LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Studienabschlussarbeiten
Fakultät für Mathematik, Informatik und Statistik

Müntefering, Eva-Maria:

Boosted Random Forest

Masterarbeit, Wintersemester 2016

Fakultät für Mathematik, Informatik und Statistik

Ludwig-Maximilians-Universität München

https://doi.org/10.5282/ubm/epub.38412

Boosted Random Forest

Masterarbeit

Eva-Maria Müntefering

Betreuer:

Prof. Dr. Anne - Laure Boulesteix, IBE München
Philipp Probst, MSc Statistik, IBE München

Institut für Statistik

Ludwig - Maximilians - Universität München

München, 19. Dezember 2016

Kurzzusammenfassung

Ziel der Arbeit ist es zu untersuchen, ob durch eine Gewichtung der Beobachtungen eines Trainingsdatensatzes eine Verringerung des Testfehlers bei Anwendung des Random–Forest–Algorithmus erreicht werden kann und wie die jeweilige Wirkungsweise der verschiedenen Gewichtungen ist. Dazu wurde eine Funktion in `R` implementiert, welche eine Anwendung der Gewichte sowohl als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe als auch für die Berechnung des *majority votes* bereitstellt. Weiter wurde eine, an die neue Funktion angepasste `predict`-Funktion implementiert. Die Berechnung der Gewichte wurde dabei von Bernard et al. (2012) und von Mishina et al. (2014) übernommen. Bei einem Vergleich der Testfehlerraten der unterschiedlichen Varianten des Algorithmus untereinander und mit denen des Random Forest nach Breiman (2001) zeigen sich keine Regelmäßigkeiten was die Überlegenheit einer Variante betrifft. Die Simulationsstudie zur Evaluierung der Verhaltensweisen der Varianten bei Daten mit Ausreißern und zunehmender Kategorienzahl im Response legt nahe, dass insbesondere eine Anwendung der Gewichte als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe bei Daten mit einem hohen Ausreißeranteil nicht empfehlenswert ist. Generell sind die Unterschiede in den Testfehlerraten der verschiedenen Varianten nur minimal, sodass weder eine deutliche Verbesserung noch eine Verschlechterung der Performance des Algorithmus durch die hier verwendeten Arten der Gewichtung festgestellt werden kann und somit weiter die bereits implementierten und schnellen Funktionen des Standard-Random-Forest vorzuziehen sind.

Inhaltsverzeichnis

1 Einleitung

Mit den immer größer werdenden Datenmengen, auch *Big Data* genannt, steigt auch die Notwendigkeit an Methoden, um mit diesen umzugehen. Häufige Anwendung findet das *statistische Lernen* daher unter anderem sowohl in vielen wissenschaftlichen Bereichen als auch im Finanz- und Marketingsektor. Ein häufiges Ziel ist es, in den Daten einen Zusammenhang zwischen Prädiktor- und Responsevariablen zu finden. [vgl. James et al. (2013)] Ein konkreter Anwendungsfall wäre hier ein Kreditunternehmen. Dieses möchte bestimmen, ob seine Kunden den Kredit zurückzahlen ($Y = 1$) oder nicht ($Y = 0$). Hierzu liegen dem Unternehmen Daten über den Kunden vor, wie etwa das Alter, das Geschlecht, das Einkommen, ..., anhand welcher dem Kunden eine Klasse (nämlich $Y = 1$ oder $Y = 0$) zugeordnet werden soll. Um das Modell zu trainieren, kann auf Daten aus der Kundendatenbank zurückgegriffen werden, zu welchen unter Umständen bereits die Beobachtung zur Kreditrückzahlung vorliegt.

Die zwei größten Untergebiete des statistischen Lernens sind das sogenannte *überwachte Lernen (supervised learning)* und das *unüberwachte Lernen (unsupervised learning)*. Beim überwachten Lernen liegt zu jedem Beobachtungsvektor der gemessenen Merkmale $\boldsymbol{x}_i$ des Trainingsdatensatzes auch eine entsprechende Beobachtung des Response y_i vor. Ziel ist es, die Beziehung zwischen den Prädiktorvariablen und der Zielvariablen möglichst genau zu modellieren, den Zusammenhang somit besser zu verstehen und weiter eine Vorhersage des Response neuer Beobachtungen zu ermöglichen. Beim unüberwachten Lernen hingegen gibt es diese Beobachtung des Response nicht. Es ist somit nicht möglich, ein Modell im Sinne des überwachten Lernens anzupassen, da das Ergebnis aufgrund der fehlenden Beobachtung von y_i nicht überprüft werden kann. Mögliche Herangehensweisen an solche Problemstellungen ist die Anwendung von Clustermethoden. [vgl. James et al. (2013), S.26 f.]

Neben linearen und logistischen Modellen, welche beim überwachten statistischen Lernen zum Einsatz kommen, ist auch der *Random Forest* (RF) eine beliebte Methode für Überlebensdaueranalysen, Regressions- und Klassifikationsprobleme. Nach seiner Einführung durch Breiman (2001), gab es viele erfolgreiche Versuche, diesen

Algorithmus weiter auszubauen und zu verbessern. Durch Kombinationen mit dem *Boosting*-Algorithmus ähnelnden Ideen, soll vor allem die Vorhersagegenauigkeit optimiert und die Anzahl der Bäume minimiert werden. Dies geschieht vor allem durch die Anwendung von Gewichten. Während die einzelnen Entscheidungsbäume in einem Random Forest unabhängig voneinander sind, soll diese Unabhängigkeit durch die Bestimmung eines Gewichts für jede Beobachtungen auf Grundlage aller bereits angepassten Bäume reduziert werden. Auf diese Weise soll vermieden werden, dass Bäume in dem Wald vorkommen, welche nichts oder nur wenig zu einer besseren Performance des gesamten Waldes beitragen [vgl. Bernard et al. (2012)].

Konkrete Veränderungen des ursprünglichen Algorithmus sind zum Beispiel durch Mishina et al. (2014) anhand einer Gewichtung sowohl der einzelnen Bäume als auch der Beobachtungen getroffen worden. Bernard et al. (2012) hingegen gewichtet mit seinem *Dynamic Random Forest (DRF)*-Algorithmus die einzelnen Beobachtungen anhand der sogenannten *Out-of-Bag*-Fehlerrate der jeweiligen Beobachtung gemessen an *allen* vorherigen Bäumen. Diese Gewichte können wiederum Einfluss auf die Wahl des Splitwertes einer Variablen (Wert, welcher das zuvor festgelegte Splitkriterium optimiert) oder, bei Klassifikation, auf die Berechnung des *majority votes* (Klasse, welcher die Beobachtung i am häufigsten zugeordnet wurde) in den Endknoten nehmen. Weitere mögliche Veränderungen des Algorithmus sind eine stärkere Randomisierung beim Baumwachstum, eine zufällige Cut-Point-Wahl oder die *Rotation-Forest-Methode*, welche mit der Hauptkomponentenanalyse arbeitet.

Ziel dieser Arbeit ist es zum einen, einen Überblick über solche, Performance verbessernde Möglichkeiten zu geben, und zum anderen den konkreten Einfluss der Gewichte beim *majority vote* sowie bei Ziehung der Bootstrap-Stichprobe zu untersuchen. Eine Kombination beider Möglichkeiten soll ebenso betrachtet werden wie auch die Gewichtung der einzelnen Bäume. Dazu wird eine Funktion in `R` implementiert, welche entsprechende Einstellungen zulässt und diese dann mit dem Standard Random Forest nach Breiman (2001) verglichen. Zusätzlich wird eine `predict()`-Funktion implementiert, mit welcher der „neue“ Random Forest auf neuen, unbekannten Daten evaluiert wird und sein Testfehler mit dem des Standard-RF (nach Breiman (2001)) verglichen werden kann. Anders als der Standard-

RF, welcher sowohl bei Regressions- als auch Klassifikationsproblemen zum Einsatz kommen kann, wird diese neue zu testende Funktion auf letztere beschränkt sein.

Im weiteren Verlauf gibt Kapitel 2 einen einleitenden, theoretischen Überblick über die drei Grundbausteine der Arbeit: *Regressions-/Klassifikationsbäume*, *Random Forest* und *Boosting*. Hierbei werden die grundlegenden Eigenschaften und Vorgehensweisen erläutert, jedoch auf eine Vertiefung der für diese Arbeit irrelevanten Möglichkeiten der jeweiligen Algorithmen verzichtet. Diese sind in einschlägiger Literatur wie zum Beispiel Hastie et al. (2009) oder auch Tutz (2012) nachzulesen. In Kapitel 3 und Abschnitt 3.2 werden dann zwei der bereits umgesetzten Möglichkeiten zur Verbesserung des RF-Algorithmus beschrieben. Dies ist zum einen der *Boosted Random Forest* nach Mishina et al. (2014) und zum anderen der *Dynamic Random Forest* nach Bernard et al. (2012). Weiter wird in Abschnitt 3.3 die neue, in `R` implementierte Funktion beschrieben und in Abschnitt 4.1 zunächst an einigen Datensätzen aus dem `R`-Paket `OpenML` angewandt. Um eventuelle unerwartete Eigenschaften des Algorithmus besser evaluieren zu könne, wird in Abschnitt 4.2 eine kurze Simulationsstudie durchgeführt. Nachdem in Unterabschnitt 4.2.2 die Ergebnisse der Simulationen evaluiert worden sind, schließt die Arbeit mit einer Diskussion, welche neben einer kritischen Zusammenfassung der Arbeit auch ein Fazit und einen Ausblick umfasst, in Kapitel 5 ab.

Alle in dieser Arbeit vorgestellten Analysen wurden mit der Statistik-Software `R` (Version 3.3.1) und `R`-Studio (Version 1.0.44) durchgeführt.

2 Statistische Methodik

Das folgende Kapitel gibt einen Überblick über die zugrunde liegende statistische Methodik. Der RF-Algorithmus gehört zu den baumbasierten Methoden innerhalb derer Algorithmen Entscheidungsbäume nach unterschiedlicher Vorgehensweise erstellt werden. Nach einer kurzen Einführung in die Thematik der *baumbasierten Methoden* in Abschnitt 2.1, werden der *Random Forest*-Algorithmus in Abschnitt 2.2 und mögliche *Boosting*-Algorithmen in Abschnitt 2.3 vorgestellt.

2.1 CART - Classification and Regression Trees

Die Stärken dieser Klassifikations- und Regressionsbäume sind unter anderem ihre leichte Interpretation, ihre Nützlichkeit als Werkzeug zur explorativen Datenanalyse, die Anwendungsmöglichkeit auf sowohl numerische als auch kategoriale Zielgrößen und die Tatsache, dass sie nicht-parametrisch sind. Letzteres setzt keine Kenntnis über die Verteilung der Daten voraus. Bezüglich der explorativen Analyse sind Entscheidungsbäume nützlich, um signifikante Variablen zu identifizieren und Beziehungen zwischen diesen aufzudecken. Beim Fit eines solchen Baumes werden, ausgehend von der Wurzel des Baumes, in welcher sich alle Beobachtungen befinden, alle diese Beobachtungsmengen anhand verschiedener *Kriterien* in Teilmengen aufgesplittet, wodurch eine Vielzahl an Knoten entsteht. [vgl. Tutz (2012)]

2.1.1 Regressionsbäume

Bei *Regressionsbäumen* liegen die Daten für Beobachtung i in der Form $(\boldsymbol{x}_i, y_i)$ vor. Hier gilt $i = 1, \ldots, N$ mit N als Anzahl der Beobachtungen und $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$ mit p als Anzahl der Variablen. Bei einer Unterteilung des Merkmalraumes in M

Regionen $R_1, \ldots, R_M$ wird die Zielgröße für jede Region als Konstante c_m modelliert:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m). \tag{2.1}$$

Diese Konstante wird mit Hilfe eines zu minimierenden Kriteriums geschätzt. Wählt man hierzu die Quadratsumme $\sum(y_i - f(\boldsymbol{x}_i))^2$, so ergibt sich der Mittelwert von y_i in der Region R_m

$$\hat{c}_m = \frac{1}{|R_m|} \sum_{i,\, x_i \in R_m} y_i,$$

als bester Wert für die Konstante c_m, welcher eben genau die Quadratsumme minimiert. Um nun den am besten geeigneten Wert s (wieder bzgl. des zu minimierenden Kriteriums) der Splitvariable j zu wählen, werden die zwei Halbebenen

$$R_1(j,s) = \{X | X_j \leq s\} \text{ und } R_2(j,s) = \{X | X_j > s\}$$

definiert. Gesucht sind nun die entsprechenden Werte für j und s, so dass

$$\min_{j,\, s} \left[\min_{c_1} \sum_{x_i \in R_1(j,\, s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,\, s)} (y_i - c_2)^2 \right]$$

erfüllt wird. Für jedes j und jedes s wird das innere Minimum erreicht durch

$$\hat{c}_1 = \frac{1}{|R_1(j,\, s)|} \sum i,\, x_i \in R_1(j,\, s) y_i \text{ und } \hat{c}_2 = \frac{1}{|R_2(j,\, s)|} \sum i,\, x_i \in R_2(j,\, s) y_i.$$

Somit wird für jede Region das Paar $(j,\ s)$ gefunden, welches das gewählte Kriterium (hier die Quadratsumme) minimiert. Auf die neu entstandenen Regionen wird das gleiche Verfahren rekursiv angewandt, wodurch immer kleinere Regionen entstehen. [vgl. Hastie et al. (2009), S.307]

Dies wird so lange fortgeführt, bis ein zuvor bestimmtes Abbruchkriterium erreicht wird. Eine Möglichkeit für ein solches Abbruchkriterium wäre, den Split nur durchzuführen, falls die Abnahme der Quadratsumme durch den Split eine bestimmte Schwelle überschreitet. Hierbei kann es jedoch passieren, dass der Algorithmus eventuell spätere Splits, welche einen großen Einfluss auf den späteren Baum haben

könnten, nicht mehr durchführt. Eine mögliche Vorgehensweise ist es daher, einen Baum so lange wachsen zu lassen, bis ein Minimum an Beobachtungen innerhalb eines Knotens erreicht ist. Der so entstandene große Baum T_0 wird mit Hilfe des sogenannten *Kosten-Komplexitäts-Prunings* zurückgeschnitten. Diese Art des Prunings zählt zu den *Bottom-Up-Prunings*, da sich hier, beginnend in den Endknoten eines Baumes, von unten nach oben gearbeitet wird. Sei dazu $T \subset T_0$ ein Unterbaum, welcher durch das Pruning entstehen kann. Die Endknoten haben die Indizes m, wobei Knoten m die jeweilige Region R_m repräsentiert. Weiter sei $|T|$ die Anzahl der Endknoten des Baumes T,

$$N_m = \#\{x_i \in R_m\},$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i \text{ und}$$

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2.$$

Das Kosten-Komplexitäts-Kriterium wird definiert als

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|.$$

N_m steht für die Anzahl der Beobachtungen in der Region R_m, $\hat{c}_m$ ist die geschätzte Konstante, $Q_m(T)$ ist der quadratische Fehler, welcher die Unreinheit des Knotens misst und $\alpha > 0$ bezeichnet den Tuningparameter, welcher die Balance zwischen Baumgröße und seiner Güte des Fits regelt. Je größer dabei α, desto kleiner wird der Baum. Der Wert $\alpha = 0$ entspricht somit dem vollen Baum T_0. Es soll für jedes α der Unterbaum $T_\alpha \subset T_0$ gefunden werden, welcher $C_\alpha(T)$ minimiert. Wie jedoch soll α bestimmt werden? Dazu wird fünf- oder zehnfache Kreuzvalidierung angewandt. Hierbei wird der Wert $\hat{\alpha}$ so gewählt, dass die kreuzvalidierte Quadratsumme minimiert wird. Der finale Baum ist somit $T_{\hat{\alpha}}$. [vgl. Hastie et al. (2009), S.308]

2.1.2 Klassifikationsbäume

Im Gegensatz zur Regression nimmt die Zielgröße bei der Klassifikation die Werte $1, \ldots, k$ an, welche für die jeweilige Kategorie stehen. Aufgrund der kategorialen Zielgröße ergeben sich Änderungen in der Vorgehensweise beim Splitting und Pruning.

Es gibt nach Tutz (2012) zwei Kriterien, nach welchen diese Splits durchgeführt werden können. Dies sind zum einen Teststatistiken. Sei dazu A ein Knoten, welcher in zwei Teilmengen A_1 und A_2 aufgeteilt werden soll. Weiter sei y_i eine eindimensionale Responsevariable mit Erwartungswert $\mu(\boldsymbol{x}_i) = E(y_i|\boldsymbol{x}_i)$. Auf der Suche nach dem optimalen Split des Knotens A werden entsprechend nur diejenigen Beobachtungen $(y_i, \boldsymbol{x}_i)$ betrachtet, für die gilt, dass $\boldsymbol{x}_i \in A$. Es wird somit der Fit des Modells

$$M_A : \mu(\boldsymbol{x}) = \mu \text{ für } \boldsymbol{x} \in A$$

mit dem des Modells

$$M_{A_1,A_2} : \mu(\boldsymbol{x}) = \mu_1 \text{ für } \boldsymbol{x} \in A_1,\ \mu(\boldsymbol{x}) = \mu_2 \text{ für } \boldsymbol{x} \in A_2$$

verglichen. A_1 und A_2 stellen dabei Teilmengen des Knotens A dar und sind definiert als $A_1 = A \cap \{x_i \leq c\}$ und $A_2 = A \cap \{x_i > c\}$. Falls die Responsevariable einer einfachen Exponentialfamilie angehört, wird nun die Differenz in der Devianz der beiden Modelle als

$$D(M_A|M_{A_1,A_2}) = D(M_A) - D(M_{A_1,A_2}), \tag{2.2}$$

berechnet, wobei $D(M_A)$, $D(M_{A_1,A_2})$ die Devianzen des jeweiligen Modells bezeichnen. Schlussendlich wird dasjenige Modell M_{A_1,A_2} gewählt, für welches die Differenz am geringsten ist. [vgl. Tutz (2012)]

Eine weiteres Hilfsmaß ist die *Unreinheit* der Knoten. Sei $Y \in \{1, \ldots, K\}$ ein mehrkategorialer Response mit den Responsewahrscheinlichkeiten $\pi_1(A), \ldots, \pi_k(A)$ innerhalb des Knotens A. Unter Verwendung einer *Unreinheitsfunktion* ϕ kann die

Unreinheit des Knotens A anhand von

$$I(A) = \phi(\pi_1(A), \ldots, \pi_k(A)) \tag{2.3}$$

gemessen werden. ϕ ist symmetrisch in den Argumenten und wird minimal, falls eine der Wahrscheinlichkeiten $P(Y = k) = \pi_k(A)$, $k = 1, \ldots, K$ den Wert 1 annimmt. Typische Messgrößen, für die $\phi(1/k, \ldots, 1/k) \geq \phi(\pi_1, \ldots, \pi_k)$ für alle $\pi_1, \ldots, \pi_k$ gilt, sind der *Gini-Index* und die *Entropie*. Ersterer nutzt die Unreinheitsfunktion $\phi(\boldsymbol{\pi}) = -\sum_{i \neq j} \pi_i \pi_j$. Für die Unreinheit des Knotens gilt damit nach Gleichung 2.3

$$I_G(A) = 1 - \sum_{r=1}^{k} (\pi_r(A))^2. \tag{2.4}$$

Für die Entropie ergibt sich mit $\phi(\boldsymbol{\pi}) = -\sum_r \pi_r \log(\pi_r)$ die Unreinheit

$$I_E(A) = -\sum_{r=1}^{k} \pi_r(A) \log(\pi_r(A)). \tag{2.5}$$

Somit ist der Knoten als „am reinsten“ zu bezeichnen, falls die gesamte Wahrscheinlichkeitsmasse auf eine Responsekategorie fällt und somit $I(A) = 0$ gilt. Umgekehrt wird die Unreinheit maximal, falls die Wahrscheinlichkeitsmasse gleichmäßig auf alle Kategorien verteilt ist und somit $\pi_1(A), \ldots, \pi_k(A) = 1/k$ gilt. Mit w_1 und w_2, so dass gilt $w_1 + w_2 = 1$, kann der Rückgang der Unreinheit mit Hilfe der Formel

$$\Delta_I(A|A_1, A_2) = I(A) - \{w_1 I(A_1) + w_2 I(A_2)\} \tag{2.6}$$

gemessen werden. Die Gewichte w_1 und w_2 werden üblicherweise über den Anteil der Beobachtungen in Knoten A an der Gesamtheit der Beobachtungen bestimmt und es gilt: $w_i = n(A_i)/n(A)$ (vgl. auch Tabelle 2.1). Man erkennt schnell, dass der Rückgang der Unreinheit maximiert wird, falls der Split mit der minimalen Unreinheit $w_1 I(A_1) + w_2 I(A_2)$ gewählt wird. [vgl. Tutz (2012), S.319 ff.]

Da die theoretische Klassenwahrscheinlichkeit in der Regel im praktischen Anwendungsfall nicht bekannt ist, kann sie bei mehrkategorialem Response $Y \in 1, \ldots, k$ über den empirischen Schätzer anhand folgender Kontingenztabelle für den Split des

Knoten A in die Tochterknoten A_1 und A_2 bestimmt werden:

	Y				
	1	2	...	k	Randsumme
A_1	$n_1(A_1)$	$n_2(A_1)$	...	$n_k(A_1)$	$n(A_1)$
A_2	$n_1(A_2)$	$n_2(A_2)$	...	$n_k(A_2)$	$n(A_2)$
	$n_1(A)$	$n_2(A)$	...	$n_k(A)$	$n(A)$

Tabelle 2.1: Kontingenztabelle bei mehrkategorialem Response und Aufspaltung des Knotens A in die Tochterknoten A_1 und A_2 [vgl. Tutz (2012), S.321]

Mit dieser Tabelle und $\hat{\pi}_1(A) = n_1(A)/n(A), \ldots, \hat{\pi}_k(A) = n_k(A)/n(A)$ ergibt sich die empirische Formel für Gleichung 2.4 als

$$I_{G,\,emp}(A) = 1 - \sum_{k=1}^{K} (n_k(A)/n(A))^2 \tag{2.7}$$

und Gleichung 2.6 bei Verwendung der Entropy aus Gleichung 2.5 als

$$\Delta_{I_{E,emp}} = \frac{1}{2n(A)} D(M_A). \tag{2.8}$$

Hierbei gilt

$$D(M_A) = -2 \sum_{k=1}^{K} n_k(A_i) \log(\pi_k(A)) \tag{2.9}$$

und somit für die Differenz der Devianz aus Gleichung 2.2

$$D(M_A | M_{A_1,A_2}) = 2 \sum_{i=1}^{2} n(A_i) \sum_{r=1}^{k} n_r(A_i) \log\left(\frac{p_k(A_i)}{p_k(A)}\right). \tag{2.10}$$

[vgl. Tutz (2012), S.321 ff.]

Bei jedem Split wird somit für jede Prädiktorvarible der optimale Splitwert ermittelt und letztendlich diejenige Kombination gewählt, welche die Unreinheit in den

Knoten minimiert. Mit Hinblick auf die an späterer Stelle der Arbeit vorgestellten Methoden von Bernard et al. (2012) und Mishina et al. (2014) sei hier explizit bemerkt, dass die Berechnungen der Unreinheit allein anhand der Klassenhäufigkeiten durchgeführt werden. In Abschnitt 3.1 und Abschnitt 3.2 werden für diese Berechnung die bereits in der Einleitung angesprochenen Gewichte zum Einsatz kommen.

Wie auch bei den Regressionsbäumen, wird das Wachstum des Baumes beendet, falls ein zuvor festgelegtes Abbruchkriterium erreicht wird. Die Größe des Baumes spielt dabei eine entscheidende Rolle, da es bei zu großen Bäumen zu einer Überanpassung kommen kann. Eine solche Überanpassung hat zur Folge, dass der Baum zwar die Trainingsbeobachtungen nahezu perfekt klassifizieren kann, bei der Klassenzuordnung neuer, unbekannter Beobachtungen jedoch zu beschränkt auf jene Trainingsbeobachtungen ist, um auch die unbekannten Klassenzugehörigkeiten richtig bestimmen zu können. Ein Indiz, dass es sich um Überanpassung handelt, ist eine sinkende Trainingsfehlerrate bei gleichzeitig steigender Testfehlerrate. Mögliche Abbruchkriterien sind daher das Unterschreiten einer Mindesanzahl an Beobachtungen n_{STOP} in einem Knoten oder das Unterschreiten einer zuvor festgelegten Schwelle, zum Beispiel, wenn der Rückgang des Testfehlers nicht mehr hoch genug ist. Eine weitere Möglichkeit, die Größe der Bäume zu optimieren, ist das sogenannte *Pruning*, das *Zurückschneiden* des Baumes. Hierbei werden, wie der Name schon sagt, Äste bereits gewachsener Bäume zurückgeschnitten. Pruning stellt somit sicher, dass auch wichtige Splits durchgeführt werden, welche ansonsten aufgrund eines zuvor eintretenden Abbruchkriteriums nicht mehr durchgeführt worden wären. [vgl. Tutz (2012), S.324]

2.2 Random Forests

Random Forests ist eine erhebliche Abwandlung des *Baggings*, welches (bei Anwendung mit Entscheidungsbäumen) eine große Anzahl (möglichst unähnlicher) Bäume produziert und diese dann mittelt [vgl. Hastie et al. (2009), S.587]. Durch die hohe

Anzahl an Entscheidungsbäumen wird erreicht, dass der hierdurch entstehende Wald *(Forest)* robust gegen eventuelle Ausreißer wird.

