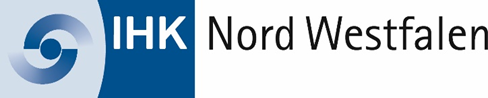
Projekte und Fallstudien der Wirtschaftsinformatik

Prüfungshausarbeit

Vorgelegt von: Maximilian Bär-Bietenbeck  
 Luca Fasselt

Studiengang: Duales Studium

Informatik-Betriebswirt/-in VWA /

Bachelor of Arts FR Wirtschaftsinformatik,

Gruppe 17

Themensteller: Prof. Dr. Priemer

Abgabedatum: 11.01.2026

Inhaltsverzeichnis

[Data Mining 2](#_Toc1815238315)

[Aufgabe a 3](#_Toc165872591)

[Aufgabe b 4](#_Toc560091002)

[Aufgabe c 5](#_Toc1759928952)

[Aufgabe d 14](#_Toc1937240881)

[Bestärkendes Lernen 18](#_Toc480908043)

[Beschreibung des Problems: 19](#_Toc915116842)

[Theoretische Grundlagen des Reinforcement Learning: 19](#_Toc458554968)

[Einsatz neuronaler Netze im Reinforcement Learning 20](#_Toc1616228651)

[Implementation eines Tic-Tac-Toe Netzes 21](#_Toc1375505393)

[Lernbeweis 24](#_Toc95938135)

[Probleme, Lösungen und Erweiterungen 25](#_Toc698343780)

## Data Mining

### Aufgabe a

1. **Analysieren Sie vor dem Durchführen des Data Minings die Trainingsdaten im Hinblick auf das Zielattribut „TARGET\_BETRUG“. Fällt Ihnen etwas auf, was das Data Mining eventuell erschweren könnte.**
2. **Massive Klassenunausgewogenheit: (Statistischen Ansatz)**

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Keine Betrugsbestellung: 28.254

Betrugsbestellung: 1.746

* Nur ca. 5,8% der Bestellungen sind betrügerisch
* Großteil davon sind Rechnungsbestellungen
* Bestellzeitpunkt ist immer ein ungültiges Datum 30.12.1899
* Daten lassen unmögliche Geburtsdaten zu
* Mögliche False Positives bei Betrugsfällen

Warum ist das ein Problem?

* Viele ML-Modelle „lernen“, immer nein vorherzusagen, weil sie damit trotzdem > 94% Accuracy erreichen würden
* Das Modell übersieht dadurch Betrugsfälle -> genau das, was wir vermeiden wollen.
* Das ML-Modell würde hauptsächlich Betrugsfälle erkennen, die über Rechnung getätigt wurden und als Bestelluhrzeit und ungültiges Datum haben -> Eigentlich nicht möglich
*  **Viele Merkmale mit extrem vielen Missing Values (NaN) – potenziell problematisch fürs Modelltraining:**  
  Besonders auffällig sind z. B. **ANUMMER\_02–ANUMMER\_10** (teils >90% fehlend; **ANUMMER\_10 sogar vollständig fehlend**) sowie **Z\_CARD\_ART (~62% fehlend)** und **Z\_LAST\_NAME (~49% fehlend)**.  
  Das kann Data Mining erschweren, weil du **Imputation/“Unknown”-Kategorien** brauchst oder Features ggf. **entfernen** musst – und weil Missingness hier vermutlich **nicht zufällig** ist (z. B. abhängig von Zahlungsart).
*  **Kodierung/Datentypen:**  
  Das Target ist **textuell (“ja/nein”)** und muss für viele Verfahren erst sauber **kodiert** werden (0/1). Außerdem sind viele Eingangsvariablen ebenfalls kategorisch („ja/nein“, Zahlungsarten etc.) und benötigen passende **Encoding-Strategien**.

1. Extrem viele Missing Values

Viele Merkmale mit extrem vielen Missing Values (NaN) – Potenziell problematisch fürs Modelltraining:

Besonders auffällig:

* ANUMMER\_02 – ANUMMER\_10 (teils > 90% fehlend)
* Z\_CARD\_ART (≈62% fehlend)
* Z\_LAST\_NAME (≈49% fehlend)
* Kann Data Mining erschweren, weil man „Unknown“-Kategorien braucht oder Features entfernen muss und weil Missingness hier nicht zufällig ist (z.B. abhängig von der Zahlungsart

1. Kodierung/Datentypen

* Target ist textuell („Ja/Nein“)
* Für viele Verfahren sauber Kodierung benötigt
* Viele Eingangsvariablen ebenfalls kategorisch („Ja/Nein“), Zahlungsarten etc.)
* Benötigen passende Encoding-Strategien

### Aufgabe b

1. **Schauen Sie sich bei den unabhängigen Attributen die Spalten ANUMMER\_01 bis ANUMMER\_10 an. Was verbirgt sich hinter diesen Spalten und warum ist die in der Tabelle gewählte Darstellung ungünstig? [Anmerkung: Sie können trotzdem mit dieser Darstellung weiter arbeiten. Es sei denn, Sie haben eine Idee, wie man die Daten umformen könnte und können diese auch praktisch umsetzen!]**
2. Variable Anzahl an Artikel

Die Spalten ANUMMER\_01 bis ANUMMER\_10 enthalten die Artikelnummern der jeweils bestellten Produkte

* Jede Spalte ist eine Bestellung
* Pro Bestellung können bis zu 10 Artikel auftauchen
* Die Artikelnummern stehen verteilt auf 10 Spalten, nicht in einzelnen Zeilen

Warum ist diese Darstellung ungünstig?

