Implementation und Evaluation eines Regressionsregellerners

Diplomarbeit von Stefan Steger
21. August 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Knowledge Engineering

Betreuer: Prof. Johannes Fürnkranz
Frederick Janssen

Erklärung zur Diplomarbeit

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

Darmstadt, den 22. August 2011

(S. Steger)

Inhaltsverzeichnis

1 Einleitung **4**
- 1.1 Motivation . . . 4
- 1.2 Überblick der Kapitel . . . 4

2 Regel-Lernen und Regression **5**
- 2.1 Separate-and-Conquer Regel-Lernen . . . 6
- 2.2 Charakteristika von Regel-Lernern . . . 8
  - 2.2.1 Repräsentationssprachen . . . 8
  - 2.2.2 Suchstrategien . . . 9
  - 2.2.3 Konzepte zur Vermeidung von Überbestimmtheit . . . 11
- 2.3 Verschiedene Algorithmen . . . 12
  - 2.3.1 AQ . . . 12
  - 2.3.2 CN2 . . . 12
  - 2.3.3 Ripper . . . 13
- 2.4 Regression . . . 13
  - 2.4.1 Regression in der Statistik . . . 14
- 2.5 Regression in maschinellem Lernen . . . 14
  - 2.5.1 Nicht regelbasierte Ansätze . . . 14
  - 2.5.2 Regelbasierte Ansätze . . . 15
    - 2.5.2.1 Diskretisierung der Klassenvariable . . . 15
    - 2.5.2.2 Dynamische Reduktion auf Klassifikation . . . 15
    - 2.5.2.3 M5Rules . . . 16
    - 2.5.2.4 R^2 . . . 16
    - 2.5.2.5 RuleFit . . . 16
    - 2.5.2.6 RegENDER . . . 16
    - 2.5.2.7 SeCoReg . . . 17
- 2.6 Evaluierung von Regressionsalgorithmen . . . 17

3 Reduce-and-Conquer **20**
- 3.1 Reduce-and-Conquer Regel-Lernen . . . 20
  - 3.1.1 Die Reduce Strategie . . . 20
  - 3.1.2 Adaption des generischen Separate-and-Conquer Algorithmus . . . 21
  - 3.1.3 Splitpoints . . . 24
- 3.2 Die Charakteristiken von Reduce-and-Conquer . . . 25
  - 3.2.1 Die Repräsentationssprache . . . 26
  - 3.2.2 Die Suche im Hypothesenraum . . . 26
  - 3.2.3 Vermeiden von Overfitting . . . 26
- 3.3 Die Konfigurationen von Reduce-and-Conquer . . . 27
  - 3.3.1 Anzahl der Regeln . . . 27
  - 3.3.2 Die Lernmenge . . . 28
  - 3.3.3 Mindestabdeckung der Regeln . . . 29
  - 3.3.4 Die Heuristiken . . . 29
- 3.4 Ähnlichkeiten und Unterschiede zu verschiedenen Algorithmen . . . 29

4 Evaluierung und Resultate **30**
4.1 Beschreibung der Datenmengen 30
4.2 Die Testkonfiguration 30
4.3 Evaluation der verschiedenen Konfigurationen 32
4.3.1 Feste Regelanzahl 33
4.3.2 Abbruchkriterium mit Vorschau 34
4.3.3 Aufteilung der Growing- und Pruningmenge 35
4.3.4 Mindestabdeckung 36
4.3.5 Die Heuristik MD 37
4.4 Vergleich der verschiedenen Konfigurationen 39
4.5 Vergleich mit anderen Regressionslernern 41
4.6 Runtime-Effizienz 43

5 Zusammenfassung und Ausblick **44**

1 Einleitung

1.1 Motivation

Methoden zur Vorhersage von Werten können in zwei Kategorien unterteilt werden. In der Klassifikation werden diskrete Werte vorhergesagt, bei der Regression dagegen kontinuierliche. Regression ist ein in der Statistik gut untersuchtes Feld, im maschinellen Lernen dagegen nicht so verbreitet. Hier liegt der Forschungsschwerpunkt eher auf Klassifikation. Dennoch sind viele Probleme aus dem Alltag Regressionsprobleme. So interessiert man sich beim Kauf von Aktien nicht nur für die Kategorien „steigt, stagniert, fällt“ des Aktienkurses, sondern will den genauen Wert zu einem bestimmten Zeitpunkt erfahren. Dies kann nur mit Regression gelöst werden. Aber das ist nicht der einzige Grund sich mit Regression zu beschäftigen. So können mit Regression auch Klassifikationsprobleme gelöst werden, indem man die Klassen als kontinuierliche Werte behandelt.

Es gibt verschiedene Regressionsansätze, in dieser Arbeit werden regelbasierte Ansätze verwendet. Der Vorteil von regelbasierter Regression liegt in der guten Möglichkeit das Gelernte zu interpretieren. Dies ist aber nicht der einzige Vorteil von Regeln. So kann durch Regeln das Gelernte sehr kompakt dargestellt werden. Klassifikation mit Hilfe von Regeln ist ein gut erforschtes Gebiet, für Regression gibt es aber nur wenige Ansätze. In dieser Arbeit wird ein Algorithmus vorgestellt, der einige Methoden der Klassifikation für Regression adaptiert. Ziel der Arbeit ist es, die Eigenschaften des Algorithmus zu untersuchen und ihn mit anderen Lernern zu vergleichen. Dazu werden Experimente mit verschiedenen Datenmengen und den verschiedenen Algorithmen durchgeführt.

1.2 Überblick der Kapitel

In Kapitel 2 wird zuerst eine Einführung in regelbasiertes Lernen gegeben. Es wird besonders auf die Separate-and-Conquer Strategie eingegangen und anhand dieser verschiedene Charakteristika von Regellernern erläutert. Im Anschluss werden einige bekannte Klassifikationsregellerner beschrieben. Im Abschnitt 2.4 wird auf die Unterschiede von Regression und Klassifikation eingegangen. Es folgt eine Auflistung von verschiedenen Regressionsalgorithmen mit besonderem Augenmerk auf die regelbasierten Regressionsalgorithmen. Der letzte Abschnitt erläutert einige gebräuchliche Evaluationsmethoden für Regression.

In Kapitel 3 wird der Reduce-and-Conquer Algorithmus vorgestellt und untersucht. Es werden die verschiedenen Konzepte vorgestellt und ein Überblick über die verwendeten Konfigurationen des Algorithmus gegeben.

Kapitel 4 beginnt mit der Beschreibung der Datenmengen und der Testkonfiguration. Die einzelnen Ausprägungen des Algorithmus werden evaluiert und verglichen. Anschließend wird die beste Version des Algorithmus mit anderen bekannten Regressionsalgorithmen verglichen.

Zum Schluss werden die Erkenntnisse der Arbeit in Kapitel 5 zusammengefasst.

2 Regel-Lernen und Regression

Das Ziel von maschinellen Lernern ist Gesetzmäßigkeiten auf einer Menge von Beispielen (engl. instances) zu erkennen und dadurch Wissen zu generalisieren. Dabei erstellt der Lerner während des Prozesses des Lernens eine Hypothese, die das erlernte Wissen möglichst genau nachbildet. Die Hypothese, auch Theorie genannt, ist eine Abbildung, die jedem Eingabewert den vermuteten Ausgabewert zuordnet.

Im Unterschied zum unüberwachten Lernen (engl. unsupervised learning) sind die Zuordnungen der Beispiele schon vor dem Lernen bekannt. Dadurch kann der Lerner die Theorie mit den tatsächlichen Ausgaben der Beispiele vergleichen und somit die Theorie verbessern. Diese Art von Lernen nennt man überwachtes Lernen (engl. supervised learning). Das Ziel dabei ist, die Ausgabe von unbekannten Beispielen vorherzusagen. Die Eingaben der einzelnen Beispiele kann man, wie in (1) zu sehen ist, als einen Vektor von Attribut-Wertepaaren modellieren.

$$(Wetter = sonnig, Temperatur = 25, Wind = ja, SpieleGolf = ja) \tag{1}$$

Die Attribute können entweder nominal als auch numerisch sein. Numerische Attribute sind solche, auf denen eine Ordnung besteht. In unserem Fall ist *Temperatur* ein numerisches Attribut, welches Werte aus dem Intervall $\{-20, ..., 40\}$ annehmen kann. Nominale Attribute, wie z.B. *Wetter* haben eine solche Ordnung nicht, sie bestehen aus Zeichensymbolen.

Im überwachten Lernen hat das letzte Element des Attributvektors meist eine besondere Bedeutung. Dieses Attribut, auch Klassenattribut genannt, ist der Wert, den das gelernte Modell vorhersagen soll. In dem Beispiel soll der Lerner eine Vorhersage über das richtige Wetter zum Golf spielen machen. Da das Klassenattribut *SpieleGolf* nur die beiden Werte *ja* und *nein* annehmen kann, handelt es sich hier um ein 2-Klassen Problem. Beispiele, bei denen der Wert des Klassenattributes *ja* lautet, werden als „positiv“ bezeichnet, Beispiele der Klasse *nein* werden als „negativ“ bezeichnet.

Während des Lernens vergleicht der Lerner die Ausgaben seiner Theorie mit den Klassenattributen der Trainingsbeispiele und passt seine Theorie an. Angestrebtes Ziel ist eine möglichst korrekte Theorie zu erstellen. Korrektheit definiert sich bei 2-Klassen-Problemen aus

- Vollständigkeit (engl. completeness)
- Konsistenz (engl. consistency)

Eine Theorie ist vollständig, wenn alle positiven Beispiele beschrieben werden und konsistent, wenn keine negativen Trainingsbeispiele enthalten sind. Gewünscht ist aber eine Korrektheit nicht nur auf den Trainingsbeispielen, sondern auf allen möglichen Beispielen. Die Zuordnung unbekannter Beispiele in Klassen nennt sich Klassifikation.

Im Gegensatz dazu steht die Regression, in der es keine Klassen mehr gibt, sondern ein numerischer Wert vorhergesagt wird. Es gibt keine Einteilung mehr in positive und negative Beispiele. Die oben beschriebene Korrektheit kann man deshalb so nicht für Regressionsprobleme verwenden. Eine Theorie für ein Regressionsproblem ist korrekt, wenn für alle Beispiele der richtige Wert vorhergesagt wird. Im Abschnitt 2.6 werden einige Methoden vorgestellt, wie man Korrektheit bei Regressionsregellernern misst.

Die Gemeinsamkeit aller Regel-Lerner besteht in der verwendeten Hypothesensprache. Die Theorie wird in Form von Regeln angegeben. Regeln haben einen Regelkörper und einen Regelkopf. Der Regelkörper besteht aus einer Reihe von Tests, auch Konditionen genannt, die miteinander verknüpft sind. Ein einzelner Test besteht aus einem Attribut, einer Relation und einem Wert, wobei getestet wird, ob die Relation erfüllt ist. Hier werden für nominale Attribute die Relationen $=$ und $\neq$, bei numerischen die Relationen $<$ und $\geq$ oder $\leq$ und $>$ verwendet. Der Regelkopf besteht aus der Vorhersage. Für (1) könnte eine vom Regel-Lerner gefundene Regel folgendermaßen aussehen:

$$(Wetter = sonnig \wedge Temperatur > 20 \wedge Wind = ja) \rightarrow SpieleGolf = ja \qquad (2)$$

Alle Beispiele, die den Test dieser Regel erfüllen, werden der Klasse *SpieleGolf* $= ja$ zugewiesen. Sie werden von der Regel abgedeckt. Der Vorteil dieser Repräsentationssprache gegenüber anderen, wie z.B. neuronalen Netzen, liegt unter anderem in der guten Interpretierbarkeit für Menschen. Eine den Regeln ähnliche Repräsentation sind Bäume. Bäume können immer auch durch Regeln beschrieben werden, umgekehrt ist dies nicht möglich. Bäume separieren den Hypothesenraum in nicht-überlappende Bereiche, Regeln können den Hypothesenraum auch in sich überlappende Bereiche separieren. So können Theorien kompakter dargestellt werden.

2.1 Separate-and-Conquer Regel-Lernen

In diesem Abschnitt werden Separate-and-Conquer Algorithmen (SeCo) genauer erläutert, da diese viele Merkmale des dieser Arbeit zu Grunde liegenden Algorithmus haben.

In den von Michalski entwickelten Algorithmen der AQ Familie [26] wird die Separate-and-Conquer Strategie erstmals noch unter den Namen Covering Strategie verwendet. Der Begriff Separate-and-Conquer wurde von Pagallo und Haussler [30] aufgrund der charakteristischen Weise der Entwicklung der Theorie während des Lernens eingeführt.

Die Funktionsweise des Algorithmus kann man in die beiden Bereiche „Abtrennen“ (separate) und „Erobern“ (conquer) unterteilen. Die beiden Bereiche werden durch zwei verschachtelte Schleifen realisiert, wobei das „Erobern“ in der inneren und das „Abtrennen“ in der äußeren Schleife passiert. Im Conquer-Schritt wird durch schrittweises Verfeinern eine Regel gefunden, die einen Teil der Trainingsbeispiele abdeckt. Danach werden im Separate-Schritt alle Beispiele, die von dieser Regel abgedeckt werden aus der Trainingsmenge gelöscht und die gefundene Regel einer Regelmenge hinzugefügt. Eine Regel wird verfeinert, indem dem Regelkörper Konditionen hinzugefügt werden. Die Regel wird spezialisiert. Man kann auch aus der Regel einzelne Bedingungen löschen, man spricht dann von generalisieren. Diese Vorgehensweise ist in allen Separate-and-Conquer Algorithmen gleich.

Algorithm 1 zeigt den generischen Separate-and-Conquer Algorithmus wie er in [19] beschrieben wird. Es werden mehrere Subroutinen aufgerufen, die für die jeweilige Ausprägung des Separate-and-Conquer Algorithmus angepasst werden können.

Es wird mit der leeren Theorie gestartet, d.h. die Regelmenge ist leer. Falls positive Beispiele in der Trainingsmenge sind, wird die Subroutine FindBestRule gestartet. Diese lernt eine Regel die einige Beispiele abdeckt. Die abgedeckten Beispiele werden entfernt, die gefundene Regel wird der Theorie hinzugefügt und FindBestRule wird erneut aufgerufen. Dies wird solange wiederholt bis es entweder keine positiven Beispiele mehr gibt oder bis ein RuleStoppingCriterion greift. Die Theorie kann dann im Anschluss noch einem PostProcessing unterworfen werden.

Algorithm 1 Separate-and-Conquer

```
procedure SeparateAndConquer(Examples)
Theory = ∅
while Positive(Examples) ≠ ∅ do
  Rule = FindBestRule(Examples)
  Covered = Cover(Rule, Examples)
  if RuleStoppingCriterion(Theory, Rule, Examples) then
    exit while
  end if
  Examples = Examples \ Covered
  Theory = Theory ∪ Rule
end while
Theory = PostProcess(Theory)
return Theory

procedure FindBestRule(Examples)
InitRule = InitializeRule(Examples)
InitVal = EvaluateRule(InitRule)
BestRule = <InitVal, InitRule>
while Rules ≠ ∅ do
  Candidates = SelectCandidates(Rules, Examples)
  Rules = Rules \ Candidates
  for Candidate ∈ Candidates do
    Refinements = RefineRule(Candidate, Examples)
    for Refinement ∈ Refinements do
      Evaluation = EvaluateRule(Refinement, Examples)
      unless StoppingCriterion(Refinement, Evaluation, Examples)
      NewRule = <Evaluation, Refinement>
      Rules = InsertSort(NewRule, Rules)
      if NewRule > BestRule then
        BestRule = NewRule
      end if
    end for
  end for
  Rules = FilterRules(Rules, Examples)
end while
return BestRule
```

Die Prozedur FindBestRule durchsucht den Hypothesenraum nach einer optimalen Regel. Die Heuristik, welche die „Güte“ einer Regel bewertet, wird in EvaluateRule definiert. Der numerische Wert dieser Heuristik ist größer je mehr positive und je weniger negative Beispiele von der Regel abgedeckt werden. Durch InitializeRules wird eine sortierte Liste mit Kandidatenregeln erzeugt. Diese Regeln heißen Kandidatenregeln, weil sie die „Kandidaten“ für die Verfeinerung sind. Dabei werden neue Regeln durch InsertSort immer an den richtigen Stellen der Kandidatenliste eingefügt. Sie werden nach dem Wert, den EvaluateRule liefert absteigend sortiert. Solange noch Kandidaten in der Kandidatenliste sind wird die *while*-Schleife ausgeführt. Bei jedem Schleifendurchgang wählt SelectCandidates die besten Regeln aus der Kandidatenliste aus, die durch RefineRules weiter verfeinert werden. Jede dieser Verfeinerungen wird wieder mit EvaluateRules ausgewertet und in die Kandidatenliste eingefügt. Dies geschieht solange bis das StoppingCriterion feuert. Wird eine Regel gefunden, die besser als die bisherigen Regeln ist, wird diese in *bestRule* zwischengespeichert. Anschließend wählt FilterRules die besten Regeln aus der Regelliste aus, die dann für die weiteren Durchgänge verwendet werden. Sobald alle Kandidatenregeln verarbeitet wurden, wird *bestRule* zurückgegeben.

