Hochschule Darmstadt

Fachbereich Mathematik und Naturwissenschaften & Informatik

Das Testen und Test-Driven Development in der Entwicklung von Conversational AI

Abschlussarbeit zur Erlangung des akademischen Grades Master of Science (M. Sc.) im Studiengang Data Science

vorgelegt von Alexander Kniesz
Mat.-Nr.: 740916

Referent: Frau Prof. Dr. Bettina Harriehausen-Mühlbauer

Korreferent: Frau Prof. Dr. Jutta Groos

Ausgabedatum: 3.2.2021
Abgabedatum: 28.7.2021

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.
Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.
Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.
Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 28.7.2021

_____

Alexander Kniesz

Zusammenfassung

Das Thema dieser Arbeit ist die Erläuterung und Anwendung der agilen Softwareentwicklung mit dem Test-Driven Development (TDD) für die Entwicklung einer Conversational AI.
Diese Arbeit wurde im Rahmen eines Projekts der Firma INNOQ erstellt, daher wird der Schwerpunkt auf die Anwendung mit dem Test-Driven Development und einer den Voraussetzungen entsprechende Implementierung des Anwendungsfalls gelegt. Die Anpassung von Softwareentwicklungsmethoden an Machine Learning (ML) ist ökonomisch sinnvoll, da ML immer häufiger in Anwendungen Verwendung findet (vgl. [1]).
Die Entwicklung von Software stellt oft einen großen finanziellen Aufwand für Unternehmen dar. Daher besteht in der Softwareentwicklung ein Bedarf an standardisierten Vorgehensweisen, wie dem TDD, da diese eine simple, strukturierte und zügige Bearbeitung der Projekte erleichtern. Die Problematik hierbei ist im Wesentlichen, dass das TDD für die Entwicklung von gleichbleibenden, eher deterministischen Systemen ausgelegt ist. Dem gegenüber stehen die ML-Systeme, welche durch die Notwendigkeit des regelmäßigen Trainierens der Modelle eine sich verändernde und statistische Natur aufweisen (vgl. [2]).
Für die Entwicklung einer Conversational AI wurde sich entschieden, da dies zum einen von der praktischen Anwendung nahegelegt ist und zum anderen bietet die Entdeckung von Transformern mit einem Aufmerksamkeitsmechanismus eine kürzliche, sehr interessante Neuerung in diesem Bereich (vgl. [3]). Des Weiteren stellt der Einsatz von Conversational AI durch die Automatisierung des Kundensupports eine große finanzielle Erleichterung für Unternehmen dar (vgl. [4]).
In dieser Arbeit wird daher eine Vorgehensweise des TDD vorgestellt, um eine Conversational AI zu implementieren. Hierbei müssen sowohl die Voraussetzungen an die Conversational AI als auch die Unterschiede und Besonderheiten des Machine Learnings beachtet werden, sodass das TDD zu einer validen Lösung führt. Bei der Vorgehensweise wurde, unter Berücksichtigung der Unterschiede, ein Fokus darauf gelegt, die Struktur des TDD zu erhalten, um somit auch möglichst viele der Vorteile bei der Entwicklung beizubehalten.

Abstract

The topic of this thesis is the explanation and application of agile software development with Test-Driven Development (TDD) for the development of a conversational AI.
This thesis was written within the framework of a project in the company INNOQ. Therefore, the focus lies on the application of the Test-Driven Development and the implementation of the use case that meets the requirements. The adaptation of software development methods to Machine Learning (ML) makes economic sense, as ML is increasingly used in applications (cf. [1]). The development of software often incurs a large financial expense for companies. Due to the large financial burden, there is a need for standardized procedures in software development, such as TDD, since these facilitate simple, structured, and smooth processing of projects. The problem with TDD is that it is designed for more constant and deterministic systems. This contrasts with ML systems, which have a changing and statistical nature because they require models to be regularly trained (cf. [2]).
The decision to develop a conversational AI was made due to the practical application and because the discovery of transformers with an attention mechanism offers a recent, very interesting innovation in this area (cf. [3]). Furthermore, the use of conversational AI represents a large financial relief for companies by automating customer support (cf. [4]).
This paper presents a TDD approach to implement conversational AI. Herefore it is essential to consider the requirements of conversational AI, as well as, the differences and peculiarities of machine learning so the TDD leads to a valid solution. In this approach, while considering the differences, a focus was placed on maintaining the structure of the TDD in order to retain as many of the advantages as possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

1.1 Motivation

Die Anwendungsbereiche des Machine Learning umfassen inzwischen sehr folgenschwere Bereiche, wie beispielsweise die Bilderkennung in der Medizin [5], das autonome Fahren [6] oder im Bereich des Natural Language Processing (NLP) [7]. Durch die Verwendung von ML in diesen Bereichen wird auch das Testen ihrer richtigen Funktionstüchtigkeit immer wichtiger. Damit ML-Modelle sicherheitskritisch funktionieren, muss z. B. sichergestellt werden, dass das Verhalten des Modells bzgl. Korrektheit, Robustheit, Privatsphäre, Effizienz und Fairness den Erwartungen entspricht (vgl. [2]).
Dadurch, dass ML immer häufiger in Software verwendet wird, wird auch die Anpassung von Softwareentwicklungsmethoden an ML an Bedeutung gewinnen. Diese versprechen eine schnelle und sichere Implementierbarkeit der Anwendungen, was für Unternehmen einen großen finanziellen Aspekt bei der Softwareentwicklung darstellt.
Die Verwendung von Test-Driven Development hat in der klassischen Softwareentwicklung viele Vorteile, welche die Fragestellung, ob das TDD anwendbar ist für eine Conversational AI, sehr interessant macht. Ein großer Vorteil des TDD ist beispielsweise die Arbeitsweise, welche sicherstellt, dass die entwickelte Anwendung zu jeder Zeit umfangreich getestet ist und somit ebenfalls dem Entwickler die Sicherheit gibt, dass die Anwendung zu jeder Zeit funktionstüchtig ist. Diese und weitere Vorteile bieten großes Potenzial für die Entwicklung von ML mit dem TDD.
Die vorgestellte Vorgehensweise wird an einem praktischen Anwendungsfall evaluiert. Der Anwendungsfall bezieht sich auf die Entwicklung einer Conversational AI, welche den Nutzer bei der Reisekostenabrechnung unterstützt. Diese Conversational AI, umgangssprachlich auch Chatbot genannt, ist im Rahmen eines Projekts der Firma INNOQ entstanden, um eine Applikation der Firma, den Reisekosten-Gorilla[1], zu erweitern. Der Chatbot wurde dabei mit dem Framework Rasa[2] umgesetzt und für die Tests mit dem Framework CheckList[3] erweitert. Mit diesem Chatbot soll die Anwendbarkeit und die Benutzerfreundlichkeit gesteigert werden. An diesem Beispiel werden ebenfalls die Vor- und Nachteile der Vorgehensweise analysiert und beschrieben.
In dieser Arbeit werden die verwendeten ML-Modelle erläutert, jedoch liegt kein Fokus auf einer umfangreichen Auswertung bzgl. aller Anwendungsbereiche des ML. Diese würden durch die grundlegenden Unterschiede in den Daten und den Modellen den Rahmen dieser Arbeit überschreiten. Grundsätzlich gibt diese Arbeit jedoch Ansätze, welche in zukünftigen Arbeiten genauer untersucht werden können.

[1] `https://www.reisekosten-gorilla.com`
[2] `https://rasa.com`
[3] `https://github.com/marcotcr/checklist`

1.2 Ziel der Arbeit

Softwareentwicklungsmethoden wie das TDD spielen in der Softwareentwicklung eine zentrale Rolle. Sie fördern eine strukturierte Arbeitsweise und nehmen somit einen essenziellen positiven Einfluss auf die erfolgreiche Implementierung der Anwendungen. Da das Machine Learning immer häufiger in Software verwendet wird, müssen auch die Entwicklungsmethoden daran angepasst werden. Daher ist die zentrale Fragestellung dieser Arbeit, ob und wie Test-Driven Development anwendbar ist, um eine Conversational AI zu entwickeln.
Im Rahmen dieser Fragestellung wird auf das Testen von ML eingegangen, welches notwendig ist, um ML erfolgreich in Software zu verwenden. Das Ziel ist es eine Vorgehensweise vorzustellen, welche nach den Prinzipien des TDD handelt und die Entwicklung einer Conversational AI ermöglicht. Die Vorgehensweise wird dabei an dem beschriebenen praktischen Anwendungsfall eines Chatbots analysiert, um die Vor- und Nachteile vergleichen.

1.3 Aufbau der Arbeit

Zur Struktur der Arbeit ist zu sagen, dass in Kapitel 2 ein grundlegendes Verständnis über die in dieser Arbeit behandelten Themen geschaffen wird. Hierzu werden insbesondere die Besonderheiten und Unterschiede des Testens von ML, das TDD und die Conversational AI im Detail erklärt.
Im Anschluss wird in Kapitel 3 auf der Schwerpunkt dieser Arbeit, die Vorgehensweise bei der Entwicklung der Conversational AI mit dem TDD, eingegangen.
Im Kapitel 4 werden die verwendeten Frameworks beschrieben, welche in der Umsetzung der Anwendung verwendet wurden.
In Kapitel 5 kommen diese Frameworks zum Einsatz, wenn die Implementierung der Vorgehensweise beschrieben wird und wie der Anwendungsfall mithilfe der Frameworks Rasa und CheckList umgesetzt wurde.
Im darauf folgenden Kapitel 6 wird untersucht, welche Voraussetzungen des TDD eingehalten wurden und welche Vor- und Nachteile die Vorgehensweise hatte. Hierbei wird ebenfalls die Performanz der entwickelten Anwendung untersucht und beschrieben.
Das abschließende Kapitel 7 beinhaltet eine Betrachtung und Zusammenfassung der Ergebnisse, um ein Fazit über diese Arbeit zu ziehen.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte dieser Arbeit erläutert, welche für das Verständnis dieser Arbeit benötigt werden. Hierzu wird in Abschnitt 2.1 das Testen von ML und im Abschnitt 2.2 das klassische TDD erläutert. Hinzu kommt, den praktischen Teil dieser Arbeit betreffend, in Kapitel 2.3 eine Erläuterung, um was es sich bei dem Begriff „Conversational AI“ handelt und wie dieser definiert werden kann.

2.1 Das Testen von Machine Learning

Viele derzeitige Anwendungen von maschinellem Lernen befinden sich in sicherheitsrelevanten Bereichen, wie beispielsweise in selbstfahrenden Autos oder medizinischer Behandlung (vgl. [2]). Die Vertrauenswürdigkeit solcher Modelle muss in diesen sicherheitskritischen Bereichen enorm hoch sein. Daher liegt ein hoher Fokus auf die Sicherstellung der Qualität dieser Modelle durch Tests. Das Testen ist jedoch nicht nur in diesen Bereichen von Bedeutung, da jegliche entwickelte Software gewisse Voraussetzungen erfüllen und nachweislich einhalten muss. Das Testen der Software soll daher sicherstellen, dass sich ein Modell in der Produktion so verhält, wie es erforderlich ist.
Das Testen von Machine Learning stellt dabei eine besondere Problemstellung dar, da die Ursache für den Fehler an verschiedenen Stellen liegen kann. Wenn ein ML-Modell einen Bug aufweist, kann dies an den Daten, an dem Modell oder am benutzten Framework und dessen inkorrekter Benutzung liegen. Unter einem „Bug“ oder „Fehler“ wird dabei jegliches inkorrektes Verhalten des Modells verstanden. Hinzu kommt, dass vor allem bei den Daten und dem Modell eine starke Abhängigkeit herrscht, welche die Identifizierung des ausschlaggebenden Faktors zusätzlich erschwert. Der fundamentale Unterschied, dass ML-Systeme weniger deterministisch und mehr statistisch orientiert sind, trägt dazu bei, dass das Testen von ML oft zu einer Herausforderung wird (vgl. [2]).
Das Testen von ML leidet dabei unter einer Form des Test-Orakel-Problems. Das Test-Orakel-Problem beschreibt allgemein die Herausforderung zwischen dem gewünschten, richtigen Verhalten des Systems und potenziell falschem Verhalten zu unterscheiden. Hierfür werden die sog. Testorakel definiert (vgl. [8]).
Da ML-Systeme jedoch die Funktion haben, eine Antwort zu generieren, für die es bisher keine Antwort gab, ist es bei solchen Systemen oft schwierig feste Testorakel für das richtige Verhalten zu definieren (vgl. [2]).
Um genauer auf die Besonderheiten des Testens von ML einzugehen, wird in den folgenden Abschnitten auf das Test-Orakel-Problem und die Unterschiede des Testens von Software mit und ohne ML eingegangen. Daraufhin werden die verschiedenen Eigenschaften des ML erläutert, welche gesondertes Testen erfordern.

2.1.1 Das Test-Orakel-Problem

Wie bereits erwähnt, leidet das Testen von ML unter einer Form des Orakel-Problems (engl. oracle problem). Testorakel werden in der Softwareentwicklung genutzt, um automatisiert zu testen. Automatisierte Tests sind wichtig, da diese das Testen schneller, billiger und verlässlicher machen. Hierbei werden Testorakel genutzt, um zwischen dem korrekten und inkorrekten Verhalten eines Systems zu entscheiden. Sie sollen bei ihrer Ausführung sowohl Bugs des Programms finden, falls solche vorhanden sind, als auch sicherstellen, dass keine Bugs vorhanden sind, falls dies zutrifft.
Diese Unterscheidung kann im ML zu einer Herausforderung werden. Zum einen arbeitet ML oft mit Wahrscheinlichkeiten, was die Abgrenzung zwischen korrektem und inkorrektem Verhalten schwieriger macht. Dies ist zwar behebbar, indem eine Softmax-Funktion verwendet wird, wodurch das Ergebnis vergleichbar wird. Jedoch ist dies nur für einzelne Testfälle möglich, da ein Test nicht den gesamten Inputraum testen kann, um somit eine absolute Sicherheit zu erreichen. Außerdem kann sich das Verhalten von einem ML-Modell in der Produktion durch die Veränderung der Trainingsdaten ändern. Das Verhalten eines ML-Systems hängt von den Trainingsdaten ab. Die Trainingsdaten bleiben jedoch nicht gleich, da diese in den Concept Drift, eine Veränderung in der Verteilung der Daten, fallen können und somit die Vorhersage des Modells verschlechtern. Um dies zu verhindern, müssen die Modelle auch in der Produktion regelmäßig neu trainiert werden, wodurch sich auch das Verhalten des Systems mit der Zeit verändern kann.
In der Literatur werden daher oft Lösungen für das Test-Orakel-Problem gesucht, indem das ML-System als eine Einheit betrachtet wird. Hierdurch können diese leichter mit den bereits bekannten Techniken des Testens von Software verglichen werden. Das System als eine Einheit zu betrachten, macht das Testen einfacher, jedoch wird es schwieriger zu bestimmen, welche Komponenten ausschlaggebend für einen gefundenen Fehler sind (vgl. [2]).
Im Machine Learning wird daher oft auf Pseudo-Orakel zurückgegriffen, wie metamorphischen Relationen, welche im Abschnitt 2.1.2 beschrieben werden.

2.1.2 Software-Testing vs. ML-Testing

Software-Testen unterscheidet sich von ML-Testen in verschiedenen Aspekten. In diesem Abschnitt werden die Eigenheiten erläutert, welche das ML-Testen vom Software-Testen unterscheiden. Zur Veranschaulichung wird Tabelle 2.1.2 vorgestellt. Diese Unterschiede werden danach in den folgenden Paragraphen genauer beschrieben.

Charakteristik	Software-Testen	ML-Testen
Zu testende Komponente	Code	Daten, Code, Algorithmus, Framework
Verhalten beim Testen	fix	veränderlich
Test-Inputs	Daten	Daten oder Code
Testorakel	einfach, da deterministisch	schwierig durch die Lernfähigkeit
Tauglichkeitskriterien	Code-Abdeckung	Neuronen-Abdeckung

Tabelle 1: Unterschiede des Software-Testens zum ML-Testen

Die zu testende Komponente: Die getestete Komponente ist im Software-Testen der Code, welcher den möglichen Bug erzeugt. Im ML-Testen kann der Bug durch die Daten, den Code, die Modell-Architektur oder das genutzte Framework entstehen, welche alle eine essenzielle Rolle für die Entstehung des Bugs einnehmen.

Verhalten während des Testens: Das Verhalten von Software ohne ML ist deterministisch, sobald die Voraussetzungen festgehalten werden. Ein ML-Modell hingegen kann sein Verhalten verändern, da sich die Datengrundlage, auf dem das Modell trainiert wird, ändert.

Test-Inputs: Die Test-Inputs sind im Software-Testen die Daten, welche den Code testen. Beim Testen von ML können diese Inputs unterschiedlicher Natur sein. Zum einen können Daten den Algorithmus testen, zum anderen können aber auch Algorithmen die Daten testen.

Testorakel: Im Software-Testen werden Testorakel definiert, um zwischen dem richtigen und falschen Verhalten der Anwendung zu entscheiden. Diese werden von den Entwicklern in Zusammenarbeit mit den Product Ownern festgelegt, um die erwartete Funktionstüchtigkeit zu gewährleisten. Im ML ist die genaue Definition solcher Testorakel jedoch schwierig und zeitaufwändig.
Eine Möglichkeit Testorakel für ML-Systeme zu erzeugen, sind metamorphische Relationen. Metamorphische Relationen sind im Allgemeinen Relationen, welche bei einer Veränderung im Input eine bekannte Veränderung im Output erzeugen. Um sich dies zu verdeutlichen, kann als Beispiel ein Programm genommen werden, welches den $sin(x)$ berechnen soll (vgl. [2]). Im Vorhinein ist bereits die periodische Eigenschaften der Sinusfunktion bekannt, dies bedeutet, dass für metamorphische Relationen in der Regel domain-spezifisches Wissen benötigt wird. Man kann also, um die richtige Funktionstüchtigkeit des Programms zu testen, das Testorakel aufstellen, dass der Output von x mit dem Output von $\pi - x$ übereinstimmen muss. Die metamorphische Relation $sin(x) = sin(\pi - x)$ kann somit als Pseudo-Orakel verwendet werden, um das Programm zu testen.
Hierbei ist anzumerken, dass das Finden von Testorakeln im ML ein umfangreiches Unterfan-

gen ist, da Fachwissen benötigt wird, um Testorakel zu erkennen. Hierbei ist es stark abhängig von der Domain, ob und welche metamorphischen Relationen gefunden werden können. Eine Folge aus fehlenden Testorakeln ist, dass die Entwickler von ML-Systemen häufiger die Rolle des Testorakels übernehmen.

Tauglichkeitskriterien der Tests: Um herauszufinden, wie ausführlich eine Software getestet ist, existieren etablierte Tauglichkeitskriterien, welche ein Maß dafür geben, wie gut ein Programm getestet ist. Beispiele solcher Kriterien sind die Zeilenabdeckung (engl. line coverage), Zweigabdeckung (engl. branch coverage) oder die Abdeckung der Datenpipeline (engl. dataflow coverage). Durch die Unterschiede des ML sind diese jedoch nicht direkt anwendbar für ML-Systeme. Ein neuronales Netz könnte z. B. abgedeckt werden, indem nur die Vorhersagefunktion getestet wird. Dies resultiert in einer Zeilenabdeckung von 100%, jedoch gibt dies keinerlei Aussagen darüber, wie gut das Modell getestet ist. Daher werden neue Kriterien für das ML benötigt.
Neue Abdeckungskriterien für neuronale Netze wurden wie folgt definiert (vgl. [2]):
Die Neuronen-Abdeckung (engl. neuron coverage) wird berechnet als das Verhältnis von allen von den Test-Inputs aktivierten Neuronen zu den gesamten Neuronen im neuronalen Netz.
Die modifizierten Konditions-/Entscheidungs-Abdeckungen (engl. modified condition/decision coverage, MC/DC) sind Kriterien, bei denen die Auswirkung einer Veränderung einer einzelnen Kondition, wie einer booleschen Variable, auf die Entscheidung des Modells beobachtet wird. Bei einem neuronalen Netz wird hierzu beobachtet, welche Neuronen von der Kondition in Form von Vorzeichen-, Wert- oder Abstandsänderungen beeinflusst werden. Somit wird versucht, kausale Veränderungen in den Test-Inputs zu erklären, wobei eine Voraussetzung ist, dass ein Fully-connected Network vorliegt.
Ein weiteres Kriterium ist die Layer-Level-Abdeckung (engl. layer-level coverage). Diese betrachtet die am häufigsten aktivierten Neuronen und deren Kombinationen miteinander, um das Verhalten des Netzes zu charakterisieren. Aufgrund der Fokussierung auf Neuronen synergiert dieses Kriterium sehr gut mit der Neuronen-Abdeckung.

Bei diesen Kriterien ist anzumerken, dass in der Studie von Zhang et al. [2] keine starke Korrelation zwischen den falsch klassifizierten Inputs und der strukturellen Abdeckung des Modells gefunden wurden. Daher ist nicht klar, wie sich solche Kriterien auf das Entscheidungsverhalten des ML-Systems übertragen lassen.

2.1.3 Online- und Offline-Testing

Beim Testen von ML kann eine Unterteilung in Online- und Offline-Testen gemacht werden. Diese unterscheiden sich im Zeitpunkt der Anwendung. Das Offline-Testen findet während der Modellentwicklung bzw. automatisiert auch während des Retrainings statt. Das Online-Testen

wird hingegen verwendet, um das Modell während seiner aktiven Phase zu beobachten, z. B. in Form von Monitoring.

Das Offline-Testen umfasst bekannte Techniken, wie z. B. die Kreuzvalidierung (engl. cross validation). Es wird angewendet, um das Modell bereits in der Entwicklungsphase den Anforderungen entsprechend zu trainieren. Hierfür werden auch die bereits beschriebenen Testorakel generiert und genutzt. Eine Ausführung der Tests generiert einen Bug-Report, welcher genutzt wird, um gefundene Bugs zu lokalisieren, replizieren und aufzulösen. Falls hierbei keine Bugs gefunden werden, ist das Modell validiert und kann in die Produktionsumgebung gestellt werden.
Nachdem das Modell somit erfolgreich validiert wurde, wird es mithilfe des Online-Testens in der Produktionsumgebung analysiert, um zu evaluieren, ob es sich auch unter realen Umständen wie erwartet verhält.
Das Online-Testen ist notwendig, da das Offline-Testen zum einen auf historischen Daten basiert, welche die zukünftigen Daten meistens nicht fehlerfrei repräsentieren. Zum anderen ist das Offline-Testen nicht in der Lage bestimmte Umstände zu simulieren, welche in der Praxis auftreten. Hierzu zählen Datenverluste aufgrund von Verbindungsproblemen oder auftretende Verzögerungen im Datenfluss, aber auch bestimmte Metriken, wie eine Klickrate (engl. clickthrough rate), können erst in der aktiven Phase des Modells evaluiert werden.
Online-Testen findet häufig in Form von Runtime-Monitoring und A/B-Testing statt. Das Runtime-Monitoring wird genutzt, um die aktuelle Performanz des Modells in der Produktion zu beobachten. Das A/B-Testen hingegen generiert Nutzergruppen und vergleicht das aktuelle Modell mit einer vorherigen Version, um zu validieren, ob sich das neue Modell verbessert hat.

2.1.4 Die zu testenden Eigenschaften des Machine Learnings

In diesem Abschnitt werden die zu testenden Eigenschaften des ML beschrieben. Diese Eigenschaften des ML-Modells müssen getestet werden, um sicherzustellen, dass ein Modell sich wie erwünscht verhält. Hierzu werden einige wichtige Eigenschaften beschrieben, welche in der Literatur häufig vertreten sind und im praktischen Teil dieser Arbeit berücksichtigt wurden. Anzumerken ist, dass die Eigenschaften je nach Anwendungsfall und Voraussetzungen unterschiedlich aussehen können. Sie reichen somit von wünschenswerten Kriterien, welche erreicht werden sollten, bis zu Ausschlusskriterien, welche die Modelle verwerfen, falls sie nicht erreicht werden. Diese Eigenschaften werden im Folgenden dieser Arbeit als Testeigenschaften des ML bezeichnet.

