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

Die vorliegende Arbeit verfolgt das Ziel der theoretischen und praktischen Vorstellung vier unterschiedlicher Data Mining Techniken. Zudem liegt ein Fokus auf der Vorbereitung, die die Datenbereinigung und die Parameterwahl umfasst. Es wird ein Datensatz betrachtet, für den verschiedene Klassifizierer erstellt werden, welche anhand mehrere Merkmale den Wert einer binären Zielvariable bestimmt.

Im ersten Teil wird das Konzept der Entscheidungsbäume vorgestellt. Es wird auf verschiedene Methoden eingegangen, durch die ein Entscheidungsbaum gekürzt werden kann, um seine Performance zu verbessern. Dabei ist ersichtlich geworden, dass im Bereich des *pre-prunings* weiche Abbruchbedingungen in einer besseren Performance resultieren als harte, da sie den Fokus auf Kriterien der einzelnen Knoten legen (z.B. Minimale Reduktion der Ungenauigkeit), statt den Baum z.B. an einer bestimmten Größe zu kappen (Maximale Tiefe). Zudem regulieren sie die Grenze zwischen Über- und Unteranpassung besser. Der beste modellierte Entscheidungsbaum erreicht eine Genauigkeit von 85,07 % durch ein *Cost-Complexity Pruning* mit dem Parameter $\alpha = 0.00025$

Der zweiten Teil beinhaltet das Konzept der neuronalen Netze. Es wird kurz auf das Konzept der Backpropagation eingegangen und wie diese an einem konkreten Beispiel implementiert werden kann. Auch in diesem Kapitel werden einige Parameter und deren Variationen besprochen. Hierbei konnten nicht alle theoretischen Annahmen bestätigt werden. Zum Beispiel konnte bei einer hohen Anzahl an Neuronen keine Überanpassung des Netzwerks festgestellt werden. Jedoch wurde bei einer zu kleinen Lernrate beobachtet, dass das Netz nur sehr langsam optimiert wird. Das beste modellierte neuronale Netz erreicht eine Genauigkeit von 85,11% bei 4 Hidden Layern mit je 3 Neuronen.

Im dritten Teil werden verschiedene Ensemblemethoden an einem Entscheidungsbaum angewandt. Dieses Kapitel legt den Fokus auf das Verfahren der Suche nach optimalen Parameter-Kombinationen, sodass hier die meisten Parameter betrachtet werden. Die Suche wird in univariat und einen multivariat aufgeteilt. Das beste Ergebnis wird mit dem *AdaBoost* Verfahren erzielt und erreicht eine Genauigkeit von 86,64%.

Der letzte Teil dieser Arbeit stellt das Konzept der *Support Vector Machines* vor. Es wird eine kurze theoretische Beschreibung gegeben und auf die unterschiedlichen Parameter und deren Einflüsse auf das Modell und dessen Komplexität eingegangen. Zudem wird der Einsatz verschiedener *Kernel* betrachtet, die jeweils eigene Parameter benutzen. Bei den Support Vector Machines wurde das beste Ergebnis mit dem *Radial Basis Function* Kernel, einem C-Wert von 10 mit dem Default-Wert für $\gamma$ erzeugt. Das erzeugte Modell ht eine Genauigkeit von 86,198%.

Anhand der Ergebnisse ist ersichtlich, dass keine der Methoden deutlich bessere Ergebnisse erzielt als die anderen. Zudem wurde ersichtlich, dass die einzelnen Parameter der Methoden starken Einfluss auf das jeweilige Modell haben.

# 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.

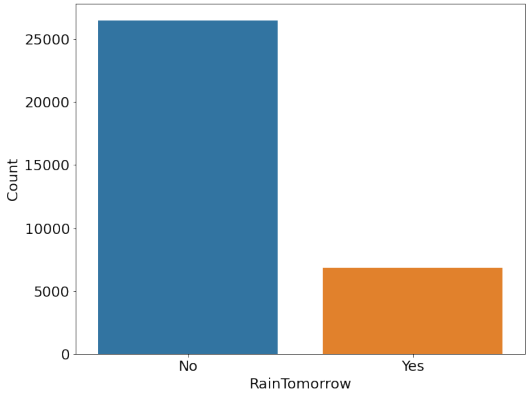



Abbildung 1: Verteilung der Zielvariable

### 1.1.2 Fehlende Werte

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

**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 diese 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. Die Idee, den Median gruppiert nach Region zu nutzen, wurde verworfen, da sie zu keinen besseren Ergebnissen geführt hat und die Berechnung verkomplizierte.

### 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 $[0, 1]$ skaliert.

## 1.2 Entscheidungsbäume

### 1.2.1 Aufteilung in Trainings- und Testdaten

Um das Modell nach Abschluss anhand der Klassifizierungsgenauigkeit bewerten zu können, wurde der Datensatz in Trainings- und Testdaten aufgeteilt. 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.

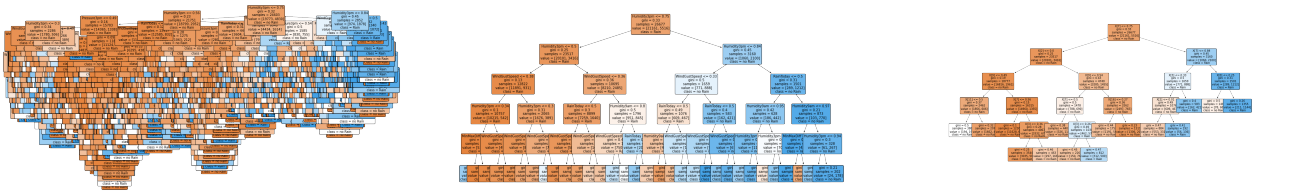
### 1.2.2 Standard Einstellungen

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. Dieser Baum ist in Abbildung 2a 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. Der Baum klassifiziert unbekannte Daten also schlechter als die Null Accuracy angibt.

### 1.2.3 Variationen

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

***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 5 aus Kapitel 3.1.1 ist die Balance zwischen



(a) Default-Einstellungen (b) *max_ depth = 5* (c) *min_ impurity_ decrease = 0.001*

Abbildung 2: Entscheidungsbäume mit Parameter-Variation

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 2b 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 2c 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 3a 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 $min\_impurity\_decrease = 10^{-3}$, sowie der Wahl des Gini-Index hat eine Klassifizierungsgenauigkeit von 84,53% wenn er auf Testdaten angewandt wird.

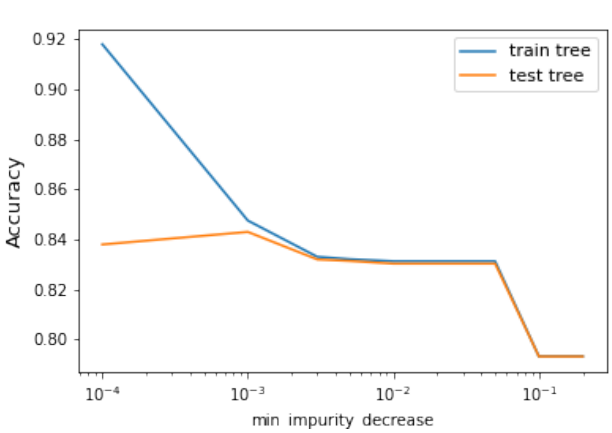
### 1.2.4 Minimal Cost-Complexity-Pruning

Eine Möglichkeit der Vermeidung der Überanpassung bietet das *Cost-Complexitiy Pruning* als sogenannte *Post-Pruning*-Methode. Dabei wird der voll ausgebildete Baum iterativ beschnitten, indem einzelne Teilbäume entfernt werden. Ziel ist es, den Baum zu finden, der die *Cost Comlexity* reduziert.

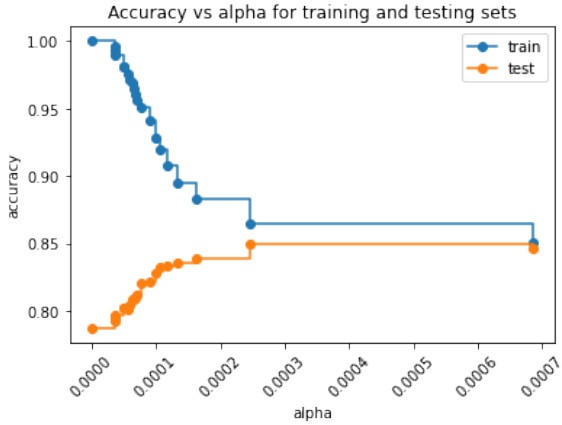
Das Modul *DecisionTreeClassifier* bestimmt für jeden Knoten den *effektiven $\alpha$-Wert ($\alpha_{eff}$)*. Dabei entpricht $\alpha_{eff}$ dem $\alpha$, für das gilt: $R_\alpha(T_t) = R_\alpha(t)$. Der Knoten mit dem geringsten effektiven $\alpha$ wird vom Baum abgeschnitten. Dieses Vorgehen wird solange wiederholt, bis der geringste effektive $\alpha$-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, wohingegen die ursprüngliche Größe keinen Einfluss auf das Endergebnis hat (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 zur Überanpassung. In Abbildung 3b ist beispielhaft für einen Entscheidungsbaum mit Default-Einstellungen die Trainings- und Testgenauigkeit (Accuracy) für verschiedene Pruning-Parameter $\alpha$ dargestellt.




