Ostfalia
Hochschule für angewandte Wissenschaften
Wolfenbüttel

Fakultät Informatik

Erkan Garan, 70467533

Entwurf eines Modells für ein unterstützendes KI-System zur Bewertung von Anwendungsregeln

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science im Studiengang Informatik im Praxisverbund

an der Ostfalia Hochschule

Hochschule Braunschweig/Wolfenbüttel

Betreuer:
Prof. Dr. Claus Fühner
Dipl. -Inf. Stefan Jung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere, dass ich alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Wolfenbüttel, den 23. März 2023

Kurzfassung

Anforderungen sind bestimmte Bedingungen oder Fähigkeiten, welche ein System erfüllen oder besitzen muss. Das korrekte Erfüllen von Anforderungen ist essenziell für den Erfolg von Projekten. Eine besondere Form von Anforderungen stellen die Anwendungsregeln dar. Um einen sicheren und zuverlässigen Einsatz von Komponenten in einem Projekt zu gewährleisten, ist eine Erfüllung der Anwendungsregeln der im Projekt genutzten Komponenten vorausgesetzt. Diese Bachelorarbeit wird sich mit der Erstellung eines unterstützenden KI-Systems beschäftigen, das in der Lage sein soll, mögliche Vorschläge für Bewertungen zu Anwendungsregeln zu liefern. Dafür wird ein Deep Learning Modell mit Daten über Projekte und ihren Anwendungsregeln der Siemens Mobility GmbH trainiert. Diese Daten müssen vorher in eine für das Anlernen eines Modells geeignete Form gebracht werden. Diese Schritte der Datenvorverarbeitung werden hier vorgestellt. Zudem werden in dieser Arbeit Kenntnisse darüber vermittelt, wie künstliche Intelligenz funktioniert, was der Unterschied zwischen KI, maschinellem Lernen und Deep Learning ist und wie solch ein Modell aufgebaut, angewendet und getestet werden kann.

Abstract

Requirements are certain conditions that a system must fulfill or capabilities that it must possess. The correct fulfillment of requirements is essential for the success of projects. Application rules are a special type of requirements. To ensure the safe and reliable use of components in a project, a fulfillment of the application rules from the components used in the project is mandatory. This bachelor thesis will deal with the creation of a supporting AI system, that should be able to predict possible suggestions for the evaluation of application rules. For this purpose, a deep learning model will be trained with data on projects and their application rules from the Siemens Mobility GmbH. This data must first be transformed into a form suitable for the training of an AI model. The required data preprocessing steps are presented here. In addition, this thesis will provide knowledge about, how artificial intelligence works, what the difference between AI, machine learning and deep learning is and how such a model can be built, applied and tested.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

DOORS IBM Rational DOORS

DXL DOORS eXtension Language

KI Künstliche Intelligenz

ML Machine Learning

DL Deep Learning

NN Neuronales Netz

ReLU Rectified Linear Unit

RE Requirements Engineering

RM Requirements Management

IREB International Requirements Engineering Boards

SMO RI Siemens Mobility Rail Infrastructure

API Application Programming Interface

1 Einleitung

1.1 Motivation

1.2 Zielsetzung

1.3 Aufbau

2 Grundlagen

Die Themenbereiche Requirements Engineering (RE), speziell das Bewerten von Anwendungsregeln, und Künstliche Intelligenz (KI) stellen den Schwerpunkt dieser Bachelorarbeit dar und werden in diesem Kapitel grundlegend vorgestellt. Im Detail wird sich dieses Kapitel damit beschäftigen, was Anforderungen und Anwendungsregeln sind und wie letztere bewertet werden, sowie was KI ist, worin sich die Themen KI, Machine Learning (ML) und Deep Learning (DL) unterscheiden und wie ein KI-Modell als Neuronales Netz (NN) dargestellt werden kann.

2.1 Requirements Engineering

Nach der Definition des International Requirements Engineering Boards (IREB) bezeichnet das RE die systematische und disziplinierte Vorgehensweise bei der Spezifikation und dem Management von Anforderungen. Das Ziel des RE ist dabei, die Wünsche und Bedürfnisse der Stakeholder zu verstehen [3, vgl. S.30]. Stakeholder sind Personen oder Organisationen, die die Anforderungen des Systems direkt oder indirekt beeinflussen oder die von dem System betroffen sind [3, vgl. S.33]. Beispielsweise können Kunden oder Nutzer, aber auch der Gesetzgeber, potenzielle Stakeholder sein. Außerdem soll das Risiko minimiert werden, dass diese Wünsche und Bedürfnisse nicht oder nur unzureichend erfüllt werden [3, vgl. S.30].

Einen Teilbereich des RE stellt das Requirements Management (RM) dar. Dieser Prozess beschreibt die Verwaltung, Speicherung, Änderung sowie die Rückverfolgung von Anforderungen [3, vgl. S.8]. Um den RM Prozess zu unterstützen, können entsprechende Tools verwendet werden. Bei der Siemens Mobility GmbH wird das Tool IBM Rational DOORS (DOORS) verpflichtend eingesetzt [4, vgl. S.18]. Näheres zu DOORS kann dem Kapitel 3.1 entnommen werden.

2.1.1 Anforderungen

Die IEEE definiert eine Anforderung wie folgt:

> „(1) A condition or capability needed by a user to solve a problem or achieve an objective.
> (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
> (3) A documented representation of a condition or capability as in (1) or (2).“ [5, S.62]

Daher bilden Anforderung die Basis eines jeden Projekts, da diese definieren, welche Bedingungen ein System erfüllen muss bzw. welche Fähigkeiten es besitzen muss. Sie werden idealerweise unter Berücksichtigung und in Zusammenarbeit mit den Stakeholdern des Projekts ermittelt. Neben den Stakeholdern können unter anderem auch Normen, Gesetze oder Vorgänger eines Systems weitere Quellen für Anforderungen sein. Um von jedem Stakeholder des Projekts verstanden zu werden, werden Anforderungen in der Regel in natürlicher Sprache formuliert. Da natürliche Sprache Raum für Interpretation bieten kann, muss darauf geachtet werden, dass die Anforderungen so klar und unmissverständlich wie möglich formuliert werden und dabei trotzdem vollständig bleiben. Zudem wird voraussichtlich nicht jeder Stakeholder über die fachlichen Kenntnisse verfügen um Fachsprache oder Konventionen zu verstehen, weshalb darauf verzichtet werden sollte [6, vgl. S.2]. Um sicherzustellen, dass Anforderungen korrekt formuliert werden, wurden in der Norm ISO/IEC/IEEE 29148:2018(E) Eigenschaften definiert, welche Anforderungen erfüllen sollen. Diese Eigenschaften werden nachfolgend dargestellt und kurz beschrieben:

Notwendig Die Anforderung definiert eine wesentliche Fähigkeit, Eigenschaft, Einschränkung und/oder einen Qualitätsfaktor [7, vgl. S.12].

Angemessen Die Anforderung verfügt über einen angemessenen Detaillierungsgrad und erlaubt dabei bei der Implementierung größtmögliche Unabhängigkeit [7, vgl. S.12].

Eindeutig Die Anforderung ist leicht zu verstehen, einfach formuliert und kann nur auf eine einzige Weise interpretiert werden [7, vgl. S.12].

Komplett Die Anforderung ist hinreichend beschrieben und benötigt keine weiteren Informationen um verstanden zu werden [7, vgl. S.12].

Atomar Die Anforderung beschreibt eine einzige Fähigkeit oder Bedingung [7, vgl. S.12].

Durchführbar Die Anforderung kann innerhalb der Beschränkungen des Systems mit akzeptablem Risiko durchgeführt werden [7, vgl. S.13].

Verifizierbar Die Umsetzung der Anforderung kann überprüft werden [7, vgl. S.13].

Korrekt Die Anforderung ist eine genaue Darstellung des Bedürfnisses ihrer Quelle [7, vgl. S.13].

Konform Die Anforderung wurde, wenn möglich, mithilfe einer genehmigten Standardvorlage und -stil verfasst [7, vgl. S.13].

Um diese Eigenschaften zu erfüllen, ist es ratsam auf Vorlagen für das Schreiben von Anforderungen zurückzugreifen. Eine Anforderung, welche mithilfe einer Vorlage verfasst wurde, könnte wie folgt aussehen:

> „The <system> shall <function> <object> every <performance> <units>.
> E.g. The coffee machine shall produce a hot drink every 10 seconds. “
> [6, S.81]

Nach einer Umfrage aus dem Chaos Report der Standish Group nennen mehr als die Hälfte der Befragten als Faktor für die Beeinträchtigung von Projekten einen Grund, der direkt im Zusammenhang mit mangelndem RE und RM steht. Dazu gehören z.B. Gründe wie unvollständige Anforderungen, Nutzer nicht ausreichend involviert, unrealistische Erwartungen oder geänderte Anforderungen und Spezifikationen [8, vgl. S.5]. Gut durchgeführtes RE und RM ist also essenziell für den Erfolg von Projekten.

2.1.2 Anwendungsregeln

Eine spezielle Form von Anforderungen stellen die Anwendungsregeln dar. Diese Anforderungen werden an Komponenten gestellt, um einen sicheren und zuverlässigen Einsatz von Komponenten im System zu gewährleisten. Dafür müssen sie von dem Kunden oder dem Projekt, welches die jeweiligen Komponenten nutzt, berücksichtigt werden [9, vgl. S.9]. Alle Anwendungsregeln von Komponenten der Siemens Mobility Rail Infrastructure (SMO RI) in all ihrer Versionen befinden sich zentral gespeichert in einem Projekt im Tool DOORS. Von dort kann der zuständige Requirements-Manager eines Projekts alle Anwendungsregeln von Komponenten importieren, die im jeweiligen Projekt genutzt werden. Als Nächstes muss er bewerten, welche Anwendungsregeln für das Projekt anwendbar sind und welche nicht. Die anwendbaren Anwendungsregeln müssen daraufhin mit einer Designlösung abgeschlossen werden oder einem oder mehreren Teilsystemen zugeordnet werden. Diese Bewertung der Anwendungsregeln erfolgt ebenfalls in DOORS wie in der Abbildung 2.1 gezeigt wird.

ID		REQ Status	REQ Statement
1	1. Die maximale Länge der Strecke, über die die Signalfrequenzen f1/f2 vom ZP 43E/V bzw. ZP D 43 zum Achszählrechner Az S 350 U ohne Abtrennung mit einem Leitungstrennübertrager erfolgen darf, beträgt 6,5 km (bei einer Kabelkapazität von 50 nF/km).	closed	Gesamtlänge der Strecke beträgt 5km.

Abbildung 2.1: Bewerten einer Anwendungsregel

Dafür werden die Attribute REQ Statement und REQ Status genutzt. Im Attribut REQ Status kann der System-Manager eine von fünf verschiedenen Ausprägungen wählen. Eine Auflistung und Erklärung dieser Ausprägungen kann der Tabelle 2.1 entnommen werden. Der System-Manager muss daraufhin ein Statement darüber abgeben, warum er welchen REQ Status gewählt hat. Dieses Statement wird in Textform im Attribut REQ Statement gespeichert. Die Anwendungsregel in der Abbildung 2.1 wurde beispielsweise mit closed bewertet, da die Anwendungsregel für das Projekt anwendbar ist und zudem auch gelöst ist, da die Streckenlänge, wie in der Anwendungsregel gefordert, weniger als 6,5 km beträgt.

Tabelle 2.1: Ausprägungen des Attributs REQ Status [9, vgl. S.28f.]

REQ Status	Erklärung
not applicable	Anwendungsregel ist nicht anwendbar
forwarded	Anwendungsregel soll an Subsystem weitergeleitet werden
closed	Anwendungsregel auf System Design Ebene mit einem Statement abgeschlossen
open	Bewertung der Anwendungsregel noch offen
compliant	Anwendungsregel anwendbar, noch nicht bewertet

2.2 Künstliche Intelligenz

Das Thema KI hat in letzter Zeit eine große mediale Aufmerksamkeit erhalten. Ein aktuelles Beispiel dafür ist der Chatbot ChatGPT der Firma OpenAI. ChatGPT wurde im November 2022 veröffentlicht und ist ein Sprachmodell, das natürliche Sprache verstehen und abhängig von der Benutzereingabe in Dialogform Antworten liefern soll. Es ist dabei in der Lage, sich an vorherige Eingaben zu erinnern und kann deshalb Anfragen in einen Kontext einordnen. Dadurch kann es auch vorherige Antworten korrigieren und auf Wünsche und Bedürfnisse des Benutzers eingehen [10].

Die Frage nach KI und nach dem ob und wie Maschinen denken können ist jedoch wesentlich älter als dieser Chatbot. Bereits Alan Turing stellte sich diese Frage in den 1950er Jahren. Er schlug vor, die Frage, ob Maschinen denken können, mit einer anderen Frage zu ersetzen. Dafür formulierte er das Problem mithilfe eines Spiels, dem „Imitation Game“, das heute eher als Turing-Test bekannt ist, um. Dieses Spiel wird mit drei Personen gespielt, einem Fragesteller, einem Mann und einer Frau. Ziel des Spiels ist es, dass der Fragesteller mithilfe von Fragen herausfinden soll, welche Person der Mann und welche Person die Frau ist. Der Mann soll dabei versuchen, den Fragesteller dazu zu bringen, ihn fälschlicherweise als die Frau zu identifizieren, während die Frau versuchen soll, dass der Fragesteller sie korrekt als Frau identifiziert. Dafür befindet sich der Fragesteller in einem anderen Raum und die Fragen werden entweder über eine außenstehende weitere Person oder einem

Fernschreiber übermittelt. Was würde nun passieren, wenn eine Maschine die Rolle des Mannes übernehmen würde? Würde der Fragesteller dann häufiger oder seltener gewinnen? Alan Turing glaubte, dass eine Maschine dann als intelligent bezeichnet werden könne, wenn eine Maschine in der Lage dazu wäre, menschliches Verhalten zu imitieren [11, vgl. S.433f.].

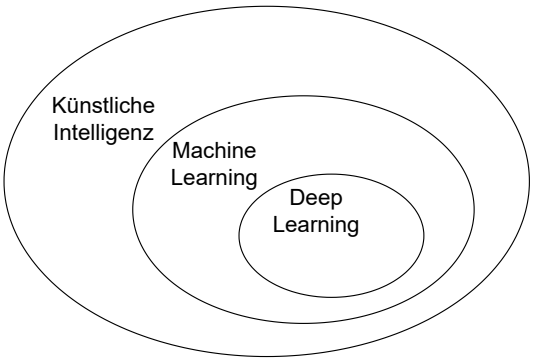

Abbildung 2.2: Beziehung zwischen KI, ML und DL [1, S.22]

François Chollet, der Entwickler der Deep-Learning-Bibliothek Keras, welche im Kapitel TODO näher beschrieben wird, definiert das Fachgebiet KI als „[den] Versuch, normalerweise von Menschen erledigte geistige Aufgaben automatisiert zu lösen“ [1, S.22]. Nach dieser Definition schließt KI weitere Themen wie das ML sowie das DL ein. Wichtig zu beachten ist dabei, dass diese Gebiete sich voneinander unterschieden. Ihre Beziehung zueinander wird in Abbildung 2.2 verdeutlicht.

Dieses Kapitel wird näher erläutern, worum es sich bei den Themen ML und DL handelt und wie ein KI-Modell als neuronales Netz erstellt wird.

2.2.1 Maschinelles Lernen

Das maschinelle Lernen ist ein Teilgebiet der KI und beschäftigt sich mit der Frage, ob ein Computer in der Lage dazu ist selbstständig eine bestimmte Aufgabe zu erlernen. Ziel dabei ist es, dass eine Maschine aus einem vorgegebenem Datensatz und den dazugehörigen Antworten Regeln extrahieren soll, die den Datensatz erklären können. Anders als bei der klassischen Programmierung soll eine Maschine hier nicht die Antworten aus den Daten und Regeln herausgeben, sondern selber nach

einer Struktur suchen, aus der die Maschine dann Regeln ableiten kann, die auch auf andere Aufgaben angewendet werden können [1, vgl. S.23f.]. Angenommen ein Datensatz bestünde aus Bildern von Hunden und Menschen. Bei der klassischen Programmierung würde der Programmierer nun selber Regeln definieren und diese programmieren müssen. Beispiele für mögliche Regeln könnten hier sein:

> Wenn Wesen Fell hat, dann ist es ein Hund.
> Wenn Wesen auf zwei Beinen läuft, dann ist es ein Mensch.

Beim ML hingegen müsste der Programmierer diese Regeln nicht selber definieren. Hier werden neben dem Datensatz noch die entsprechenden Antworten benötigt. Jedes Bild bräuchte also ein Label, also eine Information darüber, ob es sich auf dem Bild um einen Hund oder einen Menschen handelt. Aufgabe der Maschine wäre es nun, mithilfe des Datensatzes und der Label, Regeln zu definieren, ob auf einem Bild ein Hund oder ein Mensch zu sehen ist. Das ML-System wird also sozusagen trainiert. Dem trainiertem System können dann neue Bilder gezeigt werden und es wäre in der Lage anhand seiner definierten Regeln vorherzusagen, ob auf den neuen Bildern Hunde oder Menschen zu sehen sind.

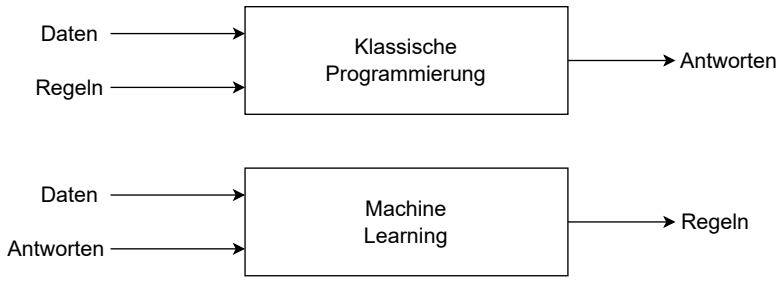

Abbildung 2.3: Unterschiedliche Programmierparadigmen [1, S.23]

Abbildung 2.3 verdeutlicht nochmal die Unterschiede zwischen der klassischen Programmierung und dem ML. Beim ML sind also drei Elemente notwendig, diese werden anhand des oben genannten Beispiels verdeutlicht:

Eingabedaten Bilder von Menschen und Hunden [1, S.24]

Antworten Label, ob auf dem Bild ein Mensch oder ein Hund abgebildet ist [1, S.24]

Metrik zur Bewertung des Algorithmus Benötigt, um die Abweichung zwischen Ausgabe des Modells und eigentlicher Antwort zu bestimmen. Wird als Feedback-Signal genutzt, um Algorithmus anzupassen. Beispielsweise wie viel Prozent der Bilder richtig zugeordnet werden. [1, S.24f.].

