Technische Universität Darmstadt

Fachbereich Informatik

Fachgebiet Knowledge Engineering

Prof. Dr. Johannes Fürnkranz

Vergleich von Pruningalgorithmen für Regellerner

Diplomarbeit

eingereicht von

Benedict Werling

am

01. Juni 2008

Betreuer: Prof. Dr. Johannes Fürnkranz

Dipl.-Inf. Frederik Janssen

Eidesstattliche Erklärung

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

Darmstadt, 01. Juni 2008 Benedict Werling

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1: Einleitung

1.1 Motivation und Ziel

Mittlerweile existiert eine Vielzahl unterschiedlicher Separate-and-Conquer (Pruning-) Algorithmen [Fürn97, Fürn99], die sich durch bestimmte Eigenschaften voneinander unterscheiden. Alle diese Algorithmen haben eine gemeinsame inhärente Struktur. Anhand dieser Grundstruktur lassen sich die Eigenschaften der Algorithmen identifizieren und entsprechend implementieren. Um die Güte der Algorithmen festzustellen werden bestimmte Merkmale dieser Algorithmen, wie die Größe der gelernten Hypothese, die erzielte Genauigkeit oder das Zeitverhalten, untereinander verglichen. So lässt sich feststellen welcher Pruningalgorithmus am besten für eine bestimmte Problemstellung geeignet ist. Dennoch lassen sich die Pruningalgorithmen aufgrund ihrer verschiedenen Eigenschaften nur bedingt untereinander vergleichen. Zu diesem Zweck sollen die Pruningalgorithmen in einem einheitlichen Framework für Separate-and-Conquer Algorithmen implementiert und verglichen werden. Die Implementierung in einem einheitlichen Framework erleichtert zum einen den Vergleich und zum anderen die Untersuchung der verschiedenen Pruningalgorithmen.

Ein besonderes Merkmal, das in allen Lernalgorithmen vorhanden ist, ist die sogenannte Lernheuristik. Die Lernheuristik soll den Lernalgorithmus auf der Suche nach einer Hypothese durch den Suchraum zu einer Hypothese führen. Ein weiteres Merkmal der Heuristiken ist, dass diese schon während der Suche prunen, indem die Heuristiken Regeln bzw. Regelverfeinerungen einen Wert zuordnen und so bestimmen welche Regel in die Hypothese mit aufgenommen wird bzw. welche Regelverfeinerung als neuer Startpunkt der Suche gewählt wird. In der Regel werden Regeln besser bewertet, die viele positive Beispiele und wenig negative Beispiele abdecken. Durch die große Vielfalt an verschiedenen Heuristiken und deren unterschiedlichen Eigenschaften verhält sich auch ein Lernalgorithmus mit der Heuristik A anders als derselbe Lernalgorithmus mit der Heuristik B. Aufgrund der Tatsache, dass Heuristiken schon während dem Lernprozess prunen, wirft das die Frage auf:

„Ist Pruning[1] überhaupt noch notwendig, wenn die Heuristiken, die zum Lernen verwendet werden, schon sehr gute Ergebnisse erzielen?“

Ziel dieser Arbeit ist es die Pruningalgorithmen in einem einheitlichen Framework zu implementieren und zu untersuchen, inwiefern das Pruning in der Lage ist die Ergebnisse, die durch die Heuristiken erzielt werden, noch zu verbessern oder ob das Pruning keine Verbesserung mehr erzielen kann. Untersucht werden dazu das Prepruning und das inkrementelle Pre- und Postpruning in Verbindung mit den Heuristiken *Laplace*, *Precision*, *Accuracy*, *Weighted Relative Accuracy*, *m-Estimate*, *Klösgen-Maß* und *Correlation*. Für das m-Estimate und das Kloesgen-Maß werden insgesamt drei verschieden Parametereinstellungen betrachtet. Für die Pruningalgorithmen werden vier verschiedene Stopkriterien und zwei Pruningoperatoren genauer betrachtet. Um einen möglichst aussagekräftigen Vergleich zu erhalten werden die verschiedenen Algorithmenkonfigurationen auf insgesamt 57 nicht verrauschten Datensätzen getestet. Um auch eine Aussage über die Güte der gelernten (und geprunten) Hypothesen für verrauschte Daten treffen zu können, werden die 57 Datensätze zusätzlich mit 5% und 10% Rauschen versehen, womit insgesamt 171 zur Verfügung stehen. Die Algorithmen werden dann anhand ihres Zeitverhaltens, der Hypothesengröße und der durchschnittlichen erzielten Genauigkeit

[1] Die Worte Pruning und Prunen werden im weiteren Verlauf der Arbeit synonym verwendet.

verglichen. Um festzustellen welcher Algorithmus nun der Beste ist, werden die Ergebnisse auch anhand einer Win-Tie-Loss Tabelle festgehalten.

1.2 Überblick über die Kapitel

Das zweite Kapitel widmet sich den Grundlagen der Separate-and-Conquer Algorithmen. Zu Beginn des Kapitels wird die grundlegende Problemstellung des Lernens beschrieben. Im weiteren Verlauf werden die Separate-and-Conquer Algorithmen und ihre Merkmale (Kapitel 2.2) beschrieben und anhand eines generischen Algorithmus beschrieben (Kapitel 2.3). Am Ende des Kapitels (Kapitel 2.4) werden Methoden zum Vergleich von Lernalgorithmen kurz vorgestellt.

Das nachfolgende Kapitel (Kapitel 3) bietet einen kurzen Überblick über Heuristiken. Dazu werden wichtige Merkmale von Regeln und Trainingsmengen definiert (Kapitel 3.1). Im weitern Verlauf des Kapitels werden PN- bzw. ROC-Räume (Kapitel 3.2) vorgestellt, die für die Visualisierung der Heuristiken verwendet werden. Abschließend werden die Heuristiken, die in dieser Arbeit verwendet wurden charakterisiert und visualisiert (Kapitel 3.3).

In Kapitel 4 werden ausführlich die verschiedenen Mechanismen zur Vermeidung von Overfitting beschrieben. Besonderes Augenmerk liegt auf dem Prepruning und dem Inkrementellen Pre- und Postpruning.

Kapitel 5 beschreibt die Umsetzung der Pruningmechanismen im SeCo-Framework. Dazu wird ein generischer Separate-and-Conquer Pruningalgorithmus definiert und auf den generischen Algorithmus des Frameworks übertragen (Kapitel 5.1). Im weiteren Verlauf des Kapitels wird definiert, welche Objekte benötigt werden um das Pruning in das Framework zu integrieren und wie die Integration bewerkstelligt wurde (Kapitel 5.2). Am Ende des Kapitels wird die Integration von vier verschiedenen Algorithmen erläutert (Kapitel 5.4 bis Kapitel 5.7).

Das sechste Kapitel beschreibt ausführlich die verwendeten Testdatensätze und die verschiedenen Konfigurationen der Algorithmen, die in dieser Arbeit verglichen werden (Kapitel 6.1 und Kapitel 6.2). Der Rest des Kapitels dient dem Vergleich der Algorithmen untereinander. Am Ende des Kapitels werden die Ergebnisse zusammengefasst und es wird versucht die Frage zu beantworten:

„Ist Pruning überhaupt noch notwendig, wenn die Heuristiken, die zum Lernen verwendet werden, schon sehr gute Ergebnisse erzielen?“

Kapitel 7 fasst nochmals kurz die gesamten Ergebnisse der Arbeit zusammen und weist auf Probleme während dem Verlauf der Arbeit hin. Abschließen werden Ausblicke für neue interessante Ansätze dargestellt.

Kapitel 2 : Separate-and-Conquer Regellernen

In diesem Kapitel wird zuerst das grundlegende Lernproblem erläutert und grundlegende Definitionen vorgenommen. Anschließend wird das Lernproblem anhand der Separate-And-Conquer Strategie genauer beleuchtet. Die Strategie wird weiter durch einen generischen Algorithmus beschrieben und eine Unterscheidung der SeCo – Algorithmen wird entlang von drei Dimensionen (Verwendete Hypothesensprache, Suchalgorithmus für Hypothesen und Vermeidung von Overfitting) gemacht, wobei das Vermeiden von Overfitting hier ausgespart wird und im Kapitel 3 ausführlich erklärt wird.

2.1 Induktives Lernen

Das induktive Lernen beschäftigt sich mit dem Problem wie man eine allgemeine Funktion $h'(\vec{e})$ aus einer Menge E von spezifischen Trainingsdaten findet. Die Trainingsdaten enthalten die Eingabewerte $\vec{e}$ verknüpft mit den korrekten Ausgabewerte $h(\vec{e})$ der Zielfunktion. Im Allgemeinen ist die Zielfunktion eine boolesche Funktion, die einer bestimmten Repräsentationssprache unterliegt, und wird als Zielhypothese bzw. Zielkonzept bezeichnet. Anhand der Ausgabewerte $h(\vec{e})$ lassen sich die Beispiele in zwei Mengen, auch Klassen genannt, einteilen. Die Klassen geben an, welches Beispiel zum Zielkonzept gehört und welche nicht. Damit können die Trainingsdaten in zwei Mengen aufgeteilt werden, nämlich die Menge der positiven Beispiele $E^{\oplus}$, für die $h(\vec{e}) = 1$ ist und zum Zielkonzept gehören, und die Menge der negativen Beispiele $E^{\ominus}$, für die $h(\vec{e}) = 0$ ist und nicht zum Zielkonzept gehören. Im Folgenden wird angenommen, dass sich das Lernen auf 2-Klassen-Probleme beschränkt. In Kapitel 2.3 werden Methoden vorgestellt, wie das Lernen von Mehr-Klassen-Problemen funktioniert.

Wie zuvor schon durch den Vektorpfeil über einem Beispiel $\vec{e}$ angedeutet, ist ein Beispiel aus der Trainingsmenge ein Vektor von Attributen, wobei jedem Attribut ein Wert aus dessen Wertebereich zugeordnet ist.

Definition 2.1.1 (Attribut): Ein *Attribut A* besitzt einen Namen, einen Wertebereich und ist einer Attributklasse zugeordnet.

Die Attributklasse lässt sich anhand des Wertebereichs zuordnen. Besteht der Wertebereich aus einer unsortierten Liste von Symbolen (z.B.: gelb, groß, sonnig etc.) und unterliegt keiner eindeutigen Ordnungsrelation, so wird das Attribut der Klasse der n*ominalen* Attribute zugeordnet. Im Gegensatz dazu stehen *numerische* Attribute, deren Wertebereich aus Zahlen besteht und einer eindeutigen Ordnungsrelation, z.B. „>“, unterliegt. Ein Attribut eines Beispiels enthält i.A. die korrekten Werte h der Zielhypothese und wird als Klassenattribut bezeichnet. Im Folgenden wird angenommen, dass das Klassenattribut das letzte Attribut eines Trainingsbeispiels ist.

Definition 2.1.2 (Beispiel, Instanz): Ein Beispiel $\vec{e}$ ist ein Vektor von Attributen $(A_1, \dots, A_n), n \in \mathbb{N}$. Den Attributen ist dabei ein bestimmter Wert aus ihrem Wertebereich zugewiesen, wobei das letzte Attribut das Klassenattribut ist:

$$\vec{e} = (A_1 = Wert_1, \dots, A_{n-1} = Wert_{n-1}, A_n = h) \quad (2.1)$$

Die Gleichheitszeichen bedeuten im Beispielvektor eine Wertzuweisung und sind nicht als Vergleich zu betrachten. Anhand der Definition der Beispiele lässt sich nun definieren was die Beispiel- bzw. Trainingsmenge ist.

Definition 2.1.3 (Trainingsmenge, Beispielmenge): Die Trainingsmenge E ist eine Menge von Beispielen $\overrightarrow{e_k}, k \in \mathbb{N}$.

$$E = \{\overrightarrow{e_1}, \overrightarrow{e_2}, \overrightarrow{e_3}, \dots, \overrightarrow{e_k}\} \quad (2.2)$$

Eine Trainingsmenge könnte somit folgendermaßen aussehen:

Beispiel	Himmel	Temperatur	Luftfeuchtigkeit	Wasser	Wind	Schwimmen gehen?
1	sonnig	20	normal	warm	stark	ja
2	sonnig	22	hoch	warm	stark	ja
3	regnerisch	14	hoch	warm	stark	nein
4	sonnig	19	hoch	kalt	stark	ja

Tabelle 1: Beispiel einer Trainingsmenge. Teile aus [Mitc97].

Im Allgemeinen wird bei der Beschreibung von Beispielen in den Vektorkomponenten nur der Wert des jeweiligen Attributes angegeben. Ein Beispiel aus der Trainingsmenge von Tabelle 2.1 sieht damit so aus:

$$(sonnig, 20, normal, warm, stark, ja) \quad (2.3)$$

In der obigen Trainingsmenge sind die Attribute der Beispiele „Himmel, Temperatur, Luftfeuchtigkeit, Wind und Wasser“, die Klasse ist „Schwimmen gehen?“. Das Attribut Temperatur ist ein numerisches Attribut, wohingegen die anderen Attribute und das Klassenattribut ausschließlich nominale Attribute sind. Eine Besonderheit des Klassenattributes hier ist, dass es ein binäres Attribut ist. Es besteht nur aus zwei Werten („ja“, „nein“).

Das Lernen einer Funktion $h'(\vec{e})$, auch Hypothese bzw. Konzept genannt, kann als Suche durch den Raum aller möglichen Hypothesen betrachtet werden. Ziel eines induktiven Lernalgorithmus ist es dann eine Hypothese zu finden, die möglichst genau die Trainingsdaten beschreibt. D.h., dass die gefundene Hypothese vollständig (Definition 2.1.4) und konsistent (Definition 2.1.5) auf den Trainingsdaten ist.

Definition 2.1.4 (Konsistenz): Eine Hypothese ist konsistent auf den Trainingsdaten, wenn sie keine negativen Beispiele abdeckt.

Definition 2.1.5 (Vollständigkeit): Eine Hypothese ist vollständig, wenn sie alle positiven Beispiele abdeckt.

Die so gefundene Hypothese wird zur Klassifikation von Beispielen benutzt, die nicht in der Trainingsmenge enthalten waren bzw. dessen Klassenwert nicht bekannt ist. Als Grundlage für das Klassifizieren von nicht bekannten Beispielen dient folgende Annahme:

Annahme 2.1.6 (Induktives Lernen Hypothese): Eine Hypothese, die die Zielhypothese auf einer ausreichend großen Menge von Trainingsdaten hinreichend genau approximiert, approximiert die Zielhypothese auch auf vorher nicht bekannten Beispielen hinreichend genau [Mitc97].

Um festzustellen wie gut die gefundene Hypothese die Zielhypothese annähert, klassifiziert man mit der gefundenen Hypothese Beispiele aus einer separaten Testmenge. Die Testmenge enthält Beispiele, für die der korrekte Wert der Zielhypothese bekannt ist, aber dem Lernalgorithmus beim Lernen vorenthalten wurde. Anhand der richtig bzw. falsch klassifizierten Beispiele in der Testmenge, kann man die Genauigkeit (Definition 2.1.7) bzw. den Fehler (Definition 2.1.8) einer Hypothese errechnen:

Definition 2.1.7 (Genauigkeit): $Genauigkeit = \frac{Anzahl\ korrekt\ klassifizierter\ Beispiele}{Anzahl\ aller\ Beispiele}$

Definition 2.1.8 (Fehler): $Fehler = \frac{Anzahl\ falsch\ klassifizierter\ Beispiele}{Anzahl\ aller\ Beispiele} = 1 - Genauigkeit$

Im Regelfall steht eine solche Testmenge nicht zur Verfügung, sodass man nur die Genauigkeit bzw. den Fehler auf den Trainingsdaten hat. Um dennoch eine verlässliche Aussage über die gelernte Hypothese treffen zu können, bietet es sich an die Trainingsdaten vor dem Lernen aufzuteilen, damit eine separate Testmenge vorhanden ist. Eine andere Möglichkeit ist die Kreuzvalidierung (*Crossvalidation*) an.

Definition 2.1.9 (Kreuzvalidierung, Crossvalidation): Bei einer Crossvalidation wird die Trainingsmenge in n gleichgroße Teile aufgeteilt. Ein Teil der Aufteilung wird als Testmenge benutzt, die anderen n-1 Teile werden benutzt um eine Hypothese zu finden. Insgesamt werden n Hypothesen gelernt, wobei jeder Teil einmal als Testmenge benutzt wird, und die Ergebnisse des Klassifizierens werden gemittelt.

Die Crossvalidation wird auch als *n-fold Crossvalidation* bezeichnet, wobei der Wert *n* die Anzahl der Teile angibt. Eine in der Praxis übliche Aufteilung ist die 10-fold Crossvalidation. Ein Spezialfall der Crossvalidation ist die *N-fold Crossvalidation*, wobei *N* die Anzahl der Beispiele in der Trainingsmenge ist. Eine andere Art der *Crossvalidation* ist die sogenannte *stratified Crossvalidation.* Neben der Aufteilung der Trainingsmenge in *n* Teile, wird darauf geachtet, dass die Verteilung der Beispiele die Klassenverteilung in der ursprünglichen Trainingsmenge widerspiegelt.

Ist die Genauigkeit auf der Testmenge niedrig bzw. der Fehler hoch, so deutet das daraufhin, dass es während dem Lernen zu einer Überanpassung der Hypothese an die Trainingsdaten gekommen ist oder die Trainingsmenge verrauscht ist. Die so gefundene Hypothese ist meist wenig aussagekräftig. Man sagt, dass die Hypothese nicht gut verallgemeinert. Dies kann mehrere Gründe haben. Zum einen die oben genannten oder zum anderen die Aufteilung der Beispielmenge in Trainings- und Testmenge oder die Aufteilung der Beispielmenge bei einer Crossvalidation. Es kann passieren, dass sich dann wichtige Beispiele, die benötigt werden um die Zielhypothese zu beschreiben, in der Testmenge befinden.

Definition 2.1.10 (Overfitting, Überangepasstheit): Für eine gegebe Menge an Trainingsdaten und genügend Freiheitsgraden, lässt sich immer eine Hypothese finden, die die Trainingsdaten genau beschreibt. Die Trainingsdaten können dabei korrekt oder fehlerhaft sein.

Definition 2.1.11 (Noise, Rauschen): Fehler in der Trainingsmenge, wie falsche oder fehlende Wert von Beispielen.

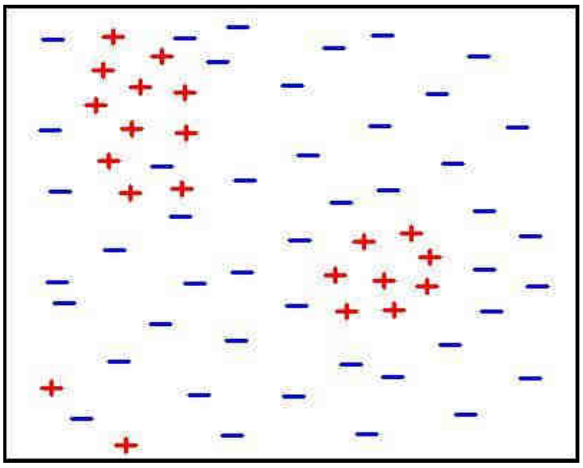

Abbildung 1: Visualisierte Trainingsmenge

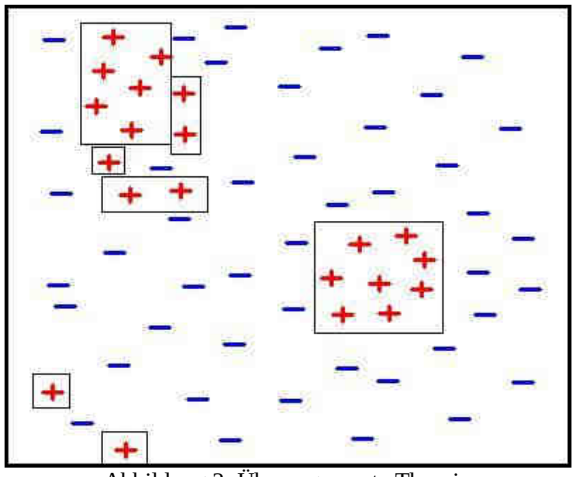

Abbildung 2: Überangepasste Theorie

Als Beispiel soll folgende Visualisierung (Abbildung 1) einer Trainingsmenge dienen. Die Minuszeichen entsprechen negativen und die Pluszeichen positiven Beispielen. Die schwarzen Vierecke entsprechen jeweils einer gelernten Regel. Eine überangepasste Hypothese könnte wie in Abbildung 2 aussehen.

Um ein Overfitting zu vermeiden oder die Auswirkung von verrauschten Daten zu vermindern, kann man versuchen die gefundene Hypothese noch zu verbessern oder die Konsistenz- und Vollständigkeitsnebenbedingungen zu lockern, sodass auch einige negative Beispiele abgedeckt werden dürfen. Dieser Vorgang wird im Allgemeinen als Pruning bezeichnet. Eine so gelernte Hypothese ist im Allgemeinen kleiner als eine überangepasste Hypothese. Die Hypothese in Abbildung 2 besteht aus 7 Regeln, wohingegen die Hypothese aus Abbildung 3 nur 3 Regeln umfasst.

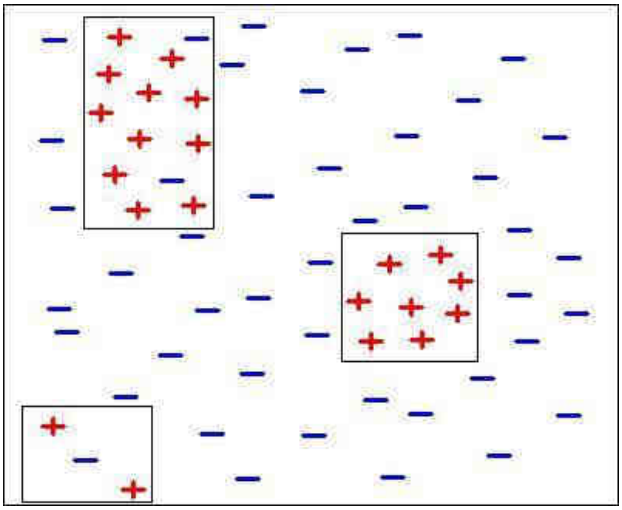

Abbildung 3: Geprunte Hypothese

Die nachfolgenden Abschnitte behandeln die Merkmale von Separate-and-Conquer Algorithmen genauer. Der Abschnitt 2.2.1 beleuchtet eine spezielle Hypothesensprache, die in dieser Arbeit benutzt wird, genauer. Der Schwerpunkt des nächsten Abschnitts (2.2.2) liegt auf den verschiedenen Suchverfahren, die benutzt werden um eine Hypothese zu lernen. Das Suchverfahren gliedert sich in drei Bereiche, den Suchalgorithmus, die Suchheuristik und die Suchstrategie. Die Suchheuristiken und ihre Charakteristika werden noch genauer in Kapitel 3 beleuchtet. Das Vermeiden von Overfitting wird kurz in Abschnitt 2.2.3 und genauer in Kapitel 4 beschrieben. Im Anschluss an diese Abschnitte, im Abschnitt 2.4, wird ein generischer Separate-and-Conquer Algorithmus vorgestellt und erklärt wie

die verschiedenen Merkmale implementiert werden können. Der letzte Abschnitt (2.5) befasst sich damit, wie man verschiedene Lernalgorithmen vergleichen kann.

2.2 Merkmale von Separate-and-Conquer Algorithmen

Separate-and-Conquer ist die Bezeichnung für eine Strategie wie ein induktiver Lernalgorithmus die Zielhypothese annähert. Diese Strategie geht auf die Algorithmen der AQ-Familie [Mich69] zurück, wo sie unter dem Namen *Covering-Strategie* eingeführt worden ist. Der Begriff Separate-and-Conquer ist von Pagallo und Haussler eingeführt worden [PaHa90]. Ein Separate-and-Conquer Algorithmus lernt zunächst eine Regel, die einen Teil der positiven Beispiele der Trainingsdaten beschreibt. Danach werden die Beispiele, welche die Regel abdeckt, aus der Trainingsmenge entfernt (Separate–Schritt) und der Algorithmus versucht rekursiv eine weitere Regel zu lernen, die die restlichen Trainingsbeispiele beschreiben (Conquer-Schritt). Die Separate- und Conquer-Schritte werden solange wiederholt, bis jedes positive Beispiel von mindestens einer Regel abgedeckt wird [Fürn99].

Obwohl die Schritte bei jedem Separate-and-Conquer Algorithmus die gleichen sind, gibt es eine Vielzahl verschiedener Separate-and-Conquer Algorithmen. Eine Auflistung findet sich in [Fürn99]. Die verschiedenen Algorithmen werden anhand von 3 Dimensionen charakterisiert.

2.2.1 Hypothesensprache

Die erste Dimension an der Separate-and-Conquer Algorithmen charakterisiert werden ist die *Hypothesensprache*, die dem Algorithmus zugrunde liegt, und das zu lernende Konzept beschreibt. Ein Separate-and-Conquer Algorithmus lernt Regeln zur Beschreibung der Zielhypothese. Die wichtigsten Hypothesensprachen, die man bei Separate-and-Conquer Algorithmen unterscheiden kann, sind dabei Regelmengen in *disjunktiver Normalform (DNF) (Definition 2.2.5) oder konjunktiver Normalform (KNF) (Definition 2.2.6), Entscheidungslisten, Logikprogramme, Regressionsregeln oder Funktionale Relationen* [Fürn99], je nachdem welche Zielhypothese gelernt werden soll. Für *kontinuierliche bzw. numerische* Zielhypothesen eigenen sich vor allem Regressionsregeln, für *nominale* Zielhypothesen eignen sich Regelmengen bzw. Entscheidungslisten.

Ein weiteres Merkmal der Hypothesensprache ist, ob sie dynamisch oder statisch ist. Bei den statischen Hypothesensprachen kommt es schnell dazu, dass der Suchraum exponentiell groß wird, und eine Hypothese nicht mehr effizient oder gar nicht gefunden werden kann. Ein weiteres Problem bei statischen Hypothesensprachen ist eine zu kurzsichtige Suche, da manche Suchalgorithmen einer Regel immer nur eine Bedingung hinzufügen. Die Größe des Suchraums wird durch die Anzahl der Attribute und Attributwerte festgelegt. Damit eine Hypothese in einem sehr großen Suchraum, dennoch effizient gefunden werden kann, kann man die maximale Anzahl von Bedingungen in einer Regel einschränken. Um eine zu kurzsichtige Suche zu vermeiden, ist es auch zulässig mehrere konjungierte Bedingungen einer Regel bei der Suche hinzuzufügen. Weitere Methoden um den Suchraum einzuschränken finden sich in [Fürn99].

Ist die Wahl einer Hypothesensprache ungünstig gewählt, so ist es bei statischen Hypothesensprachen im Nachhinein nicht mehr möglich diese zu ändern, ohne dass der Benutzer eingreift. Um dieses Problem zu umgehen kann man die verschiedenen Hypothesensprachen nach ihrer Aussagekraft sortieren und speichern. Die Suche nach der Zielhypothese beginnt mit der Hypothesensprache mit der niedrigsten Aussagekraft. Wurde mit dieser keine zufriedenstellende Theorie gelernt, so wird mit der nächsten Hypothesensprache fortgefahren. Dies ist ein Merkmal der dynamischen Hypothesensprachen. Anstatt alle Hypothesensprachen in einer Liste zu speichern, kann man automatisch neue Merkmale zu einer Hypothesensprache hinzufügen, die aus den Trainingsdaten mittels arithmetischer oder logischer Operatoren errechnet werden.

Die Hypothesensprache, die in dieser Arbeit verwendet wird, beschränkt sich auf Regelmengen, mit besonderem Augenmerk auf Entscheidungslisten, da der Algorithmus der benutzt wird auf Entscheidungslisten basiert. Dazu werden hier grundlegende Begriffe, wie Regel, Regelmenge, etc., festgelegt um dann zu definieren was eine Entscheidungsliste ist.

Entscheidungslisten und Regelmengen bestehen aus Regeln, die im Folgenden definiert werden.

Definition 2.2.1 (Regel): Eine Regel r besteht aus einem Kopf und einem Körper (Gleichung 2.3). Der Regelkörper besteht aus **Attributtests** (siehe Definition 2.2.5), die entweder konjunktiv oder disjunktiv miteinander verknüpft sind. Der Regelkopf besteht aus der Vorhersage eines Klassenwerts.

$$Regel\ r\colon Regelkörper :- Regelkopf \quad (2.3)$$

Definition 2.2.2 (Attributtest): Ein Attributtest ist ein Vergleich eines Attributes mit einem Wert aus dem Wertebereich des Attributes, der fordert, dass das Attribut mit diesem Wert übereinstimmt oder dieser Wert den Wertebereich des Attributs beschränkt.

Für Attributtest werden verschiedene Vergleichsoperatoren, auch Komparatoren genannt, verwendet. Die wichtigsten Komparatoren sind für nominale Attribute der Gleichheitsoperator $=$ bzw. der Ungleichheitsoperator $\neq$. Für numerische Attribute gelten die Komparatoren $<, \leq, >$ und $\geq$, die den Wertebereich eines Attributes einschränken. Eine Regel könnte damit wie in Gleichung 2.4 aussehen.

$$Wasser = warm \ \wedge Temperatur \geq 14 \ \wedge Wind = stark :- Sport\ macht\ Spaß? = ja \quad (2.4)$$

Der Klassenwert des Regelkopfs wird für ein Beispiel vorhergesagt (Definition 2.2.3), wenn die Regel das Beispiel abdeckt (Definition 2.2.4).

Definition 2.2.3 (Vorhersagen): Wird ein Beispiel von einer Regel abgedeckt, so wird der Wert im Regelkopf für das Beispiel vorhergesagt.

Definition 2.2.4 (Abdecken): Eine Regel deckt ein Beispiel ab, wenn der Körper der Regel wahr ist. Also wenn jeder Attribut-Wert-Vergleich wahr ist bei konjunktiven Regeln.

Mit diesen einfachen Definitionen lassen sich nun Regelmengen (Definition 2.2.7) definieren. Eine Regelmenge ist entweder in DNF (Definition 2.2.5) oder in KNF (Definition 2.2.6). Beide Repräsentationen sind äquivalent, da sich die DNF durch zweifache Negation in die KNF und die KNF durch zweifache Negation in DNF überführen lässt.

Definition 2.2.5 (disjunktive Normalform, DNF): Die disjunktive Normalform ist eine Disjunktion von konjunktiv verknüpften Literalen $L_{i,j}$. Die Literale $L_{i,j}$ bestehen dabei aus Attributtests:

$$\bigvee_i \bigwedge_j L_{i,j} = \left(L_{1,1} \wedge L_{1,2} \wedge \ldots \wedge L_{1,j}\right) \vee \ldots \vee \left(L_{i,1} \wedge L_{i,2} \wedge \ldots \wedge L_{i,j}\right) \quad (2.5)$$

Definition 2.2.6 (konjunktive Normalform, KNF): Die konjunktive Normalform ist eine Konjunktion von disjunktiv verknüpften Literalen $L_{i,j}$. Die Literale $L_{i,j}$ bestehen dabei aus Attributtests:

$$\bigwedge_i \bigvee_j L_{i,j} = \left(L_{1,1} \vee L_{1,2} \vee \ldots \vee L_{1,j}\right) \wedge \ldots \wedge \left(L_{i,1} \vee L_{i,2} \vee \ldots \vee L_{i,j}\right) \quad (2.6)$$

Definition 2.2.7 (Regelmenge): Eine Regelmenge ist eine Menge von Regeln, die entweder in KNF oder in DNF ist. Eine Regelmenge in KNF (Gleichung 2.7) besteht aus konjunktiv verknüpften Regeln, deren Attributtests disjunktiv verknüpft sind. Eine Regelmenge in DNF (Gleichung 2.6) besteht aus disjunktiv verknüpften Regeln, deren Attributtests konjunktiv verknüpft sind.

$$R = \{r_1 \vee r_2 \vee \ldots \vee r_n\} \quad (2.6)$$

$$R = \{r_1 \wedge r_2 \wedge \ldots \wedge r_n\} \quad (2.7)$$

Die Entscheidungsliste ist auch eine Regelmenge. Allerdings ist diese Regelmenge sortiert und die Reihenfolge, in der die Regeln in der Regelmenge enthalten sind, ist wichtig. Bei der Klassifikation von Beispielen wird der Reihe nach für jede Regel geprüft, ob diese das Beispiel abdeckt. Deckt eine Regel ein Beispiel ab, so wird der Klassenwert der Regel für das Beispiel vorhergesagt und die übrigen Regeln werden ignoriert. Desweiteren wird eine spezielle Regel, die sogenannte Defaultregel (Definition 2.2.8) die die Defaultklasse (Definition 2.2.9) für ein Beispiel vorhersagt, zu der Entscheidungsliste hinzugefügt.

Definition 2.2.8 (Defaultregel): Eine Defaultregel ist eine Regel deren Regelkörper keine Attributtests enthält, sondern nur den Wert *true*. Die Defaultregel deckt jedes Beispiel ab und sagt für ein Beispiel eine Defaultklasse (Definition 2.2.6) vorher.

Definition 2.2.9 (Defaultklasse): Die Defaultklasse ist die Klasse, deren Klassenwert am häufigsten in der Trainingsmenge vorkommt. Die Defaultklasse wird auch als Majorityklasse bezeichnet.

Mit den vorangegangenen Definitionen lässt sich eine Entscheidungsliste folgendermaßen definieren:

Definition 2.2.10 (Entscheidungsliste): Eine Entscheidungsliste ist eine sortierte Regelmenge. Die letzte Regel einer Entscheidungsliste ist die Defaultregel. Die Klassifizierung von Beispielen wird anhand der Reihenfolge der Regeln in der Liste gemacht. Sobald die erste Regel feuert wird einem Beispiel die Klasse der Regel zugeordnet. Regeln, die später in der Liste vorkommen, werden ignoriert.

$$R = \begin{Bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \\ true :- Defaultklasse \end{Bmatrix} \quad (2.8)$$

In den nachfolgenden Kapiteln beschränken sich die Beschreibungen und Definitionen auf konjunktive Regeln und Entscheidungslisten, da die Algorithmen, die in dieser Arbeit verglichen werden auf dieser Hypothesensprache basieren.

2.2.2 Suchverfahren

Das Suchverfahren, das ein Separate-and-Conquer Algorithmus verwendet, ist die zweite Dimension, die diese Algorithmen charakterisiert. Das Suchverfahren eines Separate-and-Conquer Algorithmus legt fest, wie der Hypothesenraum beim Lernen einer Theorie durchsucht wird. Das Suchverfahren eines Separate-and-Conquer Algorithmus unterscheidet drei Teile: Der Suchalgorithmus, die Suchstrategie und die Suchheuristik.

Der Suchalgorithmus steuert, wie der Name schon impliziert, die Suche. Eine Option ist es, einfach alle Regeln, die bei der Suche nach einer Hypothese erzeugt werden, zu generieren und nur die Regeln zu nehmen, die nur positive Beispiele abdecken. Eine andere Option wäre den Hypothesenraum erschöpfend nach einer Hypothese zu durchsuchen. Beide Optionen sind aber sehr ineffizient und werden meist nur zu theoretischen Zwecken implementiert. [Fürn99]

Die Algorithmen, die hauptsächlich bei Separate-and-Conquer Algorithmen verwendet werden, sind *Hill-Climbing, Beam Search, Best-First Search oder Stochastische Suche.* Das Hill-Climbing durchsucht den Hypothesenraum, indem es eine Regel sukzessive verfeinert bis ein Optimum erreicht ist. Die Regel wird dazu entweder generalisiert (Definition 2.2.11) oder spezialisiert (Definition 2.2.12). Dabei erzeugt die Suche alle möglichen Regeln, die von der momentanen Regel aus erzeugbar sind. Aus den erzeugten Regeln wird die beste Regel ausgewählt und die Suche startet erneut mit der besten Regel als Startpunkt.

Definition 2.2.11 (Generalisierung): Eine Regel wird generalisiert, indem ein oder mehrere Attributtests aus dem Regelkörper entfernt werden. Die Regel deckt somit mehr Beispiele ab.

Definition 2.2.12 (Spezialisierung): Eine Regel wird spezialisiert, indem ein oder mehrere Attributtests dem Regelkörper hinzugefügt werden. Die Regel deckt somit weniger Beispiele ab.

Die Hill-Climbing Suche versucht zwar ein globales Optimum zu finden, kann aber auch leicht in einem lokalen Optimum stecken bleiben. Dies geschieht, da die Suche nur einen Schritt voraussieht, d.h. die beste erzeugte Regel weiter verfeinert und die anderen Regeln vergisst. Ist das globale Optimum aber eine Verfeinerung einer vergessenen Regel, so läuft die Suche am globalen Optimum vorbei. Um diese „Kurzsichtigkeit“ der Suche zu umgehen wird die Suche so verändert, dass in einem Schritt nicht nur eine sondern *n* Regeln simultan verfeinert werden.

Im Gegensatz zum Hill-Climbing speichert die Beam-Suche nicht nur eine verfeinerte Regel, sondern die b besten Regeln in dem sogenannten Strahl (Beam). Mit der Beam-Suche will man die Auswirkung der Kurzsichtigkeit einer Hill-Climbing Suche vermindern. Dadurch, dass man einen größeren Teil des Hypothesenraums durchsucht, ist die Wahrscheinlichkeit ein globales Optimum zu finden höher bzw. die Wahrscheinlichkeit in einem lokalen Optimum festzuhängen geringer. Setzt man allerdings die Beamgröße auf $b = 1$, so erhält man wieder eine Hill-Climbing Suche. Wählt man dagegen einen zu großen Strahl, dann verschlechtern sich die Ergebnisse der Beam-Suche je nach verwendeter Suchheuristik [JaFü08].

Wird die Beamgröße bei der Beam-Suche auf $b = \infty$ festgelegt, kommt man zur Best-First-Suche. Dies führt dazu, dass der Suchraum erschöpfend durchsucht wird. Dieser Algorithmus vermeidet das Problem einer zu kurzsichtigen Suche und findet eine optimale Hypothese. Eine erschöpfende Suche ist aber sehr ineffizient, da der gesamte Hypothesenraum durchsucht wird. Die Suche kann mit dem A*-Algorithmus weiter eingeschränkt werden, indem Teile des Suchraums nicht betrachtet werden ohne dabei die Optimalität der Hypothese zu verlieren [Fürn99, HaNR68] .

Eine andere Methode den Hypothesenraum nach einer geeigneten Hypothese zu durchsuchen, ist die stochastische Suche. Das stochastische Hill-Climbing verfeinert eine Regel nicht nur Schritt für Schritt, sondern die Regel kann in einem Schritt öfters verfeinert werden. Diese Verfeinerungen werden mit einer bestimmten Wahrscheinlichkeit ausgeführt und ermöglichen es der Suche auch in andere Bereiche des Suchraums zu gelangen, die bei einer Hill-Climbing Suche nicht beachtet werden. Eine andere Möglichkeit ist es, nicht immer die beste Regel zu verfeinern, sondern auch suboptimale Regeln für zur Verfeinerung zu zulassen. Dazu wird die Wahrscheinlichkeit, dass eine Regel für einen

nächsten Schritt ausgewählt wird, mit dem Heuristikwert der Regel verknüpft. So werden gute Regeln häufiger und suboptimale Regeln weniger häufig ausgewählt. Das Simulated Annealing [Fürn99] verknüpft die Wahrscheinlichkeit, dass suboptimale Regeln ausgewählt werden mit der Zeit, sodass die Wahrscheinlichkeit mit fortlaufender Zeit abnimmt bis sie sich stabilisiert und keine suboptimalen Regeln mehr zugelassen werden.

Genetische Algorithmen [Fürn99], die auch zur stochastischen Suche zählen, verfolgen einen ganz anderen Ansatz. Die Suche speichert *s* Regeln, die sogenannte Generation, und versucht neue Nachkommen dieser *s* Regeln zu bilden, indem die Regeln generalisiert oder spezialisiert werden oder zwei Regeln tauschen einen oder mehrere zufällig ausgewählte Attributtests. Aus der Startgeneration und den neu gebildeten Regeln werden dann die *s* besten Regeln ausgesucht. Diese Regeln bilden die Generation für den nächsten Schritt der Suche. Dies geschieht solange, bis eine Regel eine vorher bestimmte Anzahl von Generationen „überlebt“. Diese Regel wird dann zur Hypothese hinzugefügt.

Die Suchstrategie eines Suchalgorithmus steuert die Art wie der Hypothesenraum nach einer geeigneten Hypothese durchsucht wird. Unterschieden werden dabei drei Vorgehensweisen. Die Top-Down Strategie beginnt bei der generellsten Regel und spezialisiert diese sukzessive, d.h. sie fügt der Regel in jedem Schritt einen Attributtest hinzu. Im Gegensatz dazu steht die Bottom-Up Strategie, die eine Regel, ausgehend von der speziellsten Regel, sukzessive generalisiert. Dazu werden von der speziellsten Regel immer mehr Attributtests entfernt. Die letzte Möglichkeit die Suche zu steuern ist eine Kombination aus den beiden oben genannten Strategien, die sogenannte bidirektionale Suche. Die bidirektionale Suche startet bei der generellsten und bei der speziellsten Regel. Diese Regeln werden sukzessive verfeinert, bis die Suche in einem Punkt zusammenläuft. Dieser Punkt ist im Allgemeinen eine Regel, die genereller als die speziellste, aber spezieller als die generellste Regel ist.

Die letzte Möglichkeit die Suche zu steuern ist die Suchheuristik, die auch den meisten Einfluss auf die Suche hat. Die Suchheuristik schätzt die Güte einer Regel, die durch verschiedene Eigenschaften der Regel festgelegt wird, und führt die Suche in die richtige Richtung im Suchraum. Die meisten Heuristiken schätzen die Güte einer Regel anhand der positiven und negativen Beispiele, die eine Regel abdeckt. Diese Abschätzungen können allerdings zu optimistisch ausfallen, da sie auf den Trainingsdaten gemacht werden und sich die Trainingsmenge von der Testmenge unterscheiden kann, d.h. die zugrunde liegende Hypothese, die zur Erstellung beider Mengen verwendet wurde kann unterschiedlich sein. Eine andere Variante um die Güte einer Regel zu schätzen wurde in [JaFü07] verfolgt. Hier wurde ein Neuronales Netz trainiert um die Genauigkeit der auf den Trainingsdaten gelernten Regeln auf der Testmenge vorherzusagen. Dieses Netz wurde dann als Suchheuristik verwendet. In Kapitel 3 werden Suchheuristiken genauer erläutert.

2.2.3 Methoden zu Vermeidung von Overfitting

Ein Separate-and-Conquer Algorithmus ist bis jetzt in der Lage konsistente und vollständige Theorien zu lernen. Was passiert aber wenn die Daten verrauscht sind oder die Trainingsdaten nur einen kleinen Teil der Zielhypothese beschreiben? In diesem Fall beschreibt die gelernte Hypothese die Trainingsdaten genau, verallgemeinert aber schlecht auf vorher nicht gesehenen Beispielen. Um diese Überanpassung zu vermeiden versucht man die gelernte Hypothese zu prunen. D.h. dass der Algorithmus versucht nicht nur konsistente und vollständige Regeln zu lernen, sondern auch Regeln, die einige negative Beispiele abdecken. Die so gelernte Hypothese deckt fast alle positiven und einige negative Beispiele ab. Es gibt mehrere Möglichkeiten um eine Hypothese zu prunen. Zum einen kann eine geeignete Suchheuristik während des Lernens dazu beitragen. In der Regel bevorzugen Heuristiken kurze, allgemeine Regel gegenüber langen, speziellen Regeln, auch wenn die Genauigkeit

der kurzen Regeln schlechter als die Genauigkeit der speziellen Regeln ist, in der Hoffnung, dass die Genauigkeit der allgemeinen Regeln auf nicht gesehenen Daten höher ist [Fünr99]. Zum anderen gibt es auch verschiedene Methoden, wie z.B. Pre- und Post-Pruning, die helfen eine überangepasste Theorie wieder zu verallgemeinern. Die verschiedenen Arten des Prunings werden im Kapitel 4

```
procedure SeparateAndConquer(Examples)

Theory = ∅
while POSITIVE(Examples) ≠ ∅
    Rule = FindBestRule(Examples)
    Covered = COVER(Rule)
    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)
            unless STOPPINGCRITERION(Refinement, Evaluation, Examples)
                NewRule = <Evaluation, Refinement >
                Rules = INSERTSORT(NewRule, Rules)
                if NewRule > BestRule
                BestRule = NewRule
    Rules = FILTERRULES(Rules,Examples)
return(BestRule)
```

Abbildung 4: Ein generischer Separate-and-Conquer Algorithmus [Fürn99]

ausführlich beschrieben.

2.3 Ein generischer Separate-and-Conquer Algorithmus

In [Fürn99] wurde ein generischer Separate-and-Conquer Algorithmus eingeführt, der eigentlich alle Algorithmen dieser Familie vereint und in Abbildung 4 zu sehen ist. Die eigentlichen Algorithmen unterscheiden in der Implementierung der Subroutinen, die der Algorithmus während des Lernens aufruft. Der generische Algorithmus gliedert sich in zwei Prozeduren. Die erste Prozedur (*Separate-and-Conquer*) bildet die äußere Schleife für den Algorithmus. Die Prozedur startet mit einer leeren Hypothese und fügt der Hypothese Regeln hinzu. Die Prozedur *FindBestRule* sucht nach geeigneten Kandidaten, die der Hypothese hinzugefügt werden sollen. Nachdem eine Regel gelernt wurde,

werden die Beispiele, die von einer Regel abgedeckt werden, aus der Menge der Trainingsbeispiele entfernt. Diese Schritte werden solange wiederholt, bis keine positiven Beispiele mehr in der Trainingsmenge vorhanden sind (für den Fall einer konsistenten und vollständigen Hypothese) oder bis das Lernen durch das *RuleStoppingCriterion* beendet wird. Am Ende des Lernens wird die gelernte Hypothese meist noch nachbearbeitet (*PostProcess*). Die Prozedur *Separate-and-Conquer* entspricht dabei dem Separate-Schritt der Algorithmen.

Die zweite Prozedur (*FindBestRule*) implementiert das Suchverfahren (siehe Kapitel 2.2.2), nach dem der Hypothesenraum durchsucht wird. Für die Implementierung des Suchalgorithmus sind die Prozeduren *SelectCandidates* und *FilterRules* zuständig. Die Suchstrategie wird durch die Prozedur *InitializeRule* und *RefineRule*, die Suchheuristik durch die Prozedur *EvaluateRule* implementiert. *FindBestRule* versucht eine Regel zu finden, deren Heuristikwert möglichst optimal ist. Dazu initialisiert die Prozedur eine Regel, die den Startpunkt der Suche markiert, setzt diese erste Regel als beste Regel fest und fügt sie einer Liste von Regeln hinzu. Solange die Liste der Regeln nicht leer ist, erzeugt die Prozedur eine Menge von Kandidatenregeln (*SelectCandidates*). Diese Kandidatenregeln werden verfeinert (*RefineRule*) und ihr Heuristikwert wird bestimmt (*EvaluateRule*). Jede der neu erzeugten Regeln wird anhand ihres Heuristikwertes an die richtige Stelle in die Liste der Regeln einsortiert (*InsertSort*). Wenn eine der verfeinerten Regeln besser ist als die momentan beste Regel, so wird die neue Regel als beste Regel definiert. Diese Schritte werden solange wiederholt bis entweder alle Verfeinerungen der Regel erzeugt, alle Kandidatenregeln verfeinert wurden oder das *StoppingCriterion* die Regelverfeinerungen abbricht. Zum Schluss wird ein Teil der neu erzeugten Regeln ausgewählt (*FilterRules*) und die Prozedur startet erneut bei den Regelverfeinerungen. Als Beispiel nehmen wir an, dass die Größe der Regelmenge auf 1 beschränkt wird, die erste Regel, bei der die Suche startet, alle Beispiele abdeckt und die Regeln spezialisiert werden. Die *StoppingCriterion*-Prozeduren geben immer *false* zurück und die Regeln werden so bewertet, dass die Regeln mit der höchsten Genauigkeit gewählt werden. Diese Beschränkungen entsprächen einer Top-Down Hill-Climbing Suche.

Bei Regelmengen, die der Beschreibung von 2-Klassen Problemen dienen, ist es unerheblich wie die Regeln in der Regelmenge enthalten sind. Deckt eine Regel innerhalb der Regelmenge ein Beispiel nicht ab, so wird dieses als negatives Beispiel klassifiziert (*negation as failure*). Dient die Regelmenge dagegen zur Beschreibung eines Problems mit mehr als zwei Klassen, ist die Sortierung der Regeln innerhalb der Regelmenge wichtig, da ein Beispiel von mehreren Regeln innerhalb einer Regelmenge abgedeckt werden kann und diese Regeln für ein Beispiel mehrere Klassen vorhersagen können. Das Problem, dass mehrere Regeln ein Beispiel abdecken können, wird in [Fürn99] auch als *overlap problem* bezeichnet. Es entsteht dadurch, dass Attributtests von Regeln gleich sind, sich sozusagen überlappen. Um diesem Problem bei der Klassifikation entgegenzusteuern kann man beim Lernen nur Regeln zulassen, die homogen sind und diese in einer homogenen Entscheidungsliste speichern. Wie in [SeEt94] gezeigt, ist eine homogene Regel, eine Regel deren Spezialisierungen denselben Heuristikwert besitzen. Auch wurde in [SeEt94] gezeigt, dass eine homogen Entscheidungsliste in eine logisch äquivalente nicht-homogene Entscheidungsliste transformiert werden kann. Der Vorteil von homogenen Entscheidungslisten ist, dass bei diesen die Sortierung der Regeln unerheblich ist.

Bis jetzt wurde angenommen, dass sich das Lernen ausschließlich auf 2-Klassen-Probleme beschränkt. Aber viele Lernprobleme lassen sich nicht mit zwei Klassen beschreiben, sondern werden durch drei, vier oder mehr, z.B. *n* Klassen beschrieben. Damit die Algorithmen mit Mehr-Klassen-Problemen umgehen können, ist es notwendig die *n* Klassen in eine Folge von 2-Klassen Problemen zu transformieren (*Class binarization*). Die erste Möglichkeit dies zu bewerkstelligen ist es Regeln ohne besonderes Augenmerk auf die Klassen zu lernen. Die Klasse die eine Regel für ein Beispiel

vorhersagt, ist die Majorityklasse der Beispiele in der Trainingsmenge, die von der Regel abgedeckt wird. Eine andere Möglichkeit bietet die sogenannte *One-against-all* Strategie, die jede Klasse für sich lernt, indem sie die Beispiele der zu lernenden Klasse als positive Beispiele betrachtet und alle anderen Beispiele als negative. Die *Round-Robin* Strategie bietet eine weitere Möglichkeit mit Mehrklassen-Problemen umzugehen. Bei diesem Verfahren wird jede Klasse gegen jede andere Klasse gelernt. In [Fürn01] wurde gezeigt, dass diese Strategie zu einer Erhöhung der Genauigkeit führen kann, aber mindestens genauso gut ist wie die anderen Strategien.

