleich mit der des BEXA.

3.1 Was zu definieren ist

Bereits gegeben ist der Kern des Algorithmus in Quelltext 2.1, allerdings wurden dort einige Funktionen nicht definiert. Diese sind in Tabelle 3.1 zu sehen.

Tabelle 3.1: Variable Funktionen eines SeCo-Algorithmus

Funktion	Beschreibung
RuleStoppingCriterion	Kriterium bestimmt, wann Lernprozeß endet.
PostProcess	Nachbearbeitung der gelernten Regelmenge.
InitializeRule	Legt Regel fest, mit der Verfeinerung begonnen wird.
EvaluateRule	Bewertet Regeln mit Hilfe einer Heuristik.
SelectCandidates	Wählt Kandidatenregeln aus, die verfeinert werden sollen.
RefineRule	Erzeugt alle Verfeinerungen für eine Kandidatenregel.
StoppingCriterion	Prüft, ob eine Kandidatenregel weiter verfeinert werden soll.
FilterRules	Filtert Kandidatenregeln nach einer Iteration der Verfeinerung.

Die Funktionen legen die Charakteristik eines SeCo-Algorithmus fest, wie sie allgemein in Abschnitt 2.2 beschrieben wurde. Um also eine Definition vornehmen zu können, muß man sich Überlegen, wie die Hypothesensprache, das Suchverfahren und die Heuristik aussehen soll.

3.2 Allgemeines zu den Hypothesensprachen

Bevor die Instanziierung der Algorithmen CN2 und BEXA erläutert wird, sollen an dieser Stelle zunächst ein paar allgemeine Definitionen erfolgen. Die im folgenden verwendeten Hypothesensprachen werden hier in der VL-Notation[10] notiert.

Definition 3.1 (Subdomäne $\mathcal{S}_A$) *Die Menge $\mathcal{S}_A$ ist eine Subdomäne von $\mathcal{D}_A$, wenn gilt $\mathcal{S}_A \subseteq \mathcal{D}_A$. Mit $\mathcal{S}_A(r)$ bezeichnen wir alle Attributwerte des Attributs A, die im Regelkörper der Regel r vorkommen. Und*
$\mathcal{S}_A(r, \square) := \{x \in \mathcal{S}_A(r) | A$ und x werden mit Komparator $\square$ verglichen$\}$.

[10] VL bezeichnet die "multiple-valued extension of propositional logic" von Michalski, siehe [12].

Definition 3.2 (Inverse Subdomäne $\overline{\mathcal{S}_A}$) *Die inverse Subdomäne $\overline{\mathcal{S}_A}$ von $\mathcal{S}_A$ ist definiert als $\overline{\mathcal{S}_A} := \{x \in \mathcal{D}_A | x \notin \mathcal{S}_A\}$.*

3.3 Eigenschaften des CN2

Zunächst werden hier die speziellen Eigenschaften des CN2 beschrieben und anschließend gezeigt, wie man diese Eigenschaften auf die allgemeine SeCo-Beschreibung aus Abschnitt 2.5 abbilden kann. Wir unterscheiden, wie bereits in Abschnitt 2.2 die drei Dimensionen: die Hypothesensprache, das Suchverfahren und die Heuristik zur Vermeidung der Überbestimmtheit.

3.3.1 Hypothesensprache

Wie bereits im Abschnitt 2.3.3 erwähnt wurde, produziert der CN2 Entscheidungslisten. Bei den L_i handelt es sich hier um Attributtests wie etwa $Aussicht = bewoelkt$ oder $Temperatur > 60$. Zur Klassifikation einer Instanz I werden die Regeln der Reihe nach angewandt. Für das jeweilige I bestimmt die erste zutreffende Regel die Klasse von I. Eine Entscheidungsliste könnte dann so aussehen

$$(Aussicht = bewoelkt \wedge Temperatur > 20) \rightarrow class(I) = + \tag{3.1}$$
$$(Aussicht = sonnig \wedge Temperatur < 15) \rightarrow class(I) = - \tag{3.2}$$
$$\rightarrow class(I) = d. \tag{3.3}$$

Man beachte die Defaultregel am Ende, die bedingungslos jedem Beispiel die Klasse d zuordnet. Die Klasse d ist diejenige, die in der Trainingsmenge am häufigsten vorkommt.

Zur Bestimmung von Attributtests wird der Attributtyp unterschieden:

Nominale Attribute Bei einem nominalen Attribut A wird ein Attributtest als Bedingung $A = x$ formuliert, welche einen bestimmten Attributwert x fordert (mit $x \in \mathcal{D}_A$). Damit ist die Anzahl der Attributtests die man für ein Attribut A erzeugen kann auf die Anzahl der möglichen Attributwerte $|\mathcal{D}_A|$ beschränkt.

Lineare Attribute Bei einem linearen Attribut B ist die Anzahl möglicher Attributwerte unendlich ($|\mathcal{D}_B| = \infty$). Man wird also o.B.d.A keine allgemeingültigen Regeln finden können, wenn man die Attributwerte von B auf Gleichheit prüft (im Sinne einer Fallunterscheidung). Darum unterteilt man den Wertebereich $\mathcal{D}_B$ in Intervalle und prüft, ob ein Attributwert $x \in \mathcal{D}_B$ innerhalb des Intervalls liegt. Der einzelne Attributtest prüft jedoch nur eine einzelne Schranke wie z.B. $temp > 20$ für alle Temperaturwerte, die über 20 liegen. Man verwendet hier also die Ordnungsrelation $>$ bzw. $\leq$, um einen Vergleich durchzuführen und die Bedingung zu formulieren. Möchte man für ein Attribut eine obere und untere Schranke fordern, dann sind zwei Attributtests erforderlich.

3.3.2 Suchverfahren

Das Suchverfahren setzt sich aus der Suchstrategie, der Suchheuristik und dem Suchalgorithmus zusammen. Die Verfeinerung der Regeln erfolgt beim CN2 mit einer *Top-Down-Strategie* (siehe Abschnitt 2.5), die ausgehend von einer

DefaultRegel der Form $true \rightarrow class = -$ Spezialisierungen erzeugt durch Hinzufügen von weiteren Bedingungen im Regelkörper. Die Verfeinerung einer Hypothese über zwei Iterationen könnte dann so aussehen:

$$true \rightarrow class = - \tag{3.4}$$

$$(Aussicht = bewoelkt) \rightarrow class = - \tag{3.5}$$

$$(Aussicht = bewoelkt \wedge Temperatur \leq 15) \rightarrow class = - \tag{3.6}$$

Um Bedingungen hinzufügen zu können, muß vorher bestimmt werden, welche Bedingungen dafür überhaupt in Frage kommen. Die Grundidee ist dabei die, alle möglichen Attributtests $\mathcal{AT}$ zu ermitteln und für jede Regel, die verfeinert werden soll, alle Kombinationen zu berechnen, die durch Hinzufügen eines Attributtests entstehen.

Sei R_i die Menge der Kandidatenregeln, die in Iteration i verfeinert werden sollen und $\mathcal{AT}$ die Menge der möglichen Attributtests. Dann erzeugt man in einer Iteration i maximal $|\mathcal{AT}| * |R_i|$ Regeln. Sei R'_i die Menge der erzeugten Regeln in Iteration i, dann gilt also

$$|R'_i| \leq |\mathcal{AT}| * |R_i|. \tag{3.7}$$

In der Regel werden weniger Regeln erzeugt, weil man an dieser sehr rechenaufwändigen Stelle den Aufwand senken möchte. Als erstes bietet es sich an, solche Regelerzeugungen zu verhindern, deren Bedingungen nicht erfüllt werden können wie z.B.

$$(Aussicht = bewoelkt \wedge Aussicht = sonnig) \rightarrow class = -. \tag{3.8}$$

Im nächsten Schritt reduziert man R'_i, indem man mittels einer Suchheuristik die Regeln bewertet und schlechte Regeln verwirft.

Der CN2[11] verwendet zur Bewertung der Regeln die *LaplacePrecision* Heuristik.

Definition 3.3 (LaplacePrecision) $LPA(r) = \frac{n_c+1}{n_{tot}+k}$, *mit*

- r *ist die zu bewertende Regel.*
- n_c *ist die Anzahl der positiven (bzw. der Klasse c zugehörigen) Beispiele, die von der Regel abgedeckt werden.*
- k *ist die Anzahl der Klassen.*
- n_{tot} *ist die Summe der Beispiele, die durch r abgedeckt werden.*

Diese Bewertungsfunktion ordnet jeder Regel r einen Wert $LPA(r)$ zu, der es ermöglicht, die Regeln nach ihrer Qualität zu ordnen.

[11] Es gibt vom CN2 eine ursprüngliche Form, die in [4] beschrieben ist und in [3] verbessert worden ist. Im folgenden beziehen wir uns auf die verbesserte Variante.

3.3.3 Heuristik zur Vermeidung von Überbestimmtheit

Zusätzlich zur Suchheuristik wird ein **Signifikanztest** durchgeführt, der verhindern soll, daß Regeln weiter verfeinert werden, die zu wenig positive, aber zu viele negative Beispiele abdecken. Der Signifikanztest des CN2 besteht aus der *Loglikelihood-Ratio-Statistic* (Definition 3.4), die einen bestimmten Grenzwert überschreiten muß, damit die jeweilige Regel als signifikant betrachtet wird.

Definition 3.4 (Loglikelihood-Ratio-Statistic)

$$LRS(r) := 2 * (p * log(\frac{p}{e_p}) + n * log(\frac{n}{e_n})), \tag{3.9}$$

wobei r die betrachtete Kandidatenregel ist und p bzw. n die Anzahl der positiven bzw. negativen durch r abgedeckten Beispiele ist. Für e_p und e_n gilt

$$e_p := (p+n) * \frac{P}{P+N} \tag{3.10}$$

$$e_n := (p+n) * \frac{N}{P+N} \tag{3.11}$$

wobei P bzw. N die absolute Anzahl der positiven bzw. negativen Beispiele in der Trainingsmenge ist.

3.4 Instanziierung des CN2

Ausgehend von dem Pseudo Code eines allgemeinen SeCo aus Quelltext 2.1, werden nun die variablen Funktionen aus Tabelle 3.1 definiert. Man kann sie unterteilen in triviale Funktionen (siehe Quelltext 3.1) und nicht-triviale Funktionen (siehe Quelltext 3.2).[12] *EvaluateRule* wurde bereits in Definition 3.3 definiert und verwendet die Funktion LPA. Ein *PostProcessing* wurde nicht definiert und das *RuleStoppingCriterion* besagt, daß der Lernprozeß erst endet, wenn alle positiven Beispiele von den Regeln abgedeckt werden. Zu den nicht-trivialen Funktionen gehört die Verfeinerung (*RefineRule*), die für eine Kandidatenregel eine Menge von verfeinerten Regeln produziert. Dabei wird davon ausgegangen, daß alle möglichen Bedingungen, die für eine Verfeinerung in Frage kommen, bereits in der Menge *CONDS* berechnet wurden. Das *StoppingCriterion* verwendet die LRS-Funktion aus Definition 3.4 und die *FilterRules*-Funktion liefert von den gegebenen Regeln (*rules*) nur die beste Regel bzgl. *EvaluateRule* zurück. *THRESHOLD* bezeichnet einen variablen Grenzwert, der festlegt, wann eine Regel signifikant ist. Damit ist die Definition des CN2 komplett.

[12] Diese Unterteilung ist für die hier betrachteten Fälle hilfreich, sollte aber nicht als allgemeingültig gesehen werden. Denkt man an eine Bottom-Up-Strategie, so könnte die Funktion InitializeRule alles andere als trivial sein.

Quelltext 3.1 Variable Funktionen des CN2 (triviale)

```
procedure EvaluateRule(rule)
{
    return LPA(rule)
}

procedure PostProcess(theory)
{
    // optionale Funktion
}

procedure InitializeRule(examples)
{
    return '→ true'
}

procedure SelectCandidates(rules, examples)
{
    // Wähle alle Regeln aus
    return rules;
}

procedure RuleStoppingCriterion(theory, rule, examples)
{
    return X_P() = ∅
}
```

Quelltext 3.2 Variable Funktionen des CN2 (nicht-triviale)

```
procedure RefineRule(candidate, examples)
{
    refinements := ∅
    foreach test ∈ CONDS
    {
        newrule := candidate ∪ test
        refinements := refinements ∪ newrule
    }
    return refinements;
}

procedure StoppingCriterion(refinement, evaluation, examples)
{
    return ( LRS(refinement) <= THRESHOLD )
}

procedure FilterRules(rules, examples)
{
    orderedRules := orderByEval(rules)
    return orderedRules.first()
}
```

3.5 Eigenschaften des BEXA

In diesem Abschnitt wird der BEXA Algorithmus beschrieben. Dabei soll die prinzipielle Funktionsweise deutlich werden und eine Abbildung auf die variablen Funktionen des allgemeinen SeCo-Algorithmus aus Abschnitt 2.5 erfolgen. BEXA [16] steht für "Basic EXclusion Algorithm", der ebenfalls zu den Separate-and-Conquer Algorithmen gehört. Die Kernidee des Algorithmus besteht in der Verwendung von disjunktiven Regeln, die eine Menge von möglichen Attributwerten für die jeweiligen Attribute beschreiben. Die Spezialisierung der Regeln erfolgt dann durch rausstreichen (exclusion) einzelner Attributwerte bzw. Disjunktionen. Das Ziel ist, diese Kernidee in einer anschließenden Fallstudie zu untersuchen.

3.5.1 Die Hypothesensprache

Der BEXA bedient sich einer erweiterten Hypothesensprache. Unterstützt werden nominale und lineare Attribute.

Nominale Attribute Bei einem nominalen Attribut A gehen wir davon aus, daß es nur eine aufzählbar, endliche Menge $\mathcal{D}_A$ von Attributwerten annehmen kann. Während andere Algorithmen wie CN2 Bedingungen formulieren wie $[A = a]$ mit $a \in \mathcal{D}_A$, so erzeugt der BEXA Disjunktionen wie $[A \in \{a, b, c\}]$ mit $a, b, c \in \mathcal{D}_A$. Anders formuliert: $A = a \vee A = b \vee A = c$. Die Hypothesensprache des BEXA ist somit also mächtiger als die von CN2. Um Disjunktionen wie $[A \in \{a, c\}]$ darstellen zu können, wird, basierend auf der Annahme, daß man

alle möglichen Attributwerte (also alle Elemente der Menge $\mathcal{D}_A$[13]) kennt, eine äquivalente Formulierung dieser Bedingung gebildet. Dafür verwenden wir die Definitionen der Subdomäne (3.1) und der inversen Subdomäne (3.2). Dann gilt

$$[A \in \mathcal{S}_A] \equiv \bigwedge_{x \in \overline{\mathcal{S}_A}} A \neq x. \tag{3.12}$$

Die Verfeinerung durch Entfernen eines Elementes aus $\mathcal{S}_A(r)$ entspricht dem Hinzufügen einer Konjunktion $A \neq x$ zum Regelkörper.

Beispiel Wir möchten den Regelkörper $[A \in \{a, b, c\}]$ verfeinern, indem wir Element b ausschließen. Desweiteren gilt $\mathcal{D} := \{a, b, c\}$. In VL lautet der spezialisierte Regelkörper $[A \in \{a, c\}]$. In einer Regel könnte man diese Spezialisierung auch durch Hinzufügen von $A \neq b$ formulieren. [14]

Lineare Attribute Für lineare (oder numerische) Attribute verwendet BEXA die Komparatoren $\leq$ und $>$, um Bedingungen bilden zu können, die prüfen, ob ein Attributwert in einem bestimmten Intervall liegt. Es entstehen also für ein numerisches Attribut B Bedingungen wie $[B \leq 10]$.

3.5.2 Suchverfahren

Auch der BEXA verwendet eine *TopDown-Strategie*. Die Verfeinerung im BEXA erfolgt durch Ausschluß möglicher Attributwerte. So kann z.B. $[A \in \{a, b, c\}]$ zu folgenden Ausdrücken verfeinert werden, indem jeweils ein Attributwert gestrichen wird:

- $[A \in \{b, c\}]$
- $[A \in \{a, c\}]$
- $[A \in \{a, b\}]$

Es ergibt sich somit für jeden Attributwert, der im Regelkörper vorkommt, eine Verfeinerungsmöglichkeit. Der Verfeinerungsprozeß folgt dem Grundprinzip, für jede dieser Möglichkeiten eine verfeinerte Regel zu erstellen. Dies soll am Beispiel in Tabelle 3.2 veranschaulicht werden. Hier haben wir 6 Beispiele und zwei Attribute, nämlich $A \in a, b, c$ und $B \in x, y$. Bei Anwendung des BEXA auf dieses Problem ergibt sich eine Suche nach den besten Konjunktionen, welche in Abbildung 3.1 dargestellt ist. Man beginnt mit der allgemeinsten[15] Konjunktion $[a, b, c][x, y]$ (oberster Knoten im Verband), die alle Attributwerte erlaubt. Die nächste Verfeinerung erfolgt durch Weglassen eines Attributwertes, woraus sich die zweite Ebene im Verband ergibt. Die folgenden Ebenen ergeben sich ebenfalls durch Verfeinerung der vorhergehenden Ebene bis schließlich für mindestens ein Attribut keine Werte mehr erlaubt sind (null). Die Mengen X_P und X_N geben die positiven bzw. negativen Beispiele an, welche die jeweilige Konjunktion erfüllen.

Man kann sich vorstellen, daß bei mehr Attributen und Attributwerten der Verband sehr stark wächst und dies zu einem zu hohen Rechenaufwand (Komplexität) führen würde. Deshalb werden folgende Einschränkungen vorgenommen:

[13] Im folgenden schreiben wir nur $\mathcal{D}$, wenn offensichtlich ist, auf welches Attribut es sich bezieht.

[14] Zum Vergleich mit der allgemeinen Herleitung, es gilt $\mathcal{S}_A(r) = \{a, c\}$ und $\overline{\mathcal{S}_A(r)} = \{b\}$.

[15] Auch most general conjunction (mgc) genannt.

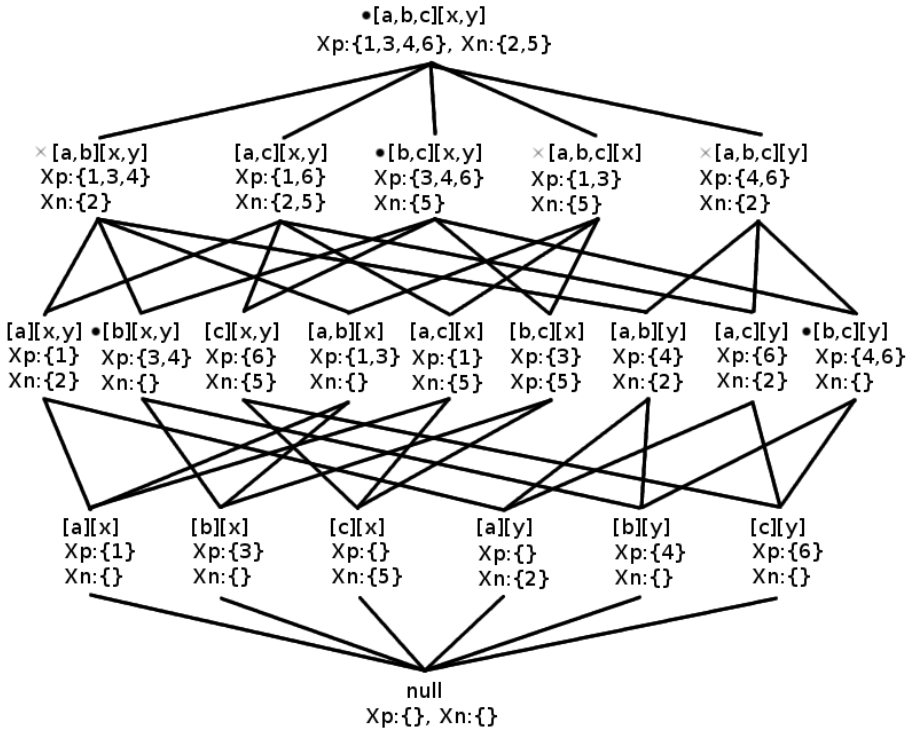

Abbildung 3.1: BEXAs Pfad im Verband der Konjunktionen

Tabelle 3.2: Kleines Lernproblem zur Erklärung der BEXA Verfeinerung

#	A	B	Class
1	a	x	+
2	a	y	-
3	b	y	+
4	b	y	+
5	c	x	-
6	c	y	+

- **(1) prevent-empty-conjunctions-restriction** verhindert, daß Regeln erzeugt werden, die keine positiven Beispiele mehr abdecken.
- **(2) uncover-new-negative-restriction** verhindert, daß Regeln erzeugt werden, die keine weiteren negative Beispiele ausschließen.
- **(3) irredundancy-restriction** vermeidet Redundanz in der Regel, die dadurch entstehen kann, daß einzelne negative Beispiele durch mehrere Konjunktionen ausgeschlossen werden, so daß schließlich einzelne Bedingungen überflüssig werden.

Um sich die Wirkung der drei Restriktionen zu veranschaulichen, betrachte man Abbildung 3.1 in der die Knoten mit · oder x diejenigen sind, die in die Menge specializations aufgenommen werden. Die mit · markierten Knoten sind diejenigen, die als beste Konjunktionen bestimmt werden. Es werden also wesentlich weniger Knoten erzeugt und evaluiert. In Abbildung 3.2 ist ein Vergleich, wie sich die einzelnen Restriktionen bei verschiedenen Lernproblemen auf die Anzahl der erzeugten Konjunktionen auswirken. Auf der x-Achse sind die verschiedenen Lernprobleme und die jeweils verwendeten Restriktionen eingetragen, während die y-Achse den Prozentsatz der erzeugten Spezialisierungen angibt. Dabei wird die Anzahl der Spezialisierungen, die ohne Verwendung von Restriktionen erzeugt werden, als Referenzwert (100 %) genommen. Man kann erkennen, daß die *prevent-empty-conjunctions-restriction* und die *uncover-new-negative-restriction* einen großen Einfluß auf die Zahl der erzeugten Spezialisierungen haben, während der Einfluß der *irredundancy-restriction* wesentlich geringer ist.

3.5.3 Heuristik zur Vermeidung von Überbestimmtheit

Wie auch der CN2, macht BEXA den Signifikanztest mit der *Likelihood-Ratio-Statistic* (Definition 3.4). Zusätzlich wird ein **Stoppe-Wachstum-Test** durchgeführt. Hierbei testet man die Kandidatenregeln, indem sie mit den jeweiligen Vorgängerregeln verglichen werden. Die **Vorgängerregel** r_v ist die Regel, aus welcher die betrachtete Regel r durch Verfeinerung entstanden ist. Eine Regel r wird mit ihrer Vorgängerregel r_v wie folgt verglichen:

Es werden folgende Werte bestimmt:

- p Die Anzahl positiver Beispiele, die durch r abgedeckt werden.
- n Die Anzahl negativer Beispiele, die durch r abgedeckt werden.

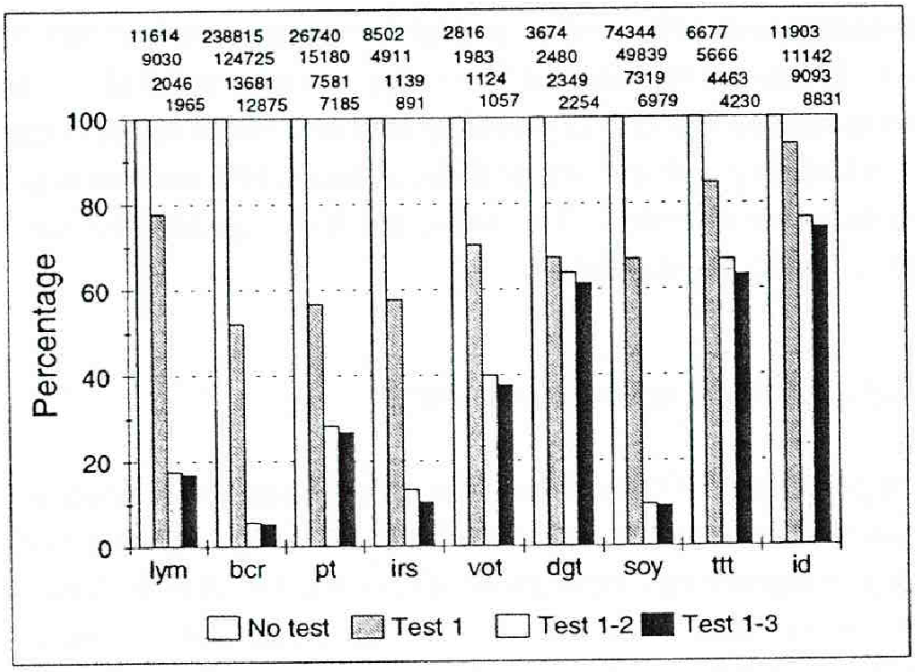

Abbildung 3.2: Gesamtzahl der erzeugten Spezialisierungen abhängig von Restriktionen aus [16]

- p_v Die Anzahl positiver Beispiele, die durch r_v abgedeckt werden.
- n_v Die Anzahl negative Beispiele, die durch r_v abgdeckt werden.

Basierend auf diesen Werten wird dann die Verteilung

$$\chi^2 = \frac{(p - p_v)^2}{p_v} + \frac{(p - n_v)^2}{n_v} \tag{3.13}$$

berechnet. Anhand eines vorgegebenen Grenzwertes (Threshold) wird dann geprüft, ob sich χ^2 innerhalb dieser Grenze befindet. Ist das der Fall, so ist die Verfeinerung r von r_v *nicht signifikant besser* als r_v und r wird daraufhin aus der Menge der Kandidatenregeln entfernt, um diese Konjunktion nicht unnötig weiter zu verfeinern.

3.6 Instanziierung des BEXA

Auch die Eigenschaften des BEXA lassen sich auf die variablen Funktionen aus Tabelle 3.1 abbilden, um mit Quelltext 2.1 einen vollständigen Pseudo-Code des Algorithmus zu erhalten. In Quelltext 3.3 sind die trivialen Funktionen definiert. Lediglich beim RuleStoppingCriterion gibt es einen Unterschied[16] zum CN2, der darin besteht, daß das Ende des Lernprozeß nicht nur durch die Abdeckung aller positiven Beispiele erreicht werden kann, sondern auch dann, wenn keine beste

[16] Im Original [16] hat der BEXA tatsächlich ein Postprocessing, was in dieser Arbeit jedoch weggelassen wurde.

Quelltext 3.3 Variable Funktionen des BEXA (triviale)

```
procedure EvaluateRule(rule)
{
    return LPA(rule)
}

procedure PostProcess(theory)
{
    // optionale Funktion
}

procedure InitializeRule(examples)
{
    return '→ true'
}

procedure SelectCandidates(rules, examples)
{
    // Wähle alle Regeln aus
    return rules;
}

procedure RuleStoppingCriterion(theory, rule, examples)
{
    return ( X_P() = ∅ || rule=null )
}
```

Regel gefunden wurde. Dadurch kann es passieren, daß die gelernten Regeln nicht alle Beispiele abdecken, wenn selbst die beste Regel als qualitativ schlecht bezeichnet wird. Dies soll der Überbestimmtheit entgegenwirken und kann sogar Fehler in der Trainingsmenge ausgleichen.

In Quelltext 3.4 und 3.5 sind die nicht-trivialen Funktionen definiert. Die

Quelltext 3.4 RefineRule Funktion des BEXA

```
procedure RefineRule(candidate, examples)
{
   refinements := ∅
   // Remove from candidate.usable all the values that will
   // lead to unnecassary specializations
   foreach a ∈ candidate.usable
   {
      if ( X_P(test) ⊆ X_P(a)                                  (1)
        || X_N(test) ∩ X_N(a) = ∅                              (2)
        || {X_N(b) | b ∈ candidate.excluded ∪ {a} }            (3)
           is a redundant partial cover of N )
      {
         candidate.usable := candidate.usable \ {a}
      }
   }

   // Generate all useful specializations of the candidate
   for a ∈ candidate.usable
   {
      c' := candidate \ a
      X_P(c') := X_P(candidate) \ X_P(a)
      X_N(c') := X_N(candidate) \ X_N(a)
      c'.usable := candidate.usable \ {a}
      c'.excluded := candidate.excluded ∪ {a}
      refinements := refinements ∪ {c'}
   }

   return refinements;
}
```

Funktion *RefineRule* erzeugt für eine Kandidatenregel (*candidate*) alle Verfeinerungen bis auf solche, die durch die Restriktionen (1), (2) und (3) ausgeschlossen werden (siehe Abschnitt 3.5.2). Für jede Regel gibt es zwei Mengen von Bedingungen: *usable* und *excluded*. Sie enthalten die verwendbaren bzw. die bereits ausgeschlossenen Bedingungen. Die initiale Regel, mit welcher der Lernprozeß startet, enthält in *usable* alle möglichen Bedingungen, während *excluded* leer ist. Das *StoppingCriterion* gleicht dem des CN2 im Gegensatz zu *FilterRules*, welches aus zwei Filtern besteht. Der erste Filter ist der bereits in Abschnitt 3.5.3 erwähnte χ^2-Filter und der Zweite ist ein Beamwidth-Filter. Das bedeutet, daß die N besten Regeln in der Menge der Kandidatenregeln verbleiben, während die restlichen entfernt werden.