Korrektheit

Die Korrektheit des Modells ist die am häufigsten vorkommende Voraussetzung. Die Korrektheit eines ML-Modells ist die funktionale Voraussetzung, welche die Richtigkeit der Vorhersage beschreibt. Sie ist eine wichtige Voraussetzung, da ein Modell, welches keine angemessene Korrektheit erfüllt, für die Produktionsumgebung nicht geeignet ist. Gleichzeitig ist sie ein wünschenswertes Kriterium, da sie unter Beachtung aller anderen Voraussetzungen, so hoch wie möglich sein soll.
Eine Definition der Korrektheit ist (vgl. [2]):

Definition 1 *(Korrektheit). Sei D die wahre Verteilung der zukünftigen Daten. Sei $x \in D$ und h das getestete Modell. $h(x)$ ist das vorhergesagte Label des Modells zu x. $c(x)$ ist das wahre Label von x. Die Korrektheit des Modells $E(h)$ ist dann die Wahrscheinlichkeit, dass $h(x)$ und $c(x)$ identisch sind:*

$$E(h) = P_{x \in D}(h(x) = c(x))$$

Da die wahre Verteilung der Daten oft nicht bekannt ist, wird meistens die empirische Korrektheit getestet:

Definition 2 *(Empirische Korrektheit) Sei $X = (x_1, ..., x_m)$ ein Set von nicht gelabelten Daten aus D. Sei h wieder das getestete Modell und $Y' = (h(x_1), ..., h(x_m))$ die vorhergesagten Label des Modells zu X. Sei $Y = (y_1, .., y_m)$ die wahren Labels, wobei $y_i \in Y$ die zugehörigen Label von $x_i \in X$ sind. Die Empirische Korrektheit $\hat{E}(h)$ ist dann:*

$$\hat{E}(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}(h(x_i) = y_i)$$

Die Korrektheit ist eine Testeigenschaft, welche sowohl offline getestet als auch online beobachtet werden muss, um die Performanz des Modells zu sichern. Sie nimmt somit eine besondere Position in den Testeigenschaften ein, da sie sowohl eine Mindestanforderung als auch ein wünschenswertes Kriterium darstellt. Dies ist von besonderem Interesse, wenn zusätzlich andere Eigenschaften betrachtet werden. Bei mehreren Modellen, welche die Mindestanforderungen erfüllen, entscheidet die Korrektheit, welches das performantere Modell ist. Die Korrektheit von Klassifikationen kann durch Metriken, wie z. B. die Accuracy, Precision, Recall oder die Area-Under-Curve (AUC) gemessen werden. Welche Metriken gewählt werden sollten, ist dabei abhängig vom Anwendungsfall und dem gewählten Modell.
Im Anwendungsfall dieser Arbeit wurde die Korrektheit durch mehrere Tests sichergestellt. Hierzu zählen sog. Minimum Functionality Tests, welche die empirische Korrektheit testen, aber auch klassische Tests, welche z. B. die Accuracy während des Trainings überprüfen.

Modell-Relevanz

Die Modell-Relevanz ist ebenfalls eine funktionale Voraussetzung. Sie befasst sich mit der Kapazität des Modells, welche nicht viel höher sein sollte als die Komplexität der Daten. Sie bildet somit ein Maß dafür, wie gut ein Modell zu den Daten passt.
Die Modell-Relevanz kann wie folgt definiert werden (vgl. [2]):

Definition 3 *(Modell-Relevanz) Sei D die Verteilung der Trainingsdaten. Sei $R(D, A)$ die minimale benötigte Kapazität eines beliebigen ML-Algorithmus A für D. $R'(D, A')$ ist die Kapazität des zu testenden ML-Algorithmus A'. Dann ist die Modell-Relevanz:*

$$f = |R(D, A) - R'(D, A')|$$

Eine schlechte Modell-Relevanz ist ein Zeichen auf Over- oder Underfitting. Im Fall des Overfittings ist die Modell-Relevanz sehr niedrig. Das Modell ist zu komplex für die Daten und lernt das Rauschen in den Trainingsdaten mit. Somit lernt es jedoch nicht die richtigen Verbindungen zwischen den Daten und den Outputs und kann daher ungesehene Datenpunkte nicht gut vorhersagen. Wenn die Modell-Relevanz auf der anderen Seite zu hoch ist, spricht dies für Underfitting. Hierbei ignoriert das Modell eventuell wichtige Informationen in den Daten und erreicht daher einen sehr hohen Wert. Die optimale Modell-Relevanz $R'(D, A)$ ist jedoch schwer herauszufinden und muss daher approximiert werden.

Die Pertubed Model Validation (PMV) ist eine Möglichkeit, die Modell-Relevanz zu validieren. Ziel der PMV ist es, einen Indikator für Over- bzw. Underfitting zu geben. Dies geschieht über erneutes Trainieren des Modells auf perturbierten Trainingsdaten, welchen ein Rauschen hinzugefügt wird. Bei diesem Training wird die Abnahme der Accuracy betrachtet, um Over- bzw. Underfitting zu erkennen. Die Intuition ist, dass sowohl bei Over- als auch Underfitting auf den absichtlich verrauschten Trainingsdaten eine, im Verhältnis zu dem hinzugefügten Rauschen, niedrige Abnahmerate der Accuracy verzeichnet wird. Dies ist darauf zurückzuführen, dass im Falle des Overfittings das hinzugefügte Rauschen in den Daten ebenfalls mit erlernt wird. In Falle des Underfittings hingegen wird das Rauschen vom Modell ignoriert und somit erfolgt keine Abnahme in der Accuracy. Die PMV bietet dabei im Vergleich zur 10-fachen Kreuzvalidierung eine bessere Performanz und ein besser erkennbares Signal für Over- als auch Underfitting (vgl. [9]).
Eine andere Möglichkeit, die Modell-Relevanz zu approximieren, ist über die Trainingszeit. Die Trainingszeit eines Modells kann als Proxy für die Komplexität des Modells dienen, da ein Modell mit einer höheren Komplexität oft eine längere Trainingszeit aufweist als ein einfacheres Modell. Somit besitzt ein Modell bei gleicher Korrektheit, aber kleinerer Trainingszeit, eine bessere Modell-Relevanz als das Modell mit einer längeren Trainingszeit.
Im praktischen Anwendungsfall wurde die Modell-Relevanz durch keinen expliziten Test si-

chergestellt. Jedoch wurde, falls mehrere Modelle die Anforderungen erfüllen, der Indikator der Trainingszeit berücksichtigt, sodass ein schneller trainierbares Modell präferiert wird.

Robustheit

Die Robustheit ist in dem IEEE Standard Glossar[4] definiert als: „The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions“. Die Robustheit eines ML-Systems ist also die Toleranz des Systems in Anwesenheit von Störungen weiterhin richtig zu klassifizieren. Hierbei kann zwischen generellen Störungen und absichtlichen Störungen unterschieden werden. Diese können dann der generellen Robustheit oder der feindlichen Robustheit zugeordnet werden.

Die generelle Robustheit beschreibt dabei die allgemeine Korrektheit des Systems in der Anwesenheit von Rauschen in den Daten. Ein robustes System sollte die Performanz erhalten, auch wenn Rauschen in den Daten existiert. Um die Robustheit eines Systems zu messen, können z. B. die folgenden Metriken verwendet werden (vgl. [2]):

- Die punktweise Robustheit: Sie misst die minimale Änderung eines Inputs, bis das System mit der Klassifizierung fehlschlägt.
- Die feindliche Frequenz: Sie misst, wie oft eine Änderung des Inputs eine Änderung im Output verursacht.
- Die feindliche Strenge: Sie misst, wie groß die Änderung eines Inputs bis zu seinem nächsten feindlichen Beispiel ist.

Im Kontext des NLP entspricht die feindliche Frequenz z. B. der Resistenz gegen Tippfehler. Hierbei wird gemessen, wie oft ein Tippfehler zu einer Fehlklassifikation des Inputs führt. Um die generelle Robustheit zu testen, wurden in der entwickelten Anwendung alle Daten der Tests der feindlichen Frequenz unterzogen.

Eine Subkategorie der Robustheit ist die feindliche Robustheit (engl. adversarial robustness). Hierbei werden absichtliche Störungen in einen Datenpunkt eingebaut. Diese sog. feindlichen Datenpunkte (engl. adversarial examples) werden in feindlichen Angriffen (engl. adversarial attacks) verwendet, um zu einer Fehlklassifikation zu führen. Das Kritische an diesen Angriffen ist es, dass das verwendete Modell dem Angreifer nicht bekannt sein muss. Es genügt ein eigenes Modell zu trainieren, welches für den gleichen Zweck erstellt wird. Mit diesem Modell können feindliche Datenpunkte generiert werden, welche in jedem Modell für diesen Zweck eine Fehlklassifikation zur Folge haben.

[4] `http://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology%20.pdf`; Stand: 14.7.2021

Wie Adversarial Attacks und die Generierung eines Adversarial Examples aussieht, ist in Abbildung 1 an einem Beispiel veranschaulicht.

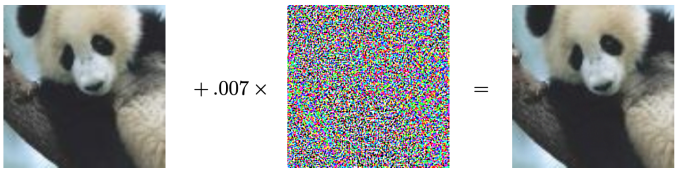

Abbildung 1: Beispiel eines Adversarial Example [10]

In der zu Grunde liegenden Studie über Adversarial Attacks [10] wurde das Bild rechts zu 57,7% als ein Panda klassifiziert. Jedoch wurde nach dem Hinzufügen des Rauschens das gleiche Bild zu 99,3% als Gibbon, eine Art der kleinen Menschenaffen, erkannt, obwohl für das menschliche Auge kaum erkennbar ist, dass die Bilder sich unterscheiden.
Adversarial Attacks sind deshalb kritisch, da durch sie bestimmte Sicherheitsmechanismen umgangen werden können. Ein Beispiel sind Spamfilter, wodurch E-Mails mit Werbung oder Trojanern nicht in das Spam-Postfach sortiert werden. Aber auch in Firewalls wird ML verwendet, wodurch schädliche Software auf den Computer geladen werden kann. Durch die Kombination könnte eine schädliche Software, wie z. B. ein Überwachungsprogramm, über eine E-Mail völlig unbemerkt auf den Computer gelangen. Da dies ein sehr umfangreiches Thema ist und nicht im Schwerpunkt dieser Arbeit liegt, wird für eine ausführliche Beschreibung, wie Adversarial Attacks funktionieren, auf die Quelle [10] verwiesen.
Allgemein kann bei einer Klassifikation auch unterschieden werden, ob es sich um einen gelenkten oder ungelenkten Angriff handelt. Ein gelenkter Angriff zielt darauf, die Klassifikation auf einen bestimmten Output fehlzuleiten. Ein ungelenkter Angriff hat nur eine Fehlklassifikation als Ziel. Somit unterscheidet man ebenfalls zwischen der lokalen feindlichen Robustheit (engl. local adversarial robustness), welche der Robustheit gegen gelenkte Angriffe entspricht, und der globalen feindlichen Robustheit (engl. global adversarial robustness), welche der Robustheit gegen eine allgemeine Fehlklassifikation entspricht.

Sicherheit

Die Sicherheit eines ML-Systems entspricht u. a. dem Schutz des Systems gegen Hacker, Datendiebstahl oder Informationsdiebstahl. Hierzu zählt auch der Schutz gegen Adversarial Attacks. Da ein System mit einer hohen Robustheit im Allgemeinen auch resistent gegen Adversarial Attacks ist und der restliche Schutz des Systems mit den üblichen Sicherheitsmaßnahmen abgedeckt ist, wird der Aspekt der Sicherheit in dieser Arbeit nicht weiter berücksichtigt.

Privatsphäre

Der Datenschutz und die Privatsphäre werden oft vom Gesetzgeber geregelt, beispielsweise durch die europäische Datenschutzgrundverordnung (DSGVO) oder das kalifornische Verbraucherschutzgesetz (CCPA).
Aktuelle Forschung fokussiert sich eher auf die Wahrung der Privatsphäre, anstatt Datenschutzverletzungen zu erkennen (vgl. [2]). In diesem Sinne wird in dieser Arbeit auf die Wahrung von persönlichen Daten geachtet, indem Tests implementiert wurden, welche in Kooperation mit der Fairness darauf achten, dass schützenswerte Attribute sowohl von der Prognose unabhängig als auch nur zwecks der Zuordnung bei der Anbindung an das vorhandene System verwendet werden.

Effizienz

Die Effizienz eines ML-Systems bezieht sich, wie bei einem normalen Softwaresystem, auf die Schnelligkeit der Ausführung. Im Falle eines ML-Systems beschreibt dies sowohl die Schnelligkeit des Trainings als auch der Vorhersage. Auch wenn durch immer höher werdende Hardwarestandards die Schnelligkeit des Systems oft kein Problem mehr darstellt, steigen gleichzeitig auch die Datenmengen, mit denen ein Modell trainiert wird. Somit kann die Effizienz weiterhin ein entscheidender Faktor sein.
Wenn die Anwendung z. B. auf einem mobilen Endgerät trainiert und benutzt werden soll, wie es im Federated Learning der Fall ist, kann die Effizienz von sehr hoher Relevanz sein. In solchen Fällen ist die Effizienz meist wichtiger als die Korrektheit, da ein Modell, welches die Effizienz nicht erfüllt, auf dem Endgerät nicht nutzbar ist (vgl. [11]).
Das Kriterium der Effizienz kann dabei über einfache Assertionen in den Tests sichergestellt werden, welche die Trainings- und Prognosezeit der Anwendung messen.

Fairness

Die Fairness und diese zu bewahren, gehört zu den schwierigen Eigenschaften eines ML-Systems. Dies ist darauf zurückzuführen, dass es im Allgemeinen schwieriger ist, Fairness zu definieren. In der Literatur werden viele Definitionen von Fairness vorgestellt, jedoch gibt es keine einheitliche Definition von Fairness (vgl. [2]). Die Fairness und diese Sicherzustellen ist dabei vom Anwendungsfall abhängig und muss daher auch abhängig von diesem behandelt werden.

Im Allgemeinen kann gesagt werden, dass sensible Charakteristiken in den Daten vorhanden sein können, welche für die Fairness des Modells geschützt werden müssen. Solche Charakteristiken werden geschützte Charakteristiken, geschützte Attribute oder sensible Attribute

genannt. Zu den sensiblen Attributen gehören die Rasse, Hautfarbe, Geschlecht, Religion, nationale Herkunft, Staatsbürgerschaft, Alter, Schwangerschaft, Familienstand, Behinderungsstatus, Veteranenstatus und genetische Informationen eines Menschen.

Zhang et al. [2] haben zur Fairness 5 verursachende Gründe aufgestellt:

- Schiefe Stichprobe: Ein kleiner initialer Bias in den Daten kann über die Zeit wachsen und somit die Fairness verletzen.
- Verschmutzte Beispiele: Das Labeln durch den Menschen kann ein Bias erzeugen, da dieser nach seinen Vorstellungen handelt.
- Limitierte Features: In den verwendeten Features können weniger Informationen enthalten sein als notwendig sind. Das Modell kann somit nicht die richtigen Verbindungen erlernen und deshalb diskriminierend sein.
- Stichprobenungleichheit: Wenn die Daten stark ungleich verteilt sind, kann das Modell die Minderheiten nicht gut vertreten.
- Proxys: Features können ein Proxy für geschützte Attribute sein, welche die Fairness des Modells verletzen, selbst wenn geschützte Attribute ausgeschlossen wurden.

Die Forschung bezüglich der Fairness fokussiert sich auf das Messen, Erkennen, Verstehen und Bewältigen von beobachteten Unterschieden verschiedener Gruppen und Einzelpersonen in der Performanz. Diese Unterschiede sind verbunden mit der Verletzung der Fairness. Die Folgen einer solchen Verletzung reichen dabei von simplen Fehlklassifizierungen, welche den Nutzer verärgern, bis zu Rechtsverletzungen, woraufhin das Unternehmen verklagt werden kann.
Um die Fairness zu sichern, werden nun einige Definitionen und Messmetriken vorgestellt (vgl. [2]):

Die Fairness duch Unwissenheit (engl. fairness through unawareness; FTU): FTU ist die einfachste Form der Fairness. Sie besagt, dass ein Algorithmus fair ist, wenn die sensiblen Attribute nicht mit in den Entscheidungsprozess einfließen. Falls dies, unter Beachtung der oben genannten Ursachen, möglich ist, ist FTU eine effektive und einfache Methode die Fairness des Algorithmus zu sichern. Hierzu muss jedoch sichergestellt sein, dass die oben beschriebenen Gründe, wie z. B. Proxys in den Daten, beachtet sind.

Gruppen-Fairness: Die Gruppen-Fairness besagt, dass ein Algorithmus fair ist, wenn er bezüglich unterschiedlicher Gruppen keine Unterschiede in der Entscheidung aufweist. Dabei gibt es zwei Typen der Gruppen-Fairness:
Die demografische Parität wird erfüllt, wenn das Modell basierend auf einem sensiblen Attribut eine gleiche Verteilung der Entscheidungen aufweist. Sie kann wie folgt definiert werden:

Definition 4 *(Demografische Parität). Seien G_1 und G_2 zwei Gruppen in den Daten X, welche unter dem sensiblen Attribut $a \in A$ geteilt sind. Ein Modell h erfüllt dann die demografische Parität, wenn gilt:*

$$P(h(x_i)|x_i \in G_1) = P(h(x_j)|x_j \in G_2).$$

Die Chancengleichheit (engl. equalised odds) besagt, dass ein Modell h unabhängig von den sensiblen Attributen ist, wenn ein Label $y \in Y$ fixiert wird. Sie kann wie folgt definiert werden:

Definition 5 *(Chancengleichheit). Seien G_1 und G_2 zwei Gruppen in den Daten X, welche unter dem sensiblen Attribut $a \in A$ geteilt sind. Ein Modell h erfüllt dann die Chancengleichheit, wenn gilt:*

$$P(h(x_i)|x_i \in G_1, Y = y_i) = P(h(x_j)|x_j \in G_2, Y = y_i).$$

Die kontrafaktische Fairness Die kontrafaktische Fairness besagt, dass ein Modell fair ist, wenn bei der Änderung eines sensiblen Attributes zu einem kontrafaktischen Wert, unter Berücksichtigung der kausalen Veränderung, die Verteilung des Outputs gleich bleibt. Hierbei werden diese Veränderungen durch ein kausales Modell bestimmt (vgl. [12]).
Ein Beispiel wäre, wenn ein Modell das Geschlecht eines Patienten berücksichtigen muss. Hierbei kann das Modell auf die kontrafaktische Fairness bzgl. des Geschlechts überprüft werden, indem andere Attribute, wie das Gewicht oder die Körpergröße, nach dem kausalen Modell angepasst werden.

Definition 6 *(Kontrafaktische Fairness). Sei a ein sensibles Attribut, a' ein kontrafaktischer Wert des Attributs. Sei des Weiteren x_i' der angepasste Wert durch die Veränderung von a zu a'. Ein Modell h ist kontrafaktisch fair, wenn gilt:*

$$P(h(x_i)_a = y_i|a \in A, x_i \in X) = P(h(x_i')_{a'} = y_i|a \in A, x_i \in X) \quad \forall x_i \in X, a \in A$$

Individuelle Fairness Ein Modell erfüllt die individuelle Fairness, wenn es ähnliche Entscheidungen für ähnliche Inputs erzeugt. Die individuelle Fairness kann wie folgt definiert werden:

Definition 7 *(Individuelle Fairness). Ein Modell h erfüllt die individuelle Fairness, wenn gilt:*

$$P(h(x_i)|x_i \in X) \approx P(h(x_j)|x_j \in X) \text{ iff } d(x_i, x_j) < \epsilon$$

Hierbei ist d eine Distanzmetrik, welche die Ähnlichkeit der Punkte misst und ϵ die Toleranz dieser Distanz.

Interpretierbarkeit

Interpretierbarkeit ist eine Testeigenschaft, welche ebenfalls abhängig von der Domain ist. Der Bedarf an Interpretierbarkeit von ML-Modellen kommt daher, dass ML als Unterstützung für Entscheidungen dienen kann. Diese Entscheidungen müssen in Fällen wie einer medizinischen Behandlung, Einkommensvorhersage oder Kreditvergabe begründet werden können. Hierzu muss nachvollzogen werden, wie das Modell zu einer Entscheidung kam.
Eine allgemeine Möglichkeit, die Interpretierbarkeit zu messen, ist diese über die Kapazität des Modells zu approximieren. Die Tiefe eines Baumes, wie in einem Entscheidungsbaum, kann ein Indikator für eine gute oder schlechte Interpretierbarkeit sein. Dieses Vorgehen wird funktional-basierte Evaluation (engl. functionally-grounded evaluation) genannt und gibt ein Maß dafür, wie gut ein Modell und dessen Entscheidungen interpretiert werden können.

Bei der Interpretierbarkeit kann im Allgemeinen zwischen zwei Arten der Interpretierbarkeit unterschieden werden: Die globale und lokale Interpretierbarkeit. Die globale Interpretierbarkeit, auch Transparenz genannt, beschreibt das Verstehen des gesamten Modells und wie das Modell Entscheidungen trifft. Für die lokale Interpretierbarkeit, auch Post-hoc-Erklärbarkeit genannt, genügt das Verständnis, wie das Modell bei spezifischen Inputs deren Outputs generiert. Entscheidungsbäume und Modelle der linearen oder logistischen Regression sind somit, im Gegensatz zu neuronalen Netzen, grundlegend besser interpretierbar, da sowohl die Entscheidung in jedem Knoten als auch diese an spezifischen Inputs zu erklären sehr logisch nachvollziehbar ist.

Für die Interpretierbarkeit kann das Buch von Christoph Molnar [13] referenziert werden. Dieses besagt, dass die einfachste Art Interpretierbarkeit zu erhalten, ist, diese durch die Wahl der infrage kommenden Algorithmen zu gewährleisten.
Es gibt auch Wege, die Interpretierbarkeit von Deep Learning Modellen zu gewährleisten. Eine Möglichkeit hierzu ist das Framework Shapley Additive Explanations (SHAP) [14].
SHAP ist ein Framework, um individuelle Vorhersagen zu erklären. Es bildet daher eine Möglichkeit, lokale Interpretierbarkeit von ML-Modellen zu erhalten. Durch Aggregation der individuellen Vorhersagen ist hierbei auch eine gewisse globale Interpretierbarkeit möglich.
SHAP verbindet die Shapely Values aus der Spieltheorie, welche mit den Parametern einer linearen Regression vergleichbar sind, mit den Local Interpretable Model-Agnostic Explanations (LIME), einer Methode zur Erstellung von Stellvertreter Modellen. Für eine ausführliche Erklärung, wie das Framework und die Methoden funktionieren, wird auf die Quelle [14] verwiesen.
Durch SHAP ist es jedoch möglich, Graphen zu erstellen, welche den Einfluss der Features im Input auf den Output erklären. Diese können dabei über mehrere Inputs aggregiert werden, um eine globale Interpretierbarkeit zu erreichen.

2.2 Test-Driven Development

Das Test-Driven Development (TDD) ist ein Vorgehen der agilen Softwareentwicklung. Hierbei wird der Code erst geschrieben, wenn ein vorhandener Test diesen erfordert. Dies unterstützt den Programmierer funktionsfähigen, aber vor allem ordentlichen, Code zu schreiben. Solcher Code wird in der Softwareentwicklung als sauberer Code (engl. clean code) bezeichnet. Ziel dabei ist es, die Software effizient und effektiv zu produzieren und dabei den Code so zu schreiben, dass er leicht lesbar, aber auch änderbar, erweiterbar und wartbar ist (vgl. [15]).
In diesem Kapitel wird beschrieben, welchen Regeln das TDD folgt, wie es anzuwenden ist und welche Vorteile es mit sich bringt. Im nächsten Kapitel 3 wird dann darauf eingegangen, wie das TDD auf die Entwicklung einer Conversational AI angewendet werden kann.