* Variable Anzahl an Artikeln -> „Wide Format”

Ein Besllung-dynamisch-in-10-SPalten-Format ist unüblich

Hier gilt:

Kunde A bestellt 1 Artikel -> 1 Spalte gefüllt

Kunde B bestellt 7 Artikel -> 7 Spalten gefüllt

* Die Anzahl der Artikel ist ein Inhaltsmerkmal, aber wird hier durch Spaltenstruktur versteckt

1. Artikelnummern sind IDs -> weder ordinal noch numerisch sinnvoll

Artikelnummern haben:

* Keinen numerischen Wert
* Keine Rangfolge
* Keine Statische Bedeutung

Aber das Modell würde sie zunächst als Zahlen interpretieren -> Bedeutungslos für ML

1. Redundanz

Von 10 möglichen Spalten sind oft:

* 80 – 90 % leer
* Gleiche Artikel tauchen in mehreren Spalten auf verschiedenen Zeilen auf
* Viele Spalten nehmen sehr selten Werte an

### Aufgabe c

1. **Führen Sie das Training mit drei unterschiedlichen und für die Problemstellung geeigneten Verfahren durch, dabei können Sie das Werkzeug selbst auswählen. Beim Training behalten Sie bitte 20% der Trainingsdaten für den Test zurück. Stellen Sie die Testergebnisse der drei Verfahren jeweils als Konfusionsmatrix dar und berechnen Sie Precision, Recall und Accuracy.**

**Nehmen Sie Stellung zu den erzielten Ergebnissen, ggf. auch im Hinblick auf die bei Teilaufgabe a) gemachte Beobachtung. Was können Sie hier eventuell zur Verbesserung des Ergebnisses tun? [Wenn es für Sie technisch möglich ist, versuchen Sie die entsprechende Maßnahme durchzuführen!]**

Das gewählte Werkzeug ist RapidMiner (bzw. jetzt AI Studio). Hierbei wurde zuerst die Möglichkeit des Auto Models ausprobiert, welches verschiedene Modelle erstellt. Aus den erstellten Modellen wurden drei ausgewählt.  
Am Beginn der Erstellung des Auto Models müssen Trainingsdaten erstellt werden. Die Spalte, die vorhergesagt werden soll, wird markiert:

Ein Bild, das Text, Screenshot, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Zahl, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Im nächsten Schritt kann geändert werden, welcher Wert, in diesem Fall „ja“ oder „nein“ bevorzugt werden soll In einem Test wurde erkannt, dass dieser Wert mit dem gewählten Daten keinen Unterschied bringt.

Ein Bild, das Text, Screenshot, Software, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

Da nicht alle Werte der bereitgestellten Daten von dem Auto Modell richtig verwendet werden können, sortiert RapidMiner die Spalten in drei Bereiche. Die Farben rot, gelb und grün stehen dafür, wie geeignet die Spalten sind, in dem Modell verarbeitet zu werden. Wenn die Werte in der Spalte sehr gestreut sind, keine Abhängigkeit zu dem vorherzusehenden Wert besitzen oder viele Nullwerte beinhaltet, sollte diese Spalte herausgenommen werden und ist rot markiert.

Die zu verarbeiteten Daten führen dazu, dass das Endergebnis immer sehr ähnlich ist, wobei die unterschiedlichsten Kombinationen benutzt werden.

Ein Bild, das Text, Screenshot, Zahl, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Der letzte Schritt bei der Erstellung des Auto Modells ist die Auswahl der zu erstellten Modelle. Es wurden alle Modelle ausgewählt.

Ein Bild, das Text, Screenshot, Zahl, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Die Ergebnisse der Modelle laden in einem Dashboard. Alle erstellten Modelle können genauer dargestellt werden. Bis alle Modelle erstellt wurden, ist einige Zeit vergangen.

So sieht das Endergebnis aus. RapidMiner bestimmt auch, welches Modell besonders schnell oder genau ist.

Ein Bild, das Text, Software, Zahl, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Auch gibt der RapidMiner eine ROC-Kurve mit allen Modellen aus.

Ein Bild, das Text, Software, Zahl, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Da die Endergebnisse alle sehr ähnlich sind, wurden hier drei Modelle als Beispiel ausgewählt. Die gewählten Modelle sind „Naive Bayes“, „Fast Large Margin“ und „Decision Tree“. Für jedes Modell sind die Endwerte ähnlich.

Ein Bild, das Text, Software, Screenshot, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Zahl, Screenshot, Quittung enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Rechteck, Quadrat enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Quittung enthält.

KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Reihe, Zahl enthält.

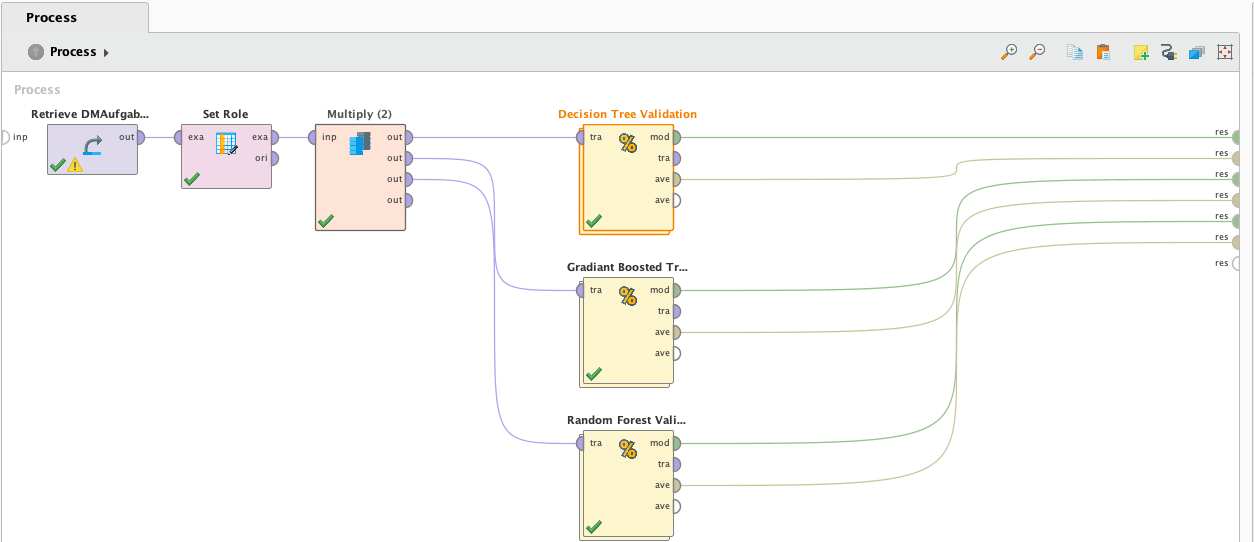
KI-generierte Inhalte können fehlerhaft sein.

Ein Bild, das Text, Screenshot, Quittung enthält.

KI-generierte Inhalte können fehlerhaft sein.