Die Aufgabe eines ML-Modells ist es passende Repräsentationen der Eingabedaten zu erlernen, das heißt, dass die Daten vom Modell sinnvoll umgewandelt werden müssen. Die systematische und automatische Suche nach der bestmöglichen Repräsentationen der Daten mithilfe eines Feedback-Signals für eine bestimmte vorgegebene Aufgabe kann als Möglichkeit verstanden werden, wie Maschinen lernen können. Mithilfe dieses Ansatzes und der in den letzten Jahren besser werdenden Hardware können große Datenmengen analysiert werden, was das Lösen von Aufgaben wie der Spracherkennung oder dem autonomen Fahren ermöglicht hat [1, vgl. S.24ff.].

Die hier beschriebene Variante des Machine Learnings wird auch Supervised Learning, also überwachtes Lernen, genannt. „Supervised“ bezieht sich hierbei darauf, dass dem Modell die Antworten vorgegeben werden und es anhand der Antworten versucht Regeln zu finden. Neben dieser Variante existiert auch noch das sogenannte Unsupervised Learning. Bei dieser Variante des Machine Learnings werden dem Modell keine Antworten gegeben. Stattdessen versucht das Modell die Daten anhand ihrer Beziehung zueinander zu analysieren und die Daten in Kategorien zu klassifizieren. Algorithmen zum Clustern von Datensätze sind typische Beispiele für das Unsupervised Learning [12, vgl. S.47ff.]. In dieser Arbeit finden Unsupervised Learning Algorithmen oder Modelle keine Anwendung, deshalb ist mit ML hier immer Supervised Learning gemeint.

2.2.2 Deep Learning und neuronale Netze

Wie Abbildung 2.2 zeigt, ist das DL ein Teilgebiet des ML und beschäftigt sich somit ebenfalls mit der Suche nach den bestmöglichen Repräsentationen von Daten. Das Besondere am DL ist, dass das Lernen des Modells in mehreren aufeinanderfolgenden Schichten, auch Layer genannt, stattfindet. Dabei können beliebig viele

Schichten eingesetzt werden. Die Anzahl an Schichten wird auch Tiefe des Modells genannt, was auch das „Deep“ in DL beschreibt. Je tiefer dabei eine Schicht liegt, desto sinnvoller sollen die Repräsentationen werden [1, vgl. S.27]. François Chollet definiert DL als „Lernen durch schichtweise Repräsentationen oder Lernen durch hierarchische Repräsentationen“ [1, S.27]. DL-Modelle werden in der Regel als NN dargestellt, das aus beliebig vielen Schichten besteht, wobei jedes NN mindestens eine Eingabe- und eine Ausgabeschicht besitzt, dazwischen kann es beliebig viele versteckte Schichten, die hidden Layer, beinhalten. Die versteckten Schichten sind für die Verarbeitung der Daten zuständig. Eine Schicht besteht dabei aus einer vom Ersteller des NN ausgewählten Anzahl an Neuronen, die miteinander verbunden sind [2, vgl. S.26]. In der Abbildung 2.4 wird der grundlegende Aufbau eines einfachen neuronalen Netzes dargestellt.

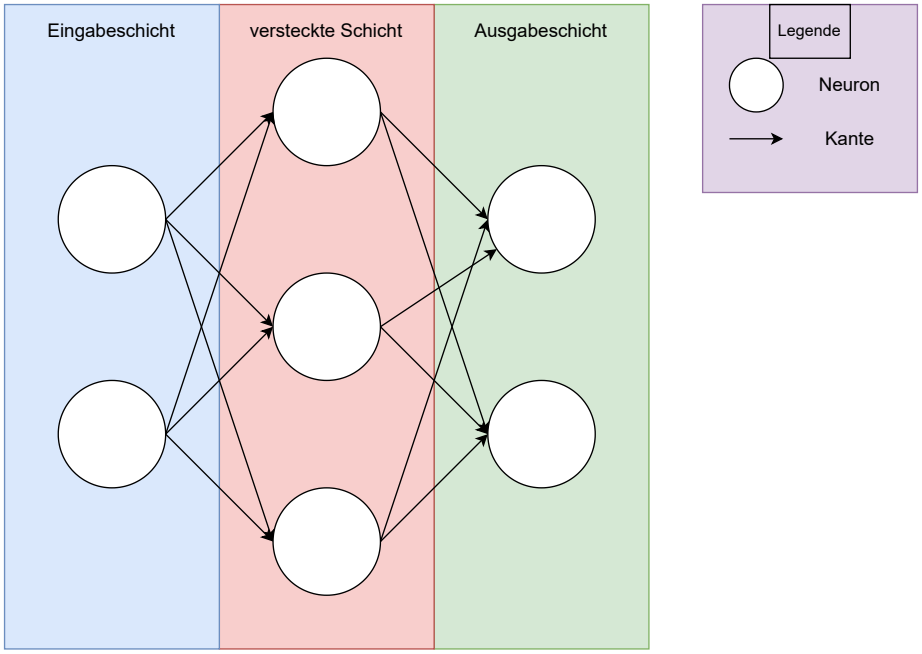

Abbildung 2.4: Aufbau eines neuronalen Netzes [2, S.27]

Die Verbindungen zwischen den Neuronen, die Kanten genannt werden, sind gerichtet und besitzen ein Gewicht. Wenn jedes Neuron eine Verbindung zu jedem Neuron in der nächsten Schicht hat, dann wird diese Schicht auch als fully-connected bezeichnet. Das NN in Abbildung 2.4 besteht aus drei Schichten, welche, bis auf die Ausgabeschicht, fully-connected sind. Unabhängig davon bekommt jedes Neuron, welches sich nicht in der Eingabeschicht befindet, eine Netzeinga-

be, welche sich aus den Ausgaben der Neuronen zusammensetzt, die eine Kante zu dem jeweiligen Neuron haben. Dabei wird jede Ausgabe eines Neurons mit dem Kantengewicht multipliziert.

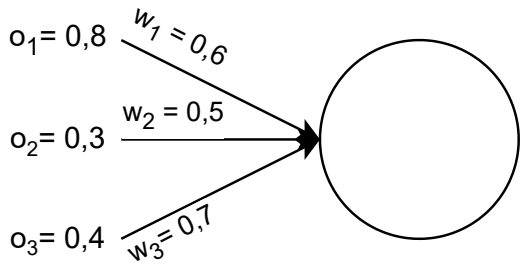

Abbildung 2.5: Berechnung Netzeingabe [2, vgl. S.29]

Abbildung 2.5 zeigt beispielhaft ein Neuron und die Ausgabe von drei vorherigen Neuronen. Die Netzeingabe des Neurons wird wie folgt berechnet [2, S.29]:

$$net = o_1 \cdot w_1 + o_2 \cdot w_2 + o_3 \cdot w_3 = 0,8 \cdot 0,6 + 0,3 \cdot 0,5 + 0,4 \cdot 0,7 = 0,91 \quad (2.1)$$

Jedes Neuron besitzt eine Aktivierungsfunktion, welche mithilfe der Netzeingabe prüft, ob das jeweilige Neuron eine Ausgabe hat und welchen Wert diese annimmt. Es existieren viele verschiedene Aktivierungsfunktionen, in dieser Arbeit wird die Rectified Linear Unit (ReLU)-Funktion genutzt. Diese ist definiert als [2, S.31]:

$$f(net) = max(0; net) \quad (2.2)$$

Diese Funktion nimmt die Netzeingabe und wenn die Netzeingabe positiv ist, dann wird die Netzeingabe als Ausgabe ausgegeben, ansonsten gibt das Neuron 0 als Ausgabe aus. Eine ReLU-Funktion als Aktivierungsfunktion eines Neurons zu benutzen ist typisch im Bereich des DL [2, vgl. S.31].

Wie kommen nun die Kantengewichte zustande? Den Kanten werden zunächst zufällige Werte als Gewicht zugeordnet. Danach wird jede Eingabe an das Modell übergeben, die daraufhin die einzelnen Layer durchläuft. Die Vorhersage wird nun mit dem tatsächlichen Wert an eine Verlustfunktion übergeben, welche dann den Verlustscore berechnet. Für eine Regression, also die Vorhersage eines numerischen Wertes, bietet sich beispielsweise die Berechnung des mittleren quadratischen

Fehlers (mean squared error) an. Dieser Verlustscore wird als Feedback-Signal genutzt und gibt an, wie gut das Modell die Aufgabe lösen kann. Das DL-Modell versucht während des Trainings diesen Wert zu minimieren, indem der Verlustscore an einen Optimierer übergeben wird, der anhand des Verlustscores die Gewichtungen der Kanten im NN aktualisiert. Am Anfang des Trainingsprozesses wird der Verlustscore relativ hoch sein, da die Kantengewichte zufällig ausgewählt wurden. Mit jeder neuen Eingabe werden die Kantengewichte jedoch aktualisiert, was zur Folge hat, dass das Modell immer genauer wird und der Verlustscore somit abnimmt [1, vgl. S.30ff.]. Abbildung 2.6 zeigt dabei grafisch den Aufbau des Trainingsprozesses eines NN. Je mehr Daten vorhanden sind, desto genauer kann das Modell somit werden. Die ImageNet-Datenbank mit ihren 1,4 Millionen Bilddateien hatte beispielsweise einen großen Einfluss auf den Erfolg von DL [1, vgl. S.45]. Wichtig zu beachten ist, dass der Datensatz in einen Trainings- und einen Testdatensatz aufgeteilt werden muss. Ein Modell darf niemals mit den Daten trainiert werden, die auch zum Testen genutzt werden. Wird ein Datensatz nicht aufgeteilt, wird das Modell logischerweise beim Testen sehr gute Ergebnisse erzielen, da es die Testdaten bereits kennt. Der Testdatensatz wird zudem genutzt, um die Genauigkeit des Modells bei unbekannten Daten zu bestimmen. Dieser Schritt ist wichtig, da so geprüft werden kann, wie gut ein Modell mit neuen Daten umgehen kann und ob es in der Lage ist diese korrekt vorherzusagen.

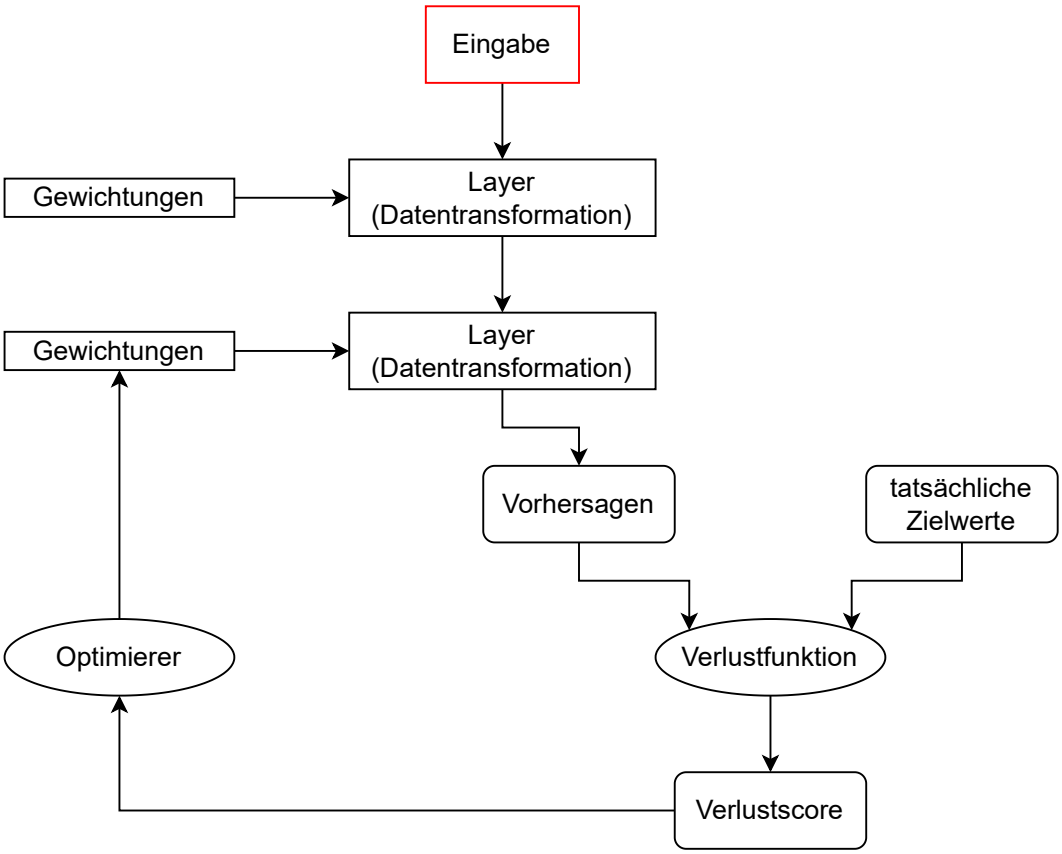

Abbildung 2.6: Anlernen eines neuronalen Netzes [1, S.31]

2.2.3 Overfitting und Underfitting

Overfitting und Underfitting, auf Deutsch Über- und Unteranpassung, können im Trainingsprozesses von ML- sowie DL-Aufgaben auftreten. Unter dem Begriff Underfitting wird verstanden, dass ein Modell noch nicht alle Zusammenhänge im Trainingsdatensatz erkannt und modelliert hat und somit der Verlustscore noch hoch ist, da das Modell für den Trainingsdatensatz noch keine sinnvollen Repräsentationen gefunden hat. Dieses Problem kann gelöst werden, indem die Anzahl an Trainingsdurchläufen erhöht oder die Komplexität des Modells erweitert wird, also weitere Schichten oder mehr Neuronen in einer Schicht hinzugefügt werden. Overfitting beschreibt dabei den Fall, dass ein Model die Eigenheiten der Trainingsdaten sozusagen auswendig lernt und mögliche irrelevante Muster im Trainingsdatensatz erlernt. Das Problem daran ist, dass das Modell dann nicht mehr in der Lage ist, das eigentliche Problem generalisiert zu betrachten und neue Daten korrekt vor-

herzusagen. Mögliche Lösungen wäre dabei mehr Daten zu beschaffen oder die Komplexität des Modells zu verringern. Zudem existiert ein Verfahren, das ebenfalls dabei hilft, Overfitting zu verhindern. Dieses Verfahren nennt sich Dropout-Regularisierung. Bei der Dropout-Regularisierung werden während des Trainings einige zufällig ausgewählte Ausgaben von Neuronen in einer Schicht auf 0 gesetzt. Das soll verhindern, dass ein NN irrelevante Muster im Trainingsdatensatz erlernt [1, vgl. S.142ff.].

Je weniger Daten vorhanden sind und je komplexer ein Modell ist, desto wahrscheinlicher läuft das Modell Gefahr, von Overfitting betroffen zu sein. Je simpler ein Modell ist und je weniger Trainingsdurchläufe durchlaufen werden, desto eher kann es zu einer Unteranpassung des Modells kommen. Es ist also wichtig, ein gutes Mittelmaß zu finden, damit ein Modell in der Lage ist ein Problem verallgemeinert und optimal lösen zu können.

2.2.4 Vor- und Nachteile

Der große Vorteil von DL und NN ist das „Universal Approximation Theorem“, welches besagt, dass jeder funktionale Zusammenhang zwischen Ein- und Ausgabe durch ein NN angenähert werden kann. Das gilt dabei nicht nur für mathematische Zusammenhänge. Voraussetzung dafür ist ein NN, dessen Komplexität groß genug gewählt wird. Dafür kann bereits eine versteckte Schicht ausreichen. Durch diese Eigenschaft kann ein NN theoretisch jedes Approximationsproblem lösen und interessiert sich dabei nicht dafür, ob eine Kausalität zwischen Ein- und Ausgabe besteht und kann ebenfalls mit unvollständigen und falschen Daten arbeiten [2, vgl. S.74ff.]. Zudem sind NN nicht anfällig gegenüber leichten Änderungen im Datensatz, stattdessen sind sie fehlertolerant. Selbst wenn ein Teil des Netzes funktionsunfähig werden sollte, spielt das keine allzu große Rolle, da die Muster und Strukturen, welches ein Modell gelernt hat, auf das gesamte Netz verteilt ist. Diese Tatsache hat zudem den Vorteil, dass wenn neue Daten dazu kommen, nicht das gesamte Netz neu trainiert werden muss, sondern das bestehende Netz upgedatet werden kann [2, vgl. S.82]. Weitere Vorteile von NN sind ihre Einfachheit und Skalierbarkeit. Das Erstellen eines Modells ist einfach und auf eine beliebige Größe skalierbar, wes-

halb NN auch mit großen Datenmengen und einer Vielzahl an Attributen angelernt werden können [1, vgl. S.47].

Neben den Vorteilen bringt die Nutzung von NN auch einige Nachteile mit sich. Zum einen ist die Genauigkeit der Prognose eines Modells stark abhängig von der Qualität der Trainingsdaten. Werden Attribute im Trainingsdatensatz genutzt, die irrelevant bei der Beschreibung des Problems sind, wirkt sich das negativ auf die Genauigkeit des Modells aus, weshalb eine vorhergehende Analyse der Attribute, die sogenannte „Feature Selection“, unausweichlich ist. Zudem kann das Training eines NN mit steigender Komplexität zeit- und rechenintensiv werden, da die Anzahl an zu berechnenden Parametern bei steigender Komplexität zunimmt. Ein komplexeres Modell hat außerdem zur Folge, dass es nicht mehr möglich ist transparent und nachvollziehbar zu Verstehen, wie ein NN eine Ausgabe berechnet. Außerdem muss beim Testen und Validieren eines Modells überprüft werden, ob das Modell nicht eventuell over- oder underfitted und somit nicht in der Lage ist, das zu lösende Problem hinreichend verallgemeinert abzubilden und somit auf neuen Daten zu fehlerhaftem Verhalten neigt [2, vgl. S.84ff.].

3 Werkzeuge

In diesem Kapitel werden die Tools vorgestellt, die in der Arbeit verwendet werden. Dazu zählt das Anforderungsmanagement-Tool DOORS, da es von der Siemens Mobility GmbH genutzt wird, um Anforderungen und somit auch Anwendungsregeln zu verwalten.

Das KI-Modell, welches in dieser Arbeit erstellt und vorgestellt wird, wird in der Programmiersprache Python geschrieben. Dieses Kapitel wird sich deshalb auch mit Python und den Bibliotheken, welche hier genutzt werden, beschäftigen und einen Überblick über diese geben.