Viele Separate-and-Conquer Regel-Lerner können durch diesen Algorithmus implementiert werden, indem die verschiedenen Prozeduren konkret ausgeformt werden. So wird die konkrete Repräsentationssprache in RefineRule festgelegt, der Suchalgorithmus in SelectCandidates und FilterRules, die Suchstrategie in InitializeRule und RefineRule, die Suchheuristik in EvaluateRule und die Overfitting Strategie kann in den Stoppingkriterien und in PostProcess implementiert werden.

2.2 Charakteristika von Regel-Lernern

Regel-Lerner und insbesondere Separate-and-Conquer Algorithmen können durch verschiedene Merkmale unterschieden werden. Die wichtigsten Merkmale hierbei sind:

- die Repräsentationssprache (engl. language bias)
- der Art der Suche (engl. search bias)
- die Strategie um Überbestimmtheit zu verhindern (engl. overfitting avoidance bias)

Im Folgenden wird auf die verschiedenen Ausprägungen detailliert eingegangen.

2.2.1 Repräsentationssprachen

Die Wahl einer passenden Repräsentationsprache beeinflußt entscheidend das Ergebnis eines Regel-Lerners [29]. Grundsätzlich kann man zwischen statischen und dynamischen Ansätzen unterscheiden. Bei statischen Verfahren wird die Repräsentationssprache im Vorfeld gewählt und danach nicht mehr verändert.

In Beispiel 2 wird die disjunktive Normalform (kurz DNF) verwendet. Dabei werden die einzelnen Bedingungen der Regel durch Disjunktionen verknüpft. Negiert man die DNF, so erhält man die konjunktive Normalform (kurz KNF). Durch Einführen von Intervallen oder auch baumartigen Strukturen kann man diese Formen erweitern.

Die Repräsentationssprache bestimmt aber nicht nur das Aussehen der Regeln, sondern auch die Form, wie diese zusammengefasst werden. So gibt es z.B. Entscheidungslisten, die die Regeln sortiert abspeichern. Die Art der Sortierung hängt von der Art der Suchstrategie (siehe Abschnitt 2.2.2) ab. Sollte nun ein unbekanntes Beispiel durch die erste Regel nicht abgedeckt werden, wird die 2. Regel getestet und so fort, bis eine Regel das Beispiel abdeckt. Die erste Regel, die das Beispiel abdeckt bestimmt die Klassenzugehörigkeit. Alle nachfolgenden Regeln haben keinen Einfluss mehr auf die Klassifizierung dieses Beispiels.

Es gibt aber auch Regellisten, die keine Sortierung benötigen. Hier ist die Reihenfolge der Regeln unerheblich. In dem in Kapitel 3 vorgestellten Algorithmus wird solch eine unsortierte Entscheidungsliste implementiert. Dort wird jede Regel, die das Beispiel abdeckt, zur Vorhersage des Klassenattributes benutzt. Durch Entscheidungslisten können Theorien meist kompakter als in anderen Repräsentationssprachen dargestellt werden.

Eine andere Repräsentationsmöglichkeit sind funktionale Relationen, wie sie in Foil [32] benutzt werden. Die Theorie wird in Form von PROLOG Relationen dargestellt.

Dynamische Ansätze haben den Vorteil, auch während des Lernens die Repräsentationssprache anzupassen oder sogar zu ändern. So werden in CLINT [9] mehrere Repräsentationssprachen anhand ihrer Darstellungsgüte aufsteigend sortiert. Wenn eine Theorie nicht komplett und konsistent mit der ersten Sprache dargestellt werden kann, so wird die Nächste verwendet. Eine andere dynamische Variante erweitert bei Bedarf die Repräsentationssprache. So werden in Utgoff [35] dynamisch neue Eigenschaften aus schon vorhandenen Attributen erzeugt.

2.2.2 Suchstrategien

Es gibt verschiedene Methoden in einem Hypothesenraum, also der Menge aller möglichen Theorien, nach Hypothesen zu suchen. Da die Hypothesen durch Regeln abgebildet werden, ist die Suche nach Hypothesen gleichzeitig auch die Suche nach den Regeln, die diese abbilden. Diese Suche kann durch 3 Merkmale beschrieben werden:

- die Suchstrategie
- der Suchalgorithmus
- die Suchheuristik

Es gibt zwei grundsätzliche Suchstrategien, Top-Down Search und Bottom-Up Search sowie die Kombination beider Varianten (Bidirektionale Suche).

Beim Top-Down Search wird der Hypothesenraum durch das wiederholte Verfeinern von Kandidatenregeln durchsucht. Kandidatenregeln sind solche, die während der Erstellung der Theorie erzeugt werden, aber noch nicht zur Theorie gehören. Dabei wird mit der generellsten Regel angefangen und diese so lange spezialisiert, bis sie der Theorie zugefügt werden kann - oder verworfen wird. Nahezu alle Separate-and-Conquer Algorithmen verwenden diese Strategie, wie z.B. CN2 [6].

Die Bottom-Up Strategie durchsucht den Hypothesenraum angefangen mit der speziellsten Regel und generalisiert diese dann. Dabei wird zumeist ein positives Beispiel zufällig ausgewählt, dass dann generalisiert wird.

Die Kombination beider Strategien, die bidirektionale Suche, kann sowohl spezialisieren als auch generalisieren. SWAP-1 [36] verwendet solch einen Ansatz, indem grundsätzlich die Regeln spezialisiert werden, aber nach jeder Spezialisierung entschieden wird, ob eine vorher gelernte Bedingung nicht gelöscht oder ersetzt werden kann.

Es gibt verschiedene Arten von Suchalgorithmen. So kann man alle möglichen Theorien miteinander vergleichen und die Beste auswählen. Dieser „Brute-Force"-Ansatz wird allerdings selten angewandt, da die Suche sehr zeitaufwendig ist. Es gibt auch noch ein anderes grundsätzliches Problem. In Hypothesenräumen mit unendlich vielen Hypothesen wird der Brute-Force Algorithmus nie eine Lösung finden.

Ein anderes Verfahren ist als „Greedy"-Suche bekannt. Hier wird schrittweise immer der beste Folgezustand ausgesucht. Die gebräuchliste „Greedy"-Suche ist Hill-climbing. Es wird so lange immer die beste Verfeinerung einer Kandidatenregel verwendet bis keine Verbesserung mehr erreicht werden kann. Dabei versucht Hill-climbing ein globales Optimum zu finden in dem während der Verfeinerung lokal optimale Schritte verwendet werden. Dabei kann es passieren das ein lokales Optimum nicht zum globalen Optimum führt. Durch „Vorausschauen" können diese Probleme verringert werden, allerdings auf Kosten der Laufzeit und des Speicherplatzes, da deutlich mehr Verfeinerungen durchgeführt und zwischengespeichert werden müssen.

Die Nachteile von Hill-climbing versucht Beam-Search zu umgehen, indem nicht nur der beste Verfeinerungsschritt gespeichert wird, sondern auch eine feste Anzahl von Alternativen, den sogenannten Beam. Hill-Climbing ist eine Sonderform von Beam Search mit einem Beam von 1. Die Laufzeit wird dadurch nur um einen konstanten Faktor erhöht, gleichzeitig wird aber ein größerer Hypothesenraum durchsucht.

In [19] gibt es einen Überblick auch über andere Suchalgorithmen wie *best-First search*, *stochastic search* oder *genetic algorithms*.

Den größten Einfluss hat aber die verwendete Such-Heuristik. Die Heuristik bewertet die Güte einer Kandidatenregel und entscheidet somit, welche Regeln der Theorie zugefügt und welche verworfen werden. In der Klassifikation verwenden die gebräuchlisten Heuristiken bestimmte Eigenschaften der Kandidatenregel, wie das Verhältnis von positiven zu negativen Beispielen. Es werden folgende Größen verwendet:

- P: Anzahl aller positiven Beispiele
- N: Anzahl aller negativen Beispiele
- r: Die Kandidatenregel
- p: Die durch r abgedeckten positiven Beispiele
- n: Die durch r abgedeckten negativen Beispiele

Accuracy (3) betrachtet eine Regel unabhängig von anderen Regeln. Es wird die Anzahl der positiv abgedeckten Beispiele im Verhältnis zu den nicht abgedeckten negativen Beispielen berechnet, wobei das Ergebnis durch die Anzahl aller vorhandenen Beispiele normiert wird. So bedeutet eine Accuracy von 1, dass eine Regel alle positiven Beispiele und keine negativen Beispiele abdeckt. Accuracy findet z.B. in I-REP [20] Verwendung.

$$A(r) = \frac{p + (N - n)}{P + N} \tag{3}$$

Eine andere Heuristik, wie sie z.B. in RIPPER verwendet wird, ist die Minimum Description Length (MDL). Hier wird versucht ein Trade-Off zwischen der Komplexität und der Accuracy einer Theorie zu finden. Dabei wird die Anzahl der Bits gemessen, die gebraucht werden um die Hypothese und die mit dieser Hypothese beschriebenen Beispiele zu kodieren. Ziel ist es die Anzahl der Bits zu minimieren.

Bei reinen Regressionsproblemen können diese Heuristiken nicht verwendet werden, da es keine positiven und negativen Beispiele gibt. Eine Übersicht der gebräuchlichsten Heuristiken für Klassifikation findet sich in [19].

2.2.3 Konzepte zur Vermeidung von Überbestimmtheit

Ein weiteres Problem von Regel-Lernern ist Überbestimmtheit (engl. Overfitting). Man spricht von Overfitting, wenn sich die Theorie zu sehr an die Trainingsmenge angepasst hat. Im speziellsten Fall gibt es für jedes Beispiel in der Trainingsmenge genau eine Regel, die auch nur dieses Beispiel abdeckt. Diese These ist korrekt, d.h. auf der Trainingsmenge wird immer die richtige Vorhersage getroffen. Die Vorhersage von unbekannten Beispielen ist mit dieser Theorie aber denkbar schlecht. Beispiele, die sehr ähnlich zu den schon vorhandenen sind, werden nicht erkannt und somit auch nicht richtig vorhergesagt.

Overfitting kann reduziert werden, indem man annähernd komplette und konsistente Theorien bevorzugt, die dafür aber deutlich einfacher sind. Einfacher heißt hier weniger und kürzere Regeln. Durch das Kürzen von Regeln werden diese genereller, es werden mehr Beispiele abgedeckt. Im Kontext von Overfitting spricht man auch von „Pruning“. Oftmals liefern diese Theorien bessere Vorhersagen auf unbekannten Beispielen.

Wird Overfitting schon während des Suchens vermieden spricht man von „Pre-Pruning“ im Gegensatz zum „Post-Pruning“, wo zuerst die Theorie erstellt wird und dann anschließend geprunt wird. Auch die Kombination beider Varianten ist möglich.

Beim Pre-Pruning wird Overfitting schon direkt bei der Erstellung der Theorie beachtet. Die Idee dabei ist die Verfeinerung von Kandidatenregeln rechtzeitig zu stoppen, auch auf die Gefahr hin, dass dadurch die Regeln zu generell werden.

Es kann aber auch eine Theorie erst nachträglich beschnitten werden. Dies wird beim Post-Pruning gemacht. So werden oftmals sich wiederholende Bedingungen im Regelkörper entfernt, aber auch unnötige Regeln. Eine Regel ist unnötig, wenn sich durch die Löschung dieser die Genauigkeit der Theorie nicht verschlechtert.

Durch Post-Pruning werden meist bessere Theorien als beim Pre-Pruning gefunden, allerdings zu Lasten der Laufzeit. Im Grow Algorithmus [7] wird deshalb der Post-Pruning Ansatz verfeinert. So wird die Trainingsmenge in eine kleinere Pruning-Menge und eine größere Growing-Menge aufgespalten. Auf der Growing-Menge wird gelernt. Das anschließende Post-Pruning und vor allem die Berechnungen dafür geschehen nur auf der Pruning-Menge.

Man kann auch beide Varianten miteinander kombinieren, so dass zuerst während des Lernens geprunt wird und die fertige Theorie dann nochmals in einer Post-Pruning Phase angepasst wird. Während der Pre-Pruning Phase soll Overfitting nicht total vermieden werden, die Theorie soll nur schlanker für die Post-Pruning Phase gemacht werden. Die Gefahr dabei ist aber

immer, dass in der Pre-Pruning Phase die Theorie zu sehr vereinfacht wird. Im von Fürnkranz vorgeschlagenen Algorithmus Top-Down Pruning [18] wird dieser Ansatz verfolgt.

2.3 Verschiedene Algorithmen

In diesem Abschnitt werden exemplarisch einige Separate-and-Conquer Algorithmen vorgestellt. All diese Algorithmen unterscheiden sich durch verschiedene Ausprägungen der Separate-and-Conquer Strategie.

2.3.1 AQ

Die AQ-Algorithmen sind eine Reihe von Algorithmen, die von Michalski in [27] und von Michalski, Mozetic, Hong und Lavrac in [28] vorgestellt werden. Diese Algorithmen gibt es in verschiedenen Ausprägungen, hier wird nur die grundsätzliche Strategie erklärt.

Für jede zu lernende Klasse bildet der AQ-Algorithmus ein eigenes Set von Regeln. Für die Klassifikation werden nun alle Regeln herangezogen, die dieses Beispiel abdecken. Wird das Beispiel nur von einer Regel abgedeckt, so wird es mit der Klassifikation versehen, die die Regel vorhersagt. Wird das Beispiel aber von mehreren Regeln abgedeckt, so bekommt es die Klasse zugewiesen, die von den meisten dieser Regeln vorhergesagt wird. Wenn das Beispiel von keiner Regel abgedeckt wird, bekommt es die Klasse zugewiesen, welche auf der Trainingsmenge am häufigsten vorkommt.

Der AQ-Algorithmus verfolgt die Separate-and-Conquer Strategie. Es gibt eine innere und eine äußere Schleife. Die äußere Schleife wird solange durchlaufen, bis es keine positiven Beispiele in der Trainingsmenge gibt. Es wird zufällig ein Beispiel, das Seed, als Startpunkt für den Conquer-Schritt gewählt, der eine Liste mit Regeln zurück gibt. Aus diesem wird die beste Regel durch eine Heuristik ausgewählt, wobei als Auswahlkriterium eine hohe Abdeckung von positiven Beispielen dienen kann (engl. coverage). Diese Regel wird der Theorie hinzugefügt und die abgedeckten Beispiele werden aus der Trainingsmenge entfernt.

Im Conquer-Schritt wird die Bottom-Up Strategie verwendet. Es wird aus dem zufällig ausgewählten Beispiel eine Regel gebildet, die nur dieses Beispiel abdeckt. Dann wird schrittweise generalisiert, d.h. es werden Konditionen aus der Regel entfernt, so dass trotzdem nur positive Beispiele abgedeckt werden. Ist keine Generalisierung mehr möglich werden die generalisierten Regeln zurückgegeben. Dieser Algorithmus findet immer eine auf der Trainingsmenge korrekte Theorie.

2.3.2 CN2

Der CN2 Algorithmus [6] bedient sich auch der Separate-and-Conquer Strategie. CN2 kann mit mehreren Klassen umgehen, während des Conquer-Schrittes sucht er Regeln, die möglichst viele Beispiele der einen Klasse und möglichst wenige der anderen Klassen abdecken. Die gefundene Regel wird an eine geordnete Entscheidungsliste angehangen und die abgedeckten Beispiele

werden entfernt. Dies wird solange ausgeführt, bis keine neuen Regeln mehr gefunden werden, bzw. bis die Heuristik abbricht. Weil unterschiedliche Klassen vorhergesagt werden, muss die Heuristik im Separate Schritt entsprechend angepasst werden. So werden die Regeln auf Signifikanz geprüft. Eine Regel ist signifikant, wenn sich die Verteilung der abgedeckten Beispiele signifikant von der Verteilung zufälliger Beispiele unterscheidet. Für den Suchalgorithmus wird Beam-Search mit der Top-Down Strategie verwendet.

2.3.3 Ripper

RIPPER [8] ist ein Regel-Lerner, der auf dem in [20] vorgestellten Konzept Incrementeal Reduced Error Pruning (IREP) beruht. Bei IREP wird die Trainingsmenge in eine Growing-Menge ($2/3$) und in eine Pruning-Menge ($1/3$) aufgeteilt. Anstatt erst die gesamte Theorie zu lernen und danach zu prunen, wird jede einzelne Regel direkt nach ihrer Erzeugung geprunt. Der fertigen Regel werden solange Bedingungen entfernt, bis jede weitere Entfernung die Accuracy auf der Pruning-Menge verringert. Die so gefundene Regel wird der Theorie hinzugefügt und alle abgedeckten Beispiele werden von der Growing- und Pruning-Menge entfernt. Die verbleibenden Beispiele werden dann wieder in eine Growing- und Pruning-Menge aufgeteilt und eine neue Regel kann gelernt werden. Der Algorithmus stoppt, sobald der Fehler der geprunten Regel $> 50\%$ auf der Pruning-Menge wird. IREP kann in verschiedenen statischen Repräsentationssprachen implementiert werden, verwendet Hill-Climbing für die Suche sowie die Top-Down Strategie und Accuracy als Suchheuristik. Um Overfitting zu vermeiden wird sowohl Pre- als auch Postpruning verwendet.