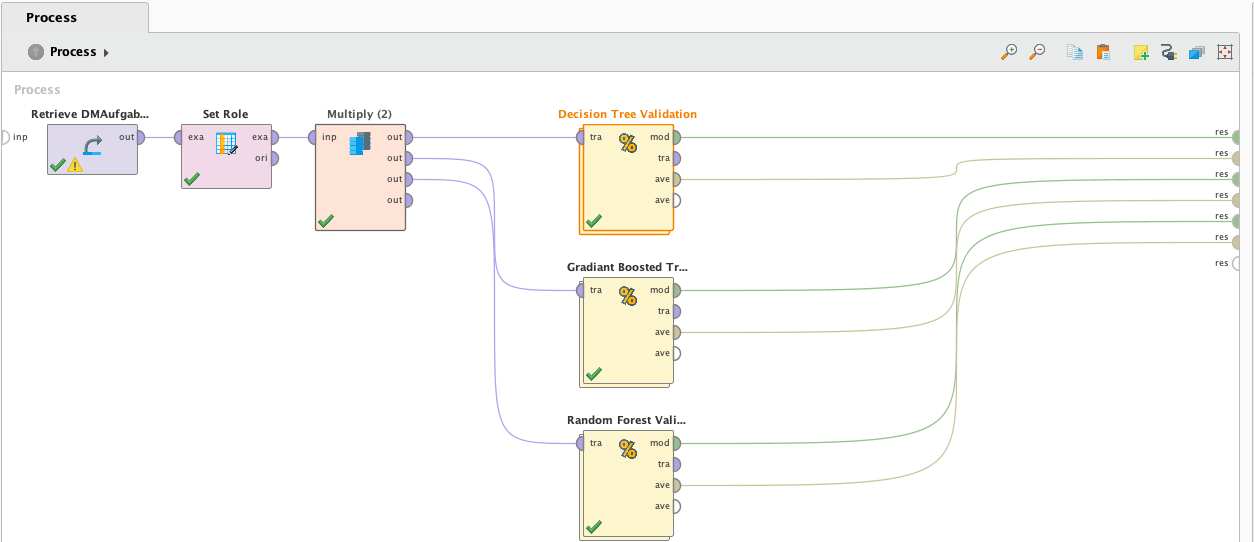
Da dieses Verfahren nicht das gewünschte Ergebnis geliefert hat, wurde das Manuelle Verfahren durchgeführt. Hier wurden die Daten so aufgeteilt, dass 80% der Daten fürs Training und 20% für den Test genutzt werden.

Der Prozess wurde mit folgenden Operatoren aufgebaut.

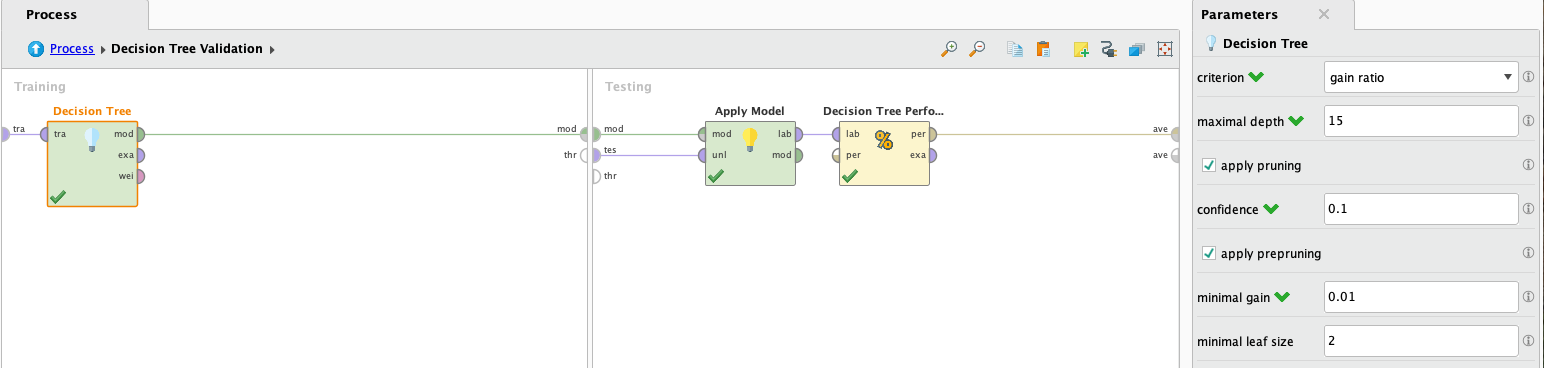


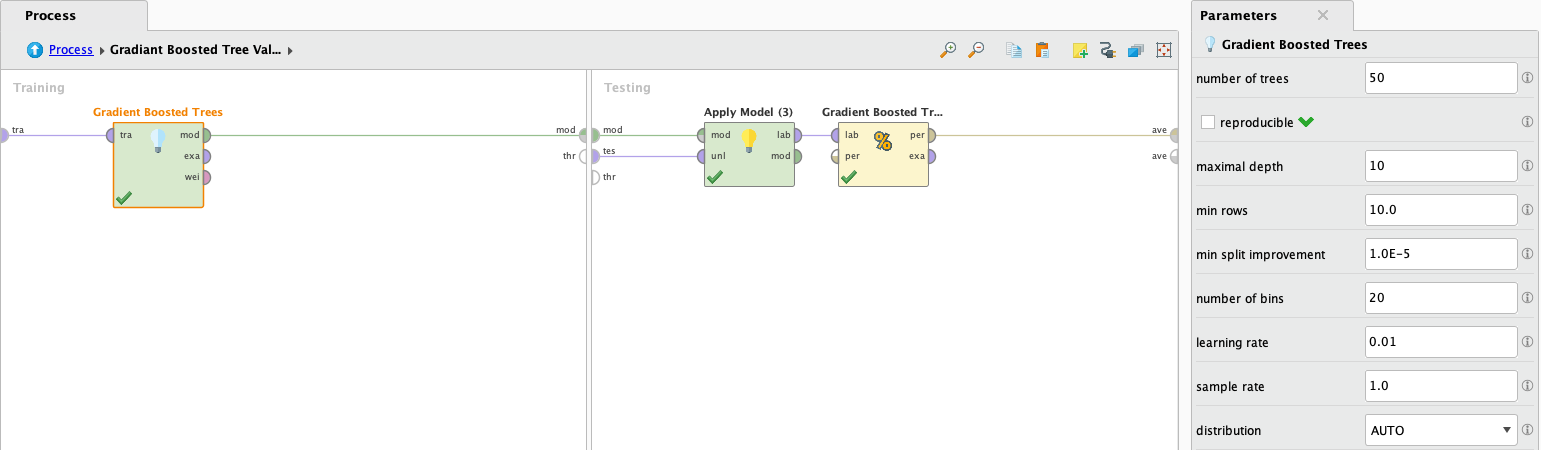
Ein Bild, das Text, Screenshot, Schrift, Software enthält.