2.2.1 Das Vorgehen im TDD

In diesem Abschnitt wird beschrieben, welchen Regeln das TDD folgt und daraufhin wird das Vorgehen anhand eines kleinen Beispiels erklärt.

Die Regeln des TDD: Das TDD folgt im Grunde den zwei folgenden Regeln:

- Schreibe neuen Code nur, wenn ein fehlgeschlagener Test vorliegt.
- Löse Duplikationen auf.

Mit diesen zwei Regeln wird das Vorgehen in der Softwareentwicklung umgedreht, da normalerweise zuerst der funktionale Code implementiert wird und dann durch Tests das Verhalten dieses Codes gesichert wird. Im TDD hingegen wird zuerst überlegt, wie die Tests für die gewünschten Funktionalitäten der Software aussehen und diese Tests dann zuerst implementiert. Dabei ist klar, dass diese Tests zunächst nicht bestehen werden, vielleicht nicht einmal compilieren, da der zugehörige funktionale Code noch nicht existiert. Jedoch darf nun nach der ersten Regel, da ein fehlgeschlagener Test vorliegt, der zugehörige Code implementiert werden.
Das Vorgehen des TDD lässt sich in drei iterierende Phasen unterteilen. Diese Phasen werden als Mantra des TDD bezeichnet. Das Mantra umfasst dabei die drei Phasen Red-Green-Refactor (vgl. [15]):

- Red: Schreibe einen Test für eine neue Funktionalität. Dabei wird davon ausgegangen, dass diese Funktionalität von dem bisherigen Programm nicht erfüllt wird. Der Test schlägt also fehl.

- Green: In dieser Phase wird der funktionale Programmcode um die von dem Test benötigte Funktionalität ergänzt. Hierbei kann dies eine neue Funktionalität des Programms sein oder ein bekannter Fehler im Programm, welcher behoben werden soll. Es wird dabei versucht mit möglichst wenig Aufwand die zusätzliche Funktionalität bereitzustellen. Ziel dabei ist es, genau so viel Code zu implementieren, dass der Test bestanden wird.
- Refactor: In dieser abschließenden Phase einer Iteration wird das Refactoring betrieben. Der Code wird nun aufgeräumt. Es werden eingebaute Wiederholungen im Code beseitigt. Es wird abstrahiert, falls es nötig ist, und es werden eventuelle Formalien an den Code beachtet.

Eine solche Iteration sollte nur wenige Minuten dauern, da beim TDD ein Fokus darauf gelegt wird, das komplexe System in möglichst kleine und handliche Komponenten zu unterteilen, welche einzeln sehr einfach getestet werden können.
Dieser Zyklus wird so lange wiederholt, bis das Programm die gewünschte Funktionalität erfüllt und alle Fehler behoben sind. Dabei ist ein wichtiger Punkt, dass nach dem Abschluss jeder Iteration das Programm ein vorläufig funktionstüchtiges System darstellt. Dies ist für die agile Softwareentwicklung sehr sinnvoll, da somit jederzeit ein Stand der Entwicklung von dem System ablesbar und durch die vorher erstellten Tests gesichert ist.

Die Pattern im TDD: Um die oben benannten Phasen zu absolvieren, werden verschiedene Patterns genutzt. Diese Patterns richten sich, ähnlich wie die Phasen, nach dem Stand des Tests. Im Folgenden werden einige wichtige Patterns erläutert. Für eine ausführliche Auflistung und Beschreibung aller Patterns wird auf das Buch von Kent Beck [15] verwiesen.
In der Red-Phase werden zwei verschiedene Patterns genutzt, die sowohl behandeln, welcher Test als Nächstes bearbeitet wird (sog. Red-Bar-Patterns) als auch wie dieser dann umgesetzt wird (sog. Testing-Patterns).
Die Red-Bar-Patterns behandeln die Problemstellungen, wie mit dem Testen begonnen wird, welcher Test als nächstes gewählt werden kann oder wie mit Fehlern umgegangen wird. Eine Übersicht, wobei die Red-Bar-Patterns Verwendung finden, ist in der folgenden Tabelle 2 gegeben.

Problem	Pattern	Beschreibung
Wahl des nächsten Tests	One-Step-Test	Wähle den Test, der dich einen Schritt voran bringt. Dabei sollte er dir etwas beibringen und für dich umsetzbar erscheinen.
Wahl eines ersten Tests	Starter-Test	Erstelle einen trivialen Test, welcher nicht mal einen Nutzen haben kann. Daraufhin kann mit einem One-Step-Test fortgefahren werden.
Neue Erkenntnisse bei der Implementierung eines Tests	Another-Test	Wenn neue Ideen aufkommen, füge sie der Test-Liste hinzu und bleibe beim Thema.
Ein Fehler ist aufgetreten	Regression-Test	Schreibe den kleinsten Test, der diesen Fehler erzeugt. Wenn dieser Test bestanden wird, ist der Fehler behoben.
Festgefahren und man kommt nicht mehr weiter	Break / Do-Over	Wenn man bei einem Problem nicht mehr weiterkommt, hilft oft eine Pause oder sogar neu anzufangen.

Tabelle 2: Eine Auswahl wichtiger Red-Bar-Patterns

Wie man erkennen kann, liegt der Fokus dieser Patterns darauf, mit welchen Tests der Programmierer fortfahren sollte. Dies ist sinnvoll, da somit ein stetiger Entwicklungsfortschritt gewährleistet werden kann.

Wenn mit den Red-Bar-Patterns ein Test gewählt wurde, werden die Testing-Patterns genutzt, um den gewählten Test zu implementieren. Welche Probleme bei der Implementierung auftreten können und wie diese zu beheben sind, wird durch die Testing-Patterns beschrieben. Eine Auswahl dieser Patterns ist in der folgenden Tabelle 3 beschrieben.

Problem	Pattern	Beschreibung
Der Test ist zu umfangreich	Child-Test	Falls ein Test zu groß erscheint, schreibe kleinere Tests, welche Teile des großen Tests enthalten.
Abhängigkeit von anderen Objekten	Mock-Object	Falls der Test eine Abhängigkeit von einem kompliziertem Objekt (wie z. B. einer Datenbank) aufweist, fälsche dieses. Schreibe ein Objekt, welche die benötigten Voraussetzungen für diesen Test mitbringt.
Testen der Kommunikation zweier Objekte	Self-Shunt	Lasse beide Objekte statt miteinander jeweils mit einem Test kommunizieren.
Testen der Reihenfolge von Methoden	Log-String	Lasse jeden Methodenaufruf in ein String-Objekt schreiben und vergleiche diesen mit der erwarteten Reihenfolge
Den Tag sinnvoll beenden	Broken-Test	Wenn man allein arbeitet, ist es sinnvoll den letzten Test des Tages fehlschlagen zu lassen, um am nächsten Tag einen guten Einstieg zu haben.
	Clean-Check-In	Wenn man in einem Team arbeitet, müssen alle Tests bestanden werden.

Tabelle 3: Eine Auswahl wichtiger Testing-Patterns

Mit diesen Patterns können die ausgewählten Tests erfolgreich implementiert werden. Nachdem die Tests implementiert wurden, folgt die Green-Phase. In dieser Phase wird versucht den implementierten Test mit so wenig Aufwand wie möglich zu bestehen. Hierbei darf der Code auf jegliche Weise manipuliert werden, solange dieses Ziel erreicht wird.
Da diese Patterns den zentralen Schritt innerhalb eines TDD Zyklus einnehmen, werden diese in den folgenden Abschnitten ausführlicher beschrieben. Die drei vorgestellten Patterns unterscheiden sich dabei in der Anwendbarkeit durch den Programmierer und dem Aufwand diese umzusetzen. Dies bedeutet, dass das Pattern der Obvious Implementation einen sehr geringen Aufwand besitzt, jedoch die Anforderung an den Programmierer stellt, dass dieser die Problemstellung, den Lösungsweg und die Implementierung umfangreich verstanden hat. Es ist dabei sinnvoll, das Pattern mit dem geringsten Aufwand zu bevorzugen, da jederzeit zu einem einfacheren Pattern gewechselt werden kann.
Bevor die Patterns erklärt werden, wird zur Orientierung die Tabelle 4 vorgestellt, welche die Anforderungen an den Programmierer für die Green-Bar-Patterns grafisch darstellt:

Problemstellung	Lösungsweg	Implementierung	Pattern
x	x	x	Obvious Implementation
x	x	-	Fake-it
x	-	-	Triangulation

Tabelle 4: Anforderungen der Green-Bar-Patterns an den Programmierer. Das x bzw. - kennzeichnet, ob diese Voraussetzung dem Programmierer bekannt oder unbekannt ist.

Obvious Implementation: Dieses Pattern kann angewendet werden, wenn der Programmierer die Problemstellung, den Lösungsweg und die performanteste Implementierung kennt. Somit kann der Programmierer schnell und sicher die benötigte Funktionalität umsetzen. Dies bringt den Nachteil mit sich, dass der Programmierer hierzu Perfektion von sich selbst voraussetzt. Es gibt keine Garantie, dass der Programmierer den besten Weg der Implementierung gewählt hat oder überhaupt zu einer zulässigen Lösung kommt. Ein Beispiel, wie die Obvious Implementation verwendet wird, ist in Appendix A zu finden.

„Fake-it (until u make it)“: Falls der Programmierer sich über die genaue Implementierung nicht sicher ist, kann er auf das „Fake-it“-Pattern zurückgreifen. In diesem Fall kennt der Programmierer zwar das Problem und wie es gelöst wird, jedoch nicht wie die Implementierung dazu aussieht.

Bei diesem Vorgehen wird es erlaubt, die Funktionalität zeitweilig zu fälschen. Wenn der Test umgesetzt wurde, ist es nach diesem Pattern dem Programmierer erlaubt, eine Funktion zu implementieren, welche die Lösung des Tests zurückgibt, z. B. in Form einer Konstante. Hierbei ist klar, dass diese Funktion zu diesem Zeitpunkt nichts anderes löst, außer den implementierten Test. Um diesen Zustand aufzulösen, wandelt der Programmierer diese Rückgabe ab, sodass die Funktionalität durch einen Ausdruck mit Variablen entsteht. Dies wird z. B. durch die Eliminierung von Duplikaten erreicht. Ein Beispiel, wie dieses Pattern verwendet wird, ist in Appendix B zu finden.

Es gibt dabei zwei wesentliche Vorteile durch dieses Pattern:

- Es ist psychologisch aufbauend: Ein bestehender Test wirkt sich positiv auf die Motivation des Programmierers aus. Ein fehlschlagender Test, ohne die Lösung zu kennen, kann hingegen zu Frustration führen.
- Es liefert Kontrolle über den Umfang: Programmierer sind gut darin, alle möglichen Situationen zu erkennen und zu berücksichtigen. Dies kann jedoch zu Hemmungen bei der Implementierung führen, wenn zu viele Sonderfälle gleichzeitig bedacht werden. Es ist also sinnvoll, von einem Normalfall aus zu generalisieren und sich die Sonderfälle für die nächsten Tests aufzuheben.

Triangulation: Das letzte und einfachste Pattern ist die Triangulation. In diesem Fall kennt der Programmierer zwar das Problem, jedoch ist ihm keine allgemeine Lösung oder Implementierung bekannt. Hierbei setzt das Pattern der Triangulation an dem simpelst möglichen Fall an, um von dort aus auf die finale Lösung zu kommen. Hierfür werden verschiedene Standpunkte mithilfe von unterschiedlichen Tests erzeugt, welche dann die richtige Funktionalität voraussetzen.
Dieses Pattern startet mit dem simpelsten Fall als Ausgangspunkt. Für diesen Fall wird, ähnlich wie beim „Fake-it"-Pattern, die Lösung genau dieses Tests implementiert. Nun wird ein weiterer Ausgangspunkt erzeugt, welcher eine Abstraktion in der aktuellen Lösung erfordert. Dieser Vorgang wird dann wiederholt, bis kein weiterer Test eine Abstraktion der Lösung erfordert und somit die finale Funktionalität implementiert ist. Der Vorteil dieses Vorgehens ist die klare und einfache Struktur:

- Schritt 1: Schreibe eine Funktion, welche den vorhandenen Test erfüllt.
- Schritt 2: Erweitere den Test um ein weiteres Beispiel.
- Schritt 3: Passe die Funktion an beide Beispiele an.
- Wiederhole die Schritte 2 und 3, falls ein weiterer Testfall fehlschlägt.

Dies ist dabei die aufwändigste Möglichkeit, verwendet jedoch einen klaren und einfachen Weg, um an eine performante Implementierung des Problems zu kommen. Ein Beispiel, wie das Pattern der Triangulation verwendet wird, ist in Appendix C zu finden.

Nachdem nun die Green-Bar-Patterns beschrieben und die Green-Phase damit absolviert wurde, folgt nun die Phase des Refactorings. In dieser Phase wird der funktionierende Code schrittweise bereinigt. In der Softwareentwicklung darf beim Refactoring die Funktionalität nicht verändert werden. Dies kann im TDD sichergestellt werden, indem nach jedem Schritt die Test-Suite ausgeführt wird.
Beim Refactoring werden die folgenden Schritte ausgeführt:

- Auflösen von doppelten Codepassagen
- Verschieben von Codepassagen an die richtigen Stellen
- Logische Benennung von Klassen und Funktionen
- Extraktion von Methoden
- Einführung und Neuanordnung von Vererbungshierarchien

Um das Vorgehen im klassischen TDD zu veranschaulichen, wird im folgenden Abschnitt eine Beispieliteration beschrieben.

Das Vorgehen im klassischen TDD

Die Beispieliteration bezieht sich auf die Implementierung von einer Addition von unterschiedlichen Währungen mit einem Währungskurs. Wir betrachten also das konkrete Beispiel (vgl. [15]):

$$5\$ + 10\ CHF = 10\$$$

bei einem imaginären Währungskurs von 2:1.

Um diese Funktionalität zu erfüllen, wird sowohl die Addition zweier gleicher Währungen benötigt als auch die Multiplikation einer Währung mit dessen Währungskurs. Diese Überlegungen werden in der Vorbereitung direkt in Tests umformuliert und mit der Erstellung der Testliste begonnen. Die Test-Liste umfasst daher die drei folgenden Testfälle:

- 5$ + 10 CHF= 10$
- 5$ * 2 = 10$
- 5$ + 5$= 10$

Nach dem Mantra des TDD, Red-Green-Refactor, beginnt die Entwicklung in der Red-Phase. Nach dem Pattern des One-Step-Tests kann sich z. B. für den zweiten Test entschieden werden. Es wird also mit einem einfachen Test für die Multiplikation begonnen:

```
1  public void testMultiplication() {
2    fiveDollar = new Dollar(5);
3    fiveDollar.times(2)
4    assertEquals(10, fiveDollar.amount)
5  }
6
```

Listing 1: Einfacher Test der Multiplikation (vgl. [15])

Wenn dieser Test nun ausgeführt wird, kommen die folgenden Fehlermeldungen:

- No class “Dollar”
- No constructor
- No method “times(int)”
- No field “amount”

Dies wird beim TDD erwartet, da in der Red-Phase noch kein funktionaler Code implementiert wird. Dieser wird in der nächsten Phase, der Green-Phase, mit möglichst wenig Aufwand ergänzt. Hierfür werden zunächst Grundkonstrukte geschaffen, welche die Fehlermeldungen beheben:

```
class Dollar{
  private int amount;

  Dollar(int amount){
  }

  public double times(int multiplier) {
  }
}

```

Listing 2: Das Konstrukt der Klasse „Dollar“ (vgl. [15])

Wenn der Test nun ausgeführt wird, erhält man das gewollte Ergebnis: Der Test compiliert, schlägt jedoch bei der Assertion fehl. Dies wird ebenfalls erwartet, da die Klasse „Dollar“ und dessen Funktionen leer sind. Um den Test nun zu bestehen, wird z. B. die Obvious Implementation verwendet, indem der Konstruktor und die Multiplikation implementiert werden:

```
class Dollar{
  private int amount;

  Dollar(int amount){
    this.amount = amount
  }

  public double times(int multiplier) {
    this.amount = amount * multiplier
  }
}

```

Listing 3: Die implementierte Klasse „Dollar“ (vgl. [15])

Während der Implementierung ist dabei auffällig, dass in der Klasse „Dollar“ ein Integer als Parametertyp gewählt wurde und dass in der *times*()-Funktion der Wert des Attributes überschrieben wird, was beim mehrfachen Ausführen der Funktion zu Problemen führen kann. Solche Auffälligkeiten sollen die aktuelle Iteration nicht aufhalten. Sie werden, dem Another-Test-Pattern entsprechend, zur Test-Liste hinzugefügt und in folgenden Iterationen behandelt.
Der aktuellen Iteration folgend, kommt nun die Refactoring-Phase. Da diese Beispieliteration nur Teile des funktionalen Codes beinhaltet, werden im Folgenden einige Beispiele und mögliche Lösungen gegeben, welche in der Refactoring-Phase möglich sind:

- Der Code von Dollar existiert bereits in einer anderen Währungs-Klasse: Hierbei könnte eine abstrakte Superklasse erstellt werden und die entsprechende Funktionalität in diese verschoben werden.

- Die Funktionen und Klassen sind für deren Kontext nicht richtig benannt: Umbenennen der Klassen und Funktionen an allen vorkommenden Stellen.
- Funktionen sind zu lang, sodass sie mehr als eine Funktionalität erfüllen: Extrahiere einen Teil der Funktion, um eine gute Lesbarkeit und Verständlichkeit zu gewährleisten.

Abschließend sieht die Test-Liste mit den Ergänzungen dann wie folgt aus:

- 5$ + 10 CHF= 10$
- ~~5$ * 2 = 10$~~
- 5$ + 5$= 10$
- Dollar.times() überschreibt amount
- Dollar int -> double + round(2)

Nun könnte mit dem nächsten Zyklus begonnen werden, wobei sich die in der letzten Iteration gefundenen Testfälle als nächste Optionen anbieten, da sie die soeben erstellte Klasse Dollar betreffen. Da dieses Beispiel nur eine Iteration beinhalten soll, ist der folgende Test in Appendix D nachzulesen.

2.2.2 Voraussetzungen des TDD

Das TDD bringt implizit die folgenden Voraussetzungen mit (vgl. [15]):

- Es muss dynamisch programmiert werden, da der laufende Code über das Feedback der Tests zwischen einzelnen Entscheidungen angepasst wird.
- Die Tests müssen selbst geschrieben werden.
- Die Tests müssen direktes Feedback geben, da auch kleine Änderungen im Code die Tests beeinflussen können.
- Das Softwaredesign muss aus lose gekoppelten Komponenten bestehen, damit sich die Tests nicht gegenseitig beeinflussen.

2.2.3 Die Vorteile im TDD

Beim Vorgehen, dass funktionaler Code nur dann geschrieben wird, wenn ein fehlgeschlagener Test diesen erfordert, wird der Programmierer dazu bewegt, eine geringere Defektdichte zu erzeugen. Mit Defektdichte ist hierbei Anzahl des unerwünschten Aufkommens von ungewolltem Verhalten im Code gemeint. Durch diese Reduzierung können folgende Errungenschaften vollzogen werden (vgl. [15]):

- Falls das Aufkommen von Defekten weit genug reduziert werden kann, bewegt sich die Qualitätssicherung von reaktiver Arbeit zu proaktiver Arbeit.
- Falls die Anzahl von unerwarteten Situationen weit genug reduziert werden kann, können die Projektleiter besser abschätzen, wann echte Kunden frühzeitig in die Entwicklung eingebunden werden können.
- Wenn die Themen technischer Konversationen durch die Kommunikation in Tests klar genug gemacht werden können, können die Programmierer in einer besseren Kollaboration miteinander arbeiten.
- Falls das Aufkommen von Defekten weit genug reduziert werden kann, kann täglich produktive Software mit neuen Funktionalitäten geliefert werden.

Neben diesen eher allgemeinen Vorteilen, welche das TDD für ein Projekt mit sich bringt, birgt es vor allem auch Vorteile für den Programmierer. Diese werden in den folgenden Paragraphen beschrieben.

Sorgenfreies Testen: Der offensichtliche Vorteil beim TDD ist, dass sich kein Abstand zwischen funktionalem und getestetem Code bilden kann. Der Programmierer muss sich im Nachhinein keine Gedanken darüber machen, ob und welche Passagen seines Codes ausreichend getestet sind.

Sicherheit im Fall der Unsicherheit: Im TDD kann jederzeit durch das Ausführen der Tests Sicherheit gewonnen werden, dass der aktuelle Code funktionstüchtig ist. Durch Unsicherheit kann der Programmierer in eine defensive Position gebracht werden, in der er vorsichtig, weniger kommunikativ und schüchtern gegenüber Feedback wird. Sobald Zweifel oder Unsicherheit entsteht, kann der Programmierer im TDD die Tests laufen lassen, um Sicherheit zu gewinnen, dass das, woran er arbeitet, funktioniert. Dies wirkt sich positiv auf die Mentalität des Programmierers aus, wodurch dieser hemmenden Angst des Programmierers entgegengewirkt wird.

Die Schwierigkeit ist variabel: Egal, ob Programmierer mit langjähriger Erfahrung oder Neuling, das Vorgehen im TDD ist variabel und kann durch die Wahl der Patterns sehr gut an die eigenen Fähigkeiten und die Schwierigkeit des zu schreibenden Codes angepasst werden.

Keine Suche nach einem Startpunkt: Beim Programmieren kann es vorkommen, dass der Programmierer sich in einer Situation wiederfindet, in der er nicht weiß, wo er anfangen soll. Dies kann daran liegen, dass der geforderte Task zu komplex ist oder zu viele Grenz- oder Sonderfälle birgt. Hierbei hilft das TDD, indem es einen strukturierten Anfang, die

Erstellung der Test-Liste, liefert. Man beginnt also bei der Frage, was die Software können soll und formuliert dies in dem ersten Test. Aus diesem kann bereits mit dem TDD gestartet werden. Im wahrscheinlichen Fall, dass dieser noch zu komplex ist, wird er in weitere kleinere Tests unterteilt und der Programmierer kann, sobald er mit der Größe der Tests zufrieden ist, mit der ersten Iteration beginnen.

Arbeiten in einem Team: Die Kommunikation innerhalb eines Teams, aber auch mit Außenstehenden, kann ebenfalls erleichtert werden, da Tests eine intuitive und klare Sprache der Kommunikation bilden. Durch die Architektur und der Orientierung an den Tests wird das Vermitteln der Anforderungen an den Code sehr einfach und verständlich. Anhand eines Tests können sowohl Programmierer als auch Nicht-Programmierer gut kommunizieren, wie die Funktionalität der Software aussehen muss. Auch die Programmierer unter sich können auf diese Weise Probleme sehr leicht kommunizieren und somit einen flüssigen Workflow gewährleisten.

