h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

Gruppe 14
Tamara Döß (767956)
Tobias Rohrer (???)

Data Mining 1

Booklet von Gruppe 14

Inhaltsverzeichnis

Zusammenfassung **iii**

1 Aufgabe 1: Entscheidungsbäume **1**
1.1 Feature Engineering . . . 1
1.1.1 Analyse der Zielvariable . . . 1
1.1.2 Fehlende Werte . . . 1
1.1.3 Merkmalserstellung und Aufteilung . . . 2
1.1.4 Diskretisierung . . . 2
1.1.5 Kodierung kategorischer Werte . . . 2
1.1.6 Bereinigung von Ausreißern . . . 2
1.1.7 Normalisierung der Daten . . . 3
1.2 Entscheidungsbäume . . . 3
1.2.1 Aufteilung in Trainings- und Testdaten . . . 3
1.2.2 Standard Einstellungen . . . 3
1.2.3 Variationen . . . 3
1.2.4 Minimal Cost-Complexity-Pruning . . . 4

2 Aufgabe 2: Neuronale Netze **5**
2.1 Neuronale Netze mit Numpy . . . 5
2.1.1 Implementierung Backprop mit einem Hidden Layer . . . 5
2.1.2 Hyperparameter . . . 6
2.2 Neuronale Netze mit TensorFlow . . . 8
2.2.1 Hyperparameter Suche . . . 8

3 Aufgabe 2: Ensemblemethoden **9**
3.1 Hyperparameter Optimierung . . . 9
3.1.1 Univariat . . . 9
3.1.2 Multivariat . . . 10
3.1.3 Ergebnisse . . . 11

4 Aufgabe 4: Support Vector Machines **12**
4.1 Beschreibung der Kernel Parameter 12
4.2 Modellergebnis 13

A Anhang **I**
A.1 Ergänzende Abbildungen zu Booklet Teil 1 I
A.2 Quellcode zu Booklet Teil 1 II
A.3 Ergänzungen zu Booklet Teil 2 III
A.4 Quellcode zu Booklet Teil 2 IV
A.5 Quellcode zu Booklet Teil 3 V
A.6 Ergänzende Tabellen zu Teil 3 V
A.7 Quellcode zu Booklet Teil 4 VII

Literatur **VIII**

Zusammenfassung

1 Aufgabe 1: Entscheidungsbäume

1.1 Feature Engineering

1.1.1 Analyse der Zielvariable

Wie Abbildung 1 entnommen werden kann, ist die Ausprägung der Zielvariable ungleich auf beide Klassen verteilt. Die Klassifizierungsgenauigkeit eines Modells muss demnach unter Berücksichtigung der sogenannten *Null Accuracy* bewertet werden. Unter *Null Accuracy* versteht man die Genauigkeit eines Modells, dass unabhängig von allen Eingaben immer die am häufigsten auftretende Klasse vorhersagt. In unserem Fall würde ein Modell, welches immer Regen vorhersagt, eine Klassifizierungsgenauigkeit von 79,39% erreichen. Das Ziel der nachfolgenden Schritte ist also ein Modell mit einer besseren Klassifizierungsgenauigkeit als die *Null Accuracy* aufzubauen.

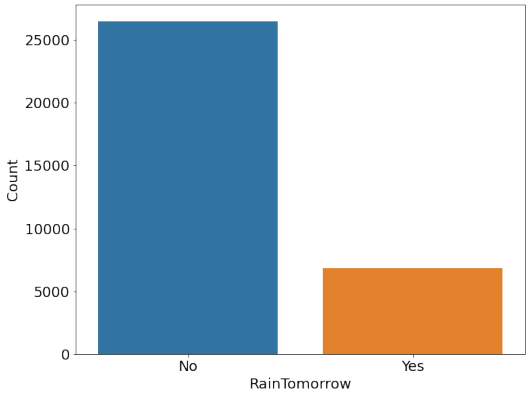


Abbildung 1: Verteilung der Zielvariable

1.1.2 Fehlende Werte

Der zu untersuchende Datensatz beinhaltet fehlende Werte. Im Folgenden werden Methoden beschrieben, wie mit den fehlenden Werte umgegangen wurde:

Fehlende Zielvariable: Im ersten Schritt wurden alle Beobachtungen, welche keinen Wert für die Zielvariable *RainTomorrow* aufweisen, aus dem Datensatz entfernt. Damit wurde die Anzahl an Beobachtungen um 834 auf 33402 reduziert.

Spalten mit fehlenden Werten: In einem nächsten Schritt werden die Spalten aus dem Datensatz entfernt, in denen mehr als 40% der beinhaltenden Variablen fehlen. Namentlich wurden somit die Spalten *Evaporation, Sunshine, Cloud9am* sowie *Cloud3pm* aus dem Datensatz entfernt. Der Schwellwert von 40% wurde empirisch festgelegt und hat zu den besten Klassifizierungsergebnissen geführt.

Beobachtungen mit fehlenden Werten Des Weiteren werden Beobachtungen aus dem Datensatz entfernt, von denen mehr als 50% der Variablen fehlen. Durch diesem Schritt wurden 55 Beobachtungen aus dem Datensatz entfernt.

Imputation Durch die zuvor beschriebenen Methoden ist der Datensatz immer noch nicht frei von fehlenden Werten. Um diese zu ersetzen, werden für kategorische und numerische

Variablen verschiedene Strategien zur Imputation verfolgt. Fehlende numerische Werte werden mit dem Median der jeweiligen Variable ersetzt. Der Median wurde gewählt, da dieser im Vergleich zum Mittelwert robuster gegenüber Ausreißern ist. Für kategorielle Variablen hingegen wird der am häufigsten vorkommende Wert verwendet. Wichtig bei der Ermittlung des Medians bzw. des häufigsten Wertes ist, dass dieser ausschließlich mit Hilfe der Trainingsdaten (siehe Abschnitt 1.2.1) ermittelt wird. Es muss davon ausgegangen werden, dass die Testdaten nicht bekannt sind. Die Ermittlung auf Basis des gesamten Datensatzes, inklusive der Testdaten, würde zu *Data Leakage* führen und ist zu vermeiden. Die auf Basis der Trainingsdaten ermittelten Werte für die Imputation werden auf die Trainings- und Testdaten angewendet.

1.1.3 Merkmalserstellung und Aufteilung

Eine weit verbreitete Technik des Feature Engineerings ist die Erstellung zusätzlicher Merkmalen. Somit wurde die Variable *MinMaxDiff* erstellt, welche die Differenz zwischen der minimalen und der maximalen Tages-Temperatur angibt. Des Weiteren wurden die Variablen *PressureDiff, HumidityDiff und WindSpeedDiff* als Differenz der Beobachtungen am Morgen und Abend erstellt. Das Feld *Datum* wurde in die Merkmale *Year, Month* und *Day* aufgeteilt.

1.1.4 Diskretisierung

Die Diskretisierung eines Merkmals kann eine Überanpassung bei der Erstellung von Modellen verhindern, indem der Wertebereich des Merkmals minimiert und somit generalisiert wird. Hierbei muss beachtet werden, dass der Informationsverlust durch die Diskretisierung nicht zu groß ist. Eine Diskretisierung wurde für das Merkmal *Month* durchgeführt, indem es in das Merkmal *Season* umgewandelt wurde. Das Merkmal *Season* fasst immer 3 Monate zu einer Jahreszeit zusammen.

