Bachelorarbeit

Ludwig-Maximilians-Universität München
Institut für Statistik

Multi-Class Classification:
Vergleich von Support Vector Machine mit anderen ausgewählten Algorithmen

---

Autor:
Thomas Westermeier

Betreuer:
Prof. Dr. Christian Heumann

Datum:
02. Mai 2019

Zusammenfassung

In dieser Arbeit soll die Performance der Support Vector Machine (SVM) analysiert werden, wenn mehr als zwei Klassen zu kategorisieren sind. Um die Leistungsfähigkeit der Support Vector Machine bei der Multi-Class Classification zu beurteilen, wird sie mit den beliebten Methoden Random Forest und Extreme Gradient Boosting verglichen. Für die Support Vector Machine wird der Ansatz One-versus-One verwendet. Die Analysen werden auf insgesamt acht Datensätzen durchgeführt. Dabei werden die erzielte Accuracy und die Zeit, die das Tuning in Anspruch nimmt, untersucht. Die Ergebnisse zeigen, dass die Support Vector Machine, in den meisten Fällen, gut abschneidet, gefolgt von Random Forest. Das Extreme Gradient Boosting kann nicht überzeugen. Allerdings deuten die Ergebnisse darauf hin, dass die Multi-Class Support Vector Machine größere Schwierigkeiten als die anderen Methoden besitzt, wenn die Klassenverteilung ein starkes Ungleichgewicht aufweist. Zudem benötigt die SVM mit steigender Anzahl an Beobachtungen erheblich mehr Zeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Bereich des Supervised Machine Learnings ist die Klassifizierung einer der Hauptaspekte. Dabei zeichnet sich das zugrunde liegende Datenproblem dadurch aus, dass eine kategoriale Zielgröße (Klasse) mithilfe von verschiedenen Einflussgrößen (Features/Attributen) vorhergesagt werden soll. Der Classifier wird mit einem Satz Trainingsdaten, die neben den Features auch die Klassenlabel beinhalten, trainiert. Dieses Modell soll dann dazu in der Lage sein, auf neuen Daten ohne Klassenlabel eine gute Vorhersage zu treffen. Dabei hängt die Güte der Vorhersage von mehreren Faktoren ab. Zum einen wird die Qualität der Vorhersage durch das zugrunde liegende Datenproblem und der Wahl des Classifiers beeinflusst. Es existiert keine Methode, die für jeden Datensatz die beste Performance liefert [11]. Hier ist grundsätzlich ein gewisses Maß an Expertise im Bereich Machine Learning nötig. Ein weiterer Faktor, der die Vorhersagequalität beeinflusst und ebenso Erfahrung benötigt, ist die Bestimmung passender Werte für einzelne Hyperparameter. Für diesen zeitintensiven Vorgang wurden verschiedene Methoden entwickelt, die sich unter dem Stichwort Tuning zusammenfassen lassen. Erst nach dieser Optimierung lässt sich ein Bild von der Performance eines Classifiers machen.

Eine etablierte Methode zur Kategorisierung von Daten stellt die Support Vector Machine dar. Der Grundgedanke dieses Classifiers besteht darin, eine Hyperebene zu finden die in der Lage ist, die Daten zu trennen, um so eine Einteilung in die einzelnen Klassen zu ermöglichen. Allerdings ist die SVM für den Fall entwickelt worden, dass nur zwei Kategorien zu klassifizieren sind. Zur Klassifizierung mehrerer Kategorien kann die Methode allerdings durch verschiedene Ansätze erweitert werden. Da die Multi-Class Classification allerdings nicht den originären Ursprung der Methode darstellt, soll in dieser Arbeit verglichen werden, wie die Multi-Class Support Vector Machine im Vergleich zu anderen Methoden abschneidet. Zu Beginn erfolgt eine detaillierte Einführung in die Support Vector Machine und der ihr zugrunde liegenden Konzepte, gefolgt von einer Beschreibung verschiedener Ansätze zur Klassifizierung von mehr als zwei Klassen. Auch die beiden Methoden, die zum Vergleich herangezogen werden, erhalten eine kurze Einführung. In Abschnitt 3 kommt es dann zu einem Vergleich der drei Methoden. Dabei werden unter anderem die behandelten Datensätze und die für diese Arbeit genutzte Software vorgestellt. Als erstes kleines Experiment wird dann evaluiert, wie die verschiedenen Methoden abschneiden, wenn den trainierten Modellen die voreingestellten Hyperparameter zugrunde liegen. Im Abschnitt "Tuning der Hyperparameter" wird dann genauer auf die einzelnen Hyperparameter eingegangen und das Prozedere des Tunings vorgestellt, bevor dann die Ergebnisse und die benötigte Zeit für die Optimierung der einzelnen Parameter sämtlicher Methoden analysiert werden. Abschließend

erfolgt ein Benchmarkexperiment, bei dem das Klassifizierungsresultat der Methoden noch etwas genauer untersucht werden soll.

2 Vorstellung der verwendeten Methoden

Im folgenden Abschnitt wird ein Überblick über die verwendeten Klassifizierungsmethoden gegeben. Das Hauptaugenmerk soll dabei auf der Support Vector Machine liegen, die den Kern der vorliegenden Arbeit darstellt. Verglichen wird die SVM dabei mit Random Forests und XGBoost, die anschließend kurz beschrieben werden.

2.1 Support Vector Machine

Zu Beginn des Abschnitts erfolgt eine Einführung in das Konzept der Hyperebene, welche die Grundlage für die Klassifizierung mittels einer Support Vector Machine (SVM) liefert. Danach wird der Support Vector Classifier behandelt, bevor dann die SVM und deren Modifizierung für die Klassifizierung von mehr als zwei Kategorien erläutert wird.

2.1.1 Hyperebene

Die grundlegende Idee besteht darin, mittels einer Hyperebene eine lineare Entscheidungsgrenze zu erhalten, die in der Lage ist eine kategoriale Zielgröße richtig zu klassifizieren. Allgemein gesprochen stellt eine Hyperebene ein $(p-1)$-dimensionales Objekt in einem p-dimensionalen Merkmalsraum dar.

Bei Vorliegen eines binären Klassifizierungsproblems mit den Klassenlabeln $y_i \in \{-1, 1\}$ und einem Trainingsdatensatz mit n Paaren $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n), \mathbf{x}_i \in \mathbb{R}^p$, ist die Hyperebene gegeben durch

$$\{x : f(x) = \langle \mathbf{w}, \mathbf{x} \rangle + b = 0\}. \tag{1}$$

Der Vektor $\mathbf{w}$ ist ein, orthogonal auf der Hyperebene stehender, Richtungsvektor dessen Länge nicht relevant ist. Der Wert b bezeichnet den sogenannten Bias. Die Entscheidungsfunktion, die eine Beobachtung einer Klasse zuordnet ist gegeben durch die Form:

$$G(x) = sgn[\langle \mathbf{w}, \mathbf{x}_i \rangle + b]. \tag{2}$$

Die Zuteilung der Beobachtung zu einer Klasse sagt aber noch nichts darüber aus, ob die Beobachtung richtig klassifiziert wurde. Um zu bestimmen, ob eine Klasse richtig klassifiziert wurde multipliziert man $f(\mathbf{x}_i)$ mit dem Klassenlabel y_i. Ist $y_i \cdot f(\mathbf{x}_i)$ positiv, so wurde die Beobachtung korrekt zugeordnet.

Es lassen sich allerdings unendlich viele solcher Hyperebenen finden, die in der Lage sind die Trainingsdaten richtig zu unterteilen. Jedoch wird nicht jede dieser Entscheidungsgrenzen gute Ergebnisse liefern, wenn unbekannte Testdaten klassifiziert werden sollen. Ziel ist folglich, die bestmögliche Trennebene für das vorliegende Datenproblem

zu finden. Diese optimale Hyperebene liegt vor, wenn sie die Trainingsdaten mit einem "Maximal Margin" trennt. Das bedeutet, dass eine Hyperebene gesucht wird, die den größtmöglichen Abstand zu den einzelnen Datenpunkten aufweist [32],[13].

2.1.2 Optimierungsproblem

Der Functional Margin f_i ist der Abstand einer Trainingsbeobachtung i zu einer bestimmten Hyperebene. Nachdem für jede einzelne Beobachtung des Trainingsdatensatzes der Functional Margin berechnet wurde, wird der kleinste Wert gewählt und als F bezeichnet. F stellt nun den Functional Margin für den gesamten Datensatz dar.

$$\begin{aligned} f_i &= y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \\ F &= \min_{i=1,\dots,n} f_i. \end{aligned} \tag{3}$$

Da der Functional Margin jedoch nicht skaleninvariant ist, wird er in den Geometric Margin transformiert. Dafür wird der Vektor $\mathbf{w}$ und der Bias b durch $\|\mathbf{w}\|$ dividiert. Aus (3) folgt:

$$\begin{aligned} \gamma_i &= y_i\left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \mathbf{x}_i + \frac{b}{\|\mathbf{w}\|}\right) \\ M &= \min_{i=1,\dots,n} \gamma_i. \end{aligned} \tag{4}$$

Dieser sog. Geometric Margin ist nun skaleninvariant. Aus den Geometric Margins M, die für jede einzelne Hyperebene berechnet wurden, wird nun das Maximum gewählt. Mit M lässt sich nun das Optimierungsproblem für den sogenannten Hard Margin Classifier formulieren als:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & M \\ \text{NB} \quad & \gamma_i \geq M, i = 1, \dots, n. \end{aligned} \tag{5}$$

Aufgrund des Zusammenhangs von (3) und (4) gilt:

$$M = \frac{F}{\|\mathbf{w}\|} \Rightarrow \begin{cases} \max\limits_{\mathbf{w},b} & \dfrac{F}{\|\mathbf{w}\|} \\ \text{NB} & \dfrac{f_i}{\|\mathbf{w}\|} \geq \dfrac{F}{\|\mathbf{w}\|},\ i = 1, \dots, n. \end{cases} \tag{6}$$

Wegen der Skaleninvarianz des Geometric Margin können $\mathbf{w}$ und b nun so gewählt werden, dass $F = 1$ gilt. Auf $\|\mathbf{w}\|$ kann in der Nebenbedingung verzichtet werden.

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \frac{1}{\|\mathbf{w}\|} \\ \text{NB} \quad & f_i \geq 1,\ i = 1, \dots, n. \end{aligned} \tag{7}$$

Es erfolgt eine Umformulierung, so dass ein konvex quadratisches Optimierungsproblem vorliegt. Dabei wird das vorliegende Maximierungsproblem in ein Minimierungsproblem umgewandelt, die Norm von $\mathbf{w}$ wird quadriert, um die Wurzel zu entfernen und das Minimierungsproblem erhält den Vorfaktor $\frac{1}{2}$. Der Vorfaktor wird verwendet, um das Optimierungsproblem mit einem QP Solver (die, in dieser Arbeit verwendete Implementierung der SVM [18] verwendet Platt's SMO Algorithmus [24]) zu lösen.
Mit $f_i = y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$, $i = 1, \ldots, n$ gilt:

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{NB} \quad & y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1,\ i = 1, \ldots, n. \end{aligned} \tag{8}$$

Da ein Optimierungsproblem unter Nebenbedingungen vorliegt, eignet sich das Verfahren der Lagrange-Multiplikatoren. $\mathcal{L}(x, \alpha) = f(x) - \alpha g(x)$ ist dabei die Lagrange-Funktion, $\nabla\mathcal{L}(x, \alpha) = \nabla f(x) - \alpha \nabla g(x)$ deren Gradient. Auf das gegebene Optimierungsproblem angewendet ergibt sich:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = f(\mathbf{w}) - \sum_{i=1}^{n} \alpha_i g_i(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i [y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) - 1]. \tag{9}$$

Für jede Nebenbedingung existiert ein Lagrange-Multiplikator α_i.

Wolfe-Dualität

Das Optimierungsproblem wird ein weiteres Mal umformuliert um ein primales (Minimierung) und ein duales Problem (Maximierung) zu erhalten. Dabei wird der Umstand ausgenutzt, dass das Maximum des dualen Problems immer kleiner oder gleich dem Minimum des primalen Problems ist. Im vorliegenden Fall liegt sogar eine starke Dualität vor, so dass das Maximum des dualen Problems dem Minimum des primalen Problems entspricht. Mit Formel (9) ergibt sich das Problem

$$\begin{aligned} \min_{\mathbf{w},b} \max_{\alpha} \quad & \mathcal{L}(\mathbf{w}, b, \alpha) \\ \text{NB} \quad & \alpha_i \geq 0,\ i = 1, \ldots, n. \end{aligned} \tag{10}$$

Das Minimierungsproblem wird gelöst, indem die partiellen Ableitungen von $\mathcal{L}$ nach $\mathbf{w}$ und b berechnet und gleich Null gesetzt werden.

$$\begin{aligned} \mathbf{w} &= \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \\ 0 &= \sum_{i=1}^{n} \alpha_i y_i. \end{aligned} \tag{11}$$

Werden die Ergebnisse aus (11) in $\mathcal{L}$ eingesetzt, kommen sowohl $\mathbf{w}$ als auch b in der Gleichung nicht mehr vor. In Verbindung mit der Formulierung des Optimierungsproblems

in (10) und Hinzunahme der weiteren Nebenbedingung, dass die Summe der Produkte $\alpha_i y_i$ gleich Null ist, gilt für das Optimierungsproblem:

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\ \text{NB} \quad & \alpha_i \geq 0 \\ & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} \tag{12}$$

Da eine Nebenbedingung eine Ungleichheitsbedingung enthält, müssen zudem die Karush-Kuhn-Tucker (KTT) Bedingungen erfüllt sein. Diese sind ebenfalls gegeben. Aus diesen Bedingungen lässt sich folgende Erkenntnis gewinnen:

1) $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) - 1 \geq 0$ und $\alpha_i \geq 0$, $\forall i = 1, \ldots, n$

2) $\alpha_i[y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) - 1] = 0$, $\forall i = 1, \ldots, n$

$$\alpha_i > 0 \Rightarrow y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = 1$$
$$\alpha_i = 0 \Rightarrow y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) > 1.$$

Der Multiplikator α_i ist nur dann größer Null, wenn $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = 1$. Beobachtungen, die diese Gleichung erfüllen, nennt man Support Vector [32].

Durch Lösung des Optimierungsproblems sind sämtliche α_i bekannt, so dass mit (11) der Vektor $\mathbf{w}$ bestimmt werden kann. Für die Berechnung von b ergibt sich nach Umformen der Gleichung $1 = y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)$:

$$b = y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle. \tag{13}$$

Hierbei wird allerdings, anstatt eines zufällig gewählten Support Vectors $\mathbf{x}_i$, der Durchschnitt über alle Support Vektoren gebildet, da man so ein numerisch stabileres Ergebnis erhält [3]

$$b = \frac{1}{N_S} \sum_{i \in S} (y_i - \langle \mathbf{w}, \mathbf{x}_i \rangle). \tag{14}$$

Dabei bezeichnet S die Indizes der Support Vektoren und N_S deren Anzahl. Als Konsequenz der Formulierung als duales Problem werden auch bei der Entscheidungsfunktion nur Support Vektoren benötigt, da die α_i nur für diese nicht Null sind:

$$G(\mathbf{x}_i) = sgn\left(\sum_{j \in S} \alpha_j y_j \langle \mathbf{x}_j, \mathbf{x}_i \rangle + b \right). \tag{15}$$

2.1.3 Support Vector Classifier

Die bisher behandelte Methode zur Klassifizierung von Daten geht davon aus, dass sich die Daten linear trennen lassen. Diese Annahme bereitet in der Realität allerdings Schwierigkeiten. So ist es durchaus denkbar, dass der resultierende Margin sehr gering ausfällt, wenn die Datenpunkte in den Trainingsdaten nahe beeinander liegen. Wird der blaue, eingekreiste Punkt in Abb. 1 etwas in Richtung der roten Punkte verschoben, so verkleinert sich der Margin, dargestellt durch die gestrichelten Linien, merklich (vgl. Abb. 2). Bei einem solchen Modell ist dann zu vermuten, dass im Allgemeinen keine guten Vorhersageergebnisse zu erwarten sind. Sollten sich die einzelnen Klassen hingegen sogar überlappen, wäre es für den Hard Margin Classifier gar nicht erst möglich, eine Hyperebene zu finden (Abb. 3). Um diesem Problem zu begegnen wird der sogenannte Support Vector Classifier (SVC), auch Soft Margin Classifier genannt, eingeführt. Der SVC lässt dabei Fehler in der Klassifizierung zu, allerdings sollen diese so gering wie möglich gehalten werden [32].

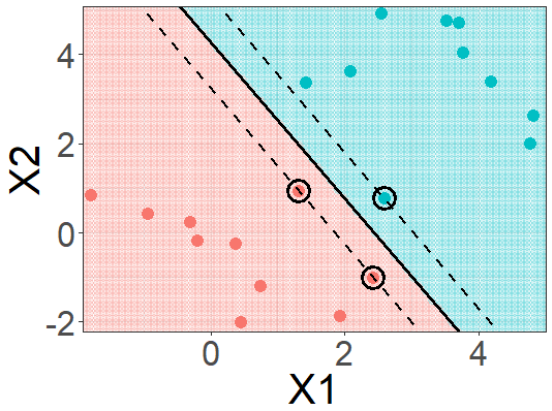


Abbildung 1

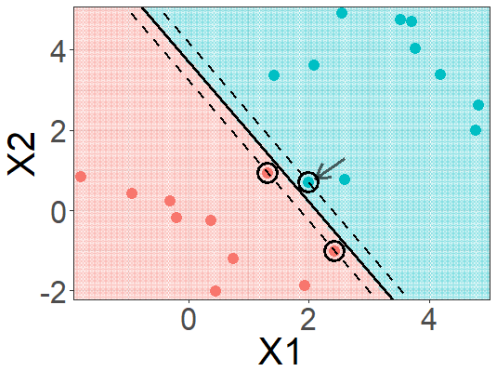


Abbildung 2

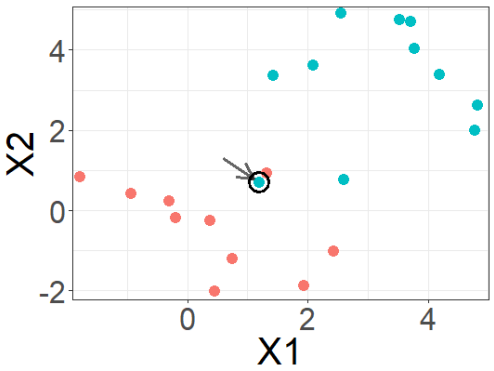


Abbildung 3

Slack Variables

Ermöglicht werden die Fehler in der Klassifizierung durch sogenannte Slack Variables ξ_i. Die ursprüngliche Nebenbedingung $y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) \geq 1$ wird nun dadurch modifiziert, dass ξ_i auf der rechten Seite der Ungleichung subtrahiert wird. Dadurch wird immernoch versucht, die Entscheidungsgrenze zu maximieren, während jedoch Punkte eine leichte Penalisierung erhalten, wenn sie auf der falschen Seite liegen [3]. Es ist nun möglich, dass einzelne Punkte innerhalb des Margins und damit zu nah an der Hyperebene oder gar auf der anderen Seite der Hyperebene liegen. Um zu verhindern, dass zu große Werte für ξ_i gewählt werden, wird die zu minimierende Funktion dahingehend angepasst, dass die Summe über sämtliche ξ_i dazu addiert wird. Darüber hinaus wird diese Summe mit dem Kostenfaktor C multipliziert um eine gewisse Kontrolle über die Strafterme zu erhalten. Damit lautet die zu minimierende Funktion

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i \tag{16}$$

mit den Nebenbedingungen

$$\begin{aligned} y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) &\geq 1 - \xi_i \\ \xi_i &\geq 0,\ i = 1, \ldots, n. \end{aligned} \tag{17}$$

Analog zur Dualform in (12) ergibt sich das Optimierungsproblem

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\ \text{NB} \quad & 0 \leq \alpha_i \leq C,\ i = 1, \ldots, n \\ & \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} \tag{18}$$

Die erste Nebenbedingung verändert sich nun dahingehend, dass sämtliche α_i nach oben durch C beschränkt sind. Diese Nebenbedingung wird auch als Box Constraint bezeichnet [3]. Die Funktionsweise des Kostenparameters C stellt sich wie folgt dar: Wird für C ein kleiner Wert gewählt, so verbreitert das den Margin und es werden relativ viele Fehler toleriert. Wird der Wert hingegen stetig erhöht, so werden immer weniger Fehler durch den Classifier erlaubt (Abb. 4, 5 und 6). Durch das Setzen von $C = \infty$ ergibt sich in der Theorie wieder der Hard Margin Classifier. In der Praxis genügt jedoch schon ein hinreichend großer Wert von C aus, um einen Classifier zu bilden, bei dem keinerlei Trainingsfehler mehr zugelassen werden (vgl. Abb. 7).

