Bachelor-Arbeit

Erweiterung des Algorithmus CHAID auf stetige Zielgrößen

Sarah Maierhofer

Ludwig-Maximilians-Universität München, Institut für Statistik

Betreuer: Prof. Dr. Torsten Hothorn

7. August 2009

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Was ist CHAID? 3
1.2 Ziel der Bachelor-Arbeit 4

2 Der Algorithmus CHAID **5**
2.1 Der Algorithmus 5
2.2 Signifikanz der Prädiktoren 6

3 Test für metrische Zielgröße und kategoriale Einflussgröße **7**
3.1 Der theoretische Rahmen 7
3.2 Auswahl eines geeigneten Tests 8

4 Implementierung im R-Paket CHAID **9**
4.1 Hilfsfunktionen 9
4.2 Programme zu den Schritten 1 bis 5 in CHAID 11

5 Simulation zum Test der Macht und Unverzerrtheit von CHAID **13**
5.1 Aufbau der Simulationsstudie 13
5.2 Ergebnisse der Simulationsstudie 14

6 Test der Vorhersagegüte von CHAID anhand realer Datensätze **15**
6.1 Aufbau der Simulation 15
6.2 Ergebnisse 17

7 Diskussion **19**

A R-Code **20**
A.1 Das Beispiel USvote 20
A.2 CHAID für stetige Zielgrößen 21
A.3 Simulation 31
A.4 Vorhersagegüte 36

B Beigefügte CD **40**

Abbildungsverzeichnis

1 Entscheidungsbaum für die US-Wahl 2000: Gore oder Bush 4
2 Simulierte Power und bedingte Wahrscheinlichkeit für den korrekten Split 15
3 Vorhersagefehler von CHAID und CTREE für BostonHousing . . 17
4 Vorhersagefehler von CHAID und CTREE für Ozone 18
5 Vorhersagefehler von CHAID und CTREE für Servo 18

Tabellenverzeichnis

1 Die Variablen im simulierten Datensatz 14
2 Wahrscheinlichkeit für die Variablenauswahl 16
3 Die Datensätze zur Simulation der Vorhersagegüte 16
4 Kruskal-Wallis-Tests auf Gleichheit der Vorhersagefehler von CHAID und CTREE 18

Zusammenfassung

Der Algorithmus CHAID berechnet einen Entscheidungsbaum für kategoriale Ziel- und Einflussgrößen. Ziel der Bachelor-Arbiet ist es, CHAID so zu erweitern, dass es auch für stetige Zielgröße anwendbar ist. Die wichtigsten Schritte zur Implementation werden kurz erläutert.
Außerdem enthält die Arbeit Benchmark-Experimente zum Vergleich von CHAID mit CTREE. Dabei werden Verzerrtheit, Power und Vorhersagegüte untersucht. Es zeigt sich, dass CHAID hin zu binären Kovariablen verzerrt ist und etwas höhere Power hat. In der Vorhersagegüte sind die beiden Verfahren jedoch ähnlich.

1 Einleitung

1.1 Was ist CHAID?

Das Akronym CHAID steht für **Ch**i-squared **A**utomatic **I**nteraction **D**etector. CHAID ist also ein Algorithmus der mittels χ^2-Tests automatisch Zusammenhänge in Datensätzen finden soll. Dazu wird ein Entscheidungsbaum erstellt, dessen Endknoten möglichst homogen bezüglich der Zielgröße sind und der zusätzlich durch die Reihenfolge der Splits zeigt, welche Prädiktoren die abhängige Variable besonders gut erklären.
Da als Entscheidungsgrundlage der χ^2-Test dient, funktioniert CHAID für kategoriale Einfluss- und Zielgrößen. Die genaue Funktionsweise wird in Kapitel 2 erklärt.

Die Anwendung von CHAID soll anhand eines Beispieles kurz illustriert werden: Der Datensatz USvote [1] enthält Daten zur Wahl des US-Präsidenten im Jahr 2000. Dabei wurde die Wahlentscheidung (Gore oder Bush), sowie soziodemografische Variablen (Geschlecht, Alter, Berufsstand, Bildung, Familienstand) erhoben. Das Ziel ist, die Wahlentscheidung anhand der soziodemografischen Variablen zu erklären. Es wurde ein Modell mit CHAID gefittet, das den Baum in Abbildung 1 liefert.

Es fällt auf, dass CHAID nicht nur binäre Splits erzeugt, sondern ein Knoten in beliebig viele Unterknoten aufgeteilt werden kann (zum Beispiel wird Knoten 2 in die Knoten 3, 4 und 5 gesplittet). Außerdem können mehrere Kategorien eines Prädiktors zusammengelegt werden und zum gleichen Unterknoten führen (zum Beispiel von Knoten 1 zu Knoten 8).
Man erkennt, dass die größte Abhängigkeit zwischen der Wahlentscheidung und dem Familienstand besteht. Der Datensatz wird zunächst nach verheiratet und verwitwet/geschieden/nie verheiratet aufgeteilt. Dann wird weiter nach Alter und Geschlecht gesplittet. Diese beiden Variablen sind also ebenfalls geeignet, um die Zielgröße zu erklären. Verheiratete im Alter von 18-34 bzw. 35-54 wählen mit einer

[1] im R-Paket CHAID als Beispieldatensatz vorhanden

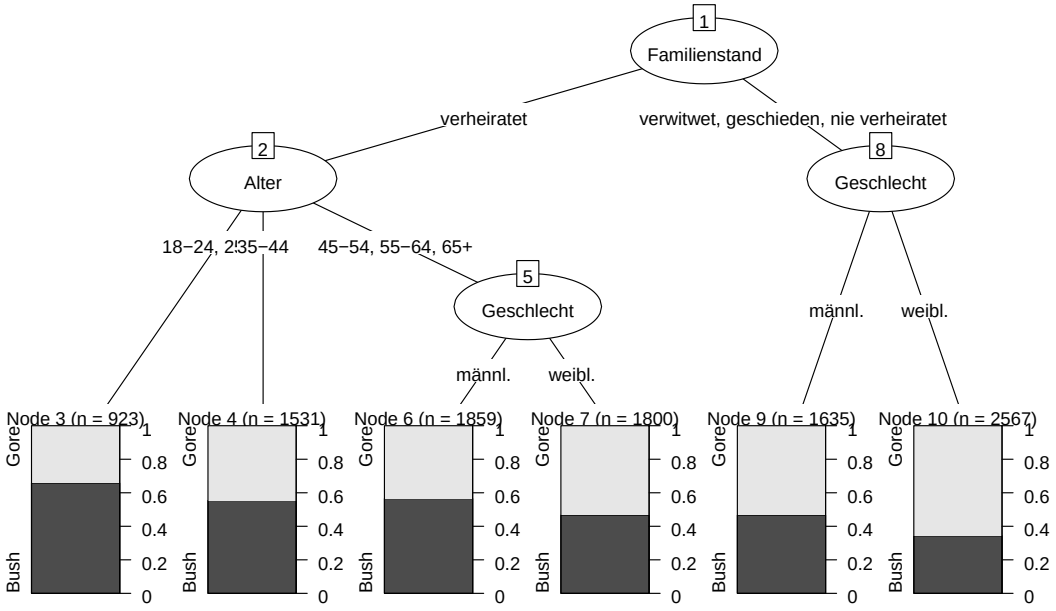

Abbildung 1: Entscheidungsbaum für die US-Wahl 2000: Gore oder Bush

Wahrscheinlichkeit von rund 60% Bush. Hingegen wählen Frauen, die verwitwet, geschieden oder nie verheiratet sind Bush mit deutlich geringerer Wahrscheinlichkeit von gut 20%.
Der Datensatz wird also in einer Baumstruktur in mehrere Endknoten aufgeteilt, die bezüglich der Wahlentscheidung möglichst homogen sein sollen. Allerdigs kann man durch die Wahl von Hyperparametern, wie Mindestanzahl an Beobachtungen pro Endknoten oder Signifikanzniveau zum Splitten, dir Tiefe des Baumes variieren. Der R-Code zu diesem Beispiel findet sich im Anhang A.1.

1.2 Ziel der Bachelor-Arbeit

In dieser Bachelor-Arbeit soll der Algorithmus CHAID so erweitert werden, dass er auch für metrische Zielgrößen funktioniert. Der wesentliche Punkt ist, dass der χ^2-Unabhängigkeitstest ausschließlich für zwei kategoriale Größen anwendbar ist. Für die Erweiterung benötigt man aber einen Test, der die Unabhängigkeit einer metrischen Größe von einer kategorialen misst. Zudem soll die alte Funktionalität erhalten bleiben.
Für R ist CHAID im Paket **CHAID** (The FoRt Student Project Team 2009) implementiert. Der bestehende R-Code soll entsprechend abgeändert werden.
Geeignete Tests lassen sich mit dem Paket **coin** (Hothorn et al. 2008) berechnen, das flexible Tests auf Unabhängigkeit von Variablen ermöglicht (siehe Kapitel 3).
In der Bachelor-Arbeit wird zunächst der Algorithmus CHAID und ein geeigneter

Unabhängigkeitstest erklärt (Kapitel 2, 3). Es geht also darum zu erklären, was schon vorhanden ist. Dann wird kurz darauf eingegangen, welche Schritte für die Implementierung notwendig sind (Kapitel 4). Abschließend folgen Tests auf Unverzerrtheit, Power (Kapitel 5) und Vorhersagegüte (Kapitel 6) von CHAID.