3.1 IBM Rational DOORS

Das Anforderungsmanagement-Tool DOORS ist ein plattformübergreifendes und unternehmensweites Tool und wird zur Erfassung, Verknüpfung, Verfolgung, Analyse und Verwaltung von Anforderungen genutzt. DOORS ist ein Akronym, das für Dynamic Object-Oriented Requirements System steht. Alle Anforderungen und weitere Informationen werden in einer zentralen Datenbank gespeichert. Innerhalb der Datenbank werden die Informationen in Modulen gespeichert. Diese Module können mithilfe von Ordnern und Projekten organisiert werden. Ordner sind vergleichbar mit den Ordnern z.B. im Windows Explorer und können andere Ordner, Projekte oder Module beinhalten. Ein Projekt hingegen ist ein spezieller Ordner, der alle Daten für ein entsprechendes Projekt beinhaltet. Sowohl für Ordner als auch für Projekte können die Zugriffsrechte individuell eingestellt werden [6, vgl. S.173]. Dabei existieren die Optionen read, modify, create, delete und administer (RMCDA).

3.1.1 Module

Es existieren zwei verschiedene Arten von Modulen im Anforderungsmanagement-Tool DOORS. Module, die die eigentlichen Anforderungen beinhalten, werden Formal Module genannt. Abbildung 3.1 zeigt ein Beispiel für so ein Modul. Zu erkennen ist dort ein geöffnetes Formal Module. Auf der linken Seite ist ein Explorer zu sehen, auf der rechten Seite die eigentlichen Inhalte des Moduls. Durch den Explorer auf der linken Seite, der in einer Baumstruktur organisiert ist, wird es dem Benutzer ermöglicht leicht zu einer bestimmten Stelle im Modul zu navigieren. Dabei können die einzelnen Sektionen auf- und zugeklappt werden [6, vgl. S.176]. Die Daten auf der rechten Seite sind tabellarisch angeordnet. Die Spalten stellen dabei die einzelnen Attribute des Moduls dar, während die Zeilen die Objekte darstellen.

Neben den Formal Modules existieren auch die Link Modules. In diesen werden Informationen über die Beziehungen zwischen einzelnen Objekten gespeichert, was die Verfolgbarkeit von Anforderungen gewährleistet.

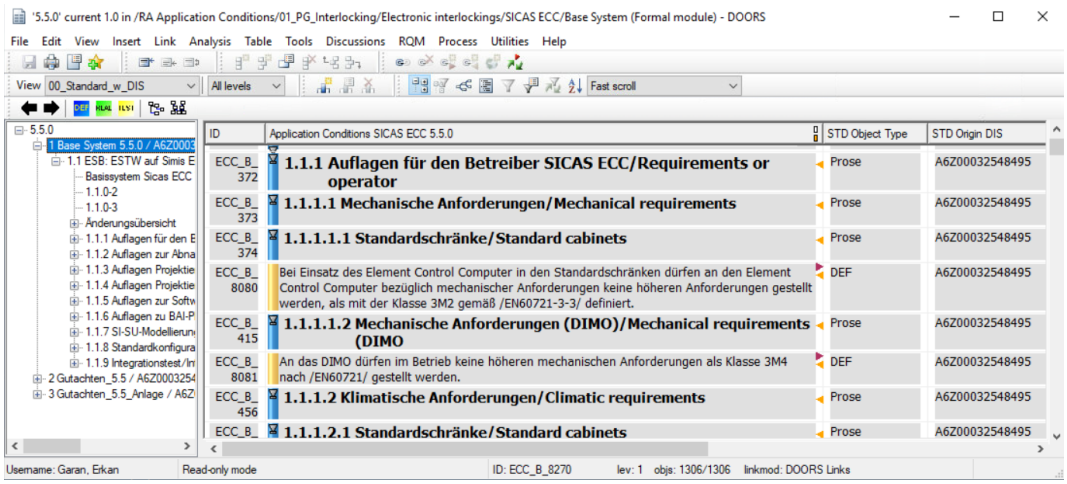

Abbildung 3.1: Geöffnetes Modul in DOORS

3.1.2 Objekte und Attribute

Innerhalb eines Moduls werden Daten in Objekten gespeichert. In der Regel bestehen Objekte aus mindestens zwei Spalten. Die erste Spalte enthält eine ID, die sich aus einem Präfix und einem Integerwert zusammensetzt. Der Integerwert wird bei jedem neu angelegtem Objekt inkrementiert, sodass jedem Objekt innerhalb eines

Moduls eine eindeutige ID zugeordnet werden kann. Die zweite Spalte besteht dabei entweder aus einer Sektions-Nummer und einer Überschrift, wie im ersten Objekt in der Abbildung 3.1 zu sehen ist, oder aus einem Objekt-Text, der beispielsweise eine Anforderung beinhalten kann. Ein Beispiel für einen Objekt-Text, der eine Anforderung beinhaltet, ist das vierte Objekt der Abbildung 3.1 [6, vgl. S.178]. Einem Objekt können beliebig viele weitere Attribute hinzugefügt werden.

Attribute beinhalten relevante Informationen über Module oder Objekte. Modulattribute speichern Informationen über das Modul, wie beispielsweise den Ersteller des Moduls, das letzte Änderungsdatum und Ähnliches. Diese Modulattribute findet der Benutzer über die Eigenschaften des Moduls, welche mit einem Rechtsklick auf das Modul in der grafischen Oberfläche geöffnet werden können. Objektattribute hingegen speichern Informationen über die Objekte. In der Abbildung 3.1 ist die Spalte STD Object Type z.B. ein Objektattribut, das definiert, ob es sich bei dem Objekt um eine Anforderung oder um Prosa, also z.B. eine Überschrift handelt.

3.1.3 Baseline

Eine Baseline friert den aktuellen Stand der Anforderungen eines Projekts mit ihren Attributen ein und ist eine nicht veränderbare Kopie von formalen Modulen [6, vgl. S.182]. Baselines werden in der Regel zu Releases von Systemen oder Subsystemen erstellt [4, vgl. S.60]. Wenn ein Formal Module geöffnet ist, kann der Benutzer unter File $\rightarrow$ Baseline eine Baseline erstellen oder eine bereits vorhandene Baseline ansehen. Abbildung 3.2 zeigt das Dialogfenster zum Öffnen bereits vorhandener Baselines. Dort wird deutlich, dass Baselines versioniert werden können.

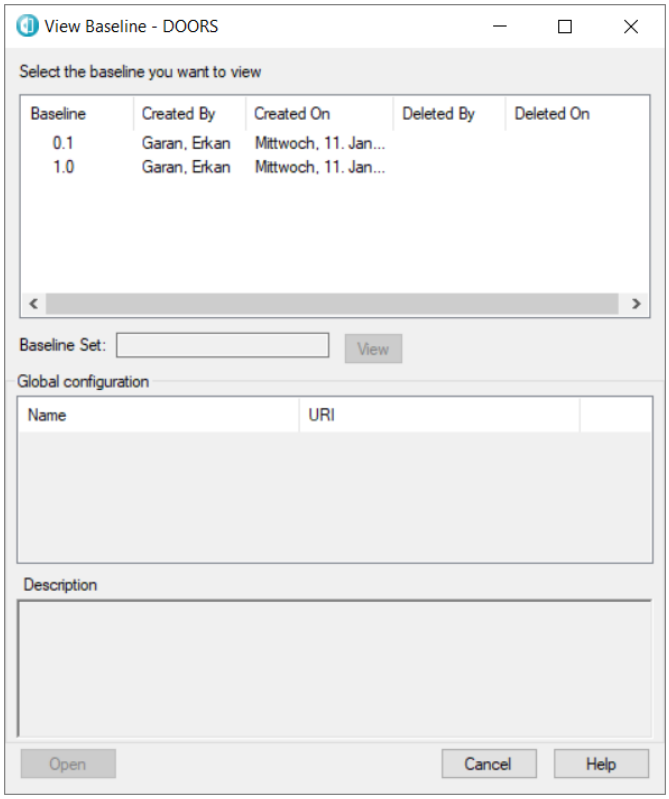

Abbildung 3.2: Baselines eines Moduls in DOORS

3.1.4 Links

In der Abbildung 3.1 sind an der rechten Kante der zweiten Spalte zwei Arten von Pfeilen zu erkennen. Diese Pfeile symbolisieren Links in DOORS. Links sind gerichtete Verbindungen von einem Quellobjekt zu einem Zielobjekt. Sie werden in DOORS genutzt, um die Verfolgbarkeit von Anforderungen zu gewährleisten. Der Benutzer kann dabei, ungeachtet von der Richtung des Links, vom Quellobjekt zum Zielobjekt oder andersherum navigieren [6, vgl. S.183]. Ein nach links zeigender gelber Pfeil ist dabei ein In-Link, das heißt, dass dieses Objekt als Zielobjekt dient und ein anderes Objekt eine Verbindung zu diesem Objekt hat. Das Gegenstück zum In-Link ist ein Out-Link. Dieser wird in der grafischen Benutzeroberfläche von DOORS als roter nach rechts zeigender Pfeil dargestellt. Hat ein Objekt einen Out-Link, heißt das, dass dieses Objekt als Quellobjekt dient und eine Verbindung zu einem anderen Objekt, welches als Zielobjekt dient, hat.

3.1.5 DXL

DOORS eXtension Language (DXL) ist eine Skript-Sprache, die ein Teil des Anforderungsmanagement-Tools DOORS ist. Durch diese Skript-Sprache können Skripte geschrieben werden, die als Batch-Skript ausgeführt werden können. Diese Skripte bieten neben der grafischen Benutzeroberfläche eine weitere Möglichkeit, um mit DOORS zu arbeiten. Zudem besteht die Möglichkeit die grafische Benutzeroberfläche von DOORS um neue, entwickelte Anwendungen zu erweitern. Von der Syntax ähnelt die Sprache den Programmiersprachen C und C++ [13, vgl. S.1]. Eine Besonderheit von DXL ist dabei der Datentyp Skip, welche eine Skiplist als Datenstruktur implementiert. Eine Skiplist besteht aus Key-Value-Paaren und ermöglicht einen schnellen Zugriff auf einzelne Elemente.

DXL wurde im Praxisprojekt zum Sammeln von Bewertungen von Anwendungsregeln aus Projekten der Siemens Mobility GmbH genutzt, indem Batch-Skripte geschrieben wurden. Diese Skripte haben im zentralen Projekt, das die Anwendungsregeln von Komponenten beinhaltet, die Links der Objekte verfolgt und dort in den Modulen nach korrekt bewerteten Anwendungsregeln gesucht. Diese wurden dann nach Projekt und Komponente gruppiert und jeweils in neue Module geschrieben, um in Zukunft als Lösungsvorschlag für neue Projekte zu dienen. Der Inhalt dieser Module wird als Datensatz für diese Bachelorarbeit genutzt. Ebenfalls wird DOORS dazu benötigt, um in der grafischen Oberfläche eines Moduls mit neu importierten Anwendungsregeln ein Programm zu starten, was mit den Informationen aus dem Modul ein weiteres Skript in der Programmiersprache Python startet. In dem Python-Skript wird das KI-Modell erstellt und angelernt. Mithilfe des Modells und mit den Daten über die Anwendungsregeln aus dem Modul soll das Modell dann Vorschläge zur Bewertung der Anwendungsregeln liefern.

3.2 Python

Python ist eine interpretierende, high-level- und objektorientierte Programmiersprache und wird von einem Artikel im International Research Journal of Engineering and Technology (IRJET) als die am schnellsten wachsende Programmiersprache bezeichnet [14, vgl. S.354]. Jacqueline Kazil, ehemaliges Vorstandsmitglied der

Python Software Foundation, nennt als Hauptgründe für das Wachstum der Programmiersprache die Beliebtheit von Python in den Themen ML und Data Science [14, vgl. S.354]. Weitere Eigenschaften der Programmiersprache sind:

- Hohe Lesbarkeit durch einfache Syntax [14, vgl. S.354]
- Programme haben weniger Zeilen Code als vergleichbare Sprachen wie C [14, vgl. S.354]
- hohe Flexibilität durch dynamische Typisierung [14, vgl. S.354]
- Automatisches Speichermanagement [14, vgl. S.354]
- läuft auf vielen verschiedenen Betriebssystemen [14, vgl. S.355]

Die dynamische Typisierung sorgt zwar auf der einen Seite für eine erhöhte Flexibilität, da Variablen kein fester Typ bei der Deklarierung zugeordnet werden muss, liefert aber auf der anderen Seite auch Nachteile. Durch die dynamische Typisierung kann die Ausführung eines Programms zeitintensiver werden und bei größeren Projekten kann es dazu kommen, dass nicht mehr genau nachvollzogen werden kann, welchen Typ eine Variable hat [14, vgl. S.355]. Beide Nachteile sind bei der Größe dieses Projekts aber vernachlässigbar.

Ein großer Vorteil von Python sind die Programmierbibliotheken, die die Sprache besitzt und die importiert werden können. Für nahezu jeden Anwendungsfall existiert eine Bibliothek die Python um weitere Klassen und Funktionen erweitert und somit die Entwicklung von Anwendungen beschleunigt und erleichtert. Durch sie muss der Entwickler die Funktionalitäten, die eine Bibliothek mitbringt, nicht selber implementieren und kann stattdessen auf diese Bibliotheken zurückgreifen. In dieser Arbeit wurden drei Bibliotheken verwendet, die für das Erstellen von KI-Modellen, für die Datenverarbeitung und das Visualisieren von Daten genutzt wurden. Diese werden in den folgenden Kapiteln näher beschrieben.

Aufgrund der Beliebtheit der Sprache im Thema KI und den Programmierbibliotheken wird Python in dieser Arbeit dazu genutzt, die Daten aus dem Praxisprojekt zu importieren, zu verarbeiten und das KI-Modell zu erstellen und mit den Daten anzulernen. Eine Alternative dazu wäre die Programmiersprache R, jedoch wird

hier Python bevorzugt, da Python weiter verbreitet ist, eine simplere Syntax hat und somit einfacher zu lesen und zu verstehen ist.

3.2.1 Keras

Keras ist eine Bibliothek für die Programmiersprache Python, die Klassen und Funktionen liefert, um verschiedene DL-Modelle zu erstellen und diese anschließend zu trainieren. Operationen zur Berechnung und Bearbeitung von beispielsweise Tensoren bringt diese Bibliothek nicht mit. Stattdessen greift Keras auf andere Bibliotheken zurück, die diese Funktionalitäten mitbringen, und nutzt diese dann. Dabei ist Keras kompatibel zu mehreren Bibliotheken dieser Art, wie zum Beispiel TensorFlow von Google oder CNTK von Microsoft. Der Entwickler kann aussuchen, welche dieser Bibliotheken er verwenden will und kann diese auch während der Entwicklung wechseln [1, vgl. S.89ff.]. François Chollet empfiehlt standardmäßig die TensorFlow-Bibliothek zu nutzen, da diese „am weitesten verbreitet, skalierbar und ausgereift“ [1, S.91] sei. Was TensorFlow genau kann und wie es genutzt wird, ist für die Erstellung eines DL-Modells mit Keras nicht relevant und wird somit nicht genauer beschrieben.

Das Erstellen eines NN mithilfe von Keras folgt dabei in der Regel den folgenden vier Schritten:

1. Trainingsdatensatz definieren und in Ein- und Ausgabewerte aufteilen [1, vgl. S.92]
2. NN definieren, indem die einzelnen Schichten konfiguriert werden [1, vgl. S.92]
3. Verlustfunktion, Optimierer und Metrik(Kenngröße) auswählen [1, vgl. S.92]
4. Modell mit Trainingsdaten anlernen [1, vgl. S.92]

Ein großer Vorteil der Bibliothek ist, dass Keras bereits die verschiedenen Schichten als Klassen mitbringt, diese also nicht vom Entwickler erst definiert werden müssen. Der Entwickler kann dadurch einem Modell beliebig Schichten hinzufügen oder entfernen und diese nach seinen Wünschen konfigurieren. Die Anzahl an Neuronen innerhalb einer Schicht und die Aktivierungsfunktion der Neuronen in

der Schicht werden der Schicht einfach als Parameter übergeben. Ebenfalls muss die Aktivierungsfunktion nicht selbst definiert werden. Da reicht es aus, den Namen der Funktion als Parameter anzugeben, denn Keras hat bereits eine Vielzahl von gängigen Aktivierungsfunktionen implementiert. Zusätzlich hat der Entwickler noch die Möglichkeit eine eigene Aktivierungsfunktion zu definieren. Das Auswählen des Optimierers und der Metrik verlaufen ebenfalls genauso einfach. Diese werden, nachdem das Modell mit seinen Schichten definiert wurde, ganz simpel, wie bei der Auswahl der Aktivierungsfunktion, als Parameter einer Funktion übergeben. Genau diese Schritte werden in dieser Arbeit durchlaufen, um das KI-System zur Bewertung von Anwendungsregeln zu erstellen und anzulernen.

3.2.2 Pandas

Um unter anderem das Importieren der Daten, welche im Praxisprojekt gesammelt wurden, zu ermöglichen, wird die Datenverarbeitungsbibliothek Pandas genutzt. Pandas ist ein Akronym, welches für panel data steht. Diese Bibliothek basiert dabei auf Tabellen, ähnlich wie bei Excel, und bietet die Möglichkeit Excel-Dateien, CSV-Dateien und weitere Dateitypen direkt zu importieren oder zu exportieren. Die beiden wichtigsten Datenstrukturen sind dabei die Series und die Dataframes [15, vgl. S.253].

Series können dabei wie eine zweispaltige Tabelle verstanden werden. Die erste Spalte beinhaltet einen Index, dieser kann dabei beliebig sein, muss also nicht wie bei einem Array aus Integer-Werten bestehen. Diese Eigenschaft unterscheidet Series von Arrays und diese bieten dadurch die Möglichkeit beliebige Indizes zu nutzen und Daten somit als Key-Value-Paare zu speichern. Wird jedoch kein spezieller Index definiert, so besteht der Index, genau wie bei einem Array, aus aufsteigenden Integer-Werten von 0 bis zur Länge der Series. In der anderen Spalte werden die eigentlichen Werte gespeichert, diese müssen dabei alle vom selben Datentyp sein [15, vgl. S.254f.]. Weitere Eigenschaften der Datenstruktur Series sind:

- Indizierung [15, vgl. S.256]
  - über Index oder Liste von Indizes auf bestimmte Werte eines Series-Objekts zuzugreifen

- Filtern nach einer Bedingung, zum Beispiel nur auf Werte zugreifen, die größer als ein Schwellwert sind [15, vgl. S.256]
- Anwendung von mathematischen Funktionen auf gesamtes Series-Objekt möglich [15, vgl. S.256]

Dataframes sind eine weitere Datenstruktur, die die Pandas Bibliothek mitbringt. Ein Dataframe hat, wie ein Series-Objekt, ebenfalls Ähnlichkeiten zu einer Tabelle, dieses Mal jedoch mit einer unbegrenzten Anzahl an Spalten. Die Werte einer Spalte müssen vom selben Datentyp sein, jedoch können verschiedene Spalten auch verschiedene Datentypen besitzen. Da nun mehrere Spalten mit Werten vorhanden sein können, besitzen Dataframes sowohl einen Zeilen- als auch einen Spaltenindex. Die Indizes sind wieder beliebig wählbar. Zudem können mindestens zwei Series-Objekte zu einem Dataframe konkateniert werden [15, vgl. S.263f.]. Also kann über Dataframes gesagt werden, dass sie eine Datenstruktur sind, die aus mehreren einzelnen Series-Objekten bestehen.