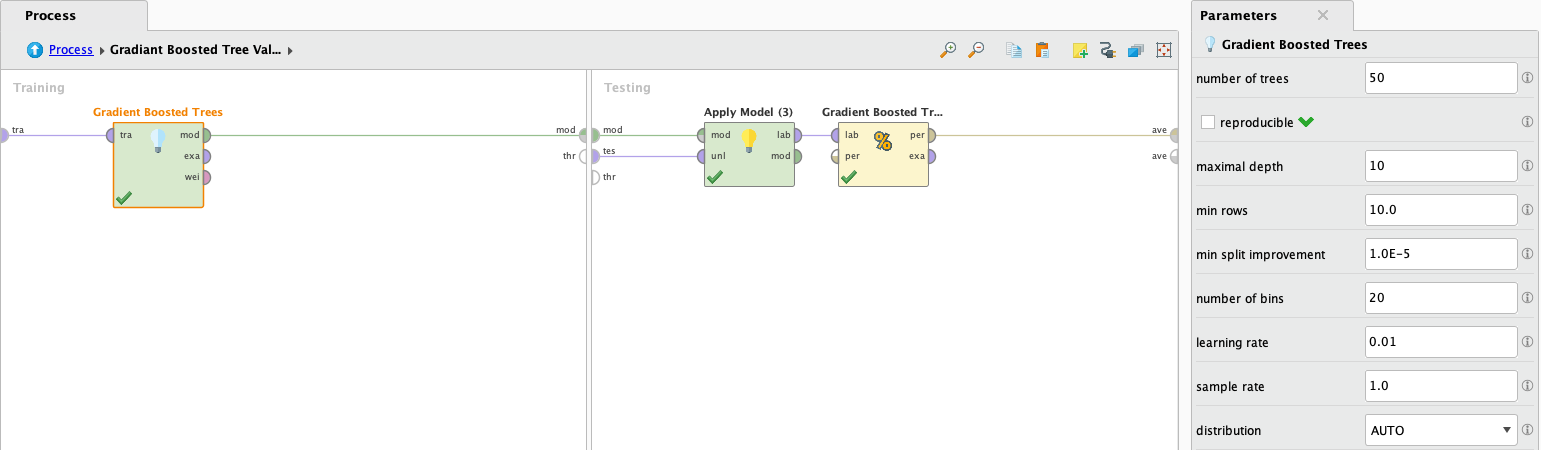
KI-generierte Inhalte können fehlerhaft sein.



Die Split Validations haben die folgenden Konfigurationen. Es wurden die Möglichkeiten des Decision Trees, des Random Forests und des Gradient Boosted Trees benutzt.







Die 20% Testdaten wurden so ausgewertet und das Modell hat gelernt, eigentlich immer nein zu sagen. Es verpasst also so gut wie alle Betrugsfälle. Also wurde das ganze mit Balancing angepasst. Dazu haben wir eine Filter Option eingebaut, die nur 20% der nicht Betrugsfälle nimmt und alle Ja Betrugsfälle. Dadurch wird dem Modell beigebracht nicht immer Nein zu sagen.   
Anpassung:

Ein Bild, das Screenshot, Text, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

Nachteil: Es werden viele zu testende Daten einfach weggeschmissen und nicht genutzt. Dadurch trainieren wir auf künstlich veränderte Klassenverteilungen.

### Aufgabe d

1. **Suchen Sie sich das Verfahren aus, das nach Ihrer Meinung am besten geeignet ist. Nutzen Sie dieses Verfahren für die Vorhersage von TARGET\_BETRUG für die neuen Bestellungen (Tabelle Klassifizierungsdaten). Wenn Ihr Werkzeug die Verbesserungsmaßnahme aus c) unterstützt, so nutzen Sie diese Verbesserungsmöglichkeit bitte.**

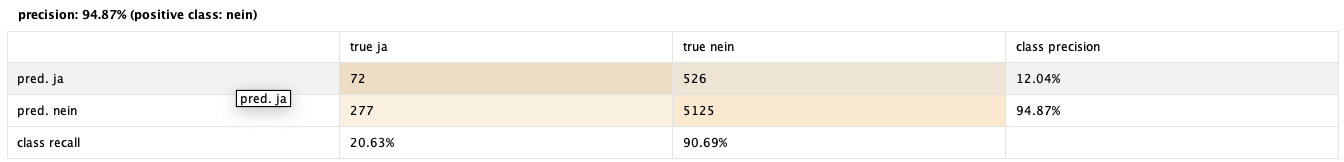
Bei dem manuellen Versuch wurden drei Methoden benutzt, der Decision Tree, der Random Forest und der Gradient Boosted Trees.

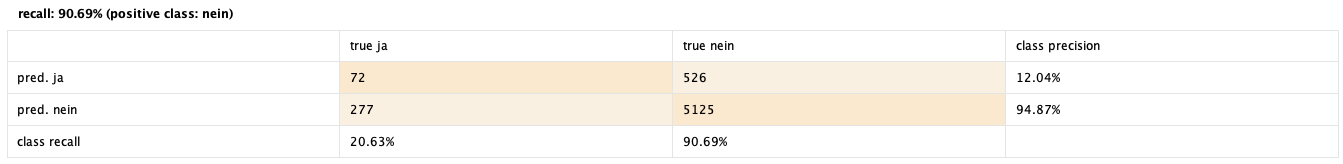
**Ergebnisse nach Anpassung:**

Das Ergebnis des Gradient Boosted Trees:

Ein Bild, das Text, Screenshot, Reihe, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.





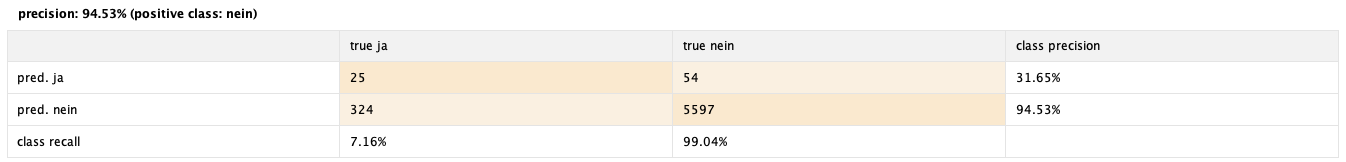
Ein Bild, das Text, Screenshot, Schrift enthält.