2.4 Vergleich von Separate-and-Conquer Algorithmen

Anhand des generischen Algorithmus aus Abbildung 4 lassen sich viele verschiedene Separate-and-Conquer Algorithmen definieren. Hat man nun einen Algorithmus ausgehend von Abbildung 4 festgelegt, indem man das Suchverfahren, Stopkriterien etc. festgelegt hat, ist es interessant festzustellen, wie gut der Algorithmus in Wirklichkeit ist. Die Güte eines Algorithmus lässt sich dabei auf verschiedene Arten feststellen und dient damit als Verlgeichsmaß mit anderen Algorithmen.

Die erste Möglichkeit ist die Genauigkeit des Algorithmus festzustellen (Definition 2.1.7). Die Genauigkeit kann entweder auf den Trainingsdaten oder auf einer separaten Testmenge festgestellt werden. Da aber im Regelfall die Genauigkeit auf der Trainingsmenge sehr hoch ist und eine extra Testmenge meist nicht zu Verfügung steht, bietet sich hier eine Kreuzvalidierung (Definition 2.1.9) an.

Eine weitere Möglichkeit die Güte eines Algorithmus zu bestimmen ist die Größe der gelernten Hypothese. Eine kleine Hypothese, die alle positiven Beispiele abdeckt, ist natürlich einer großen Hypothese vorzuziehen. Die kleine Hypothese wird weniger an Overfitting leiden und ist leichter verständlich als eine große Hypothese. Die zugrundeliegenden Parameter, die hier als Gütemaß dienen, sind die Anzahl der Regeln und die Anzahl der Attributtest der Hypothese. Aus diesen Parametern lässt sich die durchschnittliche Regellänge errechnen, die ein weiteres Maß für die Güte eines Algorithmus ist.

Die letzte Möglichkeit, mit der man die Güte eines Algorithmus feststellen kann, ist das Laufzeitverhalten des Algorithmus auf der Trainingsmenge. Das Laufzeitverhalten steigt in der Regel mit der Größe der Trainingsmenge. Ein Algorithmus, der schnell eine Hypothese findet ist einem langsamen Algorithmus dabei vorzuziehen.

2.4.1 Win-Tie-Loss Tabellen

Die Win-Tie-Loss Tabelle ist ein Mittel um mehrere Algorithmen miteinander zu vergleichen. Um die Tabellen zu erstellen lässt man mehrere Algorithmen auf einem oder mehreren Trainingsmengen lernen und stellt danach die Genauigkeit der Algorithmen fest. Im Anschluss daran zählt man wie oft ein Algorithmus besser (Win) bzw. schlechter (Loss) als ein anderer Algorithmus auf den Trainingsmengen ist. D.h. wie oft erreicht Algorithmus A eine höhere Genauigkeit als Algorithmus B auf der Beispielmenge $i, i \in \mathbb{N}$.

Anhand der Win-Tie-Loss Tabellen kann man mittels eines Vorzeichen-Tests (Sign-Test) ermitteln mit welcher Irrtumswahrscheinlichkeit ein Algorithmus A schlechter bzw. mit welcher Sicherheit ein Algorithmus A besser ist als ein anderer Algorithmus B. Der Vorzeichentest beantwortet die Frage, wie oft ein Algorithmus A besser als ein Algorithmus B sein muss, damit mit einer bestimmten Sicherheit gesagt werden kann, dass A wirklich besser ist. Als Annahme gilt, dass beide Algorithmen, A und B, gleich sind. D.h. die Wahrscheinlichkeit, dass ein Algorithmus auf einem Datensatz gewinnt ist 50%. Die Irrtumswahrscheinlichkeit, dass ein Algorithmus mindestens (N-k)-mal bzw. höchstens k-mal gewinnt, lässt sich wie in Gleichung 2.9 berechnen. N ist die Anzahl der Experimente, wobei ein

Unentschieden nicht gezählt wird, und k gibt an wie oft ein Algorithmus A gegen einen Algorithmus B gewonnen hat.

$$Signtest: P(i \leq k \vee i \geq N - k) = \frac{1}{2^{N-1}} * \sum_{i=1}^{k} \binom{N}{i} \quad (2.9)$$

Als Beispiel für den Vorzeichen-Test soll die Tabelle 2 dienen. Hier erkennt man, dass der Algorithmus Ripper im Vergleich zum Covering-Algorithmus auf allen Datensätzen, außer *monk1*, besser ist. Der Ripper-Algorithmus erzielt somit 8 Wins, 1 Loss und 0 Ties. Setzt man die Werte nun in Gleichung 2.9 ein so beträgt die Irrtumswahrscheinlichkeit p=0,0352. Daraus kann man nun schließen, dass der Algorithmus RIPPER mit 95%-iger Sicherheit besser ist als der Covering-Algorithmus.

Datensatz[2]	*Genauigkeit Ripper*[3]	*Genauigkeit Covering*[4]
balance-scale	80,80 %	73,12 %
breast-cancer	70,89 %	67,48 %
colig	84,24 %	71,20 %
horse-colic	86,96 %	71,20 %
monk1	83,06 %	95,16 %
segment	95,71 %	93,90 %
vowel	70,81 %	49,49 %
zoo	86,14 %	85,15 %

Tabelle 2: Beispiel - Genauigkeiten von Ripper und Covering

[2] Eine genaue Erklärung der Datensätze ist in Kapitel 6 zu finden.
[3] Ripper wird genauer in Kapitel 4.4.2 beschrieben.
[4] Der Covering-Algorithmus wird in Kapitel 5 und 6 genauer betrachtet

Kapitel 3 Suchheuristiken

In diesem Kapitel werden Suchheuristiken genauer betrachtet. Dazu werden bestimmte Merkmale einer Trainingsmenge definiert, sowie Regel- bzw. Regelmengenmerkmale auf der Trainingsmenge. Weiter werden Methoden wie der PN- bzw. ROC-Raum vorgestellt, anhand derer man Heuristiken visualisieren bzw. charakterisieren kann. Anschließend werden die Suchheuristiken und deren Merkmale vorgestellt die in dieser Arbeit verwendet werden.

3.1 Eigenschaften von Regeln, Regelmengen und Trainingsmengen

Eine Heuristik ordnet einer Regel r einen bestimmten Wert $h(r)$ zu, der auf den Merkmalen der Regel bzw. den Merkmalen der Trainingsmenge, auf der die Regel gelernt wurde, basiert. Die Merkmale die den Trainingsdaten und einer Regel zugeordnet werden sind:

p	Die Anzahl der abgedeckten positiven Beispiele
n	Die Anzahl der abgedeckten negativen Beispiele

Tabelle 3: Merkmale einer Regel

P	Die Anzahl aller positiven Beispiele in der Trainingsmenge
N	Die Anzahl aller negativen Beispiele in der Trainingsmenge
T = (P + N)	Die Anzahl aller Beispiele in der Trainingsmenge

Tabelle 4: Merkmale einer Trainingsmenge

Aus diesen vier grundlegenden Eigenschaften lassen sich weitere Merkmale (siehe Tabelle 5) für Regeln ableiten.

$tpr = {}^{p}/_{P}$	Die *TruePositiveRate*, der Prozentsatz der abgedeckten positiven Beispiele von allen positiven Beispielen
$fpr = {}^{n}/_{N}$	Die *FalsePositiveRate*, der Prozentsatz der abgedeckten negativen Beispiele von allen negativen Beispielen
$tp = p$	*TruePositive*. Ein positives Beispiel, dass als positives Beispiel von einer Regel vorhergesagt wurde.
$fp = n$	*FalsePositive*. Ein negatives Beispiel, dass als positives Beispiel von einer Regel vorhergesagt wurde.
$fn = (P - p)$	*FalseNegative*. Ein positives Beispiel, dass als negatives Beispiel von einer Regel vorhergesagt wurde.
$tn = (N - n)$	*TrueNegative*. Ein negatives Beispiel, dass als negatives Beispiel von einer Regel vorhergesagt wurde.

Tabelle 5: Abgeleitete Merkmale einer Regel bzw. Regelmenge

	Als positiv von der Regel vorhergesagt	Als negativ von der Regel vorhergesagt	
positives Beispiel	tp = p	fn = (P-p)	P
negatives Beispiel	fp = n	tn = (N-n)	N
	(p+n)	(P+N)-p-n	T = (P+N)

Tabelle 6: Konfusionsmatrix

Die gleichen Merkmale für Regeln lassen sich auch einer Regelmenge zuordnen. Im weiteren Verlauf gilt die Notation, die in den Tabellen 3 bis 5 benutzt wurde. Anhand der Merkmale lässt sich eine sogenannte Konfusions- oder Kontingenzmatrix erstellen.

Mit diesen grundlegenden Merkmalen lässt sich nun definieren was eine Heuristik ist:

Definition 3.1.1 (Heuristik): Eine Heuristik schätzt die Qualität bzw. Güte einer Regel, indem sie der Regel *r* einen Wert *c* zuordnet. Damit ist die Heuristik eine zweidimensionale Funktion, die eine Regel auf einen reellen Wert abbildet. Berechnet wird der Heuristikwert aus den Merkmalen einer Regel und/oder aus den Merkmalen der Trainingsmenge. Die Merkmale der Trainingsmenge können als konstant angesehen werden.

$$H(r) = H(p,n): \mathbb{N} \times \mathbb{N} \to \mathbb{R} \quad (3.1)$$

3.2 PN – und ROC – Raum

Obwohl der ROC-Raum (Definition 3.2.1) seine Ursprünge in der Singaltheorie hat, besitzt er nützliche Eigenschaften um Hypothesen[5], einzelne Regeln oder den Lernprozess einer Hypothese bzw. Regel zu analysieren, zu evaluieren und zu visualisieren. Weitere Anwendungsgebiete des ROC-Raumes sind z.B. die Klassifikation von Beispielen selbst oder das Finden von Entscheidungsbaum Splitkriterien (decision tree splitting criteria) [Flac03].

Definition 3.2.1 (ROC-Raum): Der ROC-Raum ist durch die *tpr (truepositivesrate)* und *fpr (falsepositivesrate)* einer Hypothese bzw. einer Regel definiert. Ein Punkt (x, y) im ROC-Raum entspricht einer Hypothese bzw. Regel mit einer tpr von y und fpr von x.

Im Gegensatz zum ROC-Raum wird der PN-Raum (Definition 3.2.2) nicht durch die tpr und die fpr definiert, sondern von den absoluten Zahlen der positiven und negativen Beispielen einer Trainingsmenge.

Definition 3.2.2 (PN-Raum): Der PN-Raum wird durch die Anzahl der positiven (P) und negativen (N) Beispiele definiert. Ein Punkt (x, y) im PN-Raum entspricht einer Hypothese bzw. Regel die x negative Beispiele (n) und y positive Beispiele (p) abdeckt.

Beide Räume lassen sich ineinander überführen, indem man den PN-Raum normalisiert, sodass die Achsen nur Werte zwischen 0 und 1 haben [FüFl03]. Der ROC-Raum lässt sich in den PN-Raum überführen, indem die tpr-Achse mit *P* und die fpr-Achse mit *N* multipliziert werden. Die unterschiedlichen Merkmale beider Räume sind in folgender Tabelle zusammengefasst:

Merkmal	ROC-Raum	PN-Raum
x-Achse	$fpr = {}^{n}/_{N}$	N
y-Achse	$tpr = {}^{p}/_{P}$	P
Leere Theorie	(0, 0)	(0,0)
Korrekte Theorie	(0, 1)	(0, P)
Universelle Theorie	(1, 1)	(N, P)
Auflösung	$\left(\frac{1}{N}, \frac{1}{P}\right)$	(1, 1)
Steigung der Diagonalen	1	$\frac{P}{N}$
Steigung der Geraden p=n	$\frac{N}{P}$	1

Tabelle 7: Merkmale ROC- vs. PN-Raum. (übernommen aus [FüFl05])

[5] In diesem Sinne spricht man von Klassifikatoren

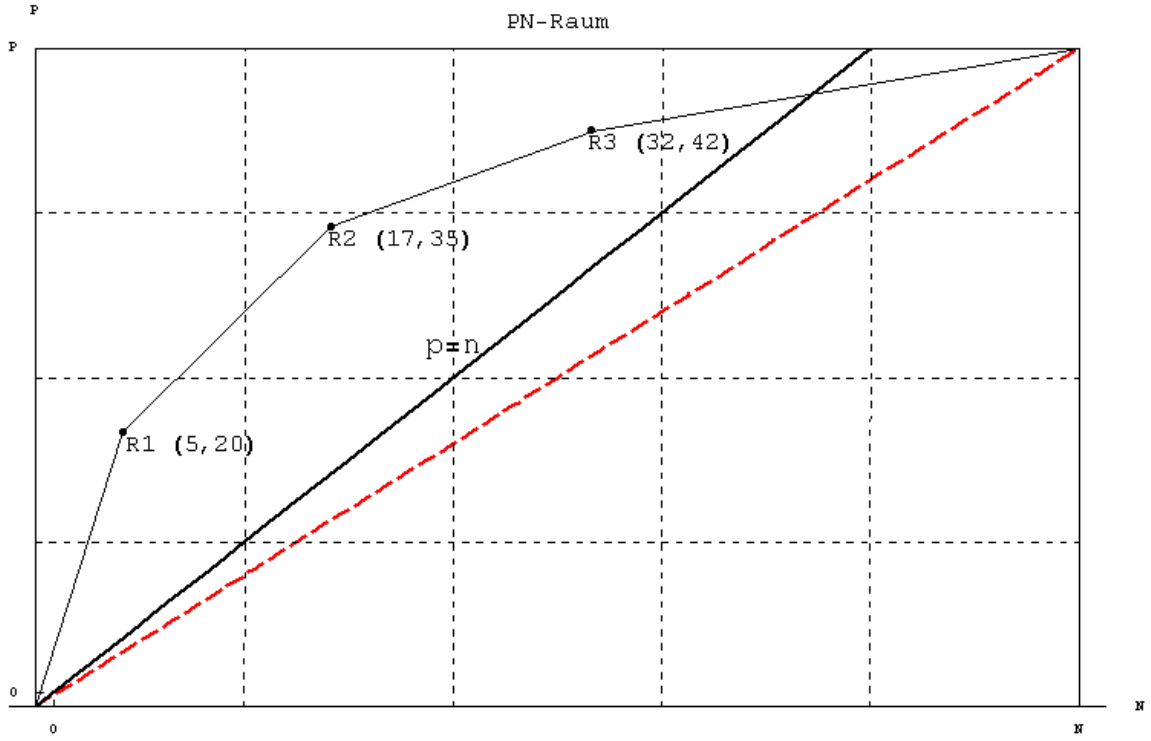

Abbildung 5: PN-Raum Beispiel

Im nachfolgenden beschränken sich die Betrachtungen auf den PN-Raum. Ausführliche Informationen zu ROC-Räumen finden sich in [FüFl05] und [Flac04].

Als Beispiel für den PN-Raum soll Abbildung 5 dienen. Die Anzahl der positiven Beispiele beträgt hier 48, die Anzahl der negativen Beispiele 60. Damit befindet sich die optimale Theorie am Punkt (0, 48), die universelle Theorie am Punkt (60, 48). Die rote gestrichelte Linie entspricht der Standardverteilung der positiven und negativen Beispiele. Eine Regel oder eine Hypothese, die auf dieser Geraden liegt, würde Beispiele zufällig anhand der Klassenverteilung einer Klasse zuordnen. Die zweite Gerade, die in Abbildung 5 zu sehen ist, ist die sogenannte Iso-Genauigkeit. Für die Iso-Genauigkeit gilt, dass $p=n$ ist. Die Steigung ist je nach Anzahl der positiven und negativen Beispiele steiler oder geringer als die der Standardverteilung.

In Abbildung 6 sind drei Punkte eingezeichnet, die jeweils einer Regel einer Hypothese entsprechen. Die Punkte geben an wie viele positive und negative Beispiele die Hypothese, nach dem Hinzufügen der Regel $R_i, i = 1,2,3$, abdeckt. Anhand des PN-Raumes lässt sich auch der Lernvorgang einer Hypothese visualisieren. Nehmen wir an, dass die Regeln in der Reihenfolge R1, R2, R3 zur Hypothese hinzugefügt wurden, dann ist der Startpunkt im PN-Raum beim Lernen (0, 0), was der leeren Theorie entspricht. Nachdem die Regel R1 zur Hypothese hinzugefügt wurde, deckt diese nun 30 negative und 90 positive Beispiele ab. Durch Hinzufügen der Regel R2 erhöht sich die Abdeckung der Hypothese auf den Trainingsdaten, der neu hinzugefügte Punkt tendiert in Richtung der universellen Theorie. Das gleiche passiert nachdem die dritte Regel hinzugefügt wurde. Jedes weitere Hinzufügen einer Regel nach der Dritten, würde die Trainingsdaten vollständig abdecken. Der Lernalgorithmus wäre bei der universellen Theorie, beim Punkt (N, P), angelangt. Man kann sich das Hinzufügen von Regeln zu einer Hypothese als Pfad durch den PN-Raum vorstellen. Genauso kann man sich das Lernen einer Regel als Pfad im PN-Raum vorstellen und visualisieren. Der Pfad beim Lernen einer Regel beginnt im Punkt (N, P) und bewegt sich in Richtung des Punktes (0, P).

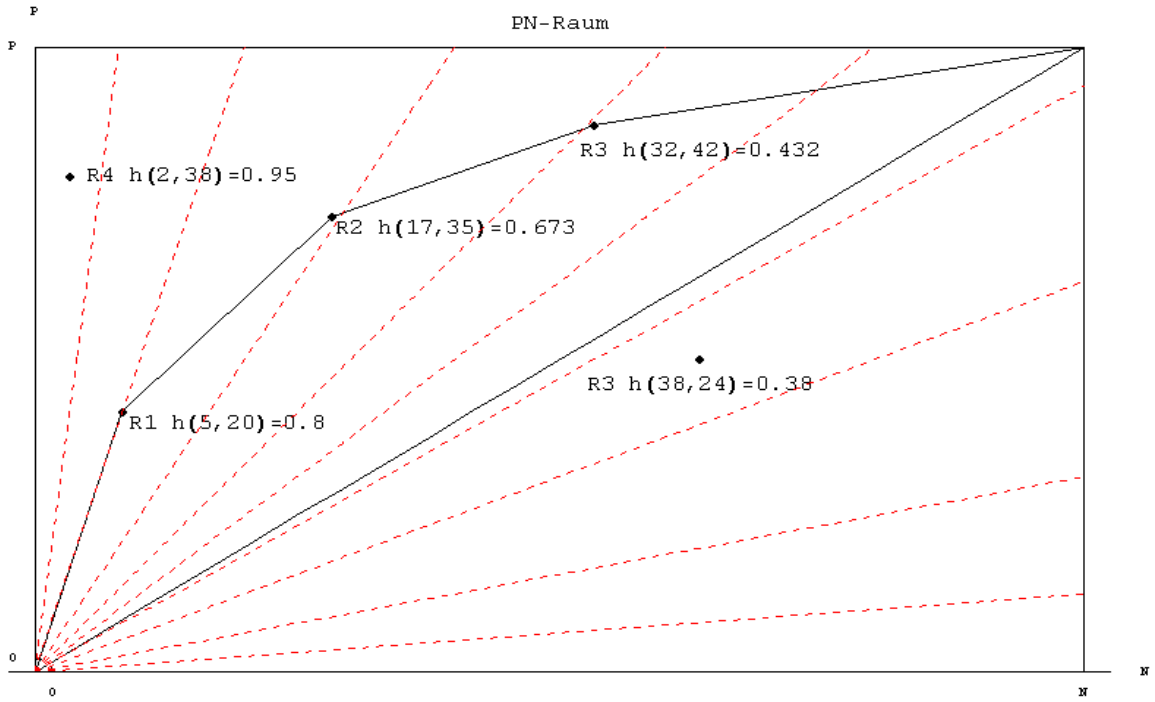

Abbildung 6: Beispiel Isometrik im PN-Raum

Nachdem eine Regel $R_i = (n_i, p_i)$ zu einer Hypothese hinzugefügt wurde, werden die Beispiele, die von dieser Regel abgedeckt werden, von der Trainingsmenge entfernt (Separate-Schritt). Für das Lernen einer neuen Regel $R_{i+1} = (n_{i+1}, p_{i+1})$ ist der PN-Raum deswegen kleiner als der vorangegangene PN-Raum PN_i. Der Punkt R_i ist dann der Ursprung eines neuen, kleineren PN-Raums PN_{i+1} und das Lernen der Regel R_{i+1} ist nun auf einen PN-Raum der Größe $N - n_i$ bzw. $P - p_i$ beschränkt. Visualisiert man diese PN-Räume in einem einzigen Diagramm, so erhält man mehrere verschachtelte PN-Räume [FüFl05] (siehe Abbildung 7).

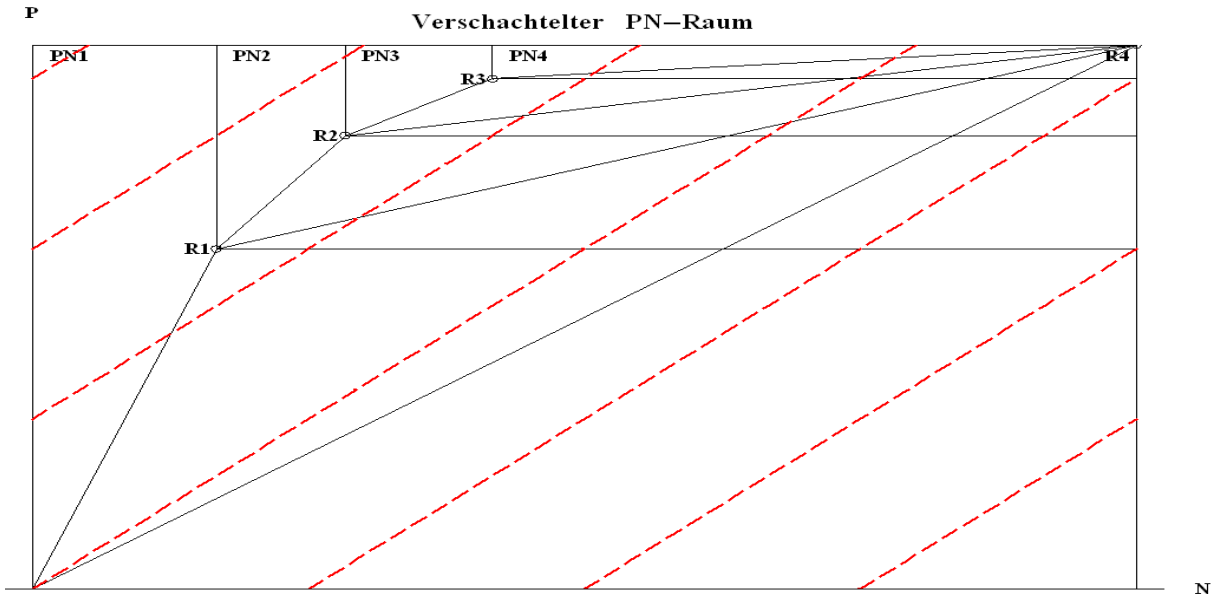

Abbildung 7: Verschachtelter PN-Raum (Accuracy Isometriken)

In Abbildung 7 sind verschiedene PN-Räume in einem PN-Raum eingezeichnet. Die Isometriken, die hier als rote gestrichelte Linien dargestellt sind, sind die Isometriken der Heuristik *Accuracy*.

Das Ziel beim Lernen ist den Punkt (0, P), die optimale Theorie, zu erreichen, was aber in den wenigsten Fällen möglich ist. Das Lernen, wie im Beispiel, tendiert meist in Richtung des Punktes (N, P). Damit das Lernen in Richtung der optimalen Theorie tendiert, werden Heuristiken verwendet. Diese lassen sich durch sogenannte Isometriken (Definition 3.2.3) im PN-Raum visualisieren. Anhand der Isometriken lässt sich leicht die Tendenz beim Lernen hin zur korrekten Theorie visualisieren.

Definition 3.2.3 (Isometrik): Eine Isometrik einer Heuristik ist eine Kurve im PN-Raum, die alle Punkte (n, p), die den gleichen Heuristikwert h besitzen, verbindet [FüFl03]. Es gilt:

$$H(r) = H(n,p) = h \quad (3.2)$$

Die Isometriken sind eigentlich dreidimensionale Funktionen und werden im zweidimensionalen durch ihre Iso-Linien dargestellt. Die Iso-Linien sind zum Beispiel wie in Abbildung 6 oder 8 die roten gestrichelten Linien.

In Abbildung 6 sind die Isometriken der Heuristik aus Gleichung 3.4 zu sehen, die als rote gestrichelte Linien dargestellt sind. Zusätzlich zu den in Abbildung 6 eingetragenen Punkten sind noch zwei weitere Regeln (R4, R5) eingetragen, die einen im Vergleich zu den Regeln R1, R2 und R3, höheren bzw. niedrigeren Heuristikwert besitzen, und für jede Regel ist der Heuristikwert angegeben. Im Normalfall würde ein Lernalgorithmus beim Lernen die Regel R4 allen anderen Regeln vorziehen, da diese Regel den Heuristikwert der Heuristik aus Gleichung 3.3 maximieren würde.

Man erkennt hier leicht, dass ein Separate-and-Conquer Algorithmen versuchen wird Regeln auszuwählen, die einen möglichst hohen Heuristikwert haben, bzw. Regeln, die nahe am Punkt (0, P) sind. Anders ausgedrückt versucht der Algorithmus, die Anzahl der abgedeckten positiven Beispiele zu maximieren und die Anzahl der negativen Beispiele zu minimieren.

3.3 Verwendete Heuristiken

In diesem Abschnitt werden die Heuristiken, die in der Diplomarbeit verwendet werden, und die Eigenschaften der Heuristik beim Lernen vorgestellt. Die Heuristiken werden zum einen anhand ihrer Formel, zum anderen anhand einer Graphik im dreidimensionalen und im PN-Raum dargestellt. Die hier verwendeten Heuristiken lassen sich grob in zwei Klassen einteilen. Die linearen Heuristiken sind Heuristiken, deren Isometriken Geraden sind. Die Isometriken der linearen Heuristiken lassen sich einem von zwei grundlegenden Modellen zuordnen. Die zweite Klasse von Heuristiken, die hier genauer vorgestellt werden, sind die sogenannten nicht-linearen Heuristiken, deren Isometriken Kurven im PN-Raum bilden.

3.3.1 Lineare Heuristiken

Die linearen Heuristiken werden anhand der Linearen Kosten Metrik (Gleichung 3.3) oder $h_{costs} - Modell$, und dem Precision Modell oder $h_{pr} - Modell$, in zwei Familien unterteilt. Die Heuristik Precision (Gleichung 3.4) ist namensgebend für diese Familie von Heuristiken. Der Beweis der Äquivalenz der Heuristiken dieser Modelle ist in [FüFl05] zu finden.

$$h_{costs} = a * p - b * n \sim c * p - (1 - c) * n \sim p - d * n \quad (3.3)$$

$$h_{pr} = \frac{p}{p + n} \quad (3.4)$$

Das Hauptmerkmal der Heuristiken des $h_{costs} - Modells$ sind Isometriken, die parallel zueinander sind. Die Isometriken des $h_{pr} - Modells$ dagegen rotieren um einen Punkt im PN-Raum. Die Isometriken aus Abbildung 6 gehören damit also dem $h_{pr} - Modell$ an. Ein weiterer Unterschied beider Modelle ist, dass die Isometriken für das $h_{costs} - Modells$ in verschachtelten PN-Räume immer die gleichen sind, wohingegen die Isometriken des $h_{pr} - Modells$ sich für jeden verschachtelten PN-Raum unterscheiden. Dies liegt daran, dass die Isometriken des $h_{pr} - Modells$ symmetrisch um einen Ursprung rotieren. Ein weiteres besonderes Merkmal der h_{costs} Isometriken ist, dass ein lokales Optimum in einem verschachtelten PN-Raum auch gleichzeitig ein globales Optimum im eigentlichen PN-Raum ist, Wohingegen ein lokales Optimum des $h_{pr} - Modell$ nicht immer ein globales Optimum ist [FüFl05].

Heuristiken des $h_{costs} - Modells$: Die grundlegendsten Heuristiken, die Basisheuristiken, dieses Modells sind MaximizePositives (Gleichung 3.5) und MinimizeNegatives (Gleichung 3.6).

$$MaximizePositives: h_p(n,p) = p \quad (3.5)$$

$$MinimizeNegatives: h_n(n,p) = -n \quad (3.6)$$

Die Isometriken der beiden Heuristiken aus Gleichung 3.5 und 3.6 sind in Abbildung 8 zu sehen. Die Isometriken auf der linken Seite sind die Isometriken der Heuristik MaximizePositives, die auf der linken Seite für MinimzeNegatives. Die Heuristik MaximizePositives bewertet Regeln, die sehr viele positive Beispiele abdecken, unabhängig davon wie viele negative Beispiele abgedeckt werden sehr gut. Die Heuristik MinimizeNegatives bewertet Regeln, die wenige negative Beispiele abdecken, unabhängig von der Anzahl der abgedeckten positiven Beispiele am besten. Für beide Heuristiken lässt sich schnell eine optimale Theorie bzw. Regel finden. h_p ist maximal für die universelle Theorie und die korrekte Theorie. h_n ist maximal für die leere Theorie, die keine negativen Beispiele abdeckt, aber auch keine positiven Beispiele. Eigentlich ist h_p maximal für Theorien, die alle positiven Beispiele abdecken und h_n ist maximal für Theorien, die keine negativen Beispiele abdecken. Beide Heuristiken lassen sich durch eine geeignete Wahl der Parameter aus Gleichung 3.5 herleiten. Dazu setzt man *a=0 und b=1* für MinimizeNegatives oder *a=1 und b=0* für MaximizePositives.

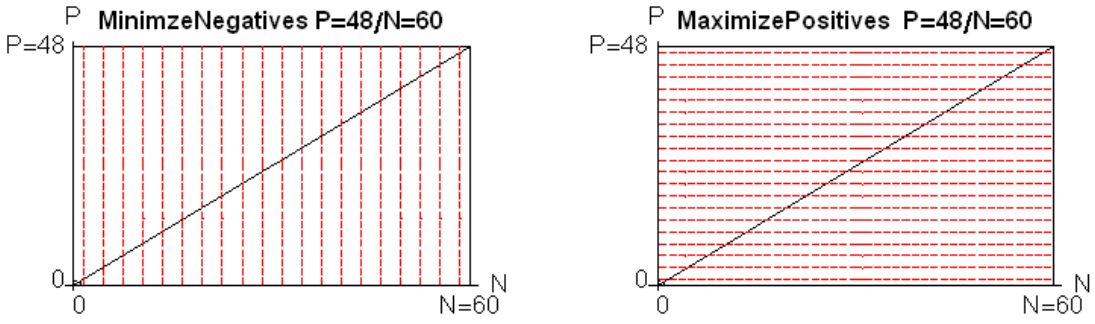

Abbildung 8: MaximizePositives und MinimizeNegatives Isometriken

Da das Ziel beim Lernen der Punkt (0, P) ist, ist es sinnvoll beide Heuristiken, h_p und h_n, gleichzeitig zu maximieren. Addiert man beide Heuristiken, erhält man daraus die Heuristik h_{acc} (Gleichung 3.7), die äquivalent zu Accuracy ist [FüFl05]. Accuracy schätzt das Verhältnis der abgedeckten positiven (p) und nicht abgedeckten negativen Beispiele (N-n) einer Regel bzw. Hypothese zu allen Beispielen (P+N). Da P und N normalerweise konstant sind, kann man die Heuristik wie in (Gleichung 3.7) durch

$p - n$ abschätzen. Die Klassenverteilung der Beispiele in der Trainingsmenge ist dabei unerheblich. Wie in Abbildung 9 zu sehen ist, schließen die Isometriken einen 45° Winkel mit der x-Achse ein. Die Isometriken sind also Geraden mit der Steigung 1. Durch Setzen der Parameter $a = b = d = 1$ oder $c = \frac{1}{2}$ erhält man aus Gleichung 3.3 die Heuristik *Accuracy*.

$$Accuracy: h_{acc}(n,p) = p - n \cong \frac{p + (N - n)}{P + N} \quad (3.7)$$

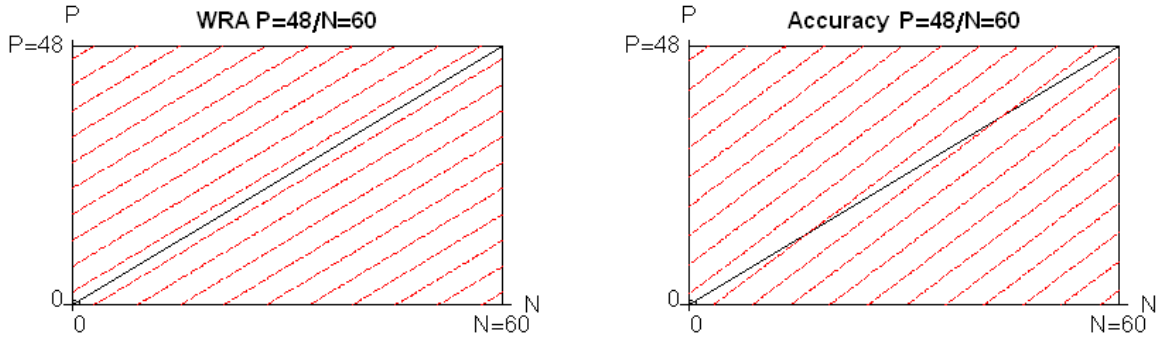

Abbildung 9: Isometriken von Accuracy und Weighted Relative Accuracy

Ein Nachteil der Heuristik ist, dass es gleich gut ist ein positives Beispiel abzudecken und ein negatives Beispiel nicht abzudecken. Dieser Nachteil kommt allerdings erst zum tragen, wenn die a priori Verteilung der Beispiele nicht repräsentativ für eine Domäne ist, oder die Kosten einer falschen Vorhersage nicht bekannt sind. Um diesem Nachteil entgegenzuwirken ist es sinnvoll die Heuristik anhand der Trainingsmenge zu normalisieren. Das Normalisieren der Heuristik *Accuracy* führt zu *Weighted Relative Accuracy (WRA)* [FüFl05]. WRA wird durch Gleichung 3.8 definiert.

$$WRA: h_{wra}(n,p) = \frac{p}{P} - \frac{n}{N} = TPR - FPR \sim \frac{p+n}{P+N}\left(\frac{p}{p+n} - \frac{P}{P+N}\right) \quad (3.8)$$

Ebenfalls in Abbildung 10 lässt sich leicht der Unterschied zwischen *Accuracy* und *WRA* feststellen. Die Isometriken von *WRA* sind alle parallel zur Standardverteilung, was gleichbedeutend damit ist das alle Isometriken die Steigung $\frac{P}{N}$ haben. Die Heuristik bewertet ein Steigen der *tpr* einer Hypothese genauso gut wie ein Absinken der *fpr* einer Hypothese. Ein weiteres Merkmal der Weighted Relative Accuracy ist, dass sie für den Fall P = N gleich zur Accuracy ist. Die WRA lässt sich durch setzen der Parameter $a = \frac{1}{P}, b = \frac{1}{N}$ oder $c = \frac{N}{P+N}$ oder $d = \frac{P}{N}$ aus Gleichung 3.3 gewinnen.

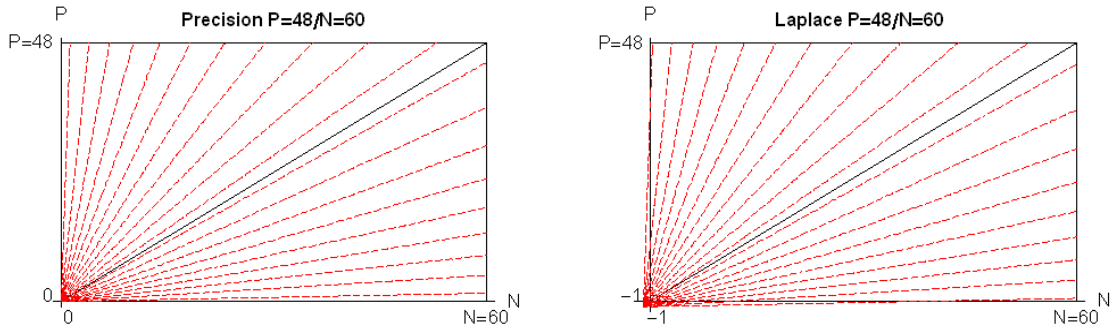

Abbildung 10: Isometriken Precision und Laplace

Heuristiken des $h_{pr} - Modells$: Die grundlegende Eigenschaft dieser Heuristiken ist, dass die Isometriken symmetrisch um einen Punkt des PN-Raumes rotieren. Die Isometriken der Heuristik h_{pr} (Gleichung 3.9) sind in Abbildung 10 links zu sehen. Die Heuristik misst das Verhältnis von allen abgedeckten positiven Beispielen zu allen abgedeckten Beispielen einer Regel bzw. Hypothese.

$$h_{pr}(n,p) = \frac{p}{p+n} \quad (3.9)$$

Die Heuristik Precision bewertet Regeln, die nur positive Beispiele abdecken, maximal und Regeln, die nur negative Beispiele abdecken, minimal. Dies entspricht Regeln, die sich auf der y-Achse (Abdeckung nur positiver Beispiele) bzw. auf der x-Achse (Abdeckung nur negativer Beispiele) befinden. Alle anderen Regeln werden somit besser als das Minimum bzw. schlechter als das Maximum bewertet. Dies ist auch gleichzeitig ein Nachteil der Heuristik. Betrachtet man z.B. die Regeln R1, die ein positives und kein negatives Beispiel abdeckt, und die Regel R2, die 1000 positive und kein negatives Beispiel abdeckt, erkennt man, dass beide Regeln gleich bewertet ($h_{pr}(n, p) = 1$) werden, obwohl die Regel R2 eindeutig besser ist als die Regel R1.

Um diesen Nachteil auszugleichen, kann man die Annahme treffen, dass eine Regel von vornherein schon eins, zwei oder m Beispiele abdeckt. Dadurch verändert sich die Bewertung für jede einzelne Regel bzw. Hypothese dahingehend, dass Regeln, die wenig positive Beispiele abdecken schlechter bewertet werden als Regeln, die viele positive Beispiele abdecken. Dies trifft unter der Annahme zu, dass n für die Regeln konstant ist. Diese Heuristiken sind Modifikationen des $h_{pr} - Modells$. Durch diese Modifikationen ist der Punkt um den die Isometriken rotieren nicht mehr der Ursprung eines PN-Raumes, sondern ein Punkt der im negativen Bereich der x- bzw. y-Achse zu finden ist. Eine sehr bekannte Modifikation des Precision-Modells ist die Laplace-Heuristik (Gleichung 3.10). Diese Heuristik geht davon aus, dass eine Regel schon mindestens ein negatives und ein positives Beispiel abdeckt. Geht man nun von den Regeln R1 und R2 aus, wie oben im Beispiel, so sieht man, dass die Regel R2 ($h_{lap}(n,p) \cong 1$) besser bewertet wird als die Regel R1 ($h_{lap}(n,p) = 2/3$). Diese Annahme hat zur Folge, dass der Punkt, um den die Precision-Isometriken rotieren, nun bei (-1, -1) liegt, wie in Abbildung 10 zu erkennen ist.

$$Laplace: h_{lap}(n,p) = \frac{p+1}{(p+1)+(n+1)} = \frac{p+1}{p+n+2} \quad (3.10)$$

Eine andere Modifikation des Precision-Modells ist das sogenannte *m-Estimate* (Gleichung 3.11). Bei dieser Heuristik ist der Ursprung der Isometriken von einem Parameter m abhängig. Je nachdem wie der Parameter gewählt wird, ändern sich die Isometriken der Heuristik im PN-Raum. Generell kann man sagen, dass der Punkt $(-m * \left(1 - \frac{P}{P+N}\right),\ -m * \frac{P}{P+N})$ der Ursprung der Precision-Isometriken des m-Estimates. Geht man von einer Gleichverteilung der Beispiele in der Trainingsmenge aus (P = N) und wählt den Parameter m=2, so kann man aus dem m-Estimate die Laplace-Heuristik herleiten. Wählt man m=0 so erhält man Heuristik Precision. Abbildung 11 zeigt die Isometriken für das m-Estimate für verschiedene Parameter, die in dieser Arbeit verwendet wurden. Der Parameter m=22.466 ist ein optimaler Parameter, der in [JaFü06] gefunden wurde.

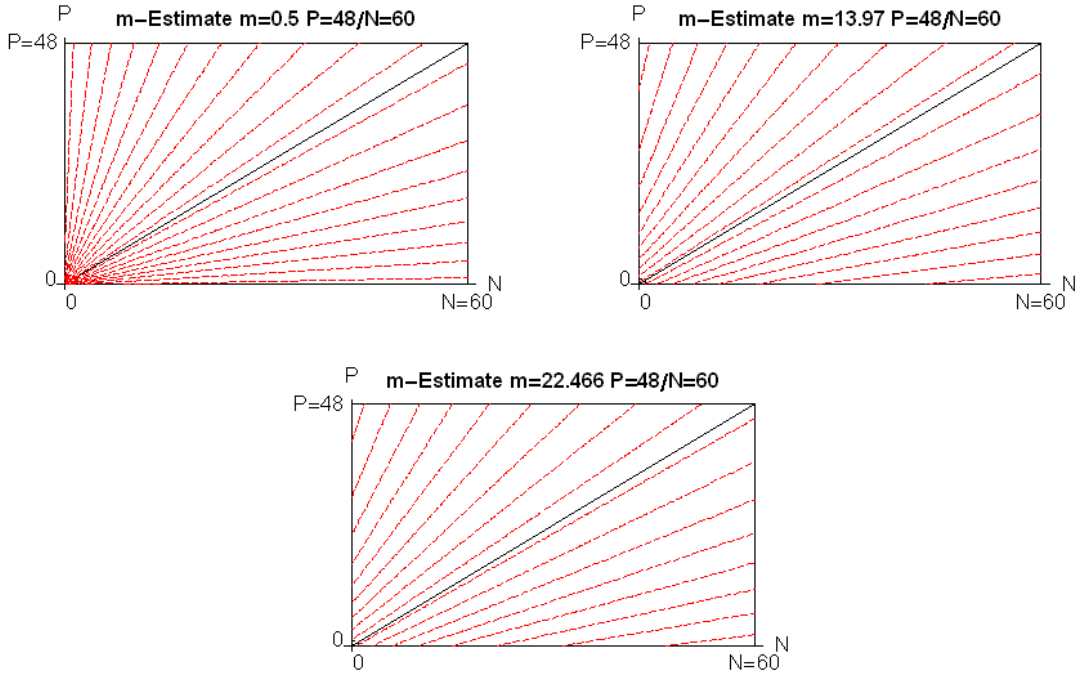

Abbildung 11: Isometriken für das m-Estimate mit verschiedenen Parametern

$$m - Estimate: h_m(n,p) = \frac{p + m * \frac{P}{P+N}}{p + n + m} \quad (3.11)$$

Die Laplace-Heuristik und das m-Estimate sind beides Spezialfälle des Generalized m-Estimates (Gleichung 3.12). Ein besonderes Merkmal des *Generalized m-Estimates* ist, dass sich die Isometriken je größer m wird immer mehr denen der Heuristik *WRA* ähneln. Für $m = \infty$ sind die Isometriken des *Generalized m-Estimate* und die Isometriken des *Generalized Cost-Measures* äquivalent und für m=0 sind die Isometriken äquivalent zu denen von *Precision* [FüFl05]. Denn es gilt, für $a = b = 1\ bzw. m = 2\ und\ c = 1/2$, dass $h_{gm} = h_{lap}$. Ein weiterer Spezialfall des Generalized m-Estimates ist das M-Estimate, der Parameter c wird dazu gleich $\frac{P}{P+N}$ gesetzt.

$$Generalized\ m - Estimate: h_{gm}(n,p) = \frac{p + m * c}{p + n + m} = \frac{p + a}{(p + a) + (n + b)} \quad (3.12)$$

Weitere Heuristiken, die dem Precision-Modell angehören sind in [FüFl05] zu finden. Das Generalized m-Estimate bzw. das m-Estimate als Spezialfall sind beide parametrisierbare Heuristiken (Definition 3.3.1).

Definition 3.3.1 (parametrisierbare Heuristik): Eine parametrisierbare Heuristik *H* wägt zwischen zwei Heuristiken, $h_1\ und\ h_2$, ab. Diese Abwägung wird durch einen frei wählbaren Parameter *p* bewerkstelligt, sodass

- Für $p \to 0$: $H\ tendiert\ zu\ h_1$ oder
- Für $p \to \infty$: $H\ tendiert\ zu\ h_2$

gilt.

Das Generalized m-Estimate wägt zwischen Heuristiken des Precision-Modells und Heuristiken des Linear-Cost-Modells ab [FüFl05]. Eine weitere parametrisierbare Heuristik ist das Klösgen-Maß (Gleichung 3.15) [Klös92], das die Heuristiken PrecisionGain (Gleichung 3.13) und Coverage (Gleichung 3.14) abwägt. Coverage misst die Anzahl der abgedeckten Beispiele im Verhältnis zur Anzahl aller Beispiele, während PrecisionGain sich ähnlich wie Precision verhält. Der Unterschied zwischen Precision und PrecisionGain ist der zusätzliche Faktor $\frac{P}{P+N}$, der noch die a priori Verteilung der Beispiele in der Trainingsmenge mit betrachtet. Somit wird eine Regel, die keine negativen Beispiel abdeckt im besten Fall mit $1 - \frac{P}{P+N}$ und eine Regel die keine positiven Beispiele abdeckt im schlechtesten Falls mit $-\frac{P}{P+N}$ bewertet. Dies soll eine zu optimistische Bewertung einer Regel bzw. Hypothese verhindern.

$$PrecisionGain: h_{pg}(n,p) = \frac{p}{p+n} - \frac{P}{P+N} \quad (3.13)$$

$$Coverage: h_{cov}(n,p) = \frac{p+n}{P+N} \quad (3.14)$$

Um die Tendenz des Klösgen-Maßes hin zu einer dieser Heuristiken festzulegen, werden beide Heuristiken mit dem Parameter ω wie folgt verknüpft:

$$Klösgen - Maß: h_{kloes}(n,p) = {h_{cov}}^{\omega} * h_{pg} = \left(\frac{p+n}{P+N}\right)^{\omega} * \left(\frac{p}{p+n} - \frac{P}{P+N}\right) \quad (3.15)$$

Auffällig ist, dass das Klösgen-Maß eigentlich drei Heuristiken abwägt. Dazu können für den Parameter ω zwei Intervalle bestimmt werden, in denen diese Abwägung zu erkennen ist. Im Intervall $\omega \in [0,1]$ verhält sich die Heuristik wie Precision bzw. WRA. Für den Fall $\omega = 0$ ist das Verhalten äquivalent zu dem von Precision. Für $\omega = 1$ verhält sich das Klösgen-Maß äquivalent zur Weighted Relative Accuracy. Das zweite Intervall, das sich für ω identifizieren lässt, liegt zwischen $[1, \infty)$. Je größer ω wird, desto mehr ähneln die Isometriken des Klösgen-Maßes denen der Heuristik *Coverage*. In Abbildung 12 ist dieses Verhalten für verschiedene Werte von ω gezeigt. Der Wert 0.4323 für ω ist ein optimaler Wert, der in [JaFü06] bestimmt wurde. Die Werte 0,3 und 0,2 sind zwei weitere Parameter, die in dieser Arbeit verwendet werden.

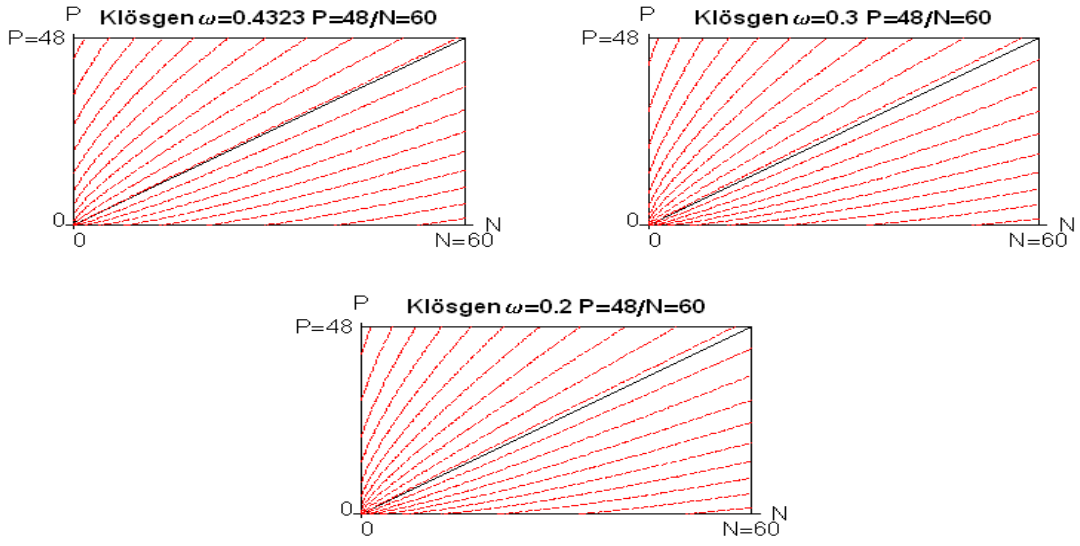

Abbildung 12: Isometriken des Klösgen-Maßes für verschiedene Werte von $\boldsymbol{\omega}$

3.3.2 Nicht-Lineare Heuristiken

Die Heuristiken, die bis jetzt betrachtet wurden, hatten alle lineare Isometriken gemeinsam. Das Klößgen-Maß zeigte lineare Isometriken nur für zwei verschiedene Werte. Die Isometriken der nicht-linearen Heuristiken sind Kurven im PN-Raum. Die zwei Vertreter dieser Familie von Heuristiken, die hier vorgestellt werden sind Correlation [FüFl05] und FOILGain [QuCJ95].