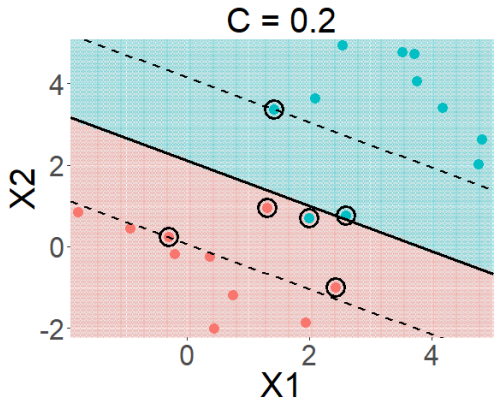


Abbildung 4

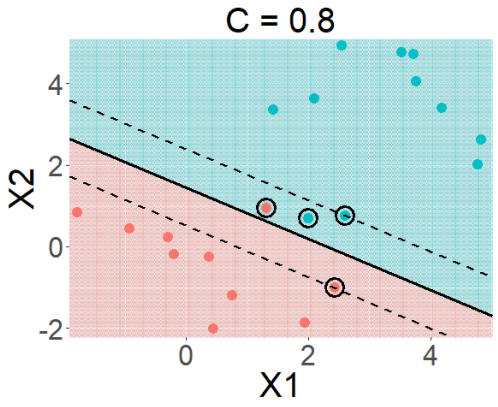


Abbildung 5

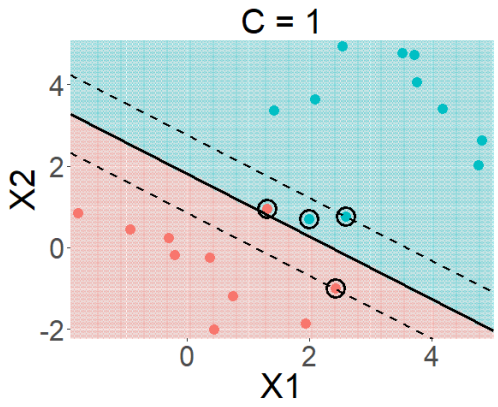


Abbildung 6

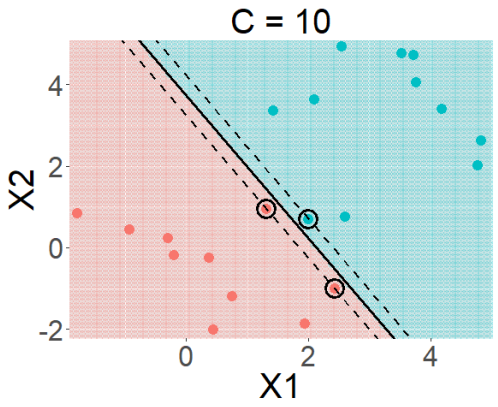


Abbildung 7

2.1.4 Support Vector Machine

Mit dem SVC ist es zwar nun möglich Fehler zuzulassen, allerdings wird weiterhin mit einer linearen Entscheidungsgrenze gearbeitet. In vielen Konstellationen liefert eine lineare Trennung keine befriedigenden Ergebnisse [17]. Eine Erweiterung der Methodik, um auch nichtlineare Entscheidungsgrenzen zu ermöglichen, trägt den Namen Support Vector Machine. Dabei ist die zugrundeliegende Idee der SVM auf Cover's Theorem [9],[30] zurückzuführen. Es besagt, dass sich Daten, die nicht linear trennbar sind, mit hoher Wahrscheinlichkeit linear trennen lassen, wenn sie in einen höherdimensionalen Raum projiziert werden. Abbildung 8 veranschaulicht dieses Vorgehen. Während es im linken Bild nicht möglich ist, eine einzige Hyperebene (in diesem Fall ein Punkt) zu finden, die rote und blaue Punkte voneinander trennt, so ist es im rechten Bild, durch Transformation der Variable x zu (x, x^2), möglich eine Hyperebene (in dem Fall eine Linie) zu finden, die in der Lage ist, die Daten linear zu separieren. Die Transformation, die für die Projektion in einen höherdimensionalen Raum benötigt wird, gestaltet sich allerdings schwierig, da ein gewisses Maß an Erfahrung nötig ist [19]. Ein weiterer

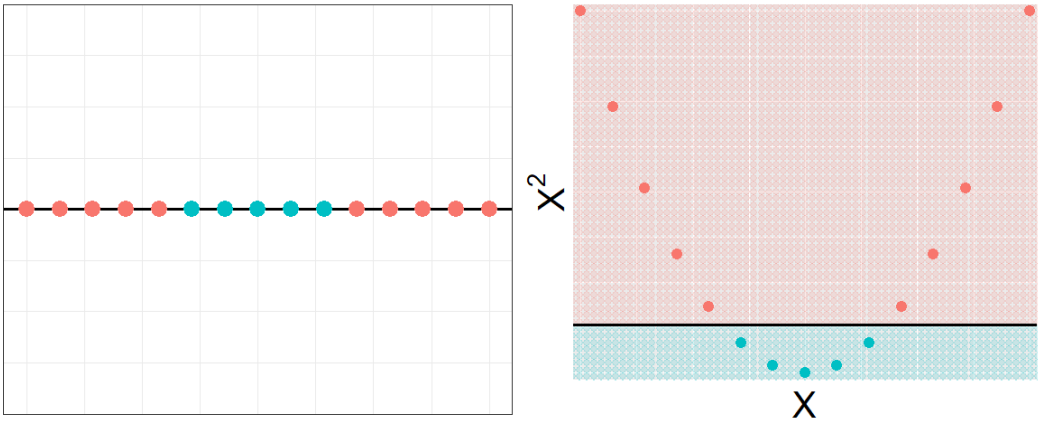


Abbildung 8: Beispiel für Cover's Theorem

Nachteil besteht darin, dass sämtliche Trainingsbeobachtungen tatsächlich transformiert werden müssen, so dass die Größe des Datensatzes und die Komplexität der Transformation als solches einen erheblichen Einfluss auf die benötigte Rechenzeit haben [17]. An dieser Stelle kommt der sogenannte Kernel-Trick ins Spiel.

Der Kernel

Zur Veranschaulichung der Funktionsweise eines Kernels soll ein kurzes Beispiel aus [12] wiedergegeben werden, dass die Transformation

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3,\ \phi(x_{i1}, x_{i2}) = (x_{i1}^2, \sqrt{2}x_{i1}x_{i2}, x_{i2}^2) = \mathbf{x}_i^* \tag{19}$$

verwirklicht. Bei dieser Transformation werden, für jede Trainingsbeobachtung, insgesamt 4 Multiplikationen durchgeführt. Zur Berechnung des Skalarprodukts $\langle \mathbf{x}_i^*, \mathbf{x}_j^* \rangle$ werden zwei transformierte Vektoren benötigt. Damit erhöht sich die Anzahl der Rechenoperation bereits auf 8. Das Skalarprodukt

$$\langle \mathbf{x}_i^*, \mathbf{x}_j^* \rangle = x_{i1}^* x_{j1}^* + x_{i2}^* x_{j2}^* + x_{i3}^* x_{j3}^* \tag{20}$$

benötigt selbst weitere 5 Rechenoperationen, so dass ingesamt 13 Operationen durchgeführt werden. Im Vergleich dazu nun die Berechnung mittels eines Kernels:

$$\begin{aligned} K(\mathbf{x}_i, \mathbf{x}_j) &= \langle \mathbf{x}_i, \mathbf{x}_j \rangle^2 \\ &= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 \\ &= x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + x_{i2}^2 x_{j2}^2 \\ &= \langle (x_{i1}^2, \sqrt{2}x_{i1}x_{i2}, x_{i2}^2), (x_{j1}^2, \sqrt{2}x_{j1}x_{j2}, x_{j2}^2) \rangle \\ &= \langle \mathbf{x}_i^*, \mathbf{x}_j^* \rangle. \end{aligned} \tag{21}$$

Die beiden grauen Zeilen werden für die Bestimmung der Anzahl der Rechenoperationen nicht herangezogen und dienen lediglich der Veranschaulichung. Das Ergebnis des Kernels stimmt mit dem Ergebnis der Berechnung durch Transformation überein, obwohl die Beobachtungen nicht explizit transformiert wurden. Die Transformation selbst muss nicht einmal bestimmt werden, nur die Kernelfunktion selbst muss bekannt sein [13]. Die zu maximierende Funktion des Optimierungsproblems in (18) und die Entscheidungsfunktion (15) werden nun lediglich dadurch modifiziert, dass $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ durch die Kernelfunktion $K(\mathbf{x}_i, \mathbf{x}_j)$ ersetzt wird.

Drei, der am häufigsten verwendeten, Kernel [13] sollen nachfolgend kurz dargestellt werden.

Linearer Kernel $K(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$ Die Support Vector Machine mit linearem Kernel entspricht dem Support Vector Classifier.

Polynomkernel $K(\mathbf{x}, \mathbf{x}') = (\mathbf{scale}\langle \mathbf{x}, \mathbf{x}' \rangle + \mathbf{offset})^{\mathbf{degree}}$ Dieser Kernel erlaubt nun auch nichtlineare Entscheidungsgrenzen. **scale**, **offset** und **degree** sind dabei die Parameter des Kernels. Mit steigendem Polynomgrad wächst die Gefahr des Overfittings. In Abbildung 10 ist die unbefriedigende Lösung des Datenproblems aus Abbildung 9 zu sehen, wenn ein linearer Kernel verwendet wird. Die Abbildungen 11 und 12 veranschaulichen die Problematik des Overfittings mit steigendem Polynomgrad.

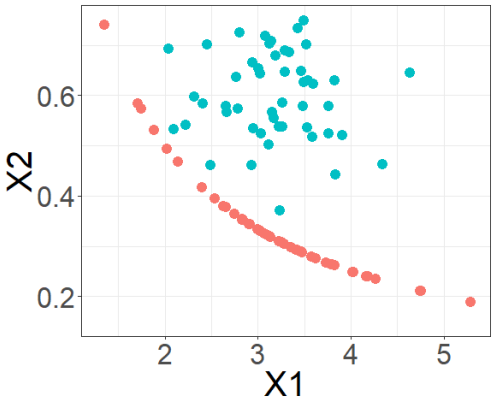


Abbildung 9

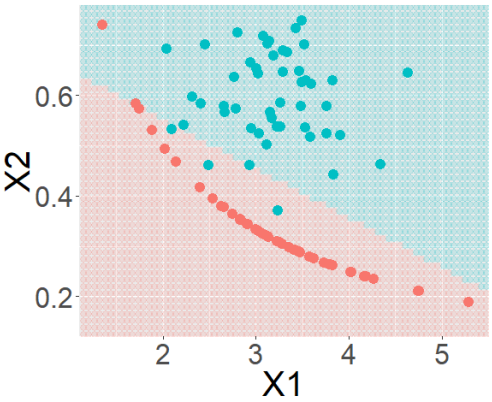


Abbildung 10

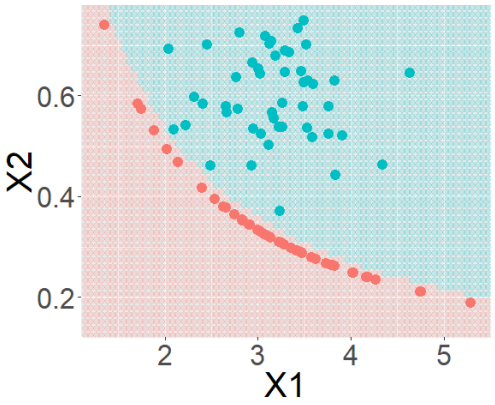


Abbildung 11: degree=2

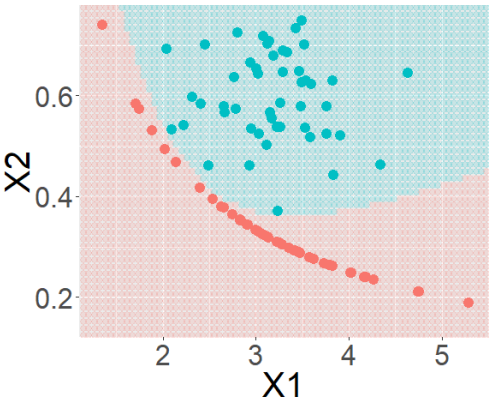


Abbildung 12: degree=4

RBF-Kernel $K(\mathbf{x}, \mathbf{x}') = \mathbf{exp}(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2)$ Der RBF-Kernel ist ein weiterer nicht-linearer Kernel, der die Besonderheit besitzt, dass er in einen unendlich-dimensionalen Raum projiziert [17],[19]. Ein Nachweis für diese Behauptung ist unter [31] zu finden. Zur Veranschaulichung sollen die Abbildungen 13 und 14 dienen. Dabei handelt es sich um Probleme, bei denen auch ein Polynomkernel keine befriedigenden Ergebnisse mehr liefern würde.

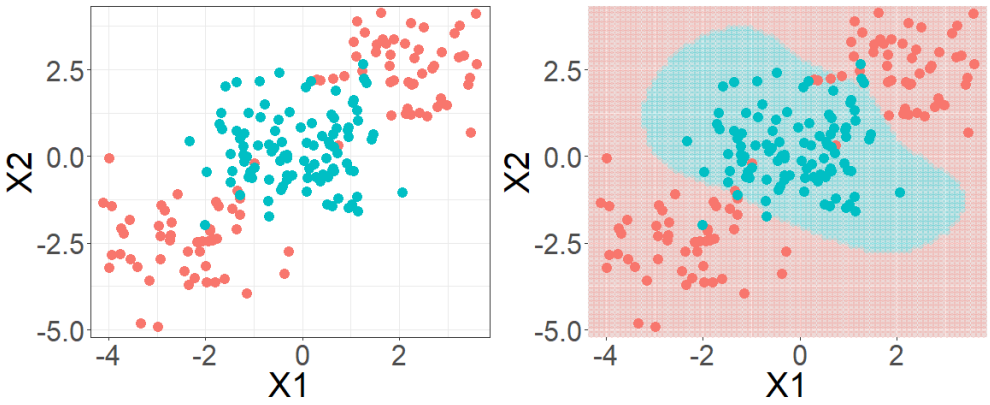


Abbildung 13: Beispiel eines RBF-Kernels

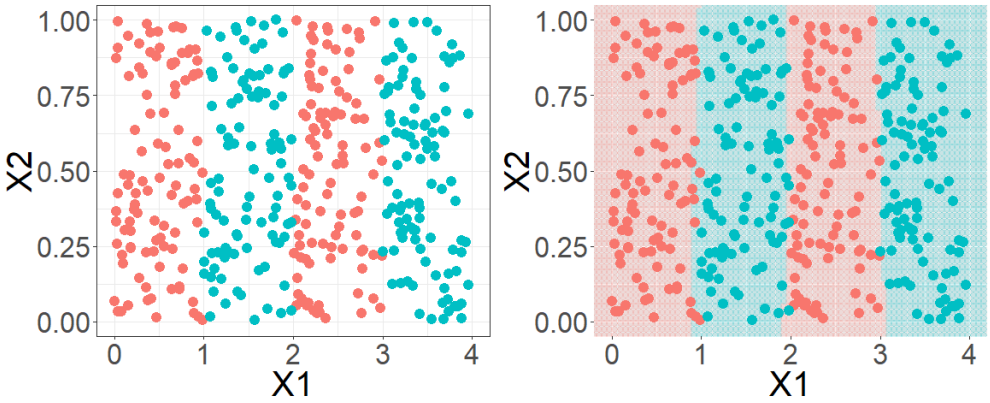


Abbildung 14: weiteres Beispiel RBF-Kernel

Es existieren noch viele weitere Kernel (vgl. z.B. die implementierten Kernels der Funktion `ksvm()` aus dem Paket **kernlab** [18]). Die Wahl des richtigen Kernels gestaltet sich mitunter etwas schwierig, da Erfahrung und spezielles Wissen über die Struktur der Daten nötig ist. Laut [14] ist der RBF-Kernel jedoch eine gute erste Wahl.

2.1.5 Multi-Class SVM

Die Support Vector Machine wurde ursprünglich für ein binäres Klassifizierungsproblem konzipiert [32]. Aufgrund einer Vielzahl von mehrkategoriellen Problem besteht reges

Interesse daran, die Support Vector Machine auch auf den Fall mit mehr als zwei Klassen auszuweiten. Die verschiedenen Methoden lassen sich dabei in zwei Kategorien, indirekte und direkte Methoden [33], unterteilen. Beispiele für indirekte Methoden sind One-versus-Rest (OvR) und One-versus-One (OvO). Für die direkten Ansätze sei hier sowohl die Methode nach Weston und Watkins (WW) [34], als auch nach Crammer und Singer (CS) [10] erwähnt.