1.1.5 Kodierung kategorischer Werte

Um kategorische Werte für weitere Analysen verwenden zu können, müssen diese in numerische Werte umkodiert werden. Hierbei wurden die folgenden Strategien Angewendet:

Binäre Kodierung Die Zielvariable *RainTomorrow*, sowie die Variable *RainToday* liegen in den Ausprägungen *Yes* und *No* vor. Für eine weitere Verarbeitung wurden die Ausprägungen in eine numerische binäre Darstellung umgewandelt.

One-Hot-Kodierung Das neu diskretisierte Merkmal *Season* wird mittels *One-Hot-Kodierung* umgewandelt. Eine *Label-Kodierung*, also eine einfache Kodierung mit einem zufälligen Zahlenwert pro auftretender Variablenausprägung, hat den Nachteil, dass dadurch eine Variable entsteht, die gegebenenfalls metrisch interpretiert wird.

Ziel-Kodierung Mit Hilfe der Ziel-Kodierung werden die Merkmale *Location, WindGustDir, WindDir9am* und *WindDir3pm* umgewandelt. Hierbei werden die Merkmalsausprägungen als ihren Einfluss auf die Zielvariable kodiert.

1.1.6 Bereinigung von Ausreißern

Ausreißer können die Performance eines Modells mindern, indem sie als Hebelwerte agieren und somit die Schätzungen der Zielvariable verzerren. Aus diesem Grund werden die Merkmale des Datensatz hinsichtlich ihrer Ausreißer begutachtet. Es wird ersichtlich, dass Merkmale wie

Rainfall und *WindGustSpeed* abweichende Werte aufweisen. Das Entfernen dieser Werte aus dem Datensatz führt jedoch zu einer schlechteren Performance der im folgenden Abschnitt besprochenen Entscheidungsbäume. Deshalb werden die Beobachtungen nicht aus dem Datensatz entfernt.

1.1.7 Normalisierung der Daten

Für Entscheidungsbäume ist eine Normalisierung der Daten nicht Notwendig. Um das *Feature Engineering* jedoch unabhängig vom gewählten Klassifizierer und auch im Hinblick auf neuronale Netze oder *Support Vector Machines* (SVMs) durchzuführen, wird es an dieser Stelle durchgeführt. Die Werte der einzelnen Variablen werden dabei auf den Wertebereich $[0, 1]$ umskaliert.

1.2 Entscheidungsbäume

1.2.1 Aufteilung in Trainings- und Testdaten

Um das Modell nach Abschluss anhand der Klassifizierungsgenauigkeit bewerten zu können, sollte der Datensatz in Trainings- und Testdaten aufgeteilt werden. Die Aufteilung und eine anschließende Bewertung anhand der Testdaten ermöglicht eine Einschätzung der Generalisierungsfähigkeit des Modells. Als Aufteilungsverhältnis wurde 20% Testdaten und 80% Trainingsdaten gewählt. 20% der Daten entsprechen 6670 Datensätzen und bilden eine ausreichend große Menge um die Modellgüte zu bestimmen. Die Wahl des Aufteilungsverhältnisses wurde außerdem nach den Empfehlungen aus Geeron (2017) gewählt.

1.2.2 Standard Einstellungen

Der Entscheidungsbaum wurde mit Hilfe der *scikit-learn*-Bibliothek erstellt Pedregosa et al. (2011). Im ersten Aufgabenteil werden dazu die Standard-Einstellungen des Moduls genutzt. Diese sehen weder eine Beschränkung in der Tiefe des Baumes, noch Kriterien für eine Aufspaltung vor. Als Resultat wächst der Baum weiter, bis alle Blätter im Baum ausschließlich Werte einer Klasse enthalten. Das Ergebnis für die hier untersuchten Wetterdaten ist ein Baum, der sich an die Trainingsdaten überangepasst hat. In Abbildung 2 ist ein Entscheidungsbaum mit den Standard-Einstellungen dargestellt. Die vielen Knoten und Blätter weisen auf eine Überanpassung an die Trainingsdaten hin. Ein weiterer Nachteil des Baums mit Standard-Einstellungen ist, dass die Entscheidungskriterien nur schwer interpretierbar sind. Ein weiterer Hinweis auf eine Überanpassung stellt die Korrektklassifizierungsrate der Trainingsdaten von 100% dar. Zum Vergleich werden nur 79,18% der Testdaten korrekt klassifiziert.

1.2.3 Variationen

Um einen leichter zu interpretierbaren und generalisierbaren Entscheidungsbaum erzeugen zu können, werden verschiedenen Einstellungen für die folgenden Hyperparameter angewandt.

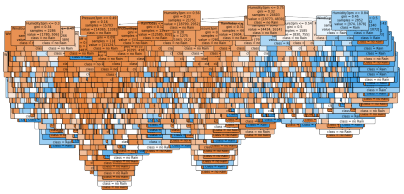


Abbildung 2: Entscheidungsbaum mit Default-Einstellungen

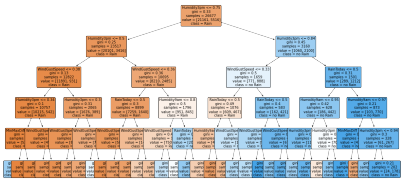


Abbildung 3: Struktur eines Entscheidungsbaums mit *max_ depth = 5*

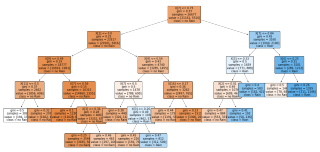


Abbildung 4: Struktur eines Entscheidungsbaums mit *min_ impurity_ decrease = 0.001*

max_ depth definiert die maximale Tiefe des Entscheidungsbaumes. Tiefe Bäume neigen dazu, sich den Trainingsdaten überanzupassen. Flache Bäume dagegen neigen dazu, die Trainingsdaten unteranzupassen. Auf Abbildung 8 aus Kapitel 3.1.1 ist die Balance zwischen über- und unteranpassung mit variierender *max_ depth* dargestellt. Die beste Klassifizierungsgenauigkeit konnte mit einer maximalen Tiefe von 5 erreicht werden. Eine exemplarische Darstellung der Struktur eines Entscheidungsbaums der Tiefe 5 ist in Abbildung 3 dargestellt.

min_ impurity_ decrease legt fest, zu welchem Anteil die Unschärfe reduziert werden muss, sodass eine Aufteilung eines Knotens stattfindet. Dies hat, wie in Abbildung 4 zu sehen, auch eine direkte Auswirkung auf die Tiefe des Entscheidungsbaums. Das Ergebnis einer *Grid Search* nach einer optimalen Einstellung von *min_ impurity_ decrease* kann Abbildung 5 entnommen werden.

criterion definiert das Kriterium, an dem die Qualität einer Aufteilung gemessen wird. Die *scikit-learn*-Bibliothek unterstützt hierbei den Gini-Index, und eine Messung des Informationsgewinns mittels der Entropie. Die Wahl des Gini-Index liefert hierbei unabhängig von den andern beiden Hyperparameter-Einstellungen ein minimal besseres Ergebnis. Außerdem ist die Berechnung des Informationsgewinns mit Hilfe der Entropie durch die logarithmische Funktion aufwändiger. In Raileanu and Stoffel (2004) wird der Unterschied zwischen Informationsgewinn und Gini-Index näher untersucht.