Die Heuristik Correlation (Gleichung 3.16) basiert auf dem Konzept der statistischen Korrelation. Der Korrelationskoeffizient zweier Zufallsvariablen X und Y berechnet den Grad der Abhängigkeit dieser Variablen. Der Wertebereich des Koeffizienten liegt zwischen -1 (perfekte negative Korrelation), 0 (keine Korrelation) und 1 (vollständige Korrelation). Dieses Konzept wurde in [Fürn94b] auf das Regellernen übertragen. Die Suchheuristik Correlation berechnet hier den Grad der Abhängigkeit zwischen der Klassenvorhersage einer Regel und den tatsächlichen Klassen der Beispiele anhand einer Konfusionsmatrix für ein 2-Klassenproblem.

$$Correlation\colon h_{corr}(n,p) = \frac{p(N-n)-n(P-p)}{\sqrt{PN(p+n)(P-p+N-n)}} = \frac{pN-nP}{\sqrt{PN(p+n)(P-p+N-n)}} \quad (3.16)$$

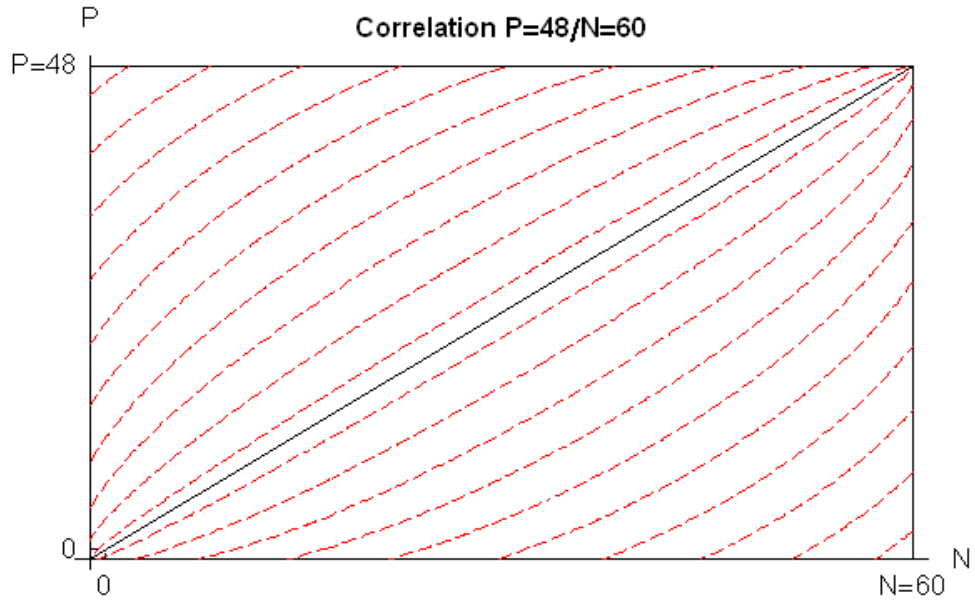

Abbildung 13: Isometriken für Correlation

Die Isometriken der Heuristik (Abbildung 13) sind symmetrisch um die Standardverteilung gebogen. Regeln, die sich auf der Geraden der Standardverteilung befinden werden von der Heuristik mit 0 bewertet, wohingegen Regeln die nahe am Punkt (0, P) zu finden sind, mit positiven Werten <1 und Regeln nahe dem Punkt (N, 0) mit negativen Werten > -1 bewertet werden. Die Heuristik tendiert dazu vollständige Regeln zu lernen, d.h. Regeln die keine negativen Beispiele abdecken. Je näher eine Regel dem Punkt (0, P) ist, desto besser wird die Heuristik Regeln bewerten, die negative Beispiele ausschließen, als Regeln die mehr positive Beispiele abdecken. Ein weiteres Merkmal der Heuristik ist, dass sie äquivalent zu einem normalisierten χ^2-Test über der Konfusionsmatrix ist [FüFl05].

Eine weitere Heuristik mit nicht-linearen Isometriken im PN-Raum wird in FOIL [QuCJ95] verwendet. Die Heuristik, hier als FOILGain (Gleichung 3.18) bezeichnet, die FOIL verwendet ist eine Modifikation der InformationGain Heuristik [FüFl05]. FOILGain bewertet nicht eine Regel, sondern die Differenz des InformationGains (Gleichung 3.17) zwischen einer Regel r und ihrer direkten Vorgängerregel r‘.

$$InformationGain: h(n,p) = \log_2 \frac{p}{p+n} \quad (3.17)$$

$$FOILGain: h_{foil}(n,p) = p\left(\log_2 \frac{p}{p+n} - \log_2 c\right) \quad (3.18)$$

Der Parameter c in Gleichung 3.18 ist der Precision-Wert der Vorgängerregel r‘ bevor diese verfeinert wurde. Die Isometriken von FOILGain zeigen im PN-Raum sowohl nicht-lineares als auch lineares Verhalten. Auch verändert sich die Landschaft der Isometriken, je nachdem wie die Vorgängerregel bewertet worden ist. Der Einfluss der Vorgängerregel und die Veränderung der Isometriken sind in Abbildung 14 visualisiert. Anhand der Isometriken lässt sich eine Kurve bzw. Basis-Isometrik identifizieren [FüFl05]. Diese Basis-Isometrik beginnt im Punkt (0, 0) und hat die Steigung $\frac{c}{c-1}$. Auf der Basis-Isometrik liegende Regeln werden mit dem Wert 0 bewertet. Befindet sich eine Regel unterhalb dieser Kurve, so wird sie negativ bewertet, was einem Verlust an Information entspricht. Regeln, die oberhalb dieser Kurve liegen werden positiv bewertet und sind somit Regeln, die zur weiteren Verfeinerung oder für die Hypothese in Frage kommen.

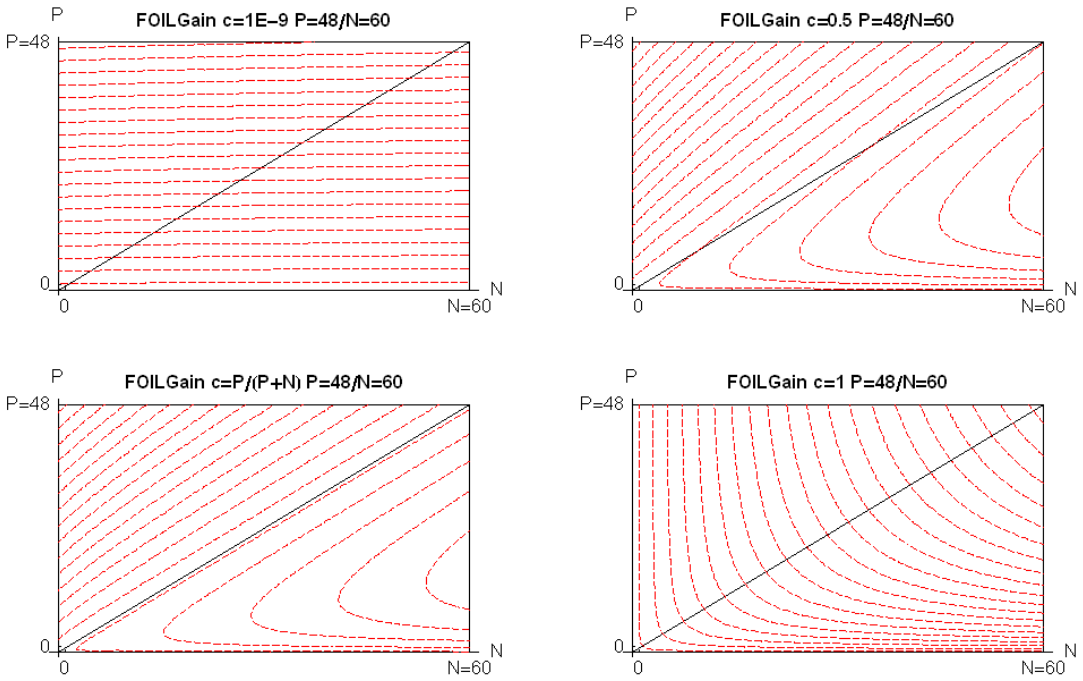

Abbildung 14: Isometriken FOILGain für verschieden Werte c

Kapitel 4 Pruning

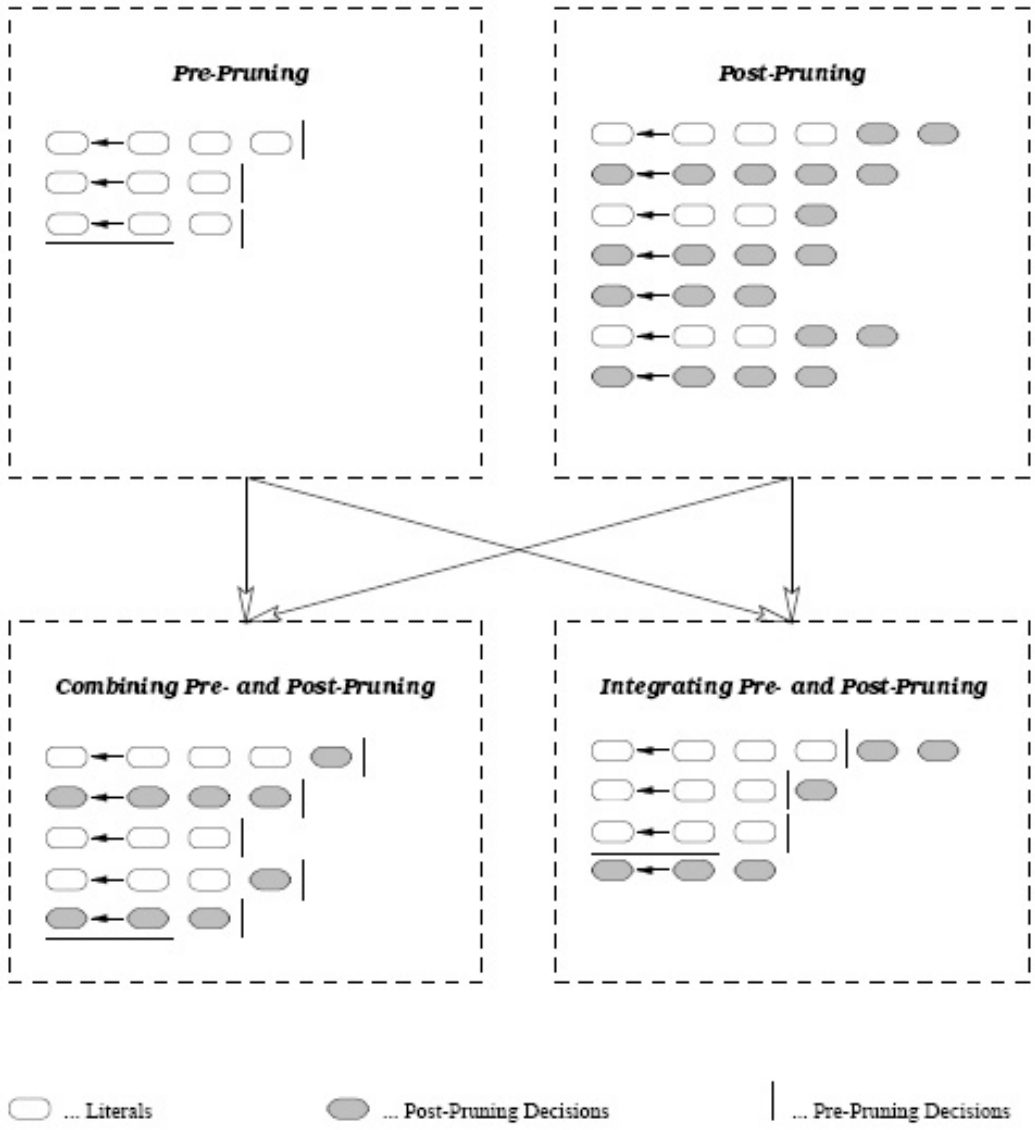

Abbildung 15: Pruningmethoden aus [Fürn97]

Eine Regel bzw. eine Hypothese, die auf verrauschten oder fehlerhaften Trainingsdaten gelernt wird, ist meist sehr komplex. Die Hypothese ist an die Trainingsdaten überangepasst und verallgemeinert nur schlecht für vorher ungesehene Beispiele. Sie enthält viele Regeln, und jede einzelne Regel deckt nur wenige positive Beispiele ab. Die Regeln enthalten zusätzlich noch viele Attributtests, die der Regel nur hinzugefügt wurden um negative Beispiele auszuschließen. Neben der Wahl einer geeigneten Heuristik für das Lernen, ist das Pruning eine weitere Methode ein Overfitting einer Hypothese an verrauschte Daten zu vermeiden und eine Hypothese zu erhalten die besser auf vorher nicht gesehenen Beispielen verallgemeinert. Das Pruning selbst greift an zwei Punkten des Lernprozesses, um eine Überangepasstheit zu vermeiden.

Um das Pruning an sich etwas zu veranschaulichen dient die Abbildung 15. Die einzelnen Felder der Abbildung enthalten Hypothesen, die mit unterschiedlichen Pruningmethoden gelernt wurden. Regeln, die in der Hypothese enthalten sind, sind in Abbildung 15 weiß dargestellt. Zusätzlich zu den weißen Regeln enthält die Abbildung noch graue Regeln, die durch Postpruning-Mechanismen aus der

endgültigen Regel entfernt wurden. Ebenso sind schwarze Striche enthalten, die eine Prepruning-Entscheidung darstellen. Horizontale Striche geben an, dass das Hinzufügen von Regeln zu einer Hypothese gestoppt wird (vgl. Kapitel 4.2.1), dagegen geben vertikale Striche an, dass eine Regel ab hier nicht weiter verfeinert wird (vgl. Kapitel 4.2.1).

Die erste grundlegende Möglichkeit des Prunens ist im linken oberen Feld der Abbildung 15 zu sehen. Das sogenannte Prepruning beschränkt sich ausschließlich auf Stopkriterien um zu entscheiden, ab wann eine Regel ausreichend verfeinert wurde bzw. ab wann eine Hypothese ausreichend Regeln enthält um die Zielhypothese zu beschreiben. Dieses Pruning beschränkt sich nur auf den Lernvorgang. Der zweite grundlegende Pruningmechanismus ist das sogenannte Postpruning (Abbildung 15, rechts oben). Das Prunen selbst greift erst nachdem eine Hypothese gelernt wurde. In Abbildung 15 erkennt man, dass die Hypothese deutlich größer ist als die gelernte Hypothese beim Prepruning. Erst nachdem Regeln und Attributtests durch das Postpruning aus der Hypothese entfernt worden sind, sind beide Hypothesen gleich groß. Die Regeln und Attributtests, die durch das Postpruning entfernt werden, sind in Abbildung 15 grau dargestellt [Fürn97].

Beide Mechanismen lassen sich auf zwei verschiedene Arten miteinander verbinden. Das kombinierte Pre- und Postpruning (vgl. Abbildung 15, links unten) erhält man, wenn man das Lernen mit Prepruningmechanismen unterstützt und die so gelernte Hypothese zusätzlich noch postprunt. Weiter kann man beide Pruningmechanismen auch integrieren (vgl. Abbildung 15, rechts unten). Das Integrierte Pre- und Postpruning lernt eine überangepasste Regel und prunt diese sofort. Dies ist der Postpruningschritt. Nach jeder Regel wird geprüft, ob die Hypothese bereits genügend Regeln enthält, d.h. verschlechtert sich die Bewertung der Hypothese, wenn die Regel mitaufgenommen wird im Vergleich zu der Hypothese ohne diese Regel. Ist dies der Fall so wird das Lernen abgebrochen [Fürn97].

Das Prepruning wird beim Lernen durch sogenannte Stopkriterien, die Entscheiden wann eine Regel bzw. Hypothese hinreichend genau ist, bewerkstelligt (siehe Kapitel 4.2.1). Im Gegensatz dazu werden beim Postpruning sogenannte Pruningoperatoren (Definition 4.1) verwendet um eine Hypothese bzw. eine Regel zu verallgemeinern.

Definition 4.1 (Pruningoperator): Ein Pruningoperator transformiert eine Hypothese/Regel in eine weitere, allgemeinere Hypothese/Regel oder lässt die Regel unverändert, falls keine allgemeinere Regel erzeugt werden kann.

Im nachfolgenden werden einige einfache Pruningoperatoren definiert und vorgestellt. Die Namensgebungen orientieren sich daran, wie der Operator eine Regel bzw. Hypothese transformiert.

- **delete-last-condition Operator [BrPa91]:** Der Operator löscht den letzten Attributtest einer Regel.
- **delete-rule Operator [BrPa91]:** Der Operator löscht eine ganze Regel.
- **delete-condition-sequence Operator [Cohe95]:** Dieser Operator löscht eine endliche Anzahl von Attributtests aus einer Regel
- **find-best-replacement Operator [WeIn91]:** Der Operator sucht auf der Pruningmenge für einen Attributtest einer Regel, einen anderen Attributtest, der die Bewertung einer Regel auf der Pruningmenge maximiert.
- **find-best-simplification Operator:** Sei *k* die Anzahl der Attributtests der Regel *R*, so erzeugt der Operator *k* neue Regeln, indem der Operator den ersten, zweiten, …, *k*-ten Attributtest aus der Regel löscht. Der Operator wählt dann die Regel, aus den *k* Regeln, die ein bestimmtes Qualitätsmaß maximiert. Die Regel, auf die der Operator

angewendet wird, wird auch mitbetrachtet. Besitzt diese Regel schon den maximalen Wert, so wird diese Regel ausgewählt.

- **identity Operator:** Dieser Operator bildet eine Regel bzw. Regelmenge auf sich selbst ab.

In diesem Kapitel werden die vier Möglichkeiten mit verrauschten Daten umzugehen und eine überangepasste Hypothese zu vermeiden genauer vorgestellt und exemplarisch ein Vertreter der Separate-and-Conquer Algorithmen vorgestellt, der diese Methode implementiert.

4.1 Postpruning

Das Postpruning ignoriert während des Lernens die Effekte, die verrauschte Daten bewirken können, und lernt eine Hypothese die konsistent und vollständig auf den Trainingsdaten ist. Die so gelernte Hypothese wird anhand eines Qualitätsmaßes bewertet und man versucht die Hypothese zu vereinfachen. Die Vereinfachungen der Hypothese werden gemacht, solange eine Vereinfachung die Bewertung des Qualitätsmaßes nicht verschlechtert. In Abbildung 16 ist ein typischer Postpruning Algorithmus zu sehen. Der erste Unterschied, der zu dem allgemeinen Separate-and-Conquer Algorithmen in Abbildung 4 auffällt, ist, dass die Hypothese nicht auf den gesamten Trainingsdaten gelernt wird. Stattdessen werden die Trainingsdaten in zwei kleinere Mengen aufgeteilt, das GrowingSet und PruningSet. Auf dem GrowingSet wird die Hypothese wie in Kapitel 2 beschrieben gelernt. Das PruningSet wird verwendet um die gelernte, überangepasste Hypothese wieder zu vereinfachen. Die Beispiele werden im Allgemeinen zufällig ohne Einhaltung der Klassenverteilung zu zwei Dritteln auf das GrowingSet und zu einem Drittel auf das PruningSet verteilt. Die Prozedur *BestSimplification* wählt die beste Vereinfachung der Hypothese anhand eines Qualitätsmaßes aus der Menge aller möglichen Vereinfachungen aus. Das am häufigsten verwendete Qualitätsmaß ist die Vorhersagegenauigkeit [Fürn97] auf der Pruningmenge. Die „loop“-Schleife entspricht der Prozedur *PostProcess* aus Abbildung 4.

```
procedure POSTPRUNING(Examples, Splitratio)

    SPLITEXAMPLES(Splitratio, Examples, GrowingSet, PruningSet)
    Theory = SEPARATEANDCONQUER(GrowingSet)
    loop
        NewTheory = BESTSIMPLIFICATION(Theory, PruningSet)
        if ACCURACY(NewTheory, PruningSet) < ACCURACY(Theory, PruningSet)
            exit loop
        Theory  = NewTheory
    return(Theory)
```

Abbildung 16: Postpruning Algorithmus [Fürn97]

4.1.1 REP

Einer der bekanntesten Postpruning-Algorithmen ist das sogenannte Reduced Error Pruning (REP). REP wird sehr häufig beim Prunen von Entscheidungsbäumen eingesetzt [Fürn97] und es gibt mehrere direkte Umsetzungen für das Regellernen, wie z.B. in [BrPa91]. REP implementiert 1:1 den Algorithmus aus Abbildung 16. Die verschiedenen Varianten unterscheiden sich lediglich in der Art wie eine Hypothese geprunt wird. Das Grundprinzip des Prunings ist hier, dass ein oder mehrere Pruningoperatoren auf eine Hypothese angewendet werden und so neue Hypothesen erzeugt werden. Aus der Menge der neu erzeugten Hypothesen wird dann die beste Hypothese gewählt. Das Prunen wird solange durchgeführt bis die Genauigkeit der geprunten Hypothese auf dem PruningSet schlechter ist, als die Genauigkeit der ungeprunten Hypothese.

Obwohl in [BrPa91] gezeigt wurde, dass REP die Genauigkeit einer Hypothese auf verrauschten Daten verbessert, gibt es einige Probleme bei der Anwendung von REP, die nicht zu vernachlässigen sind. Die Zeitkomplexität von REP ist eines der größten Probleme des Algorithmus. Sie liegt bei $O(n^4)$, wobei n die Anzahl der Beispiele ist. Ein intuitiver Beweis ist in [Fürn97] und [Cohe93] zu finden. Hier werden nur kurz die Komplexitäten die dazu beitragen zusammengefasst. Das Lernen einer Hypothese auf einer ausreichend verrauschten Trainingsmenge kann mit $O(n^2)$ abgeschätzt werden [Cohe95]. Die Annahme dahinter ist, dass jedes Beispiel der *n* Beispiele von einer Regel in der Hypothese abgedeckt wird. Die Hypothese umfasst damit *n* Regeln, die in der Pruningphase wieder verfeinert werden. In jedem Schleifendurchlauf werden *n* neue Hypothesen erzeugt, die auf der Pruningmenge getestet werden um die beste Hypothese zu finden. Nimmt man an, dass die Pruningmenge *n* Beispiele umfasst, dann ist das Testen der Hypothesen in $O(n^2)$ Zeit möglich. Geht man davon aus, dass REP die Hypothese auf die korrekte Hypothese prunt, wird jede der *n* Regeln mindesten *n*-mal betrachtet, was $O(n^2)$ Zeit benötigt. Dies führt zu einer Gesamtkomplexität von $O(n^4)$.

Ein weiteres Problem ist die Suche nach Vereinfachungen [FüWi94]. REP versucht die Genauigkeit auf der Trainingsmenge zu maximieren und macht dies von einer komplexen hin zu einer einfachen Hypothese. Die Suche ist, wie oben skizziert, sehr langsam und es gibt auf dem Weg zur geprunten Hypothese viele Möglichkeiten in einem lokalen Optimum stecken zu bleiben. Neben der Suche nach einer geprunten Hypothese ist das Aufteilen der Trainingsbeispiele schon ein Problem beim Lernen selbst. Das Aufteilen der Trainingsbeispiele kann dazu führen, dass wichtige Regeln, die die Zielhypothese beschreiben, nicht gelernt werden. Der Grund dafür ist, dass sich relevante Beispiele, die benötigt werden um die Regeln zu lernen, ausschließlich in der Pruningmenge befinden. Umgekehrt kann es sein, dass korrekt gelernte Regeln wieder vollständig aus der Hypothese entfernt werden, da sich keine Beispiele in der Pruningmenge befinden, die von diesen Regeln abgedeckt werden.

Um das Effizienzproblem von REP in den Griff zu bekommen, wurde von [Cohe93], eine Modifikation von REP vorgeschlagen. Der Algorithmus *Grow* lernt, wie REP, eine überangepasste Hypothese. Diese wird aber im Vergleich zu REP in der Pruningphase nicht sukzessive vereinfacht, sonder dient als Ausgangspunkt für die Suche nach einer neue Hypothese auf dem PruningSet. Grow generalisiert in der Pruningphase die Regeln einer Hypothese, indem eine endliche Anzahl Attributtests entfernt werden, sodass der Fehler auf der Growingmenge am geringsten ansteigt. Die generalisierten Regeln werden der Hypothese wieder hinzugefügt. Aus der so erweiterten Hypothese wird nach und nach eine Regel ausgewählt und einer neuen Hypothese hinzugefügt, bis ein weiteres Hinzufügen einer Regel die Genauigkeit auf der Pruningmenge nicht mehr erhöht. Diese Top-Down Strategie ist, wie in [Cohe93] experimentell nachgewiesen, mindestens genauso gut wie REP, dabei aber deutlich schneller. Zum Vergleich verhält sich die Zeitkomplexität des Prunens bei *Grow* proportional zu $O(n^2 * \log n)$.

Selbst die SeCo- Strategie ist für REP ein Problem [FüWi94]. Die Strategie ähnelt der Divide-and-Conquer Strategie, wie sie beim Entscheidungsbaumlernen zugrunde liegt. Beim Lernen von Entscheidungsbäumen, wird die Menge, in der sich die Trainingsbeispiele befinden, durch den Entscheidungsbaum unterteilt. Diese Unterteilung bilden disjunkte Mengen, also Mengen, in denen kein Beispiel doppelt vorkommt. Eine Unterteilung ist immer abhängig von der Unterteilung, die direkt vor dieser gemacht wurde. Nach dem Lernen eines Baumes, wird REP rekursiv auf diesen angewendet und die disjunkten Mengen werden wieder zu größeren disjunkten Mengen vereinigt. Das Prunen eines Astes hat dabei keinen Einfluss auf einen anderen Ast, da die Entscheidungen in einem Baum immer nur von der direkten Vorgängerentscheidung abhängen. Dagegen wirkt sich das Prunen

einer Regel auf andere Regeln aus, da die Regeln generalisiert werden und somit mehr Beispiele abdecken. In der Lernphase sollten eigentlich diese Beispiele schon entfernt werden, um das Lernen nachfolgender Regeln nicht zu beeinflussen, werden aber weiter mitgeführt. Durch diese Beispiele kann es sein, dass an einer Stelle während des Lernens eine falsche Entscheidung getroffen wird und ein Attributtest einer Regel hinzugefügt wird, der sich im Nachhinein als schlecht erweist. Diese Entscheidungen können auch durch das Prunen nicht wieder rückgängig gemacht werden.

4.2 Prepruning

Im Gegensatz zum Postpruning versucht das Prepruning eine Überangepasstheit der Hypothese schon während des Lernens zu vermeiden. Die Prepruning Algorithmen terminieren das Lernen nicht erst wenn alle positiven Beispiele abgedeckt oder kein negatives Beispiel mehr abgedeckt wird, sondern entscheiden durch Stopkriterien (vgl. Abbildung 4; *Literal -/ClausStoppingCriterion*), wann der Lernprozess beendet wird. In beiden Prozeduren kann eine Heuristik implementiert werden, die diese Entscheidung übernimmt. Das *LiteralStoppingCriterion* entscheidet, wann eine Regel genug verfeinert wurde, indem sie anhand einer Heuristik überprüft, ob genügend positive Beispiele abgedeckt und genügend negative Beispiele ausgeschlossen werden. Das *ClauseStoppingCriterion* entscheidet, wann genügend Regeln in der Hypothese enthalten sind, sodass die Zielhypothese hinreichend genau beschrieben wird. Dies führt dazu, dass der Lernprozess etwas schneller ist und die gelernten Hypothesen kleiner sind, als wenn eine überangepasste Hypothese gelernt wird.

4.2.1 Stopkriterien für das Prepruning

Im Wesentlichen werden hier drei verschiedene Stopkriterien für das Prepruning vorgestellt und im weiteren Verlauf untersucht. Die einfachste Art das Verfeinern einer Regel zu stoppen ist ein sogenanntes *CutOff-Kriterium* [Fürn97]. Das CutOff-Kriterium wurde als erstes im Algorithmus FOSSIL [Fürn94b] implementiert und legt einen Schwellwert für eine Heuristik fest. Die Heuristik, die in FOSSIL verwendet wird, ist die Correlation Heuristik (vgl. Gl. 3.18 / Abb. 12), aber jede andere Heuristik ist für das CutOff-Kriterium denkbar. Ein optimaler Schwellwert für Correlation liegt, wie er in [Fürn94b] gefunden wurde, bei ca. 0,3. Ist die Bewertung einer Regel hinreichend hoch bzw. niedrig, so wird die Verfeinerung der Regel abgebrochen. Analog dazu wird das Lernen einer weiteren Regel abgebrochen, falls keine Regeln gefunden werden können, die diesen Schwellwert überschreiten bzw. unterschreiten. Die Annahme dahinter ist, dass keine weitere Regel mehr gefunden werden kann, die die restlichen positiven Beispiele abdeckt bzw. negativen Beispiele ausschließt, und es sich bei diesen Beispielen um verrauschte Daten handelt. Je nach Heuristik und Schwellwert werden verschiedene und unterschiedlich komplexe Hypothese gelernt. Als Beispiel soll hier die Heuristik Correlation dienen. Setzt man den Schwellwert des Stopkriteriums auf 1.0 – den Maximalwert dieser Heuristik -, so lernt ein Lernalgorithmus ausschließlich leere Hypothesen, da keine Regel gut genug ist, um in die Hypothese aufgenommen zu werden. Ein Setzen des Schwellwerts auf 0.0 führt dazu, dass der Lernalgorithmus eine konsistente und vollständige Hypothese lernt, da jede Regel zugelassen wird.

Eine andere Möglichkeit das Lernen vorzeitig abzubrechen basiert auf dem *Minimum Description Length Prinzip (MDL)* [Riss78]. Das MDL Prinzip stammt aus der Informationstheorie. Das Grundprinzip ist hier: „Wenn Regelmäßigkeiten in einer Datenmenge vorkommen, können diese benutzt werden um die Datenmenge zu komprimieren“, d.h. es werden weniger Bits benötigt um eine Datenmenge zu beschreiben. Je mehr Regelmäßigkeiten in einer Datenmenge vorkommen, desto mehr kann man die Daten komprimieren [GrMP05]. Auf das Lernen von Regeln übertragen bedeutet dies, dass die Datenmenge die Trainingsmenge mit den Beispielen ist. Die Regelmäßigkeiten in der Trainingsmenge können durch Regeln komprimiert werden. Eine Hypothese nach dem MDL-Prinzip

ist also eine Hypothese, für die Gleichung 4.1 erfüllt ist. Eine optimale Hypothese minimiert die linke Seite der Gleichung 4.1.

$$L(H) + L(E|H) < L(E) \quad (4.1)$$

L(H) ist die Länge der Hypothese in Bits, L(E) ist die Länge der Trainingsmenge in Bits, und L(E|H) ist die Anzahl der Bits, die benötigt werden um die Beispiele zu beschreiben, wenn die Hypothese gegeben ist. Die Hypothese wird also eher kurze Regeln und wenige Regeln enthalten.

Eine Variante des MDL-Prinzips ist in FOIL [QuCJ95], unter dem Namen Encoding Length Restriction, implementiert. Diese Variante vergleicht die Anzahl der Bits, die nötig sind eine Regel zu codieren, mit der Anzahl der Bits, die benötigt werden um die Beispiele, die von der Regel abgedeckt werden, zu codieren. Die Anzahl der Bits, die benötigt werden eine Regel zu kodieren ist in FOIL durch Gleichung 4.1 gegeben.

$$FOIL\ Encoding\ Length\ Restriction: len(r) = \log_2(P+N) + \log_2 \binom{P+N}{p} \quad (4.2)$$

Die Gleichung 4.2 kodiert die Anzahl der Bits die nötig sind um die verbleibende Trainingsmenge und die Regel zu übertragen. In [FüFl05] wurde gezeigt, dass die in FOIL benutze Variante eine Anomalie aufweist, falls die Anzahl der positiven Beispiele größer ist als die Anzahl der negativen Beispiele. Gilt P>N in der Trainingsmenge so liegt das Maximum der Funktion nicht einer Abdeckung von P positiven Beispielen, sondern das Maximum liegt bei $\frac{(P+N)}{2}$. In Abbildung 17 ist die Anomalie im PN-Raum visualisiert. Das linke Bild zeigt das Verhalten der Isometriken für P<N, wohingegen das Bild auf der rechten Seite den Fall P>N zeigt. Das Maximum ist hier durch die rote Linie gekennzeichnet.

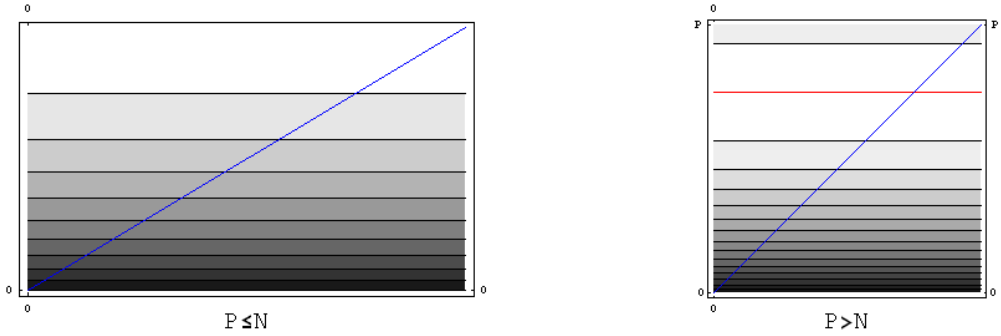

Abbildung 17: FOIL-MDL Anomalie

Eine andere Möglichkeit die Länge einer Regel zu berechnen wurde in [Pfah95] vorgestellt und enthält einige Veränderungen gegenüber der in FOIL benutzen MDL-Formel. Die Formel ist in Gleichung 4.3 aufgeführt.

$$\left.\begin{aligned} len(ruleset) &= n_{nc} * e_{nc} + \textstyle\sum_i len(r_i) \\ len(r_i) &= rc_i + n_c * e_c \end{aligned}\right\} \quad (4.3)$$

In der Formel ist n_{nc} die Anzahl der Beispiele, die nicht von der Hypothese abgedeckt werden, und e_{nc} die dazugehörige Entropie der Menge der nichtabgedeckten Beispiele. Weiter ist rc_c die Länge in Bits einer Regel, die wie in Gleichung 4.4 berechnet wird, und n_c die Anzahl der Beispiele die von der Regel abgedeckt werden und e_c die Entropie der abgedeckten Beispiele. Gleichung 4.4 berechnet die

mögliche Länge einer Regel in Bits, d.h. wie viele Möglichkeiten gibt es eine Regel der Länge $Length_i$ mit zusätzlichen Attributtests zu verlängern.

$$rc_i = \log_2\left(\binom{N_{pt}}{Length_i}\right) \quad (4.4)$$

	Wahrscheinlichkeit p=								
Freiheitsgrad	**0,005**	**0,01**	**0,025**	**0,05**	**0,1**	**0,5**	**0,9**	**0,975**	**0,99**
1	0	0	0	0	0,02	0,45	2,71	3,84	6,63

Tabelle 8: χ^2-Test Tabelle für einen Freiheitsgrad[Wiki08]

Wenn die Länge einer Regel mit MDL berechnet wird, muss diese noch um den Faktor $\log_2(Length_i!)$ verkleinert werden, da die Reihenfolge der Attributtests in einer Regel irrelevant ist.

Das dritte und letzte Stopkriterium, das hier vorgestellt wird, ist der *Signifikanztest*, der von [ClNi89] im Algorithmus CN2 oder in mFOIL von [DzBr92] benutzt wird. Der Signifikanztest vergleicht die Standardverteilung der positiven und negativen Beispiele in einer Trainingsmenge mit der Verteilung der Beispiele, die von einer Regel abgedeckt werden. Regeln oder Verfeinerungen einer Regel, deren Verteilung nicht signifikant von der Standardverteilung abweichen, werden nicht in die Hypothese mit aufgenommen bzw. nicht weiter verfeinert. Die *likelihood ration statistic (LRS)* [ClNi89] ist annähernd wie eine Chi-Quadrat-Verteilung mit einem Freiheitsgrad verteilt und kann deswegen mit dieser verglichen werden. Ist der Wert der *LRS* größer als ein vorgegebener Schwellwert, so wird angenommen, dass eine Regel bzw. Regelverfeinerung signifikant ist. Die Schwellwerte können aus Tabelle 7 abgelesen werden.

$$LRS: h_{LRS} = 2 * \left(\mathrm{p} * \log_2 \frac{p}{e_p} + n * \log_2 \frac{n}{e_n}\right) \quad (4.5)$$

$$e_p = (p+n) * \frac{P}{P+N} \quad (4.6), \quad e_n = (p+n)\frac{N}{P+N} \quad (4.7)$$

	positiv klassifiziert	negativ klassifiziert	
positives Beispiel	8	3	11
negatives Beispiel	2	7	9
	10	10	20

Tabelle 9: Beispiel LRS und Chi-Quadrat

Der χ^2-Test errechnet mit der Konfusionsmatrix die Wahrscheinlichkeit, dass eine bestimmte Häufigkeitsverteilung einer Menge signifikant ist. Die Werte eines χ^2-Test lassen sich aus einer Tabelle ablesen, als Beispiel soll hier die Tabelle 8 dienen. Die Tabelle, aus [Wiki08] übernommen, umfasst die Werte des χ^2-Test mit einem Freiheitsgrad und verschiedenen Wahrscheinlichkeiten. Der Vergleich der LRS-Heuristik mit dem χ^2 -Test liefert die Wahrscheinlichkeit, dass die Häufigkeitsverteilung einer Regel sich signifikant von der Standardverteilung der Beispiele unterscheidet.

Als Beispiel soll die Konfusionsmatrix aus Tabelle 8 dienen. Der Chi-Quadrat-Test wird wie in Gleichung (4.8) aus der Konfusionsmatrix berechnet:

$$\chi^2 = \sum_{i,j} \frac{(k_{i,j} - E(k_{i,j}))^2}{E(k_{i,j})} = \frac{(k_{0,0} + k_{0,1} + k_{1,0} + k_{1,1}) * (k_{0,0} * k_{1,1} - k_{1,0} * k_{0,1})^2}{(k_{0,0} + k_{0,1}) * (k_{1,1} + k_{1,0}) * (k_{0,0} + k_{1,0}) * (k_{1,1} + k_{0,1})} \quad (4.8)$$

Setzt man die Werte aus Tabelle 9 in die Gleichungen 4.8 ein so erhält man einen Wert von $\chi^2 = \frac{20*(56-6)^2}{(10*10*11*9)} = 5{,}051$ und für die LRS Heuristik $h_{LRS}(n,p) = 3{,}85$. Anhand der Werte kann man aus Tabelle 7 ablesen, dass die Häufigkeitsverteilung bzw. die Verteilung der positiven und negativen Beispiele der Regel mit mindestens 97,5%-iger Sicherheit von der Standardverteilung abweichen. Die Regel wird damit in die Hypothese mit aufgenommen bzw. die Regel wird weiter verfeinert und gilt damit als signifikant.

Die Isometriken der LRS sind in Abbildung 18 zu sehen. Zwei Signifikanzlevel, bei denen Regeln nicht mehr in die Hypothese aufgenommen werden, sind zusätzlich zu den Isometriken eingetragen. Regeln, die innerhalb der weißen Fläche liegen, weichen mit 95%-iger Sicherheit von der Standardverteilung ab, wohingegen Regeln, die im ersten leicht grauen Bereich liegen, mit 99%-iger Sicherheit von der Standardverteilung der Beispiele abweichen. Für Regeln, die in diesen beiden Bereichen liegen, kann man ausschließen, dass sie sehr gut für vorher nicht gesehene Beispiele verallgemeinern und können damit aus der Hypothese ausgeschlossen werden.

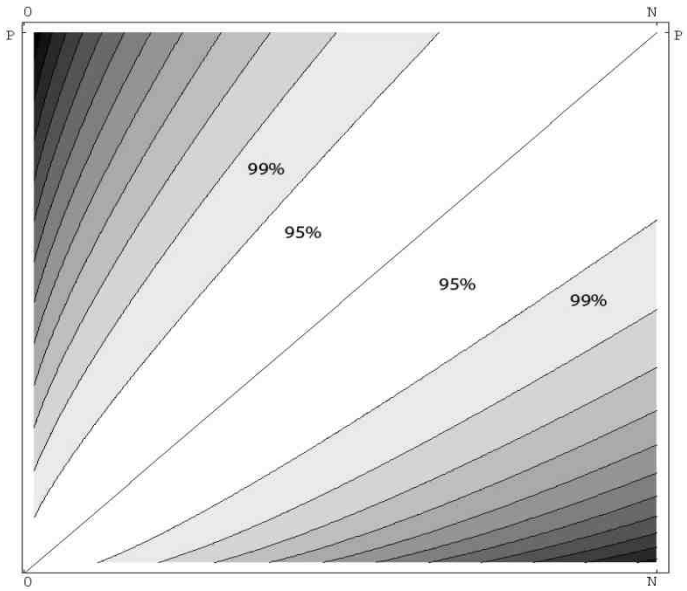

Abbildung 18: Likelihood Ratio Statistic Isometriken

4.3 Combining Post-& Prepruning

Obwohl REP die Genauigkeit einer Hypothese auf verrauschten Daten durchaus erhöhen kann, besitzt der Algorithmus einige Schwächen, wie in Kapitel 4.1 beschrieben. Um zu verhindern, dass die Postpruningphase beim Lernen mit einer viel zu komplexen Hypothese umgehen und den Großteil des Lernens wieder rückgängig machen muss, bietet es sich an, schon beim Lernen darauf zu achten, dass die Hypothese nicht zu komplex wird. Ein weiterer Vorteil den der Einsatz des Prepruning bringt, ist eine Verringerung der Laufzeit beim Lernen. Ein intuitiver Ansatz ist deshalb das Post- und das Prepruning zu kombinieren.

In [Cohe93] wurde ein solcher Ansatz vorgestellt. Die Lernphase von Grow wurde durch zusätzliche Stopkriterien, die auf dem MDL-Prinzip basieren, beschleunigt. Das Prepruning soll die Überanpassung an die Trainingsdaten nicht vollständig vermeiden, sondern nur reduzieren, damit die gelernte Hypothese weniger komplex ist. Ein Nachteil bei dieser Herangehensweise ist, dass das

Prepruning eine Hypothese finden kann, die in der Postpruningphase nicht verwendet werden kann, da sie zu simpel ist.

Ein anderer Ansatz ist in [Fürn97] beschrieben. Die Lernphase besteht hier nicht nur aus dem Lernen einer überangepassten Hypothese, sondern wird so erweitert, dass in der Lernphase ein geeigneter Startpunkt (Hypothese) für die Pruningphase gesucht wird. Das kombinierte Post- und Prepruning sei an dieser Stelle nur erwähnt, da es nicht Teil der Arbeit ist. Genauere Informationen finden sich in [Fürn94a] und [Fürn97].

4.4 Integrating Post- & Prepruning

Eine andere Art beide Pruningmethoden zusammenzuführen, bietet das integrierte Post- und Prepruning. Die Idee dahinter ist, dass nicht als erstes eine konsistente und vollständige Hypothese auf den Trainingsdaten gelernt wird und somit möglicherweise auch unnötige Attributtests einer Regel hinzugefügt werden, sondern dass zuerst eine vollständige und konsistente Regel gelernt und direkt geprunt wird. Dies soll dem Lerner ermöglichen die Beispiele zu entfernen, die von einer Regel abgedeckt werden, nachdem diese geprunt wurde, und so das Hinzufügen eines unnötigen Attributtests zu einer Regel zu vermeiden. Das sogenannte *Inkrementelle Reduced Error Pruning (IREP)* [FüWi94] implementiert diese Strategie. Der darauf basierende Algorithmus RIPPER [Cohe95] erweitert den Algorithmus so dass die Fehlerraten des Algorithmus gesenkt werden. Eine Implementierung des RIPPER, namens JRIP [WFTH99], wird in Kapitel 4.4.2 genauer vorgestellt.

4.4.1 IREP – Incremental Reduced Error Pruning

```
procedure I-REP(Examples, SplitRatio)
    Theory = ∅
    while POSITIVE(Examples) ≠ ∅
        Clause = ∅
        SPLITEXAMPLES(SplitRatio, Examples, GrowingSet, PruningSet)
        Cover = GrowingSet
        while NEGATIVE(Cover) ≠ ∅
            Clause = Clause ∪ FINDLITERAL(Clause, Cover)
            Cover = COVER(Clause, Cover)
        loop
            NewClause = BESTSIMPLIFICATION(Clause, PruningSet)
            if ACCURACY(NewClause, PruningSet) < ACCURACY(Clause, PruningSet)
                exit loop
            Clause = NewClause
        if ACCURACY(Clause, PruningSet) ≤ ACCURACY(fail, PruningSet)
            exit while
    Theory = Theory ∪ Clause
    Examples = Examples – Cover
return(Theory)
```

Abbildung 19: Pseudocode IREP

Abbildung 19 ist der Algorithmus IREP in Pseudocode gegeben, wie er auch in [Fürn97] zu finden ist. Die Trainingsmenge wird als erstes wieder in zwei kleinere Mengen aufgeteilt, die Growing- und Pruningmenge. Im Unterschied zum REP geschieht dies aber nicht nur bevor das Lernen einer Hypothese beginnt, sondern immer vor dem Lernen einer Regel. Nachdem eine Regel gelernt wurde, wird diese direkt geprunt (*BestSimplification*) und versucht die Genauigkeit der Regel auf der Pruningmenge zu erhöhen. Im ursprünglichen Algorithmus wird der *find-best-simplification*-Operator

auf eine Regel angewendet, aber andere oder mehrere Pruningoperatoren sind hier durchaus vorstellbar. Dies wird solange gemacht, bis eine geprunte Regel eine schlechtere Genauigkeit besitzt als ihre direkte Vorgängerregel. Die zweite *if*-Abfrage soll gewährleisten, dass die gefundene Regel eine Mindestgenauigkeit auf der Pruningmenge hat bzw. mehr positive als negative Beispiele abdeckt. Ist eine Regel gefunden worden und zur Hypothese hinzugefügt worden, so werden die Beispiele, die von der Regel abgedeckt werden, aus der Trainingsmenge entfernt. Die Beispiele werden nicht nur aus der Growing-Menge entfernt, sondern auch aus der Pruning-Menge. Bei der nächsten Iteration werden die Beispiele erneut in zwei Mengen geteilt und die Suche nach neuen Regeln beginnt von vorne.

Der Algorithmus besitzt Eigenschaften, die die Probleme von REP lösen, bringt aber auf der anderen Seite neue Probleme mit. Das Effizienzproblem von REP ist eindeutig reduziert worden durch die Art wie eine Hypothese bei IREP gelernt wird. Dadurch, dass die Regeln einer Hypothese gelernt werden und gleich danach geprunt werden, ist die Zeitkomplexität von IREP für den *delete-last-condition* Operator proportional zu $O(n * \log^2(n))$ [Fürn97]. Die Separate-and-Conquer Strategie ist für diesen Algorithmus kein Problem mehr, da die Beispiele erst aus der Trainingsmenge entfernt werden, nachdem eine Regel komplett fertig gelernt und geprunt wurde. Die Herangehensweise wie eine Hypothese gelernt wird, ist hier ähnlich zu der in Grow benutzten Weise mit dem Unterschied, dass nicht als erstes eine überangepasste Hypothese gelernt werden muss, sondern diese direkt aus einzelnen gelernten und geprunten Regel zusammengesetzt wird. Somit ist der Algorithmus auch weniger Anfällig für lokale Optima [Fürn97]. Auch das Problem mit der Aufteilung der Trainingsmenge in Growing- und Pruningmenge ist hier zumindest reduziert. Das Problem beschränkt sich hier nur noch auf einzelne Regeln und nicht mehr auf eine komplette Hypothese.

4.4.2 RIPPER und JRIP

Der Algorithmus RIPPER [Cohe95] ist eine Modifikation des ursprünglichen IREP [FüWi94]. Das zugrunde liegende Prinzip ist in beiden Algorithmen das Gleiche. Es werden nach und nach Regeln gelernt und gleich geprunt. Die so gelernten Regeln werden sukzessive einer Regelmenge hinzugefügt bis ein Abbruchkriterium erfüllt ist. Das Abbruchkriterium bei IREP misst den Fehler einer Regel auf der Pruningmenge. Wenn der Fehler größer als 50% ist wird das Lernen abgebrochen. In [Cohe95] wurde gezeigt, dass IREP mit diesem Abbruchkriterium oft das Lernen zu früh stoppt und dadurch anfälliger für das *Small Disjunct Problem* ist [HoAP89]. *Small Disjuncts* sind Regeln, die nur sehr wenige positive und keine negativen Beispiele abdecken. In [HoAP89] wurde gezeigt, dass Small Disjuncts, obwohl sie nur wenige positive Beispiele abdecken, für einen Großteil des Fehlers bei der Klassifikation von ungesehenen Beispielen verantwortlich sind. Um das *Small Disjunct Problem* zu umgehen führt Ripper ein auf dem MDL-Prinzip basierendes Abbruchkriterium ein. Nachdem eine Regel gelernt wurde, wird die Description Length (DL) der Regelmenge und der verbleibenden Beispiele errechnet. Ripper stoppt das Lernen von Regeln, wenn die so erhaltene DL größer als die kleinste, bis jetzt errechnete DL + d Bits ist. Ein Wert für d, der sich als besonders gut herausgestellt hat, ist 64 Bits.

RIPPER lernt Regel mit dem FOIL Algorithmus. Die Suchheuristik, die FOIL verwendet ist in Abbildung 19 visualisiert und durch die Formel aus Gleichung 3.19 gegeben. Die durch FOIL gelernten Regeln, werden dann auf der Pruningmenge vereinfacht. IREP verwendet zum Vereinfachen der gelernten Regel den *delete-last-condition* Operator, der in jedem Schritt ein Attributtest aus der Regel löscht und zwar so, dass die Heuristiken $\frac{p}{p+n}$ (IREP-2) bzw. $\frac{p+(N-n)}{P+N}$(IREP) einen bestimmten Wert nicht unterschreiten. In RIPPER ist zum Prunen ein anderes Maß implementiert worden, das in Gleichung (4.9) zu sehen ist. Die Heuristik, die vom RIPPER zum Prunen benutzt wird ist äquivalent

zu Heuristik Precision [FüFl05]. Demensprechend sind die Isometriken der Heuristik aus Gleichung 4.9 äquivalent zu denen aus Abbildung 10 (vgl. Kapitel 3.3.1).

$$RIPPER: h_{RIPPER}(n,p) = \frac{p-n}{p+n} \quad (4.9)$$

Der Algorithmus RIPPER löscht während dem Prunen nicht nur immer einen Attributtest einer Regel, sondern eine endliche Anzahl von Attributtest, sodass die Heuristik aus Gleichung 4.9 maximiert wird. Diese Modifikation bilden die Grundlage des IREP*-Algorithmus [Cohe95], der in RIPPER Verwendung findet. Die so gelernte Hypothese wird in RIPPER zusätzlich noch einer Optimierungsphase übergeben.

Die Optimierungsphase soll die gefundene Hypothese noch weiter verbessern. Dies geschieht nach einem einfachen REP-Prinzip. Die Hypothese, die während der ersten Phase gelernt wurde, wird Regel für Regel verbessert. Die Regeln aus der Hypothese werden in der Reihenfolge, in der sie gelernt wurden betrachtet. Für eine Regel R_i aus der Hypothese $\{R_1, R_2, \dots, R_i, \dots, R_n\}$ werden zwei neue Regeln erzeugt. Die erste erzeugte Regel wird Replacement von R_i genannt. Das Replacement wird auf der Growingmenge gelernt, sodass der Fehler der Hypothese ohne R_i aber mit dem Replacement verringert wird. Die zweite Regel, die aus der Regel R_i erzeugt wird, wird Revision von R_i genannt. Die Revision von R_i wird auf der Growingmenge weiter verfeinert, indem weitere Attributtests zu R_i hinzugefügt werden. Nachdem das Replacement und die Revision von R_i erzeugt wurden, wird anhand des MDL-Prinzips entschieden, welche Regel in der endgültigen Hypothese enthalten ist. Also wird entweder R_i, das Replacement oder die Revision wieder in die Hypothese aufgenommen. Die Optimierungsphase soll die Effekte des konventionellen REP annähern, sodass die Genauigkeit der Hypothese noch erhöht wird.

Sind nach der Optimierungsphase noch positive Beispiele nicht abgedeckt, wird versucht, diese durch einen weiteren Durchlauf von IREP* [Cohe95] zu beschreiben. Durch diesen Ablauf hat der Algorithmus seinen Namen. RIPPER steht für Repeated Incremental Pruning to Produce Error Reduction. Die Implementierung von RIPPER, die in dieser Arbeit als Vergleichsalgorithmus dient ist JRIP [WFTH99]. JRIP ist eine Javaimplementierung von RIPPER in der WEKA-Bibliothek [WFTH99].