(a)




(b)

Abbildung 3: (a): Klassifizierungsgenauigkeit eines Entscheidungsbaums abhängig des Hyperparameters *min_ impurity_ decrease* (b): Trainings- und Testfehler für verschiedene $\alpha$-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. Diese sind in der Aufgabenstellung gegeben. Auch alle Variablennamen und Indizes werden aus der Aufgabenstellung übernommen. Es wird mit den Gewichten zwischen dem Hidden und dem Output Layer gestartet ($\mathrm{d}W_2$), da nur für die Output Schicht ein messbarer Fehler vorliegt. Im ersten Schritt wird die Kostenfunktion in die Definition $\mathrm{d}W_2$ eingesetzt und anschließend die partielle Ableitung gebildet.

$$dW_2 = \frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$

Diese teilt sich auf in ein $\delta_2$, welches den Fehler und die abgeleitete Aktivierungsfunktion enthält, und den Outputs der Hidden Schicht ($a_1$).

$$\frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = \delta_2 \cdot a_1 = -(y - \hat{\pi}) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot a_1$$

Für die Gewichte des Bias der Hidden Schicht ($\mathrm{d}b_2$) wird der Output-Term durch 1 ersetzt, da dieser für den Bias immer konstant 1 beträgt.

$$db_2 = \frac{\partial E}{\partial b_2} = \delta_2 \cdot 1 = -(y - \hat{\pi}) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot 1$$

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.

$$dW_1 = \frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot X = \delta_2 \cdot \frac{\partial z_2}{\partial z_1} \cdot X$$

Dieser gewichtete Fehler ergibt sich aus dem $\delta$ der jeweils nachfolgenden Schicht (hier $\delta_2$). Der Output-Term der vorherigen Schicht entspricht hier den Inputs ($X$), die durch die Stichprobendaten gegeben sind, sodass gilt:

$$dW_1 = \delta_2 \cdot W_2 \cdot \sigma(X \cdot W_1 + b_1)(1 - \sigma(X \cdot W_1 + b_1)) \cdot X$$

Wobei für $db_1$ statt $X$ wieder 1 eingesetzt wird.

In Anhang A.3 ist die ausführliche 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 4 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 $\eta$ reguliert die Auswirkung eines einzelnen Schrittes im Gradientenabstiegsverfahrens. Eine hohes $\eta$ 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 $\eta$ 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 4 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.

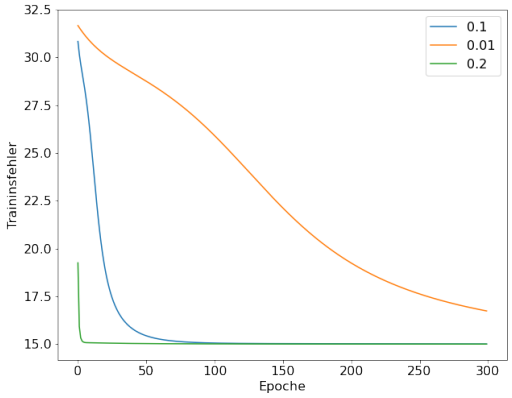



Abbildung 4: Trainingsfehler während der Trainingsepochen bei verschiedenen Lernraten

**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 A.5 zu finden.

In Abbildung 5 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.

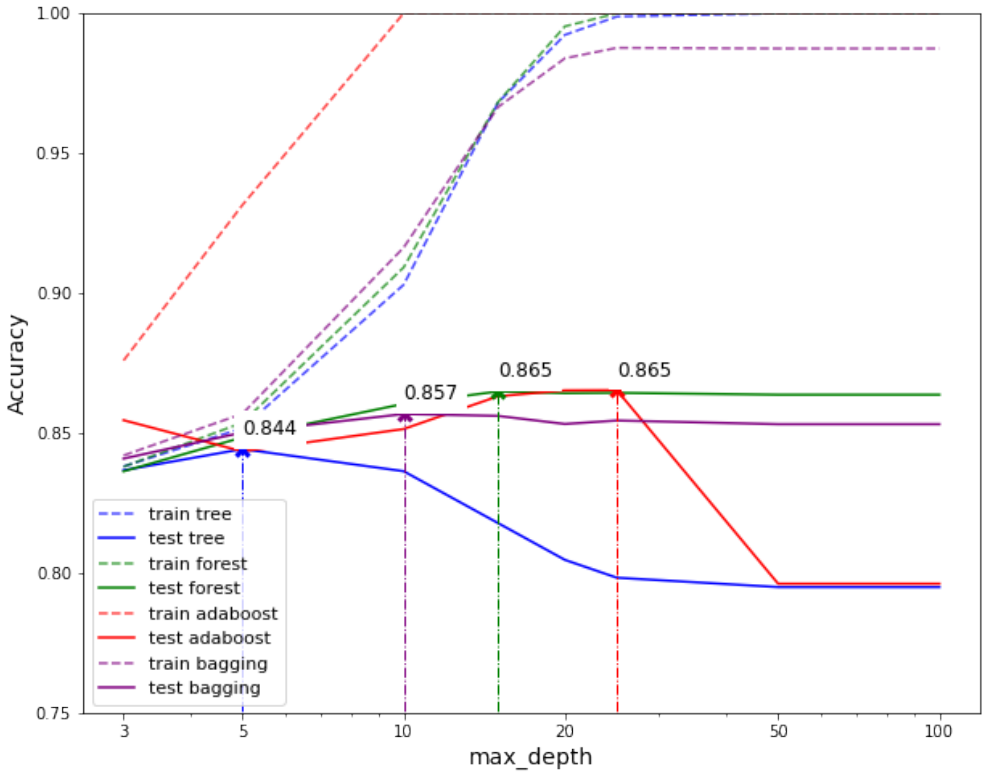



Abbildung 5: Klassifizierungsgenauigkeit abhängig des Hyperparameters *max_ depth*

### 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 A.5 zu finden.

### 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 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.

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

## 4.1 Beschreibung der Kernel Parameter

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

**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 $\gamma$-Parameter gesteuert.

## 4.2 Modellergebnis

Durch das Grid Search Verfahren wurden für jeden Kernel optimale Hyperparameter gesucht. Als bestes Modell hat der RBF Kernel mit den Parametern $C : 10$ und $\gamma :' scale'$ eine Accuracy von 86.198 % erreicht. Tabelle 11 in Anhang A.7 zeigt detailliert die untersuchten Parameterräume und Ergebnisse.

# 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 $\alpha$ wird für die beiden Baum-Variationen gut ersichtlich. Abbildungen 6 und 7 zeigen je einen vollständig ausgebildeten Baum, der durch Cost-Complexity „beschnitten“ wurden.

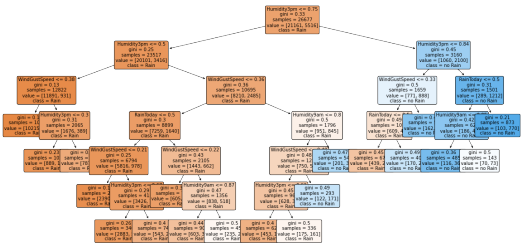


Abbildung 6: Entscheidungsbaum mit Parametern *max_depth=None* und *ccp_alpha=0.0005*

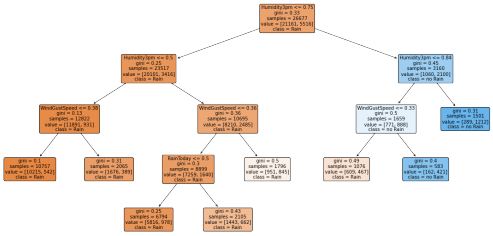


Abbildung 7: Entscheidungsbaum mit Parametern *max_depth=None* und *ccp_alpha=0.002*

Abbildungen 8 und 9 zeigen je einen beschnittenen Baum der mit der Einstellung *max_depth = 8* erstellt wurde.