Quelltext 3.5 Variable Funktionen des BEXA (nicht-triviale)

```
procedure StoppingCriterion(refinement, evaluation, examples)
{
    return ( LRS(refinement) <= THRESHOLD_LRS )
}

procedure FilterRules(rules, examples)
{
    // χ² filter
    foreach rule ∈ rules
    {
        if ( χ²(rule) <= THRESHOLD_χ² )
            rules.remove(rule)
    }

    // Beamwidth filter
    orderedRules := orderByEval(rules)
    filteredRules := ∅
    for ( i := 0, i < N, i++ )
    {
        filteredRules.add(orderedRules[i])
    }
    return filteredRules
}
```

4 Das SeCo-Framework

Dieser Abschnitt beschreibt das SeCo-Framework, das eine allgemeine Konzeption zur Umsetzung von SeCo-Algorithmen sein soll und somit auch als Grundlage der Implementation des BEXA Algorithmus aus Abschnitt 3.5 dient. Die konkrete Implementation des BEXA wird dann in Abschnitt 5 erörtert.

4.1 Motivation

In den vorangegangenen Abschnitten wurden zwei Separate-And-Conquer Algorithmen vorgestellt und auch die Unterschiede aufgezeigt. Diese finden sich bei der Hypothesensprache, Suchheuristik, Suchalgorithmus und Suchstrategie. In Abschnitt 2.4 haben wir bereits die Kriterien zum Vergleich von SeCo-Algorithmen genannt. Üblicherweise wäre dies jetzt die Stelle an der man beide Algorithmen implementiert und auf verschiedene Lernprobleme anwendet, um anschließend die Kriterien zu messen, die das Performanzmaß bestimmen. Das ist für den konkreten Fall auch bereits in [16] durchgeführt worden mit dem Ergebnis, daß BEXA in den meisten Fällen bessere Ergebnisse liefert als CN2. Der CN2 und BEXA unterscheiden sich in mehreren Punkten, die teilweise sogar im CN2 ergänzt werden könnten, ohne dadurch einen völlig neuen Algorithmus zu entwerfen.[17] Was uns vor allem interessiert hat, war inwiefern die Kernidee des BEXA, nämlich die Regelverfeinerung durch Rausstreichen von Disjunktionen, zum Erfolg beigetragen hat. Um dies messen zu können, bräuchte man also zwei Implementationen, die sich nur an dieser einen Stelle unterscheiden. Bei weiterer Überlegung stellt man fest, daß es sich hier um ein allgemeineres Problem handelt, das darin besteht, daß man bei SeCo-Algorithmen gerne wissen möchte, in welchem Ausmaß sich einzelne Ideen oder Verbesserungen auswirken.

Um eine einigermaßen komfortable Testumgebung für solche Experimente schaffen zu können, war somit ein modularer Aufbau der Implementation nötig, der eine funktionale Aufspaltung in Komponenten ermöglicht. Darüberhinaus wollte man in der Lage sein, Komponenten austauschen zu können, ohne dabei die Implementation verändern zu müssen.

4.2 Was ist das SeCo-Framework ?

Der Begriff **Framework** meint eine Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für eine bestimmte Klasse von Software darstellen [2, 9]. Das **SeCo-Framework** ist also ein Entwurf zur Implementation von Lernalgorithmen der Klasse Separate-and-Conquer.

Die Gliederung des Frameworks orientiert sich an [6], welches die allgemeine Struktur eines SeCo-Algorithmus beschreibt. Ein wesentlicher Bestandteil sind die sich daraus ergebenden, Schnittstellenbeschreibungen, die den Aufgabenbereich einer SeCo-Komponente abgrenzen und sie substituierbar machen. Darauf aufbauend beinhaltet das Framework eine Factory, die anhand einer XML-Beschreibung aus beliebigen SeCo-Komponenten einen Regellerner zusammenbaut. Voraussetzung dafür ist jedoch, daß für jede Funktion eine Komponente definiert ist, die ihre Aufgabe gemäß der Schnittstellenbeschreibung wahrnimmt.

[17] Man denke dabei an die Suchheuristik LaplaceAccuracy, die man durch LRS austauschen könnte oder auch die Suchrestriktionen des BEXA.

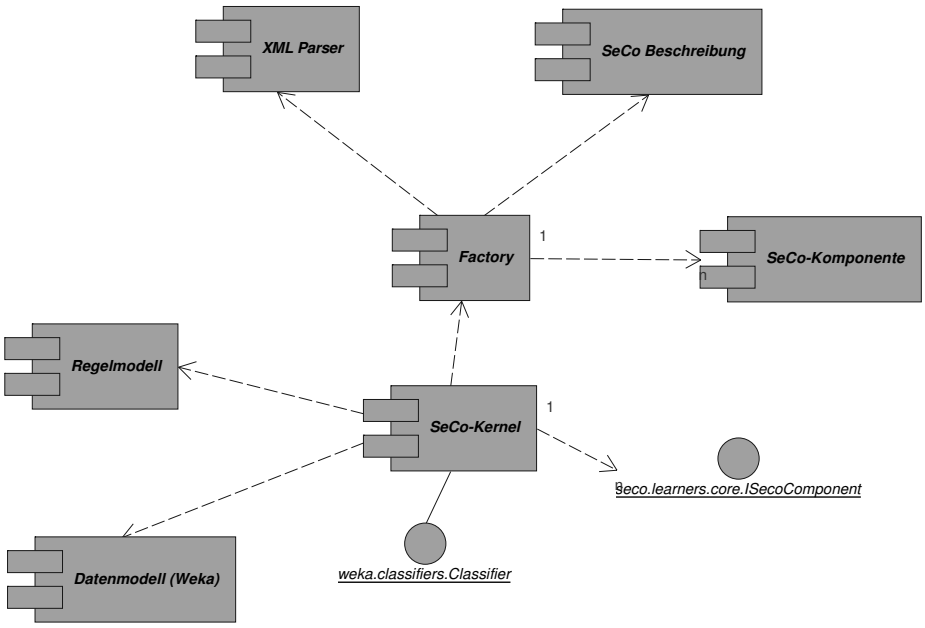

Abbildung 4.1: Komponenten des SeCo-Framework

4.3 Voraussetzungen

Die verwendete Programmiersprache ist **Java**[18]. Das Framework verwendet die Bibliothek **Weka**[17], welche bereits Java-Klassen zur Verarbeitung von Trainings- bzw. Beispielmengen sowie eine abstrakte Defintion eines Klassifizierers[19] beinhaltet. Ansonsten werden folgende Bibliotheken benötigt:

- Log4j Für das Logging. Siehe http://logging.apache.org/log4j/docs.
- XmlPullParser Zum Parsen von xml-Dokumenten.
  Siehe http://www.xmlpull.org.

4.4 Die Komponenten

Abbildung 4.1 zeigt die Komponenten des Frameworks und deren Beziehungen zueinander. Ausgehend von der SeCo-Beschreibung (in XML) des zu generierenden Algorithmus erzeugt die Factory mit Hilfe eines XML-Parsers[20] zunächst die einzelnen SeCo-Komponenten, die in der SeCo-Beschreibung angegeben sind, und mit deren Hilfe anschließend den *SeCo-Kernel*. Dieser Kernel ist dabei eine allgemeine Implementation des Kernalgorithmus, wie er in Sektion 2.5 beschrieben wird. Erst durch die Verknüpfung mit den SeCo-Komponenten wird er zu einem konkreten Regellerner. Um die SeCo-Komponenten austauschen zu können, implementieren sie alle das selbe Interface *seco.learners.core.ISecoComponent*. Da sich ihre Aufgaben unterscheiden, gibt es weitere Schnittstellenbeschreibungen, die auf dem ISecoComponent Interface aufbauen und aufgabenspezifische

[18] Entwickelt von Sun Microsystems, siehe http://java.sun.com
[19] Die entsprechende Java-Klasse nennt sich weka.classifiers.Classifier.
[20] Verwendet wurde hier ein Pullparser, siehe http://www.xmlpull.org.

Merkmale hinzufügen. Welche Schnittstellen das sind und welche Aufgaben sie haben, wird in Sektion 4.5 erläutert.

Der SeCo-Kernel, sowie dessen Teilkomponenten, operieren mit zwei Modellen:

- Das **Regelmodell** ist eine Modellierung Hypothesensprache, auch **Language Bias** genannt. Es beinhaltet sowohl Konzepte für Regeln, als auch Regelmengen, Bedingungen und Atome. (Siehe auch Sektion 4.7.)
- Das **Datenmodell** ist eine Modellierung der Beispiele und Beispielmengen, die als Trainings- und auch Testmengen verwendet werden. Es handelt sich also um eine Sprache für Beispiele. (Siehe auch Sektion 4.8.)

Damit der SeCo-Kernel nun von beliebigen Anwendungen integriert werden kann, verfügt er über das Interface *weka.classifiers.Classifier*, das bereits im Weka-Projekt definiert worden ist und somit eine kombinierte Verwendung von Weka, SeCo-Kernel und der Factory erlaubt.

4.5 SeCo-Komponenten

In Anlehnung an die variablen Funktionen aus Tabelle 3.1, die zur Instanziierung eines SeCo-Algorithmus definiert werden müssen, wurden folgende SeCo-Komponenten deklariert:

- **CandidateSelector** Das Auswahlverfahren, welches die Kandidatenregeln auswählt.
- **PostProcessor** (Optionale) Komponente, die eine Nachbearbeitung der erlernten Theory beinhaltet.
- **RuleEvaluator** Die Auswertungsfunktion zur Bewertung von Regeln.
- **RuleFilter** Ein Filter der Regeln aus einer Menge filtert.
- **RuleInitializer** Zur Erzeugung der ersten Regel.
- **RuleRefiner** Diese Komponente modifiziert/verbessert vorhandene Regeln und erzeugt somit neue Regeln.
- **RuleStoppingCriterion** Definiert, wann der Lernprozeß beendet wird.
- **StoppingCriterion** Definiert, ob eine weitere Anwendung des RuleRefiners auf eine Regel erfolgen soll.

4.6 Heuristiken

Um eine Schnittstelle für verschiedene Suchheuristiken zu schaffen, wurde von Johannes Fürnkranz eine abstrakte Suchheuristik entworfen, die als Grundlage für die Umsetzung weiterer Heuristiken dient. Die bereits implementierten Heuristiken sind (anhand der Variablen aus Tabelle 4.1) in Tabelle 4.2 formuliert.

Tabelle 4.1: Variablendefinitionen für die Heuristiken

Variable	Beschreibung
P	Die Gesamtzahl der positiven Beispiele.
N	Die Gesamtzahl der negativen Beispiele.
r	Die Kandidatenregel.
tn	Anzahl der Beispiele, die von r korrekterweise negativ klassifiziert sind (true negatives).
fn	Anzahl der Beispiele, die von r irrtümlicherweise negativ klassifiziert sind (false negatives).
tp	Anzahl der Beispiele, die von r korrekterweise positiv klassifiziert sind (true positives).
fp	Anzahl der Beispiele, die von r irrtümlicherweise positiv klassifiziert sind (false positives).

Tabelle 4.2: Heuristiken im SeCo-Framework

Name der Heuristik	Formel
Accuracy	$A(r) = \frac{tp+tn}{P+N}$
Correlation	$CORR(r) = \frac{tp*tn-fn*fp}{\sqrt{PN(tp+fp)(fn+tn)}}$
Difference	$Diff(r) = tp - fp$
FoilGain	$Foil(r) = tp(log_2 \frac{tp}{tp+fp} - log_2 c)$
Laplace	$LAP(r) = \frac{tp+1}{tp+fp+2}$
LinearCosts	$L(r) = c * tp - (1-c)fp$
MEstimate	$M(r) = \frac{tp+m\frac{P}{P+N}}{tp+fp+m}$
Precision	$Prec(r) = \frac{tp}{tp+fp}$
RateDiff	$RD(r) = \frac{tp}{tp+fn} - \frac{fp}{fp+tn}$
WRAcc	$WRAcc(r) = \frac{tp}{P+N} - \frac{tp+fp}{P+N} * \frac{P}{P+N}$

4.7 Modellierung der Regeln

Diese Modellierung wurde ebenfalls von Johannes Fürnkranz erarbeitet. Sie definiert eine Regel durch ihren Kopf H und einer Konjunktion von Bedingungen C_i.

$$C_1 \wedge C_2 \wedge ... \wedge C_n \rightarrow H \tag{4.1}$$

Dabei können sich die Bedingungen sowohl auf nominale sowie auf numerische Attribute beziehen. Im Fall der **nominalen** Attribute gibt es eine endliche Menge mit vorgegebenen, möglichen Werten ohne Ordnungsrelation, die das Attribut annehmen kann. Wir sprechen dann von der *Domäne* $\mathcal{D}_A$ des Attributs A. Somit kann eine Bedingung C_i für ein nominales Attribut A entweder $A = x$ oder $A \neq x$ lauten, mit $x \in \mathcal{D}_A$. Bei einem **numerischen**[21] Attribut B gibt es ebenfalls eine Domäne $\mathcal{D}_B$, die jedoch unendlich sein kann, aber in jedem Fall eine Ordnungsrelation besitzt. Um die Anzahl der Bedingungen, die sich auf numerische Attribute beziehen, zu verringern[22], prüfen wir dazu nicht mit den

[21] Auch lineare Attribute genannt.

[22] Verringern bezieht sich hier auf den Regelverfeinerungsprozeß, bei dem zur Spezialisierung von Regeln weniger Bedingungen zu erzeugen sind, wenn man sie in Form von Intervallen

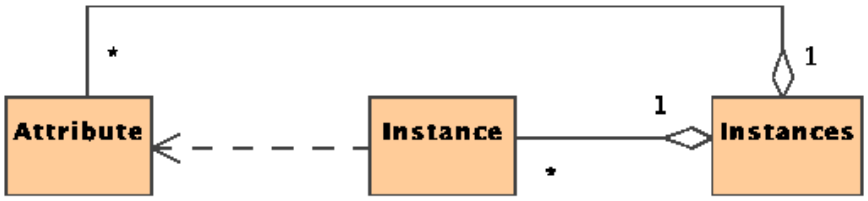

Abbildung 4.2: Modell der Beispieldaten

Operatoren $=$ und $\neq$, sondern bilden Intervalle. Dies setzt allerdings voraus, daß es eine Relation auf $\mathcal{D}_B$ gibt, die sowohl reflexiv als auch transitiv ist. In der konkreten Implementation verwenden wir für solche Attribute ausschließlich Fließkommazahlen[23], auf die wir die Vergleichsoperatoren $>$ und $\leq$ anwenden. Damit ergeben sich die beiden möglichen Bedingungen $B > x$ und $B \leq x$ für ein numerisches Attribut B mit $x \in \mathcal{D}_B$. Im **Regelkopf** H wird die Schlußfolgerung definiert, die dann gilt, wenn ein Beispiel I alle Bedingungen C_i der Regel R erfüllt. Da wir Beispiele klassifizieren möchten, hat H immer folgendes Aussehen: $H \equiv Class(I) = c$, c sei eine Klasse.

Die Regeln R_i werden als **Regelmenge** M zusammengefaßt und bzgl. des Ergebnisses des Regellerners als Theorie verwendet, um beliebige Beispiele klassifizieren zu können. Dies kann zum Testen der gelernten Regeln, sowie zur Prognose von bisher nicht klassfizierten Beispielen geschehen. Dabei erfolgt die Anwendung der Theorie $\mathcal{T}$ auf ein Beispiel I wie folgt: Im folgenden bezeichnen wir eine Regel R auf ein Beispiel I zutreffend, wenn I alle Bedingungen C_i aus R erfüllt. Die Regeln in $\mathcal{T}$ sind geordnet abgelegt in der Reihenfolge in der sie zu $\mathcal{T}$ hinzugefügt wurden. Man wendet nun die Regeln R_1 bis R_n nacheinander auf I an bis eine zutrifft, und $n := |\mathcal{T}|$ sei die Anzahl der Regeln in $\mathcal{T}$ ohne die Defaultregel. Die **Defaultregel** wird zuletzt angewendet, wenn keine andere Regel zutreffend ist und ordnet dem Beispiel I jene Klasse zu, die in der Trainingsmenge am häufigsten auftrat. Trifft vorher eine andere Regel zu, so wird die Klasse gewählt, die in deren Regelkopf H_{R_i} benannt ist.

4.8 Datenmodell

In Abbildung 4.2 sind die Java-Klassen zu sehen, die zur Verarbeitung von Beispieldaten verwendet werden. Sie befinden sich in der Weka Bibliothek, genauer: im *weka.core* package[24]. Die **Instances**-Klasse beschreibt eine Menge von Beispielen und setzt sich aus beliebig vielen Objekten der **Instance**- und der **Attribute**-Klasse zusammen. Ein Instance-Objekt beschreibt ein einzelnes Beispiel und beinhaltet die Attributwerte für alle oder einen Teil der Attribute, die zu Instances in Beziehung stehen.

formuliert anstatt $B = x$. Siehe dazu auch [16]

[23] Genauer gesagt, Java Variablen vom Typ *double*

[24] Die Weka-Bibliothek beinhaltet noch wesentlich mehr. Wir beschränken uns hier jedoch auf die Teile, die im SeCo-Framework Verwendung finden.

4.9 Konfiguration mit SeCo-Factory

Ohne Bereitstellung von Konfigurationsmöglichkeiten erfordert jede Variation eines Algorithmus die Veränderung von dessen Quellcode, was das Testen erschwert und die Gefahr birgt, Fehler einzubauen. Während Weka den Ansatz verfolgt, Eigenschaften des Klassifizierers mittels Kommandozeilenoptionen veränderbar zu machen, so sollte hier, zugunsten einer einfacheren und übersichtlicheren Handhabung, eine externe Konfigurationsdatei verwendet werden. Aufgrund der Überlegung, daß man nicht nur Eigenschaften, sondern auch Teile des Quellcodes bzw. Komponenten austauschbar haben möchte, beinhaltet die Konfiguration auch die zu verwendenden Komponenten. Als Beschreibungssprache der Konfigurationsdatei wurde XML (extensible markup language) gewählt. Das *Grundprinzip* dieser Sprache besteht darin, daß man konkrete Java-Klassen benennen und zueinander in Beziehung setzen kann. Gibt man diese Beschreibung an die SeCo-Factory so erstellt diese eine Instanz eines SeCo-Klassifizierers, die durch Aggregation mit den angegebenen SeCo-Komponenten gebildet wird. **Aggregation** ist dabei wie folgt definiert [7]: *Aggregation bedeutet, daß ein Objekt ein anderes besitzt oder dafür zuständig ist. Wir sprechen im allgemeinen davon, daß ein Objekt ein anderes hat oder daß es ein Teil von ihm ist.* Die derzeit verfügbaren Elemente und Attribute sind in Tabelle 4.3 zu sehen.

Tabelle 4.3: Elemente der SeCo XML-Beschreibung

Element	Attribut	Beschreibung
seco		Das Wurzelelement, welches die Konfiguration eines SeCo-Klassifizierers beinhaltet.
secomp		Eine SeCo-Komponente.
	interface	Die Schnittstelle bzgl. der AbstractSeco-Klasse[25]. Mögliche Werte sind candidateselector, postprocessor, ruleevaluator, rulefilter, ruleinitializer, rulerefiner, rulestoppingcriterion, stoppingcriterion.
	classname	Name der Java-Klasse, welche die Komponente implementiert.
	package	(Optional) Das Java-Package, welches die angegebene Java-Klasse beinhaltet. Als Standardwert wird seco.learners.components angenommen.
jobject		Ein beliebiges Java-Objekt.
	classname	siehe secomp
	package	siehe secomp
	setter	Teilname der setter-Methode, welche für die Aggregation dieses Objektes mit dem Objekt der nächsthöheren Ebene verwendet wird. Wenn also die Java-Methode z.B. setHeuristic heißt, so muß als setter 'heuristic' angegeben werden.
property		Zur Festlegung einer spezifischen Eigenschaft eines Objektes.
	name	Name der Eigenschaft
	value	Wert der Eigenschaft

[25] siehe seco.learners.core.AbstractSeco

Basierend auf einer XML-Beschreibung des neuen SeCo-Algorithmus soll ein Klassifizierer erzeugt werden. Der XML-Parser liest diese Beschreibung und ermöglicht die Konstruktion eines AST (*Abstract Syntax Tree*). Dieser Baum wird anschließend nach dem **post-order** Verfahren traversiert. Abbildung 4.3 zeigt das Vorgehen beim rekursiven Erzeugen der Objekte genauer. Sofern es sich bei dem besuchten Knoten um den Typ seco, secomp oder jobject handelt, wird das entsprechende Objekt O erzeugt und mit den Nachfolgeknoten NK verknüpft. Je nachdem, um welche Art von Nachfolgeknoten es sich handelt, wird die Verknüpfung bzw. Aggregation unterschiedlich durchgeführt.

- **property** Eine Property besteht aus einem Namen und einem Wert und kann mit Hilfe der setProperty-Methode von O gesetzt werden, sofern O diese Methode implementiert. Da bei jobject-Knoten keine Interfaces vorausgesetzt werden können, wird mit Hilfe des *java.lang.reflect* packages versucht, diese Methode aufzurufen.
- **secomp** beschreibt eine SeCo-Komponente und kann nur an seco-Objekte angehängt, welche immer die Methode *setSecoComponent* implementieren.
- **jobject** Um eine beliebige Modularisierung des SeCo-Algorithmus zu ermöglichen, kann ein beliebiges Java-Objekt mit einer benannten setter-Methode mit O verknüpft werden. Dazu **muß** O in jedem Fall diese setter-Methode auch implementieren, da sonst die Verknüpfung fehlschlägt und der SeCo-Algorithmus nicht erzeugt werden kann. Der Methodenaufruf lautet also z.B. O.setHeuristic(Object(NK)), wobei Object(NK) für das, durch den Nachfolgeknoten erzeugte, Objekt steht.

Als konkretes Beispiel wollen wir hier den *RuleFilter* für den BEXA Algorithmus konfigurieren. Dieser besteht, wie bereits in Abschnitt 3.6 erwähnt, aus dem χ^2-Filter und dem Beamwidth-Filter. Als RuleFilter kann jedoch nur eine Java-Klasse angegeben werden. Um dennoch eine modulare Implementation mit verschiedenen Java-Klassen zu ermöglichen, kann der RuleFilter aus mehreren Klassen zusammengebaut, indem eine weitere Klasse namens MultiRuleFilter definiert wird. Sie implementiert das Interface *rulefilter*. Das sieht in der XML Beschreibung aus wie in Quelltext 4.1. Zur Implementation des MultiRuleFil-

Quelltext 4.1 Definition eines RuleFilter

```
<secomp interface="rulefilter" classname="MultiRuleFilter">
</secomp>
```

ter seien die beiden wichtigsten Methoden in Quelltext 4.2 dargestellt. Mit der *setFilter* Methode können beliebige andere RuleFilter dem MultiRuleFilter hinzugefügt werden, während *filterRules* die Umsetzung des RuleFilter-Interface ist, das nacheinander alle hinzugefügten RuleFilter auf die gegebene Regelmenge anwendet. Was jedoch noch fehlt sind die beiden konkreten Filter. Um die SeCo-Factory zum Aufruf der *setFilter* Methode zu bewegen, werden die Filter mit dem *jobject*-Element hinzugefügt, wie in Quelltext 4.3 zu sehen ist. Das Attribut *setter* bezieht sich auf die *setFilter* Methode des MultiRuleFilter-Objekts. Das Präfix *set* wird implizit vorausgesetzt. Auf diese Weise können auch verschiedene setter-Methoden zur Aggregation verwendet werden. Nach Instanziierung

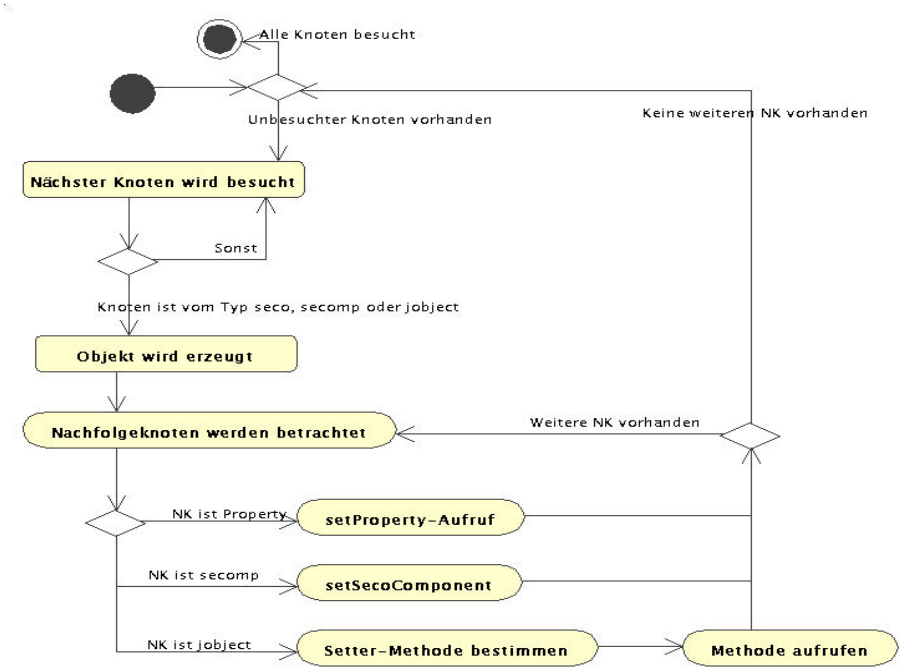

Abbildung 4.3: Traversierung des Konfigurationsbaumes (AST)

Quelltext 4.2 Implementation des MultiRuleFilter

```
public void setFilter( Object filter )
{
    if ( ! (filter instanceof IRuleFilter) )
        throw new ClassCastException(
          "setFilter(), "
          +"given Object isn't instance of IRuleFilter!");
    m_filterList.add(filter);
    m_filters = m_filterList.toArray();
}

public void filterRules(java.util.TreeSet rules,
    weka.core.Instances examples)
{
    for ( int i = 0; i < m_filters.length; i++ )
    {
        if (m_filters[i] == null ) break;
        ((IRuleFilter)m_filters[i]).filterRules(rules, examples);
    }
}
```

Quelltext 4.3 Beispiel für aggregierte SeCo-Komponente (MultiRuleFilter)

```
<secomp interface="rulefilter" classname="MultiRuleFilter">
   <jobject classname="ChiSquareFilter" setter="filter">
      <property name="threshold" value="0.9"/>
   </jobject>
   <jobject classname="BeamWidthFilter" setter="filter">
      <property name="beamwidth" value="3"/>
   </jobject>
</secomp>
```

der Objekte würden dann folgende Aufrufe durchgeführt (mrf=MultiRuleFilter, csf=ChiSquareFilter, bwf=BeamWidthFilter):

```
mrf.setFilter(csf);
mrf.setFilter(bwf);
```

Dies geschieht mittels **Reflection**, was im Abschnitt 4.10 genauer erläutert wird. Mit Hilfe des property-Elements werden spezielle Parameter der Filter-Objekte gesetzt. Grundsätzlich kann man für jedes Objekt properties setzen. Welche properties es gibt und welche Werte sie annehmen können, hängt von dem jeweiligen Objekt ab. In Tabelle 4.4 sind ein paar gängige Properties angegeben.

Tabelle 4.4: Properties der SeCo-Objekte

Property	für Element	Beschreibung
threshold	LikelihoodRatio, ChiSquareFilter	Legt den Grenzwert fest, ab welchem Kandidatenregeln als signifikant betrachtet werden.
beamwidth	BeamWidthFilter	Legt fest, die wieviel besten Kandidatenregeln nach einer Verfeinerung beibehalten werden. Die übrigen werden verworfen.
skipdefault class	AbstractSeco	Diese Einstellung für den Kernel kann auf *true* gesetzt werden, damit für die Standardklasse keine Regeln gelernt werden, sondern nur die Standardregel verwendet wird.

Die vollständige Defintion eines SeCo-Algorithmus könnte dann so aussehen wie in Quelltext 4.4. wobei einige Komponenten hier nicht explizit erwähnt wurden, sondern Defaultkomponenten (mehr dazu in Abschnitt 4.12) verwendet werden, wie z.B. der RuleInitializer, der die erste Regel einer Iteration bestimmt. In den meisten Fällen soll eine Regel erstellt werden, die keine Bedingungen beinhaltet und alle Beispiele so klassifiziert, als wenn sie der zu erlernenden Klasse angehören, z.B.: $\top \rightarrow class = A$.