Kapitel 5 Technische Umsetzung im SeCo-Framework

SeCo ist eine Abkürzung und steht für Separate-And-Conquer. Ein Framework ist eine Menge von zusammenarbeitenden Klassen, die in ihrer Gesamtheit eine generische Implementierung für eine bestimmte Klasse von Problemen bieten. Das SeCo-Framework bietet somit also eine generische Implementierung für Separate-And-Conquer Regellern Algorithmen. In diesem Kapitel wird zuerst kurz das zugrunde liegende SeCo-Framework beleuchtet. In einem weiteren Schritt werden die Prozeduren, die für das Pruning notwendig sind, identifiziert und beschrieben. Dazu wird ein generischer SeCo-Pruning Algorithmus formuliert, der auf dem SeCo-Algorithmus aus Abbildung 4 basiert. Im weiteren Verlauf des Kapitels werden die notwendigen Schnittstellen und Klassen definiert, sodass der generische Pruningalgorithmus implementiert werden kann. Am Ende des Kapitels wird gezeigt, wie durch geschickte Konfiguration und Implementierung der Klassen das Prepruning, Postpruning, die Integration und Kombination beider Methoden umgesetzt worden sind.

5.1 Das SeCo-Framework

Die dem SeCo-Framework zugrunde liegende Programmiersprache ist in Java. Zur Beschreibung der Trainingsmenge, insbesondere der Beispiele und deren Eigenschaften, verwendet das Framework, die WEKA-Bibliothek [WFTH99]. Die Lernalgorithmen in diesem Framework verwenden die Abstrakte Klasse weka.classifiers.Classifier der WEKA-Bibliothek. Mit Hilfe der WEKA-Bibliothek ist es möglich die Lernalgorithmen innerhalb des SeCo-Frameworks zu evaluieren.

Das Framework selbst ist von Prof. Dr. Johannes Fürnkranz implementiert. Die Klasse Covering bildet den Kern des Algorithmus aus Abbildung 4. Die Hypothesensprache, die das Framework unterstützt, sind Entscheidungslisten. Die Regeln einer Entscheidungsliste bestehen aus Konjunktionen von Attributtest. Am Ende einer Entscheidungsliste steht eine Defaultregel, die die Majorityklasse für ein Beispiel vorhersagt, sofern nicht eine Regel, die über der Defaultregel steht, ein Beispiel klassifiziert hat. Der Suchalgorithmus, der in diesem Framework implementiert ist, ist eine Top-Down-Beam-Suche. Bei der Suche nach Regelverfeinerungen bzw. nach Regeln, kann auf eine große Anzahl unterschiedlicher Suchheuristiken zurückgegriffen werden. Einige dieser Heuristiken sind in Kapitel 3 vorgestellt worden.

Das Framework ist in verschiedene Pakete unterteilt, in denen sich jeweils die Klassen für das Sprachmodell, die Heuristiken und die verschiedenen Lernalgorithmen befinden. In Tabelle 10 sind die Pakete mit dem Paketnamen, dem Autor und einer kurzen Beschreibung aufgelistet. Die Abkürzung JF steht für Johannes Fürnkranz.

Paketname	*Autor*	*Beschreibung*
seco.heuristics	JF	Dieses Paket enthält die Klassen für die Suchheuristiken
seco.learners	JF	Dieses Paket enthält die verschiedenen Regellernalgorithmen
seco.model	JF	Dieses Paket enthält die Klassen für das Modell der Hypothesensprache

Tabelle 10: SeCo-Framework Pakete

Methoden, die das Overfitting vermeiden, sind in diesem Framework nur an wenigen Stellen implementiert. Die wichtigsten Methoden, die implementiert sind, sind die Gewährleistung, dass das Hinzufügen einer Regel zu einer Regelmenge die Genauigkeit nicht verringert. Diese Methode ist in der Klasse Covering implementiert und wird im weiteren Verlauf als *IncAcc*[6] bezeichnet. Die zweite Methode ist im Suchalgorithmus implementiert und wird als *ForwardPruning* bezeichnet. Das *ForwardPruning* basiert dabei auf folgender Annahme:

[6] Zusammengesetzt aus „increasing“ und „accuracy“.

Prozedur	*Beschreibung*
RuleStoppingCriterion	Entscheidet, wann das Lernen von Regeln stoppt.
InitializeRule	Legt Regel fest, mit der die Verfeinerung begonnen wird.
EvaluateRule	Bewertet eine Regel mit einer Heuristik
SelectCandidates	Wählt Kandidatenregeln aus, die verfeinert werden sollen
RefineRule	Erzeugt alle Verfeinerungen einer Regel
StoppingCriterion	Entscheidet, wann das Verfeinern von Regeln stoppt
PostProcess	Ermöglicht die Nachbearbeitung von Regeln
FilterRules	Filtert Kandidatenregeln nach einer Iteration der Verfeinerung
Zusätzliche Prozeduren für das Prunen	
OuterSplit	Teilt die Trainingsmenge in 2 Mengen auf. Beispiele werden zufällig gezogen.
InnerSplit	Teilt die Trainingsmenge in 2 Mengen auf. Beispiele werden zufällig gezogen.
PostProcessRule	Ermöglicht die Nachbearbeitung einer Regel

Tabelle 11: Variable Prozeduren für das Prunen. Teile aus [THIE05]

Deckt eine Regel R_1 bereits p positive und n negative Beispiele ab und wird von einer Heuristik mit dem Wert h bewertet, so wird eine neue Regel R_P erzeugt, die p positive und keine negativen Beispiele abdeckt. Ist die Bewertung der Regel R_p schlechter als die Bewertung der momentan besten Regel im Strahl, so wird das Regellernen unterbrochen, da keine Verfeinerung der Regel R_1 eine besser Bewertung erhalten kann als die Bewertung der momentan besten Regel.

5.1.1 Ein generischer Pruningalgorithmus

```
procedure PruningSeparateAndConquer(Examples)

Theory = ∅
OUTERSPLIT (Examples, Splitratio,GrowingSet,PruningSet)
while POSITIVE(GrowingSet) ≠ ∅
    Rule = FindBestRule(GrowingSet)
    Covered = COVER(Rule, GrowingSet)
    Rule = POSTPROCESSRULE(Rule, PruningSet)
    if RULESTOPPINGCRITERION(Theory,Rule,GrowingSet, PruningSet)
        exit while
    Examples = Examples \ Covered
    INNERSPLIT (Examples, Splitratio,GrowingSet,PruningSet)
    Theory = Theory ∪ Rule
Theory = POSTPROCESS(Theory, GrowingSet, PruningSet)
return(Theory)
```

Abbildung 20: Ein generischer SeCo-Pruning-Algorithmus

Das Framework bildet den Kern des Algorithmus aus Abbildung 4. Mit „Kern des Algorithmus“ ist gemeint, dass einige Prozeduren aus Abbildung 4 nur triviale Werte zurückgegeben. Die Funktionen, die für das Prunen notwendig sind, sind in Tabelle 11 nochmals aufgeführt und beschrieben. Mit der Konfiguration dieser Prozeduren ist es möglich das Prepruning und begrenzt das Postpruning umzusetzen. Um den Kern-Algorithmus dahingehend zu erweitern, dass auch die Kombination und Integration beider Verfahren durch geschickte Konfiguration unterstützt wird, sind weitere Prozeduren notwendig, die in Tabelle 11 beschrieben sind. Eine Anpassung der Prozeduren und zwei weitere Variablen sind nötig, damit der Kernalgorithmus das Prunen vollständig unterstützt.

Mit den oben genannten und angepassten Prozeduren sieht der Kernalgorithmus wie in Abbildung 20 aus. Die *Split*-Prozeduren sorgen dafür, dass vor dem Lernen die Trainingsmenge in zwei kleinere Mengen aufgeteilt wird, sowie während dem Lernen (vgl. IREP). Die Prozedur *PostProcessRule* soll es ermöglichen eine Regel, nachdem sie gelernt wurde, nachzubearbeiten. Die meisten Funktionen erhalten sowohl das GrowingSet als auch das PruningSet als Parameter, da die Prepruning-Entscheidungen auf allen Beispielen gemacht werden, wohingegen die Entscheidungen das Lernen zu stoppen, z.B. bei IREP nur auf dem PruningSet stattfinden. Welche Menge nun wirklich verwendet wird, hängt von der Art des Algorithmus ab. Durch geschickte Konfiguration der Prozeduren ist es mit dem generischen Algorithmus nun möglich alle in Kapitel 4 genannten Arten des Prunens zu implementieren. Die Prozedur *StoppingCriterion*, welche in der Prozedur *FindBestRule* verwendet wird, muss nicht angepasst werden. Die Prozeduren, die für das Pruning wichtig sind, sind *RuleStoppingCriterion*, *StoppingCriterion*, *PostProcessRule*, *PostProcess*, *InnerSplit* und *OuterSplit*.

5.2 Realisierung des Prunings

In diesem Abschnitt werden die wichtigsten Klassen und Schnittstellen vorgestellt, die nötig sind um das SeCo-Framework so zu erweitern, dass eine Vielzahl verschiedener Pruningalgorithmen unterstützt werden. Die Grundlage für die Pruningalgorithmen bildet die Klasse Covering, die den Kernalgorithmus aus Abbildung 4 implementiert. Die Anforderung an diese Arbeit war, dass es eine Klasse geben soll, die im Prinzip die gesamte Pruningfunktionalität in sich vereinigt. Die Pruningfunktionalität beinhaltet das Aufteilen der Trainingsmenge, das Prunen von Regeln und Regelmengen, sowie verschiedene Stopkriterien, die den Lernvorgang vorzeitig abbrechen. Damit die Klasse Covering auf die Pruningfunktionalität zugreifen kann, müssen die Methodenaufrufe an der richtigen Stelle in den Quellcode des Algorithmus eingefügt werden. Die einzige Methode, die nicht in dieser Klasse verwendet wird ist die Methode *literalStop*. Diese Methode wird ausschließlich vom Suchalgorithmus verwendet und muss dort in den Algorithmus eingefügt werden. Als Entwurfsmuster bietet sich für die Pruningklassen das Muster „Template Method“ [GHJV95] an. Das Muster wurde gewählt, da sich die konkreten Implementierungen der verschiedenen Pruningalgorithmen nur an sehr wenigen Stellen unterscheiden. Meist in der Art wie geprunt wird oder welches Stopkriterium benutzt wird. Dadurch dass die konkreten Pruningalgorithmen von einer Abstrakten Pruningklasse erben, ist es gewährleistet, dass viele verschiedene Algorithmen implementiert werden können.

In Abbildung 21 sind die Pakete des SeCo-Frameworks (orange) und die Pakete, die das Framework mit Pruningfunktionalität (gelb) erweitern, zu sehen. Das Paket *seco.pruning* enthält die Klassen, die die Pruningalgorithmen implementieren. Die Pakete *seco.pruning.operator* und *seco.pruning.criterion* enthalten die Klassen für die Pruningoperatoren bzw. die Klassen für die Stopkriterien. Im Paket *seco.pruning.model* sind die abstrakten Klassen bzw. die Interfaces enthalten, die von den anderen drei Paketen benötigt werden. Die nachfolgenden Abschnitte beschreiben die Umsetzung des generischen Algorithmus aus Abbildung 20 und die Integration der Klassen in das Framework.

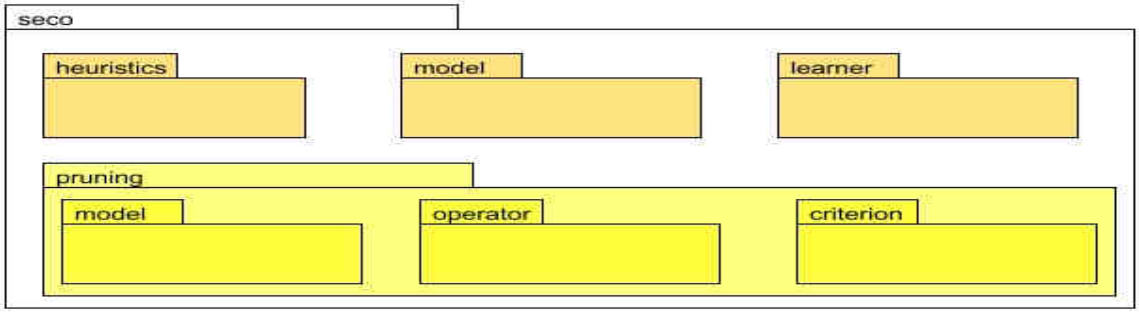

Abbildung 21: Pakete des SeCo-Frameworks

5.2.1 Pruningklassen

```
public abstract class PruningTemplate{
                ⋮
    public abstract Instances[] outerSplit(Instances);
    public abstract Instances[] innerSplit(Instances);
    public abstract CandidateRule postProcessRule(CandidateRule, Instances);
    public abstract RuleSet postProcess(RuleSet, Instances);
    // Methode für das StoppingCriterion
    public abstract boolean literalStop(CandidateRule, Instances);
    // Methode für das RuleStoppingCriterion
    public abstract boolean clauseStop(RuleSet, CandidateRule, Instances);
                ⋮
}
```

Abbildung 22: Klasse PruningTemplate

Die wichtigste Klasse für das Pruning ist die abstrakte Pruningklasse „PruningTemplate“. Die Klasse bildet das Grundgerüst für die Implementierung der Algorithmen und stellt die notwendigen Schnittstellen zur Verfügung, um die oben genannten Prozeduren (*RuleStoppingCriterion, StoppingCriterion, PostProcessRule*, *PostProcess, InnerSplit und OuterSplit*) zu implementieren. In Abbildung 22 sind die wichtigsten Methoden der Klasse im Pseudocode dargestellt. Außer den in Abbildung 22 aufgeführten Methoden stellt die Klasse weitere Methoden und Attribute zur Verfügung, die von den Pruningalgorithmen verwendet werden. Die Konfiguration der Algorithmen wird ebenso von dieser Klasse übernommen. Die später folgende Beschreibung der Algorithmen bezieht sich nur noch darauf, wie die Methoden angepasst werden müssen, um den entsprechenden Algorithmus bzw. Pruningmethode zu erhalten. Methoden, die triviale Werte, wie z.B. *true* oder *false*, zurückgeben bzw. Methoden, die mit den ihnen übergebenen Objekten nichts machen, werden bei der Beschreibung nicht berücksichtigt.

5.2.2 Pruningoperatoren

Für das Pruning werden zwei verschiedene Klassen von Pruningoperatoren unterschieden. Die erste Klasse umfasst die Operatoren, die auf Regelmengen angewendet werden. Die zweite Klasse umfasst die Operatoren, die auf Regeln angewendet werden. Ebenso wie für das Pruning bietet sich für die Pruningoperatoren das Entwurfsmuster „Template Method“ an, da alle Operatoren im Prinzip das gleiche mit einer Regel bzw. Regelmenge machen. Um die Operatoren im Framework umzusetzen sind zwei abstrakte Klassen entstanden, die zwei Methoden, *applyOperator* und *prune*, zur Verfügung stellen. Die *applyOperator*-Methode ist für jeden Regelmengen- bzw. Regeloperator gleich. Das Verhalten eines bestimmten Operators wird durch die *prune*-Methode festgelegt. Ein Regeloperator transformiert eine Regel in eine weitere geprunte Regel, wohingegen ein Regelmengenoperator eine Regelmenge in eine Menge von Regelmengen transformiert. Der Regelmengenoperator erzeugt für jede Regel R_i aus der zu prunenden Hypothese eine weitere Hypothese, in der alle Regeln der ursprünglichen Hypothese enthalten sind, außer der Regel R_i. An die Stelle der Regel R_i wird die geprunte Regel R_i' eingefügt. Die Operatorenklassen und die Erzeugungsvorschrift der geprunten Hypothesen bzw. Regeln sind in Abbildung 23 im Pseudocode zu sehen.

Die Operatoren, die im Verlauf der Arbeit implementiert wurden, sind *delete-rule, delete-last-condition, indentity und find-best-simplification*. Das Verhalten der Operatoren, entspricht dem Verhalten der Operatoren, die in Kapitel 4 beschrieben sind.

```
public abstract class RuleSetOperator{
                    ⋮
    public Vector<RuleSet> applyOperator(RuleSet, Instances){
        Vector<RuleSet> sets = new Vector<RuleSet>;
        for each Rule R_i in RuleSet{
            rs = RuleSet \ R_i;
            r‘ = prune(R_i);
            rs = rs ∪ r‘;
            sets.add(rs);
        }
        return sets;
    }
    protected abstract CandidateRule prune(CandidateRule, Instances);
                    ⋮
}

public abstract class RuleOperator{
                    ⋮
    public CandidateRule applyOperator(CandidateRule, Instances){
         r‘ = pruneRule(R_i);
         return r‘;
    }
     protected abstract CandidateRule prune(CandidateRule, Instances);
                    ⋮
}
```

Abbildung 23: Klasse RuleSet- und RuleOperator

5.2.3 Stopkriterien

Obwohl für jeden Pruningalgorithmus ein eigenes Stopkriterium implementiert werden kann, ist es dennoch sinnvoll die Methoden *literalStop* und *clauseStop* nicht nur als überschreibbare Methoden zu betrachten, sondern auch als Delegationsmethoden. Dies ist deswegen von Vorteil, da so ein Algorithmus nicht nur auf ein bestimmtes Stopkriterium festgelegt ist, sondern derselbe Algorithmus auch mit unterschiedlichen Stopkriterien implementiert werden kann. Aus diesem Grund wurde bei den Stopkriterien dasselbe Entwurfsmuster, das auch bei den Pruningoperatoren verwendet wird, zugrunde gelegt. Ob das Stopkriterium nun als RuleStoppingCriterion oder als StoppingCriterion fungiert, wird durch zwei Interfaces festgelegt, die die entsprechende Methode zur Verfügung stellen. In Abbildung 24 sind die beiden Interfaces kurz im Pseudocode dargestellt. Für die Stopkriterien existiert auch eine abstrakte Klasse. Diese dient lediglich dazu einige Parameter, wie einen Schwellwert oder eine Heuristik für ein Stopkriterium zu konfigurieren.

```
public interface IRuleStoppingCriterion{
    public boolean clauseStop(RuleSet, CandidateRule, Instances);
}
 public interface IStoppingCriterion{
    public boolean literalStop(CandidateRule, Instances);
 }
```

Abbildung 24: Interfaces für Stopkriterien

Als Stopkriterien wurden in dieser Arbeit fünf verschiedene Kriterien implementiert. Das erste Stopkriterium, das implementiert wurde, ist das Kriterium, das in IREP [FüWi94] verwendet wurde. Das Kriterium *MaximumErrorRate* stoppt Regelverfeinerungen bzw. das Hinzufügen von Regeln, wenn der Fehler der gelernten Regeln auf der Pruningmenge 50% übersteigt. Das CutOff-Kriterium

wurde gleich auf zwei Arten implementiert. Die erste stoppt das Regellernen, wenn die Bewertung einer Regel einen bestimmten Schwellwert übersteigt. Da die Bewertung der Regel auf den Regeleigenschaften einer einzigen Regel berechnet wird, d.h. den momentan abgedeckten *tp*, *fp* etc., handelt es sich um eine absolute Bewertung einer Regel. Dieses Kriterium wird im weitern Verlauf der Arbeit als *absolutes CutOff-Kriterium* bezeichnet. Das zweite CutOff-Kriterium vergleicht die gelernte Regel mit ihrer Vorgängerregel. Ist die Bewertung dieses Vergleichs größer als ein bestimmter Schwellwert, so wird das Kriterium als wahr ausgewertet und bricht das Lernen ab. Da die Bewertung einer Regel bei diesem Kriterium relativ zu ihrer Vorgängerregel gemacht wird, wird dieses Kriterium im weitern Verlauf als *relatives CutOff-Kriterium* bezeichnet. Die letzten zwei Kriterien, die implementiert wurden, sind das Significance Testing und ein auf dem MDL-Prinzip (siehe Kapitel 4.2.1) basierendes Kriterium.

Aufgrund einer Besonderheit des Covering-Algorithmus in Verbindung mit Stopkriterien, die eine Regel als signifikant oder gut bewerten, obwohl die Regel noch mehr negative als positive Beispiele abdeckt, ist es notwendig die Stopkriterien mit einer zusätzlichen Bedingung zu versehen. Diese zusätzliche Bedingung soll garantieren, dass eine Regel immer mehr positive als negative Beispiele abdeckt. Ohne diese zusätzliche Bedingung kann es leicht dazu kommen, dass nur leere Hypothesen gelernt werden, da einige der implementierten Stopkriterien im Konflikt mit dem IncAcc-Kriterium, das im Covering-Algorithmus enthalten ist, stehen.

Das Significance Testing in der vorliegenden Implementierung überprüft in diesem Sinn, mit welcher Sicherheit die Anzahl der positiven und negativen Beispiele, die von einer Regel abdeckt werden, von der Standardverteilung in Richtung des Punktes (0, P) im PN-Raum abweicht.

5.2.4 Konfiguration der Komponenten

Die Konfiguration der Algorithmen wird über einen Optionsstring bewerkstelligt. Die Syntax des Strings und die Methoden, die die Informationen aus dem String verarbeiten, werden von der WEKA-Bibliothek bereitgestellt. Eine Option setzt sich aus einem Optionstag und einem Wert, der für ein Attribut zusammengesetzt werden soll. In Gleichung 5.1 und 5.2 sind zwei Beispieloptionen angeführt.

$$-H \text{ "seco.heuristics.MEstimate -M 0.5" } (5.1)$$

$$-D \quad (5.2)$$

Die Option aus Gleichung 5.1 legt die Suchheuristik, in diesem Falle das m-Estimate, für einen Lernalgorithmus fest. Die nachfolgende Option –M 0.5 konfiguriert das m-Estimate so, dass der Parameter m gleich 0.5 gesetzt wird. Gleichung 5.2 ist ein Beispiel für eine Option, die keinen weiteren Wert mehr benötigt. Ist diese Option vorhanden so wird ein interner Wert auf *true* gesetzt, ansonsten auf *false.*

Die einzelnen Komponenten eines Pruningalgorithmus, also die Stopkriterien und Pruningoperatoren, sind für sich konfigurierbar. Um der Klasse Covering mitzuteilen, dass geprunt werden soll, muss in dem Optionsstring festgelegt werden, um welche Art des Pruning es sich handelt. Dies geschieht durch den Parameter „-PCLASS <PruningClass>“. Alle Optionen, die nach diesem String folgen, konfigurieren die weiteren Komponenten des Pruningalgorithmus. Die Stopkriterien werden über die Parameter „-LC <IStoppingCriterion>“ bzw. „-CC <IRuleStoppingCriterion>“ konfiguriert. Für jedes Stopkriterium kann, sofern dies benötigt wird, ein Schwellwert („-TH <double value>“) und eine Suchheuristik („-SH <HeuristicClass>“) festgelegt werden.

Die Pruningoperatoren für einen Pruningalgorithmus werden mit dem Parameter „-OP <OperatorClass>“ konfiguriert. Bei manchen Operatoren ist es sinnvoll auch eine Heuristik festzulegen, die für das Pruning verwendet wird. Dies wird über die Parameter „-OP <OperatorClass> -SH <HeuristicClass>“ bewerkstelligt.

5.3 Instanziierung des Preprunings

```
class Prepruning extends PruningTemplate{
                                                    ⋮
    public boolean literalStop(CandidateRule, Instances){
        return(<IStoppingCriterion>.literalStop(CandidateRule, Instances));
    }
    public boolean clauseStop(RuleSet, CandidateRule, Instances){
        return(<IRuleStoppingCriterion>.clauseStop(RuleSet, CandidateRule, Instances));
    }
                                                    ⋮
}
```

Abbildung 25: Instanziierte Methoden für das Prepruning

Wie bereits in Kapitel 4.2.1 erwähnt, wird beim Prepruning schon während des Lernens darauf geachtet keine überangepasste Hypothese zu lernen. Dies wird von den Stopkriterien maßgeblich bestimmt. Die Stopkriterien werden entweder direkt in den entsprechenden Methoden implementiert oder die Entscheidung wird zu einem Stopkriterium weitergeleitet. Die Stopkriterien können zwei unterschiedliche sein, z.B. könnte das IStoppingCriterion auf dem CutOff-Kriterium und das IRuleStoppingCriterion könnte auf dem Significance Testing basieren. Eine Aufteilung der Trainingsmenge in Growing- und Pruning-Menge ist nicht nötig. Da das Prunen schon während dem Lernen stattfindet, ist eine Nachbearbeitung von gelernten Regeln und Regelmengen nicht notwendig. Die Methoden *literalStop* und *clauseStop* aus Abbildung 21 werden für das Prepruning also wie in Abbildung 25 überschrieben.

5.4 Instanziierung des IREP

Eine Regel wird von IREP auf einer Growing-Menge gelernt und auf der Pruning-Menge nach dem Lernen wieder vereinfacht. Beide Mengen werden nach jedem Separate-Schritt wieder neu erstellt. Die Anzahl der Beispiele, die in einer der Menge enthalten sind, wird durch die Anzahl der Teile, in die die Trainingsmenge geteilt wird, festgelegt. Sei *fold* die Anzahl der Teile, in die die Trainingsmenge aufgeteilt wird, dann ist die Anzahl der Beispiele in der Growing-Menge wie folgt festgelegt:

$$\frac{folds - 1}{folds} * Anzahl\ der\ Beispiele\ in\ der\ Trainingsmenge \quad (5.3)$$

Die restlichen Beispiele, die nicht in der Growing-Menge sind, bilden die Pruning-Menge. Beide *split*-Methoden werden so überschrieben, dass die Trainingsmenge nach der Vorschrift von Gleichung 5.3 aufgeteilt wird. Die Aufteilung der Beispiele nach Gleichung 5.3 stammt aus der WEKA-Bibliothek [WFTH99] – JRip verwendet die gleiche Aufteilung der Beispiele. Wichtig bei der Aufteilung ist, dass die Beispiele aus der Trainingsmenge zufällig gezogen werden.

Aufgrund der Aufteilung der Trainingsmenge in Growing- und Pruningmenge und den im Covering-Algorithmus bzw. im IREP enthaltenen Pruningmechanismen, lernt IREP häufig leere Hypothesen oder für eine Klasse keine Regeln. Dies geschieht, wenn eine Regel soweit geprunt wird, sodass die Regel keine Attributtest mehr enthält, d.h. die Regel leer ist. Leere Regeln werden nicht in die Hypothese mit aufgenommen. Ein zweiter Grund, warum dies passiert ist, dass das Prunen dazu

führen kann, dass eine Regel mehr negative als positive Beispiele abdeckt. Diese Regeln werden vom *IncAcc* –Kriterium abgelehnt und das Lernen von Regeln für eine Klasse wird unterbrochen. Aus diesem Grund wurde IREP so erweitert, dass das Lernen neu gestartet werden kann. Wird eine leere Regel gelernt oder eine Regel, die keine Beispiele auf der Pruningmenge abdeckt, so kann das Lernen neu gestartet werden. Bei jedem Neustart wird die Trainingsmenge erneut zufällig in Growing- und Pruningmenge aufgeteilt. Dies soll die Wahrscheinlichkeit verringern, dass ein Beispiel sich in der Pruningmenge befindet, obwohl es wichtig ist, um die Zielhypothese zu beschreiben. Die Anzahl der Neustarts kann frei gewählt werden.

Da Regeln, nachdem diese gelernt wurden, geprunt werden, muss auch die Methode *postProcessRule* überschrieben werden. Das Prunen einer Regel wird im Allgemeinen solange durchgeführt, bis keine bessere Regel mehr gefunden werden kann. Eine geprunte Regel ist besser als eine andere Regel, sofern diese auf dem PruningSet eine bessere Bewertung erhält oder, bei gleicher Bewertung, kürzer ist. Bevor eine geprunte Regel zur Hypothese hinzugefügt wird, wird überprüft, ob das Hinzufügen der Regel zur Hypothese, die Hypothese verbessert. Diese Überprüfung wird durch ein Stopkriterium übernommen. Abbildung 26 zeigt wie die Methoden *outerSplit*, *innerSplit*, *postProcessRule* und *clauseStop* aus Abbildung 21 überschrieben werden müssen, um einen Algorithmus zu implementieren, der das Post- und Prepruning integriert.

```
class IREPruning extends PruningTemplate{
                                                    ⋮
    public Instances[] outerSplit(Instances){
        data[0] = draw randomly (folds−1)/folds * size(Instances) examples from Instances
        data[1] = remaining Examples in Instances
        return(data);
    }

    public Instances[] innerSplit(Instances){
        return(outerSplit(Instances));
    }

    public CandidateRule postProcessRule(CandidateRule, Instances){
        best = CandidateRule
        do{
            pruned = RuleOperator.applyOperator(best,Instances)
            if(pruned is not better than best)
                break;
            best = pruned;
        }while(true);
        return(best);
    }

    public boolean clauseStop(RuleSet, CandidateRule, Instances){
        return(<IRuleStoppingCriterion>.clauseStop(RuleSet, CandidateRule, Instances));
    }
                                                    ⋮
}
```

Abbildung 26: Instanziierte Methoden für IREP

5.5 Instanziierung des Postpruning, REP

Im Gegensatz zum IREP wird beim REP die Trainingsmenge nur einmal vor dem Lernen aufgeteilt. Die Aufteilung erfolgt auch hier nach der Vorschrift aus Gleichung 5.3. Auf Stopkriterien sowie auf das Prunen von Regeln während dem Lernen wird hier vollständig verzichtet. Die wichtigste Methode für diesen Algorithmus ist die *postProcess*-Methode. Die Methode sucht solange nach einer Verfeinerung der Hypothese bis die Genauigkeit einer geprunten Hypothese schlechter ist als die momentan beste gefundene Hypothese. Um eine bessere Hypothese zu finden, ist die Methode *findeBestSimplification* verantwortlich. Diese Methode wendet in einem Pruning-Schritt alle Pruningoperatoren einmal auf die Hypothese an und erzeugt eine Menge von möglichen besseren Hypothesen. Eine Hypothese ist besser als eine andere Hypothese, wenn die Genauigkeit der gefundenen Hypothese höher ist oder, bei gleicher Genauigkeit, die Hypothese kürzer ist. D.h. sie enthält weniger Attributtests, weniger Regeln oder beides. Abbildung 27 zeigt, wie die Methoden *outerSplit* und *postProcess* aus Abbildung 21 überschrieben werden müssen, damit man einen Algorithmus erhält, der das REP implementiert.

```
class REPruning extends PruningTemplate{
                                                    ⋮
    public Instances[] outerSplit(Instances){
        data[0] = draw randomly (folds−1)/folds * size(Instances) examples from Instances
        data[1] = remaining Examples in Instances
        return(data);
    }

    public RuleSet postProcess(RuleSet, Instances){
        do{
            best = findBestSimplification(RuleSet, Instances)
            if(accuracy(best, Instances) < accuracy(RuleSet, Instances) || best.equals(RuleSet))
                    break;
            RuleSet = best
         }while(true);
        return(RuleSet);
    }

    protected RuleSet findBestSimplification(RuleSet, Instances){
        curbest = RuleSet
        for each operator p
            possibleTheories = p.applyOperator(RuleSet, Instances);
        for each theory t in possibleTheories
            if(t is better than curbest)
                curbest =  t
        return(curbest);
    }
                                                    ⋮
}
```

Abbildung 27: Instanziierte Methoden für REP

5.6 Instanziierung des IREPOpt

Die Klasse IREPOpt, die einen Algorithmus, der an den RIPPER angelehnt ist, implementiert, ist eine Erweiterung der IREP Klasse. Zusätzlich zu dem eigentlichen IREP Algorithmus ist die Klasse so erweitert worden, dass die gelernte Hypothese nochmals eine Optimierungsphase durchläuft. Das verwendete Stopkriterium, der Pruningoperator und die Pruningheuristik sind in dieser Klasse fest

vorgeschrieben. Der Pruningoperator, der von der Klasse verwendet wird ist der delete-condition-sequence Operator. Die Heuristik, die zum Prunen verwendet wird ist die Heuristik des eigentlichen RIPPER Algorithmus (vgl. Kapitel 4.4.2). Das Stopkriterium ist das MDL-Kriterium aus Kapitel 4.2.1. Die abschließende Optimierungsphase entspricht der Phase aus [Cohe95]. Die Optimierungsphase ist in Abbildung 28 beschrieben und überschreibt die PostProcess-Prozedur aus Abbildung 20. Ansonsten sind die Methoden wie in Abbildung 26 zu sehen überschrieben.

```
class IREPOpt extends PruningTemplate{
                                                    ⋮
    public RuleSet postProcess(RuleSet, Instances){
        RuleSet best;
        for each rule r in RuleSet
            revision = learn new rule for class r.class on Instances
            replacement = refine r on Instances
            newrule = the best rule from r, revision, replacement
            best.add(newrule)
        return(best);
    }

    public CandidateRule postProcessRule(CandidateRule, Instances){
        best = CandidateRule
        do{
            pruned = deleteLastCondition(best)
            if(Evaluation(pruned) < Evaluation(best))
                break;
            best = pruned;
        }while(true);
        return(best);
    }
                                                    ⋮
}
```

Abbildung 28: Instanziierte Methoden für IREPOpt

Kapitel 6 Algorithmen im Vergleich

Dieses Kapitel widmet sich dem Vergleich der verschiedenen Algorithmen und deren Konfigurationen. Als erstes werden dazu die Datensätze beschrieben, auf denen die Algorithmen getestet werden. Im zweiten Teil des Kapitels werden die Algorithmen und deren Konfigurationen vorgestellt. Die Kapitel 6.3 bis 6.6 fassen die Ergebnisse der Vergleiche zusammen. Auf die Algorithmen REP, TDP und IREPOpt wurden in dieser Arbeit verzichtet.

6.1 Beschreibung der Testdatensätze

Um einen möglichst aussagekräftigen Vergleich zwischen den Algorithmen durchführen zu können, wurde eine große Anzahl von Datensätzen gewählt. Insgesamt sind 57 verschiedene Datensätze für den Vergleich aus dem UCI-Repository [AsNe07] gewählt worden. Das Repository ist eine Sammlung von Datensätzen für das Maschinelle Lernen um Lernalgorithmen empirisch zu untersuchen. Die Datensätze werden durch ihren Namen, die Anzahl der Attribute eines Beispiels und die Anzahl der Klassen beschrieben. Die Attribute werden zusätzlich noch in nominale *(s)* und numerische *(n)* unterteilt. |*E*| gibt die Anzahl der Beispiele, |*A*| die Anzahl der Attribute und |*K*| die Anzahl der verschiedenen Klassenwerte an. Die Tabellen 12 und 13 beschreiben die Merkmale der Datensätze.

Nr.	Name des Datensatz	|E|	|A|	|K|
1	anneal	798	32s 6n	6
2	audiology	226	69s	24
3	auto-mpg	398	3s 5n	4
4	autos	205	10s 16n	7
5	balance-scale	625	1s 4n	3
6	balloons	76	5s	2
7	breast-cancer	286	10s	2
8	breast-w	699	1s 9n	2
9	breast-w-d	699	10s	2
10	bridges2	108	11s 1n	6
11	cleveland-heart-disease	303	8s 6n	5
12	colic	368	16s 7n	2
13	colic.ORIG	368	20s 7n	2
14	contact-lenses	24	5s	3
15	credit	490	10s 6n	2
16	credit-a	690	10s 6n	2
17	credit-g	1000	14s 7n	2
18	diabetes	768	1s 8n	2
19	echocardiogram	132	2s 7n	2
20	flag	194	18s 10n	6
21	glass	214	1s 8n	7
22	glass2	163	1s 8n	2
23	hayes-roth	132	5s	3
24	heart-c	303	7s 6n	5
25	heart-h	294	7s 6n	5
26	heart-statlog	270	1s 13n	2

Tabelle 12: Teil 1 der verwendeten Datensätze

Nr.	Datensatz	|E|	|A|	|K|
27	hepatitis	155	13s 6n	2
28	horse-colic	368	15s 7n	2
29	house-votes-84	435	17s	2
30	hypothyroid	3163	19s 7n	2
31	ionosphere	351	1s 34n	2
32	iris	150	1s 4n	3
33	krkp	3196	37s	2
34	labor	57	10s 7n	2
35	labor-d	57	17s	2
36	lymph	148	16s 3n	4
37	lymphography	148	16s 3n	4
38	machine	209	1s 7n	8
39	monk1	124	7s	2
40	monk2	169	7s	2
41	monk3	122	7s	2
42	mushroom	8124	23s	2
43	primary-tumor	339	18s	22
44	promoters	106	58s	2
45	segment	2310	1s 19n	7
46	sick-euthyroid	3163	19s 7n	2
47	solar-flare	333	11s	7
48	sonar	208	1s 60n	2
49	soybean	683	36s	19
50	tic-tac-toe	958	20s	2
51	titanic	2201	4s	2
52	vehicle	846	1s 19n	4
53	vote	435	16s	2
54	vote-1	435	16s	2
55	vowel	990	1s 9n	11
56	wine	178	1s 13n	3
57	zoo	101	17s 1n	7

Tabelle 13: Teil 2 der verwendeten Datensätze

Diese 57 Datensätze bilden die Grund-Datensätze für den Vergleich der Algorithmen. Um zu sehen, wie die verschiedenen Pruningalgorithmen, ein Overfitting an verrauschte Daten vermeiden, sind die 57 Datensätze zusätzlich noch mit Noise versehen worden. Die Datensätze, wie sie im UCI-Repository enthalten sind, sind frei von verrauschten Daten (0% Noise) und werden nochmals mit 5% und mit 10% Rauschen versehen. Die Datensätze werden mit Hilfe der WEKA-Bibliothek mit Rauschen versehen. Ein bestimmter Prozentsatz der Beispiele, hier 5% und 10%, wird durch den *AddNoise*-Filter [WFTH99] verrauscht, indem ein zufälliges Attribut des Beispiels einen zufälligen neuen Wert erhält. Insgesamt sind, mit den 57 Grund-Datensätzen und den zusätzlich mit Noise versehen Datensätze, 171 Datensätze für den Vergleich der Algorithmen vorhanden.

6.2 Verwendete Algorithmen

Als Grundlage für den Vergleich der Pruningalgorithmen sind zwei verschiedene Algorithmen gewählt worden. JRip ist der erste Algorithmus, der für den Vergleich herangezogen worden ist. Wie bereits erwähnt ist JRip eine Java-Implementierung des RIPPER-Algorithmus in der WEKA-Bibliothek.

Konfigurationsname	Beschreibung
JRIP	Der RIPPER-Algorithmus mit Pruningphase und mit Optimierungsphase
JRIP-P	Der RIPPER-Algorithmus ohne Pruningphase, aber mit Optimierungsphase
JRIP-O	Der RIPPER-Algorithmus mit Pruningphase, aber ohne Optimierungsphase
JRIP-OP	Der RIPPER-Algorithmus ohne Pruningphase und ohne Optimierungsphase

Tabelle 14: JRip Konfigurationen

Der zweite Algorithmus, der für den Vergleich benutzt wird ist der Covering-Algorithmus aus dem SeCo-Framework. Dieser Algorithmus wird einmal ohne Pruning auf den Datensätzen getestet. Im weiteren Verlauf wird das Verhalten von Covering getestet, wenn dieser mit der Möglichkeit zum Prepruning erweitert ist, und mit einer Implementierung, die das integrierte Pre- und Postpruning unterstützt. Hier im Besonderen eine Implementierung des IREP, die auf dem Covering Algorithmus basiert. Da für die Datenmengen keine separaten Testmengen vorhanden sind, wird die Genauigkeit der von den Algorithmen gelernten Hypothesen mittels einer 10-fold-Crossvalidation festgestellt. Die Evaluation wird hier von den Evaluationsmechanismen der WEKA-Bibliothek vorgenommen.

Der Covering Algorithmus sucht mittels einer Top-Down Hill-Climbing Suche nach Regeln. Die Suchheuristik, die während der Suche verwendet wird, kann für den Algorithmus frei gewählt werden.

6.2.1 Konfiguration von JRip und Covering

Die Algorithmen JRip und Covering bilden das Grundgerüst des Vergleichs. Die Genauigkeiten, Hypothesengrößen und Zeiten um eine Hypothese zu finden, die von diesen Algorithmen erzeugt werden, dienen dabei als grundlegende Vergleichsparameter.

Das Pruning und die nachfolgende Optimierung der gelernten Hypothese (vgl. Kapitel 4.4.2) kann bei dem Algorithmus JRip unabhängig voneinander ein- bzw. ausgeschaltet werden. Mit diesen Möglichkeiten ergeben sich vier verschiedene Konfigurationen für JRip. In Tabelle 14 sind diese Konfigurationen aufgeführt und kurz beschrieben.

Die Konfigurationen für den Coveringalgorithmus ergeben sich ausschließlich aus den verwendeten Suchheuristiken. Insgesamt werden sieben verschiedene Suchheuristiken für den Coveringalgorithmus verwendet. Die zwei parametrisierbaren Heuristiken, Klösgen-Maß und m-Estimate, werden zusätzlich zu ihren optimalen Parametern noch mit zwei weiteren Parametern evaluiert. Die zusätzlichen Parameter sind gewählt worden, um zu sehen welche Ergebnisse von den parametrisierbaren Heuristiken für nicht optimale Parameter erzielt werden, und ob durch das Pruning die Ergebnisse für parametrisierbare Heuristiken, die eine Trainingsmenge leicht overfitten, besser sind als die Ergebnisse der parameterisierbaren Heuristiken mit optimalem Parameter. Damit gibt es 11 verschiedene Konfigurationen des Coveringalgorithmus, die in Tabelle 15 beschrieben sind.

6.2.2 Konfiguration der Prepruningalgorithmen

Die grundlegende Konfiguration der Pruningalgorithmen ist durch den Covering Algorithmus gegeben. Die Hypothesen werden mit den gleichen Suchheuristiken gelernt, die der Covering-Algorithmus verwendet. Der Unterschied hier ist, dass das Lernen der Hypothesen durch Stopkriterien unterstützt wird. Die Stopkriterien, durch die das Lernen unterstützt wird, sind beim Prepruning das

Significance Testing (vgl. Kapitel 4.2.1), das absolute CutOff-Kriterium (vgl. Kapitel 4.2.1) und ein Kriterium[7], das auf dem MDL-Prinzip basiert (vgl. Kapitel 4.2.1). Die verschiedenen Suchheuristiken werden jeweils mit einem dieser Kriterien getestet. Des Weiteren wird das Significance Testing mit drei Sicherheitswahrscheinlichkeiten getestet. Die Wahrscheinlichkeiten, mit denen das Significance Testing evaluiert wird, sind 90%, 95% und 99%.

Konfigurationsname	**Beschreibung**
Laplace	Covering mit Suchheuristik Laplace
Precision	Covering mit Suchheuristik Precision
Accuracy	Covering mit Suchheuristik Accuracy
WRAcc	Covering mit Suchheuristik Weighted Relative Accuracy
Klösgen	Covering mit Suchheuristik Klösgen-Maß. Der Parameter ω hat den Wert 0.4323 (optimaler Wert, siehe [JaFü06])
Kloesgen0.3	Covering mit Suchheuristik Klösgen-Maß. Der Parameter ω hat den Wert 0.3.
Kloesgen0.2	Covering mit Suchheuristik Klösgen-Maß. Der Parameter ω hat den Wert 0.2.
MEstimate	Covering mit Suchheuristik m-Estimate. Der Parameter m hat den Wert 22.466 (optimaler Wert, siehe[JaFü06]).
MEstimate0.5	Covering mit Suchheuristik m-Estimate. Der Parameter m hat den Wert 0.5.
MEstimate13.97	Covering mit Suchheuristik m-Estimate. Der Parameter m hat den Wert 13.97.
Correlation	Covering mit Suchheuristik Correlation.

Tabelle 15: Covering Konfigurationen

Das CutOff-Kriterium benutzt für die Entscheidung das Lernen zu unterbrechen die gleiche Heuristik, die zum Lernen verwendet wird. Zusätzlich dazu werden für das CutOff-Kriterium drei verschiedene Schwellwerte getestet. Die Cutoff - Parameter, die hier benutzt werden sind 0.3, 0.6 und 0.9. Diese Werte ergeben sich dadurch, dass die verwendeten Heuristiken Regeln im Intervall [0,1] bewerten. Der Wert 0.3 wurde gewählt, da er für das CutOff-Kriterium in Verbindung mit Correlation ein guter Wert ist [Fürn94b], und um zu sehen, ob dieser Wert auch für andere Heuristiken gut geeignet ist. Die Werte 0.6 und 0.9 wurden gewählt, um noch zwei weitere Werte innerhalb des Intervalls [0,1] zu haben und das Verhalten des CutOff-Kriteriums für größere Parameter zu untersuchen. Die Vermutung ist, dass das Verhalten gleich oder ähnlich zu dem Verhalten des CutOff-Kriteriums in Verbindung mit Correlation ist. D.h., dass für niedrige Cutoff - Parameter der Covering-Algorithmus dazu neigt, eine überangepasste Hypothese zu lernen, wohingegen für hohe Cutoff - Parameter eine leere Hypothese gelernt wird. Für das Prepruning ergeben sich somit 77 verschiedene Konfigurationen, 11 für das MDL-Kriterium, sowie jeweils 33 für das CutOff-Kriterium und Significance Testing.

	Genauigkeiten in %		
Datensatz	*CutOff mit Test fp<tp*	*CutOff ohne Test fp<tp*	*Differenz*
contact-lenses	79,17	58,33	20,48
monk1	93,55	86,30	9,03
krkp	98,75	92,68	6,07
iris	91,33	90,67	0,66
echocardiogram	71,62	74,32	-2,70

Tabelle 16: Genauigkeiten Prepruning mit CutOff Kriterium. Mit und ohne Test auf fp<tp.

[7] Hier kurz MDL-Kriterium genannt.

Das CutOff-Kriterium, das hier verwendet wird überprüft zwei Eigenschaften einer Regel. Die erste Eigenschaft, die geprüft wird, stellt sicher, dass die *truepositives* einer Regel größer sind als die *falsepositives*. Ohne das Überprüfen dieser Eigenschaft, tendiert das Lernen mit diesem Kriterium dazu, Hypothesen zu lernen, die eine geringere Genauigkeit besitzen, wenn nicht sogar eine leere Hypothese gelernt wird. Dies geschieht, da manche Heuristiken Regeln mit einem Wert größer dem CutOff-Parameter bewerten, wenn genügend positive Beispiele im Verhältnis zu den negativen Beispielen abgedeckt sind. Gilt für solche Regeln, dass *fp>tp*, so wird nicht nur die Verfeinerung unterbrochen, sondern auch das Regellernen für eine Klasse bedingt durch das *IncAcc*-Kriterium. Als Beispiel soll hier die Heuristik *Laplace* in Verbindung mit dem CutOff-Parameter 0.3 und der Regel *R*, die vier positive und sieben negative Beispiele abdeckt. Die Bewertung der Regel hätte dann einen Wert $h_{lap}(4,7) = \frac{4+1}{4+7+2} = 0.38$, womit das Lernen der Regel abgebrochen wird und die Regel vom Covering-Algorithmus abgelehnt würde. Ein ähnliches Verhalten ist auch beim Significance Testing zu sehen, weshalb auch dieses Kriterium die zusätzliche Bedingung enthält. Um den Effekt der zusätzlichen Bedingung in den Stopkriterien zu verdeutlichen, sind für das CutOff-Kriterium beispielhaft einige Ergebnisse in Tabelle 16 eingetragen. Einmal mit dem CutOff-Kriterium ohne zusätzliche Überprüfung und mit zusätzlicher Überprüfung. Gelernt wurde auf den Datensätzen mit der Heuristik *m-Estimate* und einem CutOff-Parameter von 0.3.

Die zweite Eigenschaft, die geprüft wird, ist der eigentliche CutOff, der auf einer absoluten Bewertung der Regel basiert, wie in Kapitel 5.2.3 beschrieben. Der relative CutOff wird hier nicht betrachtet, da für die meisten verwendeten Heuristiken und gelernten Hypothesen gleiche Genauigkeiten beim Lernen zu erwarten sind oder die Differenz minimal abweicht.

Im nachfolgenden wird die Bezeichnung <Heuristik>-<Stopkriterium><Parameter> verwendet, um eine Prepruningkonfiguration zu beschreiben. Die Bezeichnung <Heuristik> orientiert sich an den Bezeichnungen aus Tabelle 15. Die Stopkriterien erhalten folgende Abkürzungen. Mit „MDL“ ist das MDL-Kriterium, mit „Sig“ das Significance Testing und mit „Cut“ das CutOff-Kriterium gemeint. Die Bezeichnung <Parameter> kann die Werte 0,3, 0,6 und 0,9 für das CutOff-Kriterium und 90, 95 bzw. 99 für das Significance Testing annehmen. Die Bezeichnung Correlation-MDL bezieht sich dann auf die Prepruningkonfiguration, die mit der Heuristik Correlation lernt und das MDL-Kriterium als Stopkriterium verwendet, wohingegen die Bezeichnung Laplace-Sig90 die Prepruningkonfiguration, die mit Laplace lernt und das Significance Testing mit einer Sicherheit von 90% bezeichnet. In den nachfolgenden Tabellen können diese Bezeichnungen aus Platzgründen nicht mitgeführt werden.

6.2.3 Konfiguration der IREP Algorithmen

Den größten Teil des Vergleichs machen die Konfigurationen des IREP-Algorithmus aus. Für den Algorithmus werden vier verschiedene Stopkriterien getestet. Das erste Kriterium, mit dem der Algorithmus getestet wird, wird als *MaximumErrorRate* bezeichnet. Dieses Kriterium ist das im RIPPER verwendete Kriterium. Es stoppt das Regellernen, sobald der Fehler der besten gefundenen Regel auf der Pruningmenge 50% übersteigt. Die anderen drei Kriterien sind dieselben wie beim Prepruning. Diese Kriterien werden auch wie beim Prepruning mit drei verschiedenen Schwellwerten evaluiert (vgl. Kapitel 6.2.2).

Der IREP Algorithmus wird zusätzlich zu den Stopkriterien noch mit zwei Pruningoperatoren getestet. Die Pruningoperatoren die hier verwendet werden sind *delete-last-condition* und *find-best-simplification*. Der *find-best-simplification*-Operator wird hier mit derselben Heuristik konfiguriert mit der gelernt wird. Mit den drei Stopkriterien, den zwei Pruningoperatoren und den elf Heuristiken ergeben sich für den IREP-Algorithmus insgesamt 154 Konfigurationen.

Eine wichtige Konfiguration kann hier aufgrund der Eigenschaften des SeCo-Frameworks nicht getestet werden. Die IREP-Konfiguration mit dem MDL-Kriterium, der Suchheuristik FOILGain und der Pruningheuristik des Ripper (vgl. Kapitel 4.4.2 Gleichung 4.9). Dem aufmerksamen Leser wird nicht entgangen sein, dass der Algorithmus in dieser Konfiguration dem RIPPER ohne Optimierungsphase ähnelt. Der Grund warum diese Konfiguration nicht möglich ist, ist die *Gain-Heuristik*[8] FOILGain (vgl. Kapitel 3.3.2). Das SeCo-Framework ist für *Value-Heuristiken*[9] optimiert. Die Hypothesen, die speziell mit FOILGain gelernt werden, sind ausschließlich leere Hypothesen. Dies liegt maßgeblich an dem *ForwardPruning* des Suchalgorithmus und an dem *IncAcc*-Kriterium. Für andere *Gain-Heuristiken* ist ein ähnliches Verhalten zu erwarten.