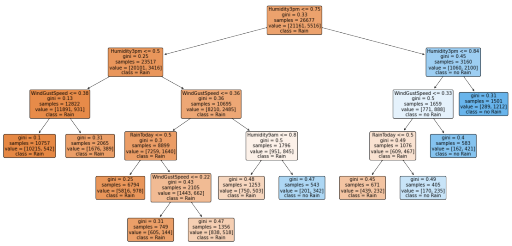


Abbildung 8: Entscheidungsbaum mit Parametern *max_depth=8* und *ccp_alpha=0.001*

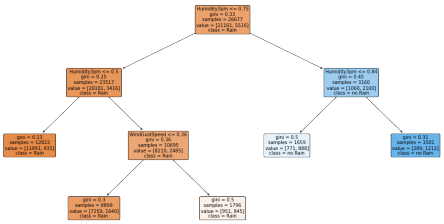


Abbildung 9: Entscheidungsbaum mit Parametern *max_depth=8* und *ccp_alpha=0.004*

Trotz dass der originale Baum ohne Tiefenbegrenzung sehr viel größer ist, als der Baum mit einer maximalen Tiefe von 8, sind die Resultate des Cost-Complexity Prunings ungefähr gleich groß. Die ursprüngliche Größe vor dem Pruning hat somit keinen Einfluss auf das Endergebnis.

## A.2 Quellcode zu Booklet Teil 1

# Booklet_1_Code

August 8, 2020

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.model_selection import train_test_split
import time
import seaborn as sns
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import MinMaxScaler
import eli5
from eli5.sklearn import PermutationImportance

from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel

import category_encoders as ce
```

## 1 Booklet Teil 1

### 1.1 Aufgabe 1: Feature Engineering

[ ]:
```
def filter_years(df):
    # filter Jahre 2013 and 2018 fuer unsere Gruppe
    df['Date'] = pd.to_datetime(df['Date'])
    df = df[df['Date'].dt.year.isin([2013, 2018])]
    return df

weather = pd.read_csv("weatherAUS.csv")  # lese CSV Daten und schreibe sie in␣
↪pandas Data Frame
weather = filter_years(weather)
weather['Date'].dt.year.value_counts()
```

[ ]:
```
# Spalten mit fehlender Zielvariable "RainTomorrow" rausschmeißen.
# Man könnte diese Auch als Testvariablen nehmen, dann wären diese aber nicht␣
↪zufällig ausgewählt...
```

```
weather = weather[weather['RainTomorrow'].notna()]
```

[ ]:
```
# Spalten, in denen mehr als 40% der Variablen fehlen rausscheißen.
# Zeilen, in denen mehr als 50% der Variablen fehlen rausschmeißen.

weather = weather[weather.columns[weather.isnull().mean() < 0.4]]
weather = weather.loc[weather.isnull().mean(axis=1) < 0.5]
```

### 1.1.1 Null Accuracy

[ ]:
```
ax = sns.countplot(x="RainTomorrow", data=weather)

ax.set_xlabel("RainTomorrow", fontsize=18)
ax.set_ylabel('Count', fontsize=18)

plt.xticks(fontsize=18)
plt.yticks(fontsize=18)

plt.savefig('distribution_target_variable.png')
```

### 1.1.2 Feate creation

Erstelle neue Merkmale anhand bereits bestehender Merkmale

[ ]:
```
weather['Year'] = weather['Date'].dt.year  # get year
weather['Month'] = weather['Date'].dt.month  # get month
weather['Day'] = weather['Date'].dt.day  # get day

weather['MinMaxDiff'] = weather['MaxTemp'] - weather['MinTemp']
weather['PressureDiff'] = weather['Pressure3pm'] - weather['Pressure9am']
weather['WindSpeedDiff'] = weather['Pressure3pm'] - weather['WindSpeed9am']
weather['HumidityDiff'] = weather['Humidity3pm'] - weather['Humidity9am']
```

### 1.1.3 Feature Binning

Diskretisiere bestehende Merkmale

[ ]:
```
def encode_season(month):
    if month >= 9 and month <= 11:
        return 'Spring'
    if month == 12 or month <= 2:
        return 'Summer'
    if month >= 3 and month <= 5:
        return 'Autumn'
    if month >= 6 and month <= 8:
        return 'Winter'
```

```
weather['Season'] = weather['Month'].apply(encode_season)
```

[ ]:
```
# Ist quasy "Target Encoding". Nur halt manuell...

def encode_rainly_month(month):
    rainy_month = [5,6, 7,8,11]
    if month in rainy_month:
        return 1
    return 0

weather['RainyMonth'] = weather['Month'].apply(encode_rainly_month)
```

### 1.1.4 Train Test split

**Wichtig:** Bevor weiteres Feature Engineering betrieben wird, müssen die Daten in eine Test- und eine Trainingsmenge aufgeteilt werden, um *Test Train Leakage* zu vermeiden. Die Testmenge darf die Trainingsdaten nicht beeinflussen. Sie gilt als unbekannt.

[ ]:
```
# Zunächst wird noch nicht die Zielvariable "abgespalten"
# Warum? Wenn die Zielvariable noch im gleichen DataFrame ist, kann man␣
↪leichter Outlier rausschmeißen.
train, test = train_test_split(weather, test_size=0.2, random_state = 0)
```

### 1.1.5 Outlier detection (Optional)

**-> Verschlechtert das Ergebnis!**

[ ]:
```
plt.figure(figsize=(15,10))

plt.subplot(2, 2, 1)
fig = train.boxplot(column='Rainfall')
fig.set_title('')
fig.set_ylabel('Rainfall')

plt.subplot(2, 2, 2)
fig = train.boxplot(column='WindGustSpeed')
fig.set_title('')
fig.set_ylabel('WindGustSpeed')
```

[ ]:
```
higher_lim = train['Rainfall'].quantile(0.995)
train = train[train['Rainfall'] < higher_lim]
higher_lim = train['WindGustSpeed'].quantile(0.995)
train = train[train['WindGustSpeed'] < higher_lim]
```

### 1.1.6 Train Test Split part 2

Hier wird jetzt die Zielvariable abgespalten

[ ]:
```
X_train = train.drop(['RainTomorrow'], axis=1)
y_train = train['RainTomorrow']

X_test = test.drop(['RainTomorrow'], axis=1)
y_test = test['RainTomorrow']
```

### 1.1.7 Impute missing Data (Univariat)

[ ]:
```
# Impute values the naiive approache without considering the locations or other␣
↪stuff like season

for dataset in [X_train, X_test]:

    colums_containing_nan = dataset.columns[dataset.isnull().any()]

    numerical_containing_nan = [col for col in colums_containing_nan if␣
↪dataset[col].dtypes != 'O']
    categorial_containing_nan = [col for col in colums_containing_nan if␣
↪dataset[col].dtypes == 'O']

    for col in numerical_containing_nan:
        col_median=X_train[col].median() #always use median from Train data !␣
↪Never impute based on Test Data ! we have to assume we dont know it.
        dataset[col] = dataset[col].fillna(col_median)

    for col in categorial_containing_nan:
        col_most_occuring = X_train[col].mode()[0]
        dataset[col] = dataset[col].fillna(col_most_occuring)
```

### 1.1.8 Encoding Categorial Variables

[ ]:
```
X_train['RainToday'] = X_train["RainToday"].replace({'No':0, 'Yes':1})
X_test['RainToday'] = X_test["RainToday"].replace({'No':0, 'Yes':1})

y_train = y_train.replace({'No':0, 'Yes':1})
y_test = y_test.replace({'No':0, 'Yes':1})
```

### 1.1.9 Target encoding

Beim target Encoding kodieren wir die Variable als Einfluss auf die Zielvariable. Wenn es also in Perth zu 20% geregnet hat, dann wird Perth mit 0.2 kodiert.

[ ]:
```
# Create the encoder
cat_features=['Location','WindGustDir',"WindDir9am", "WindDir3pm"]
for feature in [cat_features]:
```

```
    target_enc = ce.TargetEncoder(cols=feature)
    target_enc.fit(X_train[feature], y_train)

    # Transform the features, rename the columns with _target suffix, and join␣
↪to dataframe
    X_train = X_train.join(target_enc.transform(X_train[feature]).
↪add_suffix('_target'))
    X_test = X_test.join(target_enc.transform(X_test[feature]).
↪add_suffix('_target'))
```

#### 1.1.10 One Hot Encoding

[ ]:
```
# apply One Hot encoding

for col in ["Season"]:
    encoded_columns = pd.get_dummies(X_train[col], prefix=col, drop_first=True)
    X_train = X_train.join(encoded_columns).drop(col, axis=1)

    encoded_columns = pd.get_dummies(X_test[col], prefix=col, drop_first=True)
    X_test = X_test.join(encoded_columns).drop(col, axis=1)
```

#### 1.1.11 Remove Features

Einige Merkmale wurde in andere Merkmale transformiert, sodass die originalen Merkmale verworfen werden können.

[ ]:
```
columns_to_drop = ['Date', 'Location','WindGustDir',"WindDir9am", "WindDir3pm"]
X_train.drop(labels=columns_to_drop, axis=1, inplace=True)
X_test.drop(labels=columns_to_drop, axis=1, inplace=True)
```

#### 1.1.12 Scale numerical features

**Bemerkung:** Dieser Schritt ist nicht notwendig für Entscheidungsbäume. Jedoch können diese auch mit skalierten Daten arbeiten, sodass an dieser Stelle bereits skaliert wird. Die Skalierung wird benötigt, falls eine Variablenselektion durchgeführt wird. Zudem arbeiten zum Beispiel neuronale Netze besser mit skalierten Daten.

[ ]:
```
scaler = MinMaxScaler()
cols = X_train.columns