Der Name des Baggings, welches von Breiman (1996) vorgestellt wurde, kommt aus dem Englischen von *Bootstrap aggregation*. Hierbei werden anhand eines Datensatzes mit Hilfe des Bootsraps mehrere Trainingsdatensätze erstellt und diese dann gemittelt, um einen geeigneten Schätzwert zu erhalten. Dadurch wird die Varianz des Schätzers verringert, die Vorhersagegenauigkeit erhöht und auch einem Overfitting entgegengewirkt. Aus den einzelnen, anhand des entsprechenden Bootstrap-Trainingsdatensatzes b gefitteten Modellen $\hat{f}^{*1}(x), \ldots, \hat{f}^{*B}(x)$ erhält man so das finale Modell

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

als Mittelwert der einzelnen Modelle $\hat{f}^{*b}$. [vgl. Kubat (2015), S.316 f.]

Die mit Hilfe des Baggings erzeugten Bäume sind identisch verteilt, weswegen auch die gemittelte Erwartung über alle Bäume identisch mit der eines einzelnen Baumes ist. Ebenso identisch ist der Bias. Ziel des *Random Forests* ist es, die Varianz zu reduzieren, indem die Korrelation zwischen den einzelnen Bäumen reduziert wird. Dies kann durch eine zufällige Auswahl m aus der Gesamtzahl p der Prädiktorvariablen des Trainingsdatensatzes als Splitkandidaten in jedem Interationsschritt erreicht werden. Anders als beim Bagging wird hierdurch in dem Fall, dass ein starker Prädiktor vorliegt, nicht nur dieser ständig aufgrund der Splittingregel gewählt, sondern auch andere, weniger starke Prädiktoren. Auf diese Weise unterscheiden sich die entstehenden Bäume. Es gilt dabei immer $m \leq p$. Im Regressionsfall liegt der empfohlene Wert bei $m = \lfloor p/3 \rfloor$ und bei der Klassifikation bei $m = \lfloor \sqrt{p} \rfloor$. [vgl. Kubat (2015), S.320] Dieser Wert kann, ebenso wie die minimale Knotengröße (Regression: mind. fünf Beobachtungen, Klassifikation: mind. eine Beobachtung), als Tuningparameter gesehen werden. Je kleiner dabei m gewählt wird, desto geringer ist die Korrelation unter den entstehenden Bäumen und desto geringer fällt die Varianz im Mittel aus. Ein Anhaltspunkt, wann das Training des *Random Forests* beendet werden kann, ist der *Out-of-Bag-Fehler*. Sobald sich dieser stabilisiert hat, kann die Iteration abgebrochen werden. Bei den sogenannten Out-of-Bag (OOB) Samples wird die Vorhersage für die Beobachtung $z_i = (x_i, y_i)$ nur anhand derjenigen Bäume bestimmt, in welchen

z_i nicht enthalten ist. Auf diese Weise entsteht der OOB-Klassifizierer. [vgl. Hastie et al. (2009), S.587 ff.] Der OOB-Schätzer entspricht etwa der Fehlerrate der OOB-Klassifizierer des Trainingsdatensatzes, wodurch ein Testdatensatz unnötig wird. Eine weitere, vorteilhafte Eigenschaft des OOB-Schätzers ist, dass dieser (vorausgesetzt, dass die Anzahl der Bäume groß genug ist) unverzerrt ist. [vgl. Kubat (2015), S.319]

Der *Random-Forest-Algorithmus* für die Erstellung eines Waldes lautet wie folgt:

1. Für $b = 1$ bis B:
   a) Ziehe eine Bootstrap-Stichprobe $\boldsymbol{Z}^*$ der Größe N aus den Trainingsdaten.
   b) Passe einen Random-Forest-Baum T_b an die durch Bootstrap gezogenen Daten an. Wiederhole dafür die folgenden Schritte rekursiv für jeden Endknoten bis die minimale Knotengröße n_{min} in einem der Knoten erreicht ist.
      i. Wähle zufällig m Variablen aus den p Variablen aus.
      ii. Wähle den besten Split-Punkt/die beste Variable unter den m Variablen aus.
      iii. Teile den Knoten in zwei Tochterknoten.
   c) Ausgabe der Bäume $\{T_b\}_1^B$.

Handelt es sich um ein Regressionsproblem, so wird die Vorhersage der neuen Beobachtung x anhand von $\hat{f}_{rf}^B = \frac{1}{B}\sum_{b=1}^B T_b(x)$ getroffen. Handelt es sich um ein Klassifikationsproblem und sei $\hat{C}_b(x)$ die vorhergesagte Klasse des b-ten RF-Baumes, dann gilt $\hat{C}_{rf}^B(x) = \text{majority vote } \{\hat{C}_b(x)\}_1^B$. [vgl. Hastie et al. (2009), S.588] Als *majority vote (MV)* $\hat{C}_{rf}^B(\boldsymbol{x}_i)$ der Beobachtung i wird diejenige Klasse $\hat{C}(\boldsymbol{x}_i)$ bezeichnet, welcher die Beobachtung i von den Bäumen $b = 1, \ldots, B$ am häufigsten zugeordnet wurde. Für $m = p$ entspricht RF dem Bagging.

Als weitere Berechnung kann die Bedeutung der einzelnen Variablen für die Vorhersagegenauigkeit des Waldes bestimmt werden. Diese *Variablenwichtigkeit* gibt entsprechend an, wieviel die jeweilige Variable zur Vorhersage beiträgt. Auch hierzu

werden die OOB - Stichproben genutzt. Die Idee ist, dass falls eine Variable X_j keinen Einfluss auf die Schätzung hat, also nicht mit dem Response korreliert ist, eine Veränderung eines Wertes $x_{i,j}$ der Variable keine Auswirkungen auf die Schätzung hat. Die Variablenwichtigkeit kann daher anhand der Differenz der Vorhersagegenauigkeit vor und nach einer Permutation, gemittelt über alle Bäume, bestimmt werden: Sei $\overline{B}^{(t)}$ die OOB - Stichprobe eines Baumes t mit $t \in \{1, \ldots, T\}$. Die Variablenwichtigkeit der Variable X_j in Baum t ist dann

$$VI^{(t)}(\boldsymbol{X}_j) = \frac{\sum_{i \in \overline{B}^{(t)}} I\left(y_i = \hat{y}_i^{(t)}\right)}{|\overline{B}^{(t)}|} - \frac{\sum_{i \in \overline{B}^{(t)}} I\left(y_i = \hat{y}_{i,\pi_j}^{(t)}\right)}{|\overline{B}^{(t)}|}$$

mit $\hat{y}_i^{(t)} = f^t(x_i)$ als vorhergesagte Klasse für Beobachtung i vor der Permutation und $\hat{y}_{i,\pi_j}^{(t)} = f^t(x_{i,\pi_j})$ danach. Die Variablenwichtigkeit über alle Bäume wird dann berechnet als Mittel der Werte der einzelnen Bäume

$$VI(\boldsymbol{X}_j) = \frac{\sum_{t=1}^{ntree} VI^{(t)}(\boldsymbol{X}_j)}{T}.$$

[vgl. Strobl et al. (2008), S.5 f.]

2.3 Boosting

Boosting ist ein Verfahren, welches innerhalb verschiedener Methoden des statistischen Lernens angewandt werden kann. Eine dieser Methoden ist der Random Forest, beschrieben in Abschnitt 2.2. Mit den verschiedenen Boosting-Algorithmen (vgl. u.a. Unterabschnitt 2.3.1 und Unterabschnitt 2.3.2) werden sogenannte *Entscheidungsbäume* gefittet. Beim Boosting basiert der zuletzt erstellte Baum immer auf den bereits zuvor erstellten Bäumen. Wie bei *Ensemble-Methoden* üblich, wird am Ende das Ergebnis anhand aller entstandenen Bäume ermittelt. Die Boosting-Methode ist ähnlich zu der des *Bootstraps*, bei welchem aus einem vorliegenden Datensatz eine Anzahl von B verschiedenen Stichproben mit zurücklegen gezogen wird. Anders als beim Bootstrap jedoch, bei welchem die Beobachtungen mit identischer Wahrschein-

lichkeit gezogen werden, kommen beim Boosting Gewichte zum Einsatz, welche in jedem Iterationsschritt neu bestimmt werden und die Ziehung der Beobachtungen beeinflussen. Diese Methode lernt sehr langsam, wodurch ihre Performance in der Regel sehr gut ist [vgl. James et al. (2013), S.321].

Reguliert werden kann diese Lerngeschwindigkeit durch den *shrinkage-Parameter* λ. Er nimmt typischerweise einen Wert von 0.01 oder 0.001 an und ist einer der drei existierenden Tuning-Parametern. Je kleiner λ gewählt wird, desto größer muss B gewählt werden. B steht für die Anzahl der zu fittenden Bäume und wird mit Hilfe von Kreuzvalidierung bestimmt. Der dritte Tuningparameter ist die Anzahl der Splits d. Ein üblicher Wert ist $d = 1$, weshalb die Bäume nur Stümpfe sind und ein additives Modell entsteht, da mit jedem Schritt nur eine Variable hinzukommt. [vgl. James et al. (2013), S.322]

Boosting fittet eine additive Hochrechnung in einer Umgebung von elementaren Basisfunktionen. Eine solche Basisfunktion-Hochrechnung hat im Allgemeinen die Form

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m). \tag{2.11}$$

$\beta_m,\ m = 1, \ldots, M$ sind die Expansionskoeffizienten und $b(\boldsymbol{x}; \boldsymbol{\gamma}) \in \mathbb{R}$ sind für gewöhnlich einfache Funktionen des multivariaten Arguments $\boldsymbol{x}$, welches von einer Menge an Parametern $\boldsymbol{\gamma}$ charakterisiert wird. Diese additiven Modelle werden in der Regel gefittet, indem eine Verlustfunktion $L(\cdot)$, gemittelt über die Trainingsdaten, minimiert wird:

$$\min_{\{\beta_m, \gamma_m\}_1^M} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i;\ \gamma_m)\right). \tag{2.12}$$

Da die Suche nach dem Minimum intensive numerische Optimierungstechniken erfordert, wird, sofern zulässig, nur eine einzelne Basisfunktion gefittet:

$$\min_{\{\beta, \gamma\}} \sum_{i=1}^{N} L\left(y_i, \beta b(x_i;\ \gamma)\right). \tag{2.13}$$

[vgl. Hastie et al. (2009), S.341 f.]

Eine mögliche Lösung von Gleichung 2.12 ist in Hastie et al. (2009) (S.342) mit dem

Forward Stagewise Additive Modeling gegeben:

1. Initialisiere $f_0(x) = 0$.
2. Für $m = 1, \ldots, M$:
   a) Berechne
   $$(\beta_m, \gamma_m) = \underset{\beta,\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i, \gamma)).$$
   b) Setze $f_m(x) = f_{m-1}(x) + \beta_m b(x;\ \gamma_m)$.

Hier wird in jedem Schritt m die optimale Basisfunktion $b(x;\ \gamma_m)$ und der zugehörige Koeffizient β_m gesucht und diese zum gegenwärtigen $f_{m-1}(x)$ hinzuaddiert. Somit ergibt sich $f_m(x)$ und der Prozess wird wiederholt. [vgl. Hastie et al. (2009), S.342]

2.3.1 AdaBoost

Einer der bekanntesten Boosting-Algorithmen ist der *AdaBoost*-Algortihmus von Freund und Schapire (1995). Angenommen es handelt sich um ein Zwei-Klassen-Problem, bei welchem die Beobachtungen als $Y \in \{-1, 1\}$ kodiert sind. Der Klassifizierer $G(X)$, welcher der Basisfunktion in Gleichung 2.11 entspricht, nimmt somit entweder den Wert 1 oder -1 an. Von einem *schwachen Klassifizierer* spricht man, wenn dessen *Fehlerrate*

$$\overline{\text{err}} = \frac{1}{N} \sum_{i=1}^{N} I(y_i \neq G(x_i)) \tag{2.14}$$

nur minimal besser ist als die bei zufälligem Raten (50%). Ein solcher Klassifizierer $G_m(x)$ wird in jedem Schritt m für $m = 1, \ldots, M$ als Anzahl der Bäume auf Basis der mit jedem Schritt abgeänderten Version der Ausgangsdaten erstellt. Diese Veränderung entsteht durch die Anwendung von Gewichten $w_1, \ldots, w_N$ auf die Beobachtungen $(x_i, y_i),\ i = 1, \ldots, N$. Zu Beginn gilt für alle $w_i = 1/N$. Diese werden mit jedem Schritt m des Algorithmus angepasst. Die Neubestimmung der Gewichte

der Beobachtungen richtet sich danach, ob die jeweilige Beobachtung im vorherigen Schritt richtig klassifiziert wurde oder nicht. Falls diese richtig klassifiziert wurde, so wird ihr Gewicht reduziert, wurde sie der falschen Klasse zugeordnet, so wird die Wahrscheinlichkeit, dass sie im nächsten Schritt gezogen wird, größer. Somit werden schwierig zu klassifizierende Beobachtungen öfter mit einbezogen und somit von den Klassifizierern häufiger berücksichtigt. Schlussendlich wird die resultierende Vorhersage

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{2.15}$$

aus allen Klassifizierern $G_m(x)$ gebildet. [vgl. Hastie et al. (2009), S. 337 f.]

Die Gewichte $\alpha_1, \ldots, \alpha_M$ werden innerhalb des Boosting-Algorithmus bestimmt. Durch sie erhalten bessere Klassifizierer mehr Einfluss auf das Gesamtergebnis als schwächere. [vgl. Hastie et al. (2009), S.338 f.]

In Hastie et al. (2009) (S.339) ist der AdaBoost-Algorithmus wie folgt aufgeführt:

1. Initialisiere die Gewichte der Beobachtungen $i = 1, \ldots, N$ $w_i = 1/N$.
2. Für $m = 1, \ldots, M$:
   a) Fitte einen Klassifizierer $G_m(x)$ anhand der Trainingsdaten und den Gewichten w_i.
   b) Berechne
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   c) Berechne $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   d) Setze $w_i \leftarrow w_i \cdot \exp\left[\alpha_m \cdot I(y_i \neq G_m(x_i))\right]$, $i = 1, \ldots, N$.
3. Ausgabe von $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Dieser Algorithmus ist auch bekannt als *Discrete AdaBoost*, da der Basisklassifizierer $G_m(x)$ ein diskretes Klassenlabel ausgibt. Ist dieses eine reelle Zahl, so kann der oben beschriebene Algortihmus leicht abgeändert werden. Es handelt sich dann um den

sogenannten *Real AdaBoost*, welcher für Regression benutzt werden kann. [vgl. Hastie et al. (2009), S.339]

AdaBoost ist identisch zum *Forward Stagewise Additive Modeling*, wenn dort die exponentielle Verlustfunktion

$$L(y, f(x)) = \exp(-yf(x))$$

benutzt wird. Bei AdaBoost entsprechen die Klassifizierer $G_m(x) \in \{-1, 1\}$ den Basisfunktionen $b(x, \gamma)$. Somit muss in jedem Schritt

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} \exp\left[-y_i(f_{m-1}(x_i) + \beta G(x_i))\right] \tag{2.16}$$

für G_m und β_m gelöst werden. [vgl. Hastie et al. (2009), S.343]

Gleichung 2.16 kann auch als

$$(\beta_m, G_m) = \arg\min_{\beta, G} \sum_{i=1}^{N} w_i^{(m)} \exp(-\beta y_i G(x_i)) \tag{2.17}$$

mit $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$ geschrieben werden. Da $w_i^{(m)}$ weder von β noch von $G(x)$ abhängt, kann der Term als Gewicht für die entsprechende Beobachtung gesehen werden, welches von $f_{m-1}(x_i)$ abhängt und sich mit jeder Iteration m verändert. Die genaue Erklärung der Übereinstimmung ist in Hastie et al. (2009) (S.343 f.) nachzulesen.

2.3.2 Gradient Boosting

Als weiterer Boosting-Algorithmus wird hier der des *Gradient Boostings* kurz umrissen, welcher als Verallgemeinerung des *AdaBoost*-Algorithmus verstanden werden kann und seit seiner Einführung durch Friedman (2001) populär und erfolgreich ist. Er basiert auf der Methode des *Gradient Descent*. Ebenso wie beim Gradient Descent wird beim Gradient Boosting das Minimum einer Funktion, hier entsprechend der

Verlustfunktion, gesucht. Grob gesagt wird der Gradient der Verlustfunktion $L(\cdot)$ durch einen Baum approximiert.

Wenn man die Einschränkung, dass f in

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i)) \tag{2.18}$$

eine Summe aus Bäumen sein soll, ignoriert, so kann Gleichung 2.18 auch als numerisches Optimierungsproblem

$$\hat{\boldsymbol{f}} = \underset{\boldsymbol{f}}{\operatorname{argmin}}\, L(\boldsymbol{f}) \tag{2.19}$$

betrachtet werden. Hierbei sind die „Parameter“ die Werte $f(x_i)$ der Approximierungsfunktion $\boldsymbol{f} \in \mathbb{R}^N$ an jedem der N Datenpunkte x_i:

$$\boldsymbol{f} = \{f(x_1), \ldots, f(x_N)\}.$$

Gleichung 2.19 wird daher von numerischen Optimierungsprozeduren als Komponentenvektorsumme gelöst:

$$\boldsymbol{f}_M = \sum_{m=0}^{M} \boldsymbol{h}_m, \quad \boldsymbol{h}_m \in \mathbb{R}^N. \tag{2.20}$$

Dabei ist die Initialisierung von $\boldsymbol{f}_0 = \boldsymbol{h}_0$ geraten und jedes folgende $\boldsymbol{f}_m$ basiert auf dem vorherigen Parametervektor $\boldsymbol{f}_{m-1}$. Dieser ist die Summe der zuvor durchgeführten Updates (Gleichung 2.20). [vgl. Hastie et al. (2009), S.358]

Das bei jeder numerischen Optimierungsprozedur unterschiedliche $\boldsymbol{h}_m$ wird beim *Steepest Decent* als $\boldsymbol{h}_m = -\rho_m \boldsymbol{g}_m$ gewählt. Hier ist ρ ein für die Schrittlänge stehendes Skalar und $\boldsymbol{g}_m \in \mathbb{R}^N$ die Ableitung der Verlustfunktion $L(\boldsymbol{f})$ an der Stelle $\boldsymbol{f} = \boldsymbol{f}_{m-1}$. Für die einzelnen Komponenten des Gradienten ergibt sich somit die Form

$$g_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}. \tag{2.21}$$

ρ_m ergibt sich weiter als Lösung von

$$\rho_m = \underset{\rho}{\operatorname{argmin}} L(\boldsymbol{f}_{m-1} - \rho \boldsymbol{g}_m). \tag{2.22}$$

Am Ende jeder Iteration wird die aktuelle Lösung aktualisiert auf

$$\boldsymbol{f}_m = \boldsymbol{f}_{m-1} - \rho_m \boldsymbol{g}_m.$$

[vgl. Hastie et al. (2009), S.358 f.]

Entsprechend lautet der Algorithmus des Gradient Boostings wie folgt:

1. Initialisiere $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
2. Für $m = 1, \ldots, M$:
   a) Für $i = 1, \ldots, N$ berechne
   $$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}.$$
   b) Fitte einen Regressionsbaum für Zielvariablen r_{im} gegeben die Endregionen $R_{jm},\ j = 1, \ldots, J_m$.
   c) Berechne für $j = 1, \ldots, J_m$
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma).$$
   d) Aktualisiere $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Ausgabe von $\hat{f}(x) = f_M(x)$.

[vgl. Hastie et al. (2009), S.361]

3 Gewichte innerhalb des Random Forest Algorithmus

Wie bereits in Abschnitt 2.2 ausführlich beschrieben ist ein Random Forest (RF) ein Klassifizierer, welcher aus mehreren einzelnen Entscheidungsbäumen, sogenannten „weak Learnern“, besteht. Diese einzelnen schwachen Learner werden mit dem RF zu einem starken vereinigt. RF liefert eine ähnlich gute Performance wie Boosting ab, welches als sehr effizient gilt. Im Laufe der Zeit wurden bereits viele Ansätze entwickelt, um den ursprünglichen RF von Breiman (2001) weiterzuentwickeln und zu verbessern. Dazu zählt zum Beispiel der Versuch, das Baumwachstum stärker zu randomisieren. Weiter gibt es den *Extra-Trees-Algorithmus*, welcher ebenso wie RF die Random-Feature-Selektion beinhaltet, jedoch zusätzlich den Cut-Point zufällig anhand des Merkmals, welches für den Splitting-Test bestimmt wurde, wählt. Beim sogenannten *PERT*-Algorithmus wird der Cut-Point genau mittig zwischen zwei zufällig gezogenen Beobachtungen gewählt und die *Rotation-Forest-Methode* arbeitet mit der Hauptkomponentenanalyse. [vgl. Bernard et al. (2012), S.1 ff.]

Ishwaran (2015) dokumentiert eine tiefgehende Untersuchung des Einflusses der Wahl der Splitregel auf die Performance des RFs. Der Vorgehensweise bei der Bestimmung des Splitmerkmals und -wertes wird hier eine große Bedeutung beigemessen. Denn die Wahl der Splitvariablen und des zugehörigen Splitwertes leisten Studien zufolge einen großen Beitrag zur letztendlichen Vorhersagegenauigkeit des RF.

Im weiteren Verlauf dieses Kapitels werden zwei Algorithmen vorgestellt, welche laut des jeweiligen Papers die Performance des RF nach Breiman (2001) übertreffen. Es handelt sich um den *Boosted-Random-Forest-Algorithmus* von Mishina et al. (2014) und den *Dynamic-Random-Forest-Algorithmus* von Bernard et al. (2012).

3.1 Boosted Random Forest (Mishina et al. (2014))

Ebenso wie beim RF-Algorithmus werden beim *Boosted Random Forest* nach Mishina et al. (2014), um die verschiedenen Trainingsdatensätze zu erhalten, an welche die einzelnen Bäume angepasst werden, zufällige Bootstrap-Stichproben mit Zurücklegen aus der ursprünglichen Trainingsdatenmenge gezogen. Vor der Ersten von $t = 1, \ldots, T$ Iterationsschleifen des Algorithmus werden die Gewichte der einzelnen Beobachtungen $i = 1, \ldots, N$ auf $w_i = 1/N$ und der maximal mögliche Informationszugewinn aus Gleichung 3.3 auf $\Delta G_{max} = -\infty$ festgesetzt. T entspricht der Anzahl an Bäumen, welche für den Wald angepasst werden sollen. Wie beim RF-Algorithmus werden aus der Menge der P Prädiktorvariablen zufällig $m = 1, \ldots, M$ mögliche Splitvariablen gezogen. Aus diesen wird ebenso zufällig die Splitvariable für den aktuellen Knoten A_t gezogen. Für diese Splitvariable m wird nun noch zufällig ein Splitwert τ_h bestimmt. Der Knoten A_t wird dann anhand der Kombination aus m und τ_h in die zwei Tochterknoten A_{tl} und $A_t r$ aufgeteilt. Nachdem dies passiert ist, wird der durch den Split gewonnene Anstieg an Information gemessen. Dieser berechnet sich anhand der Entropie

$$E(A) = -\sum_{j=1}^{K} P(c_j) log P(c_j) \tag{3.1}$$

der Beobachtungen in den jeweiligen Knoten. K steht hier für die Anzahl an Klassen der Responsevariablen $(y_i \in (1, 2, \ldots, K))$ und c_k entsprechend für die k-te Klasse. Die Entropie wiederum wird bestimmt durch die Klassenwahrscheinlichkeiten. An dieser Stelle fließen zum ersten Mal die Gewichte w_i der Beobachtungen mit ein, denn die Wahrscheinlichkeit $P(c_k)$ der Klasse c_k ist festgelegt als

$$P(c_k) = \sum_{i \in A \wedge y_i = c_k} w_i / \sum_{i \in A} w_i. \tag{3.2}$$

Wie in Abschnitt 2.1 bereits angedeutet, nehmen hier die Gewichte explizit Einfluss auf das Splitkriterium. Der letztlich interessierende Anstieg des Informationsgehaltes

des Baumes durch den aktuell durchgeführten Split ergibt sich als

$$\Delta G = E(A) - \frac{|A_l|}{|A|}E(A_l) - \frac{|A_r|}{|A|}E(A_r). \tag{3.3}$$

Dieses neu bestimmte ΔG ersetzt den alten Wert von ΔG_{max}, sofern gilt $\Delta G > \Delta G_{max}$. Falls die minimale, a priori festgelegte Anzahl an Beobachtungen innerhalb eines Knotens durch den letzten Split erreicht wurde, oder gilt, dass $\Delta G_{max} = 0$ (durch den Split erhält der Baum keinen höheren Informationswert), dann wird die Anpassung des Baumes beendet und die zuletzt entstandenen Knoten sind die Endknoten des Baumes. Für jeden dieser Endknoten werden die einzelnen Klassenwahrscheinlichkeiten $P(c_k)$ abgespeichert, aus welchen sich später die finale Klassifikation $\hat{y}_i = \underset{c}{\text{argmax}} P_t(c|A)$ ergibt. Ist keines dieser beiden Kriterien erfüllt, werden so lange weitere Splits durchgeführt, bis dies der Fall ist. Nachdem der Baum t fertiggestellt und für jeden Beobachtungsvektor $\boldsymbol{x}_i$ die erwartete Klasse $\hat{y}_i$ bestimmt worden ist, wird die Fehlerrate ϵ_t für Baum t berechnet als

$$\epsilon_t = \frac{\sum_i^N {w_i}^{(t)} I(y_i \neq \hat{y}_i)}{\sum_{i=1}^N {w_i}^{(t)}}. \tag{3.4}$$

Hier sei angemerkt, dass dies genau der Berechnung wie beim *AdaBoost*-Algorithmus (siehe Gleichung 2.14) entspricht. Mit dieser so berechneten Fehlerrate wiederum wird das Gewicht α_t des Baumes bestimmt als

$$\alpha_t = \frac{1}{2}\log\frac{(K-1)(1-\epsilon_t)}{\epsilon_t}. \tag{3.5}$$

Wird der Wert $1 - 1/K$ (K ist Anzahl der Klassen, $1/K$ ist erwartete Rate korrekter Klassifizierungen) überschritten, so erhält der Baum eine negative Gewichtung, bei Unterschreitung entsprechend eine höhere. Falls der Baum ein Gewicht von $\alpha_t = 0$ erhält, hat er keinen Einfluss auf die Vorhersage des Waldes und die Gewichte der Beobachtungen werden für den nächsten Iterationsschritt nicht verändert. Ist hingegen $\alpha_t > 0$, so werden die Gewichte $w_i^{(t+1)}$ für die Erstellung des nächsten

Baumes angepasst:

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} exp(\alpha_t), & \text{falls } y_i \neq \hat{y}_i \\ w_i^{(t)} exp(-\alpha_t), & \text{sonst.} \end{cases} \tag{3.6}$$

Ist eine Beobachtung zuvor falsch klassifiziert worden, so erhöht sich das Gewicht, wohingegen bei korrekter Klassifizierung das Gewicht der Beobachtung entsprechend geringer wird. Somit liegt der Fokus des nächsten Baumes auf denjenigen Beobachtungen, welche im aktuellen Baum schwieriger richtig zu klassifizieren waren als die Übrigen.