Wie bereits erwähnt haben die beiden Hyperparameter *max_ depth* und *min_ impurity_ decrease* eine Auswirkung auf die Tiefe des Entscheidungsbaums. Das Setzen von *min_ impurity_ decrease* anstelle von *max_ depth* hat den Vorteil, dass dies eine eher weiche Abbruchbedingung während des Trainings darstellt. Somit ist die Gefahr der Unteranpassung geringer. Die Wahl des Gini-Index als Entscheidungskriterium für eine Aufteilung liefert minimal bessere Ergebnisse und ist unaufwändiger zu berechnen. Ein Entscheidungsbaum mit der Einstellung *min_ impurity_ decrease* = 5, sowie der Wahl des Gini-Index hat eine Klassifizierungsgenauigkeit von 84,5% auf die Testdaten.

1.2.4 Minimal Cost-Complexity-Pruning

Eine Möglichkeit der Vermeidung des Overfittings bietet das *Cost-Complexitiy Pruning* als sogenannte *Post-Pruning*-Methode. Dabei wird der voll ausgebildete Baum iterativ beschnitten, indem diejenigen Teilbäume entfernt werden, die einen festgelegten penalisierten Fehlerterm minimieren.

Das Modul *DecisionTreeClassifier* bestimmt für jeden Knoten den effektiven α-Wert. Dabei entpricht α_{eff} dem α, für das gilt: $R_\alpha(T_t) = R_\alpha(t)$. Der Knoten mit dem geringsten effektiven

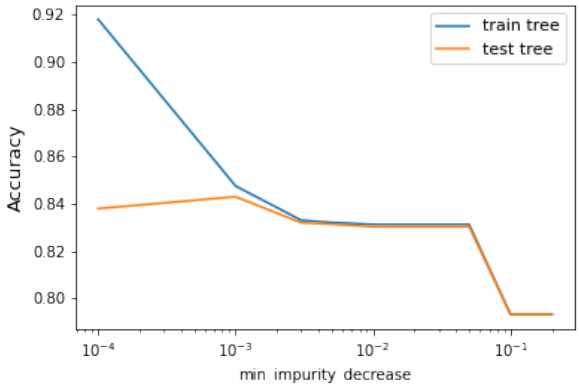


Abbildung 5: Klassifizierungsgenauigkeit eines Entscheidungsbaums abhängig des Hyperparameters *min_impurity_decrease*

α wird vom Baum abgeschnitten. Dieses Vorgehen wird solange wiederholt, bis der geringste effektive α-Wert größer als der gegebenen Penalisierungs-Term *ccp_alpha* ist.

Dabei ist zu beachten, dass die Bäume stärker beschnitten werden, je höher der gegebene Penalisierungs-Term ist, da dieser eine hohe Anzahl an Knoten bestraft (siehe auch A.1).

Das Pruning hat Auswirkungen auf die Modellgüte, indem es die Anpassung an die Trainingsdaten reguliert. Wird der Penalisierungs-Term zu hoch gesetzt, besteht die Gefahr der Unteranpassung, ein sehr niedriger Term führt zu Overfitting. In Abbildung 6 ist beispielhaft für einen Entscheidungsbaum mit Default-Einstellungen die Trainings- und Testgenauigkeit (Accuracy) für verschiedene Pruning-Parameter α dargestellt.

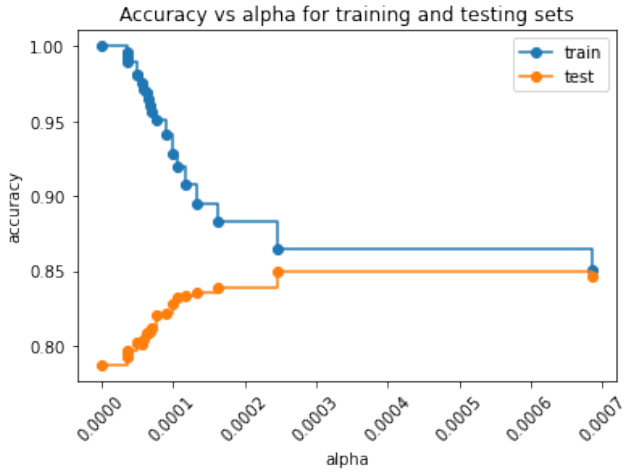


Abbildung 6: Trainings- und Testfehler für verschiedene α-Werte

2 Aufgabe 2: Neuronale Netze

2.1 Neuronale Netze mit Numpy

2.1.1 Implementierung Backprop mit einem Hidden Layer

Die Herleitung der Backpropagation basiert auf der Kostenfunktion, die den Fehler der Output Schicht beschreibt, sowie den Definitionen der einzelnen Gradienten. Es wird mit den Gewichten

zwischen dem Hidden und dem Output Layer gestartet (dW_2), da nur für die Output Schicht ein messbarer Fehler vorliegt. Im ersten Schritt wird die Kostenfunktion in die Definition dW_2 eingesetzt und anschließend die partielle Ableitung gebildet. Diese teilt sich auf in ein δ_2, welches den Fehler und die abgeleitete Aktivierungsfunktion enthält, und den Outputs der Hidden Schicht (a_1). Für die Gewichte des Bias der Hidden Schicht (db_2) wird der Output-Term durch 1 ersetzt, da dieser für den Bias immer konstant 1 beträgt.

Für die Gradienten zwischen der Input Schicht und der Hidden Schicht wir das gleiche Vorgehen genutzt. Da kein direkter Fehler gemessen werden kann, wird der Fehler-Term durch den backpropagierten gewichteten Fehler der Output Schicht ersetzt. Dieser gewichtete Fehler ergibt sich aus dem δ der jeweils nachfolgenden Schicht. Der Output-Term entspricht hier den Inputs (X), die durch die Stichprobendaten gegeben sind.

In Anhang A.3 ist die mathematische Ausarbeitung des beschriebenen Vorgehens zu finden. Zusätzlich findet sich in Anhang A.4 die dazugehörige Python-Implementierung.

2.1.2 Hyperparameter

Im folgenden werden Hyperparameter beschrieben, sowie ihre Auswirkungen auf das neuronale Netzwerk diskutiert.

Anzahl der Neuronen im Hidden Layer

Wenn keine l_1 oder l_2 Regularisierung, Dropout Srivastava et al. (2014) oder andere Regularisierungstechniken eingesetzt werden, steigt die Gefahr einer Überanpassung mit steigender Anzahl an Neuronen im Netzwerk. Werden zu wenige Neuronen im Hidden Layer eingesetzt, steigt hingegen die Gefahr von Underfitting. In der Praxis wählt man eher ein Netzwerk mit zu vielen Neuronen in den Hidden Layern und wirkt Overfitting mit Hilfe von Regularisierungstechniken entgegen Geeron (2017). Tabelle 1 zeigt den Netzwerkfehler abhängig von der Anzahl an Neuronen im Hidden Layer. Anhand der beobachteten Werte aus der Tabelle kann man jedoch nicht auf ein Overfitting bei steigender Neuronen-Anzahl schließen. Denn ab einer Neuronen-Anzahl von 100 steigt der Fehler auf die Trainingsdaten ebenso, wie der Fehler auf die Testdaten. Es ist eher zu beobachten, dass das Netzwerk mit steigender Neuronen Anzahl generell Schwierigkeiten hat, den Fehler zu minimieren.