2.2.4 Die Limitationen und Gegenargumente beim TDD

Nach den Vorteilen müssen nun auch die Limitationen und Argumente gegen das TDD erläutert werden.
Ein wesentlicher und das häufig aufgeführte Argument gegen das TDD ist, dass es durch das exzessive Testen mit deutlich mehr Aufwand verbunden ist. Dieses Argument kann zum Teil entkräftet werden, da in der Vorbereitung auch ohne das TDD eine Aufzählung der zu erfüllenden Voraussetzungen gemacht wird. Diese können mit nur ein wenig mehr Aufwand über Beispiele in die Form einer Test-Liste gebracht werden. Dies birgt dabei den Vorteil einer besseren gedanklichen Strukturierung der Anwendung. Des Weiteren führt dies mit dem TDD zu einem Verständnis, in welcher Reihenfolge die Funktionen implementiert werden müssen, da sich hierfür an einem Testfall orientiert werden kann.
Ein weiteres Argument gegen das TDD ist die Voraussetzung, dass der Entwickler seine Tests selbst schreiben muss. Da der Entwickler sowohl Tests als auch den zugehörigen Code schreibt, können sich sogenannte blinde Flecken (engl. blind spots) bilden, falls der Entwickler einen Sonderfall nicht berücksichtigt hat. Dies geschieht z. B. wenn der Entwickler die Validierung eines bereits bestehenden Parameters nicht überprüft. Das Problem dabei ist, dass der Test wahrscheinlich trotzdem bestanden wird, wodurch ein falsches Gefühl der Korrektheit entsteht.
Des Weiteren birgt die exzessive Nutzung von Tests einen laufenden Mehraufwand, da diese gewartet werden müssen. Die Wartung von vielen detaillierten Tests kostet Zeit. Daher kann dies als Argument gegen das TDD aufgeführt werden. Falls es die Situation und die Anwendung erlaubt, kann es mehr Sinn machen, generellere Tests zu schreiben, welche ledig-

lich extreme Konditionen und gewählte Stichproben der Daten testen. Dieses Vorgehen birgt dabei jedoch das Risiko der Unvollständigkeit.
Ein weiterer kritischer Punkt im TDD ist, dass die Tests, welche in frühen Iterationen geschrieben wurden, zu einem späteren Zeitpunkt schwierig zu reproduzieren sind. Daher sind diese Tests und deren Korrektheit zu einem späteren Zeitpunkt in der Entwicklung sehr wertvoll. Diese aufgrund von einer Änderung, welche viele Tests fehlschlagen lässt, zu korrigieren, ist mit einem großen Arbeitsaufwand verbunden, da jeder Test individuell geändert werden muss. Außerdem können bei solchen Veränderungen unbemerkte blind spots in der Testabdeckung aufkommen.

Bezüglich der Limitationen vom TDD ist zu sagen, dass es in manchen Situationen, durch die Fokussierung auf Unit-Tests, schwierig umzusetzen sein kann. Dies betrifft besonders Fälle, in denen mittels TDD User-Interfaces, Programme mit Datenbanken oder bestimmte Netzwerkkonfigurationen getestet werden sollen. In diesen Fällen wird allerdings auch die Voraussetzung des TDD verletzt, dass die Anwendung aus lose gekoppelten Komponenten besteht.

2.3 Conversational AI

In diesem Abschnitt wird beschrieben, was Conversational AI umfasst und wie sie in Unternehmen verwendet werden kann. Hierzu wird darauf eingegangen, wie eine Conversational AI funktioniert und wie sie sich in den letzten Jahren entwickelt hat, sodass sie nun vielversprechend in vielen Anwendungsbereichen verwendet wird.
Zunächst sollte der Begriff „Conversational AI“ genauer definiert werden, da es viele unterschiedliche Definitionen gibt. Die in dieser Arbeit verwendete Definition ist: Conversational AI sind durch künstliche Intelligenz (KI) gestützte automatisierte Dialogsysteme (vgl. [16]). Diese werden umgangssprachlich auch Chatbots bzw. text- und sprach-basierte Assistenten genannt. Bei diesen kann unterschieden werden, ob sie mit oder ohne eine KI arbeiten. Chatbots ohne KI werden regelbasierte Chatbots genannt, da sie auf festgelegten Regeln basieren, welche den Nutzer durch die vordefinierten Dialoge führen. Die KI-basierten Chatbots verwenden Künstliche Intelligenz, um sowohl den Input des Nutzers zu erkennen als auch um eine dem Kontext entsprechende Antwort darauf zu geben. Hierin liegen auch die Vorteile der KI-gestützten Variante. Diese kann den Nutzer durch deutlich diversere Dialoge führen und diese an die Antworten des Nutzers anpassen. Im Folgenden dieser Arbeit wird der Begriff „Chatbot“ mit der KI-gestützten Variante verbunden.

2.3.1 Die Funktionsweise einer Conversational AI

Eine Conversational AI besteht aus zwei primären Tasks: Erkennung des Inputs des Nutzers in Form von Intentionen und Entitäten und die Reaktion auf diesen Input im Kontext der bisherigen Konversation. Für die Erkennung der Intentionen und Entitäten bedient sich die Conversational AI bekannter Methoden des NLU, wie der Tokenisierung und Featurization. Mit diesen Methoden wird der Textinput in die Form von numerischen Vektoren gebracht, um somit von Maschinen verarbeitet werden zu können. Diese Vektoren werden Worteinbettungen (engl. word embeddings) genannt. Die einfachste Form von Word-Embeddings sind Bag-of-Words-Vektoren. Diese Vektoren bestehen aus einem One-hot-Encoding der im Input vorkommenden Wörter in Bezug zu allen bekannten Wörtern. Hierin liegt auch der Nachteil dieser Vektoren, da keinerlei Unterschied gemacht wird, wie ein Wort im Kontext des Satzes verwendet wird. Um dies zu verbessern, können sogenannte Transformer (siehe Kap. 4.3) verwendet werden, um kontextualisierte Word-Embeddings zu erzeugen. Diese erzeugten Vektoren werden dann verwendet, um mithilfe von ML sowohl die Intentionen der Nutzer zu klassifizieren als auch eventuell vorkommende Entitäten, wie z. B. Namen oder Ortsangaben, zu erkennen.

Ein Beispiel wäre der Satz: „Spiele die Bohemian Rhapsody von Queen“. Hierbei sollte das

Modell sowohl die Intention „etwas abspielen“ als auch die Entitäten Songname „Bohemian Rhapsody“ und Band „Queen“ erkennen.

Für die Reaktion der Conversational AI werden dann sogenannte Richtlinien (engl. policies) verwendet. Diese Richtlinien entscheiden mithilfe der Intentionen und Entitäten, welche Aktion die Conversational AI zu jeden Zeitpunkt der Konversation unternimmt. Neben den regelbasierten Richtlinien gibt es hierbei ebenfalls ML-basierte Richtlinien, welche eine Reaktion vorhersagen. Im obigen Beispiel sollte dieses Modell erkennen, dass eine Aktion gefordert wird, welche den Song aus den Entitäten erkennt und abspielt.
Wie eine Conversational AI genau funktioniert, wird im Kapitel 5 dieser Arbeit genauer beschrieben und erklärt.

2.3.2 Die Entwicklung von Conversational AI

Die Fragestellung, warum Conversational AI verstärkt verwendet wird, stammt aus der übermäßigen Nutzung von Apps für Kunden. Durch die permanente Nutzung von Smartphones im Alltag haben Unternehmen versucht sich durch die Bereitstellung Apps an diesen Trend anzupassen. Der Erfolg dieser Apps blieb jedoch aus. Das Problem war, dass sehr viele Apps auf den Markt kamen und der Nutzer nur ungern weitere hinzugefügt hat. Die Lösung dieses Problems verspricht die Bereitstellung von Conversational AI, welche über Messaging Plattformen agieren können, welche üblicherweise bereits installiert sind (vgl. [17]).
Hinzu kommt, dass die Nutzer durch große Unternehmen, wie Google und Apple, bereits an die Verwendung von sprach- oder textbasierten Assistenten gewöhnt sind. 21% der Erwachsenen in den USA und mehr als 80% der Generation Z nutzt bereits solche Assistenten, um Informationen zu bekommen oder online einzukaufen. Hierdurch hat sich der Marktumfang von Conversational AI in den letzten Jahren ständig vergrößert. 2017 waren es rund 250 Millionen US-Dollar und bis 2024 soll dieser Wert auf ca. 1,34 Milliarden US-Dollar anwachsen (vgl. [4]).

Aufgrund dieser Nutzungstrends und dem finanziellen Nutzen wird Conversational AI immer häufiger verwendet. Im Folgenden wird beschrieben, wie sich Conversational AI entwickelt hat und wie sie State-of-the-art verwendet wird.
Die Entwicklung der Conversational AI lässt sich durch Stufen der Komplexität beschreiben. Hierbei wurden 5 Stufen identifiziert, welche sich durch die jeweilige Leistung der Conversational AI unterscheiden (vgl. [18]):

Benachrichtigungsbots: Die erste Stufe der Conversational AI, welche von Unternehmen bereits vor ca. 10 Jahren genutzt wurde, sind Benachrichtigungsbots. Diese versenden automatisiert Benachrichtigungen an Kunden. Sie beschreiben die simpelste Form von

Conversational AI und fallen unter die Kategorie der regelbasierten Chatbots. Bei diesen Bots werden, nach vorheriger Absprache, automatisierte Erinnerungen und Benachrichtigungen über Änderungen und Neuerungen der abonnierten Themengebiete versendet.

FAQ-Bots: Die zweite Stufe der Conversational AI sind die FAQ-Bots. Diese für den heutigen Standard noch sehr simplen Bots werden genutzt, um den Kundensupport zu unterstützen. Mit ihnen wird den Kunden eine Möglichkeit gegeben, Antworten zu erhalten, ohne sich direkt mit einem Mitarbeiter des Unternehmens in Verbindung setzen zu müssen. Die Basisstufe dieser Bots sind vergleichbar mit der FAQ-Seite einer Website mit einer Suchfunktion. Eine regelbasierte Erweiterung dieser Bots erlaubt es ihnen, auf simple Folgefragen zu reagieren. Ein Beispiel einer solchen Konversation könnte sein:

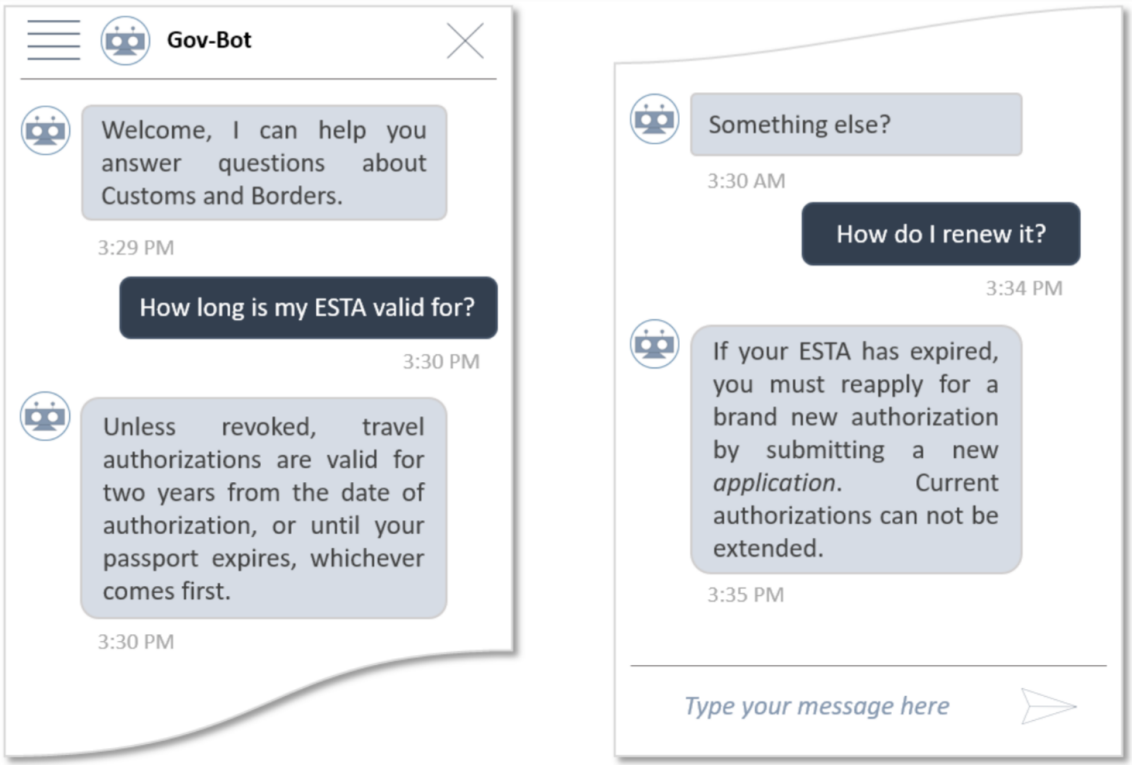

Abbildung 2: Beispiel eines FAQ-Chatbots mit einer Rückfrage [19]

Slot-basierte Bots: Die dritte Stufe der Conversational AI sind die slot-basierten Chatbots. Diese versuchen die Tasks zu vollziehen, welche bei einem FAQ-Bot nur angefragt wurden. Hierzu benötigt die Conversational AI eine Art Gedächtnis. Dieses Gedächtnis äußert sich in Form von Slots. Der Assistent kann, ähnlich wie in der vorherigen Stufe, Sprache klassifizieren, um zu erkennen, welche Intention oder Frage der Nutzer verfolgt.

Nun können jedoch zusätzlich Entitäten, wie z. B. Namen oder Orte, erkannt und in Slots gespeichert werden. Somit können notwendige Slots für einen Task abgefragt und dieser dann ausgeführt werden. Ein Beispiel hierfür sind Wetterbots, wobei der Nutzer einen Ort und ein Datum für die Wettervorhersage geben muss:

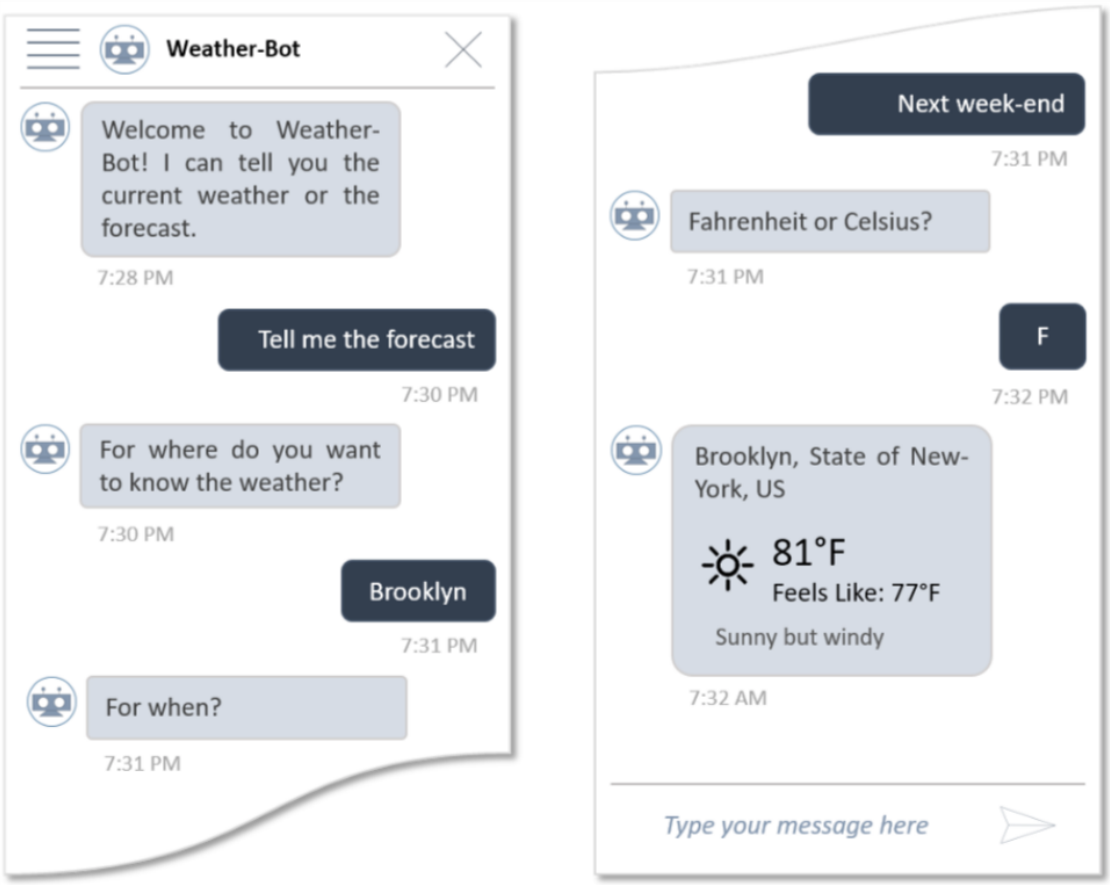

Abbildung 3: Beispiel eines slot-basierten Chatbots [19]

Kontext-Assistenten: Eine Erweiterung der slot-basierten Form der Conversational AI sind die Kontext-Assistenten (engl. contextual assistants). Diese Form unterbricht den statischen Aufbau der Konversationen. Dies ermöglicht es zuverlässiger auf Zwischenfragen zu reagieren oder sogar temporär das Thema zu wechseln. Dies wird in Form von baumartigen Dialogstrukturen umgesetzt, daher werden diese Assistenten auch baumbasierte Chatbots (engl. tree-based chatbots) genannt. Da Konversationen durch ihre Intentionen abstrahiert sind, können hierdurch mehrere Szenarien abgefangen werden, indem die Konversationen sich nicht grundsätzlich unterscheiden müssen, sondern baumartig aufgebaut werden. Somit ist es z. B. möglich während eines Dialogs in eine andere Konversation zu wechseln und nach Abschluss dieser mit der ursprünglichen Konversation fortzufahren.

Eine Beispiel einer solchen Konversation könnte der folgende Verlauf sein:

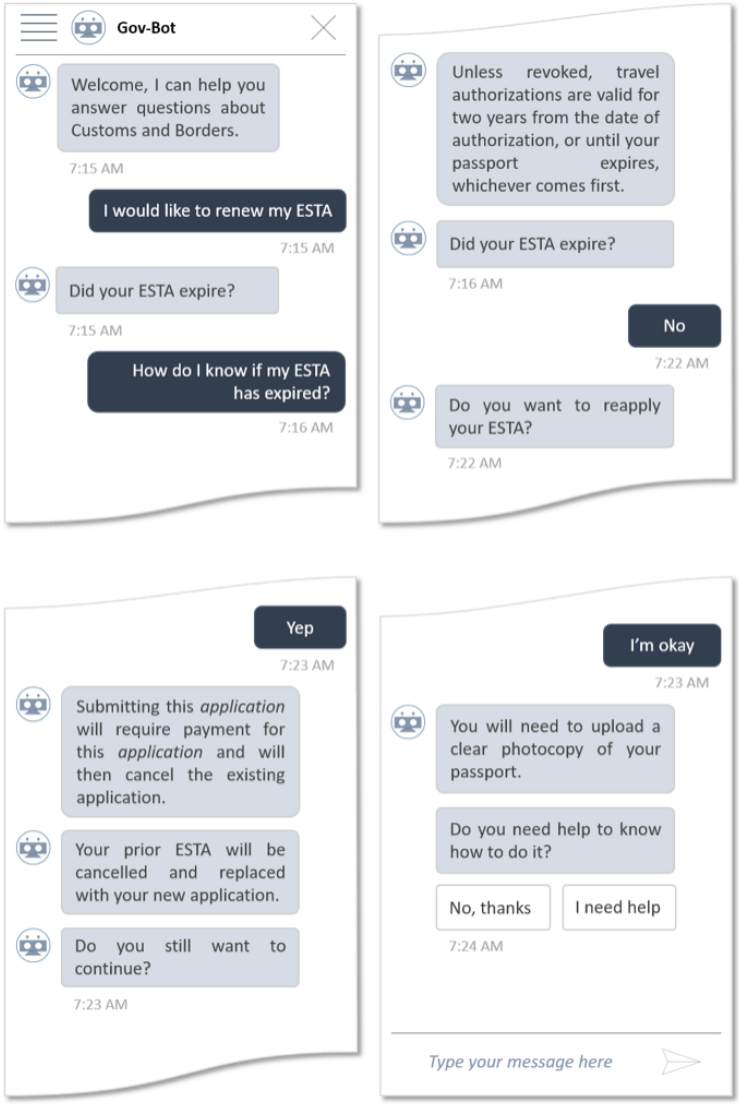

Abbildung 4: Beispiel eines Kontext-Assistenten für die Einreisegenehmigungen der USA [19]

Diese Form der Conversational AI ist für kleine bis mittelständige Unternehmen die praktikabelste Form und wurde in dieser Arbeit durch den praktischen Anwendungsfall umgesetzt. Natürlich gibt es spezialisierte Unternehmen, welche auch höhere Formen der Conversational AI anbieten können. Solche Anwendungen umfassen eine AI, welche mehrere Domains und viele Tasks gleichzeitig bedienen kann. Beispiele hierfür sind der Google-Assistant, Amazon's Alexa oder Siri von Apple. Solche Assistenten sind für die viele mittelständige Unternehmen jedoch nicht umsetzbar, da sie einen hohen Aufwand und Expertise in der Entwicklung benötigen und zusätzlich eine sehr große Datengrund-

lage für das ML benötigt wird.

Personalisierte Assistenten: Diese angesprochenen Formen sind die vierte Stufe der Conversational AI. Sie sind sogenannte personalisierte Assistenten. Diese zeichnen sich dadurch aus, dass sie den Nutzer kennenlernen und sich an dessen Präferenzen anpassen. Diese sind für Unternehmen erstrebenswert, da sie einzelnen Nutzern personalisierte Angebote vermarkten können. Die Problematik hierbei ist, dass diese personalisierten Assistenten einen höheren Standard der Konversation erfordern, da eine Vielzahl von Eigenschaften der Nutzer erkannt und ausgewertet werden müssen, um eine positive Zufriedenheit der Nutzer zu erreichen. Dies resultiert in an einzelne Nutzer angepasste Konversationsverläufe, welche, um einen fließenden Dialog zu erhalten, sehr große Mengen an Trainingsdaten benötigen. Eine Hürde ist hierbei noch, dass diese Assistenten stark mit dem Datenschutz kollidieren, da viele personenbezogene Daten erlernt, gespeichert und ausgewertet werden müssen.

Autonome Organisationen: Die letzte Stufe der Conversational AI sind komplette autonome Organisationen. Dies ist eine Zukunftsvision von Rasa, einem deutschen Open-Source-Plattform-Betreiber für Conversational AI. Autonome Organisationen sind heutzutage schon gut vorstellbar an einem Beispiel eines Unternehmens in der Versicherungsbranche. Mithilfe von Assistenten im Bereichen des Marketings, Verkauf und Finanzen könnten Unterschriften eingeholt, Schadensfälle aufgenommen und personalisiertes Marketing betrieben werden, um einen großen Teil des Tätigkeitsfeldes dieser Unternehmen abzudecken.

2.3.3 Kategorien und fachlicher Fokus von Conversational AI

Einer der größten Anwendungsbereiche, wie auch der praktische Anwendungsfall dieser Arbeit, befindet sich im Kundensupport. In diesem Bereich kann Conversational AI in vielen Wegen verwendet werden, um dem Kunden, aber auch den Mitarbeitern, einen Nutzen und einzigartige Vorteile zu bringen. Diese befinden sich dabei vor allem in der Automatisierung des Kundensupports und der vom Unternehmen initiierten Kommunikation. In diesem Bereich hat Conversational AI den Vorteil, dass sie sowohl viele Kunden auf einmal bedienen kann als auch nie frustriert oder müde wird, wie es ein Nachteil menschlicher Mitarbeiter sein kann (vgl. [4]).
Im Folgenden wird eine Übersicht gegeben, welche Kategorien von Conversational AI es gibt. Diese Kategorien der Conversational AI richten sich dabei nach dem fachlichen Fokus des Assistenten:

- Virtuelle persönliche Assistenten (VPAs): Diese sind dazu konzipiert, dem Nutzer zu helfen, verschiedenste Aufgaben zu erfüllen. Sie basieren auf gesprochenem oder geschriebenem Input, welcher mithilfe von Datenbanken und Präferenzen des Benutzers zu einem bestimmten Task zugeordnet wird. Beispiele sind die üblichen Sprachassistenten Amazon Alexa, Siri von Apple oder der Google-Assistant.
- Spezialisierte digitale Assistenten (SDAs): Diese Assistenten fokussieren sich auf einen spezifischen, von der Domain abhängigen, Task. Beispiele wären Wetterbots oder spezifische Chatbots im Kundensupport. Unter diese Kategorie fällt auch der für diese Arbeit umgesetzte Chatbot.
- Verkörperlichte Assistenten (engl. embodied conversational agents, ECAs): Diese Assistenten haben einen animierten Avatar im Fokus. Dieser kann dabei die Sprache mit Mimik und Gestik kombinieren.
- Chatterbots: Diese Chatbots haben den Fokus auf Small-Talk und realistischen Gesprächen. Ein Beispiel hierfür ist Cleverbot[5], eine Conversational AI, welche versucht möglichst menschlich zu wirken.