2 Der Algorithmus CHAID

Der Algorithmus CHAID wurde erstmals von G. V. Kass in dem Paper "An Exploratory Technique for Investigating Large Quantities of Categorical Data" vorgestellt. Wie oben erwähnt, steht das Akronym CHAID für "**Ch**i-squared **A**utomatic **I**nteraction **D**etector".
Anwenden kann man CHAID, wenn man eine kategoriale Zielgröße durch mehrere kategoriale Einflussgrößen erklären möchte. Es ist insbesondere auch geeignet, wenn man sehr viele Prädiktoren hat und herausfinden möchte, welche davon die abhängige Variable am besten erklären.
CHAID ist ein rekursiver Algorithmus, der in mehreren Schritten abläuft. Zuerst wird für jeden Prädiktor die optimale Aufteilung bestimmt. Das heißt, es wird getestet, welche Faktorstufen zusammengelegt werden können, um die größte Abhängigkeit mit der Zielgröße zu erzeugen. In einem zweiten Schritt wird getestet, welcher optimal zusammengelegte Prädiktior die größte Abhängigkeit mit der Zielgröße aufweist und der Datensatz entsprechend geteilt. Jeder dieser Teildatensätze wird unabhängig von den anderen, wie eben beschrieben, weiter analysiert und so gegebenenfalls wieder aufgeteilt.
Das Skalenniveau des Prädiktors bestimmt dabei die möglichen erlaubten Aufteilungen seiner Kategorien. Handelt es sich um einen nominalen Prädiktor dürfen beliebige Kategorien zusammengelegt werden. Bei einer ordinalen Variable hingegen ist nur das Zusammenlegen benachbarter Kategorien zulässig. Die Kategorien werden so zusammengelegt, dass die Kontingenztabelle mit der Zielgröße den kleinsten p-Wert bei einem χ^2-Test (zur Berechnung: siehe Kapitel 2.2) hat.

2.1 Der Algorithmus

Man betrachte zunächst eine abhängige Variable mit $d \geq 2$ Kategorien und einen bestimmten Prädiktor mit $c \geq 2$ Kategorien. Der Algorithmus lüuft in fünf Schritten ab (Kass 1980, S. 121):

Schritt 1: Für jeden Prädiktor wird eine Kreuztabelle mit der abhängigen Variable angelegt und die Schritte 2 und 3 ausgeführt.

Schritt 2: Man sucht das Paar von Kategorien (nur zulässige Paare je nach Skalenniveau der Kovariabel, siehe oben), deren $2 \times d$ Untertabelle sich

am wenigsten signifikant unterscheidet. Wenn der p-Wert einen kritischen Wert nicht erreicht, dann fügt man die zwei Kategorien zusammen. Im weiteren werden die zusammengefügten Kategorien als eine Kategorie betrachtet und Schritt 2 wiederholt, bis kein weiteren Kategorien mehr zusammengefügt werden können.

Schritt 3: (optional) Für jede zusammengefügte Kategorie, die drei oder mehr der ursprünglichen Kategorien enthält, sucht man den signifikantesten binären Split (der zulässig ist). Wenn der p-Wert kleiner als ein kritischer Wert ist, dann splittet man die Kategorie und geht zurück zu Schritt 2. Um eine Teilung auszuschließen, die nur eine vorangegangene Situation wieder herstellt, muss das Kriterium zum Splitten strikter sein, als das Kriterium für das Zusammenlegen von Kategorien.

Schritt 4: Man berechnet die Signifikanz (siehe unten, Kapitel 2.2) für jeden optimal zusammengefügten Prädiktor und speichert denjenigen mit dem kleinsten p-Wert. Wenn der p-Wert kleiner ist, als ein kritischer Wert, teilt man die Daten entsprechend der (zusammengefügten) Kategorien des gewählten Prädiktors.

Schritt 5: Für jeden Teil der Daten, der noch nicht analysiert wurde, beginnt man wieder mit Schritt 1. Man kann Datenteile mit einer zu geringen Fallzahl aus der weiteren Analyse ausschließen.

2.2 Signifikanz der Prädiktoren

Es ist nötig in Schritt 4 die Signifikanz der Unabhängigkeit von einem optimal zusammengefassten Prädiktor und der Zielgröße zu bestimmen (siehe Kass 1980, S. 122). Das Problem ist, dass die Kategorien des Prädiktors so zusammengelegt wurden, dass die Unabhängigkeit von der Zielgröße möglicht unwahrscheinlich ist. Die einfache Berechnung des p-Wertes in einem χ^2-Test würde den p-Wert also systematisch unterschätzen.
Wenn es keine Zusammenlegung im Prädiktor gab, kann natürlich der normale χ^2-Test verwendet werden. Wenn jedoch Kategorien zusammengelegt wurden, wird der p-Wert mittels Bonferroni adjustiert. Die Idee ist es, den p-Wert mit der Anzahl möglicher Zusammenlegungen von c Kategorien auf r Gruppen (mit $1 \leq r \leq c$) zu standardisieren. Wobei c die Anzahl der ursprünglichen Kategorien ist und r die Anzahl der Kategorien nach dem Zusammenlegen. Der angepasste p-Wert wird berechnet, indem man den p-Wert des χ^2-Test, mit der entsprechenden Anzahl von Möglichkeiten multipliziert. Die Anzahl möglicher Zusammenlegungen lässt sich für nominale und ordinale Variablen wie folgt berechnen:

(1) *Ordinale Variablen:* In diesem Falle sind nur Zusammenlegungen benachbarter

Kategorien zulässig.

$$B_{ordinal} = \binom{c-1}{r-1}.$$

(2) *Nominale Variablen:* Hier sind alle Gruppierungen von Kategorien möglich.

$$B_{nominal} = \sum_{i=0}^{r-1} (-1)^i \frac{(r-i)^c}{i!(r-i)!}.$$

Kass nennt zudem floating predictors. Das sind Variablen, die ordinales Skalenniveau haben, die allerdings auch eine Kategorie besitzen, die sich nicht in die ordinale Struktur einordnen lässt. Zum Beispiel eine Kategorie "sonstiges" auf einer Meinungsskala von 'schlecht' bis 'gut'. Allerdings ist dieser Fall nicht implementiert und wird daher nicht näher erläutert.

3 Test für metrische Zielgröße und kategoriale Einflussgröße

Wie bereits erwähnt, wird der nötige Test für metrische Zielgröße und eine kategoriale Einflussgröße mit dem Paket **coin** berechnet. Daher wird zunächst die Theorie, die hinter diesem Paket steckt, kurz erklärt. Davon ausgehend kann man einen geeigneten Test für das vorliegende Testproblem auswählen.

3.1 Der theoretische Rahmen

Das Paket **coin** implementiert Permutationstests nach der Theorie von Strasser und Weber (Strasser und Weber 1999, siehe Hothorn et al. 2008 S. 3f und Hothorn et al. 2006 S. 257f). Es geht darum Tests auf Unabhängigkeit von beliebigen Variablen $\mathbf{Y}$ und $\mathbf{X}$ zu konstruieren, die aus Zufallsräumen $\mathcal{Y}$ und $\mathcal{X}$ stammen. Diese Variablen können ein beliebiges Skalenniveau haben und auch multivariat sein.
Die zu testende Nullhypothese auf Unabhängigkeit ist:

$$H_0 : D(\mathbf{Y}|\mathbf{X}) = D(\mathbf{Y})$$

Zum Testen dieser Hypothese werden Teststatistiken folgender Form verwendet:

$$\mathbf{T} = vec\left(\sum_{i=1}^{n} w_i g(\mathbf{X}_i) h(\mathbf{Y}_i)\right) \in \mathbb{R}^{pq \times 1}.$$

Dabei ist vec der Operator, der die Spalten einer Matrix zu einem Vektor zusammenfügt. Die Funktion $g : \mathcal{X} \to \mathbb{R}^{p \times 1}$ ist die Transformation auf die Ausprägungen von $\mathbf{X}$. Zum Beispiel für Faktoren ist diese Transformation meist die Bildung von

Indikatorvariablen.
Die Funktion $h : \mathcal{Y} \to \mathbb{R}^{q \times 1}$ wird auch Einflussfunktion (engl. influence function) genannt. Diese Funktion kann von allen Ausprägungen von $\mathbf{Y}$ abhängen, also $h(\mathbf{Y}_i) = h(\mathbf{Y}_i, (\mathbf{Y}_1 \dots \mathbf{Y}_n))$. Allerdings nur in der Form, dass die Reihenfolge, in der die $\mathbf{Y}$-Werte beobachtet wurden keine Rolle spielt. Für Faktoren kann diese Transformation wieder die Kodierung in Indikatorvariablen sein. Für metrische Zielgrößen sind die Identitätsfunktion oder die Abbildung auf die Ränge üblich.
Die Gewichte w_i sind ganze Zahlen, die angeben, wie oft eine Beobachtung $(\mathbf{Y}_i, \mathbf{X}_i)$ gemessen wurde. Per default ist $w_i \equiv 1$ (Hothorn et al. 2006, S. 258).

Natürlich ist die Verteilung von $\mathbf{T}$ abhängig von der gemeinsamen Verteilung von $\mathbf{Y}$ und $\mathbf{X}$. Allerdings kann man diese Abhängigkeit zumindest unter der Nullhypothese beseitigen, indem man $\mathbf{X}_1, \dots, \mathbf{X}_n$ festhält und auf alle möglichen Permutationen S von $\mathbf{Y}_1, \dots, \mathbf{Y}_n$ bedingt. Der konditionale Erwartungswert $\mu \in \mathbb{R}^{pq}$ und die Kovarianzmatrix $\Sigma \in \mathbb{R}^{pq \times pq}$ von $\mathbf{T}$ wurden unter H_0 gegeben alle Permutationen $\sigma \in S$ von Weber und Strasser (1990) abgeleitet.
Mit dem konditionalen Erwartungswert und der Kovarianzmatrix kann man eine beobachtete Statistik $\mathbf{t} \in \mathbb{R}^{pq}$ standardisieren und zu einer eindimensionalen Teststatistik $c = c(\mathbf{t}, \mu, \Sigma)$ aggregieren. Für $c(\cdot)$ kann man beispielsweise eine quadratische Form oder das Maximum wählen (Hothorn et al. 2006, S. 258).

Durch die Wahl von $g(\cdot)$, $h(\cdot)$, und $c(\cdot)$ kann man flexibel Unabhängigkeitstests für beliebige $\mathbf{Y}$- und $\mathbf{X}$-Variablen konstruieren. Selbsverständlich ist es auch möglich in diesem theoretischen Rahmen bereits bekannte Unabhängigkeitstest, wie den χ^2-Test zu berechnen.