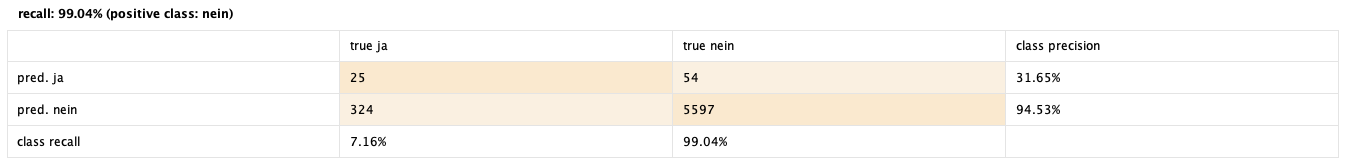
KI-generierte Inhalte können fehlerhaft sein.

Das Ergebnis des Random Forests:

Ein Bild, das Text, Screenshot, Reihe, Quittung enthält.

KI-generierte Inhalte können fehlerhaft sein.



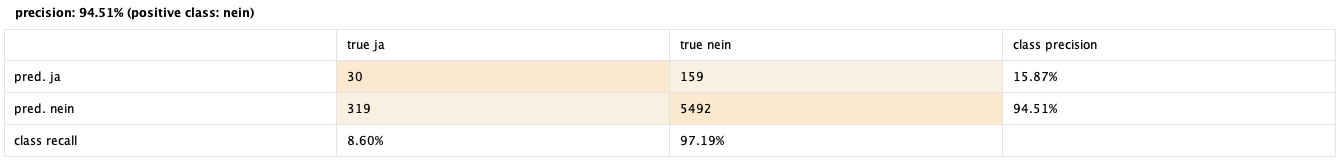


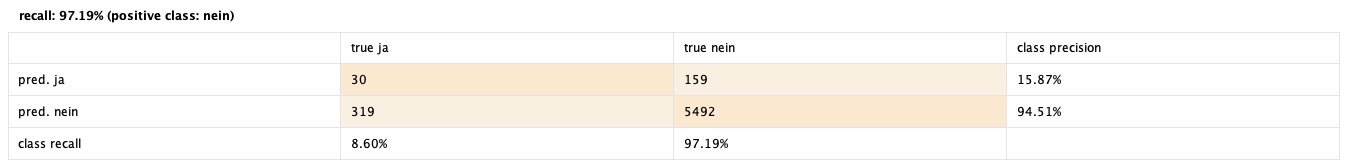
Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Das Ergebnis des Decision Trees:







Ein Bild, das Text, Screenshot, Schrift, Karte Menü enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Ergebnis:**

Der Oberen Erklärung kann man entnehmen, dass Gradient Boosted Tree am besten geeignet ist, um Betrugsfälle zu identifizieren

**Ohne Balancing:**

Ein Bild, das Text, Reihe, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Mit Balancing:

Ein Bild, das Text, Reihe, Schrift, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Es wurden 2620 Bestellungen als Betrügerisch eingeordnet und 17360 wurden von dem Modell als in Ordnung eingeordnet.

Auffällig ist, dass sich das Modell in den Ergebnissen nicht ganz „sicher“ ist. Das Modell hat im Schnitt einen Confidence (Ja) Score von 0,239. Selbst bei den „Eindeutigsten“ Betrügerischen Indizien eines Betrugs hat das Modell einen Confidence (Ja) Score von 0,554. Heißt das Modell erkennt Merkmale, ist jedoch in den meisten Fällen zu unsicher und tendiert dann zu keinen Betrug. Im Schnitt hat es einen Confidence (Nein) Socre von 0,761.

Wichtig für die Auswertung und das Ergebnis ist, dass wir uns für das Modell entschieden haben, welches am ehesten dazu tendiert einen Betrug vorherzusagen. Das führt dazu, dass die Accuracy, Recall Nein und Precision Werte niedriger sind, als bei den anderen Modellen. Im Vergleich sind jedoch die Recall Ja Werte mehr als Doppelt so hoch.

## Bestärkendes Lernen

### Beschreibung des Problems:

Das betrachtete Problem ist die Entwicklung eines selbstlernenden Programms für das Spiel Tic-Tac-Toe mithilfe von Reinforcement Learning. Tic-Tac-Toe ist ein deterministisches, rundenbasiertes Zwei-Personen-Nullsummenspiel mit vollständiger Information. Zwei Spieler setzen abwechselnd ihre Symbole („X“ und „O“) auf ein 3x3-Spielfeld. Ziel ist es, drei eigene Symbole in einer Reihe (horizontal, vertikal oder diagonal) zu platzieren. Das Spiel endet entweder mit einem Sieg eines Spielers oder mit einem Unentschieden, wenn alle Felder belegt sind, ohne dass eine Gewinnbedingung erfüllt wurde.

Die Herausforderung besteht darin, einen Spieler entwickeln, der ohne explizit vorgegebene Strategien oder Spielbäume lernt, möglichst erfolgreich zu spielen. Das Programm kennt zu Beginn ausschließlich die erlaubten Züge (Setzen auf freie Felder) sowie die Spielendbindungen und deren Bewertung (Sieg, Niederlage, Unentschieden). Eine klassische, regelbasierte Lösung wie das Minimax-Verfahren ist ausdrücklich ausgeschlossen, sodass der Spieler seine Strategie ausschließlich durch Erfahrung entwickeln muss.

Ein zentrales Merkmal des Problems ist die Anzahl möglicher Spielsituationen. Jedes der neun Felder kann einen von drei Zuständen annehmen: leer, mit „X“ belegt oder mit „O“ belegt. Daraus ergibt sich theoretisch eine obere Schranke von 3^9=19.683 möglichen Spielfeldkonfigurationen. Nicht alle diese Zustände sind jedoch tatsächlich erreichbar, da beispielsweise die Anzahl der gesetzten „X“- und „O“-Steine immer um höchstens eins differieren darf und viele Konfigurationen erst nach Spielende entstehen würden. Die reale Anzahl gültiger Spielsituationen liegt daher deutlich darunter, wird aber dennoch als ausreichend groß angesehen, um ein Lernen durch Ausprobieren sinnvoll zu machen.

Ziel des Systems ist es, eine Strategie zu erlernen, die langfristig zu möglichst vielen Siegen beziehungsweise mindestens zu Unentschieden führt. Der Spieler soll sowohl als „X“ als auch als „O“ agieren können und durch wiederholtes Spielen – etwa gegen sich selbst oder gegen einen festen Gegner – seine Entscheidungen schrittweise verbessern.