IREP tendiert aufgrund seines Abbruchkriteriums zum frühzeitigen Abbruch, wie in [8] gezeigt wurde. So besteht die Möglichkeit, dass IREP frühzeitig abbricht, wenn komplexe Regeln gefunden werden, die nur wenige positive Beispiele auf der Grow-Menge abdecken.

Deshalb benutzt Cohen in RIPPER die MDL als Abbruchkriterium. Die MDL wird nach jedem Hinzufügen einer neuen Regel gemessen. Ist die neu gemessene MDL um einen konstanten Faktor größer als die bisher kleinste gefundene MDL stoppt RIPPER.

2.4 Regression

Im Abschnitt Regression werden die Unterschiede zu der vorher vorgestellten Klassifikation erläutert.

In praktischen Anwendungen kommen oftmals numerische Klassenvariablen vor. Klassifikation kann da meist nur sehr eingeschränkt verwendet werden, da es oftmals keine Möglichkeit gibt den numerischen Beispielen Klassen zuzuordnen. Abgesehen davon können Regressionslerner einen konkreten Wert vorhersagen, mit Klassifikation können nur Intervalle angegeben werden. Möchte man beispielsweise die Höhe eines Kredits vorhersagen, so ist eine tatsächliche Größe aussagekräftiger als Kategorien wie „klein“, „mittel“ oder „groß“.

Viele Methoden der Klassifikation können bei Regressionsproblemen nicht verwendet werden. Hier müssen andere Strategien benutzt werden. So kann keine der Heuristiken aus der Klassifikation verwendet werden. Stattdessen wird z.B. der Fehler der Vorhersage berechnet.

Die verwendeten Methoden zur Bestimmung von Fehlern werden im Abschnitt 2.6 erläutert. Andere Konzepte wie z.B. die Suchstrategien und Suchalgorithmen werden auch in der Regression verwendet. Die Repräsentationssprachen ändern sich insofern, als dass keine Klassen mehr vorhergesagt werden, sondern ein numerischer Wert. Das Prunen von Regeln gestaltet sich schwieriger. Da durch das Prunen mehr Beispiele abgedeckt werden, muss die Vorhersage der Regel neu berechnet werden.

2.4.1 Regression in der Statistik

Die Vorhersage von numerischen Werten wird in der Statistik Regression genannt. Es sollen durch mathematische Verfahren Funktionen gefunden werden, die mit möglichst geringem Fehler die Daten abbilden. So werden z.B. Lineare Regression, nicht-parametrische Regression wie Multivariante Adaptive Regressions-Splines [13] und semi-parametrische Regression wie Projection Pursuit Regression [17] verwendet. Hier wird stellvertretend nur die Lineare Regression vorgestellt, da das Hauptaugenmerk dieser Arbeit auf Regressionsregellernern liegt.

Bei der einfachen Linearen Regression wird nach einem linearen Zusammenhang auf der Datenmenge gesucht. Die Daten liegen in der Form (x_i, y_i) vor, wobei x_i die Eingabe und y_i die Ausgaben der Daten sind. Es wird ein linearer Zusammenhang zwischen x_i und Y_i vermutet, wobei Y_i die Vermutung und y_i die tatsächliche Ausprägung ist. Es wird als Modell $Y_i = \beta_0 + \beta_1 x_1 + \epsilon_i$ angenommen. Dabei versucht man β_0 und β_1 zu bestimmen. Es wird also ein Gerade durch die Datenpunkte (x_i, y_i) gelegt, die einen möglichst geringen Abstand zu allen Punkten hat. Durch β_0 und β_1 wird der Ursprung und die Steigung der Geraden bestimmt. Dabei kann der Abstand durch verschiedene Methoden bestimmt werden, wie z.B. der Minimum-Quadrat-Methode.

Wird als Eingabe nicht nur x_i sondern mehrere Variablen verwendet, spricht man von multipler Regression. Solch eine multiple Regression wird auch im Kapitel 4 verwendet.

2.5 Regression in maschinellem Lernen

In den folgenden Abschnitten werden sowohl nicht regelbasierte als auch regelbasierte Regressionsalgorithmen vorgestellt. Dabei wird kurz auf die Unterschiede zwischen den verschiedenen Ansätzen eingegangen.

2.5.1 Nicht regelbasierte Ansätze

Hier werden einige Regressionslerner beschrieben, die nicht auf Regeln beruhen. So gibt es den Nearest Neighbour Algorithmus [24], der die Abstände eines Beispiels zum Nächsten misst und somit Ähnlichkeiten auf Datenmengen erkennt. Dabei werden alle Beispiele gespeichert und die Berechnung der Abstände erfolgt erst während der Vorhersage. Auch Locally Weighted Regression [5] funktioniert ähnlich.

Eine andere Methode verwendet MultilayerPerceptron [33]. MultilayerPerceptron ist ein Neuronales Netz [4] mit mehreren nicht rückgekoppelten Schichten. Neuronale Netzwerke können meist gute Ergebnisse erzielen, allerdings kann man das Modell schwer interpretieren.

Für die Tests im Kapitel 4 wird auch SVMreg verwendet. Dies ist eine Support Vector Machine für Regressionsprobleme. Die Beispiele werden als Vektoren in einem Vektorraum dargestellt. Die SVM soll im Vektorraum eine Hyperebene finden, die die Beispiele möglichst gut voneinander trennt. Eine Umsetzung der ursprünglich für Klassifikation genutzten SVM für Regressionsprobleme wird in [34] beschrieben.

2.5.2 Regelbasierte Ansätze

Grundsätzlich kann man zwischen zwei Ansätzen unterscheiden. Bei dem in Abschnitt 2.5.2.1 vorgestellten Algorithmus wird die Regression auf eine Klassifikation zurückgeführt, die dann gelöst wird. Dies kann vorteilhaft sein, da das Feld der Klassifikation sehr gut untersucht ist und es sehr leistungsfähige Klassifikationslerner gibt. Durch die Diskretisierung gehen aber immer auch Informationen verloren. Hier werden mit verschiedenen Methoden die numerischen Klassenwerte der Beispiele durch eine Klasse ersetzt. In den folgenden zwei Abschnitten werden unterschiedliche Ansätze dafür gezeigt.

Die andere Methode geht nicht den Umweg über die Klassifikation sondern löst das Regressionsproblem direkt. Die meisten der hier vorgestellten Regressionsregellerner verfolgen diese Strategie.

2.5.2.1 Diskretisierung der Klassenvariable

Weiss und Indurkhya stellen in [37] den P-Class Algorithmus vor, der zuerst Pseudo-Klassen erstellt und diese den Beispielen zuordnet und danach klassifiziert. Die Beispiele werden nach ihrem Klassenattribut sortiert und jeweils gleich viele Beispiele einer Pseudo-Klasse zugewiesen. Die Vorhersage der Pseudo-Klasse ist dabei der Durchschnitt aller enthaltenen Beispiele. Ein Beispiel wird in eine benachbarte Klasse verschoben, wenn dadurch der Fehler auf allen Beispielen sinkt. Klassen mit gleichen Vorhersagen werden vereinigt. Während der Klassifikation wird dieses Multiklassenproblem auf ein Binäres reduziert, indem alle Klassen nacheinander behandelt werden. Beispiele der aktuellen Klasse sind dabei positiv, alle anderen negativ.

2.5.2.2 Dynamische Reduktion auf Klassifikation

In [23] wird Regression durch Klassifikation gelöst. Hierbei werden alle Beispiele, die in der Nähe der Vorhersage liegen, als positiv markiert und alle anderen als negativ. In jedem Verfeinerungsschritt werden die Beispiele neu markiert. Dadurch verändern sich mit jeder Verfeinerung die positiven und negativen Beispiele dynamisch. Zur Evaluation der Regeln können die bekannten Heuristiken für Klassifikation genutzt werden.

2.5.2.3 M5Rules

M5Rules [12] ist kein reiner Regellerner. So wird auf den Beispielen zuerst ein Model-Tree [31] gelernt. Bäume haben den Vorteil, dass durch ihre Struktur besser geprunt werden kann, allerdings haben sie auch den Nachteil, dass die Theorie nicht so kompakt dargestellt werden kann wie mit Regeln. Aus dem geprunten Baum wird das beste Blatt ausgewählt und daraus eine Regel erstellt. Der Baum wird verworfen, alle durch die Regel abgedeckten Beispiele werden entfernt und der nächste Baum wird auf den verbleibenden Beispielen gelernt. Die Regel wird in einer Entscheidungsliste gespeichert. Dies wiederholt sich so lange, bis alle Beispiele durch mindestens eine Regel abgedeckt sind. Dieser Lerner verfolgt somit auch die Separate-and-Conquer Strategie, nur werden hier im Conquer-Schritt keine Regeln gelernt sondern Bäume. Somit können die Vorzüge von Bäumen mit denen von Regeln kombiniert werden.

2.5.2.4 R^2

R^2 findet Regeln in mehreren Schritten. Im ersten Schritt werden Regionen auf der Beispielmenge gesucht, die noch von keiner Regel abgedeckt sind. Dabei werden von den schon gefundenen Regeln alle Konditionen negiert. Die Kondition, welche die meisten nicht abgedeckten Regeln findet wird verwendet. So passiert es, dass mehrere Regeln die gleichen Beispiele abdecken, R^2 bildet also verschiedene Regel-Modelle für die gleichen Daten.

Nun muss ein Regressionsmodell für die gefundene Kondition erstellt werden. R^2 hat dafür eine Menge von immer komplexer werdenden Sprachen. Mit jeder Sprache wird ein Regressionsmodell erstellt, die Sprache mit dem geringsten Fehler wird verwendet.

Dieses Regressionsmodell wird spezialisiert, indem durch Hinzufügen von Bedingungen versucht wird das am schlechtesten beschriebene Beispiel aus dem Regressionsmodell zu entfernen. Eine Heuristik vergleicht die Güte der Spezialisierung mit der ursprünglichen Regel und entscheidet, welche besser ist. Ist die spezialisierte Regel besser, wird diese weiter spezialisiert bis keine weiteren Verbesserungen mehr möglich sind.

2.5.2.5 RuleFit

RuleFit [16] basiert auf der Ensemble-Technik. Hier gibt es verschiedene Ansätze wie Random Forests [3] und Bagging [2]. All diesen Methoden ist gemein, dass die Beispiele nicht durch eine einzige Regel vorhergesagt werden sondern durch alle Regeln, die dieses Beispiel abdecken. Es wird ein ganzes „Ensemble“ von Regeln zur Vorhersage genutzt. Das Ensemble in RuleFit wird wie in [15] beschrieben durch Entscheidungsbäume erstellt.

2.5.2.6 RegENDER

RegENDER [10] verwendet auch Ensembles. Das Ensemble wird mit der Default-Regel, die alle Beispiele abdeckt initialisiert. Dann werden in jedem Schritt des Algorithmus weitere Regeln

hinzugefügt, wobei jede neue Regel nur im Kontext der anderen Regeln im Ensemble erstellt wird. Die Regel, die den geringsten Fehler hat wird dem Ensemble hinzugefügt. Eine Regel wird wie folgt erstellt. Im ersten Schritt werden aus allen Konditionen Regeln erstellt. Die Regel, welche den geringsten Fehler auf der Beispielmenge hat wird im nächsten Schritt weiter verfeinert, indem nun Regeln aus dieser Kondition gepaart mit allen anderen möglichen Konditionen erstellt werden. Dies wiederholt sich so lange bis keine weitere Verbesserung mehr möglich ist. RegENDER gibt es in 4 Ausprägungen. So kann der absolute und der quadratische Fehler mit gradient boosting [14] und least angle technique [25] kombiniert werden.

2.5.2.7 SeCoReg

SeCoReg [22] ist eine Adaption des in dieser Arbeit vorgestellten 2.1 auf Regression. Es wird die Top-Down Strategie mit Hill-climbing verwendet. Für die Fehlerberechnung wird eine Kombination von RRSE (13) und der Abdeckung verwendet. Durch einen Parameter kann man dabei das Verhältnis zwischen RRSE und Abdeckung steuern. Weiterhin wird eine neue Art für die Berechnung der Splitpoints verwendet.

2.6 Evaluierung von Regressionsalgorithmen

In der Klassifikation beruhen Heuristiken auf positiven und negativen Beispielen, wie es im Abschnitt 2.2.2 vorgestellt wird. Da es diese in der Regression nicht gibt, wird hier ein anderes Verfahren verwendet. Um zu entscheiden wie gut eine Regel ist, wird der Fehler auf den Beispielen verwendet. Allerdings sollte man beachten, dass nicht auf den Beispielen evaluiert wird, die fürs Lernen verwendet werden. Ansonsten ist die Gefahr sehr hoch, dass auf den Beispielen der Lernmenge sehr gute Ergebnisse erzielt werden, die Vorhersage für unbekannte Beispiele aber sehr schlecht ist.

Die Vorhersage einer Regel wird durch eine Schätzfunktion berechnet. Oftmals wird der Durchschnitt aller abgedeckten Beispiele für die Vorhersage verwendet. Es gibt aber auch andere Methoden zur Berechnung der Vorhersage wie den Median.

Weiter unten werden einige Funktionen beschrieben, mit denen man die Güte eines Schätzers messen kann. Dabei wird der Fehler des Schätzers berechnet.

In diesem Abschnitt werden folgende Bezeichner verwendet:

- n: Anzahl der (abgedeckten) Beispiele
- $i \in \{1, ..., n\}$: Index für das i-te Beispiel
- y_i: tatsächlicher Wert des i-ten Beispiels
- $\bar{y}_i$: Vorhersage für Beispiel i
- $\acute{y}$: Mittelwert von allen (abgedeckten) Beispielen

Für die Vorhersage der Regeln werden hier zwei Schätzer vorgestellt. Am häufigsten wird das arithmetische Mittel (engl. mean) verwendet. Hier werden die tatsächlichen Werte aller

abgedeckten Beispiele addiert und durch die Anzahl der abgedeckten Beispiele geteilt. In nahezu allen Regressionsregellernern wird diese Methode zur Vorhersage der Regeln benutzt.

$$mean = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{4}$$

Man kann auch den Median (5) verwenden. Hierbei werden alle Beispiele nach ihrem Wert sortiert und der Wert des mittleren Beispiels wird als Vorhersage für alle abgedeckten Beispiele verwendet. Der Median ist sehr robust gegenüber Ausreißern, da diese ignoriert werden.

$$\text{median} = \begin{cases} y_{(\frac{n+1}{2})}, & \text{n ungerade,} \\ \frac{1}{2}\left(y_{(\frac{n}{2})} + y_{(\frac{n}{2}+1)}\right), & \text{n gerade.} \end{cases} \tag{5}$$

Nun werden die verschiedenen Methoden zur Berechnung von Fehlern erläutert. Prinzipiell können diese mit beiden vorgestellten Methoden zur Ermittlung des Mittelwertes verwendet werden, allerdings wird fast ausschließlich das arithmetische Mittel verwendet.

Der mittlere absolute Fehler (engl. mean absolute error - MAE) wird aus dem Durchschnitt der absoluten Fehler berechnet.

$$L_{MAE} = \frac{1}{n}\sum_{i=1}^{n} \left|y_i - \bar{y}_i\right| \tag{6}$$

Die Mittlere absolute Abweichung bezüglich des Medians berechnet sich ähnlich wie der MAE, hier wird der Median verwendet.

$$L_{MD} = \frac{1}{n}\sum_{i=1}^{n} \left|y_i - median\right| \tag{7}$$

Die mittlere quadratische Abweichung (engl. mean squared error - MSE) berechnet den Durchschnitt der quadrierten Fehler der abgedeckten Beispiele. Der MSE wird als Fehlerkriterium in Regressionsalgorithmen oft benutzt.

$$L_{MSE} = \frac{1}{n}\sum_{i=1}^{n} \left(y_i - \bar{y}_i\right)^2 \tag{8}$$

Mit Varianz (engl. variance) wird die Summe der quadratischen Fehler bezüglich des Durchschnitts bezeichnet.

$$L_{VAR} = \frac{1}{n}\sum_{i=1}^{n} \left(y_i - \acute{y}\right)^2 \tag{9}$$

Zieht man vom MSE die Wurzel erhält man den root mean squared error (RMSE)

$$L_{RMSE} = \sqrt{L_{MSE}} \tag{10}$$

All diese Funktionen berechnen den Fehler des Schätzers. Allerdings haben alle einen großen Nachteil - sie sind immer von der Anwendungsdomäne abhängig. So kann man die Ergebnisse für eine Datenmenge nicht mit denen für andere Datenmengen vergleichen. Der Fehler ist auf Datenmengen, deren Klassenattribut sich im Intervall $[0, 1]$ befindet, deutlich geringer als Datenmengen mit einem Klassenattribut im Intervall $[1000, 10000]$. Normiert man diese Funktionen, kann man dieses Problem umgehen. Die Werte befinden sich im Intervall $[0, 1]$ und können somit als Prozentwert ausgegeben werden. Alle Werte der relativen Funktionen in Kapitel 4, also vor allem der RRSE, werden prozentual angegeben.