	10000	1000	100	10	5	2
Test Fehler	62	19,9	14,9	14.9	14.9	15
Trainings Fehler	188	43,5	35,1	35,0	35,0	35,1

Tabelle 1: Netzwerkfehler abhängig von der Anzahl an Neuronen im Hidden Layer.

Anzahl an Iterationen

Die Anzahl der Iterationen, die benötigt werden, bis der Trainingsfehler nicht mehr sinkt, hängt stark von der gewählten Lernrate ab. Im weiteren Verlauf wird das einmalige Iterieren durch den gesamten Trainingsdatensatz als Epoche bezeichnet. In Abbildung 7 ist der Trainingsverlauf während mehrerer Epochen für verschiedene Lernraten abgebildet. Bei einer Lernrate von 0.1 sinkt der Trainingsfehler nach 100 Epochen nicht mehr. Sobald der Trainingsfehler während mehrerer aufeinanderfolgenden Epochen nicht weiter sinkt, sollte das Training beendet werden, da sonst die Gefahr von Overfitting steigt.

Lernrate

Die Lernrate η reguliert die Auswirkung eines einzelnen Schrittes im Gradientenabstiegsverfahrens. Eine hohes η führt zwar zu einer schnellen Minimierung des Trainingsfehlers, Jedoch steig die Gefahr, dass das Verfahren gute Minima überspringt, oder auch um ein gutes Minima pendelt. Ein kleineres η findet mit einer hohen Wahrscheinlichkeit ein besseres Minima, jedoch werden dafür mehr Iterationen und damit auch mehr Rechenleistung benötigt Günter Daniel Rey (2018).

In Abbildung 7 ist der Trainingsfehler während des Trainingsverlauf bei verschiedenen Lernraten dargestellt. Der Abbildung kann entnommen werden, dass für $\eta = 0.01$ das Training nach 300 Epochen immer noch nicht konvergiert ist. Für $\eta = 0.2$ ist zu sehen, dass das Verfahren sehr früh konvergiert und daher das Risiko besteht, dass wie oben beschrieben, gute Minima übersprungen wurden. Für komplexere Eingabedaten und Netzwerkarchitekturen würde dies ein reales Problem darstellen. In diesem einfachen Netzwerk und den einfachen Eingabedaten aus der Aufgabenstellung, kann auch mit $\eta = 0.2$ ein gutes Ergebnis erreicht werden. Der Mittelweg beider Extreme bildet $\eta = 0.1$ und wird als Einstellung vorgenommen.

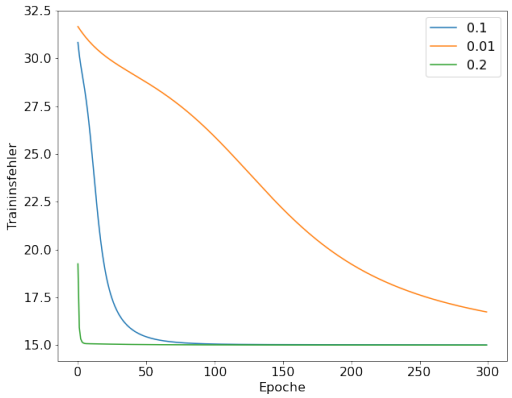


Abbildung 7: Trainingsfehler während der Trainingsepochen bei verschiedenen Lernraten

Initialisierung der Gewichte

Vor Allem beim Trainieren von tiefen neuronalen Netzen spielt die Initialisierung der Gewichte eine entscheidende Rolle um das Auftreten des Problems der explodierenden beziehungsweise verschwindenden Gradienten entgegen zu wirken Geeron (2017). In Glorot and Bengio (2010) wird die Glorot-Initialisierung, ein Verfahren zur Initialisierung der Gewichte bei Verwendung der Sigmoid Aktivierungsfunktion, beschrieben. Hierbei werden die Werte der initialen Gewichte aus einer Normalverteilung $\mathcal{N}(\mu, \sigma^2)$ mit $\mu = 0$ und $\sigma^2 = \frac{1}{fan_{avg}}$ entnommen. Hierbei gilt:

$$fan_{avg} = \frac{(fan_{in} + fan_{out})}{2}$$

Wobei fan_{in} der Anzahl an eingehenden Gewichte in einer Schicht entspricht und fan_{out} der Anzahl an Neuronen in der Schicht. Die Initialisierung der Gewichte in der vorliegenden Arbeit wurde nach Glorot implementiert. Die Gewichte der Biase wurden mit 0 initialisiert.

Würden die Gewichte mit Null initialisiert werden, würde der Gradient für Alle Schichten, bis auf die Ausgabeschicht, Null sein. Ein Lernen würde daher nur sehr begrenzt stattfinden. Der Gradient für die Aktualisierung der Gewichte zwischen versteckter Schicht und Ausgabeschicht würde für alle partiellen Ableitungen gleich sein. Auch eine Initialisierung mit einer anderen Konstanten würde dazu führen, dass alle Gewichte in einer Schicht gleich aktualisiert werden würden. Sie könnten dann auch einfach durch ein einzelnes Neuron mit einer einzigen Verbindung ersetzt werden.

Mithilfe der Verwendung eines *Seeds* wird sicher gestellt, dass die zufällige Initialisierung der Gewichte bei jedem Durchlauf mit den gleichen Parametern die gleichen zufälligen Werte liefert. Das Netzwerk liefert also mit jedem Durchlauf bei gleichen Daten und Parametern die gleichen Ergebnisse. Somit wird die Reproduzierbarkeit im Netzwerk gefördert, was bei einer Fehlersuche und der Hyperparameter Optimierung hilfreich sein kann.

2.2 Neuronale Netze mit TensorFlow

Im Folgenden wird die Implementierung eines voll vernetzen Neuronalen Netzwerks mit Hilfe der *TensorFlow* Bibliothek beschrieben Abadi et al. (2015). Unterstützend wurde für den Aufbau des neuronalen Netzes die *Keras*-API verwendet, die seit *TensorFlow* Version 2 standardmäßig integriert ist.
Um die Daten für das Training des neuronalen Netzes effizient vorzubereiten, wurde die *Dataset*-API verwendet. Mit Hilfe der *Dataset*-API wurde das zufällige Mischen der Trainingsdaten, sowie die Bereitstellung in Form von *Mini-Batches* implementiert. Die Verwendung der *Dataset*-API hat noch zusätzlich den Vorteil, dass die Ausführung von Vorbereitungsschritten der Daten perfekt mit dem Training des neuronalen Netzes koordiniert und parallelisiert werden können.

2.2.1 Hyperparameter Suche

Um ein optimales Modell für die vorliegenden Daten zu finden, wurde eine automatisierte Suche nach den besten Hyperparameter-Kombination implementiert. Die zur Auswahl stehenden Hyperparameter mit ihren möglichen Ausprägungen sind in Tabelle 2 aufgeführt.