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = pd.DataFrame(X_train, columns=cols)
X_test = pd.DataFrame(np.array(X_test), columns=cols)
```

### 1.1.13 Feature Selection (Optional)

**Bemerkung**: wurde im finalen Stand nicht ausgeführt

[ ]:
```
selected_columns = weather.columns
```

**Univariat**

[ ]:
```
# zeige correlations matrix
train = X_train
train["RainTomorrow"] = y_train.values
train.corr()
```

[ ]:
```
# zeige heatmap
plt.imshow(train.corr(), cmap='hot', interpolation='nearest')
plt.show()
```

[ ]:
```
NUM_FEATURES_TO_SELECT = 5

selector = SelectKBest(f_classif, k=NUM_FEATURES_TO_SELECT)

X_new = selector.fit_transform(X_train, y_train)

selected_features = pd.DataFrame(selector.inverse_transform(X_new),
                                 index=X_train.index,
                                 columns=X_train.columns)

selected_columns = selected_features.columns[selected_features.var() != 0]

X_train[selected_columns].head()
```

**Multivariat**

[ ]:
```
# Regularisation strengh is set by regularization parameter C.
# Note: the lower C, the higher the regularization
logistic = LogisticRegression(C=0.01, penalty="l1", solver='liblinear',␣
 ↪max_iter=10000, random_state=7).fit(X_train, y_train)
model = SelectFromModel(logistic, prefit=True)

X_new = model.transform(X_train)

selected_features = pd.DataFrame(model.inverse_transform(X_new),
                                 index=X_train.index,
                                 columns=X_train.columns)

# Dropped columns have values of all 0s, keep other columns
selected_columns = selected_features.columns[selected_features.var() != 0]
selected_columns
```

[ ]:
```
# getting accuracy for a logistic regression model
logistic = LogisticRegression().fit(X_train[selected_columns], y_train)

test_predictions = logistic.predict(X_test[selected_columns]).round().
 ↪astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.2 Aufgabe 2 Entscheidungsbäume

[ ]:
```
# Default-Einstellungen
model = tree.DecisionTreeClassifier()
model.fit(X_train, y_train)

# Errechne Genauigkeit und Testfehler
test_predictions = model.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (20,10))
tree.plot_tree(model, ax=ax,
               feature_names=selected_columns,
               filled = True,
               rounded = True,
               precision =2,
               fontsize = 10,
               class_names=["no Rain", "Rain"]
               );
```

[ ]:
```
# max_depth Variation
model1 = tree.DecisionTreeClassifier(max_depth=5)
model1.fit(X_train, y_train)

test_predictions1 = model1.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions1))
mean_absolute_error(y_test, test_predictions1)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (20,10))
tree.plot_tree(model1, ax=ax,
                feature_names=selected_columns,
                class_names= ["Rain", "no Rain"],
                filled = True,
                rounded = True,
                precision = 2,
                fontsize = 10);
```

[ ]:
```
# min_impurity_increase variation
model2 = tree.DecisionTreeClassifier(min_impurity_decrease=0.001)
model2.fit(X_train, y_train)

test_predictions2 = model2.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions2))
mean_absolute_error(y_test, test_predictions2)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (20,10))
tree.plot_tree(model2, ax=ax,
               feature_names=selected_columns,
               class_names= ["Rain", "no Rain"],
               filled = True,
               rounded = True,
               precision = 2,
               fontsize = 10);
```

[ ]:
```
# criterion variation
model3 = tree.DecisionTreeClassifier(criterion="entropy", max_depth=5)
model3.fit(X_train, y_train)

test_predictions3 = model3.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions3))
mean_absolute_error(y_test, test_predictions3)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (20,10))
tree.plot_tree(model3, ax=ax,
               feature_names=selected_columns,
               class_names= ["Rain", "no Rain"],
               filled = True,
               rounded = True,
               precision = 2,
               fontsize = 10);
```

### 1.2.1 Analysis

[ ]:
```
perm = PermutationImportance(model, random_state=1).fit(X_test, y_test)
eli5.show_weights(perm, feature_names = X_test.columns.tolist())
```

### 1.2.2 Cost-Complexity Pruning

Führe mit mind. 3 Bäumen unterschiedlicher Tiefe aus Aufgabenteil c) ein Minimal Cost Complexity Pruning durch. Wie verändern dich die Bäume bei Variation des Prunings? Welche Auswirkung auf die Modellgüte hat das?

[ ]:
```
model = tree.DecisionTreeClassifier()
path = model.cost_complexity_pruning_path(X_train, y_train)
```

```
ccp_alphas, impurities = path.ccp_alphas, path.impurities
ccp_alphas.size
```

[ ]:
```
# nur einen Teil der Ergebnisse nutzen zur Beschleunigung
ccp_alphas_part = ccp_alphas[[0, 68, 136, 204, 272, 340, 408, 476, 544, 612,␣
↪680, 748, 816, 884, 952, 1020, 1088, 1156, 1224, 1292, (ccp_alphas.size-20)]]
```

[ ]:
```
fig, ax = plt.subplots()
ax.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
```

[ ]:
```
model = tree.DecisionTreeClassifier(ccp_alpha=0.00025)
model.fit(X_train, y_train)

test_predictions = model.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

[ ]:
```
fig, ax = plt.subplots(figsize = (20,10))
tree.plot_tree(model, ax=ax,
               feature_names=selected_columns,
               class_names= ["Rain", "no Rain"],
               filled = True,
               rounded = True,
               precision = 2,
               fontsize = 10);
```

[ ]:
```
clfs = []
for ccp_alpha in ccp_alphas_part:
    clf = tree.DecisionTreeClassifier(ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
      clfs[-1].tree_.node_count, ccp_alphas[-1]))
```

[ ]:
```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas_part, train_scores, marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas_part, test_scores, marker='o', label="test",
```

```
        drawstyle="steps-post")
ax.legend()
plt.setp(ax.xaxis.get_majorticklabels(), rotation=45)