Hat der Wald die gewünschte Größe erreicht, so ist er bereit für die Klassifikation neuer, unbekannter Beobachtungen. Für die Klassifikation der Beobachtung $\boldsymbol{x}_i$ werden für jeden Baum des Waldes zunächst für jede Klasse c die Klassenwahrscheinlichkeiten $P_t(c|\boldsymbol{x}_i)$ des Endknotens, in welchen diese für den jeweiligen Baum t fällt, benötigt. Diese werden im nächsten Schritt entsprechend der während des Trainings bestimmten Gewichte der einzelnen Bäume gewichtet und dann die Summe dieser Wahrscheinlichkeiten über alle Bäume gemittelt. Für jede Klasse erhält man daher die finale Klassenwahrscheinlichkeit der Klasse c als

$$P(c|\boldsymbol{x}_i) = \frac{1}{T} \sum_{t=1}^{T} \alpha_t P_t(c|\boldsymbol{x}_i). \tag{3.7}$$

Wie auch während des Trainings wird die finale Klasse der Beobachtung $\boldsymbol{x}_i$ bestimmt als

$$\hat{y}_i = \underset{c}{\text{argmax}}\ P(c|\boldsymbol{x}_i). \tag{3.8}$$

Für einen Vergleich des Boosted-RF mit dem Standard-RF wurden von Mishina et al. (2014) sowohl beide Algorithmen als auch zusätzlich der Boosted-RF ohne Aktualisierung der Gewichte auf fünf ausgewählten Datensätzen angewandt. Je Datensatz wurde die Tiefe der Bäume variiert. Diese lag pro Datensatz jeweils bei $5, 10, 15$ und 20 Splits. Eine graphische Darstellung der Fehlklassifikationsraten aller Methoden zeigt, dass bei zunehmender Tiefe der Bäume die Performance aller drei Algorithmen nahezu identisch ist, der Boosted-RF jedoch vor allem bei einer Parameterwahl der

Splittiefe von 5 und 10 wesentlich genauere Ergebnisse erzielt. Ein weiterer, nicht zu verachtender Vorteil des Boosted-RF gegenüber dem Standard-RF ist der geringere beanspruchte Speicherplatz während der Ausführung.

3.2 Dynamic Random Forest (Bernard et al. (2012))

Mit Hilfe des *Sequential-Forward-Search*-Algorithmus können aus einem bestehenden Wald nur diejenigen Bäume selektiert werden, welche tatsächlich zu einer besseren Performance dessen beitragen. Es ist festzustellen, dass immer eine Untermenge an Bäumen gefunden werden kann, welche den gesamten Wald aussticht. Ursache hierfür ist die Unabhängigkeit der einzelnen Bäume voneinander. In Bernard et al. (2012) wird daher ein Ansatz vorgeschlagen, welcher diese „Performance mindernden", Bäume gar nicht erst in das Ensemble von Bäumen mit aufnimmt, indem jeder Baum auf Grundlange der Gesamtmenge der bereits angepassten Bäume entsteht. Auf diese Weise könnte mit weniger Bäumen eine bessere Performance erzielt werden. Dieser neue Algorithmus nennt sich *Dynamic Random Forest* und wird im Folgenden näher beschrieben.

Beim Dynamic-RF besteht das Resampling aus einer Vorgehensweise, die sowohl etwas mit Bagging als auch mit Boosting gemeinsam hat. Auf diese Weise werden zum einen die zwei effizienten randomisierenden Techniken des RFs erhalten und zusätzlich das adaptive Resampling des Boostings genutzt. Jede Beobachtung wird dabei gemäß des Anteils gewichtet, zu welchem der zugehörige Response bereits richtig vorhergesagt wurde. Um eine Vorhersage für die jeweilige Beobachtung $\boldsymbol{x}_i$ treffen zu können, werden nur jene Bäume hinzugezogen, in welchen $\boldsymbol{x}_i$ *out-of-bag (OOB)* war. Diese Vorgehensweise ermöglicht eine Berechnung ohne zusätzlichen Testdatensatz (vgl. Abschnitt 2.2). Somit wird auch das Risiko der Überanpassung (Overfitting) limitiert. Für die Gewichtung wird die Funktion

$$c(\boldsymbol{x}_i, y_i) = \frac{1}{|h_{oob}|} \sum_{h_t \in h_{oob}} I\left(h_i(\boldsymbol{x}_i) = y_i\right) \tag{3.9}$$

vorgeschlagen. Hier steht $|h_{oob}|$ für die Anzahl aller Bäume, für welche $\boldsymbol{x}_i$ OOB war und $h_t(\boldsymbol{x}_i)\,, t = 1, \ldots, T$, für die der Beobachtung $\boldsymbol{x}_i$ durch den Baum h_t zugeordneten Klasse. Je kleiner Gleichung 3.9, desto stärker muss die jeweilige Beobachtung im nächsten Baum mit einbezogen werden. Die finale Gewichtsfunktion ergibt sich somit als

$$W(c(\boldsymbol{x}_i, y_i)) = 1 - c(\boldsymbol{x}_i, y_i). \tag{3.10}$$

[vgl. Bernard et al. (2012), S.5 ff.]
Hier sei anzumerken, dass diese Gewichtsfunktion nicht die einzige mögliche ist. Während beim Boosted-RF eine ähnliche Grundlage für die Gewichtung der Beobachtungen besteht, ist hier hervorzuheben, dass in Gleichung 3.6 nur die Fehlerrate des letzten Baumes betrachtet wird. Durch die Berechnung der Fehlerrate wie in Gleichung 3.9 werden hier jedoch alle bereits angepassten Bäume für die Berechnung der Fehlerrate hinzugezogen.

Um einen Dynamic-RF zu erstellen wird eine *Trainingsmenge* Θ benötigt, welche aus den N Beobachtungen $(\boldsymbol{x}_i, y_i)$ mit Ausprägungen der P Prädiktorvariablen besteht. T gibt die gewünschte Anzahl an Bäumen an, welche für den Wald angepasst werden sollen. Die Anpassung der einzelnen Bäume $t = 1, \ldots, T$ erfolgt auf Grundlage des folgenden Algorithmus:

1. Initialisierung des Gewichtsvektors $\boldsymbol{w}$: $w_i^{(0)} = \frac{1}{N}$ für $i = 1, \ldots, N$.
2. Ziehung der Bootstrap-Stichprobe Θ_t mit Zurücklegen.
3. Gewichtung von Θ_t durch $\boldsymbol{w}^{(t-1)}$.
4. Anpassen des Baumes t an Θ_t und Hinzufügen des Baumes zum bereits bestehenden Wald.
5. $w_i^{(t)} = \begin{cases} W(c(\boldsymbol{x}, y)), & falls\ \boldsymbol{x}_i\ OOB \\ w_i^{(t-1)}, & sonst \end{cases}, i = 1, \ldots, N.$
6. Normalisieren der Gewichte: $w_i^{(t)} = \frac{w_i^{(t)}}{\sum_{i=1}^{N} w_i^{(t)}}$
7. T-malige Wiederholung der Schritte $2 - 6$.

8. Ausgabe des Waldes.

9. Klassifizierung jeder Beobachtung x_i anhand des *majority votes*:
   $\hat{C}^T_{drf}(x) = \text{majority vote } \{\hat{C}_t(x)\}^T_1$

Dabei werden die Bäume mit Hilfe der sogenannten *Random Tree Induction*-Prozedur angepasst. Um einen verlässlichen Wald zu erhalten, sollte mit der Anpassung der Gewichte jedoch erst begonnen werden, wenn bereits 20 oder mehr Bäume gefittet wurden. Auf diese Weise soll die Wahrscheinlichkeit erhöht werden, dass jede Beobachtung zumindest einmal *out of bag* gewesen ist. Für die Anpassung eines Entscheidungsbaumes durch die *Random Tree Induction*-Prozedur wird für jedes Merkmal der Informationszugewinn berechnet, welcher entsteht, wenn das jeweilige Merkmal als Splitkriterium gewählt werden würde. Anhand dieses Wertes wird der Anteil π derjenigen Merkmale ermittelt, welche den geringsten Zugewinn aufweisen. Für diesen Anteil π muss zuvor ein Schwellenwert festgelegt werden. Für jeden Knoten des Baumes wird nun π mit diesem Schwellenwert verglichen. Ist der Wert von π größer als der zuvor bestimmte Schwellenwert, so wird die spätere Anzahl M der möglichen Splitkandidaten aus der Gleichverteilung $M \sim U(\frac{1}{P})$ bestimmt, wobei P für die Anzahl aller möglichen Prädiktorvariablen steht. Wird der Schwellenwert jedoch andererseits vom Wert π unterschritten, so ergibt sich M aus einer Normalverteilung für die $M \sim N(\sqrt{P}, \frac{P}{50})$ gilt. Nachdem zufällig M der P Merkmale als Splitkanditaten gezogen wurden, wird nun anhand der Unreinheitsregel (vgl. Gleichung 2.3) die Kombination aus Splitvariable und Splitpunkt gewählt, welche eben diese minimiert und der Knoten anhand dieser in zwei Tochterknoten aufgeteilt. Wie die Gewichte genau angewandt werden, ist in Bernard et al. (2012) nicht beschrieben. Im Vergleich mit dem Standard-RF schneidet der Dynamic-RF jedoch überwiegend besser ab. Der Vergleich ist anhand realer Datensätze erfolgt. Eine Übersicht über diese und die Ergebnisse ist in Bernard et al. (2012) zu finden.

3.3 Boosted Random Forest dieser Arbeit

Die in dieser Arbeit implementierte Methode soll dazu dienen, den Einfluss einer Gewichtung der einzelnen Beobachtungen x_i, welche sowohl Einfluss auf die Bestimmung des *majority votes* als auch die Ziehung der Bootstrap-Stichprobe haben kann, auf die Gesamtperformance des Waldes zu untersuchen. Dazu werden die Beobachtungen einmal so wie in Mishina et al. (2014) (vgl. Abschnitt 3.1) und einmal wie in Bernard et al. (2012) (vgl. Abschnitt 3.2) gewichtet. Außerdem soll der Einfluss einer Gewichtung der einzelnen Bäume anhand ihrer Vorhersagegenauigkeit untersucht werden. Auch diese Vorgehensweise wird von Mishina et al. (2014) übernommen. Der einzige Unterschied liegt hier in der Vorgehensweise, wie die Knoten der einzelnen Bäume aufgeteilt werden. Während bei Mishina et al. (2014) auch hier schon die Gewichte zum Einsatz kommen (vgl. u.a. Gleichung 3.3), werden die Bäume innerhalb der neuen Funktion unter Verwendung der `rfsrc()`-Funktion aus dem `R`-Paket `randomForestSRC` (Ishwaran und Kogalur (2016)) angepasst. Dadurch entfällt zwar die Anwendung der Gewichte in den Splits, ermöglicht aber eine unverzerrte Evaluation der Wirkung von Gewichten in den Endknoten. Wie genau die Gewichte bei Bernard et al. (2012) angewandt wurden geht aus dem Paper nicht eindeutig hervor. Ebenso wird bei Bernard et al. (2012) die *Random-Tree-Induction*-Prozedur zum Fit der Bäume angewandt, bei welcher die Anzahl der als Splitvariablen in Frage kommenden Prädiktorvariablen anhand vorausgegangener Berechnungen bestimmt wird (vgl. Abschnitt 3.2). Bei der `rfsrc()`-Funktion kommen standardmäßig $\sqrt{p}$ der p Prädiktorvariablen als Splitvariablen in Frage.

3.3.1 Mögliche Parametereinstellungen

Innerhalb der Funktion sind verschiedene Einstellungen möglich. Eine Übersicht über diese ist in Tabelle 3.1, sowie eine Übersicht über die Ausgabewerte in Tabelle 3.2 zu finden.

Parameter	Beschreibung
`TY`	Zielvariablenvektor der Trainingsdaten
`TX`	Matrix der Prädiktorvariablen der Trainingsdaten
`forest.size`	Integer; bestimmt Anzahl der Bäume im Random Forest; Default ist 300
`leaf.weights`	falls TRUE (Default), Anwendung der Gewichte in den Endknoten
`sample.weights`	falls TRUE, Anwendung der Gewichte bei Ziehung der Bootsrap-Stichprobe (Default: FALSE)
`init.weights`	Vektor, welcher die Gewichte der einzelnen Beobachtungen spezifiziert, Default ist $1/N$
`weight.threshold`	Integer; bestimmt, nach welchem Baum die Gewichtung einsetzt (nur bei `method = "bernard"`, Default ist 20)
`smoothness`	falls `converge = TRUE`: Integer; bestimmt Anzahl der voherigen Bäume, anhand welcher der mittlere OOB-Fehler bestimmt wird (Default ist 100)
`conv.threshold`	erlaubte mittlere Abweichung des OOB-Fehlers (Default ist 0.001)
`converge`	falls TRUE, bricht Algorithmus bei Erreichen von `conv.threshold` ab (Default ist TRUE)
`stopTreeOut`	falls TRUE, `converge = FALSE` und `forest.size` groß genug gewählt: gibt an, nach wie vielen Bäumen Algorithmus bei `converge = TRUE` abgebrochen hätte
`method`	Gibt an, auf welche Weise die Gewichte berechnet werden sollen, entweder `"bernard"` oder `"mishina"`. Default ist `"mishina"`.
`TreeWeights`	Falls TRUE, gewichtet die Bäume für spätere `predict.brf()`-Funktion. Nur möglich, falls `method = "mishina"`.
`vote.type`	Entweder `"hard"` (Einfacher Majority Vote, anhand der Klassenhäufigkeiten) oder `soft` (Majority Vote anhand der Summe der Klassenwahrscheinlichkeiten über alle Bäume)
`invertWeights`	Default ist FALSE, falls TRUE: richtig klassifizierte Beobachtungen werden stärker gewichtet als falsch klassifizierte (nur bei `method = "bernard"`)

Tabelle 3.1: Parametereinstellungen der neuen Funktion

Ausgabewerte	Beschreibung
`oob.error`	Entwicklung der Out-of-Bag-Fehlerrate über die Bäume
`leaf.weights`	Wert des Parameters `leaf.weights` innerhalb der Funktion
`predictions`	Matrix der Vorhersagen der einzelnen Bäume
`forest`	Liste aller Bäume
`num.trees`	Anzahl der Bäume im Wald
`num.levels`	Anzahl der Klassen der Zielvariablen
`levels`	Klassennahmen der Zielvariablen
`method`	Methode, welche innerhalb der Funktion benutzt wurde (`"mishina"` oder `"bernard"`)
`vote.type`	Art der Berechnung des *majority votes*
`classProbs`	Liste der einzelnen Klassenwahrscheinlichkeiten je Baum
`x.vars`	Name der Prädiktorvariablen
`treeWeight`	Gewichte der einzelnen Bäume, falls `TreeWeights = T`
`weights`	Matrix der Gewichte pro Iterationsschritt

Tabelle 3.2: Ausgabewerte der neuen Funktion

Die Parametereinstellung `sample.weights = TRUE`

Hier sollen nun einige Vorüberlegungen getroffen werden, was von der Parametereinstellung `sample.weights = T` zu erwarten ist. Bei dieser Einstellung fungieren die Gewichte, welche den Beobachtungen am Ende eines jeden Interationsdurchlaufs t zugeteilt werden, als Wahrscheinlichkeiten dieser bei der Bootstrap-Stichprobe des folgenden Iterationsdurchlaufs $t + 1$ gezogen zu werden. Sobald der durch `weight.threshold` festgelegte Schwellenwert der minimalen Baumanzahl erreicht ist, beginnt die Gewichtung anhand der OOB-Fehlerrate der jeweiligen Beobachtung (falls `method = "bernard"`) bzw. anhand der in Abschnitt 3.1 vorgestellten Gewichtung (falls `method = "mishina"`). Auf diese Weise sollen Beobachtungen, welche in den vorausgegangenen Bäumen falsch klassifiziert worden sind, einen stärkeren Einfluss auf das Training der folgenden Bäume nehmen. Bei diesen Beobachtungen

kann es sich demnach um solche handeln, deren Klassenhäufigkeit in Relation zu den übrigen Klassen sehr gering ist. Falls diese Ungleichheit in den Klassenhäufigkeiten bei der Initialisierung der Gewichte nicht beachtet wird, kann es passieren, dass keine oder nur wenige Beobachtungen mit der entsprechenden Ausprägung in der Zielvariablen in die Bootstrap-Stichprobe gelangen. Somit ist es unausweichlich, dass dieser Baum sie falsch klassifiziert. Durch die höhere Gewichtung steigt die Wahrscheinlichkeit, Einfluss auf das Training des nächsten Baumes zu nehmen, wodurch in den Endknoten dieses Baumes auch Ausprägungen der nicht so häufig auftretenden Klasse vorherrschen. Eine weitere Ursache, wieso Beobachtungen falsch klassifiziert werden, könnte eine eventuelle Ausreißereigenschaft sein. Hier seien Ausreißer als solche definiert, bei welchen die Prädiktorvariablen in keinem Zusammenhang mit der Zielvariable stehen. Durchläuft eine solche Beobachtung einen Baum, wird sie erwartungsgemäß in dem der Struktur ihrer Prädiktorvariablen entsprechenden Endknoten landen. Ihr wird nun die Klasse zugeordnet, welche in diesem Knoten dominiert. Da die sich in diesem Endknoten befindenden Ausprägungen der Zielvariablen jedoch aufgrund des anderen Zusammenhangs zwischen Prädiktor- und Zielvariable der Trainingsbeobachtungen von der tatsächlichen Klasse des Ausreißers unterscheiden sollten, wird die Ausreißerbeobachtung falsch klassifiziert. Somit ist zu vermuten, dass auch den eventuell in den Daten vorkommenden Ausreißern eine höhere Gewichtung zugeteilt werden wird.

Aufgrund der Eigenschaften der Bootstrap-Ziehung (Ziehen mit Zurücklegen) und der voraussichtlich hohen Gewichtung, und damit hohen Wahrscheinlichkeit der Beobachtungen, in die Bootstrap-Stichprobe gezogen zu werden, könnten nun ausschließlich solche, zuvor falsch klassifizierten Beobachtungen für das Training des Baumes verwendet werden. Folglich wird der entstehende Baum die verbleibenden, „normalen“ Beobachtungen falsch klassifizieren. Da der Anteil dieser Beobachtungen (nicht-Ausreißer) an der Stichprobe wesentlich höher ausfällt als der Anteil der Ausreißer, sollte dies einen Anstieg der OOB-Fehlerrate zur Folge haben. Ebenso sollte später bei Vorhersage der Klassenzugehörigkeiten des Testdatensatzes die Testfehlerrate mit zunehmender Größe des Waldes ansteigen. Dies ist eine nicht wünschenswerte Eigenschaft, welche sich in Abschnitt 4.2 bestätigt.

Die Parametereinstellung `leaf.weights = TRUE`

Wie in Tabelle 3.1 nachzulesen, bewirkt die Einstellung `leaf.weights = TRUE` einen Einfluss der Gewichte auf die Klassenhäufigkeiten in den Endknoten und somit unweigerlich auch auf den *majority vote (MV)*. Die Gewichtung in den Endknoten kommt dadurch zustande, dass für den MV nicht mehr diejenige Klasse der Trainingsbeobachtungen gewählt wird, welche in dem Endknoten (absolut) am häufigstens vorkommt, sondern jene, für welche die Klassenwahrscheinlichkeit $P(c_j) = \sum_{i \in A \wedge y_i = c_j} w_i / \sum_{i \in A} w_i$ am höchsten ist. Wie auch in Abschnitt 3.1 steht y_i für die beobachtete Klasse der jeweiligen Beobachtung i und $c_j,\ j = 1, \ldots, M$ für die j-te der M Klassen der Zielvariable. Wie auch durch die Parametereinstellung `sample.weights = TRUE` wird durch diese Art der Bestimmung des MV solchen Beobachtungen, welche sich aufgrund ihrer sich von der der übrigen Beobachtungen unterscheidenden Struktur schwierig klassifizieren lassen, ein höheres Gewicht zukommengelassen. Gibt es nur wenige Ausprägungen einer bestimmten Klasse in einem Endknoten, so wird den den Baum durchlaufenden Testbeobachtungen aufgrund der höheren Gewichtung der zugehörigen Trainingsbeobachtung unter Umständen dennoch eben diese Klasse zugewiesen.

Die Parametereinstellung `converge = TRUE`

Die Parametereinstellung `converge = TRUE` bietet einen Vorteil gegenüber den üblichen RF-Funktionen. Durch diese ist die Möglichkeit gegeben, die Anpassung des Waldes zu beenden, sobald die OOB-Fehlerrate konvergiert. Dadurch wird es unnötig, die genaue Zahl der gewünschten Bäume a priori festzulegen. Diese Angabe a priori zu treffen, ist zum Teil schwierig, da die für eine maximale Genauigkeit des Waldes nötige Größe unbekannt ist. Grundsätzlich gilt, dass die OOB-Fehlerrate beim RF mit zunehmender Anzahl an Bäumen konvergiert. Dennoch ist ein automatischer Abbruch wünschenswert, um Rechenzeit einzusparen, welcher durch die `converge = TRUE` Einstellung der Funktion gegeben ist (vgl. Tabelle 3.1). Um zu bestimmen, ob die OOB-Fehlerrate (ϵ_{oob}) bereits konvergiert, wird,

sobald $T = \texttt{smoothness} + 1$ Bäume angepasst wurden, die über alle diese Bäume gemittelte OOB-Fehlerrate $moob = \frac{1}{T}\sum_{t=1}^{T} \epsilon_{oob}^{(t)}$ bestimmt. Im nächsten Schritt wird für jeden Baum $t \in 1, \ldots, T$ die Abweichung der OOB-Fehlerrate der Bäume $1, \ldots, t$ von dem zuvor berechneten Mittelwert aller T Bäume bestimmt als $diffoob^{(t)} = |\epsilon_{oob}^{(t)} - moob|$. Im letzten Schritt wird diese über alle betrachteten Bäume gemittelt und als $mdiffoob = \frac{1}{T}\sum_{t=1}^{T} diffoob^{(t)}$ notiert. Die OOB-Fehlerrate wird als konvergierend eingestuft, falls gilt, dass $mdiffoob < \texttt{conv.threshold}$. Zur Veranschaulichung dieser Prozedur diene Abbildung 3.1.

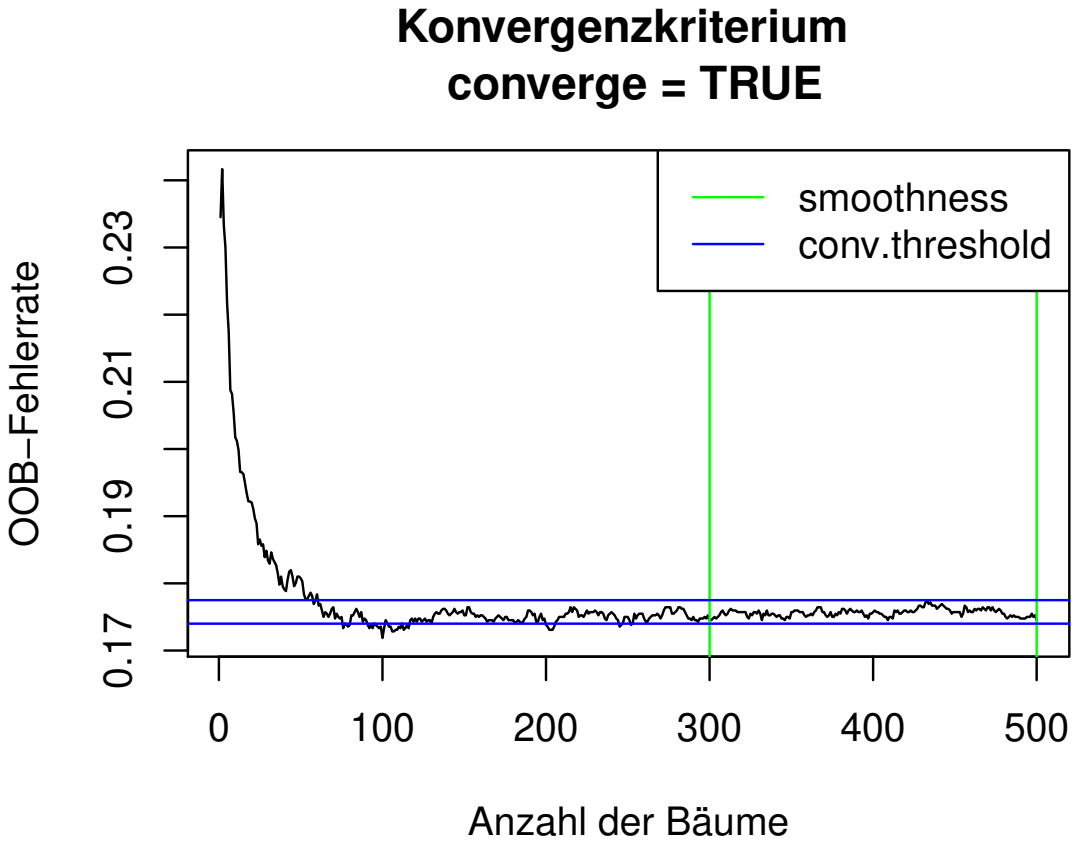

Abbildung 3.1: Allgemeine Darstellung des Konvergenzkriteriums `converge = TRUE`

Die Parametereinstellung `method`

Durch den Parameter `method` wird vorgegeben, auf welche Art die Gewichte der Beobachtungen bei Anpassung des Waldes bestimmt werden. Für `method = "bernard"` werden die Gewichte anhand der OOB-Fehlerrate der jeweiligen Beobachtung i über alle bereits erstellten Bäume berechnet (vgl. Abschnitt 3.2), für `method = "mishina"` entprechend wie in Abschnitt 3.1 unter Verwendung der jeweiligen Baumfehlerrate. Der größte Unterschied dabei ist, dass bei erstgenannter Methode *alle* bereits trainierten Bäume mit einbezogen werden, bei zweiter jedoch immer nur der aktuelle Baum. Dass dies einen Unterschied auf die Vorhersagegenauigkeit haben kann, zeigt sich in Abschnitt 4.1 und Abschnitt 4.2.