Hyperparameter	**Ausprägungen**
Anzahl Hidden Layer	0,1,2,3,4
Anzahl Neuronen pro Schicht	1,3,5,10,20,50,100
Lernrate	0,1; 0,05; 0.01
Aktivierungsfunktion	ReLU, Sigmoid, ELU
Dropout Wahrscheinlichkeit	0; 0.25; 0.5

Tabelle 2: Parameterraum der Hyperparameter Suche

Durch begrenzte Ressourcen wäre eine Evaluierung aller möglichen Hyperparameter-Kombinationen mittels einer *Grid Search* nicht möglich. In der vorliegenden Arbeit wurde deshalb eine zufällige Suche nach Bergstra and Bengio (2012) implementiert. Das Sieger-Modell aus 10 zufälligen Hyperparameter-Kombinationen besteht aus 4 Hidden Layern mit jeweils 3 Neuronen. Die Lernrate wurde auf 0.1 festgelegt und das Netzwerk wurde ohne die Verwendung von Dropout trainiert. Als effektivste Aktivierungsfunktion hat sich die ELU-Funktion herausgestellt Djork-Arné Clevert (2016). Das Sieger-Modell konnte eine Klassifizierungsgenauikgkeit von 85,11%

auf die Testdaten erreichen. Die Daten zum trainieren und testen des neuronalen Netzwerks wurden analog zu Aufgabenstellung 1.2 verwendet. Im Vergleich zu der Klassifizierungsgenauigkeit des Entscheidungsbaums aus 1.2.3 schneidet das entworfene neuronale Netzwerk um 0,61 Prozentpunkte besser ab. Der Nachteil des neuronalen Netzwerks ist jedoch, dass die Entscheidungskriterien im Vergleich zum Entscheidungsbaum nur sehr schwer interpretierbar sind.

3 Aufgabe 2: Ensemblemethoden

	Entscheidungsbaum	AdaBoost	Random Forest	Bagging
n_estimators		100*	100	**100***
criterion	gini	gini	gini	entropy*
max_depth	**5***	**20***	None	10*
min_samples_split	40*	2	**5***	20*
min_samples_leaf	100*	1	1	10*
min_weight_fraction_leaf	0	0	0	0
max_features	25*	None	5*	20*
max_leaf_nodes	None	None	None	None
min_impurity_decrease	0	0	0	0
min_impurity_split	0.1*	0	0	0.1*

Tabelle 3: Optimale Hyperparameter-Einstellung pro Modell durch die univariate *Grid Search*

Im Folgenden Abschnitt wird die Implementierung und Optimierung verschiedener Ensemblemethoden und eines Entscheidungsbaums mit Hilfe der *scikit-learn* Bibliothek beschrieben. Um einen Vergleich zwischen verschiedenen Ensemblemethoden herstellen zu können wurde das *Bagging*- und *AdaBoost*-Verfahren, sowie ein *Random-Forest* implementiert. Alle drei *Ensemblemethoden* wurden mit einem Klassifizierungsbaum als Basisklassifizierer erstellt. Die Klassifizierungsgenauigkeit der einzelnen Verfahren mit Standardeinstellungen, sowie nach einer Hyperparameter-Optimierung können in Tabelle 4 gefunden werden.

3.1 Hyperparameter Optimierung

Um eine optimale Einstellung der Hyperparameter pro Verfahren zu finden, wurde mit Hilfe der *scikit-learn* Bibliothek das *Grid Search*-Verfahren implementiert. Im Gegensatz zu dem *Random Search*-Verfahren, dass in Kapitel 2.2.1 verwendet wurde, sucht *Grid Search* allen möglichen Hyperparameter-Kombinationen in einem gegebenen Parameterraum. Als Metrik für die Optimierung wurde auf Grund der einfachen Interpretierbarkeit die Klassifizierungsgenauigkeit (engl. *Accuracy*) gewählt.

3.1.1 Univariat

Die aus der Aufgabenstellung angegebenen Hyperparameter wurden zunächst einzeln (univariat) variiert, um nach einem optimalen Wert pro Hyperparameter zu suchen. Pro Hyperparameter wird also ein einzelner Parameterraum definiert. Hierbei wurde die Suche in zwei

Durchläufen durchgeführt. Der Hyperparameter, durch dessen Konfiguration weg von seiner Standardeinstellung, die größte positive Auswirkung auf die Klassifizierungsgenauigkeit erzielt werden konnte, wurde für den nächsten Durchgang fest konfiguriert und aus dem zu suchenden Parameterraum entfernt. Die Ergebnisse der *Grid Search* pro Modell und Hyperparameter sind in Tabelle 3 dargestellt. Die **fett** geschriebenen Werte wurden jeweils nach dem Ersten der beiden *Grid Search* Durchläufe festgelegt. Die Werte, die mit * gekennzeichnet sind, weichen von den Standardeinstellungen ab. Zu beachten ist, dass der Hyperparameter *n_estimators* durch eingeschränkte Rechenkapazitäten auf ein oberes Limit von 100 begrenzt wurde. Eine vollständige Auflistung der durchsuchten Parameterräume pro Hyperparameter ist in Tabelle 6 im Anhang zu finden.
In Abbildung 8 ist beispielhaft der erste Durchlauf einer *Grid Search* nach dem Hyperparameter *max_depth* dargestellt. Es ist zu sehen, dass das *Bagging*-Verfahren, sowie der *Random Forest* robuster gegenüber einer Überanpassung an die Trainingsdaten sind. Zu beachten ist, dass die in der Abbildung dargestellten Kurven auf den Train- und Testdaten aus der 10-fachen Kreuzvalidierung basieren.

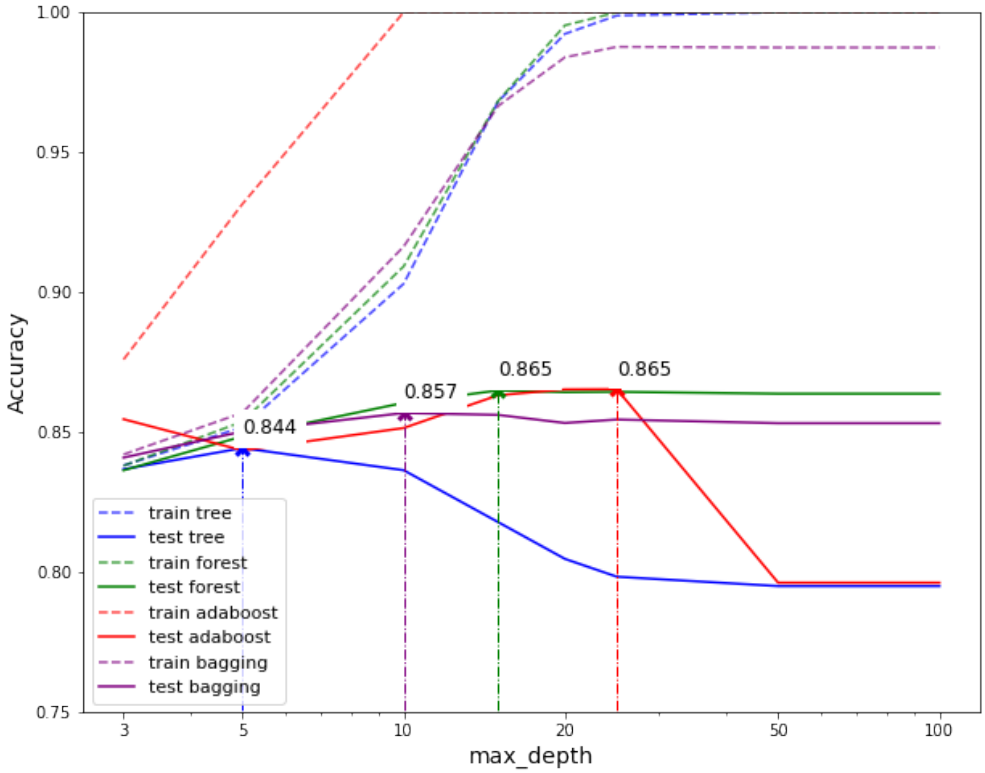


Abbildung 8: Klassifizierungsgenauigkeit abhängig des Hyperparameters *max_depth*