### Theoretische Grundlagen des Reinforcement Learning:

Reinforcement Learning (bestärkendes Lernen) ist ein Teilgebiet des maschinellen Lernens, bei dem ein Agent durch Interaktion mit einer Umgebung lernt, optimale Entscheidungen zu treffen. Im Gegensatz zum überwachten Lernen existieren keine vorab gelabelten Trainingsdaten. Stattdessen erhält der Agent nach jeder Aktion oder nach Abschluss einer Handlung eine Rückmeldung in Form einer Belohnung oder Bestrafung.

Ein Reinforcement-Learning-Problem besteht typischerweise aus den folgenden Komponenten:

Der Agent ist das lernende System, hier der Tic-Tac-Toe-Spieler. Die Umgebung ist das Spiel selbst, einschließlich Spielfeld und Gegner. Ein Zustand beschreibt die aktuelle Spielsituation, also die Belegung des Spielfelds. Eine Aktion entspricht einem möglichen Spielzug. Nach jeder Aktion erhält der Agent eine Belohnung, beispielsweise +1 für einen Sieg, −1 für eine Niederlage und 0 für ein Unentschieden. Ziel des Agenten ist es, eine Strategie (Policy) zu lernen, die den erwarteten kumulierten Belohnungswert maximiert.

Ein zentrales Spannungsfeld im Reinforcement Learning ist der sogenannte Exploration-Exploitation-Trade-off. Der Agent muss einerseits neue Aktionen ausprobieren (Exploration), um bessere Strategien zu entdecken, andererseits bekannte gute Aktionen bevorzugen (Exploitation), um erfolgreich zu spielen. Besonders zu Beginn des Lernprozesses ist Exploration entscheidend, da der Agent noch keine Kenntnisse über gute oder schlechte Züge besitzt.

Im Kontext von Tic-Tac-Toe eignet sich Reinforcement Learning besonders gut, da das Spiel klar definierte Zustände, Aktionen und Endbewertungen besitzt. Der Lernprozess kann durch viele automatisch durchgeführte Spiele realisiert werden, ohne dass menschliche Eingriffe notwendig sind. Durch Selbstspiel kann der Agent sowohl offensive als auch defensive Strategien entwickeln.

### Einsatz neuronaler Netze im Reinforcement Learning

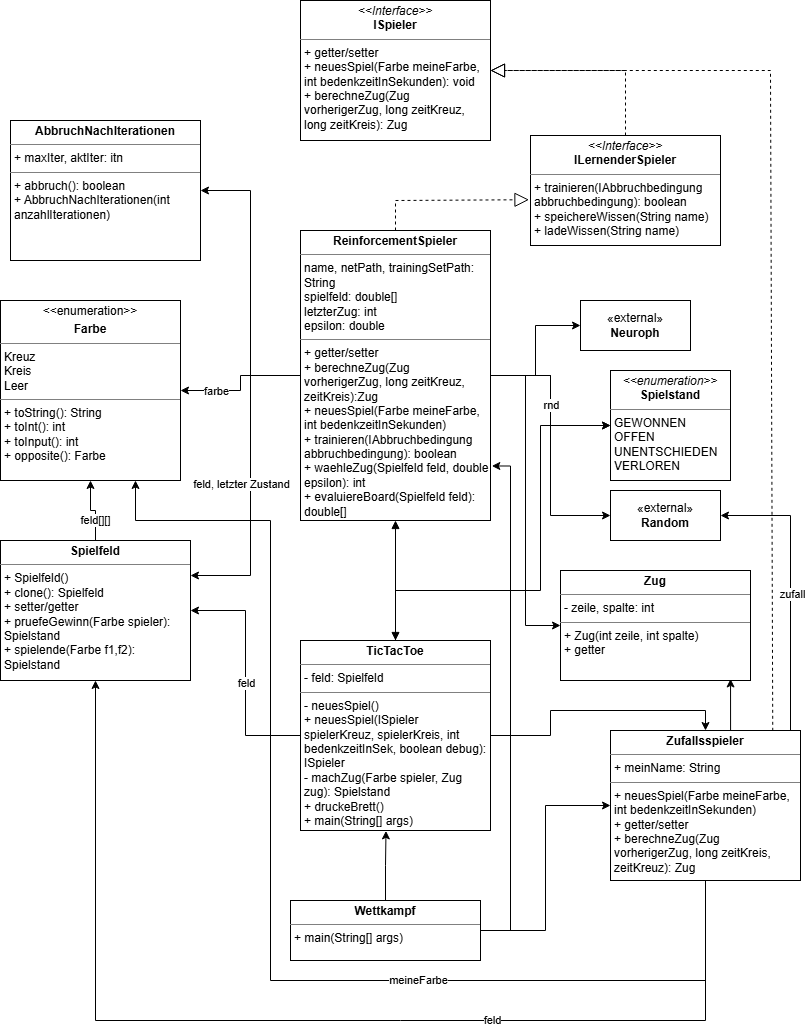
Neuronale Netze werden im Reinforcement Learning eingesetzt, um komplexe Zusammenhänge zwischen Zuständen und Aktionen zu approximieren. Anstatt für jede Spielsituation explizite Werte zu speichern (z. B. in einer Tabelle), kann ein neuronales Netz lernen, den Wert eines Zustands oder die Qualität einer Aktion zu schätzen. Dies ist insbesondere dann sinnvoll, wenn die Anzahl möglicher Zustände groß ist oder eine Generalisierung auf unbekannte Situationen erwünscht ist.

Im vorliegenden Problem kann ein neuronales Netz beispielsweise das Spielfeld als Eingabe erhalten und als Ausgabe eine Bewertung möglicher Züge oder eine Wahrscheinlichkeitsverteilung über Aktionen liefern. Dadurch ist der Spieler in der Lage, auch aus ähnlichen, aber noch nicht exakt gesehenen Spielsituationen sinnvolle Entscheidungen abzuleiten.

Der Einsatz neuronaler Netze ist im Zusammenhang mit Reinforcement Learning besonders dann sinnvoll, wenn eine vollständige Enumeration aller Zustände unpraktikabel ist oder wenn das System skalierbar auf komplexere Spiele erweitert werden soll. Obwohl Tic-Tac-Toe ein vergleichsweise einfaches Spiel ist, bietet es eine gute Grundlage, um die Prinzipien von Reinforcement Learning in Kombination mit neuronalen Netzen praktisch zu erproben und zu demonstrieren.