3.3.2 Der Algorithmus in Pseudocode

Im Folgenden ist der Algorithmus der Funktion in Pseudocode beschrieben. Dabei wird zwischen der Variante mit a priori vorgegebener Anzahl an Bäumen und der Variante mit Konvergenzkriterium unterschieden.

Variante mit a priori vorgegebener Anzahl an Bäumen

Für $t = 1, \ldots, T$:

1. Anpassung eines Baumes an die zugehörige Bootstrap-Stichprobe durch die Funktion `rfsrc()`:
   - falls `sample.weights = FALSE`: Die Ziehungswahrscheinlichkeiten der Beobachtungen für die Bootstrap-Stichprobe entsprechen einer Gleichverteilung.
   - falls `sample.weights = TRUE`: Die Ziehungswahrscheinlichkeiten der Beobachtungen für die Bootstrap-Stichprobe entsprechen dem vorgegebenen Gewichtsvektor $\boldsymbol{w}^{(t)}$.

2. Hinzufügen des neu erstellten Baumes zum Wald.

3. Klassifizierung der OOB-Beobachtungen anhand des t-ten Baumes und Bestimmung des *majority votes* anhand der Bäume $1, \ldots, t$:

   - falls `leaf.weights = FALSE`:

     a) Bestimmung des *majority votes* des t-ten Baumes als die Klasse, welche im zugehörigen Knoten die größte Wahrscheinlichkeit zugewiesen bekommt

     b) Bestimmung des *majority votes* der Bäume $1, \ldots, t$ als die Klasse, in welche die entsprechende Beobachtung am häufigsten klassifiziert wurde.

   - falls `leaf.weights = TRUE`:

     a) Für jede Klasse k aus $1, \ldots, K$ und jeden Endknoten l aus $1, \ldots, L$: Berechnung der Summe der Gewichte der Trainingsbeobachtungen $\boldsymbol{x}_{i,\Theta}$ in Knoten l, welche aus Klasse k sind

     $$S_{l,k,i} = \sum_{\boldsymbol{x}_{i,\Theta} \in l \wedge y_{i,\Theta} = k} w_{i,\Theta}^{(t)},$$

     $w_{i,\Theta}^{(t)}$ ist Gewicht der Beobachtung in Iterationsschleife t.

     b) Für jeden Endknoten l aus $1, \ldots, L$: Berechnung der Gesamtsumme der Gewichte aller Trainingsbeobachtungen $\boldsymbol{x}_{i,\Theta}$ in Knoten l

     $$S_{l,i} = \sum_{\boldsymbol{x}_{i,\Theta} \in l} w_{i,\Theta}^{(t)}$$

     c) Berechnung der Klassenwahrscheinlichkeiten des t-ten Baumes für jeden Endknoten l als

     $$P(k|l)^{(t)} = \sum_{\boldsymbol{x}_{i,\Theta} \in l \wedge y_{i,\Theta} = k} \frac{S_{l,k,i}}{S_{l,i}}.$$

d) Abspeichern der Klassenwahrscheinlichkeiten für Baum t in Liste.

- falls `vote.type = soft`:

  i. Berechnung der über alle Bäume $1, \ldots, t$ gemittelten Klassenwahrscheinlichkeiten jeder OOB-Beobachtung $\boldsymbol{x}_{i,oob}$ für jede Klasse k anhand der einzelnen Klassenwahrscheinlichkeiten im entsprechenden Endknoten:

$$P(k)_{i,oob} = \frac{1}{t} \sum_{j=1}^{t} P(k|l)^{(j)} I(\boldsymbol{x}_{i,oob} \in l \wedge y_{i,\Theta} = k)$$

  ii. Bestimmung des *majority votes* als

$$\hat{y}_{i,oob} = \arg\max_{k} P(k)_{i,oob}.$$

- falls `vote.type = hard`:

  i. Bestimmung des *majority votes* für jede OOB-Beobachtung $\boldsymbol{x}_{i,oob}$ anhand der einzelnen *votes* der Bäume $1, \ldots, t$

$$\hat{y}_{i,t} = \operatorname{argmax}_{k} P(k|l)^{(t)}.$$

  Es gilt

$$\hat{y}_i = \operatorname{argmax}_{k} \sum_{j=1}^{t} P(k|l)^{(j)} I(y_{i,oob} \in l).$$

4. Bestimme für jede Beobachtung, für wieviele Bäume sie OOB war: $\#oob_i$.

5. Bestimme die OOB-Fehlerrate

$$\epsilon_{oob} = \sum_{i,\, \#oob_i \neq 0} \frac{I(\hat{y}_i \neq y_i)}{\#oob_i}.$$

6. Berechnung des Gewichtsvektors $\boldsymbol{w}^{(t+1)}$:

- Falls `method = mishina`:

  a) Bestimmung des Baumfehlers

  $$\epsilon_t = \frac{\sum_{i,\, \hat{y}_i \neq y_i} w_i^{(t)}}{\sum_i w_i^{(t)}}.$$

  b) Bestimmung des Baumgewichts:

  $$\alpha_t = \begin{cases} \frac{1}{2}\log\left(\frac{(K-1)(1-\epsilon_t)}{\epsilon_t}\right), & \text{falls } \epsilon_t > 0 \\ \frac{1}{2}\log\left(\frac{(K-1)(1-\epsilon_t)}{10^{-5}}\right), & \text{falls } \epsilon_t = 0 \end{cases}$$

  c) Anpassen der Gewichte:

  $$w_i^{(t+1)} = \begin{cases} w_i^{(t)} \cdot \exp(\epsilon_t), & \text{falls } \hat{y}_i \neq y_i \\ w_i^{(t)} \cdot \exp(\epsilon_t), & \text{falls } \hat{y}_i = y_i \end{cases}$$

- Falls `method = bernard`:

  a) Bestimmung, für wie viele Bäume $\hat{y}_i = y_i$ gilt, falls $\boldsymbol{x}_i$ für den Baum OOB war: $\gamma_i = \sum_{j=1}^{t} I(\hat{y}_i = y_i \wedge \boldsymbol{x}_i \, OOB)$

  b) $w^{(t+1)} = 1 - \frac{\sum_{i, \#oob_i \neq 0} \gamma_i}{\#oob_i}$
  Falls $w^{(t+1)} = 0$ setze $w^{(t+1)} = 10^{-12}$.

  c) Normalisieren der Gewichte: $w_i^{(t+1)} = \frac{w_i^{(t+1)}}{\sum_{i=1}^{N} w_i^{(t+1)}}$

Variante mit Konvergenzkriterium

Solange $t \leq$ `smoothness`: Schritt 1 bis 6 wie bei der Variante mit a priori festgelegter Anzahl an Bäumen.

7. Falls $t >$ `smoothness`, Bestimmung von:

   a) $moob = \frac{1}{t} \sum_{j=1}^{t} \epsilon_{oob}^{(j)}$

   b) $diffoob^{(t)} = |\epsilon_{oob}^{(t)} - moob|$

c) $mdiffoob = \frac{1}{t} \sum_{j=1}^{t} diffoob^{(j)}$

8. Falls $mdiffoob <$ `conv.threshold`, setze `convergence = TRUE` und beende Algorithmus. Ansonsten springe zu Schritt 1.

4 Performance-Evaluierung der neuen Funktion

In diesem Kapitel werden die verschiedenen Möglichkeiten der neuen Funktion zunächst auf einige wenige Datensätze aus dem `R`-Paket `OpenML` (Casalicchio et al. (2015)) angewandt und die resultierenden Testfehlerraten dabei sowohl untereinander als auch mit denen des Standard-RFs verglichen. Nachdem anhand der graphischen Darstellung der Testfehlerraten ein erster Eindruck der Performance der unterschiedlichen Möglichkeiten der Funktion gewonnen werden konnte, werden in Abschnitt 4.2 Datensätze simuliert und erneut Wälder angepasst. Anhand der simulierten Daten können eventuell auftretende Ungereimtheiten besser auf die Beschaffenheit der Daten zurückgeführt werden, da diese bei den simulierten Daten, anders als bei den realen Datensätzen, bekannt ist.

4.1 Testfehlerraten im Vergleich

In diesem Abschnitt sollen die Testfehlerraten unter Verwendung des Standard-RF und der unterschiedlichen Gewichtungen innerhalb der neuen Funktion miteinander verglichen werden, um einen ersten Eindruck der jeweiligen Performance zu erhalten. Die entsprechenden Parametereinstellungen sind in Tabelle 4.2 nachzulesen. Für den Vergleich werden fünf verschiedene Datensätze aus dem `R`-Paket `OpenML` (Casalicchio et al. (2015)) verwendet. Da diese nur einen ersten Eindruck verschaffen sollen, wurden sie aufgrund ihrer geringen Beobachtungszahl ausgewählt. Für weitere sechs Datensätze, welche mehr Beobachtungen enthalten und für welche darüber hinaus die Kombination aus Klassen- und Merkmalzahl (Prädiktorvariablen) variiert, werden die mittleren Testfehler unter Verwendung von 5-facher Kreuzvalidierung berechnet und im Anschluss präsentiert. Eine Übersicht über diese Datensätze und ihre Eigenschaften (Anzahl der Klassen, Beobachtungen und Merkmale) ist in Tabelle 4.1 zu finden.

	TaskID	DataID	Name	#Klassen	#Merkmale	#Beob.
1	59	61	iris	3	5	150
2	52	53	heart-statlog	2	14	270
3	38	39	ecoli	8	8	336
4	11	11	balance-scale	3	5	625
5	37	37	diabetes	2	9	768
6	12	12	mfeat-factors	10	217	2000
7	14	14	mfeat-fourier	10	77	2000
8	16	16	mfeat-karhunen	10	65	2000
9	18	18	mfeat-morphological	10	7	2000
10	20	20	mfeat-pixel	10	241	2000
11	22	22	mfeat-zernike	10	48	2000

Tabelle 4.1: Eigenschaften der Datensätze

Objektname	leaf.weights	sample.weights	method	vote.type	TreeWeights
BRF1	TRUE	FALSE	Bernard	hard	FALSE
BRF2	TRUE	TRUE	Bernard	hard	FALSE
BRF3	TRUE	FALSE	Bernard	soft	FALSE
BRF4	TRUE	TRUE	Bernard	soft	FALSE
BRF5	TRUE	FALSE	Mishina	hard	FALSE
BRF6	TRUE	TRUE	Mishina	hard	FALSE
BRF7	TRUE	FALSE	Mishina	soft	FALSE
BRF8	TRUE	TRUE	Mishina	soft	FALSE
BRF9	TRUE	FALSE	Mishina	soft	TRUE
BRF10	TRUE	TRUE	Mishina	soft	TRUE

Tabelle 4.2: Auflistung der Parametereinstellungen der miteinander verglichenen Objekte

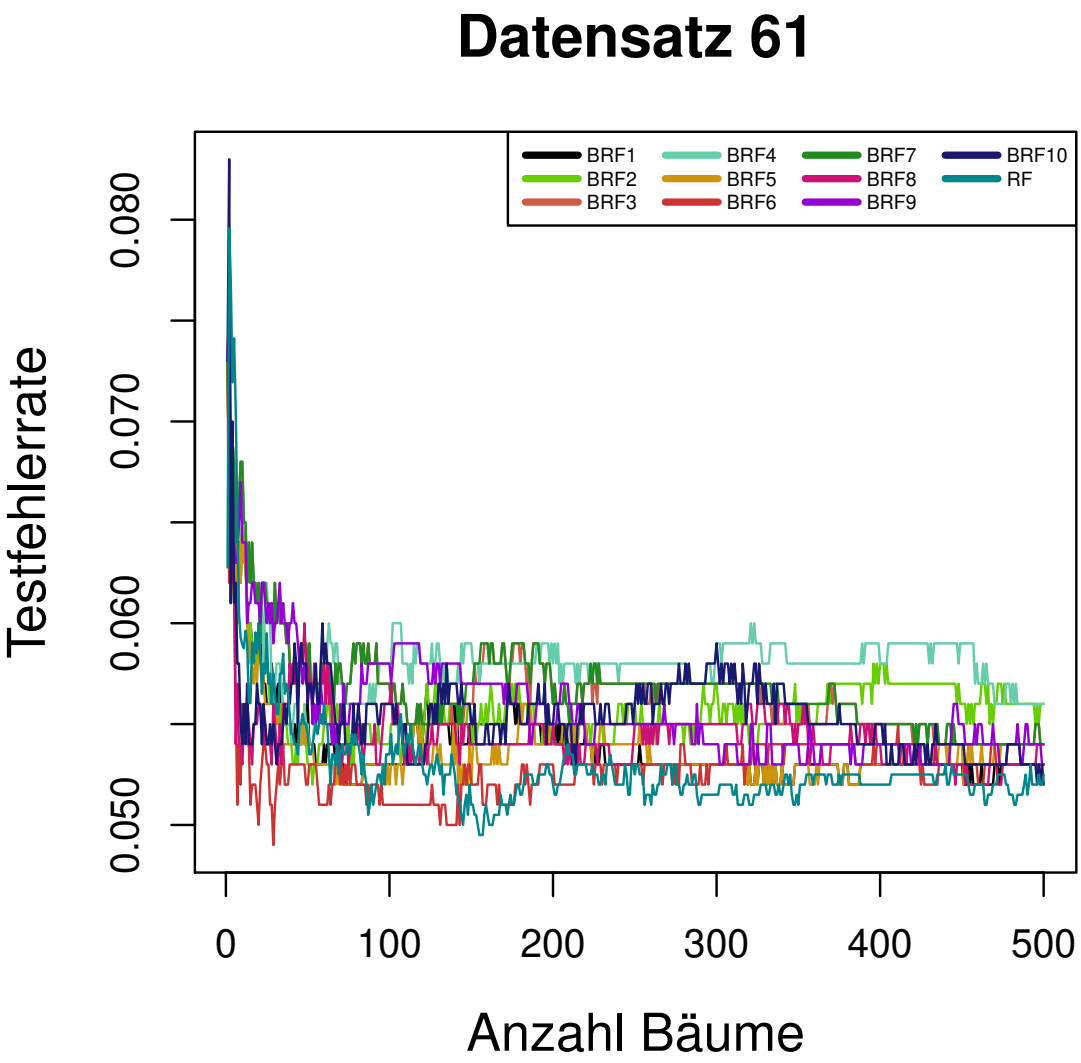

Abbildung 4.1: Gemittelte Testfehlerrate für Datensatz 61 auf Basis von 20 Forests mit je 500 Bäumen.

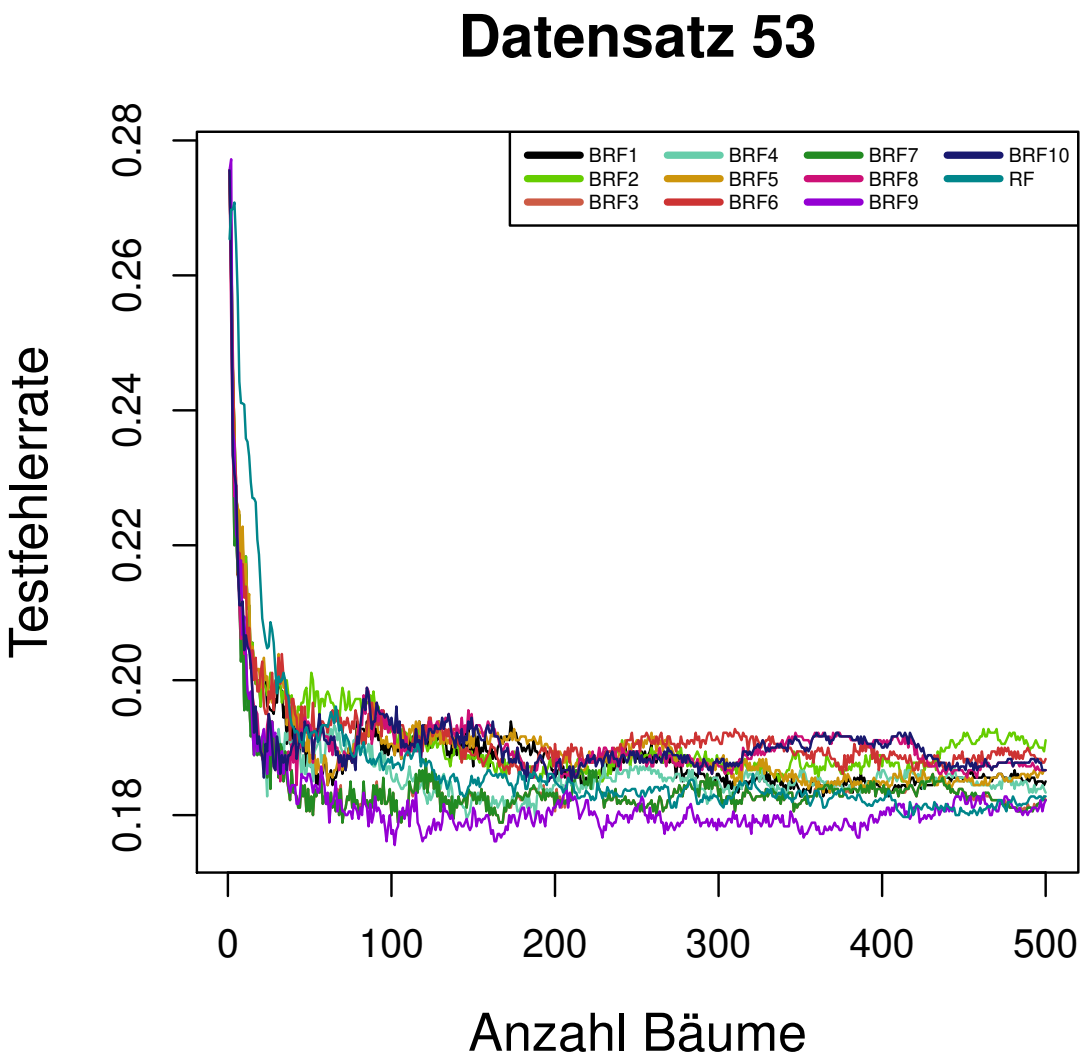

Abbildung 4.2: Gemittelte Testfehlerrate für Datensatz 53 auf Basis von 20 Forests mit je 500 Bäumen.

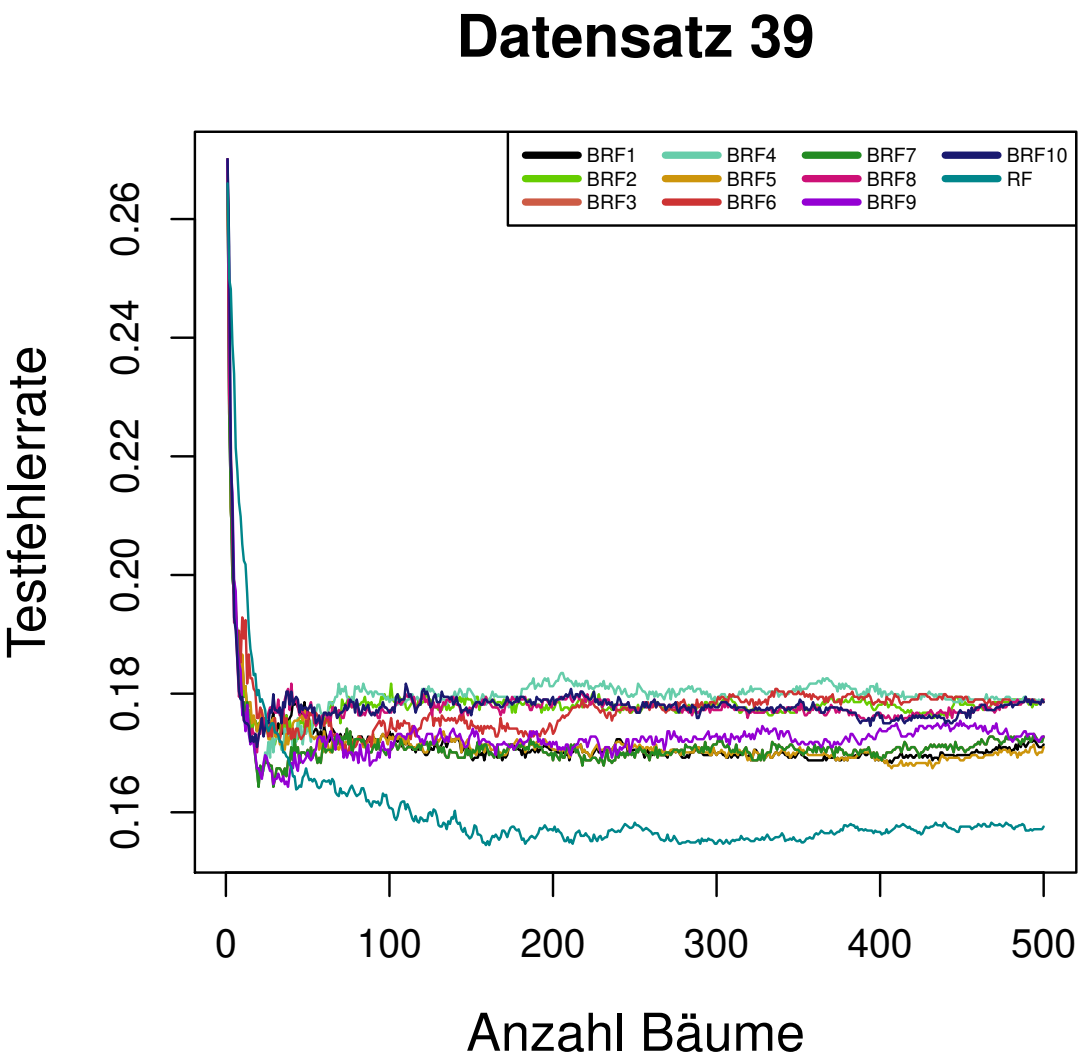

Abbildung 4.3: Gemittelte Testfehlerrate für Datensatz 39 auf Basis von 20 Forests mit je 500 Bäumen.

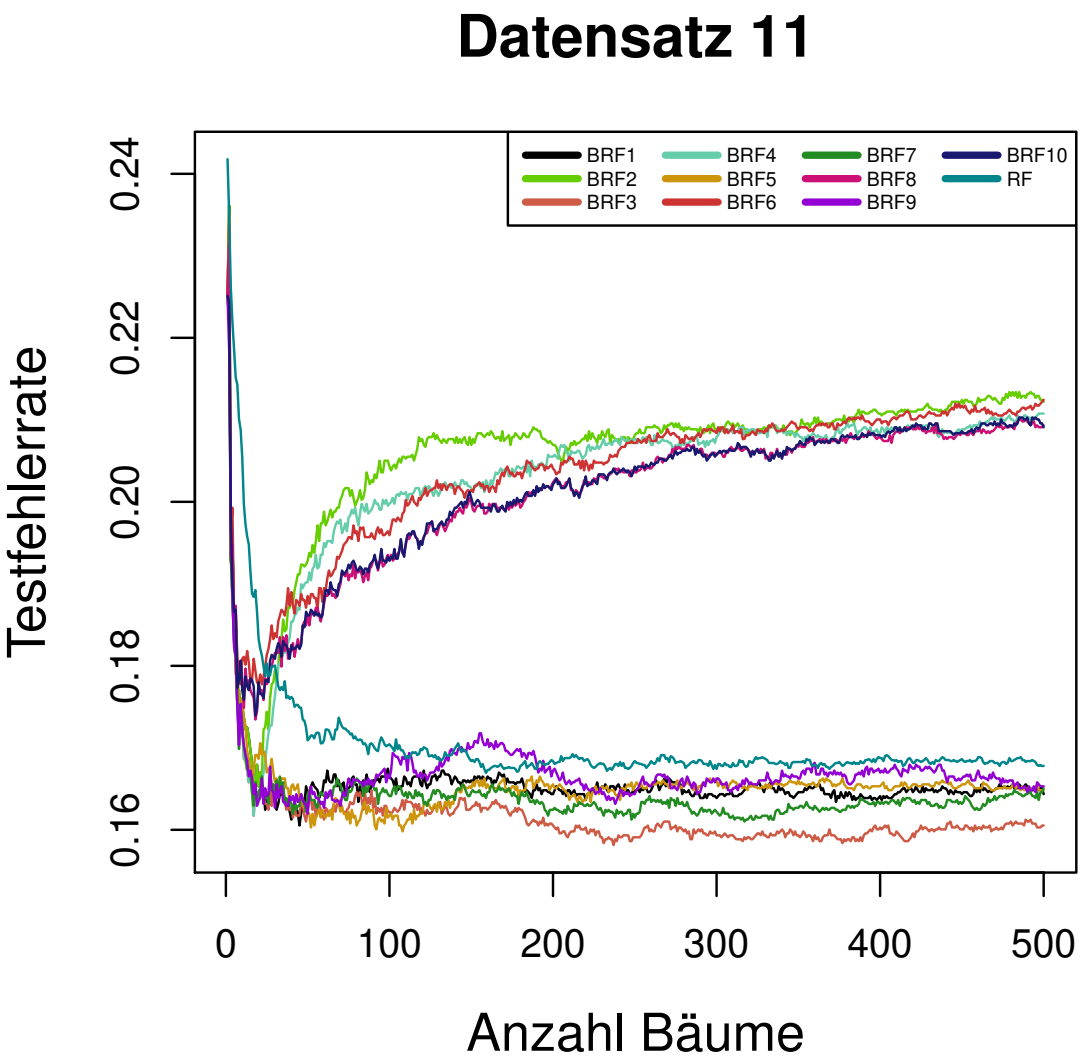

Abbildung 4.4: Gemittelte Testfehlerrate für Datensatz 11 auf Basis von 20 Forests mit je 500 Bäumen.