Der relative Standardfehler (engl. relative standard error - RSE) normiert den MSE durch die Summe der quadratischen Abweichungen vom Mittelwert.

$$L_{RSE} = \frac{\sum_{i=1}^{n} \left(y_i - \bar{y}_i\right)^2}{\sum_{i=1}^{n} \left(y_i - \acute{y}\right)^2} \tag{11}$$

Der relative absolute Fehler (engl. relative absolute error - RAE) wird durch die Summe der absoluten Abweichungen vom Mittelwert normiert.

$$L_{RAE} = \frac{L_{MAE}}{\frac{1}{n} \sum_{i=1}^{n} \left|y_i - \acute{y}\right|} \tag{12}$$

Der root relative standard error (RRSE) wird durch die Wurzel des RSE berechnet. Mit dieser Funktion werden im Kapitel 4 die verschiedenen Tests evaluiert.

$$L_{RRSE} = \sqrt{L_{RSE}} \tag{13}$$

3 Reduce-and-Conquer

Im Kapitel 3 werden sowohl der Algorithmus Separate-and-Conquer als auch die zugrunde liegenden Ideen vorgestellt. Es wird gezeigt, wie Reduce-and-Conquer (ReCo) von Separate-and-Conquer Algorithmen beeinflusst ist. Es werden aber auch die Unterschiede zwischen der Separate- und der Reduce-Strategie aufgezeigt. Es folgt eine Skizze des Algorithmus mit besonderem Augenmerk auf die verschiedenen Charakterisierungen aus Abschnitt 2.2, sowie ein Überblick über die verschiedenen Konfigurationen des Algorithmus.

3.1 Reduce-and-Conquer Regel-Lernen

SeCo Algorithmen sind ein gut erforschtes Gebiet im maschinellen Lernen. Dabei liegt der Fokus vor allem auf Klassifikationsalgorithmen. In ReCo wird eine neue Methode zur Adaption von SeCo auf Regression implementiert. Hierbei werden Verfahren von RIPPER, RegENDER und eine neuartige SeCo-Strategie verwendet.

SeCo beruht auf den zwei Schritten Entfernen (Separate) und Erobern (Conquer). Die Grundidee hinter Reduce-and-Conquer ist sehr ähnlich. In der inneren Schleife werden genau wie bei SeCo neue Bereiche der Datenmenge „erobert“. Im Reduce-Schritt werden aber nicht wie bei SeCo die abgedeckten Beispiele der gefundenen Regel des Conquer-Schrittes aus der Trainingsmenge entfernt, vielmehr werden diese Beispiele selber verändert.

3.1.1 Die Reduce Strategie

In SeCo Algorithmen werden alle Beispiele, die von einer Regel abgedeckt werden gelöscht. Für die weitere Entwicklung der Theorie spielen sie keine Rolle mehr. Dies ist bei ReCo anders. Die abgedeckten Beispiele verbleiben in der Trainingsmenge. Es wird allerdings das Klassenattribut dieser Beispiele verändert. Eine Regel sagt immer den Durchschnitt aller abgedeckten Beispiele vorher. Dieser Wert wird von dem Klassenattribut der abgedeckten Beispiele abgezogen. Soll das Zielattribut von unbekannten Beispielen vorhergesagt werden, so werden die Vorhersagen aller Regeln, die dieses Beispiel abdecken, addiert. Der so gefundene Wert ist die Vorhersage dieses Beispiels. Dadurch kann auf eine sortierte Regelliste verzichtet werden, wo die Regeln eingefügt werden spielt keine Rolle. Alle Beispiele, die von keiner anderen Regel abgedeckt werden, werden von der Default-Regel abgedeckt. Die Default-Regel deckt alle Beispiele ab und sagt den Durchschnitt dieser vorher.

Der Algorithmus startet anders als bei SeCo nicht mit einer leeren Theorie, sondern mit der Default-Regel. Dadurch wird gleich zu Beginn von allen Beispielen der Durchschnittswert abgezogen. Der Durchschnitt aller Beispiele ist danach Null. Als Schätzfunktion wird der MSE verwendet. Dabei werden die Abweichungen der Vorhersagen der einzelnen Beispiele vom tatsächlichen Wert berechnet. Durch das Starten der Theorie mit der Default-Regel ist die Vorhersage jedes Beispiels immer der Wert Null. Deshalb ist der tatsächliche Wert eines Beispiels zugleich auch seine Abweichung vom zu schätzenden Wert. Eine Theorie wird dabei besser, wenn der MSE auf den Beispielen sinkt. Der MSE wird immer entweder von allen Beispielen der

Pruning-Menge oder allen Beispielen der Growing-Menge berechnet, je nachdem in welchem Schritt sich der Algorithmus gerade befindet. Wenn eine Regel evaluiert werden soll, wird zunächst von allen abgedeckten Beispielen temporär die Vorhersage der Regel abgezogen. Dann wird der MSE von allen Beispielen berechnet. Es wird also durch jede Regel die Menge der Beispiele verändert.

Durch dieses Vorgehen sinkt sukzessive mit jeder gefundenen Regel die Varianz der Beispiele. Separate-and-Conquer kann man so interpretieren, dass die Informationen der entfernten Regeln auf die Regelmenge übergeht, da jede Regel die von ihr entfernten Beispiele beschreibt. In ReCo verhält es sich ähnlich, jede Regel beinhaltet Teilinformationen der von ihr abgedeckten Beispiele. Allerdings kann eine Regel allein nicht alle von ihr abgedeckten Beispiele beschreiben, dazu sind auch alle anderen Regeln notwendig, die diese Beispiele abdecken. Eine einzelne Regel kann deshalb immer nur im Kontext aller anderen Regeln betrachtet werden. Wenn alle Regeln, die ein Beispiel abdecken zur Vorhersage verwendet werden, spricht man auch von Ensemblelernen. Ein ähnlicher Ansatz wird in RegENDER verwendet.

3.1.2 Adaption des generischen Separate-and-Conquer Algorithmus

Der Reduce-and-Conquer Algorithmus basiert auf dem in [19] beschriebenen generischen Separate-and-Conquer Algorithmus. Die Änderungen kann man in drei Bereiche aufteilen:

- Umsetzung der Reduce-Strategie
- Umsetzung einer zu RIPPER ähnlichen Pruning Strategie
- Benutzen von Ensembles

In der Funktion SeparateAndConquer des generischen Separate-and-Conquer Algorithmus muss der Seperate Schritt durch den oben beschriebenen Reduce Schritt ersetzt werden. Dadurch ändert sich auch die „while-Schleife“, da Beispiele jetzt nicht mehr entfernt werden, sondern immer in der Pruningmenge verbleiben. Ferner wird durch die Pruning-Strategie sowohl die Growing- und Pruningmenge eingeführt als auch die Liste der Verfeinerungsregeln. Auch startet der Algorithmus nicht mit der leeren Theorie, sondern mit der Default-Regel, welche alle Bespiele abdeckt.

Bevor Regeln gefunden werden können, muss die Menge der Beispiele „gesäubert“ werden. So werden alle Beispiele ohne Klassenattribut ignoriert. Auch alle Beispiele ohne numerisches Klassenattribut werden vorher entfernt. Beispiele mit Attributen ohne zugehörigen Wert werden allerdings nicht vorher behandelt, sondern immer erst direkt im Conquer-Schritt. Wird solch ein Attribut mit fehlendem Wert von der Regel getestet, wird das Beispiel ignoriert.

Des weiteren wird wie bei RIPPER auch in ReCo die Menge der Beispiele in eine Growing-Menge und eine Pruning-Menge aufgeteilt. Auf der Growing-Menge werden die Regeln gelernt, die Pruning-Menge dient dann zum anschließenden Prunen der Regeln, um das Risiko von Overfitting zu vermindern. Dabei wird wie bei RIPPER $2/3$ der Beispiele für die Growing-Menge und $1/3$ der Beispiele für die Pruning-Menge verwendet. Das Aufteilen der Beispiele ist in SplitExamples realisiert. Der Frage, ob eine Durchmischung der beiden Mengen nach jeder gefundenen Regel sinnvoll ist, wird im Abschnitt 4.3.3 nachgegangen.

Da nicht mit einer leeren Theorie gestartet wird, muss *DefRule* initialisiert werden. Dabei wird in InitializeRule die Default-Regel initialisiert, d.h. es werden alle Beispiele von dieser Regel abgedeckt. Diese Regel sagt den Durchschnitt aller Beispiele aus der Pruning- und der Growing-Menge vorher. In ReduceExampleValues wird von allen Beispielen der Durchschnitt abgezogen, so dass der Durchschnitt aller Beispiele Null ergibt. Die Default-Regel wird dann auf der Pruning-Menge evaluiert, indem der MSE berechnet wird. Die Evaluation wird erst in der äußeren Schleife in RuleStoppingCriterion verwendet. Die Default-Regel wird der Theorie hinzugefügt. Die Theorie sagt also im schlechtesten Fall den Durchschnitt aller Beispiele vorher.

In der äußeren Schleife werden die im Conquer Schritt FindRefinementRules gefundenen Verfeinerungsregeln auf der Pruning-Menge evaluiert. Es wird für jede Verfeinerungsregel temporär die Vorhersage der Regel von den abgedeckten Beispielen in der Pruning-Menge abgezogen und der MSE berechnet. Die Verfeinerungsregel, welche den geringsten MSE auf der Pruning-Menge erzielt, wird in *BestRule* gespeichert. In FindBestRefinementRule wird dies realisiert.

In der ersten Version des Algorithmus sollte *BestRule* der Theorie nur dann zugefügt werden, wenn der MSE mit dieser Regel geringer ist als der MSE ohne diese Regel. Ansonsten soll der Algorithmus abbrechen und die Theorie ausgeben, da ja augenscheinlich keine Verbesserung mehr gefunden werden kann. Dies funktioniert so nicht, so gibt es Regeln, die auf der Growing-Menge einen besseren MSE erzielen, auf der Pruning-Menge allerdings einen schlechteren. Dies kommt durch die zufällige Verteilung der Beispiele in beiden Mengen zustande. Eine später gefundene Regel kann aber durchaus wieder den MSE auf der Pruning-Menge verbessern. In der Variante bricht der Algorithmus zu früh ab. Deshalb werden in RuleStoppingCriterion verschiedene Varianten des Abbruchkriteriums implementiert, die im Abschnitt 3.3 beschrieben werden.

Falls das RuleStoppingCriterion nicht feuert, wird *BestRule* zu der Theorie hinzugefügt. Nun müssen noch von allen abgedeckten Beispielen, also sowohl die der Pruning- als auch die der Growing-Menge, die Vorhersage von *BestRule* von den Klassenattributen abgezogen werden (ReduceExampleValues), dies ist der eigentliche Reduce-Schritt. Die Beispielmengen haben sich verändert, so dass im nächsten Durchlauf der Schleife andere Regeln gefunden werden können.

Im „Conquer-Schritt“ werden die Regeln gelernt. Gelernt wird immer nur auf der Growing-Menge. Es wird auch im Conquer-Schritt mit der Default-Regel begonnen, die alle Beispiele abdeckt. Die Evaluation der Default-Regel findet allerdings auf der Growing-Menge statt. Während des Verfeinerns werden aufeinander aufbauende Regeln gefunden, die in der Liste *Rules* gespeichert werden. Die Liste wird mit der Default-Regel initialisiert und jede folgende Regel ist die beste Verfeinerung der vorhergehenden Regeln. So könnte eine im Conquer-Schritt gefundene Regelliste aussehen:

$$\begin{aligned} \{&(TRUE \rightarrow 0,52),\\ &(Wetter = sonnig \rightarrow 0,8),\\ &(Wetter = sonnig \wedge Temperatur \leq 15 \rightarrow 0,63),\\ &(Wetter = sonnig \wedge Temperatur \leq 15 \wedge Wind = ja \rightarrow 0,49),\} \end{aligned} \tag{14}$$

In *Candidate* wird die Kandidatenregel gespeichert, also die Regel welche verfeinert werden soll. Die erste Kandidatenregel ist die Default-Regel, sie ist auch zu Beginn von FindRefinementRules die beste Regel und wird in *BestRule* gespeichert.

Algorithm 2 Reduce-and-Conquer

```
procedure ReduceAndConquer(Examples)
GrowingSet = SplitExamples(Examples)
PruningSet = Examples \ GrowingSet
DefRule = InitializeRule(Examples)
ReduceExampleValues(Cover(DefRule, GrowingSet, PruningSet))
DefVal = EvaluateRule(DefRule, PruningSet)
DefRule = <DefVal, DefRule>
Theory = {DefRule}
loop
  RefinementRules = FindRefinementRules(GrowingSet)
  BestRule = FindBestRefinementRule(RefinementRules, PruningSet)
  if RuleStoppingCriterion(BestRule, PruningSet) then
    exit loop
  end if
  Theory = Theory ∪ BestRule
  ReduceExampleValues(Cover(BestRule, GrowingSet, PruningSet))
end loop
return Theory

procedure FindRefinementRules(GrowingSet)
DefVal = EvaluateRule(DefRule, GrowingSet)
DefRule = <DefVal, DefRule>
Rules = {DefRule}
Candidate = DefRule
BestRule = DefRule
loop
  Refinements = RefineRule(Candidate, GrowingSet)
  for Refinement ∈ Refinements do
    Evaluation = EvaluateRule(Refinement, GrowingSet)
    NewRule = <Evaluation, Refinement>
    if NewRule > BestRule then
      BestRule = NewRule
    end if
  end for
  if StoppingCriterion(BestRule, GrowingSet ) then
    exit loop
  end if
  Rules = Rules ∪ BestRule
  Candidate = BestRule
end loop
return Rules
```

In der „loop“-Schleife wird die Kandidatenregel verfeinert. Es werden alle Regeln, die durch eine Verfeinerung gebildet werden, erzeugt. Diese Regeln haben genau eine Bedingung mehr als die nicht verfeinerte Regel. Die Bedingungen werden entweder aus numerischen oder nominalen Attributen gebildet. Bei nominalen Attributen wird für jeden möglichen Wert, den dieses Attribut annehmen kann, jeweils eine Verfeinerungsregel erzeugt. Dabei wird darauf geachtet, dass keine doppelten Bedingungen innerhalb einer Regel erzeugt werden. Numerische Attribute werden durch einen Splitpoint geteilt. Es werden genau zwei Verfeinerungen erzeugt, jeweils eine für ($Attribut \leq Splitpoint$) und eine für ($Attribut > Splitpoint$). Splitpoints werden im Abschnitt 3.1.3 erläutert.

Jede der gefundenen Verfeinerungsregeln werden nun auf der Growing-Menge evaluiert, d.h. es werden temporär die Vorhersagen der einzelnen Regeln von den abgedeckten Beispielen abgezogen und anschließend der MSE auf den Beispielen der Growing-Menge berechnet. Für jede Verfeinerungsregel wird der MSE der bis dahin besten Regel (*BestRule*) mit dem MSE der Verfeinerungsregel verglichen. Ist der MSE der Verfeinerungsregel geringer, wird diese Regel in *BestRule* gespeichert. Nachdem also alle Verfeinerungsregeln, die durch einen Verfeinerungsschritt entstanden sind, verglichen wurden, befindet sich in *BestRule* die beste Verfeinerungsregel. Wenn der MSE von *BestRule* sich im Vergleich zum vorherigen Verfeinerungsschritt nicht verbessert hat, bricht die Verfeinerung ab. Ansonsten ist die soeben gefundene beste Regel die neue Kandidatenregel und wird weiter verfeinert.

Die gefundenen Verfeinerungsregeln aus den einzelnen Verfeinerungsschritten werden in der Verfeinerungsregelliste gespeichert. Diese wird zurückgegeben, wenn keine besseren Verfeinerungen mehr möglich sind. Die Liste der Verfeinerungen enthält genau so viele Regeln wie es Verfeinerungsschritte gibt. Es befindet sich zwar die Default-Regel in der Liste, so dass es eigentlich eine Regel mehr sein müsste, allerdings wird die Regel aus dem letzten Verfeinerungsschritt nicht in die Verfeinerungsregelliste eingetragen, da es ja in diesem Schritt zum Abbruch kommt und der MSE mit dieser Regel somit schlechter ist. Mit dem Ensemble von Regeln können nun unbekannte Beispiele vorhergesagt werden.

3.1.3 Splitpoints

Im Abschnitt 3.1.2 wurden Splitpoints erwähnt. Splitpoints teilen die Werte eines numerischen Attributes in zwei Mengen auf. Mit den Relationen $\leq$ und $>$ kann man somit Bedingungen erzeugen. Bedingt durch die Regression kann der Fehler nicht iterativ berechnet werden, so dass die effizienten Methoden zur Splitpointberechnung aus der Klassifikation hier nicht verwendet werden können.