### Implementation eines Tic-Tac-Toe Netzes

Um ein neuronales Netz für einen Tic-Tac-Toe-Spieler zu entwickeln, wurde zunächst auf das Neuroph-Framework zurückgegriffen, was erste Funktionen für Reinforced Learning und neuronale Netze bereitstellt.  
Die Lösung basiert auf dem zur Verfügung gestellten .jar-Datei. Diese beinhaltet vorgefertigte Klassen, auf die bei der Entwicklung zurückgegriffen wurde.  
Des Weiteren wurde die Logik zum Trainieren und Lernen in der bereitgestellten Klasse ReinforcementSpieler implementiert. Damit dieser Spielertyp korrekt und fehlerfrei agieren kann wurden Hilfsmethoden z.B. zur Berechnung der Züge bereitgestellt.  
Der gesamte Aufbau des Programmes mit den relevanten Klassen wird der folgenden Darstellung entnommen.



Damit nun ein neuronales Netz auf das Tic-Tac-Toe-Spiel trainiert wird hat die Klasse ReinforcementSpieler die Methode “trainieren”. Diese benötigt als Parameter eine Abbruchbedingung, die vom Interface IAbbruchbedingung stammt und gibt ein boolean zurück. Das hierbei verfolgte Trainingsprinzip ist das Lernen durch wiederholte Episoden. Das heißt das Training erfolgt über eine große Anzahl an Spielwiederholungen, in dem das neuronale Netz immer wieder Züge berechnet, die im Anschluss in das Datenset einfließen. Bei dem Training werden die Aktionen bewertet und das neuronale Netz “belohnt”. Die Belohnung erfolgt über die Rückgabe eines positiven und oder negativen Wertes, je nachdem, ob ein Sieg, eine Niederlage oder ein Unentschieden das Ergebnis war. Ziel des neuronalen Netzes ist es diesen resultierenden Belohnungswert zu maximieren und damit die möglichst relevanten Züge auszuwählen, die eine positive Belohnung als Ergebnis hatten.  
Da Tic-Tac-Toe ein endliches und vergleichsweise kleines Zustands- und Aktionsmodell besitzt, kann das neuronales Netz durch die wiederholte Konfrontation mit ähnlichen Spielsituationen schrittweise robuste Strategien entwickeln und diese später in der Spielsimulation anwenden.  
Während des Trainings wird ein neuer ReinforcementSpieler instanziiert, der, sofern vorhanden ein bestehendes Datenset lädt, um dieses weiter zu trainieren.  
Damit die Züge bei den ersten Iterationen eher zufällig ausfallen, wird ein Grenzwert, in dem Fall Epsilon gleich eins gesetzt. Dieser Wert wird nach jeder Iteration verringert und bei der Berechnung eines Zuges immer an das neuronale Netz übergeben. Dies hat den Hintergrund, dass je geringer Epsilon ist, desto weniger zufällig sind die Züge. Ist Epsilon null, so greift das neuronale Netz nur auf bestehende Daten zurück und es wird kein Zufall für die Auswahl des nächsten Zuges genutzt.  
Hiernach wird in jeder Trainingsiteration ein neues Spiel mit einem Zufallsspieler und dem ReinforcementSpieler gestartet und ausgespielt.  
Nach jeder Iteration eines Trainings wird eine Belohnungsfunktion “checkReward” aufgerufen, die berechnet, welcher Belohnungswert das neuronale Netz übergeben bekommt. Dabei resultiert, je nach Ergebnis, die folgende Aufstellung:

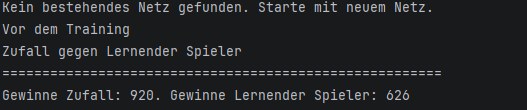
|  |  |
| --- | --- |
| Spielausgang | Belohnungswert |
| Illegaler Zug | -1 |
| Sieg | 1 |
| Unentschieden | 0.2 |
| Niederlage | -1 |

Für den Fall, dass die Exception IllegalerZugException geworfen wird, gibt es eine Try-Catch-Logik, die dies abfängt. Dies sollte jedoch nicht auftreten, da nur offene Felder dem neuronalen Netz übergeben werden.  
Wenn der Belohnungswert nun berechnet ist, gibt es eine Sub-Methode “trainNetz”, die das Netz selbst trainiert. Diese existiert zur Übersichtlichkeit und Vereinfachung beim Programmieren. Die Methode übernimmt den letzten Zustand des Feldes, sowie den letzten Zug des ReinforcementSpieler und den Belohnungswert.  
In der Methode wird das aktuelle Spielfeld in eine für das neuronale Netz verständliche Darstellung konvertiert, die ein double-Array ist.  
Danach kalkuliert das neuronale Netz den nächsten Zug und gibt das angepasste Array als Output aus.  
Außerdem werden belegte Felder maskiert, was verhindert, dass diese als “gut” verstanden werden. Diese weitere Darrstellung wird in einer weiteren Variable gespeichert, die zum Lernen später benötigt wird.  
Hiernach wird das Datenset erstellt mit einer Input- und Outputgröße von neun. Dies entspricht der Anzahl der verfügbaren Felder bei Tic-Tac-Toe. Das Datensetz bekommt eine sog. DataSetRow zugewiesen, die den letzten Zustand des Feldes beinhaltet und die maskierte Form, die zusätzlich nun auch den Zug mitsamt dem Belohnungswert inkludiert. Mithilfe von der von der bereitgestellten “learn”-Methode von Neuroph erfolgt die Übernahme des Datensets in das neuronale Netz. Wichtig ist hier, dass die learn-Methode durch das Setzen von “MaxIterations” begrenzt ist, da es sonst zu einer endlosen Laufzeit des Programms kommt.  
Neben des Trainings ist ebenfalls die Berechnung des Zuges wichtig, auf dem das Training letztendlich basiert. Dafür gibt es die Methode “berechneZug”, die durch die .jar vorgegeben war. Sie übernimmt den vorherigen Zug, sowie zwei Long-Datentypen, die die Bedenkzeiten von Kreis und Kreuz repräsentieren. Diese beiden Parameter sind für den aktuellen Stand jedoch irrelevant.  
die Sub-Methode “waehleZug” aufgerufen. Diese übernimmt als Parameter das aktuelle Spielfeld und den Wert für Epsilon. Dieser ist standardmäßig null, wie weiter oben beschrieben.  
WaehleZug lässt zunächst das aktuelle Spielfeld vom neuronalen Netz bewerten und berechnet alle verfügbaren Züge und fügt diese einer Liste hinzu. Danach beginnt der Schritt der “Exploration”, was das zufällige Ausprobieren neuer Aktionen bezeichnet.   
Mithilfe der Random Funktionalität von java.util wird ein zufälliges Double zwischen 1 und 0 erzeugt und geprüft, ob dieses kleiner ist als Epsilon. Wenn dies der Fall ist, wird ein zufälliger zulässiger Zug erstellt und zurückgegeben an die “berechneZug”-Methode. Wenn dies sonst nicht der Fall ist, dann wird der beste vom neuronalen Netz bewertete Zug gesucht und wenn dieser nicht in den verfügbaren Zügen beinhaltet ist, wird ein zufälliger Zug ausgewählt. Abschließend wird in der Methode der letzte Zustand des Feldes und des Zuges zwischengespeichert und ein Integer an berechneZug zurückgegeben.  
Das Integer wird dann in einen Zug umgewandelt auf Basis des Feldes, was der Zahl zugewiesen ist. Nun wird das intern gespeicherte Feld mit dem eigenen Zug aktualisiert und der erstellte Zug wird letztendlich als Wert an die jeweilige Tic-Tac-Toe-Instanz zurückgegeben. Damit ist der gegnerische Spieler nun am Zug.