3.1.2 Multivariat

Im Gegensatz zu Abschnitt 3.1.1 werden im folgenden Parameterräume definiert, die mehr als einen Hyperparameter umfassen. Somit können auch optimale Kombinationen von Hyperparameter-Einstellungen untersucht werden. Hierbei sollte darauf geachtet werden, dass die Rechenzeit

einer *Grid Search* exponentiell mit der Größe des Parameterraums ansteigt. Demnach werden folgende Hyperparameter von der Suche ausgeschlossen: *n_ estimators*, *min_ samples_ leaf*, *criterion*, *min_ weigh_ fraction_ leaf*, *max_ leaf_ nodes* und *min_ impurity_ split*. Die gewählten Parameterräume, sowie die gewählte Einstellung pro Hyperparameter ist in Tabellen 7 - 10 im Anhang zu finden. Die Klassifizierungsergebnisse, der durch die multivariate *Grid Serach* optimierten Verfahren, können in Tabelle 4 gefunden werden.

3.1.3 Ergebnisse

Die Klassifizierungsergebnisse der Verfahren mit Standard Einstellungen, sowie nach einer multivariaten und univariaten *Grid Search* sind in Tabelle 4 zu finden. Wie der Tabelle entnommen werden kann, konnten die besten Klassifizierungsergebnisse mit der zweistufigen univariaten *Grid Search* erzielt werden. Die Ergebnisse der Verfahren, die durch die multivariate *Grid Search* optimiert wurden, können wahrscheinlich mit einer Erweiterung des Parameterraums verbessert werden. Dies würde jedoch auf der eingesetzten Hardware zu einem starken Anstieg der Berechnungsdauer führen. Als Alternative für weitergehende Untersuchungen könnte das in Abschnitt 2.2.1 eingesetzte *Random Search* Verfahren angewendet werden.

	Entscheidungsbaum	AdaBoost	Random Forest	Bagging
Default Einstellungen	79.07%	78,64%	86,36%	85,67%
Nach zweistufiger univariater *Grid Search*	84.77%	86,64%	86,43%	86,13%
Nach multivariater *Grid Search*	84,53%	86,64%	86,34%	85,97%

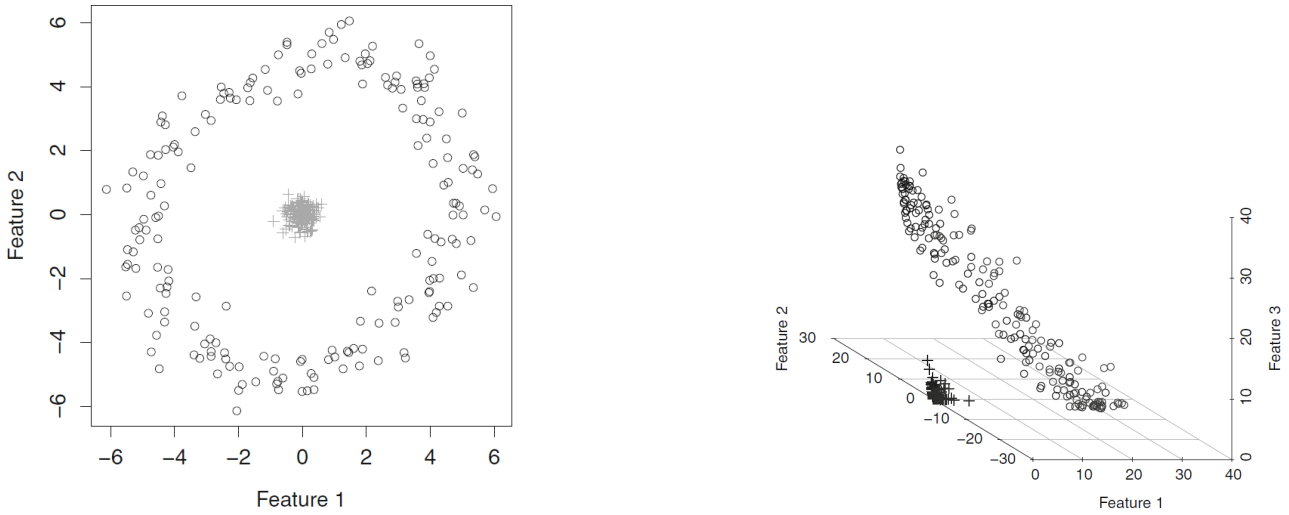
Tabelle 4: Klassifizierungsgenauigkeiten der Ensemblemethoden vor und nach der Optimierung durch *Grid Search*

4 Aufgabe 4: Support Vector Machines

Support Vector Machines (SVMs) werden zur binären Klassifikation eingesetzt. Sie modellieren Hyperebenen, die den Datenraum in zwei zwei Teile einteilen, die jeweils eine der Zielklassen vertreten. Die Idee besteht darin, Trainingsdaten durch eine Hyperebene zu trennen, sodass ein symmetrischer Bereich um diese Hyperebene (*Margin*), welcher keine Datenpunkte enthält, möglichst groß wird (*Maximum Margin Hyperplane*). Da in der Praxis meist keine linear trennbaren Daten vorliegen, wird in der Trainingsphase eine Missklassifikation von Trainingsdaten toleriert. Jeder Datenpunkt, der dabei nicht außerhalb der Margin liegt oder falsch klassifiziert wird, wird entsprechend durch sogenannte Schlupfvariablen penalisiert. Dabei wird auch von einer *Soft Margin* gesprochen (vgl. Aggarwal (2015)).

Ist eine lineare Trennung der Daten nicht sinnvoll, kann der *Kernel Trick* angewandt werden. Dabei wird eine definierte Kernfunktion eingesetzt, die die Daten in eine höhere Dimension transformiert, indem die vorhandenen Merkmale (*Input Space*) in weiteren Merkmalen vereint werden (*Feature Space*). In der höheren Dimension können die Daten dann linear geteilt werden. Das Prinzip ist in den Abbildungen 9a und 9b visualisiert. Die linke Abbildung zeigt die Daten im zweidimensionalen Raum, in welchem sie nicht linear geteilt werden können. Die rechte Abbildung zeigt die transformierten, drei-dimensionalen Daten. Diese können durch eine Hyperebene geteilt werden.

Bei SVMs werden nach der Trainingsphase nicht alle Trainingsdaten zur Klassifizierung neuer Daten genutzt. Diejenigen Datenpunkte, die zur Klassifikation betrachtet werden heißen *Support Vectors*.