One-vs-Rest Multi-Class SVM

Die wohl erste Implementierung, einer Support Vector Machine zur Klassifizierung von mehr als zwei Klassen, ist der One-vs-Rest Ansatz [15]. Bei dieser Variante werden insgesamt K SVMs trainiert, wobei K der Anzahl der Klassen entspricht. Das k-te SVM-Modell wird trainiert, indem die Trainingsbeobachtungen der k-ten Klasse das Label $+1$ und die restlichen $(K-1)$ Klassen das Label -1 erhalten. Eine Trainingsbeobachtung $\mathbf{x}_i$ gehört dann zu der Klasse k, wenn die Entscheidungsregel einen positiven Wert annimmt. Allerdings führt diese Vorgehensweise dazu, dass bei der Klassifizierung einer Testbeobachtung mehrere oder gar keine Klasse gefunden wird, wenn die K einzelnen Entscheidungsfunktionen zur Klassifizierung herangezogen werden [3]. Im ersten Fall kann die Beobachtung nicht eindeutig zugeordnet werden und im letzteren kommt erst gar keine Klasse in Frage. Um diesem Problem Rechnung zu tragen, wird auf die Signumsfunktion in der Entscheidungsregel $G(x)$ aus (1) verzichtet, so dass nun der eigentliche Wert der Entscheidungsfunktion $f(x)$ betrachtet wird. Eine Testbeobachtung $\mathbf{x}_i$ wird der Klasse k zugeteilt, wenn deren Entscheidungsfunktion maximal wird.

$$f(\mathbf{x}_i) = \max_{k=1,\dots,K} f_k(\mathbf{x}_i) \tag{22}$$

Beobachtungen die eigentlich keiner Klasse zugeteilt würden, werden nun der Klasse zugeteilt, der sie am nächsten sind. Problematisch ist bei diesem Ansatz, dass die Trainingsdaten der einzelnen binären Classifier unbalanciert sind, erst recht, wenn die K Klassen ursprünglich ausgeglichen waren [33].

One-vs-One Multi-Class SVM

Ein weiterer Ansatz ist die One-vs-One Methode. Bei diesem Ansatz kommt es zu einem paarweisen Vergleich. Es wird bei K Klassen ein Classifier pro Klassenpaar, also insgesamt $\frac{K(K-1)}{2}$, trainiert. Ein Nachteil gegenüber OvR ist, dass die Anzahl der zu trainierenden Classifier deutlich stärker ansteigt, wenn viele Klassen vorliegen und dies negative Auswirkungen auf die Trainingszeit haben kann. Allerdings relativiert sich dieser Nachteil etwas, da zum einen in der Regel weniger Beobachtungen für die einzelnen Classifier herangezogen werden und zum anderen die Anzahl der gefundenen Support

Vektoren meist geringer ist [20]. Hsu und Lin konnten in [15] nachweisen, dass die Trainingszeit tatsächlich geringer ausfällt, obwohl $\frac{K(K-1)}{2}$ Classifier trainiert werden müssen. Die Klassifizeriung geschieht dann über einen einfachen Mehrheitsentscheid. Eine Testbeobachtung fällt in die Klasse, der sie beim Durchlaufen der einzelnen Classifier am häufigsten zugeteilt wurde. Auch hier ist es theoretisch möglich, dass keine eindeutige Entscheidung möglich ist, da eine Beobachtung mehreren Klassen gleich häufig zugeteilt wird. Es wird vorgeschlagen in diesem Fall die Klasse zu wählen, die den niedrigsten Index besitzt [15].

Weston & Watkins Multi-Class SVM

Eine direkte Methode wurde von Weston und Watkins [34] vorgeschlagen. Die Idee dahinter ist, dass nur ein einziges Optimierungsproblem gelöst werden soll. Der Grundgedanke ist dem des OvR-Ansatzes recht ähnlich. Es werden K binäre Classifier trainiert, wobei die k-te Funktion $\langle \mathbf{w}_k, \mathbf{x} \rangle + b_k$ die Klasse k von allen anderen separiert. Es existieren demnach K Entscheidungsfunktionen, die aber durch Lösung eines einzigen Optimierungsproblems erhalten werden.

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}\|\mathbf{w}_k\|^2 + C\sum_{i=1}^{n}\sum_{k\neq y_i}\xi_{i,k} \\ \text{NB} \quad & \langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle + b_{y_i} \geq \langle \mathbf{w}_k, \mathbf{x}_i \rangle + b_k + 2 - \xi_{i,k} \\ & \xi_{i,k} \geq 0 \quad i = 1, \ldots, n \quad k \in \{1, \ldots, K\} \setminus y_i. \end{aligned} \tag{23}$$

Die Entscheidungsfunktion lautet dann

$$\underset{k}{\operatorname{argmax}} \left(\langle \mathbf{w}_k, \mathbf{x} \rangle + b_k \right) \quad k = 1, \ldots, K. \tag{24}$$

Crammer & Singer Multi-Class SVM

Ein weiterer Ansatz, der nur ein einziges Optimierungsproblem zur Lösung benötigt, ist der von Crammer und Singer[10]. Das Optimierungsproblem stellt sich wie folgt dar:

$$\begin{aligned} \min_{\mathbf{w},\xi} \quad & \frac{1}{2}\|\mathbf{w}_k\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{NB} \quad & \langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle - \langle \mathbf{w}_k, \mathbf{x}_i \rangle \geq 1 - \delta_{y_i,k} - \xi_i \\ & i = 1, \ldots, n \quad k \in \{1, \ldots, K\} \end{aligned} \tag{25}$$

Dabei ist $\delta_{y_i,k}$ das Kronecker-Delta (definiert als 1, falls $y_i = k$ und 0 andernfalls). Damit lässt sich zugleich erklären, wieso in (25) die Nebenbedingung $\xi_i \geq 0$ im Gegensatz zu (23) fehlt. Für den Fall, dass $y_i = k$ gilt, ergibt sich die Nebenbedingung in (25) zu $0 \geq 0 - \xi_i$ und damit zu $\xi_i \geq 0$. Der Hauptunterschied zwischen CS und WW ist,

dass in (25) nur n Slack Variables verwendet werden, während (23) insgesamt $n(K-1)$, also $(K-1)$ pro Datenpunkt, benötigt [28]. Dies soll zu einem Algorithmus mit verringertem Speicherverbrauch und einer effizienteren Trainingszeit führen [10],[19]. Bei dem Vergleich verschiedener Multi-Class Support Vector Machines in [15] wurde allerdings festgestellt, dass die Methode von Crammer & Singer sehr langsam ist, wenn ein sehr großer Wert für den Parameter C gewählt wird.

Weitere Multi-Class SVM Ansätze

Die Liste der vorgestellten Multi-Class SVMs erhebt keinen Anspruch auf Vollständigkeit. Ein weiteres Beispiel wäre die in [25] vorgestellte Directed Acyclic Graph SVM (DAGSVM), die eine Erweiterung des OvO-Ansatzes darstellt. In [15] ist zu erkennen, dass sich die Testzeit gegenüber OvO mit der DAGSVM verringern lässt. Wang und Xue hingegen haben in [33] die Simplified MultiClass SVM (SimMSVM) vorgeschlagen, die eine Erweiterung des Ansatzes von Crammer und Singer darstellt und im Vergleich dazu den Trainingsprozess deutlich beschleunigen kann.

Es lässt sich festhalten, dass die direkten Methoden zumeist deutlich mehr Rechenzeit benötigen als die indirekten Ansätze. Allerdings wird durch dieses Mehr an Rechenzeit kein substanzieller Zugewinn an Vorhersagegenauigkeit erzielt. Deshalb kommen verschiedene Autoren zu dem Ergebnis, dass kein Ansatz existiert, dessen Performance die der anderen Methoden überflügelt und OvO in der Praxis wohl zu bevorzugen ist [15],[33],[30]. Aus diesem Grund soll in der vorliegenden Arbeit lediglich die Methode One-vs-One, für den Vergleich mit anderen Klassifizierungsmethoden, herangezogen werden.

2.2 Random Forests

Die erste Methode, die für einen Vergleich mit der SVM herangezogen wird, sind Random Forests [4]. Grundlage der Random Forests sind **C**lassification **A**nd **R**egression **T**rees (CART). Solche Entscheidungsbäume sind allerdings sehr instabil, da sie eine hohe Varianz besitzen. Dieser Tatsache wird durch Bootstrap Aggregation (Bagging) Rechnung getragen. Die grunsätzliche Idee des Baggings besteht darin, dass viele instabile Modelle mit relativ geringem Bias gemittelt werden, um so eine Reduktion der Varianz zu erzielen. Daher eignen sich Trees für Bagging, da sie, sollten sie tief genug gewachsen sein, einen geringen Bias aufweisen. Allerdings sind die generierten Bäume nur identisch aber nicht unabhängig verteilt [13]. Die daraus resultierende Korrelation der einzelnen Bäume limitiert die Reduktion der Varianz. Um die Korrelation zwischen den Bäumen zu verringern, wird vor jedem Split nur ein Teil der vorhandenen Features

zufällig ausgewählt und verwendet [13]. Da der Random Forest in [11] sehr gut abschneiden konnte, erscheint er ein passender Kandidat für den Vergleich mit der Multi-Class Support Vector Machine zu sein.

2.3 Extreme Gradient Boosting

Zusätzlich muss sich die Multi-Class SVM mit dem XGBoost [7] messen. Der XGBoost stellt eine Erweiterung des Boostings bzw. des Gradient Boostings dar. Die grundsätzliche Idee des Boostings besteht darin, durch die Kombination vieler schwacher Classifier einen starken Classifier zu bilden [13]. Das Boosting funktioniert dabei sequentiell. In jeder Iteration werden Informationen aus dem vorhergehenden Baum benutzt, um die Rate der Fehlklassifizierung Stück für Stück zu verringern. Dabei wird den falsch klassifizierten Beobachtungen im nachfolgenden Baum ein höheres Gewicht zugeteilt [29].

Die primäre Erweiterung, gegenüber anderen Implementierungen des Gradient Boostings, liegt beim XGBoost in seiner Effizienzsteigerung bei extrem großen Datensätzen. Auch wenn in dieser Arbeit keine derart großen Datensätze verwendet werden, so hat der XGBoost gegenüber Gradient Boosting einen weiteren Vorteil. Der XGBoost zeichnet sich zusätzlich dadurch aus, dass er noch mehr Regularisierung benutzt, um Overfitting zu vermeiden [6],[29]. So wird unter anderem auch die Idee des Random Forest entliehen, nur einen Teil der Features für das Fitting zu verwenden, um die Gefahr des Overfittings weiter zu dezimieren [7].

Da der XGBoost sich derzeit sehr großer Beliebtheit erfreut und bereits sehr erfolgreich in verschiedenen Machine Learning Challenges eingesetzt wurde [36], scheint er eine ausgesprochen gute Wahl für den Vergleich darzustellen.

3 Vergleich der einzelnen Classifier

In diesem Abschnitt kommt es nun zu dem Vergleich der Classifier Random Forest und XGBoost mit der Multi-Class Support Vector Machine. Zuerst wird ein Einblick in die verwendete Software gewährt, gefolgt von einer kurzen Auflistung der verwendeten Datensätze. Im Anschluss daran wird erläutert, wie die einzelnen Methoden mit den voreingestellten Hyperparametern bei den einzelnen Datenproblemen abschneiden. Schließlich wird dann auf das Tuning dieser Parameter eingegangen. Nach der Optimierung der Hyperparameter wird dann die Performance anhand der ermittelten Accuracy und der Zeitverbrauch des Tunings bewertet.

3.1 Überblick über verwendete Software und Datensätze

Für die Analysen in dieser Arbeit wurde ausschließlich die Programmiersprache **R** (3.5.3) [26] verwendet. Dabei wird hauptsächlich auf das Package **mlr** (2.13) [1] zurückgegriffen. Das Paket gewährleistet, dass verschiedene Aufgaben im Bereich des Machine Learnings in einem einheitlichen Rahmen ablaufen. Das Trainieren einzelner Learner (hier: Classifier), Methoden zur Variablenselektierung, unterschiedliche Resampling-Strategien und das Tunen von Hyperparametern folgt einer gleichbleibenden und damit leichter nachvollziehbaren Syntax, anstatt den Voraussetzungen und Limitierungen einzelner Programmpakete anheim zu fallen, da **mlr** die Nutzung verschiedener Algorithmen aus diversen R-Paketen ermöglicht.

3.1.1 Weitere verwendete R-Packages

Mit dem Package **kernlab** (0.9-27) [18] wird die Support Vector Machine realisiert. Dabei nutzt **kernlab** intern die in C und C++ implementierten Libraries **LIBSVM** [5] und **BSVM** [16]. Für die Klassifizierung von mehr als 2 Klassen verwendet das Paket standardmäßig One-Versus-One. Es sind allerdings auch die Methoden nach Crammer & Singer (`spoc-svc`) und Weston & Watkins (`kbb-svc`) verfügbar. Für die Implementierung von Random Forests wird **ranger** (0.11.2) [35] gewählt. **ranger** wurde in C++ programmiert und die R-Version mit dem Package **Rcpp** umgesetzt. Die Verwendung von Extreme Gradient Boosting wird durch **xgboost** (0.82.1) [8] ermöglicht. Die grundlegende Basis ist ebenfalls wieder in C++ implementiert [22].

Im Bereich des Tunings der Hyperparameter einzelner Classifier wurden zusätzlich die Pakete **mlrMBO** (1.1.2) [2] und **mlrHyperopt** (0.1.1) [27] verwendet. Mit dem Package **mlrMBO** wird die modelbasierte Optimierung von Hyperparametern ermöglicht,

während mittels **mlrHyperopt** vorgefertigte Parameterräume für das Tuning extrahiert werden.

3.1.2 Übersicht der Datensätze

Für den nachfolgenden Vergleich der einzelnen Klassifizierungsmethoden wurden verschiedene Datensätze ausgewählt (vgl. Tabelle 1), die allesamt Teil des *UCI Machine Learning Repository* sind [23]. Die Datensätze **Glass**, **Letter**, **Satellite**, **Shuttle**, **Vehicle**, **Vowel** wurden über das R-Paket **mlbench** (2.1-1) [21] aufgerufen. Der Datensatz **Wave** wurde mit der Funktion `mlbench.waveform()` generiert, die auf Grundlage der Implementierung des Waveform Generators in C basiert, der ebenfalls vom *UCI Machine Learning Repository* stammt. Der Datensatz **Iris** entstammt dem R-Paket **base**.

Datensatz	*#Beobachtungen (n)*	*#Klassen (K)*	*#Attribute (p)*
Glass	214	6	9
Iris	150	3	4
Letter	20.000	26	16
Satellite	6.435	6	36
Shuttle	58.000	7	9
Vehicle	846	4	18
Vowel	990	11	9(10)
Wave	5.000	3	21

Tabelle 1: Verwendete Datensätze

Bei dem Datensatz **Vowel** liegen 10 Attribute vor. Es werden allerdings nur 9 der Features verwendet, da die Variable `V1` keinerlei zusätzlichen Informationsgewinn bietet (vgl. Appendix). Bei dieser Variable handelt es sich um einen Index, der Auskunft darüber gibt, welche Person den entsprechenden Vokal eingesprochen hat.

3.2 Baseline Performance

Anfänglich soll die Performance der Classifier untersucht werden, wenn die voreingestellten Hyperparameter angewendet werden. Die Datensätze wurden mit Hilfe der **mlr**-Funktionen `makeResampleDesc()` und `makeResampleInstance()` zerteilt, so dass der Trainingsdatensatz 2/3 aller Beobachtungen enthält und für jede Auswertung exakt die gleichen Splits verwendet werden.

Die relevanten und voreingestellten Parameter für die Support Vector Machine sind `C = 1`, `kernel = 'rbfdot'` und `sigma` (entspricht γ in der Definition des RBF-Kernels in Abschnitt 2.1.4). Die Funktion `ksvm()` verwendet für `sigma` allerdings keinen festen voreingestellten Wert, sondern berücksichtigt die empirische Beobachtung, dass gute Werte für das `sigma` des RBF-Kernels zwischen dem 10%- und 90%-Quantil der $\|x-x'\|$-Statistik liegen. `ksvm()` greift auf die Funktion `sigest()` zurück, um aus einem Teil der Daten die Statistik zu errechnen und wählt dann den Median der Beobachtungen, die innerhalb der Quantile liegen, als Defaultwert für `sigma` (vgl. die Hilfe von `?ksvm`)[18].

Die in dieser Arbeit relevanten Parameter der Funktion `ranger()` sind die Anzahl der verwendeten Features je Split `mtry` $= \lfloor\sqrt{p}\rfloor$ und `min.node.size = 1`. Der Wert für die Anzahl der Bäume `num.trees` wird bei 500 belassen.

Die Parameter des XGBoost, die später getuned werden sollen, sind `eta = 1`, `gamma = 0`, `colsample_bytree = 1`, `max_depth = 6`, `min_child_weight = 1`, `subsample = 1` und abweichend von der Defaulteinstellung des Packages **mlr** der Wert 100 für `nrounds`. Im Abschnitt "Tuning der Hyperparameter" wird noch genauer auf die einzelnen Parameter eingegangen.

Einen Überblick der Ergebnisse liefert Tabelle 2. Es fällt auf, dass der Random Forest, vor Tuning der einzelnen Classifier, in fünf von acht Fällen die beste Performance erzielt und dreimal als einziger die höchste Accuracy aufweist. Die Multiclass-SVM schafft bei drei von acht Datenproblemen die beste Performance, in zwei Fällen exklusiv. Der XGBoost erreicht bei zwei Datensätzen die höchste Accuracy, in einem Fall zusammen mit dem Random Forest. Dabei darf aber nicht übersehen werden, dass die drei Classifier grundsätzlich zu vergleichbaren Ergebnissen kommen und die Differenzen teilweise marginal ausfallen. So ist beispielsweise die SVM beim Datensatz **Shuttle** nur um 0.16% schlechter als Random Forest oder XGBoost.

Eine deutlich bessere Performance liefert der Random Forest allerdings bei den Datenproblemen **Glass** und **Vowel**, wobei bei **Glass** nur die SVM deutlich schlechter

Datensatz	*SVM (OvO)*	*Random Forest*	*XGBoost*
Glass	66.22	**74.32**	72.97
Iris	**96.00**	**96.00**	94.00
Letter	92.46	95.55	**95.56**
Satellite	89.75	**91.01**	90.87
Shuttle	99.82	**99.98**	**99.98**
Vehicle	**79.08**	76.24	76.95
Vowel	82.73	**91.21**	84.55
Wave	**86.08**	84.88	85.12

Tabelle 2: Accuracy der einzelnen Classifier mit Defaultwerten in %.
Beste Ergebnisse fett gedruckt.

abschneidet. XGBoost und Random Forest sind beim Datenproblem **Letter** fast gleich stark, jedoch liegt die SVM, mit einer Differenz von 3.1%, deutlich zurück. Im Fall des **Vehicle**-Datensatzes kann die Multiclass-SVM am deutlichsten überzeugen.