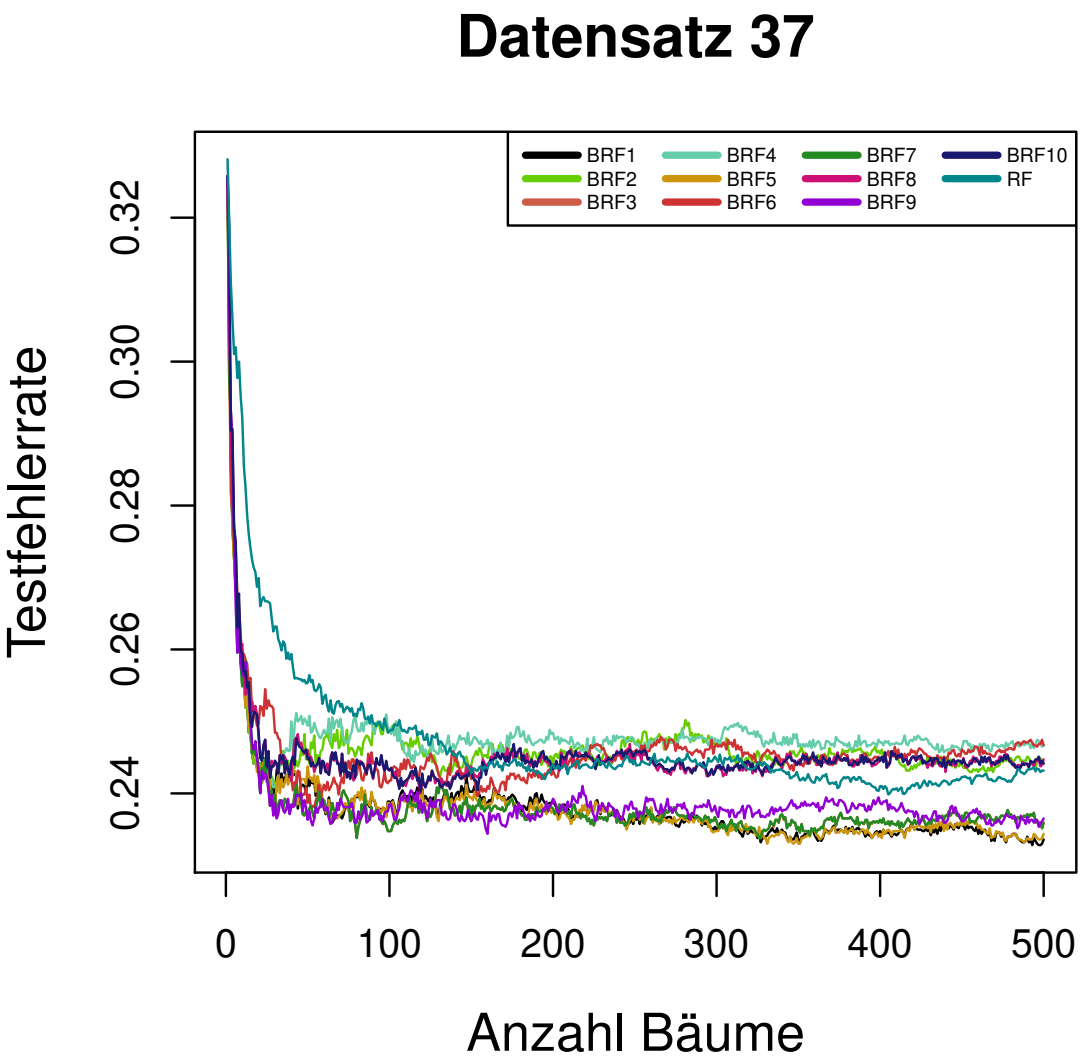

Abbildung 4.5: Gemittelte Testfehlerrate für Datensatz 37 auf Basis von 20 Forests mit je 500 Bäumen.

Bei Betrachtung der Testfehlerraten bei Anwendung der verschiedenen Varianten der Funktion (*BRF1 - BRF10*) an den Datensätzen aus Tabelle 4.1 fällt direkt auf, dass die Ergebnisse sich je Datensatz unterscheiden. Während die Testfehlerraten in Abbildung 4.1, 4.2 und 4.5 relativ nah beieinander liegen, liegt die Testfehlerrate des Standard-RF in Abbildung 4.3 deutlich unter denen der Varianten *BRF1 - BRF10*.

In Abbildung 4.4 steigen fünf der elf Testfehlerraten mit steigender Anzahl an Bäumen, welche für die Vorhersage hinzugezogen werden. Bei diesen fünf Varianten der Funktion handelt es sich um *BRF2*, *BRF4*, *BRF6*, *BRF8* und *BRF10* und somit kam jedes Mal die spezielle Parametereinstellung `sample.weight = T` zur Anwendung. Da diese Beobachtung nur in Abbildung 4.3 zu machen ist, liegt die Vermutung nahe, dass die Struktur des hier verwendeten Datensatzes sich von denen der übrigen Datensätze unterscheidet. Die Vorüberlegungen zur Wirkungsweise des Parameters `sample.weights` aus Abschnitt 3.3 aufgreifend wäre es möglich, dass in diesem Datensatz Ausreißerbeobachtungen vorliegen. Dass solche Ausreißerbeobachtungen für die höhreren Testfehlerraten der entsprechenden Varianten verantwortlich sein können, bestätigt sich durch die Simulationen in Abschnitt 4.2. Dies ist ein wesentlicher Nachteil der Variante mit `sample.weight = TRUE` im Vergleich zum Standard-RF, dessen Vorteil es unter anderem gerade ist, dass er eine sogenannte *off-the-shelf*-Methode ist, also direkt auf Daten angewandt werden kann, ohne zuvor konkrete Annahmen über diese treffen zu müssen. Vielmehr ist somit die Einstellung `sample.weights = TRUE` nur zu verwenden, wenn in den Daten keine oder nur wenige Ausreißer vorkommen. Da die Testfehlerraten dieser Variationen jedoch auch auf den übrigen drei Datensätzen höher sind als der Rest, scheint diese Variante der Gewichtung generell nicht empfehlenswert zu sein. Eine Möglichkeit, die Gewichte dennoch bei Ziehung der Bootstrap-Stichprobe zu verwenden, wäre eine genau umgekehrte Gewichtung der Beobachtungen. „Ausreißer“, also Beobachtungen welche von anderer Struktur wie die restlichen sind, nur einen geringen Anteil der Gesamtdatenmenge ausmachen und falsch klassifiziert werden, würden durch eine geringe Gewichtung vom Training der zukünftigen Bäume ausgeschlossen werden. Diese würden sich somit stärker auf die Mehrheit der Beobachtungen konzentrieren und entsprechend unbekannte Daten besser klassifizieren können. Für Datensätze, in welchen keine oder nur sehr wenige Ausreißer vorhanden sind, hätte dies jedoch eine leichte Verschlechterung der Genauigkeit der Klassifikation zur Folge. Abbildung 4.6 zeigt die Testfehlerraten der Varianten *BRF2* und *BRF4* im Vergleich mit der des Standard-RF unter Verwendung einer umgekehrten Gewichtung. Eine genauere Beschreibung der Graphiken findet sich an entsprechender Stelle.

Im Vergleich mit dem Standard-RF bleibt anzumerken, dass dieser vor allem auf den

Datensätzen 61 und 39 (Abbildung 4.1 und 4.3) niedrigere Testfehlerraten erzielt als die Varianten *BRF1* bis *BRF10*. Der Unterschied ist insbesondere bei Datensatz 39 recht groß. Bei Betrachtung von Tabelle 4.1 fällt auf, dass dieser Datensatz eine verhältnismäßig hohe Klassenanzahl in der Zielvariable besitzt. Dies lässt vermuten, dass die Varianten *BRF1* bis *BRF10* bei solchen Daten Probleme haben, die Beobachtungen richtig zu klassifizieren. Auch diese Überlegungen sollen mit Hilfe einiger simulierter Datensätze in Abschnitt 4.2 überprüft werden. Bei Datensatz 53 (Abbildung 4.2) liegt die Testfehlerraten des Standard-RF nur minimal unter denen der Varianten *BRF1* bis *BRF10*, bei den Datensätzen 11 und 37 (Abbildung 4.4 und 4.5) hingegen sind die Testfehlerraten der Varianten *BRF1, BRF3, BRF5, BRF7* und *BRF9* leicht niedriger als die des Standard-RF. Hier fällt auf, dass dies alles Varianten sind, bei welchen die Einstellung `sample.weights = FALSE` gewählt wurde.

In Abbildung 4.6 sind die Testfehlerraten der Varianten *BRF2* und *BRF4* jeweils im Vergleich mit der des Standard-RF dargestellt. Wie erwartet sind die Ergebnisse genau entgegengesetzt zu denen aus den Berechnungen mit der Gewichtsfunktion, bei welcher die falsch klassifizierten Beobachtungen ein hohes Gewicht erhalten. Genauer heißt das, dass die Testfehlerraten für den Datensatz 61 nun oberhalb der des Standard-RF liegen, wohingegen sie bei stärkerer Gewichtung der „Ausreißer“ ähnlich ausgefallen sind. Bei Strukturen wie der des Datensatzes 61 erscheint eine entgegengesetzte Gewichtung somit wenig sinnvoll. Ein anderes Bild zeichnet sich bei Datensätzen 53, 11 und 37. Insbesondere die Testfehlerraten aus den Berechnungen aus Datensatz 11 liegen dieses Mal deutlich unter der des Standard-RF und auch unter denen aller anderen Varianten (vgl. Abbildung 4.4). Was auch immer dafür verantwortlich ist, dass sich Testfehlerraten wie hier ergeben: Fest steht, dass, sobald bekannt ist, bei welcher Art von Datenstruktur die Funktion mit der Einstellung `sample.weights = TRUE` wie reagiert, zwar diese Art der Gewichtung nicht mehr „blind“ auf die Daten angewandt werden kann (einer der Vorteile des RF!). Jedoch könnte die Genauigkeit des Waldes durch die entsprechend angebrachte Gewichtung wesentlich verbessert werden können.

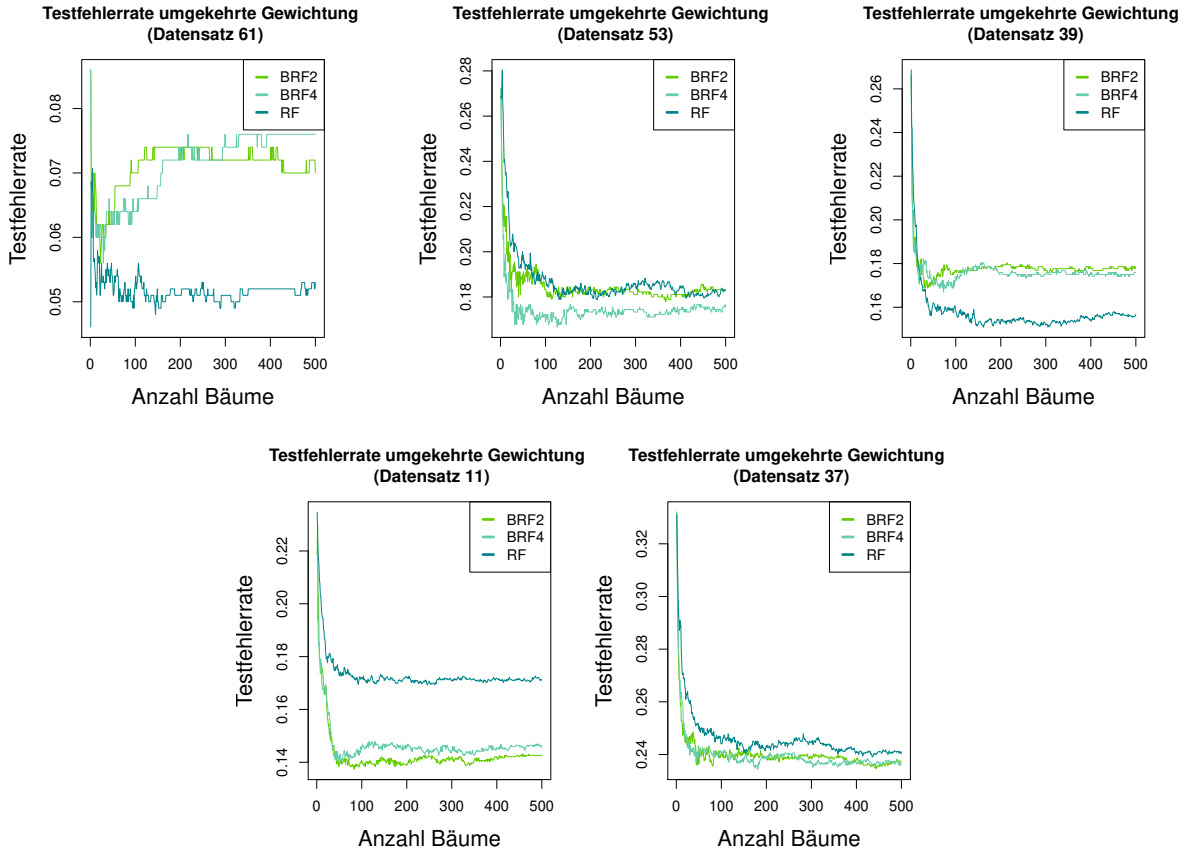

Abbildung 4.6: Gemittelte Testfehlerrate für die Datensätze 61, 53, 39, 11 und 37 bei umgekehrter Gewichtsfunktion auf Basis von zehn Forests mit je 500 Bäumen.

Während die Einstellung des Parameters `sample.weights` einen stärkeren Einfluss auf die Performance des Waldes zu haben scheint, macht dies für die Parameter `method` und `vote.type` nicht den Eindruck.

In Tabelle 4.3 sind die Ergebnisse der 5-fachen Kreuzvalidierung der Funktion auf den 11 Datensätzen aus Tabelle 4.1 aufgeführt.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.049	0.052	0.048	0.049	0.049	0.049	0.050	0.050	0.049	0.053	0.046
2	0.177	0.194	0.177	0.196	0.178	0.196	0.178	0.191	0.182	0.192	0.177
3	0.155	0.166	0.157	0.166	0.156	0.164	0.158	0.160	0.156	0.159	0.155
4	0.174	0.235	0.170	0.227	0.175	0.228	0.175	0.226	0.177	0.226	0.166
5	0.241	0.248	0.241	0.252	0.242	0.246	0.241	0.249	0.243	0.249	0.240
6	0.034	0.028	0.034	0.028	0.034	0.028	0.034	0.030	0.034	0.030	0.033
7	0.170	0.165	0.172	0.166	0.170	0.164	0.172	0.165	0.171	0.165	0.171
8	0.038	0.034	0.040	0.033	0.038	0.032	0.040	0.034	0.040	0.034	0.037
9	0.301	0.308	0.301	0.307	0.301	0.307	0.302	0.304	0.302	0.305	0.302
10	0.027	0.023	0.027	0.023	0.027	0.024	0.027	0.024	0.028	0.024	0.027
11	0.223	0.212	0.224	0.213	0.223	0.211	0.224	0.212	0.224	0.212	0.222

Tabelle 4.3: Testfehlerraten aus 5-facher Kreuzvalidierung für jede Variante der Funktion und den Standard-RF, rot: die niedrigste Testfehlerrate je Datensatz.

Bei Betrachtung der Testfehlerraten in Tabelle 4.3 fällt auf, dass es zwischen den verschiedenen Varianten fast keine Unterschiede gibt. Im Vergleich mit den Ergebnissen aus Abbildung 4.1 bis 4.5 ist nun immer die Testfehlerrate des Standard-RF die niedrigste. Wie auch zuvor fällt hier auf, dass es einzig bei Datensatz 11 größere Abweichungen der Testfehlerraten voneinander gibt. Bei denjenigen Varianten, bei welchen die Gewichte als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe genutzt wurden, fallen die Testfehlerraten (im Vergleich zu den übrigen Datensätzen) deutlich höher aus als bei den restlichen Varianten sowie dem Standard-RF. Für die weiteren Datensätze, welche mehr Beobachtungen aufweisen hat die Variante *BRF6* am häufigsten die niedrigste Testfehlerrate. Alles in allem sind die Unterschiede wie bereits erwähnt jedoch nicht nennenswert und auf Grundlage dieser Ergebnisse scheint somit keine Verbesserung des Algorithmus durch eine Anwendung von Gewichten wie sie hier implementiert wurde erreicht worden zu sein.

4.2 Simulationsstudie

In diesem Abschnitt soll eine kleine Simulationsstudie durchgeführt werden. Hierzu werden sechs verschiedene Simulationsszenarien angewandt, welche an entsprechender Stelle separat vorgestellt werden. Innerhalb der Simulationsstudie sollen die bereits in Abschnitt 4.1 betrachteten Varianten der Funktion miteinander und zusätzlich mit dem Standard-RF verglichen werden. Anders als im vorherigen Kapitel ist nun jedoch bekannt, ob in den Daten Ausreißer vorliegen, wodurch explizit die These der Ausreißerempfindlichkeit bei Anwendung der Parametereinstellung `sample.weights = TRUE` überprüft werden kann. Ebenso kann die Verhaltensweise der Funktion mit anderen Parametereigenschaften bei Daten mit und ohne Ausreißer sowie zunehmender Klassenanzahl im Response beobachtet werden. Da die Ausführung der Funktion noch sehr rechenintensiv ist, wird die Anzahl der Iterationen hier entsprechend niedrig gehalten. Auch diese wird bei Vorstellung des jeweiligen Szenarios aufgeführt. Zur Datensimulation wurden zwei Funktionen implementiert, mit welchen die Daten mit den dem Simulationsszenario entsprechenden Einstellungen

erzeugt werden. Hierbei handelt es sich zum einen um die Funktion `createBinData` und zum anderen um die Funktion `createMultData`. Wie die Namen vermuten lassen, wird mit erstgenannter ein Datensatz mit binärem, mit zweitgenannter ein Datensatz mit multikategorialem Response erzeugt. Beide Funktionen sind sowohl in Abschnitt A.1 als auch auf der beigefügten DVD zu finden. Aus Reproduzierbarkeitsgründen wird ein entsprechender *Seed* gesetzt.

4.2.1 Aufbau

Für jede der Simulationen werden zunächst mit Hilfe der entsprechenden Funktion (binärer Response: `createBinData`, multikategorialer Response: `createMultData`) zehn verschiedene Datensätze unter Verwendung des jeweiligen Simulationsszenarios erstellt. Mit Hilfe des `R`-Pakets `mlr` (Bischl et al. (2016)) werden die elf bereits kennengelernten Varianten des RF (*BRF1 - BRF10, RF*) zweifach kreuzvalidiert. Die Anzahl der Wiederholungen liegt bei fünf. Über diese so erfolgenden insgesamt zehn Durchläufe (2 Folds, 5 Wiederholungen) wird der mittlere Testfehler berechnet. Die sich ergebenden Testfehlerraten der zehn Iterationen werden mit Hilfe von Boxplots dargestellt. Anders als bei der Berechnung der Testfehlerrate aus Abschnitt 4.1, bei welcher jeweils 500 Bäume je Wald erstellt wurden, wird hier für den Standard-RF eine Waldgröße von `ntree = 5000` Bäumen vorgegeben, um sicherzustellen, dass der Wald seine maximale Genauigkeit erreicht. Für die Varianten *BRF1 - BRF10* wird nun das in Unterabschnitt 3.3.1 vorgestellte Konvergenzkriterium mit den Parametern `conv.threshold = 0.001` und `smoothness = 200` verwendet. Um sicherzustellen, dass durch diese Werte adäquate Einstellungen vorgenommen werden, wurde zuvor an einigen Beispieldatensätzen getestet, bei welcher Anzahl an erstellten Bäumen der Algorithmus mit genau diesen Werten das Training des Waldes beendet hätte (Parametereinstellung `stoptreeOut = TRUE`). Dabei stellte sich heraus, dass der Algorithmus immer erst abbricht, nachdem sich die OOB-Fehlerrate schon stabilisiert hat.

Da für die Datensimulation auf das *binäre* bzw. *multinomiale Logit-Modell* zurückgegriffen wird, sollen diese zwei Modelle im Folgenden kurz erläutert werden.

Das binäre Logit-Modell

Das binäre Logit-Modell wird verwendet, wenn der Zusammenhang zwischen den p Prädiktorvariablen $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$ einer Beobachtung i und ihrem binären Response y_i modelliert werden soll. Genauer wird hierbei die Wahrscheinlichlichkeit modelliert, dass eine Beobachtung i, gegeben ihren Prädiktorvariablen $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})$, Klasse 1 angehört anstatt Klasse 0. Die grundlegende Darstellung dieser Wahrscheinlichkeit lautet

$$\pi(\boldsymbol{x}) = F(\eta(\boldsymbol{x})). \tag{4.1}$$

Innerhalb des Logit-Modells hat die Verteilungsfunktion $F(\cdot)$ die Form

$$F(\eta) = \frac{\exp(\boldsymbol{x}^T\boldsymbol{\beta})}{1 + \exp(\boldsymbol{x}^T\boldsymbol{\beta})} \tag{4.2}$$

mit dem linearen Prädiktor $\eta(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{\beta}$. Weiter sind die folgenden Darstellungen äquivalent zueinander:

$$\pi(\boldsymbol{x}) = F(\eta(\boldsymbol{x})), \tag{4.3}$$

$$\frac{\pi(\boldsymbol{x})}{1 - \pi(\boldsymbol{x})} = \exp(\boldsymbol{x}^T\boldsymbol{\beta}), \tag{4.4}$$

und

$$\log\left(\frac{\pi(\boldsymbol{x})}{1 - \pi(\boldsymbol{x})}\right) = \boldsymbol{x}^T\boldsymbol{\beta}. \tag{4.5}$$

[vgl. Tutz (2012), S.37]

Das multinomiale Logit-Modell

Während der Response Y beim Logit-Modell binär ist, d.h. nur zwei mögliche Klassen annehmen kann, gilt für das multinomiale Logit-Modell $Y \in \{1, \ldots, k\}$. Der Response besteht demnach aus den Klassen $1, \ldots, k$. Sei $\boldsymbol{x}$ wieder ein Vektor der Prädiktorvariablen. Für jede Klasse r und Referenzkategorie k gilt für die logarithmierte Chance,

dass die Beobachtung aus Klasse r anstatt aus Klasse k ist,

$$\log\left(\frac{P(Y=r|\boldsymbol{x})}{P(Y=k|\boldsymbol{x})}\right) = \boldsymbol{x}^T\boldsymbol{\beta}_r,\ r=1,\ldots,q. \tag{4.6}$$

Diese ergibt sich somit aus den Ausprägungen der Prädiktorvariablen $\boldsymbol{x}$ und dem für Klasse r spezifischen Koeffizientenvektor $\boldsymbol{\beta}_r$. Durch einfaches Umstellen von Gleichung 4.6 und der Verwendung aller $q = k-1$ Logits $\log(P(Y=1|\boldsymbol{x})/P(Y=k|\boldsymbol{x}), \ldots, \log(P(Y=q|\boldsymbol{x})/P(Y=k|\boldsymbol{x})$ ergibt sich die Wahrscheinlichkeit, dass die Beobachtung zur Referenzkategorie k gehört, gegeben der Prädiktorvariablen $\boldsymbol{x}$, als

$$P(Y=k|\boldsymbol{x}) = \frac{1}{1+\sum_{r=1}^{k-1}\exp(\boldsymbol{x}^T\boldsymbol{\beta}_r)}. \tag{4.7}$$

Diese eingesetzt in Gleichung 4.6 ergibt die Wahrscheinlichkeit für Klasse r gegeben $\boldsymbol{x}$ und es gilt

$$P(Y=r|\boldsymbol{x}) = \frac{\exp(\boldsymbol{x}^T\boldsymbol{\beta}_r)}{1+\sum_{s=1}^{k-1}\exp(\boldsymbol{x}^T\boldsymbol{\beta}_s)}. \tag{4.8}$$

[vgl. Tutz (2012), S.210 f.]