(a) zwei-dimensionaler Input-Space (b) drei-dimensionaler Feature-Space

Abbildung 9: Transformation der Daten durch Kernfuntionen (Mello and Ponti (2018))

4.1 Beschreibung der Kernel Parameter

Folgend sind in Tabelle 5 die mathematischen Definitionen der verschiedenen Kernel gelistet. Dabei ist die Einstellung *precomputed* nicht enthalten, da sie keinen definierten Kernel benutzt, sondern nur eine bereits durch einen beliebigen Kernel verarbeitete Matrix entgegennimmt. Zum Beispiel stellt XX^T einen linearen Kernel dar. Die Kernel enthalten Parameter die im Folgenden kurz beschrieben werden.

Linear (linear):	$\langle x, x^{'} \rangle$
Polynomial (poly):	$(\gamma \langle x, x^{'} \rangle + r)^d$
Radial Basis Function (rbf):	$exp(-\gamma \|\|x - x^{'}\|\|^2)$
Sigmoid (sigmoid):	$tanh(\gamma \langle x, x^{'} \rangle) + r)$

Tabelle 5: Mathematische Definitionen der Kernel

c (cost): Dieser Parameter wird bei allen Kernen eingesetzt und gibt an, wie strikt die Margin durchgesetzt wird. Das bedeutet, dass bei kleinem C eine große Margin genutzt wird, das eine höhere Anzahl an Datenpunkten innerhalb der Margin toleriert wird. Bei einem großen Wert in C wird dagegen eine kleine Margin bevorzugt, da Missklassifikationen und Datenpunkte innerhalb der Margin während der Trainingsphase weniger toleriert werden (vgl. Aggarwal (2015)).

gamma (γ): γ definiert den Einfluss der Trainingsdaten auf eine neue Klassifizierung. Je größer γ, desto näher müssen Trainingsdaten an dem neuen Datenpunkt liegen, um einen Einfluss zu haben. Damit steigt die Wahrscheinlichkeit des Overfittings bei einem großen γ.

degree (d): Der Parameter d gibt für den polynomialen Kernel den Grad des Polynoms an. Er kontrolliert die Flexibilität des modellierten Klassifizierers (vgl. Ben-Hur and Weston (2009)). Damit steigt die Wahrscheinlichkeit des Overfittings bei hohem d.

coef0 (r): r kann verwendet werden, um die Datenpunkte zu 'skalieren'. Das heißt, sie werden in einen anderen Wertebereich verschoben. Das kann z.B. bei dem polynomialen Kern verhindern, dass Datenpunkte, für die $\langle x, x^{'} \rangle < 1$ gilt, stark von denen Datenpunkten mit $\langle x, x^{'} \rangle > 1$ separiert werden. Durch r können alle Datenpunkte größer 1 gesetzt werden, sodass dieses Problem umgangen wird.

Die Kernel unterscheiden sich vor allem in ihrer Komplexität. Der Lineare Kernel ist der einfachste und somit robust gegen Overfitting. Er ist jedoch nicht geeignet, wenn die nicht-transformierten Daten nicht linear trennbar sind. Der polynomiale Kernel erhöht die Flexibilität bei $d > 1$, sodass dessen Komplexität von Parameter d bestimmt wird. Der RBF Kernel bildet einen Feature Space mit unendlich vielen Dimensionen und verhält sich ähnlich zu einem *weighted-nearest-neighbour* Klassifizierer, da er mit der euklidischen Distanz ($||x - x^{'}||^2$) arbeitet. Daher hat auch dieser Kernel eine höhere Komplexität. Diese wird indirekt über den γ-Parameter gesteuert. Auch die Komplexität des sigmoid Kernels kann durch *gamma* reguliert werden.

4.2 Modellergebnis

Bei einem Durchlauf mit $C = 0.1$ und allen anderen Parametern auf Default-Einstellung zeigte der polynomiale Kernel die besten Ergebnisse. Durch das GridSearch Verfahren wurde ein X Kernel mit den Parametern a,b,c als bestes Modell geschätzt. Dieses Modell hat eine *accuracy* von X

A Anhang

A.1 Ergänzende Abbildungen zu Booklet Teil 1

Folgende Abbildungen stellen die Entscheidungsbäume der Cost-Complexity Aufgabe dar. Der Unterschied in den Bäumen mit geringerem und höherem α wird für die beiden Baum-Variationen gut ersichtlich. Abbildungen 10 und 11 zeigen je einen vollständig ausgebildeten Baum, der durch Cost-Complexity „beschnitten“ wurden.

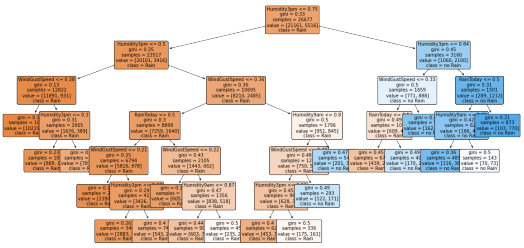


Abbildung 10: Entscheidungsbaum mit Parametern *max_depth=None* und *ccp_alpha=0.0005*

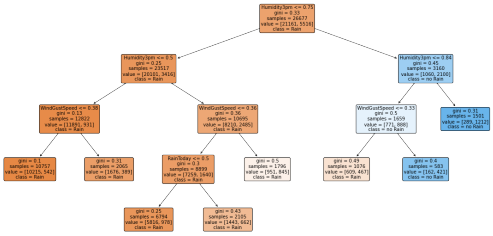


Abbildung 11: Entscheidungsbaum mit Parametern *max_depth=None* und *ccp_alpha=0.002*

Abbildungen 12 und 13 zeigen je einen beschnittenen Baum der mit der Einstellung *max_depth = 8* erstellt wurde.

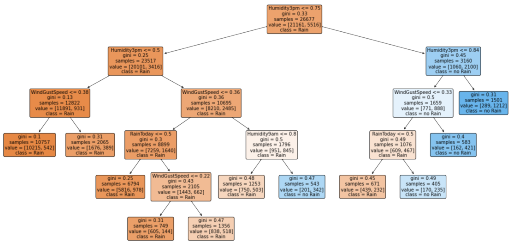


Abbildung 12: Entscheidungsbaum mit Parametern *max_depth=8* und *ccp_alpha=0.001*

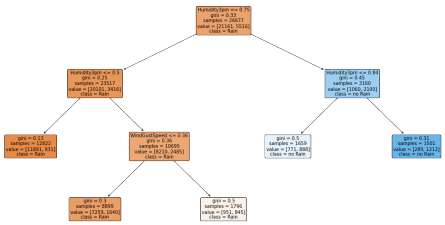


Abbildung 13: Entscheidungsbaum mit Parametern *max_depth=8* und *ccp_alpha=0.004*

A.2 Quellcode zu Booklet Teil 1

A.3 Ergänzungen zu Booklet Teil 2

Folgend ist die mathematische Herleitung der Backpropagation für das in Aufgabe 1 gegebene Modell aufgeführt.