plt.show()
```

## A.3 Ergänzungen zu Booklet Teil 2

Als Ergänzung zur Aufgabenstellung und der folgenden mathematischen Herleitung der Backpropagation ist das beschriebene Model in Abbildung 10 schematisch mit den entsprechenden Variablennamen dargestellt.

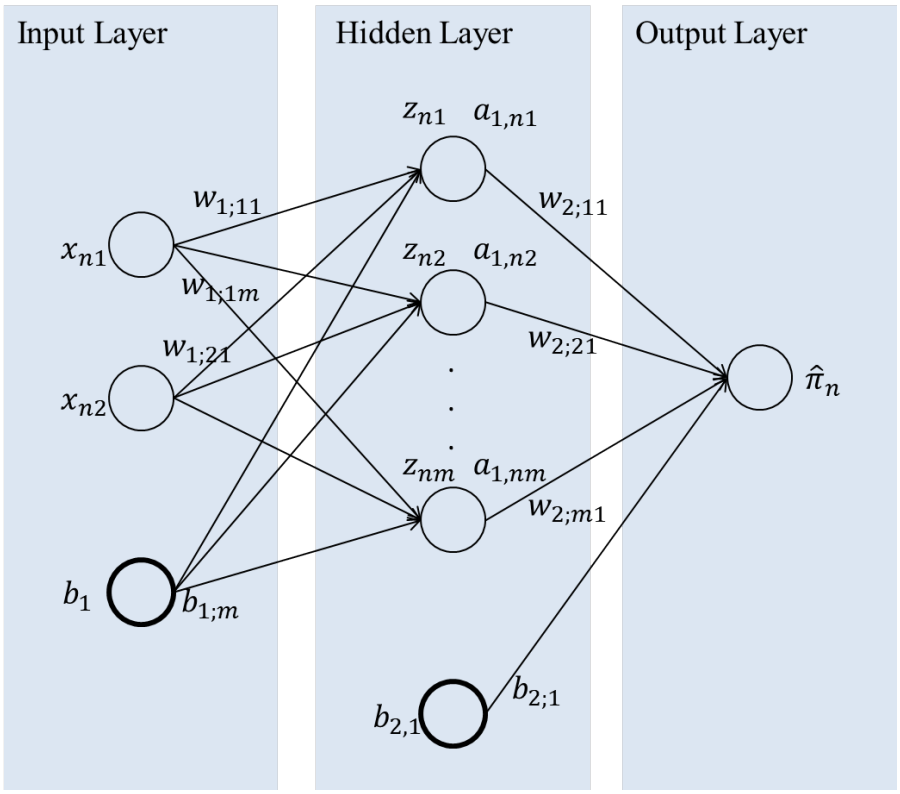



Abbildung 10: Schematische Darstellung des betrachteten Neuronalen Netzes

y: Beobachtete Werte der Stichprobe
$\sigma$ = Sigmoid-Funktion

$$
\begin{aligned}
dW_2 &= \frac{\partial E}{\partial W_2} \\
&= \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \\
&= \frac{\partial E}{\partial \hat{\pi}} \cdot \frac{\partial \hat{\pi}}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \\
&= \frac{\partial}{\partial \hat{\pi}} \frac{1}{2}(y - \hat{\pi} i)^2 \cdot \frac{\partial}{\partial z_2} \sigma(a_1 W_2 + b_2) \cdot \frac{\partial}{\partial W_2}(a_1 W_2 + b_2) \\
&= -(y - \hat{\pi}) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot a_1 \\
\Rightarrow \delta_2 &= \frac{\partial E}{\partial z_2} \\
\delta_2 &= -(y - \hat{\pi}) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2))
\end{aligned}
$$

$$
\begin{aligned}
db_2 &= \frac{\partial E}{\partial b_2} \\
&= \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} \\
&= \frac{\partial E}{\partial \hat{\pi}} \cdot \frac{\partial \hat{\pi}}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} \\
&= \delta_2 \cdot 1 \\
&= -(y - \hat{\pi}) \cdot \sigma(a_1 W_2 + b_2)(1 - \sigma(a_1 W_2 + b_2)) \cdot 1
\end{aligned}
$$

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

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

Bemerkung zu $\delta_1$: der Parameter $k$ entspricht der Anzahl an Knoten im Output Layer. Da das betrachtete Modell nur einen Knoten im Output Layer besitzt, kann diese Summe weggelassen

werden, da es nur einen Summanden gibt.

$$\begin{aligned} db_1 &= \delta_1 \cdot \frac{\partial}{\partial b_1} \sigma(X \cdot W_1 + b_1) \\ &= \delta_1 \cdot 1 \end{aligned}$$

## A.4 Quellcode zu Booklet Teil 2

# Booklet_2_Code_A1

August 8, 2020

## 1 Neuronale Netze - Beispielprogramm

### 1.0.1 Erweiterung des Beispielprogramms um einen Hidden Layer

### 1.0.2 1 - Preparations

### 1.1 - Imports

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
import sklearn
import sklearn.datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score
import pandas as pd

# Display plots inline and change default figure size
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
```

### 1.2 - Generating a dataset

[ ]:
```
# Generate a dataset and plot it
n          = 1000
data_seed  = 1337
split_seed = 42
test_size  = 0.25

np.random.seed(data_seed)
X, y = sklearn.datasets.make_moons(n, noise=0.25)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral);
```

### 1.3 - Split the dataset

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=test_size,
```

```
                                                    random_state=split_seed)
```

### 1.0.3 2- Implementierung

**2.1 - Aktivierungsfunktion (activation function)** Als Aktivierungsfunktion des Output-Layers wird die logistische Funktion

$$\sigma\colon \mathbb{R} \to (0,1)\,,\ z \mapsto \frac{1}{1+\exp(-z)}$$

verwendet.

[5]:
```
def sigmoid(z, derivation = False):
    if derivation:
        return sigmoid(z)*(1-sigmoid(z))
    else:
        return 1/(1+np.exp(-z))

value = 1
print(sigmoid(value))
print(sigmoid(value,True))
```

```
0.7310585786300049
0.19661193324148185
```

**2.2 Kostenfunktion (cost function)** Als Kostenfunktion wird folgende quadratische Fehlerfunktion verwendet:

$$E(y,\widehat{\pi}) = \frac{1}{2}\sum_{k=1}^{n}(y_k - \widehat{\pi_k})^2$$

wobei $y = (y_1,\ldots,y_n)^T$ und $\widehat{\pi} = (\widehat{\pi_1},\ldots,\widehat{\pi_n})^T$.

# 2 Booklet Teil 2

**hier startet unsere Implementierung:**

[6]:
```
class NeuralNet:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate=0.
↪1):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.learning_rate = learning_rate

        # set random seed to get reproducable results
        np.random.seed(5)

        # Glorot initialisation
```

```
        self.W_input_hidden = np.random.normal(0.0,
                                               1/((self.input_nodes*self.
↪hidden_nodes+self.hidden_nodes)/2),
                                               (self.hidden_nodes, self.
↪input_nodes))

        self.W_hidden_output = np.random.normal(0.0,
                                               1/((self.hidden_nodes*self.
↪output_nodes+self.output_nodes)/2),
                                               (self.output_nodes, self.
↪hidden_nodes))

        self.bias_input = np.zeros((self.hidden_nodes, 1))

        self.bias_hidden = np.zeros((self.output_nodes, 1))

    def calculate_loss(self, inputs,labels):
        predictions = self.predict(inputs)

        # Berechnung des Kostenfunktionswertes
        cost = np.power(predictions-labels,2)
        cost = np.sum(cost)/2

        return cost


    def fit(self, inputs, label):
        label = np.array(label, ndmin=2).T
        output, hidden_outputs = self.predict(inputs, backprop=True)

        # Update weights from hidden to output layer
        output_error = output - label  #getting this from the derived squared␣
↪error loss function
        # note that "output_error*outputs*(1.0-outputs)" comes from the␣
↪derivate of sigmoid acrivation.
        gradient_weights_hidden_output = np.dot((output_error*output*(1.
↪0-output)), np.transpose(hidden_outputs))
        self.W_hidden_output += -self.learning_rate *␣
↪gradient_weights_hidden_output

        # update weights of bias_hidden
        gradient_weights_bias_hidden = output_error*output*(1.0-output)
        self.bias_hidden += - self.learning_rate * gradient_weights_bias_hidden

        # Update weights from input to hidden layer
```

```
        # first "propagate" the errors back to the hidden layer.
        hidden_errors = np.dot(self.W_hidden_output.T, (output_error*output*(1.
↪0-output)))

        # this step is the same as from the previous update. just one layer␣
↪closer to the input.
        gradient_weights_input_hidden = np.dot((hidden_errors * hidden_outputs␣
↪* (1.0 - hidden_outputs)),
                                               np.array(inputs, ndmin=2))
        self.W_input_hidden += -self.learning_rate *␣
↪gradient_weights_input_hidden

        # update weights of bias_input
        gradient_weights_bias_input = (hidden_errors * hidden_outputs * (1.0 -␣
↪hidden_outputs))
        self.bias_input = self.bias_input - self.learning_rate *␣
↪gradient_weights_bias_input

        pass

    def predict(self, inputs, backprop=False):

        inputs = np.array(inputs, ndmin=2).T

        # feedforward from input layer to hidden layer
        hidden_inputs = np.dot(self.W_input_hidden, inputs) + self.bias_input
        hidden_outputs = sigmoid(hidden_inputs)

        # feedforward from hidden layer to output layer
        output_inputs = np.dot(self.W_hidden_output, hidden_outputs) + self.
↪bias_hidden
        output_outputs = sigmoid(output_inputs)

        if backprop == True:
            # returning hidden outputs which we need in case of backpropagation
            return (output_outputs, hidden_outputs)

        return output_outputs
```

[7]:
```
# create nn
nn = NeuralNet(input_nodes=2, hidden_nodes=100, output_nodes=1, learning_rate=0.
↪1)
```

[8]:
```
# train nn
for epoch in range(15):
    for sample, sample_label, in zip(X_train, y_train):
```