3.2 Auswahl eines geeigneten Tests

Für die Erweiterung von CHAID auf metrische Zielgrößen wird ein Test benötigt, der die Unabhängigkeit einer metrischen Zielgröße von einer kategorialen Einflussgröße testet. Dazu verwendet man für die Einflussgröße $\mathbf{X}$ die Transformation auf Indikatorvariablen. Für einen Faktor auf k Stufen werden k Indikatorfunktionen benutzt. Nur für $k = 2$ wird eine eindimensionale Indikatorfunktion benutzt. Ein Faktor mit $k = 3$ Kategorien wird beispielsweise folgendermaßen transformiert:
$g(\mathbf{X}_i) = (1, 0, 0)^T$ für $\mathbf{X}_i$ in der ersten Kategorie.
Die Funktion $h(\cdot)$ setzt man auf die Identität. Man rechnet also dirket mit den beobachteten Werten.
Für die Berechnung der Teststatisitk wird die Funktion $c(\cdot)$ auf die quadratische Form gesetzt. Es gilt also:

$$c_{quad}(\mathbf{T}, \mu, \Sigma) = (\mathbf{T} - \mu)\Sigma^+(\mathbf{T} - \mu)^T$$

mit der Moore-Penrose-Inversen Σ^+ von Σ.

Damit ist der neue Test möglichst nahe am ursprünglich verwendeten χ^2-Test, den man durch folgende Transformationen erhält: $g(\cdot)$ und $h(\cdot)$ sind die Abbildung auf Indikatorvariablen und die Teststatisitk wählt man quadratisch. Mit einer geeigneten Skalierung erhält man exakt den altbekannten χ^2-Unabhängigkeitstest.

4 Implementierung im R-Paket CHAID

Der Algorithmus CHAID wurde im Rahmen des Kurses "Fortgeschrittene Programmierung mit R" im Wintersemester 2008/2009 bei Prof. Dr. Hothorn an der LMU für R implementiert. Die nötigen Funktionen stehen im Package **CHAID** auf der Seite: `http://r-forge.r-project.org/projects/chaid/` zum Download zur Verfügung. Eine kurze Beschreibung findet sich in "CHAID: CHi-squared Automated Interaction Detection." (The FoRt Student Project Team 2009).
Zunächst eine Übersicht zu den nötigen Unterprogrammen. Aufgeteilt nach Hilfsfunktionen und Funktionen, die direkten Bezug zu den Schritten im Algorithmus haben.

4.1 Hilfsfunktionen

Für die Implementaion von CHAID ist es nötig die Kategorien von Variablen zusammenzulegen und auch zu splitten. Dazu wird mit einem Index gearbeitet. Der Index ist zunächst ein Vektor, der die Zahlen von 1 bis zur Anzahl der Kategorien enthält.
Der Index zu einer Variable mit vier Levels ist also der Vektor $(1, 2, 3, 4)$. Levels, die zusammengelegt werden sollen, erhalten den gleichen Index. Sollen nun die ersten beiden Kategorien zusammengelegt werden, so wird der Index zu $(1, 1, 2, 3)$. Für das Zusammenlegen der letzten drei Kategorien ist der Index $(1, 2, 2, 2)$. Auf diese Art können beliebige Zusammenlegungen und auch wieder Aufteilungen der Kategorien einfach gespeichert werden.

mergelevels (index, merge) legt zwei Levles (Übergabe in merge) des Indexes (index) zusammen. Als Beispiel: $mergelevels(c(1, 2, 3, 4), c(1, 2))$ gibt die Ausgabe $(1, 1, 2, 3)$. Es werden also die ersten beiden Kategorien zusammegelegt.

splitlevels (index, level, split) splittet zusammengelegte Levels eines Indexes wieder auf. Zum Beispiel: $splitlevels(c(1, 1, 1, 2, 2), 1, 2)$ gibt als Ausgabe den neuen Index $(1, 2, 1, 3, 3)$. Es wird also die zweite Kategorie in der ersten Zusammenlegung wieder abgespalten.

mergex (x, index) legt die Kategorien der Variable x entsprechend des Indexes zusammen. Diese Funktion überträgt die Information aus dem Index also tatsächlich auf die Daten.

Weiterhin ist es nötig den χ^2-Test beziehungsweise den Unabhängigkeitstest für eine metrische von einer kategorialen Größe zu berechnen. Dabei müssen auch Fälle berücksichtigt werden, die beim Ausführen den Tests zu einer Fehlermeldung führen würden. Beispielsweise, wenn in einer Variable nur noch Beobachtungen in einer Kategorie vorhanden sind. Zudem können sehr kleine p-Werte auftreten. Diese möchte man trotzdem miteinander vergleichen können. Die Genauigkeit kann erhöht werden, indem man statt der p-Werte die logarithmierten p-Werte betrachtet. Man erhält so einen Wertebereich von minus unendlich bis null. Da der Logarithmus eine monotone Transformation ist, ist es äquivalent, ob man die logarithmierten p-Werte oder die p-Werte sortiert.

logid_test (x, response) ersetzt **logchisq.test**(x): zunächst zu logchisq.test(): diese Funktion nimmt als Übergabeparameter eine Kontingenztabelle x, die eine Einflussgröße und die Zielgröße enthält. Für Tabellen mit vielen Beobachtungen wird der χ^2-Test berechnet. Bei geringer Fallzahl ($n < 100$) oder geringer Zellbesetzung ($n_{ij} < 10$) wird die Verteilung der Teststatistik simuliert. Wenn in einer Variable nur noch Ausprägungen in einer Kategorie vorhanden sind, wird null zurückgegeben. Das entspricht dem maximal möglichen p-Wert auf der log-Skala. In diesem Fall ist schließlich kein Zusammenhang mehr zwischen den Variablen feststellbar.
Der Funktion logid_test() übergibt man die Zielgröße und den Response als Vektoren. Dafür spielt das Skalenniveau der Variablen keine Rolle. Ähnlich wie in logchisq.test() wird die Verteilung der Teststatistik für geringe Fallzahlen ($n < 100$) simuliert. Wenn alle Wete von response gleich sind, oder x nur noch eine Kategorie hat, wird null zurückgegeben.

Einer weiteren Hilfsfunktion kann man alle Hyperparameter des Algorithmus CHAID übergeben, sodass man an jeder Stelle des Codes auf die nötigen Parameter zugreifen kann. Es sind jedoch für alle Parameter Default-Werte vorgegeben (Zur Bedeutung der Hyperparameter: siehe auch die Hilfe zur Funktion chaid() im Package CHAID).

chaid_control (alpha2 = 0.05, alpha3 = -1, alpha4 = 0.05,
minsplit = 20, minbucket = 7, minprob = 0.01, stump = FALSE)
speichert alle Hyperparameter als Liste.

- **alpha2** Signifikanzniveau, das zum Zusammenlegen der Einflussgrößen in Schritt 2 verwendet wird.
- **alpha3** Wenn alpha3 einen positiven Wert < 1 hat, ist es das Signifikanzniveau, das für das Teilen zuvor zusammengelegter Kategorien in Schritt 3 verwendet wird. Sonst wird Schritt 3 nicht ausgeführt.
- **alpha4** Signifikanzniveau, das in Schritt 5 mindestens erreicht werden muss, damit in einem Knoten nach dem signifikantesten Prädiktor gesplittet wird.

minsplit Minimale Anzahl an Beobachtungen in einem Knoten, bei der kein weiterer Split mehr ausgeführt wird.

minbucket Minimale Anzahl an Beobachtungen in einem Endknoten.

minprob Minimaler Anteil an Beobachtungen in einem Endknoten.

stump wenn stump TRUE, dann wird nur ein Split ausgeführt (sogenannter Root-Split).

4.2 Programme zu den Schritten 1 bis 5 in CHAID

Mit dem Parameter *response* wird immer die Zielvariable bezeichnet, die sich für einen Funktionsaufruf nicht ändert. Mit *weights* wird jeweils ein Vektor von Gewichten übergeben. Das ermöglicht zum einen die direkte Übergabe eines Datensatzes, der mehrfach gemachte Beobachtungen mit einer Gewichtsvariable kennzeichnet. Zum anderen kann man so berücksichtigen, dass in der Baumstruktur nur noch auf den Beobachtungen gerechnet wird, die sich in dem Knoten befinden, der weiter gesplittet werden soll. Dazu wird das Gewicht aller anderen Beobachtungen auf 0 gesetzt. Mit *ctrl* wird die Liste der Hyperparameter bezeichnet, die mitteles chaid_control() erstellt wird.
Die folgenden Funktionen mussten teilweise abgeändert werden, damit sie auch für metrische Zielgrößen funktionieren. Zum einen wurde der logchisq.test() immer durch den logid_test() ersetzt. Zum anderen müssen die Funktionen insgesamt auf Vektoren arbeiten und nicht mehr wie zuvor auf Kontingenztabellen.
Der häufig vorkommende *weights*-Parameter wurde mittels rep() auf die Vektoren angewendet: Wenn eine Variable x mit *weights* gewichtet werden soll, wurde das mit dem Befehl: $x < -rep(x, weights)$ erreicht.

step1internal (response, x, weights, index = NULL, ctrl) berechnet für eine Variable x, wenn notwendig die Schritte 2 und 3. Ausgegeben wird der Index zum optimalen Zusammenlegen der Kategorien. Wenn keine Kategorien zusammengelet werden sollen, enthält der Index die Zahlen 1,2,3... bis zur Anzahl der Kategorien von x.

step1 (response, xvars, weights, indices = NULL, ctrl) iteriert step1internal für alle Prädiktorvariablen xvars und gibt eine Liste zurück, die für jeden Prädiktor den Index enthält.

step2 (response, x, weights, index = 1:nlevels(x), ctrl) legt zwei Kategorien einer Variable x zusammen, wenn sich die Zielgröße in diesen Kategorien nicht stark unterscheidet. Dazu werden alle möglichen Zusammenlegungen der Kategorien ausprobiert und jeweils der Unabhängigkeitstest berechnet. Ist der maximale p-Wert größer als das vorgegebene alpha2, so werden diese beiden Kategorien zusammengelegt. step2 wiederholt das so lange bis keine weiteren Kategorien mehr zusammengelegt werden können oder die minimale Anzahl an Beobachtungen in einem Endknoten unterschritten ist.

step3 (x, y, weights, alpha3 = 0.049, index, kat) berechnet den Index für die Variable x neu, wenn mehr als drei Kategorien zusammengelegt wurden. Dazu werden mittels step3intern() die Kategorien berechnet, die aus einer Kategorie kat abgesplittet werden sollen und dann der neue Index bestimmt.