y: Beobachtete Werte der Stichprobe
$a_2 = \hat{\pi}$
σ = Sigmoid-Funktion

$$
\begin{aligned}
dW_2 &= \frac{\partial E^n}{\partial W_2} \\
&= \frac{\partial E^n}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \\
&= \frac{\partial E^n}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \\
&= \frac{\partial}{\partial a_2} \frac{1}{2}(y - a_2)^2 \cdot \frac{\partial}{\partial z_2} \sigma(a_1 W_2 + b_2) \cdot \frac{\partial}{\partial W_2}(a_1 W_2 + b_2) \\
&= -(y - a_2) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot a_1 \\
&= \left[- \begin{pmatrix} y_1 - a_{2;11} \\ \vdots \\ y_n - a_{2;n1} \end{pmatrix} \cdot \begin{pmatrix} \sigma(z_{2;11}) \\ \vdots \\ \sigma(z_{2;n1}) \end{pmatrix} \cdot \begin{pmatrix} 1 - \sigma(z_{2;11}) \\ \vdots \\ 1 - \sigma(z_{2;n1}) \end{pmatrix} \right]^T \cdot \begin{pmatrix} a_{1;11} & \cdots & a_{1;1m} \\ \vdots & \ddots & \vdots \\ a_{1;n1} & \cdots & a_{1;nm} \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \delta_2 &= \frac{\partial E^n}{\partial z_2} \\
\delta_2 &= -(y - a_2) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2))
\end{aligned}
$$

$$
\begin{aligned}
db_2 &= \frac{\partial E^n}{\partial b_2} \\
&= \frac{\partial E^n}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} \\
&= -(y - a_2) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot 1 \\
&= \left[- \begin{pmatrix} y_1 - a_{2;11} \\ \vdots \\ y_n - a_{2;n1} \end{pmatrix} \cdot \begin{pmatrix} \sigma(z_{2;11}) \\ \vdots \\ \sigma(z_{2;n1}) \end{pmatrix} \cdot \begin{pmatrix} 1 - \sigma(z_{2;11}) \\ \vdots \\ 1 - \sigma(z_{2;n1}) \end{pmatrix} \right]^T \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
dW_1 &= \frac{\partial E^n}{\partial W_1} \\
&= \delta_1 \cdot X
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \delta_1 &= \frac{E^n}{\partial z_1} \\
&= \sum_k \frac{\partial E^n}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \\
&= \sum_k \delta_2 \cdot \frac{\partial z_2}{\partial z_1} \\
&= \sum_k \delta_2 \cdot \frac{\partial}{\partial a_1}(a_1 W_2 + b_2) \cdot \frac{\partial}{\partial z_1}\sigma(X \cdot W_1 + b_1) \\
&= \sum_k \delta_2 \cdot W_2 \cdot \sigma(X \cdot W_1 + b_1)(1 - \sigma(X \cdot W_1 + b_1))
\end{aligned}
$$

$$
\Rightarrow dW_1 = \delta_2 \cdot \begin{pmatrix} w_{2;11} \\ \vdots \\ w_{2;m1} \end{pmatrix} \cdot \begin{pmatrix} \sigma(z_{1;11}) & \cdots & \sigma(z_{1;1m}) \\ \vdots & \ddots & \vdots \\ \sigma(z_{1;n1}) & \cdots & \sigma(z_{1;nm}) \end{pmatrix} \cdot \begin{pmatrix} 1 - \sigma(z_{2;11}) \\ \vdots \\ 1 - \sigma(z_{2;n1}) \end{pmatrix} \cdot \begin{pmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix}
$$

$$
\begin{aligned}
db_1 &= \delta_1 \cdot \frac{\partial}{\partial b_1}\sigma(X \cdot W_1 + b_1) \\
&= \delta_1 \cdot 1 \\
&= \delta_2 \cdot \begin{pmatrix} w_{2;11} \\ \vdots \\ w_{2;m1} \end{pmatrix} \cdot \begin{pmatrix} \sigma(z_{1;11}) & \cdots & \sigma(z_{1;1m}) \\ \vdots & \ddots & \vdots \\ \sigma(z_{1;n1}) & \cdots & \sigma(z_{1;nm}) \end{pmatrix} \cdot \begin{pmatrix} 1 - \sigma(z_{2;11}) \\ \vdots \\ 1 - \sigma(z_{2;n1}) \end{pmatrix} \cdot 1
\end{aligned}
$$

A.4 Quellcode zu Booklet Teil 2

A.5 Quellcode zu Booklet Teil 3

A.6 Ergänzende Tabellen zu Teil 3

Hyperparameter	Parameterraum
n_estimators	[10, 50, 100]
criterion	[gini, entropy]
max_depth	[None, 3,5,10, 15, 20, 25, 50]
min_samples_split	[2, 5,10, 20, 30, 40]
min_samples_leaf	[1, 2, 5, 10, 20, 40, 100, 200]
min_weight_fraction_leaf	[0, 0.2, 0.4, 0.5]
max_features	[None, 5, 10, 15 ,20, 25]
max_leaf_nodes	[None, 2 ,10, 100, 150, 200, 300, 500]
min_impurity_decrease	[0.0, 0.001, 0.002, 0.01, 0.1]
min_impurity_split	[0.0, 0.1, 0.2, 0.5, 0.8, 1, 2, 3]

Tabelle 6: Parameterräume der univariaten *Grid Search*

Hyperparameter	Parameterraum	Gewählter Parameter
max_depth	[None, 5, 10, 15]	None
min_samples_split	[2, 5, 10]	5
max_features	[None, 5, 20, 25]	None
min_impurity_decrease	[0.00005, 0.0001, 0.001, 0.003]	0.00005
n_estimators		100

Tabelle 7: Parameterräume und Ergebnisse der multivariaten *Grid Search* des Random Forest

Hyperparameter	Parameterraum	Gewählter Parameter
max_depth	[None, 5, 10, 15]	None
min_samples_split	[2, 5, 10]	5
max_features	[None, 5, 20, 25]	None
min_impurity_decrease	[0.00005, 0.0001, 0.001, 0.003]	0.00001

Tabelle 8: Parameterräume und Ergebnisse der multivariaten *Grid Search* des Entscheidungsbaums

Hyperparameter	Parameterraum	Gewählter Parameter
max_depth	[None, 5, 10, 20. 25]	20
min_samples_split	[2, 5, 10, 20]	2
max_features	[None, 5, 20, 25]	None
min_impurity_decrease	[0.0, 0.0001, 0.0005, 0.001]	0.0
n_estimators		100

Tabelle 9: Parameterräume und Ergebnisse der multivariaten *Grid Search* des Adaboost-Verfahrens

Hyperparameter	Parameterraum	Gewählter Parameter
max_depth	[None, 5, 10, 15]	None
min_samples_split	[2, 5, 10, 20]	10
max_features	[None, 5, 20, 25]	5
min_impurity_decrease	[0.00005, 0.0001, 0.0005, 0.001]	0.0001
n_estimators		100

Tabelle 10: Parameterräume und Ergebnisse der multivariaten *Grid Search* des Bagging-Verfahrens

A.7 Quellcode zu Booklet Teil 4

Literatur

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, and Others (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Aggarwal, C. (2015). *Data Mining: The Textbook*. Springer.

Ben-Hur, A. and J. Weston (2009). A user's guide to support vector machines. *Data Mining Techniques for the Life Sciences*, 223–239.

Bergstra, J. and Y. Bengio (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*(10), 281–305.

Djork-Arné Clevert, Thomas Unterthiner, S. H. (2016). Fast and accurate deep network learning by exponential linear units (elus).

Geeron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*.

Günter Daniel Rey, K. F. W. (2018). *Neuronale Netze*. hogrefe.

Mello, R. and M. Ponti (2018). *Machine Learning: A Practical Approach on the Statistical Learning Theory*. Springer.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

Raileanu, L. E. and K. Stoffel (2004). Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence 41*(1), 77–93.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*(56), 1929–1958.