```
        nn.fit(sample, sample_label)
```

[9]:
```
nn.calculate_loss(X_test, y_test)
```

[9]: 14.84704733671989

[10]:
```
# calculate the accuracy
predictions = nn.predict(X_test)
rounded_predictions = np.where(predictions > 0.5,1,0)
rounded_predictions[0]

accuracy = (rounded_predictions[0] == y_test).mean()
accuracy
```

[10]: 0.824

# Booklet_2_Code

August 8, 2020

[ ]:
```
import numpy as np
import pandas as pd
import time

import tensorflow as tf
from tensorflow import keras

import category_encoders as ce

from sklearn.model_selection import RandomizedSearchCV
```

## 1 Booklet 2 Neuronale Netze Aufgabe 2

### 1.0.1 Using RandomSearch to find the perfect Hyperparameter combination

[ ]:
```
# Erstelle Modell innerhalb einer Funktion, um es für Grid Search nutzen zu␣
 ↪können
def build_model(n_hidden = 1, n_neurons=30, learning_rate=0.1,␣
 ↪activation_function='relu', dropout_prop=0.25):
    model = keras.models.Sequential()
    model.add(keras.layers.Dense(2, activation='relu'))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation=activation_function))
    model.add(keras.layers.Dense(1))
    model.compile(optimizer=tf.keras.optimizers.
 ↪SGD(learning_rate=learning_rate),
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    model.add(keras.layers.Dropout(dropout_prop))
    return model

nn = keras.wrappers.scikit_learn.KerasClassifier(build_model)
```

[ ]:
```
# zu untersuchende Parameter mit jeweiligem Parameterraum
params = {
    'n_hidden': [0,1,2,3,4],
    'n_neurons': [1,3,5,10,20,50,100],
```

```
    'learning_rate': [0.1, 0.05, 0.01],
    'activation_function': ['relu', 'sigmoid', 'elu'],
    'dropout_prop': [0, 0.25, 0.5]
}
```

[ ]:
```
random_search = RandomizedSearchCV(nn, params, n_iter=20)
random_search.fit(X_train.values,
                  y_train.values,
                  validation_data=(X_test.values, y_test.values),
                  callbacks=[keras.callbacks.EarlyStopping(patience=6)],
                  batch_size=32,
                  epochs=100
                 )
```

[ ]:
```
random_search.best_params_
```

### 1.1 Building the winner Model

[ ]:
```
#dont need this anymore...
train_dataset = tf.data.Dataset.from_tensor_slices((X_train.values, y_train.
 ↪values))
train_dataset = train_dataset.shuffle(len(X_train)).batch(32)

test_dataset = tf.data.Dataset.from_tensor_slices((X_test.values, y_test.
 ↪values))
test_dataset = test_dataset.shuffle(len(X_test)).batch(len(X_test))
```

[ ]:
```
nn = keras.models.Sequential()
nn.add(keras.layers.Dense(2, activation='elu'))
for layer in range(4):
        nn.add(keras.layers.Dense(3, activation='elu'))
nn.add(keras.layers.Dense(1))

nn.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
           loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
           metrics=['accuracy'])
```

[ ]:
```
nn.fit(train_dataset,
       validation_steps=1,
       validation_data=test_dataset,
       batch_size=32,
       callbacks=[keras.callbacks.EarlyStopping(patience=6)],
       epochs=50)
```

## A.5 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.6 Quellcode zu Booklet Teil 3

# Booklet3Code

August 8, 2020

## 1 Booklet 3 Ensemblemethoden

[ ]:
```
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,␣
↪AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
import matplotlib
```

### 1.1 Univariat Grid Search

[ ]:
```
params_bagging = [{'n_estimators': [10, 50, 100]},
                  {"base_estimator__criterion": ["gini", "entropy"]},
                  {"base_estimator__max_depth": [None, 3,5,10, 15, 20, 25,␣
↪50]},
                  {"base_estimator__min_samples_split":␣
↪[2,5,10,20,30,40]},#The minimum number of samples required to split an␣
↪internal node:
                  {"base_estimator__min_samples_leaf": [1,2,5,10,20,40]},#The␣
↪minimum number of samples required to be at a leaf node
                  {"base_estimator__min_weight_fraction_leaf": [0, 0.2, 0.4, 0.
↪5]},#The minimum weighted fraction of the sum total of weights (of all the␣
↪input samples) required to be at a leaf node.
                  {"base_estimator__max_features": [5,10,15,20, 25, None]},
                  {"base_estimator__max_leaf_nodes": [None, 2␣
↪,10,100,150,200,300,500]},
                  {"base_estimator__min_impurity_decrease": [0.0, 0.1, 0.2, 0.
↪5, 0.8, 1, 2, 3]},
                  {"base_estimator__min_impurity_split": [0.0, 0.1, 0.2, 0.5,␣
↪0.8, 1, 2, 3]} #dericated option
                 ]

params_adaboost = [{'n_estimators': [10, 50, 100]},
                  {'learning_rate': [0.1, 0.5, 1]},
                  {"base_estimator__criterion": ["gini", "entropy"]},
                  #{"base_estimator__max_depth": [None, 3,5,10, 15, 20, 25,␣
↪50]},
```

```
                    {"base_estimator__min_samples_split":␣
↪[2,5,10,20,30,40]},#The minimum number of samples required to split an␣
↪internal node:
                    {"base_estimator__min_samples_leaf": [1,2,5,10,20,40]},#The␣
↪minimum number of samples required to be at a leaf node
                    {"base_estimator__min_weight_fraction_leaf": [0, 0.2, 0.4, 0.
↪5]},#The minimum weighted fraction of the sum total of weights (of all the␣
↪input samples) required to be at a leaf node.
                    {"base_estimator__max_features": [1,2,10,15,20, None]},
                    {"base_estimator__max_leaf_nodes":␣
↪[None,2,5,10,100,150,200]},
                    {"base_estimator__min_impurity_decrease": [0.0, 0.1, 0.2, 0.
↪5, 0.8, 1, 2, 3]},
                    {"base_estimator__min_impurity_split": [0.0, 0.1, 0.2, 0.5,␣
↪0.8, 1, 2, 3]} #dericated option
                   ]

params_random_forest = [{'n_estimators': [10, 50, 100, 200, 500]},
                    {"criterion": ["gini", "entropy"]},
                    {"max_depth": [None, 3,5,10, 15, 20, 25, 50]},
                    #{"min_samples_split": [2,5,10,20,30,40]},#The minimum␣
↪number of samples required to split an internal node:
                    {"min_samples_leaf": [1,2,5,10,20,40]},#The minimum number␣
↪of samples required to be at a leaf node
                    {"min_weight_fraction_leaf": [0, 0.2, 0.4, 0.5]},#The␣
↪minimum weighted fraction of the sum total of weights (of all the input␣
↪samples) required to be at a leaf node.
                    {"max_features": [5,10,15,20, 25, None]},
                    {"max_leaf_nodes": [None, 2 ,10,100,150,200,300,500]},
                    {"min_impurity_decrease": [0.0, 0.1, 0.2, 0.5, 0.8, 1, 2,␣
↪3]},
                    {"min_impurity_split": [0.0, 0.1, 0.2, 0.5, 0.8, 1, 2, 3]}␣
↪#dericated option
                   ]

params_tree = [{"criterion": ["gini", "entropy"]},
                    #{"max_depth": [None, 3,5,10, 15, 20, 25, 50]},
                    {"min_samples_split": [2,5,10,20,30,40]},#The minimum number␣
↪of samples required to split an internal node:
                    {"min_samples_leaf": [1,2,5,10,20,40]},#The minimum number␣
↪of samples required to be at a leaf node
                    {"min_weight_fraction_leaf": [0, 0.2, 0.4, 0.5]},#The␣
↪minimum weighted fraction of the sum total of weights (of all the input␣
↪samples) required to be at a leaf node.
                    {"max_features": [5,10,15,20, 25, None]},
                    {"max_leaf_nodes": [None, 2 ,10,100,150,200,300,500]},
```

```
                    {"min_impurity_decrease": [0.0, 0.1, 0.2, 0.5, 0.8, 1, 2,␣
↪3]},
                    {"min_impurity_split": [0.0, 0.1, 0.2, 0.5, 0.8, 1, 2, 3]}␣
↪#dericated option
                   ]
```

[ ]:
```
#looping through grid ourselfes for better interpretation of outputs.
for param in params_random_forest:

    bagging_classifier = BaggingClassifier(
        base_estimator=DecisionTreeClassifier(),
        bootstrap=True, #replace training samples
        n_jobs=-1, #use all available cores
        random_state=42,
  #      n_estimators=100
    )

    bagging_gs = GridSearchCV(bagging_classifier, param, cv=10)
    bagging_gs.fit(X_train, y_train)
    print(param)
    print(bagging_gs.best_params_)
    print(bagging_gs.best_score_)
    print(bagging_gs.cv_results_['mean_test_score'])
    print(" ")
```

[ ]:
```
start_time = time.time()

for param in params_random_forest:

    adaboost_classifier = AdaBoostClassifier(
        base_estimator=DecisionTreeClassifier(
        max_depth=5,
        max_features=25,

            min_impurity_decrease=0.0005,
            min_samples_split= 10
        ),
        random_state=42,

    )

    adaboost_gs = GridSearchCV(adaboost_classifier, param, cv=10)
    adaboost_gs.fit(X_train, y_train)
    print(param)
    print(adaboost_gs.best_params_)
    print(adaboost_gs.best_score_)
    print(adaboost_gs.cv_results_['mean_test_score'])
```