In Abbildung 5 wird eine Übersicht über diese vier Kategorien grafisch mit einem skalierten Fokus verdeutlicht. Hierbei ist zu erkennen, dass der Anwendungsfall dieser Arbeit zu der in gelb markierten Kategorie der SDAs gehört, da dieser sich auf eine textbasierte Task-Orientierung fokussiert.

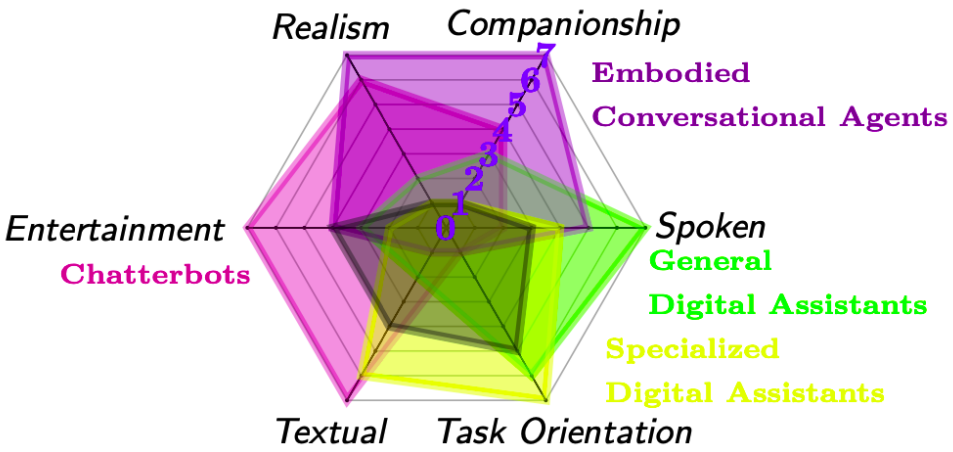

Abbildung 5: Verteilung des Fokus der verschiedenen Kategorien von Chatbots [17]

[5] https://www.cleverbot.com

3 Methodik des TDD im Kontext des Machine Learning

Nachdem die Grundlagen erläutert wurden, wird nun der Schwerpunkt dieser Arbeit angegangen. In diesem Kapitel wird beschrieben, wie das TDD an die Entwicklung einer Conversational AI angepasst werden kann. Hierfür wird das Vorgehen beschrieben, wie das TDD in der Entwicklung einer Conversational AI verwendet wird und, ähnlich wie beim klassischen TDD, auf unterschiedliche Patterns eingegangen, welche für die Implementierung verwendet werden. Dieses Vorgehen wird dann in Kapitel 5 angewendet.

3.1 Die Vorgehensweise

Um die Vorgehensweise zu veranschaulichen, werden die folgenden Tabellen 5 und 6 vorgestellt. In diesen wird das Vorgehen des klassischen TDD, der Spalte (A), dem Vorgehen für die Entwicklung einer Conversational AI, der Spalte (B), gegenübergestellt.
In Tabelle 5 wird veranschaulicht, welche allgemeinen Vorbereitungen und Überlegungen notwendig sind, um eine Test-Liste zu erstellen. Diese Test-Liste wird daraufhin in den Zyklen des TDD bearbeitet und umgesetzt. Der Fokus dieser Tabellen liegt dabei darauf, eine Funktionalität in die benötigten Operationen aufzuteilen und diese umzusetzen. Im Fall (A) entspricht dies der Aufteilung der Funktionalität von $5\$ + 10\,CHF = 10\$$ in eine Addition gleicher Währungen und einer Multiplikation einer Währung mit dem entsprechenden Währungskurs. Im Fall (B) werden, um einen Chatbot umzusetzen, ML-Modelle und eine Datengrundlage benötigt. Hierfür kann eine Auswahl passender ML-Modelle recherchiert und die voraussichtliche Datengrundlage evaluiert werden.
Der nächste Schritt in der Vorbereitung ist es, aus diesen Funktionalitäten die Überlegungen anzustellen, wie diese in der praktischen Anwendung der Software umgesetzt werden. Hierzu kann sich an Beispielen oder Testfällen orientiert werden, welchen die Anwendung in der Produktion begegnen wird. Im Fall (A) werden hierfür die benötigten Klassen und Funktionen gefunden und beschrieben, wie diese miteinander interagieren. Im Fall (B) können Überlegungen getroffen werden, welche Intentionen das Modell verstehen muss und wie die typischen Konversationen verlaufen.

Die Vorbereitung für die Erstellung der Test-Liste	
(A) TDD Klassisch	(B) TDD für eine Conversational AI
Benötigte Funktionalität: z. B.: 5$ + 10 CHF = 10$ Herunterbrechen: Funktionalitäten: Addition und Multiplikation von Währungen Zusätzliche Überlegungen: Währungen - Addition, Multiplikation; Exchange Rates für die Multiplikation	Benötigte Funktionalität: z. B.: Chatbot zur Reisekostenabrechnung Herunterbrechen: Auswahl passender Modell-Architekturen und benötigter/vorhandener Daten Zusätzliche Überlegungen: Benötigte Intentionen; Typische Konversationen
Erstellen der Test-Liste	

Tabelle 5: Vorbereitung für die Erstellung der Test-Liste im TDD

Aus dieser Vorbereitung kann nun eine Test-Liste erstellt werden, welche den Ansatzpunkt für die Entwicklung in den Zyklen des TDD gibt. Die Test-Liste für den klassischen Fall (A) könnte aus dem Test für die gesamte Funktionalität, einem Test für die Multiplikation und einem Test für die Addition von Währungen bestehen. Im Fall des TDD für eine Conversational AI wird ebenfalls eine Test-Liste erstellt, jedoch müssen die Tests an das ML angepasst werden. Die Unit-Tests, wie sie im klassischen Fall erstellt wurden, können für das ML nicht verwendet werden, da ein Test nicht die gesamte Datengrundlage eines ML-Modells abdecken kann. Für eine Conversational AI können hierfür z. B. sogenannte Minimum Functionality Tests (MFTs) verwendet werden, welche die Vorhersage für ausgewählte Daten sicherstellt. Diese und weitere Testarten werden im Kapitel 4.1 dieser Arbeit noch genauer beschrieben. Mit solchen Tests kann dann eine Test-Liste erstellt werden, welche z. B. aus den MFTs der Intentionen und End-to-End-Tests für typische Konversationen besteht.

Iteratives Implementieren der Tests durch den TDD-Zyklus	
(A) TDD Klassisch	(B) TDD für eine Conversational AI
Test-Liste: Addition: $5\$ + 5\$ = 10\$$ Multiplikation: $5\$ * 2 = 10\$$	Test-Liste: MFTs für die Intentionen End-to-End-Tests für Konversationen
Red-Phase - get test to compile and fail: Wähle und implementiere den Test Deklariere Klassen und Funktionen	Red-Phase - get test to compile and probably fail: Wähle und implementiere einen Test Wähle oder erstelle ein Modell
Green-Phase - get test to pass: Implementiere die Funktionalität in den Klassen und Funktionen	Green-Phase - get test to pass: Importiere validierte NLU-Daten, erstelle die benötigten Storys und trainiere das Modell
Refactor-Phase: Duplikationen im Code ersetzen Coding Konventionen erfüllen	Refactor-Phase: Tuning der Modellparameter Analyse der Fehler

Tabelle 6: Gegenüberstellung des Vorgehens im TDD

Die Tabelle 6 zeigt das iterative Vorgehen, mit der die Test-Liste bearbeitet wird. Dieser Zyklus wird genutzt, um die Tests und den funktionalen Code zu implementieren.
Nachdem eine Test-Liste erstellt wurde, kann mit der Red-Phase begonnen werden. In dieser Phase wird im klassischen TDD, wie in Kapitel 2.2.1 beschrieben, der in diesem Zyklus bearbeitete Test gewählt und die benötigten Klassen und Funktionen deklariert, sodass das Ziel eines fehlschlagenden Tests erreicht wird. Für den Fall einer Conversational AI muss in dieser Phase zusätzlich eine Funktion deklariert werden, welche eine Vorhersage trifft. Dies könnte zunächst eine leere Funktion sein. In der Praxis bieten sich an dieser Stelle jedoch die Default-Modelle an, welche viele Frameworks zur Verfügung stellen. Dieser Schritt muss dabei nur einmalig ausgeführt werden, da in weiteren Iterationen die Modelle bereits zur Verfügung stehen.
Nachdem ein Test compiliert und wahrscheinlich fehlschlägt, wird mit der Green-Phase begonnen. In dieser Phase wird das Modell angepasst und mit den benötigten Daten angereichert, um den Test zu bestehen. Hierbei unterscheidet sich das Vorgehen vom klassischen TDD, da nicht ein minimaler Aufwand angestrebt wird, sondern ein benötigter Aufwand geleistet werden muss. Es wird also z. B. nicht die minimale Anzahl der Daten verwendet, sondern das Modell wird so angepasst und mit so vielen Daten angereichert, sodass es alle implementierten Tests erfüllt. Hierbei können vorherige Tests gegebenenfalls auch wieder fehlschlagen, wobei diese direkt bearbeitet werden müssen, sodass diese Phase mit einem validen Modell beendet wird, welches alle Tests bzgl. der minimalen Anforderungen besteht.
In der abschließenden Phase des Refactorings wird nun der Code und das Modell finalisiert, sodass die Funktionalität gewährleistet wird und der Format den Ansprüchen entspricht. Dies

äußert sich beim ML deutlich umfangreicher, da hierbei sowohl der Code als auch das Modell und die Daten berücksichtigt werden müssen. Somit muss in dieser Phase eine Optimierung der Modellwahl und Parameter getätigt werden. In dieser Phase sollte also für jedes Modell aus den passenden Modellen ein Parametertuning vollzogen werden, um zu evaluieren, welches Modell mit welchen Parametern am besten geeignet ist. Da dies jedoch mit einem erheblichen Zeitaufwand verbunden ist und in frühen Phasen der Entwicklung keinen ersichtlichen Mehrwert bringt, ist es sinnvoll diese Phase zu einem frühen Zeitpunkt in der Entwicklung zu verkürzen. Hierbei ist es ausreichend, ein zulässiges Modell zu finden, welches die aktuellen Anforderungen möglichst erfolgreich erfüllt. Die Modellwahl und Optimierung wird dann zu einem späteren Zeitpunkt, z. B. in Verbindung mit Meilensteinen, in vollem Umfang vollzogen.
Zu diesem Zeitpunkt muss außerdem die Test-Liste der Testeigenschaften des ML aus Abschnitt 2.1.4 ergänzt werden. Um den flüssigen Beginn des TDD nicht zu unterbrechen und da diese Tests ebenfalls erst mit einem hinreichend funktionalen Modell sinnvoll sind, werden diese ergänzt, sobald die ersten grundlegenden Funktionalitäten des Modells implementiert wurden. Diese Ergänzungen dürfen dabei nicht zu spät gemacht werden, da sie Ausschlusskriterien für die Auswahl der Modelle darstellen und somit einen starken Einfluss auf die Entwicklungsphase der Anwendung haben können.

3.2 Patterns des TDD für Machine Learning

Im Folgenden wird genauer erläutert, wie die Patterns des TDD für die Entwicklung von ML übernommen und angepasst werden können. Hierbei ist zu beachten, dass im ML eine starke Abhängigkeit von der Datengrundlage herrscht und somit immer sowohl die Daten als auch das Modell ausschlaggebend sein können, dass ein Test fehlschlägt. Wie die Programmierkenntnisse im klassischen TDD sind daher die Kenntnisse des ML wichtig, da der Entwickler oft aus dem Trainingsprozess die Informationen ziehen muss, woran ein Test fehlschlägt.

Für die Auswahl und Erstellung der Tests gibt es keine Anpassungen, da die Red-Bar-Patterns und die Testing-Patterns sich auch für ML verwenden lassen. Hierbei verändert sich, wie bereits beschrieben, nur, dass hierfür andere Tests implementiert werden, da Unit-Tests sich nicht für ML-Modelle eignen.
In den Green-Bar-Patterns müssen jedoch Anpassungen getroffen werden, da spätestens hier ein funktionstüchtiges ML-Modell angefertigt oder bearbeitet werden muss. In der Green-Phase muss nicht nur der Code des Modells angepasst werden, sondern es müssen ggf. auch die Daten, mit denen das Modell trainiert wird, validiert werden.
Für die Validierung der Daten kann unabhängig von dem gewählten Pattern eine Datenpipeline erstellt werden, welche ebenfalls für die spätere Instandhaltung des Modells in der Produktion verwendet werden kann. Wie diese Datenpipeline genau aussieht, ist abhängig von

der Datengrundlage und kann während der Entwicklung in der Green- und Refactoring-Phase anhand des Feedbacks des Modells genauer definiert werden. Die Datenpipeline wird für die Bereinigung der Datengrundlage verwendet, sodass diese zum Trainieren genutzt werden können. Das Ziel der Datenpipeline ist es, die Daten zu bereinigen und in eine einheitliche Form für die weitere Verarbeitung zu bringen.
Im Allgemeinen können unabhängig von der Datenpipeline Datenvorverarbeitungsschritte, wie z. B. Korrelationstests, angewendet werden, um die allgemeine Aussagekraft der Daten zu prüfen. Diese sind notwendig, um eine Einschätzung zu erhalten, ob und welche Features der Daten für die Entwicklung eines ML-Modells geeignet sind.

Nachdem die Daten validiert und bereinigt sind, kann mit der Entwicklung des ML-Modells in der Green-Phase begonnen werden. Hierfür wurden im praktischen Teil dieser Arbeit die folgenden zwei Patterns verwendet:

Obvious Implementation: Das Pattern der Obvious Implementation kann vom TDD übernommen werden. Dieses Pattern besagt, dass die Anpassung des Modells an den Test in der Weitsichtigkeit des Programmierers liegt. Somit kann auch bei der Verwendung von ML der Programmierer die Tests direkt implementieren. Hierbei gelten jedoch wieder die Voraussetzungen an den Programmierer zu wissen, welches Modell mit welchen Daten in der Lage ist, den Test zu bestehen.

„Fake-it“: Das „Fake-it“-Pattern kann an die Entwicklung von ML angepasst werden, indem das Prinzip des ursprünglichen Patterns erhalten wird. Somit wird bei dieser Anpassung versucht, durch das Fälschen des validen Modells einen Fortschritt zu erreichen. Dementsprechend wird bei diesem Pattern zunächst versucht, durch das Anreichern von möglichst vielen validierten Daten und das Ignorieren von Einschränkungen, wie z. B. dem Effizienzkriterium, so viele Tests zu bestehen wie möglich. Nachdem das Modell trainiert wurde, kann dann untersucht werden, warum bestimmte Tests fehlschlagen oder ignorierte Einschränkungen nicht eingehalten werden können. Nachdem die Ursachen analysiert wurden, müssen diese z. B. durch Anpassung der Parameter des Modells oder einer gründlicheren Bereinigung der Daten behoben werden, um somit zu einem validen Modell zu gelangen.
Um diesen Fortschritt zu erreichen, können ebenfalls die Daten aus den Tests als Trainingsdaten verwendet werden, da die Tests des TDD als eine vom Training unabhängige Instanz betrachtet werden können und somit die Auswertung der Metriken des Modells nicht beeinflussen.

Falls kein valides Modell trainiert werden kann, muss eine detaillierte Auswertung angefertigt werden, warum jedes einzelne Modell fehlschlägt. Mit dieser Auswertung kann dann Rücksprache mit dem Kunden gehalten werden, um durch eine Lockerung der gesetzten Voraussetzungen oder eine Veränderung der Datengrundlage, zu einem Ergebnis zu kommen.

4 Das Setup des Anwendungsfalls

In diesem Abschnitt wird beschrieben, was die Frameworks Rasa und CheckList sind und wie sie für die Implementierung einer Conversational AI genutzt wurden. Für die Implementierung des Chatbots wurde das Framework Rasa verwendet, welches für die Entwicklung der Tests um das Framework CheckList ergänzt wurde. Diese werden in den folgenden Abschnitten 4.1 und 4.2 genauer beschrieben. Im Abschnitt 4.3 wird dann auf die zu Grunde liegenden ML-Modelle und deren Konfiguration eingegangen.

4.1 Framework CheckList

Da das Testen eine besondere Rolle in dieser Arbeit einnimmt, wurde für die Tests das Framework CheckList hinzugezogen, um die Test-Funktionalitäten von Rasa zu erweitern [20]. CheckList ist ein Testing-Tool, welches sich auf das Testen von NLP-Modellen spezialisiert hat. Hierfür hält es verschiedene Funktionalitäten bereit, welche das ausführliche Testen vereinfachen. Diese Funktionalitäten reichen dabei von der Generierung und Perturbierung von Testdaten über die Erstellung der Tests bis zu einer Test-Suite zur einfachen Ausführung und Evaluation aller implementierten Tests. Dies ist von besonderer Bedeutung, da diese Test-Suite auch für die spätere Instandhaltung verwendet werden kann.
Da nicht alle Funktionalitäten von CheckList auch für die deutsche Sprache vorhanden sind und nur einige Funktionalitäten in dieser Arbeit verwendet wurden, wird für eine ausführliche Beschreibung auf das Paper des Frameworks in Quelle [20] verwiesen.
Grundsätzlich gibt es in CheckList drei unterschiedliche Test-Typen. Diese können nach ihrer Erstellung sowohl einzeln als auch in einer Test-Suite ausgeführt werden. Wenn einzelne Tests oder eine Test-Suite ausgeführt wird, kann eine Zusammenfassung ausgegeben werden, welche im Detail darstellt, welche Testdaten fehlgeschlagen und welche bestanden werden. In einem Jupyter Notebook ist auch eine interaktive visuelle Darstellung der Daten möglich.
In CheckList werden folgende drei Test-Typen definiert:

Minimum Functionality Tests (MFT): Die MFTs testen eine minimale Funktionalität des Modells. Hierzu werden Testdaten erstellt, welche gegen einen bestimmten Output des Modells getestet werden. Diese Tests werden z. B. genutzt, um sicherzustellen, dass ein NLU-Modell grundlegend eine bestimmte Intention des Nutzers aus einem Input erkennen kann.

Invariance Tests (INV): Die Invarianztests sind Tests, welche die Testdaten gegen eine bestimmte Veränderung testen. Sie testen, ob eine Änderung im Input eine Änderung im Output erzeugt. Bei den Invarianztests wird dabei erwartet, dass keine Änderung im

Output stattfindet. Ein Beispiel für Invarianztests sind Tests für die Robustheit gegen Tippfehler.

Directional Expectation Tests (DIR): Die Directional Expectation Tests testen, ähnlich wie die Invarianztests, ob eine Änderung im Input eine Änderung im Output erzeugt. Hierbei wird jedoch erwartet, dass die Änderung den Output verändert. Diese Tests werden z. B. verwendet, um Negationen zu erkennen, falls das Modell solche erkennen muss.

Für die effiziente Implementierung dieser Test-Typen hält CheckList einige nützliche Funktionalitäten bereit. Hierzu zählen das Einbauen von Tippfehlern, das Vertauschen von Namen oder Orten, das Löschen und Einsetzen von Punktierung und das gezielte Verändern von Sätzen. Bei Letztem werden gezielt Satzteile oder Wörter in den Daten ausgetauscht, um somit die Testdaten zu diversifizieren. Dies kann manuell durch den Nutzer sein, aber auch mithilfe des vorgefertigten Lexikons, welches vertauschbare Wortgruppen bereithält, wie z. B. Namen, Nachnamen, Orte, Nationalitäten oder Religionen.
Um zu verdeutlichen, wie die Tests mit CheckList erstellt werden, wird im Folgenden je ein Beispiel anhand eines Tests aus der praktischen Anwendung vorgestellt. Hierfür werden repräsentativ die Testdaten für Begrüßungen in je einem Test-Typ implementiert.
Zunächst werden die Testdaten für die Intention „begrüßen“ in einer Liste gespeichert:

```
begruessungen = ['guten Tag','guten Morgen','guten Abend','grüß Gott',
'ich grüße Sie','seien Sie gegrüßt', 'grüß dich','hallihallo','hallo',
'hallöchen','hallöle','hei','hey','heyho','hi','hoi','howdie','huhu',
'moin','moin moin','na du','servus','Tach','Tachchen','Tag',
'salut','sei gegrüßt','Ei Gude''Grüße','Gude','Gude wie',
moin','moin moin','servus','Tach auch','alaaf','helau']

```

Listing 4: Testdaten für Begrüßungen

Mit dieser Liste kann ein MFT für die Intention einer Begrüßung erstellt werden, sodass das Modell bei einem Input aus dieser Liste eine Begrüßung erkennen soll:

```
from checklist.editor import Editor
from checklist.test_types import MFT

#Formatierung der Begrüßungen für einen MFT
ret = editor.template('{hallo}', hallo = begruessungen, labels=2, save=True)
#Erstellen des MFT
MFT_Hallo = MFT(ret.data, labels=ret.labels, name='Begrüßungen', capability='
  Intent Recognition', description='Einfachster Fall')

```

Listing 5: Erstellen eines MFTs für die Intention „begrüßen“

Die Erstellung eines MFTs besteht also aus 3 Zeilen Code. Dies ist wichtig, da im TDD viele solcher Tests implementiert werden und somit eine schnelle Implementierung eines Tests den Zeitaufwand deutlich verringert. Für den Test müssen lediglich die Testdaten in einer Liste gespeichert werden (vgl. Listing 4). Diese Liste wird mithilfe des Editors aus dem CheckList Framework automatisch für die Erstellung des Tests formatiert und mit dem entsprechenden Label versehen (vgl. Listing 5, Zeile 2). Abschließend werden die Daten dieser Formatierung in einem MFT-Objekt gespeichert. Hierbei kann dem Test ein Name, Beschreibung und Kompetenz hinzugefügt werden, um an einem späteren Zeitpunkt zugeordnet werden zu können (vgl. Listing 5, Zeile 5).
Aus den Daten kann wie folgt ein Invarianztest implementiert werden:

```
1  from checklist.perturb import Perturb
2  from checklist.test_types import INV
3
4  t = Perturb.perturb(begruessungen, Perturb.add_typos)
5
6  INV_Robust_Tipp = INV(**t, name = "INV Test Typos", capability="INV")
7
```

Listing 6: Erstellen eines Invarianztests für die Robustheit gegen Tippfehler

Hierbei wird in Zeile 1 die Perturbationsfunktion *add_typos* des CheckList-Frameworks genutzt. Die *pertub*-Funktion gibt jedoch allgemein die Möglichkeit an dieser Stelle jede gewollte Perturbierung durch eine Funktion zu verwenden. In dem Beispiel wird die vorgefertigte Funktion für Tippfehler verwendet, welche zwei benachbarte Zeichen in den Daten vertauscht, um somit Tippfehler zu simulieren. Daraufhin wird das Objekt, analog zu einem MFT, in einem Invarianztest gespeichert.

Ein Directional Expectation Test wird ähnlich wie ein Invarianztest aufgesetzt, jedoch wird nun eine Veränderung im Output erwartet. Daher wird eine Erwartungsfunktion benötigt, welche diese Veränderung misst. In diesem Beispiel werden die Daten durch die Beispielfunktion *add_other_intent* perturbiert, sodass eine Abnahme der Prognosesicherheit erwartet wird:

```
1   from checklist.perturb import Perturb
2   from checklist.test_types import DIR
3   from checklist.expect import Expect
4
5  t = Perturb.perturb(begruessungen, add_other_intent)
6
7  monotonic_decr = Expect.monotonic(label=1, increasing=False, tolerance=0.1)
8
9  DIR_Robust_Intent = DIR(**t, expect=monotonic_decr)
10
```

Listing 7: Erstellen eines DIR Tests

Hierbei wird durch die *monotonic*-Funktion die Abnahme der Sicherheit erwartet, wobei eine Toleranz von $0,1$ berücksichtigt wird.