4.10 Objekterzeugung mit *reflection*

Um beliebige Java-Objekte in den SeCo-Algorithmus aufnehmen zu können, ohne daß diese spezielle Interfaces implementieren, wird das java package *java.lang.reflect* verwendet. Es erlaubt der SeCo-Factory, spezielle Merkmale, wie Methodensignaturen, von einem fremden[26] Objekt zu erfragen und diese dann auch anzuwenden. In der SeCo-Factory gibt es zwei Fälle, bei denen von *reflection* Gebrauch gemacht wird: Beim Erzeugen fremder Objekte und bei der Aggregation von Objekten.

[26] Mit fremd ist hier gemeint, daß zunächst nichts über das Objekt bekannt ist, außer, daß es ein *java.lang.Object* ist.

Quelltext 4.4 Beschreibung des **BEXA-Algorithmus**

```
<seco>
  <secomp interface="ruleevaluator"
    classname="DefaultRuleEvaluator">
     <jobject package="seco.heuristics" classname="Laplace"
       setter="heuristic"/>
  </secomp>

  <secomp interface="rulerefiner" classname="BexaRefinerTopDown"
   package="seco.learners.bexa"/>
  <secomp interface="selector" classname="DefaultSelector"/>

  <secomp interface="stopcriterion" classname="LikelihoodRatio">
    <property name="threshold" value="0.9"/>
  </secomp>
  <secomp interface="rulefilter" classname="MultiRuleFilter">
    <jobject classname="ChiSquareFilter" setter="filter">
      <property name="threshold" value="0.9"/>
    </jobject>
    <jobject classname="BeamWidthFilter" setter="filter">
      <property name="beamwidth" value="3"/>
    </jobject>
  </secomp>

</seco>
```

4.10.1 Constructor-Aufruf von fremden Objekten

Zunächst ist nur der Name der Klasse bekannt, der das fremde Objekt angehört. Dieser Name ist der XML-Beschreibung entnommen. Mit Hilfe der Metaklasse *java.lang.Class* ist es dann möglich, mit der statischen Methode *Class.forName()* eine Instanz von *java.lang.Class* zu erhalten, die der Klasse des fremden Objektes entspricht. Durch die *Class.getConstructor* Methode wird der Konstruktor bestimmt, mit welchem dann das fremde Objekt erzeugt wird. Im SeCo-Framework wird dabei ausschließlich der DefaultConstructor[27] verwendet. Im Beispiel des MultiRuleFilter erfolgt die Instanziierung eines Filters mit Namen *ChiSquareFilter* wie in Quelltext 4.5 dargestellt.

Quelltext 4.5 Konstruktoraufruf mit reflection

```
Class cl       = Class.forName("ChiSquareFilter");
Constructor c  = cl.getConstructor(new Class[0]);
Object jobject = c.newInstance(new Object[0]);
```

4.10.2 Aggregation von Objekten

Zur Aggregation wird die, in der XML-Beschreibung angegebene,
Setter-Methode verwendet. Da diese jedoch in keinem Interface erwähnt ist, muß sie mittels *reflection* bestimmt werden. Es wird im SeCo-Framework vereinfachend davon ausgegangen, daß die Setter-Methode nur ein Objekt als Argument bekommt, nämlich genau das Objekt mit dem es verknüpft werden soll. Da die Methodensignatur keine weiteren Anforderungen an das Objekt stellt, kann hier ein beliebiges übergeben werden. In der Regel wird jedoch in dem Objekt, welches die Setter-Methode implementiert, ein Cast[28] stattfinden. Im Quelltext 4.6 ist der entsprechende Java Code für das MultiRuleFilter Beispiel. Dabei sind das MultiRuleFilterobjekt, das Filterobjekt, sowie der Name der setter-Methode gegeben.

Quelltext 4.6 Methodenaufruf mit reflection

```
Object mrf       = ... // MultiRuleFilter Object
Object filter    = ... // Filter Object
Class cl         = mrf.getClass();
Class[] argtype  = new Class[1];
argtype[0]       = Object.getClass();
Method m         = cl.getMethod("setFilter", argtype);
Object[] args    = new Object[1];
args[0]          = filter;
m.invoke(mrf, args);
```

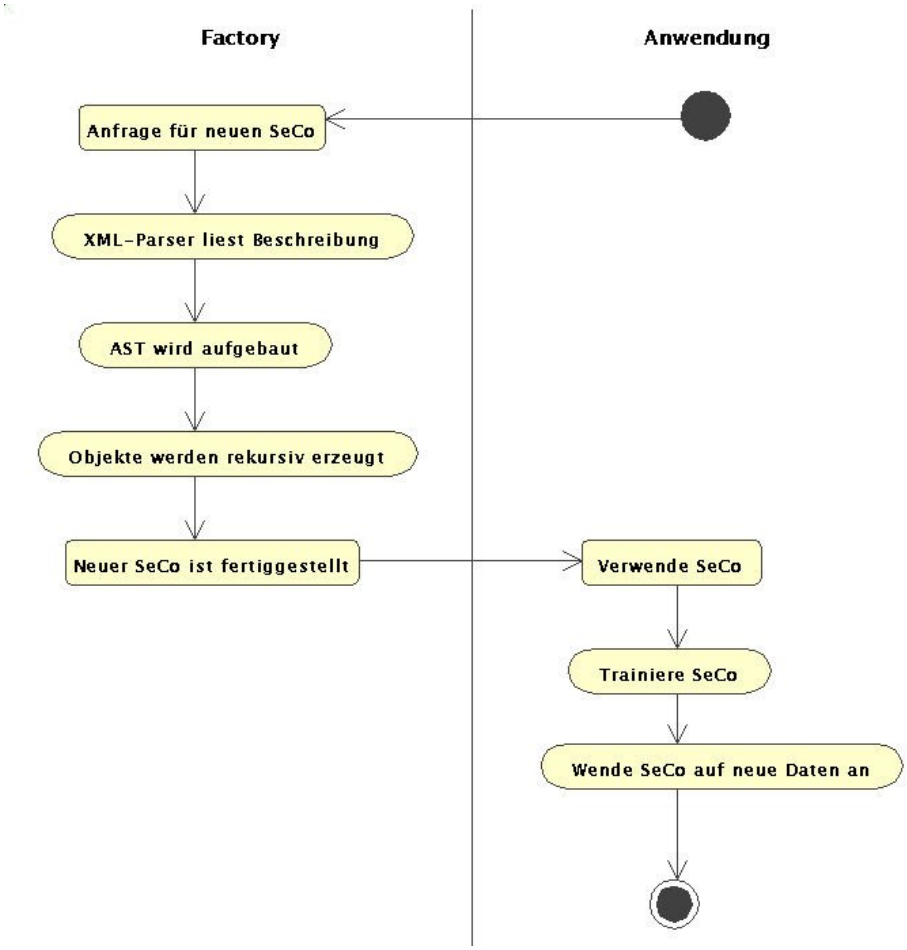

Abbildung 4.4: Verwendung der SeCo-Factory in einer Anwendung

4.11 Integration der SeCo-Factory

Abbildung 4.4 zeigt den Ablauf bei Verwendung der SeCo-Factory innerhalb einer Anwendung. Basierend auf einer XML-Beschreibung des neuen SeCo-Algorithmus soll ein Klassifizierer erzeugt werden. Der XML-Parser liest diese Beschreibung und ermöglicht die Konstruktion eines AST (*Abstract Syntax Tree*). Ist die rekursive Erzeugung der Objekte abgeschlossen, so gibt es ein seco-Objekt, das als Ergebnis an die Anwendung übergeben werden kann. Zu beachten ist, daß wir jetzt zwar einen Klassifizierer haben, dieser aber noch nicht trainiert ist. Dies muß nun durch die Anwendung erfolgen. Anschließend kann der SeCo-Algorithmus zur Klassifikation neuer Daten verwendet werden.

4.12 Default-Komponenten

Im SeCo-Framework gibt es einige Default-Komponenten, die nicht explizit in der XML-Beschreibung angegeben werden müssen, da sie in vielen Fällen gleich sind. Tabelle 4.5 gibt eine Übersicht über diese Komponenten. Diese Komponenten befinden sich alle im *seco.learners.components* package.

Tabelle 4.5: Default-Komponenten des SeCo-Framework

Name der Komponente	**Beschreibung**
DefaultSelector	Dieser Candidateselector wählt alle Kandidatenregeln aus, die daraufhin verfeinert werden.
DefaultRuleEvaluator	Hier wird die LaPlace Heuristik zur Bewertung von Regeln verwendet.
DefaultRuleInitializer	Wählt als initiale Regel eine leere Regel, die keine Bedingungen enthält und im Regelkopf die zu erlernende Klasse enthält.
DefaultRuleStop	Dieses Kriterium prüft, ob eine gegebene Regel besser ist als eine leere Regel, die im Regelkopf die aktuell zu erlernende Klasse beinhaltet. Bewertet werden dabei die Regeln mit der LaPlace Heuristik, sofern nicht eine andere angegeben ist.

4.13 Anwendung im Client-/Server-Betrieb

Im Rahmen dieser Arbeit wurden sehr viele Experimente mit der SeCo-Factory durchgeführt. Allein für die abschließenden Tests wurde die Software über 2000 mal auf Lernprobleme angewandt. Um diese Experimente besser verarbeiten zu können, wurden ergänzende Skripte angelegt, welche einen Client-/Server-Betrieb möglich machten. Der Betriebsablauf ist in Abbildung 4.5 zu sehen. Dabei werden auf der Workstation Rechenaufträge in XML formuliert, die einerseits die Beschreibung des anzuwendenden Algorithmus und andererseits eine Liste der Lernpobleme beinhaltet. Der Auftrag wird dann an den Server geschickt

[27] Der DefaultConstructor ist der Constructor, der keine Argumente bekommt. Man könnte dies jedoch erweitern, um auch andere Konstruktoren aufrufen zu können.
[28] Zuordnung eines spezielleren Datentyps/Klasse.

und in einer Warteschlange abgelegt in der die Aufträge in ihrer Reihenfolge abgearbeitet werden. Für jeden Auftrag wird dabei zunächst der Algorithmus mit der SeCo-Factory erzeugt und anschließend auf die Liste der Lernprobleme angewandt. Die Konsolenausgabe, die alle Ergebnisse und auch evtl. Fehlermeldungen erfaßt, wird als Ergebnisdatei auf dem Server abgelegt und ist auf Abruf verfügbar. Auf der Workstation werden diese Ergebnisdateien nach den interessierenden Informationen durchsucht, die dann in SQL formuliert werden. Diese SQL Statements können dann in die Datenbank gegeben werden. Das hierfür verwendete Datenbankdesign ist im Anhang B auf Seite 86 zu finden.

Durch die Speicherung in einer Datenbank konnten die Ergebnisse mittels Datenbankabfragen angefordert und verglichen werden.

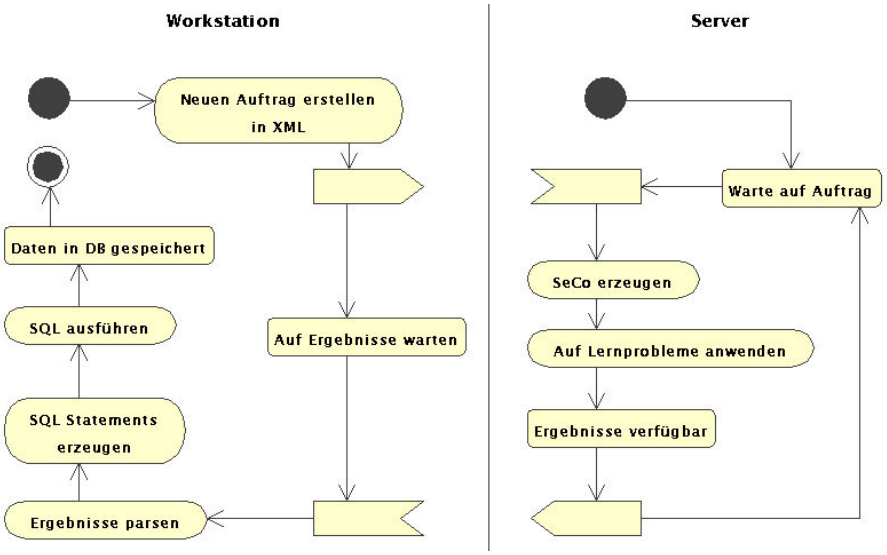

Abbildung 4.5: Betriebsablauf mit Server

5 Realisierung des BEXA mit SeCo-Factory

Dieser Abschnitt beschreibt die Umsetzung des BEXA aus Abschnitt 3.5 mittels der SeCo-Factory aus Abschnitt 4. Diese Implementation wird dann in den folgenden Abschnitten für empirische Untersuchungen verwendet. Der BEXA-Algorithmus wurde so umgesetzt, wie in Abschnitt 3.5 beschrieben, bis auf die *irredundancy restriction*, die nicht so großen Einfluß auf die Zahl der erzeugten Spezialisierungen hat (siehe Abbildung 3.2). Eine XML-Beschreibung des Algorithmus für die Factory befindet sich in Quelltext 4.4.

5.1 Die Hypothesensprache

Die Java-Klassen der Hypothesensprache befinden sich im *seco.models* package. Sie bilden die Konzepte der Regeln, Regelmengen und Bedingungen ab. Die Regeln werden dargestellt wie in Formel 2.9 und bestehen aus einem Regelkörper (Konjunktionen von Bedingungen) und dem Regelkopf (Klassenzuweisung). Die einzelnen Bedingungen sind Attributtests (siehe Definition 2.5), die auf Gleichheit (z.B. $A_1 = w_{1,1}$ mit $w_{1,1} \in \mathcal{D}_A$) oder auf Ungleichheit (z.B. $A_1 \neq w_{1,1}$) prüfen können.

Ob eine Bedingung, die in *seco.models.NominalCondition* implementiert ist, auf Gleichheit oder Ungleichheit prüft, wird durch den dort genannten *cmpmode* beschrieben. Es ist ein boolscher Wert, der *true* oder *false* annehmen kann. Im Fall *true* wird auf Gleichheit geprüft, ansonsten auf Ungleichheit.

5.2 Umsetzung der SeCo-Komponenten

Im folgenden werden die Details hinsichtlich der Implementation der verwendeten SeCo-Komponenten erläutert. Die Umsetzung orientiert sich an den Funktionen, die in Abschnitt 3.6 definiert wurden. Darum konzentriert sich diese Sektion auf die etwas umfangreicheren SeCo-Komponenten **RuleFilter** und **RuleRefiner**.

5.2.1 RuleFilter

Der RuleFilter setzt sich hier zusammen aus dem ChiSquareFilter und dem BeamWidthFilter. Um beide Filter als RuleFilter einsetzen zu können, wurde der MultiRuleFilter erzeugt, dem beliebig viele RuleFilter hinzugefügt werden können, die dann in der gegebenen Reihenfolge angewendet werden. Der entsprechende Konfigurationsabschnitt ist in Quelltext 5.1 zu sehen.

Quelltext 5.1 Konfigurationsabschnitt für MultiRuleFilter

```
<secomp interface="rulefilter" classname="MultiRuleFilter">
  <jobject classname="ChiSquareFilter" setter="filter">
    <property name="threshold" value="0.995"/>
  </jobject>
  <jobject classname="BeamWidthFilter" setter="filter">
    <property name="beamwidth" value="3"/>
  </jobject>
</secomp>
```

ChiSquareFilter Dieser Filter implementiert den Stoppe-Wachstum-Test, der bereits in Abschnitt 3.5.3 beschrieben wurde.

BeamWidthFilter Dieser Filter definiert eine Ordnungsrelation $\geq_{eval}$ auf den Kandidatenregeln:

$$x, y \in \mathcal{T}.x \geq_{eval} y \Leftrightarrow E(x) \geq E(y) \tag{5.1}$$

wobei $\mathcal{T}$ die Theorie und $E(x)$ die Bewertungsfunktion[29] ist. Anhand dieser Relation werden die Regeln in $\mathcal{T}$ sortiert. Die ersten *bw* Regeln werden in $\mathcal{T}$ behalten, die anderen werden herausgefiltert, wobei *bw* die vorgegebene beamwidth ist, die über die Property *beamwidth* variiert werden kann.

5.2.2 RuleRefiner

Der RuleRefiner des BEXA, wie er in 3.5.2 beschrieben ist, ist in der Java-Klasse *seco.learners.bexa.BexaRefinerTopDown* implementiert, mit Ausnahme der irredundancy restriction, wie bereits erwähnt. Ausgangspunkt des Refiners ist die Methode *refineRule()*, die eine Kandidatenregel und die Beispielmenge als Argumente erhält. Der Pseudo-Code ist in Quelltext 5.2. Das Ergebnis ist

Quelltext 5.2 Pseudo-Code der refineRule() Methode

```
procedure refineRule(CandidateRule, examples)
{
    ruleset := ∅
    usable := createUsableConditions()
    usable' := filterUsableConditions(CandidateRule,
                usable, examples)
    foreach usable' u
    {
        ruleset.add(CandidateRule.specialize(u))
    }
    return ruleset
}
```

eine Menge von Kandidatenregeln, die spezieller als die gegebene *CandidateRule* sind. Die Methode *specialize*, welche eine neue Kandidatenregel erzeugt, ist trivial, denn wie in Abschnitt 3.5.1 erklärt ist, kann man die Regelverfeinerung des BEXA so darstellen, daß Bedingungen der Form $A_1 \neq w_{1,1}$ hinzugefügt werden, was im Falle der oben beschriebenen Implementation dem Hinzufügen einer *Condition* mit *cmpmode=false* entspricht.

Aufwändiger hingegen ist die Berechnung der Menge *usable'*, die alle Bedingungen enthält, mit denen verfeinert werden soll. In Abbildung 5.1 ist dieser Vorgang als Aktivitätsdiagramm dargestellt. Der Vorgang beginnt damit, daß geprüft wird, ob die Menge aller Bedingungen bereits erzeugt wurde. Dieser Caching-Mechanismus spart viel Rechenzeit, da die Menge aller möglichen Bedingungen (**usable**) nur von den Trainingsdaten abhängt. Es reicht also, diese

[29] Die Bewertungsfunktion wird im RuleEvaluator gesetzt und ist beim BEXA auf die LaPlace- Heuristik eingestellt.

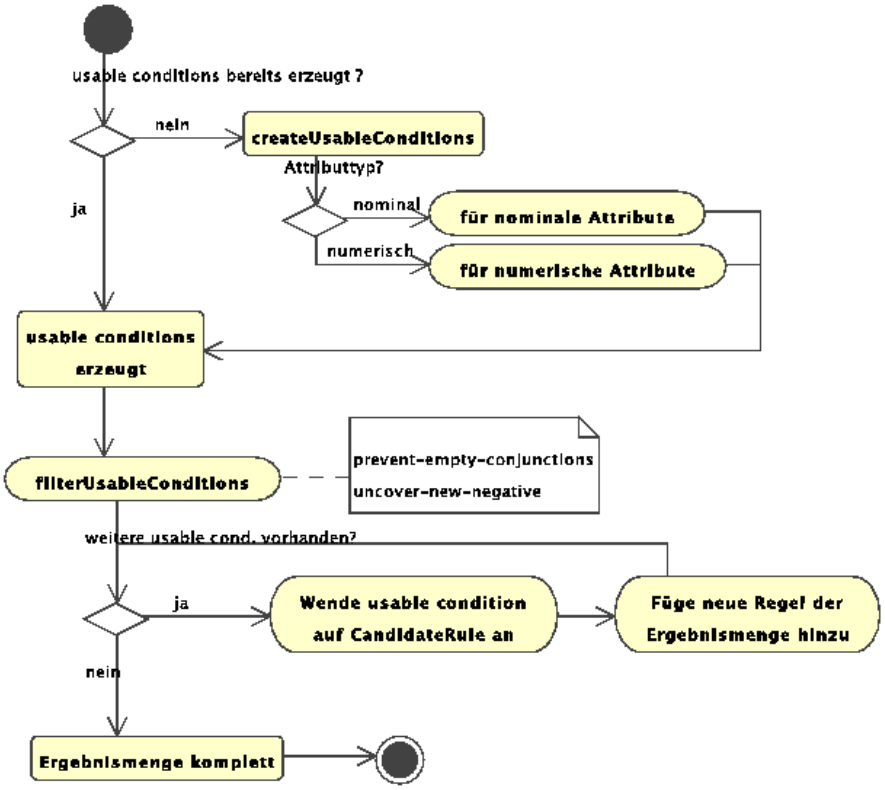

Abbildung 5.1: Aktivitätsdiagramm des BexaRefinerTopDown

Menge nur einmal zu erzeugen. Da diese Menge jedoch zu groß ist und im folgenden Ablauf zu viel Rechenzeit verursachen, wird eine gefilterte Version **usable'** für jede Kandidatenregel berechnet, mittels *filterUsableConditions()*.

createUsableConditions Diese Methode erzeugt für alle Attribute Bedingungen, wobei dieser Vorgang für nominale und numerische Attribute unterschiedlich abläuft. Für **nominale** Attribute A_{nom} wird die Menge $\{A_{nom} \neq w_{nom,i} | \forall w_{nom,i} \in \mathcal{D}_{A_{nom}}\}$ erzeugt. Bei **numerischen** Attributen A_{num} ist dieses Vorgehen i.A. nicht möglich, da die Menge der möglichen Werte $\mathcal{D}_{A_{num}}$ zu mächtig ist. Wie in Abschnitt 3.5.1 erwähnt, verwendet man darum Vergleichsoperatoren $\leq$ und $>$, um die Zugehörigkeit zu einem Intervall zu überprüfen. Dies führt zu der Frage, wie die Intervallgrenzen zu bestimmen sind. Hierzu werden alle Beispiele der Trainingsmenge nach dem numerischen Attribut A_{num} sortiert. Anschließend werden die Werte ermittelt, bei denen ein Wechsel der Klasse stattfindet.

Zur Veranschaulichung betrachten wir das Lernproblem aus Abschnitt 2.1, deren Beispiele für die Intervallbestimmung des Attributs *Temperatur* in Tabelle 5.1 sortiert werden. Nun durchsucht man die Liste von oben nach unten bis man einen Klassenwechsel feststellt. Bei folgenden Temparaturänderungen erfolgt ein Klassenwechsel: von 23 auf 23, von 23 auf 27, von 27 auf 27 und von 28 auf 29. Für jedes dieser Zahlenpaare bildet man das arithmetische Mittel (23, 25, 27, 28.5), welches dann gleichzeitig eine Intervallgrenze darstellt. Für diese Grenzen werden dann Bedingungen generiert:

- Temperatur ≤ 23, Temperatur > 23,
- Temperatur ≤ 25, Temperatur > 25,
- Temperatur ≤ 27, Temperatur > 27,
- Temperatur ≤ 28.5, Temperatur > 28.5.

filterUsableConditions Diese Methode muß für jede Kandidatenregel r erneut angewendet werden, da hier zunächst die Mengen[30] $X_P(r)$ und $X_N(r)$ gebildet werden. Als notwendiges Kriterium, daß eine Bedingung aus *usable* für die betrachtete Kandidatenregel r hilfreich sein kann, wird die *prevent-empty-conjunctions-restriction* und die *uncover-new-negative-restriction* gefordert, wie sie bereits in Abschnitt 3.5.2 beschrieben wurden. Geprüft werden diese Restriktionen, indem ein positives Beispiel gesucht wird, welches $X_P(r')$ angehört und ein negatives Beispiel $i \in X_N(r)$, für welches $i \notin X_N(r')$ gilt, wobei r' die Regel ist, die sich ergibt, wenn man zu r noch die jeweilige Bedingung aus *usable* hinzufügt.

Als Beispiel betrachten wir beim bekannten Lernproblem aus Abschnitt 2.1 die Regel $r = Temperatur > 25$. Wir finden in *usable* die Bedingung $sonnig = nein$ und bilden die Regel $r' = Temperatur > 25$ *und* $sonnig = nein$. Es gilt:

[30] Zur Erinnerung: Es handelt sich um die Menge der positiven bzw. negativen Beispiele, die von der Regel r abgedeckt werden.

Tabelle 5.1: Sortierung der Beispiele zur Intervallbestimmung

#	**Aussicht**	**Herbst**	**Temperatur**	**Klasse**
3	regen	ja	16	-
1	sonnig	ja	17	-
2	bewölkt	nein	18	-
4	sonnig	ja	22	-
8	regen	ja	23	-
11	bewölkt	nein	23	+
13	regen	nein	23	+
9	regen	nein	27	-
12	sonnig	nein	27	+
10	sonnig	ja	28	+
5	sonnig	nein	29	-
6	bewölkt	nein	30	-
7	bewölkt	nein	35	-

$$
\begin{aligned}
X_P(r) &= \{10, 12\} \\
X_N(r) &= \{5, 6, 7, 9\} \\
X_P(r') &= \{12\} \\
X_N(r') &= \{5\}
\end{aligned}
$$

Wir stellen fest, daß es ein positives Beispiel in $X_P(r')$ gibt, nämlich #12 (*prevent-empty-conjunctions*), und daß es ein negatives Beispiel in $X_N(r)$ gibt, welches in $X_N(r')$ nicht mehr vorkommt, nämlich #6, #7 und #9 (*uncover-new-negative*).

Würden wir beispielsweise die Bedingung *bewoelkt* = *nein* betrachten, so wäre $X_P(r') = \emptyset$ und somit die *prevent-empty-conjunctions-restriction* verletzt. Die Bedingung käme dadurch dann nicht mehr in Betracht.

6 Evaluierung der Implementation

Ausgehend von der Implementation der SeCo-Factory und des BEXA können nun Experimente gemacht und Ergebnisse analysiert werden. Die allgemeine Problematik hierbei ist, daß man einerseits nie ausschließen kann, daß Programme fehlerfrei sind, andererseits sind Meßergebnisse nur dann vergleichbar und interpretierbar, wenn die Art und Weise, wie Versuche durchgeführt werden, sowie die Meßmethoden bekannt sind. Dieser Abschnitt möchte darum nicht einfach Zahlen präsentieren, sondern insbesondere die Vorgehensweise beschreiben und durch Vergleich mit anderen Implementationen anderer Lernalgorithmen eine Plausibilität der Ergebnisse zeigen.

6.1 Die Lernprobleme

Die verwendeten Lernprobleme stammen aus dem UCI-Repository [8]. Die konvertierten Daten im ARFF-Format wurden von [1] bezogen. Tabelle 6.2 gibt eine Übersicht über die verwendeten Lernprobleme und deren Eigenschaften. Die Erklärung der Tabellenspalten befindet sich in Tabelle 6.1.

Tabelle 6.1: Legende zu Eigenschaften von Lernproblemen

Spalte	Beschreibung
Problem	Name des Lernproblems.
Attr.	Gesamtanzahl der Attribute.
Nom.	Anzahl der nominalen Attribute.
Num.	Anzahl der numerischen Attribute.
Kl.	Gesamtanzahl der Klassen.
Beisp.	Anzahl der vorhandenen Beispiele.
größte Kl.	Prozentsatz des Vorkommens der größten Klasse.

6.2 Was wird gemessen ? Wie wird gemessen ?

In Abschnitt 2.4 wurden bereits einige Kriterien vorgestellt wie die Korrektheit und die Größe der Regelmenge. Im folgenden wird erläutert, wie diese Werte gemessen werden.

6.2.1 Korrektheit

Messung mit Trainingsdaten Eine Möglichkeit die Korrektheit zu messen ist, eine Überprüfung mittels der Trainingsdaten durchzuführen und zu sehen, ob alle Beispiele, mit denen gelernt wurde, auch richtig von den gelernten Regeln klassifiziert werden. Viele Algorithmen entsprechen der Annahme, daß die Korrektheit hier bei 100% liegen muß, da die Regeln beliebig differenziert werden können durch Hinzufügen weiterer Bedingungen bis keine Falschaussagen mehr gemacht werden. Bei konsequenter Umsetzung dieser Strategie ist jedoch häufig zu beobachten, daß die gelernten Regeln zu sehr an die Trainingsdaten angepaßt sind und somit viel spezieller sind als das eigentliche Konzept ist. Dieser Effekt verstärkt sich insbesondere dann, wenn ein Teil der Daten fehlerhaft ist. Man spricht dann von **Overfitting**. Der BEXA gehört zu denjenigen Algorithmen, die einem Overfitting durch vorzeitigen Abbruch der Regelverfeinerung vorbeu-

Tabelle 6.2: Eigenschaften der Lernprobleme des UCI

Problem	Attr.	Nom.	Num.	Kl.	Beisp.	größte Kl.
anneal	38	32	6	6	898	76
anneal.ORIG	38	32	6	6	898	76
audiology	69	69	0	24	226	25
autos	25	10	15	7	205	33
balance-scale	4	0	4	3	625	46
breast-cancer	9	9	0	2	286	70
breast-w	9	9	0	2	699	66
colic	22	15	7	2	368	63
colic.ORIG	27	20	7	2	368	66
credit-a	15	9	6	2	692	55
credit-g	20	13	7	2	1000	70
diabetes	8	0	8	2	768	65
glass	9	0	9	7	214	36
heart-c	13	7	6	5	303	54
heart-h	13	7	6	5	294	64
heart-statlog	13	0	13	2	270	56
hepatitis	19	13	6	2	155	79
hypothyroid	29	22	7	4	3772	92
ionosphere	34	0	34	2	352	64
iris	4	0	4	3	150	33
kr-vs-kp	36	36	0	2	3196	52
labor	16	8	8	2	57	65
lymph	18	15	3	4	148	55
mushroom	22	22	0	2	8124	52
primary-tumor	17	17	0	22	339	25
sick	29	22	7	2	3772	94
sonar	60	0	60	2	208	53
soybean	35	35	0	19	684	13
splice	61	61	0	3	3190	52
tic-tac-toe	9	9	0	2	958	65
vehicle	18	0	18	4	846	26
vote	16	16	0	2	435	61
vowel	13	3	10	11	990	09
zoo	17	17	0	7	101	41

gen, wenn die weitere Verfeinerung nur relativ wenige Regeln abdecken würde. Dadurch gibt es im folgenden Meßergebnisse, die nicht bei 100% liegen, aber nicht auf fehlerhafte Algorithmen zurückzuführen sind.