Um Splitpoints zu erzeugen, werden die Werte eines Attributes aufsteigend sortiert. Alle doppelten Werte werden zu einem zusammengefasst. Nun wird der Mittelwert zwischen zwei aufeinander folgenden Werten ermittelt, dies wird für alle Werte gemacht. Somit hat man bei n Werten $n - 1$ Splitpoints. In Tabelle 1 wird dies an einem Beispiel verdeutlicht. Hierbei ist auch zu beachten, dass man mindestens zwei Beispiele mit unterschiedlichem Attributwert braucht um einen Splitpoint zu berechnen. Wenn dies nicht möglich ist, werden keine Splitpoints erzeugt und somit auch keine Konditionen für dieses Attribut erstellt.

Um nun die besten Splitpoints zu finden, wird die Kandidatenregel aus dem Verfeinerungsschritt temporär verfeinert. Es wird für jeden Splitpoint eine neue Regel mit der zusätzlichen

Tabelle 1: Splitpoints

Wert	1		2	2		4		5		7	7		8
Splitpoint		1,5			3		4,5		6			7,5	

Bedingung ($Attributwert > Splitpoint$) für jeden Splitpoint erzeugt. Genauso verfährt man mit der Bedingung ($Attributwert \leq Splitpoint$). Alle Regeln werden auf der Growing-Menge evaluiert und die jeweils besten $>$ und $\leq$ -Regeln werden ermittelt. Nur diese beiden Verfeinerungen werden im Verfeinerungsschritt weiter benutzt.

Das Problem dabei ist die Laufzeit. Hat man eine Growing-Menge von n Beispielen, die k numerische Attribute mit jeweils l verschiedenen Splitpoints haben, so werden für einen einzigen Verfeinerungsschritt $2 * l * k$ temporäre Regeln erzeugt und evaluiert. Für jeden weiteren Verfeinerungsschritt müssen wieder $2 * l * k$ Regeln erzeugt und evaluiert werden. In großen Datenmengen ist dabei l oftmals auch sehr groß. Dies wirkt sich sehr ungünstig auf die Laufzeit aus. Deshalb wurde hier ein etwas gröberes Verfahren zum Finden von guten Splitpoints gewählt.

Wie oben gezeigt, wird zuerst eine Liste aller möglichen Splitpoints erzeugt. Es werden eine obere und untere Schranke eingeführt. Die untere wird auf den kleinsten und die obere auf den größten Splitpoint gesetzt. Nun werden $\sqrt{m}$ Splitpoints gleichverteilt auf der Menge aller Splitpoints ausgewertet, wobei m die Anzahl aller möglichen Splitpoints innerhalb der beiden Schranken ist. Der beste Splitpoint wird gewählt, die untere Schranke wird auf den nächst kleineren soeben ausgewerteten Splitpoint gesetzt und die Obere auf den nächst Größeren. Die Anzahl der Splitpoints innerhalb beider Schranken wird neu berechnet, und es werden wiederum $\sqrt{m}$ Splitpoints ausgewählt und evaluiert. Das wiederholt sich solange, bis innerhalb der Schranken nur noch 10 mögliche Splitpoints vorhanden sind. Diese werden alle evaluiert und der beste Splitpoint wird dann zum Verfeinern genommen. Dies wird wiederum sowohl für die Relation „$>$“ als auch für „$\leq$“ gemacht.

Die Laufzeit des Algorithmus kann so merklich verringert werden. So werden für 1000 Splitpoints nur 49 Evaluationen durchgeführt. Die Gefahr bei dieser Methode ist genauso wie bei Hill-Climbing, dass das globale Optimum dabei nicht gefunden wird. Dies wirkt sich bei diesem Algorithmus allerdings nicht so stark aus. Da sich ständig die Beispielmenge verändert und die Splitpoints in jedem Verfeinerungsschritt neu berechnet werden müssen, ist die Chance doch noch das globale Maximum zu finden sehr hoch.

3.2 Die Charakteristiken von Reduce-and-Conquer

In diesem Abschnitt wird der vorgestellte Reduce-and-Conquer Algorithmus anhand der Kriterien aus Abschnitt 2.2 untersucht.

3.2.1 Die Repräsentationssprache

ReCo verwendet als Repräsentationssprache die Disjunktive Normalform. Die einzelnen Bedingungen sind entweder nominale oder numerische Tests, wie im Beispiel (2) zu sehen ist. Der Kopf der Regel ist ein numerischer Wert. Es wird eine unsortierte Regelliste zum Speichern der einzelnen Regeln verwendet. Die Repräsentationssprache wird in RefineRule und InitializeRule implementiert.

3.2.2 Die Suche im Hypothesenraum

Wie bei vielen anderen Regel-Lernern wird auch hier Hill-Climbing als Algorithmus für die Suche aufgrund des guten Trade-Offs zwischen Laufzeit und Performance verwendet. In der Funktion FindRefinementRules wird immer die beste Kandidatenregel für den nächsten Verfeinerungsschritt ausgewählt. Die Gefahr des „Hängenbleibens“ in einem lokalen Optimum besteht auch hier, wird aber durch die Reduce-Strategie abgeschwächt. Da die Beispiele nicht entfernt werden, sondern sich nur ihr Klassenattribut verringert, kann schon im nächsten Conquer-Schritt ein „verpasstes“ Optimum gefunden werden.

Es wird die Top-Down Suchstrategie verwendet, im Conquer-Schritt wird mit der generellsten Regel begonnen und die Kandidatenregeln werden durch schrittweises Verfeinern spezialisiert. Die Top-Down Suchstrategie spiegelt sich sowohl in der Initialisierung von RefinementRules als auch in RefineRules wieder.

Anders als beim Klassifizieren können hier keine der in [19] vorgestellten Heuristiken verwendet werden. Da aufgrund der Reduce-Strategie auch niemals eine einzelne Regel für sich allein evaluiert wird, muss immer die gesamte Theorie bewertet werden, also alle bis dahin gefundenen Regeln zusammen mit der aktuellen Regel. Die Heuristik soll dabei den Fehler der Theorie minimieren. Hier kommen sowohl die mittlere quadratische Abweichung (MSE) als auch die mittlere absolute Abweichung bezüglich des Median (MD) zum Einsatz. Im Abschnitt 4.3.5 werden die Ergebnisse für beide Heuristiken vorgestellt. Die Heuristik wird in EvaluateRule für die Verfeinerung verwendet. Für das Pruning ist sie in FindBestRefinementRule implementiert.

3.2.3 Vermeiden von Overfitting

Das Pruning findet hauptsächlich im Reduce-Schritt statt. Geprunt wird immer nur auf der Pruning-Menge. Oftmals wird geprunt, indem man einzelne Bedingungen sukzessive so lange löscht, bis sich der Wert der Evaluation dieser Regel verschlechtert. Dies geschieht hier nicht explizit. Die im Conquer-Schritt gefundene Verfeinerungsregelliste enthält die sortierten Regeln, wobei die erste Regel die Generellste ist und die letzte die Speziellste. Die einzelnen Regeln werden zusammen mit der bisher gefundenen Theorie evaluiert und die beste Regel wird der Theorie hinzugefügt. Die restlichen Regeln aus der Verfeinerungsregelliste werden verworfen.

Das eben beschriebene Verfahren geht von der Annahme aus, dass durch die Evaluation auf der Pruningmenge zu komplexe Regeln vermieden werden. So wird fürs Pruning nicht explizit

die Abdeckung der Beispiele benutzt. Es wird aber wie im Abschnitt 3.1.3 beschrieben eine Mindestabdeckung von mindestens 2 Beispielen verlangt.

Dies ist aber nur ein Teil der Overfitting-Strategie. Auch RuleStoppingCriterion dient der Vermeidung von Overfitting. So ist das Abbruchkriterium des Algorithmus verantwortlich für die Regelanzahl der Theorie. Ein zu restriktives Abbruchkriterium fördert eine kleine Regelanzahl in der Theorie, dafür ist die Gefahr der Ungenauigkeit groß. Ein zu weit gefasstes Abbruchkriterium führt zu einer großen Regelanzahl und die Gefahr von Overfitting steigt.

3.3 Die Konfigurationen von Reduce-and-Conquer

Im Abschnitt „Die Konfigurationen von Reduce-and-Conquer“ werden die verschiedenen Ausformungen von Reduce-and-Conquer beschrieben. Es wird aufgezeigt, welche Probleme sich dabei ergeben und wie sie gelöst werden können. Die erste Version des Algorithmus zeigte ein eher ungewöhnliches Verhalten, so dass dies genauer untersucht werden musste. Die Schritte, die zu der endgültigen Konfiguration führten werden chronologisch beschrieben.

3.3.1 Anzahl der Regeln

Die ersten Konfigurationen des Algorithmus verändern nur das RuleStoppingCriterion aus dem Reduce-Schritt. Der Algorithmus soll eine Regel zur Theorie nur dann hinzufügen, wenn der MSE auf der Pruning-Menge geringer ist als der MSE ohne diese Regel. Das Abbruchkriterium RuleStoppingCriterion lautet demnach

$$(newMSE(Theory, PruningSet) \geq oldMSE(Theory, PruningSet)) \tag{15}$$

Ist diese Bedingung erfüllt, bricht der Algorithmus ab und die Theorie wird ausgegeben. Die Bedingung darf auch nicht $(newMSE(Theory, PruningSet) > oldMSE(Theory, PruningSet))$ lauten, weil irgendwann keine Verbesserungen mehr gefunden werden. In diesem Fall bleibt der MSE immer gleich, egal wie lange der Algorithmus weiter sucht. Somit bricht der Algorithmus niemals ab.

Eine weitere Idee beruht auf der Einführung eines Grenzwertes, wenn der MSE kleiner als dieser wird, bricht der Algorithmus ab. Da aber der MSE auf verschiedenen Datenmengen sehr unterschiedlich ist, wurde diese Idee nicht weiter verfolgt. Man könnte statt des MSE auch den domänenunabhängigen RRSE verwenden, aber auch dieser fällt auf verschiedenen Datenmengen unterschiedlich aus, wie im Kapitel 4 zu sehen ist. Mit dem oben vorgestellten RuleStoppingCriterion bricht der Algorithmus im Allgemeinen sehr schnell ab, meistens findet er nur 1-4 Regeln abhängig von der Datenmenge.

Um das Ergebnis zu interpretieren wurde das RuleStoppingCriterion folgendermaßen abgeändert

$$\text{(Anzahl der Regeln} > \text{n)} \tag{16}$$

Sobald die Theorie mehr Regeln beinhaltet als der vorher festgelegte Grenzwert n bricht der Algorithmus ab. Bei der Untersuchung der gefunden Regeln tritt dabei ein unerwarteter Effekt auf.

Die Regeln, die der „Abbruchregel“ folgen, haben teilweise einen geringeren MSE und könnten die Theorie verbessern. Wenn die Beispiele durch den Reduce-Schritt nicht verändert werden würden, würde dieser Effekt nicht auftreten. Durch den verwendeten Hill-Climbing Suchalgorithmus werden immer nur die besten Verfeinerungen für den nächsten Verfeinerungsschritt ausgewählt. Verändert sich jetzt aber die Beispielmenge nicht, so werden auch immer genau die selben Verfeinerungsregeln gefunden. Dies würde in einer Endlosschleife resultieren. Da sich aber auch durch das Einfügen von schlechteren Regeln zur Theorie die Beispielmenge ändert, können im nächsten Verfeinerungsschritt wieder andere Regeln gefunden werden. Diese können wiederum einen besseren MSE auf der Pruning-Menge haben. Somit kann das Abbruchkriterium (15) nicht verwendet werden.

Das Abbruchkriterium (16) wurde mit verschieden großen n getestet. Dabei kommt ein zweiter Effekt zum Tragen. So gibt es einige kleinere Datenmengen, auf denen bei größeren n die Theorie allgemein schlechter wird. Bei großen Datenmengen ist dies meist nicht der Fall. Deshalb wurde die Idee der festen Regelanzahl auch nicht weiter verfolgt, da sie sowieso nur dazu diente, das frühe Abbrechen von RuleStoppingCriterion (15) zu verstehen. Dennoch finden sich im Abschnitt 4.3.1 die Evaluationen für verschieden große n.

Da die optimale Anzahl der Regeln für viele Datenmengen unterschiedlich ist, braucht man einen Mechanismus, der eine variable Regelanzahl für unterschiedliche Datenmengen erlaubt. Hier wird eine „Vorschaufunktion“ verwendet. Der Algorithmus bricht nicht ab, wenn eine Regel mit schlechterem MSE gefunden wird. Stattdessen läuft er weiter und beobachtet die nächsten Regeln. Dabei bestimmt ein vorher fest gewählter Grenzwert für die Vorschaufunktion, wie viele Regeln beobachtet werden sollen. Die Vorschaufunktion startet erst, wenn eine Regel mit schlechteren MSE gefunden wird. Mit jeder neuen Regel, die den MSE der Theorie nicht verbessern kann, wird die Vorschaufunktion um eins hochgezählt. Wenn die Vorschaufunktion den Grenzwert erreicht und keine bessere Regel gefunden wird, bricht der Algorithmus ab und gibt die Theorie ohne die beobachteten Regeln zurück. Wird allerdings eine Regel mit besserem MSE gefunden, obwohl die Vorschaufunktion den Grenzwert noch nicht erreicht hat, wird dieser auf Null zurückgesetzt. Diese Regel wird der Theorie hinzugefügt und der Algorithmus läuft weiter, bis die nächste Regel mit schlechteren MSE gefunden wird. Die Vorschaufunktion wird wieder gestartet usw.

Im Abschnitt 4.3.2 werden die Ergebnisse für unterschiedliche Grenzwerte der Vorschaufunktion präsentiert.

3.3.2 Die Lernmenge

Die Growing- und Pruningmenge wird beim Start des Algorithmus aufgeteilt. Danach bleibt die Verteilung der Beispiele immer gleich. In dieser Konfiguration soll untersucht werden, wie sich die Ergebnisse des Algorithmus verändern, wenn nach jeder gefundenen Regel im Reduce-Schritt die Mengen durchmischt und wieder im Verhältnis 1 zu 2 aufgeteilt werden.

Im Algorithmus müssen dafür nur nach ReduceExampleValues die beiden Mengen neu aufgeteilt werden. Da für jede gefundene Regel immer alle abgedeckten Beispiele der beiden Mengen verändert werden müssen, wird keine weitere Anpassung benötigt. Die Ergebnisse finden sich im Abschnitt 4.3.3.

3.3.3 Mindestabdeckung der Regeln

Eine weitere Idee betrifft die Mindestabdeckung der Regeln. So werden im Verfeinerungsschritt nur Verfeinerungsregeln verwendet, die eine Mindestanzahl von Beispielen abdecken. Durch eine größere Mindestabdeckung werden Regeln bevorzugt, die viele Beispiele abdecken. Bei einer kleinen Abdeckung gibt es vermehrt sehr spezielle Regeln. So ist die Veränderung der Mindestabdeckung auch eine Möglichkeit, Overfitting zu verringern. Die kleinst mögliche Mindestabdeckung ist dabei 2, da ansonsten die Berechnung der Splitpoints nicht mehr funktioniert. Auch im Reduce-Schritt wird mit Mindestabdeckung gearbeitet. So werden Regeln aus der Verfeinerungsregelliste nur dann auf der Pruning-Menge evaluiert, wenn die vorher festgelegte Mindestanzahl von Beispielen abgedeckt wird. Die Mindestabdeckung wird in beiden Schritten immer gleich gewählt.

Die Ergebnisse finden sich im Abschnitt 4.3.4.

3.3.4 Die Heuristiken

Der MSE beruht auf dem Durchschnittswert der Beispiele, also der Summe aller Beispiele geteilt durch die Anzahl der Beispiele. Da einige Datenmengen eine hohe Varianz haben, wurde der Algorithmus auch mit einer anderen Heuristik getestet.

Die mittlere absolute Abweichung bezüglich des Medians (MD) bietet sich da an. Sie beruht nicht wie MSE auf dem Durchschnitt der Beispiele, sondern auf dem Median der Beispiele. Dazu muss sowohl FindBestRefinementRule als auch EvaluateRule angepasst werden. Auch in der Berechnung der Regelvorhersage muss der Durchschnitt durch den Median ersetzt werden. Die Evaluation ist im Abschnitt 4.3.5 zu finden. Dort wird auch MD verknüpft mit einigen der anderen Konfigurationen getestet.

3.4 Ähnlichkeiten und Unterschiede zu verschiedenen Algorithmen

ReCo verfolgt wie schon beschreiben eine dem Separate-and-Conquer ähnliche Strategie. Hier wird das Aufteilen der Mengen in die Pruning- und Growingmenge wie bei RIPPER benutzt. Das in ReCo verwendete Ensemble ist durch RegENDER inspiriert. Neu ist hierbei nicht nur die Kombination dieser Techniken sondern auch die Reduce-Strategie. Diese Strategie ist nur mit Ensembles kombinierbar, da jede Regel das Klassenattribut der abgedeckten Beispiele verändert.