3.3 Tuning der Hyperparameter

Das Tuning der Hyperparameter wird mittels nested Resampling durchgeführt. Dabei wird der Holdout-Split verwendet, der bereits im vorhergehenden Unterabschnitt verwendet wurde. Auf den 2/3 der Trainingsdaten wird dann, für jede getestete Parameterkombination, eine Kreuzvalidierung durchgeführt. Dabei wird für die meisten Datensätze eine 10-fold Cross-Validation gewählt und gegebenenfalls stratifiziert, um der Unbalanciertheit mancher Datensätze Rechnung zu tragen. Lediglich für die Datensätze **Glass**, **Letter** und **Shuttle** wurde von diesem Vorgehen abgewichen. Durch die Beachtung der ungleichen Klassenverteilung bei dem Problem **Glass** ist nur noch eine 6-fold Cross-Validation möglich. Bei **Letter** und **Shuttle** wurde aufgrund der Größe beider Datensätze eine 5-fold Cross-Validation gewählt.

3.3.1 Die einzelnen Hyperparameter im Detail

Support Vector Machine

In Abschnitt 2.1.3 wurde bereits erwähnt, dass der Kostenparameter `C` kontrolliert wie viele Fehler der Classifier auf den Trainingsdaten erlaubt, um so die Breite des Margins zu erweitern. Der Parameter `sigma` bestimmt zusätzlich die Funktionsweise der Support Vector Machine. Wird der Wert für `sigma` zu klein gewählt, dann ähnelt das Verhalten dem einer linearen SVM und die Struktur der Daten wird möglicherweise nicht mehr hinreichend berücksichtigt. Falls ein zu großer Wert für `sigma` benutzt wird, so wird das Model zu stark von einzelnen Support Vektoren beeinflusst [19]. Dies lässt sich damit begründen, dass nur nahegelegene Trainingsbeobachtungen einen Einfluss auf die Klassifizierung einer Testbeobachtung haben [17].

Random Forest

Der Parameter `mtry` bestimmt die Anzahl der zufällig ausgewählten Features, die für jeden Split verwendet werden, um zur Dekorrelation der einzelnen Bäume beizutragen. Der Parameter `min.node.size` bestimmt die Mindestgröße die eine Node besitzen muss. Damit kontrolliert der Parameter, wie tief die Bäume wachsen. Je kleiner der Wert von `min.node.size`, desto tiefer wächst der einzelne Baum des Random Forest [13].

XGBoost

`eta` stellt den Shrinkage Parameter dar (manchmal auch als Learning Rate bezeichnet). Die Idee dahinter ist, dass die neu hinzugefügten Gewichte mit dem Faktor `eta` modifiziert werden. Damit wird der Einfluß einzelner Bäume reduziert und gewährleistet, dass nachfolgende Bäume das Modell noch weiter optimieren können. Kleinere Werte von `eta` führen zu einer längeren Rechenzeit. Der Parameter `gamma` stellt eine weitere Form der Regularisierung dar, um Overfitting zu vermeiden. `colsample_bytree` setzt die Idee der Random Feature Selection des Random Forest um. Laut [7] soll damit das Overfitting noch weiter reduziert werden. Die Tiefe des einzelnen Trees wird durch `max_depth` gesteuert. Je größer der Wert, desto tiefer wachsen die Bäume, die dann zu einem komplexeren Modell führen. Mit `min_child_weight` wird gesteuert, ab welchem Punkt die Partitionierung gestoppt werden soll. Dazu wird die Summe des sog. "instance weight" (durch die partiellen Ableitungen zweiter Ordnung) berechnet und überprüft, ob dieser Wert `min_child_weight` unterschreitet. Der Parameter `subsample` gibt an, wie groß der Anteil an Trainingsbeobachtungen sein soll, der zum Trainieren der Bäume verwendet wird. `nrounds` bestimmt die Anzahl der Iterationen, die durchgeführt werden sollen und entspricht bei einem Klassifizierungsproblem der Anzahl der Bäume die gefittet werden.

3.3.2 Verwendung von mlrMBO und mlrHyperopt

Um einen möglichst fairen Vergleich der einzelnen Methoden zu gewährleisten, sollen alle Classifier auf die gleiche Art und Weise optimiert werden. In einem ersten Schritt werden, mittels der Funktion `getDefaultParConfig()` des Pakets **mlrHyperopt**, vorgefertigte Parameterräume für das Tuning eingelesen, die der Grundeinstellung des Pakets entsprechen. Wegen der Komplexität und der Vielzahl von Hyperparametern des XGBoost fällt die Wahl der Tuningmethode auf die modelbasierte Optimierung des Pakets **mlrMBO**. So soll gewährleistet werden, dass SVM und RF, die deutlich weniger Parameter für das Tuning benötigen, nicht dadurch einen Vorteil gegenüber dem XGBoost erhalten, dass bei ihnen Gridsearch verwendet wird.

Mit modellbasierter Optimierung werden teure Black-Box Funktionen optimiert. Es wird versucht ein oder mehrere Objective(s) mit Hilfe eines sog. Surrogate Regression Models zu verbessern. Im Bereich des Machine Learnings stellt der Learner die Black-Box Funktion und das ausgewählte Performance Measure (hier: Accuracy) das Objective dar. Die Funktionsweise lässt sich vereinfacht wie folgt darstellen [2]:
In einem Startdesign werden Punkte aus den zu tunenden Parametern ausgewählt und das Objective durch den Learner evaluiert. Mit diesem Ergebnis und den ausgewählten Punkten wird nun das Regressionsmodell trainiert. Ein Infill Criterion schlägt nun neue Punkte vor, mit denen der Classifier trainiert und evaluiert wird. Das Ergebnis wird nun, inklusive der vorgeschlagenen Punkte, dem Design hinzugefügt und dem Regressionsmodell zur Berechnung übergeben. Dieses Vorgehen wird wiederholt, bis ein bestimmtes Stopkriterium erfüllt ist (hier: Anzahl der vorgegebenen Iterationen).

Für jeden Hyperparameter setzt **mlrMBO** aufgrund des Startdesigns vier Iterationen an. Folglich besitzen RF und SVM von Haus aus, neben der vorgegebenen Anzahl an Wiederholungen, acht Iterationen, während der XGBoost 28 benötigt. Bei jedem Datenproblem wird für jeden Learner mit zehn Iterationen gestartet. Sollte sich eine Verbesserung gegenüber dem Defaultfit zeigen, so wird das Ergebnis festgehalten und es findet kein weiteres Tuning für den entsprechenden Classifier mehr statt. Bei Methoden, die sich nicht verbessern, werden die Iterationen in 10er-Schritten (bis maximal 100 Iterationen) erhöht, bis eine Verbesserung eintritt. Lediglich bei **Letter** und **Shuttle** wird maximal bis 40 Wiederholungen erhöht, um der Größe der Datensätze Rechnung zu tragen. Sollte selbst nach der maximalen Anzahl an Durchläufen keine Verbesserung eintreten, so wird im weiteren Verlauf der Arbeit mit den Default Parametern gearbeitet. Abweichend davon wird für die Messung der benötigten Zeit die Iteration verwendet, die im Tuning das beste Ergebnis erzielt hat. Mit diesem Vorgehen wird Learnern, die bspw. nach 18/38 Durchläufen bereits einen Tuningerfolg aufweisen, zwar die Möglich-

keit verwehrt, sich mit zusätzlichen Iterationen zu verbessern. Dieses Vorgehen lässt sich allerdings damit rechtfertigen, dass mit Erhöhung der Durchläufe nicht automatisch eine weitergehende Optimierung einhergeht. Das Erhöhen der Iterationen kann durchaus zu einer Verschlechterung der Accuracy führen. Es ist also nicht angebracht, auf einem Datensatz für jeden Learner die Anzahl an Wiederholungen zu wählen, die der tuningintensivste Algorithmus benötigt hat.
Für die Messung der Dauer des Tunings wird davon abgewichen, da sonst in manchen Fällen keine Werte vorliegen würden. Es erscheint sinnvoll, dann den Wert zu wählen, der innerhalb des Tuningprozesses das beste Ergebnis erzielt hat.

3.3.3 Ergebnisse des Tunings

Im folgenden Abschnitt sollen nun die Ergebnisse des Tunings genauer untersucht werden. Ein Überblick über die Ergebnisse liefert Tabelle 3. Nach Durchführung der Optimierung der Hyperparameter liefert die Support Vector Machine bei fünf von acht Datensätzen das beste Ergebnis. Bei dem Datensatz **Wave** ist allerdings zusätzlich anzumerken, dass das Tuning keine Verbesserung des ursprünglichen Ergebnisses erbracht hat. Sowohl Random Forest als auch XGBoost erzielen in zwei von acht Fällen das beste Ergebnis, allerdings nur in einem Fall als einziger Learner.

Datensatz	*SVM (OvO)*		*Random Forest*		*XGBoost*	
Glass	67.57	(+1.35)	75.68	(+1.36)	**78.38**	(+5.41)
Iris	**98.00**	(+2.00)	96.00	(+0.00)	96.00	(+2.00)
Letter	**96.87**	(+4.41)	95.67	(+0.12)	95.25	(−0.31)
Satellite	90.22	(+0.47)	**91.10**	(+0.09)	**91.10**	(+0.23)
Shuttle	99.88	(+0.06)	**99.99**	(+0.01)	99.94	(−0.04)
Vehicle	**82.98**	(+3.90)	76.95	(+0.71)	78.01	(+1.06)
Vowel	**96.97**	(+14.24)	92.42	(+1.21)	82.42	(−2.13)
Wave	**85.96**	(−0.12)	84.82	(−0.06)	85.18	(+0.06)

Tabelle 3: Ergebnisse nach Tuning in %. Beste Ergebnisse fett gedruckt. Veränderung in Klammern.

Tabelle 4 enthält eine Auflistung darüber, wieviel Zeit (in Sekunden) für das Tuningergebnis aus Tabelle 3 benötigt wurde. Zusätzlich wird in Klammern angegeben, wieviele Iterationen, exklusive der Durchläufe des Startdesigns, dem einzelnen Learner zugeteilt wurden. Ohne die Anzahl der Iterationen zu berücksichtigen, lässt sich festhalten, dass das Tuning der Support Vector Machine bei fünf Datenproblemen am schnellsten abgeschlossen war. Der Random Forest benötigt bei **Glass** und **Letter** am wenigsten Zeit und XGBoost ist nur bei dem Datensatz **Shuttle** am schnellsten. Besonders Hervorzuheben sind dabei die Ergebnisse für **Shuttle** und **Vehicle**. Bei diesen zwei Datenproblemen ist jeweils einer der Learner deutlich schneller als der Rest und das bei einer identischen Anzahl an Iterationen. Bei **Letter** ist anzumerken, dass der XGBoost mehr Iterationen als Support Vector Machine und Random Forest benötigt hat und es zusätzlich mehr Iterationen für das Startdesign bedarf, weshalb vorerst nur die eindeutig schnellere Rechenzeit des Random Forest gegenüber der Support Vector Machine hervorgehoben werden soll.

Datensatz	*SVM (OvO)*	*Random Forest*	*XGBoost*
Glass	35.10 (90)	**9.83** (10)	29.97 (10)
Iris	**3.35** (10)	8.38 (10)	18.09 (10)
Letter	2198.74 (10)	**470.96** (10)	5235.10 (40)
Satellite	**100.19** (10)	640.33 (60)	714.52 (60)
Shuttle	4282.80 (10)	410.47 (10)	**237.89** (10)
Vehicle	**8.40** (10)	58.12 (10)	98.75 (10)
Vowel	**15.12** (10)	168.91 (20)	675.62 (70)
Wave	**63.57** (20)	486.95 (40)	102.51 (10)

Tabelle 4: Dauer des Tunings in Sekunden. Anzahl der gewählten Iterationen in Klammern

Es bietet sich an, die Tuningdauer noch etwas genauer zu beleuchten, um ein besseres Bild der Zeitintensität des Optimierungsprozesses zu erhalten. Dabei liefert Tabelle 5 einen Überblick über die durchschnittliche Zeit (in Sekunden), die eine Tuningiteration benötigt hat. Direkt unter diesen Werten befindet sich die gemessene Standardabwei-

chung. Die Durchschnittszeit fällt dabei in fast allen Fällen für die SVM am geringsten aus. Lediglich bei **Letter** benötigte der Random Forest und bei **Shuttle** der XGBoost weniger Zeit. Dieses Ergebnis stützt die Erkenntnisse aus Tabelle 4 bzgl. der Gesamtdauer des Tuningprozesses.

Datensatz	*SVM (OvO)*	*Random Forest*	*XGBoost*
Glass	**0.108** ± 0.008	0.341 ± 0.066	0.673 ± 0.417
Iris	**0.101** ± 0.009	0.274 ± 0.012	0.387 ± 0.101
Letter	120.523 ± 105.759	**25.645** ± 7.454	76.443 ± 89.949
Satellite	**5.444** ± 4.506	9.043 ± 2.569	7.600 ± 3.726
Shuttle	237.724 ± 447.653	22.268 ± 3.904	**6.184** ± 2.140
Vehicle	**0.373** ± 0.067	3.008 ± 0.980	2.482 ± 1.304
Vowel	**0.739** ± 0.142	5.744 ± 2.501	6.261 ± 1.596
Wave	**2.126** ± 1.069	9.799 ± 11.376	2.588 ± 1.289

Tabelle 5: Durchschnittliche Zeit in Sekunden pro Iteration und Standardabweichung. Beste Ergebnisse fett gedruckt.

Glass

Durch das Tuning konnte bei jedem Learner eine Verbesserung erzielt werden. Die Accuracy des XGBoost steigert sich um 5.41%, während Support Vector Machine und Random Forest nur eine Steigerung um 1.35% bzw 1.36% erfahren. XGBoost und Random Forest tauschen damit die Plätze, der Boostingalgorithmus schneidet nach dem Tuning, mit einem Vorsprung von 2.7%, am besten ab. Der Random Forest war mit 9.83 Sekunden schneller als Support Vector Machine (35.10 Sekunden) und XGBoost (29.97 Sekunden). Allerdings benötigt die SVM neunmal so viele Iterationen als die anderen beiden Learner. Ein Blick auf den Mittelwert der einzelnen Ausführungszeiten

zeigt, dass die Support Vector Machine eine einzelne Iteration schneller abschließt, als Random Forest und XGBoost. Allerdings darf nicht vergessen werden, dass die Support Vector Machine bei **Glass** deutlich mehr Parameterkombinationen testen musste, damit das Tuning einen Erfolg zeigt.

Iris

Bei dem Datensatz **Iris** brachte das Tuning nur bei Support Vector Machine und XGBoost eine Verbesserung von jeweils 2%. Das Ergebnis des Random Forest bleibt unverändert. Für jeden Learner wurden zehn Iterationen für das Tuning verwendet. Bei der SVM war das Finden der optimalen Hyperparameter deutlich schneller abgeschlossen als bei den anderen Methoden. Dies spiegelt sich auch in der durchschnittlichen Dauer einer Iteration wider. Die geringe Anzahl der Beobachtungen, Features und Klassen begünstigen dabei den geringen Zeitverbrauch der Support Vector Machine im Gegensatz zu den anderen Methoden, bei denen zum einen die Anzahl der gefitteten Bäume und zum anderen die Tiefe der Bäume selbst eine Auswirkung auf die Dauer des Tunings haben sollten. Zwar wurde bei Random Forest der Wert für `num.trees` bei 500 fixiert und im Gegensatz dazu beim XGBoost auch die Anzahl der zu fittenden Bäume optimiert, jedoch lässt sich damit nicht erklären, dass der XGBoost langsamer als der Random Forest ist, da mit Parameter dem `early_stopping_rounds = 10` deutlich weniger Bäume gefitted werden sollten, selbst wenn in einer Iteration wesentlich mehr als 500 Bäume trainiert werden.

Letter

Auch wenn die Performance vor Tuning bei Random Forest und XGBoost noch deutlich besser ausgefallen ist, so erzielt die Multi-Class SVM nach der Optimierung ein besseres Ergebnis. Durch das Tuning kann eine Verbesserung von 4.41% verzeichnet werden. Der Tuningerfolg des Random Forest hält sich mit einer Steigerung von 0.12% in Grenzen und XGBoost kann keine Steigerung der Accuracy verbuchen. Das beste Ergebnis, bei 40 Iterationen, ist um -0.31% schlechter als der Fit mit Defaultwerten. Bei Support Vector Machine und Random Forest wurden lediglich zehn Tuningiterationen verwendet. In Bezug auf die benötigte Zeit für das Tuning ist festzuhalten, dass SVM mit fast 37 Minuten und XGBoost mit etwas mehr als 87 Minuten sehr viel langsamer waren als der Random Forest, der nur knapp unter acht Minuten benötigt hat. Der Blick auf die durchschnittlich benötigte Zeit pro Iteration lässt allerdings erkennen, dass der XGBoost mit 76.443 Sekunden deutlich schneller war als die SVM, die durchschnittlich 120.523 Sekunden gebraucht hat. Der Datensatz **Letter** besitzt zum einen die meisten Klassen (26) und zum anderen besitzt er insgesamt 20000 Beobachtungen. Hier dürfte

sich der zeitliche Nachteil für die Support Vector Machine damit begründen lassen, dass sie mit dem One-versus-One Ansatz für jede paarweise Klassenkombination einen Classifier trainieren muss. In Verbindung mit der relativ hohen Anzahl an Beobachtungen verstärkt sich dieses Problem nur noch weiter.

Satellite

Die Verbesserung der Performance durch Optimierung der Hyperparameter, für den Datensatz **Satellite**, ist bei den Classifiern relativ gering ausgefallen. Die Steigerung der Accuracy liegt zwischen 0.096 bis 0.47%. Der XGBoost kann durch das Tuning mit dem Random Forest gleichziehen, während die Support Vector Machine den Abstand auf die beiden anderen Methoden etwas verringern konnte. Allerdings hat die Support Vector Machine mit 100.19 Sekunden deutlich weniger Zeit benötigt als Random Forest (640.33 Sekunden) und XGBoost (714.52 Sekunden). Zwar wurden für diese beiden Classifier, im Gegensatz zur SVM, sechsmal so viele Tuningiterationen benötigt, ein Blick auf den durchschnittlichen Zeitverbrauch pro Durchlauf gibt allerdings Aufschluss darüber, dass die Support Vector Machine insgesamt etwas schneller ist. Die etwas längere Berechnungszeit eines einzelnen Durchlaufs lässt sich damit begründen, dass der Datensatz 36 Attribute besitzt. Da Random Forest und XGBoost mit Bäumen arbeiten, müssen dementsprechend, abhängig von dem gewählten Wert für die Hyperparameter `mtry` und `colsample_bytree`, verhältnismäßig viele Attribute für die einzelnen Splits evaluiert werden. Kombiniert mit der Summe an Bäumen die trainiert werden, ergibt sich dann die etwas längere Rechenzeit im Vergleich zur Support Vector Machine.