Die Bezeichnungen der IREP-Konfigurationen ähneln denen der Prepruningkonfigurationen. Die generische Bezeichnung ist <Heuristik>-<Operator>-<Stopkriterium><Parameter>. Die Pruningoperatoren erhalten die Abkürzungen „FIND“ für den Operator *find-best-simplification* und „DEL“ für den Operator *delete-last-condition*. Das vierte Stopkriterium, das für IREP evaluiert wird, wird mit „Max“ bezeichnet und steht für *MaximumErrorRate*. Die Bezeichnung WRA-DEL-Cut0,3 bezieht sich auf die IREP-Konfiguration, die mit WRA lernt, den *delete-last-condition* Operator zum Prunen verwendet und als Stopkriterium das CutOff-Kriterium mit dem Schwellwert 0,3 benutzt. In den nachfolgenden Tabellen können diese Bezeichnungen aus Platzgründen nicht mitgeführt werden.

6.3 Evaluation der Algorithmen

Die insgesamt 268 Konfigurationen werden nach den Kriterien aus Kapitel 2.5 ausgewertet. Durch die Masse an Daten[10] werden nur interessante Aspekte exemplarisch aufgeführt und evaluiert. Win-Tie-Loss Tabellen (WTL-Tabellen) werden in den nachfolgenden Kapiteln nur eine bzw. gar keine aufgeführt[11]. An dieser Stelle sollte noch gesagt werden, dass die Auswertungen anhand der WTL-Tabellen eher als qualitative Bewertung anstatt einer quantitativen Bewertung gesehen werden sollte, da der Vorzeichentest nur aussagekräftig ist, wenn man die Algorithmen paarweise betrachtet. Der Vergleich der Algorithmen wird anhand ihres Zeitverhaltens (angegeben in Sekunden), der Größe der gelernten Hypothese und der erzielten Genauigkeit vorgenommen. Die Größe der Hypothese wird in der durchschnittlichen Anzahl von gelernten Regeln angegeben (kurz als |R| bezeichnet). Zu Beginn werden die Konfigurationen separat für jedes Datensatzpaket evaluiert und abschließend auf allen Datensatzpaketen die Ergebnisse erneut betrachtet und zusammengefasst. Ein Datensatzpaket besteht aus den 57 Datensätzen, die in Kapitel 6.1 beschrieben sind. Der Unterschied zwischen den Datensatzpaketen ist der Prozentsatz mit wie viel verrauschten Daten die 57 Datensätze aus einem Paket versehen sind. Im weiteren Verlauf bezeichnet *n%-Paket*, ein Datenpaket dessen Datensätze mit n% Noise versehen ist. Besonderes Augenmerk liegt auf den Fragen:

1. Welche Genauigkeiten erzielen die Algorithmen? Wie groß ist die gelernte Hypothese? Wie lange braucht ein Algorithmus um die Hypothese zu lernen?
2. Welche Heuristiken erzielen die besten Ergebnisse? In welcher Konfiguration werden von den Heuristiken die besten Ergebnisse erzielt?

[8] Von Prof. Fürnkranz eingeführter Begriff. Eine *Gain-Heuristik* bewertet eine Regel im Verhältnis zu ihrer Vorgängerregel, d.h. wie gut die Verfeinerung einer Regel ist. Im Gegensatz dazu stehen die Value-Heuristiken, die die Bewertung einer Regel nur auf diese Regel stützen, d.h. wie gut die aktuelle Regel ist.
[9] Siehe Fußnote 7.
[10] Um dies zu verdeutlichen hier eine Rechnung. Die Tabelle mit allen Daten ist 268 (Konfigurationen) *4 (Meßwerte) *171 (Datensätze) = 183312 Zellen groß. Wohingegen die Win-Tie-Loss Tabelle nur knapp 72000 Zellen umfasst.
[11] Die WTL-Tabellen befinden sich auf der beiliegenden CD, da alle Tabellen bzw. die gesamte WTL-Tabelle 574 Seiten groß ist. Verweise auf die Tabellen sind im WTL-Verzeichnis im Anhang A zu finden.

3. Wie verändern sich die Ergebnisse des Covering-Algorithmus ohne Pruning, wenn dieser um Pruningfunktionalität erweitert ist? Welche Stopkriterien und welche Parameter für die Kriterien eignen sich besonders?
4. Wie gut sind die Pruningalgorithmen, die in dieser Arbeit implementiert wurden, im Vergleich zum Covering-Algorithmus ohne Pruning, dem Ripper und untereinander?

Am Ende des Kapitels wird anhand der Ergebnisse versucht eine Antwort auf folgende Frage zu geben:

„Ist Pruning überhaupt noch notwendig, wenn die Heuristiken, die zum Lernen verwendet werden, schon sehr gute Ergebnisse erzielen?“

6.3.1 Ergebnisse JRIP vs. Covering

JRip und Covering									
	Zeitverhalten in sec			***Hypothesengröße in |R|***			***Genauigkeit in %***		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
JRip	0,20	0,20	0,20	7,30	6,84	6,51	81,60	77,34	73,12
JRip-P	0,09	0,25	0,27	14,70	22,25	22,86	81,06	75,12	69,66
JRip-O	0,05	0,05	0,06	8,68	8,79	9,44	80,86	76,03	70,98
JRip-OP	0,09	0,23	0,25	14,70	22,25	22,86	81,06	75,12	69,66
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97

Tabelle 17: Durchschnittliches Zeitverhalten, durchschnittliche Hypothesengröße und Genauigkeit von JRip und Covering

Für die Vergleiche in diesem Kapitel ist es wichtig zu sagen, dass der Covering-Algorithmus hier gänzlich ohne Pruning getestet wurde. Mit diesem Wissen sind auch die Ergebnisse bei diesen Testläufen zu betrachten. Die Hypothese, die der Algorithmus lernt, wird eine Trainingsmenge garantiert in einem gewissen Maß overfitten. Somit ist das Zeitverhalten, die Größe der Hypothese und die Genauigkeit bei Heuristiken die stark verallgemeinern entsprechend hoch bzw. niedrig. Der Algorithmus versucht jedes positive Beispiel einer Klasse in der Trainingsmenge abzudecken und hat bis auf die doch recht schwachen Pruningkriterien *IncAcc*, *ForwardPruning* bzw. die verwendete Suchheuristik keine Möglichkeit ein Overfitting an verrauschte Daten zu vermeiden.

Das Zeitverhalten der Algorithmen in Sekunden ist für die verschiedenen Datenpakete in Tabelle 17 und nochmals in Tabelle B-1 (siehe Anhang B) zu sehen. Wie man hier leicht erkennt ist der Algorithmus Ripper wesentlich schneller als der Covering-Algorithmus. Die meiste Arbeit während dem Lernen einer Hypothese liegt beim Ripper in der abschließenden Optimierungsphase. Sehr auffällig beim Ripper ist, dass die Konfiguration JRip-P und die Konfiguration JRip-OP (vgl. Tabelle 17 – blau hinterlegt) die gleichen Ergebnisse auf allen Datensätzen liefern. Der einzige minimale

Unterschied, der hier festzustellen ist, liegt im Zeitverhalten. Die Konfiguration ohne Pruning ist etwas schneller als die Konfiguration mit Pruning. Auch wenn man die Genauigkeiten und die Größe der Hypothesen betrachtet unterscheiden sich beide Konfigurationen nicht. Dies wird besonders deutlich an der Tatsache, dass die Win-Tie-Loss Ergebnisse [RIP-COV][12] der Algorithmen mit 0 Wins, 171 Ties und 0 Losses gegeben ist. Diese Ergebnisse legen nahe, dass die Optimierungsphase des Ripper nur sinnvoll ist, wenn diese durch ein Pruning unterstützt wird, da durch das Pruning bessere Startpunkte für die abschließende Optimierungsphase gefunden werden. Anders Ausgedrückt heißt es auch, dass es ohne Pruning egal ist, ob die gelernte Regelmenge nochmals optimiert wird oder nicht.

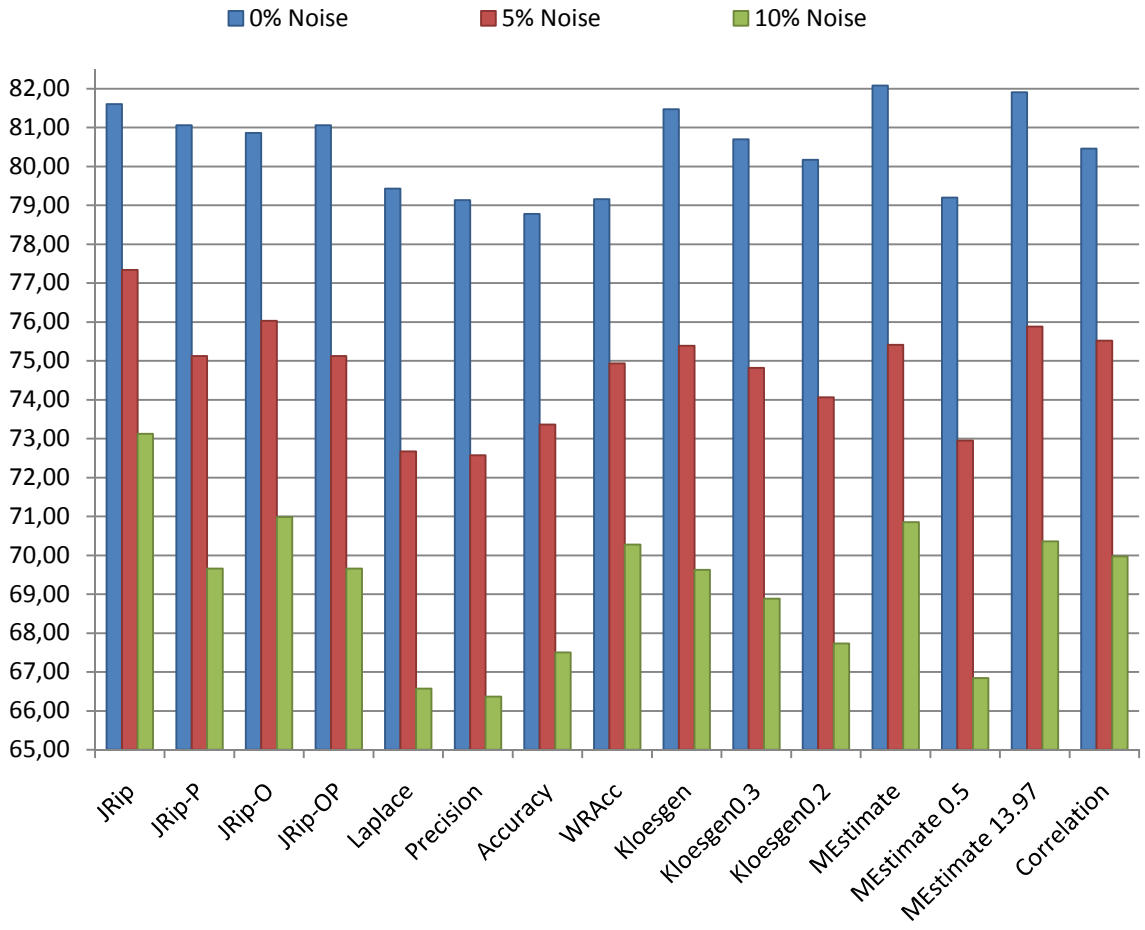

Abbildung 29: Durchschnittliche Genauigkeit der Covering- und JRip-Konfigurationen auf den Datenpaketen

In Abbildung 29 ist die Tabelle 17 für die durchschnittlichen Genauigkeiten nochmals visualisiert. In dem Diagramm sind auf der x-Achse die jeweiligen Covering- bzw. JRip-Konfigurationen aufgetragen. Auf der y-Achse erkennt man die durchschnittliche Genauigkeit, die von den Konfigurationen, jeweils auf dem 0%-, 5%- bzw. 10%-Paket, erzielt wurden. Auch hier erkennt man nochmals, dass die Ergebnisse der Konfigurationen JRip-P und JRip-OP gleich sind. Anhand der Abbildung lässt sich feststellen, was ein Rauschen innerhalb der Trainingsdaten bewirkt. Je verrauschter die Daten sind, desto geringer wird auch die Genauigkeit, die ein Lernalgorithmus erzielt. Besonders deutlich wird das, wenn man die Covering Konfigurationen betrachtet, die mit Heuristiken lernen, die zum Overfitting neigen, wie zum Beispiel die Konfigurationen Laplace, Precision oder MEstimate0.5. Die Eigenschaft der Heuristik sehr spezielle Hypothesen zu lernen, liefert auch den

[12] Diese Tabellen sind auf der beiliegenden CD zu finden. Die Bezeichnungen sind im Anhang A aufgelistet einschließlich der Tabellen, auf die sie sich beziehen.

Grund für das Verhalten, da der Lernalgorithmus mit diesen Heuristiken auch versucht, die verrauschten Daten zu beschreiben. Betrachtet man das Verhalten der Konfigurationen, die mit Heuristiken lernen, die eher allgemeine Hypothesen lernen, wie z.B. *WRA*, das *m-Estimate* oder *Correlation*, erkennt man das Absinken der Genauigkeiten immer noch, aber nicht mehr in diesem Maße, wie bei den Heuristiken, die stark overfitten.

Der Algorithmus Ripper steht im Vergleich auf allen Datensätzen zum Covering ohne Pruning außerhalb jeder Konkurrenz. Die Konfiguration JRip ist mit 99%-iger besser als der Covering-Algorithmus mit den gewählten Heuristiken. Die einzige Konfiguration, bei der man das nicht sagen kann ist MEstimate13.97. Auf dem 0%-Paket erzielen einige Konfigurationen (MEstimate, Kloesgen) bessere durchschnittliche Ergebnisse als die Konfiguration JRip. Ein Blick auf die WTL-Tabelle [RIP-COV] zeigt aber nur, dass man nicht mit Sicherheit sagen kann welche der Konfigurationen die bessere ist. Das volle Potential entfaltet der Ripper-Algorithmus je verrauschter die Daten werden. Auf dem 10%-Paket ist die Konfiguration JRip mit 99%-iger Sicherheit besser als die Covering-Konfigurationen.

Im Gegensatz zum Ripper wird das Zeitverhalten von Covering maßgeblich durch die verwendete Heuristik bestimmt. Heuristiken, die dazu neigen eine überangepasste Hypothese zu lernen, verlangsamen den Algorithmus. Im Gegensatz dazu werden Hypothesen, die mit Heuristiken gelernt werden, die stark verallgemeinern, schneller gefunden. Dieses Verhalten wird umso deutlicher je verrauschter die Daten sind. Gleiches gilt für die Größe der gelernten Regelmenge und die Genauigkeiten. Die Regelmengen sind größer je mehr die Isometriken einer Heuristik den Isometriken des Precision-Modells gleichen. Dies sind auch die Heuristiken, die zum Overfitting neigen. Erstaunlicherweise sind die Größen der Regelmengen bei der Heuristik *WRA* nahezu konstant. Die Ergebnisse, die hier für *WRA* gefunden wurden, sind sehr nahe an den Ergebnissen, die in [ToFL00] gefunden wurden. Insgesamt wurden in [ToFL00] 21 Datensätze, von denen 8 hier nicht verwendet wurde, für die Auswertung benutzt. Die besten Ergebnisse werden durch die parametrisierbaren Heuristiken *Klösgen-Maß* und *m-Estimate* erzielt, insbesondere aber in Kombination mit deren optimalen Parametern, die in [JaFü06] gefunden worden sind. Diese Ergebnisse deuten darauf hin, dass es sich bei den gefundenen optimalen Parametern auch wirklich um gute Parameter handelt.

Die Konfiguration, deren Ergebnisse auf allen Datensätzen am besten sind, ist MEstimate. Die Konfiguration ist im Vergleich mit 99%-iger Sicherheit besser als die restlichen Covering-Konfigurationen. Die einzigen Ausnahmen bilden die Konfigurationen Kloesgen, WRA und das MEstimate13.97. Durch die WTL-Tabelle sieht man, dass die Konfiguration MEstimate mit 95%-iger Sicherheit besser als Kloesgen ist. Gegen die anderen beiden Konfigurationen sind die Gewinne der Konfiguration MEstimate größer als die Verluste (siehe auch [RIP-COV]).

6.3.2 Ergebnisse Prepruning

Anhand der Tabelle B-1 bis B-4 (siehe Anhang B) erkennt man deutlich den Effekt, den das Prepruning auf das Lernen einer Hypothese hat. Besonders sticht hier ins Auge, dass die Lerngeschwindigkeit auf allen Datenpaketen im Schnitt drastisch erhöht hat und die Hypothesengröße verringert wurde. Betrachtet man jedoch die durchschnittlichen Genauigkeiten auf dem 0%-Paket, erkennt man schnell, dass das Pruning auf nicht verrauschten Daten nur selten etwas oder gar nichts gebracht hat, d.h. dass die durchschnittlichen Genauigkeiten nicht oder nur sehr wenig gestiegen sind. Um dieses Ergebnis zu verdeutlichen dient die Abbildung 30. Hier sind die durchschnittlichen Genauigkeiten der Covering und Prepruning Konfigurationen visualisiert.

Anhand der Abbildung 30 lässt sich schnell sehen, dass das Prepruning bei Konfigurationen, die überspezialisierende Heuristiken verwenden, eher zu einer Verringerung der Genauigkeiten führt. Im

Grunde genommen verringert sich der Grad der Überanpassung der gelernten Hypothesen. Auch bei Heuristiken, die stark verallgemeinern, wie *WRA*, werden keine Verbesserungen erzielt. Die Konfigurationen, die die Heuristik *Klösgen-Maß* und das Stopkriterium Significance Testing verwenden, können neben den Konfigurationen Correlation-MDL und MEstimate0.5-Sig95 auf dem 0%-Paket die durchschnittliche Genauigkeit im Vergleich zum Covering erhöhen.

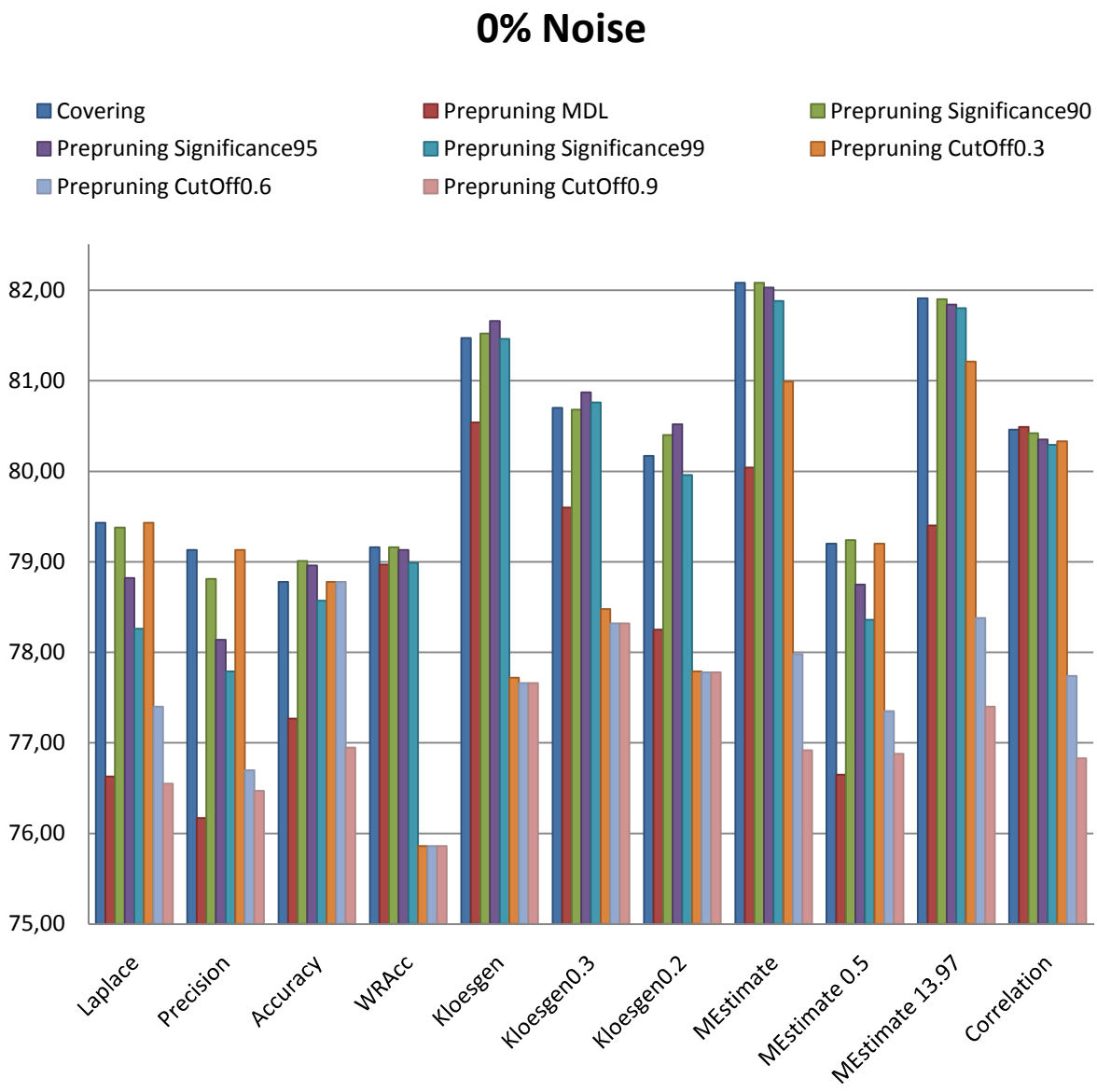

Abbildung 30: durchschnittliche Genauigkeiten des Preprunings auf dem 0% Paket

Betrachtet man die Konfigurationen, die als Stopkriterium das MDL-Kriterium verwenden, so erkennt man, dass die Hypothesengröße auf verrauschten Daten erst anwächst und dann wieder abfällt (vgl. Anhang B Tabelle B-1). Das ist auf die Eigenschaft des MDL-Kriteriums zurückzuführen, die versucht Regeln zu finden, die die Datenmenge komprimieren. Je verrauschter die Datenmenge ist, desto weniger Regeln können gefunden werden, die die Datenmenge auch wirklich komprimieren. Also Regeln die viele Beispiele abdecken. Die Regeln, die nur kleine Teile der Trainingsmenge beschreiben, werden abgelehnt und nicht in die Regelmenge aufgenommen. Da das MDL-Kriterium als *ClauseStoppingCriterion* hier nicht betrachtet worden ist, bedeutet das, dass die gelernten Regeln vom *IncAcc*-Kriterium im Covering-Algorithmus abgelehnt werden. Es werden also mehr Regeln

gelernt, deren *fp*-Anzahl größer als deren *tp*-Anzahl ist bzw. es werden mehr leere Regeln, also Regeln deren Körper den Wert *true* enthält, gelernt.

Ein Blick auf die Tabellen B-3 und B-4 zeigt, wie sich das Significance Testing mit zunehmendem Signifikanzlevel verhält. Im Wesentlichen sinkt die benötigte Zeit zum Lernen und die Größe der Hypothesen nimmt ab. Nichtsdestoweniger werden die durchschnittlichen Genauigkeiten mit dem Significance Testing auf dem 0%-Paket nur für die Heuristiken *Klößgen-Maß* und *m-Estimate* mit Parameter 0.5 erhöht (vgl. Abbildung 30).

Im Vergleich zu den beiden vorherigen Stopkriterien ist das Verhalten des CutOff-Kriteriums ähnlich. Ein gravierender Unterschied lässt sich allerdings in den durchschnittlichen Genauigkeiten feststellen. Auf nicht verrauschten Daten, also auf dem 0%-Paket, sind die gelernten Hypothesen maximal genauso gut, wie der Covering-Algorithmus ohne Pruning. Die durchschnittlichen Genauigkeiten pro Heuristik, die mit einem höheren CutOff-Parameter auf dem 0%-Paket gelernt werden, sind zwischen 0% und 5,5% niedriger als die Genauigkeiten des Covering-Algorithmus, wobei die Genauigkeiten mit zunehmenden CutOff-Parameter absinken.

Aus den Tabellen B-2 und B-3 (siehe Anhang B) kann man erkennen, dass die Hypothesengröße mit steigendem CutOff-Parameter abnimmt. Dies deutet daraufhin, dass für einen CutOff-Parameter von 0 vollständige und konsistente Hypothesen gelernt werden, wohingegen für einen maximalen[13] CutOff-Parameter leere Hypothesen gelernt werden. Direkt nachvollziehen kann man das Verhalten wegen der zusätzlichen Bedingung im CutOff Kriterium nicht, da auch für einen maximalen CutOff-Wert, wegen den zusätzlichen Bedingungen in den Kriterien, nicht leere Hypothesen gelernt werden.

Besonders interessant ist die Tatsache, dass sich die gewählten CutOff-Parameter für *WRA* und das *Klösgen-Maß* nicht eigenen, wie man in Abbildung 30 und [TAB-VS] erkennt. Für die Konfigurationen, die mit *WRA* lernen, werden, egal für welchen CutOff-Parameter, immer die selben Hypothesen gelernt, da das Kriterium bei *WRA* nur sicherstellt, dass eine Regel mehr *tp* als fp abdeckt, die Bewertung einer Regel im Allgemeinen aber niedriger als 0,3 - bzw. 0,6 oder 0,9 – ausfällt, und das Lernen somit abgebrochen wird, sobald gilt *tp* > *fp*. Ähnliches gilt für die Konfigurationen mit dem *Klösgen-Maß*. Die CutOff-Parameter 0.6 bzw. 0.9 eignen sich hier nicht als Parameter, da die gleichen Hypothesen auf den Datensätzen gelernt werden (vgl. Abbildung 30 bis 32, [TAB-VS]).

Interessant ist auch die Tatsache, dass die Konfigurationen mit Precision-Cut0.3, Laplace-Cut0.3, Mestimate0.5-Cut0.3 und Accuracy-Cut0.3 dieselben Ergebnisse erzielen, wie die Covering Konfigurationen Precision, Laplace und Accuracy. Gleiches gilt auch für das 5%- und 10%-Paket (vgl. Abbildung 30 bis 32). Durch die zusätzliche Bedingung, dass bei jeder Regel *tp>fp* gelten muss, werden minimale Heuristikwerte für die drei Heuristiken festgelegt. Seien nun die Gleichungen der Heuristiken wie in Kapitel 3 und die Bedingung $p = n + 1$, was gleichbedeutend damit ist, dass eine Regel genau ein positives Beispiel mehr als negative Beispiele abdeckt. Setzt man die Bedingung in die Gleichungen der Heuristiken ein, verändern sich die Heuristiken wie folgt:

- $Precision: h_{pr}(n,p) = \frac{n+1}{2*n+1}$
- $Laplace: h_{lap}(n,p) = \frac{n+2}{2*n+3}$
- $Accuracy: h_{acc}(n,p) = \frac{N+1}{N+P}$

[13] „maximal“ bedeutet hier, dass der maximale Wert einer Heuristik benutzt wird, z.B. 1 für Laplace, Precision oder Correlation.Für andere Heuristiken müsste der maximale Wert während der Auswertung jedesmal neu berechnet werden.

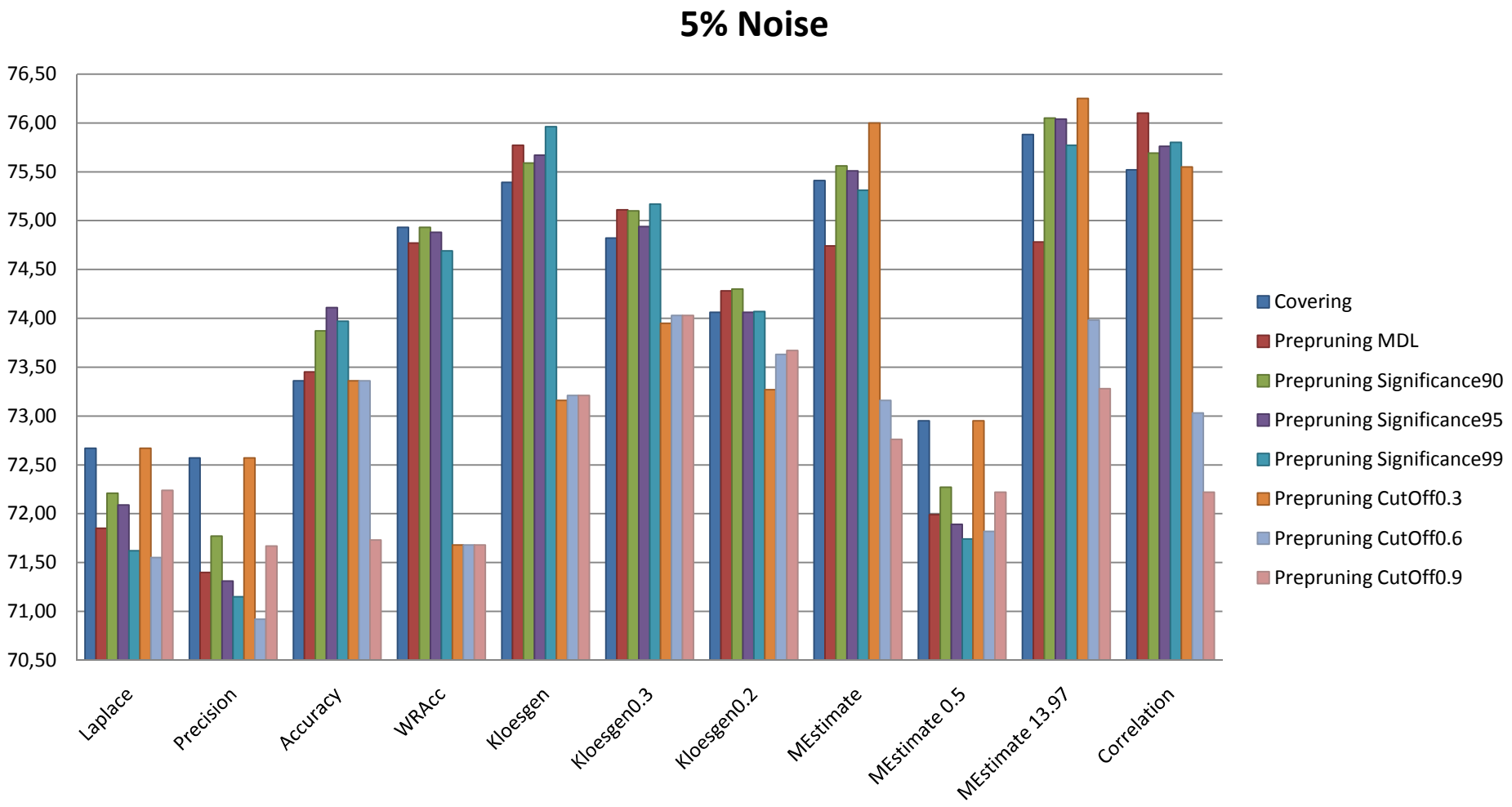

Abbildung 31: durchschnittliche Genauigkeiten des Preprunings auf dem 5% Paket

Nimmt man nun an, dass eine Regel unendlich viele negative Beispiele abdeckt, dann sind die minimalen Heuristikwerte für *Precision* und *Laplace* jeweils 0,5. Für *Accuracy* nimmt man an, dass die Trainingsmenge *N+1* positive Beispiele enthält, und erhält als minimalen Wert, wenn eine Regel alle negativen Beispiele abdeckt, auch 0,5. Das CutOff-Kriterium mit dem Parameter 0.3 wird in diesen Konfigurationen also niemals als wahr ausgewertet und es werden somit die gleichen Hypothesen, wie bei den Covering-Konfigurationen ohne Pruning gelernt.

Da das Prepruning auf dem 0%-Paket keine wirkliche Verbesserung gegenüber den Covering-Algorithmus erzielen konnte außer die Verringerung der Laufzeit und der Hypothesengröße, stellt sich nun die Frage, ob das Prepruning auf verrauschten Daten die durchschnittliche Genauigkeit noch erhöhen kann. Um diese Frage zu beantworten dient die Abbildung 31. Die Abbildung visualisiert die durchschnittlichen Genauigkeiten der Prepruning- und Covering-Konfigurationen auf dem 5%-Paket.

Hier erkennt man schon eine Tendenz, die auf dem 10%-Paket erst richtig deutlich wird. Anhand der Abbildung 31 sieht man, dass die durchschnittlichen Genauigkeiten durchaus erhöht werden können, dennoch ist das Prepruning in Verbindung mit Heuristiken, die stark verallgemeinern, wie *WRA*, nicht in der Lage die Genauigkeiten zu erhöhen. Auch bei den Heuristiken, die stark zum Overfitting neigen kann das Prepruning die Genauigkeiten auf dem 5%-Paket nicht erhöhen. Nichtsdestoweniger zeigt sich, dass, je nachdem welches Stopkriterium verwendet wurde, die Genauigkeiten bei den Konfigurationen Accuracy, Kloesgen, Kloesgen0.2, Kloesgen0.3, MEstimate, MEstimate13.97 und Correlation erhöht werden können. Betrachtet man in Abbildung 31 die durchschnittliche Genauigkeiten der Konfiguration MEstimate und MEstimate13.97, sieht man, dass das *m-Estimate* mit einem suboptimalen Parameter bessere Ergebnisse erzielt als das *m-Estimate* mit optimalem Parameter. Durch den suboptimalen Parameter der parametrisierbaren Heuristik overfittet der Lernalgorithmus die Trainingsdaten etwas, was zur Folge hat, dass die Entscheidung des Prunings, das Lernen zu unterbrechen, an anderen Punkten im Suchraum stattfindet. Das Verhalten des Preprunings mit CutOff-Kriterium ist hier ähnlich zu dem Verhalten auf dem 0%-Paket, wie man anhand der Abbildung 31 erkennt.

Betrachtet man die Abbildung 32, die die durchschnittlichen Genauigkeiten auf dem 10%-Paket visualisiert, sieht man sofort, dass das Prepruning, die durchschnittlichen Genauigkeiten fast jeder Konfiguration erhöht hat. Hier zeigt sich auch, dass das Prepruning bei den Konfigurationen, die stark overfitten, noch zu einer Erhöhung der durchschnittlichen Genauigkeit führen. Besonders durch das MDL-Kriterium und das CutOff-Kriterium können die Genauigkeiten der Konfigurationen erhöht werden. Dennoch kann man für Heuristiken, die sehr stark verallgemeinern, sagen, dass das Prepruning keine Veränderung mehr erzielen kann. Auch die durchschnittlichen Genauigkeiten der Konfigurationen, die mit parametrisierbaren Heuristiken lernen, werden erhöht. Das MDL-Kriterium sticht hier für die Konfigurationen mit dem *Klösgen-Maß* und das CutOff-Kriterium für das *m-Estimate* heraus.

Ein Blick auf Tabelle B-2 und B-3 (Anhang B) zeigt, dass die durchschnittlichen Genauigkeit der Konfiguration Accuracy für die drei CutOff-Parameter erst leicht ansteigt und dann wieder abfällt. Diese Beobachtung legt nahe, dass ein optimaler CutOff-Parameter für Accuracy zwischen 0,6 und 0,9 liegt. Auch der CutOff-Parameter 0,3, der in [Fürn94b] als guter Parameter für *Correlation* gefunden wurde, ist hier ein guter Wert. Hier sollte aber noch erwähnt werden, dass das CutOff-Kriterium in [Fürn94b] zwei Regeln miteinander verglichen hat und nicht auf einer absoluten Bewertung einer Regel gemacht wurde. Dennoch ist dieser Wert auch für den absoluten CutOff in Verbindung mit Correlation der beste Wert.

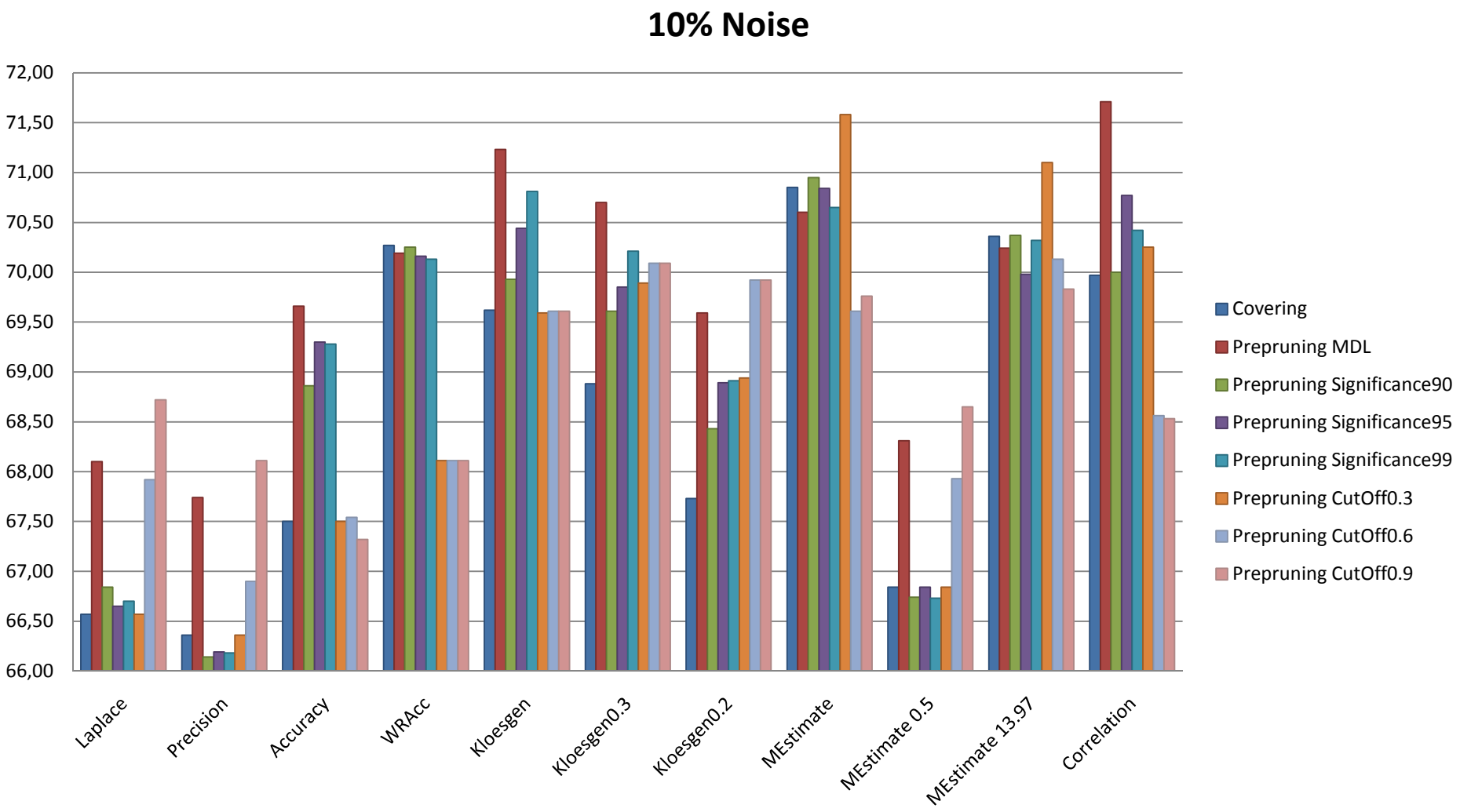

Abbildung 32: durchschnittliche Genauigkeiten des Preprunings auf dem 10% Paket

Wie man gesehen hat, ist es durchaus möglich, die Genauigkeiten der Covering-Konfigurationen durch Pruning noch zu erhöhen. Die Frage, die sich hier nun stellt, ist, welcher der Konfigurationen denn nun signifikant besser als die Covering-Konfiguration mit gleicher Heuristik ist. Für die Heuristik *Correlation* ist dies die Konfiguration Correlation-MDL. Diese Konfiguration ist auf dem 5%-Paket mit 95%-iger Sicherheit und auf dem 10%-Paket mit 99%-iger Sicherheit besser als die Covering-Konfiguration mit *Correlation* [COV-PRE]. Für das *Klösgen-Maß* sind das die Konfigurationen in Kombination mit dem Significance Testing und einem Signifikanzlevel von 95% oder 99%, die signifikant bessere Ergebnisse erzielen, als der Covering-Algorithmus mit der gleichen Heuristik. Für das CutOff-Kriterium lässt sich nur für die Heuristik *Correlation* eine Konfiguration finden, die signifikant besser als Covering mit gleicher Heuristik. Die Konfiguration, die sich hier erkennen lässt, ist Correlation-Cut0.3 [COV-PRE].

Auch wenn das Prepruning die Ergebnisse des Covering-Algorithmus verbessern konnte, kann es mit dem RIPPER-Algorithmus nicht mithalten. Einige der Prepruning-Konfigurationen, wie Kloesgen-Sig99, MEstimate13.97-Sig95, Correlation-MDL oder MEstimate13.97-Cut0.3, sind zwar signifikant besser als die Konfigurationen JRip-P und JRip-OP [RIP-PRE], aber Hier muss gesagt werden, dass bei den Konfigurationen des Ripper das Pruning ausgeschaltet ist. Anhand der Tabelle [RIP-PRE] lässt sich feststellen, dass die Konfiguration JRip hier eindeutig besser als die evaluierten Prepruning-Konfigurationen ist.

Insgesamt lässt sich sagen, dass das Prepruning im Vergleich zum Ripper keine signifikante Verbesserung erzielt, selbst dann nicht wenn Heuristiken zum Lernen benutzt werden, für die bereits optimale Parameter gefunden wurden. Auch bei Heuristiken die stark verallgemeinern oder zum Overfitting neigen, ist der Algorithmus Ripper signifikant besser. Im Vergleich zu den Covering-Konfigurationen wurden durch das Prepruning signifikante Verbesserungen erzielt, vor allem durch die parametrisierbaren Heuristiken und Correlation. Das Prepruning-Stopkriterium, das hier die besten Ergebnisse erzielt, ist das MDL-Kriterium (siehe auch [TAB-VS]). Bei dem Vergleich der Stopkriterien untereinander lässt sich eine Rangfolge erstellen, welches Kriterium am besten geeignet ist. Das MDL-Kriterium, vor allem in Verbindung mit den Heuristiken *Klösgen-Maß* und *Correlation*, eignet sich hier am besten [TAB-VS]. Danach folgt das Significance Testing, wobei man feststellt, dass mit steigendem Signifikanzlevel auch bessere Ergebnisse erzielt werden. Vor allem aber in Kombination mit den parametrisierbaren Heuristiken und optimalem Parameter bzw. einem Parameter, der leicht unter dem optimalem Parameter liegt. Am wenigsten geeignet ist hier das CutOff-Kriterium, besonders mit sehr hohem CutOff-Parameter. Die Konfigurationen mit einem CutOff-Parameter von 0.3 sind hier die Konfigurationen, die für das Kriterium die besten Ergebnisse erzielen, insbesondere aber die Konfigurationen MEstimate-Cut0.3 und MEstimate13.97-Cut0.3 [TAB-VS].

6.3.3 Ergebnisse IREP

Die Ergebnisse, die von den IREP Konfigurationen erzielt werden, sind im gesamten gesehen sehr erstaunlich. Die Zeiten, die zum Lernen einer Hypothese benötigt werden, und die endgültigen Hypothesengrößen werden hier drastisch reduziert (siehe Anhang B, Tabellen B-3 bis B-12). Das verblüffende aber ist, dass auch die durchschnittlichen Genauigkeiten sehr stark abfallen. Betrachtet man die Datensätze im Einzelnen, so gibt es auch hier einige Ausreißer - nicht nur ins Negative. Dennoch kann man sehen, dass die einzelnen Genauigkeiten eher schlechter sind als die Genauigkeiten, die von Covering ohne Pruning erzielt werden.

Aufgrund der stark abfallenden Genauigkeiten wurde für einen weiteren Vergleich auch IREPOpt (vgl. Kapitel 5.7) implementiert, um zu sehen, welche Auswirkung eine abschließende Optimierungsphase der gelernten Hypothese auf die Genauigkeiten hat. Dennoch sind die Ergebnisse

auch hier so wie beim IREP selbst, teilweise sogar etwas schlechter. Aus diesem Grund wurden die Konfigurationen von IREPOpt hier nicht weiter evaluiert. Auch das Entfernen der im Covering Algorithmus enthaltenen Pruningmechanismen – das *IncAcc*-Kriterium - in Kombination mit IREP führte eher zu schlechteren Ergebnissen als zu besseren. Die Ergebnisse, die in den B-5 bis B-12 (siehe Anhang B) enthalten sind, basieren auf IREP-Konfigurationen, die die grundlegenden Pruningmechanismen des Covering-Algorithmus enthalten. Zusätzlich dazu wird das Lernen einer Regel neu gestartet, sobald entweder eine leere Regel gelernt wurde oder eine Regel keine Beispiele auf der Pruningmenge abdeckt. Ist dies der Fall, wird insgesamt fünf mal versucht eine neue Regel zu lernen, in dem die Trainingsmenge erneut zufällig in Growing- und Pruningmenge aufgeteilt wird.

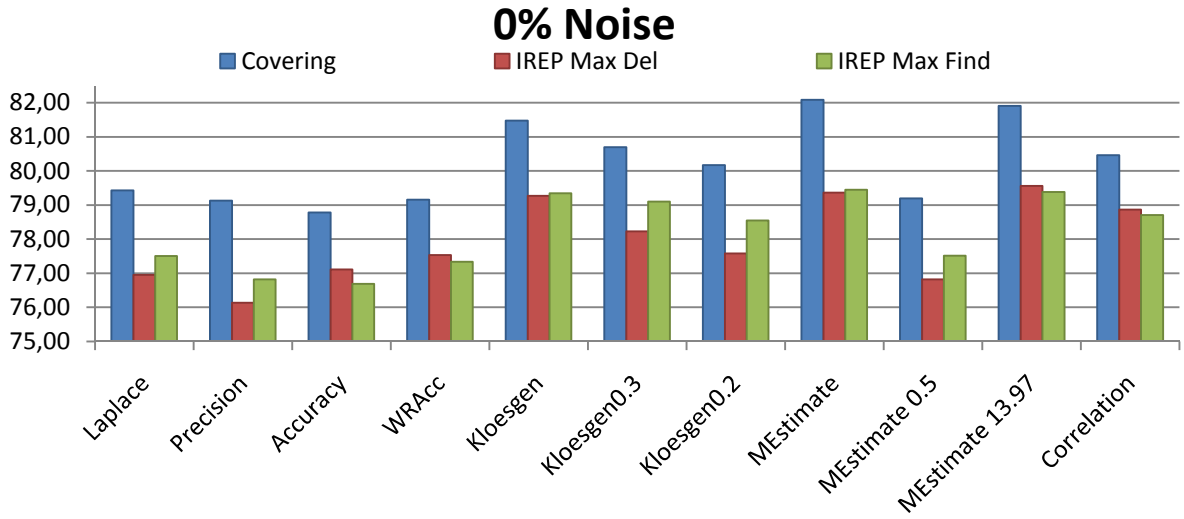

Abbildung 33: durchschnittliche Genauigkeiten Von Covering und IREP mit Stopkriterium MaximumErrorRate auf dem 0%-Paket

Die Abbildungen 33 bis 35 visualisieren die durchschnittlichen Genauigkeiten der IREP-Konfigurationen mit dem *MaximumErrorRate* Stopkriterium. Diese Konfigurationen dienen sozusagen als Startpunkt für den Vergleich der IREP-Konfigurationen, da das hier verwendete Stopkriterium auch innerhalb des Algorithmus RIPPER [Cohe95] und dem ursprünglichen IREP-Algorithmus [FüWi94] verwendet wurde. Anhand der Abbildung 33 sieht man, dass die erzielten Genauigkeiten auf dem 0%-Paket ausschließlich schlechter sind als die der Covering-Konfigurationen mit der gleichen Heuristik. Das Gleiche stellt man auch für das 5%-Paket fest, wie man in Abbildung 34 sieht.

Die höchsten durchschnittlichen Genauigkeiten werden auf dem 0%-Paket neben der Heuristik *Correlation* von den parametrisierbaren Heuristiken, mit optimalem Parameter oder einem Parameter der leicht unter dem optimalen Parameter liegt, erzielt. Aufgrund der Ergebnisse, die die IREP-Konfigurationen erzielt haben, legt dies den Schluss nahe, dass die Genauigkeiten nur so „hoch“ sind, weil die verwendeten Heuristiken schon sehr gute Ergebnisse erzielt haben. Selbst wenn der Lernalgorithmus Heuristiken zum Lernen verwendet, die zum Overfitting neigen, konnte die Genauigkeit nicht erhöht werden. Auf nicht verrauschten Daten eignet sich der *delete-last-condition* Operator zum Prunen besser als der *find-best-simplification* Operator, da mit dem *delete-last-condition* Operator bei mehr Konfigurationen eine höhere Genauigkeit erzielt wurde als bei den Konfigurationen, die den *find-best-simplification* Operator verwendeten. Das gleiche gilt auch für das 5%-Paket.

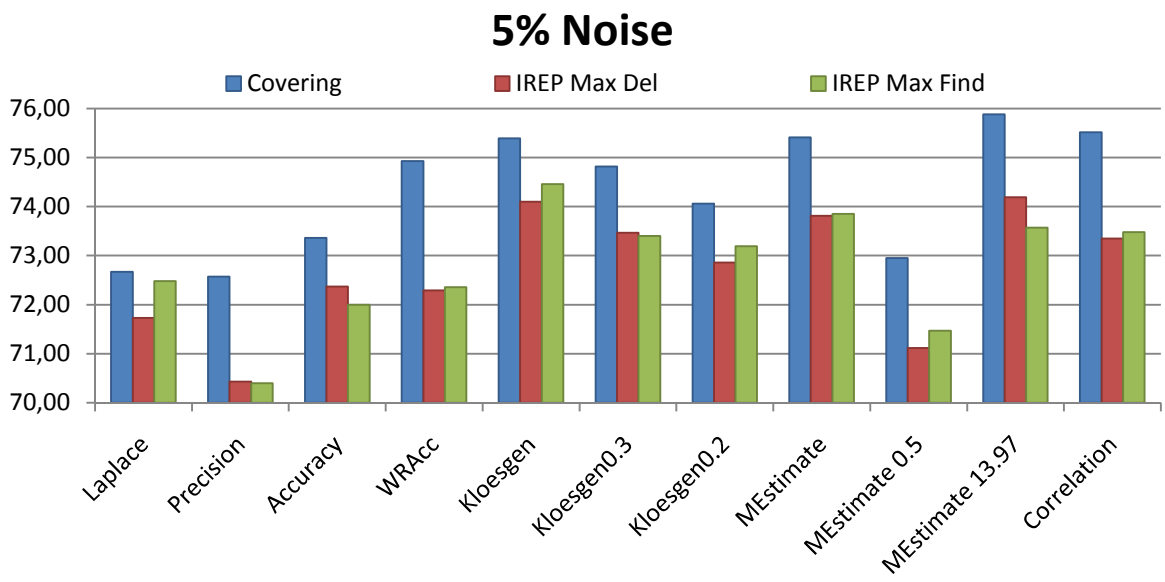

Abbildung 34: durchschnittliche Genauigkeiten Von Covering und IREP mit Stopkriterium MaximumErrorRate auf dem 5%-Paket

Betrachtet man die Abbildung 35, so erkennt man, dass einige IREP-Konfigurationen bessere Genauigkeiten erzielen konnten als die Covering-Konfigurationen mit gleicher Heuristik. Dennoch sind die Verbesserungen so gering und wenige, dass man diese als statistische Ausreißer ansehen kann. Damit gilt für die IREP-Konfigurationen mit dem Stopkriterium *MaximumErrorRate* auf dem 10%-Paket dasselbe wie auf dem 0%- und 5%-Paket. D.h., dass selbst auf stark verrauschten Daten durch das Pruning keine Verbesserung erzielt werden konnte.