step3intern (x, y, weights, alpha3=0.05, index, kat) probiert alle binären Splits der zusammengelegten Kategorie kat einer Variable x, wenn in Schritt 2 mehr als drei Kategorien vereinigt wurden. Für diese neue Zusammenlegung der Kategorien wird jeweils der p-Wert auf Unabhängigkeit von der Zielgröße berechnet. Wenn der minimale p-Wert kleiner ist als alpha3, so werden die Kategorien zurückgegeben, die abgesplittet werden sollen.

step4internal (response, x, weights, index) berechnet für die Variable x, entsprechend ihrer zusammengelegten Kategorien, den bonferroni-adjustierten p-Wert des Unabhägigkeitstest mit der Zielgröße.

step4 (response, xvars, weights, indices) iteriert step4internal über alle Einflussgrößen xvars und gibt den Vektor mit den p-Werten aller optimal zusammengelegter Prädiktoren zurück.

step5 (id = 1L, response, x, weights = NULL, indices = NULL, ctrl = chaid_control()) implementiert die Rekursion, indem für jeden Datenteil, der noch nicht analysiert wurde wieder bei step1 begonnen wird. Dabei werden die Abbruchkriterien aus ctrl beachtet.
Die Baumstruktur wird mittels des Pakets partysplit (Hothorn et al. 2009) abgespeichert. Dieses Paket ermöglicht auch die graphische Ausgabe der Ergebnisse.

chaid (formula, data, subset, weights, na.action = na.omit, control = chaid_control()) dient dazu die vom Benutzer beim Funktionsaufruf eingegebenen Informationen zu verarbeiten. So werden aus formula und data die Zielgröße (der response) und die Einflussgrößen bestimmt. Dem Parameter weights kann man optional einen Gewichtsvektor übergeben. Na.action legt fest, wie mit fehlenden Werten umgegangen wird. Per default werden Beobachtungen mit fehlenden Werten aus der Analyse ausgeschlossen. Control kann man ein Objekt der Funktion chaid_control übergeben, um Hyperparameter zu spezifizieren.
Außerdem wird die Funktion step5() aufgerufen, sodass die Rekursion in Gang kommt.
Ausgegeben wird ein Baumobjekt, das man direkt anschauen oder mittels plot() graphisch darstellen kann.

5 Simulation zum Test der Macht und Unverzerrtheit von CHAID

Zunächst soll mittels einer Simulation geprüft werden, ob die neue Implementierung des Algorithmus CHAID unverzerrt ist. Ein Algorithmus für rekursives Partitionieren heißt unverzerrt, wenn für m Kovariablen, die alle unabhängig von der Zielgröße sind, die Wahrscheinlichkeit in einer Kovariable zu splitten immer gleich $1/m$ ist. Das muss unabhängig davon gelten, auf welchem Skalenniveau die Variable erhoben wurde und ob es fehlende Werte gibt (vergleiche Hothorn et al. 2006 b, S. 664). In der momentanen Implementierung von CHAID werden Beobachtungen mit fehlenden Werten nicht berücksichtigt. Von Bedeutung ist hier also, ob CHAID unverzerrt ist bezüglich ordinaler und nominaler Kovariablen und auch für Kovariablen mit unterschiedlich vielen Faktorstufen.
Die Macht (englisch power) gibt an bei wie kleinen Unterschieden in der Zielgröße bezüglich einer Kovariable der Algorithmus splittet.
Um die Ergebnisse der Simulation besser einordnen zu können, werden die Regressionssbäume nicht nur mit CHAID, sondern auch mit CTREE berechnet. CTREE steht für Conditional Inference Trees und ist ein Verfahren zur Berechnung von Bäumen, das auf Variablen unterschiedlicher Skalenniveaus angewendet werden kann. Insbesondere kann man CTREE auch für kategoriale Einflussgrößen und einen metrischen Response anwenden. CTREE wurde von Hothorn et al. entwickelt und ist im R-Paket **party** implementiert (Hothorn et al. 2006 b). Dass CTREE unverzerrt ist zeigten Hothorn et. al. (2006 b).

5.1 Aufbau der Simulationsstudie

Um nun Unverzerrtheit und Macht von CHAID zu testen wird eine Simulationsstudie durchgeführt (Aufbau und R-Code zur Simulationsstudie wie in Hothorn et al. 2006 b und Thiemichen 2009). Eine Auswahl des benötigten R-Codes ist im Anhang A.3 abgedruckt. Die vollständige Simulation ist auf der beigefügten CD gespeichert.
Zunächst werden $B = 5000$ Datensätze mit jeweils $n = 100$ Beobachtungen und acht Variablen zufällig erzeugt. Die Einflussgrößen werden mit V1 bis V7 bezeichnet und unabhängig voneinander simuliert. Es handelt sich um ordinale und nominale Variablen mit unterschiedlich vielen Kategorien. Tabelle 1 gibt einen Überblick über den Aufbau des Datensatzes. Y ist die Zielgröße. Sie wird in Abhängigkeit von V7 folgendermaßen erzeugt:

$$Y \sim \begin{cases} \mathcal{N}(0,1), & \text{für V7} = 0 \\ \mathcal{N}(\mu,1), & \text{für V7} = 1 \end{cases}$$

Das heißt für $\mu = 0$ ist Y von allen Kovariablen unabhängig. Ist μ größer als 0 so ist Y abhängig von X7.

Variable	Skalenniveau	Faktorstufen
V1	ordinal	4
V2	ordinal	6
V3	ordinal	8
V4	nominal	4
V5	nominal	6
V6	nominal	8
V7	nominal	2
Y	metrisch	-

Tabelle 1: Die Variablen im simulierten Datensatz

Zum Test der Unverzerrheit wird für $\mu = 0$ ein Split erzwungen, indem kein Abbruchkriterium übergeben wird. Wenn der Algorithmus unverzerrt ist, so wird jede der Kovariablen mit der Wahrscheinlichkeit $1/m = 1/7 \approx 0.14$ zum Splitten ausgewählt.
Für die Macht des Testes betrachtet man die Wahrscheinlichkeit, dass gegeben es wird eine Kovariable zum Splitten ausgewählt es sich um die richtige Kovariable handelt. Dazu wird μ von 0 in Schritten von 0.1 bis 1 erhöht. Bei je kleinerem μ der Algorithmus dann in der richtigen Kovariable splittet, oder zumindest mit möglichst hoher Wahrscheinlichkeit in der richtigen Kovariable splittet, umso höher die Macht.

5.2 Ergebnisse der Simulationsstudie

In Abbildung 2 links ist die Wahrscheinlichkeit für einen Root-Split in Abhängigkeit vom Parameter μ dargestellt. Man erkennt, dass für μ gleich null, also Unabhängigkeit des Responses von allen Kovariablen, CHAID mit einer Wahrscheinlichkeit von gut 20% dennoch fälschlicherweise splittet. Bei CTREE wird diese Wahrscheinlichkeit durch $\alpha = 0.05$ (gepunktete Linie) kontrolliert. Für μ größer als null ist die Power von CHAID jeweils höher, als die von CTREE. Wobei der Unterschied mit steigendem μ abnimmt.
Die rechte Graphik von Abbildung 2 zeigt, dass die bedingte Wahrscheinlichkeit den korrekten Split auszuwählen für μ kleiner als 0.1 für CHAID größer ist als für CTREE. Ab diesem Wert liegt allerdings CTREE über CHAID. Ab μ gleich 0.7 ist die bedingte Wahrscheinlichkeit für einen korrekten Split annähernd eins.
In Tabelle 2 sind die geschätzten Wahrscheinlichkeiten für einen Split in den Variablen V1 bis V7 für Unabhängigkeit vom Response abgedruckt. Zusätzlich ist jeweils ein 95%-Konfidenzintervall nach Goodman (Goodman 1965) angegeben. Man erkennt, dass bei CHAID die Wahrscheinlichkeit in der binären Kovariable (V7) zu splitten deutlich höher ist, als in den anderen Variablen. Diese beträgt knapp 30% und das Konfidenzintervall enthält den Wert 14%, der bei gleicher Auswahlwahrscheinlichkeit für jede Variable gilt, nicht. Es zeigt sich also, dass

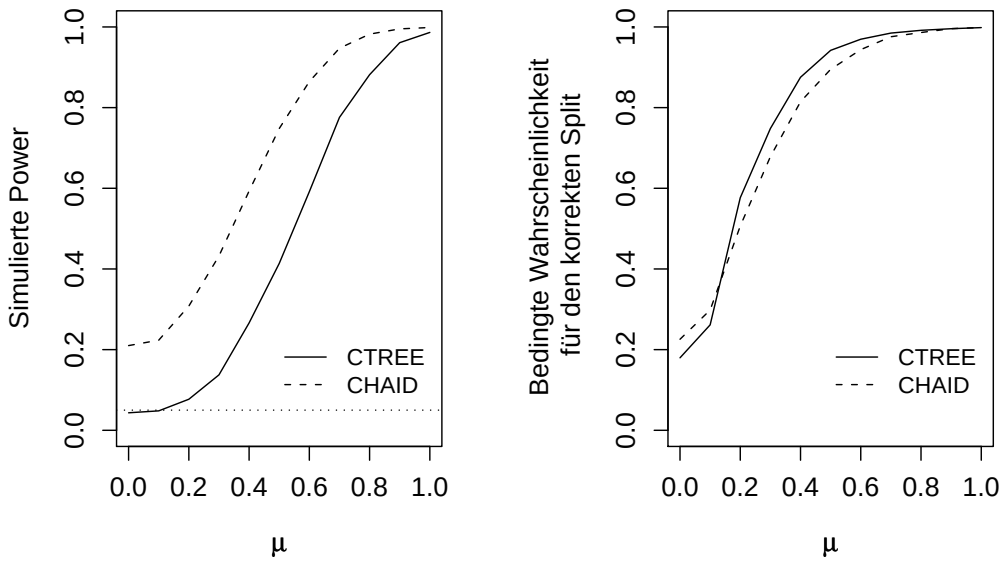

Abbildung 2: Simulierte Powerer (links) und bedingte Wahrscheinlichkeit für den korrekten Split (rechts) von CHAID und CTREE in Abhängigkeit von μ. Basierend auf 5000 Simulationsdurchläufen

CHAID zu binären Kovariablen verzerrt ist. Außerdem kann man ablesen, dass CHAID eher Variablen mit wenigen Faktorstufen auswählt, allerdings enthalten die Konfidenzintervalle den Wert 14%. Dieses Ergebnis deckt sich mit denen aus der Bachelor-Arbeit von Thiemichen zur Verzerrtheit von CHAID für kategoriale Einflussgrößen hin zu Kovariablen mit wenigen Kategorien (S. 18 f).