Shuttle

Beim Datensatz **Shuttle** konnten nur noch geringe Verbesserungen erzielt werden, wobei das Tuning bei XGBoost erfolglos verlaufen ist. Die geringen Verbesserungen von 0.01% (RF) und 0.06% (SVM) sind allerdings in diesem Fall wünschenswert, da bereits vor Tuning eine sehr hohe Vorhersagegenauigkeit gegeben war, die zwischen 99.82 bis 99.98% lag. Der Random Forest schneidet mit einer Accuracy von 99.99% am besten ab. Bei jedem Learner hat das Minimum an Iterationen bereits das beste Ergebnis hervorgebracht. Die Support Vector Machine war dabei sehr langsam und hat insgesamt 4282.80 Sekunden für das Tuning benötigt. Random Forest und XGBoost waren mit 410.47 bzw. 237.89 Sekunden ungleich schneller. Zieht man einen Vergleich zu den Ergebnissen des Datensatzes **Letter**, so verwundert der Zeitverbrauch der Support Vector Machine nicht. Die Anzahl der Beobachtungen hat sich fast verdreifacht. Daran ändert auch die deutlich geringere Anzahl an Klassen bei **Shuttle** nichts. Der Random Forest hingegen lässt erkennen, dass ihm die deutlich höhere Anzahl an Beobachtungen keinen

Nachteil einbringt. Ebensowenig ist das beim XGBoost der Fall. Auf dem Datensatz **Shuttle** ist er sogar der schnellste Classifier. Eine geringere Anzahl an Klassen und Attributen führt dazu, dass er das Tuning wesentlich schneller abschließen kann.

Vehicle

Bei dem Problem **Vehicle** konnten Random Forest (0.71%), Support Vector Machine (3.9%) und XGBoost (1.06%) bereits nach zehn Iterationen eine Verbesserung erzielen. Mit fast 4% war dabei der Tuningerfolg der SVM am größten. Somit bleibt sie, nach Tuning, der beste Classifier für den Datensatz. Mit insgesamt 8.4 Sekunden und 0.373 Sekunden pro Iteration war sie auch der mit Abstand schnellste Learner. Wegen der geringen Anzahl an Beobachtungen (846) und Klassen (4) kann die Support Vector Machine hier einen deutlichen Vorteil gegenüber Random Forest (+49.72 Sekunden) und XGBoost (+90.35 Sekunden) verbuchen. Für die beiden letztgenannten Methoden sollte, während des Tunings, unteranderem die Anzahl von 18 Attributen ausschlaggebend für die Dauer der Optimierung sein. Bei Betrachtung der durchschnittlichen Zeit, die für eine Iteration benötigt wird, ist zu erkennen, dass der XGBoost diesmal ein wenig schneller war als der Random Forest. Da der XGBoost aber aufgrund des Startdesigns insgesamt 20 Iterationen mehr benötigt, war das Tuning des Random Forest im Endergebnis schneller abgeschlossen.

Vowel

Die Multi-Class SVM erfährt, mit der Mindestanzahl an Tuningdurchläufen, eine starke Verbesserung mit +14.24%. Bei der Boostingmethode ist kein Erfolg zu verzeichnen: das beste Ergebnis ist um −2.13% schlechter als die Variante mit Defaultparametern. Der Random Forest kann eine leichte Erhöhung der Accuracy von +1.21% verzeichnen. Auffallend ist, dass die Support Vector Machine das Tuning mit Abstand am schnellsten abgeschlossen hat. Während Random Forest (168.91 Sekunden) und XGBoost (675.62 Sekunden) im Mittel 5.744 bzw. 6.261 Sekunden pro Iteration benötigen, beendet die Support Vector Machine das Tuning bereits nach 15.12 Sekunden mit durchschnittlich 0.739 Sekunden pro Iteration.

Wave

Das Tuning des Datensatzes **Wave** gestaltet sich mittels modelbasierter Optimierung als schwierig. Random Forest und Support Vector Machine konnten keine Verbesserung erzielen, die Verbesserung des XGBoost fällt mit +0.06% sehr gering aus. Im Ergebnis bleibt die SVM der stärkste Classifier nach Tuning. Die Support Vector Machine benötigt auch hier am wenigsten Zeit für die Optimierung der Hyperparameter. Der

XGBoost benötigt insgesamt 38.94 Sekunden mehr, das Tuning des Random Forest ist sogar erst mehr als 7 Minuten später abgeschlossen. Ein Blick auf die durchschnittliche Zeit pro Iteration gibt zu erkennen, dass der XGBoost nicht wesentlich langsamer wäre als die Support Vector Machine. Der Mehrverbrauch an Zeit für das Tuning lässt sich wieder damit erklären, dass der XGBoost am Ende zehn Iterationen mehr benötigt. Der Random Forest lässt, unabhängig der benötigten 40 Tuningiterationen, erkennen, dass er auch im Durchschnitt 9.799 Sekunden benötigt hat. Zusätzlich besitzen die Ausführungszeiten der einzelnen Iterationen mit 11.376 Sekunden eine sehr starke Streuung. Das ist vorallem darauf zurückzuführen, dass in den ersten 9 Tuningiterationen Werte zwischen 3 und 21 für den Parameter `mtry` gewählt wurden. Bei diesem speziellen Datenproblem scheint das Ermitteln der einzelnen Splitkandidaten am meisten Zeit in Anspruch genommen zu haben.

3.3.4 Benchmarkergebnisse

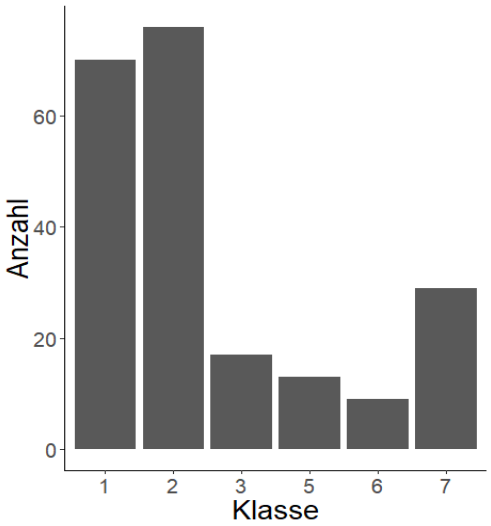
Abschließend wird, mit der **mlr**-Funktion `benchmark()` ein kleines Benchmarkexperiment durchgeführt. Dabei werden nun die Datensätze mit sämtlichen Beobachtungen verwendet. Auf diesen Daten wird eine wiederholte Kreuzvalidierung durchgeführt. Insgesamt werden fünf Wiederholungen mit je zehn Folds ausgeführt. Einzige Ausnahme

Datensatz	*SVM (OvO)*	*Random Forest*	*XGBoost*
Glass	71.58	**79.43**	64.56
Iris	**96.93**	95.20	94.13
Letter	**97.84**	96.88	96.52
Satellite	**92.08**	91.93	92.02
Shuttle	99.91	**99.99**	99.55
Vehicle	**83.43**	75.27	77.35
Vowel	**99.15**	96.22	90.69
Wave	**85.81**	85.13	84.89

Tabelle 6: Accuracy der Benchmarkergebnisse in %.

von dieser Vorgehensweise stellt der Datensatz **Glass** dar, bei dem wegen der unausgeglichenen Klassenverteilung stratifiziert wurde und deshalb nur neun Folds möglich sind. Es muss dabei beachtet werden, dass nun jeder Classifier sämtliche Daten zum Trainieren des Modells erhält und kein gesonderter Testsplit mehr vorliegt. Als Folge dessen ist zu beachten, dass die Ergebnisse des Benchmarks etwas zu optimistisch ausfallen sollten. Nichtsdestotrotz soll mit diesem Experiment versucht werden, die Variabilität der Vorhersageergebnisse des Tunings, etwas zu verringern. In Tabelle 6 sind die Ergebnisse aufgelistet. Zusätzlich befinden sich im Anhang Grafiken der Konfusionsmatrizen für jeden einzelnen Datensatz und Classifier, um ein eingehenderes Bild der Benchmarkergebnisse zu liefern.

Die Support Vector Machine erzielt nun auf jedem Datensatz das beste Ergebnis außer bei **Glass** und **Shuttle**. Auf diesen beiden Datensätzen liegt der Random Forest vorne. Der XGBoost übertrumpft in keinem der acht Fälle die anderen Classifier. Bei den Problemen **Glass** und **Satellite** kam es zu einer Veränderung in den Platzierungen. Hat der XGBoost nach Tuning auf den **Glass**-Daten noch deutlich besser abgeschnitten als Support Vector Machine und Random Forest, so ist er nun deutlich abgeschlagen im Gegensatz zu den anderen beiden Methoden.



Klasse	*SVM*	*RF*	*XGB*
1	22.86	12.29	13.71
2	32.37	19.47	21.32
3	57.65	68.24	82.35
5	29.23	32.31	64.62
6	22.22	13.33	71.11
7	15.86	13.10	73.79

Abbildung 15 & Tabelle 7: Glass: Klassenverteilung (links) und Fehlklassifizierung der einzelnen Klassen in % (rechts).

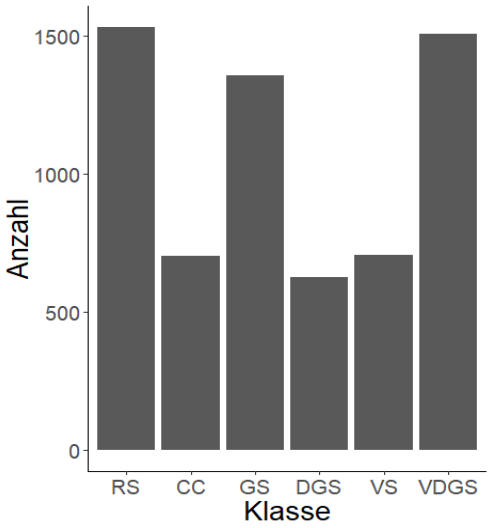
Abbildung 15 und nebenstehende Tabelle geben einen genaueren Einblick in die Klassenverteilung und die prozentuale Fehlerquote je Klasse. Es ist zu erkennen, dass der XGBoost gerade bei den Klassen, die im Datensetz deutlich seltener vorkommen, zwischen 64.62% und 82.35% Fehlerquote aufweist. Allerdings ist anzumerken, dass Support Vector Machine und Random Forest bei Klasse `3` auch erhebliche Probleme mit

der Klassifizierung besitzen. Insgesamt scheinen die **Glass**-Daten relativ schwierig zu klassifizieren. Die SVM teilt bspw. bei den Klassen `1-6` teilweise deutlich mehr als 1/5 der Beobachtungen falsch zu. Insbesondere scheinen alle drei Classifier besonders häufig Kategorie `1` und `2` vorherzusagen, die beiden Klassen, die ein deutliches Übergewicht besitzen.

Bei dem Datensatz **Iris** gibt es keine großen Überraschungen. Zwar sind die Ergebnisse des Benchmarks etwas unter den Ergebnissen des Tunings, die Support Vector Machine liefert nach wie vor die beste Accuracy, allerdings verliert der XGBoost deutlich mehr gegenüber den anderen Classifiern und fällt so hinter dem Random Forest zurück. Sämtliche Methoden konnten die Klasse `setosa` ohne Probleme vorhersagen. Bei den verbleibenden Klassen hatten Random Forest und XGBoost deutlich größere Probleme als die SVM.

Der Benchmark für **Letter** liefert ein ähnliches Bild wie die Ergebnisse des Tunings. Allerdings sind hier etwas optimistischere Ergebnisse zu verzeichnen. Die Konfusionsmatrizen im Anhang geben Aufschluss darüber, dass im Speziellen Buchstabenpaare wie z.B. `I-J`, `P-F` oder `R-B` Probleme bei der Klassifizierung bereiten. Da gerade solche Buchstabenpaare, aufgrund ihrer Verwechslungsgefahr, von besonderem Interesse sind, ist fraglich, ob die erhaltenden Resultate befriedigend sind.

Bei **Satellite** konnten Support Vector Machine und XGBoost den Random Forest überholen. Abbildung 16 inkl. der Tabelle rechts zeigt, dass besonders die Klasse `DGS (damp grey soil)` Probleme in der Klassifizierung bereitet. Hier scheint jeder Classifier ein



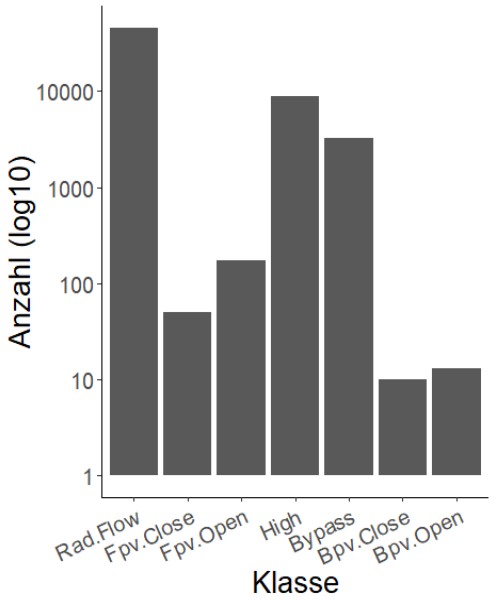
Klasse	*SVM*	*RF*	*XGB*
RS	1.68	1.90	1.93
CC	2.22	2.42	2.73
GS	4.61	3.86	4.71
DGS	34.25	37.06	33.83
VS	7.30	9.76	9.28
VDGS	9.27	7.92	8.20

Abbildung 16 & Tabelle 8: Satellite: Klassenverteilung (links) und Fehlklassifizierung der einzelnen Klassen in % (rechts).

Problem damit zu haben, dass die Klasse im Vergleich zu `GS (grey soil)` und `VDGS`

`(very damp grey soil)` verhältnismäßig selten vorkommt und sich verschiedene Variationen von grauem Boden (hier feucht und sehr feucht) schwer unterscheiden lassen.

Der Datensatz **Shuttle** weist eine extrem ungleiche Klassenverteilung auf. Daher soll mit Abbildung 17 beurteilt werden, wie die Ergebnisse der Classifier im Benchmarkexperiment zu bewerten sind. Nach dem Benchmark ist die Support Vector Machine nun etwas besser als der XGBoost. Random Forest behält das gute Ergebnis von 99.99% Accuracy bei. Wie bereits an Beispielen **Glass** und **Satellite** zu sehen, scheint der XGBoost etwas Probleme mit Datensätzen zu haben, die sehr unbalanciert sind.



Klasse	*SVM*	*RF*	*XGB*
Rad.Flow	0.01	0.00	0.00
Fpv.Close	2.00	2.00	2.00
Fpv.Open	6.55	0.59	2.34
High	0.11	0.01	2.10
Bypass	0.39	0.00	1.90
Bpv.Close	38.00	24.00	24.00
Bpv.Open	58.46	3.08	40.00

Abbildung 17 & Tabelle 9: Shuttle: Klassenverteilung (links) und Fehlklassifizierung der einzelnen Klassen in % (rechts).

Diese Vermutung bestätigt sich bei **Shuttle**. Interessant ist dabei allerdings, dass die Support Vector Machine sich prozentual mehr Fehler, bei den Klassen `Bpv.Close` und `Bpv.Open`, erlaubt als der XGBoost. Dennoch erscheint das Ergebnis des Benchmark für die SVM zu sprechen. Hierbei ist allerdings zu beachten, dass nur 10 (`Bpv.Close`) bzw. 13 (`Bpv.Open`) von insgesamt 58000 Beobachtungen auf diese Klassen entfallen. So verwundert es nicht, dass dieser prozentuale Unterschied kaum ins Gewicht fällt. Ein weiteren Knackpunkt, im Vergleich von SVM und XGBoost, stellt die Variable `High` mit 8903 Beobachtungen dar. Da XGBoost bei dieser Kategorie fast 2% mehr Fehlklassifikationen aufweist, fällt er im Ergebnis hinter die SVM zurück. Ein Blick auf die Konfusionsmatrizen liefert zusätzlich die Erkenntnis, dass die Support Vector Machine sehr viele Klassen der am häufigsten vertretenen Kategorie `Rad.Flow` zuweist. Dies ist als eindeutiger Hinweis zu verstehen, dass ein extremes Ungleichgewicht in

der Verteilung der Klassen die Support Vector Machine negativ beeinflusst und dazu führt, dass die dominierende Klasse einen viel zu großen Einfluss auf das Bilden der Entscheidungsgrenze besitzt.

Der Benchmark fällt bei **Vehicle** für die Support Vector Machine etwas besser aus als das Tuningergebnis. Die anderen beiden ML-Methoden büßen hingegen etwas an Accuracy ein. Bei diesem Datensatz geht es darum, vier Fahrzeuge (Cheverolet Van, Doppeldeckerbus, Saab 9000 und Opel Manta 400) aufgrund ihrer Silhouette richtig einzuteilen. Die vier Fahrzeugtypen wurden dabei so gewählt, dass davon auszugehen ist, dass sich `van`, `bus` und eines der Fahrzeuge leicht unterscheiden lassen, die Klassifizierung von `opel` und `saab` allerdings schwer fällt. Dieses Ergebnis bestätigt ein Blick auf die Konfusionsmatrizen. Desweiteren scheinen alle drei Methoden die Neigung zu besitzen, jeden der vier Fahrzeugtypen als `van` zu klassifizieren.

Die Ergebnisse des Benchmarkexperiments für die **Vowel**-Daten liefern deutlich höhere Accuracies als im Tuning ermittelt. Dennoch ändert sich an der Reihenfolge der Performance nichts. der XGBoost konnte allerdings seinen Abstand zum Random Forest etwas verringern, auch wenn er immernoch deutlich abgeschlagen hinter den anderen Methoden liegt. Hatte das Tuning bereits einen sehr großen Erfolg bei der Support Vector Machine zur Folge, so ist das Benchmarkergebnis von 99.15% nochmal eine deutliche Steigerung gegenüber den 96.96% nach dem Tuning. Durch die Betrachtung der Konfusionsmatritzen erkennt man, dass alle Classifier dazu neigen den Vokal `had` als `hed` zu klassifizieren. Die fehlerhafte Einteilung von `hed` als `had` ist allerdings nur bei XGBoost zu erkennen. Auch wenn die Boostingmethode, im Vergleich zu den anderen Learnern, deutliche Defizite bei der Klassifizierung von **Vowel** aufweist, ist anzumerken, dass sowohl XGB als auch Random Forest dazu neigen, viele Beobachtungen der Kategorie `hed` zuzuteilen. Eine Erkenntnis, die bei der Multi-Class SVM nicht zu beobachten ist.

Der Datensatz **Wave** liefert im Benchmark vergleichbare Resultate wie in Tabelle 3 nach Abschluss des Tunings. Der Random Forest kann etwas zulegen und somit den XGBoost überholen, der etwas an Accuracy einbüßt. Die Support Vector Machine bleibt dabei der Classifier mit dem besten Ergebnis. Allerdings beträgt die Differenz der Ergebnisse der SVM und Random Forest weniger als 1%. Die Konfusionsmatrizen zeigen, dass sämtliche Learner vorallem mit Klasse `1` Probleme haben. Tabelle 10 gibt an wieviel Prozent der Beobachtungen einer Klasse falsch kategorisiert wurden. Es fällt auf, dass der XGBoost mit Klasse `1` weniger Probleme hat als Support Vector Machine und Random Forest. Allerdings büßt er diesen Vorteil wieder ein, da er die Klassen `2` und `3` öfter falsch klassifiziert als die anderen beiden Learner.