Es stellt sich nach diesen Ergebnissen nun die Frage, warum die IREP-Konfigurationen keine Verbesserungen erzielen konnten. Werden Heuristiken verwenden, die eine Trainingsmenge overfitten, ist die Anzahl der abgedeckten Beispiele einer gelernten Regel schon sehr gering. Durch die Aufteilung der Trainingsmenge in Growing- und Pruningmenge kann man sagen, dass solche Regeln auf der Growingmenge nur wenige Beispiele abdecken. Dies führt dazu, dass dieselbe Regel auf der Pruningmenge nur wenige oder eher sogar kein Beispiel abdeckt. Solche Regeln werden durch das *IncAcc*-Kriterium abgelehnt und das Lernen wird abgebrochen. Selbst wenn die Regel auf der Pruningmenge geprunt wird, und damit auf der Pruningmenge mehr Beispiele abdeckt, so deckt diese Regel auch mehr Beispiele auf der Growingmenge ab. So kann es sein, dass diese Regel auf der Pruningmenge mehr *tp* als *fp* abdeckt, auf der Growingmenge der Fall aber genau entgegengesetzt ist und diese Regel mehr *fp* als *tp* abdeckt. Auch solche Regeln werden vom *IncAcc*-Kriterium abgelehnt und das Lernen wird unterbrochen. Eine ähnliche Erklärung lässt sich für Heuristiken, die übergeneralisieren, finden. Neben Regeln, die keine Abdeckung auf der Pruningmenge haben, kann es passieren, dass bei einer schon sehr generellen Regel ein Attributtest geprunt wird, der dafür verantwortlich war, dass auf der Growingmenge *tp>fp* gilt. Fehlt dieser Attributtest in der Regel, so gilt tp<fp und die Regel wird abgelehnt und das Lernen unterbrochen.

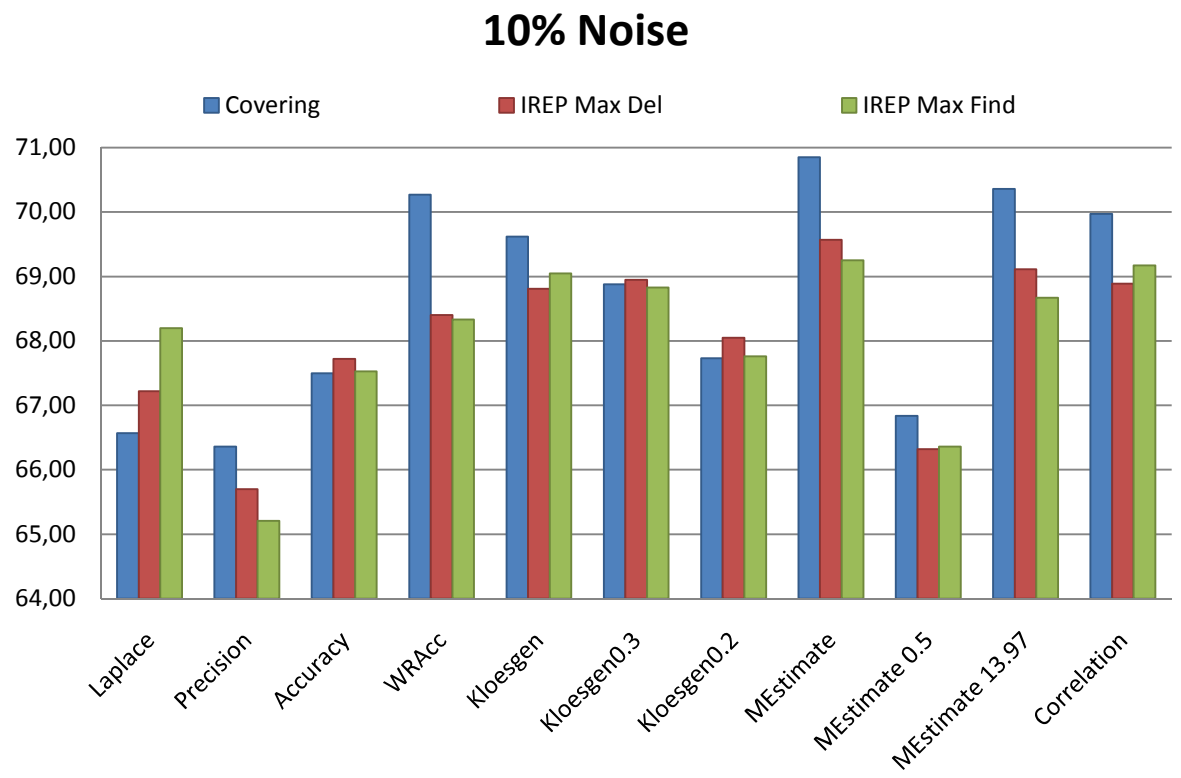

Abbildung 35: durchschnittliche Genauigkeiten Von Covering und IREP mit Stopkriterium MaximumErrorRate auf dem 10%-Paket

Eine weitere Frage, die sich stellt, ist, ob es vielleicht ein Stopkriterium gibt, das bessere Ergebnisse erzielt als das *MaximumErrorRate* Stopkriterium. Aus diesem Grund wurden auch die Prepruning Stopkriterien für IREP untersucht. Das zweite Stopkriterium, das für die IREP-Konfigurationen untersucht wurde, ist das MDL-Kriterium. Hier muss allerdings jetzt schon gesagt werden, dass die Ergebnisse nur der Vollständigkeit wegen aufgeführt werden. Wie man an Tabelle B12 (siehe Anhang B) erkennt, werden im Großen und Ganzen nur leere Hypothesen gelernt bzw. im Schnitt ca. eine Regel pro Hypothese. Dies liegt im Wesentlichen daran, dass die Heuristiken Regeln lernen, die nur sehr wenige Beispiele innerhalb der Trainingsmenge abdecken. Deckt eine Regel schon wenige Beispiele der Trainingsmenge ab, so ist es sehr wahrscheinlich, dass auch sehr wenige bis keine Beispiele der Pruningmenge abgedeckt werden. Also wird der Test $L(R)+L(E|R)<L(E)$ nahezu immer als falsch ausgewertet, da die gelernten Regeln die Pruningmenge nicht komprimieren können. Selbst die zusätzlichen Restarts verändern an dem Ergebnis nichts.

In Abbildung 36 sind wiederum die durchschnittlichen Genauigkeiten der Konfigurationen visualisiert. Aber diese Abbildung ist etwas kritischer zu betrachten, da hier aufgrund der sehr schlechten Ergebnisse, die durchschnittliche Genauigkeit über alle 171 Datensätze gezeigt ist, dennoch kann man sich anhand der Tabelle B-12 davon überzeugen, dass keine der IREP-Konfigurationen, die das MDL-Kriterium verwendet haben, bessere Ergebnisse als die Covering-Konfigurationen mit gleicher Heuristik. Die Schwankungen der durchschnittlichen Genauigkeiten liegen, wenn man die einzelnen Datenpakete betrachtet zwischen -7% und -20% im Vergleich zum Covering.

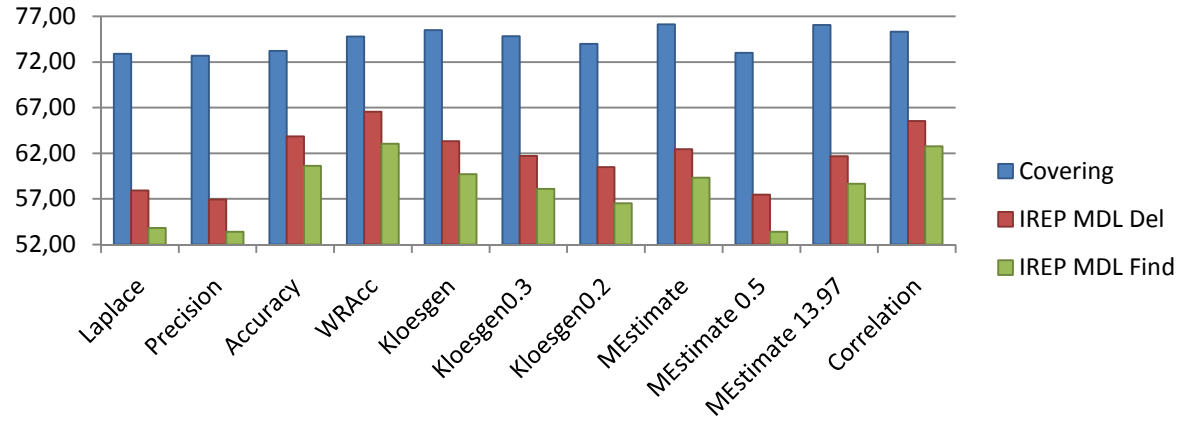

Abbildung 36: durchschnittliche Genauigkeiten von Covering und IREP mit Stopkriterium MDL auf allen Datenpaketen

Für die letzten beiden Stopkriterien, die in dieser Arbeit untersucht wurden, wird auf eine Visualisierung der durchschnittlichen Genauigkeiten auf dem 0%- und 5%-Paket verzichtet, da keine der IREP-Konfigurationen eine Verbesserung gegenüber dem Covering erzielen konnte (vgl. auch Tabelle B-6 bis B-11, Anhang B). Anhand von Abbildung 37 sieht man, dass auch das Significance Testing in Kombination mit den IREP-Konfigurationen auf dem 10%-Paket eher keine Verbesserung erzielen konnte. Die einzigen Ausnahmen, die man hier erkennt, sind die IREP-Konfigurationen, die mit den Heuristiken *Accuracy*, *Klösgen-Maß* und *Correlation* lernen. Besonders interessant ist die Tatsache, dass die IREP-Konfigurationen mit dem *delete-last-condition* Operator für eine Heuristik die gleichen Hypothesen lernen. Das Signifikanzlevel ist hierbei unerheblich. Die Konfiguration Laplace-DEL-Sig90 lernt die gleichen Hypothesen wie die Konfigurationen Laplace-DEL-Sig95 und Laplace-DEL-Sig99. Dies stellt man auch für die übrigen 10 Heuristiken fest. Dieses Verhalten wird durch die Tabellen B-9 bis B-10 (siehe Anhang B) und die Tabelle [TAB-VS] noch untermauert. Für den *find-best-simplification* Operator, erkennt man ein Absinken der Genauigkeit je höher das Signifikanzlevel ist (siehe Abb. 37).

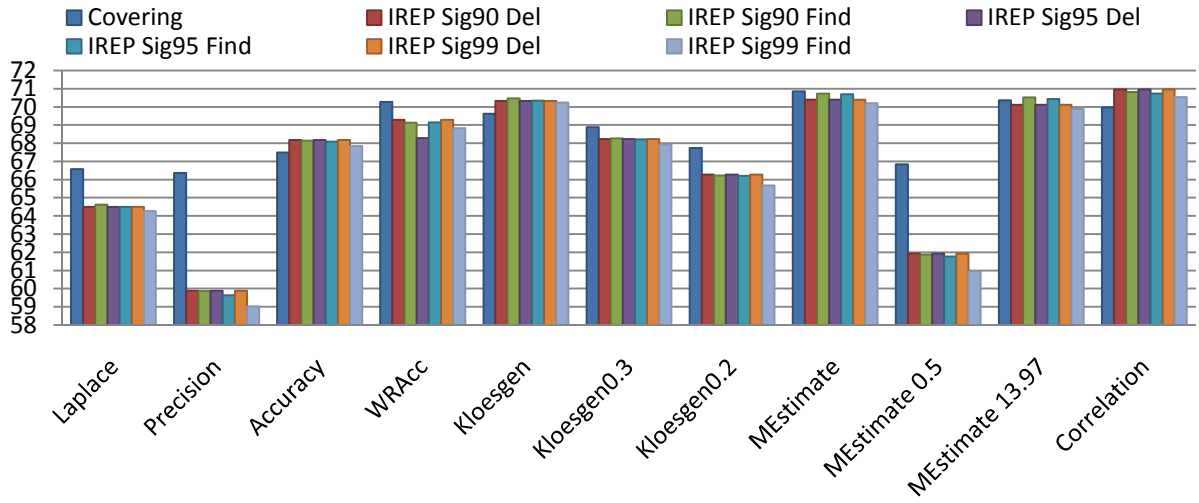

Abbildung 37: durchschnittliche Genauigkeiten von Covering und IREP mit Stopkriterium Significance Testing auf dem 10%-Paket

Die letzten Konfigurationen für IREP, die hier verglichen werden verwenden als Stopkriterium das absolute *CutOff*-Kriterium. Wie man an den Tabellen B-6 bis B-8 (Anhang B, blau hinterlegt) erkennt, werden bei den Konfigurationen, die mit dem *Klösgen-Maß* und *WRA* lernen, für die CutOff-Parameter 0.3, 0.6 und 0.9 nur leere Hypothesen gelernt. Die einzige Ausnahme sind die IREP-Konfigurationen, die zum Lernen das *Klösgen-Maß* verwenden, für einen CutOff-Parameter von 0.3.

Auch ein Blick auf die WTL-Tabelle [TAB-VS] verdeutlicht dieses Ergebnis und Abbildung 37 zeigt das Verhalten nochmals für das 10%-Paket. Diese Beobachtung zeigt wie beim Prepruning, dass diese Heuristiken in Verbindung mit dem *CutOff*-Kriterium und den gewählten CutOff-Parametern nicht geeignet sind. Regeln, die mit diesen Heuristiken gelernt werden, erhalten im Allgemeinen eine Bewertung, die unterhalb der CutOff-Parameter liegt. Dies führt dazu, dass das Lernen von Regeln für eine Klasse zu früh abgebrochen wird und keine Regeln in die Hypothese mit aufgenommen werden.

Auch erkennt man an Abbildung 37, dass zwei IREP-Konfigurationen die Genauigkeiten im Vergleich zum Covering erhöhen konnten. Die Konfigurationen mit den Heuristiken *Correlation* und *Accuracy* für einen CutOff-Parameter von 0.3 bzw. bei *Accuracy* für einen die Parameter 0.3 und 0.6. Auch wenn die Konfigurationen mit dem *Klösgen-Maß* keine Verbesserung erzielen konnten, sieht man an Abbildung 37 doch, dass es durchaus sinnvoll ist, diese Heuristik mit einem suboptimalen Parameter zu verwenden. Die durchschnittlichen Genauigkeiten steigen mit sinkendem Parameter für das CutOff-Kriterium mit CutOff-Parameter 0.3. Auch sieht man, dass die Genauigkeiten der Konfigurationen, ähnlich denen des Preprunings, mit steigendem CutOff-Parameter sinken.

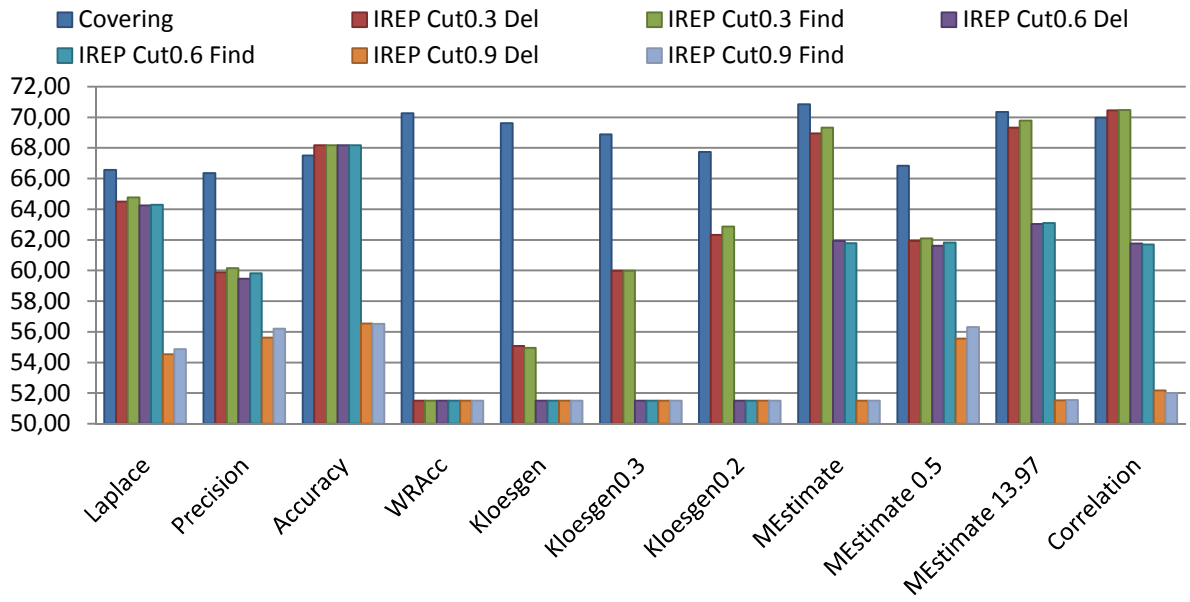

Abbildung 38: durchschnittliche Genauigkeiten von Covering und IREP mit Stopkriterium absolute CutOff auf dem 10%-Paket

Nichtsdestoweniger kann man anhand der Ergebnisse feststellen, welcher der IREP-Konfigurationen im Vergleich zu den anderen signifikant besser sind. Vergleicht man die IREP-Konfigurationen mit MaximumErrorRate untereinander, so sieht man, dass die Konfigurationen mit parametrisierbarer Heuristik die besten Ergebnisse erzielen. Die besten Ergebnisse werden hier von MEstimate13.97-DEL-Max und MEstimate-FIND-Max erzielt (vgl. [TAB-VS]), die Vermutung liegt hier nahe, dass diese Konfigurationen besonders gut sind, da die Heuristik, die hier verwendet wird schon sehr gute Ergebnisse erzielt. Im Grunde genommen sind beide Operatoren ungefähr gleich gut geeignet, wobei die Konfiguration MEstimate13.97-DEL-Max und die Konfiguration MEstimate-FIND-Max, im Vergleich der Operatoren untereinander die besten Ergebnisse erzielen (vgl. [TAB-VS]).

Betrachtet man die WTL Ergebnisse der IREP-Konfigurationen mit MDL-Kriterium für die zwei Operatoren [TAB-VS], zeigt sich, dass die Heuristiken *Correlation*, *Accuracy* und *WRA* für beide Operatoren am geeignetsten sind. Die Konfiguration WRA-FIND-MDL sticht hier besonders hervor, da sie mit 99% Sicherheit besser als die Konfigurationen mit dem *delete-last-condition* Operator sind.

Welcher Operator aber nun wirklich in Verbindung mit dem MDL-Kriterium am besten geeignet ist, lässt sich auf Grund der generell schlechten Ergebnisse dieser IREP-Konfigurationen nicht feststellen.

Vergleicht man die IREP-Konfigurationen mit Significance Testing untereinander für die verschiedenen Pruningoperatoren und Signifikanzlevel, erkennt man, dass die Konfiguration Correlation-DEL-Sig neben den Konfigurationen mit parametrisierbaren Heuristiken und optimalem Parameter besonders hervorsticht [TAB-VS]. Diese Konfiguration ist mit 99%-iger Sicherheit besser als die Konfigurationen mit dem *find-best-simplification* Operator. Die einzigen Ausnahmen sind hier die Konfigurationen MEstimate-FIND-Sig, MEstimate13.97-FIND-Sig und Kloesgen-FIND-Sig. Für den *find-best-simplification* Operator sticht die Konfiguration Kloesgen-FIND-Sig, neben den Konfigurationen MEstimate-FIND-Sig, MEstimate13.97-FIND-Sig, besonders hervor. Die Konfiguration ist mit 99%-iger Sicherheit besser als die Konfigurationen mit dem *delete-last-condition* Operator. Ausnahmen sind hier Konfigurationen mit dem *MEstimate*, dem *MEstimate13.97* und *Correlation*. Auch lässt sich für den *find-best-simplification* Operator feststellen, dass für ein niedriges Signifikanzlevel die besten Ergebnisse erzielt werden. Betrachtet man die durchschnittlichen Schwankungen stellt man fest, dass diese für das Signifikanzlevel 90% am niedrigsten sind, wohingegen für das Signifikanzlevel 99% am höchsten.

Anhand der Tabellen B-6 bis B-8 sieht man, dass die Konfigurationen Correlation-FIND-Cut0.3 und Correlation-DEL-Cut0.3 die höchsten durchschnittlichen Genauigkeiten erzielen. Betrachtet man das Verhalten dieser Konfigurationen für steigende CutOff-Parameter, sieht man, dass die Ergebnisse kontinuierlich schlechter werden. Dieses Verhalten legt nahe, dass der CutOff-Parameter 0.3 für *Correlation* auch in Verbindung mit IREP ein guter Parameter ist. Aber selbst für diese Konfigurationen gilt, dass die Genauigkeiten im Vergleich zum Covering-Algorithmus erst auf stark verrauschten Daten verbessert werden. Für stark verrauschte Daten ist die Konfiguration Correlation-FIND-Cut0.3 etwas besser als die Konfiguration mit dem *delete-last-condition* Operator, was durch die WTL-Tabelle [TAB-VS] bestätigt wird.

Die nächste Frage, die sich stellt, ist, welche Ergebnisse werden eigentlich von den IREP-Konfigurationen im Vergleich zu den Prepruning-Konfigurationen erzielt. Nahezu alle Prepruning-Konfigurationen sind mit 99%-iger Sicherheit besser als die IREP-Konfigurationen mit MDL-Kriterium. Kann man dies nicht mit einer gewissen Sicherheit, hier mindestens 95%, sagen, sieht man trotzdem, dass die Prepruning-Konfigurationen mehr Gewinne als Verluste erzielen [PRE-IREP].

In groben Zügen lässt sich sagen, dass die IREP-Konfigurationen nur für die parametrisierbaren Heuristiken und *Correlation* bessere Ergebnisse erzielen als die Prepruning-Konfigurationen, die mit Heuristiken lernen, die zum Overfitting neigen. Besonderes Augenmerk liegt hier auf den IREP-Konfigurationen mit Significance Testing und eingeschränkt auf den Konfigurationen mit *CutOff*-Kriterium für einen CutOff-Parameter von 0.3 [PRE-IREP].

Die IREP-Konfigurationen, die das Stopkriterium *MaximumErrorRate* verwenden, schneiden ähnlich zu den Konfigurationen mit dem MDL-Kriterium ab. Diese IREP-Konfigurationen erzielen im Vergleich zu den Prepruning-Konfigurationen nur Verbesserung, sofern die IREP-Konfigurationen mit einer parametrisierbaren Heuristik lernen. Die Verbesserungen werden hier allerdings nur gegen Prepruning-Konfigurationen erzielt, die mit Heuristiken, die zum Overfitting neigen, lernen [PRE-IREP].

Im Folgenden Abschnitt wird versucht für jeden Pruningoperator die besten Konfigurationen zu ermitteln. Die Konfiguration, die sich für den *delete-last-condition* Operator als am geeignetsten herausstellt ist Correlation-DEL-Sig [TAB-VS]. Diese Konfiguration ist mit 99%-iger Sicherheit

besser als die IREP-Konfigurationen mit einem anderen Stopkriterium [TAB-VS]. Ausnahmen sind hier die Konfigurationen, die mit parametrisierbaren Heuristiken lernen und dasselbe Stopkriterium verwenden, dennoch sieht man hier, dass die Konfiguration Correlation-DEL-Sig mehr Gewinne als Verluste erzielt (gegen die Konfiguration MEstimate-DEL-Max ist die Konfiguration Correlation-DEL-Sig nur mit 95%-iger Sicherheit besser).

Für den *find-best-simplification* Operator existieren mehrere Konfigurationen, die ungefähr gleich gute Ergebnisse erzielen. Die erste Konfiguration, die hier ins Auge sticht ist Kloesgen-FIND-Sig90, deren WTL-Verhalten ähnlich zu der Konfiguration Correlation-DEL-Sig ist. Die zweite Konfiguration, die hier signifikant gute Ergebnisse erzielt ist MEstimate13.97-FIND-Cut0.3. Auch diese Konfiguration ist im Vergleich zu den anderen IREP-Konfigurationen sehr oft mir 99%-iger Sicherheit besser. Welcher der drei Konfigurationen nun die beste ist erkennt man anhand der Tabelle [TAB-VS]. Hier zeigt sich, dass die Konfigurationen Kloesgen-FIND-Sig90 und Correlation-DEL-Sig mit 99%-iger Sicherheit besser sind als die Konfiguration mit dem MEstimate. Vergleicht man beide Konfigurationen, Kloesgen-FIND-Sig90 und Correlation-DEL-Sig, kann man für die verschiedenen Datenpakete nicht feststellen, welche der beiden Konfigurationen nun wirklich die bessere ist. Man sieht nur, dass die Konfiguration Correlation-DEL-Sig mehr Gewinne als Verluste gegenüber der Konfiguration Kloesgen-FIND-Sig90 erzielt. Betrachtet man allerdings alle drei Datenpakete insgesamt sieht man, dass die Konfiguration Correlation-DEL-Sig mit 95%-iger Sicherheit besser ist als die Konfiguration Kloesgen-FIND-Sig90 [TAB-VS].

Generell lässt sich sagen, dass die IREP-Konfigurationen keine signifikanten Verbesserungen gegenüber dem Covering mit gleicher Heuristik erzielen konnten [COV-IREP]. Die hier gefundenen Ergebnisse stehen in direktem Widerspruch zu den früher gefundenen Ergebnissen für den IREP Algorithmus. Eigentlich sollte sich die Genauigkeit wie in [FüWi94] erhöhen bzw. sollten die IREP-Konfiguration ähnlich gute Ergebnisse liefern, wie der Algorithmus Ripper, der auf dem gleichen Prinzip basiert. So ist auch das Ergebnis nicht verwunderlich, dass die Konfiguration JRip, die im Vergleich zu den IREP-Konfigurationen „bessere“ Konfiguration ist. Nur sofern das Pruning beim RIPPER-Algorithmus ausgeschaltet wird, sieht man, dass einige der IREP-Konfigurationen mit Significance Testing signifikant besser sind [RIP-IREP].

Auch die wenigen Konfigurationen, die im Vergleich zu Covering auf verrauschten Daten eine Verbesserung erzielen konnten, sind in ihrer Anzahl und der Verbesserung so gering, dass diese als statistische Ausreißer, nicht nur für das Stopkriterium *MaximumErrorRate*, sondern für alle Stopkriterien, betrachtet werden können. Betrachtet man dagegen die IREP-Konfigurationen deren durchschnittliche Genauigkeit nahe an denen des Covering mit gleicher Heuristik liegen, so kann man darauf schließen, dass diese Ergebnisse nur erzielt wurden, weil die verwendete Heuristik schon gute Ergebnisse erzielt hat und durch das Prunen nur eine Verschlechterung erreicht wurde. Anders ausgedrückt, das Pruning konnte an der endgültigen Hypothese nicht viel „kaputt“ machen. Welcher der zwei Pruningoperatoren nun am besten für IREP geeignet ist, lässt sich hier nur bedingt feststellen, da die Ergebnisse unterschiedlich für die Stopkriterien und verwendeten Heuristiken ausfallen. Beide Operatoren sind ungefähr gleich gut geeignet, wobei der *delete-last-condition* Operator etwas besser abschneidet.

6.4 Ist Pruning überhaupt noch notwendig?

Die Beantwortung der Frage gestaltet sich als nicht einfach, da man nur für eine Pruningmethode eine definitive Aussage treffen kann. Aufgrund der sehr schlechten Ergebnisse der IREP-Konfigurationen lässt sich für diesen Teil die Frage nur bedingt beantworten. Die Testreihen haben hier gezeigt, dass IREP deutlich schlechter abschneidet als der Regellerner Covering mit gleicher Heuristik, was darauf

deutet dass das Pruning nichts mehr bringt. Diese Ergebnisse stehen allerdings im Widerspruch zu den bisherigen Resultaten, in denen gezeigt wurde, dass der Algorithmus bessere Ergebnisse erzielt als ein Regellerner ohne Pruning. Der Grund, weshalb die Ergebnisse so stark von denen in [Fürn97] und [Cohe95] bzw. von den Ergebnissen der Konfiguration JRip-O abweichen, konnte im Verlauf dieser Diplomarbeit nicht festgestellt werden. Auch weitere Modifikationen der IREP-Konfigurationen durch eine abschließende, dem RIPPER-Algorithmus ähnliche, Optimierung der Regelmenge oder das Neustarten des Lernens falls eine „schlechte“ Regel gefunden wurde, hat an den Ergebnissen nichts ändern können. Der Widerspruch der Ergebnisse der IREP-Konfigurationen zu den Ergebnissen von [Fürn97] und [Cohe95], lässt auf eine Unstimmigkeit[14] innerhalb der Implementierung im SeCo-Framework schließen, die im Verlauf der Arbeit aber nicht gefunden werden konnte.

Für das Prepruning lässt sich eine Tendenz feststellen, die in Verbindung mit dem Grad des Rauschens der Daten steht. Für nicht verrauschte Daten (0% Noise) bringt das Prepruning nichts - weder für übergeneralisierende noch überspezialisierende Heuristiken. Je verrauschter die Daten werden umso mehr eignet sich ein Prunen. Auf den verrauschten Daten (5% bzw. 10% Noise) erkennt man, dass das Prepruning in Kombination mit Heuristiken, die stark overfitten die Ergebnisse noch verbessern kann. Im Schnitt lag die Verbesserung hier bei ca. 2%. Interessant ist hier, dass auch die Ergebnisse, die durch die parametrisierbaren Heuristiken mit optimalem Parameter durch das Prepruning noch verbessert werden konnten. Dies legt die Vermutung nahe, dass ein optimaler Parameter, wie er in [JaFü06] auf nicht verrauschten Daten gefunden wurde, kein Garant dafür ist, dass dieser optimale Parameter auch optimale Ergebnisse auf verrauschten Daten erzielt. Wählt man für die parameterisierbaren Heuristiken einen suboptimalen Parameter, sodass das Lernen mit der Heuristik dazu tendiert etwas speziellere Theorien zu lernen, kann das Prepruning auch die Ergebnisse noch verbessern. Auch die Ergebnisse der einzigen verwendeten, nicht-linearen Heuristik Correlation konnten durch das Prepruning noch verbessert werden. Allerdings ist ein Pruning nicht mehr notwendig, wenn die Heuristik, die verwendet wird stark verallgemeinert.

[14] Der Begriff Unstimmigkeit wurde hier gewählt, da es sich um viele Möglichkeiten handeln kann. Dies können zum Beispiel ein Defekt, ein Fehler, ein Bedingung an der falschen Stelle etc. sein.

Kapitel 7 Schlusswort

7.1 Zusammenfassung

In Kapitel 2 wurden die Separate-and-Conquer Strategie für das Regellernen genauer vorgestellt. Dazu wurde eine kurze Einführung in das Thema des induktiven Lernens gegeben (Kapitel 2.1) und die Merkmale und Eigenschaften der Separate-and-Conquer Algorithmen genauer beleuchtet (Kapitel 2.2). Der Kern, den alle Separate-and-Conquer Algorithmen gemeinsam haben, wurde anhand eines generischen Algorithmus vorgestellt (Kapitel 2.3) und die Grundlagen um die Algorithmen zu vergleichen wurde erklärt (Kapitel 2.4).

Das dritte Kapitel widmet sich ausschließlich den Suchheuristiken, die einen Teil des Suchverfahrens der Separate-and-Conquer Algorithmen ausmacht. In Kapitel 3.1 wurden grundlegende Eigenschaften für Regeln und Trainingsmengen festgehalten, auf die sich die Bewertung durch Heuristiken stützt. Eine Methode zur Visualisierung von Heuristik wurde in Kapitel 3.2 kurz beleuchtet. Das Ende des dritten Kapitels beleuchtete kurz die in dieser Arbeit verwendeten Heuristiken und deren Eigenschaften (Kapitel 3.3).

Das Pruning wurde eingehend in Kapitel 4 beleuchtet und deren Merkmale sowie Vorteile und Nachteile eingehend beleuchtet. Die ersten Methoden die vorgestellt wurden und wie diese mit verrauschten Daten umgehen, waren das Postpruning (Kapitel 4.1) und das Prepruning (Kapitel 4.2). Die Kapitel 4.3 und 4.4 befassen sich mit der Kombination und der Integration beider Methoden. In Kapitel 4.4 wurden auch die bekanntesten Algorithmen für diese Methode vorgestellt.

Um eine technische Grundlage für die Umsetzung der Pruningalgorithmen zu haben, wurde in Kapitel 5.1 das SeCo-Framework kurz vorgestellt und ein generischer Pruningalgorithmus definiert, der die Prozeduren umfasst, die jedem dieser Algorithmen zugrunde liegen. In einem weiteren Schritt wird die Realisierung der Komponenten (Kapitel 5.2), die für das Pruning notwendig sind, vorgestellt. Der Abschluss des Kapitels widmet sich ausschließlich der Umsetzung der Algorithmen mittels dieser Komponenten (Kapitel 5.3 bis 5.6).

Das sechste Kapitel hält die Ergebnisse des Vergleichs der Pruningalgorithmen fest. Dazu wurden die 57 verwendeten Datensätze des UCI-Repository kurz beschrieben und wie diese mit Rauschen versehen worden sind (Kapitel 6.1). Das Kapitel 6.2 dient der Beschreibung der 268 Konfigurationen, die in dieser Arbeit verglichen wurden. Das Grundgerüst des Vergleichs ist hier durch vier Konfigurationen des RIPPER-Algorithmus und elf Konfigurationen des Covering-Algorithmus gegeben. Die restlichen 253 Konfigurationen sind Erweiterungen des Covering-Algorithmus. In einem ersten Schritt werden die Algorithmen RIPPER und Covering miteinander verglichen (Kapitel 6.3.1), der zweite Schritt erweitert den Vergleich um die Prepruning-Konfigurationen (Kapitel 6.3.2). Der letzte Schritt erweitert den Vergleich um die IREP-Konfigurationen (Kapitel 6.3.3). Das nachfolgende Kapitel ermittelt welche der (Pruning-)Algorithmen die besten Ergebnisse erzielen und vergleicht diese nochmals untereinander um festzustellen, welcher der Algorithmen wirklich der beste ist. Abschließend wird versucht eine Antwort auf die Frage zu geben (Kapitel 6.5):

„Ist Prunen überhaupt noch notwendig, wenn die Heuristiken, die zum Lernen verwendet werden, schon sehr gute Ergebnisse erzielen?“

7.2 Schlussfolgerungen

Das Ziel der Arbeit die verschiedenen Pruningalgorithmen innerhalb eines einheitlichen Regellern-Frameworks, dem SeCo-Framework, zu integrieren ist mit Erfolg erreicht worden. Aufgrund der

Modularität der Komponenten und der Identifikation der veränderlichen Teile der Pruningalgorithmen anhand eines generischen Algorithmus, kann eine Vielzahl verschiedener Pruningalgorithmen innerhalb des Frameworks implementiert und konfiguriert werden. Die hier getesteten 264 Konfigurationen des Frameworks sind nur ein Anfang.

Die Antwort auf die Frage: *„Ist Prunen überhaupt noch notwendig, wenn die Heuristiken, die zum Lernen verwendet werden, schon sehr gute Ergebnisse erzielen?“*, konnte nur teilweise gegeben werden. Eine definitive Antwort für das inkrementelle Pre- und Postpruning kann hier nicht gegeben werden. Die Frage lässt sich nur anhand der hier gesammelten Daten geben und ist ein eindeutiges *„Nein, es ist nicht notwendig!“*. Wie bereits erwähnt stehen die Daten für das inkrementelle Pre- und Postpruning im Widerspruch zu den bereits in [Fürn97] und [Cohe95] gefundenen Ergebnissen. Deswegen bedarf es hier einer erneuten, genaueren Betrachtung der Methode. Andererseits konnte die Frage eindeutig für das Prepruning beantwortet werden (vgl. Kapitel 6.4). Je verrauschter die Trainingsdaten sind, desto notwendiger ist ein Pruning.

Innerhalb des vorliegenden Vergleichs hat sich gezeigt, dass das Prepruning die Pruningstrategie der Wahl ist, da bei 10 von 11 untersuchten Heuristiken eine Steigerung der Genauigkeit möglich ist. Nichtsdestoweniger hat der Vergleich der Algorithmen gezeigt, dass der RIPPER-Algorithmus der momentan beste Pruningalgorithmus für das Regellernen ist, vor allem auf verrauschten Daten.

7.3 Offene Punkte

7.3.1 Heuristiken im Framework

Das SeCo-Framework ist momentan stark an Value-Heuristiken angepasst, was dazu führt, dass Gain-Heuristiken, wie *FoilGain*, nicht oder eingeschränkt benutzt werden können. Folglich ist der nächste Schritt der Erweiterung des Frameworks eine Anpassung an Gain-Heuristiken. Aufgrund der fehlenden Möglichkeit Gain-Heuristiken effektiv zu nutzen, können viele Algorithmenkonfigurationen innerhalb des Frameworks nicht umgesetzt werden. Interessant wäre es den hier vorliegenden Vergleich um eine Konfiguration zu erweitern, die dem RIPPER-Algorithmus entspricht.

Man erkennt auch, dass es für manche Heuristik-Stopkriterium-Kombination einen optimalen Parameter für das Stopkriterium gibt bzw. dass für manche dieser Kombinationen die gewählten Parameter für das Stopkriterium nicht geeignet waren. Interessant wäre es für diese Kombinationen die optimalen Parameter zu finden.

Interessant wäre es auch für die parametrisierbaren Heuristiken optimale Parameter auf verrauschten Daten zu finden. Die Annahme dahinter ist, dass die Ergebnisse, die die Heuristik mit diesem Parameter erzielt, auch durch das Pruning nicht mehr verbessert werden können. Ein weiterer Aspekt wäre herauszufinden, wie sich die „optimalen“ Parameter auf verrauschten Daten verändern, um mit diesen Informationen vielleicht einen allgemeinen optimalen Parameter für die Heuristiken zu finden.

7.3.2 Pruning im SeCo-Framework

Auch was die Möglichkeiten des Prunings betrifft, die während dieser Arbeit implementiert wurden, ist die Spitze des Eisbergs noch nicht erreicht, d.h. es sind noch nicht alle Pruningalgorithmen, etc. implementiert. Einige Punkte, die noch realisiert werden können, sind:

- Ein anderes auf dem MDL-Prinzip basierendes Stopkriterium. Das hier verwendete Stopkriterium ist an manchen Stellen etwas zu stark gewesen. Interessant wären die Ergebnisse, die man erhält, wenn man ein weniger starkes Kriterium verwenden würde

- Die Erweiterung der vorhandenen Stopkriterien, sodass man diese auch für Regelmengen verwenden kann.
- Aufgrund des Widerspruchs der IREP-Ergebnisse ist es durchaus sinnvoll hier nochmals zu testen, warum dieser Widerspruch zustande kommt.
- Eine Optimierung von Regelmengen, wie sie zum Beispiel im RIPPER-Algorithmus verwendet wird, fehlt noch vollständig.
- Auch sind noch nicht alle Pruningoperatoren vollständig implementiert.

7.3.3 Der vorliegende Vergleich

Selbst für den Vergleich gibt es noch offene Punkte. Einige interessante Aspekte wären z.B.:

- Die Erweiterung des Vergleichs mit den Algorithmen TDP und REP.
- Eine Erweiterung des Vergleichs, sodass auch die Stopkriterien für Regelmengen genauer untersucht werden können.
- Auch wieder eine Erweiterung des Vergleichs, sodass für die Optimierungsphase für Regelmengen hinzugenommen und genauer untersucht werden kann.
- Erweiterung des Vergleichs durch einen Konfiguration, die an den RIPPER-Algorithmus angelehnt ist.

Literaturverzeichnis

[AsNe07] A. Asuncion & D.J. Newman. (2007). UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, School of Information and Computer Science.

[BrPa91] C. A. Brunk and Michael J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 389--393, Evanston, Illinois, 1991.

[ClNi89] P. Clark, T. Niblett. The CN2 induction Algorithm. *Machine Learning*, 3(4):261-283, 1989.

[Cohe93] W. W. Cohen (1993): Efficient pruning methods for separate-and-conquer rule learning systems *in International Joint Conferences on Artificial Intelligence* 1993, IJCAI 1993: 988-994.

[Cohe95] W. W. Cohen. Fast effective rule induction. In Machine Learning: *Proceedings of the Twelfth International Conference*, Lake Tahoe, California, 1995.

[DzBM92] S. Dzeroski and I. Bratko. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, Report ICOT TM-1182, 1992.

[Flac03] P.A. Flach. The geometry of ROC Spaces: Understanding maschine learning metrics through ROC isometrics. In *Proceedings 20th International Conference on Maschine Learning (ICML`03)*, pages 194-201. AAAI Press, 2003.

[Flac04] P. A. Flach. Tutorial on „The many faces of ROC analysis in maschine learning.“, *International Conference on Machine Learning 2004* (ICML`04 Tutorial), 2004. Notes available at http://www.cs.bris.ac.uk/~flach/ICML04tutorial/index.html

[FüFl03] J. Fürnkranz, P.A. Flach. An Analysis of Rule Evaluation Metrics. In *Proceedings 20th International Conference on Maschine Learning (ICML`03).* 2003.

[FüFl05] J. Fürnkranz, P.A. Flach. ROC `n` Rule Learning – Towards a better understanding of covering algorithms. *Maschine Learning*, 58(1): 39-77, 2005.

[Fürn94a] J. Fürnkranz. *Efficient Pruning Methods for Relational Learning*. PhD thesis, Vienna University of Technology, 1994.

[Fürn97] J. Fürnkranz. Pruning Algorithms for Rule Learning. *Machine Learning*, 27 (2):139--171, 1997.

[Fürn94b] J. Fürnkranz. FOSSIL: A robust relational learner. Lecture Notes in Computer Science, 784:122-137, 1994.

[Fürn03] J. Fürnkranz. Modeling Rule Precision. Technical Report OEFAI-TR-2003-35, Research Institute for Artificial Intelligence, Wien, Austria, 2003.

[FüWi94] J. Fürnkranz, G. Widmer. Incremental Reduced Error Pruning. In *Proceedings the Eleventh International Conference on Machine Learning*, pages 70-77,New Brunswick, NJ, 1994.

[Fürn99] J. Fürnkranz. Separate-and-Conquer Rule Learning. *Artificial Intelligence Review*,13(1): 3-54, 1999.

[Fürn01] J. Fürnkranz. Round Robin Rule Learning. In C. Brodley and A. Danyluk (Eds.), *Proceedings of the 18th International Conference on Machine Learning (ICML-01)*, pp. 146-153, Williamstown, MA, 2001. Morgan Kaufmann.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*, Addison-Wesley,1995.

[GrMY05] P.D. Grünwald, I.J. Myung, M. Pitt (editors). *Advances in Minimum Description Length: Theory and Applications*. 452 pages. MIT Press, April 2005.

[HaNR68] P. E. Hart, N.J. Nilsson, B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transaction on System Science and Cybernetics,* 4(2): 100-107, 1968.

[HoAP89] R. Holte, L. Acker, and B. Porter. Concept Learning and the Problem of Small Disjuncts. Technical report, Austin, TX, USA, 1989. Pages 813-818.

[JaFü06] F. Janssen, J. Fürnkranz. On Trading Off Consistency and Coverage in Inductive Rule Learning. In Althoff, K.-D. and Schaaf, M., editors, *Proceedings of the LWA 2006, Lernen Wissensentdeckung Adaptivität*, pages 306-313, Hildesheim, Germany. Gesellschaft für Informatik e. V. (GI), 2006.

[JaFü07] F. Janssen and J. Fürnkranz (2007c). On meta-learning rule learning heuristics. In *Proceedings of the 7th IEEE Conference on Data Mining (ICDM-07)*, pages 529-534, Omaha, NE.

[JaFü08] F. Janssen, J. Fürnkranz *An Empirical Comparison of Hill-Climbing and Exhaustive Search in Inductive Rule Learning*, Technical Report TUD-KE-2008-02, Knowledge Engineering Group, TU Darmstadt, 2008.

[Klös92] W. Klösgen. Problems for Knowledge Discovery in Databases and their Treatment in the Statistics Interpreter Explora. *International Journal of Intelligent Systems,* 7: 649-673, 1992.

[Mich69] R. S. Michalski. On the quasi minimal solution of the covering algorithm. In *Proceedings oft he 5th international symposiumon information processing (FCIP-69)*, volume 3A, pages 125-128, Bled, Yugoslavia, 1969.

[Mitc97] T. M. Mitchell. Maschine Learning, International Edition, McGraw-Hill Inc., 1997

[PaHa90] G. Pagallo, D. Haussler. Boolean feature discovery in empirical Learning. *Maschine Learning, 5,71-99, 1990.*

[Pfah95] B. Pfahringer. A New MDL Measure for Robust Rule Induction (Extended Abstract), in Lavrac N. and Wrobel S.(eds.), *Machine Learning: ECML-95*, Springer, Berlin/Heidelberg/New York/Tokyo, pp.331-334, 1995.

[QuCJ95] J.R. Quinlan, R.M. Cameron-Jones. Induction of Logic Programs: FOIL and Related Systems. *NewGenerationComputing,* 13(3,4): 287-312,1995.

[Riss78] J. Rissanen. Modeling by shortest data description, *Automatica*, Vol. 14, pp. 465-471, 1978.

[Thie05] M. Thiel. Separate and Conquer Framework und disjunktive Regeln. Master's thesis, TU Darmstadt, 2005.

[ToFL00] L. Todorovski, P. A. Flach, and N. Lavrac. Predictive performance of weighted relative accuracy. In Djamel A. Zighed, Jan Komorowski, and Jan Zytkow, editors, *4th European Conference on Principles of Data Mining and Knowledge Discovery* (PKDD2000), pages 255 - 264. Springer-Verlag, September 2000.

[SeEt94] R. Segal, O. Etizioni. Learning Decision Lists Using Homogenous Rules. In *National Confernce on Artificial Intelligenc e(AAAI-94),* pages 619-625, 1994.

[WeIn91] S. M. Weiß, N. Indurkyha. Reduced Complexity Rule Induction. In *International Joint Conference on Artificial Intelligence* (IJCAI-91) , pages 678-684, 1991.

[WFTH99] I.H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. Cunningham. Weka: Practical Machine Learning Tools and Techniques with Java implementations. In *Proceedings ICONIP/ANZIIS/ANNES'99 Int. Workshop: Emerging Knowledge Engineering and Connectionist-Based Info*, pages 192-196, 1999.

[Wiki08] Artikel *Chi-Quadrat-Test.* In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 2. Februar 2008, 20:51 UTC. URL: http://de.wikipedia.org/w/index.php?title=Chi-Quadrat-Test&oldid=41984417

Anhang A – WTL-Verzeichnis

In dieser Tabelle werden kurz die Namen der WTL-Tabellen aufgeführt und wie im Literaturverzeichnis aufgelistet. Jede Tabelle erhält dazu ein bestimmtes Kürzel. Der Ordner, in dem sich die Tabellen auf der CD befinden, heißt „<CD-Laufwerk>:\Tabellen\WTL\“.

[RIP-COV]	WTL_JRip_Covering.xls
[RIP-PRE]	WTL_JRip_Prepruning.xls
[RIP-IREP]	WTL_JRip_IREP.xls
[COV-PRE]	WTL_Covering_Prepruining.xls
[COV-IREP]	WTL_Covering_IREP.xls
[PRE-IREP]	WTL_Prepruning_IREP.xls
[TAB-VS]	WTL_Algorithmen_gegen_sichselbst.xls

Anhang B – Ergebnistabellen

Dieser Teil des Anhangs enthält alle Ergebnistabellen. Die Tabellen enthalten redundante Daten, damit ein Vergleich zwischen dem Covering-Algorithmus und den Erweiterungen des Algorithmus zu erleichtern. Die Tabellen fassen das Zeitverhalten in Sekunden, die durchschnittliche Hypothesengröße und die durchschnittliche Genauigkeit in % der Konfigurationen auf den drei Datenpaketen zusammen.