Szenario 1

Die Anzahl der simulierten Datensätze beträgt $S = 10$. Dazu werden für 200 Beobachtungen fünf stetige, normalverteilte Prädiktorvariablen aus einer $N(0,1)$-Verteilung erzeugt. Anhand der daraus resultierenden Designmatrix $\boldsymbol{X}_{200x5}$, des über alle Datensätze $S = 1,\ldots,10$ gleichbleibenden Koeffizientenvektors $\boldsymbol{\beta}_1 = (5, 8, 3.5, 6.8, 7)$ und des Logit-Modells wird ein binärer Response $\boldsymbol{y}$ generiert. Für die Zuweisung der Klassenzugehörigkeit y_i einer Beobachtung i gilt:

$$y_i = \begin{cases} 1, & \text{falls } P(Y_i=1|\boldsymbol{X},\beta) = \frac{\exp(\boldsymbol{X}\beta)}{1+\exp(\boldsymbol{X}\beta)} \geq 0.5 \\ 0, & \text{sonst} \end{cases},\ i=1,\ldots,N.$$

Weiter sollen keine Ausreißer in den Daten vorkommen.

Das Szenario in Stichpunkten:

- Anzahl der simulierten Datensätze: $S = 10$
- binärer Response, generiert anhand eines binären Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0,1)$, $i = 1, \ldots, 5$
- 0% Ausreißer
- Koeffizientenvektor $\boldsymbol{\beta} = (5, 8, 3.5, 6.8, 7)$

Szenario 2

Wie in Szenario 1 werden auch hier zehn Datensätze erzeugt. Die Designmatrix $\boldsymbol{X}$ besteht dabei wieder aus fünf $N(0,1)$-verteilten Prädiktorvariablen, der Responsevektor $\boldsymbol{y}$ wird über das Logit-Modell bestimmt. Der Koeffizientenvektor $\boldsymbol{\beta}$ ist identisch zu dem aus Szenario 1. Bei fünf Prozent der Beobachtungen soll es sich nun jedoch um Ausreißer handeln, das heißt, dass es bei entsprechenden Beobachtungen i keinen Zusammenhang zwischen den Prädiktorvariablen $\boldsymbol{x}_i$, und dem Response y_i gibt. Hierzu werden 5% der Beobachtungen aus Klasse 0 der Klasse 1 zugeorndet und umgekehrt.

- Anzahl der simulierten Datensätze: 10
- binärer Response, generiert anhand eines binären Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0,1)$, $i = 1, \ldots, 5$
- 5% Ausreißer
- Koeffizientenvektor $\boldsymbol{\beta} = (5, 8, 3.5, 6.8, 7)$

Szenario 3

Die Prädiktorvariablen, der Response und der Koeffizientenvektor werden wie in den zwei vorangegangenen Szenarien simuliert. Der einzige Unterschied liegt im Anteil der Ausreißer. Dieser beträgt hier 30%.

- Anzahl der simulierten Datensätze: 10
- binärer Response, generiert anhand eines binären Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0,1)$, $i = 1,\ldots,5$
- 30% Ausreißer
- Koeffizientenvektor $\boldsymbol{\beta} = (5, 8, 3.5, 6.8, 7)$

Szenario 4

Das vierte Szenario simuliert einen multikategorialen Response mit fünf Kategorien $k = 1,\ldots,5$ anhand des multinomialen Logit-Modells. Hierzu werden für 200 Beobachtungen fünf normalverteilte, und somit stetige, Prädiktorvariablen $X_i \sim N(0,1)$, $i = 1,\ldots,5$ erzeugt.

- Anzahl der simulierten Datensätze: 10
- multinomialer Response (5 Kategorien), generiert anhand eines multinomialen Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0,1)$, $i = 1,\ldots,5$
- $\boldsymbol{X} \sim N(\mathbf{0}, \mathbf{1})$
- Koeffizientenvektoren:
  - $\boldsymbol{\beta}_1 = (0, 0, 0, 0, 0)$ (Referenzkategorie)
  - $\boldsymbol{\beta}_2 = (-4, 5, 7, 6, 3)$
  - $\boldsymbol{\beta}_3 = (5, 2, -6, 4, 3)$

- $\beta_4 = (4, 7, 3, 8, -2)$
- $\beta_5 = (-6, 9, 2, 6, 1)$
- $\beta_6 = (8, 4, -2, 2, 7)$
- $\beta_7 = (3, 1, 3, 1, -4)$
- $\beta_8 = (-2, 4, 6, 1, 4)$
- $\beta_9 = (1, -2, 7, 3, 1)$
- $\beta_1 0 = (4, -7, 2, -4, 5)$

Szenario 5

Das fünfte Szenario ist identisch mit dem vierten Szenario, mit Ausnahme der Anzahl der Responsekategorien, welche hier bei sieben liegt.

- Anzahl der simulierten Datensätze: 10
- multinomialer Response (7 Kategorien), generiert anhand eines multinomialen Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0, 1)$, $i = 1, \ldots, 5$
- $\boldsymbol{X} \sim N(\mathbf{0}, \mathbf{1})$
- Koeffizientenvektoren: wie Szenario 4.

Szenario 6

Die Daten werden simuliert wie in Szenario 5. Der Response nimmt zehn verschiedene Kategorien an.

- Anzahl der simulierten Datensätze: 10
- multinomialer Response (10 Kategorien), generiert anhand eines multinomialen Logit-Modells
- fünf stetige, normalverteilte Prädiktorvariablen: $X_i \sim N(0,1)$, $i = 1, \ldots, 5$
- $\boldsymbol{X} \sim N(\mathbf{0}, \mathbf{1})$
- Koeffizientenvektoren: wie Szenario 4.

4.2.2 Ergebnisse

Es werden zunächst die Ergebnisse der Szenarien $1 - 3$ betrachtet. Dazu erfolgt die Darstellung der Testfehlerraten, erhalten aus den zehn simulierten Datensätze, anhand eines Boxplots in Abbildung 4.7 bis 4.13. Ein genauer Überblick über die einzelnen Testfehlerraten findet sich in den Tabellen in Abschnitt A.2.

Abbildung 4.7 zeigt die Ergebnisse aus Szenario 1. Auffällig ist, dass die Werte der Varianten *BRF3 und BRF7* im Gegensatz zu den übrigen nicht nach oben streuen. Ihr Maximum (0.2150) liegt knapp oberhalb des Medians (0.1840). Darüber hinaus sind alle Werte dieser beiden Varianten identisch. Alle Varianten betrachtend liegen die Mittelwerte der Testfehlerraten im Intervall $[0.1592, 0.1806]$ und der betrachtete Wertebereich ist somit sehr klein. Ebenso sind die Werte der Varianten *BRF2, 4, 6, 8* und *10* etwas niedriger. Die mittlere Testfehlerrate des Standard-RF liegt hier mit einem Wert von 0.1800 nur minimal unter denen der Varianten *BRF1, 3, 5* und *7* und im Umkehrschluss oberhalb jener der Varianten *BRF2, 4, 6* und *8*. Die niedrigste Testfehlerrate erzielt dabei die Variante *BRF4* mit einem durchschnittlichen Wert von 0.1592, die höchsten sind die der Varianten *BRF3* und *BRF7* mit einem durchschnittlichen Wert von jeweils 0.1806. Zusammenfassend lässt sich feststellen, dass

alle Varianten, in welchen die Parametereinstellung `sample.weight = TRUE` gewählt wurde, geringere Testfehler verursachen als der Standard-RF. Jene Varianten, bei welchen die Gewichte keinen Einfluss auf die Ziehung der Bootstrap-Stichprobe hatten, unterscheiden sich in ihrem Ergebnis nur minimal von dem des Standard-RF. Unterteilt man nun die Varianten *BRF2*, *4*, *6* und *8* noch einmal nach der Art, wie die Gewichtung durchgeführt wurde (d.h. `method = bernard` bzw. `method = mishina`), so fällt auf, dass jene mit einer Gewichtung, wie sie in Bernard et al. (2012) vorgeschlagen wird, noch einmal bessere Prognosen stellen als solche mit einer Gewichtung wie sie Mishina et al. (2014) vorschlägt. An dieser Stelle sei noch einmal in Erinnerung gerufen, dass diese beiden Arten der Gewichtung sich insbesondere dahingehend unterscheiden, dass bei Bernard et al. (2012) alle bereits angepassten Bäume für die Berechnung der neuen Gewichte involviert werden, wohingegen bei Mishina et al. (2014) nur der jeweils aktuelle Baum Einfluss auf die verwendete Gewichtsfunktion nimmt. Bei alleiniger Betrachtung der Ergebnisse aus Szenario 1 scheint die retrospektive Variante die Anpassung der Bäume besser regulieren zu können. Eine weitere, interessante Beobachtung ist, dass einige der mittleren Testfehlerraten identisch sind. Dies sind die Varianten *BRF1* und *BRF5* und (wie bereits erwähnt) *BRF3* und *BRF7*. Sie unterscheiden sich jeweils nur in der Art der Gewichtung. Diese scheint bei Verwendung der Parametereinstellung `sample.weights = FALSE` und indentischer Wahl des `vote.type`'s keinen Unterschied zu machen. Die Wälder, bei welchen der MV mit `vote.type = soft` bestimmt wurde, weisen jedoch eine minimal geringere Testfehlerrate auf als jene, bei welchen `vote.type = hard` verwendet wurde. Eine Gewichtung der einzelnen Bäume (`treeWeights = TRUE`, *BRF10*) scheint dabei den Rückstand auf die retrospektive Gewichtung nicht kompensieren zu können.

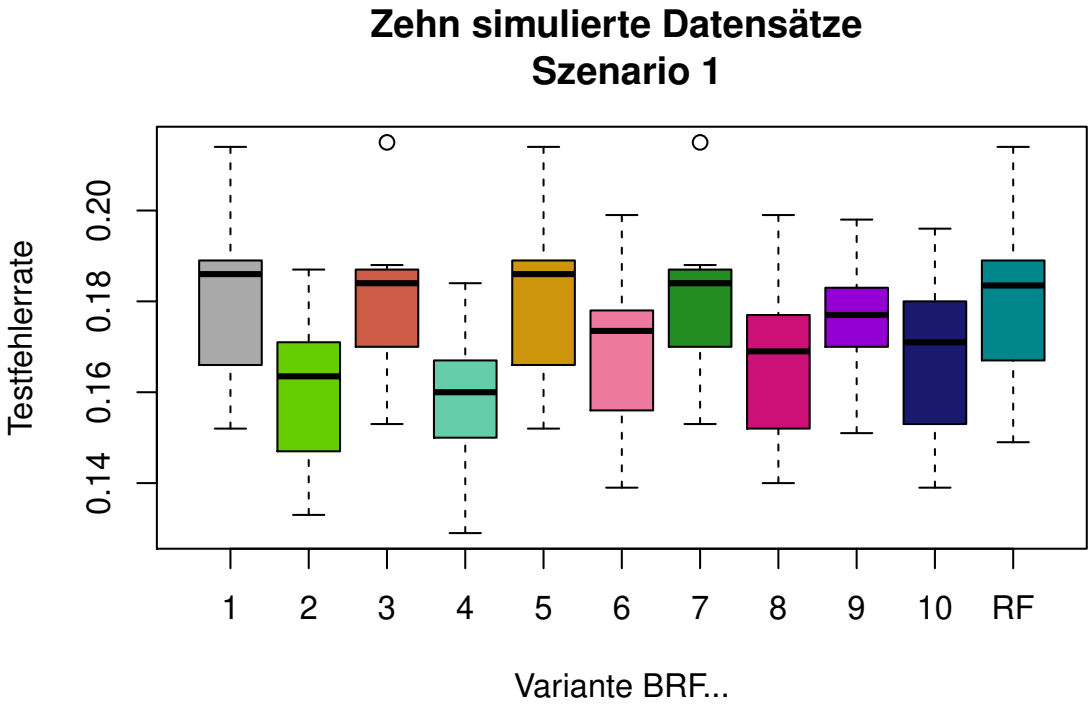

Abbildung 4.7: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Für das Resümee der Ergebnisse aus Szenario 2 wird nun Abbildung 4.8 betrachtet. Hier liegen die Mittelwerte der Testfehlerraten im Intervall $[0.1894, 0.2018]$ und sind somit generell höher als jene aus Szenario 1. Der betrachete Wertebereich fällt diesmal etwas größer aus. Die insgesamt niedrigsten Testfehlerraten erzielt hier die Variante *BRF2* (Median ist 0.1815), die höchsten *BRF3* und *BRF7* (Mediane jeweils 0.1950). Auch hier fallen die Ergebnisse für die Varianten, in welchen die Gewichte einen Einfluss auf die Ziehung der Bootstrap-Stichprobe haben besser aus als für jene, bei welchen die Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe aus einer Gleichverteilung stammen. Der Standard-RF gehört mit einer mittleren Testfehlerrate von 0.1997 und einem Median von 0.1955 zu den sechs Varianten, welche höhere Testfehlerraten erzielen. Aus dem Vergleich der zwei Gewichtsfunktionen geht erneut

jene nach Bernard et al. (2012) als die stärkere hervor.

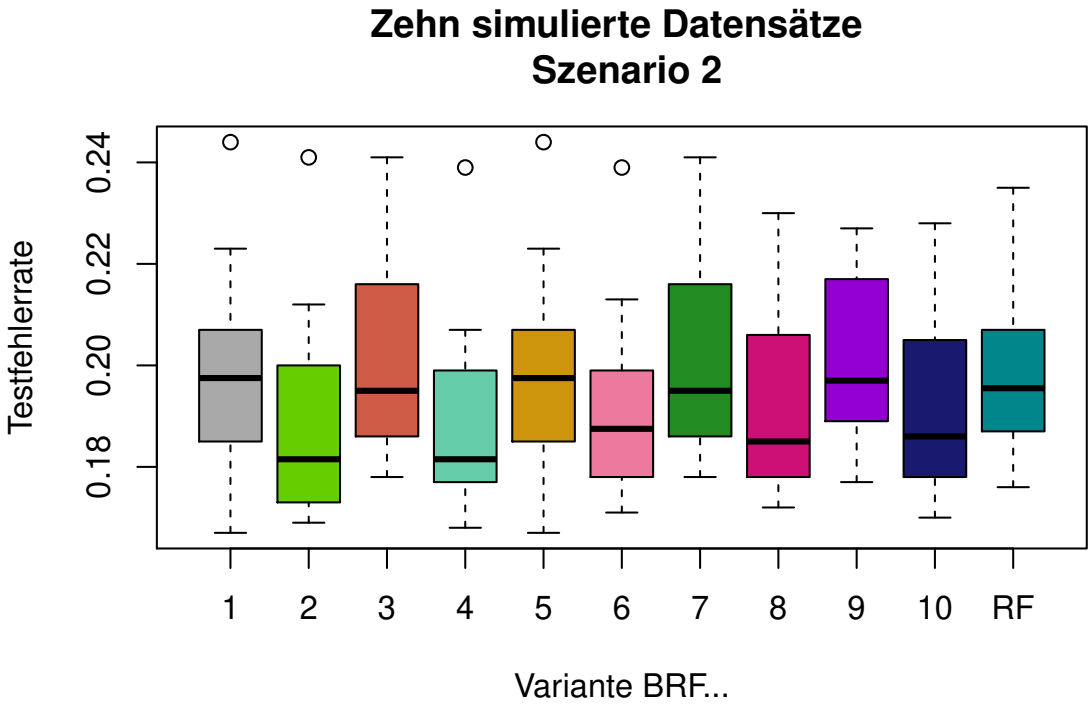

Abbildung 4.8: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Für die Daten, welche nach dem Szenario 3 simuliert wurden, zeichnet sich in Abbildung 4.9 ein anderes Bild. Hier liegen die Testfehlerraten im Intervall $[0.4085, 0.4233]$ und somit wesentlich über jenen aus den Szenarien 1 und 2. Während die Variante *BRF2* in den vorausgegangen Szenarien noch Testfehlerraten erzielte, welche (deutlich) unter denen der anderen Varianten lagen, so ist ihre Testfehlerrate mit einem Wert von 0.4233 nun die höchste. Generell sind alle Varianten, bei welchen die Gewichte als Wahrscheinlichkeiten für die Ziehung der Bootstrap-Stichprobe verwendet wurden, schlechter als jene, bei welchen dies nicht der Fall war. Jene Varianten, bei welchen der MV anhand der Summe der Klassenwahrscheinlichkeiten über alle Bäume bestimmt wurde (`vote.type = soft`), liefern im direkten Vergleich

mit jenen, bei welchen `vote.type = hard` verwendet wird und alle verbleibenden Einstellungen identisch sind immer besser Ergebnisse (das heißt, ihre Testfehlerraten sind geringer).

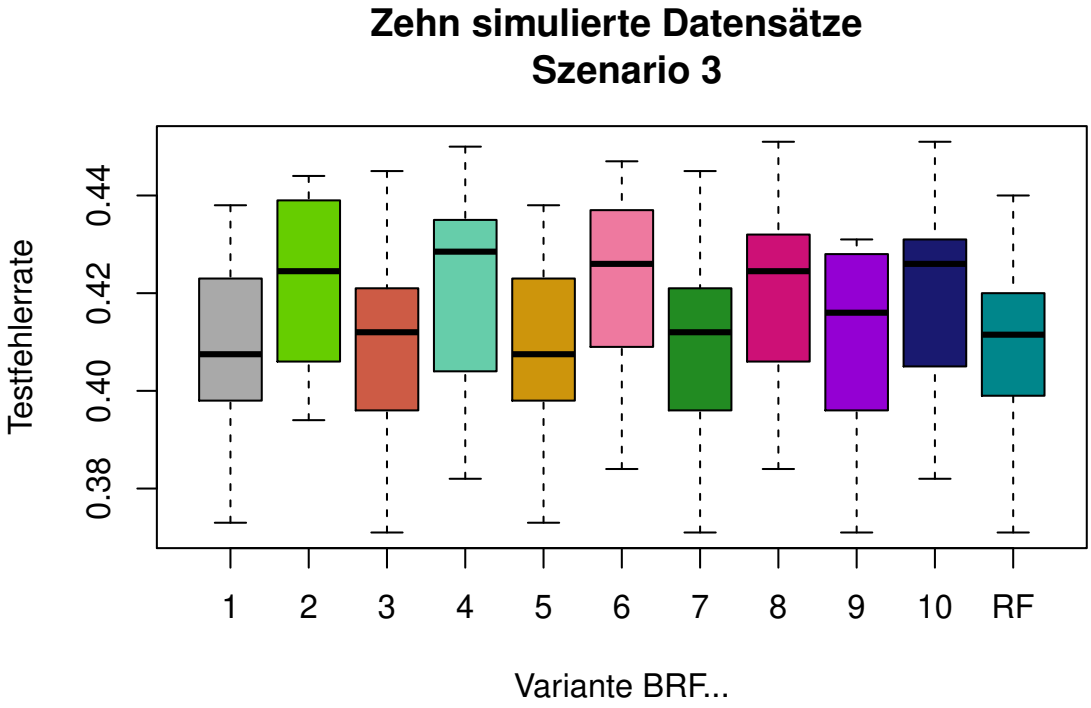

Abbildung 4.9: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Die Ergebnisse der Szenarien $1 - 3$ zusammenfassend lässt sich feststellen, dass die Testfehlerraten mit zunehmendem Ausreißeranteil in den Daten größer werden. Während es bei geringem Ausreißeranteil geeignet erscheint, die Gewichte der Beobachtungen als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe zu nutzen, sprechen die Ergebnisse aus Szenario 3 bei einem hohen Ausreißeranteil eher gegen einen solchen Einsatz der Gewichte. Weiter scheint es, als sei die Parametereinstellung `vote.type = soft` der Einstellung `vote.type = hard` vorzuziehen. Die in Abschnitt 4.1 aufgestellte These, dass eine Verwendung der Gewichte zur Ziehung

der Bootstrap-Stichprobe bei Ausreißern in den Daten nicht empfehlenswert sein könnte, scheint sich somit zu bestätigen.

Anhand der Tabellen in Abschnitt A.2, welche die jeweiligen Testfehlerraten der einzelnen Simulationsdurchläufe darstellen, ergibt sich Abbildung 4.10. Hier ist abgetragen, wie oft die jeweilige Variante über alle Simulationsdurchläufe der Szenarien 1 bis 3 die niedrigste Testfehlerrate erzielt hat. Identische Testfehlerraten werden entsprechend berücksichtigt. Deutlich sticht hier die Variante *BRF4* hervor, welche 12 mal die niedrigste Testfehlerrate erzielt, gefolgt von *BRF2*. Für beide Varianten wurden die Gewichte als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe benutzt und die Parametereinstellungen `vote.type = soft` sowie `method = bernard` gewählt.

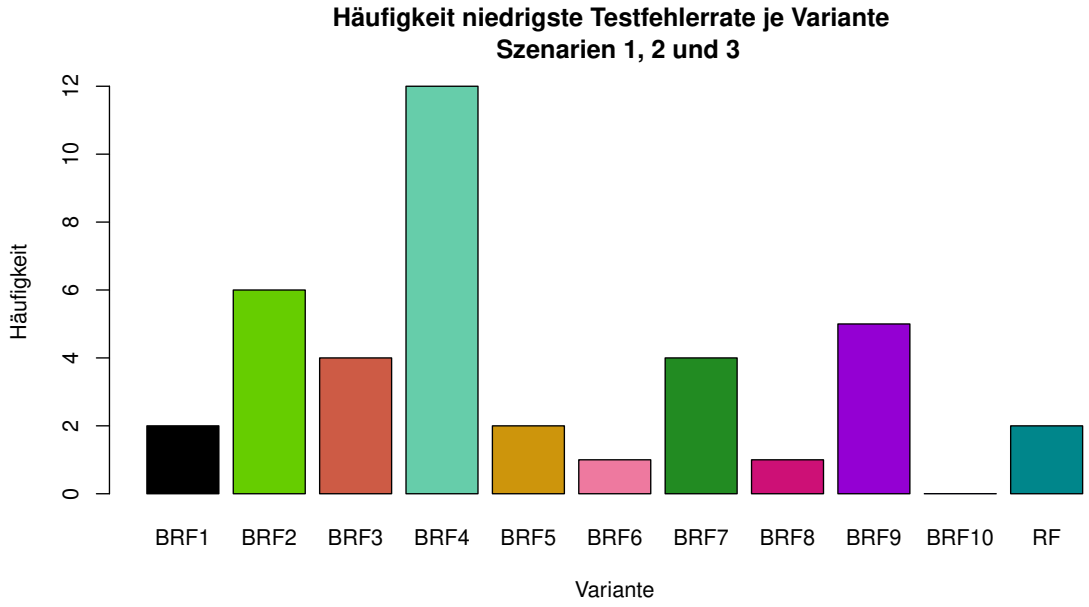

Abbildung 4.10: Balkendiagramm, welches die Häufigkeiten anzeigt, wie oft eine Variante die niedrigste Testfehlerrate erzielt hat (Szenario 1 bis 3).

Eine weitere Überlegung aufgrund der Betrachtung der Testfehlerraten in Abschnitt 4.1 war, dass bei Ausreißervorkommen in den Daten eine umgekehrte Gewichtung der Beobachtungen empfehlenswert sein könnte, da diese auf diese Weise keinen Ein-

fluss auf das Training der Bäume haben würden und diese folglich die Mehrheit der Beobachtungen, welche keine Ausreißer sind, noch besser klassifizieren können. Um diese Überlegung zu überprüfen, wurden die Simulationen $1 - 3$ für die Varianten mit einer Gewichtsberechnung nach Bernard et al. (2012) erneut ausgeführt, dieses Mal jedoch mit entsprechend entgegengesetzter Gewichtung (Parametereinstellung `invertWeights = TRUE`). Das heißt, dass Beobachtungen, welche falsch klassifiziert wurden, ein geringeres Gewicht bekommen als solche, welche richtig klassifiziert wurden. Die sich ergebenden Testfehlerraten sind erneut anhand von Boxplots in Abbildung 4.11 bis 4.13 zu finden.

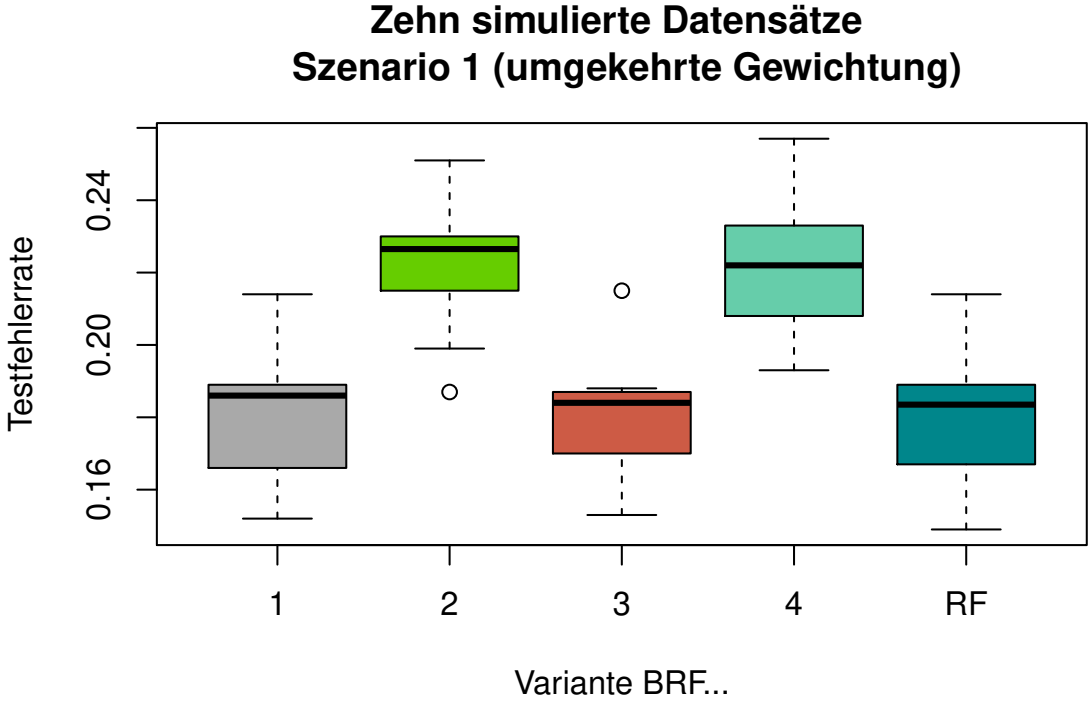

Abbildung 4.11: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Es ist sofort zu erkennen, das sich die Testfehlerraten der Varianten mit angewandten Gewichten als Ziehwahrscheinlichkeiten der Beobachtungen für die Bootstrap-

Stichprobe mit zunehmendem Ausreißeranteil immer mehr an die der Varianten ohne eine solche Nutzung der Gewichte annähern. Die Streuung der Werte der Variante *BRF2* fällt darüber hinaus für Szenario 3.2 sehr gering aus.