6 Test der Vorhersagegüte von CHAID anhand realer Datensätze

6.1 Aufbau der Simulation

Nun soll die Vorhersagegüte von CHAID auf realen Datensätzten geprüft werden (Aufbau und R-Code zum Test der Vorhersagegüte wie in Hothorn et al. 2006 b und Thiemichen 2009). Dazu wird in einem Benchmark-Experiment die Vorhersagegüte von CHAID mit der von CTREE verglichen.
Als Beispieldatensätze dienen BostonHousing, Servo und Ozone. Diese Datensätzte sind im R-Paket **mlbench** enthalten (zum Paket mlbench: Leisch und Dimitriadou, 2009). Die Zielgröße ist jeweils metrisch. Wenn es metrische Kovariablen

	CHAID		**CTREE**	
	Schätzer	95% KI	Schätzer	95% KI
V1	0.185	[0.168, 0.204]	0.152	[0.137, 0.169]
V2	0.135	[0.121, 0.151]	0.140	[0.125, 0.156]
V3	0.107	[0.095, 0.121]	0.136	[0.122, 0.152]
V4	0.140	[0.126, 0.156]	0.142	[0.128, 0.159]
V5	0.091	[0.080, 0.104]	0.137	[0.123, 0.153]
V6	0.056	[0.047, 0.066]	0.141	[0.126, 0.157]
V7	0.285	[0.263, 0.310]	0.153	[0.138, 0.170]

Tabelle 2: Simulierte Wahrscheinlichkeit (mit Konfidenzintervall) für einen Split in den Variablen V1 bis V7, die alle unabhängig vom Response sind. Bei Unverzerrtheit müsste jede Variabel mit der selben Wahrscheinlichkeit, also $1/7 \approx 0.14$ ausgewählt werden. Die Ergebnisse basieren auf 5000 Simulationsdurchläufen

gibt, so werden diese in ordinale Variablen mit zehn Kategorien umgewandelt, indem man sie an den entsprechenden 10%-Quantilen teilt. Dadurch erhält man drei Datensätze mit ausschließlich kategorialen Kovariablen und kann daher CHAID anwenden. Einen Überblick zu den verwendeten Datensätzen gibt Tabelle 3.

Die Vorhersagegüte wird simuliert, indem man zunächst aus einem Datensatz

Datensatz	n	NA	m	nominal	ordinal	metrisch
Boston Housing	506	-	13	-	-	13
Ozone	361	158	12	3	-	9
Servo	167	-	4	4	-	-

Tabelle 3: Die Datensätze zur Simulation der Vorhersagegüte: n bezeichnet die Beobachtungszahl, NA die Zahl der fehlenden Werte, m die Zahl der Kovariablen, nominal, ordinal und metrisch geben an wie viele Kovariablen es pro Skalenniveau gibt

eine nonparametrische Bootstrap-Stichprobe als Lerndatensatz zieht (allgemeines zu Bootstrap-Verfahren Hastie et al. 2008). Dabei zieht man mit Zurücklegen so viele Beobachtungen, wie im Originaldatensatz vorhanden sind. Auf dieser Lernstichprobe werden je ein Baum mit CHAID und CTREE berechnet. Für beide Verfahren werden die Default-Einstellungen der Hyperparameter verwendet. Der Testdatensatz wird out-of-bag bestimmt. Das heißt alle Beobachtungen aus dem Originaldatensatz, die nicht in der Lernstichprobe sind, kommen in die Teststichprobe. Für diese Stichprobe wird die Zielgröße mit beiden Verfahren prognostiziert und mit den wahren Werten verglichen. Als Maß für die Güte der Vorhersage wird das Mittel der quadrierten Abweichungen von prognostiziertem und beobachteten Wert betrachtet. Hier wurden $B = 100$ Bootstrap-Datensätzte gezogen.

6.2 Ergebnisse

Die Abbildungen 3, 4 und 5 zeigen jeweils den mittleren quadrierten Vorhersagefehler für die drei Datensätze. Man erkennt, dass die beiden Verfahren im Mittel ähnlich gut prognostizieren. Allerdings hat CHAID eine etwas größere Streuung der Fehler und mehr Ausreißer nach oben. Für den Datensatz BostonHousing sind die Vorhersagefehler ähnlich verteilt. Beim Datensatz Ozone hat CHAID tendenziell größere Fehler in der Prognose. Beim Datensatz Servo hat es jedoch geringere Abweichungen von den wahren Werten als CTREE. Insgesamt scheint die Vorhersagegüte der beiden Verfahren ähnlich zu sein.

Um zu testen, ob die beiden Verfahren im Mittel die gleiche Vorhersagegüte

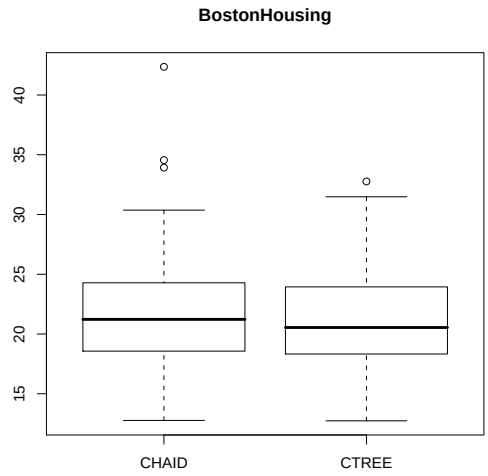

Abbildung 3: Simulierter Vorhersagefehler von CHAID und CTREE für den Datensatz BostonHousing, basierend auf B=100 Bootstrap-Stichproben

besitzen, wird ein Kruskal-Wallis-Rangsummentest durchgeführt (Kruskal-Wallis-Test Toutenbourg 2003 S. 221 ff). Dieser nonparametrische Test prüft durch den Vergleich von Rängen in unabhängigen Gruppen, ob die Verteilungen gleich sind. Die Ergebniss in Tabelle 4 zeigen, dass man die Gleichheit für die Datensätze Servo und Ozone ablehnen kann. Für das Beispiel BostonHousing wird die Nullhypothese beibehalten.

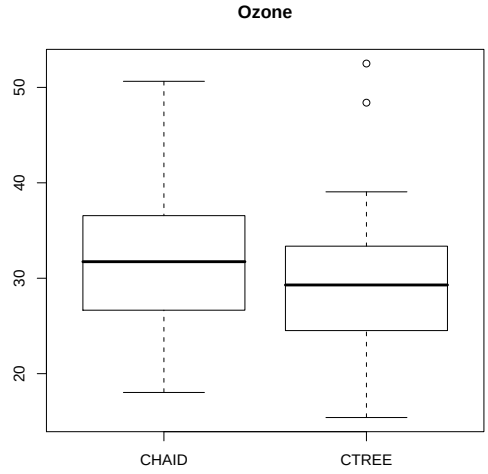

Abbildung 4: Simulierter Vorhersagefehler von CHAID und CTREE für den Datensatz Ozone, basierend auf B=100 Bootstrap-Stichproben

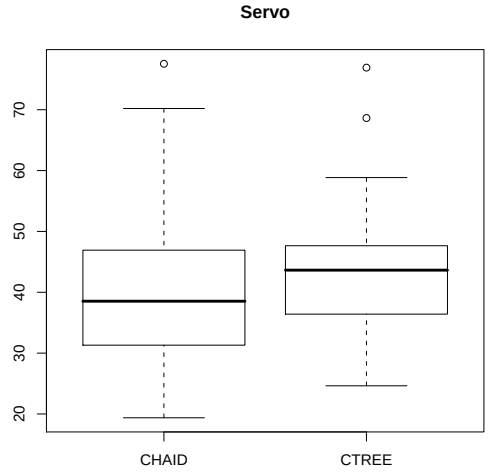

Abbildung 5: Simulierter Vorhersagefehler von CHAID und CTREE für den Datensatz Servo, basierend auf B=100 Bootstrap-Stichproben

Datensatz	Statistik	p-Wert
BostonHousing	0.71	0.40
Ozone	11.53	0.00
Servo	7.25	0.01

Tabelle 4: Kruskal-Wallis-Tests auf Gleichheit der Vorhersagefehler von CHAID und CTREE

7 Diskussion

Die Erweiterung von CHAID auf stetige Zufallsgrößen ist im Rahmen der vorhandenen Implementierung durch relativ wenige Änderungen der R-Codes möglich gewesen. Allerdings hat die neue Implementierung den Nachteil, dass sie mehr Rechenzeit benötigt, als die alte. Für die Berechnung des Unabhängigkeitstestes wird auf eine High-Level-Funktion zugegriffen und nicht mehr auf den einfachen Chi^2-Test. Weil für ein Baum-Modell viele Tests nötig sind, verlängert sich die Rechenzeit deutlich.
Außerdem steht mit dem Verfahren CTREE ein wesentlich allgemeineres Werkzeug zur Verfügung: CTREE ist auf unterschiedliche Kombinationen von Kovariablen und Zielvariabeln anwendbar und nicht wie CHAID auf kategoriale Kovariablen beschränkt. CTREE liefert zudem insgesamt ähnliche oder bessere Ergebnisse: Es hat sich gezeigt, dass CHAID im Gegensatz zu CTREE bei der Variablenselektion verzerrt ist. Das ist ein großer Nachteil auch bei der Interpretation der Ergebnisse. Die Vorhersagegüte ist bei den Beispieldatensätzen ähnlich.