Messung mit Testdaten Es wird eine *10-fold stratified cross validation* durchgeführt wie sie in der Weka-Bibliothek implementiert ist. Dabei wird die Trainingsmenge in 10 gleich große Mengen aufgeteilt. 9 von 10 Mengen werden zum Lernen verwendet, während die Zehnte zum Messen des Lernerfolgs verwendet wird. Dies wird 10 mal durchgeführt, so daß jede Menge einmal als Testmenge verwendet worden ist. Schließlich werden die 10 Ergebnisse mit Hilfe des arithmethischen Mittels zu einem zusammengefaßt.

6.2.2 Größe der Regelmenge

Außer der Korrektheit wird auch die gelernte Regelmenge ausgegeben. Zu beachten ist, daß bei den hier präsentierten Experimenten, für **jede** Klasse Regeln gelernt werden, obwohl es möglich ist, die Regeln für die letzte Klasse wegzulassen und durch eine Defaultregel zu ersetzen, wie das bei vielen Algorithmen üblich ist. Bzgl. der Anwendung der Regelmenge auf ein Beispiel wird zwar wie bei einer *Entscheidungsliste*[31] vorgegangen, jedoch dieser Sachverhalt nicht zur Optimierung beim Lernen ausgenutzt. Dadurch behält jede Regel für sich ihre Gültigkeit ohne, daß geprüft werden müßte, ob eine vorhergehende Regel zutreffend ist. Diese Art der Implementation soll es ermöglichen, die Qualität der Regelmengen genauer untersuchen zu können. Die Größe der Regelmengen wurde hier als Anzahl der Regeln in der erlernten Regelmenge definiert.

6.2.3 Anzahl der Bedingungen

Da Regeln hinsichtlich ihrer Länge sehr unterschiedlich sein können, wird auch die Anzahl der Bedingungen gezählt. Es handelt sich hierbei einfach um die Summe aller Bedingungen, die in der Regelmenge auftauchen.

6.3 Komplexität

Auf einen detaillierten Vergleich der Komplexität der Algorithmen wurde an dieser Stelle verzichtet. Um aber eine Vorstellung zu bekommen, wie sich das Laufzeitverhalten unterscheidet, wurde die reelle Rechenzeit gemessen. Dabei wurde für jedes Problem separat die Zeit erfaßt, jedoch für die gesamte Dauer des Java-Prozeß. Somit beinhalten die Zeitintervalle auch die *cross-validation*. Durchgeführt wurden die Messungen mit dem Unix-Programm **time**.

6.4 Vergleich von BEXA mit JRip

Um die Ergebnisse auf Plausibilität zu prüfen, wurde der Ripper [5] Algorithmus zum Vergleich herangezogen. Verwendet wurde die Implementation JRip aus der Weka-Bibliothek [17].

Bei den Vergleichen bzgl. der Komplexität, Anzahl der Bedingungen und Größe der Regelmenge wäre ein direkter Vergleich zwischen JRip und BEXA[32]

[31] Bei der Entscheidungsliste werden die Regeln gemäß ihrer Reihenfolge angewandt und die erste Regel, die zutrifft, bestimmt die Klasse des Beispiels.

[32] So wie er hier implementiert wurde.

nicht möglich, da JRip, im Gegensatz zu BEXA, für die Standardklasse[33] keine Regeln lernt. Darum wurde in der SeCo-Factory eine weitere Property eingeführt namens *skipdefaultclass*, die eine Regelerzeugung für die Standardklasse verhindert. Dadurch wird ein Vergleich von Laufzeit und Regelgröße möglich.

6.4.1 Korrektheit

Die Tabelle 6.3 zeigt die Ergebnisse des Vergleichs sortiert nach Lernproblemen. Die Spalten *BEXA* und *JRip* zeigen jeweils die Korrektheit der Algorithmen auf der Trainingsmenge. Die Spalte *diff* zeigt die Differenz der Korrektheit. Die Spalten *LRS* und χ^2 geben die Grenzwerte an, die für den jeweiligen Versuch verwendet wurden. Der BEXA wurde mit allen Grenzwertkombinationen von 0, 0.7, 0.9, 0.99 und 0.995 evaluiert und die jeweils besten Ergebnisse sind in der Tabelle gelistet. Von diesen 34 Versuchen hat der BEXA 15 mal gewonnen, und 17 mal verloren.

[33] Die Standardklasse ist normalerweise diejenige, die in der Trainingsmenge am häufigsten vorkommt. Mit ihrer Hilfe wird eine Standardregel erstellt, die bedingungslos ein Beispiel mit der Standardklasse klassifiziert, sofern keine andere Regel zutreffend ist.

Tabelle 6.3: Vergleich der Korrektheit von BEXA disj. und JRip

problem	BEXA	JRip	diff	LRS	χ^2
anneal	98.44	98.33	0.11	0.995	0
anneal.ORIG	95.55	95.32	0.23	0.995	0.7
audiology	69.91	73.89	-3.98	0	0
autos	76.1	73.17	2.93	0	0.9
balance-scale	84.48	80.8	3.68	0.7	0.7
breast-cancer	74.83	70.98	3.85	0.7	0
breast-w	95.71	95.42	0.29	0	0.7
colic	78.8	84.24	-5.44	0.99	0.995
colic.ORIG	81.25	82.61	-1.36	0.9	0.99
credit-a	84.06	85.8	-1.74	0.995	0
credit-g	72	71.7	0.30	0.995	0.9
diabetes	71.22	76.04	-4.82	0.99	0.99
glass	64.49	68.69	-4.20	0.7	0
heart-c	76.57	81.52	-4.95	0.995	0.99
heart-h	77.21	78.91	-1.70	0.995	0.9
heart-statlog	79.63	78.89	0.74	0.9	0
hepatitis	83.87	78.06	5.81	0.995	0
hypothyroid	97.14	99.34	-2.20	0.995	0.7
ionosphere	94.02	89.74	4.28	0.9	0.9
iris	93.33	94.67	-1.34	0.7	0.9
kr-vs-kp	99.19	99.19	0.00	0.9	0.7
labor	89.47	77.19	12.28	0.9	0.995
lymph	86.49	77.7	8.79	0.9	0.995
mushroom	100	100	0.00	0.9	0.99
primary-tumor	38.94	39.23	-0.29	0.9	0.9
sick	97.75	98.22	-0.47	0.995	0.9
sonar	69.71	73.08	-3.37	0.99	0
soybean	89.31	92.53	-3.22	0.7	0.995
splice	89.53	94.45	-4.92	0.9	0
tic-tac-toe	98.54	97.81	0.73	0	0.7
vehicle	60.28	68.56	-8.28	0.995	0.9
vote	96.55	95.4	1.15	0.995	0.995
vowel	51.52	69.09	-17.57	0.99	0.7
zoo	92.08	87.13	4.95	0	0.995
Durchschnitt	82.59	83.17	-0.58		
Gewonnen	15	17			

Tabelle 6.4: Vergleich von BEXA disj. und JRip bei fester Einstellung

problem	BEXA	JRip	diff	lrs	chi
anneal	98.33	98.33	0.00	0.995	0.9
anneal.ORIG	94.88	95.32	-0.44	0.995	0.9
audiology	61.95	73.89	-11.94	0.995	0.9
autos	62.44	73.17	-10.73	0.995	0.9
balance-scale	80.96	80.8	0.16	0.995	0.9
breast-cancer	73.08	70.98	2.10	0.995	0.9
breast-w	94.56	95.42	-0.86	0.995	0.9
colic	76.63	84.24	-7.61	0.995	0.9
colic.ORIG	78.53	82.61	-4.08	0.995	0.9
credit-a	82.75	85.8	-3.05	0.995	0.9
credit-g	72	71.7	0.30	0.995	0.9
diabetes	69.53	76.04	-6.51	0.995	0.9
glass	61.68	68.69	-7.01	0.995	0.9
heart-c	74.59	81.52	-6.93	0.995	0.9
heart-h	77.21	78.91	-1.70	0.995	0.9
heart-statlog	74.81	78.89	-4.08	0.995	0.9
hepatitis	82.58	78.06	4.52	0.995	0.9
hypothyroid	96.87	99.34	-2.47	0.995	0.9
ionosphere	91.74	89.74	2.00	0.995	0.9
iris	92	94.67	-2.67	0.995	0.9
kr-vs-kp	98.47	99.19	-0.72	0.995	0.9
labor	85.96	77.19	8.77	0.995	0.9
lymph	77.7	77.7	0.00	0.995	0.9
mushroom	100	100	0.00	0.995	0.9
primary-tumor	36.58	39.23	-2.65	0.995	0.9
sick	97.75	98.22	-0.47	0.995	0.9
sonar	63.46	73.08	-9.62	0.995	0.9
soybean	87.41	92.53	-5.12	0.995	0.9
splice	85.86	94.45	-8.59	0.995	0.9
tic-tac-toe	82.88	97.81	-14.93	0.995	0.9
vehicle	60.28	68.56	-8.28	0.995	0.9
vote	95.63	95.4	0.23	0.995	0.9
vowel	49.09	69.09	-20.00	0.995	0.9
zoo	84.16	87.13	-2.97	0.995	0.9
Durchschnitt	79.48	83.17	-3.69		
Gewonnen	7	24			

Man sollte aus diesen Ergebnissen jedoch nicht folgern, daß die Algorithmen etwa gleich gut sind, da die Einstellungen des JRip nicht variiert wurden. Um einen besseren Vergleich zu bekommen, wurde in Tabelle 6.4 eine Einstellung des BEXA beibehalten, die im Durchschnitt die besten Erfolge erzielt hat[34], nämlich lrs=0.995 und chi=0.9. Das Ergebnis sieht dann für BEXA deutlich schlechter aus: 7 mal gewonnen, 3 mal unentschieden und 24 mal verloren.

6.4.2 Größe der Regelmenge

In Tabelle 6.5 ist ein Vergleich der Regelmengen zu sehen. Es sind für jedes Problem jeweils die Anzahl der erzeugten Regeln und die Summe der Bedingungen angegeben. Zusätzlich wird zum Vergleich das Verhältnis $\frac{BEXA}{JRip}$ berechnet, das die Anzahl der Regeln (bzw. Bedingungen) von BEXA durch die Anzahl der Regeln (bzw. Bedingungen) von JRip dividiert. Ein Wert kleiner 1 bedeutet, daß BEXA weniger Regeln (bzw. Bedingungen) produziert hat als JRip. Entsprechend gilt bei einem Wert größer 1 das Gegenteil. Am Ende der Tabelle werden die Regeln und Bedingungen summiert, während bzgl. der Verhältnisse das geometrische Mittel berechnet wird. Man kann erkennen, daß von wenigen Ausnahmen abgesehen, JRip kompaktere Konzeptbeschreibungen liefert.

6.4.3 Komplexität

Die Tabelle 6.6 gibt eine Übersicht über die Laufzeit der Algorithmen. Das Format der Zeiten ist hh:mm:ss.msec, was jeweils für Stunde, Minuten, Sekunden und Millisekunden steht. Da die Optimierung hinsichtlich der Laufzeit bei der Implementation des BEXA vernachlässigt wurde, war zu erwarten, daß der BEXA hier sehr schlecht abschneidet, dennoch gibt es ein paar wenige Lernprobleme bei denen er schneller als JRip war. Auffällig sind die Probleme *hypothyroid*, *sick*, *splice* und *vowel* bei denen die Laufzeit extrem erhöht ist.

6.4.4 Zusammenfassung

Der Vergleich hat gezeigt, daß JRip hinsichtlich der Korrektheit dem BEXA überlegen ist und kompaktere Regelmengen produziert. Erklären läßt sich dies mit dem PostProcessing, das im JRip implementiert ist, jedoch nicht in der vorliegenden BEXA Implementation.

[34] Beste Erfolge wurde daran gemessen, wieviele Lernprobleme mit der jeweiligen Einstellung für sich gewonnen werden konnten. Also Maximierung der Anzahl positiver *diff*-Werte. Der Betrag von *diff* wurde außer acht gelassen.

Tabelle 6.5: Vergleich der Regelmenge, BEXA disj. und JRip

	Regeln			Bedingungen		
Problem	BEXA	JRip	$\frac{BEXA}{JRip}$	BEXA	JRip	$\frac{BEXA}{JRip}$
anneal	15	7	2.14	35	8	4.37
anneal.ORIG	23	14	1.64	70	37	1.89
audiology	40	15	2.66	80	27	2.96
autos	21	13	1.61	34	25	1.36
balance-scale	26	12	2.16	81	39	2.07
breast-cancer	2	3	0.66	8	4	2.00
breast-w	18	6	3.00	41	9	4.55
colic	14	4	3.50	40	6	6.66
colic.ORIG	13	5	2.60	28	9	3.11
credit-a	32	4	8.00	90	7	12.85
credit-g	47	3	15.66	174	5	34.80
diabetes	44	4	11.00	136	9	15.11
glass	14	8	1.75	33	18	1.83
heart-c	13	4	3.25	32	6	5.33
heart-h	15	3	5.00	35	4	8.75
heart-statlog	8	5	1.60	19	8	2.37
hepatitis	4	4	1.00	8	5	1.60
hypothyroid	50	5	10.00	156	11	14.18
ionosphere	10	3	3.33	12	2	6.00
iris	5	4	1.25	10	3	3.33
kr-vs-kp	29	16	1.81	98	44	2.22
labor	4	4	1.00	6	4	1.50
lymph	6	6	1.00	14	8	1.75
mushroom	7	9	0.77	18	12	1.50
primary-tumor	17	7	2.42	75	18	4.16
sick	39	4	9.75	124	10	12.40
sonar	21	5	4.20	29	9	3.22
soybean	44	28	1.57	196	52	3.76
splice	16	14	1.14	216	55	3.92
tic-tac-toe	2	9	0.22	6	24	0.25
vote	8	4	2.00	29	6	4.83
vowel	196	51	3.84	468	145	3.22
zoo	7	6	1.16	12	6	2.00
Summe/Geo. Mittel	810	2413	2.28	289	635	3.54

Tabelle 6.6: Vergleich der Laufzeit, BEXA disj. und JRip (hh:mm:ss.msec)

Problem	BEXA	JRip	Differenz
anneal	00:00:32.320	00:00:11.110	00:00:21.210
anneal.ORIG	00:00:40.400	00:00:28.280	00:00:12.120
audiology	00:00:10.100	00:00:10.100	00:00:00.000
autos	00:00:25.250	00:00:06.060	00:00:19.190
balance-scale	00:00:14.140	00:00:10.100	00:00:04.040
breast-cancer	00:00:01.010	00:00:01.010	00:00:00.000
breast-w	00:00:11.110	00:00:05.050	00:00:06.060
colic	00:00:35.350	00:00:04.040	00:00:31.310
colic.ORIG	00:01:08.080	00:00:08.080	00:01:00.000
credit-a	00:03:31.310	00:00:06.060	00:03:25.250
credit-g	00:13:07.070	00:00:12.120	00:12:54.950
diabetes	00:14:39.390	00:00:07.070	00:14:32.320
glass	00:00:30.300	00:00:04.040	00:00:26.260
heart-c	00:00:23.230	00:00:03.030	00:00:20.200
heart-h	00:00:16.160	00:00:02.020	00:00:14.140
heart-statlog	00:00:18.180	00:00:03.030	00:00:15.150
hepatitis	00:00:03.030	00:00:01.010	00:00:02.020
hypothyroid	00:40:19.190	00:00:50.500	00:39:28.690
ionosphere	00:00:29.290	00:00:10.100	00:00:19.190
iris	00:00:01.010	00:00:01.010	00:00:00.000
kr-vs-kp	00:01:01.010	00:01:24.240	-00:00:23.230
labor	00:00:01.010	00:00:01.010	00:00:00.000
lymph	00:00:02.020	00:00:01.010	00:00:01.010
mushroom	00:01:12.120	00:01:25.250	-00:00:13.130
primary-tumor	00:00:06.060	00:00:06.060	00:00:00.000
sick	00:32:45.450	00:01:07.070	00:31:38.380
sonar	00:02:20.200	00:00:10.100	00:02:10.100
soybean	00:00:49.490	00:00:14.140	00:00:35.350
splice	03:48:07.000	00:08:23.230	03:39:43.770
tic-tac-toe	00:00:02.020	00:00:12.120	-00:00:10.100
vote	00:00:02.020	00:00:02.020	00:00:00.000
vowel	01:08:52.000	00:01:20.200	01:07:31.800
zoo	00:00:02.020	00:00:01.010	00:00:01.010
Summe	06:52:58.340	00:17:31.280	

7 Fallstudie: Hypothesensprache

Wie in Abschnitt 3.5 gezeigt wurde, besteht eine wesentliche Innovation des BEXA-Algorithmus darin, im Verfeinerungsprozeß einzelne Attributwerte auszuschließen, anstatt einen konkreten Wert zur Bedingung zu machen. Man spricht dabei auch von disjunktiven Regeln der Form

$$(A = a \vee A = b \vee A = c) \wedge (B = x \vee B = y) \rightarrow Class = Cl_1, \tag{7.1}$$

während klassische Regellerner wie der CN2 ausschließlich Konjunktionen im Regelkörper verwenden.

Die Frage ist nun, welche Vor- bzw. Nachteile die modifizierte Hypothesensprache tatsächlich bringt. In [16] wurde zwar bereits ein Vergleich zwischen BEXA und anderen Regellernen gemacht, jedoch wurden in der dortigen (sowie auch in dieser) BEXA-Implementation noch andere Verbesserungen eingebaut, die einen großen Einfluß auf die empirischen Ergebnisse gehabt haben. In dieser Fallstudie sollen darum nicht zwei grundsätzlich verschiedene Algorithmen verglichen werden, sondern die vorhandene Implementation soll lediglich im interessierenden Punkt verändert werden.

7.1 Modifizierte BEXA-Varianten

Die modifizierten BEXA-Algorithmen, die in dieser Studie der Ausgangsimplementation gegenübergestellt werden, unterscheiden sich nur an einer Stelle vom Original, und zwar bei der Verfeinerung von Bedingungen für nominale Attribute. Die Ursprungsform des BEXA wird im folgenden als **BEXA disj.** bezeichnet, wobei disj. für die disjunktiven Regeln steht, die hierbei gebildet werden.

7.1.1 BEXA konj.

Wie bereits in Sektion 3.5.2 beschrieben wurde, findet die Verfeinerung einer Regel durch Hinzufügen von Bedingungen der Form $A \neq a$ statt. In der modifizierten Variante **BEXA konj.**[35] werden diese Spezialisierungen nicht erzeugt. Statt dessen wird der $=$ Operator verwendet. Es entstehen also Bedingungen der Form $A = a$. Zu diesem Zweck wurde die Refiner-Komponente des BEXA um eine Konfigurationsmöglichkeit, nämlich die Property *nominal.cmpmode* erweitert, was in der XML-Beschreibung die Ergänzung aus Quelltext 7.1 möglich macht, um zur modifizierten Variante BEXA konj.zu kommen.

Quelltext 7.1 Modifikation der XML Beschreibung für BEXA konj.

```
<secomp interface="rulerefiner" classname="BexaRefinerTopDown"
package="seco.learners.bexa">
    <property name="nominal.cmpmode" value="equal"/>
</secomp>
```

[35] konj. steht für die konjunktiven Regeln, die bei dieser Variante gebildet werden.

7.1.2 BEXA mixed

Die Variante BEXA mixed verwendet zur Erzeugung der Spezialisierungen beide Komparatoren, also = und $\neq$, womit sie doppelt so viele Spezialisierungen erzeugt, wie Variante BEXA konj. und BEXA disj.Es ist klar, daß ohne weitere Einschränkungen die Variante BEXA mixed bzgl. der Performanz nicht mit den beiden anderen mithalten kann, aufgrund der höheren Komplexität. Bei dieser Variante interessiert vielmehr, inwiefern, durch Erhöhung der Ausdrucksstärke der Hypothesensprache, die Qualität der Regeln verbessert werden kann. In Quelltext 7.2 ist der veränderte Teil der XML-Beschreibung des RuleRefiner für die Variante BEXA mixed. Die property *nominal.cmpmode* nimmt hier den Wert *both* an.

Quelltext 7.2 Modifikation der XML Beschreibung für BEXA mixed

```
<secomp interface="rulerefiner" classname="BexaRefinerTopDown"
package="seco.learners.bexa">
    <property name="nominal.cmpmode" value="both"/>
</secomp>
```

7.2 Weitere Qualitätskriterien

Durch die Veränderung der Hypothesensprache ergibt sich die Frage, wie die Anzahl der Regelbedingungen zu messen ist, denn wie in Abschnitt 3.5.1 bereits gezeigt wurde, kann man bei nominalen Attributen die Darstellung der Bedingungen ändern. So kann man statt $A \neq b$ auch $A = a \vee A = c$ schreiben mit $\mathcal{D}_A := \{a, b, c\}$. Während sich die bereits vorgestellte Meßmethode, aus Abschnitt 6.2.3 für die Anzahl der Bedingungen, an der gerade verwendeten Darstellungsform orientiert[36], sollen weitere Meßmethoden auch andere Darstellungsformen berücksichtigen, die im folgenden erläutert werden.

7.2.1 Referenzierte Attribute

Die Anzahl der referenzierten Attribute[37] ergibt sich anhand der verschiedenen Attribute, die im Regelkörper getestet werden. Durch Minimierung dieses Meßwertes werden Regelmengen bevorzugt, mit denen man die Klassifikation anhand weniger Attribute durchführen kann.

7.2.2 Normalisierte Regellänge

Bei der normalisierten Regellänge[38] wird davon ausgegangen, daß die Anzahl der Attributtests, die zur Klassifikation durchgeführt werden müssen, maßgebend für die Qualität der Regelmenge ist. Darum wird für jede Regel die minimale Anzahl an Attributtests berechnet, die bei günstigster Formulierung nötig sind. Die günstigste Formulierung wird für jedes Attribut A bestimmt, das in dem

[36] Im Fall BEXA disj. also die Darstellung mit $\neq$ und im Fall BEXA konj. mit =.
[37] Im folgenden auch mit *referred_atts* abgekürzt.
[38] Im folgenden auch mit *norm_length* abgekürzt.

Regelkörper vorkommt, indem (im Fall BEXA disj.) die Mächtigkeit der Subdomäne[39] $|\mathcal{S}_A(r,\neq)|$ mit der Mächtigkeit der inversen Subdomäne $|\overline{\mathcal{S}_A(r,\neq)}|$ verglichen wird. Die jeweils kleineren Werte bzgl. jedes Attributs A werden summiert. Die Summe entspricht der **normalisierten Regellänge**. Für BEXA konj. gilt die Rechnung analog mit dem $=$-Komparator, während bei BEXA mixed für das jeweilige Attribut geprüft werden muß, welcher Komparator verwendet wird. Die Berechnung läßt sich damit in folgender Definition zusammenfassen:

Definition 7.1 (Normalisierte Regellänge $norm\_length$**)** *Die normalisierte Regellänge berechnet sich für BEXA disj. mit*

$$norm\_length(r) := \sum_{\forall A \in r} min\{|\mathcal{S}_A(r,\neq)|, |\overline{\mathcal{S}_A(r,\neq)}|\}$$

Für BEXA konj.:

$$norm\_length(r) := \sum_{\forall A \in r} min\{|\mathcal{S}_A(r,=)|, |\overline{\mathcal{S}_A(r,=)}|\}$$

Für BEXA mixed:

$$norm\_length(r) := \sum_{\forall A \in r} \begin{cases} min\{|\mathcal{S}_A(r,\neq)|, |\overline{\mathcal{S}_A(r,\neq)}|\} & falls\ \mathcal{S}_A(r,\neq) \neq \emptyset \\ min\{|\mathcal{S}_A(r,=)|, |\overline{\mathcal{S}_A(r,=)}|\} & falls\ \mathcal{S}_A(r,=) \neq \emptyset \\ 0 & sonst \end{cases}$$

Für numerische Attribute wird jede Bedingung einzeln gezählt und zu $norm\_length$ *addiert.*[40]

7.2.3 Ein Beispiel

An dieser Stelle soll die Bestimmung der Meßwerte anhand des Lernproblems aus Abschnitt 2.1 veranschaulicht werden. Wir nehmen an, daß folgende Regelmenge gelernt worden ist:

$$Aussicht \neq sonnig \wedge Herbst \neq ja \wedge Aussicht \neq regen \rightarrow class = + \tag{7.2}$$

$$Aussicht \neq bewoelkt \rightarrow class = - \tag{7.3}$$

Wir ermitteln nun die jeweiligen Werte:

Anzahl der Regeln Offensichtlich 2 Regeln (r_1 und r_2).

Anzahl der Bedingungen In der ersten Regel sind es 3 und in der zweiten Regel eine Bedingung. Macht in der Summe also 4 Bedingungen.

Referenzierte Attribute In der ersten Regel werden die Attribute *Aussicht* und *Herbst* erwähnt und in der zweiten Regel lediglich *Aussicht*. Da die Anzahl der referenzierten Attribute für jede Regel ermittelt wird (also 2 und 1) und anschließend summiert wird, ist das Ergebnis 3.

[39] siehe Abschnitt 3.5.1

[40] Wegen der Übersichtlichkeit wurde dieser Fall nicht in den Formeln berücksichtigt.

Normalisierte Regellänge Wir wenden die Formel aus der Definition an:

$$\begin{aligned} norm\_length(r_1) &:= min\{|\mathcal{S}_{Aussicht}(r_1, \neq)|, |\overline{\mathcal{S}_{Aussicht}(r_1, \neq)}|\} \\ &+ min\{|\mathcal{S}_{Herbst}(r_1, \neq)|, |\overline{\mathcal{S}_{Herbst}(r_1, \neq)}|\} \\ &= min\{|\{sonnig, regen\}|, |\{bewoelkt\}|\} \\ &+ min\{|\{ja\}|, |\{nein\}|\} \\ &= min\{2, 1\} + min\{1, 1\} \\ &= 1 + 1 = 2 \end{aligned} \tag{7.4}$$

$$\begin{aligned} norm\_length(r_2) &:= min\{|\mathcal{S}_{Aussicht}(r_2, \neq)|, |\overline{\mathcal{S}_{Aussicht}(r_2, \neq)}|\} \\ &= min\{|\{bewoelkt\}|, |\{sonnig, regen\}|\} \\ &= min\{1, 2\} = 1 \end{aligned} \tag{7.5}$$

Die normalisierte Regellänge für die gesamte Regelmenge ergibt sich wiederum aus der Summe und ist somit 3.

7.3 Vergleich von BEXA disj. mit BEXA konj.

Für BEXA konj. wurden ebenfalls alle Kombinationen der Grenzwerte 0, 0.7, 0.9, 0.99, 0.995 für LRS und χ^2 getestet. Die besten Ergebnisse von BEXA disj. aus Tabelle 6.3 sind in Tabelle 7.1 den jeweils besten Ergebnissen von BEXA konj. gegenübergestellt. Die LRS und χ^2 Spalte geben die jeweils optimalen Einstellungen an. Im Vergleich zu Abschnitt 6 wurden einige Lernprobleme weggelassen, um Rechenzeit für die Experimente einzusparen.

Tabelle 7.1: Korrektheit von BEXA disj. und BEXA konj.