Auswertung der Tests

Bei jedem Test kann die *summary*-Funktion verwendet werden, um eine Übersicht über die Ergebnisse des Tests zu erhalten. Diese gibt aus, wie viele Testfälle geprüft wurden, wie viele fehlgeschlagen sind und einige Beispiele der Fehlschläge. Wenn in einem Jupyter Notebook gearbeitet wird, kann auch eine ausführliche visuelle Übersicht erstellt werden, welche in Abbildung 6 veranschaulicht ist.

Abbildung 6: Visuelle interaktive Übersicht eines Beispieltests im Jupyter Notebook, welche Begrüßungen erwartet

Bei dieser Funktion wird auf der linken Seite eine Übersicht über den Test gegeben und auf der rechten Seite befindet sich die Liste aller Testfälle und deren Ergebnisse, wobei in diesem Fall eingestellt ist, dass nur fehlgeschlagene Testfälle angezeigt werden. Diese Liste kann durch das Textfeld unten rechts gefiltert werden, wodurch ist eine detaillierte Auswertung der fehlgeschlagenen Testfälle möglich.
Ein wesentlicher Vorteil des Frameworks ist, dass es durch die vielen Optionen gut an die Anwendung angepasst werden kann. Ähnlich wie die *add_other_intent*-Funktion in Listing 7 kann in dem Framework jede Funktion durch selbstgeschriebenen Code ersetzt werden, solange ein passendes Format eingehalten wird. Somit können die genannten Perturbationsfunktionen, aber auch die Erwartungsfunktion des DIR und die Vorhersagefunktion des Modells durch den Entwickler selbst definiert werden.

4.2 Framework Rasa

Rasa ist ein Framework für die Entwicklung von NLU, Dialogmanagement und Integration von textbasierten Assistenten. Rasa stellt die notwendige Infrastruktur und Tools bereit, um textbasierte Assistenten zu entwickeln. Rasa X ist ein ergänzendes Tool, welches ermöglicht, den mit Rasa entwickelten Assistenten interaktiv an Testnutzern zu verbessern. Zusammen ergeben Rasa und Rasa X eine gute Infrastruktur für die Entwicklung und stetige Verbesserung von textbasierten Assistenten.
Nach einer ausgiebigen Recherche nach Tools für die Entwicklung von Chatbots wurde sich für das Framework Rasa entschieden. Rasa hat dabei die Vorteile, dass es eine kostenlose Open-Source-Software ist und zusätzlich über eine ausführliche Dokumentation und aktive Support-Community verfügt. Open Source ist dabei ein Vorteil, da der Code somit einsehbar ist und die Implementierung der Modelle und Parameter nachvollzogen werden können. Außerdem hält Rasa eine Vielfalt an Anpassungsoptionen bereit und unterstützt die Erkennung von einzelnen als auch mehreren Intentionen und sowohl vor-trainierte als auch benutzerdefinierte Entitäten. Ein weiterer Vorteil, welcher Rasa von Konkurrenzsoftware abhebt, ist, dass Rasa ein breites Spektrum an Kanälen bereitstellt, um den entwickelten Assistenten zu veröffentlichen.
Diese Vorteile werden dabei nur unwesentlich durch den Nachteil geschmälert, dass Rasa durch die vielen Anpassungsoptionen als eine verhältnismäßig schwierig zu verstehende Software bewertet wurde (vgl. [21]).

Bevor die Arbeitsweise mit Rasa beschrieben wird, werden einige Bestandteile eines Chatbots erklärt:

Intentionen (engl. intents): Damit ein Chatbot einen Input verstehen kann, müssen die Intentionen des Nutzers definiert werden. Diese Intentionen sind dabei die Output-Klassen des NLU-Modells, in welche die Inputs des Nutzers klassifiziert werden.

Entitäten (engl. entities): Zu den Intentionen kommen Entitäten. Entitäten sind zusätzliche Informationen, welche der Nutzer in seinem Input übergibt. Hierzu zählen z. B. Ortsangaben, Namen oder ein Datum.

Slots (engl. slots): Slots agieren als Gedächtnis des Chatbots. In Slots werden Entitäten für die weitere Verarbeitung gespeichert. Mit den Slots kann der Chatbot auf die Existenz oder auf den Wert in einem Slot reagieren, falls dies notwendig ist.

Storys (engl. stories): Die Storys sind abstrahierte Konversationen, welche der Bot behandeln können soll. Sie bilden die Trainingsdaten für das Dialogmanagement und somit dem Dialog-Modell. Die Konversationen werden dabei durch die Verläufe der Intentionen des Nutzers mit den entsprechenden Reaktionen abstrahiert.

Aktionen (engl. actions): Aktionen sind jegliche Aktionen des Chatbots, welche er als Reaktion auf einen bestimmten Input ausführen kann. Sie bilden somit die Output-Klassen des Dialog-Modells. Aktionen können dabei statische Antworten sein, wie z. B. eine Vorstellung von sich selbst, aber auch selbstdefinierte Reaktionen, welche in sog. Custom-Actions implementiert werden.

Antworten (engl. responses): Antworten sind die statischen Aktionen des Chatbots. In den Storys werden die Antworten bestimmten Dialogen zugeordnet, um nach dem Erkennen eines Verlaufs gegeben zu werden. In den Antworten, wie auch in den Custom-Actions, kann dabei auf die Slots zugegriffen werden, um somit z. B. Feedback für den Nutzer zu geben, was aus dem Input erkannt wurde.

Formen (engl. forms): Mit Formen wird die Funktionalität von Formularen umgesetzt. Wenn z. B. mehrere Slots für einen Task besetzt sein müssen, können diese mithilfe von Formen vom Nutzer abgefragt werden. Ohne eine Form müssten hierfür jegliche Permutationen von bereits erfassten und bisher nicht erfassten Slots in einer Story beschrieben sein. Dies wird durch Formen erleichtert, indem definiert wird, welche Slots von einer Form erfragt werden müssen und diese dann so lange nachfragt, bis jeder Slot besetzt wurde. Somit wird die Anzahl der benötigten Storys erheblich reduziert.

Regeln (engl. rules): Die Regeln bilden eine weitere Verkürzung der Storys, indem bestimmten Intentionen eine gleichbleibende Antwort zugeordnet wird. Somit muss z. B. die Begrüßung und die Verabschiedung nicht am Anfang und Ende jeder Story definiert werden.

Um die Verwendung von Rasa zu beschreiben, werden die benötigten Dateien beschrieben, mit denen der Chatbot durch Rasa trainiert wird.
Diese Dateien werden im Folgenden beschrieben und ein Auszug der Textdateien aus dem praktischen Anwendungsfall gezeigt:

NLU-Datei: Die NLU-Datei enthält die Trainingsdaten für das NLU-Modell. In dieser Datei werden die Intentionen deklariert und mit Beispielen, wie ein Nutzer diese Intention ausdrücken kann, versehen. Hierbei wird in Klammern eine Entität definiert, welche der Extraktor für Entitäten gesondert erkennt, um sie in einer Konversation zu berücksichtigen.
Ein Ausschnitt der NLU-Datei ist in Listing 8 gegeben:

```
nlu:
- intent: begrüßen
examples: |
  - hallo
  - hi
  - guten Tag
  ...
- intent: einreichen
examples: |
  - Ich will ein [Meeting](Anlass) mit einem Kunden abrechnen
  - Ich habe ein [Training](Anlass) gehalten und will dies abrechnen
  - Ich muss eine [Reise](Anlass) abrechnen
  ...

```

Listing 8: Ausschnitt der NLU-Datei

Storys-Datei: Die Storys-Datei beinhaltet die Trainingsdaten für das Dialog-Modell. In dieser Datei wird die Struktur von typischen Konversationen durch deren Intentionen und den entsprechenden Aktionen gespeichert. Hierbei wird zwischen den erhofften Pfaden (engl. happy paths) und unerhofften Pfaden (engl. unhappy paths) einer Konversation unterschieden. Die erhofften Pfade sind dabei die erwarteten Fälle, in denen der Nutzer wie gewollt auf den Bot reagiert. Bei den unerhofften Pfaden handelt es sich um Fälle, in denen der Nutzer z. B. mit Zwischenfragen reagiert oder die Konversation vorzeitig beenden will.

Ein Beispiel, wie eine Story aussieht, ist in Listing 9 gegeben:

```
stories:
- story: beleg happy path
  steps:
  - intent: begrüßen
  - action: utter_hallo
  - intent: einreichen
  - action: beleg_form
  - active_loop: beleg_form
  - active_loop: null
  - action: utter_zeig_Daten
  - action: utter_fragen_korrekt
  - intent: bestätigen
  - action: utter_fertig_bedanken
  - intent: verabschieden
  - action: utter_bye

```

Listing 9: Story des Happy-Paths zum Einreichen einer Abrechnung

Domain-Datei: In der Domain-Datei werden alle Eigenschaften des Chatbots aufgelistet, um somit den Rahmen des Chatbots zu definieren. Sie fungiert dabei auch als eine Übersichtsdatei in welcher alle Intentionen, Entitäten, Slots, Aktionen, Antworten und Formen aufgelistet werden. In ihr selbst werden lediglich die Antworten und Slots definiert, da die anderen Eigenschaften in der NLU-Datei oder in einem Python-Skript beschrieben werden. Bei den Slots wird hierbei angegeben, welchen Datentyp sie beinhalten und ob sie den Verlauf einer Konversation beeinflussen sollen. Bei den Antworten wird angegeben, welchen Textoutput sie ausgeben, wobei Formatierungen unterstützt werden, sodass z. B. die Werte von Slots abgerufen werden können oder dem Nutzer Buttons zum Anklicken zur Auswahl gestellt werden.
Ein schematischer Ausschnitt aus der Domain-Datei ist in folgendem Listing 10 gegeben:

```
intents:
- begrüßen
...
entities:
- Anlass
...
slots:
  Anlass:
    type: any
  ...
responses:
  utter_hallo:
  - text: Hallo, ich bin ein Reisekosten-Assistent. Ich kann dich bei der Abrechnung deiner Reisekosten unterstützen. Schreibe mir einfach, dass du etwas abrechnen willst und wir legen los :)
  utter_default:
  - text: Sorry, ich habe diesen Input leider nicht verstanden.
  ...
actions:
  - action_slot_reset
  - utter_hallo
  - utter_default
  ...
forms:
  beleg_form:
    Anlass:
    - entity: Anlass
      type: from_entity
    ...

```

Listing 10: Ausschnitt aus der Domain-Datei

Rules-Datei: In der Rules-Datei werden die Regeln des Chatbots definiert. Diese Regeln ordnen bestimmten Intentionen eine statische Antwort zu. Somit werden statische Reaktionen abgedeckt, wie z. B. dass auf eine Begrüßung mit einer Vorstellung des Bots reagiert wird, sodass die Storys diese Teile der Konversationen nicht mehr abdecken müssen.
Beispiele dieser Regeln sind in Listing 11 gegeben:

```
rules:
- rule: Rule to map `begrüßen` intent to `utter_hallo`
  steps:
  - intent: begrüßen
  - action: utter_hallo
- rule: Rule to map `verabschieden` intent to `utter_bye`
  steps:
  - intent: verabschieden
  - action: utter_bye
...

```

Listing 11: Beispielregeln aus der Rules-Datei

Durch diese Regeln kann die Beispielstory aus Listing 9 auf den wesentlichen Part, dass die Intention „einreichen“ die Form des Belegs einleiten soll, verkürzt werden:

```
stories:
- story: beleg happy path
  steps:
  - intent: einreichen
  - action: beleg_form
  - active_loop: beleg_form
  - active_loop: null
  - action: utter_zeig_Daten
  - action: utter_fragen_korrekt
  - intent: bestätigen
  - action: utter_fertig_bedanken

```

Listing 12: Verkürzung der Beispielstory durch die Regeln

Config-Datei: Die Config-Datei ist die Konfigurationsdatei. In dieser Datei werden die Komponenten der Datenpipeline und die Konfiguration der ML-Modelle definiert. Zum einen wird in der Pipeline definiert, wie die natürliche Sprache verarbeitet wird, um abschließend von dem NLU-Modell klassifiziert zu werden. Zum anderen werden die Richtlinien definiert, welche das Dialogmanagement bestimmen.
Im folgenden Abschnitt 4.3 wird beschrieben, wie die Modelle konfiguriert werden und wie dies in der Konfigurationsdatei implementiert ist.

4.3 Die ML-Modelle einer Conversational AI

In diesem Abschnitt wird beschrieben, welche ML-Modelle verwendet werden, um einen textbasierten Assistenten zu implementieren. Um den Umfang dieser Arbeit zu beschränken, wird hierbei der Fokus auf die für den praktischen Anwendungsfall ausgewählten Modelle gelegt.

Im Allgemeinen können für einen Chatbot zwei Klassifikationsmodelle trainiert werden. Das erste Modell, das NLU-Modell, bekommt als Input die verarbeiteten Aussagen des Nutzers, um diese den Intentionen zu zuzuordnen. Das zweite Modell, das Dialog-Modell, wird verwendet, um das Dialogmanagement umzusetzen und die Aktionen des Chatbots vorherzusagen.

Datenvorverarbeitung

Um natürliche Sprache für eine Maschine verständlich zu machen, müssen die Inputs des Nutzers in die Form von numerischen Vektoren gebracht werden. In Rasa wird hierzu eine Pipeline verwendet, welche verschiedene Optionen für diese Verarbeitung bereitstellt. Im Allgemeinen wird ein Tokenizer benötigt, welcher die Sprache in sog. Tokens unterteilt und ein Featurizer, welcher die Tokens in die Form eines Vektors umformt. Diese erzeugten Vektoren können dann dem NLU-Modell als Trainingsdaten und Inputs übergeben werden.

Der Tokenizer Der Tokenizer teilt die natürliche Sprache in Tokens. Der einfachste und in den westlichen Sprachen am häufigsten verwendete ist der Whitespace-Tokenizer. Dieser teilt den Satz an den Leerzeichen getrennt in Tokens auf. Wenn ein vortrainiertes Sprachmodell (engl. language model), wie z. B. von Spacy oder MITIE, verwendet wird, kann auch ein Spacy oder MITIE-Tokenizer verwendet werden, welche für das jeweilige Modell optimiert sind. Diese Tokenizer basieren zwar auf dem Whitespace-Tokenizer, haben jedoch weitere vordefinierte Regeln, welche z. B. Abkürzungen wie „U.K.“ berücksichtigen, sodass diese nicht durch die Punkte voneinander getrennt werden.

Der Featurizer Nachdem mit einem Tokenizer der Input in Tokens aufgeteilt wurde, wird ein Featurizer verwendet, um die Tokens in die für eine Maschinen lesbare Form eines Vektors zu bringen. Hierbei kann der Featurizer sowohl Satz- als auch Sequenz-Features erzeugen. Die Satz-Features sind eine Matrix Repräsentation des Inputs in der Form $1 \times \mathit{Feature\ Dimension}$. Jeder Input wird also durch seine in ihm vorkommenden Tokens repräsentiert. Bei den Sequenz-Features wird eine Matrix in der Form $\mathit{Anzahl\ der\ Tokens} \times \mathit{Feature\ Dimension}$ erzeugt. Hierbei kann nachvollzogen werden, in welcher Sequenz die Tokens zueinander standen. Welche dieser Repräsentationen

verwendet wird, wird durch das NLU-Modell entschieden, somit ist es in Rasa möglich, sowohl Sequenz- als auch Bag-of-Words-Modelle zu trainieren.

4.3.1 Das NLU-Modell

Nachdem die natürliche Sprache in die Form von Vektoren gebracht wurde, folgt das NLU-Modell, um den abstrahierten Input der Nutzer in die Intentionen zu klassifizieren. Hierfür stellt Rasa eine Auswahl an Klassifikatoren bereit. Im Folgenden wird der Sklearn-Klassifikator, welcher auf einer Support Vektor Machine (SVM) basiert, und der Dual Intent and Entity Transformer (DIET) genauer beschrieben, da diese im Kapitel 6 miteinander verglichen werden.

SklearnIntentClassifier: Der Sklearn-Klassifikator nutzt die Sklearn-Bibliothek, um eine lineare SVM zu trainieren, welche mit einem Grid-Search-Verfahren optimiert wird.
Eine SVM versucht eine möglichst gute Trennung zwischen zwei Klassen durch eine $n-1$-dimensionale Hyperebene zu finden. Um das Grundkonzept einer SVM zu verdeutlichen, wird die folgende Abbildung 7 herangezogen:

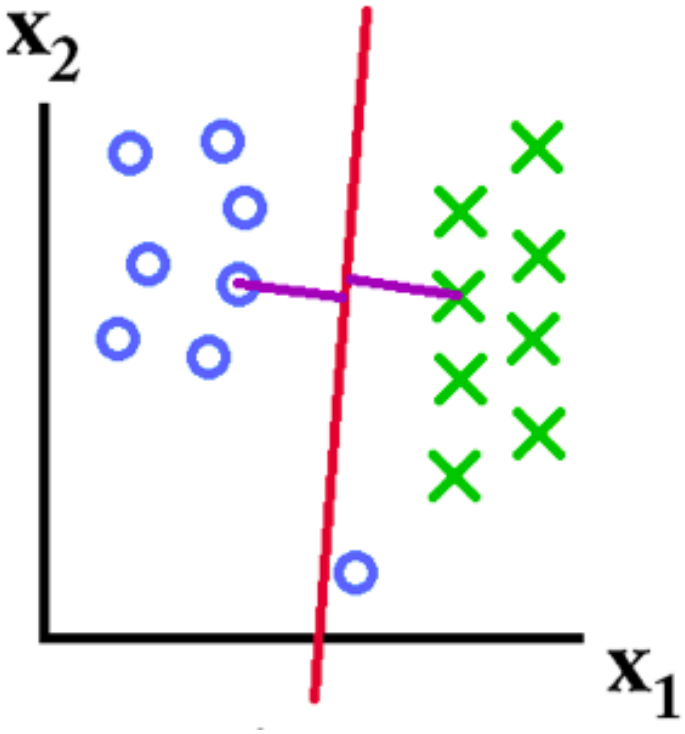

Abbildung 7: Veranschaulichung einer SVM im zwei-dimensionalen Raum

In dieser Abbildung wird veranschaulicht, wie eine SVM im zwei-dimensionalen Raum

versucht eine Trennlinie durch die zwei Klassen zu ziehen. Hierbei versucht das Verfahren eine Hyperebene, in diesem Fall die rote Gerade, zu finden, welche die größte minimale Spanne zwischen den Klassen, die sog. margin in Lila, besitzt. Im Falle des SklearnIntentClassifiers werden die Inputs mit einem Dense-Featurizer verarbeitet. Diese somit erstellten Word-Embeddings, welche zwei Intentionen angehören, versucht der Klassifikator durch die Hyperebene zu trennen, sodass eine Klassifikation anhand dieser Hyperebene möglich wird.
Bei diesem Verfahren wird ein Parameter berücksichtigt, welcher die Schwere einer Fehlklassifikation bestraft, sodass die Breite der Spanne bestimmt werden kann. Dieser Parameter wird dabei durch das Grid-Search-Verfahren mit einer Kreuzvalidierung bestimmt.
Da eine solche SVM nur eine binäre Klassifikation löst, gibt es Erweiterungen des Verfahrens, sodass auch Multi-Class-Probleme mithilfe von SVMs gelöst werden können. In der verwendeten Sklearn-Bibliothek wird dabei das One-vs-One-Schema angewendet, welches je eine SVM für jedes Klassenpaar erstellt. Im Kontext des NLP werden somit für k-Intentionen insgesamt $k(k-1)/2$ Klassifikatoren erstellt. Um zu entscheiden, zu welcher Klasse ein Input gehört, wird dann auf eine Max-Win-Voting-Strategie zurückgegriffen. Da auf diesen Klassifikator nur für Vergleichszwecke zurückgegriffen wird, wird für eine ausführlichere Beschreibung des Verfahrens auf die Quelle [22] verwiesen.
Im praktischen Teil wurde der Klassifikator mit der folgenden Konfiguration verwendet:

```
pipeline:
- name: "SklearnIntentClassifier"
#Einstellungen für den Grid Search für die SVM
#Schwere der Missklassifikation (klein => große Margin)
C: [1, 2, 5, 10, 20, 25, 50, 75, 100]
# Welche Kerne werden verwendet
kernels: ["linear"]
# Maximale Anzahl der k-fachen CV
"max_cross_validation_folds": 5
# Scoring Funktion für die Evaluierung
"scoring_function": "f1_weighted"

```

DIETClassifier: Der DIET-Klassifikator ist die aktuelle state-of-the-art Multi-Task-Architektur von Rasa, welche sowohl für die Klassifikation der Intentionen als auch für die Extraktion der Entitäten verwendet wird (vgl. [23]). Diese Architektur basiert auf einem Transformer. Transformer haben den Vorteil, dass sie durch einen Aufmerksamkeitsmechanismus (engl. attention mechanism) die Sequenzen der Tokens in einem Input berücksichtigen. Sie haben dabei gleichzeitig Vorteile gegenüber den bisher für diesen Task verwendeten rekurrenten neuronalen Netzen, da sie dadurch, dass sie parallelisierbar sind, eine schnellere Trainingszeit aufweisen und bei langen Sequenzen nicht unter verschwinden-

den Gradienten und dem Verlust von Information leiden.

Transformer arbeiten mit einem Self-Attention-Mechanismus, welcher das Ziel hat, die Tokens miteinander in Beziehung zu setzen, um somit kontextualisierte Word-Embeddings zu generieren (vgl. [3]).
Bei einem Self-Attention-Mechanismus werden die Vektoren der Input-Tokens v_i als Skalarprodukt mit dem Bezugsvektor v_j verrechnet. Diese Skalare s_{ji} werden daraufhin normalisiert und als Gewichte w_{ji} mit dem ursprünglichen Inputvektor des Tokens multipliziert. Zuletzt werden diese multiplizierten Vektoren addiert, sodass der entstehende Vektor y_j nun einer gewichteten Kombination aller Tokens des Inputs entspricht. Dieses Schema des Self-Attention-Mechanismus ist für das Beispiel eines Bezugsvektors v_3 in Abbildung 8 veranschaulicht.

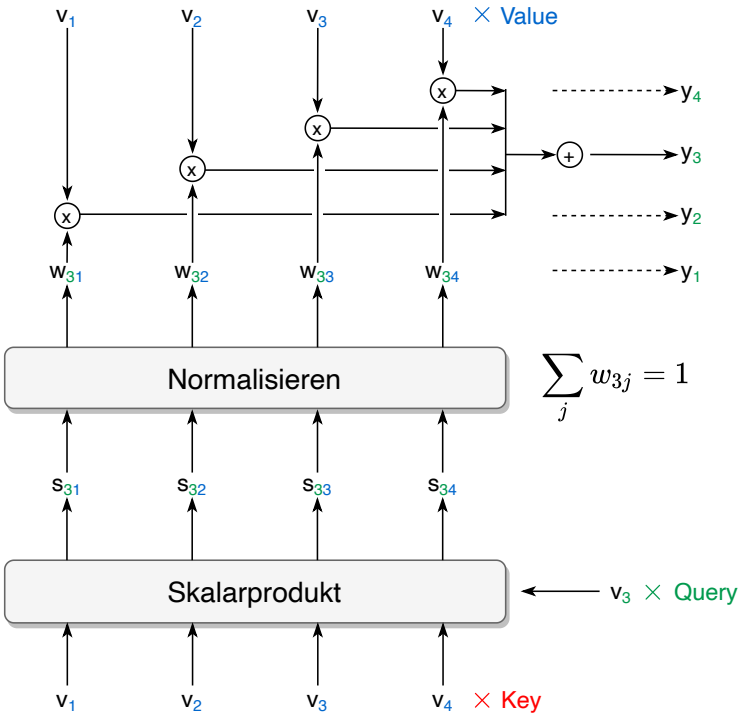

Abbildung 8: Schema des Self-Attention-Mechanismus

Um diesen Mechanismus nun trainierbar zu machen, werden die Vektoren v als Zeilenvektoren an den drei Stellen, an denen sie vorkommen, jeweils mit einer Matrix aus Gewichten multipliziert. Hierbei werden diese Matrizen in der Literatur mit Key, Query und Value benannt. Somit kann gesagt werden, dass der Output des Self-Attention-Mechanismus aus der gewichteten Summe der Values besteht, wobei die Gewichte jeder Value aus dem normalisierten Skalarprodukt der Querys mit dem entsprechenden Key gebildet wird (vgl. [3]).