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
JRip	0,20	0,20	0,20	7,30	6,84	6,51	81,60	77,34	73,12
JRip-P	0,09	0,25	0,27	14,70	22,25	22,86	81,06	75,12	69,66
JRip-O	0,05	0,05	0,06	8,68	8,79	9,44	80,86	76,03	70,98
JRip-OP	0,09	0,23	0,25	14,70	22,25	22,86	81,06	75,12	69,66
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-MDL	0,18	0,20	0,19	13,11	14,19	14,07	76,63	71,85	68,10
Precision-MDL	0,24	0,26	0,25	17,02	18,63	18,39	76,17	71,40	67,74
Accuracy-MDL	0,23	0,26	0,28	7,35	7,88	7,75	77,27	73,45	69,66
WRAcc-MDL	0,48	0,71	0,79	4,16	4,19	4,09	78,97	74,77	70,19
Kloesgen-MDL	0,35	0,42	0,41	9,00	9,47	8,74	80,54	75,77	71,23
Kloesgen0.3-MDL	0,26	0,38	0,37	9,96	11,28	11,23	79,60	75,11	70,70
Kloesgen0.2-MDL	0,24	0,34	0,32	12,56	12,79	12,28	78,25	74,28	69,59
MEstimate-MDL	0,36	0,45	0,40	7,95	8,30	7,46	80,04	74,74	70,60
MEstimate 0.5-MDL	0,22	0,24	0,23	15,18	16,14	16,32	76,65	71,99	68,31
MEstimate 13.97-MDL	0,28	0,34	0,35	8,04	8,18	7,96	79,40	74,78	70,24
Correlation-MDL	0,43	0,52	0,54	8,37	8,11	8,81	80,49	76,10	71,71

Anhang B - 1: Ergebnisse Ripper, Covering, Prepruning mit MDL-Kriterium

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-CUT0.3	1,35	3,29	5,37	41,07	60,72	78,51	79,43	72,67	66,57
Precision-CUT0.3	1,49	3,76	5,85	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy-CUT0.3	1,41	2,32	3,31	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc-CUT0.3	0,21	0,39	0,46	3,88	3,86	3,35	75,86	71,68	68,11
Kloesgen-CUT0.3	0,26	0,38	0,45	9,70	11,82	11,98	77,72	73,16	69,59
Kloesgen0.3-CUT0.3	0,31	0,46	0,51	11,46	13,46	14,60	78,48	73,95	69,89
Kloesgen0.2-CUT0.3	0,39	0,52	0,60	14,21	15,60	16,75	77,79	73,27	68,94
MEstimate-CUT0.3	0,57	0,87	1,03	9,26	11,04	11,84	80,99	76,00	71,58
MEstimate 0.5-CUT0.3	1,41	3,51	5,66	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97-CUT0.3	0,60	0,95	1,16	11,40	13,65	15,46	81,21	76,25	71,10
Correlation-CUT0.3	0,64	0,85	0,92	11,25	13,28	14,37	80,33	75,55	70,25
Laplace-CUT0.6	0,33	0,78	0,94	16,42	23,02	23,84	77,40	71,55	67,92
Precision-CUT0.6	0,39	0,89	1,11	20,18	27,75	29,32	76,70	70,92	66,90
Accuracy-CUT0.6	1,40	2,32	3,30	35,37	42,44	54,63	78,78	73,36	67,54
WRAcc-CUT0.6	0,20	0,39	0,46	3,88	3,86	3,35	75,86	71,68	68,11
Kloesgen-CUT0.6	0,24	0,31	0,35	9,65	11,84	12,00	77,66	73,21	69,61
Kloesgen0.3-CUT0.6	0,26	0,33	0,33	11,00	13,72	14,49	78,32	74,03	70,09
Kloesgen0.2-CUT0.6	0,25	0,31	0,38	12,70	14,25	15,70	77,78	73,63	69,92
MEstimate-CUT0.6	0,26	0,47	0,54	7,79	8,89	9,19	77,98	73,16	69,61
MEstimate 0.5-CUT0.6	0,33	0,83	1,02	18,02	25,23	26,61	77,35	71,82	67,93
MEstimate 13.97-CUT0.6	0,30	0,52	0,59	9,25	10,88	11,30	78,38	73,98	70,13
Correlation-CUT0.6	0,33	0,49	0,49	8,44	10,39	11,11	77,74	73,03	68,56

Anhang B - 2: Ergebnisse Covering und Prepruning mit CutOff-Kriterium, CutOff-Parameter 0.3 und 0.6

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-CUT0.9	0,18	0,37	0,26	14,37	18,47	16,72	76,55	72,24	68,72
Precision-CUT0.9	0,21	0,41	0,31	17,09	21,89	20,39	76,47	71,67	68,11
Accuracy-CUT0.9	0,72	1,27	0,66	22,11	25,18	22,09	76,95	71,73	67,32
WRAcc-CUT0.9	0,20	0,39	0,46	3,88	3,86	3,35	75,86	71,68	68,11
Kloesgen-CUT0.9	0,24	0,31	0,34	9,65	11,84	12,00	77,66	73,21	69,61
Kloesgen0.3-CUT0.9	0,26	0,33	0,33	11,00	13,72	14,49	78,32	74,03	70,09
Kloesgen0.2-CUT0.9	0,25	0,31	0,38	12,70	14,25	15,70	77,78	73,67	69,92
MEstimate-CUT0.9	0,20	0,30	0,29	7,60	8,63	8,47	76,92	72,76	69,76
MEstimate 0.5-CUT0.9	0,20	0,37	0,28	15,60	18,89	17,86	76,88	72,22	68,65
MEstimate 13.97-CUT0.9	0,19	0,31	0,29	8,37	10,39	10,37	77,40	73,28	69,83
Correlation-CUT0.9	0,18	0,32	0,32	7,74	9,09	10,19	76,83	72,22	68,53
Laplace-Sig90	0,90	2,47	1,95	28,54	43,70	41,30	79,38	72,21	66,84
Precision-Sig90	1,03	2,89	2,21	33,93	53,98	47,77	78,81	71,77	66,14
Accuracy-Sig90	1,04	1,63	2,15	25,81	28,56	30,58	79,01	73,87	68,86
WRAcc-Sig90	0,54	0,81	0,93	4,25	4,46	4,39	79,16	74,93	70,25
Kloesgen-Sig90	0,84	2,09	1,57	18,77	30,32	28,84	81,52	75,59	69,93
Kloesgen0.3-Sig90	1,05	2,15	3,05	25,60	35,19	40,05	80,68	75,10	69,61
Kloesgen0.2-Sig90	1,06	2,17	2,67	28,63	37,98	42,23	80,40	74,30	68,43
MEstimate-Sig90	0,80	1,20	1,48	13,25	16,46	18,49	82,08	75,56	70,95
MEstimate 0.5-Sig90	0,99	2,64	2,12	31,26	46,56	47,23	79,24	72,27	66,74
MEstimate 13.97-Sig90	0,79	1,31	1,55	15,30	20,54	22,72	81,90	76,05	70,37
Correlation-Sig90	0,82	1,19	1,46	14,30	18,72	22,07	80,42	75,69	70,00

Anhang B - 3: Ergebnisse Covering, Prepruning mit SignificanceTesting, SignifikanzLevel 90% und Prepruning mit CutOff-Kriterium, CutOff-Parameter 0.9

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-Sig95	0,83	1,40	1,82	27,28	33,46	37,32	78,82	72,09	66,65
Precision-Sig95	0,92	1,67	1,98	31,04	42,96	44,54	78,14	71,31	66,19
Accuracy-Sig95	0,92	1,41	0,95	23,51	23,91	22,51	78,96	74,11	69,30
WRAcc-Sig95	0,53	0,79	0,93	4,26	4,40	4,37	79,13	74,88	70,16
Kloesgen-Sig95	0,79	2,01	1,36	17,30	27,84	25,37	81,66	75,67	70,44
Kloesgen0.3-Sig95	0,95	2,01	1,24	22,60	31,46	26,88	80,87	74,94	69,85
Kloesgen0.2-Sig95	1,00	1,99	1,28	27,12	33,84	32,72	80,52	74,06	68,89
MEstimate-Sig95	0,80	1,18	1,44	13,14	16,26	18,11	82,03	75,51	70,84
MEstimate 0.5-Sig95	0,88	1,57	2,03	28,32	38,63	44,09	78,75	71,89	66,84
MEstimate 13.97-Sig95	0,77	1,28	1,47	14,96	19,96	21,32	81,84	76,04	69,98
Correlation-Sig95	0,78	1,11	1,44	13,96	16,91	21,82	80,35	75,76	70,77
Laplace-Sig99	0,63	1,12	1,42	23,09	28,28	30,70	78,26	71,62	66,70
Precision-Sig99	0,75	1,36	1,83	27,42	35,75	40,65	77,79	71,15	66,18
Accuracy-Sig99	0,61	0,69	0,77	16,11	17,49	17,58	78,57	73,97	69,28
WRAcc-Sig99	0,53	0,80	0,93	4,26	4,35	4,30	78,99	74,69	70,13
Kloesgen-Sig99	0,72	1,05	1,17	15,40	19,19	20,89	81,46	75,96	70,81
Kloesgen0.3-Sig99	0,80	0,92	0,98	19,84	21,23	20,65	80,76	75,17	70,21
Kloesgen0.2-Sig99	0,81	0,85	1,02	22,88	22,42	26,70	79,96	74,07	68,91
MEstimate-Sig99	0,79	1,14	1,39	12,84	15,35	16,86	81,88	75,31	70,65
MEstimate 0.5-Sig99	0,72	1,34	1,70	25,02	33,23	36,82	78,36	71,74	66,73
MEstimate 13.97-Sig99	0,74	1,18	1,38	14,49	18,32	19,74	81,80	75,77	70,32
Correlation-Sig99	0,74	1,00	1,15	12,65	15,28	17,53	80,29	75,80	70,42

Anhang B - 4: Ergebnisse Covering und Prepruning mit SignificanceTesting, SignifikanzLevel 99% und 95%

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Max	0,34	0,43	0,48	16,75	18,63	19,98	77,51	72,48	68,20
Precision-FIND-Max	0,53	0,71	0,78	25,49	30,63	33,74	76,82	70,40	65,21
Accuracy-FIND-Max	0,46	0,51	0,54	12,79	11,60	10,81	76,69	72,00	67,53
WRAcc-FIND-Max	0,46	0,67	0,75	3,89	4,12	4,09	77,34	72,36	68,33
Kloesgen-FIND-Max	0,48	0,66	0,82	11,89	14,74	16,47	79,35	74,46	69,05
Kloesgen0.3-FIND-Max	0,51	0,61	0,81	16,75	17,28	20,68	79,10	73,40	68,83
Kloesgen0.2-FIND-Max	0,53	0,64	0,85	19,84	22,14	26,39	78,55	73,19	67,76
MEstimate-FIND-Max	0,49	0,72	0,79	8,37	10,12	10,19	79,45	73,85	69,25
MEstimate 0.5-FIND-Max	0,45	0,62	0,67	21,16	25,53	28,18	77,52	71,47	66,36
MEstimate 13.97-FIND-Max	0,49	0,68	0,80	10,04	11,05	12,05	79,38	73,57	68,67
Correlation-FIND-Max	0,52	0,76	0,79	9,74	12,89	12,95	78,71	73,48	69,17
Laplace-DEL-Max	0,35	0,45	0,53	16,30	19,74	21,68	76,95	71,73	67,22
Precision-DEL-Max	0,37	0,49	0,52	19,26	23,39	24,18	76,14	70,43	65,70
Accuracy-DEL-Max	0,48	0,54	0,59	13,00	12,72	12,49	77,11	72,37	67,72
WRAcc-DEL-Max	0,49	0,67	0,80	4,16	4,28	4,35	77,54	72,29	68,40
Kloesgen-DEL-Max	0,56	0,68	0,86	12,47	14,96	16,89	79,27	74,10	68,81
Kloesgen0.3-DEL-Max	0,50	0,68	0,78	15,14	18,09	18,05	78,23	73,47	68,95
Kloesgen0.2-DEL-Max	0,46	0,60	0,68	16,63	19,39	21,65	77,58	72,86	68,05
MEstimate-DEL-Max	0,50	0,70	0,88	8,37	9,77	11,40	79,36	73,81	69,57
MEstimate 0.5-DEL-Max	0,38	0,47	0,55	18,63	21,54	21,79	76,82	71,12	66,32
MEstimate 13.97-DEL-Max	0,52	0,72	0,76	9,96	11,40	11,58	79,56	74,19	69,11
Correlation-DEL-Max	0,54	0,74	0,81	9,72	12,74	12,25	78,87	73,35	68,89

Anhang B - 5: Ergebnisse Covering und IREP mit MaximumErrorRate

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Cut0.3	0,29	0,46	0,45	4,25	4,95	4,84	73,41	69,17	64,76
Precision-FIND-Cut0.3	0,26	0,36	0,39	3,84	3,56	3,67	70,11	64,63	60,15
Accuracy-FIND-Cut0.3	0,45	0,60	0,65	3,61	3,25	3,12	76,03	71,77	68,17
WRAcc-FIND-Cut0.3	0,44	0,64	0,69	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen-FIND-Cut0.3	0,29	0,43	0,43	0,47	0,33	0,23	62,49	58,95	54,95
Kloesgen0.3-FIND-Cut0.3	0,27	0,43	0,43	1,72	1,54	1,00	70,12	65,45	59,99
Kloesgen0.2-FIND-Cut0.3	0,29	0,44	0,47	3,39	3,02	2,96	73,21	68,44	62,86
MEstimate-FIND-Cut0.3	0,56	0,83	0,91	3,72	3,81	3,65	76,99	72,35	69,32
MEstimate 0.5-FIND-Cut0.3	0,28	0,40	0,42	4,26	4,49	4,11	71,64	66,99	62,09
MEstimate 13.97-FIND-Cut0.3	0,56	0,79	0,90	4,23	4,18	4,18	77,79	73,36	69,79
Correlation-FIND-Cut0.3	0,63	0,77	0,82	4,04	3,51	3,44	78,83	74,13	70,48
Laplace-DEL-Cut0.3	0,29	0,43	0,43	3,82	3,96	4,30	72,63	68,42	64,49
Precision-DEL-Cut0.3	0,26	0,36	0,40	3,34	3,98	3,18	69,38	64,07	59,88
Accuracy-DEL-Cut0.3	0,46	0,60	0,71	3,59	3,63	3,60	76,03	72,23	68,18
WRAcc-DEL-Cut0.3	0,44	0,64	0,69	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen-DEL-Cut0.3	0,30	0,44	0,44	0,50	0,32	0,21	62,38	58,91	55,07
Kloesgen0.3-DEL-Cut0.3	0,29	0,47	0,46	1,68	1,42	1,11	69,07	65,10	59,98
Kloesgen0.2-DEL-Cut0.3	0,31	0,44	0,50	2,95	2,37	2,32	72,10	67,27	62,32
MEstimate-DEL-Cut0.3	0,62	0,84	0,89	4,04	3,70	3,56	76,70	72,65	68,95
MEstimate 0.5-DEL-Cut0.3	0,29	0,39	0,40	3,84	3,75	3,61	71,10	66,57	61,92
MEstimate 13.97-DEL-Cut0.3	0,60	0,82	0,96	4,50	4,18	4,28	77,32	73,35	69,33
Correlation-DEL-Cut0.3	0,69	0,78	0,92	4,14	3,51	3,44	78,76	74,65	70,45

Anhang B - 6: Ergebnisse Covering und IREP mit CutOff-Kriterium, CutOff-Parameter 0.6

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Cut0.6	0,27	0,44	0,42	4,00	4,74	4,54	73,07	69,00	64,29
Precision-FIND-Cut0.6	0,25	0,35	0,39	3,79	3,46	3,65	69,89	64,42	59,81
Accuracy-FIND-Cut0.6	0,45	0,60	0,66	3,61	3,25	3,11	76,01	71,77	68,17
WRAcc-FIND-Cut0.6	0,44	0,64	0,69	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen-FIND-Cut0.6	0,26	0,39	0,42	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen0.3-FIND-Cut0.6	0,22	0,33	0,39	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen0.2-FIND-Cut0.6	0,18	0,28	0,31	0,00	0,00	0,00	54,29	52,88	51,49
MEstimate-FIND-Cut0.6	0,38	0,52	0,57	1,30	1,07	1,23	68,55	64,89	61,79
MEstimate 0.5-FIND-Cut0.6	0,26	0,39	0,42	4,05	4,30	4,11	71,23	66,81	61,83
MEstimate 13.97-FIND-Cut0.6	0,36	0,49	0,59	1,74	1,42	1,65	69,85	66,08	63,11
Correlation-FIND-Cut0.6	0,47	0,62	0,66	2,12	1,68	1,44	71,89	66,23	61,69
Laplace-DEL-Cut0.6	0,28	0,42	0,42	3,75	3,77	4,21	72,47	68,24	64,24
Precision-DEL-Cut0.6	0,26	0,35	0,38	3,18	3,00	3,00	69,22	63,67	59,47
Accuracy-DEL-Cut0.6	0,45	0,61	0,71	3,59	3,63	3,60	76,03	72,23	68,18
WRAcc-DEL-Cut0.6	0,44	0,64	0,69	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen-DEL-Cut0.6	0,23	0,39	0,44	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen0.3-DEL-Cut0.6	0,23	0,35	0,39	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen0.2-DEL-Cut0.6	0,20	0,31	0,33	0,00	0,00	0,00	53,90	52,88	51,49
MEstimate-DEL-Cut0.6	0,39	0,54	0,58	1,34	1,14	1,16	68,29	65,03	61,93
MEstimate 0.5-DEL-Cut0.6	0,27	0,36	0,39	3,64	3,54	3,47	70,80	66,29	61,62
MEstimate 13.97-DEL-Cut0.6	0,39	0,53	0,62	1,89	1,58	1,75	69,63	66,36	63,04
Correlation-DEL-Cut0.6	0,48	0,63	0,67	2,20	1,70	1,49	71,91	66,82	61,75

Anhang B - 7: Ergebnisse Covering und IREP mit CutOff-Kriterium, CutOff-Parameter 0.6

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Cut0.9	0,18	0,24	0,24	1,68	1,21	0,82	65,55	59,62	54,87
Precision-FIND-Cut0.9	0,21	0,29	0,28	3,21	2,75	2,44	68,52	60,70	56,20
Accuracy-FIND-Cut0.9	0,33	0,48	0,52	1,46	1,33	1,07	64,02	60,13	56,50
WRAcc-FIND-Cut0.9	0,44	0,64	0,69	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen-FIND-Cut0.9	0,27	0,39	0,42	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen0.3-FIND-Cut0.9	0,22	0,34	0,39	0,00	0,00	0,00	54,29	52,88	51,49
Kloesgen0.2-FIND-Cut0.9	0,18	0,29	0,31	0,00	0,00	0,00	54,29	52,88	51,49
MEstimate-FIND-Cut0.9	0,30	0,41	0,46	0,07	0,04	0,00	55,50	53,56	51,50
MEstimate 0.5-FIND-Cut0.9	0,18	0,27	0,25	2,49	2,28	1,82	68,02	60,94	56,31
MEstimate 13.97-FIND-Cut0.9	0,28	0,41	0,43	0,14	0,09	0,00	56,37	53,84	51,53
Correlation-FIND-Cut0.9	0,37	0,51	0,49	0,61	0,37	0,05	59,95	55,45	52,01
Laplace-DEL-Cut0.9	0,18	0,24	0,23	1,41	0,95	0,72	63,98	58,21	54,52
Precision-DEL-Cut0.9	0,23	0,27	0,28	2,59	2,14	2,18	66,75	59,47	55,61
Accuracy-DEL-Cut0.9	0,33	0,48	0,53	1,43	1,49	1,04	63,98	60,36	56,54
WRAcc-DEL-Cut0.9	0,44	0,39	0,69	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen-DEL-Cut0.9	0,27	0,39	0,38	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen0.3-DEL-Cut0.9	0,23	0,35	0,32	0,00	0,00	0,00	53,90	52,88	51,49
Kloesgen0.2-DEL-Cut0.9	0,20	0,31	0,45	0,00	0,00	0,00	53,90	52,88	51,49
MEstimate-DEL-Cut0.9	0,30	0,41	0,45	0,07	0,04	0,00	55,13	53,56	51,49
MEstimate 0.5-DEL-Cut0.9	0,20	0,25	0,24	2,04	1,58	1,35	66,18	59,70	55,55
MEstimate 13.97-DEL-Cut0.9	0,28	0,40	0,42	0,16	0,07	0,00	55,85	53,81	51,51
Correlation-DEL-Cut0.9	0,38	0,52	0,48	0,63	0,42	0,09	59,53	55,60	52,17

Anhang B - 8: Ergebnisse Covering und IREP mit CutOff-Kriterium, CutOff-Parameter 0.9

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Sig90	0,29	0,46	0,45	4,25	4,91	4,84	73,41	69,14	64,62
Precision-FIND-Sig90	0,26	0,36	0,39	3,84	3,53	3,67	70,08	64,55	59,86
Accuracy-FIND-Sig90	0,45	0,60	0,65	3,61	3,23	3,07	76,03	71,76	68,14
WRAcc-FIND-Sig90	0,69	1,24	1,39	2,88	2,67	2,79	76,97	72,92	69,13
Kloesgen-FIND-Sig90	0,51	0,71	0,77	4,26	4,40	4,40	78,81	74,82	70,47
Kloesgen0.3-FIND-Sig90	0,40	0,59	0,63	4,14	4,37	3,93	77,29	73,24	68,27
Kloesgen0.2-FIND-Sig90	0,33	0,50	0,58	4,30	4,09	4,23	75,15	73,24	66,22
MEstimate-FIND-Sig90	0,70	1,01	1,12	5,11	5,02	4,79	78,92	74,47	70,74
MEstimate 0.5-FIND-Sig90	0,28	0,40	0,42	4,26	4,46	4,11	71,61	67,00	61,87
MEstimate 13.97-FIND-Sig90	0,59	0,85	1,00	4,91	4,91	4,81	78,74	74,10	70,53
Correlation-FIND-Sig90	0,63	0,81	0,87	4,25	3,89	3,77	78,79	74,36	70,83
Laplace-DEL-Sig90	0,28	0,43	0,44	3,93	3,96	4,30	72,99	68,42	64,49
Precision-DEL-Sig90	0,27	0,36	0,40	3,46	3,09	3,18	69,80	64,07	59,88
Accuracy-DEL-Sig90	0,45	0,61	0,71	3,65	3,63	3,60	76,34	72,23	68,18
WRAcc-DEL-Sig90	0,76	1,33	1,49	3,12	2,86	2,88	77,23	72,88	69,29
Kloesgen-DEL-Sig90	0,57	0,80	0,82	4,54	4,40	4,19	78,62	74,24	70,32
Kloesgen0.3-DEL-Sig90	0,44	0,61	0,65	4,37	3,89	3,53	76,82	72,79	68,24
Kloesgen0.2-DEL-Sig90	0,36	0,53	0,57	4,05	3,51	3,44	74,81	70,85	66,28
MEstimate-DEL-Sig90	0,77	1,06	1,15	5,25	5,12	4,88	79,11	74,85	70,40
MEstimate 0.5-DEL-Sig90	0,28	0,39	0,40	3,95	3,75	3,61	71,49	66,57	61,92
MEstimate 13.97-DEL-Sig90	0,67	0,91	1,01	5,30	5,11	4,81	78,79	74,57	70,12
Correlation-DEL-Sig90	0,71	0,86	1,01	4,32	4,23	3,93	79,11	74,83	70,97

Anhang B - 9: Ergebnisse Covering und IREP mit SignificanceTesting, Signifikanzlevel 90%

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Sig95	0,29	0,46	0,44	4,21	4,91	4,82	73,28	69,15	64,49
Precision-FIND-Sig95	0,25	0,36	0,39	3,81	3,53	3,61	69,90	64,43	59,63
Accuracy-FIND-Sig95	0,45	0,60	0,65	3,60	3,23	3,07	76,00	71,77	68,09
WRAcc-FIND-Sig95	0,69	1,24	1,39	2,86	2,67	2,79	77,10	73,04	69,14
Kloesgen-FIND-Sig95	0,51	0,72	0,77	4,23	4,40	4,40	78,76	74,79	70,34
Kloesgen0.3-FIND-Sig95	0,40	0,58	0,63	4,11	4,37	3,93	77,23	73,22	68,21
Kloesgen0.2-FIND-Sig95	0,34	0,50	0,58	4,26	4,09	4,16	75,12	71,11	66,20
MEstimate-FIND-Sig95	0,70	1,01	1,12	5,09	5,02	4,79	78,74	74,41	70,69
MEstimate 0.5-FIND-Sig95	0,27	0,40	0,42	4,19	4,40	4,05	71,56	66,95	61,76
MEstimate 13.97-FIND-Sig95	0,59	0,85	1,01	4,89	4,91	4,81	78,57	74,07	70,43
Correlation-FIND-Sig95	0,63	0,80	0,87	4,23	3,89	3,77	78,72	74,33	70,73
Laplace-DEL-Sig95	0,28	0,44	0,44	3,93	3,96	4,30	72,99	68,42	64,49
Precision-DEL-Sig95	0,26	0,36	0,40	3,46	3,09	3,18	69,80	64,07	59,88
Accuracy-DEL-Sig95	0,45	0,61	0,71	3,65	3,63	3,60	76,34	72,23	68,18
WRAcc-DEL-Sig95	0,75	1,34	1,49	3,12	2,86	2,88	77,23	72,88	68,29
Kloesgen-DEL-Sig95	0,57	0,80	0,82	4,54	4,40	4,19	78,62	74,24	70,32
Kloesgen0.3-DEL-Sig95	0,44	0,60	0,65	4,37	3,89	3,53	76,82	72,79	68,24
Kloesgen0.2-DEL-Sig95	0,35	0,53	0,57	4,05	3,51	3,44	74,81	70,85	66,28
MEstimate-DEL-Sig95	0,77	1,06	1,15	5,25	5,12	4,88	79,11	74,85	70,40
MEstimate 0.5-DEL-Sig95	0,28	0,39	0,40	3,95	3,75	3,61	71,49	66,57	61,92
MEstimate 13.97-DEL-Sig95	0,68	0,91	1,01	5,30	5,11	4,81	78,79	74,57	70,12
Correlation-DEL-Sig95	0,71	0,85	1,01	4,32	4,23	3,93	79,11	74,83	70,97

Anhang B - 10: Ergebnisse Covering und IREP mit SignificanceTesting, Signifikanzlevel 95%

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-Sig99	0,28	0,45	0,43	4,18	4,81	4,61	72,93	68,76	64,26
Precision-FIND-Sig99	0,25	0,35	0,37	3,65	3,39	3,28	69,36	63,56	59,01
Accuracy-FIND-Sig99	0,44	0,58	0,64	3,44	3,11	2,96	75,40	71,15	67,85
WRAcc-FIND-Sig99	0,69	1,24	1,38	2,79	2,61	2,68	76,45	72,49	68,83
Kloesgen-FIND-Sig99	0,50	0,71	0,76	4,19	4,37	4,26	78,35	74,46	70,24
Kloesgen0.3-FIND-Sig99	0,39	0,57	0,62	4,02	4,16	3,74	76,91	72,87	67,93
Kloesgen0.2-FIND-Sig99	0,33	0,49	0,57	4,19	3,84	3,96	74,75	70,65	65,68
MEstimate-FIND-Sig99	0,70	1,01	1,11	5,02	4,93	4,74	78,19	73,93	70,21
MEstimate 0.5-FIND-Sig99	0,27	0,40	0,38	4,12	4,25	3,42	71,18	66,44	60,95
MEstimate 13.97-FIND-Sig99	0,59	0,85	0,99	4,86	4,88	4,70	78,02	73,89	69,88
Correlation-FIND-Sig99	0,63	0,80	0,86	4,21	3,86	3,72	78,25	73,96	70,54
Laplace-DEL-Sig99	0,28	0,43	0,44	3,93	3,96	4,30	72,99	68,42	64,49
Precision-DEL-Sig99	0,26	0,36	0,40	3,46	3,09	3,18	69,80	64,07	59,88
Accuracy-DEL-Sig99	0,45	0,61	0,71	3,65	3,63	3,60	76,34	72,23	68,18
WRAcc-DEL-Sig99	0,76	1,34	1,49	3,12	2,86	2,88	77,23	72,88	69,29
Kloesgen-DEL-Sig99	0,57	0,80	0,82	4,54	4,40	4,19	78,62	74,24	70,32
Kloesgen0.3-DEL-Sig99	0,44	0,61	0,65	4,37	3,89	3,53	76,82	72,79	68,24
Kloesgen0.2-DEL-Sig99	0,36	0,53	0,57	4,05	3,51	3,44	74,81	70,85	66,28
MEstimate-DEL-Sig99	0,77	1,07	1,15	5,25	5,12	4,88	79,11	74,85	70,40
MEstimate 0.5-DEL-Sig99	0,28	0,39	0,40	3,95	3,75	3,61	71,49	66,57	61,92
MEstimate 13.97-DEL-Sig99	0,67	0,92	1,01	5,30	5,11	4,81	78,79	74,57	70,12
Correlation-DEL-Sig99	0,71	0,85	1,01	4,32	4,32	3,93	79,11	74,83	70,97

Anhang B - 11: Ergebnisse Covering und IREP mit SignificanceTesting, Signifikanzlevel 99%

	Zeitverhalten in sec			*Hypothesengröße in |R|*			*Genauigkeit in %*		
Konfiguration	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***	***0%-Paket***	***5%-Paket***	***10%-Paket***
Laplace	1,45	3,47	5,54	41,07	60,72	78,51	79,43	72,67	66,57
Precision	1,60	3,92	6,06	45,82	70,95	88,60	79,13	72,57	66,36
Accuracy	1,47	2,42	3,46	35,37	42,44	54,79	78,78	73,36	67,50
WRAcc	0,57	0,87	1,02	4,25	4,46	4,39	79,16	74,93	70,27
Kloesgen	0,98	2,21	1,81	15,32	30,60	31,58	81,47	75,39	69,62
Kloesgen0.3	1,19	2,42	3,52	29,07	40,07	49,04	80,70	74,82	68,88
Kloesgen0.2	1,34	2,50	3,78	35,44	45,19	59,04	80,17	74,06	67,73
MEstimate	0,88	1,29	1,60	13,25	16,46	18,53	82,08	75,41	70,85
MEstimate 0.5	1,41	3,60	5,77	42,30	64,33	83,09	79,20	72,95	66,84
MEstimate 13.97	0,80	1,31	1,58	15,30	20,54	22,75	81,91	75,88	70,36
Correlation	0,89	1,28	1,60	14,42	18,72	22,07	80,46	75,52	69,97
Laplace-FIND-MDL	0,19	0,27	0,27	0,91	0,84	0,53	60,55	57,44	53,83
Precision-FIND-MDL	0,16	0,25	0,25	0,65	0,53	0,40	59,86	55,42	53,40
Accuracy-FIND-MDL	0,34	0,47	0,53	1,28	1,12	1,07	68,20	63,77	60,63
WRAcc-FIND-MDL	0,56	1,04	1,18	1,86	1,67	1,39	71,12	66,73	63,02
Kloesgen-FIND-MDL	0,32	0,52	0,57	1,44	1,32	1,21	66,88	63,08	59,71
Kloesgen0.3-FIND-MDL	0,25	0,40	0,46	1,04	1,12	0,79	64,35	61,20	58,11
Kloesgen0.2-FIND-MDL	0,21	0,36	0,39	1,00	0,93	0,61	63,14	59,31	56,53
MEstimate-FIND-MDL	0,37	0,57	0,66	1,44	1,33	1,11	66,24	62,72	59,31
MEstimate 0.5-FIND-MDL	0,17	0,27	0,27	0,75	0,68	0,47	60,27	56,24	53,41
MEstimate 13.97-FIND-MDL	0,35	0,55	0,63	1,30	1,23	1,09	65,14	61,25	58,64
Correlation-FIND-MDL	0,45	0,62	0,68	1,70	1,63	1,44	70,12	65,80	62,76
Laplace-DEL-MDL	0,20	0,30	0,28	0,93	0,79	0,58	61,46	57,88	54,44
Precision-DEL-MDL	0,18	0,27	0,29	0,74	0,58	0,49	60,72	56,29	53,76
Accuracy-DEL-MDL	0,33	0,46	0,53	1,25	1,04	0,96	67,73	63,41	60,41
WRAcc-DEL-MDL	0,57	1,08	1,23	1,84	1,68	1,37	70,69	66,21	62,68
Kloesgen-DEL-MDL	0,32	0,52	0,56	1,40	1,32	1,11	66,69	63,51	59,75
Kloesgen0.3-DEL-MDL	0,26	0,43	0,47	1,07	1,11	0,86	64,96	61,62	58,54
Kloesgen0.2-DEL-MDL	0,24	0,39	0,43	0,98	0,86	0,74	64,00	60,37	57,04
MEstimate-DEL-MDL	0,36	0,58	0,64	1,40	1,32	1,05	65,82	62,30	59,21
MEstimate 0.5-DEL-MDL	0,19	0,29	0,30	0,82	0,70	0,56	61,20	56,96	54,28
MEstimate 13.97-DEL-MDL	0,34	0,53	0,95	1,28	1,21	1,53	65,10	61,26	58,60
Correlation-DEL-MDL	0,44	0,60	0,67	1,70	1,56	1,35	69,32	65,37	61,86

Anhang B - 12: Ergebnisse Covering und IREP mit MDL-Kriterium

Anhang C – API-Dokumentation

In diesem Anhang findet sich die Dokumentation der API. In Tabelle Anhang C-1 sind die Pakete, deren Autoren und eine kurze Beschreibung der Pakete zu finden.

Paket	***Autor***	***Beschreibung***
seco	J. Fürnkranz, B.Werling	Enthält statistische Klassen für Regeln und Regelmengen
seco.heuristics	J. Fürnkranz, B.Werling	Enthält die Suchheuristiken
seco.learners	J. Fürnkranz	Enthält die Lernaglorithmen
seco.models	J. Fürnkranz	Enthält das Modell der Hypothesensprache
seco.pruning	B.Werling	Enthält die implementierten Pruningmechanismen
seco.pruning.criterion	B.Werling	Enthält die Stopkriterien
seco.pruning.model	B.Werling	Enthält das Modell der Pruningmechanismen
seco.pruning.operator	B.Werling	Enthält die Pruninhgoperatoren

Anhang C - 1: Übersicht über die Pakete

Paketverzeichnis

Klassen- und Interfaceverzeichnis

C1 – Paket „seco“

Klassen	*Beschreibung*
RuleSetStats	This class counts the statistics (tp – TruePositives, fp – FalsePositives, fn – FalseNegatives, tn – TrueNegatives) for one ruleset for a given class.
TwoClassStats	Encapsulates performance functions for two-class problems.

Anhang C - 2: Paketinhalte des Pakets "seco"

Class / Interface 1: RuleSetStats

public class **RuleSetStats**
extends java.lang.Object
implements java.io.Serializable
This class counts the statistics
tp - TruePositives
fp - FalsePositives
fn - FalseNegatives
tn - TrueNegatives
for one ruleset for a given class.

Constructor Detail

RuleSetStats
public **RuleSetStats**()

Method Detail

updateCounts
public void **updateCounts**(RuleSet rs,
weka.core.Instances data,
double currClass)
Updates the counts for a 2-class problem for the ruleset.
Parameters:
rs - The ruleset to calculate the statistics for.
data - The data on which the statistics are calculated
currClass - The current class which is learned. The examples of this class count as tp.

---

toTwoClassStats
public TwoClassStats **toTwoClassStats**()
This method returns a TwoClassStats Object with the tp,fp,fn,tn of the ruleset
Returns:
a TwoClassStats object with the tp,fp,fn,tn of the ruleset

---

toString
public java.lang.String **toString**()
Overrides:
toString in class java.lang.Object

Class / Interface 2: TwoClassStats

public class **TwoClassStats**
extends java.lang.Object
implements java.lang.Cloneable, java.io.Serializable
Encapsulates performance functions for two-class problems.

Constructor Detail

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

TwoClassStats
public **TwoClassStats**(double tp,
double fp,
double tn,
double fn)
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

Method Detail

clone
public java.lang.Object **clone**()
Overrides:
clone in class java.lang.Object
Returns:
a deep copy of the statistics

---

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

---

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

---

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

---

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

---

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

---

incTruePositive
public double **incTruePositive**(double tp)

---

incTrueNegative
public double **incTrueNegative**()
Increments the number of true negatives

---

incTrueNegative
public double **incTrueNegative**(double tn)

---

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

---

incFalsePositive
public double **incFalsePositive**(double fp)

---

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

---

incFalseNegative
public double **incFalseNegative**(double fn)

---

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

---

getFalsePositive
public double **getFalsePositive**()
Gets the number of negative instances predicted as positive

---

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

---

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

---

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

---

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

---

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

---

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

---

getPredictedPositive
public double **getPredictedPositive**()

Gets the total number of examples predicted positive

---

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

---

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

---

getErrorRate
public double **getErrorRate**()
Calculate the error rate This is defined as
incorrectly classified examples
---
total examples

Returns:
the error rate

---

getAccuracy
public double **getAccuracy**()
Calculate the accuracy This is defined as
correctly classified examples
---
total examples

Returns:
the accuracy

---

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

Returns:
the prior

---

getTruePositiveRate
public double **getTruePositiveRate**()
Calculate the true positive rate. This is defined as
correctly classified positives
---
total positives

Returns:
the true positive rate

---

getFalsePositiveRate
public double **getFalsePositiveRate**()
Calculate the false positive rate. This is defined as
incorrectly classified negatives
---

total negatives

Returns:
the false positive rate

---

getPrecision
public double **getPrecision**()
Calculate the precision. This is defined as
correctly classified positives
---
total predicted as positive

Returns:
the precision

---

getRecall
public double **getRecall**()
Calculate the recall. This is defined as
correctly classified positives
---
total positives

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

---

getFMeasure
public double **getFMeasure**()
Calculate the F-Measure. This is defined as
2 * recall * precision
---
recall + precision

Returns:
the F-Measure

---

getFallout
public double **getFallout**()
Calculate the fallout. This is defined as
incorrectly classified negatives
---
total predicted as positive

Returns:
the fallout

---

getConfusionMatrix
public weka.classifiers.evaluation.ConfusionMatrix **getConfusionMatrix**()
Generates a ConfusionMatrix representing the current two-class statistics, using class names "negative" and "positive".
Returns:
a ConfusionMatrix.

---

toString
public java.lang.String **toString**()

Returns a string containing the various performance measures for the current object **Overrides:** toString in class java.lang.Object

C2 – Paket „seco.heuristics“

Klassen	***Beschreibung***
Accuracy	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Correlation	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Cost_Measure	The seco package implements generic functionality for simple separate-and-conquer rule learning.
F_Measure	The seco package implements generic functionality for simple separate-and-conquer rule learning.
FoilGain	The seco package implements generic functionality for simple separate-and-conquer rule learning.
GeneralizedM	The seco package implements generic functionality for simple separate-and-conquer rule learning.
J_Measure	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Kloesgen	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Laplace	The seco package implements generic functionality for simple separate-and-conquer rule learning.
LinearCosts	The seco package implements generic functionality for simple separate-and-conquer rule learning.
LinearRegression	The seco package implements generic functionality for simple separate-and-conquer rule learning.
LinearRegression_Testversion	The seco package implements generic functionality for simple separate-and-conquer rule learning.
MEstimate	The seco package implements generic functionality for simple separate-and-conquer rule learning.
MetaMultilayerPerceptron	The seco package implements generic functionality for simple separate-and-conquer rule learning.
MetaSVMreg	The seco package implements generic functionality for simple separate-and-conquer rule learning.
MPrecision	The seco package implements generic functionality for simple separate-and-conquer rule learning.
OptCandRuleMEstimate	The seco package implements generic functionality for simple separate-and-conquer rule learning.
OptCandRulePrec	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Precision	The seco package implements generic functionality for simple separate-and-conquer rule learning.
RateDiff	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Relative_Cost_Measure	The seco package implements generic functionality for simple separate-and-conquer rule learning.
RIPPERPrune	The seco package implements generic functionality for simple separate-and-conquer rule learning.
RuleMDL	The seco package implements generic functionality for simple separate-and-conquer rule learning.
SearchHeuristic	The seco package implements generic functionality for simple separate-and-conquer rule learning.
TileRateDiff	The seco package implements generic functionality for simple separate-and-conquer rule learning.
WRAcc	The seco package implements generic functionality for simple separate-and-conquer rule learning.120

Anhang C - 3: Paketinhalte des Pakets "seco.heuristics"

Class / Interface 3: Accuracy

public class **Accuracy**
extends SearchHeuristic
implements java.io.Serializable
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).

Constructor Detail

Accuracy
public **Accuracy**()
Constructor

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
return the accuracy of the rule
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 4: CombinedMetaMultilayerPerceptron

public class **CombinedMetaMultilayerPerceptron**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail

CombinedMetaMultilayerPerceptron
public **CombinedMetaMultilayerPerceptron**()
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

getOptions

public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

Class / Interface 5: Correlation

public class **Correlation**
extends SearchHeuristic
implements java.io.Serializable
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)

Constructor Detail

Correlation
public **Correlation**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with the Correlation estimate
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 6: Cost_Measure

public class **Cost_Measure**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail

Cost_Measure
public **Cost_Measure**()

Cost_Measure
public **Cost_Measure**(double c)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 7: F_Measure

public class **F_Measure**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable
The seco package implements generic functionality for simple separate-and-conquer rule learning.
This file implements a generic class for evaluating a rule with F-Measure.

Constructor Detail

F_Measure
public **F_Measure**()

F_Measure
public **F_Measure**(double n)

throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates the F-Measure of the rule
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 8: FoilGain

public class **FoilGain**
extends SearchHeuristic
implements java.io.Serializable
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.

Constructor Detail

FoilGain
public **FoilGain**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)

evaluates a rule with the Foil's information gain heuristic. Rules without predecessors (getPredecessor() == null) are evaluated with 0.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 9: GeneralizedM

public class **GeneralizedM**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable
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.

Constructor Detail
GeneralizedM
public **GeneralizedM**()
Empty constructor, c will be set to 0.5 and m to 2.0.

GeneralizedM
public **GeneralizedM**(double m,
double c)
throws java.lang.Exception
Constructor
Parameters:
m - the value for m (0 <= m <= NaN, default 1)
c - the cost factor (0 <= c <= 1, default 0.5)
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with the generalized m-estimate
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
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).
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options.
Specified by:
setOptions in interface weka.core.OptionHandler
Parameters:
options - the list of options as an array of strings
Throws:
java.lang.Exception - if an option is not supported

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 10: J_Measure

public class **J_Measure**
extends SearchHeuristic
implements java.io.Serializable
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 kloesgen-estimate. The default value of n is 2. It can be changed via setOptions (the class implements the OptionHandler interface).

Constructor Detail

J_Measure
public **J_Measure**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 11: Kloesgen

public class **Kloesgen**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable
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 kloesgen-estimate. The default value of n is 2. It can be changed via setOptions (the class implements the OptionHandler interface).

Constructor Detail

Kloesgen
public **Kloesgen**()

Kloesgen
public **Kloesgen**(double n)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
returns an enumeration of the available options, which are: -M number
The n-value used in the Kloesgen-heuristic (Default: 2.0).
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 12: Laplace

public class **Laplace**
extends SearchHeuristic
implements java.io.Serializable
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)

Constructor Detail

Laplace
public **Laplace**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with the Laplace estimate
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 13: LineareCosts

public class **LinearCosts**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable
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).

Constructor Detail

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

LinearCosts
public **LinearCosts**(double c)
throws java.lang.Exception
Constructor.
Parameters:
c - the cost trade-off.
Throws:

java.lang.Exception - unless 0 <= c <= 1.

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with a linear cost metric
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
returns an enumeration of the available options. which are: -c number
The cost trade-off in the heuristic. 0 <= c <= 1 (Default: 0.5).
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options.
Specified by:
setOptions in interface weka.core.OptionHandler
Parameters:
options - the list of options as an array of strings
Throws:
java.lang.Exception - if an option is not supported

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 14: LinearRegression

public class **LinearRegression**
extends SearchHeuristic
implements java.io.Serializable

Constructor Detail

LinearRegression
public **LinearRegression**()

Method Detail

evaluateRule

public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 15: LinearRegression_Testversion

public class **LinearRegression_Testversion**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail

LinearRegression_Testversion
public **LinearRegression_Testversion**()

LinearRegression_Testversion
public **LinearRegression_Testversion**(double[] values)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

setValues
public void **setValues**(double[] values)
throws java.lang.Exception
Throws:
java.lang.Exception

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:

java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 16: MEstimate

public class **MEstimate**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable
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).

Constructor Detail

MEstimate
public **MEstimate**()
Empty constructor, m will be set to 1.

MEstimate
public **MEstimate**(double m)
throws java.lang.Exception
Constructor
Parameters:
m - the value for m
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with the m-estimate
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
returns an enumeration of the available options, which are: -M number
The m-value used in the m-heuristic (Default: 2).
Specified by:
listOptions in interface weka.core.OptionHandler

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options.
Specified by:
setOptions in interface weka.core.OptionHandler
Parameters:
options - the list of options as an array of strings
Throws:
java.lang.Exception - if an option is not supported

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 17: MetaMultilayerPerceptron

public class **MetaMultilayerPerceptron**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail

MetaMultilayerPerceptron
public **MetaMultilayerPerceptron**()

MetaMultilayerPerceptron
public **MetaMultilayerPerceptron**(java.lang.String ModelName)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 18: MetaSVMreg

public class **MetaSVMreg**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail
MetaSVMreg
public **MetaSVMreg**()

MetaSVMreg
public **MetaSVMreg**(java.lang.String ModelName)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

Class / Interface 19: MPrecision

public class **MPrecision**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail
MPrecision
public **MPrecision**()

MPrecision
public **MPrecision**(double c)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:

an array of strings suitable for passing to setOptions

Class / Interface 20: OptCandRuleMEstimate

public class **OptCandRuleMEstimate**
extends SearchHeuristic
implements java.io.Serializable
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)

Constructor Detail

OptCandRuleMEstimate
public **OptCandRuleMEstimate**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates the refinement of a rule by trading off the precision and WRA which both got their origin in the point (N,P) - that's because in this case refinements are evaluated with a Top-Down Strategy
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

listOptions
public java.util.Enumeration **listOptions**()
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)
throws java.lang.Exception
parse a list of options.
Parameters:
options - the list of options as an array of strings
Throws:
java.lang.Exception - if an option is not supported

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 21: OptCandRulePrec

public class **OptCandRulePrec**
extends SearchHeuristic
implements java.io.Serializable

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)

Constructor Detail

OptCandRulePrec
public **OptCandRulePrec**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates the precision of a candidate refinement; this works by maximizing the angle between the (P-p)-axis and the (N-n)-axis; thus, if a rule is successive specialized starting with the most general rule, the specialization which is located nearest to the (N-n)-axis is chosen
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 22: Precision

public class **Precision**
extends SearchHeuristic
implements java.io.Serializable
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)

Constructor Detail

Precision
public **Precision**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates the precision of the rule
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 23: RateDiff

public class **RateDiff**
extends SearchHeuristic
implements java.io.Serializable
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.

Constructor Detail

RateDiff
public **RateDiff**()
Constructor

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with weighted relative accuracy
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 24: Relative_Cost_Measure

public class **Relative_Cost_Measure**
extends SearchHeuristic
implements weka.core.OptionHandler, java.io.Serializable

Constructor Detail

Relative_Cost_Measure
public **Relative_Cost_Measure**()

Relative_Cost_Measure
public **Relative_Cost_Measure**(double c)
throws java.lang.Exception
Throws:
java.lang.Exception

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

setOptions
public void **setOptions**(java.lang.String[] options)

throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
get the current configuration
Specified by:
getOptions in interface weka.core.OptionHandler
Returns:
an array of strings suitable for passing to setOptions

Class / Interface 25: RIPPERPrune

Evaluates a rule with the Ripper pruning heuristic.

$$h(n,p) = \frac{p - n}{p + n}$$

Constructor Detail

RIPPERPrune
public **RIPPERPrune**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 26: RuleMDL

public class **RuleMDL**
extends SearchHeuristic
implements java.io.Serializable

MDL based Cost Measure based on Bernd Pfahringers MDL formula.

Constructor Detail

RuleMDL
public **RuleMDL**()

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)

Description copied from class: SearchHeuristic
computes the evaluation for a possible rule.
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 27: SearchHeuristic

public abstract class **SearchHeuristic**
extends java.lang.Object
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)

Constructor Detail

SearchHeuristic
public **SearchHeuristic**()

Method Detail

evaluateRule
public abstract double **evaluateRule**(CandidateRule r)
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)
throws java.lang.Exception
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.
Throws:
java.lang.Exception - if the classifier name is invalid, or the options supplied are not acceptable to the classifier

toOptionString
public java.lang.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 java.lang.String **toString**()
Overrides:
toString in class java.lang.Object
Returns:
a string representation of a search heuristic, including all parameters (if any). The parameters are added in parentheses, like a constructor call.

Class / Interface 28: TileRateDiff

public class **TileRateDiff**
extends SearchHeuristic
implements java.io.Serializable
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, multiplied by the size of the rules. This is motivated by Bart Goethals' idea of tiles as an interestingness measure for rules.

Constructor Detail

TileRateDiff
public **TileRateDiff**()
Constructor

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with weighted relative accuracy
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

Class / Interface 29: WRAcc

public class **WRAcc**
extends SearchHeuristic
implements java.io.Serializable
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).

Constructor Detail

WRAcc
public **WRAcc**()
Constructor

Method Detail

evaluateRule
public double **evaluateRule**(CandidateRule r)
evaluates a rule with weighted relative accuracy
Specified by:
evaluateRule in class SearchHeuristic
Parameters:
r - the candidate rule

C3 – Paket „seco.learners“

Klassen	*Beschreibung*
Covering	The seco package implements generic functionality for simple separate-and-conquer rule learning.
GreedyTopDown	The seco package implements generic functionality for simple separate-and-conquer rule learning.
TopDownBeamSearch	The seco package implements generic functionality for simple separate-and-conquer rule learning.

Anhang C - 4: Paketinhalte des Pakets "seco.learners"

Class / Interface 30: Covering

public class **Covering**
extends weka.classifiers.Classifier
implements weka.core.WeightedInstancesHandler, weka.core.AdditionalMeasureProducer, weka.core.OptionHandler, java.io.Serializable

The seco package implements generic functionality for simple separate-and-conquer rule learning. Covering implements the covering strategy. It relies on external methods for learning a single rule. The covering strategy learns one rule at a time, each time removing all examples covered by the rule from the training set. The current implementation only uses the simplest stopping criterion: Learning stops whenever the best rule found leads to a decrease in accuracy on the training set (i.e., when the best rule does not cover more positive than negative examples). The class is intended as an outer loop for separate-and-conquer rule learning algorithms, but in principle other classifiers could be used as internal classifiers as well (this only makes sense if the classifier is partial, i.e., if it returns the missing class value for some of the instances). Currently, however, it requires a method for returning the learned model, which is not supported by the general Classifier class. So for the moment, the internal classifier has to be of type Rule, and the learner needs to be of type TopDown BeamSearch. TODO: - maybe be more specific upon which classifier is valid as the single-rule learner (must be one that does not classify all examples). The internal learner currently must be TopDownBeamSearch. This should be a generic class (like a "PartialClassifier") that has certain methods defined.

Constructor Detail

Covering
public **Covering**()

Method Detail

getRuleLearner
public TopDownBeamSearch **getRuleLearner**()
Returns:
the current rule learner

---

setRuleLearner
public void **setRuleLearner**(TopDownBeamSearch l)
specify the learner that will be used for inducing single rules
Parameters:
l - the new rule learner (must be a Rule)

---

getRuleSet
public RuleSet **getRuleSet**()
Returns:
the learned set of rules

---

classifyInstance
public double **classifyInstance**(weka.core.Instance inst)
throws java.lang.Exception
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.
Overrides:
classifyInstance in class weka.classifiers.Classifier
Parameters:
inst - the instance
Returns:
the class of the instance or a missing value
Throws:
java.lang.Exception

---

buildClassifier
public void **buildClassifier**(weka.core.Instances instances)
throws java.lang.Exception
build a classifier using the covering strategy. The covering strategy calls a partial classifier on the training data, removes all examples that are covered by it from the dataset, and repeats until no progress is made (i.e., until no more rules are learned.
Specified by:
buildClassifier in class weka.classifiers.Classifier
Throws:
java.lang.Exception

---

toString
public java.lang.String **toString**()
Overrides:
toString in class java.lang.Object

---

listOptions
public java.util.Enumeration **listOptions**()
Returns an enumeration describing the available options. Valid options are:
-D
turn on the debug mode and produce a lot of output -L rule learner specify the internal rule learner (default: TopDownBeamSearch) --
option separator. All options after this will be passed to the rule learner.
Specified by:
listOptions in interface weka.core.OptionHandler
Overrides:
listOptions in class weka.classifiers.Classifier
Returns:
an enumeration of all available options

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options. The Options for Pruning must come before the Learneroptions
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class weka.classifiers.Classifier

Parameters:
opts - the list of options an array of strings
Throws:
Exeption - if an option is not supported
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
the current settings of the classifier
Specified by:
getOptions in interface weka.core.OptionHandler
Overrides:
getOptions in class weka.classifiers.Classifier
Returns:
an array of strings suitable for passing to setOptions

---

enumerateMeasures
public java.util.Enumeration **enumerateMeasures**()
Specified by:
enumerateMeasures in interface weka.core.AdditionalMeasureProducer
Returns:
an enumeration of the additional measure names

---

getMeasure
public double **getMeasure**(java.lang.String additionalMeasureName)
Returns the value of the named measure
Specified by:
getMeasure in interface weka.core.AdditionalMeasureProducer
Parameters:
measureName - the name of the measure to query for its value
Returns:
the value of the named measure
Throws:
java.lang.IllegalArgumentException - if the named measure is not supported

---

main
public static void **main**(java.lang.String[] args)
Main method.

---

getPruner
public PruningTemplate **getPruner**()
Returns:
the m_prune

---

setPruner
public void **setPruner**(PruningTemplate m_prune)
Parameters:
m_prune - the m_prune to set

Class / Interface 31: GreedyTopDown

public class **GreedyTopDown**
extends weka.classifiers.Classifier
implements weka.core.WeightedInstancesHandler, weka.core.AdditionalMeasureProducer, weka.core.OptionHandler
The seco package implements generic functionality for simple separate-and-conquer rule learning. GreedyTopDown implements the greedy top-down (general-to-specific) induction of a single rule. In principle, a Rule can be used as a classifier in its own right, but the main purpose of this is to provide a learner for the inner loop of the separate-and-conquer/covering loop. TODO: - Stopping heuristics - Filtering heuristics (at the place where we are now " ---> ignored") - maybe initialization with a given rule (be careful not to re-initialize the rule, init used vector) - beam search - local optimization (gain heuristics do not allow a global comparison between rules, such as Foil-Gain) - LEFs (multiple evaluation heuristics) - cleaner interface should maybe store a Rule (and not a CandidateRule) internally and copy it to a CandidateRule for learning - split value can be chosen from domain or interpolate between neighboring values - Maybe the search should maintain a general confusion matrix so that CN2 can implemented (both, the entropy heuristic and the decision list search that auto-picks the class) - have I covered missing class values? - the two code blocks for finding the best condition should probably be separate routines/methods of some class. Parts of it is based on code for JRip and for Prism.