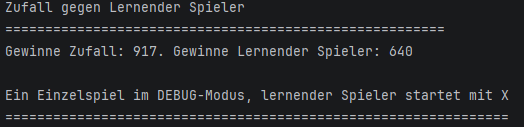
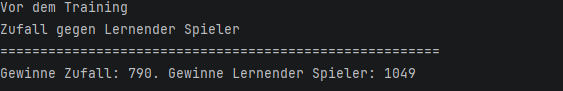
### Lernbeweis

Um zu beweisen, dass das Programm tatsächlich lernt, ist der Beweis in die folgenden Schritte aufgebaut. Hierbei wird die Wettkampf-Klasse genutzt, die schon bereitgestellt wurde.

1. Wettkampf ohne bestehendes Netz starten
2. Ergebnis der Siege evaluieren
3. Training starten (200000 Iterationen)
4. Neue Ergebnisse evaluieren
5. Wettkampf mit bestehenden Datenset neustarten

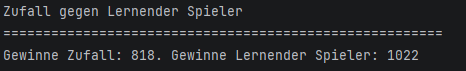
Ergebnis nach dem ersten Wettkampf:  


Es ist deutlich, dass das neuronale Netz dem Zufallspieler ohne bestehende Daten in den meisten Fällen unterliegt. Eine Differenz von 294 Siegen ist eine große Spanne, die mit Reinforced Learning verringert werden kann.  
Das Training dauert ca. 20 Minuten kann aber variieren. Zu Debugging-Zwecken wurden ein paar Zeilen in der Konsole ausgegeben, um die Funktionalität zu gewährleisten.  
Nach dem Training sieht das Ergebnis wie folgt aus:

  
Das Ergebnis ist hier leider nicht zufriedenstellend, weshalb es noch einmal gestartet wird, diesmal jedoch mit bestehenden Daten. Diese sind auch im Github-Repository hinterlegt und können nachträglich hinzugefügt werden.  
Um das Training besser zu gestalten, werden nun die Farben beim Training umgekehrt.  
Das jetzige Ergebnis vor dem Training sieht so aus:  


Hier gewinnt der lernende Spieler schon deutlich mehr als der Zufallsspieler.

Nach dem Training sehen die Daten dann wie folgt aus:



Hiermit lässt sich sagen, dass das neuronale Netz tatsächlich lernt, jedoch benötigt es sehr viele Iterationen, bis ein signifikanter Anstieg festzustellen ist, jedoch sind 204 Siege mehr ein Unterschied.

### Probleme, Lösungen und Erweiterungen

Zunächst sind bei der Entwicklung verschiedene Probleme und Fragen aufgekommen, die geklärt werden mussten. Zum einen hat eine falsche Integration von Neuroph dafür gesorgt, dass das Programm nicht ausgeführt werden konnte. Nach Umstellung der Quelle für Neuroph auf ein Github Repository, was der pom.xml entnommen werden kann, konnte das Programm mit Neuroph starten.  
Danach kam es dazu, dass, durch wenig Erfahrung mit dem Framework, das Training in eine Endlosausführung gelaufen ist und nicht nach einer festgelegten Anzahl an Iterationen geendet hat. Mithilfe des Setzens dieser, konnte das Training ebenfalls korrekt abgeschlossen werden.  
Eine große Herausforderung, war das ständige Umwandeln von Inputs bzw. Outputs für das neuronale Netz, was für viele Hilfsmethoden gesorgt hat, die die Klasse ReinforcementSpieler aufblähen. Hier wäre eine mögliche Lösung in einer zukünftigen Version, diese in eine Helper-Klasse auszulagern, die es ermöglicht alle Hilfsmethoden direkt aufzurufen.  
Eine Erweiterung, die bisher nicht umgesetzt wurde, ist die Integration von Bedenkzeiten für Spieler. Dadurch dauern die Runden u.U. nicht zu lange und die Dauer des Trainings kann genauer bestimmt werden. Hierbei wäre es jedoch nötig die Laufzeiten in den einzelnen Methoden zu berücksichtigen und mit in die Berechnung des neuen Zuges mitaufzunehmen. Dabei ist jedoch nicht geklärt, was passiert, wenn während eines Zuges die Zeit ausläuft. Es wären viele Prüfungen nötig oder man müsste eine Überwachungsklasse implementieren, die die jeweiligen Zeiten überwacht und das laufende Spiel frühzeitig beendet.  
Das Training könnte ebenfalls so erweitert werden, dass nicht eine Person auswählen muss, wogegen das neuronale Netz mit welcher Farbe trainiert, sondern es könnte mehr dem Zufall überlassen werden. D.h. es würde zufällig gewählt werden, welche Farbe der ReinforcementSpieler hat und ob er gegen einen anderen ReinforcementSpieler spielt oder gegen einen Spieler vom Typ Zufallsspieler.

Quellen:

Theoretische Grundlagen

<https://zsalloum.medium.com/basics-of-reinforcement-learning-the-easy-way-fb3a0a44f30e>

<https://de.wikipedia.org/wiki/Q-Lernen>

<https://safe-intelligence.fraunhofer.de/artikel/sicheres-reinforcement-learning>

<https://www.ibm.com/de-de/think/topics/reinforcement-learning>