Die entstehenden drei Matrixmultiplikationen können mit linearen Layern eines neuronalen Netzes ohne einen Bias Term verglichen werden. Wenn der Rest des Mechanismus nun ebenfalls in Matrixoperationen umgeschrieben wird, erhält man einen Block, welcher als Self-Attention-Block bezeichnet wird. Eines solcher Self-Attention-Block ist in der folgenden Abbildung 9 veranschaulicht.

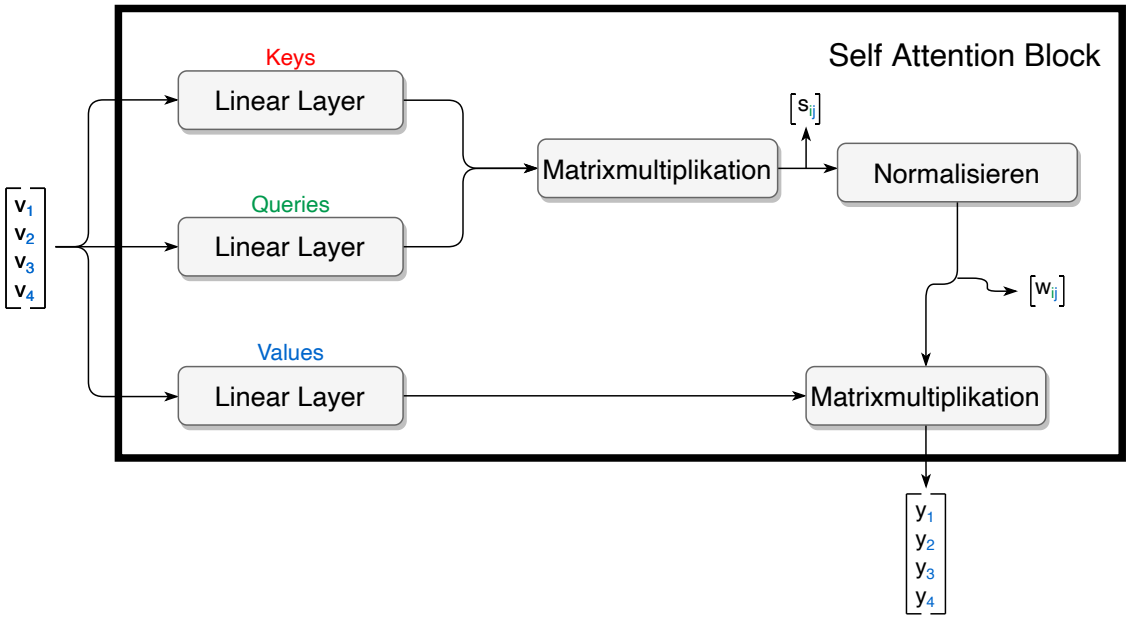

Abbildung 9: Schematische Veranschaulichung eines Self-Attention-Blocks

Die Erweiterung der Self-Attention, welche in einem Transformer genutzt wird, ist die Multi-Head-Attention. Der Grundgedanke hierbei ist, dass ein Token in einem Input von mehreren Tokens Aufmerksamkeit bekommen kann. Die Multi-Head-Attention vervielfältigt daher die linearen Layer, sodass die verschiedenen Layer ihre Aufmerksamkeit auf verschiedene Teile des Inputs richten können.
Ein Beispiel, warum mehrere Aufmerksamkeiten nötig sein können, ist in Abbildung 10 veranschaulicht. Hieran ist zu erkennen, dass das Wort „gab“ von mehreren Wörtern Aufmerksamkeit bekommen muss, um den Kontext des Satzes richtig zu erfassen.

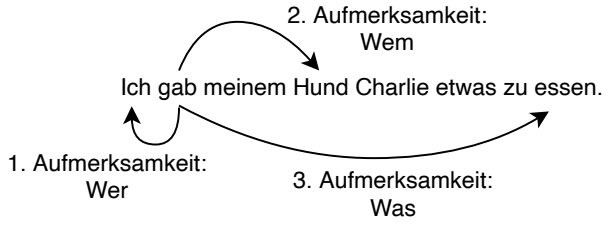

Abbildung 10: Veranschaulichung für mehrere Aufmerksamkeiten

Ein Multi-Head-Attention-Block ist in der folgenden Abbildung 11 veranschaulicht:

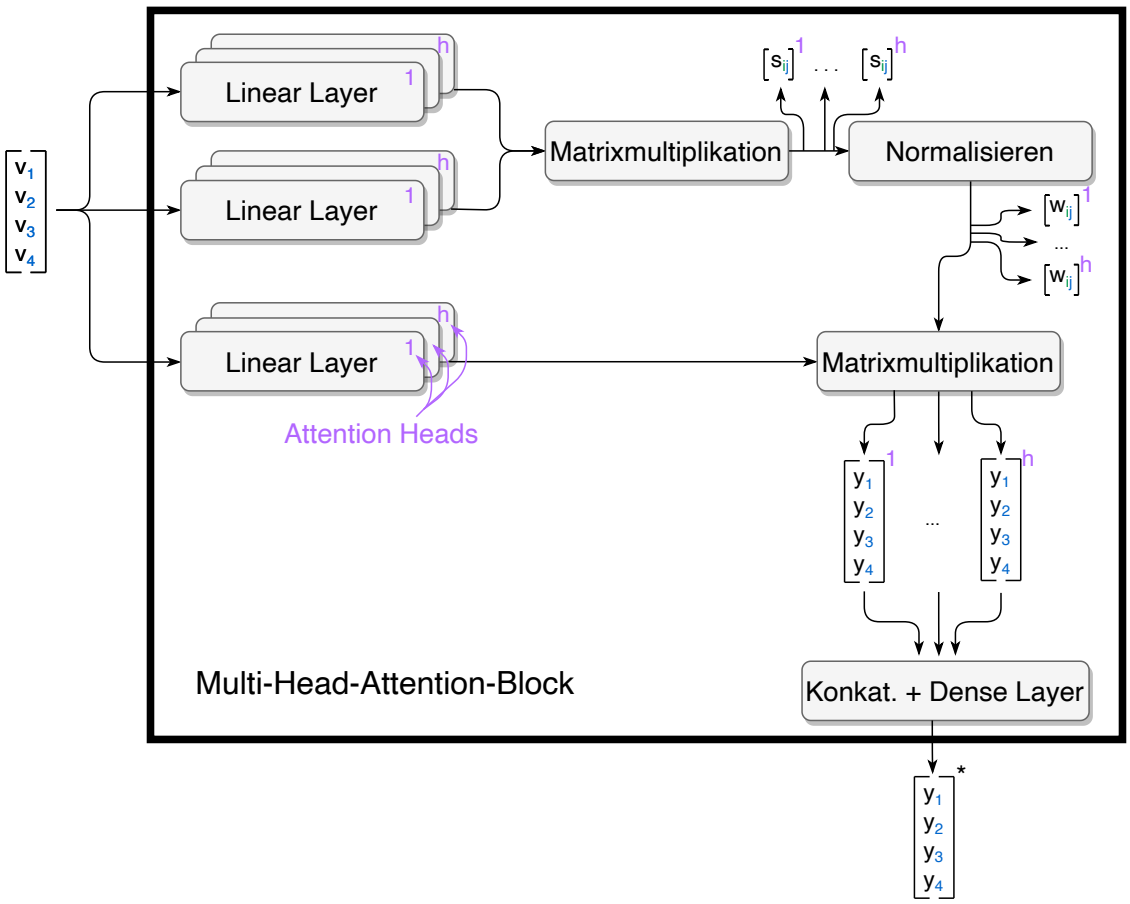

Abbildung 11: Veranschaulichung des Multi-Head-Attention-Blocks

Durch die mehreren Aufmerksamkeiten in diesem Block muss den Self-Attention-Mechanismen ein weiterer Schritt hinzugefügt werden, um die Outputs der einzelnen Aufmerksamkeiten zusammenzufassen. Hierzu werden diese konkateniert und durchlaufen ein Dense Layer, um eine Äquivalenz mit dem Input aufzuweisen. Hierdurch können diese Blöcke bei Bedarf gestapelt werden. Die Aufmerksamkeiten des Multi-Head-Attention-Blocks können dabei durch die Anzahl der Heads konfiguriert werden.
Ein solcher Multi-Head-Attention-Block wird in den Transformer Layern des DIET-Algorithmus verwendet.

Um nun den DIET-Algorithmus zu beschreiben, wird in der folgenden Abbildung 12 eine schematische Repräsentation der Architektur vorgestellt:

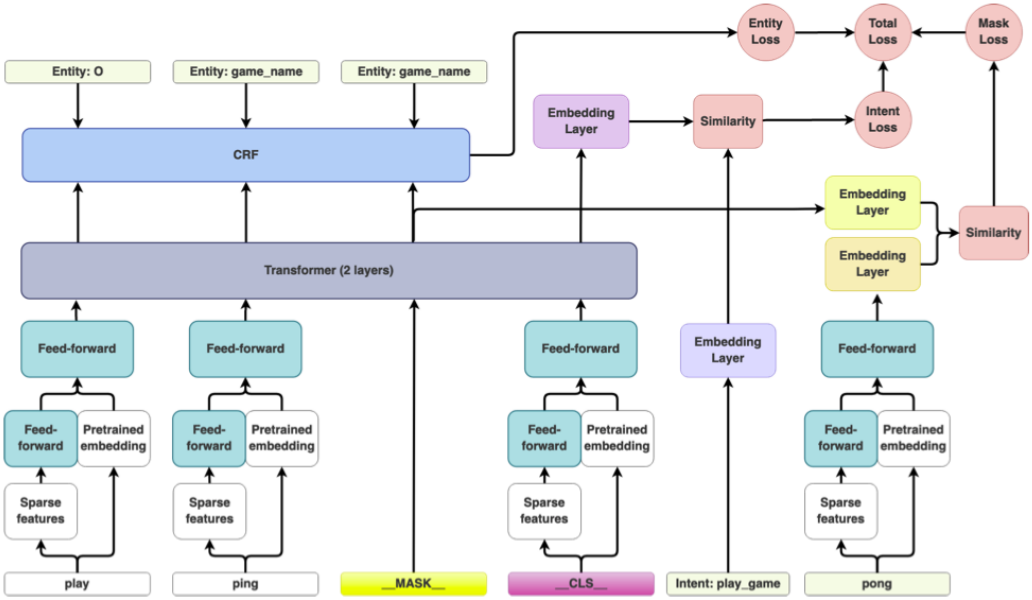

Abbildung 12: Veranschaulichung der DIET-Architektur anhand des Beispiels „play ping pong“ aus dem zugehörigen Research-Paper von Rasa [23]

Die DIET-Architektur kann in einer Conversational AI die beiden primären Tasks des NLU erfüllen: Die Klassifikation der Intentionen und das Erkennen der Entitäten. Im Folgenden wird der Algorithmus anhand des Weges des Inputs erklärt.
Die untere Hälfe des Schemas aus Abbildung 12 beschäftigt sich mit der Verarbeitung der Tokens. Hierbei ist erkennbar, dass die meisten Tokens zunächst einen gleichen Block durchlaufen. In diesem Token-Block werden die Tokens sowohl an ein vortrainiertes Modell, wie GloVe oder BERT, übergeben als auch als Sparse-Features an ein Feed-forward Layer (FF). Somit wird gewährleistet, dass die Architektur sowohl allgemeine Sprache als auch domain-spezifische Fachsprache erlernen kann. Die Sparse-Features können dabei One-hot-Encodings der Tokens sein oder Zeichen n-Gramme, je nachdem welcher Featurizer verwendet wird. Das darauf folgende Feed-forward Layer ist ein Fully-connected Layer, wobei ein Dropout von 80% eingestellt wird, um die Architektur leichtgewichtig zu halten. Die Gewichte dieser Feed-forward Layer werden dabei mit allen Feed-forward Layern an dieser Stelle geteilt. Da nun zwei Repräsentationen des Tokens existieren, werden diese konkateniert und durchlaufen ein weiteres Feed-forward Layer, um in eine einheitliche Form für den Transformer gebracht zu werden. Diese Layer besitzen ebenfalls 80% Dropout und teilen die Gewichte mit den anderen Layern an dieser Stelle. Der Output dieser Token-Blocks ist dann ein standardisierter Vektor gleicher Länge für jeden Token, welcher an den Transformer übergeben wird.
Neben den Input-Tokens werden die zwei speziellen Tokens $\_\_\_MASK\_\_\_$ und $\_\_\_CLS\_\_\_$

ergänzt. Der CLS-Token ist dabei ein Klassen-Token, welcher eine Aussage über den gesamten Input ermöglicht. Dieser Token durchläuft ebenfalls einen Token-Block, wobei hier die Vektoren aller Tokens durch den Mittelwert zusammengefasst werden. Die Besonderheit liegt darin, dass falls ein vortrainiertes Modell, wie BERT oder ConveRT, verwendet wird, an dieser Stelle ein modellabhängiger spezieller Vektor eingesetzt wird, welchen diese Modelle trainiert haben. Der Output dieses Blocks durchläuft dann ebenfalls den Transformer, wobei dieser Token als Repräsentant für den gesamten Input gesehen werden kann. Dieser Token wird dann als solcher Repräsentant mit der Intention des Inputs verglichen, woraus wird ein Verlust für die Klassifizierung der Intentionen berechnet wird.
Der andere spezielle Token ist der MASK-Token. Diese standardmäßig deaktivierte Einstellung maskiert während des Trainings ein zufälligen Token aus dem Input. Die Idee dahinter ist, dass der Algorithmus während des Trainings versuchen soll diesen Token vorherzusagen. Hierdurch wird erreicht, dass der Algorithmus ein weiteres Trainingsziel bekommt, welches die Vorhersage von diesem Wort im Input ist. Falls notwendig kann die Aktivierung dieser Einstellung den Algorithmus regularisieren, um allgemeinere Features aus dem Text zu lernen und sich nicht zu sehr auf eventuell diskriminierende Features für die Klassifikation zu konzentrieren. Diese maskierten Tokens werden, nachdem sie den Transformer durchlaufen haben, ähnlich wie beim CLS-Token, mit dem ursprünglichen Wort verglichen, welches einen Token-Block durchlaufen hat. Dieser Weg führt zu einem Verlust für diese Vorhersage, welcher beim Trainieren des Algorithmus optimiert wird.
Nachdem die Tokens den Transformer durchlaufen haben, werden diese an ein Conditional-Random-Field (CRF) weitergegeben, um die Entitäten des Inputs zu bestimmen (vgl. [24]). In diesem Block werden die vom Transformer erzeugten Word-Embeddings mit den Entitäten der Trainingsdaten verglichen, um somit eine Vorhersage für jeden Token zu treffen, ob dieser zu einer Entität gehört. Der CRF-Block besteht dabei aus einem Feed-forward Layer, welches für jeden Token eine Vorhersage trifft, zu welcher Entität dieser Token gehört. Diese Vorhersagen werden mit einer Überleitungsmatrix kombiniert, welche zusammen mit den Feed-forward Layern trainiert wird. Die Überleitungsmatrix versucht dabei zu erlernen, falls eine Entität in einem der Inputs erkannt wurde, ob der vorherige oder der darauf folgende Token ebenfalls zu einer Entität gehört. Diese Vorhersagen werden daraufhin mit den wirklichen Entitäten des Inputs verglichen und ein Verlust für diese Klassifikation berechnet. Eine Veranschaulichung des CRF-Blocks ist in Abbildung 13 gegeben.

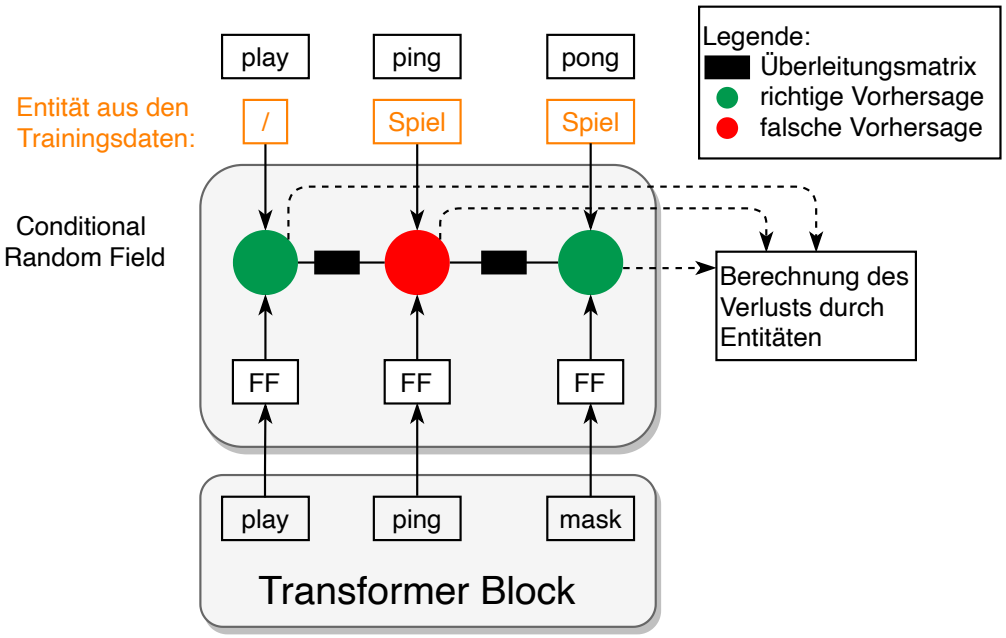

Abbildung 13: Schema des CRF-Blocks innerhalb der DIET-Architektur

Wie oben rechts in der Abbildung 12 erkennbar ist, versucht der DIET-Algorithmus die drei verschiedenen Verluste der einzelnen Tasks in Kombination miteinander zu optimieren. Der Algorithmus versucht daher den folgenden Ausdruck zu minimieren:

$$L_{total} = L_{Intent} + L_{Entity} + L_{Mask}$$

Dabei ist die Architektur des Algorithmus so aufgebaut, dass sowohl jeder dieser Verluste herausgenommen als auch jedes Layer und jeder Block individuell konfiguriert werden können. Somit ist sichergestellt, dass der Algorithmus sowohl in einer großen und umfangreichen als auch in einer leichtgewichtigen und einfachen Anwendung optimal eingesetzt werden kann.

4.3.2 Das Dialog-Modell

Das Dialog-Modell, wird verwendet, um das ML-gestützte Dialogmanagement umzusetzen. Hierbei werden die Aktionen des Chatbots von dem Dialog-Modell vorhergesagt. Die finale Entscheidung, welche Aktion der Chatbot ausführt, wird von den Richtlinien bestimmt. Hierbei können für einen Chatbot mehrere Richtlinien parallel genutzt werden. Diese Richtlinien sind teilweise regelbasierte Richtlinien, können aber auch auf einem ML-Modell basieren.
Die regelbasierten Richtlinien entscheiden anhand fester Regeln, welche Aktion der Chatbot einleiten soll. So entscheidet z. B. die Memoization-Policy, ob die bisherige Konversation in

den Storys vorkommt und wählt die in dieser Story vorkommende Aktion aus. Eine weitere regelbasierte Richtlinie ist die Rule-Policy. Diese bestimmt z. B. die in den Regeln definierte Zuweisung von Intentionen zu bestimmten Aktionen und die Formen des Chatbots.
Neben diesen können ML-basierte Richtlinien verwendet werden, um Konversationsverläufe zu behandeln, welche nicht direkt in den Trainingsdaten vorhanden sind. Rasa verwendet hierfür standardmäßig die Transformer Embedding Dialogue Policy (TED-Policy) [25]. Diese ebenfalls auf einem Transformer basierende Architektur wurde für die Vorhersage der Aktionen des Chatbots designt. In einer früheren Version von Rasa gab es auch eine LSTM-basierte Richtlinie, welche jedoch in neueren Versionen nicht weiter unterstützt wird, da sie in praxisnahen Anwendungen der TED-Policy unterlegen ist. Dies wird vor allem darauf begründet, dass die TED-Policy eine bessere Robustheit gegen unerwarteten Input aufweist, womit z. B. Small-Talk besser abgefangen werden kann. Außerdem ist die TED-Policy hinsichtlich der Schnelligkeit und Interpretierbarkeit den Vergleichsarchitekturen überlegen (vgl. [25]).
Als Input für die TED-Policy dienen die bisherigen Aktionen in der Konversation, die Intention und Entitäten des letzten Inputs und die belegten Slots, welche in einem vereinten Vektor einen unidirektionalen Transformer durchlaufen. Dies wird für jeden Schritt der Konversation ausgeführt und am Ende dient der Durchschnitt der Verluste dazu, den Transformer zu trainieren. Eine Veranschaulichung der TED-Architektur ist in der folgenden Abbildung 14 gegeben.

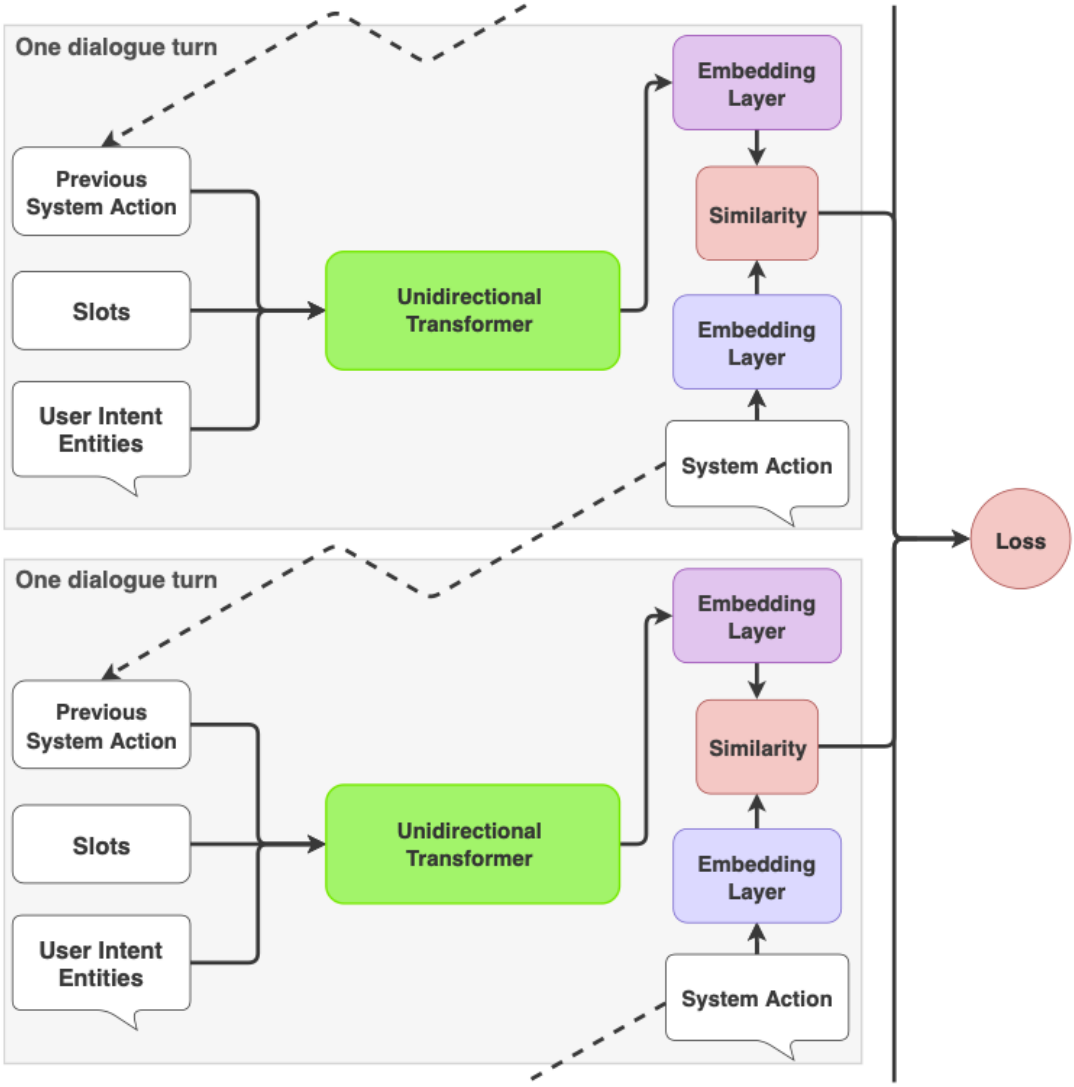

Abbildung 14: Veranschaulichung der Architektur der TED-Policy aus dem zugehörigen Research-Paper von Rasa [25]

In dieser Abbildung ist zu erkennen, dass die Features, welche dem Transformer als Input dienen, zunächst zusammengefasst werden, dann den unidirektionalen Transformer durchlaufen. Zuletzt werden diese über ein Embedding Layer und die Punktproduktähnlichkeit (engl. dot product similarity) mit den Aktionen des Chatbots verglichen, um einem Verlust zu berechnen mit welchem der Transformer und die Embedding Layer trainiert werden können. In dem unidirektionalen Transformer sind dabei die Verbindungen, welche zukünftige Inputs berücksichtigen auf Null gesetzt, sodass die Aufmerksamkeit nur auf vergangene Aktionen und Inputs gelenkt wird.
In Abbildung 15 ist ein Schema der TED-Policy abgebildet. Hieran ist erkennbar, dass die

zusammengefassten Vektoren f_t an den Transformer gegeben werden, um zunächst kontextualisierte Embeddings $\hat{f}_t$ für den aktuellen Dialogstatus zu generieren. Diese Embeddings durchlaufen dann, analog zu den Aktionen a_t, ein Embedding Layer, um dann über die Punktproduktähnlichkeit miteinander verglichen zu werden. Wenn die TED-Policy trainiert wurde, dient diese Ähnlichkeit, verglichen mit allen Aktionen, als Vorhersage des Modells.