```
    print(" ")

elapsed_time = time.time() - start_time
print(elapsed_time)
```

[ ]:
```
# multivariat parameter space
params_tree = [{"max_depth": [None,3,5,7],
     "min_samples_leaf": [50, 100, 200]
              }]

params_random_forest = [{
    "base_estimator__max_depth": [None,3, 5, 10],
    "base_estimator__min_samples_split": [5, 10],
    "base_estimator__max_features": [None, 5, 10, 25],
    "base_estimator__min_impurity_decrease": [0.0001, 0.0005]
              }]
```

[ ]:
```
start_time = time.time()

for param in params_random_forest:

    tree_classifier = DecisionTreeClassifier(
        random_state=42,
    )


    tree_gs = GridSearchCV(tree_classifier, params_random_forest, cv=10)
    tree_gs.fit(X_train, y_train)
    print(param)
    print(tree_gs.best_params_)
    print(tree_gs.best_score_)
    print(tree_gs.cv_results_['mean_test_score'])
    print(" ")

elapsed_time = time.time() - start_time
print(elapsed_time)
```

[ ]:
```
start_time = time.time()

for param in params_random_forest:

    random_forest_classifier = RandomForestClassifier(
        bootstrap=True, #replace training samples
        random_state=42,
        n_jobs=-1, #use all available cores,
        n_estimators=100,
    )
```

```
    random_forest_gs = GridSearchCV(random_forest_classifier, param, cv=10)
    random_forest_gs.fit(X_train, y_train)
    print(param)
    print(random_forest_gs.best_params_)
    print(random_forest_gs.best_score_)
    print(random_forest_gs.cv_results_['mean_test_score'])
    print(" ")

elapsed_time = time.time() - start_time
print(elapsed_time)
```

### 1.1.1 Optimized Bagging Classifier

[ ]:
```
# Ist das hier nicht im Prinzip das Gleiche wie ein Random Forest mit␣
 ↪splitter="random"?
start_time = time.time()

bagging_classifier = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(
     max_features=5,
        min_impurity_decrease=0.0001,
        min_samples_split=10
    ),
    bootstrap=True, #replace training samples
    n_jobs=-1, #use all available cores
    random_state=42,
    n_estimators=100
)
bagging_classifier.fit(X_train, y_train)

test_predictions = bagging_classifier.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)

elapsed_time = time.time() - start_time
print(elapsed_time)
```

### 1.1.2 Optimized AdaBoost Classifier

[ ]:
```
start_time = time.time()
adaboost_classifier = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(
        max_depth=20,
        #max_features=25,
```

```
        #   min_impurity_decrease=0.0005,
         #  min_samples_split= 10
        ),
    n_estimators=100,
    random_state=42
)

adaboost_classifier.fit(X_train, y_train)

test_predictions = adaboost_classifier.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)

elapsed_time = time.time() - start_time
print(elapsed_time)
```

### 1.1.3 Optimized Random Forest

[ ]:
```
random_forest_classifier = RandomForestClassifier(
    bootstrap=True, #replace training samples
    random_state=42,
    n_jobs=-1, #use all available cores,
    min_impurity_decrease=0.00005,
    min_samples_split=5,
    n_estimators=100
)

random_forest_classifier.fit(X_train, y_train)

test_predictions = random_forest_classifier.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)

# Feature importances with random forest
for name, score in zip(X_train.columns, random_forest_classifier.
 ↪feature_importances_):
    print(name, score)
```

### 1.1.4 Optimized Tree

[ ]:
```
tree_classifier = DecisionTreeClassifier(
    random_state=42,
    min_impurity_decrease=0.0002,
    min_samples_split=6
    )
```

```
tree_classifier.fit(X_train, y_train)

test_predictions = tree_classifier.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.1.5 Visualisation of GridSearch results

[ ]:
```
parameter = {"max_depth": [3,5,10, 15, 20, 25, 50,100]}
#parameter = {"min_impurity_decrease": [0.0001, 0.001, 0.003, 0.005, 0.01, 0.
 ↪05, 0.1, 0.2]}
parameter_ = {"base_estimator__max_depth": [3,5,10, 15, 20, 25, 50,100]}

## Tree
tree_classifier = DecisionTreeClassifier(
    random_state=42,
)

tree_gs = GridSearchCV(tree_classifier, parameter, cv=10,␣
 ↪return_train_score=True)
tree_gs.fit(X_train, y_train)

results_tree = tree_gs.cv_results_
test_scores_tree = results_tree['mean_test_score']
train_scores_tree = results_tree['mean_train_score']


random_forest_classifier = RandomForestClassifier(
    bootstrap=True, #replace training samples
    random_state=42,
    n_jobs=-1, #use all available cores,
)

## Forest
random_forest_gs = GridSearchCV(random_forest_classifier, parameter, cv=10,␣
 ↪return_train_score=True)
random_forest_gs.fit(X_train, y_train)

results_forest = random_forest_gs.cv_results_
test_scores_forest = results_forest['mean_test_score']
train_scores_forest = results_forest['mean_train_score']

## bagging
bagging_classifier = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    bootstrap=True, #replace training samples
```

```
    n_jobs=-1, #use all available cores
    random_state=42
)

bagging_gs = GridSearchCV(bagging_classifier, parameter_, cv=10,␣
 ↪return_train_score=True)
bagging_gs.fit(X_train, y_train)

results_bagging = bagging_gs.cv_results_
test_scores_bagging = results_bagging['mean_test_score']
train_scores_bagging = results_bagging['mean_train_score']

## Adaboost
adaboost_classifier = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(),
    random_state=42
)

adaboost_gs = GridSearchCV(adaboost_classifier, parameter_, cv=10,␣
 ↪return_train_score=True)
adaboost_gs.fit(X_train, y_train)

results_adaboost = adaboost_gs.cv_results_
test_scores_adaboost = results_adaboost['mean_test_score']
train_scores_adaboost = results_adaboost['mean_train_score']


X_axis = np.array(results_tree['param_max_depth'].data, dtype=float)
```

[ ]:
```
results_tree
```

[ ]:
```
best_index_tree = np.nonzero(results_tree['rank_test_score'] == 1)[0][0]
best_score_tree = results_tree['mean_test_score'][best_index_tree]

best_index_adaboost = np.nonzero(results_adaboost['rank_test_score'] == 1)[0][0]
best_score_adaboost = results_adaboost['mean_test_score'][best_index_adaboost]

best_index_forest = np.nonzero(results_forest['rank_test_score'] == 1)[0][0]
best_score_forest = results_forest['mean_test_score'][best_index_forest]

best_index_bagging = np.nonzero(results_bagging['rank_test_score'] == 1)[0][0]
best_score_bagging = results_bagging['mean_test_score'][best_index_bagging]
```

[ ]:
```
matplotlib.rcParams['figure.figsize'] = (10.0, 8.0)
fig1, ax1 = plt.subplots()
```

```
ax1.plot(X_axis, train_scores_tree, color='blue', alpha=0.7, linestyle='dashed'␣
 ↪)
ax1.plot(X_axis, test_scores_tree, color='blue')

ax1.plot(X_axis, train_scores_forest, color = 'green', alpha=0.7,␣
 ↪linestyle='dashed')
ax1.plot(X_axis, test_scores_forest, color = 'green')

ax1.plot(X_axis, train_scores_adaboost, color = 'red', alpha=0.7,␣
 ↪linestyle='dashed')
ax1.plot(X_axis, test_scores_adaboost, color = 'red')

ax1.plot(X_axis, train_scores_bagging, color = 'purple', alpha=0.7,␣
 ↪linestyle='dashed')
ax1.plot(X_axis, test_scores_bagging, color = 'purple')

ax1.set_xscale('log')
ax1.set_xticks([3,5,10,20,50,100])
ax1.get_xaxis().set_major_formatter(matplotlib.ticker.ScalarFormatter())
ax1.set_xlabel("max_depth", fontsize=14)
ax1.set_ylabel('Accuracy', fontsize=14)
plt.legend(['train tree','test tree', 'train forest', 'test forest','train␣
 ↪adaboost', 'test adaboost', 'train bagging', 'test bagging'], fontsize=11)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

ax1.plot([X_axis[best_index_tree], ] * 2, [0, best_score_tree], linewidth=1,
        linestyle='-.', color='blue', marker='x', markeredgewidth=3, ms=8)
ax1.annotate("%0.3f" % best_score_tree,
            (X_axis[best_index_tree], best_score_tree + 0.005),␣
 ↪backgroundcolor="w", fontsize=12)

ax1.plot([X_axis[best_index_adaboost], ] * 2, [0, best_score_adaboost],␣
 ↪linewidth=1,
        linestyle='-.', color='red', marker='x', markeredgewidth=3, ms=8)
ax1.annotate("%0.3f" % best_score_adaboost,
            (X_axis[best_index_adaboost], best_score_adaboost + 0.005),␣
 ↪backgroundcolor="w", fontsize=12)

ax1.plot([X_axis[best_index_forest], ] * 2, [0, best_score_forest], linewidth=1,
        linestyle='-.', color='green', marker='x', markeredgewidth=3, ms=8)
ax1.annotate("%0.3f" % best_score_forest,
            (X_axis[best_index_forest], best_score_forest + 0.005),␣
 ↪backgroundcolor="w", fontsize=12)
```