Klasse	*SVM (OvO)*	*Random Forest*	*XGBoost*
1	20.24	21.99	19.41
2	11.27	10.88	12.45
3	11.10	11.77	13.48

Tabelle 10: Fehlklassifizierung der einzelnen Klassen in % (Wave).

4 Fazit

In dieser Arbeit war es das Ziel herauszufinden, wie die Support Vector Machine, im direkten Vergleich mit anderen Klassifizierungsmethoden, abschneidet, wenn es sich nicht um ein binäres Datenproblem handelt und eine Vielzahl von Klassen vorhergesagt werden sollen. Originär wurde die Support Vector Machine für den Zwei-Klassen-Fall konzipiert und es ist nur mit speziellen Methoden möglich sie auf den Multi-Class Fall zu erweitern. Als kompetitive Kontrahenten wurden der Random Forest und das Extreme Gradient Boosting gewählt. Dabei wurden sowohl die erzielte Accuracy als auch der Zeitverbrauch des Tunings in Betracht gezogen.
Nachdem eine ausführliche Einführung in das Konzept der Hyperebene, des Optimierungsproblems und der Erweiterung des Hard Margin Classifiers hin zum Support Vector Classifier erfolgte, wurde die Support Vector Machine und die Verwendung von Kernelfunktionen erläutert. Darauf erfolgte ein Überblick über verschiedene Ansätze, die der SVM die Multi-Class Classification ermöglichen. Nach einem kurzen Einblick in die anderen beiden Klassifizierungsmethoden, erfolgte in Abschnitt 3 der Vergleich der einzelnen Classifier. Dabei wurde auf die verwendete Software und die zu klassifizierenden Datensätze eingegangen.
Als Erstes wurde die Performance der einzelnen Methoden mit den voreingestellten Hyperparametern verglichen. Dabei zeigte sich, dass der Random Forest die beste Leistung erbrachte, gefolgt von der Support Vector Machine und dem XGBoost als Schlusslicht. Sodann wurde das Tuning der einzelnen Hyperparameter für alle drei Methoden durchgeführt. Um einen fairen Vergleich der Methoden zu ermöglichen, wurde die modelbasierte Optimierung durch das R-Paket **mlrMBO** gewählt. Es wurden vorgefertigte Parameterräume für die einzelnen Classifier durch das Paket **mlrHyperopt** verwendet, um so zu gewährleisten, dass alle Learner gleiche Startbedingungen besitzen. Nach einer ausführlichen Analyse erfolgte zusätzlich noch ein kleines Benchmarkexperiment. Diese fünfmal wiederholte k-fold Cross-Validation hatte den Hintergrund, etwaige Variabilität der Ergebnisse zu verringern.

Die Ergebnisse von Tuning und Benchmark haben gezeigt, dass die Support Vector Machine in fünf bzw. sechs von acht Fällen das beste Ergebnis liefert. Random Forest und XGBoost teilten sich nach dem Tuning einmal den ersten Platz in Bezug auf die Accuracy, allerdings konnte der Random Forest den Tuningprozess etwas schneller beenden, so dass es nur fair erscheint, dem Random Forest zweimal die beste Performance zuzuschreiben. Der XGBoost konnte im Tuning lediglich auf dem Datensatz **Glass** überzeugen. Was die Dauer des Tunings angeht, so konnte auch dort die Support Vector Machine überzeugen. In fünf von acht Fällen war sie am schnellsten. Lediglich bei Datensätzen

mit 20000 oder mehr Beobachtungen besitzt die SVM einen deutlichen Nachteil gegenüber den anderen Methoden. Bei diesen Datensätzen waren je einmal Random Forest und XGBoost deutlich schneller. Der Random Forest konnte zudem auf dem Datensatz **Glass** mit einer sehr kurzen Zeit aufwarten. Bei den Benchmarkexperimenten konnte der Random Forest zweimal die höchste Accuracy aufweisen. Der XGBoost schaffte in keinem einzigen Fall das beste Ergebnis.
Die Ergebnisse des XGBoost erscheinen nicht befriedigend. Es ist zu vermuten, dass das modelbasierte Optimieren von Hyperparametern bei der Vielzahl von zu tunenden Parametern des XGBoost nur bedingt geeignet ist. Auch bei der Dauer des Tunings muss berücksichtigt werden, dass bei XGBoost ingesamt sieben Parameter optimiert wurden, wohingegen für Random Forest und Support Vector Machine nur zwei Parameter nötig waren.

Ingesamt kann die Support Vector Machine als Multi-Class Classifier überzeugen, allerdings mit Einschränkungen, die das gute Ergebnis zu trüben vermögen. Zum einen fällt auf, dass das Tunen/Trainieren der Support Vector Machine auf größeren Datensätzen sehr zeitintensiv ist. Dabei muss ausdrücklich herausgestellt werden, dass mit **Letter** (20000 Beobachtungen) und **Shuttle** (58000 Beobachtungen) zwar etwas umfangreichere Datensätze vorliegen, diese aber keineswegs als besonders groß zu bezeichnen sind. Eben dieser Umstand ist einer der größten Hindernisse für die Support Vector Machine, gerade in Zeiten, in denen der Begriff Big Data eine immer größer werdende Rolle spielt. Die Tatsache, dass zusätzlich, je nach Multi-Class Methode, mehrere Classifier gefitted werden müssen oder ein einziges großes Optimierungsproblem berechnet werden muss, verstärkt diesen Effekt noch zusätzlich.
Die Ergebnisse des Benchmarks haben zudem gezeigt, dass besonders die Support Vector Machine Probleme mit unbalancierten Datensätzen besitzt. Konnte man bei den **Glass**-Daten mit 214 Ausprägungen noch feststellen, dass alle drei Methoden Testbeobachtungen sehr oft den dominierenden Klassen zugeteilt haben, so wurde das Problem der Support Vector Machine bei dem Datensatz **Shuttle** offenkundig. Auch wenn ihre Accuracy auf dem Blatt, gerade im Vergleich zu XGBoost, zufriedenstellend aussieht, so ist zu erkennen, dass die gefundene Entscheidungsgrenze der SVM sehr stark von der dominierenden Klasse `Rad.Flow` beeinflusst wurde und die SVM dazu neigt viele der anderen Klassen dieser Kategorie zuzuteilen.
Grundsätzlich scheint die Support Vector Machine mit mehreren Klassen zurecht zu kommen. Es deutet jedenfalls nichts darauf hin, dass die Support Vector Machine mit steigender Anzahl an Kategorien an Zuverlässigkeit einbüßt. Diese Erkenntnis wird jedenfalls auch von [11] gestützt. Die erhaltenen Ergebnisse lassen den Verdacht aufkeimen, dass die Support Vector Machine mit steigender Anzahl an Attributen (vgl. Tabel-

len 1 und 5) weniger Zeit benötigt als die anderen beiden Classifier. Abschließend wäre es noch von Interesse zu überprüfen, wie sich die unterschiedlichen Methoden verhalten, wenn den Datensätzen deutlich mehr Features zugrunde liegen oder die Anzahl der Klassen stark ansteigt. Zusätzlich wäre es wissenswert, wie sich die einzelnen Methoden im Vergleich schlagen, wenn für jede Methode gesondert das optimale Tuningprozedere gefunden und angewendet wird. Diese zusätzlichen Erkenntnisse müssen vorliegend allerdings offen bleiben, da sie den zeitlichen Rahmen der Arbeit gesprengt hätten.

Literatur

[1] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine Learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[2] Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006.

[4] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[5] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[6] Tianqi Chen. What is the difference between the R gbm (gradient boosting machine) and xgboost (extreme gradient boosting)? `https://bit.ly/2EEMRK4`. [Online; Zugriff am 31. März 2019].

[7] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[8] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, Mu Li, Junyuan Xie, Min Lin, Yifeng Geng, and Yutian Li. *xgboost: Extreme Gradient Boosting*, 2018. R package version 0.82.1.

[9] Thomas M Cover. Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition. *IEEE transactions on electronic computers*, (3):326–334, 1965.

[10] Koby Crammer and Yoram Singer. On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines. *Journal of Machine Learning Research*, 2(Dec):265–292, 2001.

[11] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we Need Hundreds of Classiers to Solve Real World Classication Problems? *The Journal of Machine Learning Research*, 15(1):3133–3181, 2014.

[12] Abhishek Ghose. Support Vector Machine (SVM) Tutorial. `https://blog.statsbot.co/support-vector-machines-tutorial-c1618e635e93`, 2017. [Online; Zugriff am 04 März 2019].

[13] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, 2nd edition, 2009.

[14] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A Practical Guide to Support Vector Classification. Technical report, Department of Computer Science, National Taiwan University, 2003.

[15] Chih-Wei Hsu and Chih-Jen Lin. A Comparison of Methods for Multiclass Support Vector Machines. *IEEE transactions on Neural Networks*, 13(2):415–425, 2002.

[16] Chih-Wei Hsu and Chih-Jen Lin. A Simple Decomposition Method for Support Vector Machines. *Machine Learning*, 46:291–314, 2002.

[17] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer-Verlag, 2013.

[18] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab – An S4 Package for Kernel Methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004.

[19] Alexandre Kowalczyk. Support Vector Machines Succinctly. *Succintly E-Book Series. Syncfusion*, 2017.

[20] Ulrich H-G Kreßel. *Advances in Kernel Methods - Support Vector Learning*, chapter Pairwise Classification and Support Vector Machines, pages 255–268. MIT Press, Cambridge, MA, 1999.

[21] Friedrich Leisch and Evgenia Dimitriadou. *mlbench: Machine Learning Benchmark Problems*, 2010. R package version 2.1-1.

[22] DMLC(distributed machine learning common). A Introduction to XGBoost R package. `https://bit.ly/2XpZRIv`, March 2016.

[23] D.J. Newman, S. Hettich, C.L. Blake, and C.J. Merz. UCI Repository of Machine Learning Databases, 1998.

[24] John Platt. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Technical report, April 1998.

[25] John C Platt, Nello Cristianini, and John Shawe-Taylor. Large Margin DAGs for Multiclass Classification. In *Advances in Neural Information Processing Systems*, pages 547–553, 2000.

[26] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[27] Jakob Richter. *mlrHyperopt: Easy Hyperparameteroptimization with mlr and mlrMBO*, 2019. R package version 0.1.1.

[28] Ryan Rifkin. 9.520: Statistical Learning Theory and Applications, Spring 2008: Multiclass Classification. `http://www.mit.edu/~9.520/spring09/Classes/multiclass.pdf`, February 2008. [Online; Zugriff am 16.03.2019].

[29] Manish Saraswat. Beginners Tutorial on XGBoost and Parameter Tuning in R. `https://bit.ly/2pHRLxU`. [Online; Zugriff am 31. März 2019].

[30] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels : Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning. The MIT Press, 2002.

[31] Unbekannt. The Radial Basis Function Kernel. `http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/svms/RBFKernel.pdf`. [Online; Zugriff am 06 März 2019].

[32] Vladimir Vapnik and Corinna Cortes. Support-Vector Networks. *Machine Learning*, 20:273–297, 1995.

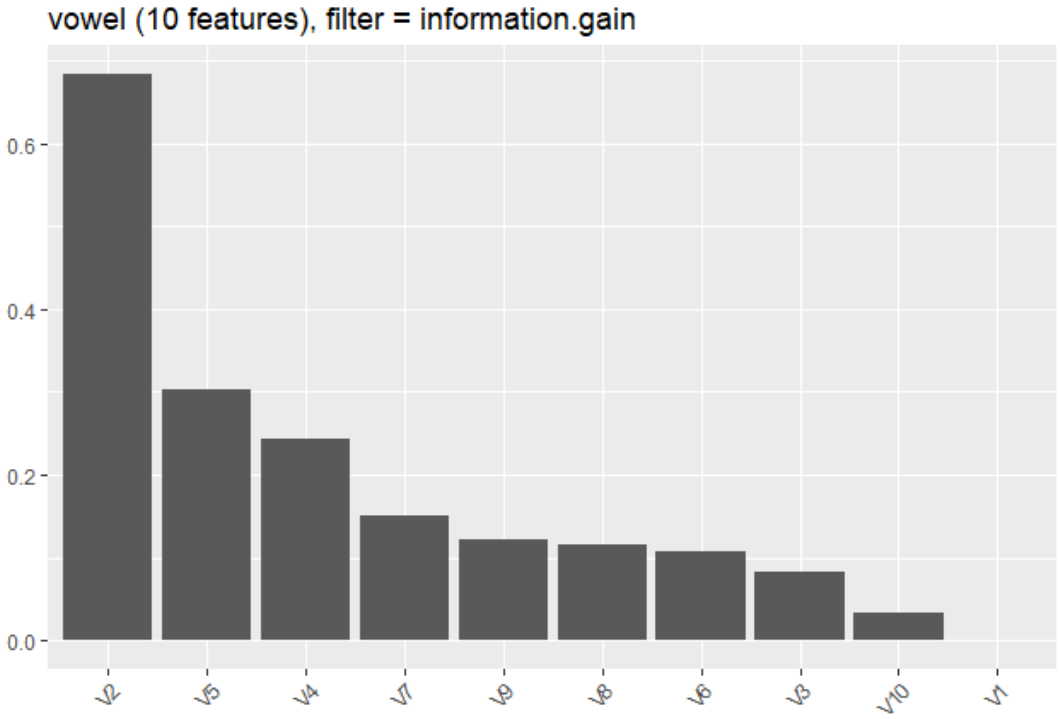
[33] Zhe Wang and Xiangyang Xue. Multi-Class Support Vector Machines. In *Support Vector Machines Applications*, pages 23–48. Springer, 2014.

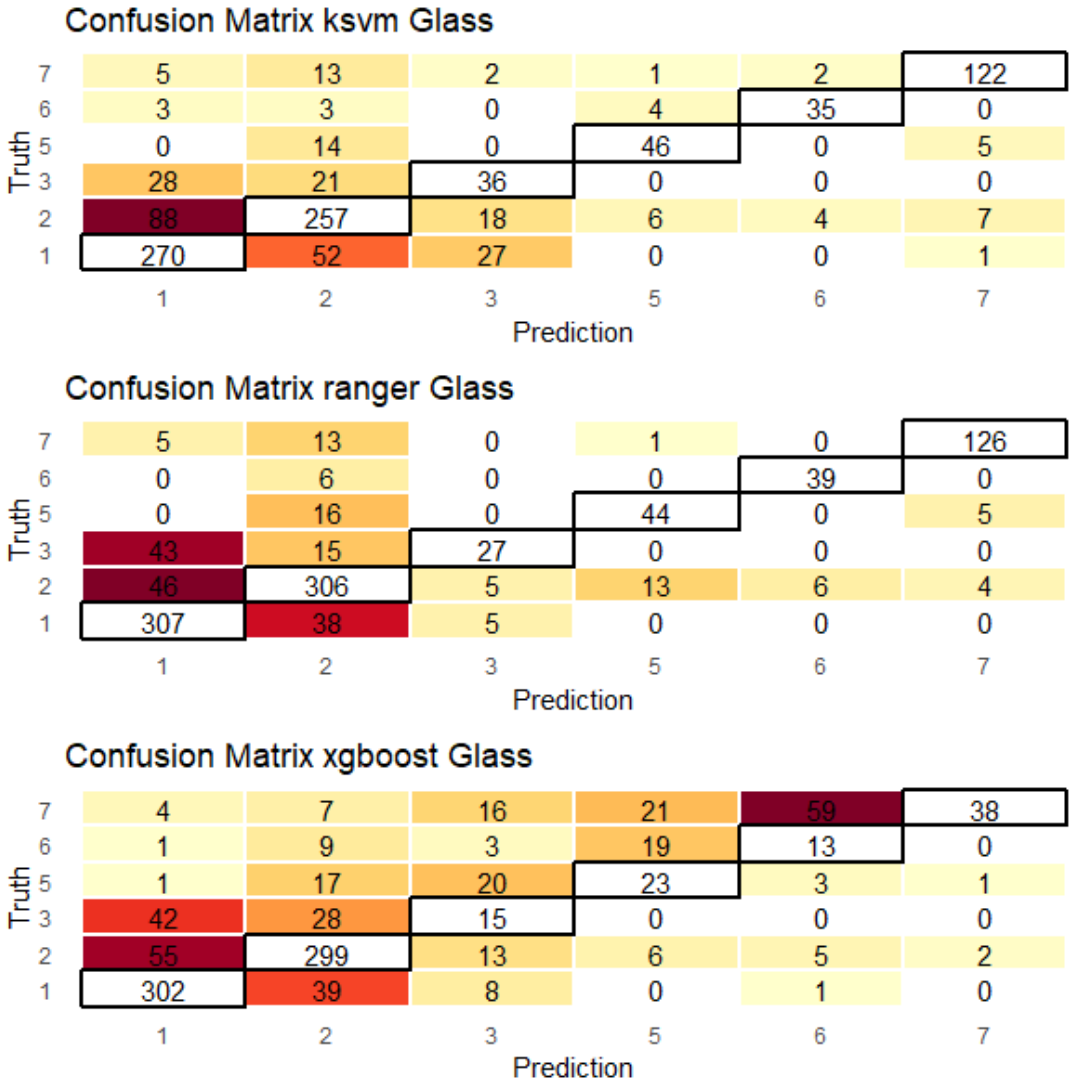
[34] J Weston and C Watkins. Multi-class Support Vector Machines. 1998.