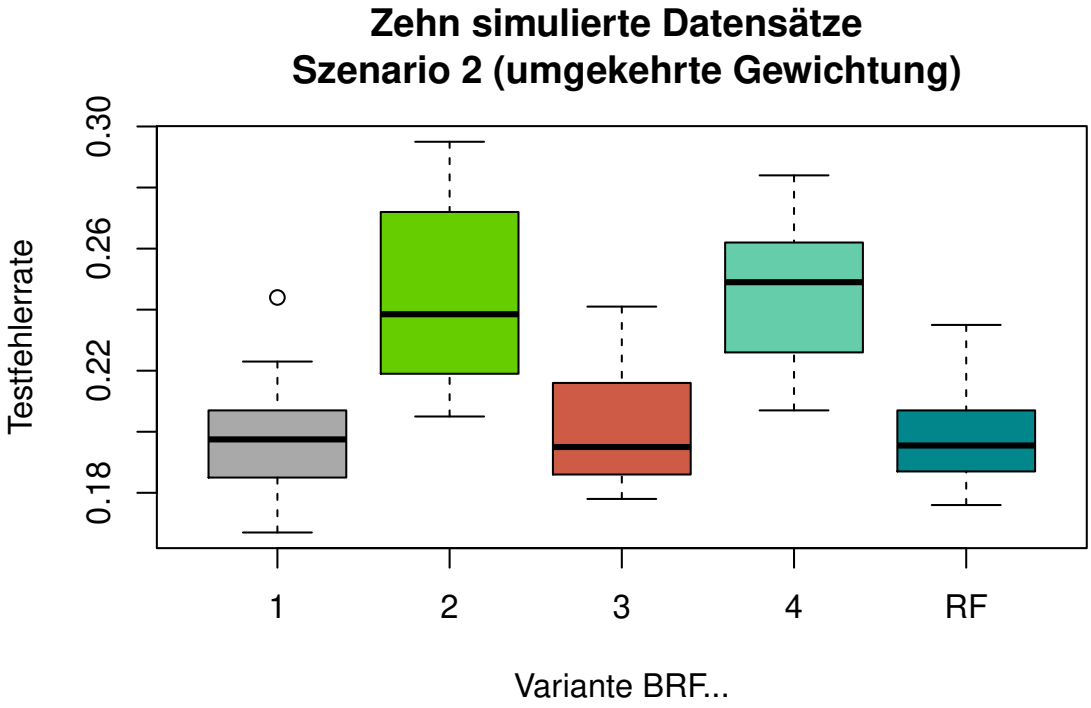

Abbildung 4.12: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

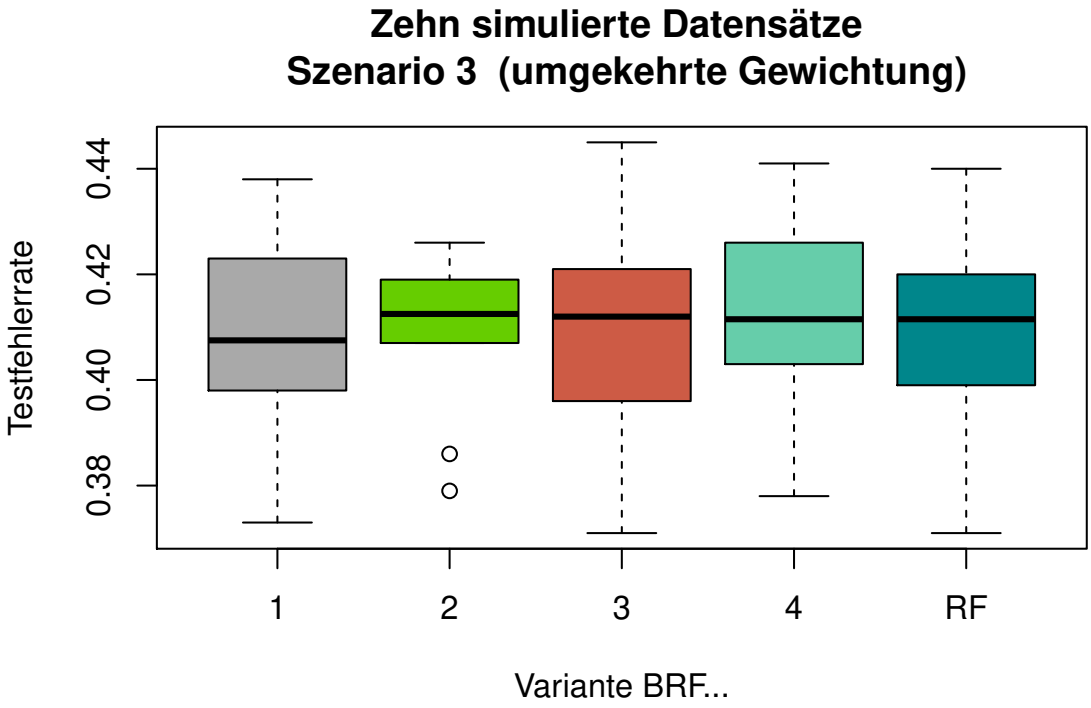

Abbildung 4.13: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Auch für diese Simulationsergebnisse ist eine Rangfolge der Varianten anhand der Häufigkeiten der niedrigsten Testfehlerraten erstellt worden. Diese ist in Abbildung 4.14 zu finden.

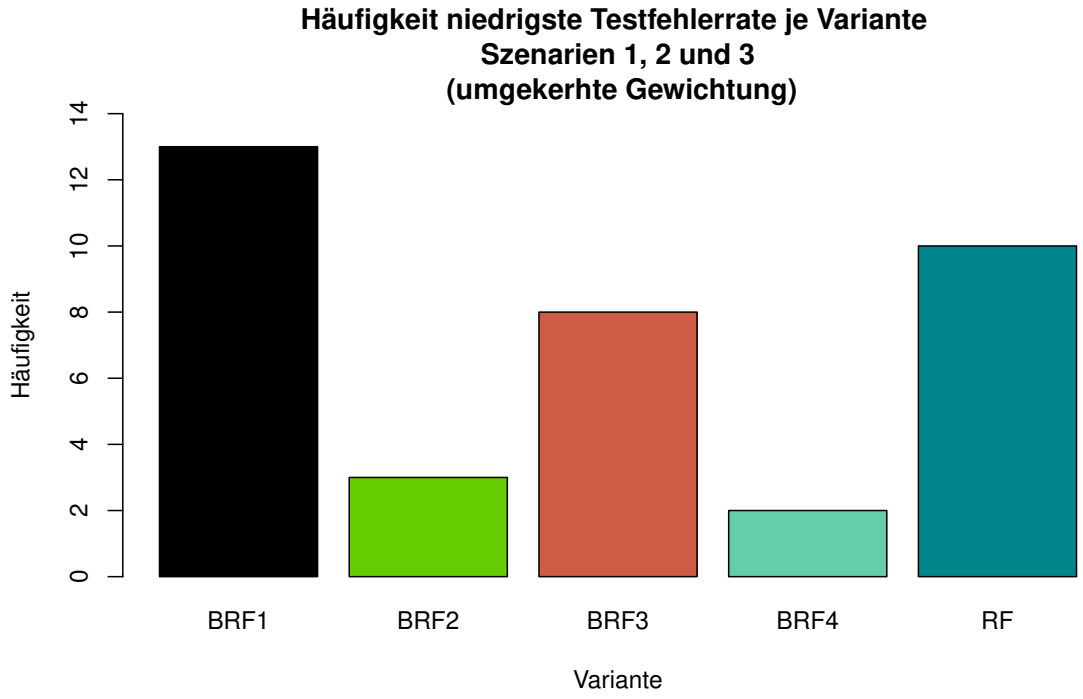

Abbildung 4.14: Balkendiagramm, welches die Häufigkeiten anzeigt, wie oft eine Variante die niedrigste Testfehlerrate erzielt hat (Szenario 1 bis 3; umgekehrte Gewichtung).

Der *BRF1* gefolgt vom Standard-RF sind hier die zwei Varianten, welche am häufigsten die niedrigste Testfehlerrate erzielen.

Im Folgenden sind die Ergebnisse der Szenarien $4-6$ beschrieben. Hierbei wurde die Anzahl der Kategorien im Response variiert, um eine Aussagen über den Einfluss dieser auf die Performance der einzelnen Varianten machen zu können.

In Abbildung 4.15 sind zunächst die Boxplots der Werte des vierten Szenarios dargestellt. Der betrachtete Wertebereich ist erneut sehr klein und die Unterschiede zwischen den Testfehlerraten der verschiedenen Varianten nicht sehr groß. Dennoch erzielen die Varianten *BRF2, 4, 6, 8* und *10* erneut die geringsten Testfehler. Hier ist deutlich zu erkennen, dass ihre Mediane (0.1815, 0.1815, 0.1875 und 0.185) niedriger als die der Varianten *BRF1, 3, 5, 7, 9* und des Standard-RF (0.1975, 0.1950, 0.1975, 0.1950, 0.1970 und 0.1955) liegen.

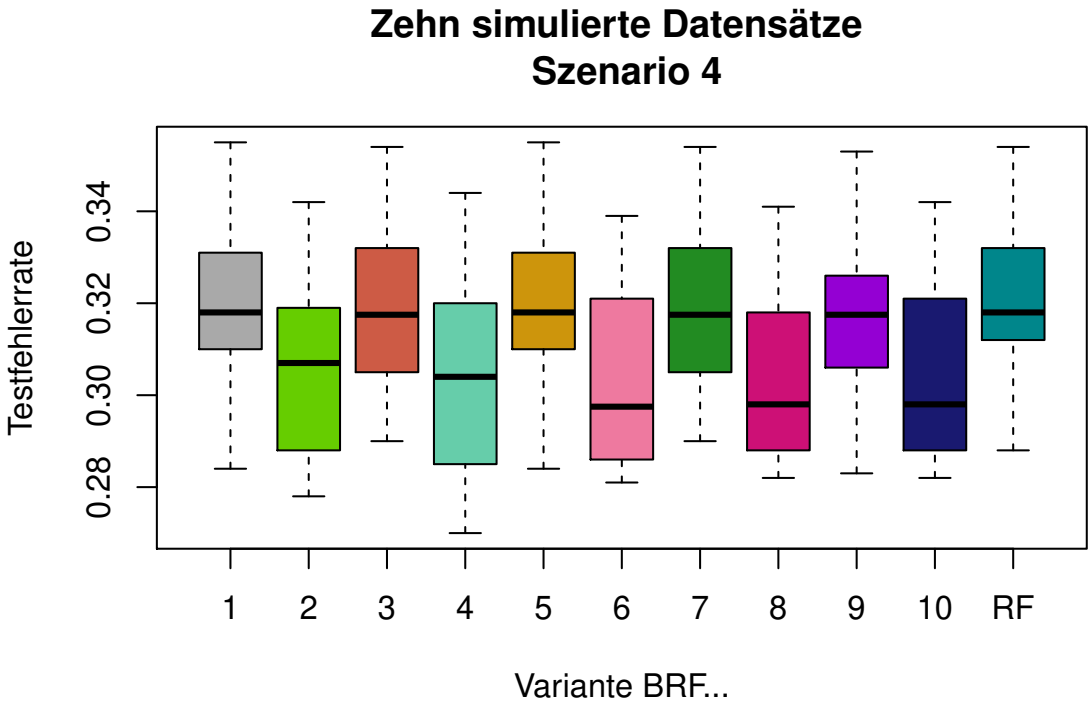

Abbildung 4.15: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Die Ergebnisse aus Szenario 5 (vgl. Abbildung 4.16) unterscheiden sich nur insofern von jenen aus Szenario 4 als dass der Interquartilsabstand größer geworden ist. Die resultierenden Testfehlerraten streuen somit breiter als noch zuvor. Da es sich bei den verwendeten Datenpunkten jedoch nur um eine Anzahl von zehn handelt, ist die Ausssagekraft hier generell nicht sehr hoch und somit nicht überzubewerten.

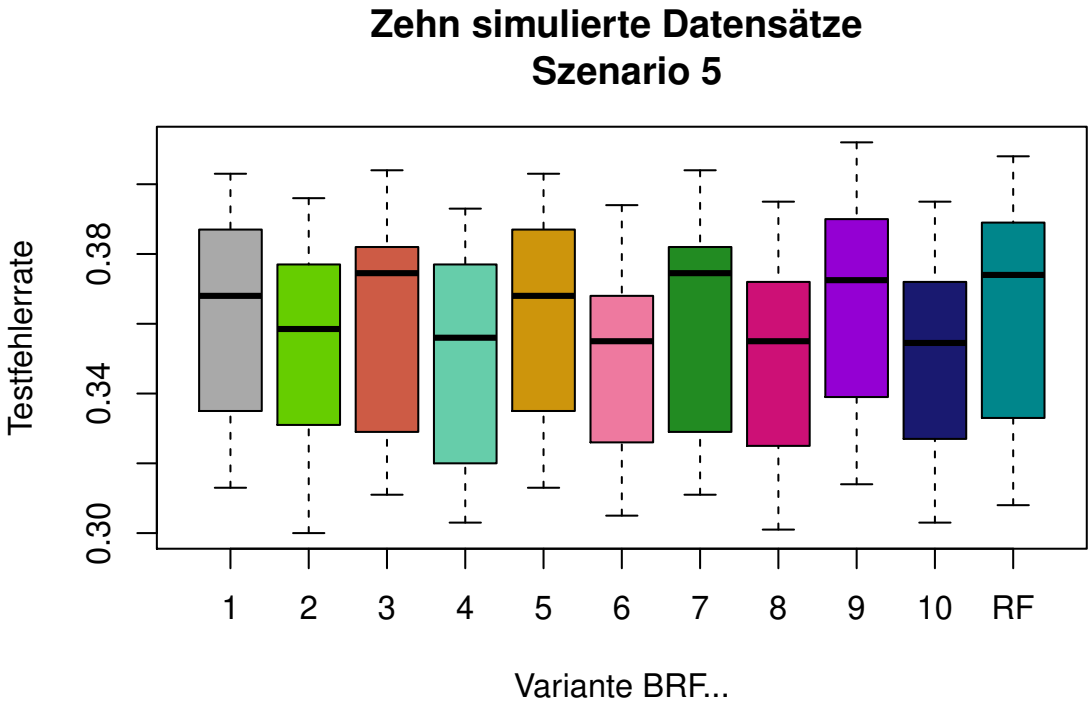

Abbildung 4.16: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

In Szenario 6 (Abbildung 4.17) rücken die Mediane der einzelnen Varianten noch näher zueinander. Der Unterschied in den Testfehlerraten wird somit noch einmal geringer. Es bleibt einzig zu bemerken, dass erneut die Varianten, in welchen die Gewichte als Ziehwahrscheinlichkeiten der Bootstrap-Stichprobe genutzt werden, minimal öfter genauere Vorhersagen zu treffen scheinen als die restlichen.

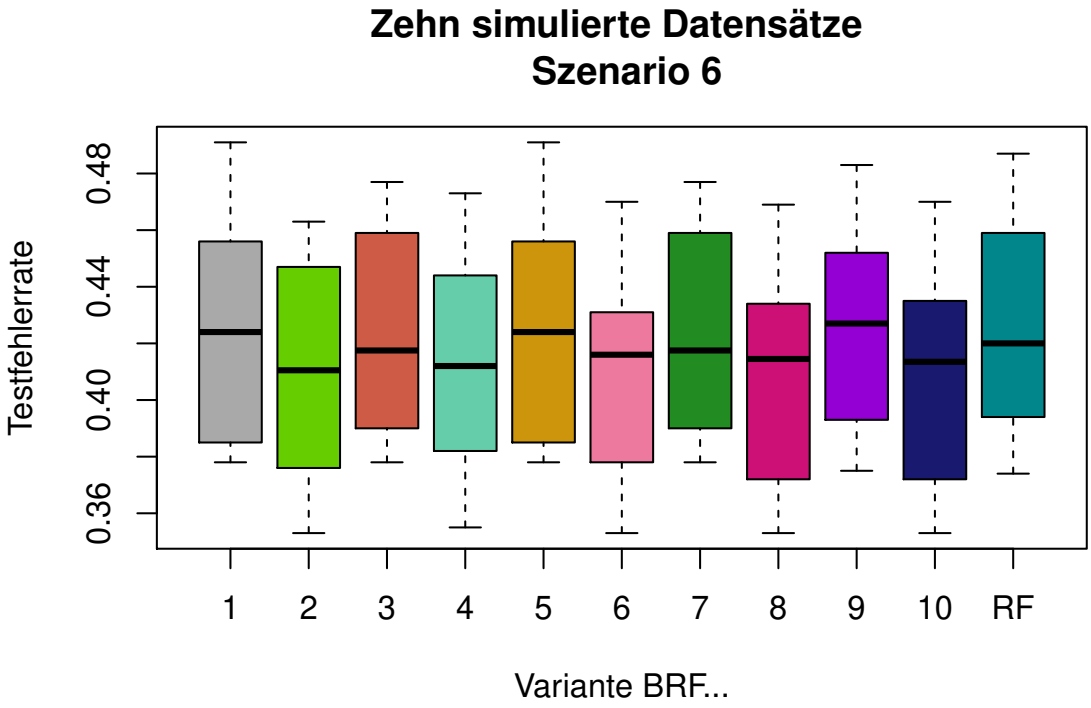

Abbildung 4.17: Boxplots der gemittelten Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF.

Abschließend stellt Abbildung 4.18 die Häufigkeiten der niedrigsten Testfehlerraten aus den Szenarien $4 - 6$ anhand eines Balkendiagramms dar. Auch hier stechen die Varianten, in welchen die Gewichte als Ziehwahrscheinlichkeiten benutzt wurden, hervor. Am häufigsten erzielt die Variante *BRF6* die niedrigste Testfehlerrate. Es handelt sich hier um eine Variante mit `method = mishina` und `vote.type = hard`.

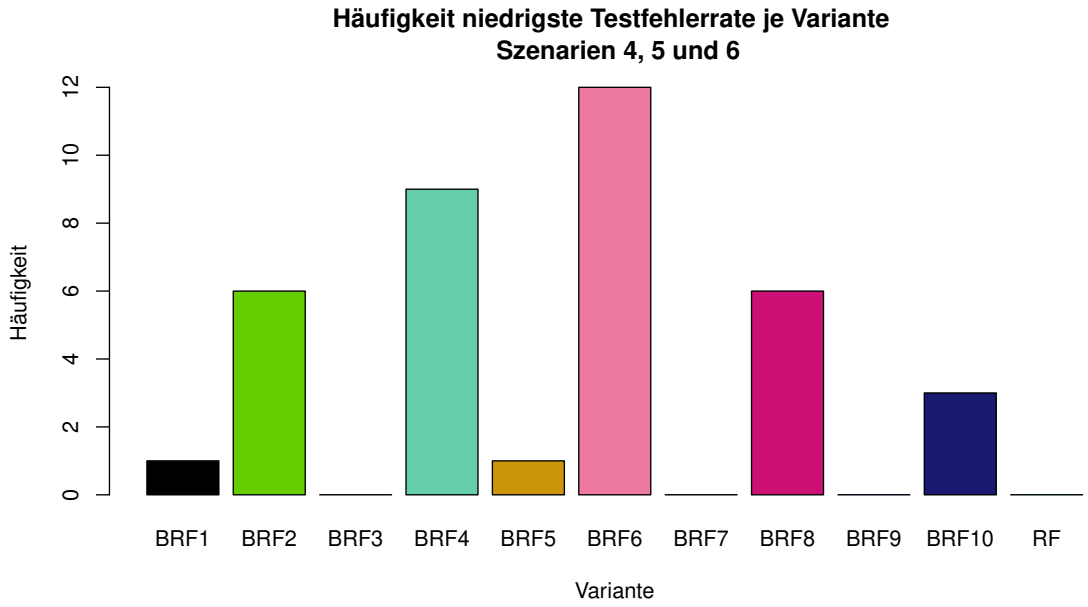

Abbildung 4.18: Balkendiagramm, welches die Häufigkeiten anzeigt, wie oft eine Variante die niedrigste Testfehlerrate erzielt hat (Szenario 4 bis 6).

Soll nun eine Aussage über den Einfluss der Zahl der Responsekategorien getroffen werden, so lässt sich beobachten, dass es mit zunehmender Kategorienanzahl nur noch einen geringen Unterschied zu machen scheint, welche Art des Algorithmus gewählt wird. Die Testfehler werden zudem mit zunehmender Kategorienanzahl größer was im Umkehrschluss bedeutet, dass die Vorhersagegenauigkeit der Varianten abnimmt. Um jedoch die bestmögliche Klassifikation zu erreichen scheint eine Wahl einer der Varianten, in welchen die Gewichte auch als Ziehwahrscheinlichkeiten verwendet werden, empfehlenswert zu sein.

5 Diskussion

In dieser Arbeit wurden verschiedene Varianten implementiert, wie Gewichte innerhalb des Random Forests angewandt werden können. Möglichkeiten hierzu sind sowohl bei der Berechnung des *majority votes* und des Splitkriteriums als auch als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe gegeben. Die implementierte Funktion deckt die Verwendung der Gewichte für letzteres und die Bestimmung des *majority votes* ab. Diese wurden anhand ihrer Testfehlerraten mit dem Standard-RF nach Breiman (2001) verglichen. Die zunächst für die Berechnung der Testfehlerrate verwendeten Datensätze aus dem `R`-Paket `OpenML` deuteten bereits an, dass die Performance der unterschiedlichen Varianten von der Beschaffenheit der jeweiligen Daten abhängen kann, sich die einzelnen Varianten hinsichtlich korrekter Klassifikation jedoch weitesgehend ähneln. In Hinblick auf eine eventuelle Überlegenheit einer der Varianten gegenüber dem Standard-RF zeichnete sich somit kein eindeutiges Bild. Vielmehr zeigte die kleine, im Anschluss durchgeführte Simulationsstudie, dass insbesondere bei einer Verwendung der Gewichte als Ziehwahrscheinlichkeiten der Beobachtungen in die Bootstrap-Stichprobe darauf zu achten ist, dass in den Daten keine, oder zumindest nur wenige, Ausreißer vorhanden sind. Andernfalls resultiert die Verwendung dieser Varianten in einem hohen Testfehler und somit einer schlechten Vorhersagegenauigkeit. Die Art der Berechnung der Gewichtsfunktion betreffend scheint die Vorgehensweise nach Bernard et al. (2012) genauere Ergebnisse zu erzielen, da hier alle bereits angepassten Bäume mit einbezogen werden. Bei der Berechnung der Gewichte nach Mishina et al. (2014) hingegen wurde jeweils nur der aktuelle Baum betrachtet und resultierte bei identischen Einstellungen der verbleibenden Parameter in höheren Testfehlerraten als die retrospektive Variante. Die Simulationen zur Untersuchung des Einflusses der Kategorienanzahl des Responses auf die Vorhersagegenauigkeit verzeichnete ähnliche Resultate. Mit zunehmender Kategorienanzahl im Response wurden die mittleren Testfehlerraten höher, die Vorhersagegenauigkeit der Methoden somit geringer. Hier waren die Varianten, in welchen die Gewichte als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe verwendet wurden, konstant besser als die restlichen. Allerdings ist hierbei anzumerken, dass die für die Bestimmung der Testfehlerraten simulierten Daten frei

von Ausreißern waren. Eine Umkehrung der Gewichtung (das heißt, dass falsch klassifizierte Beobachtungen ein geringeres Gewicht bekommen als richtig klassifizierte) scheint allerdings erst sinnvoll, wenn bekannt ist, von welcher Struktur die Daten sein müssen, damit eine solche Änderung der Gewichtsfunktion eine positive Auswirkung hat. Nach der Betrachtung der Ergebnisse aus den Szenarien 1.2, 2.2 und 3.2 scheint dies bei einem mittleren Ausreißeranteil (bis 30%) nicht der Fall zu sein. Alles in allem waren die Unterschiede in den Testfehlerraten darüber hinaus sehr gering.

Ein abschließendes Gesamtfazit ziehend lässt sich festhalten, dass es keinen großen Unterschied zu machen scheint, welche der miteinander verglichenen Varianten verwendet wird. Daher ist auf Grundlage der Ergebnisse dieser Arbeit der Standard-RF der neuen Funktion vorzuziehen, da dieser konstant gute Ergebnisse liefert und die bereits implementierten Funktionen schnell sind. Dennoch erscheint eine Verwendung von Gewichten innerhalb des RF-Algorithmus durchaus sinnvoll. Hier muss jedoch vor allem bei der Verwendung der Gewichte als Ziehwahrscheinlichkeiten für die Bootstrap-Stichprobe darauf geachtet werden, dass zuvor mögliche Ausreißerbeobachtungen aus den Daten entfernt wurden. Weiter ist eine retrospektive Gewichtsfunktion wie sie in Bernard et al. (2012) vorgeschlagen wird zu empfehlen, da diese alle bereits angepassten Bäume für die Berechnung der Gewichtsfunktion berücksichtigt und somit einzelne, schlechter klassifizierende Bäume weniger stark ins Gewicht fallen. Anders als in den Papern von Bernard et al. (2012) und Mishina et al. (2014) wurden die Gewichte nicht bei der Berechnung des Splitkriteriums verwendet. Da die erhaltenen Ergebnisse der neuen Funktion zwar teilweise besser waren als die des Standard-RF's, jedoch nicht in einem solchen Ausmaß, wie es bei Bernard et al. (2012) der Fall war, bleibt anzunehmen, dass eine zusätzliche Verwendung der Gewichte für die Splits eine weitere Verbesserung mit sich bringen würde.