4 Evaluierung und Resultate

In diesem Kapitel wird sowohl der Aufbau der Tests als auch die Resultate der Experimente dargestellt. Hier werden zuerst die verschiedenen Konfigurationen von ReCo miteinander verglichen, um die Beste zu ermitteln. Anschließend wird diese Konfiguration mit anderen Regressionslernern verglichen und eingeordnet.

4.1 Beschreibung der Datenmengen

Zum Testen der verschiedenen Konfigurationen des Reduce-and-Conquer Algorithmus wurden 21 Datensätze aus dem UCI-Repositorium [1] und von Luis Torgos Webseite[1] ausgewählt. Diese Datensätze kommen beim maschinellen Lernen häufig zum Einsatz. Die ausgewählten Datensätze sollen dabei möglichst unterschiedlich sein, um verlässliche Testergebnisse zu ermöglichen. So haben sie nicht nur unterschiedlich viele Beispiele, sondern sind auch aus ganz unterschiedlichen Bereichen, wie z.B. Daten zu Brustkrebserkrankungen oder Immobilienpreisen. Die Datensätze sind größtenteils der Wirklichkeit entlehnt. Die Eigenschaften der Datensätze sind in Tabelle 2 aufgelistet.

4.2 Die Testkonfiguration

Dieser Abschnitt beschreibt die Konfiguration der Experimente sowie die Evaluationsmethoden. Die Tests der einzelnen Konfigurationen von Reduce-and-Conquer und die Vergleiche mit anderen Regressionslernern wurde auf den im Abschnitt 4.1 vorgestellten Datenmengen durchgeführt.

Zur Bewertung der verschiedenen Ergebnisse wurden verschiedene Evaluationen verwendet. Für jede Kombination aus Datenmenge und Algorithmus wurde der RRSE berechnet. Die Vorteile von RRSE gegenüber anderen Evaluationsmethoden liegen in der Unabhängigkeit von den Datenmengen. RRSE bewertet immer die gesamte Theorie. Im weiteren Kapitel wird der RRSE immer prozentual angegeben. Werte über 100 bedeuten, das die Theorie schlechter ist als die Vorhersage des Durchschnitts.

Im Abschnitt 4.3 werden zunächst die Evaluationen der verschiedenen Konfigurationen von ReCo erläutert. Hierbei wird versucht durch bessere Konfigurationen den RRSE zu minimieren. Trotz der Unabhängigkeit von RRSE kann man immer nur auf der gleichen Datenmenge vergleichen. So ist der Fehler auf Datenmengen, die nur sehr wenig Regelmäßigkeiten aufweisen, deutlich höher als auf Datenmengen, die sehr gut durch Regeln beschrieben werden können. Da dies aber bei allen Konfigurationen passiert gleicht sich dieser Effekt wieder aus.

Es wird als Anhaltspunkt auch der Durchschnitt der Evaluation aller Datenmengen angegeben. Dieser Wert ist allerdings nur bedingt aussagekräftig. Wenn ein Algorithmus auf den meisten Datenmengen sehr gut abschneidet und auf einer sehr schlecht, wird das Ergebnis durch

[1] Die Datensätze können heruntergeladen werden unter
http://www.liaad.up.pt/~ltorgo/Regression/DataSets.html

Tabelle 2: Die 21 Datensätze

Name	Beispiele	Attribute	numerische Attribute	nominale Attribute
auto93	93	23	17	6
auto-horse	205	26	18	8
auto-mpg	398	8	5	3
auto-price	159	16	15	1
cloud	108	7	5	2
compressive	1030	9	9	0
concrete-slump	103	10	10	0
cpu	209	8	7	1
delta-elevators	9517	7	7	0
diabetes	43	3	3	0
echo-month	130	10	7	3
housing	506	14	13	1
machine	209	7	7	0
meta	528	22	20	2
pyrim	74	28	28	0
r-wpbc	194	33	33	0
stock	950	10	10	0
strike	625	7	6	1
triazines	186	61	61	0
veteran	137	8	4	4
winequality-red	1599	12	12	0

die Verwendung des Durchschnittes verzerrt. Deshalb wird auch ein anderes Verfahren zum Vergleichen der Algorithmen verwendet. Für jede Datenmenge wird der Rang berechnet, den der jeweilige Algorithmus erreicht. So bekommt der Algorithmus mit dem geringsten RRSE den 1. Rang, der Algorithmus mit dem schlechtesten Ergebnis landet auf dem letzten Rang. Haben mehrere Algorithmen das gleiche Ergebnis, so wird jedem der Mittelwert der Ränge zugewiesen. Wenn z.B. 3 Algorithmen mit dem selben Ergebnis die Ränge 3, 4 und 5 belegen, wird allen Dreien der Rang 4 zugewiesen.

Um die Ergebnisse zu vergleichen wird der Friedman-Test verwendet, wie er in [11] beschrieben wird. Es wird der Durchschnitt der Ränge auf allen Datenmengen berechnet. Für diesen Abschnitt werden folgende Bezeichner verwendet:

- n: Anzahl der Datenmengen
- k: Anzahl der Algorithmen
- i: Datenmenge i
- j: Algorithmus j
- r_i^j: Rang von Algorithmus i auf Datenmenge j
- $R_j = \frac{1}{n} \sum_{i=1}^{n} r_{ij}$: Durchschnitt der Ränge des Algorithmus j

Die Friedman Statistik wird berechnet durch

$$\chi_F^2 = \frac{12n}{k(k+1)} \left[\sum_{j=1}^{n} R_j^2 - \frac{k(k+1)^2}{4} \right] \tag{17}$$

Dabei sollten n und k groß genug sein, als Daumenregel gelten hier $n > 10$ und $k > 5$, da ansonsten die Aussagekraft des Friedman-Tests zu gering ist. Iman und Davenport haben in [21] eine bessere Statistik vorgeschlagen, da die Friedman Statistik zu zurückhaltend ist.

$$F_F = \frac{(n-1)\chi_F^2}{n(k-1) - \chi_F^2} \tag{18}$$

Dieser Wert ist ein Maß für die Signifikanz der Tests. Wenn der Wert über dem der F-Verteilung für $(k-1)$ mit $(k-1)(n-1)$ Freiheitsgraden für ein vorgegebenes Signifikanzniveau liegt, unterscheiden sich die Algorithmen signifikant. Allerdings gibt F_F keine Auskunft darüber, wie viele und welche Algorithmen signifikant sind.

Deshalb wird bei festgestellter Signifikanz im Anschluss der Nemenyi-Test durchgeführt.

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6n}} \tag{19}$$

q_α sind dabei die kritischen Werte der studentisierten Spannweitenverteilung geteilt durch $\sqrt{2}$. *CD* gibt die kritische Distanz an. Algorithmen, deren Ränge innerhalb dieser Distanz liegen können zu einer Gruppe zusammengefasst werden, die keine signifikanten Unterschiede haben. Ist der Abstand zwischen 2 Algorithmen größer als die kritische Distanz, so unterscheiden sie sich signifikant.

Jeder Test wurde mit 10-facher Cross-Validation durchgeführt. Hierfür wird jede Datenmenge in 10 Partitionen aufgeteilt. Auf den letzten 9 wird gelernt, die Erste dient zum Testen der gefundenen Theorie. Danach wird auf allen außer der 2. Partition gelernt und auf dieser getestet usw. Es wird der Durchschnitt aller zehn Durchgänge berechnet. Crossvalidation wird verwendet, da auf den zu lernenden Daten keine Evaluation gemacht werden kann, weil sich ansonsten die Algorithmen zu sehr an die Lernmenge anpassen.

Ziel der Arbeit ist aber der Vergleich von Reduce-and-Conquer mit anderen Regressionslernern. Dabei wurden die in *weka* [38] vorhandenen Regressionsregellerner *ConjunctiveRule, DecisionTable, M5Rules* und RegENDER [10] sowie *Lineare Regression, MPL und SVM* verwendet.

ConjunctiveRule lernt nur eine einzige Regel, ist also ein sehr schwacher Regellerner. M5Rules-R ist eine Konfiguration von M5Rules die einen anderen Baum zum Regelerstellen benutzt. In DecisionTable wird eine Entscheidungsliste gelernt. Die Ergebnisse finden sich im Abschnitt 4.5.

4.3 Evaluation der verschiedenen Konfigurationen

In diesem Abschnitt werden die einzelnen im Abschnitt 3.3 beschriebenen Konfigurationen getestet. Es werden folgende Abkürzungen verwendet:

- RL: Konfiguration mit fester Regelanzahl
- Pv: Konfiguration mit Vorschaufunktion
- RD: Konfiguration mit Durchmischen der Beispiele
- Cv: Konfiguration mit Mindestabdeckung
- MSE: Konfiguration mit MSE
- MD: Konfiguration mit MD
- RA: Regelanzahl
- KF: Konfiguration

4.3.1 Feste Regelanzahl

Tabelle 3: Ergebnisse für vorgegebene Regelanzahl

RL =	5	10	15	20	25	30	35	40	beste KF
auto93	99	100,3	99,5	99,8	99,8	99,8	99,9	100,4	5
auto-horse	60,3	58,1	56	55,4	55,2	54,7	54,5	54,1	40
auto-mpg	60,2	50,8	48,1	48	48	46,8	46,8	46,3	40
auto-price	51,9	47,4	45,6	47	46,6	46,6	46,7	47,1	15
cloud	71,3	77,9	77,4	77,7	77,5	76,9	76,4	76,3	5
compressive	66,3	54,9	52,4	49	47,3	46,2	45,2	44,3	40
concrete-slump	84,9	65,8	67,9	66,9	68,1	68,8	68,9	69,1	10
cpu	53	50,8	52	51,6	50,6	50,4	50,7	50,7	30
delta-elevators	70,2	66,2	64,6	63,8	63,5	63,3	63,2	63	40
diabetes	103,3	96	101	103,6	105,6	106,7	106,3	106,5	10
echo-month	86	98,5	105,6	110	107,9	108,6	108,9	110,1	5
housing	61,3	55,2	53,6	54,5	54,1	53,9	52,7	53,5	35
machine	42,4	38,5	39,8	41,6	41,1	40,4	41,4	42,1	10
meta	117,4	125,5	133,7	136,4	141,3	141,6	142,1	142,7	5
pyrim	91,9	81,9	81,8	80,8	84	82,3	82,3	82,1	20
r-wpbc	106,8	117,5	123,7	125,8	128,4	129,9	130,3	131,1	5
stock	37,7	29,8	27	26	25,9	25,9	25,5	25,1	40
strike	95,3	96,4	99,7	100,4	103,3	104,1	105,7	107	5
triazines	98,9	93,8	91,6	92,3	92,3	93,9	94,5	94	15
veteran	115,7	122,2	121,4	120,6	121,2	121	122,5	122,6	5
winequality-red	86,7	85,2	85,6	85,3	85,4	85,7	85,9	86,3	10
Durchschnitt	79,1	76,8	77,5	77,9	78,4	78,5	78,6	78,8	10

Wie in Abschnitt 3.3.1 erläutert wird die Anzahl der Regeln vorgegeben. Dabei soll festgestellt werden, welche Regelanzahl auf den einzelnen Datenmengen optimal ist. In Tabelle 3 wird der

RRSE für alle Datenmengen der Anzahl der Regeln gegenübergestellt. Es wurde mit 5 Regeln begonnen und dann in Fünferschritten bis zu einer Anzahl von vierzig Regeln evaluiert. In der letzten Spalte wird jeweils die mit dieser Methode gefundene beste Regelanzahl für die jeweilige Datenmenge angegeben.

Auffällig ist, dass mehr Regeln nicht gleichzeitig eine höhere Genauigkeit bedeuten. So ist in den Mengen *auto93, cloud, echo-month, meta, r-wbpc, strike, veteran* schon mit 5 Regeln das Optimum erreicht. Andere Mengen hingegen wie *auto-horse, auto-mpg, compressive, delta-elevators, stock* können mit 40 Regeln besser beschrieben werden. Auf diesen Mengen nimmt auch der RRSE stetig ab, auf den anderen Mengen ist dies selbst bis zur optimalen Regelanzahl meistens nicht so.

Die beste Konfiguration auf allen Datenmengen ist die Konfiguration mit 10 Regeln. Es ist leicht zu erkennen, dass die Konfigurationen mit einer vorgegebenen Regelanzahl viel zu statisch ist und zu sehr von der Datenmenge abhängt.

Aus der Tabelle 2 der Datensätze lässt sich keine Regelmäßigkeit zwischen Anzahl der Regeln und Größe der Datenmengen oder der Anzahl der Attribute herleiten. Auch ist auffällig, dass der RRSE auf den verschiedenen Datenmengen sehr unterschiedlich ist, im Durchschnitt liegt er zwischen 76 und 79. Es gibt aber einige Datenmengen mit Werten über 100 und unter 50. Das spricht auch dafür, dass die Datenmengen unterschiedlich sind.

4.3.2 Abbruchkriterium mit Vorschau

In dieser Konfiguration wird das im Abschnitt 3.3.1 beschriebene Verfahren zum Anpassen der Regelanzahl auf den Datenmengen getestet. Es wurde mit einer Vorschau von 5 bis 20 Regeln gearbeitet. Tabelle 4 zeigt sowohl den RRSE als auch die Größe der Regelmenge. In der letzten Spalte ist wiederum das beste Ergebnis notiert. Die unterste Zeile gibt den Durchschnitt über alle Datenmengen an.

Schon auf der Datenmenge *auto-horse* fällt auf, dass der beste RRSE mit 10 Vorschauregeln erreicht wird. Die Regelanzahl hat sich aber zum Test vorher nicht verändert, d.h. die optimale Regelmenge wurde schon mit der ersten Konfiguration gefunden, obwohl der beste RRSE mit der 2. Konfiguration erreicht wird. Dabei werden auch genau die selben Regeln gefunden. Dieser Effekt kommt durch die Crossvalidation zustande. In der Crossvalidation werden nicht alle Beispiele zum Lernen verwendet. Dadurch können andere Theorien gefunden werden als bei der Verwendung aller Beispiele. So kann durchaus eine Theorie gefunden werden, die durch die Partitionierung der Beispiele besser ist. Dadurch sinkt auch der Wert des RRSE. Die optimale Regelmenge wird auf allen Beispielen erzeugt. Deshalb ist als bestes Ergebnis immer das erste Auftreten der optimalen Regelmenge angegeben.

Die meisten Datenmengen, die im vorherigen Abschnitt den besten Wert mit 40 Regeln haben, profitieren auch von einer größeren Vorschaufunktion. Die Datemengen, die im vorherigen Abschnitt eher zu einer kleinen Regelanzahl tendieren, haben auch mit dieser Konfiguration eher wenig Regeln.

Auf 16 von 21 Datenmengen wird die beste Regelmenge schon mit 5 Vorschauregeln gefunden. Der Gewinn durch ein größeres Vorschaufenster wiegt dabei nicht die stark ansteigende Laufzeit auf. So wird für die Berechnung auf allen Datenmengen für *(Vorschau = 5 Regeln)* 3:31 Minuten benötigt, für *(Vorschau = 20 Regeln)* hingegen schon 8:07 Minuten. Deshalb werden

Tabelle 4: Ergebnisse mit verschiedenen Vorschaugrößen

PV =	5	RA	10	RA	15	RA	20	RA	beste KF
auto93	95,3	3	95,5	3	95,5	3	97,5	3	5
auto-horse	52,8	5	52,5	5	52,5	5	52,5	5	5
auto-mpg	47,9	12	47,6	51	47	69	46,8	69	20
auto-price	47,6	21	45,8	25	46,1	25	45,5	25	20
cloud	74,7	3	74,7	3	75,1	3	73,2	3	5
compressive	43,6	27	41,4	27	39,3	122	38,3	122	20
concrete-slump	70,3	26	71,7	26	71,7	26	71,7	26	5
cpu	48,9	19	48,2	19	48,1	19	46,7	19	5
delta-elevators	62,9	48	63	48	63,2	48	63,3	48	5
diabetes	98,9	3	100,5	3	100,5	3	100,5	3	5
echo-month	94,2	3	94,2	3	94,2	3	95,3	3	5
housing	52,2	23	52,7	23	53,3	23	53,6	23	5
machine	40,9	12	40	12	40	12	40	12	5
meta	110,2	2	110,2	2	110,2	2	110,2	2	5
pyrim	85,9	12	84,2	15	84,2	15	84,2	15	10
r-wpbc	106,1	3	106,1	3	106,1	3	106,1	3	5
stock	25,2	33	24,7	72	24,6	72	24,6	116	20
strike	96,4	2	95,9	2	97,7	2	97,8	2	5
triazines	98,8	6	97,5	6	91,8	6	92,4	6	5
veteran	111,2	4	111,2	4	111,2	4	111,2	4	5
winequality-red	84,8	13	85,4	84	86	100	86,1	164	5
Durchschnitt	73,7	13,3	73,5	20,8	73,2	26,9	73,2	32	15

die weiteren Test mit *(Vorschau = 5 Regeln)* durchgeführt. Dies hat auch den Effekt, dass die Theorie eher zu kleineren Regelmengen tendiert, was ja nicht schlecht sein muss, da eine zu große Anpassung an die Datenmengen so weniger unterstützt wird.