	BEXA disj.			BEXA konj.			
Problem	Korr.	LRS	χ^2	Korr.	LRS	χ^2	Diff.
anneal	98.44	0.995	0	99.78	0.9	0.99	-1.34
anneal.ORIG	95.55	0.995	0.7	95.43	0.995	0	0.12
audiology	69.91	0	0	69.03	0.7	0.7	0.88
autos	76.1	0	0.9	80.98	0	0.99	-4.88
breast-cancer	74.83	0.7	0	73.43	0.99	0	1.40
breast-w	95.71	0	0.7	95.99	0	0.7	-0.28
colic	78.8	0.99	0.995	79.62	0.99	0.99	-0.82
colic.ORIG	81.25	0.9	0.99	74.46	0.99	0.99	6.79
credit-a	84.06	0.995	0	84.78	0.99	0.9	-0.72
credit-g	72	0.99	0.995	74.4	0.99	0.9	-2.40
heart-c	76.57	0.995	0.99	77.89	0.995	0.995	-1.32
heart-h	77.21	0.995	0.9	79.25	0.995	0.995	-2.04
hepatitis	83.87	0.995	0.995	85.16	0.99	0	-1.29
hypothyroid	97.14	0.995	0.7	97.38	0.995	0.9	-0.24
kr-vs-kp	99.19	0.9	0.7	99.31	0.99	0.7	-0.12
labor	89.47	0.9	0.995	92.98	0	0.7	-3.51
lymph	86.49	0.9	0.995	81.76	0.9	0.995	4.73
mushroom	100	0.995	0.995	100	0	0	0.00
primary-tumor	38.94	0.9	0	39.23	0.9	0.995	-0.29
sick	97.75	0.995	0.9	97.48	0.995	0.7	0.27
soybean	89.31	0.7	0.995	90.63	0	0.7	-1.32
splice	89.53	0.9	0	52.92	0.7	0.995	36.61
tic-tac-toe	98.54	0	0.7	98.33	0.99	0	0.21
vote	96.55	0.995	0.995	96.09	0.99	0.99	0.46
vowel	51.52	0.99	0.7	57.17	0.7	0.995	-5.65
zoo	92.08	0.7	0.995	87.13	0.9	0	4.95
Durchschnitt	84.26			83.10			1.16
Gewonnen	10			15			

Bei dem Vergleich hat BEXA disj. bei 15 von 26 Lernproblemen verloren und scheint somit die schlechtere Wahl zu sein. Besonders auffällig ist jedoch das Problem *splice*, bei welchem BEXA disj. um 36.61% besser ist als BEXA konj.In Tabelle 7.2 ist für die selben Durchläufe die entsprechende Anzahl an Regeln und Bedingungen gegenübergestellt, die in den entstandenen Regelmengen vorkommen. Obwohl im Durchschnitt von BEXA disj. weniger Regeln und Bedingungen erzeugt wurden, kann man bzgl. dieses Aspekts nicht generell sagen, daß BEXA disj. hier im Vorteil ist, da es nur etwa die Hälfte der Lernprobleme

Tabelle 7.2: Regelmenge von BEXA disj. und BEXA konj.

	Anzahl der Regeln			Anzahl der Bedingungen		
Problem	disj.	konj.	$\frac{disj.}{konj.}$	disj.	konj.	$\frac{disj.}{konj}$
anneal	27	31	0.87	62	51	1.21
anneal.ORIG	35	46	0.76	96	132	0.72
audiology	94	94	1.00	261	204	1.27
autos	69	65	1.06	141	122	1.15
breast-cancer	19	20	0.95	124	56	2.21
breast-w	37	39	0.94	95	102	0.93
colic	31	48	0.64	84	118	0.71
colic.ORIG	62	193	0.32	206	198	1.04
credit-a	64	72	0.88	203	209	0.97
credit-g	125	139	0.89	432	525	0.82
heart-c	28	25	1.12	71	60	1.18
heart-h	29	34	0.85	72	80	0.90
hepatitis	15	15	1.00	22	22	1.00
hypothyroid	61	52	1.17	177	142	1.24
kr-vs-kp	73	57	1.28	301	225	1.33
labor	10	12	0.83	18	18	1.00
lymph	18	15	1.20	55	35	1.57
mushroom	19	22	0.86	47	28	1.67
primary-tumor	63	77	0.81	315	294	1.07
sick	62	62	1.00	171	164	1.04
soybean	87	95	0.91	381	313	1.21
splice	55	2185	0.02	521	2202	0.23
tic-tac-toe	45	15	3.00	233	45	5.17
vote	17	14	1.21	49	39	1.25
vowel	250	324	0.77	616	743	0.82
zoo	11	26	0.42	22	29	0.75
Mittel[41]	54.07	145.26	0.78	183.65	236.76	1.08

ist, bei denen die Anzahl der Regeln bzw. Bedingungen geringer ist. Das Gleiche gilt für die Anzahl der referenzierten Attribute und der normalisierten Regellänge, die in Tabelle 7.3 gelistet ist. Auch hier sprechen die Durchschnittswerte für BEXA disj., der aber bei etwa der Hälfte der Lernprobleme komplexere Regelmengen erzeugt. In allen drei Tabellen schneidet BEXA disj. bzgl. des Problems *splice* deutlich besser ab.

[41] Für Regeln und Bedingungen wurde das arithmetische Mittel gebildet. Für $\frac{disj.}{konj.}$ jedoch das geometrische Mittel.

Tabelle 7.3: Normalisierte Regellänge von BEXA disj. und BEXA konj.

	Referenzierte Attribute			Normalisierte Regellänge		
Problem	disj.	konj.	$\frac{disj.}{konj.}$	disj.	konj.	$\frac{disj.}{konj.}$
anneal	52	48	1.08	62	51	1.21
anneal.ORIG	75	111	0.67	96	125	0.76
audiology	247	204	1.21	250	204	1.22
autos	128	120	1.06	141	122	1.15
breast-cancer	98	56	1.75	117	56	2.08
breast-w	90	97	0.92	95	102	0.93
colic	78	114	0.68	84	118	0.71
colic.ORIG	151	198	0.76	206	198	1.04
credit-a	167	185	0.90	203	209	0.97
credit-g	315	419	0.75	432	525	0.82
heart-c	69	58	1.18	71	60	1.18
heart-h	66	74	0.89	71	80	0.88
hepatitis	22	22	1.00	22	22	1.00
hypothyroid	149	119	1.25	177	142	1.24
kr-vs-kp	298	225	1.32	298	225	1.32
labor	18	18	1.00	18	18	1.00
lymph	50	35	1.42	53	35	1.51
mushroom	45	28	1.60	46	28	1.64
primary-tumor	308	294	1.04	308	294	1.04
sick	149	147	1.01	171	164	1.04
soybean	363	313	1.15	369	313	1.17
splice	367	2202	0.16	468	2202	0.21
tic-tac-toe	190	45	4.22	190	45	4.22
vote	49	39	1.25	49	39	1.25
vowel	465	597	0.77	616	743	0.82
zoo	22	29	0.75	22	29	0.75
Mittel[42]	155.03	222.96	1.00	178.26	236.50	1.06

Betrachtet man nun die jeweils gelernte Regelmenge für *splice* wird deutlich, warum hier so große Unterschiede vorhanden sind. Hier ein Auszug der Regelmenge, die von BEXA konj. gelernt wurde[43]:

```
Class = EI :- attribute_50 = N.  [3|0] Val: 0.8
Class = EI :- Instance_name = HUMALBGC-DONOR-17044.  [2|0]
Val: 0.75
Class = EI :- Instance_name = HUMMYLCA-DONOR-2559.  [2|0]
Val: 0.75
```

Und hier ein Auszug der Regelmenge von BEXA disj.[44]:

```
Class = EI :- attribute_35 != C, attribute_35 != T,
```

[42] Auch hier wurde wieder für die Verhältnisse $\frac{disj.}{konj.}$ das geometrische anstatt dem arithmetischen Mittel gebildet.

[43] Aus Platzgründen sind hier nur die ersten drei Regeln notiert. Die Eigenschaften, die davon abgeleitet werden, sind jedoch auch für die restliche Regelmenge gültig.

[44] Auch hier wieder nur die ersten drei.

```
attribute_35 != A, attribute_32 != C, attribute_34 != T,
attribute_32 != G, attribute_32 != A, attribute_34 != G,
attribute_31 != T, attribute_53 != A, attribute_2 != G,
attribute_44 != A, attribute_17 != A.  [216|0] Val: 0.995

Class = EI :- attribute_32 != A, attribute_32 != C,
attribute_32 != G, attribute_31 != C, attribute_31 != A,
attribute_31 != T, attribute_18 != T, attribute_35 != C,
attribute_35 != T, attribute_35 != A, attribute_24 != C,
attribute_6 != T, attribute_41 != A.  [198|0] Val: 0.995

Class = EI :- attribute_32 != A, attribute_32 != C,
attribute_32 != G, attribute_31 != C, attribute_31 != T,
attribute_31 != A, attribute_22 != T, attribute_21 != T,
attribute_33 != G, attribute_33 != C, attribute_33 != T,
attribute_10 != G, attribute_60 != C.  [87|0] Val: 0.989
```

In dieser Notation kommt zuerst der Regelkopf, dann der Regelkörper. Schließlich folgt in eckigen Klammern die Anzahl abgedeckter, positiver Beispiele[45] und die Anzahl abgedeckter, negativer Beispiele[46]. *Val* gibt die Bewertung durch die Bewertungsfunktion wieder.

Es fällt auf, daß mit Forderungen nach bestimmten Attributwerten gar keine Abdeckung vieler positiver Beispiele möglich ist. Selbst bei Prüfung eines Attributs (siehe erster Auszug) ist $|X_P| \leq 3$, während im zweiten Fall bereits mit der ersten Regel 216 Beispiele abgedeckt werden. Das Problem *splice* besitzt also eine Eigenschaft, die im folgenden als **Ausschluß-Eigenschaft** bezeichnet werden soll.

7.4 Ausschluß-Eigenschaft

Woran man bestimmen kann, ob ein Lernproblem die Ausschluß-Eigenschaft besitzt oder nicht, ist zunächst unklar. Die Beobachtung des vorangegangenen Abschnitts zeigt, daß es eine Eigenschaft gibt, die einen großen Einfluß auf den Lernerfolg der jeweils verwendeten Hypothesensprache hat. Obwohl das Problem *splice* diese Eigenschaft sehr deutlich aufzeigt, kann nicht ausgeschlossen werden, daß es sich hier um einen fließenden Übergang zwischen "Problem hat Ausschluß-Eigenschaft" und "Problem hat nicht die Ausschluß-Eigenschaft" handelt. Für die weitere Betrachtung soll folgende Definition eine Hilfe sein.

Definition 7.2 (Ausschluß-Eigenschaft) *Ein Problem P besitzt die Ausschluß-Eigenschaft, wenn beim Vergleich von Regeln R_{IN}, die* ***ohne*** *Ausschluß von Attributwerten formuliert sind, mit Regeln R_{EX}, die* ***mit*** *Ausschluß von Attributwerten formuliert sind, gilt:*
$|X_P(r_{IN})| << |X_P(r_{EX})|$ *und* $|X_N(r_{IN})| \approx |X_N(r_{EX})|$, *wobei r_{IN} die beste Regel aus R_{IN} und r_{EX} die beste Regel aus R_{EX} ist.*[47]

Um sich die Definition 7.2 zu verdeutlichen, kann man folgendes Beispiel betrachten: Sei P ein Lernproblem mit einem Attribut A und Domäne $\mathcal{D}_A := \{\omega_1..\omega_5\}$. Es gibt die Klassen a und b. Es sei folgende Trainingsmenge gegeben:

[45] Auch als X_P bezeichnet.
[46] Auch als X_N bezeichnet.
[47] $<<$ bedeutet hier "wesentlich kleiner als".

Tabelle 7.4: Trainingsmenge für kleines Ausschluß-Eigenschaft Beispiel

#	A	class
1	ω_1	a
2	ω_2	b
3	ω_3	b
4	ω_4	b
5	ω_5	b

Wollen wir nun Regeln für Klasse b aufstellen und wählen eine Hypothesensprache wie in BEXA konj., so benötigen wir 4 Regeln, die jeweils ein Beispiel abdecken. Das ist dann die Regelmenge R_{IN}, bei welcher für die beste Regel[48] r_{IN} gilt: $|X_P(r_{IN})| = 1$, weil nur ein positives Beispiel abgedeckt wird. Verwendet man jedoch die Hypothesensprache von BEXA disj., so reicht eine Regel r_{EX} aus mit $A \neq \omega_1$. Es gilt also $X_N(r_{IN}) = 0 = X_N(r_{EX})$ und $|X_P(r_{IN})| = 1 << 4 = |X_P(r_{EX})|$, womit laut Definition 7.2 P die Ausschluß-Eigenschaft besitzt.

7.5 Vergleich von BEXA konj. mit BEXA mixed

Der BEXA mixed versucht, die positiven Eigenschaften von beiden Hypothesensprachen zu vereinen, indem er Bedingungen mit dem =- und dem $\neq$-Komparator erzeugt. Als Konsequenz wird hierfür natürlich eine wesentlich höhere Laufzeit benötigt, da im Verfeinerungsprozeß erst mal alle möglichen Bedingungen erzeugt werden. Diese Menge ist nun jeweils doppelt so groß, wie im BEXA disj.. Dieser Nachteil wird hier zunächst in Kauf genommen, darf aber bei der Bewertung des Algorithmus nicht außer acht gelassen werden. Da insbesondere das Problem *splice* sehr viel Rechenzeit konsumiert, wurde es nicht für alle Kombinationen der Grenzwerte berechnet. Aufgrund seiner Besonderheit wird es in Abschnitt 7.5.2 gesondert behandelt.

[48] In diesem Fall sind alle Regeln gleich gut. Wir wählen also eine beliebige.

7.5.1 Vergleich mit allen Grenzwertkombinationen

Für beide Algorithmen wurden hier die verschiedenen Grenzwertkombinationen getestet und die jeweils besten Ergebnisse bzgl. der Korrektheit verwendet. In Tabelle 7.5 ist der Vergleich der Korrektheit zu sehen. Sowohl der Durchschnitt als auch Häufigkeit, mit der ein Algorithmus den anderen übertrifft sind hier nicht signifikant.

Tabelle 7.5: Korrektheit von BEXA konj. und BEXA mixed

	BEXA konj.			BEXA mixed			
Problem	Korr.	LRS	χ^2	Korr.	LRS	χ^2	Diff
anneal	99.78	0.9	0.99	99.78	0	0	0.00
anneal.ORIG	95.43	0.995	0	95.99	0.995	0.99	-0.56
audiology	69.03	0.7	0.7	67.7	0	0	1.33
autos	80.98	0	0.99	80	0.7	0	0.98
breast-cancer	73.43	0.99	0	73.43	0.99	0	0.00
breast-w	95.99	0	0.7	96.14	0.7	0	-0.15
colic	79.62	0.99	0.99	79.35	0.99	0.9	0.27
colic.ORIG	74.46	0.99	0.99	72.55	0.9	0.99	1.91
credit-a	84.78	0.99	0.9	84.64	0.99	0.99	0.14
credit-g	74.4	0.99	0.9	74.3	0.995	0.995	0.10
heart-c	77.89	0.995	0.995	77.23	0.995	0.995	0.66
heart-h	79.25	0.995	0.995	77.89	0.995	0.995	1.36
hepatitis	85.16	0.99	0	84.52	0.995	0.995	0.64
hypothyroid	97.38	0.995	0.9	97.53	0.995	0	-0.15
kr-vs-kp	99.31	0.99	0.7	99.22	0.99	0	0.09
labor	92.98	0	0.7	92.98	0.7	0	0.00
lymph	81.76	0.9	0.995	80.41	0.9	0.995	1.35
mushroom	100	0	0	100	0.995	0.995	0.00
primary-tumor	39.23	0.9	0.995	39.82	0.9	0.7	-0.59
sick	97.48	0.995	0.7	97.67	0.995	0	-0.19
soybean	90.63	0	0.7	90.92	0	0	-0.29
tic-tac-toe	98.33	0.99	0	98.23	0.9	0.99	0.10
vote	96.09	0.99	0.99	95.86	0.99	0.7	0.23
vowel	57.17	0.7	0.995	57.78	0	0.99	-0.61
zoo	87.13	0.9	0	86.14	0.9	0.99	0.99
Durchschnitt	84.31			84.00			0.31
Gewonnen	14			7			

In Tabelle 7.6 sind wieder die Anzahl der Regeln und Bedingungen gegenübergestellt. Hier gibt es bei einigen Problemen größere Abweichungen, aber von einem allgemeinen Vorteil des BEXA mixed kann man hier auch nicht sprechen, da es weniger als die Hälfte der Probleme sind, bei denen er im Vorteil ist. Für die Ergebnisse in Tabelle 7.7 gilt entsprechendes.

Tabelle 7.6: Regelmenge von BEXA konj. und BEXA mixed

	Anzahl der Regeln			Anzahl der Bedingungen		
Problem	konj.	mixed	$\frac{konj.}{mixed}$	konj.	mixed	$\frac{konj.}{mixed}$
anneal	31	31	1.00	51	52	0.98
anneal.ORIG	46	42	1.09	132	110	1.20
audiology	94	99	0.94	204	217	0.94
autos	65	63	1.03	122	118	1.03
breast-cancer	20	13	1.53	56	30	1.86
breast-w	39	37	1.05	102	99	1.03
colic	48	47	1.02	118	109	1.08
colic.ORIG	193	192	1.00	198	211	0.93
credit-a	72	88	0.81	209	246	0.84
credit-g	139	120	1.15	525	427	1.22
heart-c	25	27	0.92	60	68	0.88
heart-h	34	35	0.97	80	80	1.00
hepatitis	15	15	1.00	22	22	1.00
hypothyroid	52	52	1.00	142	145	0.97
kr-vs-kp	57	70	0.81	225	291	0.77
labor	12	10	1.20	18	15	1.20
lymph	15	21	0.71	35	50	0.70
mushroom	22	22	1.00	28	28	1.00
primary-tumor	77	75	1.02	294	330	0.89
sick	62	54	1.14	164	139	1.17
soybean	95	90	1.05	313	301	1.03
tic-tac-toe	15	22	0.68	45	68	0.66
vote	14	15	0.93	39	44	0.88
vowel	324	325	0.99	743	754	0.98
zoo	26	26	1.00	29	29	1.00
Mittel[49]	63.68	63.64	0.98	158.16	159.32	0.98

[49] Für Regeln und Bedingungen das arithmetische Mittel. Für die Verhältnisse $\frac{konj.}{mixed}$ das geometrische Mittel.

Tabelle 7.7: Normalisierte Regellänge von BEXA konj. und BEXA mixed

	Referenzierte Attribute			Normalisierte Regellänge		
Problem	konj.	mixed	$\frac{konj.}{mixed}$	konj.	mixed	$\frac{konj.}{mixed}$
anneal	48	48	1.00	51	52	0.98
anneal.ORIG	111	95	1.16	125	103	1.21
audiology	204	217	0.94	204	217	0.94
autos	120	116	1.03	122	118	1.03
breast-cancer	56	30	1.86	56	30	1.86
breast-w	97	92	1.05	102	99	1.03
colic	114	108	1.05	118	109	1.08
colic.ORIG	198	211	0.93	198	211	0.93
credit-a	185	223	0.82	209	246	0.84
credit-g	419	364	1.15	525	427	1.22
heart-c	58	66	0.87	60	68	0.88
heart-h	74	73	1.01	80	80	1.00
hepatitis	22	22	1.00	22	22	1.00
hypothyroid	119	124	0.95	142	145	0.97
kr-vs-kp	225	291	0.77	225	291	0.77
labor	18	15	1.20	18	15	1.20
lymph	35	50	0.70	35	50	0.70
mushroom	28	28	1.00	28	28	1.00
primary-tumor	294	330	0.89	294	330	0.89
sick	147	121	1.21	164	139	1.17
soybean	313	301	1.03	313	301	1.03
tic-tac-toe	45	68	0.66	45	68	0.66
vote	39	44	0.88	39	44	0.88
vowel	597	631	0.94	743	754	0.98
zoo	29	29	1.00	29	29	1.00
Mittel[50]	143.80	147.88	0.98	157.88	159.04	0.98

7.5.2 Spezialfall: splice

Die Erwartung, daß sich BEXA mixed den $\neq$-Komparator zu Nutze machen würde blieb unerfüllt. Stattdessen wurde für die Regelmenge nur der $=$-Komparator verwendet, mit dem selben Ergebnis wie bei BEXA konj.Die Erklärung für dieses Phänomen ist einfach: Im Rahmen des Verfeinerungsprozesses werden die Bedingungen einzeln hinzugefügt und die resultierenden Regeln werden nach jedem Schritt neu bewertet. Wie in den beiden Auszügen aus den Regelmengen in Abschnitt 7.3 zu erkennen ist, bestehen die Regeln von BEXA disj. aus relativ vielen Bedingungen, während die Regeln von BEXA konj. nur eine Bedingung haben. Da die einzelnen Bedingungen bereits 2 positive und kein negatives Beispiel abdecken, bekommen die Regeln mit der *LaPlaceAccuracy* eine Bewertung von 0.75 und mehr, während eine einzelne Bedingung mit dem $\neq$-Komparator neben vielen positiven auch noch sehr viele negative Regeln abdeckt und darum

[50] Für Regeln und Bedingungen das arithmetische Mittel. Für die Verhältnisse $\frac{konj.}{mixed}$ das geometrische Mittel.

niedriger bewertet ist. Aufgrund der hohen Zahl von Kandidatenregeln, kann das Problem nicht mit der *beamwidth* (siehe Abschnitt 3.5) kompensiert werden. Für den konkreten Fall könnte hier also eine andere Suchheuristik nützlich sein.

7.6 Weighted Relative Accuracy

Zum Vergleich wurden die Messungen noch einmal mit einer anderen Heuristik durchgeführt, der Weighted Relative Accuracy aus Tabelle 4.2. Verwendet wurden diesmal alle Grenzwertkombinationen aus 0, 0.9, 0.995 für das Stopping-Criterion, sowie den Stop-Growth-Test. Die Beam-Width betrug diesmal nur 1. Gegenübergestellt wurden die jeweils korrektesten Regelmengen.

7.6.1 Vergleich von BEXA disj. mit BEXA konj.

In Tabelle 7.8 ist ein Vergleich zwischen BEXA disj. und BEXA konj. hinsichtlich der Korrektheit.

Tabelle 7.8: Korrektheit von BEXA disj. und BEXA konj.(WRAcc)

	BEXA disj.			BEXA konj.			
Problem	Korr.	LRS	χ^2	Korr.	LRS	χ^2	Diff.
anneal	97.55	0.9	0	97.22	0	0	0.33
anneal.ORIG	88.42	0.995	0	81.96	0.995	0	6.46
audiology	80.53	0	0	82.3	0.995	0	-1.77
autos	69.76	0.9	0	64.39	0	0	5.37
breast-cancer	71.33	0.995	0.995	72.03	0.995	0	-0.70
breast-w	94.56	0.995	0	94.56	0.9	0	0.00
colic	83.97	0.995	0.9	82.07	0.995	0	1.90
colic.ORIG	79.62	0.995	0.9	76.9	0.995	0	2.72
credit-a	80.87	0.995	0	80.87	0.995	0	0.00
credit-g	66.7	0.995	0	62.9	0.995	0	3.80
heart-c	80.53	0.995	0	72.28	0.995	0	8.25
heart-h	73.13	0.995	0.9	79.59	0.995	0.995	-6.46
hepatitis	78.71	0.995	0	78.06	0.995	0	0.65
hypothyroid	92.6	0.9	0.9	92.66	0.9	0.9	-0.06
kr-vs-kp	89.55	0.995	0	90.21	0	0	-0.66
labor	77.19	0.995	0	80.7	0.995	0	-3.51
lymph	78.38	0.995	0.9	79.05	0.995	0	-0.67
mushroom	99.93	0.995	0	96.45	0.9	0	3.48
primary-tumor	38.64	0.995	0	41.3	0.995	0	-2.66
sick	97.69	0.9	0	97.75	0	0.9	-0.06
soybean	90.19	0.995	0	72.62	0	0	17.57
splice	91.16	0.995	0	89.44	0.995	0	1.72
tic-tac-toe	69.94	0.995	0	56.78	0.9	0	13.16
vote	95.63	0.995	0	95.63	0.995	0	0.00
vowel	61.11	0	0	60.71	0	0	0.40
zoo	94.06	0	0	93.07	0	0	0.99
Durchschnitt	81.47			79.67			1.8
Gewonnen	13			10			

Diesmal schneidet BEXA disj. besser ab, da er bei 13 Problemen gewinnt und nur bei 10 verliert. Es fällt auf, daß bei den meisten Problemen die besten Ergebnisse ohne Stop-Growth-Test erzielt wurden. Desweiteren ist der Vorsprung bei **soybean** und **tic-tac-toe** besonders hoch.

Tabelle 7.9: Regelmenge von BEXA disj. und BEXA konj.(WRAcc)

	Anzahl der Regeln			Anzahl der Bedingungen		
Problem	disj.	konj.	$\frac{disj.}{konj.}$	disj.	konj.	$\frac{disj.}{konj}$
anneal	12	14	0.85	43	42	1.02
anneal.ORIG	10	16	0.62	27	34	0.79
audiology	28	23	1.21	127	73	1.73
autos	11	15	0.73	73	67	1.08
breast-cancer	2	6	0.33	5	10	0.50
breast-w	4	6	0.66	16	26	0.61
colic	5	8	0.62	9	17	0.52
colic.ORIG	9	127	0.07	13	134	0.09
credit-a	6	6	1.00	21	20	1.05
credit-g	11	11	1.00	97	50	1.94
heart-c	6	8	0.75	23	27	0.85
heart-h	5	8	0.62	13	12	1.08
hepatitis	10	10	1.00	19	19	1.00
hypothyroid	11	11	1.00	25	24	1.04
kr-vs-kp	7	8	0.87	35	42	0.83
labor	7	7	1.00	11	11	1.00
lymph	5	6	0.83	14	12	1.16
mushroom	4	6	0.66	25	8	3.12
primary-tumor	9	9	1.00	40	37	1.08
sick	10	12	0.83	34	34	1.00
soybean	21	23	0.91	83	63	1.31
splice	6	34	0.17	1015	44	23.06
tic-tac-toe	3	6	0.50	8	6	1.33
vote	3	3	1.00	3	3	1.00
vowel	29	29	1.00	233	223	1.04
zoo	7	7	1.00	22	19	1.15
Mittel[51]	9.26	16.11	0.68	78.23	40.65	1.07

In Tabelle 7.9 und 7.10 erhält man einen Überblick, wie groß die gelernten Regelmengen sind. Man erkennt, daß die auffälligen Probleme soybean und tic-tac-toe hier keine großen Abweichungen haben. Dafür sind es hier die Probleme **splice** und **colic.ORIG**, die hier sehr unterschiedliche Ergebnisse liefern.

Im Fall **splice** liegt dies daran, daß nach der Korrektheit maximiert wurde und ohne Stop-Growth-Test die besten Ergebnisse erzielt wurden. Verwendet man jedoch den Stop-Growth-Test, dann erreicht BEXA disj. immer noch eine Korrektheit von etwa 90,5 %, aber die erlernte Regelmenge beinhaltet nur noch 27 Bedingungen, die sogar kompakter als die von BEXA konj. ist, selbst dann, wenn man für BEXA konj. ebenfalls den Stop-Growth-Test verwendet.

[51] Für Regeln und Bedingungen wurde das arithmetische Mittel gebildet. Für $\frac{disj.}{konj.}$ jedoch das geometrische Mittel.

Im Fall **colic.ORIG** ist es BEXA konj. welcher die größeren Regelmengen erzeugt, was auch hier mit dem Stop-Growth-Test verbessert werden kann, aber mit einem deutlichen Verlust an Korrektheit erkauft werden muß. Statt ca. 76 % sind es dann etwa nur noch etwa 63 %. Auch hier ist der disjunktive Regellerner also im Vorteil.

Tabelle 7.10: Normalisierte Regellänge, BEXA disj. BEXA konj.(WRAcc)

	Referenzierte Attribute			Normalisierte Regellänge		
Problem	disj.	konj.	$\frac{disj.}{konj.}$	disj.	konj.	$\frac{disj.}{konj.}$
anneal	37	39	0.94	43	42	1.02
anneal.ORIG	22	32	0.68	27	34	0.79
audiology	118	73	1.61	124	73	1.69
autos	55	59	0.93	73	67	1.08
breast-cancer	5	10	0.50	5	10	0.50
breast-w	16	24	0.66	16	26	0.61
colic	8	16	0.50	8	17	0.47
colic.ORIG	13	133	0.09	13	134	0.09
credit-a	19	20	0.95	21	20	1.05
credit-g	73	41	1.78	94	50	1.88
heart-c	22	24	0.91	23	27	0.85
heart-h	12	12	1.00	12	12	1.00
hepatitis	17	17	1.00	19	19	1.00
hypothyroid	24	24	1.00	25	24	1.04
kr-vs-kp	35	42	0.83	35	42	0.83
labor	11	11	1.00	11	11	1.00
lymph	14	12	1.16	14	12	1.16
mushroom	16	8	2.00	25	8	3.12
primary-tumor	39	37	1.05	39	37	1.05
sick	28	31	0.90	33	34	0.97
soybean	71	63	1.12	77	63	1.22
splice	29	44	0.65	1011	44	22.97
tic-tac-toe	7	6	1.16	7	6	1.16
vote	3	3	1.00	3	3	1.00
vowel	179	171	1.04	233	223	1.04
zoo	21	19	1.10	22	19	1.15
Mittel[52]	34.38	37.34	0.88	77.42	40.65	1.05

[52] Auch hier wurde wieder für die Verhältnisse $\frac{disj.}{konj.}$ das geometrische anstatt dem arithmetischen Mittel gebildet.