A R-Code

A.1 Das Beispiel USvote

Zur Berechnung des Baumes für den Datensatz USvote wurde folgende Syntax verwendet. Man benötigt das Paket **CHAID** (The FoRt Student Project Team 2009).

```
library("CHAID")
data("USvote")

names(USvote) <- c("Wahl","Geschlecht", "Alter",
                   "Arbeitslos", "Bildung", "Familienstand")
levels(USvote$Geschlecht) <- c("männl.", "weibl.")
levels(USvote$Arbeitslos) <- c("ja", "nein", "in Rente")
levels(USvote$Bildung) <- c("< High School", "High School",
                            "> High School", "College", "Post College")
levels(USvote$Familienstand) <- c("verheiratet", "verwitwet",
                                  "geschieden", "nie verheiratet")
summary(USvote)

ctrl <- chaid_control(minsplit = 3000)
chaidUS <- chaid(Wahl ~ ., data = USvote, control = ctrl)

print(chaidUS)
plot(chaidUS)
```

A.2 CHAID für stetige Zielgrößen

Der R-Code für CHAID basierend auf der Implementierung aus dem Kurs "Fortgeschrittene Programmierung mit R" im Wintersemester 2008/2009. Damit der Algorithmus funktioniert, benötigt man die Pakete **coin** (Hothorn T. et al. 2006 a und Hothorn T.et al. 2008) und **partykit** (Hothorn und Zeileis 2009)).

```
library(coin)          # Package for independence test
library(partykit)      # Package for tree-structure

### check if two objects are identical and print differences else
isequal <- function(a, b) {
    attributes(a) <- NULL
    attributes(b) <- NULL
    if (!isTRUE(all.equal(a, b))) {
        print(a, digits = 10)
        print(b, digits = 10)
        return(FALSE)
    } else {
        return(TRUE)
    }
}

### function calculating the logarithm of p-values of tests
logpval <- function(object) {
    p <- NA
    if (extends(class(object), "QuadTypeIndependenceTest")) {
      if (extends(class(object@distribution), "AsymptNullDistribution")) {
        p <- pchisq(statistic(object),
                    df = object@distribution@parameters$df,
                    log.p = TRUE, lower.tail = FALSE)
        return(p)
      }
      if (extends(class(object@distribution), "ApproxNullDistribution")) {
        return(log(pvalue(object)))
        }
    }
    if (is.na(p)) warning("could not compute log-pvalue")
    return(NA)
}

### calculationg the logpvalue of metric and categorical response
logid_test <- function(x, response) {
```

```
    # check that there are different values in x
    if (nlevels(x[, drop = TRUE]) < 2) return(0)

    # check that there are different values in response
    if (is.factor(response)){
      if (nlevels(response[, drop = TRUE]) < 2) return(0)
    }else{
      if(sum(response == rep(response[1], length(response)))
          == length(response))
        return(0)
    }

    if(length(x) < 100){
      itest <- independence_test(response ~ x,
                  distribution = approximate(B=9999), teststat="quad")
    }else{
      itest <- independence_test(response ~ x,
                  distribution = asymptotic(), teststat="quad")
    }

    # Calculate logarithm of pvalue
    ret <- as.numeric(logpval(itest))

    names(ret) <- "logp"
    stat <- statistic(itest)
    attr(ret, "Chisq") <- as.numeric(stat)

    return (ret)
}


### merge two levels of a factor
mergelevels <- function(index, merge) {

    stopifnot(length(unique(merge)) == 2)
    stopifnot(all(merge %in% index))

    # which original levels are to be merged
    ml <- index %in% merge
    # which levels must be relabeled
    gr <- index > max(merge)
```

```
    # merge levels
    index[ml] <- min(index[ml])
    # relabel
    index[gr] <- index[gr] - 1

    return(index)
}

### split a merger in two groups
splitlevels <- function(index, level, split) {

    stopifnot(sum(index == level) > length(split))

    # which levels must be relabeled
    gr <- index > level
    # relabel
    index[gr] <- index[gr] + 1
    # split
    index[split] <- level + 1

    return(index)
}

### actually merge factor levels
mergex <- function(x, index) {

    # extract levels and save observations as character
    lev <- levels(x)
    if (is.ordered(x))
        stopifnot(all(diff(index) %in% c(0, 1)))
    chrx <- as.character(x)
    newlev <- rep("", length(unique(index)))
    for (i in unique(index)) {
        indx <- index == i
        # assign merged levels to observations
        chrx[chrx %in% lev[indx]] <- paste(lev[indx], collapse = "+")
        # merge levels itself
        newlev[i] <- paste(lev[indx], collapse = "+")
    }
    if (is.ordered(x)) return(ordered(chrx, levels = newlev))
    return(factor(chrx, levels = newlev))
}
```

```
#### functions corresponding to the steps in CHAID

step1internal <- function(response, x, weights, index = NULL, ctrl) {

    alpha2 <- ctrl$alpha2
    alpha3 <- ctrl$alpha3
    stopifnot(alpha2 > alpha3)
    stopifnot(is.factor(x))
    if (is.null(index))
        index <- 1:nlevels(x)

    while(TRUE) {

         ### nothing to do for two categories
         if (max(index) == 2) break()

         ### merge levels
         mlev <- step2(response, x, weights, index, ctrl)

         ### nothing to merge, return index
         if (is.null(mlev)) break()

         ### step 3 necessary?
         runstep3 <- sum(mlev[1] == index) > 1 ||
                     sum(mlev[2] == index) > 1
         runstep3 <- runstep3 && (alpha3 > 0)

         ### actually merge levels
         kati <- index %in% mlev
         index <- mergelevels(index, mlev)
         kat <- unique(index[kati])

         ### perform step 3 if necessary
         if (runstep3)
             index <- step3(response, x, weights, index, alpha3 = alpha3, kat)
    }
    return(index)
}

# indices for optimally merged variables
step1 <- function(response, xvars, weights, indices = NULL, ctrl) {

    ret <- vector(mode = "list", length = length(xvars))
```

```
    for (i in 1:length(xvars))
        ret[[i]] <- step1internal(response, xvars[[i]],
                                  weights, indices[[i]], ctrl)
    ret
}


step2 <- function(response, x, weights, index = 1:nlevels(x), ctrl) {

    stopifnot(is.factor(x))

    x <- mergex(x, index)
    if (nlevels(x[drop = TRUE]) < 3) return(NULL)

    xlev <- levels(x)
    xclass <- class(x)[1]

    x <- rep(x, weights)
    response <- rep(response, weights)

    comb <- switch(xclass,
        "factor" = lapply(1:(nlevels(x) - 1), function(i)(i + 1):nlevels(x)),
        "ordered" = lapply(1:(nlevels(x) - 1), function(i) i + 1),
        stop("unknown class")
    )

    logpmax <- -Inf
    levindx <- c(NA, NA)
    for (i in 1:length(comb)) {         # all combinations for merging
        for (j in comb[[i]]) {

            response1 <- response[which(as.integer(x) %in% c(i,j))]
            x1 <- x[ which(as.integer(x) %in% c(i,j)) ]

            logp <- logid_test(x1, response1)
            if (logp > logpmax) {
                logpmax <- logp             # maximum p value
                levindx <- c(i, j)
            }
        }
     }

     # in case logp-value -Inf, nothing to merge
```

```
    if(is.na(levindx)[1]) return (NULL)

    ### sample size stopping criteria
    nmin <- min(c(ceiling(ctrl$minprob * sum(weights)), ctrl$minbucket))

    if (exp(logpmax) > ctrl$alpha2 ||
        length(x[which(as.integer(x) %in% levindx)]) < nmin ||
        length(x[-(which(as.integer(x) %in% levindx))]) < nmin){
        return(levindx)
        }

    return(NULL)
}

step3 <- function(x, y, weights, alpha3 = 0.049, index, kat) {

    split_indx <- index
    if (sum(index == kat) > 2) {
        sp <- step3intern(x, y, weights, alpha3, index, kat)
        ### compute minimum p-value and split
        if(!is.null(sp))
            split_indx <- splitlevels(index, level = kat, sp$split)
    }
    ## return new index
    return(split_indx)
}

step3intern <- function(x, y, weights, alpha3=0.05, index, kat){

           ### determine all admissible combinations
           foo <- function(nll) {
                   if (is.ordered(x)){

                     ret<-matrix(FALSE,ncol=nll,nrow=nll-1)
                     for(i in 1:(nll-1)) {
                     for(j in 1:(nll-1))  {
                     if(i<=j){
                     ret[j,i]<-TRUE}}}

                    else{
                     indl <- rep(FALSE, nll)
                     indl[1] <- TRUE
```

```
                    mi <- 2^(nll - 1)
                    ret <- matrix(FALSE, ncol = nll, nrow = mi - 1)

                    for (i in 1:(mi - 1)) {
                        ii <- i
                        for(l in 1:(nll-1)) {
                            indl[l] <- as.logical(ii%%2)
                            ii <- ii %/% 2
                        }
                       ret[i,] <- indl
                } }
                return(ret)
            }

            subsetx <- x %in% levels(x)[index == kat]
            ytmp <- y[subsetx, drop = TRUE]
            xtmp <- x[subsetx, drop = TRUE]
            wtmp <- weights[subsetx]
            xlev <- levels(xtmp)
            ret <- foo(nlevels(xtmp))

            logp <- numeric(nrow(ret))

            for (i in 1:nrow(ret)){

                tmpx <- as.factor(xtmp %in% xlev[ret[i, ]])

                matx <- rep(tmpx, wtmp)
                matresponse <- rep(ytmp, wtmp)
                logp[i] <- logid_test(matx, matresponse)
            }

            logp_min <- min(logp)
            if (exp(logp_min) > alpha3) return(NULL)

            splitlev <- xlev[ret[which.min(logp),]]
            return(list(logp = logp_min,
                        split = which(levels(x) %in% splitlev)))

}

step4internal <- function(response, x, weights, index) {
```

```
    mx <- mergex(x, index)
    c_levels <- nlevels(x[weights > 0, drop = TRUE])
    r_levels <- nlevels(mx)

    x1 <- rep(mx, weights)
    response1 <- rep(response, weights)

    ### p-value on log-scale!
    logp <- logid_test(x1, response1)

    ### for log-p-vlaue = 0 no adjustment necessary
    if(logp == 0) return (0)

    if (is.ordered(x)) {
        ### formula (3.1) in Kass (1980)
        ret <- logp + lchoose(c_levels - 1,r_levels - 1)
    } else {
        i <- 0:(r_levels - 1) ### formula (3.2)
        fact <- sum((-1)^i * ((r_levels - i)^c_levels) /
                              (factorial(i) * factorial(r_levels - i)))
        ret <- logp + log(fact)
    }
    attr(logp, "Chisq") <- attr(logp, "Chisq")
    return(ret)
}

step4 <- function(response, xvars, weights, indices) {

    p <- numeric(length(xvars))
    X2 <- rep(NA, length(xvars))
    for (i in 1:length(xvars)) {
        tmp <- step4internal(response, xvars[[i]], weights, indices[[i]])
        p[i] <- tmp
        if (!is.null(attr(tmp, "Chisq")))
            X2[i] <- attr(tmp, "Chisq")
    }
    names(p) <- names(xvars)
    attr(p, "Chisq") <- X2

    return(p)
}

step5 <- function(id = 1L, response, x, weights = NULL, indices = NULL,
```

```
              ctrl = chaid_control()) {

if (is.null(weights)) weights <- rep.int(1, length(response))

# stopping criteria if too few observations in node
if (sum(weights) < ctrl$minsplit){
    return(partynode(id = id))
}

# stopping criteria for root-split
if (ctrl$stump && id > 1){
    return(partynode(id = id))}

indices <- step1(response, x, weights, indices = indices, ctrl)

logpvals <- step4(response, x, weights, indices)
info <- list(adjpvals = exp(logpvals))

if (exp(min(logpvals)) > ctrl$alpha4) {
    return(partynode(id = id, info = info))
}

sp <- partysplit(varid = which.min(logpvals),
                 index = as.integer(indices[[which.min(logpvals)]]))

### FIXME: remove???
### retain index only for split variable
newindices <- lapply(x, function(x) 1:nlevels(x))
### newindices[[which.min(logpvals)]] <- indices[[which.min(logpvals)]]

kidids <- kidids_split(sp, data = x)

kids <- vector(mode = "list", length = max(sp$index))
for (kidid in 1:max(sp$index)) {
      w <- weights
      w[kidids != kidid] <- 0
      if (kidid > 1) {
          myid <- max(nodeids(kids[[kidid - 1]]))
      } else {
          myid <- id
      }

      kids[[kidid]] <- step5(id = as.integer(myid + 1), response, x,
```

```
                                    weights = w, newindices, ctrl)

    }

    return(partynode(id = as.integer(id), split = sp, kids = kids, info = info))
}

chaid_control <- function(alpha2 = 0.05, alpha3 = -1, alpha4 = 0.05,
                          minsplit = 20, minbucket = 7, minprob = 0.01,
                          stump = FALSE) {

    ret <- list(alpha2 = alpha2, alpha3 = alpha3, alpha4 = alpha4,
         minsplit = minsplit, minbucket = minbucket, minprob = minprob,
         stump = stump)
    class(ret) <- "chaid_control"
    return(ret)
}


chaid <- function(formula, data, subset, weights, na.action = na.omit,
                  control = chaid_control())
{
      mf <- match.call(expand.dots = FALSE)
      m <- match(c("formula", "data", "subset", "weights", "na.action"),
               names(mf), 0)
      mf <- mf[c(1, m)]
      mf$drop.unused.levels <- FALSE
      mf[[1]] <- as.name("model.frame")
      m <- eval.parent(mf)
      y <- model.response(m)
      x <- m[, c(-1, -which(names(m) == "(weights)")), drop = FALSE]
      w <- model.weights(m)
      chaidtree <- step5(1L, y, x, weights = w, ctrl = control)
      tree <- party(chaidtree, data = x,
                fitted = data.frame("(fitted)" = fitted_node(chaidtree, data = x)
                                    "(response)" = y, check.names = FALSE),
                    terms = terms(formula, data = data))
      if (!missing(weights))
          tree$fitted[["(weights)"]] <- w
      class(tree) <- c("constparty", "party")
      tree
}
```