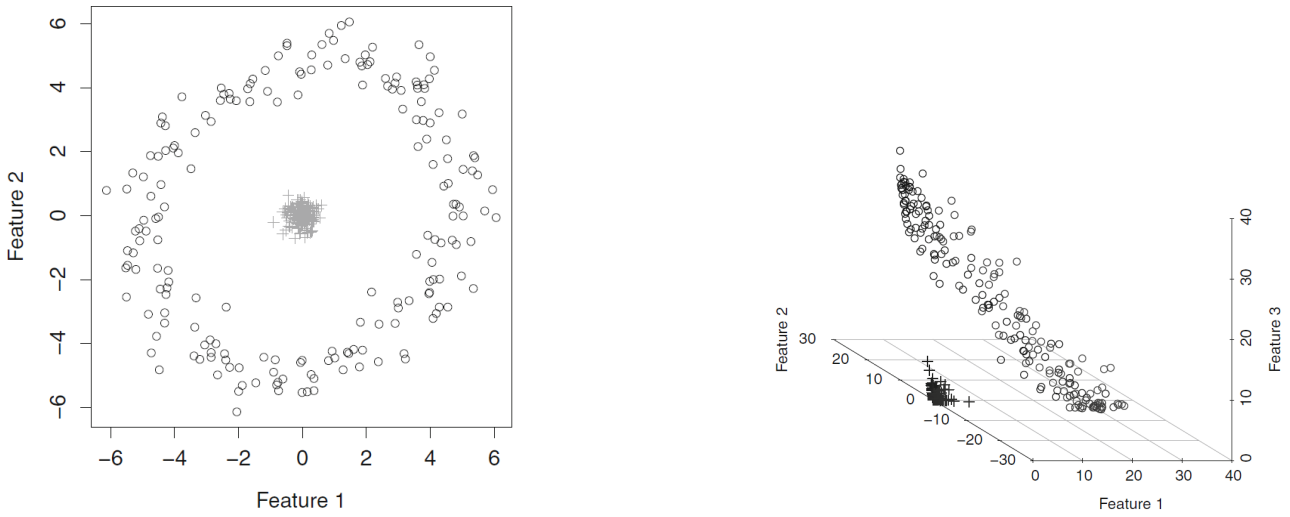
```
ax1.plot([X_axis[best_index_bagging], ] * 2, [0, best_score_bagging],␣
 ↪linewidth=1,
        linestyle='-.', color='purple', marker='x', markeredgewidth=3, ms=8)
ax1.annotate("%0.3f" % best_score_bagging,
             (X_axis[best_index_bagging], best_score_bagging + 0.005),␣
 ↪backgroundcolor="w", fontsize=12)

ax1.set_ylim(0.75,1)

plt.savefig('grid_search_max_depth.png')
```

## A.7 Ergänzungen zu Booklet Teil 4

Abbildung 11a zeigt einen zweidimensionalen *Input Space* dessen Daten durch Transformation in einem dreidimensionalen *Feature Space* linear trennbar sind (s. Abbildung 11b)
Tabelle 11 zeigt die genauen Parameterräume, die wurch das *Grid Search* Verfahren untersucht




(a) zwei-dimensionaler Input-Space

(b) drei-dimensionaler Feature-Space

Abbildung 11: Transformation der Daten durch Kernfuntionen (Mello and Ponti (2018))

wurden. Zudem sieht man für jeden Kernel die besten Parameter und die Genauigkeit des trainierten Modells. Der Kerneltyp *precomputed* wurde hierbei nicht beachtet.

| Kernel | Parameterräume | Parameterergebnisse | Accuracy |
|---|---|---|---|
| Linear | C: [0.1, 1.0, 10.0, 100.0] | C: 1.0 | 0.85047 |
| Polynomial | C: [0.1, 1.0, 10.0, 100.0] | C: 1.0 | 0.86153 |
| | coef0: [0, 0.5, 1.0] | coef0: 1.0 | |
| | degree: [2.0, 3.0, 4.0, 5.0] | degree: 4.0 | |
| | gamma: ['scale', 'auto', 0.01, 1.0] | gamma: 'scale' | |
| RBF | C: [0.1, 1.0, 10.0, 100.0] | C: 10.0 | 0.86198 |
| | gamma: ['scale', 'auto', 0.01, 1.0] | gamma: 'scale' | |
| Sigmoid | C: [0.1, 1.0, 10.0, 100.0] | C: 10.0 | 0.85028 |
| | gamma: ['scale', 'auto', 0.01, 1.0] | gamma: 0.01 | |
| | coef0: [0.0, 0.5, 1.0] | coef0: 0.0 | |

Tabelle 11: Parameterräume und Ergebnisse der Grid Search

## A.8 Quellcode zu Booklet Teil 4

# Booklet4Code

August 8, 2020

[1]:
```
import numpy as np
import pandas as pd
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV

from sklearn.svm import SVC
```

## 1 SVM Booklet Teil 4

### 1.0.1 linear Kernel

$\langle x, x^{'} \rangle$

[ ]:
```
svm_linear = SVC(kernel="linear", C=0.1)
svm_linear.fit(X_train, y_train)

test_predictions = svm_linear.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.0.2 Polynominal Kernel

$(\gamma \langle x, x^{'} \rangle + r)^d$

$d$ : degree

$r$: coef0

[ ]:
```
svm_poly = SVC(kernel="poly", gamma=0.1, C=0.1)
svm_poly.fit(X_train, y_train)

test_predictions = svm_poly.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.0.3 Gaussian RBF Kernel

$exp(-\gamma ||x - x^{'}||^2)$

$\gamma$: gamma (>0)

$\gamma$ definiert den Einfluss der einzelenen Trainingsdaten Je größer $\gamma$ ist, desto näher müssen andere Trainingsdatenpunkte sein, um einen Effekt zu haben $\rightarrow$ $\gamma$ gibt invertiert den Einfluss-Radius der Datenpunkte an, die als Support Vectors bestimmt wurden (vom Modell).

[ ]:
```
svm_rbf = SVC(kernel="rbf", C=0.01)
svm_rbf.fit(X_train, y_train)

test_predictions = svm_rbf.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.0.4 Sigmoid Kernel

$tanh(\gamma\langle x, x^{'}\rangle) + r)$

$r$: coef0

[ ]:
```
svm_sigmoid = SVC(kernel="sigmoid", C=0.01)
svm_sigmoid.fit(X_train, y_train)

test_predictions = svm_sigmoid.predict(X_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

### 1.0.5 precomputed Kernel

[ ]:
```
gram_train = np.dot(X_train, X_train.T)
gram_test = np.dot(X_test, X_train.T)

svm_precomputed = SVC(kernel="precomputed", C=0.01)

svm_precomputed.fit(gram_train, y_train)

test_predictions = svm_precomputed.predict(gram_test).round().astype(int)
print(accuracy_score(y_test, test_predictions))
mean_absolute_error(y_test, test_predictions)
```

Um ein 'gute' Modell zu trainieren, werden die Hyperparameter mit GridSearch bestimmt. Diese Parameterräume werden untersucht.

[ ]:
```
#Tamara
param_linear = {'C': [0.1, 1.0, 10.0, 100.0]}
param_poly = {'C': [0.1, 1.0, 10.0, 100.0],
              'gamma': ['scale', 'auto', 0.01, 1.0],
              'degree': [2.0, 3.0, 4.0, 5.0],
              'coef0': [0, 0.5, 1.0]}
param_rbf = {'C': [0.1, 1.0, 10.0, 100.0],
```

```
              'gamma': ['scale', 'auto', 0.01, 1.0]}
param_sigmoid = {'C': [0.1, 1.0, 10.0, 100.0],
              'gamma': ['scale', 'auto', 0.01, 1.0],
              'coef0': [0.0, 0.5, 1.0]}
grid = GridSearchCV(SVC(kernel='poly', C=1), {'coef0': [0.0, 0.5, 1.0]}, refit␣
 ↪= True, verbose = 3)

# fitting the model for grid search
grid.fit(X_train, y_train)
# print best parameter after tuning
print(grid.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid.best_estimator_)
```

# 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.