7.6.2 Vergleich von BEXA konj. mit BEXA mixed

Nun soll ein Vergleich mit BEXA mixed erfolgen, um zu sehen, inwiefern die *Weighted Relative Accuracy* Heuristik in der Lage ist, die Vorteile beider Hypothesensprachen auszunutzen. In Tabelle 7.11 ist die Korrektheit für das jeweilige Lernproblem gelistet. Man sieht, daß BEXA mixed bei vielen Problemen eine etwas höhere Korrektheit erreicht. Bei **soybean** hat die Heuristik besonders gut funktioniert, indem sie die bessere Hypothesensprache gewählt hat, jedoch bei **tic-tac-toe** ist genau das Gegenteil der Fall[53].

Tabelle 7.11: Korrektheit von BEXA konj. und BEXA mixed(WRAcc)

	BEXA konj.			BEXA mixed			
Problem	Korr.	LRS	χ^2	Korr.	LRS	χ^2	Diff
anneal	97.22	0.9	0	98.66	0.9	0	-1.44
anneal.ORIG	81.96	0.995	0	88.31	0	0	-6.35
audiology	82.3	0.995	0	82.74	0.995	0	-0.44
autos	64.39	0	0	71.22	0.9	0	-6.83
breast-cancer	72.03	0.995	0	72.73	0.995	0.9	-0.70
breast-w	94.56	0.9	0	94.71	0	0	-0.15
colic	82.07	0.995	0	82.07	0.995	0	0.00
colic.ORIG	76.9	0.995	0	78.53	0.995	0.9	-1.63
credit-a	80.87	0.995	0	80.87	0.995	0.9	0.00
credit-g	62.9	0.995	0	66.4	0.995	0	-3.50
heart-c	72.28	0.995	0	76.57	0.995	0.9	-4.29
heart-h	79.59	0.995	0.995	74.49	0.995	0.995	5.10
hepatitis	78.06	0.995	0	79.35	0.995	0	-1.29
hypothyroid	92.66	0.9	0.9	92.58	0.9	0.9	0.08
kr-vs-kp	90.21	0.9	0	90.83	0	0	-0.62
labor	80.7	0.995	0	80.7	0.995	0.995	0.00
lymph	79.05	0.995	0	81.08	0.995	0.9	-2.03
mushroom	96.45	0.995	0	99.98	0	0	-3.53
primary-tumor	41.3	0.995	0	40.41	0.995	0	0.89
sick	97.75	0	0.9	97.75	0.9	0.9	0.00
soybean	72.62	0	0	91.07	0.995	0	-18.45
splice	89.44	0.995	0	91.25	0.995	0	-1.81
tic-tac-toe	56.78	0.995	0	55.95	0.9	0.9	0.83
vote	95.63	0.995	0	95.63	0.995	0.9	0.00
vowel	60.71	0	0	61.82	0.9	0	-1.11
zoo	93.07	0	0	93.07	0	0	0.00
Durchschnitt	79.67			81.49			-1.82
Gewonnen	4			16			

[53]Siehe Ergebnis des BEXA disj. mit 69,94 % aus Tabelle 7.8.

Die Tabellen 7.12 und 7.13 geben die Größe der Regelmengen wieder.

Tabelle 7.12: Regelmenge von BEXA konj. und BEXA mixed(WRAcc)

	Anzahl der Regeln			Anzahl der Bedingungen		
Problem	konj.	mixed	$\frac{konj.}{mixed}$	konj.	mixed	$\frac{konj.}{mixed}$
anneal	14	12	1.16	42	34	1.23
anneal.ORIG	16	13	1.23	34	33	1.03
audiology	23	23	1.00	73	77	0.94
autos	15	12	1.25	67	75	0.89
breast-cancer	6	2	3.00	10	5	2.00
breast-w	6	7	0.85	26	33	0.78
colic	8	8	1.00	17	20	0.85
colic.ORIG	127	135	0.94	134	140	0.95
credit-a	6	6	1.00	20	13	1.53
credit-g	11	14	0.78	50	90	0.55
heart-c	8	9	0.88	27	26	1.03
heart-h	8	7	1.14	12	13	0.92
hepatitis	10	10	1.00	19	19	1.00
hypothyroid	11	11	1.00	24	24	1.00
kr-vs-kp	8	7	1.14	42	35	1.20
labor	7	7	1.00	11	10	1.10
lymph	6	7	0.85	12	16	0.75
mushroom	6	6	1.00	8	23	0.34
primary-tumor	9	9	1.00	37	38	0.97
sick	12	10	1.20	34	28	1.21
soybean	23	22	1.04	63	70	0.90
splice	34	23	1.47	44	1335	0.03
tic-tac-toe	6	13	0.46	6	39	0.15
vote	3	3	1.00	3	3	1.00
vowel	29	29	1.00	223	231	0.96
zoo	7	7	1.00	19	22	0.86
Mittel[54]	16.11	15.84	1.03	40.65	94.30	0.77

Auffallend sind hier die Probleme **splice** und **tic-tac-toe**, bei denen der BEXA mixed eine wesentlich größere Regelmenge produziert hat. Ursache ist dabei der Stop-Growth-Test, der für maximale Korrektheit deaktiviert worden ist. Verwendet man den Test, so werden die Regelmengen deutlich kleiner, ohne viel Korrektheit zu verlieren.

[54] Für Regeln und Bedingungen das arithmetische Mittel. Für die Verhältnisse $\frac{konj.}{mixed}$ das geometrische Mittel.

Tabelle 7.13: Normalisierte Regellänge, BEXA konj. BEXA mixed(WRAcc)

	Referenzierte Attribute			Normalisierte Regellänge		
Problem	konj.	mixed	$\frac{konj.}{mixed}$	konj.	mixed	$\frac{konj.}{mixed}$
anneal	39	30	1.30	42	34	1.23
anneal.ORIG	32	28	1.14	34	32	1.06
audiology	73	76	0.96	73	77	0.94
autos	59	59	1.00	67	75	0.89
breast-cancer	10	5	2.00	10	5	2.00
breast-w	24	29	0.82	26	33	0.78
colic	16	18	0.88	17	20	0.85
colic.ORIG	133	140	0.95	134	140	0.95
credit-a	20	13	1.53	20	13	1.53
credit-g	41	74	0.55	50	90	0.55
heart-c	24	23	1.04	27	26	1.03
heart-h	12	13	0.92	12	13	0.92
hepatitis	17	18	0.94	19	19	1.00
hypothyroid	24	24	1.00	24	24	1.00
kr-vs-kp	42	35	1.20	42	35	1.20
labor	11	10	1.10	11	10	1.10
lymph	12	16	0.75	12	16	0.75
mushroom	8	17	0.47	8	23	0.34
primary-tumor	37	38	0.97	37	38	0.97
sick	31	24	1.29	34	27	1.25
soybean	63	64	0.98	63	70	0.90
splice	44	58	0.75	44	1335	0.03
tic-tac-toe	6	39	0.15	6	39	0.15
vote	3	3	1.00	3	3	1.00
vowel	171	176	0.97	223	231	0.96
zoo	19	20	0.95	19	22	0.86
Mittel[55]	37.34	40.38	0.91	40.65	94.23	0.77

[55]Für Regeln und Bedingungen das arithmetische Mittel. Für die Verhältnisse $\frac{konj.}{mixed}$ das geometrische Mittel.

8 Schlußwort

8.1 Zusammenfassung

Begonnen wurde mit einer formalen Beschreibung von **Lernproblemen** und einer Einführung in die Klasse der **Separate and Conquer Algorithmen** in Abschnitt 2. Es wurde erläutert, worin sich die verschiedenen Algorithmen unterscheiden und anhand welcher Kriterien sie verglichen werden. Dabei wurde die allgemeine Form eines Separate and Conquer Algorithmus halbformal beschrieben.

In Abschnitt 3.3 wurde einer der einfachsten und bekanntesten SeCo-Algorithmen (der **CN2**) beschrieben. Im Abschnitt 3.5 wurde dann der **BEXA** Algorithmus eingeführt und seine Besonderheiten bzgl. der Hypothesensprache und seines Verfeinerungsprozesses erläutert. Vor allem die Idee, Bedingungen durch Ausschluß von Attributwerten zu bilden[56], stand hierbei im Vordergrund, da es den Hauptunterschied zum CN2 ausmacht. Für beide Algorithmen wurde eine Abbildung auf die allgemine Form der Separate and Conquer Algorithmen durchgeführt.

Um eine modulare und übersichtliche Implementation der beiden Algorithmen zu bekommen, die sich auch zum empirischen Vergleich eignet, wurde in Abschnitt 4 das **SeCo-Framework** beschrieben. Hier wurde die allgemeine Formulierung eines SeCo-Algorithmus aus Abschnitt 2.5 auf Java-Klassen abgebildet, die mittels einer XML-Beschreibung konfigurierbar und austauschbar gehalten wurden. In dem Framework ebenfalls enthalten ist ein Modell zur Formulierung von Regeln und eine Menge von häufig verwendeten Suchheuristiken.

Im Abschnitt 5 wurde dann erklärt, wie der BEXA Algorithmus mittels des SeCo-Framework umgesetzt wurde. Anschließend wurde in Abschnitt 6 die **BEXA Implementation** mit einer unabhängigen Umsetzung des Ripper [5] aus der Weka-Bibliothek verglichen, um Sicherheit zu gewinnen, daß die BEXA Implementation fehlerfrei ist.[57] Es wurde bei den Versuchen festgestellt, daß die BEXA Implementation grundsätzlich funktioniert, aber schlechtere Ergebnisse als JRip liefert und weniger performant ist.

Anstatt die funktionierende Implementation weiter zu optimieren, konzentrierte man sich in der Fallstudie aus Abschnitt 7 auf den **Vergleich der Hypothesensprache**, indem die vorhandene BEXA Implementation nur in diesem Punkt verändert wurde. Dabei sind neue Bewertungskriterien eingeführt worden, um einen fairen Vergleich der gelernten Regelmengen machen zu können.

8.2 Schlußfolgerung

Die Verwendung des **SeCo-Frameworks** hat sich bei der wissenschaftlichen Arbeit bewährt, da

- durch den strukturierten Aufbau wenige Implementationsfehler auftraten, bzw. diese schnell gefunden werden konnten. Der Einsatz der Logging-Bibliothek log4j war dabei sehr hilfreich.
- ein Großteil des Quelltextes immer wieder verwendet werden kann.

[56] Was wir auch als disjunktives Regellernen bezeichnen.
[57] Ein hinreichendes Kriterium ist dies natürlich nicht.

- man einzelne Aspekte, wie die Hypothesensprache, isoliert betrachten kann, was bei völlig unterschiedlichen Implementationen von Algorithmen in der Regel nicht möglich ist, da sie sich in mehreren Punkten unterscheiden und Wirkungen nicht eindeutig auf Ursachen abgebildet werden können.
- aufgrund der XML-Beschreibung Eigenschaften eines Algorithmus leicht veränderbar sind, ohne das Quelltext verändert und erneut kompiliert werden muß.
- durch die vorhandene XML-Beschreibung zu den jeweiligen Ergebnissen jederzeit eine Rekonstruktion des Versuchs möglich ist.

Die **Untersuchung der Hypothesensprache** hat gezeigt, daß es unter den betrachteten Varianten keinen eindeutigen Sieger gibt, der in allen Fällen bessere Ergebnisse liefert. Obwohl nach den Meßergebnissen auf dem UCI Repository der BEXA konj. deutlich besser abschneidet, hat sich gezeigt, daß bei Vorliegen der Ausschluß-Eigenschaft (Definition 7.2) der BEXA disj. wesentlich bessere Ergebnisse liefert. Im UCI Repository liegt diese Eigenschaft bei den meisten Lernproblemen jedoch nicht vor. Sucht man also für ein neues Problem die richtige Hypothesensprache (bzw. Algorithmus), so kann sich ein Vergleich der beiden Varianten lohnen.

Verwendet man jedoch die **Weighted Relative Accuracy** Heuristik, so ist die Hypothesensprache von BEXA disj. im Vorteil.

8.3 Offene Punkte

8.3.1 SeCo-Framework

Bzgl. des **SeCo-Frameworks** gibt es sicherlich vieles was man verbessern oder ergänzen könnte. Einige Punkt seien hier erwähnt:

- Optimierung der Rechenzeit.
- Die Erarbeitung einer allgemeineren Fassung des Frameworks, welches eine noch größere Zahl von Algorithmen realisieren kann, also auch solche die nicht zu den Separate and Conquer Algorithmen gehören.
- Der Entwurf eines abstrakteren Regelmodells, mit dem noch mehr Hypothesensprachen unterstützt werden können.
- Standardisierung einer XML-Beschreibungssprache, die bei vielen Wissenschaftlern Anerkennung findet.
- Entwurf eines Meta-Algorithmus, der mittels XML-Beschreibung und SeCo-Factory Meta-Learning durchführen kann.

8.3.2 Ausschluß-Eigenschaft

An dieser Stelle soll nochmal erwähnt sein, daß es sich bei der Definition der **Ausschluß-Eigenschaft** (Definition 7.2) nicht um ein genau beschriebenes Merkmal handelt, anhand dessen man Lernprobleme in Kategorien einteilen kann, sondern daß es vielmehr ein beobachtetes Phänomen ist, welches genauerer Untersuchung bedarf, um es einerseits genauer beschreiben zu können und

andererseits einen effizienten Weg zu finden, der z.B. im Rahmen eines Meta-Learning Prozeß die Bestimmung der Eigenschaft im Vorfeld möglich macht.

8.3.3 Fallstudie

Der **BEXA mixed** soll ein Ansatz sein, der sich beider Hypothesensprachen bedient und jeweils die beste Formulierung finden soll. Es hat sich jedoch gezeigt, daß die bloße Einbeziehung beider Hypothesensprachen nicht den gewünschten Erfolg bringt. Hier gilt es, eine Heuristik zu finden, welche diese mächtige Hypothesensprache optimal ausnutzen kann, aber gleichzeitig den Rechenaufwand begrenzt, da der BEXA mixed den beiden anderen Varianten in Punkto Komplexität im Nachteil ist. Ein weiterer interessanter Aspekt könnte die Verwendung eines BottomUp-Verfeinerungsprozeß sein, der mit sehr spezifischen Regeln beginnt und diese generalisiert.

Literatur

[1] http://www.sc.ehu.es/ccwbayes/docencia/mmcc/files/, http://www.hakank.org/weka/.

[2] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability*, volume 2. Addison-Wesley, Reading, MA, 1989.

[3] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proc. Fifth European Working Session on Learning*, pages 151–163, Berlin, 1991. Springer.

[4] Peter Clark and Tim Niblett. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[5] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, July 9–12, 1995. Morgan Kaufmann.

[6] Johannes Fürnkranz. Separate-and-conquer rule learning. Technical report, Austrian Research Institude for Artificial Intelligence, 1996.

[7] Erich Gamma, Richard Helm, and Ralph Johnson und John Vlissides. *Entwurfsmuster*, volume 5. Addison-Wesley, 2001.

[8] S. Hettich, C.L. Blake, and C.J. Merz. UCI repository of machine learning databases, 1998.

[9] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, Juni/Juli 1988.

[10] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.

[11] R.S. Michalski. On the quasi-minimal solution of the covering algorithm. In *Proceedings of the 5th International Symposium on Information Processing (FCIP-69)*, volume A3, pages 125–128, Bled, Yugoslavia, 1969.

[12] R.S. Michalski. *Variable-valued logic and its applications to pattern recognition and machine learning.* North-Holland, D.C. Rhine, 1975.

[13] J.R. Quinlan. *Learning logical definitions from relations*, volume 5. 1990.

[14] J.R. Quinlan. Learning first-order definitions of functions. *Artificial Intelligence Research*, 5:139–161, 1996.

[15] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 5th Workshop on Algorithmic Learning Theory (ALT-95).* Springer, 1995.

[16] Hendrik Theron and Ian Cloete. Bexa: A covering algorithm for learning propositional concept descriptions. *Journal of Machine Learning*, pages 5–40, 1996.

[17] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools with Java implementations.* Morgan Kaufmann, San Francisco, 2000.

A SeCo-Factory

A.1 Konfiguration eines SeCo-Algorithmus

Die Beschreibung eines SeCo-Algorithmus erfolgt in XML und beinhaltet derzeit folgende Elemente und Attribute:

Tabelle A.1: Elemente der SeCo XML-Beschreibung

Element	Attribut	Beschreibung
seco		Das Wurzelelement, welches die Konfiguration eines SeCo-Klassifizierers beinhaltet.
secomp		Eine SeCo-Komponente.
	interface	Die Schnittstelle bzgl. der AbstractSeco-Klasse[58]. Mögliche Werte sind candidateselector, postprocessor, ruleevaluator, rulefilter, ruleinitializer, rulerefiner, rulestoppingcriterion, stoppingcriterion.
	classname	Name der Java-Klasse, welche die Komponente implementiert.
	package	(Optional) Das Java-Package, welches die angegebene Java-Klasse beinhaltet. Als Standardwert wird seco.learners.components angenommen.
jobject		Ein beliebiges Java-Objekt.
	classname	siehe secomp
	package	siehe secomp
	setter	Teilname der setter-Methode, welche für die Aggregation dieses Objektes mit dem Objekt der nächsthöheren Ebene verwendet wird. Wenn also die Java-Methode z.B. setHeuristic heißt, so muß als setter 'heuristic' angegeben werden.
property		Zur Festlegung einer spezifischen Eigenschaft eines Objektes.
	name	Name der Eigenschaft
	value	Wert der Eigenschaft

A.2 Properties

Grundsätzlich kann man für jedes Objekt properties setzen. Welche properties es gibt und welche Werte sie annehmen können, hängt von dem jeweiligen Objekt ab. In Tabelle A.2 sind ein paar gängige Properties angegeben.

A.3 Anforderungen an die Java-Klassen

Um die, in der Konfiguration benannten, Java-Klassen mit der SeCo-Factory verwenden zu können, müssen folgende Anforderungen erfüllt werden:

- **Die SeCo-Komponenten**, die direkt an das seco-Element angehängt werden, müssen das entsprechende Interface des seco.learners.core packages implementieren.

[58] siehe seco.learners.core.AbstractSeco

Tabelle A.2: Properties der SeCo-Objekte

Property	für Element	Beschreibung
threshold	LikelihoodRatio, ChiSquareFilter	Legt den Grenzwert fest, ab welchem Kandidatenregeln als signifikant betrachtet werden.
beamwidth	BeamWidthFilter	Legt fest, die wieviel besten Kandidatenregeln nach einer Verfeinerung beibehalten werden. Die übrigen werden verworfen.
nominal .cmpmode	BexaRefiner	Wenn diese Property nicht gesetzt wird, dann arbeitet der BEXA wie gewöhnlich mit dem $\neq$-Komparator. Bei Angabe von *equal* wird der $=$-Komparator verwendet und bei *both* sind beide Komparatoren in Gebrauch, was dem BEXA mixed entspricht.
skipdefault class	AbstractSeco	Diese Einstellung für den Kernel kann auf *true* gesetzt werden, damit für die Standardklasse keine Regeln gelernt werden, sondern nur die Standardregel verwendet wird.

- Objekte, die **Properties** erhalten sollen, müssen eine Methode mit folgender Signatur implementieren:

  ```
  public void setProperty(Object).
  ```

  Diese wird dann bei Generierung des Klassifizierers von der SeCo-Factory aufgerufen, um properties zu übergeben, sofern welche gesetzt wurden. Ansonsten sollte jedes Objekt Standardwerte für seine Properties vorsehen.

- Objekte, die andere Objekte, im Sinne einer **Aggregation**, beinhalten, müssen die entsprechende setter-Methode implementieren, wie im Fall *setter* $='$ *heuristic*$'$:

  ```
  public void setHeuristic(Object).
  ```

Column	Type	Comment
run_name	character varying(30)	The name of the run. Just a unique identifier.
algorithm	character varying(30)	The name of the algorithms that was used for the run.

Abbildung B.1: Datenbanktabelle *runs*

Column	Type	Comment
run_name	character varying(30)	The name of the run.
problem	character varying(30)	The learning problem.
output	text	The raw console ouput of the run.

Abbildung B.2: Datenbanktabelle *raw_output*

B Datenbankdesign

Hier soll kurz das Datenbankdesign beschrieben werden, das zur Erfassung und Verarbeitung der Meßergebnisse verwendet wurde. In Abbildung B.1 ist die Tabellenbeschreibung für die Tabelle *runs*, welche die Durchläufe mit ihrem Durchlaufnamen (*run_name*) und dem Algorithmusnamen (*algorithm*) beinhaltet. Ein Durchlauf umfaßt die Anwendung einer Algorithmusbeschreibung auf mehrere Lernprobleme. In der Regel wurde als *run_name* ein Zeitstempel verwendet, der den Beginn des Durchlaufs darstellt. Die Datenbanktabelle *properties* (siehe Abbildung B.3) enthält Einstellungen des Algorithmus, wie z.B. Grenzwerte für das StoppingCriterion oder die beamwidth. Die Konsolenausgabe eines Durchlaufs und des jeweiligen Lernproblems sind in Datenbanktabelle *raw_output* (siehe Abbildung B.2). Hier bilden der Durchlaufname und der Name des Lernproblems den Primärschlüssel. So kann gezielt für ein Tupel bestehend aus Duchlaufname und Lernproblemname die entsprechende Ausgabe erfragt werden, die neben der gelernten Regelmenge auch statistische Messungen wie Korrektheit und Anzahl der Regeln beinhaltet. In der Datenbanktabelle *stats* (siehe Abbildung B.4) kommt der selbe Primärschlüssel bestehend aus Duchlaufname und Lernproblemname zum Einsatz. Hier sind nun die, aus der Konsolenausgabe extrahierten, statistischen Informationen gelistet, die auch für die Ergebnistabellen in den Abschnitten 6 und 7 verwendet wurden. Die **XML-Beschreibungen** zu den Durchläufen sind in separaten Dateien gespeichert. Somit sind alle Informationen verfügbar, um jede Messung erneut durchführen zu können.

Column	Type	Comment
run_name	character varying(30)	The name of the run.
property_name	character varying(30)	Name of the property.
property_value	character varying	Value of the property.

Abbildung B.3: Datenbanktabelle *properties*

Column	Type	Comment
run_name	character varying(30)	The name of the run.
problem	character varying(30)	Name of the learning problem.
trainset	double precision	Correctness on training set.
testset	double precision	Correctness on test set.
rules	integer	Amount of rules in learned theory.
conditions	integer	Sum of conditions in rule set.
referred_atts	integer	Sum of referred attributes in rule set.
norm_length	integer	Sum of normalized rule length's.

Abbildung B.4: Datenbanktabelle *stats*

C API Dokumentation des SeCo-Framework

Dieser Anhang enthält die Dokumentation der Java-Klassen des SeCo-Framework. Einen Überblick über die Pakete gibt Tabelle C.1. Nicht alles ist im Rahmen dieser Arbeit entstanden. In der Tabelle sind die jeweiligen Authoren Johannes Fürnkranz und Matthias Thiel mit ihren Initialen eingetragen.

Tabelle C.1: Übersicht über Pakete des SeCo-Framework

Name	Author	Beschreibung
seco.models	JF	Das Regelmodell für die Hypothesensprachen.
seco.learners.factory	MT	Beinhaltet die SeCo-Factory und deren Hilfsklassen.
seco.learners.core	MT	Die abstrakte Abbildung eines SeCo-Algorithmus mit den notwendigen Schnittstellen für die SeCo-Komponenten.
seco.logger	MT	Das Logging Paket, um einzelne Rechenschritte nachvollziehen zu können.
seco.learners.bexa	MT	Klassen zur Umsetzung des BEXA.
seco.stats	JF	Enthält eine Klasse zur Berechnung von Statistiken.
seco.learners.components	MT	Die SeCo-Komponenten.
seco.heuristics	JF	Die Suchheuristiken.

C.1 Package seco.models

Package Contents *Page*

Classes

C.1.1 Class CandidateRule

The seco package implements generic functionality for simple separate-and-conquer rule learning on top of weka.

CandidateRule is a subclass of Rule for candidate rule. A candidate rule contains additional information like - a slot for storing the result of the rule - the history of the rule (a pointer back to the predecessor)

Parts of it is based on code for JRip and for Prism.

Declaration

public class CandidateRule
extends seco.models.Rule
implements java.lang.Comparable, weka.core.Copyable, java.lang.Cloneable

Constructors

- *CandidateRule*
  `public` **CandidateRule**`( )`
- *CandidateRule*
  `public` **CandidateRule**`( seco.models.Condition` **head** `)`

Methods

- *betterRule*
  `public static CandidateRule` **betterRule**`(`
  `seco.models.CandidateRule` **r1**`, seco.models.CandidateRule` **r2**
  `)`
  - **Usage**
    - return the better of two rules. It is assumed that computeRuleValue has been previously called! A rule is better if its evaluation is higher or if the evalution is the same, but it has shorter length, or the same length and a higher random tie break value. If one of the two is null, the other is returned.
  - **Parameters**
    - `r1` - the first candidate rule
    - `r2` - the second candidate rule
  - **Returns** - the better of the two rules
- *clone*
  `public Object` **clone**`( )`
  - **Returns** - a deep copy of the candidate rule

    the clone contains also contains fresh copies of the conditions and the coverage stats.

- *compareTo*
  `public int` **compareTo**`( seco.models.CandidateRule r )`
  - **Usage**
    - compare two rules. It is assumed that computeRuleValue has been previously called! A rule is better if its evaluation is higher or if the evalution is the same, but it has shorter length, or the same length and a higher random tie break value.
  - **Parameters**
    - `o` - the rule to compare to
  - **Returns** - -1 if the o is better, 1 if the old rule is better, 0 else
  - **Exceptions**
    - `java.lang.NullPointerException` - if the object is null

- *compareTo*
  `public int` **compareTo**`( java.lang.Object o )`
  - **Usage**
    - compare a rule to another object. If the object is another CandidateRule this function behaves like compareTo(CandidateRule). Otherwise, it throws a ClassCastException (as CandidateRules are comparable only to other CandidateRules).
  - **Parameters**
    - `o` - the object to compare to
  - **Returns** - -1 if the o is better, 1 if the old rule is better, 0 else
  - **Exceptions**
    - `java.lang.ClassCastException` - if the object is not a rule

- *computeRuleValue*
  `public double` **computeRuleValue**`(`
  `seco.heuristics.SearchHeuristic h )`
  - **Usage**
    - compute the heuristic evaluation of the rule with the provided rule value. This calls passes the object itself to the provided heuristic and internally stores the result.
  - **Parameters**
    - `h` - a search heuristic
  - **Returns** - the computed rule value

- *copy*
  `public Object` **copy**`( )`
  - **Returns** - a shallow copy of the candidate rule

    The copy does not copy the conditions, while the clone does.

- *getPredecessor*
  `public CandidateRule` **getPredecessor**`( )`
  - **Usage**
    - get the predecessor of the candidate rule
  - **Returns** - the predecessor or null

- *getRuleValue*
  `public double` **getRuleValue**`( )`
  - **Usage**
    - get the rule value that has been previously computed.

- *getTieBreaker*
  `public double` **getTieBreaker**`( )`
  - **Usage**
    - get a tie break value, i.e., a random number between 0 and 1. This number is always the same unless it is reset in the mean-time.

- *initBody*
  `public void` **initBody**`( )`
  - **Usage**
    - reset the body of the rule to an empty vector and reset all statistics to 0.

- *resetTieBreaker*
  `public void` **resetTieBreaker**`( )`
  - **Usage**
    - reset the tie break value. This causes the next call to getTieBreaker to return a new value. resetTieBreaker should be called whenever the rule candidate changes (e.g., by adding or deleting conditions). This is *not* done automatically. resetTieBreaker is automatically called only when copying or constructing a rule candidate.

- *setPredecessor*
  `public void` **setPredecessor**`( seco.models.CandidateRule r )`

- *setRandom*
  `public void` **setRandom**`( java.util.Random r )`
  - **Usage**
    - set the internal random generator to an initialized Random object.

- *shadow*
  `public Object` **shadow**`( )`

  - **Returns** - a copy of the rule, but with empty body and empty stats.

- *specialize*
  `public CandidateRule` **specialize**`( seco.models.Condition` **c** `)`
  - **Usage**
    - return a specialization of the current rule. The specialization will be a fresh copy and the current rule will be its predecessor.
  - **Parameters**
    - `c` - a condition
  - **Returns** - a specialization of the rule that results from adding c

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - print out a candidate rule with coverage statistics and the heuristic value

      '@return a printable representation of the rule

C.1.2 Class Condition

The seco package implements generic functionality for simple separate-and-conquer rule learning.