A.3 Simulation

Dieser Abschnitt enthält den R-Code für die Simulation der Unverzerrtheit und Power von CHAID und CTREE. Man benötigt für CHAID die Pakete **coin** und **partykit** (siehe oben Anhang A.2).
CTREE wird mit dem Paket **party** (Hothorn et al. 2006 b und Hothorn et al. 2009) berechnet. Der Funktionsaufruf für CTREE erfolgt mit der Funktion ctree():

```
ctree(formula, data,
      controls = ctree_control(mincriterion = 1, stump = TRUE))
```

Wobei formula die Ziel- und Einflussgrößen spezifiziert und data den Datensatz angibt. Mit den Übergabeparametern in der Funktion ctree_control() erzwingt man, dass genau ein Root-Split durchgeführt wird.

sim_bin.R

Die Datei sim_bin.R enthält den Simulationscode, der für die verschiedenen Werte des Mittelwerts von y jeweils aufgerufen wird. Die Funktion foo(n,p) generiert einen Datensatz mit n Beobachtungen und mit dem Mittelwert p von y für V7 gleich 1. Die Funktion loop(Nsim,p) berechnet Nsim mal einen Root-Split in einem Datensatz, der mittels der Funktion foo() erstellt wird. Der Parameter p bezeichnet wieder den Mittelwert von y für V7 gleich eins.

```
# Simulation y~N(p,vari)

library("party")            # Paket zur Berechnung von CTREE
source("chaidmetrisch.R")   # CHAID für metrische Zielgrößen

Nsim <- 5000    # Anzahl der Simulationsdurchläufe

vari <- 1          # Varianz von y

set.seed(290875)

# Generiert Datensatz mit n Beobachtungen
# und y ~ N(0, vari) für X7 = 1 und y ~ N(p, vari) für X7 = 1

foo <- function(n, p = 1) {
    df <- as.data.frame(matrix(0, nrow = n, ncol = 7))

    df[1] <- sample(gl(4,n,ordered=TRUE))[1:n]
    df[2] <- sample(gl(6,n,ordered=TRUE))[1:n]
    df[3] <- sample(gl(8,n,ordered=TRUE))[1:n]
    df[4] <- sample(gl(4,n))[1:n]
```

```
    df[5] <- sample(gl(6,n))[1:n]
    df[6] <- sample(gl(8,n))[1:n]

    df$V7 <- sample(gl(2, n/2))
    df$y <- rep(0, n)

    df$y[df$V7 == "1"] <- rnorm(n/2, 0, vari)
    df$y[df$V7 == "2"] <- rnorm(n/2, p, vari)

    df
}

# Generiert Tabelle mit gewählter Split-Variable
# und deren p-Wert für CHAID und CTREE

loop <- function(Nsim, p = 1) {

    sctree <- vector(mode = "character", length = Nsim)
    pvalue_ctree <- vector(mode = "numeric", length = Nsim)
    schaid <- vector(mode = "character", length = Nsim)
    pvalue_chaid <- vector(mode = "numeric", length = Nsim)

    for (i in 1:Nsim) {

        print(i)
        df <- foo(100, p = p)

        ctrl <- party::ctree_control(stump = TRUE, mincriterion = 0)
        mod <- party::ctree(y ~ ., data = df, controls = ctrl)
        sctree[i] <- names(df)[which.max(mod@tree[[3]][[2]])]
        pvalue_ctree[i] <- (1 - mod@tree[[3]][[3]])

        mod <- chaid(y ~ ., data = df,
                     control = chaid_control(alpha4 = 1, stump = TRUE))
        schaid[i] <- names(node_party(mod)$split$varid)
        pvalue_chaid[i] <- min(node_party(mod)$info$adjpvals)
    }

    ret <- data.frame(sctree = factor(sctree,
                                      levels = paste("V", 1:7, sep = "")),
                      pvalue_ctree = pvalue_ctree,
                      schaid = factor(schaid,
                                      levels = paste("V", 1:7, sep = "")),
```

```
                    pvalue_chaid = pvalue_chaid
                    )
    attr(ret, "p") <- p
    return(ret)
}
```

run_0.R

Als Beispiel für eine Simulation der Code für $p = 0$. Mit p wird in der Syntax der Parameter μ bezeichnet, der den Mittelwert von Y gegeben die Variable V7 festlegt. Die Ergebnisse der Simulation werden in der Datei "ret_0.rda" gespeichert.

```
p <-  0

source("sim_bin.R")

ret <- loop(Nsim, p)
attach(ret)

table(sctree) / Nsim
mean(pvalue_ctree < 0.05)
mean(sctree == "V7" & pvalue_ctree < 0.05)

table(schaid) / Nsim
mean(pvalue_chaid < 0.05)
mean(schaid == "V7" & pvalue_chaid < 0.05)

save(ret, file = paste("ret_", p, ".rda", sep = ""))
```

goodman.R

Die Funktion goodman(freq, alpha = 0.05) dient der Berechnung der Konfidenzintervalle nach Goodman (Goodman 1965).

```
goodman <- function(freq, alpha = 0.05){
k <- length(freq)
int <- matrix(nrow = k, ncol = 2)
n <- sum(freq)
B <- qchisq(p = 1 - alpha / k, df = 1)
for(i in 1:k){
a <- (1 + B / n)
b <- (- 2 * as.numeric(freq[i]) - B) / n
c <- (as.numeric(freq[i]) / n)^2
```

```
int[i,1] <- ((-b) - sqrt(b^2 - 4 * a * c)) / (2 * a)
int[i,2] <- ((-b) + sqrt(b^2 - 4 * a * c)) / (2 * a)
}
return(int)
}
```