4.3.3 Aufteilung der Growing- und Pruningmenge

RIPPER teilt die Growing- und Pruningmenge nach jedem Schritt erneut auf. Es soll dadurch einer zu großen Anpassung an die Growingmenge entgegengewirkt werden. Da auch RECO die Aufteilung in eine Lern- und eine Pruningmenge benutzt, lag es nahe dies auch zu testen. Es wurde die Konfiguration aus dem vorherigen Abschnitt mit einer Vorschaugröße von 5 Regeln gewählt, einmal mit Mischen der Mengen und einmal ohne. Die Ergebnisse finden sich in Tabelle 5.

Es wird für jede Datenmenge sowohl der RRSE als auch die Anzahl der gefundenen Regeln angegeben. In der letzten Zeile findet sich wieder der Durchschnitt aller Datenmengen.

Durch das Mischen werden durchschnittlich 4 Regeln mehr gefunden. Insgesamt schneiden 14 Datenmengen schlechter ab, nur 7 profitieren vom Durchmischen der Beispiele. Mit dieser Konfiguration wurden zumindest ähnlich gute Ergebnisse wie mit die Variante ohne neues Aufteilen

Tabelle 5: Ergebnisse ohne und mit Mischen der Beispiele

RD:	ohne	RA	mit	RA
auto93	95,3	3	91,3	3
auto-horse	52,8	5	54,8	23
auto-mpg	47,9	12	47,5	12
auto-price	47,6	21	48,8	32
cloud	74,7	3	66,7	7
compressive	43,6	27	46,3	52
concrete-slump	70,3	26	79,7	38
cpu	48,9	19	41,9	8
delta-elevators	62,9	48	63,9	11
diabetes	98,9	3	113	9
echo-month	94,2	3	99,2	14
housing	52,2	23	53,7	24
machine	40,9	12	48,2	10
meta	110,2	2	133,7	2
pyrim	85,9	12	90,1	9
r-wpbc	106,1	3	142,8	23
stock	25,2	33	24,2	30
strike	96,4	2	102,1	10
triazines	98,8	6	92,5	28
veteran	111,2	4	108,3	6
winequality-red	84,8	13	86,6	11
Durchschnitt	73,7	13,3	77,9	17,24

der Mengen erwartet. Allein aus den Daten lässt sich dieser Effekt schwer deuten. Er könnte mit der Mindestabdeckung zusammenhängen. Die Mindestabdeckung beträgt aufgrund der Berechnung der Splitpoints mindestens 2 Beispiele. So werden Regeln im Verfeinerungsschritt sofort verworfen, wenn nur ein Beispiel abgedeckt wird. Genauso sieht es im Reduce-Schritt aus. Daraus folgt, das mindestens auf der Growing-Menge 2 Beispiele und auf der Pruning-Menge auch 2 Beispiele abgedeckt werden müssen, ansonsten hat diese Regel keine Chance zur Theorie hinzugefügt zu werden. Durch das Mischen werden evt. ungünstige Beispielverteilungen erzeugt, so dass auf einer der Mengen durch eine Regel nur ein Beispiel abgedeckt wird, auf der anderen Menge aber beliebig viele abgedeckte Beispiele sein können. Diese Regel wird immer verworfen. Je öfter gemischt wird, desto öfter kann dieser Effekt auftreten. Dies ist aber nur eine Vermutung, die Tests dazu würden den Rahmen dieser Arbeit sprengen.

Aufgrund der schlechten Ergebnisse wurde diese Konfiguration wieder verworfen.

4.3.4 Mindestabdeckung

In dieser Testreihe wird als Basis die Konfiguration aus Abschnitt 4.3.2 mit einer Vorschauregelanzahl von 5 gewählt. Wie schon erwähnt müssen in der Grundkonfiguration immer mindestens

2 Beispiele von einer Regel abgedeckt werden, und zwar sowohl auf der Growing- als auch auf der Pruningmenge. Die Mindestabdeckung wird für beide Mengen schrittweise auf 4 erhöht.

In Tabelle 6 sind für alle Datenmengen sowohl der RRSE als auch die Anzahl der gefundenen Regeln angegeben. Die untere Zeile enthält den Durchschnitt aller Datenmengen.

Der Durchschnitt für *Abdeckung = 3* ist etwas besser als in den beiden anderen Tests. Vergleicht man aber, mit welcher Konfiguration die einzelnen Datenmengen besser werden, sieht das Ergebnis nicht mehr so eindeutig aus. So sind 5 Datenmengen mit *Abdeckung = 2*, 8 mit *Abdeckung = 3* und 8 mit *Abdeckung = 4* besser. Eine leichte Verbesserung durch eine größere Mindestabdeckung war erwartet worden, da oftmals dadurch generellere Regeln entstehen. So müssen mit einer Mindestabdeckung von 4 mindestens 8 Beispiele vorhanden sein, die von dieser Regel abgedeckt werden können. Einzelne Ausreißer in der Datenmenge werden somit ignoriert.

Tabelle 6: Ergebnisse für verschiedene Mindestabdeckung

Cv =	2	RA	3	RA	4	RA
auto93	95,3	3	92,4	3	87,6	3
auto-horse	52,8	5	51,1	31	53,6	24
auto-mpg	47,9	12	47,3	12	46,7	41
auto-price	47,6	21	47,2	15	54,2	12
cloud	74,7	3	75	3	75,8	3
compressive	43,6	27	45	27	43,1	52
concrete-slump	70,3	26	69,5	7	67,5	25
cpu	48,9	19	62,5	16	56,2	8
delta-elevators	62,9	48	62,9	51	62,8	46
diabetes	98,9	3	93,9	3	87,7	3
echo-month	94,2	3	90,2	3	88,5	3
housing	52,2	23	52,4	16	54,8	18
machine	40,9	12	54	25	59,6	11
meta	110,2	2	98,7	2	99,8	2
pyrim	85,9	12	82,1	8	90,7	16
r-wpbc	106,1	3	104,8	3	104,8	3
stock	25,2	33	25,2	31	25,4	59
strike	96,4	2	94,7	4	95,8	4
triazines	98,8	6	93,3	2	91,2	4
veteran	111,2	4	102,2	4	107	4
winequality-red	84,8	13	84,7	16	84,8	14
Durchschnitt	73,7	13,3	72,8	13,43	73,2	16,9

4.3.5 Die Heuristik MD

In den vorherigen Konfigurationen wurde das Abbruchkriterium, die Beispielmenge und die Mindestabdeckung verändert und getestet. Durch alle Varianten wird versucht Overfitting zu

verringern. In diesem Abschnitt wird die Konfiguration aus dem vorherigen Abschnitt mit einer Regelvorschau von 5, kein Durchmischen der Beispielmengen und mit einer Mindestabdeckung von 3 auf beiden Beispielmengen verwendet. In dieser Konfiguration wird noch eine andere Heuristik getestet. Statt des MSE wird MD basierend auf dem Median verwendet. Die Vermutung ist, dass durch die Verwendung des Median für die Regelvorhersage gerade Testmengen mit vielen Ausreißern in der Datenmenge besser abschneiden.

Die Tabelle 7 enthält aufgrund besserer Übersichtlichkeit nur den RRSE der verschiedenen Konfigurationen.

Der Vergleich der beiden Konfigurationen ist in den ersten beiden Spalten zu finden. Auf 8 Datenmengen ist die Konfiguration mit MD besser. Dabei ist der Durchschnitt aller Datenmengen bei beiden Konfigurationen gleich. MSE scheint leicht im Vorteil zu sein, dennoch wurden weitere Test mit MD und anderen Konfigurationen durchgeführt.

Die weiteren Spalten aus Tabelle 7 enthalten jeweils die verschiedenen Konfigurationen mit MSE und MD. In den ersten vier Spalten wurde die Konfiguration mit der Vorschaufunktion aus Abschnitt 4.3.2 mit einer Vorschau von *(Pv = 5)* und *(Pv = 10)* verwendet, in den folgenden 2 Spalten stehen die Ergebnisse mit einer Vorschaufunktion von *(Pv = 5)* und einer Mindestabdeckung von *(Cv = 3)* (Abschnitt 4.3.4) und die letzten beiden Spalten enthalten die Konfiguration aus Abschnitt 4.3.3 mit der Durchmischung der Growing- und Pruningmenge.

Tabelle 7: Ergebnisse für MSE-MD mit anderen Konfigurationen

	MSE Pv = 5	MD Pv = 5	MSE Pv = 10	MD Pv = 10	MSE Cv = 3	MD Cv = 3	MSE RD	MD RD
auto93	95,3	73,8	95,5	74,5	92,4	71,2	91,3	79,2
auto-horse	52,8	55,5	52,5	54	51,1	51,1	54,8	52,3
auto-mpg	47,9	51	47,6	51,2	47,3	50,8	47,5	49,2
auto-price	47,6	48,1	45,8	48,7	47,2	52,9	48,8	53,5
cloud	74,7	56,3	74,7	56,1	75	55,3	66,7	56,5
compressive	43,6	55,7	41,4	53,8	45	55,1	46,3	55,4
concrete-slump	70,3	77,2	71,7	75,8	69,5	66,9	79,7	79,6
cpu	48,9	60,5	48,2	60,5	62,5	70,1	41,9	33,1
delta-elevators	62,9	70,8	63	70,8	62,9	70,8	63,9	70,3
diabetes	98,9	94,8	100,5	94,8	93,9	96,4	113	95,2
echo-month	94,2	77,9	94,2	78,4	90,2	78	99,2	85,1
housing	52,2	55,9	52,7	56	52,4	55,7	53,7	56,5
machine	40,9	59	40	55,6	54	51,3	48,2	61,9
meta	110,2	100,3	110,2	99,9	98,7	100,2	133,7	100,3
pyrim	85,9	89,5	84,2	80,7	82,1	83,5	90,1	99,5
r-wpbc	106,1	109,5	106,1	110,6	104,8	108,7	142,8	142,6
stock	25,2	31,2	24,7	30,6	25,2	29,7	24,2	27,9
strike	96,4	91,6	95,9	91,4	94,7	91,5	102,1	93,1
triazines	98,8	94,3	97,5	93,5	93,3	97	92,5	102,5
veteran	111,2	99,2	111,2	98,8	102,2	100,7	108,3	100,3
winequality-red	84,8	95,1	85,4	95,1	84,7	95,3	86,6	94,9
Durchschnitt	73,7	73,7	73,5	72,9	72,8	73	77,9	75,7

Auf 8 Datenmengen ist die Evaluation mit MSE besser, bei 9 Datenmengen ist es unterschiedlich und auf 4 Datenmengen ist MD besser. Schaut man sich die 9 Datenmengen an, die unterschiedliche Ergebnisse haben so sind 5 in drei Evaluationen mit MSE besser, mit MD sind es nur 2. Die beiden restlichen Datenmengen schneiden mit MSE und MD ähnlich ab. Insgesamt verstärkt sich der Eindruck, dass MSE in ReCo eine bessere Heuristik als MD ist.

4.4 Vergleich der verschiedenen Konfigurationen

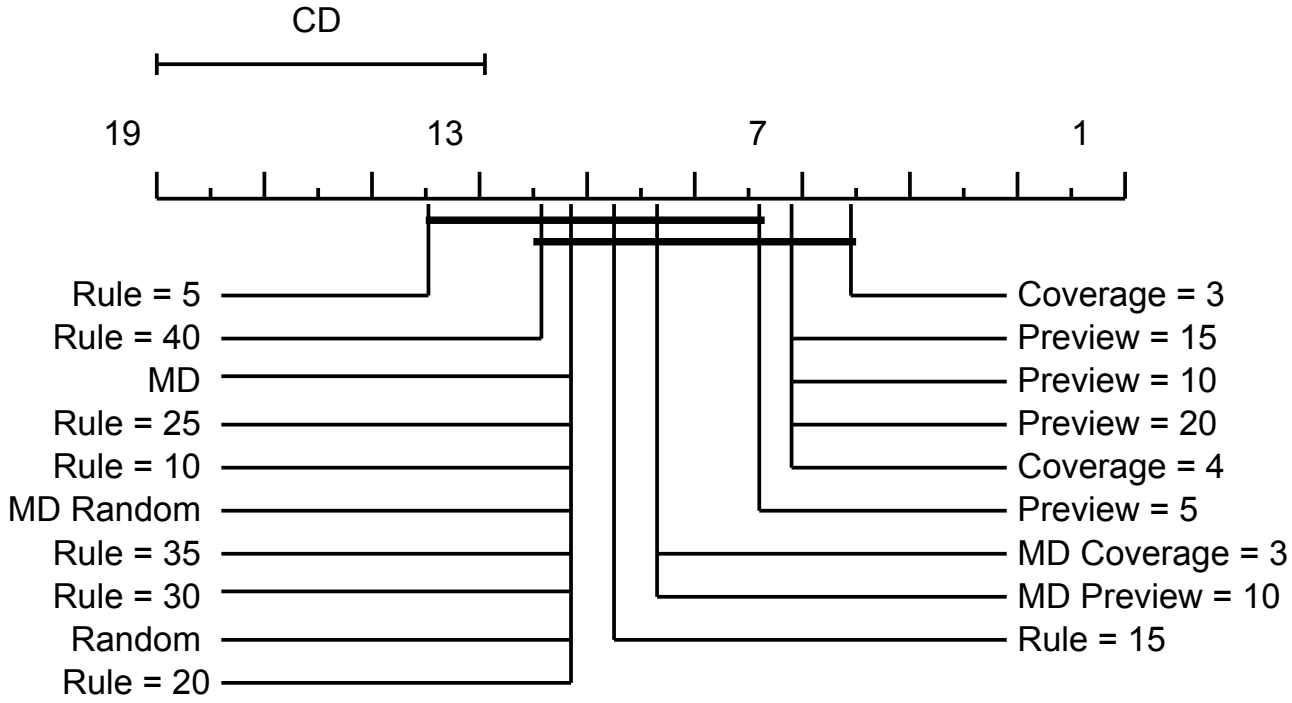
Es wurden alle beschriebenen Konfigurationen getestet, doch stellt sich die Frage welche am Besten ist. Schaut man sich den Durchschnitt der getesteten Konfigurationen auf allen Datenmengen an, so ist mit 72,8 die Konfiguration mit der Mindestabdeckung von 3, keine Durchmischung, MSE und mit einer Vorschaufunktion von 5 am besten. Wie aber schon am Anfang des Kapitels erwähnt, ist der Durchschnitt aller Datenmengen ein nicht so zuverlässiges Kriterium. Deshalb werden in diesem Abschnitt alle bisherigen Konfigurationen mit einem Rankingverfahren verglichen, wobei der durchschnittliche Rang für alle Datenmengen berechnet wird.

Tabelle 8: Ranking der Konfigurationen

Datenmenge	RRSE	Rang
CV = 3	72,81	6,26
PV = 15	73,24	7,21
PV = 10	73,46	7,26
PV = 20	73,21	7,29
CV = 4	73,20	7,31
PV = 5	73,74	7,86
MD & CV=3	72,97	9,76
MD & PV=10	72,90	9,86
RL = 15	77,53	10,52
RD	77,88	11,24
RL = 20	77,91	11,24
RL = 30	78,46	11,29
MD & RD	75,67	11,38
RL = 35	78,59	11,38
MD	73,67	11,43
RL = 10	76,79	11,43
RL = 25	78,43	11,43
RL = 40	78,78	11,86
RL = 5	79,06	13,95

Aus Tabelle 8 ergibt sich, dass die Konfiguration *(CV = 3)* die Beste ist. Konfigurationen mit dem gleichen durchschnittlichen Rang wurden anschließend nach dem RRSE sortiert. Auch die Verteilung der anderen Konfigurationen hat sich schon während der Evaluation so abgezeichnet. So sind die Varianten mit statischer Regelanzahl eher am Ende der Tabelle zu finden. Erstaunlich ist das Abschneiden von MD. Nach dem Durchschnittswert zu urteilen, müsste diese Konfigura-

Abbildung 1: Nemenyi-Test mit den verschiedenen Konfigurationen von RECO für p = 0,05



tion deutlich weiter vorne liegen. Auch dies gibt einen Hinweis darauf, das die Wahl von MD nicht zu einem besseren Algorithmus führt. Auch die anderen Konfigurationen mit MD liegen zum Teil deutlich unter den selben Konfigurationen mit MSE.