Condition implements conditions for rules

Parts of it is based on code for JRip and for Prism.

Declaration

public abstract class Condition
extends java.lang.Object
implements weka.core.Copyable

Constructors

- *Condition*
  `public` **Condition**`( weka.core.Attribute` **a** `)`
- *Condition*
  `public` **Condition**`( weka.core.Attribute` **a**`, double` **value** `)`
- *Condition*
  `public` **Condition**`( weka.core.Attribute` **a**`, double` **value**`,`
  `boolean` **cmp** `)`

METHODS

- *cmp*
  `public boolean` **cmp**`( )`
- *copy*
  `public Object` **copy**`( )`
  - **Usage**
    - implements Copyable
  - **Returns** - a shallow copy of itself
- *coveredInstances*
  `public Instances` **coveredInstances**`( weka.core.Instances` **data** `)`
  - **Usage**
    - return the list of instances that are covered by the condition
  - **Parameters**
    - `data` - the list of instances
  - **Returns** - a new list of covered instances
- *covers*
  `public abstract boolean` **covers**`( weka.core.Instance` **inst** `)`
- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - Compares this object with the given one.
  - **Parameters**
    - `o` - The other object.
  - **Returns** - false for all objects that are not instance of Condition. It returns true, if the attribute, value and cmp are equal, otherwise false.
- *getAttr*
  `public Attribute` **getAttr**`( )`
- *getValue*
  `public double` **getValue**`( )`
- *setCmp*
  `public void` **setCmp**`( boolean` **c** `)`
- *setValue*
  `public void` **setValue**`( double` **v** `)`
- *toString*
  `public abstract String` **toString**`( )`

C.1.3 Class NominalCondition

Conditions with nominal values

Declaration

public class NominalCondition
extends seco.models.Condition
implements java.lang.Cloneable

Constructors

- *NominalCondition*
  `public` **NominalCondition**`( weka.core.Attribute` **a** `)`
- *NominalCondition*
  `public` **NominalCondition**`( weka.core.Attribute` **a**`, double` **value** `)`
- *NominalCondition*
  `public` **NominalCondition**`( weka.core.Attribute` **a**`, double` **value**`, boolean` **cmp** `)`

Methods

- *covers*
  `public boolean` **covers**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - check whether the instance is covered by this condition
  - **Parameters**
    - `inst` - the instance in question
  - **Returns** - the boolean value indicating whether the instance is covered by this antecedent
- *toString*
  `public String` **toString**`( )`

C.1.4 Class NumericCondition

Conditions with numeric values

Declaration

public class NumericCondition
extends seco.models.Condition
implements java.lang.Cloneable

CONSTRUCTORS

- *NumericCondition*
  `public` **NumericCondition**`( weka.core.Attribute` **a** `)`
- *NumericCondition*
  `public` **NumericCondition**`( weka.core.Attribute` **a**`, double` **value** `)`
- *NumericCondition*
  `public` **NumericCondition**`( weka.core.Attribute` **a**`, double` **value**`, boolean` **cmp** `)`

METHODS

- *covers*
  `public boolean` **covers**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - check Whether the instance is covered by this condition
  - **Parameters**
    - `inst` - the instance in question
  - **Returns** - the boolean value indicating whether the instance is covered by this antecedent
- *toString*
  `public String` **toString**`( )`

C.1.5 CLASS Rule

The seco package implements generic functionality for simple separate-and-conquer rule learning on top of weka.

Rule implements the representation of a single rule. Note that in principle, a Rule can be used as a classifier in its own right.

Parts of it is based on code for JRip and for Prism.

DECLARATION

public class Rule
extends java.lang.Object
implements weka.core.Copyable, java.lang.Cloneable

CONSTRUCTORS

- *Rule*
  `public` **Rule**`( )`
- *Rule*
  `public` **Rule**`( seco.models.Condition` **head** `)`

METHODS

- *addCondition*
  `public void` **addCondition**`( seco.models.Condition` **c** `)`
  - **Usage**
    - add a condition to the body of the rule

- *attributeSet*
  `public HashSet` **attributeSet**`( )`
  - **Usage**
    - The set of attributes that are to be tested by this rule.
  - **Returns** - The set of attributes belonging to this rule.

- *classifyInstance*
  `public double` **classifyInstance**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - classify the passed Instance, i.e. return the class value if the instance is covered by the rule, or the missing value if it is not covered by the rule.
  - **Parameters**
    - `inst` - the instance
  - **Returns** - the class of the instance or a missing value

---

- *clone*
  `public Object` **clone**`( )`
  - **Returns** - a deep copy of the list.

    the clone contains also contains fresh copies of the conditions.

- *copy*
  `public Object` **copy**`( )`
  - **Returns** - a shallow copy of the list.

    the copy is shallow in the sense that the conditions are not copied (i.e., both the original and the copy point to the same conditions). However, all statistics are copied, so that both rules can be evaluated and used independently.

- *coveredInstances*
  `public Instances` **coveredInstances**`( weka.core.Instances` **data** `)`
  - **Usage**
    - return the set of Instances that are covered by the rule.
  - **Parameters**
    - `data` - the set of instances

  - **Returns** - the set of instances taht are not covered.

- *covers*
  `public boolean` **covers**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - check whether the rule covers an instance
  - **Parameters**
    - `inst` - the instance to check
  - **Returns** - true if the rule covers the instance, false else

---

- *deleteCondition*
  `public void` **deleteCondition**`( int` **n** `)`
  - **Usage**
    - delete the nth condition from the body of the rule
  - **Parameters**
    - `n` - number of the condition to delete

- *deleteLastCondition*
  `public void` **deleteLastCondition**`( )`
  - **Usage**
    - delete the last condition from the body of the rule

- *getBody*
  `public FastVector` **getBody**`( )`
  - **Returns** - the body of the rule, a FastVector of Conditions

---

- *getCondition*
  `public Condition` **getCondition**`( int` **n** `)`
  - **Parameters**
    - `n` - the number of the condition that should be returned
  - **Returns** - the nth condition of the body.

- *getHead*
  `public Condition` **getHead**`( )`
  - **Returns** - the head of the rule, a condition on the class attribute

---

- *getLastCondition*
  `public Condition` **getLastCondition**`( )`
  - **Returns** - the final condition of the body.

- *getPredictedValue*
  `public double` **getPredictedValue**`( )`
  - **Returns** - the class value predicted by the rule

- *getStats*
  `public TwoClassStats` **getStats**`( )`
  - **Usage**
    - return the TwoClassStats object containing the coverage counts

---

- *initBody*
  `public void` **initBody**`( )`
  - **Usage**
    - reset the body of the rule to an empty vector and the stats to 0

- *length*
  `public int` **length**`( )`
  - **Usage**
    - get the length of the rule, the number of conditions in the body

---

- *normalizedLength*
  `public int` **normalizedLength**`( )`
  - **Usage**
    - The normalized rule length checks, whether the rule's body can be optimized by substitution of conditions with other conditions that have inverted cmp. It computes the rule length by regarding the inverted rule representation.
  - **Returns** - The normalized rule length.

- *referredAttributes*
  `public int` **referredAttributes**`( )`
  - **Usage**
    - Determines the amount of different attributes that are to be tested when applying this rule. So only the mentioned attributes will be counted.
  - **Returns** - Amount of referred attributes.

- *replaceCondition*
  `public void` **replaceCondition**`( seco.models.Condition c, int n )`
  - **Usage**
    - replace the nth condition from the body of the rule with a new condition
  - **Parameters**
    - `c` - new condition
    - `n` - number of condition to replace

- *replaceLastCondition*
  `public void` **replaceLastCondition**`( seco.models.Condition c )`

  - **Usage**
    * replace the last condition from the body of the rule with a new condition
  - **Parameters**
    * `c` - new condition

- *setHead*
`public void` **setHead**`( seco.models.Condition` **h** `)`
  - **Usage**
    * set the head of the rule to a new class condition

- *setStats*
`public void` **setStats**`( seco.stats.TwoClassStats` **stats** `)`
  - **Usage**
    * set the TwoClassStats object to a new object

- *shadow*
`public Object` **shadow**`( )`
  - **Returns** - a copy of the rule, but with empty body and empty stats.

- *toString*
`public String` **toString**`( )`
  - **Usage**
    * print out a rule with coverage statistitics

      ‘@return a printable representation of the rule

- *uncoveredInstances*
`public Instances` **uncoveredInstances**`( weka.core.Instances` **data** `)`
  - **Usage**
    * return the set of Instances that are not covered by the rule.
  - **Parameters**
    * `data` - the set of instances
  - **Returns** - the set of instances taht are not covered.

- *uncoveredInstancesWithNeg*
`public Instances` **uncoveredInstancesWithNeg**`(`
`weka.core.Instances` **data**`, double` **posClass** `)`
  - **Usage**
    * return the set of Instances that are not covered by the rule or are negative.
  - **Parameters**
    * `data` - the set of instances
    * `posClass` - The positive class.
  - **Returns** - the set of instances taht are not covered.

C.1.6 Class **RuleSet**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

RuleSet implements the representation of a rule set.

TODO: - Should the default rule be stored as another rule?

Declaration

public class RuleSet
extends java.lang.Object

Constructors

- *RuleSet*
  `public` **RuleSet**`( )`

Methods

- *addRule*
  `public void` **addRule**`( seco.models.Rule` **c** `)`
  - **Usage**
    - add a rule to the set
  - **Parameters**
    - `r` - the rule to be added

- *classifyInstance*
  `public double` **classifyInstance**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - classify the passed Instance with the rule set. Currently we assume classification as a decision list, i.e., the prediction of the first rule that doesn't predict the missing value is returned. Eventually, this should probably be a parameter or maybe even a separate subclass.
  - **Parameters**
    - `inst` - the instance
  - **Returns** - the class of the instance or a missing value

- *deleteLastRule*
  `public void` **deleteLastRule**`( )`
  - **Usage**
    - delete the last rule

- *deleteRule*
  `public void` **deleteRule**`( int  n )`
  - **Usage**
    - delete the nth rule
  - **Parameters**
    - `n` - number of the rule to delete

- *getCoveringRules*
  `public FastVector` **getCoveringRules**`( weka.core.Instance  inst )`
  - **Usage**
    - return all rules that cover a given instance
  - **Parameters**
    - `inst` - the instance
  - **Returns** - a FastVector of rules that cover the instance

---

- *getDefaultPrediction*
  `public double` **getDefaultPrediction**`( )`
  - **Usage**
    - get the default prediction

- *getDefaultRule*
  `public Rule` **getDefaultRule**`( )`
  - **Usage**
    - get the default rule

- *getFirstCoveringRule*
  `public Rule` **getFirstCoveringRule**`( weka.core.Instance  inst )`
  - **Usage**
    - return the first rule that covers a given instance or null if no instance covers the instance. The default rule is not tested.
  - **Parameters**
    - `inst` - the instance
  - **Returns** - a the first rule that cover the instance

- *getLastRule*
  `public Rule` **getLastRule**`( )`
  - **Returns** - the final rule of the set

- *getRule*
  `public Rule` **getRule**`( int  n )`
  - **Parameters**
    - `n` - the number of the rule should be returned

  - **Returns** - the nth rule

- *getRules*
`public FastVector` **getRules**`( )`
  - **Returns** - the rule set, a FastVector of Rules. Note that the default rule is notreturned.

- *normalizedLength*
`public int` **normalizedLength**`( )`
  - **Usage**
    - The sum of normalized rule lengths. The normaliation regards alternative rule representations that would shorten the rule.
  - **Returns** - The sum of normalized rule lengths.

- *numConditions*
`public int` **numConditions**`( )`
  - **Returns** - the number of conditions in the rules

- *numRules*
`public int` **numRules**`( )`
  - **Returns** - the number of rules (excluding the default rule)

- *referredAttributes*
`public int` **referredAttributes**`( )`
  - **Usage**
    - Sums the number of attribute references in the rules which is the amount of different attributes that are to be tested when applying the rule.
  - **Returns** - The number of attribute references in the rules.

- *replaceCondition*
`public void` **replaceCondition**`( seco.models.Rule r, int n )`
  - **Usage**
    - replace the nth rule with a new rule
  - **Parameters**
    - `r` - new rule
    - `n` - number of rule to replace

- *replaceLastRule*
`public void` **replaceLastRule**`( seco.models.Rule r )`
  - **Usage**
    - replace the last rule with a new rule
  - **Parameters**
    - `r` - new rule

- *setDefaultRule*
  `public void` **setDefaultRule**`( seco.models.Rule r )`
  - **Usage**
    - set the default rule to a new rule
  - **Parameters**
    - `r` - the new rule

- *toString*
  `public String` **toString**`( )`
  - **Returns** - a printable version of a rule set.

C.2 Package seco.learners.factory

Package Contents *Page*

Classes

C.2.1 Class **ConfigNode**

Represents a node of a learner configuration. It is used for building a tree of the xml document.

Declaration

public class ConfigNode
extends java.lang.Object

Fields

- public static final int ROOT
  - –
- public static final int SECO
  - –
- public static final int SECOMP
  - –
- public static final int JOBJECT
  - –
- public Hashtable properties
  - – The appended property elements as name/value pairs in a table.
- public ConfigNode pred
  - – The predecessor node.
- public LinkedList succ
  - – All successor nodes in the order as they appear in the document.
- public int type
  - – The type of node.
- public Hashtable attrs
  - – The attributes of the xml element as name/value pairs in the table.

Constructors

- *ConfigNode*
  `public` **ConfigNode**`( int  type )`
  - **Usage**
    - Creates a new instance of ConfigNode
  - **Parameters**
    - `type` - The type of node.

C.2.2 Class **ConfigurableSeco**

This kind of AbstractSeco is configurable by setting the Seco components. It uses some default components that may be optionally overwritten.

Declaration

public class ConfigurableSeco
extends seco.learners.core.AbstractSeco

Fields

- public static final String SELECTOR
  –
- public static final String POSTPROCESSOR
  –
- public static final String RULEEVALUATOR
  –
- public static final String RULEFILTER
  –
- public static final String RULEINITIALIZER
  –
- public static final String RULEREFINER
  –
- public static final String RULESTOPCRITERION
  –
- public static final String STOPCRITERION
  –

Constructors

- *ConfigurableSeco*
  `public` **ConfigurableSeco**`( )`
  - **Usage**
    * Creates a new instance of ConfigurableSeco and some default components.

- *ConfigurableSeco*
  `public` **ConfigurableSeco**`( java.lang.String` **id** `)`

- **Usage**
  - Creates a new instance of ConfigurableSeco
- **Parameters**
  - `id` - Optional identifier for this Classifier.

METHODS

- *setSecoComponent*
  `public void` **setSecoComponent**`(`
  `seco.learners.core.ISecoComponent` **secomp**`, java.lang.String` **ifaceName** `)`
  - **Usage**
    - This will be used for adding a new component by its interface name. It will overwrite a previously set (or default) component, if any.
  - **Parameters**
    - `secomp` - The new seco component.
    - `ifacename` - The interface name which must be one of selector, postprocessor, rulerefiner, rulefilter, ruleinitializer, stopcriterion, rulestopcriterion, ruleevaluator

C.2.3 CLASS SecoFactory

This factory will produce instances of ConfigurableSeco (or AbstractSeco) which could be any variant of a separate and conquer (or covering) algorithm. As description for the algorithm, it needs a xml document.

DECLARATION

public class SecoFactory
extends java.lang.Object

CONSTRUCTORS

- *SecoFactory*
  `public` **SecoFactory**`( )`
  - **Usage**
    - Creates a new instance of SecoFactory

METHODS

- *addObjToObj*
  `public static void` **addObjToObj**`( java.lang.Object` **o1**`, java.lang.Object` **o2**`, java.lang.String` **setter** `)`

  - **Usage**
    * This will add o1 to o2 by using a setter method.
  - **Parameters**
    * `o1` - The object that will be used as argument.
    * `o2` - The object that has to contain the setter method.
    * `setter` - The name of the setter method without the prefix 'set'. For example: heuristic, if you want to use the setHeuristic method. Since there's no information about required type of o1, the setter method must have exactly one argument of type Object.

- *addPropertiesToNode*
  `protected void` **addPropertiesToNode**`( java.lang.Object` **o**`, java.util.Hashtable` **props** `)`
  - **Usage**
    * This will add the given properties to the object, if it has a method setProperty that may receive such properties.
  - **Parameters**
    * `o` - The object that will get the properties via setProperty method, if it exists.
    * `props` - The properties as name/value pairs in the table.

- *appendNode*
  `protected void` **appendNode**`( int` **type**`, java.util.Hashtable` **attrs** `)`
  - **Usage**
    * This will create a new current node. It updates the pred and succ variable of the nodes.
  - **Parameters**
    * `type` - The node type.
    * `attrs` - The attributes of the xml element.
  - **See Also**
    * `seco.learners.factory.ConfigNode` (in C.2.1, page 106)

- *createByClassname*
  `protected static Object` **createByClassname**`( java.lang.String` **name** `)`
  - **Usage**
    * This will create a new instance of the named class using a default constructor without arguments.
  - **Parameters**
    * `name` - The classname.
  - **Returns** - The new object.

- *createSeco*
  `public ConfigurableSeco` **createSeco**`( java.io.Reader` **reader** `)`

  - **Usage**
    * Creates a new seco by parsing the xml input and using the first seco element as description of the seco that is to be built.
  - **Parameters**
    * `reader` - A Reader providing the xml data.
  - **Returns** - The new seco classifier.

- *main*
`public static void` **main**`( java.lang.String [] args )`

  - **Usage**
    * Main method.

- *processConfigNode*
`protected Object` **processConfigNode**`(`
`seco.learners.factory.ConfigNode` **n** `)`

  - **Usage**
    * This will create a new object that is represented by the configuration node.
  - **Parameters**
    * `n` - The node of the xml tree that represents the new object.
  - **Returns** - The new object.

- *processDocument*
`public void` **processDocument**`( org.xmlpull.v1.XmlPullParser` **xpp** `)`

  - **Usage**
    * This will process the tags of the document.
  - **Parameters**
    * `xpp` - The pull parser.

- *processEndElement*
`public void` **processEndElement**`( org.xmlpull.v1.XmlPullParser` **xpp** `)`

  - **Usage**
    * This will process the end elements.
  - **Parameters**
    * `xpp` - The pull parser.

- *processSecoNode*
`protected ConfigurableSeco` **processSecoNode**`(`
`seco.learners.factory.ConfigNode` **secoNode** `)`

  - **Usage**
    * This will build the seco classifier.
  - **Parameters**
    * `secoNode` - The xml tree node that represents the seco.

  - **Returns** - The new seco classifier.

- *processStartElement*
  `public void` **processStartElement(**
  `org.xmlpull.v1.XmlPullParser` **xpp)**
  - **Usage**
    - This will process all start elements.
  - **Parameters**
    - `xpp` - The pull parser.

- *processText*
  `public void` **processText(** `org.xmlpull.v1.XmlPullParser` **xpp)**
  - **Usage**
    - Not used yet.

C.3 Package seco.learners.core

Package Contents *Page*

C.3.1 Interface ICandidateSelector

An interface for implementations that select candidate rules for a separate and conquer algorithm.

Declaration

public interface ICandidateSelector
implements ISecoComponent

Methods

- *selectCandidates*
  `public RuleSet` **selectCandidates**`( java.util.TreeSet` **rules**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will determine the CandidateRules for the next iteration of the algorithm.
  - **Parameters**
    - `rules` - A sorted set of rules.
    - `examples` - The training set of examples.
  - **Returns** - A set of CandidateRules.

C.3.2 Interface IPostProcessor

The interface for an implementation that will perform postprocessing on the learned theory.

Declaration

public interface IPostProcessor
implements ISecoComponent

Methods

- *postProcessTheory*
  `public RuleSet` **postProcessTheory**`( seco.models.RuleSet` **theory** `)`
  - **Usage**
    - Processes the given theory after learning.
  - **Parameters**
    - `theory` - The learned theory.
  - **Returns** - The postprocessed Theory.

C.3.3 Interface **IRuleEvaluator**

Interface for an implementation of a rule evaluator. It will enable the seco algorithm to compare rules by assigning evaluation values to the rules determined by a heuristic.

Declaration

public interface IRuleEvaluator
implements ISecoComponent

Methods

- *evaluateRule*
  `public void` **evaluateRule**`( seco.models.CandidateRule` **r**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will evaluate a rule and store the value in the CandidateRule object.
  - **Parameters**
    - `r` - The rule that is to be evaluated.
    - `examples` - The training set.
- *setHeuristic*
  `public void` **setHeuristic**`( seco.heuristics.SearchHeuristic` **heuristic** `)`
  - **Usage**
    - Setter for the heuristic that is to be used. It will be called before the first call of evaluateRule and everytime when another heuristic is to be used.
  - **Parameters**
    - `heuristic` - The heuristic for evaluation of the rules.

C.3.4 Interface **IRuleFilter**

The implementation of a rule filter as it will be usually called after Refinement.

Declaration

public interface IRuleFilter
implements ISecoComponent

METHODS

- *filterRules*
  `public void` **filterRules**`( java.util.TreeSet` **rules**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This remove the filtered rules from the given TreeSet.
  - **Parameters**
    - `rules` - The rule set that is to be filtered.
    - `examples` - The training set.

C.3.5 INTERFACE IRuleInitializer

Interface for a rule initializer that is called at the beginning of the Find-Best-Rule method.

DECLARATION

public interface IRuleInitializer
implements ISecoComponent

METHODS

- *initializeRule*
  `public CandidateRule` **initializeRule**`( weka.core.Instances` **examples** `)`
  - **Usage**
    - This will determine an initial rule.
  - **Parameters**
    - `examples` - The training set.
  - **Returns** - The first CandidateRule.

C.3.6 INTERFACE IRuleRefiner

Interface for a rule refiner.

DECLARATION

public interface IRuleRefiner
implements ISecoComponent

METHODS

- *refineRule*
  `public RuleSet` **refineRule**`( seco.models.CandidateRule` **c**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will create a set of rules that have been refined from the given CandidateRule.
  - **Parameters**
    - `c` - The CandidateRule that is to be refined.
    - `examples` - The training set.
  - **Returns** - A set of refinements.

C.3.7 INTERFACE IRuleStoppingCriterion

An interface for objects that provide a rule stopping criterion.

DECLARATION

public interface IRuleStoppingCriterion
implements ISecoComponent

METHODS

- *checkForRuleStop*
  `public boolean` **checkForRuleStop**`( seco.models.RuleSet` **theory**`, seco.models.CandidateRule` **rule**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - Determines whether the rule stopping criterion fulfilled.
  - **Parameters**
    - `theory` - The current theory.
    - `rule` - The new best rule that has been just evaluated, not yet in theory.
    - `examples` - The training set.
  - **Returns** - true, if the criterion is fulfilled, otherwise false.

C.3.8 INTERFACE ISecoComponent

A common interface for all components of a separate and conquer algorithm.

DECLARATION

public interface ISecoComponent

METHODS

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - This will be called by the separate and conquer algorithm before using this component.
  - **Parameters**
    - `classVal` - The value of the class that is to be learned.
- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - All components are able to receive configurable properties via this method that will overwrite the default values.
  - **Parameters**
    - `name` - The name of the property.
    - `value` - The value of the property.

C.3.9 INTERFACE IStoppingCriterion

A stopping criterion may prevent further processing of a refined rule.

DECLARATION

public interface IStoppingCriterion
implements ISecoComponent

METHODS

- *checkForStop*
  `public boolean` **checkForStop**`( seco.models.CandidateRule` **refinement**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - A stopping criterion may prevent further processing of a refined rule.
  - **Parameters**
    - `refinement` - The current refined CandidateRule.
    - `examples` - The training set.
  - **Returns** - true, if the criterion is fulfilled, otherwise false.

C.3.10 CLASS AbstractSeco

Declaration

public abstract class AbstractSeco
extends weka.classifiers.Classifier

Fields

- public static final int UNSORTED
  –
- public static final int SORTED
  –

Constructors

- *AbstractSeco*
  `public` **AbstractSeco**`( )`
  - **Usage**
    - Creates a new instance of AbstractSeCo

Methods

- *buildClassifier*
  `public void` **buildClassifier**`( weka.core.Instances` **instances** `)`
  - **Usage**
    - The interface method for weka. It will execute the separate and conquer algorithm for each class that appears in the training set.
  - **Parameters**
    - `instances` - The training set.

- *classifyInstance*
  `public double` **classifyInstance**`( weka.core.Instance` **inst** `)`
  - **Usage**
    - classify the passed Instance, i.e. return the class value if the instance is covered by the rule, or the missing value if it is not covered by the rule. Currently we assume classification as a decision list, i.e., the prediction of the first rule that doesn't predict the missing value is returned. Eventually, this should probably be a parameter or maybe even a separate subclass.
  - **Parameters**
    - `inst` - the instance
  - **Returns** - the class of the instance or a missing value

---

- *containsPositive*
  `public static boolean` **containsPositive**`( weka.core.Instances` **examples**`, double` **classVal** `)`
  - **Usage**
    - Checks, whether the given examples contain positive ones according to classVal.
  - **Parameters**
    - `examples` - The training set.
    - `classVal` - The class value that is to be considered positive.
  - **Returns** - true, if there are positive examples, false otherwise.

- *findBestRule*
  `public CandidateRule` **findBestRule**`( weka.core.Instances` **examples** `)`
  - **Usage**
    - This will determine the best CandidateRule for the given training set.
  - **Parameters**
    - `examples` - The training set.
  - **Returns** - The best CandidateRule.

- *getTheory*
  `public RuleSet` **getTheory**`( )`
  - **Usage**
    - Getter for learned theory.

- *orderInstancesByClasses*
  `public Instances` **orderInstancesByClasses**`(`
  `weka.core.Instances` **examples** `)`
  - **Usage**
    - This will order the given examples in ascending frequency of their classes.
  - **Parameters**
    - `examples` - Unsorted examples.
  - **Returns** - The sorted examples.

- *removeRuleSetFromTreeSet*
  `protected void` **removeRuleSetFromTreeSet**`(`
  `seco.models.RuleSet` **rset**`, java.util.TreeSet` **tset** `)`
  - **Usage**
    - Removes all rules of the rset from the TreeSet.
  - **Parameters**
    - `rset` - The set of rules that is to be removed.
    - `tset` - The TreeSet that is to be modified.

- *separateAndConquer*
  `public RuleSet` **separateAndConquer**`( weka.core.Instances` **examples** `)`
  - **Usage**
    - The Separate and Conquer algorithm. The class, that is to be learned, has to be set up with setClassVal before calling this method.
  - **Parameters**
    - `examples` - The training set.
  - **Returns** - The theory that has been learned.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - This will set the classVal in m_classVal and it will propagate it to all seco components.
  - **Parameters**
    - `classVal` - The class that is to be learned.

- *setDefaultRule*
  `public void` **setDefaultRule**`( seco.models.RuleSet` **theory**`, weka.core.Instances` **examples**`, weka.core.Attribute` **cl** `)`
  - **Usage**
    - This will create the default rule that will assign the most frequent class to any example. The default rule will be put into the theory. The examples must be ordered in ascending frequency of their classes before applying this method.
  - **Parameters**
    - `theory` - The rule set that will get the default rule.
    - `examples` - The training set.
    - `cl` - original class attribute.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - This will be used for setting properties.
  - **Parameters**
    - `name` - The name of the property.
    - `value` - The value of the property.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Creates a string representation of the learned classifier. It contains the ruleset and some statistical information.
  - **Returns** - String representation of learned classifier.

C.4 Package seco.logger

Package Contents *Page*

Classes

C.4.1 Class **Logger**

The logger that will be used by this application. It's a variant of a log4j Logger.

Declaration

public class Logger
extends org.apache.log4j.Logger

Methods

- *error*
  `public void` **error**`( java.lang.Exception` **e** `)`
- *getLogger*
  `public static Logger` **getLogger**`( java.lang.Class` **c** `)`
- *getLogger*
  `public static Logger` **getLogger**`( java.lang.String` **name** `)`
- *info*
  `public void` **info**`( java.lang.Exception` **e** `)`
- *main*
  `public static void` **main**`( java.lang.String [] args )`
- *trace*
  `public void` **trace**`( java.lang.Object` **message** `)`
- *warn*
  `public void` **warn**`( java.lang.Exception` **e** `)`

C.4.2 Class **LoggerFactory**

The Logger factory for the Logger class.

Declaration

public class LoggerFactory
extends java.lang.Object
implements org.apache.log4j.spi.LoggerFactory

Constructors

- *LoggerFactory*
  `public` **LoggerFactory**`( )`
  - **Usage**
    - Creates a new instance of LoggerFactory

METHODS

- *makeNewLoggerInstance*
  `public Logger` **makeNewLoggerInstance**`( java.lang.String` **str** `)`

C.4.3 CLASS LogLevel

This class has been used for adding loglevel trace to log4j.