Neben den beiden Datenstrukturen liefert die Pandas Bibliothek zahlreiche Funktionen zur Analyse, Bearbeitung und Verwaltung der gespeicherten Daten. Die in dieser Arbeit verwendeten Funktionen werden in Kapitel 4 an den gesammelten Daten aus dem Praxisprojekt vorgestellt und erläutert.

3.2.3 Matplotlib

Zur Visualisierung von Daten wird in dieser Arbeit die Bibliothek Matplotlib für Python genutzt. Matplotlib bietet die Möglichkeit verschiedenste Diagramme und Darstellungen, wie zum Beispiel Linien-, Balken-, Tortendiagramme und viele mehr, mit wenig Code zu erstellen. Die erstellten Diagramme können vom Entwickler zudem noch beliebig konfiguriert werden [15, vgl. S.167.]. Das Visualisieren der Daten sorgt für ein besseres Verständnis der Daten im Vergleich zu einer rein textuellen Beschreibung.

Matplotlib eignet sich vor allem in der Verwendung zusammen mit Pandas. Pandas listet Matplotlib als „optionale Abhängigkeit“, was bedeutet, dass für die Verwendung von Pandas Matplotlib nicht zwingend benötigt wird, aber es empfohlen wird [15, vgl. S.253]. Beide Datenstrukturen der Pandas-Bibliothek besitzen zudem

eine Plot-Funktion, also eine Funktion um ein Diagramm aus den Daten zu erstellen, die genutzt werden kann, wenn sowohl Pandas als auch Matplotlib als Bibliotheken importiert werden.

4 Datensatz

In dem Praxisprojekt, auf das diese Bachelorarbeit aufbaut, wurden Bewertungen von Anwendungsregeln aus vorherigen Projekten der Siemens Mobility GmbH gesammelt und in eine CSV-Datei geschrieben. Wie im Kapitel 3.2.1 beschrieben, ist der erste Schritt beim Erstellen eines DL-Modells als NN das Definieren des Datensatzes sowie die Aufteilung dessen in Eingabe- und Ausgabewerte. Dieses Kapitel wird sich mit diesem Schritt beschäftigen und den Datensatz so aufbereiten, dass er für das Anlernen des KI-Modells genutzt werden kann. Zudem werden in diesem Kapitel wichtige Eigenschaften des Datensatzes visualisiert.

4.1 Datensatz aus dem Praxisprojekt

Das Ergebnis aus dem Praxisprojekt war ein Datensatz mit 195.518 Bewertungen zu Anwendungsregeln. Beim Erstellen des Datensatzes wurde jedoch ein entscheidender Punkt nicht beachtet. Als die Daten aus den Projekten gesammelt wurden, wurde dabei auch jede Baseline eines Moduls berücksichtigt ohne zu überprüfen, ob sich an der Anwendungsregeln und ihrer Bewertung etwas geändert hat. Die Folge dessen war, dass der Datensatz eine Vielzahl von Duplikaten enthalten hat, was dazu geführt hätte, dass Projekte mit mehreren Baselines stärker ins Gewicht gefallen wären, obwohl die Anzahl an Baselines nichts über die Signifikanz der Bewertung aussagt. Deshalb mussten die erstellten Module mit den bewerteten Anwendungsregeln in DOORS überarbeitet werden.

Dafür wurde ein Skript in DXL geschrieben, was in den Modulen nach Duplikaten sucht und diese löscht. Die äußere Schleife durchläuft dazu alle Elemente in dem Ordner, in welchem sich die Module mit den bewerteten Anwendungsregeln befinden. Wenn ein Element ein Formal Module ist, dann wird dieses Modul geöffnet und in einer inneren Schleife werden alle Objekte des Moduls durchlaufen. Für

```
1     // ...
2     for it in f do{
3         if (type(it) == "Formal"){
4             m = edit(fullName(it), false)
5             for o in entire m do{
6                 szData = o."ObjectText""" o."Status""" o."
    Statement""";
7                 if(find(slUnique, szData)){
8                     softDelete(o);
9                 }else{
10                    put(slUnique, szData, szData)
11                }
12                szData = ""
13            }
14            purgeObjects_(m)
15            delete(slUnique)
16        // ...
```

Quellcode 4.1.1: Duplikate in Modulen löschen

jedes Objekt wird eine Zeichenfolge erstellt, die aus den Attributen ObjectText, Status und Statement besteht. Anschließend wird überprüft, ob diese Zeichenfolge in einer Skiplist bereits vorhanden ist. Wenn dies der Fall ist, dann wird das Objekt als Duplikat erkannt und mittels der softDelete()-Funktion als gelöscht gekennzeichnet. Wenn diese Zeichenfolge noch nicht in der Skiplist vorhanden ist, dann ist dieses Objekt noch einzigartig und wird der Skiplist hinzugefügt, um sicherzustellen, dass zukünftige Duplikate erkannt werden. Anschließend wird die Zeichenfolge geleert. Nachdem die innere Schleife durchlaufen wurde, werden alle Objekte, welche als gelöscht gekennzeichnet wurden, endgültig aus dem Modul entfernt. Zudem wird der Inhalt der Skiplist entfernt, bevor das nächste Modul durchlaufen wird, um zu gewährleisten, dass diese bei der nächsten Verwendung keine Elemente aus vorherigen Modulen mehr enthält.

Das Entfernen der Duplikate hatte zur Folge, dass der im Praxisprojekt gesammelte Datensatz von ursprünglich 195.518 auf 14.572 bewertete Anwendungsregeln reduziert wurde. Der Grund, weshalb der Datensatz auf rund 7,5 % seiner eigentlichen Größe geschrumpft ist, liegt darin, dass einige Projekte bis zu 40 Baselines hatten, wo sich aber die meisten Anwendungsregeln nicht verändert hatten. Dieser Schritt war wichtig, um dafür zu sorgen, dass bestimmte Bewertungen nicht stärker gewichtet werden, als andere.

Nun besteht der Datensatz aus 14.572 Einträgen, die jeweils aus drei Attributen, nämlich dem eigentlichen Text der Anwendungsregel sowie dem Status und dem Statement bestehen. Wie in der Abbildung 2.3 dargestellt, benötigt ein Modell Eingabedaten und die dazugehörigen Antworten. Die Eingabedaten stellen in diesem Fall die Texte dar, während die Antworten hier in Form des Status und des Statements dargestellt werden. Angenommen der Status einer Anwendungsregel X wird 15-mal mit closed und zehnmal mit open bewertet und ein Modell wird mit diesen Daten angelernt. Würde ein neues Projekt nun diese Anwendungsregel X importieren und einen Vorschlag vom Modell generieren lassen, dann würde das Modell eine Mehrheitsentscheidung durchführen und prüfen, wie oft die Anwendungsregel in der Vergangenheit mit welchem Status bewertet wurde. Das Modell würde aufgrund der Mehrheitsentscheidung diese Anwendungsregeln immer mit closed bewerten. Das wäre ein legitimer Ansatz um Vorschläge zu generieren. Dahinter steckt jedoch keine Struktur, die das Modell erkennen könnte, und somit auch keine Intelligenz. Dieses Problem könnte auch mit einer einfachen Tabelle gelöst werden, das Nutzen eines NN wäre hier unnötig. Daher benötigt der Datensatz noch weitere Eigenschaften, da die Bewertung von Anwendungsregeln nicht alleine durch die reine Anzahl an Bewertungen in der Vergangenheit prognostiziert werden kann. Beispielsweise spielen regionale Gegebenheiten eine große Rolle, da, wenn Projekte im selben Land durchgeführt werden, es wahrscheinlicher ist, dass sie Anwendungsregeln ähnlich bewerten.

Um an mehr Informationen über die vorherigen Projekte zu gelangen, muss das Skript aus dem Praxisprojekt erweitert werden. Neben dem Text der Anwendungsregel, dem Status und dem Statement muss auch das Produkt und die Version der Komponente berücksichtigt werden, von der die Anwendungsregel stammt. Ebenso spielt der Name des Projekts eine Rolle, da durch ihn in Kapitel 4.4 in einer Access-Datenbank nach Informationen zu dem Projekt gesucht werden kann. Das Produkt und die Version der Komponente lässt sich durch den Link bestimmen, den jedes Objekt auf das zentrale Projekt hat, in der die Anwendungsregeln gespeichert werden. Dieses Projekt heißt RA Application Conditions und ist dabei wie folgt aufgebaut:

\RA Application Conditions\XX_PG\Kategorie\Produkt...

```
1     // ...
2     for l in all o -> "*" do{
3         mnTarget = target(l)
4         if(fullName (getParentProject(mnTarget)) == fullName (
    project("RA Application Conditions"))){
5             szProduct = getProduct(target(l))
6             szVersion = target(l)
7             break;
8         }
9     }
10    // ...
```

Quellcode 4.1.2: Produkt- und Versionsbezeichnung bestimmen

Aus dem Zielobjekt des Links lässt sich das Modul bestimmen, in dem die jeweilige Anwendungsregel gespeichert ist. Das Modul trägt dabei als Namen die Versionsbezeichnung eines Produkts. Über den Speicherpfad lässt sich außerdem das Produkt, als das dritte Element des Pfades, bestimmen. Im DXL-Skript muss also eine weitere Schleife hinzugefügt werden. Diese Schleife soll alle ausgehenden Links eines Objekts durchlaufen, bis ein Link gefunden wurde, dessen Zielobjekt sich im Projekt RA Application Conditions befindet. Wenn so ein Link gefunden wurde, dann wird der vollständige Pfad an eine spezielle Funktion übergeben, die als Rückgabewert den Namen des Produkts hat. Außerdem wird die Versionsbezeichnung ebenfalls dem Link entnommen und in einer Variable gespeichert. Dabei ist zu beachten, dass die Funktion target(Link) überladen ist. Im ersten Fall wird eine Referenz auf das Modul zurückgegeben, auf das der Link zeigt. Diese Referenz beinhaltet den kompletten Pfad. Im zweiten Fall wird lediglich der Name des Zielmoduls zurückgegeben [13, vgl. S.391]. Da der Name des Moduls bereits die Produktbezeichnung beinhaltet, ist hier der volle Pfad nicht relevant.

Während des Praxisprojekts wurden alle bewerteten Anwendungsregeln nach Projekt und Komponente gruppiert. Dabei liegen alle bewerteten Anwendungsregeln eines Projekts jeweils in einem Ordner. Der Name dieses Ordners trägt den Namen des Projekts. Um nun an den Namen eines Projekts zu kommen, muss lediglich der Name des Ordners geprüft werden, in dem sich das aktuelle Modul befindet. Dies wird ermöglicht durch die getParentFolder()-Funktion, wie dem Quellcode 4.1.3 entnommen werden kann. In der Access-Datenbank steht vor dem Namen des Projekts noch ein Slash, weshalb dieser noch vor den Namen gesetzt werden muss.

```
    // ...
    for it in f do{
        if (type(it) == "Formal"){
            m = edit(fullName(it), false)
            Folder fParent = getParentFolder(it)
            String szPath = "/" name fParent
            // ...
```

Quellcode 4.1.3: Projektnamen bestimmen

Das Ergebnis dieser beiden Schritte ist eine CSV-Datei mit 14.572 Einträgen, die jeweils 6 Attribute (Text der Anwendungsregel, Produkt, Version, Pfad, Status, Statement) besitzen. Diese Datei kann nun mittels der Pandas-Bibliothek in ein Python-Skript importiert werden.

4.2 Importieren des Datensatzes

Um die Pandas-Bibliothek zu nutzen, muss diese zunächst importiert werden. Beim Importieren einer Bibliothek besteht die Möglichkeit dieser Bibliothek einen Alias zuzuweisen, wie in Zeile 1 des Quellcodes 4.2.1 zu erkennen ist. Für die Pandas-Bibliothek ist dabei pd als Alias gebräuchlich.

```
1   import pandas as pd
2   df = pd.read_csv('SAR_Data.csv')
3   df.shape
4   ---------------------------------------
5   Output:
6   (14572, 6)
```

Quellcode 4.2.1: Pandas und den Datensatz importieren

df.shape gibt hierbei die Anzahl der Zeilen und Spalten des Dataframes zurück. Das Tupel kann dabei als (Anzahl Zeilen, Anzahl Spalten) gelesen werden. An der Ausgabe können die 14.572 Zeilen und die 6 Attribute erkannt werden.

4.3 Datensatz auf Fehler prüfen

Nach dem Einlesen des Datensatzes sollte der Datensatz auf mögliche Fehler und Unregelmäßigkeiten überprüft werden. Die erste potenzielle Fehlerquelle können die verschiedenen Ausprägungen des Attributs Status sein. Alle erlaubten Ausprägungen nach dem Process Manual zu Anwendungsregeln der Siemens Mobility

GmbH können der Tabelle 2.1 entnommen werden. Um zu überprüfen, welche Ausprägungen das Attribut Status hat und wie oft jede Ausprägung auftritt, kann der Quellcode 4.3.1 genutzt werden. Dort wird eine Liste erstellt, welche alle einzigartigen Werte der Spalte Status beinhaltet. Im Anschluss wird eine Schleife definiert, die über all diese Werte iteriert. Bei jedem Durchgang wird der aktuelle Wert und die Anzahl an Zeilen, bei denen die Spalte Status den aktuellen Wert annimmt, auf der Konsole ausgegeben.

```
    status = df['Status'].unique()
    for x in status:
        print(x + " " + str(len(df[df['Status'] == x])))
    ----------------------------------------
    Output:
    In creation 773
    non applicable 4052
    closed 6772
    forwarded 1339
    open 111
    postponed 12
    compliant 1413
    partly closed 92
    partly open 8
```

Quellcode 4.3.1: Häufigkeit der Ausprägungen von Status bestimmen

An der Ausgabe des Codes wird deutlich, dass bei der Bewertung der Anwendungsregeln vom Process Manual abgewichen wurde. Um das zu beheben, müssen die Statuswerte gemapped werden. Dafür werden die Ausprägungen „In Creation“, „postponed“ und „partly open“ als „open“ definiert. Die Ausprägung „partly closed“ wird zu „closed“ geändert. Zudem wird die Schreibweise „non applicable“ in „not applicable“ abgeändert, um die Vorgaben des Process Manuals zu erfüllen.

Um die Häufigkeit der einzelnen Ausprägungen zu visualisieren wird der Quellcode 4.3.2 ausgeführt. Damit wird eine neue Liste erstellt, in der die Häufigkeiten der verschiedenen Statuswerte gespeichert werden. Im Anschluss wird wieder die Schleife aus dem Quellcode 4.3.1 durchlaufen. Dieses Mal werden die Häufigkeiten aber nicht auf der Konsole ausgegeben, sondern sie werden der neuen Liste hinzugefügt.

```
    status = df['Status'].unique()
    hauefigkeit = []
    for x in status:
        hauefigkeit.append(len(df[df['Status'] == x]))
    fig, ax = plt.subplots()
    ax.pie(hauefigkeit, labels=status, autopct='%1.1f%%')
```

Quellcode 4.3.2: Visualisierung des Attributs Status

Mit den Listen „status“ und „hauefigkeit“ kann nun ein Tortendiagramm mithilfe von Matplotlib erstellt werden. Dafür muss ein Plot erstellt werden und die beiden Listen müssen als Parameter an die pie()-Methode übergeben werden. Der letzte Parameter bietet die Möglichkeit das Format der Prozentangaben der einzelnen Tortensegmente anzugeben. In dem Fall hier werden die Prozentangaben auf eine Nachkommastelle gerundet. Die Abbildung 4.1 zeigt das erstellte Tortendiagramm an. Durch diese Abbildung wird zum Beispiel deutlich, dass fast die Hälfte der Anwendungsregeln in der Vergangenheit mit „closed“ bewertet wurden.

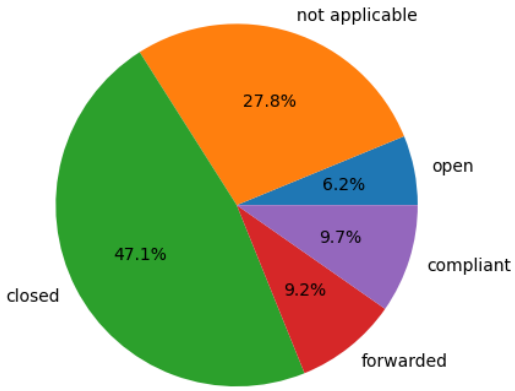

Abbildung 4.1: Tortendiagramm des Attributs Status

Als Nächstes werden mithilfe der unique()-Methode alle Ausprägungen der weiteren Attribute überprüft. Beim Quellcode 4.3.3 fallen zwei Ausprägungen auf, diese wurden zur Verdeutlichung in der Ausgabe rot markiert. Die Produktbezeichnung ist bei manchen Fällen „nan“ oder „/“ und stellen somit keine wirklichen Produkte dar. Um die Ursache dafür herauszufinden, ist es sinnvoll, sich die Zeilen anzusehen, in der diese Probleme auftreten. Dafür wird die loc()-Methode aus der Pandas-Bibliothek genutzt. Diese Methode wird genutzt, um gezielt Datenbereiche eines Dataframes auszuwählen. Zudem können mithilfe dieser Methode Datenoperationen auf den ausgewählten Daten ausgeführt werden. In Zeile zwei des Quellcodes 4.3.3 werden alle Zeilen ausgewählt, die als Produkt den Wert „nan“ besitzen. Dabei werden nur die Spalten „Product“, „Version“ und „Path“ausgewählt und zum Schluss wird die head()-Methode genutzt, um nur die ersten 5 Zeilen zurückzugeben. Die Ausgabe dieser Zeile kann der unteren Tabelle entnommen werden. In

dieser Tabelle wird deutlich, dass die Produktbezeichnung in der Version enthalten ist. Da die Werte der Spalte „Path“ sensible Unternehmensdaten sind, wurden diese hier entfernt.

```
1     print(df['Product'].unique())
2     df.loc[df['Product'].isna(), ['Product', 'Version', 'Path']].
      head()
3     ---------------------------------------
4     Output:
5     ['DCM' 'ACM300' 'Trainguard 200 RBC' 'SICAS ECC' 'SIMIS IS'
6     'ETCS Engineering Process' 'CG ETCS Workstation' 'AzS350U' '
      TGMT'
7     'ACM200' 'Clearguard TCM 100' 'VICOS OC111' 'LED_70_at_Som6'
8     'LED_Anzeigemodul' 'LEU S21 MS MC MT 208-233_-433_-533'
9     'Eurobalise S21 und S22' 'SIMIS W' 'VICOS NCU' 'LED_70' '
      LED_136'
10    'POM4 S700K BSG9' 'Key Switch' 'VICOS OC501' 'LZB700M' nan
11    'LEU S21 MS MC MT 208-203_-403_-503' 'Eurobalise S21' '/' 'DTS
      '
12    'SIMIS LC' 'AC100' 'LEU S21 M ME 208-201_-401']
13
14    Product          Version     Path
15        NaN  VICOS_S_D_02.14      ...
16        NaN  VICOS_S_D_02.14      ...
17        NaN  VICOS_S_D_02.14      ...
18        NaN  VICOS_S_D_03.00      ...
19        NaN  VICOS_S_D_03.00      ...
```

Quellcode 4.3.3: Ausprägungen des Attributs Product

Die Anwendungsregeln zum Produkt „VICOS_S_D“ liegen in der DOORS-Datenbank in Modulen unter dem Pfad: /RA Application Conditions/03_PG_OCS/Service and diagnostic systems. Dieser Pfad weicht von der Ordnerstruktur, die in Kapitel 4.1 beschrieben wurde, ab. Es fehlt hier der Ordner mit dem Namen des Produkts. Stattdessen enthält hier das Modul sowohl den Namen des Produkts als auch seine Version. Deshalb müssen die Einträge, die als Produkt den Wert „nan“ besitzen, überarbeitet werden.

```
1     df.loc[df['Version'].str.contains('VICOS_S_D'), 'Product'] = '
      VICOS_S_D'
2     df.loc[df['Version'].str.contains('VICOS_S_D'), 'Version'] =
      df['Version'].str[-5:]
```

Quellcode 4.3.4: Eintragen der korrekten Produkt- und Versionsbezeichnung

In den beiden Zeilen im Quellcode 4.3.4 werden alle Zeilen überarbeitet, deren Versionsbezeichnung „VICOS_S_D“ enthält. Alle gefundenen Zeilen bekommen als Produkt den korrekten Wert „VICOS_S_D“ zugeordnet. Die Versionsbezeichnung besteht aus den letzten 5 Zeichen der Spalte „Version“ aus der Tabelle im

Quellcode 4.3.3. Deshalb wird nur ein Teil der ursprünglichen Versionsbezeichnung beibehalten. str[-5:] sorgt nämlich dafür, dass nur die letzten 5 Zeichen der Zeichenkette genutzt werden.