summary.R

Der R-Code in summary.R dient dazu die Ergebnisse der Simulation für alle Werte von p, hier $p \in \{0, 0.1, 0.2, \ldots, 1\}$ zusammenzufassen.

```
source("goodman.R")

p <- seq(from = 0, to = 1, by = 0.1)
try(rm(ret))
Nsim <- 5000
out <- c()
fctree <- c()
fchaid <- c()

for (m in p) {
    load(paste("ret_", m, ".rda", sep =""))
    out <- rbind(out,c(mean(ret$pvalue_ctree < 0.05),
            if(sum(ret$pvalue_ctree < 0.05) == 0) {0}
      else{sum(ret$sctree == "V7" &
               ret$pvalue_ctree < 0.05) /
                  sum(ret$pvalue_ctree < 0.05)},
    mean(ret$pvalue_chaid < 0.05),
    if(sum(ret$pvalue_chaid < 0.05) == 0) {0}
      else{sum(ret$schaid == "V7"
               & ret$pvalue_chaid < 0.05) /
                 sum(ret$pvalue_chaid < 0.05)}
   ))
    fctree <- rbind(fctree, table(ret$sctree) / Nsim)
    fchaid <- rbind(fchaid, table(ret$schaid) / Nsim)
    rm(ret)
}

rownames(out) <- p
colnames(out) <- c("split_ctree", "correct_ctree",
                   "split_chaid", "correct_chaid")
out <- as.data.frame(out)
```

```
pdf("power_split.pdf", width = 8, height = 5)
cex <- 1.15

layout(matrix(1:2, nc = 2))
par(mar=c(5, 6, 4, 2) + 0.1)
plot(p, out[["split_ctree"]], type = "l", lty = 1, ylim = c(0,1),
     ylab = "Simulierte Power", xlab = expression(mu), cex.axis = cex,
     cex.lab = cex)
lines(p, out[["split_chaid"]], lty = 2)
abline(h = 0.05, lty = 3)
legend(0.45, 0.25, lty = c(1, 2), legend = c("CTREE", "CHAID"),
       cex = 1, bty = "n")

plot(p, out[["correct_ctree"]], type = "l", lty = 1, ylim = c(0,1),
     ylab = "Bedingte Wahrscheinlichkeit\n für den korrekten Split",
     xlab = expression(mu), cex.axis = cex, cex.lab = cex)
lines(p, out[["correct_chaid"]], lty = 2)
legend(0.45, 0.25, lty = c(1, 2), legend = c("CTREE", "CHAID"),
       cex = 1, bty = "n")

dev.off()

print(out)
print(fctree)
print(fchaid)

c1 <- round(goodman(fctree[1,] * Nsim), 3) ## für p = 0
c2 <- round(goodman(fchaid[1,] * Nsim), 3)

selprob <- cbind(fchaid[1,], c2[,1], c2[,2], fctree[1,], c1[,1], c1[,2])
```

A.4 Vorhersagegüte

Dieser Abschnitt enthält den R-Code zur Prüfung der Vorhersagegüte von CHAID im Vergleich zu CTREE anhand der Beispieldatensätze BostonHousing, Ozone und Servo.

data.R

Code zur Aufbereitung der Datensäzte. Die Zielgröße erhält den Namen y. Metrische Kovariablen werden kategorisiert. Abgedruckt ist die Aufbereitung von BostonHousing. Servo und Ozone wurden analog bearbeitet.

```
library("mlbench")

data(BostonHousing)

names(BostonHousing)[14] <- "y"
any(is.na(BostonHousing))
summary(BostonHousing)

## alle Einflussgroessen kategorisieren (außer Var4. sie ist schon Faktor)
for(i in c(1:3,5:13)) {
BostonHousing[[i]] <- cut(BostonHousing[[i]],
                       breaks =   unique(quantile(BostonHousing[[i]],
                                          probs = seq(0,1,0.1))),
                             ordered_result = TRUE, include.lowest = TRUE)
}

summary(BostonHousing)

save(BostonHousing, file = "data/BostonHousing.rda")
```

cmp.R

Die Funktion plrpbench(dat,B) zieht $B = 100$ Bootstrap-Datensätze aus dem Datensatz dat und berechnet jeweils einen Baum mittels CHAID und CTREE. Dann wird für den out-of-bag Testdatensatz der mittlere quadrierte Fehler berechnet.

```
library("party")
source("chaidmetrisch.R")

B <- 100
set.seed(290875)
```

```
plrpbench <- function(dat, B = 100) {

    dat <- dat[complete.cases(dat),]
    ## complete cases, da CHAID Beobachtungen mit NAs sowieso entfernt

    n <- nrow(dat)
    bs = rmultinom(B, n, rep.int(1, n)/n) ## Bootstrap

    perf <- matrix(0, ncol = 2, nrow = B)
    ptree <- vector(length = B, mode = "list")
    chtree <- vector(length = B, mode = "list")

    for (i in 1:B) {              # Schleife über Bootstrap-Samples
        print(i)
        oob = (bs[,i] == 0)       # Erzeugt Testdatensatz

        ## CTREE
        pt <- party::ctree(y ~ ., data = dat, weight = bs[,i])
        pr <- predict(pt, newdata = dat)
        ptree[[i]] <- pt
        perf[i,2] <- mean((pr[oob]-dat$y[oob])^2)

        ## CHAID
        cht <- chaid(y ~ ., data = dat, weights = bs[,i])
        chtree[[i]] <- cht
        chr <- predict(cht, newdata = dat)
        perf[i,1] <- mean((chr[oob] - dat$y[oob])^2)

        cat(perf[i,1], " ", perf[i,2], " ", "\n");
    }
    list(perf = perf, bs = bs)
}
```

rundat_BostonHousing.R

Als Beispiel für den Simulationsaufruf der Code für den Datensatz BostonHousing.

```
try(system("rm perf/perfBostonHousing.rda"))

source("cmp.R")
load("data/BostonHousing.rda")

perf <- plrpbench(BostonHousing, B = B)
```

```
apply(perf$perf, 2, summary)
perf$name = "BostonHousing"

save(perf, file = "perf/perfBostonHousing.rda")
```

summary.R

Zur Zusammenfassung der Ergebnisse aus den drei Datensätzen. Es werden Boxplots der quadrierten Fehler erstellt und der Kruskal-Wallis-Test wird berechnet.

```
sets <- c("BostonHousing", "Ozone", "Servo")

try(system("rm pdf/*.pdf"))

# Boxplots

for (m in sets) {
load(paste("perf/perf", m, ".rda", sep = ""))
pdf(file = paste("pdf/bp_", m, ".pdf", sep = ""))
boxplot(list(perf$perf[,1],perf$perf[,2]),
         names=c("CHAID","CTREE"), main = paste(m))
dev.off()
}

# KW-Test

total <- data.frame()

for (m in sets) {
    load(paste("perf/perf", m, ".rda", sep = ""))

    ch <- data.frame(rep(paste(m), length(perf$perf[,1])),
                     rep("chaid", length(perf$perf[,1])),
                     perf$perf[,1])
    ct <- data.frame(rep(paste(m), length(perf$perf[,2])),
                     rep("ctree", length(perf$perf[,2])),
                     perf$perf[,2])

    names(ch) <- c("dataset","algorithm","error")
    names(ct) <- c("dataset","algorithm","error")

    verf <- rbind(ch,ct)
```

```
    ## Gesamtdatensatz erstellen
    total <- rbind(total,verf)
}

kwresult <- data.frame()

for (m in sets) {
    kwt <- kruskal.test(total$error[total$dataset == paste(m)]
                        ~ total$algorithm[total$dataset == paste(m)])

    ## Ergebnisse in Tabelle schreiben
    kwres <- data.frame(paste(m), kwt$statistic ,kwt$p.value)
    names(kwres) <- c("dataset", "statistic", "p.value" )
    kwresult <- rbind(kwresult, kwres)
}
rownames(kwresult) <- NULL
```

B Beigefügte CD

Die beigefügte CD enthält allen R-Code, der für die Bachelor-Arbeit verwendet wurde. Die Ordner enthalten folgede Inhalte:

- *CHAID:* das Package CHAID, wie es im Wintersemester 2008/2009 im Kurs FoRt bei Prof. Dr. Hothorn programmiert wurde

- *CHAIDmetrisch:* die neue Implementation von CHAID mit Komponententests

- *VerzerrungPower:* den R-Code und die R-Outputs zur Simulation der Verzerrung und Power

- *Vorhersage:* die Datensätze, den R-Code und die R-Outputs zur Bestimmung der Vorhersagegüte

Literatur

[1] Goodman L. A.(1965): *On simultaneous confidence intervals for multinomial proportions.* Technometrics, Mai 1956, Vol. 7, No. 2, S. 247-254 Hastie T., Tibshirani R. und Friedman J. (2008): *The elements of statistical learning : data mining, inference and prediction.* Springer Series in Statistics. Springer New York, 2 edition.

[2] Hothorn T., van de Wiel M. A., Zeileis A. (2006 a): *A Lego System for Conditional Inference.* The American Statistician, August 2006, Vol. 60, No. 3, S. 257-263 Hothorn T., Hornik K., und Zeileis A.(2006 b): *Unbiased recursive partitioning: A conditional inference framework.* Journal of Computational and Graphical Statistics, Vol. 15, No. 3, S. 651-674.

[3] Hothorn T., Hornik K., van de Wiel M. A., Zeileis A. (2008): *Implementing a Class of Permutation Tests: The coin Package.* Journal of Statistical Software, November 2008, Volume 28, Isssue 8.

[4] Hothorn T. und Zeileis A. (2009): *partykit: A Toolkit for Recursive Partytioning.* R Paket Version 0.0-1.

[5] Hothorn T., Hornik K., Strobl C. und Zeileis A (2009): *party: A Laboratory for Recursive Partytioning.* R package version 0.9-995.

[6] Kass G. V. (1980): *An Exploratory Technique for Investigating Large Quantities of Categorical Data.* Applied Statistics, 29(2), S. 119-127.

[7] Leisch F. und Dimitriadou E. (2009): *mlbench: Machine Learning Benchmark Problems.* R package version 1.1-6.

[8] Strasser, H., Weber, C. (1999): *On the Asymptotic Theory of Permutation Statistics.* Mathematical Methods of Statistics, 8, S. 220-250.

[9] The FoRt Student Project Team (2009): *CHAID: CHi-squared Automated Interaction Detection.* R Paket Version 0.1-0.

[10] Thiemichen S. (2009): *Unverzerrtes rekursives Partitionieren: Ein empirischer Vergleich von CHAID und CTREE.* Bachelor-Arbeit an der Ludwig-Maximilians-Universität München, Institut für Statistik, betreut von Prof. Dr. Torsten Hothorn, unveröffentlicht.

[11] Toutenbourg H. (2003): *Lineare Modelle. Theorie und Anwendungen.* Physica-Verlag. Heidelberg. 2. Auflage.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelor-Arbeit selbstständig verfasst habe und, dass ich ausschließlich die angegebenen Quellen und Hilfsmittel verwendet habe.

Freising, den 7. August 2009

Sarah Maierhofer