Constructor Detail

GreedyTopDown
public **GreedyTopDown**()

Method Detail

initRule
public void **initRule**()
initialize the rule learner with a fresh, empty rule.

---

initRule
public void **initRule**(CandidateRule r)
initialize the rule learner with
Parameters:
rule - a new starting point for the refinement

---

getRule
public CandidateRule **getRule**()
Returns:
the learned rule

---

getHeuristic
public SearchHeuristic **getHeuristic**()
return the heuristic used for evaluating rules
Returns:
a seco.heuristics.SearchHeuristic

---

setHeuristic
public void **setHeuristic**(SearchHeuristic h)
set the heuristic used for evaluating rules
Parameters:
h - a seco.heuristics.SearchHeuristic

---

getTargetClass

public double **getTargetClass**()
Returns:
the index of the class for which a rule is learned

---

setTargetClass
public void **setTargetClass**(double i)
set the index of the class to a new value. it also resets the rule (initRule).

---

debug
public boolean **debug**()
Returns:
true if debugging is on, false if not

---

setDebug
public void **setDebug**(boolean d)
Overrides:
setDebug in class weka.classifiers.Classifier
Parameters:
d - a boolean that indicates whether debugging should be on or off

---

getSeed
public long **getSeed**()
Returns:
the current seed for the random number generator

---

setSeed
public void **setSeed**(long seed)
Parameters:
seed - set a new seed for the random number generator

---

classifyInstance
public double **classifyInstance**(weka.core.Instance inst)
throws java.lang.Exception
classify the passed Instance
Overrides:
classifyInstance in class weka.classifiers.Classifier
Parameters:
inst - the instance to classify
Returns:
the class that is assigned to this instance or -1x
Throws:
java.lang.Exception

---

buildClassifier
public void **buildClassifier**(weka.core.Instances data)
throws java.lang.Exception
induce a single rule. Note that the head of the rule should be set first, so that we know for which class to learn. Otherwise, we learn for the first class in the data (class value 0)
Specified by:
buildClassifier in class weka.classifiers.Classifier
Parameters:
data - the training set
Throws:
java.lang.Exception

---

toOptionString
public java.lang.String **toOptionString**()
Returns:
a string representation of the Learner, 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 java.lang.String **toString**()
Overrides:
toString in class java.lang.Object
Returns:
a string representation of the learner. Currently it returns the string representation of the current rule.

---

listOptions
public java.util.Enumeration **listOptions**()
Returns an enumeration describing the available options. Valid options are:
-D
turn on the debug mode and produce a lot of output -H heuristic
the heuristic used for evaluating candidate rules. If the heuristic takes parameters, these can be added as a string (e.g., -H "seco.heuristics.MEstimate -M 2") (Default: seco.heuristics.Laplace) -S
the seed for the random number generator (Default: 1) -C
the index of the class for which the rule should be learned (Default: 0) *
Specified by:
listOptions in interface weka.core.OptionHandler
Overrides:
listOptions in class weka.classifiers.Classifier
Returns:
an enumeration of all available options

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class weka.classifiers.Classifier
Parameters:
opts - the lis of options an array of strings
Throws:
Exeption - if an option is not supported
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
the current settings of the classifier
Specified by:
getOptions in interface weka.core.OptionHandler
Overrides:
getOptions in class weka.classifiers.Classifier
Returns:
an array of strings suitable for passing to setOptions

---

enumerateMeasures

public java.util.Enumeration **enumerateMeasures**()
Returns an enumeration of the additional measure names
Specified by:
enumerateMeasures in interface weka.core.AdditionalMeasureProducer
Returns:
an enumeration of the measure names

---

getMeasure
public double **getMeasure**(java.lang.String additionalMeasureName)
Returns the value of the named measure
Specified by:
getMeasure in interface weka.core.AdditionalMeasureProducer
Parameters:
measureName - the name of the measure to query for its value
Returns:
the value of the named measure
Throws:
java.lang.IllegalArgumentException - if the named measure is not supported

---

main
public static void **main**(java.lang.String[] args)
Main method.

Class / Interface 32: TopDownBeamSearch

public class **TopDownBeamSearch**
extends weka.classifiers.Classifier
implements weka.core.WeightedInstancesHandler, weka.core.AdditionalMeasureProducer, weka.core.OptionHandler
The seco package implements generic functionality for simple separate-and-conquer rule learning. TopDownBeamSearch implements a top-down (general-to-specific) induction of a single rule via beam search. In principle, a Rule can be used as a classifier in its own right, but the main purpose of this is to provide a learner for the inner loop of the separate-and-conquer/covering loop. TODO: - Stopping heuristics - Filtering heuristics (at the place where we are now " ---> ignored") - maybe initialization with a given rule (be careful not to re-initialize the rule, init used vector) - local optimization (gain heuristics do not allow a global comparison between rules, such as Foil-Gain) - LEFs (multiple evaluation heuristics) - cleaner interface should maybe store a Rule (and not a CandidateRule) internally and copy it to a CandidateRule for learning - split value can be chosen from domain or interpolate between neighboring values - Maybe the search should maintain a general confusion matrix so that CN2 can implemented (both, the entropy heuristic and the decision list search that auto-picks the class) - have I covered missing class values? - the two code blocks for finding the best condition should probably be separate routines/methods of some class. - is it possible to order to conditions so that certain combinations do not need to be re-searched? I guess that works only for exhaustive OPUS... However, there might be duplicate rules in the beam now! Parts of it is based on code for JRip and for Prism.

Constructor Detail

TopDownBeamSearch
public **TopDownBeamSearch**()

Method Detail

initRule
public void **initRule**()
initialize the rule learner with a fresh, empty rule.

---

initRule
public void **initRule**(CandidateRule r)
initialize the rule learner with
Parameters:
rule - a new starting point for the refinement

---

getRule
public CandidateRule **getRule**()
Returns:
the learned rule

---

getHeuristic
public SearchHeuristic **getHeuristic**()
return the heuristic used for evaluating rules
Returns:
a seco.heuristics.SearchHeuristic

---

setHeuristic
public void **setHeuristic**(SearchHeuristic h)
set the heuristic used for evaluating rules
Parameters:
h - a seco.heuristics.SearchHeuristic

---

getBeamWidth
public long **getBeamWidth**()
Returns:
the beam width (the number of candidate rules remembered for learning)

---

setBeamWidth
public void **setBeamWidth**(int beam)
Parameters:
beam - set the beam width

---

getTargetClass
public double **getTargetClass**()
Returns:
the index of the class for which a rule is learned

---

setTargetClass
public void **setTargetClass**(double i)
set the index of the class to a new value. it also resets the rule (initRule).

---

debug
public boolean **debug**()
Returns:
true if debugging is on, false if not

---

setDebug
public void **setDebug**(boolean d)
Overrides:
setDebug in class weka.classifiers.Classifier

Parameters:
d - a boolean that indicates whether debugging should be on or off

---

getSeed
public long **getSeed**()
Returns:
the current seed for the random number generator

---

setSeed
public void **setSeed**(long seed)
Parameters:
seed - set a new seed for the random number generator

---

classifyInstance
public double **classifyInstance**(weka.core.Instance inst)
throws java.lang.Exception
classify the passed Instance
Overrides:
classifyInstance in class weka.classifiers.Classifier
Parameters:
inst - the instance to classify
Returns:
the class that is assigned to this instance or -1x
Throws:
java.lang.Exception

---

buildClassifier
public void **buildClassifier**(weka.core.Instances data)
throws java.lang.Exception
induce a single rule. Note that the head of the rule should be set first, so that we know for which class to learn. Otherwise, we learn for the first class in the data (class value 0)
Specified by:
buildClassifier in class weka.classifiers.Classifier
Parameters:
data - the training set
Throws:
java.lang.Exception

---

toOptionString
public java.lang.String **toOptionString**()
Returns:
a string representation of the Learner, 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 java.lang.String **toString**()
Overrides:
toString in class java.lang.Object
Returns:
a string representation of the learner. Currently it returns the string representation of the current rule.

---

listOptions
public java.util.Enumeration **listOptions**()
Returns an enumeration describing the available options. Valid options are:

-D
turn on the debug mode and produce a lot of output -H heuristic
the heuristic used for evaluating candidate rules. If the heuristic takes parameters, these can be added as a string (e.g., -H "seco.heuristics.MEstimate -M 2") (Default: seco.heuristics.Laplace) -B beam-width
the beam width used for searching. This specifies the number of candidates that will be remembered after each refinement. If the value is inf (infinity), an exhaustive best-first search is performed. A beam width of 1 results in hill-climbing or gradient ascent search. (Default: 1) -S
the seed for the random number generator (Default: 1) Not yet valid are: -C
the index of the class for which the rule should be learned (Default: 0) *
Specified by:
listOptions in interface weka.core.OptionHandler
Overrides:
listOptions in class weka.classifiers.Classifier
Returns:
an enumeration of all available options

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
parse a list of options.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class weka.classifiers.Classifier
Parameters:
opts - the lis of options an array of strings
Throws:
Exeption - if an option is not supported
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
the current settings of the classifier
Specified by:
getOptions in interface weka.core.OptionHandler
Overrides:
getOptions in class weka.classifiers.Classifier
Returns:
an array of strings suitable for passing to setOptions

---

enumerateMeasures
public java.util.Enumeration **enumerateMeasures**()
Returns an enumeration of the additional measure names
Specified by:
enumerateMeasures in interface weka.core.AdditionalMeasureProducer
Returns:
an enumeration of the measure names

---

getMeasure
public double **getMeasure**(java.lang.String additionalMeasureName)
Returns the value of the named measure
Specified by:
getMeasure in interface weka.core.AdditionalMeasureProducer
Parameters:

measureName - the name of the measure to query for its value
Returns:
the value of the named measure
Throws:
java.lang.IllegalArgumentException - if the named measure is not supported

---

main
public static void **main**(java.lang.String[] args)
Main method.

---

getPruner
public PruningTemplate **getPruner**()
Returns:
the m_prune

---

setPruner
public void **setPruner**(PruningTemplate m_prune)
Parameters:
m_prune - the m_prune to set

C4 – Paket „seco.models“

Klassen	***Beschreibung***
CandidateRule	The seco package implements generic functionality for simple separate-and-conquer rule learning.
Condition	The seco package implements generic functionality for simple separate-and-conquer rule learning.
NominalCondition	Conditions with nominal values
NumericCondition	Conditions with numeric values
Rule	The seco package implements generic functionality for simple separate-and-conquer rule learning.
RuleSet	The seco package implements generic functionality for simple separate-and-conquer rule learning.
TrueCondition	The seco package implements generic functionality for simple separate-and-conquer rule learning.
ValueTestCondition	The seco package implements generic functionality for simple separate-and-conquer rule learning.

Anhang C - 5: Paketinhalte des Pakets "seco.models"

Class / Interface 33: CandidateRule

public class **CandidateRule**
extends Rule
implements java.lang.Comparable, weka.core.Copyable, java.lang.Cloneable, java.io.Serializable
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.

Constructor Detail

CandidateRule
public **CandidateRule**()

CandidateRule
public **CandidateRule**(Condition head)

Method Detail

copy
public java.lang.Object **copy**()
Specified by:
copy in interface weka.core.Copyable
Overrides:
copy in class Rule
Returns:
a shallow copy of the candidate rule The copy does not copy the conditions, while the clone does.

---

clone
public java.lang.Object **clone**()
throws java.lang.CloneNotSupportedException
Overrides:
clone in class Rule
Returns:

a deep copy of the candidate rule the clone contains also contains fresh copies of the conditions and the coverage stats.
Throws:
java.lang.CloneNotSupportedException

---

shadow
public java.lang.Object **shadow**()
throws java.lang.CloneNotSupportedException
Overrides:
shadow in class Rule
Returns:
a copy of the rule, but with empty body and empty stats.
Throws:
java.lang.CloneNotSupportedException

---

initBody
public void **initBody**()
reset the body of the rule to an empty vector and reset all statistics to 0.
Overrides:
initBody in class Rule

---

betterRule
public static CandidateRule **betterRule**(CandidateRule r1,
CandidateRule r2)
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

---

compareTo
public int **compareTo**(java.lang.Object o)
throws java.lang.ClassCastException
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).
Specified by:
compareTo in interface java.lang.Comparable
Parameters:
o - the object to compare to
Returns:
-1 if the o is better, 1 if the old rule is better, 0 else
Throws:
java.lang.ClassCastException - if the object is not a rule

---

compareTo
public int **compareTo**(CandidateRule r)
throws java.lang.NullPointerException
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
Throws:
java.lang.NullPointerException - if the object is null

---

equals
public boolean **equals**(CandidateRule r)
compare two rules, if the class and all conditions are equal the two rules are considered to be the same.
Parameters:
r - the rule to test the equality with
Returns:
true if the rules are equal

---

sort
public CandidateRule **sort**()

---

computeRuleValue
public double **computeRuleValue**(SearchHeuristic h)
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

---

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

---

getTieBreaker
public double **getTieBreaker**()
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.

---

resetTieBreaker
public void **resetTieBreaker**()
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.

---

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

---

getPredecessor
public CandidateRule **getPredecessor**()
get the predecessor of the candidate rule
Returns:
the predecessor or null

---

setPredecessor
public void **setPredecessor**(CandidateRule r)

specialize
public CandidateRule **specialize**(Condition c)
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 java.lang.String **toString**()
print out a candidate rule with coverage statistics and the heuristic value `@return a printable representation of the rule
Overrides:
toString in class Rule

Class / Interface 34: Condition

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

Constructor Detail

Condition
public **Condition**()

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)

Method Detail

getAttr
public weka.core.Attribute **getAttr**()

---

getValue
public double **getValue**()

---

setValue
public void **setValue**(double v)

cmp
public boolean **cmp**()

setCmp
public void **setCmp**(boolean c)

coveredInstances
public weka.core.Instances **coveredInstances**(weka.core.Instances data)
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)

toString
public abstract java.lang.String **toString**()
Overrides:
toString in class java.lang.Object

copy
public java.lang.Object **copy**()
implements Copyable
Specified by:
copy in interface weka.core.Copyable
Returns:
a shallow copy of itself

Class / Interface 35: NominalCondition

public class **NominalCondition**
extends Condition
implements java.lang.Cloneable, java.io.Serializable
Conditions with nominal values

Constructor Detail

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)

Method Detail

covers
public boolean **covers**(weka.core.Instance inst)
check whether the instance is covered by this condition
Specified by:
covers in class Condition
Parameters:
inst - the instance in question
Returns:
the boolean value indicating whether the instance is covered by this antecedent

---

toString
public java.lang.String **toString**()
Specified by:
toString in class Condition

Class / Interface 36: NumericCondition

public class **NumericCondition**
extends Condition
implements java.lang.Cloneable, java.io.Serializable
Conditions with numeric values

Constructor Detail

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)

Method Detail

covers
public boolean **covers**(weka.core.Instance inst)
check Whether the instance is covered by this condition
Specified by:
covers in class Condition
Parameters:
inst - the instance in question
Returns:
the boolean value indicating whether the instance is covered by this antecedent

---

toString
public java.lang.String **toString**()

Specified by:
toString in class Condition

Class / Interface 37: Rule

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

Constructor Detail

Rule
public **Rule**()

Rule
public **Rule**(Condition head)

Method Detail

getMaxLen
public int **getMaxLen**()

---

setMaxLen
public void **setMaxLen**(int maxLen)

---

copy
public java.lang.Object **copy**()
Specified by:
copy in interface weka.core.Copyable
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.

---

clone
public java.lang.Object **clone**()
throws java.lang.CloneNotSupportedException
Overrides:
clone in class java.lang.Object
Returns:
a deep copy of the list. the clone contains also contains fresh copies of the conditions.
Throws:
java.lang.CloneNotSupportedException

---

shadow
public java.lang.Object **shadow**()
throws java.lang.CloneNotSupportedException
Returns:
a copy of the rule, but with empty body and empty stats.
Throws:

java.lang.CloneNotSupportedException

---

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

---

setHead
public void **setHead**(Condition h)
set the head of the rule to a new class condition

---

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

---

getBody
public weka.core.FastVector **getBody**()
Returns:
the body of the rule, a FastVector of Conditions

---

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

---

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

---

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

---

addCondition
public void **addCondition**(Condition c)
add a condition to the body of the rule

---

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

---

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

---

replaceLastCondition
public void **replaceLastCondition**(Condition c)
replace the last condition from the body of the rule with a new condition
Parameters:

c - new condition

---

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

---

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

---

setStats
public void **setStats**(TwoClassStats stats)
set the TwoClassStats object to a new object

---

coveredInstances
public weka.core.Instances **coveredInstances**(weka.core.Instances data)
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.

---

uncoveredInstances
public weka.core.Instances **uncoveredInstances**(weka.core.Instances data)
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.

---

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

---

classifyInstance
public double **classifyInstance**(weka.core.Instance inst)
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

---

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

---

toString
public java.lang.String **toString**()
print out a rule with coverage statistitics `@return a printable representation of the rule
Overrides:
toString in class java.lang.Object

Class / Interface 38: RuleSet

public class **RuleSet**
extends java.lang.Object
implements java.io.Serializable
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?

Constructor Detail

RuleSet
public **RuleSet**()

Method Detail

setRuleSetStats
public void **setRuleSetStats**(RuleSetStats s)

getStats
public RuleSetStats **getStats**()

getRules
public weka.core.FastVector **getRules**()
Returns:
the rule set, a FastVector of Rules. Note that the default rule is *not* returned.

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

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

numConditions
public int **numConditions**()
Returns:

the number of conditions in the rules

---

addRule
public void **addRule**(Rule c)
add a rule to the set
Parameters:
c - the rule to be added

---

deleteLastRule
public void **deleteLastRule**()
delete the last rule

---

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

---

replaceLastRule
public void **replaceLastRule**(Rule r)
replace the last rule with a new rule
Parameters:
r - new rule

---

replaceCondition
public void **replaceCondition**(Rule r,
int n)
replace the nth rule with a new rule
Parameters:
r - new rule
n - number of rule to replace

---

getDefaultPrediction
public double **getDefaultPrediction**()
get the default prediction

---

getDefaultRule
public Rule **getDefaultRule**()
get the default rule

---

setDefaultRule
public void **setDefaultRule**(Rule r)
set the default rule to a new rule
Parameters:
r - the new rule

---

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

---

getFirstCoveringRule

public Rule **getFirstCoveringRule**(weka.core.Instance inst)
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

---

classifyInstance
public double **classifyInstance**(weka.core.Instance inst)
throws java.lang.Exception
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
Throws:
java.lang.Exception

---

toString
public java.lang.String **toString**()
Overrides:
toString in class java.lang.Object
Returns:
a printable version of a rule set.

Class / Interface 39: TrueCondition

public abstract class **TrueCondition**
extends Condition
implements weka.core.Copyable
The seco package implements generic functionality for simple separate-and-conquer rule learning. Condition implements conditions for rules. True is a condition that is always true.

Constructor Detail

TrueCondition
public **TrueCondition**()

Method Detail

coveredInstances
public weka.core.Instances **coveredInstances**(weka.core.Instances data)
return the list of instances that are covered by the condition
Overrides:
coveredInstances in class Condition
Parameters:
data - the list of instances
Returns:
a new list of covered instances

---

covers
public boolean **covers**(weka.core.Instance inst)
Specified by:
covers in class Condition

---

toString
public java.lang.String **toString**()
Specified by:
toString in class Condition

Class / Interface 40: ValueTestCondition

public abstract class **ValueTestCondition**
extends Condition
implements weka.core.Copyable
The seco package implements generic functionality for simple separate-and-conquer rule learning. Condition implements conditions for rules. ValueTestCondition implements conditions that test for certain value of attributes. They may be Nominalconditions or NumericConditions. Parts of it is based on code for JRip and for Prism.

Constructor Detail

ValueTestCondition
public **ValueTestCondition**(weka.core.Attribute a)

ValueTestCondition
public **ValueTestCondition**(weka.core.Attribute a,
double value)

ValueTestCondition
public **ValueTestCondition**(weka.core.Attribute a,
double value,
boolean cmp)

Method Detail

getAttr
public weka.core.Attribute **getAttr**()
Overrides:
getAttr in class Condition

---

getValue
public double **getValue**()
Overrides:
getValue in class Condition

---

setValue
public void **setValue**(double v)
Overrides:
setValue in class Condition

---

cmp

public boolean **cmp**()
Overrides:
cmp in class Condition

---

setCmp
public void **setCmp**(boolean c)
Overrides:
setCmp in class Condition

C5 – Paket „seco.pruning“

Klassen	***Beschreibung***
IREPOpt	The IREPOpt Pruningclass implements nearly the same pruning methods as the RIPPER from William W. Cohen
IREPruning	Implements the IREP pruning behavior.
NoPruning	Dummyclass if no Pruning is specified.
Prepruning	Class implementing Prepruning behavior.
REPruning	Implements the REP Pruningalgorithm.
TDPruning	This Class implements the TDP-Pruning Algorithm

Anhang C - 6: Paketinhalte des Pakets "seco.pruning"

Class / Interface 41: IREPOpt

public class **IREPOpt**
extends PruningTemplate
The IREPOpt Pruningclass implements nearly the same pruning methods as the RIPPER from William W. Cohen. Fast effective rule induction. In Machine Learning: Proceedings of the Twelfth International Conference, Lake Tahoe, California, 1995.
Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.

Constructor Detail

IREPOpt
public **IREPOpt**()

Method Detail

outerSplit
public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Specified by:
outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
the trainingexamples in the covering -loop Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.

The growing set is used for learning and the pruning set is used for pruning
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
A generic method.Implements the ClauseStoppingCriterion. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - - The RuleSet to prune.
data - - The test set
Returns:
Returns Whether to stop rule learning (true) or not (false)
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Implements the LiteralStoppingCriterion. Is always false
Specified by:
literalStop in class PruningTemplate
Parameters:
r - The CandidateRule
data - The test set
Returns:
Returns Whether to stop rule refining (true) or not (false)
Throws:
java.lang.Exception

---

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Optimization Stage like the RIPPER's
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - The ruleset to be optimized
data - The data for optimization
Returns:

The optimized ruleset
Throws:
java.lang.Exception

---

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Prunes a rule on the given data.
Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - The rule to prune
data - The pruning set
Returns:
the pruned rule or r, iff no better rule is found
Throws:
java.lang.Exception

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Description copied from class: PruningTemplate
Sets the options for the Pruningalgorithms.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class PruningTemplate
Parameters:
options - - The options
Throws:
java.lang.Exception

Class / Interface 42: IREPruning

public class **IREPruning**
extends PruningTemplate
Implements the REP Pruningalgorithm. Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.
-P pruning flag
Flag wether pruning is enabled or not.
-N number
Set number of folds for pruning. One fold is used as the pruning set. (Default: 3)
-SH Set the heuristic for Pruning (Default: seco.heuristics.Laplace) -LC Literalstopping Criterion
Set the Literalstopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-CC Clausestopping Criterion
Set the Clausestopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-OP
Specifies the RuleOperator for Pruning (Default: seco.pruning.operator.RuleDeleteLastCondition).
TODO:
- Specification of pruning operators. Use 1,2,...or all operators.

Constructor Detail

IREPruning
public **IREPruning**()

Method Detail

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Prunes a Rule.

1: curBest = r
2: r' = r
3: do
4: r' = apply operator p to r'
5: if Accuracy(r') < Accuracy(curBest) then
6: break;
7: curBest = r';
8: while(a better rule can be found)

Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - - The CandidateRule to prune.
data - - The pruningset
Returns:
Returns a pruned CandidateRule
Throws:
java.lang.Exception

---

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - The ruleset to prune
data - The pruning set
Returns:
The best simplification for a ruleset
Throws:
java.lang.Exception

---

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Implements the ClauseStoppingCriterion. Defers the decision to the LiteralStoppingCriterion,

because the decision for the RuleStop is only based on one Rule
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - The RuleSet
r - The CandidateRule
data - The test set
Returns:
Whether to stop rule learning (true) or not (false)
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
Implements the LiteralStoppingCriterion. Is always false
Specified by:
literalStop in class PruningTemplate
Parameters:
r - The CandidateRule
data - The test set
Returns:
Returns Whether to stop rule refining (true) or not (false)
Throws:
java.lang.Exception

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Description copied from class: PruningTemplate
Sets the options for the Pruningalgorithms.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class PruningTemplate
Parameters:
options - - The options
Throws:
java.lang.Exception

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler
Overrides:
listOptions in class PruningTemplate

---

outerSplit
public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains (num_folds-1)/num_folds of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/num_folds of the initial set size.
The growing set is used for learning and the pruning set is used for pruning. If num_folds is smaller

than zero then the set is not split.
Specified by:
outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains (num_folds-1)/num_folds of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/num_folds of the initial set size.
The growing set is used for learning and the pruning set is used for pruning. If num_folds is smaller than zero then the set is not split.
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

Class / Interface 43: NoPruning

public class **NoPruning**
extends PruningTemplate
Dummyclass if no Pruning is specified. All Methods are dummy Methods which do nothing or return a default value

Constructor Detail

NoPruning
public **NoPruning**()

Method Detail

postProcess
public java.lang.Object **postProcess**(java.lang.Object o,
weka.core.Instances data)
throws java.lang.Exception
Does nothing
Throws:
java.lang.Exception

---

clauseStop

public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
A generic method.Implements the ClauseStoppingCriterion. Is always false.
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - The RuleSet
r - The CandidateRule
data - The test set
Returns:
false
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
A generic method.Implements the LiteralStoppingCriterion. Is always false.
Specified by:
literalStop in class PruningTemplate
Parameters:
r - The CandidateRule
data - The test set
Returns:
false
Throws:
java.lang.Exception

---

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning.
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - The RuleSet to prune.
data - The pruningset
Returns:
Returns rs
Throws:
java.lang.Exception

---

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning.
Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - - The CandidateRule to prune.
data - - The pruningset
Returns:

Returns r
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
the initial set of examples
Throws:
java.lang.Exception

---

outerSplit
public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
the initial set of examples
Throws:
java.lang.Exception

Class / Interface 44: PrePruning

public class **PrePruning**
extends PruningTemplate
Class implementing Prepruning behavior. The Examples arent split and only the Methods for the Stoppingcriteria are overwritten.

Constructor Detail

PrePruning
public **PrePruning**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
Implements the ClauseStoppingCriterion. Is always false.
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - The RuleSet

r - The CandidateRule
data - The test set
Returns:
false
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Implements the LiteralStoppingCriterion. Delegates the decision to the StoppingCriterion
Specified by:
literalStop in class PruningTemplate
Parameters:
r - The CandidateRule
data - The test set
Returns:
Returns Whether to stop rule refining (true) or not (false)
Throws:
java.lang.Exception

---

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - - The RuleSet to prune.
data - - The pruningset
Returns:
Returns a pruned RuleSet
Throws:
java.lang.Exception

---

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - - The CandidateRule to prune.
data - - The pruningset
Returns:
Returns a pruned CandidateRule
Throws:
java.lang.Exception

---

outerSplit

public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
the initial set of examples
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
the initial set of examples
Throws:
java.lang.Exception

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Description copied from class: PruningTemplate
Sets the options for the Pruningalgorithms.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class PruningTemplate
Parameters:
options - - The options
Throws:
java.lang.Exception

Class / Interface 45: REPruning

public class **REPruning**
extends PruningTemplate
Implements the REP Pruningalgorithm. Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.
-P pruning flag
Flag wether pruning is enabled or not.
-N number
Set number of folds for pruning. One fold is used as the pruning set. (Default: 3)
-SH Set the heuristic for Pruning (Default: seco.heuristics.Laplace) -LC Literalstopping Criterion
Set the Literalstopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-CC Clausestopping Criterion

Set the Clausestopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-OP
If not specified REP uses the Operators delete-last-condition and delete-rule operator. Else it uses all Operators. find-best-simplification, delete-rule, delete-last-condition
TODO: - Specification of pruning operators. Use 1,2,...or all operators.

Constructor Detail

REPruning
public **REPruning**()

Method Detail

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
This Method processes a rule set, after the theory is learned. In the case of REP it tries to find the best simplification for the RuleSet.
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - The ruleset to prune
data - The pruning set
Returns:
The best simplification for a ruleset
Throws:
java.lang.Exception

---

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Does nothing.
Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - the rule to prune
data - the pruning set
Returns:
the rule r
Throws:
java.lang.Exception

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Sets the options for REP
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class PruningTemplate
Parameters:
options - The options String
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler
Overrides:
getOptions in class PruningTemplate

---

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
Implements the ClauseStoppingCriterion. Is always false.
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - The RuleSet
r - The CandidateRule
data - The test set
Returns:
false
Throws:
java.lang.Exception

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler
Overrides:
listOptions in class PruningTemplate

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
Implements the LiteralStoppingCriterion. Is always false.
Specified by:
literalStop in class PruningTemplate
Parameters:
r - The CandidateRule
data - The test set
Returns:
false
Throws:
java.lang.Exception

---

outerSplit
public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains (num_folds-1)/num_folds of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/num_folds of the initial set size.
The growing set is used for learning and the pruning set is used for pruning. If num_folds is smaller than zero then the set is not split.
Specified by:

outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Does nothing
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
the initial set of examples
Throws:
java.lang.Exception

Class / Interface 46: TDPruning

public class **TDPruning**
extends PruningTemplate
This Class implements the TDP-Pruning Algorithm

Constructor Detail

TDPruning
public **TDPruning**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
A generic method.Implements the ClauseStoppingCriterion. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
clauseStop in class PruningTemplate
Parameters:
rs - - The RuleSet to prune.
data - - The test set
Returns:
Returns Whether to stop rule learning (true) or not (false)
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
A generic method. Implements the LiteralStoppingCriterion. Must be overwritten for each new
Specified by:
literalStop in class PruningTemplate
Parameters:
r - - The CandidateRule to prune.
data - - The test set
Returns:
Returns Whether to stop rule refinement (true) or not (false)
Throws:
java.lang.Exception

---

postProcess
public RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
postProcess in class PruningTemplate
Parameters:
rs - - The RuleSet to prune.
data - - The pruningset
Returns:
Returns a pruned RuleSet
Throws:
java.lang.Exception

---

postProcessRule
public CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Specified by:
postProcessRule in class PruningTemplate
Parameters:
r - - The CandidateRule to prune.
data - - The pruningset
Returns:
Returns a pruned CandidateRule
Throws:
java.lang.Exception

---

innerSplit
public weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
the trainingexamples in the covering -loop Splits the examples into to set.

The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Specified by:
innerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

outerSplit
public weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Description copied from class: PruningTemplate
Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Specified by:
outerSplit in class PruningTemplate
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Description copied from class: PruningTemplate
Sets the options for the Pruningalgorithms.
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class PruningTemplate
Parameters:
options - - The options
Throws:
java.lang.Exception

C6 – Paket „seco.pruning.criterion“

Klassen	***Beschreibung***
CutOff	CutOff Stoppingcriterion.
MaximumErrorRate	The MaximumErrorRate StoppingCriterion.
MDL	The MDL-Stoppingcriterion based on Bernhard Pfahringer MDL-formula.
NoStopping	The StoppingCriterion which doesnt stop anything.
RelativeCutOff	CutOff Stoppingcriterion.
Significance	The Significance Test.

Anhang C - 7: Paketinhalte des Pakets "seco.pruning.criterion"

Class / Interface 47: CutOff

public class **CutOff**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
CutOff Stoppingcriterion.
Performs the CutOff based on an absolute evaluation of a rule.

Constructor Detail

CutOff
public **CutOff**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop adding rules to a ruleset, based on an absolute CutOff.
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
rs - The ruleset
r - The rule for which the CutOff is decided
data - Can be ignored here.
Returns:
true, iff the rules value is larger than the given threshhold.
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop refining a rule.
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule for which the CutOff is decided
data - Can be ignored here.
Returns:
true, iff the rules value is larger than the given threshhold.

Throws:
java.lang.Exception

Class / Interface 48: MaximumErrorRate

public class **MaximumErrorRate**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
The MaximumErrorRate StoppingCriterion. Decides when to stop adding rules to a ruleset, based on a rules errorrate. Also decides when to stop refining a rule based on the rules errorrate.

Constructor Detail

MaximumErrorRate
public **MaximumErrorRate**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop adding rules to a ruleset, based on a rules errorrate.
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
rs - The ruleset
r - The rule for which the error is estimated
data - the data on which the error is calculated.
Returns:
true, iff the rules error is larger than 50%
Throws:
java.lang.Exception

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule for which the error is estimated
data - the data on which the error is calculated.
Returns:
true, iff the rules error is larger than 50%
Throws:
java.lang.Exception

Class / Interface 49: MDL

public class **MDL**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
The MDL-Stoppingcriterion based on Bernhard Pfahringer MDL-formula. *B. Pfahringer. A New MDL Measure for Robust Rule Induction (Extended Abstract), in Lavrac N. and Wrobel S.(eds.), Machine Learning: ECML-95, Springer, Berlin/Heidelberg/New York/Tokyo, pp.331-334, 1995.*

Constructor Detail

MDL
public **MDL**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop adding rules to a ruleset, based on a MDL-Criterion.
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
rs - The ruleset
r - The rule
data - the data which is tried to compress
Returns:
true, iff L(rule)+L(data|rule) < L(data)
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop refining rules based on a MDL-Criterion.
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule
data - the data which is tried to compress
Returns:
true, iff L(rule)+L(data|rule)<L(data)
Throws:
java.lang.Exception

Class / Interface 50: NoStopping

public class **NoStopping**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
The StoppingCriterion which doesnt stop anything. All Methods return false.

Constructor Detail

NoStopping
public **NoStopping**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from interface: IRuleStoppingCriterion
Decides when to stop adding rules to a ruleset.
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
rs - The Ruleset
r - The rule
data - The data
Returns:
true iff the criterion is satisfied
Throws:
java.lang.Exception

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule
data - The data
Returns:
true iff the criterion is satisfied
Throws:
java.lang.Exception

Class / Interface 51: RelativeCutOff

public class **RelativeCutOff**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
CutOff Stoppingcriterion.
Performs the CutOff based on a relative evaluation of a rule. The rule is compared to its drect predecessor rule.

Constructor Detail

RelativeCutOff
public **RelativeCutOff**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop adding rules to a ruleset, based on an absolute CutOff.
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
rs - The ruleset
r - The rule for which the CutOff is decided
data - Can be ignored here.
Returns:
true, iff the rules value is larger than the given threshhold.
Throws:
throws - an Exception, because it is not yet implemented
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule for which the CutOff is decided
data - Can be ignored here.
Returns:
true, iff the rules value is larger than the given threshhold.
Throws:
java.lang.Exception

Class / Interface 52: Significance

public class **Significance**
extends Criterion
implements IRuleStoppingCriterion, IStoppingCriterion
The Significance Test.

Constructor Detail

Significance
public **Significance**()

Method Detail

clauseStop
public boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception

decides whether the likelihood ratio of a ruleset exceeds the threshhold or not
Specified by:
clauseStop in interface IRuleStoppingCriterion
Parameters:
r - The rule
data - The data
rs - The Ruleset
Returns:
true, iff the likelihood ratio exceeds the threshhold
Throws:
java.lang.Exception

---

literalStop
public boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
decides whether the likelihood ratio of a rule exceeds the threshhold or not
Specified by:
literalStop in interface IStoppingCriterion
Parameters:
r - The rule
data - The data
Returns:
true, iff the likelihood ratio exceeds the threshhold

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler
Overrides:
getOptions in class Criterion

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class Criterion
Throws:
java.lang.Exception

C7 – Paket „seco.pruning.model“

Klassen/Interfaces	*Beschreibung*
IRuleStoppingCriterion	Offers the Methods needed for the RuleStoppingCriterion
IStoppingCriterion	Offers the Methods needed for the StoppingCriterion
PruningTemplate	Template Class for Pruning behavior in the Covering class.
RuleOperator	This is a generic class for Ruleoperators
RuleSetOperator	This is a generic class for Rulesetoperators
Criterion	This is a generic class for a stopping criterion.

Anhang C - 8: Paketinhalte des Pakets "seco.pruning.model"

Class / Interface 53: IRuleStoppingCriterion

public interface **IRuleStoppingCriterion**
Offers the Methods needed for the RuleStoppingCriterion

Method Detail

clauseStop
boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Decides when to stop adding rules to a ruleset.
Parameters:
rs - The Ruleset
r - The rule
data - The data
Returns:
true iff the criterion is satisfied
Throws:
java.lang.Exception

Class / Interface 54: IStoppingCriterion

public interface **IStoppingCriterion**
Offers the Methods needed for the StoppingCriterion

Method Detail

literalStop
boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Parameters:
r - The rule
data - The data
Returns:
true iff the criterion is satisfied
Throws:
java.lang.Exception

Class / Interface 55: Criterion

public abstract class **Criterion**
extends java.lang.Object
implements java.io.Serializable, weka.core.OptionHandler
This is a generic class for a stopping criterion. It is in fact only a container class, which holds the options for a Criterion
Possible options are:
-TH - Sets the threshhold for a Stoppingcriterion (Default: 0)

-SH - Sets the Searchheuristic for a Stoppingcriterion (Default: seco.heuristics.Laplace)

Constructor Detail

Criterion
public **Criterion**()

Method Detail

setHeuristic
public void **setHeuristic**(SearchHeuristic h)

---

setThreshhold
public void **setThreshhold**(double th)

---

getHeuristic
public SearchHeuristic **getHeuristic**()

---

getThreshhold
public double **getThreshhold**()

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

---

toOptionString
public java.lang.String **toOptionString**()
Returns:
a string representation of the Criterion, 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).

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler

Throws:
java.lang.Exception

Class / Interface 56: PruningTemplate

public abstract class **PruningTemplate**
extends java.lang.Object
implements weka.core.OptionHandler, java.lang.Cloneable, java.io.Serializable
Template Class for Pruning behavior in the Covering class. Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.
-P pruning flag
Flag wether pruning is enabled or not.
-N
Set number of folds for pruning. One fold is used as the pruning set. (Default: 3)
-SH
Set the heuristic for Pruning (Default: seco.heuristics.Laplace) -LC Literalstopping Criterion
Set the Literalstopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-CC Clausestopping Criterion
Set the Clausestopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-LC Literalstopping Criterion
Set the Literalstopping Criterion. (Default: seco.pruning.criterion.NoStopping)
-RE
Set the number of restarts. (Default: 0)

Constructor Detail

PruningTemplate
public **PruningTemplate**()
A Standardconstructor

Method Detail

postProcess
public abstract RuleSet **postProcess**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Parameters:
rs - - The RuleSet to prune.
data - - The pruningset
Returns:
Returns a pruned RuleSet
Throws:
java.lang.Exception

---

postProcessRule
public abstract CandidateRule **postProcessRule**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
A generic method for pruning. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Parameters:

r - - The CandidateRule to prune.
data - - The pruningset
Returns:
Returns a pruned CandidateRule
Throws:
java.lang.Exception

---

literalStop
public abstract boolean **literalStop**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
A generic method. Implements the LiteralStoppingCriterion. Must be overwritten for each new
Parameters:
r - - The CandidateRule to prune.
data - - The test set
Returns:
Returns Whether to stop rule refinement (true) or not (false)
Throws:
java.lang.Exception

---

clauseStop
public abstract boolean **clauseStop**(RuleSet rs,
CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
A generic method.Implements the ClauseStoppingCriterion. Must be overwritten for each new Pruningalgorithm or at least return the Object which is passed to it.
Parameters:
rs - - The RuleSet to prune.
data - - The test set
Returns:
Returns Whether to stop rule learning (true) or not (false)
Throws:
java.lang.Exception

---

innerSplit
public abstract weka.core.Instances[] **innerSplit**(weka.core.Instances data)
throws java.lang.Exception
the trainingexamples in the covering -loop Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

outerSplit
public abstract weka.core.Instances[] **outerSplit**(weka.core.Instances data)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,

and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

split
public static weka.core.Instances[] **split**(weka.core.Instances data,
int num_folds)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains (num_folds-1)/num_folds of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/num_folds of the initial set size.
The growing set is used for learning and the pruning set is used for pruning. If num_folds is smaller than zero then the set is not split.
Parameters:
data - the initial set of examples
num_folds - the number of folds used for splitting
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

split
public static weka.core.Instances[] **split**(weka.core.Instances data)
throws java.lang.Exception
Splits the examples into to set.
The growing set (Instances[0]), which contains 2/3 of the initial instance set size,
and the pruning set (Instances[1]), which contains 1/3 of the initial set size.
The growing set is used for learning and the pruning set is used for pruning
Parameters:
data - the initial set of examples
Returns:
instances[0]: the growing set;
instances[1]: the pruning set
Throws:
java.lang.Exception

---

calculateRuleStats
public CandidateRule **calculateRuleStats**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Copies a CandidateRule and calculates the TwoClassStats an the given data.
Parameters:
r - The CandidateRule
data - The given data
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Sets the options for the Pruningalgorithms.
Specified by:
setOptions in interface weka.core.OptionHandler
Parameters:
options - - The options
Throws:
java.lang.Exception

---

getHeuristic
public SearchHeuristic **getHeuristic**()
Returns:
the m_heuristic

---

setHeuristic
public void **setHeuristic**(SearchHeuristic m_heuristic)
Parameters:
m_heuristic - the m_heuristic to set

---

getNumFolds
public int **getNumFolds**()
Returns:
the m_setFolds

---

setNumFolds
public void **setNumFolds**(int folds)
Parameters:
folds - the m_setFolds to set

---

toOptionString
public java.lang.String **toOptionString**()
Returns:
a string representation of the Pruningalgorithm, 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).

---

isRestartEnabled
public boolean **isRestartEnabled**()
Returns:
the m_restartEnabled

---

setRestartEnabled
public void **setRestartEnabled**(boolean enabled)

Parameters:
enabled - the m_restartEnabled to set

---

getNumRestarts
public int **getNumRestarts**()
Returns:
the m_numRestarts

---

setNumRestarts
public void **setNumRestarts**(int restarts)
Parameters:
restarts - the m_numRestarts to set

Class / Interface 57: RuleOperator

public abstract class **RuleOperator**
extends java.lang.Object
implements java.io.Serializable, weka.core.OptionHandler
This is a generic class for Ruleoperators Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.
-SH Set the heuristic for Pruning (Default: seco.heuristics.Laplace)

Constructor Detail

RuleOperator
public **RuleOperator**()
A defaul constructor

Method Detail

applyOperator
public abstract CandidateRule **applyOperator**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
This Method applies a specific Pruningoperator to a CandidateRule
Parameters:
r - The rule to prune
data - The pruning set
Returns:
The pruned rule or null if the operator cannot be applied
Throws:
java.lang.Exception

---

getHeuristic
public SearchHeuristic **getHeuristic**()
Returns:
the m_heuristic

---

setHeuristic
public void **setHeuristic**(SearchHeuristic m_heuristic)
Parameters:
m_heuristic - the m_heuristic to set

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

Class / Interface 58: RuleSetOperator

public abstract class **RuleSetOperator**
extends java.lang.Object
implements java.io.Serializable, weka.core.OptionHandler
This is a generic class for RuleSet operators. A RuleSetOperator applied to a RuleSet returns a set of new RuleSets. The operator is applied to each rule in the ruleset and creates a new RuleSet with one transformed rule.

1: Theories = empty
2: For Each Rule r in RuleSet rs
3: r' = apply Operator to r
4: rs' = rs \ r
5: rs' = rs' + r'
6: Theories = Theories + rs'
7: return(Theories)

Valid options are:
-D debug flag
Flag wether debug mode is enabled or not.
-SH Set the heuristic for Pruning (Default: seco.heuristics.Laplace)

Constructor Detail

RuleSetOperator
public **RuleSetOperator**()

Method Detail

applyOperator
public abstract java.util.Vector<RuleSet> **applyOperator**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Returns a set of rulesets where every Rule is simplified according to the operator

Parameters:
rs - The Ruleset which the operator is applied to
data - The pruning data in case needed
Returns:
A set of RuleSets
Throws:
java.lang.Exception

---

getOptions
public java.lang.String[] **getOptions**()
Specified by:
getOptions in interface weka.core.OptionHandler

---

listOptions
public java.util.Enumeration **listOptions**()
Specified by:
listOptions in interface weka.core.OptionHandler

---

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Throws:
java.lang.Exception

---

getHeuristic
public SearchHeuristic **getHeuristic**()
Gets the Searchheuristic
Returns:
the m_heuristic

---

setHeuristic
public void **setHeuristic**(SearchHeuristic heur)
Sets the Searchheuristic
Parameters:
heur - the m_heuristic to set

C8 – Paket „seco.pruning.operator“

Klassen	***Beschreibung***
RuleDeleteLastCondition	This is a Rule operator, which deletes the last condition of a rule
RuleFindBestSimplification	This is a rule operator, which searches for the best simplification of a rule given the data.
RuleFindBestReplacement	This ruleoperator tries to find the best replacement of a literal in a rule.
RuleIdentity	This is a rule operator which does nothing.
RuleSetDeleteLastCondition	This Ruleset operator returns a set of RuleSets.
RuleSetDeleteRule	This Ruleset operator returns a set of RuleSets.
RuleSetFindBestSimplification	This Ruleset operator returns a set of RuleSets.
RuleSetIdentity	This Ruleset operator returns a set containing one RuleSet.

Anhang C - 9: Paketinhalte des Pakets "seco.pruning.operator"

Class / Interface 59: RuleDeleteLastCondition

public class **RuleDeleteLastCondition**
extends RuleOperator
implements java.io.Serializable
This is a Rule operator, which deletes the last condition of a rule.

Constructor Detail

RuleDeleteLastCondition
public **RuleDeleteLastCondition**()

Method Detail

applyOperator
public CandidateRule **applyOperator**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleOperator
This Method applies a specific Pruningoperator to a CandidateRule
Specified by:
applyOperator in class RuleOperator
Parameters:
r - The rule to prune
data - The pruning set
Returns:
a candidateRule with the last condition deleted or r if the operator cant be applied
Throws:
java.lang.Exception

Class / Interface 60: RuleFindBestReplacement

public class **RuleFindBestReplacement**
extends RuleOperator
This ruleoperator tries to find the best replacement of a literal in a rule. not yet implemented properly.
TODO: Implemation work.

Constructor Detail

RuleFindBestReplacement
public **RuleFindBestReplacement**()

Method Detail

applyOperator
public CandidateRule **applyOperator**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleOperator
This Method applies a specific Pruningoperator to a CandidateRule
Specified by:
applyOperator in class RuleOperator
Parameters:
r - The rule to prune
data - The pruning set
Returns:
The pruned rule or null if the operator cannot be applied
Throws:
java.lang.Exception

Class / Interface 61: RuleFindBestSimplification

public class **RuleFindBestSimplification**
extends RuleOperator
This is a rule operator, which searches for the best simplification of a rule given the data.

Constructor Detail

RuleFindBestSimplification
public **RuleFindBestSimplification**()

Method Detail

applyOperator
public CandidateRule **applyOperator**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleOperator
This Method applies a specific Pruningoperator to a CandidateRule
Specified by:
applyOperator in class RuleOperator
Parameters:
r - The rule to be pruned
data - The data for the search of the best simplification
Returns:
the best simplification of the rule r or the r itself if no simplification is found
Throws:
java.lang.Exception

Class / Interface 62: RuleIdentity

public class **RuleIdentity**
extends RuleOperator
implements java.io.Serializable
This is a rule operator which does nothing. It returns a rule without changing it

Constructor Detail

RuleIdentity
public **RuleIdentity**()

Method Detail

applyOperator
public CandidateRule **applyOperator**(CandidateRule r,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleOperator
This Method applies a specific Pruningoperator to a CandidateRule
Specified by:
applyOperator in class RuleOperator
Parameters:
r - The Rule to be pruned
data - The data on which the rule is pruned
Returns:
The rule
Throws:
java.lang.Exception

Class / Interface 63: RuleSetDeleteLastCondition

public class **RuleSetDeleteLastCondition**
extends RuleSetOperator
This RuleSet operator generates a set of new RuleSets. For each rule the last condition is deleted an a new Ruleset is generated

Constructor Detail

RuleSetDeleteLastCondition
public **RuleSetDeleteLastCondition**()

Method Detail

applyOperator
public java.util.Vector<RuleSet> **applyOperator**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleSetOperator
Returns a set of rulesets where every Rule is simplified according to the operator
Specified by:
applyOperator in class RuleSetOperator

Parameters:
rs - The RuleSet to be pruned
data - The data on which the ruleset is pruned
Returns:
A set of new RuleSets
Throws:
java.lang.Exception

Class / Interface 64: RuleSetDeleteRule

public class **RuleSetDeleteRule**
extends RuleSetOperator
This Ruleset operator returns a set of RuleSets. each rule is deleted an a new Ruleset is generated

Constructor Detail

RuleSetDeleteRule
public **RuleSetDeleteRule**()

Method Detail

applyOperator
public java.util.Vector<RuleSet> **applyOperator**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleSetOperator
Returns a set of rulesets where every Rule is simplified according to the operator
Specified by:
applyOperator in class RuleSetOperator
Parameters:
rs - The RuleSet to be pruned
data - The data on which the ruleset is pruned
Returns:
A set of new RuleSets
Throws:
java.lang.Exception

Class / Interface 65: RuleSetFindBestSimplification

public class **RuleSetFindBestSimplification**
extends RuleSetOperator
This Ruleset operator returns a set of RuleSets. For each rule it is tried to find the best simplification and a new Ruleset is created

Constructor Detail

RuleSetFindBestSimplification
public **RuleSetFindBestSimplification**()

Method Detail

applyOperator
public java.util.Vector<RuleSet> **applyOperator**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleSetOperator
Returns a set of rulesets where every Rule is simplified according to the operator
Specified by:
applyOperator in class RuleSetOperator
Parameters:
rs - The RuleSet to be pruned
data - The data on which the ruleset is pruned
Returns:
A set of new RuleSets
Throws:
java.lang.Exception

setOptions
public void **setOptions**(java.lang.String[] options)
throws java.lang.Exception
Specified by:
setOptions in interface weka.core.OptionHandler
Overrides:
setOptions in class RuleSetOperator
Throws:
java.lang.Exception

Class / Interface 66: RuleSetIdentity

public class **RuleSetIdentity**
extends RuleSetOperator
This RuleSet operator returns a Set with one RuleSet.

Constructor Detail

RuleSetIdentity
public **RuleSetIdentity**()

Method Detail

applyOperator
public java.util.Vector<RuleSet> **applyOperator**(RuleSet rs,
weka.core.Instances data)
throws java.lang.Exception
Description copied from class: RuleSetOperator
Returns a set of rulesets where every Rule is simplified according to the operator
Specified by:
applyOperator in class RuleSetOperator
Parameters:

rs - The Ruleset to be pruned data - The data on which the ruleset is pruned **Returns:** A Set containing the RuleSet rs **Throws:** java.lang.Exception