Dasselbe Verfahren wird auch für die Fälle durchgeführt, die als Produkt den Wert „/“ besitzen. Dieser Sonderfall tritt bei 147 bewerteten Anwendungsregeln aus zwei ungarischen Projekten auf. Einige Anwendungsregeln stammen nicht aus dem Projekt RA Application Conditions, sondern vermutlich aus einem anderen Projekt oder sie sind speziell für diese beiden Projekte erstellt worden. Die ausgehenden Links der Objekte können über die grafische Oberfläche von DOORS verfolgt werden, bei diesen beiden Projekten wird das Zielobjekt aber aufgrund mangelnder Rechte nicht angezeigt, wie in Abbildung 4.2 gesehen werden kann.

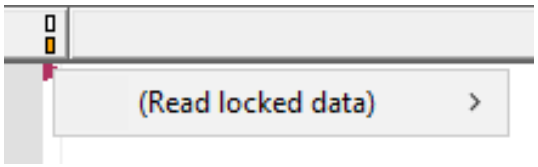

Abbildung 4.2: Link auf gesperrte Daten

4.4 Sammeln zusätzlicher Daten

Wie in Kapitel 4.1 bereits erwähnt, besteht die Möglichkeit über eine Access-Datenbank noch weitere Daten über die Projekte zu sammeln. Aus den Inhalten dieser Access-Datenbank wird jeden Tag eine XML-Datei erstellt, welche ebenfalls in das Python-Skript mit der Pandas-Bibliothek importiert werden kann. In dieser XML-Datei kann dann nach den Projekten aus dem Datensatz gesucht werden und der Datensatz kann dann mit den Daten aus der XML-Datei erweitert werden. Zunächst wird eine Liste erstellt, die alle eindeutigen Projekte aus dem Datensatz enthält. Diese Liste wird dann verwendet, um eine Schleife durchzuführen, in der jeder Projektname nacheinander durchlaufen wird. In der Schleife wird für jedes Projekt der Eintrag aus der Access-Datenbank gesucht. Dabei muss der Eintrag drei Bedingungen erfüllen:

1. „Type“ muss den Wert „Real“ besitzen

- Projekte mit dem „Type“ „Real“ befinden sich in der Projektabwicklungsphase

2. „Location“ muss den Wert „BWG“ besitzen
   - „BWG“ steht hierbei für die RM Datenbank Braunschweig
3. „Offset“ muss den Wert „path“ besitzen
   - stellt sicher, dass der Eintrag zum richtigen Projekt gehört
   - In einem Land existieren mehrere Projekte mit demselben Namen, im Datensatz ist damit jedoch ein bestimmtes Projekt gemeint, die Namen mussten hier anonymisiert werden

Das Prüfen dieser Bedingungen ist eingebettet in einen try-except-Block. Der try-except-Block sorgt dafür, dass eine Fehlermeldung auftritt, falls in der Access-Datenbank ein Eintrag nicht gefunden wird. Wenn hingegen ein Eintrag gefunden wird, dann wird die erste gefundene Zeile in einer Variable „result“ gespeichert. Quellcode 4.4.1 zeigt den Code, um diese Schritte durchzuführen.

```
1     paths = df['Path'].unique()
2     accessDB = pd.read_xml(".../Input_BWG_Combined_Access.xml")
3     for path in paths:
4         try:
5             if(path == "..."):
6                 result = accessDB.loc[(accessDB['Type'] == "Real")
7                 & (accessDB['Location'] == "BWG")
8                 & (accessDB['Offset'] == "...")].iloc[0]
9             else:
10                result = accessDB.loc[(accessDB['Type'] == "Real")
11                & (accessDB['Location'] == "BWG")
12                & ((accessDB['Offset'] == str(path)) | (accessDB['
    Offset'] == (str(path) + "/")))].iloc[0]
13        except:
14            print(str(path) + " has no entry in the AccessDB!")
```

Quellcode 4.4.1: Importieren der Access-Datenbank

„result“ enthält nun alle Attribute über das Projekt, die in der Access-Datenbank vorhanden sind. Diese relevanten Informationen müssen nun dem ursprünglichen Datensatz hinzugefügt werden. Noch in derselben Schleife wie im Quellcode 4.4.1, werden dem Datensatz neue Spalte hinzugefügt und mit den Werten aus der Access-Datenbank gefüllt. Quellcode 4.4.2 zeigt in Zeile drei ein Beispiel dafür. Durch diese Zeile werden alle Zeilen aus dem Datensatz aktualisiert, die in der „Path“-Spalte den aktuellen „path“-Wert haben. Der Wert der neuen Spalte wird dabei aus der entsprechenden Spalte von „result“ übernommen. Dieser Schritt wird ebenfalls

für die anderen Spalten aus der Access-Datenbank ausgeführt. Zum Schluss wird noch die Versionsbezeichnung angepasst. Es kann der Fall auftreten, dass zwei verschiedene Produkte dieselbe Versionsbezeichnung besitzen, zum Beispiel „2“ oder „01.5“. Um das zu verhindern, wird vor die Versionsbezeichnung noch der Name des Produkts gestellt, damit sich Versionsbezeichnungen von verschiedenen Produkten nicht mehr gleichen können.

```
    for path in paths:
        #...
        df.loc[df['Path'] == str(path), 'Project_category']
        = result['Project_category']
        #...
        df['ProductVersion']
        = df["Product"].str.cat(df["Version"], sep = "-")
```

Quellcode 4.4.2: Erweiterung des Datensatzes

Das Ergebnis dieser Schritte ist ein Datensatz in Form eines Dataframes mit 14.572 Zeilen und 11 verschiedenen Spalten. Diese zusätzlichen Daten wurden gesammelt, um Ähnlichkeiten von Projekten nutzen zu können. Je stärker sich Projekte ähneln, desto eher werden die Anwendungsregeln der Projekte ähnlich bewertet. Die Spalten dieses Dataframes sind die folgenden und dienen, bis auf die Spalte „Text“, als Datensatz für das Anlernen des KI-Modells:

Text	Text der Anwendungsregel
Product	Name des Produkts
ProductVersion	Version des Produkts
Project_name	Name des Projekts, das die Anwendungsregel bearbeiten musste
section	Sektion, die das Projekt durchführt, zum Beispiel ML für MainLine oder MT für MassTransit
Project_category	Kategorie des Projekts, gibt Aufschluss über die Größe eines Projekts
BS	Business Segment, in dem das Projekt angesiedelt ist
RU	Region, in der das Projekt durchgeführt wird
ProjectYear	Jahr, in dem das Projekt gestartet wurde
Status	Status der Anwendungsregel
Statement	Begründung zum Status

Eine weitere Möglichkeit zur Erzeugung weiterer Daten aus dem bestehenden Datensatz wäre die Data Augmentation. Bei diesem Verfahren werden die bestehenden Daten leicht abgeändert, damit weitere Daten synthetisch erstellt werden können, was zu einem größeren Datensatz führt. Diese Praxis wird häufig im Bereich der Computer Vision, also dem „maschinellen Sehen“, genutzt. Computer Vision beschäftigt sich mit der Verarbeitung von Bildern und Videos, weshalb dabei die Datensätze aus Bildern und Videos bestehen. Auf den Daten können dann verschiedene Transformationen ausgeführt werden, zum Beispiel kann ein Bild gespiegelt oder gedreht werden, um aus einem Bild mehrere zu erzeugen [1, vgl. S.184f.]. In dem in dieser Arbeit beschriebenem Anwendungsfall bietet sich Data Augmentation jedoch nicht an. Da der Datensatz überwiegend aus kategorischen Attributen besteht, müssten einige Kategorien andere Werte bekommen, um aus den vorhandenen Daten weitere zu erzeugen. Dies könnte zum Beispiel dazu führen, dass eine Zeile im Datensatz, die beispielsweise zu einem Projekt, das eigentlich der Kategorie „A“ zugehört, unterschiedlichen Kategorien zugeordnet wird, was dann aber zu einem Widerspruch im Datensatz führen könnte. Zudem könnten durch das Manipulieren des Datensatzes zufällige Strukturen und Zusammenhänge erzeugt werden, die in der Realität nicht existieren. Da der Datensatz relativ klein ist, könnte ein NN dann diese falschen Strukturen erkennen und diese lernen. Dies wäre kritisch zu betrachten, da Anwendungsregeln sicherheitsrelevant sein können. Deshalb wurde in dieser Arbeit davon abgesehen, dieses Verfahren zu verwenden.

4.5 Codierung der Attribute

Im nächsten Schritt muss der Datensatz in eine Form gebracht werden, die für das Anlernen eines KI-Modells geeignet ist. Die ausgewählten Attribute stellen alle Kategorien in Textform oder als Jahreszahl dar. Beide Darstellungsformen sind ungeeignet, um als Trainingsdatensatz zu dienen, da ein NN mit numerischen Werten arbeitet. Eine Möglichkeit zur Codierung von kategoriellen Attributen ist die One-Hot-Codierung. Dabei wird jeder Ausprägung eines Attributs eine eigene neue Spalte zugeordnet und für jede Zeile hat nur eine Spalte den Wert 1, alle anderen Spalten nehmen den Wert 0 an. Die beiden nachfolgenden Tabellen zeigen

anhand eines Beispiels die One-Hot-Codierung des Attributs „ProjectYear“.

Text	ProjectYear
Text1	2019
Text2	2021
Text3	2022

Tabelle 4.1: Ursprügliche Tabelle

Text	2019	2021	2022
Text1	1	0	0
Text2	0	1	0
Text3	0	0	1

Tabelle 4.2: Nach One-Hot-Codierung

Um auf den Spalten des Dataframes in Python die One-Hot-Codierung auszuführen, wurde eine neue Funktion geschrieben, die dem Quellcode 4.5.1 entnommen werden kann. Diese Funktion erhält als Parameter einen Dataframe sowie eine Liste an Spalten, auf denen die One-Hot-Codierung durchgeführt werden soll. In einer Schleife iteriert die Funktion über jede Spalte, die in der Liste übergeben wurde und prüft anschließend, ob diese Spalte im Dataframe vorhanden ist. Wenn dies der Fall ist, dann wird mithilfe der get_dummies()-Methode aus der Pandas-Bibliothek die One-Hot-Codierung durchgeführt. Anschließend wird die ursprüngliche Spalte aus dem Dataframe entfernt und dafür werden die neuen codierten Spalten dem Dataframe hinzugefügt. Dafür werden die drop()- und concat()-Methode der Pandas-Bibliothek genutzt, die es ermöglichen Spalten und Zeilen hinzuzufügen oder zu entfernen [16].

```
1     def column_one_hot (dataframe, columns):
2         for column in columns:
3             if column in dataframe:
4                 one_hot = pd.get_dummies(dataframe[column])
5                 dataframe = dataframe.drop(column,axis = 1)
6                 dataframe = pd.concat([dataframe, one_hot], axis
    =1)
7         return dataframe
```

Quellcode 4.5.1: Funktion zur One-Hot-Codierung

Die Attribute „Product“ und „ProductVersion“ stellen hier jedoch einen Sonderfall dar. Aufgrund der One-Hot-Codierung kann in jeder Zeile nur eine Ausprägung eines Attributs den Wert 1 annehmen, jedoch benutzen Projekte mehrere Produkte und verschiedene Produktversionen. Deshalb muss der Datensatz für diese beiden Attribute nochmal überarbeitet werden. Zunächst wird über alle Projekte und Produkte iteriert. Für jedes Produkt wird nun geprüft, ob es in einem Projekt vorhanden ist. Wenn dies der Fall ist, dann wird der Wert für dieses Produkt in jeder Zeile des

Projekts auf eins gesetzt. Äquivalent wird mit dem Attribut „ProductVersion“ verfahren.

```
    df = column_one_hot(df, ['Product'])
    for project in projects:
        for product in products:
            df.loc[df['Project_name'] == project, product] = 1 if
            (df.loc[df['Project_name'] == project][product].sum())
    >= 1 else 0
    df = column_one_hot(df, ['ProductVersion', 'Project_name', '
    section',
    'Project_category', 'BS', 'RU', 'ProjectYear'])
    ----------------------------------------
    Output:
    (14572, 180)
```

Quellcode 4.5.2: Anpassung der Spalte „Product“

Nachdem die beiden Attribute korrekt codiert wurden, können die restlichen Attribute ebenfalls codiert werden, wie im Quellcode 4.5.2 gezeigt wird. Aus den ursprünglichen 11 Spalten wurden nach der Codierung der Attribute 180 verschiedene Spalten.

Eine Variante der One-Hot-Codierung ist der One-Hot-Hashing-Trick, der sich anbietet, wenn die Anzahl an Attributen in einem Datensatz zu hoch ist. Anstatt jede Ausprägung eines Attributs als neue Spalte zu definieren, kann auch eine Hashfunktion genutzt werden, um die Ausprägungen abzubilden. François Chollet nutzt diese Variante der One-Hot-Codierung bei der Codierung von Wörtern und Zeichen. In seinem Beispiel codiert er die 1000 am häufigsten vorkommenden Wörter [1, vgl. S.236]. Da der Datensatz in dieser Arbeit aber nicht über so viele verschiedene Spalten verfügt, wurde diese Variante der One-Hot-Codierung hier nicht genutzt.

Neben den beiden Varianten existieren noch weitere Möglichkeiten zur Codierung von kategorischen Attributen. Ein weiteres Beispiel wäre das Label-Encoding. Dabei wird jeder Ausprägung eines Attributs ein eindeutiger Integer-Wert zugeordnet. Die nachfolgenden Tabellen zeigen ein Beispiel für das Label-Encoding eines Attributs. Dem Produkt „SICAS ECC“ wird dabei der Wert 0 zugeordnet, dem Produkt „LZB700m“ 1 und so weiter.

Text	Product
Text1	SICAS ECC
Text2	LZB700M
Text3	SIMIS IS

Tabelle 4.3: Ursprügliche Tabelle

Text	Product
Text1	0
Text2	1
Text3	2

Tabelle 4.4: Nach Label-Encoding

Im Vergleich zur One-Hot-Codierung besteht hier der Vorteil, dass nicht für jede Ausprägung eine neue Spalte hinzugefügt wird und die Dimension des Datensatzes somit nicht größer wird. Der entscheidende Nachteil des Label-Encodings tritt jedoch auf, wenn die Ausprägungen nicht in einer Beziehung zueinander stehen. Durch das Zuweisen von aufsteigenden Integer-Werten erweckt diese Art der Codierung den Eindruck, dass die Beziehung „SICAS ECC“ < „LZB700m“ < „SIMIS IS“ besteht, obwohl das gar nicht der Fall ist. Deshalb wurde in dieser Arbeit das Label-Encoding verworfen und stattdessen die One-Hot-Codierung genutzt.

4.6 Aufteilung des Trainingsdatensatzes

Nachdem die Attribute codiert wurden und der Datensatz somit in eine Form gebracht wurde, die zum Anlernen eines KI-Modells geeignet ist, muss nun als Nächstes eine Aufteilung in Trainings- und Testdaten sowie in Eingabe- und Ausgabewerte erfolgen. Da die Bewertung einer Anwendungsregel unabhängig von der Bewertung anderer Anwendungsregeln ist, darf das KI-Modell nur mit den Bewertungen zu der spezifischen Anwendungsregel angelernt werden. Wie Abbildung 4.3 jedoch zeigt, wurden nur 38 Anwendungsregeln mehr als zehnmal bewertet, die allermeisten Anwendungsregeln wurden hingegen weniger als fünfmal bewertet. Das sorgt dafür, dass beim KI-Modell mit Überanpassung gerechnet werden muss, da der Datensatz zu klein ist.