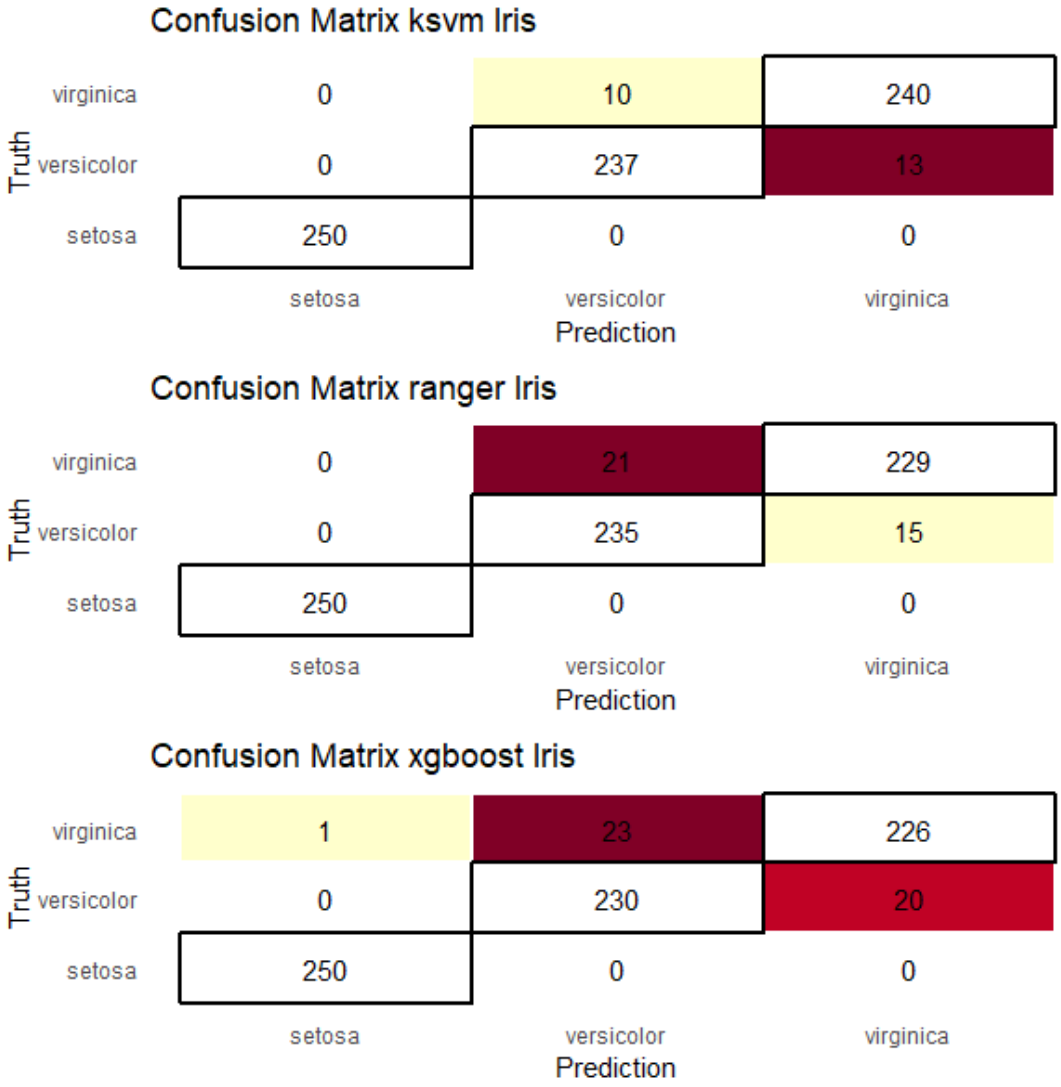
[35] Marvin N. Wright and Andreas Ziegler. ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1):1–17, 2017.

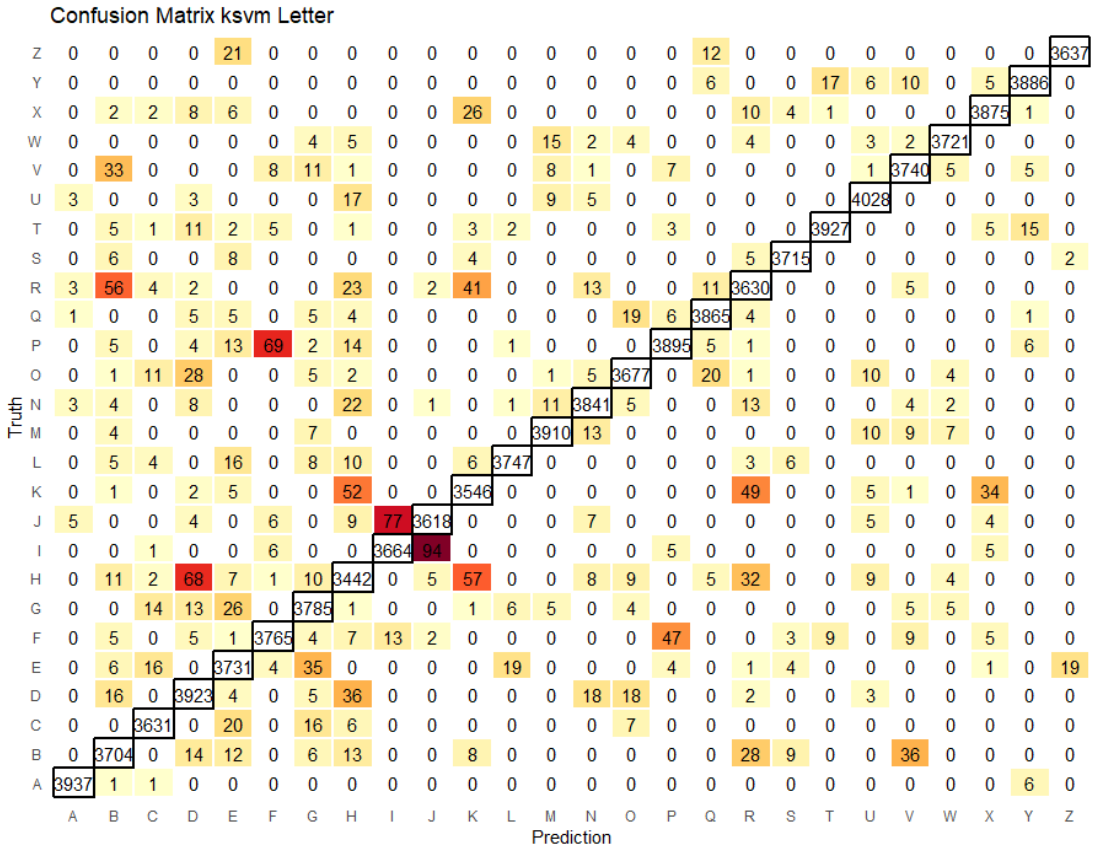
[36] xgboost github. Machine Learning Challenge Winning Solutions. `https://bit.ly/2k7W3Jh`. [Online; Zugriff am 31. März 2019].

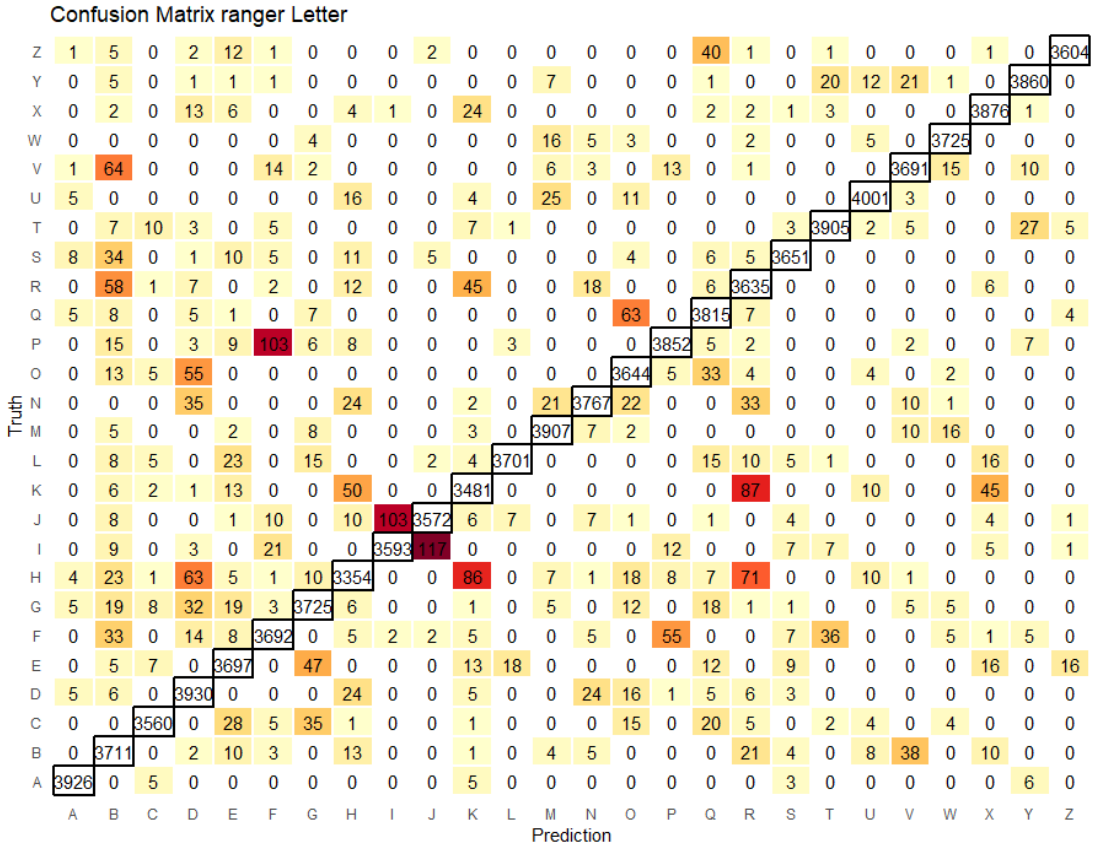
A Appendix

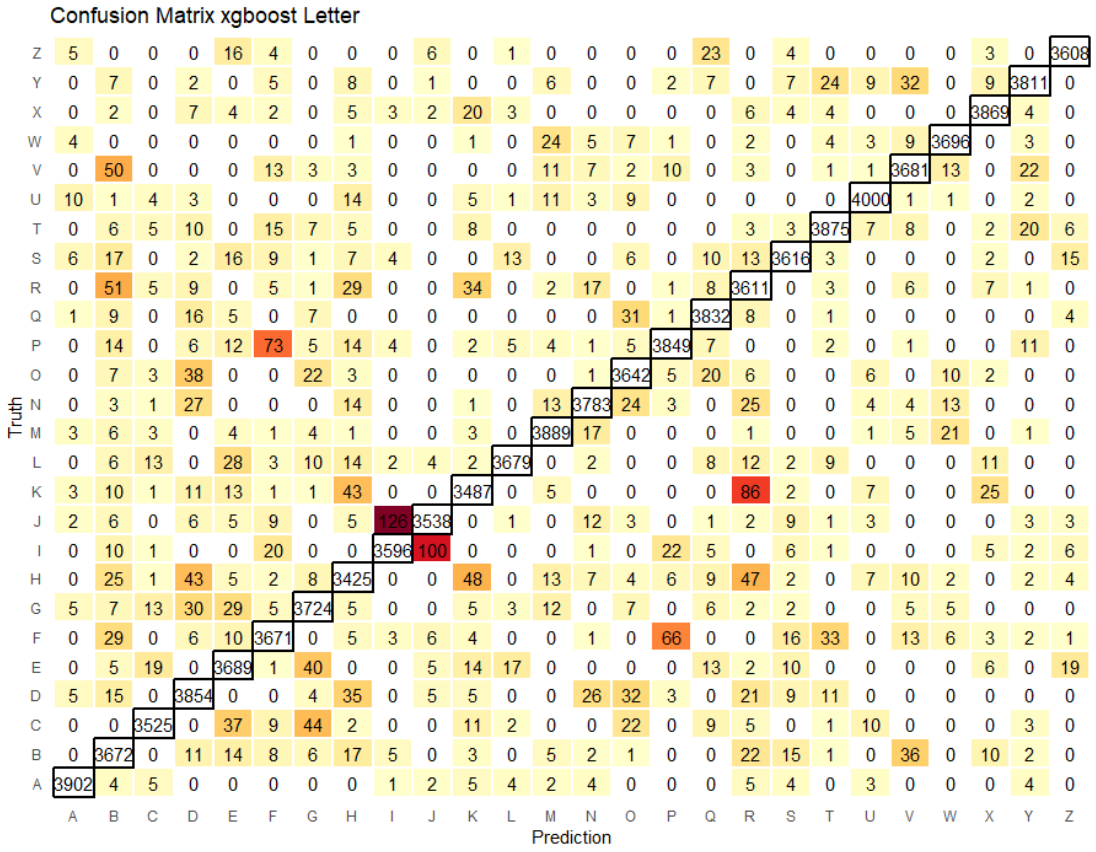

vowel (10 features), filter = information.gain
0.6
0.4
0.2
0.0
V2
V5
V4
V7
V9
V8
V6
V3
V10
V1

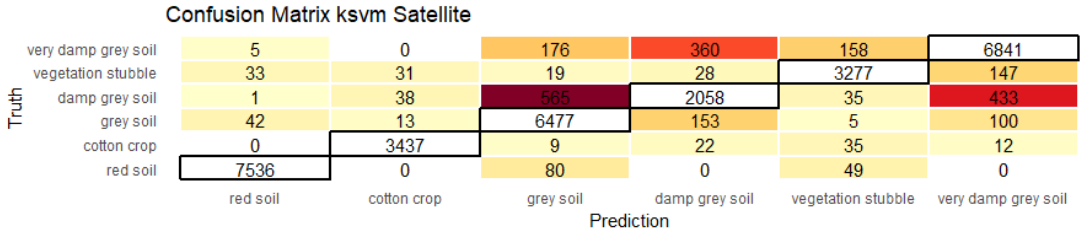

Confusion Matrix ksvm Glass
Truth
7 5 13 2 1 2 122
6 3 3 0 4 35 0
5 0 14 0 46 0 5
3 28 21 36 0 0 0
2 88 257 18 6 4 7
1 270 52 27 0 0 1
1 2 3 5 6 7
Prediction
Confusion Matrix ranger Glass
Truth
7 5 13 0 1 0 126
6 0 6 0 0 39 0
5 0 16 0 44 0 5
3 43 15 27 0 0 0
2 46 306 5 13 6 4
1 307 38 5 0 0 0
1 2 3 5 6 7
Prediction
Confusion Matrix xgboost Glass
Truth
7 4 7 16 21 59 38
6 1 9 3 19 13 0
5 1 17 20 23 3 1
3 42 28 15 0 0 0
2 55 299 13 6 5 2
1 302 39 8 0 1 0
1 2 3 5 6 7
Prediction

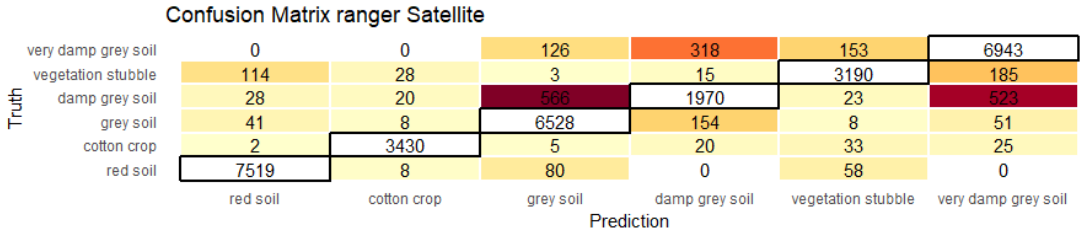

Confusion Matrix ksvm Iris
virginica
versicolor
setosa
Truth
0
10
240
0
237
13
250
0
0
setosa
versicolor
virginica
Prediction
Confusion Matrix ranger Iris
virginica
versicolor
setosa
Truth
0
21
229
0
235
15
250
0
0
setosa
versicolor
virginica
Prediction
Confusion Matrix xgboost Iris
virginica
versicolor
setosa
Truth
1
23
226
0
230
20
250
0
0
setosa
versicolor
virginica
Prediction

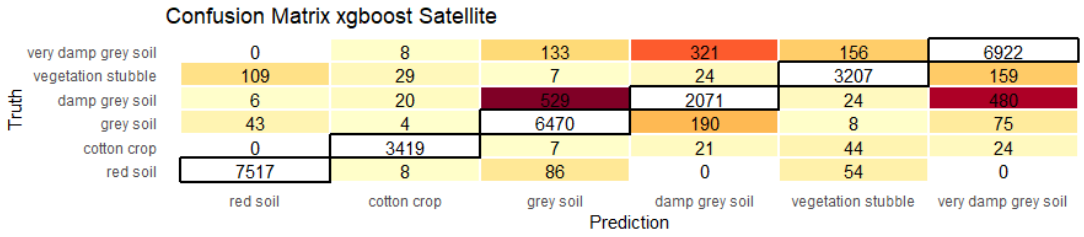

Confusion Matrix ksvm Letter
Truth
Prediction
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

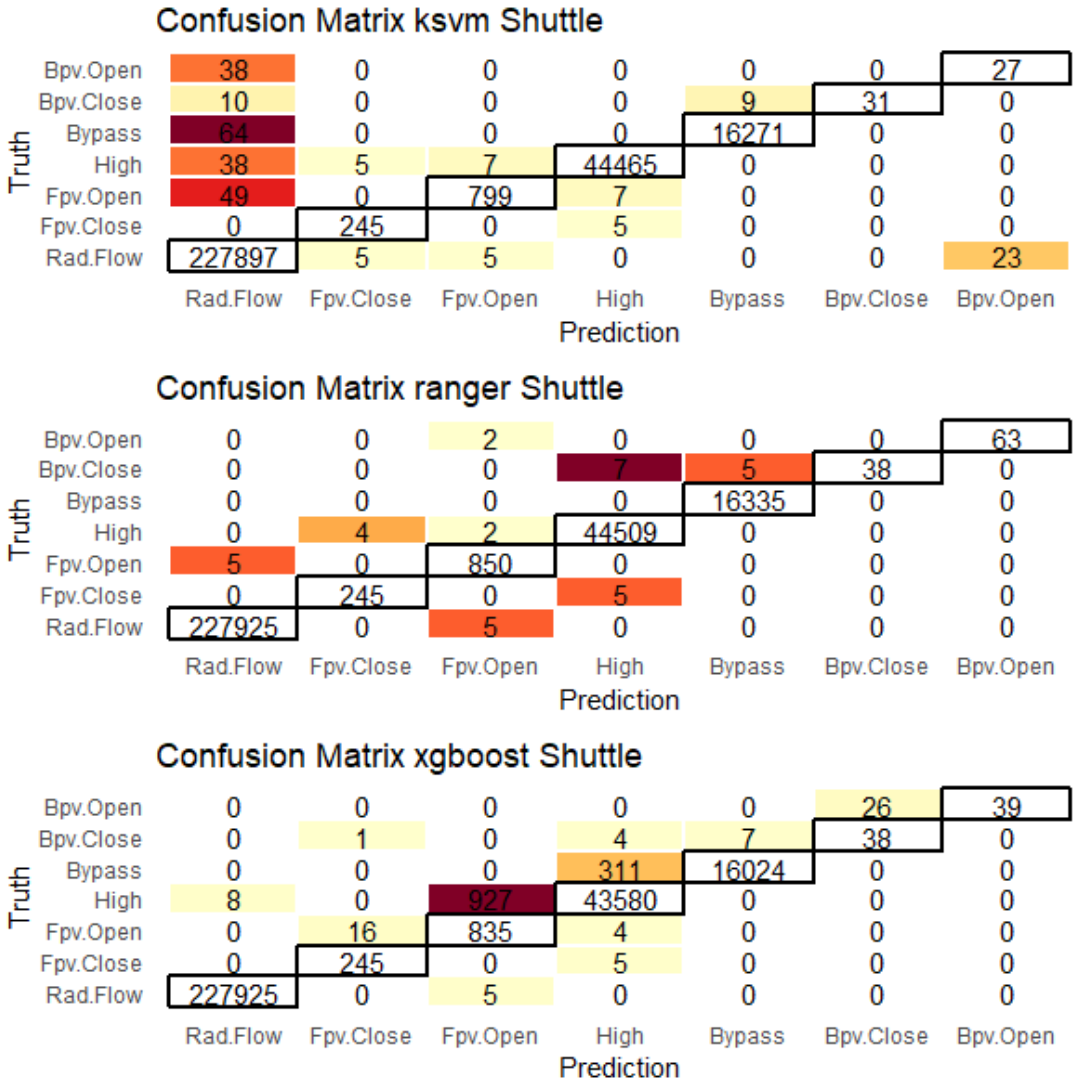

Confusion Matrix ranger Letter
Truth
Prediction
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