Folglich wäre es interessant, diese auch für diese Art der Verwendung der Gewichte zu implementieren und mit den bereits vorgestellten Varianten zu vergleichen. Hierzu kann direkt auf den C-Code der `rfsrc()`-Funktion zugegriffen und dieser entsprechend abgeändert werden. Ebenso wäre eine komplette Übertragung des

R-Codes in C denkbar, wodurch die Rechenzeit erheblich verkürzt werden würde und die Funktion innerhalb eines entsprechenden R-Pakets zur Verfügung gestellt werden könnte. Da die Funktion zur Zeit nur bei Klassifikationsproblemen anwendbar ist, wäre es wünschenswert, eine entsprechende Variante für Regressionsprobleme zur Verfügung zu stellen. Hier ist zu überlegen, wie die Gewichte berechnet werden könnten. Eine erste, einfache Möglichkeit wäre der *Mean Squared Error (MSE)*. Jedoch würden Ausreißer hier eine wesentlich höhere Gewichtung bekommen. Wie auch bei der Klassifikation, könnte dies einen negativen Einfluss auf die Vorhersagegenauigkeit des Waldes haben. Um dieses Problem zu umgehen, wäre zum einen denkbar, ein anderes Maß zu verwenden, welches robust gegen Ausreißer ist, oder zum anderen Ausreißer vorab als solche zu definieren, und bei der Gewichtung nicht zu berücksichtigen. Dies wäre auch eine Überlegung für die Klassifikation mit der Parametereinstellung `sample.weights = TRUE` um die sich in den Simulationen gezeigten Problemen der Funktion mit ausreißerbehafteten Daten zu eliminieren.

Die Simulationen in Abschnitt 4.2 haben darauf abgezielt zu überprüfen, ob Ausreißer oder eine hohe Kategorienanzahl im Response einen Einfluss auf die Vorhersagegenauigkeit des Waldes haben. Weitere mögliche Einflüsse sind die Klassenhäufigkeiten oder die Anzahl der Prädiktorvariablen. Diese könnten mit Hilfe weiterer Simulationen überprüft werden. Hierzu würde sich erneut das multinomiale Logit-Modell aus Abschnitt 4.2 anbieten, über welches die Wahrscheinlichkeiten für die unterschiedlichen Klassen des Response und somit die Klassenhäufigkeiten reguliert werden können. Wie bereits in Abschnitt 3.2 angemerkt, ist die von Bernard et al. (2012) vorgeschlagene Gewichtsfunktion nicht die einzige denkbare. Hier könnten weitere Varianten implementiert und mit der bereits vorhandenen verglichen werden. Auch über diese könnten Ausreißer weniger starken Einfluss auf die Gesamtperformance des Waldes ausüben. Eine Möglichkeit wäre, wie es auch in Mishina et al. (2014) vorgeschlagen wird, den Logarithmus zu verwenden, da durch diese großen Werten automatisch kein unverhältnismäßiges Gewicht zugeteilt werden würde. Eine Kombination der Methoden von Bernard et al. (2012) und Mishina et al. (2014), das heißt eine retrospektive Gewichtung mit Hilfe des natürlichen Logarithmus könnte einen Lösungsansatz darstellen, welcher anhand von Simulationen ähnlich jenen dieser

Arbeit überprüft werden kann.

Generell könnten die Simulationen erneut mit mehreren Iterationen und anderen Seeds durchgeführt werden, um ihre Stabilität zu überprüfen und ihre Aussagekraft zu erhöhen.

Literaturverzeichnis

Bernard, S., S. Adam, und L. Heutte (2012). Dynamic random forests. *Pattern Recognition Letters 33*(12), 1580–1586.

Bischl, B., M. Lang, L. Kotthoff, J. Schiffner, J. Richter, Z. Jones, und G. Casalicchio (2016). *mlr: Machine Learning in R*. R package version 2.9.

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

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

Casalicchio, G., B. Bischl, D. Kirchhoff, M. Lang, B. Hofner, J. Bossek, P. Kerschke, und J. Vanschoren (2015). *OpenML: Exploring Machine Learning Better, Together*. R package version 1.0.

Freund, Y. und R. E. Schapire (1995). A desicion-theoretic generalization of on-line learning and an application to boosting. In P. Vitányi (Ed.), *Computational learning theory*, Volume 904 of *Lecture notes in computer science Lecture notes in artificial intelligence*, S. 23–37. Berlin: Springer.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics 29*(5), 1189–1232.

Hastie, T., R. Tibshirani, und J. H. Friedman (2009). *The elements of statistical learning: Data mining, inference and prediction* (Second edition, corrected 7th printing ed.). Springer series in statistics. New York: Springer.

Ishwaran, H. (2015). The effect of splitting on random forests. *Machine Learning 99*(1), 75–118.

Ishwaran, H. und U. Kogalur (2007, October). Random survival forests for r. *R News 7*(2), 25–31.

Ishwaran, H. und U. Kogalur (2016). *Random Forests for Survival, Regression and Classification (RF-SRC)*. R package version 2.3.0.

Ishwaran, H., U. Kogalur, E. Blackstone, und M. Lauer (2008). Random survival forests. *Ann. Appl. Statist. 2*(3), 841–860.

James, G., D. Witten, T. Hastie, und R. Tibshirani (2013). *An introduction to statistical learning: With applications in R*, Volume 103 of *Springer texts in statistics*. New York, NY: Springer.

Kubat, M. (2015). *An introduction to machine learning*. Cham: Springer.

Mishina, Y., M. Tsuchiya, und H. Fujiyoshi (2014). Boosted random forest. In *Proceedings of the 9th International Conference on Computer Vision Theory and Applications - Volume 2: VISAPP, (VISIGRAPP 2014)*, S. 594–598.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing.

Strobl, C., A.-L. Boulesteix, T. Kneib, T. Augustin, und A. Zeileis (2008). Conditional Variable Importance for Random Forests. Technical report, University of Munich, Department of Statistics.

Tutz, G. (2012). *Regression for categorical data*. Cambridge series in statistical and probabilistic mathematics. Cambridge: Cambridge University Press.

Abbildungsverzeichnis

Tabellenverzeichnis

A Anhang

Im folgenden Anhang sind neben einigen der für die Erstellung der Arbeit implementierten Funktionen noch ergänzende Abbildungen und Tabellen angefügt. Ebenso ist der Inhalt der beigefügten DVD aufgelistet und beschrieben.

A.1 Funktionen

Die Funktion `createBinData`

Diese Funktion simuliert Daten mit einem binären Response. Über `niter` kann die Anzahl der zu simulierenden Datensätze bestimmt werden, über `size` die Anzahl der Beobachtungen und `outlier.prob` reguliert den Anteil an Ausreißern. `mX` stellt die Prädiktormatrix, bestehend aus fünf Prädiktorvariablen, dar. `vbeta` bezeichnet den Koeffizientenvektor für Klasse 1. In `vProb` werden die anhand der Prädiktormatrix und dem Koeffizientenvektor berechneten Wahrscheinlichkeiten gespeichert, anhand welcher die letztendliche Klasse bestimmt wird. Dabei gilt, dass eine Beobachtung Klasse 1 angehört, falls die Wahrscheinlichkeit $P(Y = 1|\boldsymbol{x}) \geq 0.5$ ist. Falls Ausreißer in den Daten gewünscht sind, so wird der vorgegebene Anteil an Beobachtungen innerhalb jeder Klasse in die jeweils andere Klasse verschoben. Die entsprechenden Beobachtungen werden mit `out1` und `out0` zufällig festgelegt. Die Ausgabe der Funktion ist eine Liste mit der über `niter` festgelegten Anzahl an Datensätzen.

```
createBinData = function(niter, size, outlier.prob){
simData = list()
for(j in 1:niter){
mX = matrix(rnorm(size·5, 0, 1), size, 5)
vbeta = c(5,8,3.5,6.8,7)
vProb = (exp(mX·vbeta)/(1+exp(mX·vbeta)))
class = as.numeric(vProb ≥ 0.5)
```

```
out1 = sample(which(class == 1), round(outlier.prob·sum(class == 1)))
out0 = sample(which(class == 0), round(outlier.prob·sum(class == 0)))
class[out1] = 0
class[out0] = 1
mdata = cbind.data.frame(class = factor(class),mX)
colnames(mdata) = c("class",paste("V",1:5,sep='' ''))
simData[[j]] = mdata
}


return(simData)}
```

Die Funktion *createMultData*

Diese Funktion generiert Datensätze mit multinomialem Response. Die Anzahl der Responsekategorien `numclass` kann dabei aus der Menge $\{1,\ldots,10\}$ gewählt werden. `niter` bestimmt die Anzahl der zu simulierenden Datensätze und `size` die Anzahl der Beobachtungen. Ausreißer können hier nicht mitsimuliert werden. `mX` ist die Prädiktormatrix bestehend aus fünf Prädiktorvariablen und die Vektoren `vCoef1,...,vCoef10` sind die Koeffizientenvektoren der jeweiligen Klassen mit Klasse 1 als Referenzkategorie. Die Matrix `pi` besteht in den Spalten aus den Wahrscheinlichkeitsvektoren der einzelnen Klassen. Auf den Nenner wird verzichtet, da die im Folgenden zur Ziehung der Responsekategorien genutzte Funktion `rmultinom` die Wahrscheinlichkeit intern normiert. Ausgabe der Funkion ist eine Liste bestehend aus `niter` Datensätzen.

```
createMultData <- function(niter, size, numclass){


simData <- list()
for(i in 1:niter){
mX <- matrix(rnorm(size·5, 0, 1), size, 5)
vCoef1 <- c(0,0,0,0,0)
vCoef2 <- c(-4,5,7,6,3)
```

```
vCoef3 <- c(5,2,-6,4,3)
vCoef4 <- c(4,7,3,8,-2)
vCoef5 <- c(-6,9,2,6,1)
vCoef6 <- c(8,4,-2,2,7)
vCoef7 <- c(3,1,3,1,-4)
vCoef8 <- c(-2,4,6,1,4)
vCoef9 <- c(1,-2,7,3,1)
vCoef10 <- c(4,-7,2,-4,5)
mCoef <- cbind(vCoef1, vCoef2, vCoef3, vCoef4, vCoef5, vCoef6, vCoef7,
                        vCoef8, vCoef9, vCoef10)
mCoef <- mCoef[,c(1:numclass)]
pi = exp(mX·mCoef)
mChoices = t(apply(pi, 1, rmultinom, n = 1, size = 1))
mdata = cbind.data.frame(class = factor(apply(mChoices, 1,
                        function(x) which(x==1))), mX)
colnames(mdata) <- c("class",paste("V",1:5,sep=))
simData[[i]] <- mdata
}
return(simData)
}
```

Die Funktion `predict.brf`

Die Funktion `predict.brf` ist eine Funktion zur Klassifikation von unbekannten Beobachtungen auf Grundlage eines Modells der Klasse `brf`. Dabei werden die durch die `brf`-Funktion (vgl. Abschnitt 3.3) übergebenen Parametereinstellungen weiter berücksichtigt. Durch die Parametereinstellung `prob = TRUE` (Default ist FALSE) kann trotz zuvor gewählter Einstellung `vote.type = hard` eine Berechnung des *majority votes* entsprechend der Einstellung `vote.type = soft` erreicht werden. Die komplette Funktion ist auf der DVD einzusehen. Siehe hierzu auch Abschnitt A.3.

A.2 Weitere Abbildungen und Tabellen

Darstellung der Mittelwerte der mittleren Testfehlerraten aus den Simulationen

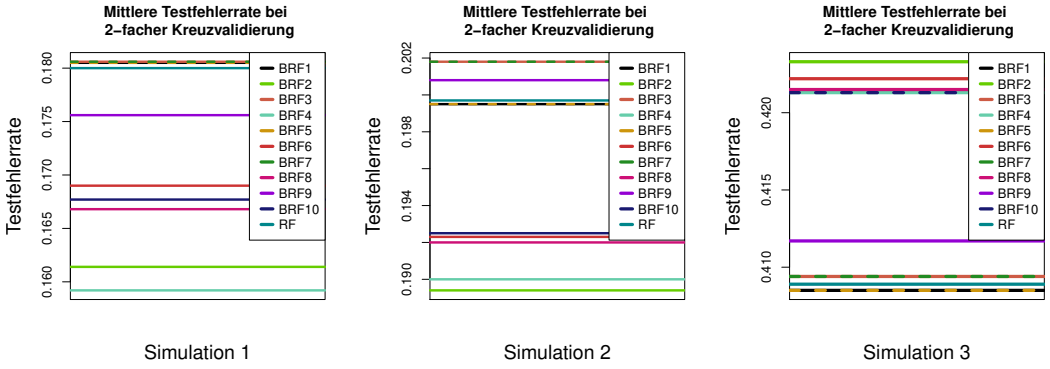

Abbildung A.1: Gemittelte Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF über 10 Datensätze.

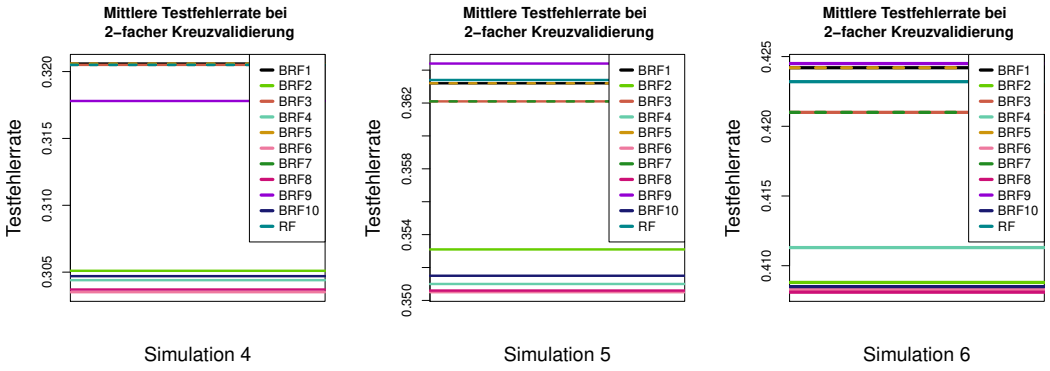

Abbildung A.2: Gemittelte Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF über 10 Datensätze.

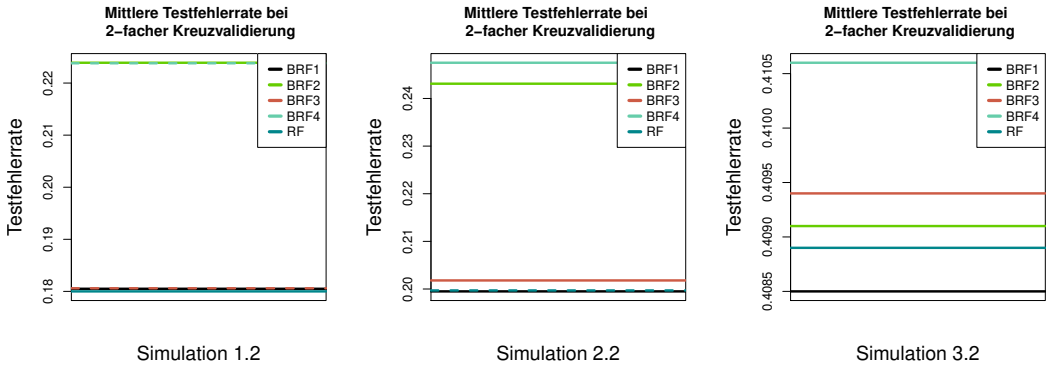

Abbildung A.3: Gemittelte Testfehlerraten aus zweifacher Kreuzvalidierung mit fünfmaliger Wiederholung der verschiedenen Varianten im Vergleich mit dem Standard-RF über 10 Datensätze.

Simulationsergebnisse

Auf den folgenden Seiten sind die Tabellen mit den Simulationsergebnisse zu finden. Aus Platzgründen sind diese im Querformat abgebildet.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.214	0.187	0.215	0.184	0.214	0.199	0.215	0.199	0.198	0.196	0.214
2	0.187	0.164	0.183	0.158	0.187	0.179	0.183	0.169	0.174	0.172	0.182
3	0.161	0.143	0.160	0.146	0.161	0.152	0.160	0.149	0.161	0.148	0.162
4	0.173	0.147	0.179	0.150	0.173	0.156	0.179	0.152	0.171	0.153	0.174
5	0.190	0.163	0.188	0.159	0.190	0.171	0.188	0.169	0.180	0.170	0.189
6	0.189	0.175	0.187	0.174	0.189	0.178	0.187	0.177	0.186	0.180	0.189
7	0.152	0.133	0.153	0.129	0.152	0.139	0.153	0.140	0.151	0.139	0.149
8	0.185	0.171	0.186	0.167	0.185	0.177	0.186	0.176	0.183	0.177	0.185
9	0.166	0.161	0.170	0.161	0.166	0.163	0.170	0.159	0.170	0.162	0.167
10	0.188	0.170	0.185	0.164	0.188	0.176	0.185	0.178	0.182	0.180	0.189

Tabelle A.1: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 1, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.223	0.212	0.232	0.207	0.223	0.213	0.232	0.219	0.227	0.222	0.224
2	0.207	0.200	0.216	0.199	0.207	0.199	0.216	0.206	0.217	0.205	0.207
3	0.195	0.171	0.189	0.177	0.195	0.178	0.189	0.178	0.189	0.179	0.188
4	0.200	0.182	0.201	0.177	0.200	0.196	0.201	0.188	0.206	0.188	0.201
5	0.167	0.181	0.178	0.185	0.167	0.174	0.178	0.172	0.177	0.170	0.176
6	0.244	0.241	0.241	0.239	0.244	0.239	0.241	0.230	0.227	0.228	0.235
7	0.185	0.174	0.189	0.178	0.185	0.179	0.189	0.182	0.190	0.184	0.187
8	0.185	0.169	0.184	0.168	0.185	0.171	0.184	0.173	0.189	0.175	0.186
9	0.185	0.173	0.186	0.176	0.185	0.178	0.186	0.178	0.182	0.178	0.190
10	0.204	0.191	0.202	0.194	0.204	0.196	0.202	0.194	0.204	0.196	0.203

Tabelle A.2: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 2, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.438	0.444	0.445	0.434	0.438	0.435	0.445	0.432	0.427	0.431	0.440
2	0.408	0.402	0.412	0.400	0.408	0.409	0.412	0.406	0.421	0.405	0.408
3	0.400	0.420	0.396	0.423	0.400	0.419	0.396	0.422	0.406	0.424	0.399
4	0.398	0.419	0.403	0.409	0.398	0.421	0.403	0.421	0.396	0.419	0.399
5	0.423	0.439	0.421	0.434	0.423	0.437	0.421	0.437	0.396	0.438	0.420
6	0.373	0.394	0.371	0.382	0.373	0.384	0.371	0.384	0.371	0.382	0.371
7	0.419	0.438	0.417	0.442	0.419	0.440	0.417	0.429	0.428	0.430	0.419
8	0.394	0.406	0.393	0.404	0.394	0.399	0.393	0.406	0.411	0.405	0.395
9	0.407	0.429	0.412	0.435	0.407	0.431	0.412	0.427	0.430	0.428	0.415
10	0.425	0.442	0.424	0.450	0.425	0.447	0.424	0.451	0.431	0.451	0.423

Tabelle A.3: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 3, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.316	0.280	0.315	0.278	0.316	0.282	0.315	0.284	0.306	0.282	0.320
2	0.355	0.342	0.354	0.344	0.355	0.339	0.354	0.341	0.353	0.342	0.354
3	0.317	0.314	0.316	0.308	0.317	0.302	0.316	0.303	0.315	0.303	0.316
4	0.328	0.311	0.325	0.320	0.328	0.322	0.325	0.318	0.326	0.321	0.332
5	0.310	0.296	0.304	0.300	0.310	0.291	0.304	0.293	0.305	0.292	0.312
6	0.304	0.303	0.305	0.297	0.304	0.293	0.305	0.291	0.310	0.293	0.300
7	0.342	0.320	0.345	0.318	0.342	0.321	0.345	0.316	0.326	0.318	0.339
8	0.331	0.319	0.332	0.324	0.331	0.318	0.332	0.321	0.334	0.325	0.330
9	0.284	0.278	0.290	0.270	0.284	0.281	0.290	0.282	0.283	0.283	0.288
10	0.319	0.288	0.319	0.285	0.319	0.286	0.319	0.288	0.320	0.288	0.314

Tabelle A.4: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 4, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.353	0.345	0.350	0.341	0.353	0.346	0.350	0.347	0.346	0.347	0.349
2	0.382	0.377	0.379	0.377	0.382	0.367	0.379	0.368	0.382	0.368	0.381
3	0.374	0.367	0.381	0.362	0.374	0.360	0.381	0.363	0.370	0.362	0.382
4	0.403	0.396	0.404	0.393	0.403	0.394	0.404	0.395	0.412	0.395	0.408
5	0.362	0.350	0.370	0.350	0.362	0.350	0.370	0.342	0.375	0.345	0.367
6	0.387	0.370	0.382	0.370	0.387	0.368	0.382	0.372	0.390	0.372	0.389
7	0.330	0.331	0.325	0.320	0.330	0.326	0.325	0.325	0.324	0.327	0.324
8	0.313	0.300	0.311	0.303	0.313	0.305	0.311	0.301	0.314	0.303	0.308
9	0.335	0.310	0.329	0.309	0.335	0.310	0.329	0.310	0.339	0.312	0.333
10	0.393	0.385	0.390	0.385	0.393	0.379	0.390	0.383	0.392	0.384	0.393

Tabelle A.5: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 5, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	BRF5	BRF6	BRF7	BRF8	BRF9	BRF10	RF
1	0.412	0.407	0.409	0.404	0.412	0.414	0.409	0.412	0.414	0.411	0.413
2	0.491	0.463	0.477	0.473	0.491	0.470	0.477	0.469	0.483	0.470	0.487
3	0.385	0.389	0.391	0.393	0.385	0.385	0.391	0.393	0.401	0.393	0.394
4	0.438	0.414	0.426	0.420	0.438	0.420	0.426	0.418	0.450	0.417	0.427
5	0.436	0.424	0.438	0.425	0.436	0.418	0.438	0.417	0.440	0.416	0.437
6	0.398	0.376	0.390	0.382	0.398	0.378	0.390	0.372	0.393	0.372	0.398
7	0.381	0.353	0.378	0.357	0.381	0.353	0.378	0.353	0.377	0.353	0.380
8	0.467	0.460	0.460	0.460	0.467	0.457	0.460	0.457	0.460	0.459	0.463
9	0.456	0.447	0.459	0.444	0.456	0.431	0.459	0.434	0.452	0.435	0.459
10	0.378	0.355	0.382	0.355	0.378	0.357	0.382	0.356	0.375	0.359	0.374

Tabelle A.6: Simulationsergebnisse (Testfehlerrate der Datensätze 1 bis 10) aus Szenario 6, in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	RF
1	0.214	0.251	0.215	0.257	0.214
2	0.187	0.226	0.183	0.218	0.182
3	0.161	0.215	0.160	0.208	0.162
4	0.173	0.225	0.179	0.230	0.174
5	0.190	0.250	0.188	0.254	0.189
6	0.189	0.230	0.187	0.233	0.189
7	0.152	0.199	0.153	0.201	0.149
8	0.185	0.229	0.186	0.219	0.185
9	0.166	0.187	0.170	0.193	0.167
10	0.188	0.227	0.185	0.225	0.189

Tabelle A.7: Simulationsergebnisse (Testfehler der Datensätze 1 bis 10) aus Szenario 1 (umgekehrte Gewichtung), in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	RF
1	0.223	0.295	0.232	0.284	0.224
2	0.207	0.245	0.216	0.262	0.207
3	0.195	0.239	0.189	0.246	0.188
4	0.200	0.274	0.201	0.260	0.201
5	0.167	0.205	0.178	0.225	0.176
6	0.244	0.272	0.241	0.279	0.235
7	0.185	0.225	0.189	0.234	0.187
8	0.185	0.219	0.184	0.226	0.186
9	0.185	0.238	0.186	0.252	0.190
10	0.204	0.219	0.202	0.207	0.203

Tabelle A.8: Simulationsergebnisse (Testfehler der Datensätze 1 bis 10) aus Szenario 2 (umgekehrte Gewichtung), in rot: die niedrigste Testfehlerrate.

	BRF1	BRF2	BRF3	BRF4	RF
1	0.438	0.425	0.445	0.441	0.440
2	0.408	0.426	0.412	0.426	0.408
3	0.400	0.379	0.396	0.378	0.399
4	0.398	0.419	0.403	0.413	0.399
5	0.423	0.407	0.421	0.415	0.420
6	0.373	0.386	0.371	0.390	0.371
7	0.419	0.415	0.417	0.404	0.419
8	0.394	0.408	0.393	0.403	0.395
9	0.407	0.410	0.412	0.410	0.415
10	0.425	0.416	0.424	0.426	0.423

Tabelle A.9: Simulationsergebnisse (Testfehler der Datensätze 1 bis 10) aus Szenario 3 (umgekehrte Gewichtung), in rot: die niedrigste Testfehlerrate.

A.3 Beschreibung der beigefügten DVD

Die beigefügte DVD beinhaltet alle Funktionen, Grafiken und R-Codes die zur Erstellung dieser Arbeit verwendet wurden. Weiter enthält sie die vorliegende Arbeit im pdf-Format und eine *Readme*-Datei, welche eine Beschreibung aller weiteren Dateien enthält.

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

München, 19.12.2016

Eva-Maria Müntefering