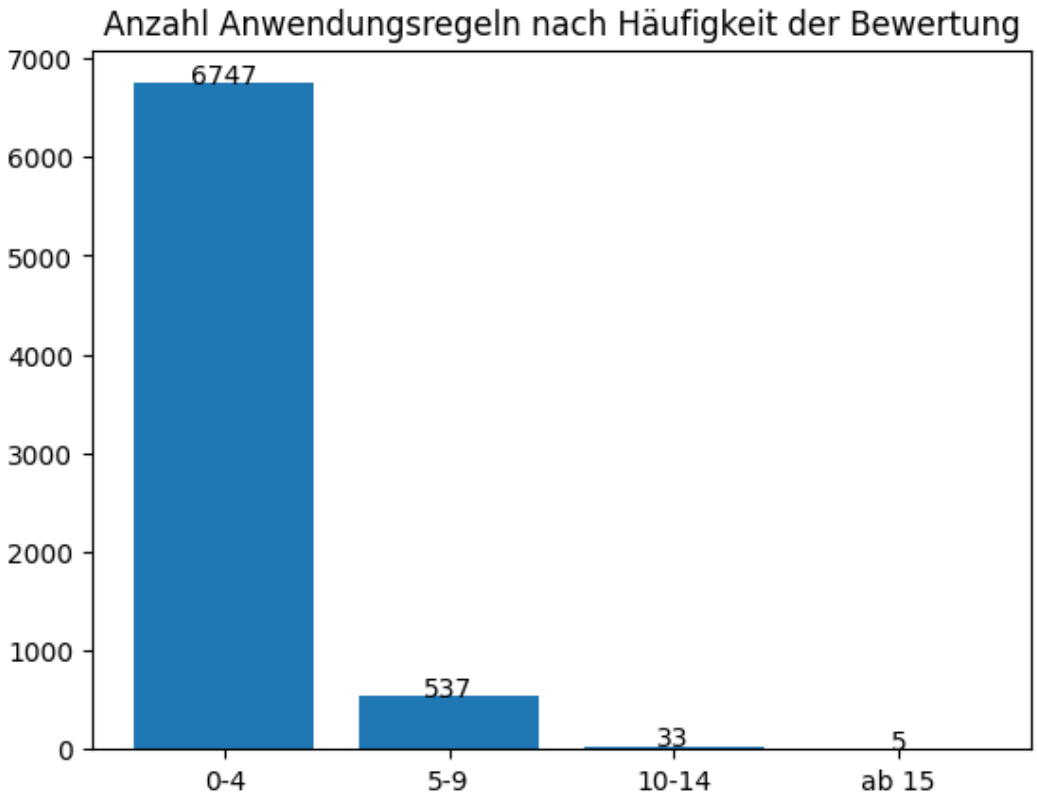

Abbildung 4.3: Säulendiagramm zur Häufigkeit der Bewertung von Anwendungsregeln

Um den Datensatz auf die Bewertungen von Anwendungsregeln zu beschränken, welche gerade bewertet werden soll, wurde zur Demonstration dieses Schritts beispielhaft die Anwendungsregel ausgewählt, welche am häufigsten im Datensatz vorhanden war. Der Quellcode 4.6.1 zeigt den benötigten Code-Ausschnitt, um diese Schritte durchzuführen.

```
1 df_training = df
2 text = "Zur Anschaltung des Antriebes in der Außenanlage muessen
     Signalkabel nach VDE 0816/2 oder Kabel mit vergleichbaren
     Eigenschaften verwendet werden. Die Verlegevorschriften des
     Kabels sind einzuhalten."
3 df_training = df_training.loc[df_training['Text'] == text]
4
5 trainX = drop_columns(df_training, ['Status', 'Text', 'Statement'
     ])
6 trainYStatus = drop_column(column_one_hot(df_training[['Text', '
     Status']], ['Status']), "Text")
7 trainYStatement = drop_column(column_one_hot(df_training[['Text',
     'Statement']], ['Statement']), "Text")
```

Quellcode 4.6.1: Definieren des Trainingsdatensatzes

Zeile 2 des Quellcodes 4.6.1 speichert den beispielhaft ausgewählten Text einer Anwendungsregel in eine Variable. Bei der späteren Anwendung würde dieser Text aus dem Modul in DOORS extrahiert werden. In der nächsten Zeile wird der Datensatz auf die Zeilen beschränkt, die als Text den ausgewählten Text besitzen. Die letzten drei Zeilen werden für die Aufteilung in Eingabe- und Ausgabewerte benötigt. Alle Spalten, bis auf die beiden Ausgabewerte „Status“ und „Statement“ sowie der Text der Anwendungsregel, stellen die Eingabewerte dar. Die beiden Ausgabewerte mussten in jeweils eigene Dataframes aufgeteilt werden, der Grund dafür wird in Kapitel 5.1.1 beschrieben.

Nun sind die Daten korrekt codiert und aufgeteilt in einen Trainingsdatensatz mit getrennten Ein- und Ausgabewerten. Der Testsplit wird beim Anlernen definiert, näheres dazu ebenfalls in Kapitel 5.1.1.

5 Modellierung

Dieses Kapitel wird sich mit der Erstellung eines KI-Modells als NN mithilfe der Keras-Bibliothek in Python beschäftigen. Dabei werden die verschiedenen Möglichkeiten zur Modellierung des Modells vorgestellt, erläutert und diskutiert. Zum Schluss wird gezeigt, wie so ein Modell verwendet werden kann und wie gut es seine Aufgabe lösen kann.

5.1 Definieren des ersten neuronalen Netzes

Nachdem der Trainingsdatensatz definiert und in Ein- und Ausgabewerte aufgeteilt wurde, muss als Nächstes das NN definiert werden, wie es Kapitel 3.2.1 beschriebt. Dafür bietet die Keras-Bibliothek die Klasse „Sequential“, die genutzt werden kann, wenn Schichten sequentiell angeordnet werden sollen. Laut Chollet ist diese Art der Netzarchitektur die am häufigsten genutzte [1, vgl. S.92].

```
1    from keras.models import Sequential
2    from keras.layers import Dense
3    from keras.losses import CategoricalCrossentropy
4
5    model = Sequential()
6    model.add(Dense(8, input_shape=(trainX.shape[1],), activation=
     'relu'))
7    model.add(Dense(16, activation='relu'))
8    model.add(Dense(trainYStatus.shape[1], activation='softmax'))
```

Quellcode 5.1.1: Erstellung eines sequentiellen Modells

Der Quellcode 5.1.1 zeigt die benötigten Imports und wie ein Modell zur Vorhersage des Statuswerts definiert werden kann. Zuerst wird ein Objekt vom Typ Sequential erstellt. Diesem Objekt können dann die verschiedenen Schichten hinzugefügt werden. Dabei muss bestimmt werden, wie viele Schichten das Modell haben soll und wie viele Neuronen jede Schicht besitzen soll. Als erste Schicht wird eine Dense-Layer als Eingabeschicht mit acht Neuronen definiert. Eine Dense-Layer

ist eine fully-connected Schicht, wie beispielsweise die Schichten in der Abbildung 2.4, und erwartet als Parameter mindestens die Anzahl an Neuronen. Optional können noch weitere Parameter angegeben werden. Die Eingabeschicht benötigt als Besonderheit noch zusätzlich die Anzahl an Attributen, diese können dem Datafrane mit den Eingabewerten entnommen werden. Als Aktivierungsfunktion wird die ReLU-Funktion genutzt, die bereits in Kapitel 2.2.2 vorgestellt wurde und auch die verbreitetste Aktivierungsfunktion beim DL ist [1, vgl. S.102]. Als erste versteckte Schicht wird eine weitere Dense-Layer genutzt, hier mit 16 Neuronen. Die Anzahl an Neuronen und die Tiefe des Modells bestimmt die Komplexität des Modells. Ein zu komplexes Modell kann zu Überanpassung und ein zu simples Modell zu Unteranpassung führen. Im Vorhinein ist es schwierig die benötigte Komplexität zu bestimmen, weshalb die Anzahl an Schichten und Neuronen, in Abhängigkeit zu dem Ergebnis des Modells, im Laufe der Erstellung noch geändert werden sollten. Als letzte Schicht und somit als Ausgabeschicht wird ebenfalls eine Dense-Layer verwendet. Die Anzahl an Neuronen in der Ausgabeschicht ist abhängig von der Anzahl an möglichen Ausprägungen des zu bestimmenden Wertes. Die ausgewählte Anwendungsregel wurde in der Vergangenheit mit „closed“, „compliant“ und „not applicable“ bewertet, weshalb in diesem Fall die Ausgabeschicht drei Neuronen besitzen muss. Jedes Neuron stellt dabei eine mögliche Ausprägung dar. Anders als bei den vorherigen Schichten wurde bei dieser Schicht als Aktivierungsfunktion „Softmax“ gewählt. Diese Aktivierungsfunktion weist jedem Neuron eine Wahrscheinlichkeit zwischen null und eins zu, wobei die Summe aller Wahrscheinlichkeiten eins ergibt [17]. Diese Eigenschaft der Funktion ist auch der Grund, weshalb für Status und Statement eigene Modelle erstellt werden müssen, da das Sequential-Modell nur eine Ausgabeschicht erlaubt und es somit nicht möglich ist, mit dieser Funktion zwei Kategorien vorherzusagen.

Die Auswahl der Aktivierungsfunktion der Ausgabeschicht ist abhängig von der Aufgabe des Modells. „Softmax“ bietet sich zum Beispiel ideal als Möglichkeit zur Klassifikation an, wenn mehrere verschiedene Klassifizierungen vorhanden sind. Wenn nur zwei verschiedene Zustände möglich sind, also ein binäres Klassifikationsproblem vorliegt, würde sich als Aktivierungsfunktion eine Sigmoidfunktion

```
model.summary()
---------------------------------------
Output:
Model: "sequential"


_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 1424

dense_1 (Dense)              (None, 16)                144

dense_2 (Dense)              (None, 3)                 51


=================================================================
Total params: 1,619
Trainable params: 1,619
Non-trainable params: 0

_________________________________________________________________
```

Quellcode 5.1.2: Zusammenfassung des Modells

anbieten, da diese beliebige Werte einen Ausgabebereich zwischen null und eins zuordnet [1, vgl. S.100].

Eine Zusammenfassung wurde mit der summary()-Methode im Quellcode 5.1.2 ausgegeben. Diese Zusammenfassung zeigt nochmal die verschiedenen Schichten, die Dimension der Ausgabe der Schichten sowie die Anzahl an Parametern in jeder Schicht an. Zu erkennen sind dort die drei hinzugefügten Dense-Layer. Die erste Dimension der Ausgabe ist „None“, da die Anzahl an Einträgen vorher nicht festgelegt wurde. Die zweite Dimension wird bestimmt durch die Anzahl an Neuronen in einer Schicht.

Je nach Anwendungsfall würden andere Schichten als die Dense-Layer infrage kommen. Stehen die Daten in einem sequentiellen Zusammenhang, beispielsweise eine Zeitreihe von Wetterdaten, dann würden die sogenannten LSTM-Layer infrage kommen. LSTM steht für Long Short-Term-Memory(auf Deutsch: langes Kurzzeitgedächtnis) und diese Art von Schicht ist in der Lage Informationen mehrere Zeitschritte lang zu erhalten [1, vgl. S.260]. Bei der Computer Vision werden häufig CNNs (Convolutional Neuronal Networks) genutzt. Ein CNN besteht dabei nicht aus Dense-Layern, wie in diesem Anwendungsfall, sondern aus Convolutional-

Layern. Diese Schichten können zum Beispiel lokale Muster in Bildern erlernen und diese in neuen Bildern wiedererkennen, auch wenn sie nicht an derselben Stelle sind [1, vgl. S.164]. Diese Schichten werden also in anderen Anwendungsfällen genutzt, für diesen Anwendungsfall eignen sich jedoch die Dense-Layer am besten.

Bevor das Modell mit den Daten angelernt werden kann, müssen zunächst noch die Verlustfunktion und ein Optimierer ausgewählt werden. Da der Aufgabentyp eine Single-Label-Mehrfachklassifizierung ist, also eine Klassifizierung wo eine Klasse aus mehreren Klassen gewählt werden muss, wird als Verlustfunktion die kategorische Kreuzentropie genutzt. Eine Kreuzentropie misst die Differenz zwischen den vorhergesagten Wahrscheinlichkeiten und dem tatsächlichen Wert [1, vgl. S.102]. Bei der kategorischen Kreuzentropie wird erwartet, dass die Ausgabewerte in der One-Hot-Codierung vorliegen. Sind die Ausgabewerte mit dem Label-Encoding codiert, müsste als Verlustfunktion die „Sparse Categorical Crossentropy“ genutzt werden [17]. Für andere Aufgabentypen werden andere Verlustfunktionen genutzt. Zum Beispiel bei einer Regression bietet sich der mittlere quadratische Fehler an oder bei der Binärklassifizierung die binäre Kreuzentropie [1, vgl. S.155]. Während des Trainings wird das Modell versuchen das Ergebnis der kategorischen Kreuzentropie zu minimieren.

Als Optimierer soll, laut Chollet, in den meisten Fällen der „rmsprop“-Optimierer mit der voreingestellten Lernrate verwendet werden können [1, vgl. S.155]. Ähnlich wie bei der Komplexität des Modells ist es schwierig vorher genau zu bestimmen, welcher Optimierer für die spezifische Anwendung die besten Ergebnisse liefert. Deshalb muss auch hier ausprobiert werden. Zunächst wird sich an die Empfehlung von Chollet gehalten, es existieren jedoch auch weitere Optimierer, wie zum Beispiel „Adam“ oder „SGD“, die berücksichtigt werden sollten [17].

Die Verlustfunktion sowie der Optimierer können nun, wie im Quellcode 5.1.3 gezeigt, ausgewählt werden. Diese werden der compile()-Methode als Parameter übergeben. Zudem muss noch eine Kenngröße ausgewählt werden, die während des Anlernens überwacht wird. In diesem Fall wird dafür die „categorical_accuracy“ verwendet. Diese Kenngröße gibt an, wie oft Vorhersagen den One-Hot codierten Ausgabewerten entsprechen.

```
    model.compile(optimizer='rmsprop',
        loss=CategoricalCrossentropy(),
        metrics=['categorical_accuracy'])
```

Quellcode 5.1.3: Auswahl des Optimierers sowie der Verlustfunktion

Nun ist das Modell fertig konfiguriert und kann auf den Trainingsdaten trainiert werden.

5.1.1 Training des Modells

Zum Trainieren des Modells mit den vorher definierten und codierten Daten wird die fit()-Methode genutzt. Dieser Methode werden die Ein- und Ausgabewerte übergeben, mit dem das Modell trainiert werden soll. Weitere Parameter sind:

batch_size Anzahl an Trainingsbeispielen, bevor die Gewichte des NN geupdatet werden [17]

epochs Anzahl an Iterationen über den gesamten Trainingsdatensatz [17]

verbose Bestimmt die Menge an Terminalausgaben [17]

validation_split Anteil des Trainingsdatensatzes, der zum Testen benutzt werden soll [17]

Quellcode 5.1.4 zeigt, wie hier die Parameter definiert wurden. In diesem ersten Trainingsansatz wurde als „batch_size“ zwei gewählt, was bedeutet, dass das Modell nach jedem zweiten Trainingsbeispiel die Gewichte des Modells ändert. Da die ausgewählte Anwendungsregel lediglich 17-Mal bewertet wurde, muss die „batch_size“ dementsprechend niedrig gewählt werden. Je niedriger die „batch_size“ ist, desto genauer kann das Modell die Trainingsdaten erlernen, was jedoch auch zu Overfitting führen kann. Zudem kann ein niedrigerer Wert die Laufzeit des Trainingsprozesses negativ beeinflussen, da das Modell häufiger geupdatet werden muss.

Die Anzahl an zu durchlaufenden Epochen wurde zunächst mit 50 gewählt. Hier muss ebenfalls im Nachhinein geprüft werden, wie viele Epochen benötigt werden. Eine höhere Anzahl an Epochen kann zu Overfitting führen, eine zu niedrige Anzahl zu Underfitting. Hier muss also mithilfe der Kenngröße geprüft werden, wie sich das Modell mit zunehmender/abnehmender Anzahl an Epochen verhält.

```
    history = model.fit(trainX, trainYStatus,
                        batch_size=2,
                        epochs=50,
                        verbose=2,
                        validation_split=0.25)
```

Quellcode 5.1.4: Trainieren des Modells

Für den Parameter „verbose“ wurde der Wert zwei gewählt. Dieser sorgt dafür, dass die maximale Menge an Terminalausgaben ausgegeben wird. Ein Wert von eins würde nur einen Fortschrittsbalken ausgeben, ein Wert von null würde gar keine Ausgabe produzieren [17].

Da ein Datensatz niemals mit den Trainingsdaten getestet werden sollte, muss eine Aufteilung in Testdaten erfolgen. Wie in Kapitel 4.6 erwähnt, kann der Testsplit während des Anlernens erfolgen. Dafür bietet Keras die Möglichkeit automatisch einen Teil der Trainingsdaten zum Testen zu benutzen. In diesem Fall werden 25 % des Datensatzes als Testdaten verwendet. Bei 17 Einträgen wären das 4,25 Einträge, die aber aufgerundet werden, also werden fünf Einträge nicht zum Trainieren benutzt, sondern werden fürs Testen zurückgehalten.

5.1.2 Testen des Modells

Die Ausgabe des Trainingsprozesses kann dabei der Abbildung 5.1 entnommen werden. Da der „verbose“-Wert auf zwei gesetzt wurde, wird für jede Epoche eine Ausgabe generiert. Diese Ausgabe beinhaltet die benötigte Zeit für jede Epoche sowie den Wert der Verlustfunktion und der gewählten Kenngröße für die Trainings- und Testdaten.

```
Epoch 1/50
6/6 - 2s - loss: 1.0931 - categorical_accuracy: 0.3333 - val_loss: 1.0934 - val_categorical_accuracy: 0.4000 - 2s/epoch - 373ms/step
Epoch 2/50
6/6 - 0s - loss: 1.0761 - categorical_accuracy: 0.5000 - val_loss: 1.0859 - val_categorical_accuracy: 0.4000 - 63ms/epoch - 10ms/step
```

Abbildung 5.1: Ausgabe des Trainingsprozesses

Der Wert der Verlustfunktion sowie die Genauigkeit können auch zusätzlich visualisiert werden, um den Verlauf darzustellen. Dazu wird das Objekt genutzt, welches bei Aufruf der fit()-Methode im Quellcode 5.1.4 zurückgegeben und in der Variable „history“ gespeichert wurde. Dieses Objekt beinhaltet den Wert der Verlustfunktion und die Genauigkeit für die Trainings- und Testdaten. Mithilfe der

```
1     plt.plot(history.history['loss'], label = 'Training loss')
2     plt.plot(history.history['val_loss'], label = 'Validation loss
      ')
3     plt.title('Wert der Verlustfunktion')
4     plt.xlabel('Epochen')
5     plt.legend()
```

Quellcode 5.1.5: Trainieren des Modells

Matplotlib-Bibliothek können diese Werte dann in einem Liniendiagramm visualisiert werden, der Quellcode 5.1.5 zeigt die benötigten Schritte für das Erstellen eines Liniendiagramms für die Verlustfunktion, die Genauigkeit kann jedoch äquivalent dazu dargestellt werden. Neben der Übergabe der Daten und der dazugehörigen Label können auch ein Titel und eine Achsenbeschriftung hinzugefügt werden, wie Zeile drei und vier des Quellcodes zeigen. Abbildungen 5.2 und 5.3 zeigen die beiden erstellten Diagramme. Dort ist zu erkennen, dass mit zunehmender Anzahl an Epochen der Wert der Verlustfunktion abnimmt, während die Genauigkeit zunimmt, was ein idealer Fall wäre. Die Werte sind jedoch mit Vorsicht zu betrachten, da sie nur ein Beispiel für ein Modell darstellen. Da der Datensatz sehr klein ist, kann es sein, dass zufällig Testdaten ausgewählt wurden, die das Modell leichter erkennen konnte. Bei erneuter Ausführung könnten die Werte deshalb stark schwanken.