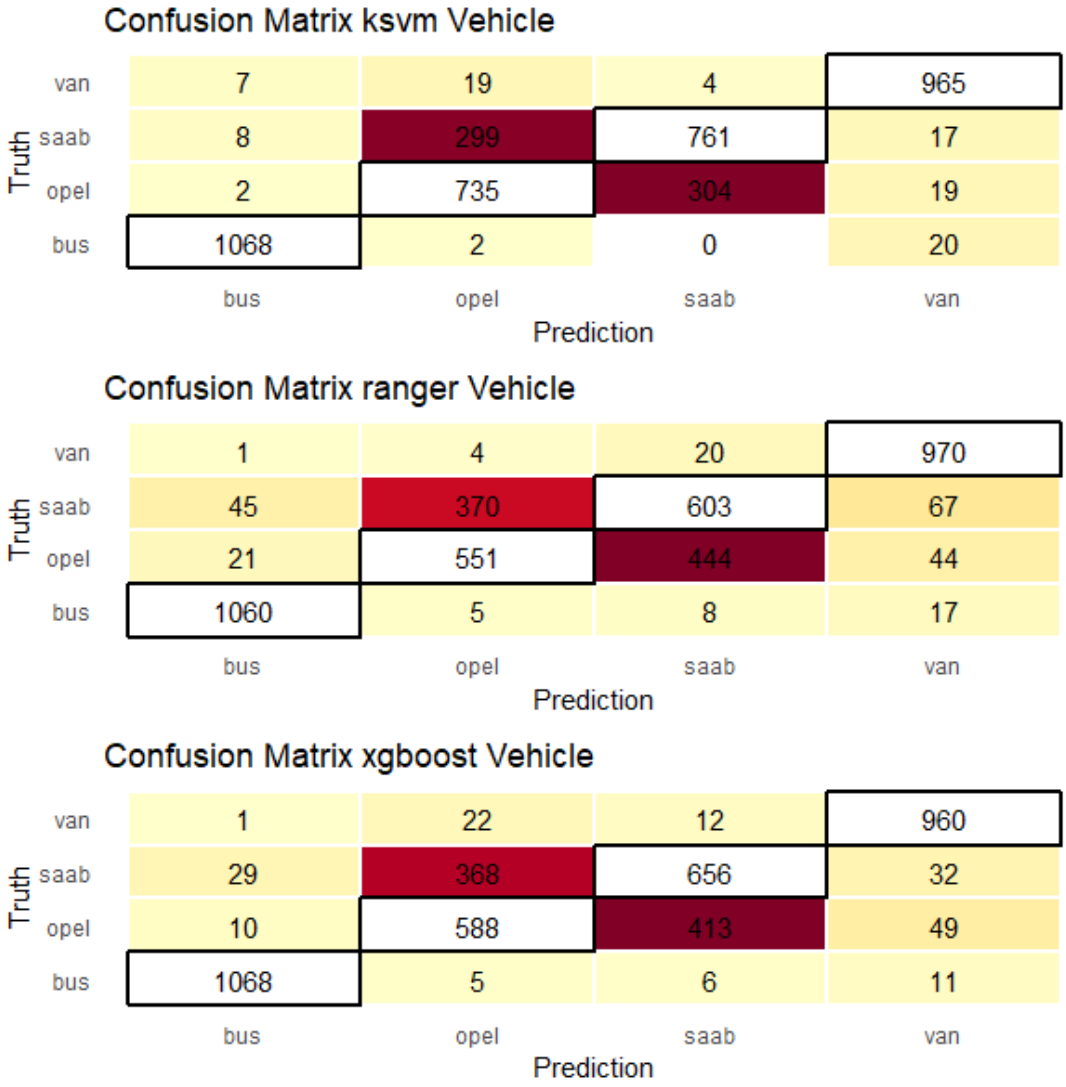

Confusion Matrix xgboost Letter
Truth
Prediction
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

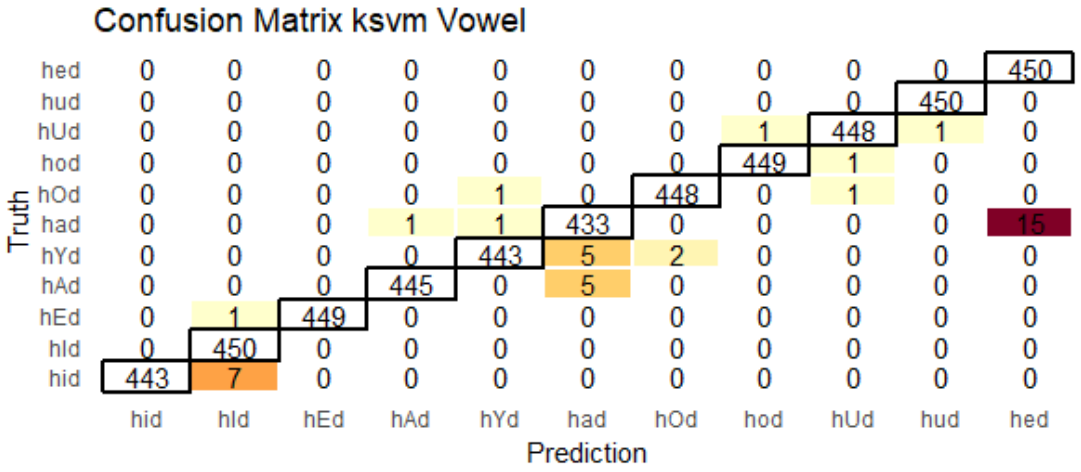

Confusion Matrix ksvm Satellite
very damp grey soil
vegetation stubble
damp grey soil
grey soil
cotton crop
red soil
Truth
red soil
cotton crop
grey soil
damp grey soil
vegetation stubble
very damp grey soil
Prediction

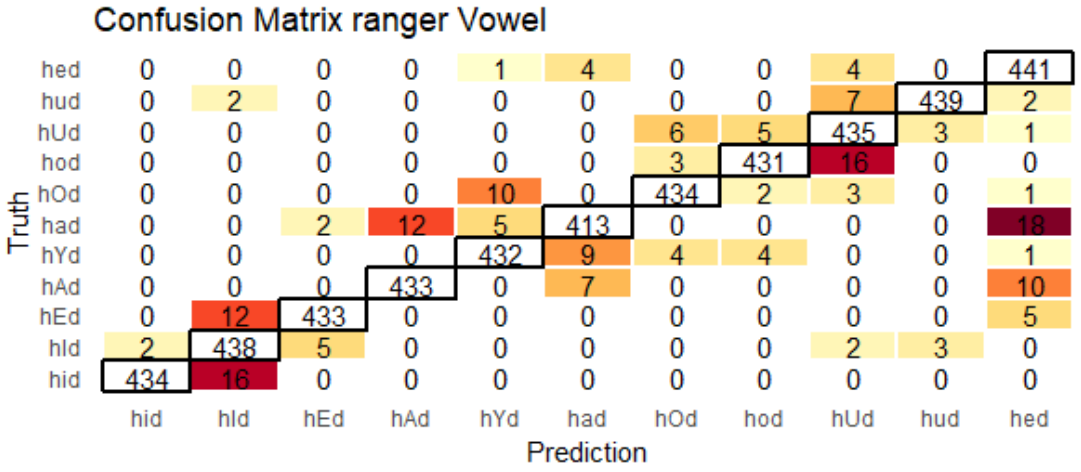

Confusion Matrix ranger Satellite
very damp grey soil
vegetation stubble
damp grey soil
grey soil
cotton crop
red soil
Truth
red soil
cotton crop
grey soil
damp grey soil
vegetation stubble
very damp grey soil
Prediction

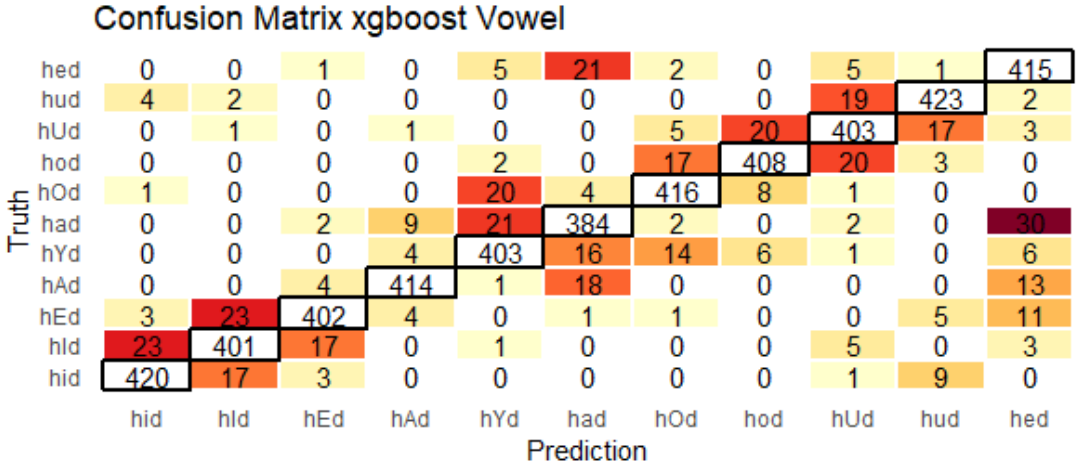

Confusion Matrix xgboost Satellite
very damp grey soil
vegetation stubble
damp grey soil
grey soil
cotton crop
red soil
Truth
red soil
cotton crop
grey soil
damp grey soil
vegetation stubble
very damp grey soil
Prediction

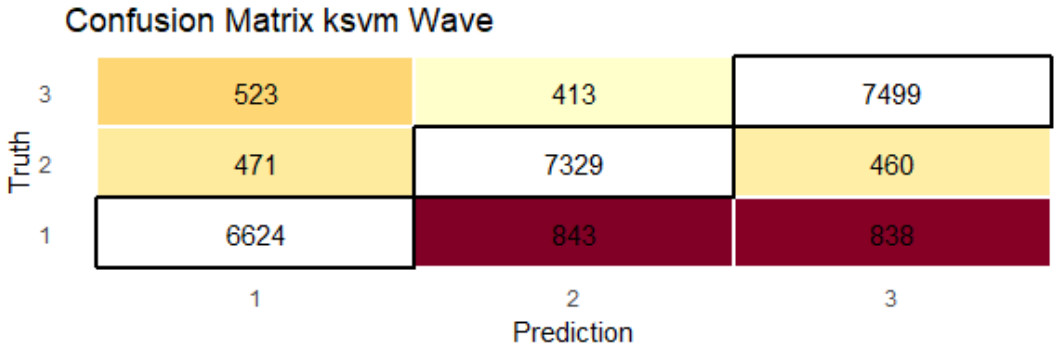

Confusion Matrix ksvm Shuttle
Truth
Bpv.Open 38 0 0 0 0 0 27
Bpv.Close 10 0 0 0 9 31 0
Bypass 64 0 0 0 16271 0 0
High 38 5 7 44465 0 0 0
Fpv.Open 49 0 799 7 0 0 0
Fpv.Close 0 245 0 5 0 0 0
Rad.Flow 227897 5 5 0 0 0 23
Rad.Flow Fpv.Close Fpv.Open High Bypass Bpv.Close Bpv.Open
Prediction
Confusion Matrix ranger Shuttle
Truth
Bpv.Open 0 0 2 0 0 0 63
Bpv.Close 0 0 0 7 5 38 0
Bypass 0 0 0 0 16335 0 0
High 0 4 2 44509 0 0 0
Fpv.Open 5 0 850 0 0 0 0
Fpv.Close 0 245 0 5 0 0 0
Rad.Flow 227925 0 5 0 0 0 0
Rad.Flow Fpv.Close Fpv.Open High Bypass Bpv.Close Bpv.Open
Prediction
Confusion Matrix xgboost Shuttle
Truth
Bpv.Open 0 0 0 0 0 26 39
Bpv.Close 0 1 0 4 7 38 0
Bypass 0 0 0 311 16024 0 0
High 8 0 927 43580 0 0 0
Fpv.Open 0 16 835 4 0 0 0
Fpv.Close 0 245 0 5 0 0 0
Rad.Flow 227925 0 5 0 0 0 0
Rad.Flow Fpv.Close Fpv.Open High Bypass Bpv.Close Bpv.Open
Prediction

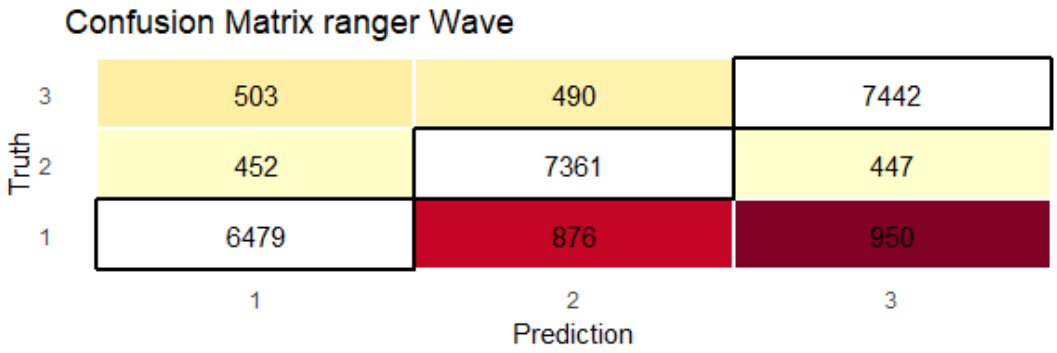

Confusion Matrix ksvm Vehicle
van 7 19 4 965
saab 8 299 761 17
opel 2 735 304 19
bus 1068 2 0 20
Truth
bus opel saab van
Prediction
Confusion Matrix ranger Vehicle
van 1 4 20 970
saab 45 370 603 67
opel 21 551 444 44
bus 1060 5 8 17
Truth
bus opel saab van
Prediction
Confusion Matrix xgboost Vehicle
van 1 22 12 960
saab 29 368 656 32
opel 10 588 413 49
bus 1068 5 6 11
Truth
bus opel saab van
Prediction

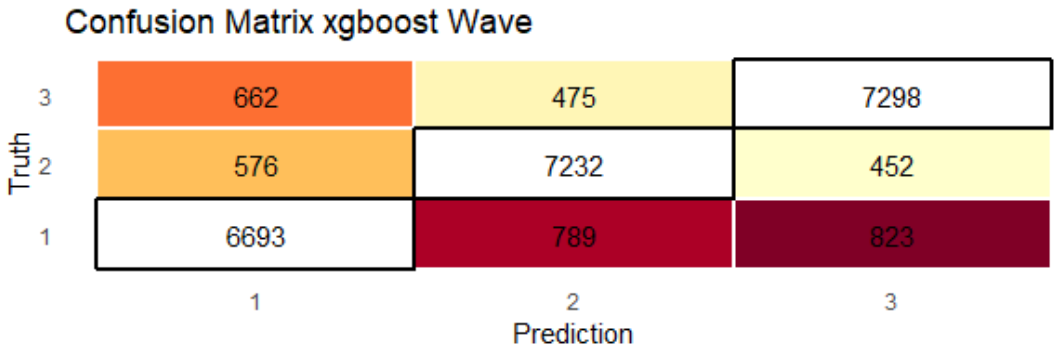

Confusion Matrix ksvm Vowel
Truth
hed 0 0 0 0 0 0 0 0 0 0 450
hud 0 0 0 0 0 0 0 0 0 450 0
hUd 0 0 0 0 0 0 0 1 448 1 0
hod 0 0 0 0 0 0 0 449 1 0 0
hOd 0 0 0 0 1 0 448 0 1 0 0
had 0 0 0 1 1 433 0 0 0 0 15
hYd 0 0 0 0 443 5 2 0 0 0 0
hAd 0 0 0 445 0 5 0 0 0 0 0
hEd 0 1 449 0 0 0 0 0 0 0 0
hId 0 450 0 0 0 0 0 0 0 0 0
hid 443 7 0 0 0 0 0 0 0 0 0
hid hId hEd hAd hYd had hOd hod hUd hud hed
Prediction


Confusion Matrix ranger Vowel
Truth
hed 0 0 0 0 1 4 0 0 4 0 441
hud 0 2 0 0 0 0 0 0 7 439 2
hUd 0 0 0 0 0 0 6 5 435 3 1
hod 0 0 0 0 0 0 3 431 16 0 0
hOd 0 0 0 0 10 0 434 2 3 0 1
had 0 0 2 12 5 413 0 0 0 0 18
hYd 0 0 0 0 432 9 4 4 0 0 1
hAd 0 0 0 433 0 7 0 0 0 0 10
hEd 0 12 433 0 0 0 0 0 0 0 5
hId 2 438 5 0 0 0 0 0 2 3 0
hid 434 16 0 0 0 0 0 0 0 0 0
hid hId hEd hAd hYd had hOd hod hUd hud hed
Prediction


Confusion Matrix xgboost Vowel
Truth
hed 0 0 1 0 5 21 2 0 5 1 415
hud 4 2 0 0 0 0 0 0 19 423 2
hUd 0 1 0 1 0 0 5 20 403 17 3
hod 0 0 0 0 2 0 17 408 20 3 0
hOd 1 0 0 0 20 4 416 8 1 0 0
had 0 0 2 9 21 384 2 0 2 0 30
hYd 0 0 0 4 403 16 14 6 1 0 6
hAd 0 0 4 414 1 18 0 0 0 0 13
hEd 3 23 402 4 0 1 1 0 0 5 11
hId 23 401 17 0 1 0 0 0 5 0 3
hid 420 17 3 0 0 0 0 0 1 9 0
hid hId hEd hAd hYd had hOd hod hUd hud hed
Prediction


Confusion Matrix ksvm Wave
Truth
3
2
1
523
413
7499
471
7329
460
6624
843
838
1
2
3
Prediction


Confusion Matrix ranger Wave
Truth
3
2
1
503
490
7442
452
7361
447
6479
876
950
1
2
3
Prediction


Confusion Matrix xgboost Wave
Truth
3
2
1
662
475
7298
576
7232
452
6693
789
823
1
2
3
Prediction

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

München, am ....

Thomas Westermeier