Da aber noch nicht bekannt ist, ob sich die Konfigurationen signifikant voneinander unterscheiden wird der Friedman-Test durchgeführt. Mit den aufgelisteten Rängen, den 19 verschiedenen Konfigurationen und den 21 Datenmengen ergibt sich der Wert $F_F = 3{,}3393$. Es muss nun überprüft werden, ob es überhaupt eine Signifikanz gibt. Für einen p-Wert von 0,05 ergibt sich eine Signifikanzgrenze von 1.6326. Da der F_F-Wert darüber liegt gibt es signifikante Konfigurationen. Nun wird mit dem Nemenyi-Test untersucht welche Konfigurationen signifikant unterschiedlich sind und welche sich nicht signifikant voneinander unterscheiden. Der Test ergibt eine Kritische Distanz von 6,1079. Alle Konfigurationen, die sich im Ranking innerhalb dieser Distanz befinden sind nicht signifikant unterschiedlich.

Abbildung 1 stellt den Nemenyi-Test grafisch dar. Der Zahlenstrahl stellt die 19 möglichen Ränge dar. Auf diesem sind die einzelnen durchschnittlichen Ränge der verschiedenen Konfigurationen aufgetragen. Ränge, die zu dicht beieinander lagen wurden zusammengefasst. Mit der Strecke CD ist die Kritische Distanz dargestellt. Verschiebt man nun diese entlang des Zahlenstrahls kann man die Konfigurationen zu verschiedenen Gruppen zusammenfassen. In diesem Beispiel gibt es genau zwei Gruppen. Die erste Gruppe beinhaltet die 14 schlechtesten Ränge. Alle diese Konfigurationen unterscheiden sich nicht signifikant voneinander. Daraus kann man ableiten, dass die Konfigurationen der ersten fünf Ränge sich signifikant von der schlechtesten Konfiguration unterschieden.

Die zweite Gruppe enthält die besten 18 Konfigurationen. Diese unterschieden sich nicht signifikant voneinander. Auch hier kann man daraus ableiten, dass die schlechteste Konfiguration sich signifikant von der Besten unterscheidet.

Im folgenden Abschnitt wird die beste Konfiguration mit anderen Regellernern verglichen.

Tabelle 9: Evaluation der verschiedenen Regressionslerner

	ReCo	LR	MPL	CR	DT	M5R	M5-R	SVM	RegE
auto93	92,4	59,5	55,8	89,3	77,6	54,1	81,1	62,5	80,4
auto-horse	51,1	31,1	33,3	71,3	48,9	51,1	54,8	31,5	43,7
auto-mpg	47,3	37,9	48,2	67,3	44,4	50,8	47,5	38,9	49,1
auto-price	47,2	48,6	45,4	62,8	54,7	52,9	48,8	46,6	48,3
cloud	75	38,2	63,7	88,3	61,2	55,3	66,7	37,2	62,7
compressive	45	62,9	47,5	84,1	67,6	55,1	46,3	65,5	40,9
concrete-slump	69,5	36,2	8,7	93,2	69,2	66,9	79,7	39	56
cpu	62,5	42,5	8	80,2	37,4	70,1	41,9	29,3	46,1
delta-elevators	62,9	61	67,4	75,1	63,4	70,8	63,9	61,1	63,6
diabetes	93,9	91,8	102,2	95,4	114,1	96,4	113	92,4	104,7
echo-month	90,2	68,8	122,6	71,7	67,9	78	99,2	74,1	98,2
housing	52,4	53,1	48,3	77,3	59,7	55,7	53,7	54,5	53,9
machine	54	46,2	41,9	85	60,4	51,3	48,2	41,9	47,5
meta	98,7	203,6	122	116,4	174,8	100,2	133,7	81,6	261,9
pyrim	82,1	96	96,9	85,4	79,4	83,5	90,1	91,6	71,1
r-wpbc	104,8	97,2	184,8	102	107,7	108,7	142,8	97,1	126,8
stock	25,2	35,9	18,5	62,1	22,2	29,7	24,2	36,8	24,6
strike	94,7	85,6	110,3	96,8	84,8	91,5	102,1	82,3	106,2
triazines	93,3	106,1	122,9	99,5	93,3	97	92,5	95,9	84,9
veteran	102,2	94,7	218,8	101	101,8	100,7	108,3	89,1	124,8
winequality-red	84,7	80,8	90,8	91,2	85,3	95,3	86,6	81,5	89,3
Durchschnitt	72,8	70,4	79	85,5	75	66,7	76,6	63,3	80,2

4.5 Vergleich mit anderen Regressionslernern

Die im Abschnitt 4.4 gefundene beste Konfiguration von ReCo wird in diesem Abschnitt mit anderen Regressionslernern verglichen. ReCo wird mit der besten Konfiguration aus Abschnitt 4.4 verwendet. Der Algorithmus wird mit den in *weka* implementierten Regellernern *ConjunctiveRule, DecisionTable, M5Rules, M5Rules-R* sowie mit *RegENDER* verglichen. Zusätzlich wird ReCo auch noch mit anderen Regressionsverfahren wie *LinearRegression, MultilayerPerceptron, SVMreg* verglichen. Alle Algorithmen werden mit den Standardkonfigurationen und einer 10-fachen Crossvalidation evaluiert. Wie im vorherigen Abschnitt wird zur Evaluation der RRSE verwendet. Tabelle 9 zeigt die Ergebnisse der Regressionslerner auf den 21 Datenmengen. Dabei werden für die Lerner folgende Abkürzungen verwendet:

- ReCo: Regression-and-Conquer
- LR: LinearRegression
- MPL: MultilayerPerceptron
- CR: ConjunctiveRule
- DT: DecisionTable
- M5R: M5Rules
- M5R-R: M5Rules-R
- SVM: SVMreg
- RegE: RegENDER

Tabelle 10: Ranking der Regressionslerner

M5R	66,66	2,90
SVM	63,35	3,48
LR	70,36	3,62
ReCo	72,81	5,24
MPL	78,96	5,24
DT	75,03	5,38
RegE	80,22	5,67
M5R-R	76,63	6,05
CR	85,50	7,43

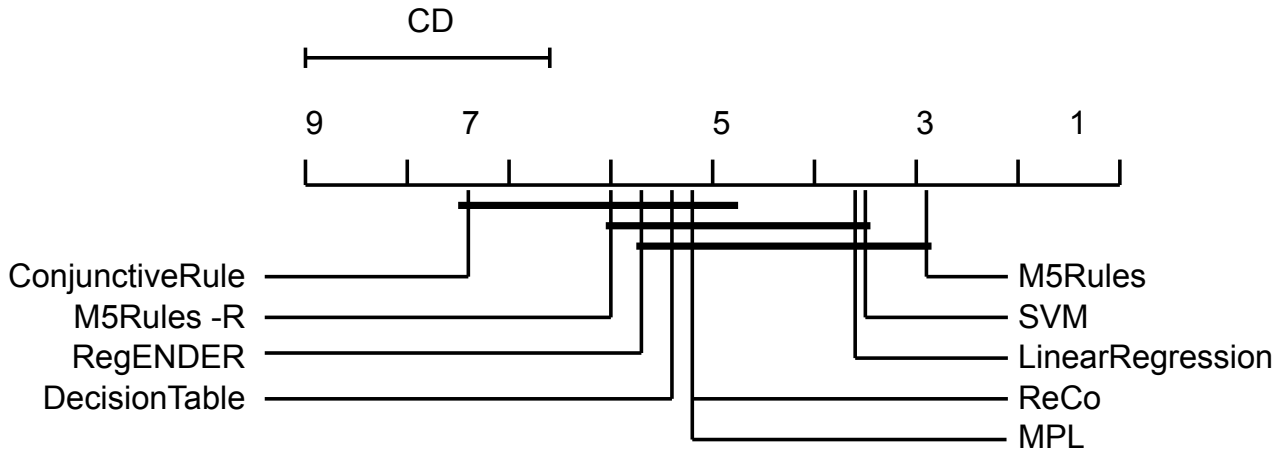
ReCo ist auf 8 Datenmengen besser als der Durchschnitt, auf 4 Datenmengen liegt ReCo im Durchschnitt und auf 9 Datenmengen ist er schlechter. Somit liegt ReCo im Mittelfeld der hier getesteten Algorithmen. Auf den Datenmengen *auto93, auto-horse, cloud, concrete-slump, cpu, echo-month, veteran* fällt das Ergebnis schlecht aus, dafür schneidet ReCo auf *compressive, diabetes, housing, meta, triazines* gut ab.

ConjunctiveRule ist der schlechteste Algorithmus, dies war aber auch zu erwarten, die anderen Algorithmen deutlich stärkere Modelle lernen. Erstaunlich ist das Abschneiden von RegENDER. Nach [10] müsste RegENDER deutlich besser abschneiden als Lineare Regression. Hier wurde aber im Gegensatz zu der Version aus [10] der nicht optimierte RegENDER mit Standardparametern getestet.

Bei diesem Test ist auch besonders auffällig, dass der Durchschnitt des RRSE von allen Datenmengen kein gutes Maß zum Vergleichen ist. Aufgrund des Durchschnitts müsste RegENDER und MPL hinter M5Rule-R liegen. MPL ist auch nach dieser Methode deutlich schlechter als ReCo. Mit dem Ranking-Verfahren liegen sie aber auf dem gleichen Rang.

Genauso wie im Abschnitt 4.4 wurde auch hier der Friedman-Test durchgeführt. Der Friedman-Test mit 9 verschiedenen Algorithmen und 21 Datenmengen ergibt den Wert $F_F = 7{,}47$. Für ein Signifikanzniveau von $p = 0,05$ ergibt sich eine Signifikanzgrenze von 1,997. Daraus folgt, dass es signifikante Algorithmen bei diesem Test gibt. Der Nemenyi-Test ergibt eine

Abbildung 2: Nemenyi-Test der verschiedenen Regressionslerner für p = 0,05



Kritische Distanz von 2,62. Aus der Grafik 2 kann man entnehmen, dass M5Rules, SVM und Lineare Regression signifikant besser sind als ConjunctiveRule. ConjunctiveRule und M5Rules-R sind signifikant schlechter als M5Rules. RegENDER, DecisionTable, MPL, ReCo, LinearRegression, SVM, M5Rules liegen alle in einer Signifikanzgruppe. Diese Algorithmen unterscheiden sich also nicht signifikant voneinander. Insgesamt kann man feststellen, das ReCo sich nicht signifikant von den getesteten Algorithmen unterscheidet.

4.6 Runtime-Effizienz

In Tabelle 11 ist eine Auflistung der benötigten Rechenzeit zum Erstellen eines Modells für alle 21 Datenmengen. Dies sind die Werte ohne Crossvalidation. Hier gibt es keine große Überraschung, so sind ConjunktiveRule und Lineare Regression wie erwartet sehr schnell, MLP und SVM brauchen am längsten. ReCo ist zwar der Langsamste der hier verwendeten Regellerner, allerdings wurde ReCo auch nicht auf Schnelligkeit optimiert, so wie M5Rules oder RegENDER.

Tabelle 11: Runtime der Algrithmen

Algorithmus	Zeit in min
CR	0:01
LR	0:02
RegE	0:20
M5R	0:49
DT	1:12
M5R-R	2:19
ReCo	3:37
MPL	6:24
SVM	10:04

5 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde erfolgreich eine Adaption von Separate-and-Conquer auf Regression implementiert und getestet. Dabei sollte das Lernen und die Evaluation der gelernten Regeln auf unterschiedlichen Beispielmengen erfolgen. Es wurde die neuartige Reduce-Strategie eingeführt, mit der es möglich ist, während des Lernens die Beispiele zu verändern. Der Algorithmus wurde mit den verschiedenen Charakterisierungen von Separate-and-Conquer verglichen und die Unterschiede wurden aufgezeigt.

Das ursprüngliche Abbruchkriterium erwies sich als ungünstig, da schon nach wenigen Regeln abgebrochen wurde. Dieser Effekt wurde untersucht und mit den Vorschauregeln eine Lösung für dieses Problem vorgestellt. Weiterhin wurden verschiedene Konfigurationen des Algorithmus implementiert und getestet. So wurde die Mindestabdeckung der Regeln verändert, die Beispiele nach jeder gefundenen Regel gemischt und die Heuristik verändert. Es wurde eine Methode zur schnelleren Berechnung von Splitpoints vorgestellt. Somit konnte die Laufzeit von ReCo deutlich verbessert werden, ohne dass gleichzeitig die Genauigkeit der gefundenen Theorie abnimmt.

Diese insgesamt 19 verschiedenen Konfigurationen wurden getestet und miteinander verglichen. Die Beste dieser Konfigurationen wurde dann mit einigen in *weka* implementierten Regressionslernern verglichen. Es konnte festgestellt werden, das ReCo nicht signifikant unterschiedlich zu diesen Algorithmen ist.

Somit konnte gezeigt werden, das diese Adaptionen des Separate-and-Conquer Algorithmus auf Regression ähnlich gut lernt wie die vorhandenen Regressionslerner.

Dieser Algorithmus lässt aber auch noch Platz für weitere Verbesserungen. Zum einen wäre es interessant zu sehen inwieweit die Änderung des Suchalgorithmus sich auswirkt. So könnte man das in ReCo verwendeten Hill-Climbing leicht durch Beam Search ersetzen. Auch die in [22] vorgeschlagene Methode zur Splitpointberechnung kann mit ReCo verwendet werden. Es wäre weiter zu untersuchen, ob die in [22] vorgeschlagene Fehlerberechnung mithilfe von RRSE und der Abdeckung ReCo verbessert.

Literatur

[1] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[2] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[3] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] R. Callan. *The essence of neural networks*. The essence of computing series. Prentice Hall Europe, 1999.

[5] S. S. Chris Atkeson, Andrew Moore. Locally weighted learning. *AI Review*, 11:11–73, April 1997.

[6] P. Clark and T. Niblett. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[7] W. W. Cohen. Efficient pruning methods for separate-and-conquer rule learning systems. In *IJCAI*, pages 988–994, 1993.

[8] W. W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995.

[9] L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor clint. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–191. Academic Press, London, 1992.

[10] K. Dembczynski, W. Kotlowski, and R. Slowinski. Solving regression by learning an ensemble of decision rules. In *ICAISC*, pages 533–544, 2008.

[11] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[12] N. Y. Foo, editor. *Advanced Topics in Artificial Intelligence, 12th Australian Joint Conference on Artificial Intelligence, AI '99, Sydney, Australia, December 6-10, 1999, Proceedings*, volume 1747 of *Lecture Notes in Computer Science*. Springer, 1999.

[13] J. H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19:1–67, 1991.

[14] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[15] J. H. Friedman and B. E. Popescu. Importance sampled learning ensembles, 2003.

[16] J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008.

[17] J. H. Friedman and W. Stuetzle. Projection pursuit regression. *Journal of the American Statistical Association*, 76:817–823, 1981.

[18] J. Fürnkranz. Top-down pruning in relational learning. In *ECAI*, pages 453–457, 1994.

[19] J. Fürnkranz. Separate-and-conquer rule learning. *Artif. Intell. Rev.*, 13(1):3–54, 1999.

[20] J. Fürnkranz and G. Widmer. Incremental reduced error pruning. In *ICML*, pages 70–77, 1994.

[21] L. Iman and J. M. Davenport. Approximations of the critical region of the friedman statistic. *Communications in Statistics*, pages 571–595, 1980.

[22] F. Janssen and J. Fürnkranz. Separate-and-conquer regression. Technical Report TUD-KE-2010-01, TU Darmstadt, Knowledge Engineering Group, 2010.

[23] F. Janssen and J. Fürnkranz. Heuristic rule-based regression via dynamic reduction to classification. In T. Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1330–1335, 2011.

[24] D. F. Kibler, D. W. Aha, and M. K. Albert. Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5:51–57, 1989.

[25] L. Mason, J. Baxter, P. Bartlett, and M. Frean. Functional gradient techniques for combining hypotheses. In A. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*, pages 221–246, Cambridge, MA, 2000. MIT Press.

[26] R. S. Michalski. On the quasi-minimal solution of the covering problem. In *FCIP*, pages 125–128, 1969.

[27] R. S. Michalski. A theory and methodology of inductive learning. *Artif. Intell.*, 20(2):111–161, 1983.

[28] R. S. Michalski, I. Mozetic, J. Hong, and N. Lavrac. The multi-purpose incremental learning system aq15 and its testing application to three medical domains. In *AAAI*, pages 1041–1047, 1986.

[29] T. M. Mitchell. The need for biases in learning generalizations. Technical report, Rutgers University, New Brunswick, NJ, 1980.

[30] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–99, 1990.

[31] J. R. Quinlan. *Learning with Continuous Classes*, volume 92, pages 343–348. Citeseer, 1992.

[32] J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: Foil and related systems. *New Generation Comput.*, 13(3&4):287–312, 1995.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 673–695. MIT Press, Cambridge, MA, USA, 1988.

[34] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14:199–222, August 2004.

[35] P. E. Utgoff. Adjusting bias ln concept learning. In *IJCAI*, pages 447–449, 1983.

[36] S. M. Weiss and N. Indurkhya. Reduced complexity rule induction. In *IJCAI*, pages 678–684, 1991.

[37] S. M. Weiss and N. Indurkhya. Rule-based machine learning methods for functional prediction. *J. Artif. Intell. Res. (JAIR)*, 3:383–403, 1995.

[38] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.