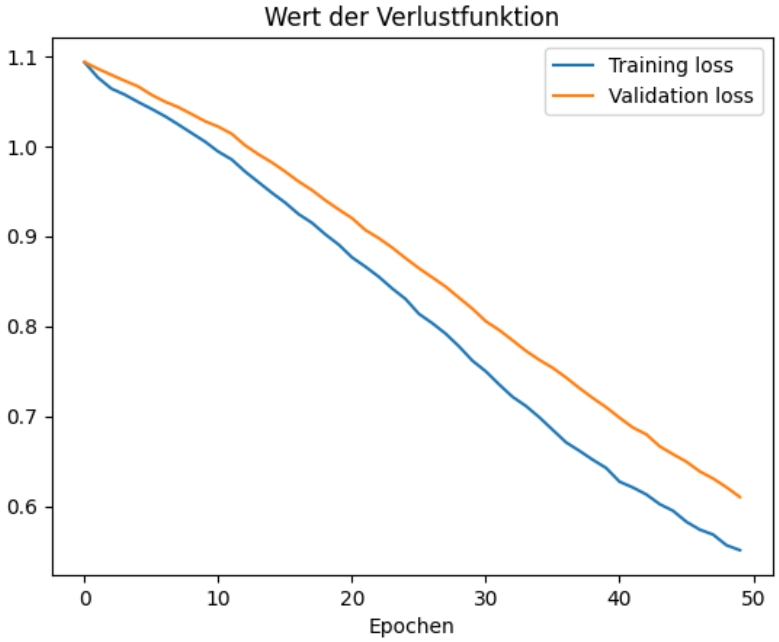

Abbildung 5.2: Plot der Verlustfunktion

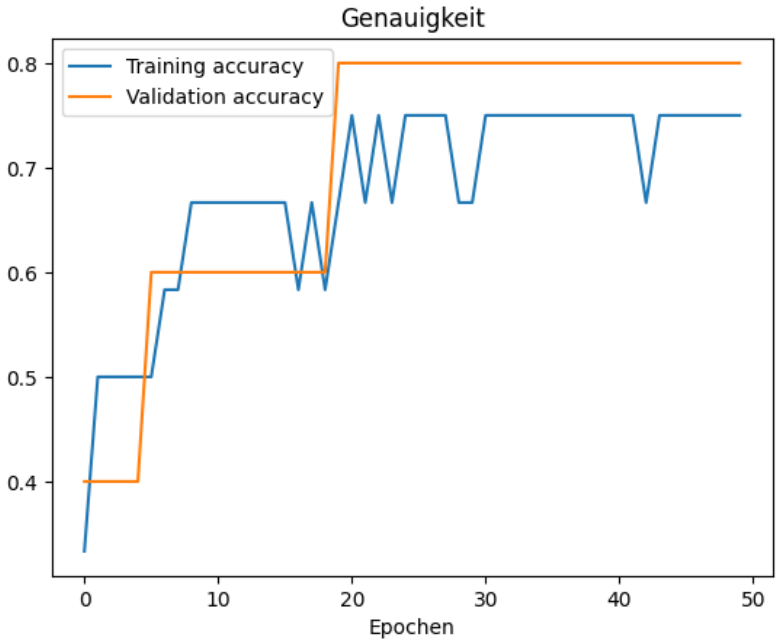

Abbildung 5.3: Plot der Genauigkeit

5.1.3 K-Cross-Validierung

Eine Möglichkeit zur besseren Beurteilung eines Modells wäre die k-cross-Validierung. Bei diesem Verfahren werden die Daten in k Teilmengen aufgeteilt und k Modelle erstellt. Jedes Modell wird mit k-1 Teilmengen trainiert, die letzte Teilmenge wird zum Testen verwendet [1, vgl. S.121f.]. Abbildung 5.4 zeigt schematisch das Verfahren für k=3. Das endgültige Ergebnis ist definiert als der Durchschnitt der einzelnen Teilergebnisse der Modelle.

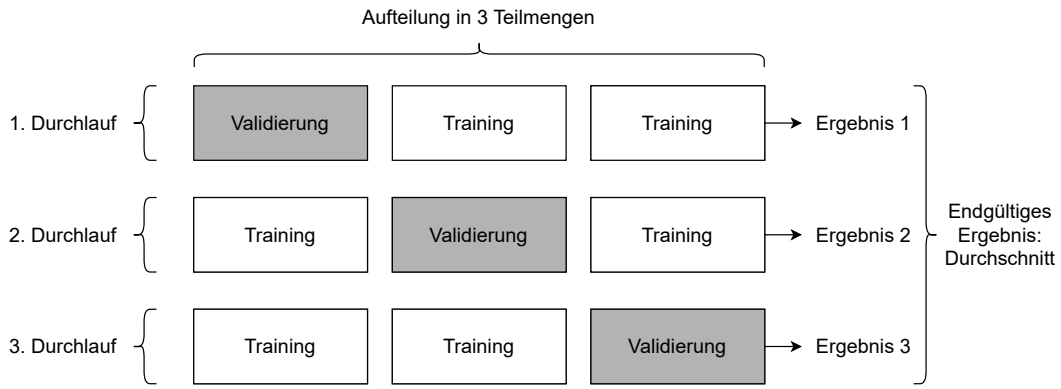

Abbildung 5.4: k-cross-Validierung [1, vgl. S.122]

```
    num_epochs = 250
    k = 4
    for i in range(k):
        start = int(i/k * len(df_training))
        end = int((i/k + 0.25) * len(df_training))
        val_data = df_training.iloc[start:end]
        train_data = df_training.drop(range(start,end))
        #...
        history = model.fit(trainX, trainYStatus,
                            batch_size=2,
                            epochs=num_epochs,
                            verbose=2,
                            validation_data=(valX, valYStatus))
        all_val_loss.append(history.history['val_loss'])
        all_loss.append(history.history['loss'])
        all_acc.append(history.history['categorical_accuracy'])
        all_val_acc.append(history.history['
    val_categorical_accuracy'])
```

Quellcode 5.1.6: Aufteilung des Datensatzes in Teilmengen

Um eine k-cross-Validierung durchzuführen, müssen nacheinander Modelle erstellt werden, die mit verschiedenen Teilmengen des Datensatzes trainiert und getestet werden. Die dafür benötigten Schritte werden im Quellcode 5.1.6 gezeigt. Dafür wird als Erstes definiert, wie viele Modelle und Teilmengen genutzt werden sollen, hier wird sich für vier entschieden. Danach wird eine Schleife viermal durchlaufen und bei jeder Iteration wird der Trainings- und Testdatensatz neu definiert. Der Testdatensatz besteht dabei aus 25 % des Datensatzes, in der ersten Iteration aus den ersten 25 %, bei der zweiten die nächsten 25 % und so weiter. Der Trainingsdatensatz beinhaltet die restlichen Daten. Innerhalb der Schleife werden danach die Daten wieder in Ein- und Ausgabewerte aufgeteilt, dieses Mal müssen auch die Testdaten entsprechend aufgeteilt werden. Danach kann wieder das Modell definiert und angelernt werden. Da der Testdatensatz nun selber definiert wurde, kann nicht mehr auf die automatische Aufteilung zurückgegriffen werden. Jetzt müssen die Testdaten, aufgeteilt in Ein- und Ausgabewerte, der Methode übergeben werden. Zudem wurde hier die Anzahl an Epochen auf 250 erhöht, damit beobachtet werden kann, wie das Modell sich bei steigender Anzahl an Epochen verhält. Das zurückgegebene Objekt der fit()-Methode wird genutzt, um die vier verschiedenen Werte jeweils in Listen speichern zu können, damit der Verlauf wieder visualisiert werden kann.

Der Wert der Verlustfunktion sowie die Genauigkeit müssen noch gemittelt werden. Dafür wird, wie der Quellcode 5.1.7 zeigt, der Durchschnitt der vier Kenngrößen pro Epoche gebildet. Mit den Werten können im nächsten Schritt wieder Diagramme zur Visualisierung erstellt werden.

```
1    avg_val_loss = [np.mean([x[i] for x in all_val_loss])
2                    for i in range(num_epochs-1)]
3    avg_loss     = [np.mean([x[i] for x in all_loss])
4                         for i in range(num_epochs-1)]
5    avg_acc      = [np.mean([x[i] for x in all_acc])
6                         for i in range(num_epochs-1)]
7    avg_val_acc  = [np.mean([x[i] for x in all_val_acc])
8                         for i in range(num_epochs-1)]
```

Quellcode 5.1.7: Mitteln der Ergebnisse [1]

Die Abbildungen 5.5 und 5.6 zeigen das Ergebnis der k-cross-Validierung. Bei der Verlustfunktion wird deutlich, dass der Wert der Verlustfunktion auf den Testdaten bis ca. 80 Epochen leicht abnimmt, danach aber ansteigend, während der Wert der Verlustfunktion auf den Trainingsdaten abnimmt und gegen ca. 0,2 konvergiert. Das Ansteigen auf den Testdaten und Abnehmen auf den Trainingsdaten ist ein Zeichen dafür, dass das Modell die Trainingsdaten auswendig lernt und die Generalisierungsfähigkeit des Modells darunter leidet und somit Overfitting auftritt.

Das Diagramm zur Genauigkeit zeigt, dass mehr Epochen nicht automatisch ein besseres Ergebnis liefern. Auf den Testdaten ändert sich die Genauigkeit ab der fünfzigsten Epoche nicht mehr wesentlich und stagniert ab Epoche 150. Im Vergleich zur Genauigkeit in der Abbildung 5.3 ist der Durchschnitt der Genauigkeit von vier Modellen geringer. Daran kann erkannt werden, dass die Testdaten bei 5.3 für das Modell günstig waren und das Modell auf diesen Daten gute Leistungen bringen konnte, was jedoch nicht so viel über die Leistung auf anderen Daten aussagt, da der Datensatz zu klein ist. Gerade in diesem Fall bietet sich die k-cross-Validierung sehr an und bietet wesentlich verlässlichere Kenngrößen als das Testen mit einem Modell auf einem Testdatensatz [1, vgl. S.123].

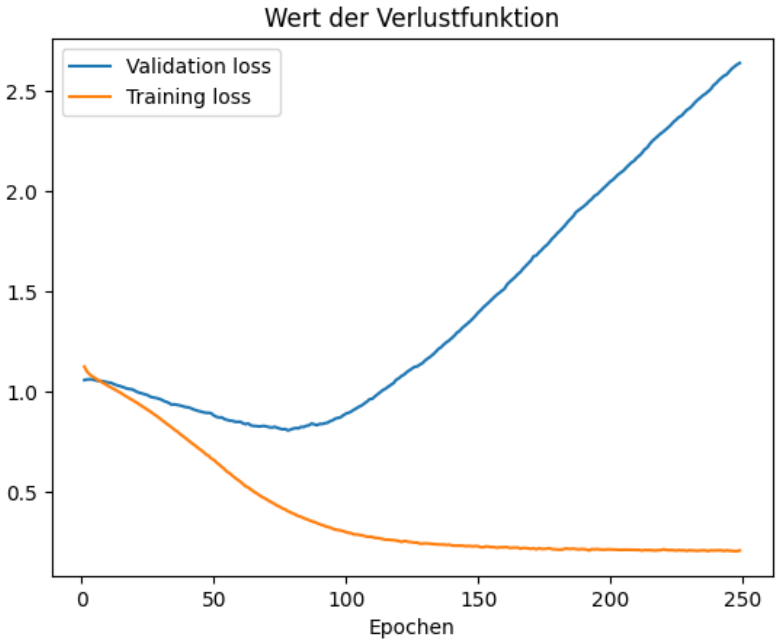

Abbildung 5.5: Verlustfunktion nach k-cross-Validierung

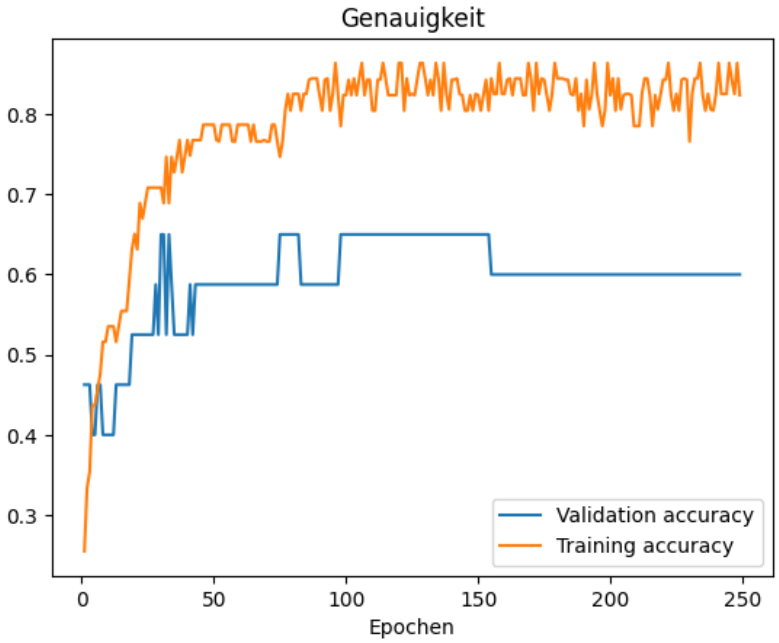

Abbildung 5.6: Genauigkeit nach k-cross-Validierung

5.2 Verbesserungen des ersten Modells

In Kapitel 5.1 wurde gezeigt, wie ein Modell für die Vorhersage für den Statuswert aussehen kann. Diese Schritte müssten ebenfalls für das dazugehörige Statement durchlaufen werden. Das Ergebnis wären zwei unabhängige Modelle, die beide jeweils einzeln und nacheinander trainiert werden müssen. Diese Aufteilung ist nötig gewesen, da die „Sequential“-Klasse es nicht erlaubt mehrere Ausgabeschichten zu definieren und da die „softmax“-Funktion allen Ausgabeneuronen einen Wert zuordnet, die addiert eins ergeben. Um dieses Problem zu umgehen, kann statt der „Sequential“-Klasse von Keras, ein Application Programming Interface (API), nämlich die funktionale Keras-API verwendet werden. Dadurch wird es ermöglicht Modelle zu definieren, die mehr als eine Eingabe- oder Ausgabeschicht haben oder Verzweigungen zwischen den Layern besitzen [1, vgl. S.299f.]. Abbildung 5.7 zeigt den Aufbau eines NN mit zwei Ausgabeschichten. Genau so ein Modell wird im nächsten Schritt mit der funktionalen Keras-API erstellt.

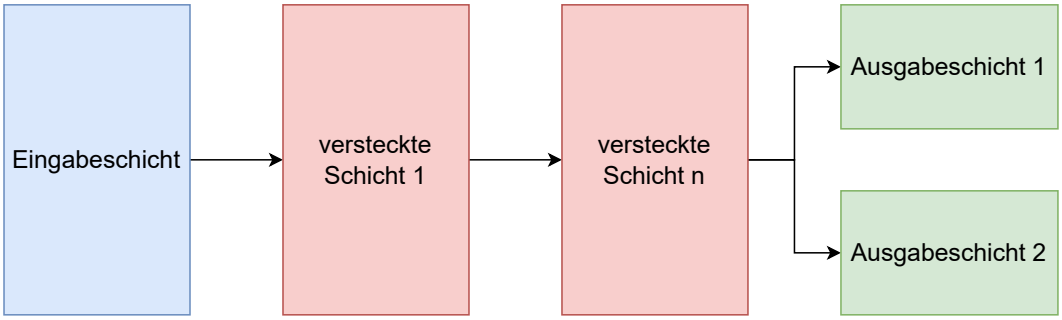

Abbildung 5.7: Modell mit zwei Ausgabeschichten

Grundsätzlich ist es möglich jedes Modell, welches mit der „Sequential“-Klasse erstellt wurde, in ein Modell mit der funktionalen Keras-API zu übersetzen. Deshalb wird als Erstes das bestehende und in Kapitel 5.1 beschriebene Modell übersetzt und im Anschluss daran wird dem Modell eine zweite Ausgabeschicht hinzugefügt. Mit der funktionalen API werden Tensoren direkt bearbeitet und können den Schichten, wie bei einer Funktion, übergeben und entgegengenommen werden. Ein Tensor ist eine Datenstruktur, welche als n-dimensionales-Array beschrieben werden kann. Skalare sind Tensoren der Stufe 0, Vektoren sind Tensoren der Stufe 1, Matrizen sind Tensoren der Stufe 2 und so weiter [12, vgl. S.128]. Ziel dabei ist es aus ei-

nem Eingabetensor einen Ausgabetensor zu erzeugen. Dafür ruft die Bibliothek alle Schichten ab, die an dieser Transformation beteiligt sind, und fasst diese Struktur dann als Modell zusammen. Der Ausgabetensor entsteht also durch aufeinanderfolgende Transformationen des Eingabetensors [1, vgl. S.305]. Quellcode 5.2.1 zeigt die Definition des Modells aus Kapitel 5.1 mit der funktionalen API und kann mit Quellcode 5.1.1 verglichen werden.

```
1  input = Input(shape=(trainX.shape[1],))
2  x = Dense(8, activation='relu')(input)
3  x = Dense(16, activation='relu')(x)
4  output = Dense(trainYStatus.shape[1], activation='softmax')(x)
5  model = Model(input, output)
6  model.summary()
7  ---------------------------------------
8  Output:
9
10 _________________________________________________________________
11 Layer (type)                 Output Shape              Param #
12 =================================================================
13 input_9 (InputLayer)         [(None, 177)]             0
14
15 dense_36 (Dense)             (None, 8)                 1424
16
17 dense_37 (Dense)             (None, 16)                144
18
19 dense_38 (Dense)             (None, 3)                 51
20
21 =================================================================
22 Total params: 1,619
23 Trainable params: 1,619
24 Non-trainable params: 0
   _________________________________________________________________
```

Quellcode 5.2.1: Modell mit funktionaler API darstellen

Anders als bei der Erstellung des Modells mit der „Sequential“-Klasse, wird hier die Dimension der Eingabedaten nicht in der ersten Schicht als Parameter übergeben, sondern wird noch vorher festgelegt. Wie in der Zusammenfassung des Modells zu sehen ist, stellt „Input“ jedoch keine wirkliche erste Schicht dar, da sie keine Parameter besitzt. Bis auf diesen Unterschied ist die Zusammenfassung beider Modelle gleich. Der Prozess des Trainierens und die Auswahl der Verlustfunktion sowie des Optimierers sind ebenfalls identisch. Deshalb sind beide Modelle in der Anwendung äquivalent. Nun muss noch eine weitere zusätzliche Ausgabeschicht hinzugefügt werden, welche das Statement zur Bewertung der Anwendungsregel prognostizieren soll. Dafür werden Zeile vier und fünf des Quellcodes 5.2.1 überarbeitet, was in Quellcode 5.2.2 gezeigt wird.

```
1    output1 = Dense(trainYStatus.shape[1], activation='softmax',
     name='status')(x)
2    output2 = Dense(trainYStatement.shape[1], activation='softmax'
     , name='statement')(x)
3    model = Model(inputs=input, outputs=[output1, output2])
```

Quellcode 5.2.2: Zweite Ausgabeschicht hinzufügen

Beide Ausgabeschichten erhalten als Parameter die Anzahl an möglichen Ausprägungen ihres Zielattributs sowie eine Aktivierungsfunktion. An dieser Stelle wäre es möglich auch verschiedene Aktivierungsfunktionen auszuwählen, sollte beispielsweise ein Zielattribut das Ergebnis einer Regression sein, könnte hier auch „mse“ gewählt werden. Zudem erhalten die beiden Layer noch einen eindeutigen Namen. Beim Aufruf der compile()-Methode besteht dadurch die Möglichkeit den beiden Schichten unterschiedliche Verlustfunktionen zuzuweisen [1, vgl. S.308f.]. Da beide Layer hier jedoch eine Single-Label-Mehrfachklassifizierung lösen sollen, wird das nicht benötigt.