DECLARATION

public class LogLevel
extends org.apache.log4j.Level

FIELDS

- public static final int TRACE_INT
  –
- public static final String TRACE_STR
  –
- public static final Level TRACE
  –

CONSTRUCTORS

- *LogLevel*
  `protected` **LogLevel**`( int` **level**`, java.lang.String` **name**`, int` **sysLogEquiv** `)`
  - **Usage**
    - Creates a new instance of LogLevel

METHODS

- *toLevel*
  `public static Level` **toLevel**`( int` **i** `)`
- *toLevel*
  `public static Level` **toLevel**`( java.lang.String` **sArg** `)`
  - **Usage**
    - Convert the string passed as argument to a level. If the conversion fails, then this method returns #CONSTRUCT .
- *toLevel*
  `public static Level` **toLevel**`( java.lang.String` **sArg**`, org.apache.log4j.Level` **defaultValue** `)`

C.5 Package seco.learners.bexa

Package Contents *Page*

Classes

C.5.1 Class BexaRefinerTopDown

This is a refiner like it has been described in the original bexa paper. It uses a top down strategy.

Declaration

public class BexaRefinerTopDown
extends java.lang.Object
implements seco.learners.core.IRuleRefiner

Constructors

- *BexaRefinerTopDown*
  `public` **BexaRefinerTopDown**`( )`
  - **Usage**
    - Creates a new instance of BexaRefiner

Methods

- *createUsableConditions*
  `protected HashSet` **createUsableConditions**`(`
  `weka.core.Instances` **examples** `)`
  - **Usage**
    - This will create all usable conditions. Means the conditions that result from combining all attributes with all of their values.
  - **Parameters**
    - `examples` - The training set.
  - **Returns** - A HashSet containing the Condition objects.

- *filterUsableConditions*
  `public void` **filterUsableConditions**`( seco.models.CandidateRule` **c**`, java.util.HashSet` **usable**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - Remove from usable all values that will lead to unnecessary specializations by iterating over examples X usable conditions. Includes the prevent-empty-conjunctions restriction and the uncover new negatives restriction.
  - **Parameters**
    - `c` - The CandidateRule
    - `usable` - A set of usable conditions for c.
    - `examples` - The training set.

- *refineRule*
  `public RuleSet` **refineRule**`( seco.models.CandidateRule` **c**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will create specializations of the CandidateRule excluding some obviously useless specializations.
  - **Parameters**
    - `c` - The CandidateRule
    - `examples` - The training set.
  - **Returns** - The rule set containing the specialized rules.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - Presets the target class.
  - **Parameters**
    - `classVal` - The value of the target class.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - No properties yet.

C.5.2 Class **NumericComparator**

Declaration

public class NumericComparator
extends java.lang.Object
implements java.util.Comparator

Constructors

- *NumericComparator*
  `public` **NumericComparator**`( weka.core.Attribute` **att** `)`
  - **Usage**
    - Creates a new instance of NumericComparator

Methods

- *compare*
  `public int` **compare**`( java.lang.Object` **o1**`, java.lang.Object` **o2** `)`

C.6 Package seco.stats

Package Contents *Page*

Classes

Encapsulates performance functions for two-class problems.

C.6.1 Class **TwoClassStats**

Encapsulates performance functions for two-class problems.

Declaration

public class TwoClassStats
extends java.lang.Object
implements java.lang.Cloneable

Constructors

- *TwoClassStats*
  `public` **TwoClassStats**`( )`
  - **Usage**
    - Initializes a TwoClassStats with all 0s.

- *TwoClassStats*
  `public` **TwoClassStats**`( double` **tp**`, double` **fp**`, double` **tn**`, double` **fn** `)`
  - **Usage**
    - Creates the TwoClassStats with the given initial performance values.
  - **Parameters**
    - `tp` - the number of correctly classified positives
    - `fp` - the number of incorrectly classified negatives
    - `tn` - the number of correctly classified negatives
    - `fn` - the number of incorrectly classified positives

Methods

- *clone*
  `public Object` **clone**`( )`
  - **Returns** - a deep copy of the statistics

- *getAccuracy*
  `public double` **getAccuracy**`( )`
  - **Usage**
    - Calculate the accuracy This is defined as

```
correctly classified examples
---------------
      total examples
```

  - **Returns** - the accuracy

- *getConfusionMatrix*
`public ConfusionMatrix` **getConfusionMatrix**`( )`
  - **Usage**
    * Generates a `ConfusionMatrix` representing the current two-class statistics, using class names 'negative' and 'positive'.
  - **Returns** - a `ConfusionMatrix`.

- *getCorrect*
`public double` **getCorrect**`( )`
  - **Usage**
    * Gets the number of correctly predicted instances

- *getErrorRate*
`public double` **getErrorRate**`( )`
  - **Usage**
    * Calculate the error rate This is defined as

```
incorrectly classified examples
----------------
      total examples
```

  - **Returns** - the error rate

- *getFallout*
`public double` **getFallout**`( )`
  - **Usage**
    * Calculate the fallout. This is defined as

```
incorrectly classified negatives
----------------
  total predicted as positive
```

  - **Returns** - the fallout

- *getFalseNegative*
`public double` **getFalseNegative**`( )`
  - **Usage**
    * Gets the number of positive instances predicted as negative

---

- *getFalsePositive*
`public double` **getFalsePositive**`( )`
  - **Usage**

* Gets the number of negative instances predicted as positive

---

- *getFalsePositiveRate*
`public double` **getFalsePositiveRate**`( )`
  - **Usage**
    * Calculate the false positive rate. This is defined as

```
incorrectly classified negatives
----------------
       total negatives
```

  - **Returns** - the false positive rate

- *getFMeasure*
`public double` **getFMeasure**`( )`
  - **Usage**
    * Calculate the F-Measure. This is defined as

```
2 * recall * precision
-----------
  recall + precision
```

  - **Returns** - the F-Measure

- *getIncorrect*
`public double` **getIncorrect**`( )`
  - **Usage**
    * Gets the number of incorrectly predicted instances

- *getIsNegative*
`public double` **getIsNegative**`( )`
  - **Usage**
    * Gets the total number of negative instances

- *getIsPositive*
`public double` **getIsPositive**`( )`
  - **Usage**
    * Gets the total number of positive instances

- *getPrecision*
`public double` **getPrecision**`( )`
  - **Usage**

* Calculate the precision. This is defined as

```
correctly classified positives
---------------
 total predicted as positive
```

  - **Returns** - the precision

- *getPredictedNegative*
`public double` **getPredictedNegative**`( )`
  - **Usage**
    * Gets the total number of examples predicted negative

- *getPredictedPositive*
`public double` **getPredictedPositive**`( )`
  - **Usage**
    * Gets the total number of examples predicted positive

- *getPrior*
`public double` **getPrior**`( )`
  - **Usage**
    * Estimate the prior probability of positive examples This is defined as

```
positive examples
---------
 total examples
```

  - **Returns** - the prior

- *getRecall*
`public double` **getRecall**`( )`
  - **Usage**
    * Calculate the recall. This is defined as

```
correctly classified positives
---------------
      total positives
```

      (Which is also the same as the truePositiveRate.)
  - **Returns** - the recall

- *getTotal*
  `public double` **getTotal**`( )`
  - **Usage**
    - Gets the total number of examples

- *getTrueNegative*
  `public double` **getTrueNegative**`( )`
  - **Usage**
    - Gets the number of negative instances predicted as negative

---

- *getTruePositive*
  `public double` **getTruePositive**`( )`
  - **Usage**
    - Gets the number of positive instances predicted as positive

---

- *getTruePositiveRate*
  `public double` **getTruePositiveRate**`( )`
  - **Usage**
    - Calculate the true positive rate. This is defined as

```
correctly classified positives
---------------
      total positives
```

  - **Returns** - the true positive rate

- *incFalseNegative*
  `public double` **incFalseNegative**`( )`
  - **Usage**
    - Increments the number of false negatives

- *incFalseNegative*
  `public double` **incFalseNegative**`( double` **fn** `)`

- *incFalsePositive*
  `public double` **incFalsePositive**`( )`
  - **Usage**
    - Increments the number of false positives

- *incFalsePositive*
  `public double` **incFalsePositive**`( double` **fp** `)`

- *incTrueNegative*
  `public double` **incTrueNegative**`( )`
  - **Usage**

        * Increments the number of true negatives

- *incTrueNegative*
`public double` **incTrueNegative**`( double tn )`

- *incTruePositive*
`public double` **incTruePositive**`( )`
  - **Usage**
    * Increments the number of true positives

- *incTruePositive*
`public double` **incTruePositive**`( double tp )`

- *setFalseNegative*
`public void` **setFalseNegative**`( double fn )`
  - **Usage**
    * Sets the number of positive instances predicted as negative

---

- *setFalsePositive*
`public void` **setFalsePositive**`( double fp )`
  - **Usage**
    * Sets the number of negative instances predicted as positive

---

- *setTrueNegative*
`public void` **setTrueNegative**`( double tn )`
  - **Usage**
    * Sets the number of negative instances predicted as negative

---

- *setTruePositive*
`public void` **setTruePositive**`( double tp )`
  - **Usage**
    * Sets the number of positive instances predicted as positive

---

- *toString*
`public String` **toString**`( )`
  - **Usage**
    * Returns a string containing the various performance measures for the current object

C.7 Package seco.learners.components

Package Contents *Page*

Classes

C.7.1 Class BeamWidthFilter

This filter will reduce the rule set to the beam.

Declaration

public class BeamWidthFilter
extends java.lang.Object
implements seco.learners.core.IRuleFilter

Constructors

- *BeamWidthFilter*
  `public` **BeamWidthFilter**`( )`
  - **Usage**
    - Creates a new instance of BeamWidthFilter

Methods

- *filterRules*
  `public void` **filterRules**`( java.util.TreeSet` **rules**`,`
  `weka.core.Instances` **examples** `)`
  - **Usage**
    - This will remove all rules outside the preset beam width.
  - **Parameters**
    - `rules` - The rule set that is to be modified.
    - `examples` - The training set.
- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - Not needed for this filter.
- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`,`
  `java.lang.String` **value** `)`

C.7.2 Class ChiSquareFilter

This filter will check, whether a rule is significant in comparison to its predecessor.

Declaration

public class ChiSquareFilter
extends java.lang.Object
implements seco.learners.core.IRuleFilter

Constructors

- *ChiSquareFilter*
  `public` **ChiSquareFilter**`( )`
  - **Usage**
    - Creates a new instance of ChiSquareFilter

Methods

- *chiSquareK2*
  `public static double` **chiSquareK2**`( double` **p**`, double` **n**`, double` **ep**`, double` **en** `)`
  - **Usage**
    - The formula for computing the chi square.
  - **Parameters**
    - `p` - Predicted positives.
    - `n` - Predicted negatives.
    - `ep` - Predicted positives of predecessor.
    - `en` - Predicted negatives of predecessor.
  - **Returns** - The chi square value.

- *filterRules*
  `public void` **filterRules**`( java.util.TreeSet` **rules**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will remove insignificant rules from the given rule set. Uses the preset threshold that will be mapped by LikelihoodRatio Class.
  - **Parameters**
    - `rules` - The rule set that is to be modified.
    - `examples` - The training set.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - Not needed here.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`

C.7.3 Class DefaultRuleEvaluator

A rule evaluator whose heuristic is configurable so that it should match to most of the use cases.

Declaration

public class DefaultRuleEvaluator
extends java.lang.Object
implements seco.learners.core.IRuleEvaluator

Constructors

- *DefaultRuleEvaluator*
  `public` **DefaultRuleEvaluator**`( )`
  - **Usage**
    - Creates a new instance of DefaultRuleEvaluator

Methods

- *evaluateRule*
  `public void` **evaluateRule**`( seco.models.CandidateRule` **r**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - This will setup the TwoClassStats of the rule and will call the computeRule method. The result will be stored in the rule. The setClassVal and setHeuristic methods have to be called first.
  - **Parameters**
    - `r` - The CandidateRule that has to be evaluated.
    - `examples` - The training set.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - This will setup the class value of the class that is to be learned.
  - **Parameters**
    - `classVal` - The new class value.

- *setHeuristic*
  `public void` **setHeuristic**`( java.lang.Object` **heuristic** `)`

- *setHeuristic*
  `public void` **setHeuristic**`( seco.heuristics.SearchHeuristic` **heuristic** `)`
  - **Usage**

    * This will set the heuristic that is to be used for evaluation of the rules.
  - **Parameters**
    * `heuristic` - The heuristic.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`

C.7.4 Class DefaultRuleInitializer

This initializer will create a first rule that classifies everything as the current target class.

Declaration

public class DefaultRuleInitializer
extends java.lang.Object
implements seco.learners.core.IRuleInitializer

Constructors

- *DefaultRuleInitializer*
  `public` **DefaultRuleInitializer**`( )`
  - **Usage**
    * Creates a new instance of DefaultRuleInitializer

Methods

- *initializeRule*
  `public CandidateRule` **initializeRule**`( weka.core.Instances` **examples** `)`
  - **Usage**
    * Generates the first CandidateRule that classifies all examples with the target class.
  - **Parameters**
    * `examples` - Training set.
  - **Returns** - The CandidateRule.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    * Sets the target class.
  - **Parameters**

    * `Value` - of target class.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`

C.7.5 Class DefaultRuleStop

The DefaultRuleStop checks, whether a given rule is better than the default rule according to a preset rule evaluator.

Declaration

public class DefaultRuleStop
extends java.lang.Object
implements seco.learners.core.IRuleStoppingCriterion

Constructors

- *DefaultRuleStop*
  `public` **DefaultRuleStop**`( )`
  - **Usage**
    * Creates a new instance of DefaultRuleStop Initializes with a DefaultRuleEvaluator using Laplace heuristic.

Methods

- *checkForRuleStop*
  `public boolean` **checkForRuleStop**`( seco.models.RuleSet` **theory**`, seco.models.CandidateRule` **rule**`, weka.core.Instances` **examples** `)`
  - **Usage**
    * Compares the given rule with the default rule using the preset rule evaluator.
  - **Parameters**
    * `theory` - The current theory.
    * `rule` - The CandidateRule that is to be compared.
    * `examples` - The training set.
  - **Returns** - true, if it should stop, false otherwise.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    * Sets the target class.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - Not used yet.

- *setRuleEvaluator*
  `public void` **setRuleEvaluator**`( java.lang.Object` **eval** `)`
  - **Usage**
    - Setter for the rule evaluator that is to be used for the rule stop check.
  - **Parameters**
    - `eval` - The rule evaluator that has to be instance of IRuleEvaluator !

C.7.6 Class DefaultSelector

The default candidate selector for the most common cases.

Declaration

public class DefaultSelector
extends java.lang.Object
implements seco.learners.core.ICandidateSelector

Constructors

- *DefaultSelector*
  `public` **DefaultSelector**`( )`
  - **Usage**
    - Creates a new instance of DefaultSelector

Methods

- *selectCandidates*
  `public RuleSet` **selectCandidates**`( java.util.TreeSet` **rules**`, weka.core.Instances` **examples** `)`
  - **Usage**
    - Selects all candidates.
  - **Parameters**
    - `rules` - The rule set.
    - `examples` - The training set.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`

C.7.7 Class LikelihoodRatio

Implementation that will calculate the likelihood ratio. As being an implementation of IStoppingCriterion it is able to perform a check, whether that value exceeds a preset threshold.

Declaration

public class LikelihoodRatio
extends java.lang.Object
implements seco.learners.core.IStoppingCriterion

Constructors

- *LikelihoodRatio*
  `public` **LikelihoodRatio**`( )`
  - **Usage**
    - Creates a new instance of LikelihoodRatio

Methods

- *checkForStop*
  `public boolean` **checkForStop**`( seco.models.CandidateRule` **refinement**`, weka.core.Instances` **examples** `)`
- *evalLikelihoodRatioStatistic*
  `public static double` **evalLikelihoodRatioStatistic**`( double` **p**`, double` **n**`, double` **ep**`, double` **en** `)`
  - **Usage**
    - Calculates the likelihood ratio statistic.
- *mapThreshold*
  `public static double` **mapThreshold**`( double` **t** `)`
  - **Usage**
    - This maps the threshold.
  - **Parameters**
    - `t` - The threshold that has to be one of: 0.7, 0.75, 0.8, 0.85, 0.9 0.95, .0.975, 0.99, 0.995
  - **Returns** - The border that has to be compared with the lrs-value.

  - **Exceptions**
    - `java.lang.Exception` - If the border for the argument (threshold) is not defined.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - Sets the target class.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - Sets properties for the likelihood check. Currently supported properties: threshold
  - **Parameters**
    - `name` - name of property.
    - `value` - new value of property
  - **See Also**
    - `mapThreshold`

C.7.8 Class MultiRuleFilter

This filter will combine multiple other filters. The filters will be applied in a row.

Declaration

public class MultiRuleFilter
extends java.lang.Object
implements seco.learners.core.IRuleFilter

Constructors

- *MultiRuleFilter*
  `public` **MultiRuleFilter**`( )`
  - **Usage**
    - Creates a new instance of MultiRuleFilter

Methods

- *filterRules*
  `public void` **filterRules**`( java.util.TreeSet` **rules**`, weka.core.Instances` **examples** `)`
  - **Usage**

    - This will apply all added filters on the rule set.
  - **Parameters**
    - `rules` - the rule set.
    - `examples` - The training set.

- *setClassVal*
  `public void` **setClassVal**`( double` **classVal** `)`
  - **Usage**
    - This will just propagate the current target class value to the added filters.
  - **Parameters**
    - `classVal` - The target class value.

- *setFilter*
  `public void` **setFilter**`( java.lang.Object` **filter** `)`
  - **Usage**
    - This will add another filter. The filters will be applied in the order in which they will be added by this method.
  - **Parameters**
    - `filter` - The new filter.

- *setProperty*
  `public void` **setProperty**`( java.lang.String` **name**`, java.lang.String` **value** `)`
  - **Usage**
    - For setting special properties for this filter. Currently not implemented.

C.8 Package seco.heuristics

Package Contents *Page*

Classes

*Linear Costs implements a class for evaluating a rule with a linear cost function (c*tp - (1-c)*fp), where 0 <= c <= 1.*

c = 1 means that only covered positives counts, c = 0 means that only excluding negatives counts, values inbetween trade off between these two extremes.

This file implements a generic class for evaluating a rule with the m-estimate, i.e.

This file implements a generic class for evaluating a rule with weighted relative accuracy.

C.8.1 Class **Accuracy**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with accuracy. The accuracy of a single rule is (tp + tn) / (tp + fp + fn + tn).

You may also want to consider the equivalent but faster Difference. For more details on the the equivalences between search heuristics see (Fürnkranz & Flach, ICML-03).

Declaration

public class Accuracy
extends seco.heuristics.SearchHeuristic

Constructors

- *Accuracy*
  `public` **Accuracy**`( )`
  - **Usage**
    - Constructor

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - return the accuracy of the rule

C.8.2 Class **Correlation**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the Correlation-estimate, i.e. tp*tn - fp*fn/sqrt(Pos*Neg*Covered*Uncovered)

DECLARATION

public class Correlation
extends seco.heuristics.SearchHeuristic

CONSTRUCTORS

- *Correlation*
  `public` **Correlation**`( )`

METHODS

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with the Correlation estimate

C.8.3 CLASS Difference

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the difference between the covered positive and negative examples, i.e., tp - fp.

Note, however, that for rules with the same example distribution (tp + fn) and (fp + tn) are constant), this is eqivalent to accuracy, but presumbably somewhat faster.

For more details on the the equivalences between search heuristics that are mentioned below, see (Fürnkranz & Flach, ICML-03)

DECLARATION

public class Difference
extends seco.heuristics.SearchHeuristic

CONSTRUCTORS

- *Difference*
  `public` **Difference**`( )`
  - **Usage**
    - Constructor

METHODS

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - return the difference between the number of positive and negative examples covered by the rule. This is equivalent but presumably faster than accuracy.

C.8.4 CLASS FoilGain

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with Foil's information gain, i.e. tp*(log2(tp/(tp+fp)) - log2(tp'/(tp'+fp'))) where tp' and fp' are the true and false positives of the parent rule.

DECLARATION

public class FoilGain
extends seco.heuristics.SearchHeuristic

CONSTRUCTORS

- *FoilGain*
  `public` **FoilGain**`( )`

METHODS

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with the Foil's information gain heuristic. Rules without predecessors (getPredecessor() == null) are evaluated with 0.

C.8.5 CLASS GeneralizedM

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the generalized m-estimate, i.e. (tp+m*c)/(tp+fp+m).

c may be interpreted as a general linear cost factor, just like in LinearCosts.

m is the m-value as in the m-heuristic. It may be viewed as a trade-off between LinearCost (which assumes a cost value c) and Precision (which does not make any cost assumptions).

If m is NaN, it will be interpreted as infinity and the LinearCosts heuristic will be called. If m is 0, you get Precision.

The default values are m = 2 and c = 0.5, which results in the Laplace heuristic.

See (Fürnkranz & Flach, ICML-03) for details on the Generalized MEstimate.

Declaration

public class GeneralizedM
extends seco.heuristics.SearchHeuristic
implements weka.core.OptionHandler

Constructors

- *GeneralizedM*
  `public` **GeneralizedM**`( )`
  - **Usage**
    - Empty constructor, c will be set to 0.5 and m to 2.0.

- *GeneralizedM*
  `public` **GeneralizedM**`( double` **m**`, double` **c** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `m` - the value for m (0 <= m <= NaN, default 1)
    - `c` - the cost factor (0 <= c <= 1, default 0.5)

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule` **r** `)`
  - **Usage**
    - evaluates a rule with the generalized m-estimate

- *getOptions*
  `public String` **getOptions**`( )`
  - **Usage**

* get the current configuration

– **Returns** - an array of strings suitable for passing to setOptions

---

- *listOptions*
  `public Enumeration` **listOptions**`( )`
  – **Usage**
    * returns an enumeration of the available options. which are:

      -M number

      The m-value used in the m-heuristic (Default: 2).

      -C number

      The cost trade-off in the heuristic. 0 <= c <= 1 (Default: 0.5).

- *setOptions*
  `public void` **setOptions**`( java.lang.String [] options )`
  – **Usage**
    * parse a list of options.
  – **Parameters**
    * `options` - the list of options as an array of strings
  – **Exceptions**
    * `java.lang.Exception` - if an option is not supported

C.8.6 Class Laplace

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the Laplace-estimate, i.e. (tp+1)/(tp+fp+2)

Declaration

public class Laplace
extends seco.heuristics.SearchHeuristic

Constructors

- *Laplace*
  `public` **Laplace**`( )`

METHODS

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with the Laplace estimate

C.8.7 CLASS LinearCosts

The seco package implements generic functionality for simple separate-and-conquer rule learning.

Linear Costs implements a class for evaluating a rule with a linear cost function (c*tp - (1-c)*fp), where 0 <= c <= 1.

c = 1 means that only covered positives counts, c = 0 means that only excluding negatives counts, values inbetween trade off between these two extremes. The default value of c is 0.5, which corresponds to the Accuracy heuristic.

The parameter can be canged via setOptions (the class implements the OptionHandler interface).

DECLARATION

public class LinearCosts
extends seco.heuristics.SearchHeuristic
implements weka.core.OptionHandler

FIELDS

- public double m_c
  –

CONSTRUCTORS

- *LinearCosts*
  `public` **LinearCosts**`( )`
  - **Usage**
    - Empty constructor, c will be set to 0.5.

- *LinearCosts*
  `public` **LinearCosts**`( double c )`
  - **Usage**
    - Constructor.
  - **Parameters**
    - `c` - the cost trade-off.

METHODS

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with a linear cost metric
- *getOptions*
  `public String` **getOptions**`( )`
  - **Usage**
    - get the current configuration
  - **Returns** - an array of strings suitable for passing to setOptions
- *listOptions*
  `public Enumeration` **listOptions**`( )`
  - **Usage**
    - returns an enumeration of the available options. which are:

      -c number

      The cost trade-off in the heuristic. 0 <= c <= 1 (Default: 0.5).
- *setOptions*
  `public void` **setOptions**`( java.lang.String [] options )`
  - **Usage**
    - parse a list of options.
  - **Parameters**
    - `options` - the list of options as an array of strings
  - **Exceptions**
    - `java.lang.Exception` - if an option is not supported

C.8.8 Class MEstimate

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the m-estimate, i.e. (tp+m*prior)/(tp+fp+m). The prior probability is (tp + fn) / (tp + fp + fn +tn). If you don't want to recompute this every time around, better use the GeneralizedM.

The default value of m is 2. It can be canged via setOptions (the class implements the OptionHandler interface).

Declaration

public class MEstimate
extends seco.heuristics.SearchHeuristic
implements weka.core.OptionHandler

Constructors

- *MEstimate*
  `public` **MEstimate**`( )`
  - **Usage**
    - Empty constructor, m will be set to 1.
- *MEstimate*
  `public` **MEstimate**`( double` **m** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `m` - the value for m

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule` **r** `)`
  - **Usage**
    - evaluates a rule with the m-estimate
- *getOptions*
  `public String` **getOptions**`( )`

  - **Usage**
    * get the current configuration
  - **Returns** - an array of strings suitable for passing to setOptions

---

- *listOptions*
  `public Enumeration` **listOptions**`( )`
  - **Usage**
    * returns an enumeration of the available options, which are:

      -M number

      The m-value used in the m-heuristic (Default: 2).

- *setOptions*
  `public void` **setOptions**`( java.lang.String [] options )`
  - **Usage**
    * parse a list of options.
  - **Parameters**
    * `options` - the list of options as an array of strings
  - **Exceptions**
    * `java.lang.Exception` - if an option is not supported

C.8.9 Class **Precision**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file provides evaluation for evaluating a rule with precision, i.e. tp/(tp+fp)

Declaration

public class Precision
extends seco.heuristics.SearchHeuristic

Constructors

- *Precision*
  `public` **Precision**`( )`

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    * evaluates the precision of the rule

C.8.10 Class **RateDiff**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with the difference of the true positive rate and the true negative rate.
For rules with the same example distribution ((tp + fn) and (fp + tn) are constant), this is eqivalent to weighted relative accuracy (WRAcc), but presumably faster.

Declaration

public class RateDiff
extends seco.heuristics.SearchHeuristic

Constructors

- *RateDiff*
  `public` **RateDiff**`( )`
  - **Usage**
    - Constructor

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with the difference of true positive rate and false negative rate.

C.8.11 Class **SearchHeuristic**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

SearchHeuristic implements a generic class for search heuristics for rule learning. A Search Heuristic takes a TwoClassStats confusion matrix and the predecessor rule as an argument and returns an evaluation. Most heuristics simply ignore the rule, but some will use it (e.g., information gain for getting the parent gain or MDL-based heuristics for getting the rule length)

For more details on rule learning search heuristics and their equivalences, see (Fürnkranz & Flach, ICML-03)

Declaration

public abstract class SearchHeuristic
extends java.lang.Object

Constructors

- *SearchHeuristic*
  `public` **SearchHeuristic**`( )`

Methods

- *evaluateRule*
  `public abstract double` **evaluateRule**`(`
  `seco.models.CandidateRule` **r** `)`
  - **Usage**
    - computes the evaluation for a possible rule.
  - **Parameters**
    - `r` - the candidate rule

- *forName*
  `public static SearchHeuristic` **forName**`( java.lang.String` **name**`, java.lang.String [] options )`
  - **Usage**
    - Creates a new instance of the heuristic given it's class name and (optional) arguments to pass to it's setOptions method. If the heuristic implements OptionHandler and the options parameter is non-null, the heuristic will have it's options set.
  - **Parameters**
    - `name` - the fully qualified class name of the heuristic
    - `options` - an array of options suitable for passing to setOptions. May be null.
  - **Returns** - the newly created heuristic, ready for use.
  - **Exceptions**
    - `java.lang.Exception` - if the classifier name is invalid, or the options supplied are not acceptable to the classifier

---

- *toOptionString*
  `public String` **toOptionString**`( )`
  - **Returns** - a string representation of a search heuristic, including all parameters (if any). If there are parameters, the returned string will start and end with quotes. Thus the representation is suitable for the command-line (e.g., for initializing other objects).

- *toString*
  `public String` **toString**`( )`

- **Returns** - a string representation of a search heuristic, including all parameters (if any). The parameters are added in parentheses, like a constructor call.

C.8.12 Class **WRAcc**

The seco package implements generic functionality for simple separate-and-conquer rule learning.

This file implements a generic class for evaluating a rule with weighted relative accuracy. weighted relative accuracy is (tp+fp)/total (tp/(tp+fp) - prior) where total is tp+fp+fn+tn and prior is the prior probability (tp+fn)/total.

You may also want to consider the equivalent but faster RateDiff. For more details on the the equivalences between search heuristics see (Fürnkranz & Flach, ICML-03).

Declaration

public class WRAcc
extends seco.heuristics.SearchHeuristic

Constructors

- *WRAcc*
  `public` **WRAcc**`( )`
  - **Usage**
    - Constructor

Methods

- *evaluateRule*
  `public double` **evaluateRule**`( seco.models.CandidateRule r )`
  - **Usage**
    - evaluates a rule with weighted relative accuracy