Das Übersetzen der beiden Modelle mit der „Sequential“-Klasse in ein Modell mit der funktionalen API hat den Vorteil, dass nun nur noch ein Modell erstellt und trainiert werden muss, was die Laufzeit des Modells halbiert und zudem Zeilen an Code spart.

5.2.1 Wahl des Optimierers

Wie in Kapitel 5.1 beschrieben, ist es schwierig vorher zu bestimmen, welcher Optimierer die beste Performance liefert. Die Herangehensweise um den idealen Optimierer zu finden ist daher Trial-and-Error. Dafür wird mit einer k-cross-Validierung der mittlere Wert der Verlustfunktion für den Statuswert über mehrere Epochen und mit verschiedenen Optimierern visualisiert und anschließend geprüft, welcher Optimierer den niedrigsten Wert liefert. Es werden die Optimierer „Adam“, „RMSprop“ und „SGD“ ausprobiert. Hier wird nur der Statuswert betrachtet, da das Statement bei den meisten Anwendungsregeln einzigartig ist und somit selten eine genaue Übereinstimmung vorhanden ist. Deshalb ist beim Statement die Genauigkeit in der Regel null und darum ist der Wert der Verlustfunktion für das Statement nicht aussagekräftig.

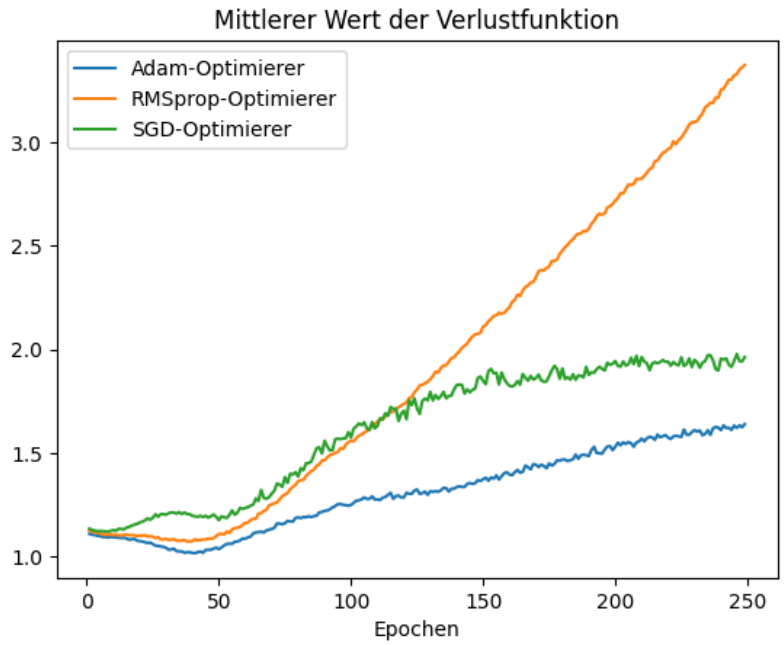

Abbildung 5.8: Vergleich verschiedener Optimierer

Das Diagramm in der Abbildung 5.8 zeigt den Verlauf des mittleren Werts der Verlustfunktion über mehrere Epochen. Dort ist zu erkennen, dass der Adam-Optimierer insgesamt den niedrigsten Wert erreicht, weshalb dieser im Folgenden genutzt wird. Jedoch unterscheiden sich die Werte mit den verschiedenen Optimierern bis zur ca. Epoche 50 nicht stark, weshalb davon auszugehen ist, dass die Wahl des Optimierers in diesem spezifischen Anwendungsfall keine allzu große Rolle spielt.

Neben der Auswahl des Optimierers kann ebenfalls die Lernrate eine Rolle spielen. Die Lernrate ist ein Parameter, der den Optimierer beeinflusst, weshalb verschiedene Lernraten ausprobiert werden sollten. Standardmäßig beträgt die Lernrate des Adam-Optimierers 0,001 [17]. Als weitere Lernraten werden 0,01 und 0,0001 gewählt, um zu prüfen, wie sich der Wert der Verlustfunktion ändert, wenn eine größere bzw. kleinere Lernrate gewählt wird. Die Lernrate des Optimierers kann bei Aufruf der compile()-Methode festgelegt werden, wie der Quellcode 5.2.3 zeigt.

```
1    model.compile(optimizer=optimizers.Adam(learning_rate=0.0001),
2        loss    ={'status': CategoricalCrossentropy(),
3                  'statement': CategoricalCrossentropy()},
4        metrics=['categorical_accuracy'])
```

Quellcode 5.2.3: Wahl der Lernrate

Abbildung 5.9 zeigt den Verlauf der Verlustfunktion mit den drei verschiedenen Lernraten. Deutlich zu erkennen ist, dass ein Vergrößern der Lernrate zu einem wesentlich schlechteren Ergebnis führt. Die standardmäßige sowie die kleinere Lernrate ähneln sich in ihrem Verlauf, wobei die standardmäßige Lernrate ein tieferes Minimum erreicht, weshalb der Wert der Lernrate in diesem Fall nicht verändert werden sollte.

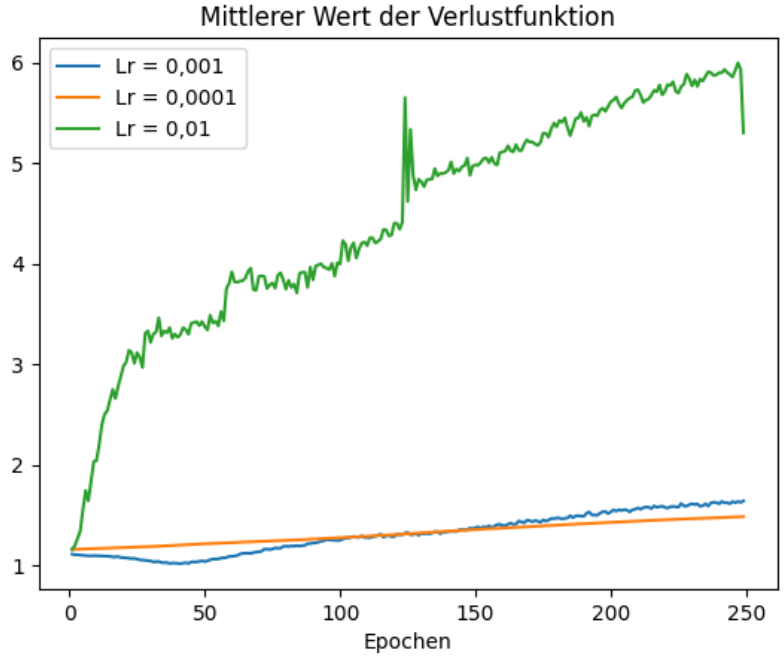

Abbildung 5.9: Vergleich verschiedener Lernraten

5.2.2 Anpassen der Modellarchitektur

Nachdem mehrere Optimierer ausprobiert und sich für einen entschieden wurde, sollte im nächsten Schritt noch geprüft werden, ob durch eine Anpassung der Modellarchitektur gegebenenfalls ein besseres Ergebnis erzielt werden kann. Ähnlich wie bei der Wahl des Optimierers im vorherigen Schritt, wird dies wieder durch Ausprobieren und anschließendes Visualisieren entschieden. Dabei wird die Anzahl an Schichten sowie die Anzahl an Neuronen in einer Schicht angepasst. Zudem wird geprüft, ob mittels einer Dropout-Regularisierung ein besseres Ergebnis erzielt werden kann. Folgende Modellarchitekturen werden dargestellt:

M1 Modell mit zwei versteckten Schichten (32, 16 Neuronen)

M2 breiteres und tieferes Modell mit vier Schichten (256, 128, 64, 32 Neuronen) mit Dropout-Regularisierung

M3 breiteres und tieferes Modell mit vier Schichten (256, 128, 64, 32 Neuronen) ohne Dropout-Regularisierung

M4 Modell mit einer versteckten Schicht mit 16 Neuronen

M5 Modell mit einer versteckten Schicht mit 512 Neuronen

Ein Beispiel für eine Dropout-Regularisierung zeigt der Quellcode 5.2.4. Die Dropout-Klasse erwartet als Parameter einen Wert zwischen null und eins. Dieser Wert stellt den Anteil an Neuronen dar, dessen Ausgabe während des Trainings auf null gesetzt wird.

```
1     x = Dense(256, activation='relu')(inputs)
2     x = Dropout(0.2)(x)
3     x = Dense(128, activation='relu')(x)
```

Quellcode 5.2.4: Dropout-Regularisierung mit Keras

Der Verlauf des mittleren Wertes der Verlustfunktion wird in Abbildung 5.10 für alle fünf Modelle dargestellt. Zu erkennen ist, dass die beiden Modelle mit der geringsten Anzahl an Neuronen (M1 und M4) die besten Ergebnisse erzielen, während die beiden Modelle mit der höchsten Komplexität (M2 und M3) die schlechtesten Ergebnisse erzielen. Der Grund dafür ist vermutlich, dass die komplexeren Modelle zu komplex sind um das Problem zu lösen. Zudem wird der kleine Datensatz ebenfalls dafür sorgen, dass simplerer Modelle in diesem Anwendungsfall bessere Leistungen erzielen. Auch die Dropout-Regularisierung bringt in diesem Anwendungsfall kein besseres Ergebnis. Das Minimum des Modells M1 lässt sich mit der argmin()-Methode der Numpy-Bibliothek bestimmen. Bei Epoche 23 hat M1 sein globales Minimum. Deshalb sollte dieses Modell weiterhin genutzt werden und beim Trainieren des Modells sollte als Anzahl an Epochen ein Wert um 23 gewählt werden, um ein bestmögliches Ergebnis in diesem Anwendungsfall zu erreichen.

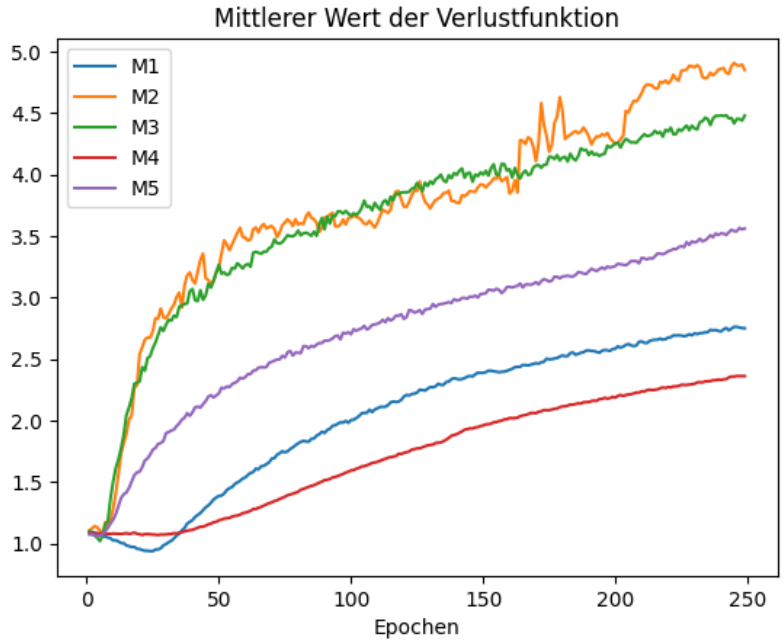

Abbildung 5.10: Vergleich verschiedener Modellarchitekturen

5.3 Vorhersagen treffen

Da nun ein Modell definiert und mit Daten angelernt wurde, sollen mögliche Vorhersagen zur Bewertung von Anwendungsregeln getroffen werden. Um die Anwendung des Modells vorzustellen, wurde vor dem Trainingsprozess eine Zeile aus dem Trainingsdatensatz entfernt. Das Modell soll nun zu dieser Zeile einen Vorschlag zur Bewertung abgeben. Zum Treffen von Vorhersagen kann die Methode predict() der Keras-Bibliothek benutzt werden. Mit dieser Methode können einem trainierten Modell neue Daten gegeben werden. Anhand dieser Daten erstellt das Modell dann eine Vorhersage. Quellcode 5.3.1 zeigt, wie so eine Vorhersage getroffen werden kann. Da das Modell zwei Ausgabeschichten besitzt, gibt diese Methode auch zwei Objekte zurück. Diese Objekte sind Listen, welche die Wahrscheinlichkeiten der einzelnen Ausprägungen beinhalten. Über diese Liste kann iteriert werden, um diese Wahrscheinlichkeiten auszugeben. Für den Statuswert wurden alle möglichen Ausprägungen ausgegeben, da dort nur drei Verschiedene vorhanden sind. Aufgrund der Übersichtlichkeit wird für das Statement nur der Text ausgegeben, der die höchste Wahrscheinlichkeit besitzt.

```
    prediction_status, prediction_statement = model.predict(test)

    for val in prediction_status:
        for col in range(len(val)):
            print (str(trainYStatus.columns[col]) + " " +
                   '{:.1%}'.format(val[col]))

    index_max = np.argmax(prediction_statement)
    for val in prediction_statement:
        print (trainYStatement.columns[index_max] + " " +
               '{:.1%}'.format(val[index_max]))
```

Quellcode 5.3.1: Vorhersage über neue Daten treffen

Die Ausgabe für den Statuswert sieht wie folgt aus:

Real compliant

Prediction closed 2.0 %

compliant 88.2 %

not applicable 9.8 %

Vergleicht man diese Ausgabe mit der Verteilung der Statuswerte der beispielhaft bewerteten Anwendungsregel in Abbildung 5.11, erkannt man eindeutig, dass das Modell eine Struktur erkennt und in der Lage ist, den Statuswert einer Anwendungsregel vorherzusagen.

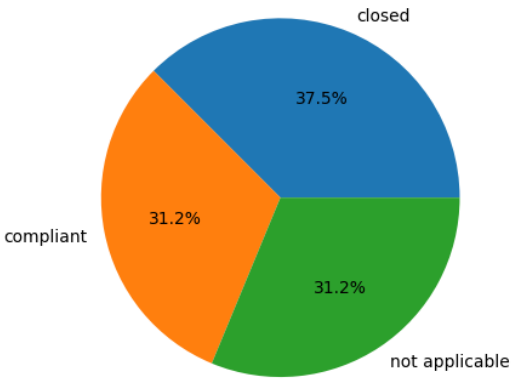

Abbildung 5.11: Verteilung des Statuswerts der betrachteten Anwendungsregel

Das dazugehörige Statement wurde folgendermaßen prognostiziert:

Real Im Projekt ... sind die Signalkabel nach VDE0816/2 verwendet. Die Verlegevorschriften des Kabels sind beim Verlegen eingehalten. Daher ist diese Auflage erfüllt. Bitte sieh Kabelspezifi-

kation (Aussenkabel) A6Z00038004325 - Outdoor Cable Plan A6Z00037954383 -

Prediction Zur Anschaltung der Weichen in der Außenanlage kommen Signalkabel nach VDE 0816/2 oder Kabel mit vergleichbaren Eigenschaften zum Einsatz. Siehe auch Dokument Requirement for Signalling cable G68167-S0100-U-5009 A6Z08110839230 01. 20.8 %

Wie bereits erwähnt sind die meisten Statements einzigartig, hier wurde jedoch ein Statement gewählt, welches vom Inhalt her identisch ist. Anhand der beiden vorhergesagten Attribute kann schlussfolgert werden, dass das Modell in der Lage ist eine Struktur beim Bewerten von Anwendungsregeln zu erkennen und Ähnlichkeiten in den Projekten festzustellen. Das war jedoch nur ein Beispiel für eine bewertete Anwendungsregel. Im Anschluss gilt noch dieses Modell zu testen, im Idealfall mithilfe eines größeren Datensatzes.

6 Fazit

6.1 Ausblick

Literaturverzeichnis

[1] François Chollet, *Deep Learning mit Python und Keras*. mitp, 2018, vol. 1.

[2] Daniel Sonnet, *Neuronale Netze kompakt : Vom Perceptron zum Deep Learning*. Springer, 2022, vol. 1.

[3] IREB(International Requirements Engineering Board), "Wörterbuch der Requirements Engineering Terminologie," 2022.

[4] Siemens Mobility GmbH, *RM Process Manual - Project Execution*, 2021.

[5] IEEE(The Institute of Electrical and Electronics Engineers), *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[6] Elizabeth Hull, Ken Jackson, Jeremy Dick, *Requirements Engineering*. Springer, 2005, vol. 2.

[7] ISO/IEC/IEEE International Standard, "29148-2018 - Systems and software engineering – Life cycle processes – Requirements engineering," 2018.

[8] The Standish Group, "The CHAOS Report," 1994.

[9] Siemens Mobility GmbH, *RM Process Manual - Application Rules*, 2022.

[10] OpenAI, "Introducing ChatGPT," https://openai.com/blog/chatgpt, accessed: 2023-03-02.

[11] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[12] Huawei Technologies Co., Ltd, *Artificial Intelligence Technology*. Springer, 2023.

[13] IBM, *The DXL Reference Manual*, https://www.ibm.com/docs/en/SSYQBZ_9.5.0/com.ibm.doors.requirements.doc/topics/dxl_reference_manual.pdf, 2012, accessed: 2023-03-01.

[14] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[15] B. Klein, *Numerisches Python*. Carl Hanser Verlag GmbH & Co. KG, 2019.

[16] "Pandas Documentation," https://pandas.pydata.org/docs/, accessed: 2023-03-15.

[17] "Keras API reference," https://keras.io/api/, accessed: 2023-03-17.