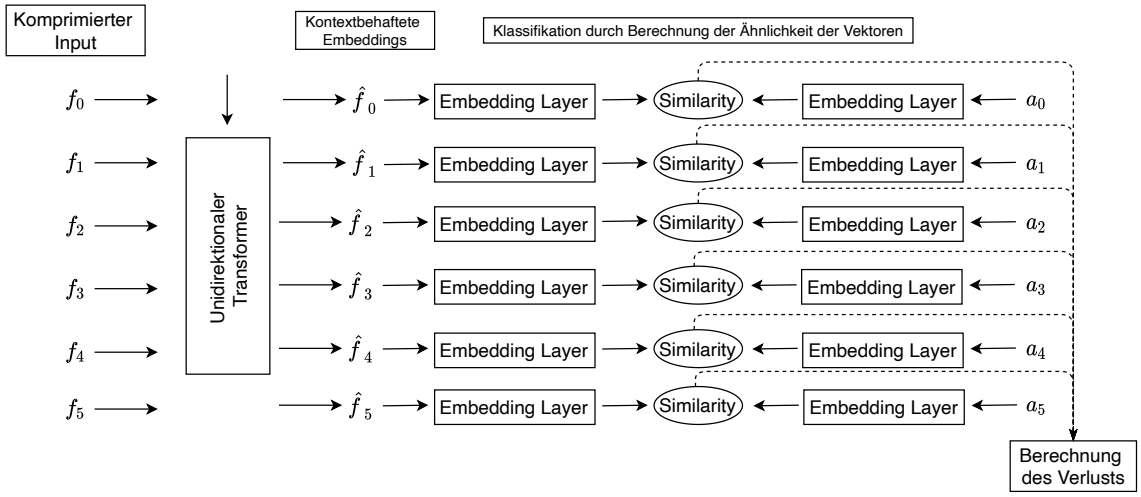

Abbildung 15: Schematische Abbildung der TED-Policy

Hierbei kann die Breite des Transformers über einen Parameter bestimmt werden, wodurch bestimmt wird, wie viele Konversationsschritte der Transformer berücksichtigen kann.

Da ein Chatbot mehrere Richtlinien nutzen kann, gibt es Prioritäten zwischen den einzelnen Richtlinien. Im Allgemeinen entscheidet diejenige Richtlinie, welche die höchste Sicherheit für die nächste Aktion aufweist. Falls jedoch zwei Richtlinien eine gleich hohe Sicherheit aufweisen, wird nach der folgenden Priorität entschieden:

1. Rule-Policy
2. Memoization-Policy
3. Mapping-Policy
4. ML-Policies (z. B. TED-Policy)

5 Die praktische Umsetzung

In diesem Kapitel wird genauer auf den praktischen Anwendungsfall und auf dessen Implementierung eingegangen. Hierfür wird der Kontext des Chatbots beschrieben, auf die Vorbereitung und Zielsetzung des Chatbots eingegangen und abschließend die Implementierung des Chatbots beschrieben.

5.1 Kontext und Domainbeschreibung

Die praktische Anwendung befindet sich im Bereich des Kundensupports. Der Chatbot wurde für den Reisekosten-Gorilla entwickelt, um die Nutzerfreundlichkeit der Applikation zu verbessern. Der Reisekosten-Gorilla wird betrieben, um die Reisekostenabrechnung sowohl für den Reisenden als auch für die Buchhaltung zu erleichtern. Hierbei wird dem Reisenden auf der Website ein Formular bereitgestellt, welches ausgefüllt direkt an die Buchhaltung des Unternehmens geschickt werden kann.

In einem Brainstorming, wie man das Produkt weiter verbessern kann, wurde eine firmeninterne Umfrage über die Benutzerzufriedenheit gemacht. Diese ergab aus Sicht der Nutzer zwei zentrale Verbesserungsmöglichkeiten des Produkts: Zum einen müssen sog. Poweruser, also Nutzer, welche häufig Reisekosten abrechnen müssen, zu viel mit der Website interagieren. Zum anderen haben Gelegenheitsnutzer Verständnisprobleme mit den Begriffen und der Fachsprache des Systems.
Um diese Problemstellungen anzugehen, wurde in der Vorbereitung auf das Projekt entschieden, dass ein Chatbot entwickelt wird, welcher den Powerusern ermöglicht, schneller ihre Abrechnungen zu tätigen, aber auch Erklärungen und einen Leitfaden gibt, damit Gelegenheitsnutzern die Abrechnung erklärt und erleichtert wird.

5.2 Die Vorbereitungsphase

In der Vorbereitungsphase, auch Design-Phase genannt, wurde in einem Kick-off Meeting entschieden, dass ein Chatbot die genannten Funktionalitäten erfüllen kann. In diesem Meeting wurde eine Liste mit Anforderungen erstellt, welche die Anwendung erfüllen soll. Diese initiale Liste besteht aus den Punkten, dass als Grundkonstrukt ein Chatbot verwendet wird, welcher einen Leitfaden der Reisekostenabrechnung gibt, wobei Korrekturen bei falschen Angaben möglich sind, er Fachbegriffe erklären kann und außerdem die Möglichkeit bietet, schnell Abrechnungen zu tätigen:

- Chatbot zur Reisekostenabrechnung

- Korrekturen bei falschen Angaben
- Erklären von Fachbegriffen
- Möglichkeit einer schnellen Abrechnung

Auf dieses Meeting folgend wurde eine Recherchephase bezüglich der Entwicklung von Chatbots gemacht, woraus folgte, dass ein Framework für die Entwicklung benutzt wird, wobei dieses unterstützen muss, dass auf einer kleinen Datengrundlage gearbeitet wird. Nach einer ausgiebigen Recherche wurde sich für das Framework Rasa entschieden, da es durch viele Anpassungsoptionen eine hohe Konfigurierbarkeit bereitstellt, welche unter anderem das Arbeiten auf einer kleinen Datengrundlage durch die Bereitstellung von vor-trainierten Einstellungen erleichtert. Da Rasa leider nicht die notwendigen Voraussetzungen für das TDD mitbringt, wurde ein weiteres Framework hinzugezogen, welches das explizite Testen vor der Modellentwicklung ermöglicht. Hierzu wurde das Framework CheckList ausgewählt, da es ein spezialisiertes Framework ist, welches das Testen von NLP-Anwendungen erleichtert.

5.3 Die Implementierung der Anwendung

In diesem Abschnitt wird die Implementierung der praktischen Anwendung mit dem TDD beschrieben. Hierzu wird die Entwicklung von dem Erstellen einer Test-Liste für die erste Funktionalität über die Implementierung eines Tests bis zum Trainieren des Chatbots beschrieben.

In der Design-Phase des Projekts wurden aus den Anforderungen die folgenden Funktionalitäten für den Chatbot definiert:

- Chatbot mit grundlegenden Intentionen: Hallo, Tschüss, Ja, Nein, Danke
- Intentionen, um den Task zu erfüllen: „einreichen“ als Trigger für das Formular, „informieren“ als Antworten des Nutzers
- Das Formular: Form für einen Beleg
- Korrekturen ermöglichen
- FAQs beantworten

Hierbei wird die Anforderung, dass eine schnelle Abrechnung möglich ist, durch die Funktionalität des Formulars abgedeckt, da dieses nur nach bisher nicht belegten Slots fragt. Durch die Angabe mehrerer Entitäten, auch wenn nicht danach gefragt wurde, wird somit eine schnellere Abrechnung möglich. Dies wurde bei der Implementierung der Funktionalität für das Formular durch eine entsprechende Information an den Nutzer kenntlich gemacht.

Zu diesen Funktionalitäten kommen die zusätzlichen Rahmenbedingungen für die Testeigenschaften des ML-Modells:

- Korrektheit: Metriken Accuracy, Precision, F1_Score jeweils > 0.9
- Robustheit: Robustheit gegen Tippfehler durch Vertauschung von Zeichen.
- Privatsphäre/Fairness: FTU, also das Sicherstellen der Unabhängigkeit der Prognose bzgl. sensibler Attribute
- Effizienz: Einschränkungen an Trainingszeit $<$ 1h und Prognosezeit $<$ 5sec

Um mit der zyklischen Bearbeitung im Sinne des TDD zu beginnen, wurden aus diesen Voraussetzungen Tests extrahiert, welche die Funktionalitäten gewährleisten. Hierzu wurden die Auswahl der NLU-Modelle und die verwendeten Richtlinien aus der Recherche über Rasa ergänzt. Als Beispiel wird die erste Funktionalität vorgestellt, welche in die Form der Tabellen 5 und 6 gebracht wurde:

Die Vorbereitung für die Erstellung der Test-Liste:
Benötigte Funktionalität: Funktionalität: Chatbot, welcher die Grundintentionen in einer Konversation verstehen kann Herunterbrechen: Auswahl der NLU-Modelle: DIET, Sklearn, jeweils mit Spacy Verwendete Richtlinien: Memoization-Policy, Rule-Policy, TED-Policy Überlegungen zur Implementierung: Einführen der Intentionen für *Hallo*, *Tschüss*, *Ja*, *Nein* und *Danke*, sodass das Grundgerüst einer Konversation verstanden wird.

Tabelle 7: Die Vorbereitung für die Test-Liste der ersten Funktionalität

Aus dieser Tabelle wurde dann die Test-Liste für diese Funktionalität erstellt, welche aus den MFTs für die eingeführten Intentionen und die Verwendung dieser Intentionen in einem Grundgerüst einer Konversation besteht:

Das Erstellen der Test-Liste
Testliste: MFTs für die Intentionen: begrüßen verabschieden bestätigen verneinen bedanken End-to-End-Tests für Konversationen: „Hallo“ - „Hi, kann ich helfen?“ - „Ja“ - „Platzhalter wird ausgeführt, bis zum nächsten Mal“ - „Ciao“ - „Tschüss“ „Hallo“ - „Hi, kann ich helfen?“ - „Nein“ - „Okay, bis zum nächsten Mal“ - „Ciao“ - „Tschüss“

Tabelle 8: Die Erstellung der Test-Liste von der ersten Funktionalität

Diese Vorbereitung wurde analog für alle Funktionalitäten und den zusätzlichen Testeigenschaften gemacht, sodass jeweils eine Test-Liste entstand, welche mit dem Zyklus des TDD bearbeitet werden konnte. Hierbei kann jede Funktionalität getrennt voneinander betrachtet werden und bei Bedarf schrittweise in leichtere Funktionalitäten umgeschrieben werden. Um den Umfang zu beschränken und Repetitionen zu vermeiden, wird dieser Schritt beispielhaft für diese erste Funktionalität dargestellt. Die weiteren Tabellen der Funktionalitäten sind im Appendix E veranschaulicht.
Wenn die Test-Liste angefertigt wurde, kann mit dem Zyklus des TDD begonnen werden:

Zyklus des TDD
Red-Phase - get test to compile and probably fail: Auswahl eines Tests mit den Red-Bar-Patterns aus 2 Implementierung des Tests mit den Testing-Patterns aus 3
Green-Phase - get test to pass: Implementierung des Codes durch die Obvious Implementation oder „Fake-it“
Refactor-Phase: Parametertuning und Fehleranalyse für das Modell, falls notwendig auch für andere Modelle aus der Auswahl.

Tabelle 9: Das Vorgehen im TDD mit ML

Jeder Zyklus beginnt mit dem Anlegen eines Tests. Hierbei wird ein Test ausgewählt, welcher den Red-Bar-Patterns entspricht und dann mithilfe der Testing-Patterns umgesetzt wird. Als Beispiel kann der MFT für die Begrüßungen gewählt werden, da er dem One-Step-Test entspricht und die Implementierung der Anwendung einen Schritt voranbringt. Die Implementierung dieser Tests wurde bereits im Abschnitt 4.1 genauer beschrieben, daher wird der Test der Begrüßungen im folgenden Listing 13 zusammengefasst dargestellt:

```
from checklist.editor import Editor
from checklist.test_types import MFT

begruessungen = ['guten Tag','guten Morgen','guten Abend','grüß Gott',
'ich grüße Sie','seien Sie gegrüßt', 'grüß dich','hallihallo','hallo',
'hallöchen','hallöle','hei','hey','heyho','hi','hoi','howdie','huhu',
'moin','moin moin','na du','servus','Tach','Tachchen','Tag',
'salut','sei gegrüßt','Ei Gude''Grüße','Gude','Gude wie',
moin','moin moin','Tach auch','alaaf','helau']

ret = editor.template('{gude}',gude = begruessungen, labels=2, save=True)

MFT_Hallo = MFT(ret.data, labels=ret.labels, name='Begrüßungen',
    capability='Intent Recognition', description='Häufige Begrüßungen')

```

Listing 13: Der MFT für die Intention „begrüßen“

Wenn der Test erstellt wurde, konnte dieser dann ausgeführt und ausgegeben werden, um das erwartete Ergebnis eines Fehlschlags zu erhalten:

```
MFT_Hallo.run(wrapped_prediction)

MFT_Hallo.summary() //zum Anzeigen der Ergebnisse

```

Listing 14: Grundaufbau für das Ausführen eines Tests

Da zu Beginn der Implementierung noch kein ML-Modell existiert, musste für die Ausführung der Tests im ersten Zyklus ein Modell gewählt werden, welches der *run*-Funktion übergeben wird. Hierfür wurde auf die Initialisierungsfunktion von Rasa zurückgegriffen, welche alle benötigten Dateien initialisiert. Diese Standardeinstellung von Rasa wurde im Anschluss an das Erstellen dieses ersten Tests angepasst.
An dieser Stelle wurde der Test zur Test-Suite hinzugefügt. Dies ermöglicht es die gesamte Testfunktionalität auszuführen, um somit alle Auswirkungen einer Änderung auf das Modell zu erkennen. Nach dem erreichen des ersten Meilensteins, welcher nach dem Implementieren des Formulars gesetzt wurde, sah die Test-Suite nach dem ersten ausführlichen Parametertuning wie folgt aus:

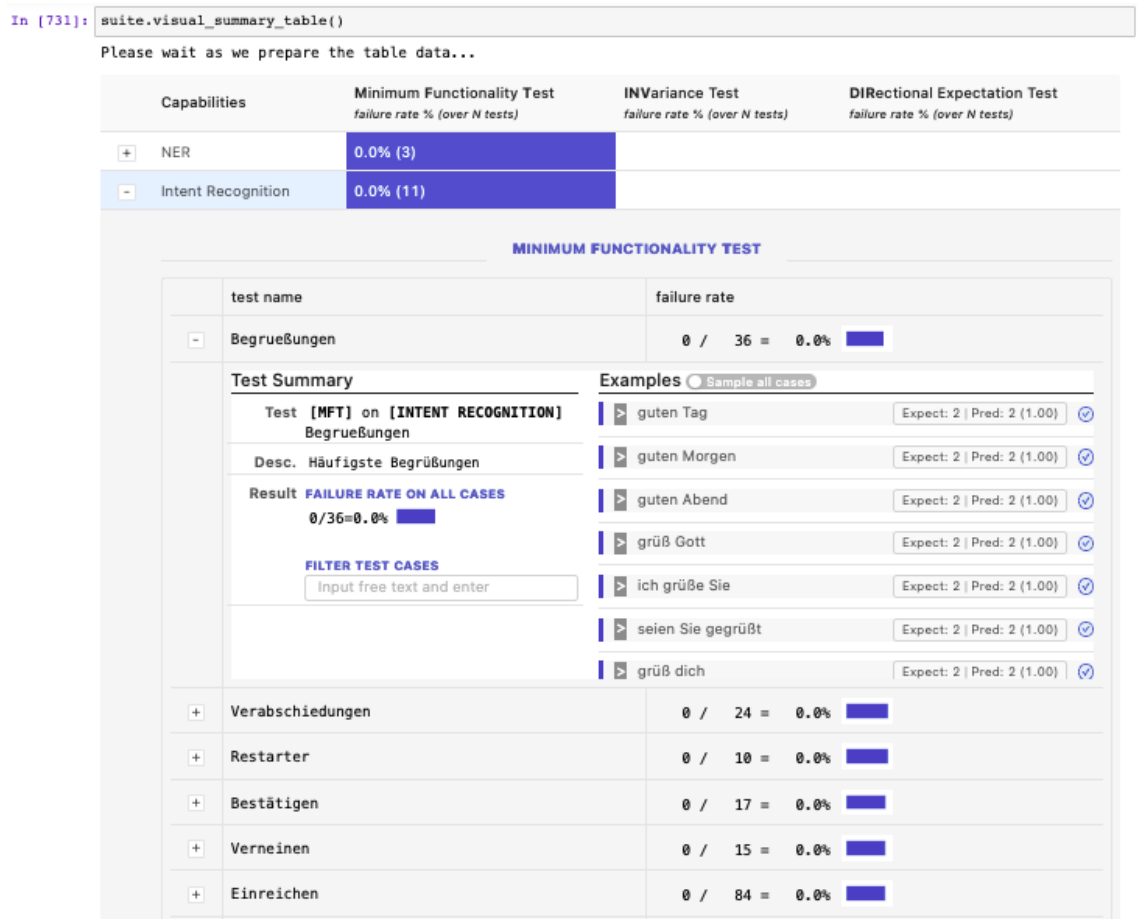

Abbildung 16: Visualisierung der Test-Suite des DIET-Klassifikators

Durch diese Visualisierung der Tests war es sehr schnell möglich, einen Überblick darüber zu bekommen, welche Funktionalitäten das Modell in seiner aktuellen Form beherrscht.
Da in dem ersten Zyklus bisher noch die Standardeinstellung von Rasa verwendet wurde, wird diese für den Vergleich mit dem Sklearn-Klassifikator angepasst. Das initiale Modell des TDD bestand daher aus der folgenden Konfiguration:

```yaml
pipeline:
- name: SpacyNLP
  model: "de_core_news_md"
- name: SpacyTokenizer
- name: SpacyFeaturizer
- name: RegexFeaturizer
- name: LexicalSyntacticFeaturizer
- name: CountVectorsFeaturizer
- name: CountVectorsFeaturizer
  analyzer: char_wb
  min_ngram: 1
    max_ngram: 4
- name: DucklingEntityExtractor
  url: http://localhost:8000
  dimensions:
  - time
  - amount-of-money
  locale: de_DE
  timezone: Europe/Berlin
- name: DIETClassifier
  epochs: 100
- name: EntitySynonymMapper
- name: ResponseSelector
  epochs: 100
  retrieval_intent: faq
- name: FallbackClassifier
  threshold: 0.3
  ambiguity_threshold: 0.1

policies:
- name: MemoizationPolicy
  max_history: 3
- name: TEDPolicy
- name: RulePolicy

```

Listing 15: Konfiguration des initialen Modells mit dem DIET-Klassifikator und der TED-Policy

Diese Konfiguration bestand sowohl aus dem DIET-Klassifikator als auch aus der TED-Policy. Diese Wahl der Konfiguration ist dabei auf die Research-Paper der beiden Architekturen gestützt, da diese in deren Tests in der Performanz und weiteren Punkten wie Trainingszeit, Flexibilität und Interpretierbarkeit besser abschnitten als andere state-of-the-art Architekturen (vgl. [23, 25]). In dieser Konfiguration wurde ein erstes Modell trainiert, mit welchem die weiteren Tests direkt kompilieren konnten.
Da der geschriebene Test kompiliert, konnte mit der Green-Phase begonnen werden. In der

Green-Phase wurde das Modell soweit verfeinert, dass die Funktionalität erfüllt wird.
Da zu diesem Zeitpunkt keine bzw. nur grundlegende Intentionen entwickelt wurden, konnte für diese Intentionen das Prinzip der Obvious Implementation verwendet werden. Hierfür wurde die Intention und deren Trainingsdaten dem Modell hinzugefügt und die entsprechenden Dateien von Rasa angepasst.
Um die Rasa-Anwendung weiterzuentwickeln, müssen in den meisten Fällen die drei Dateien, NLU, Storys und Domain, angepasst werden.
Um, wie in diesem Beispiel, die Intention „begrüßen“ dem Chatbot hinzuzufügen, wurde in der NLU-Datei die Intention benannt und deren Trainingsdaten ergänzt. Zusätzlich musste in der Domain-Datei die Intention dann aufgeführt werden, sodass diese im Training berücksichtigt wird. Um eine Konversation der Anwendung hinzuzufügen, wurde die Story der Storys-Datei hinzugefügt. Falls die Story dabei Aktionen benötigte, welche noch nicht vorhanden waren, mussten diese in der Domain-Datei definiert werden. Mussten bei diesen Änderungen Entitäten oder Slots erkannt bzw. berücksichtigt werden, wurden diese in der Domain-Datei spezifiziert. Falls eine Intention hingegen immer eine gleiche Aktion des Chatbots aufrufen sollte, konnte das Hinzufügen der Story durch eine Regel in der Rules-Datei ersetzt werden, wobei darauf geachtet werden musste, dass diese Regel den implementierten Storys nicht widerspricht.
Wenn diese Anpassungen gemacht wurden, sollte der Test nun erfolgreich kompilieren und nicht mehr fehlschlagen. Falls dies für einen Tests nicht der Fall war, wurde zu dem „Fake-it“-Pattern gewechselt, wobei zunächst versucht wurde durch Anpassung der Parameter des Modells zu einem validen Modell zu kommen. Falls dies nicht möglich war, wurde im Detail geschaut, warum der Test fehlschlägt. Hierzu konnte im Falle der Konversationen im ersten Schritt geschaut werden, ob die Intention oder die Aktion nicht richtig erkannt wurde. Somit lag die Fehlerquelle im NLU-Modell und dessen Trainingsdaten oder in den Richtlinien und den Storys-Daten, welche daraufhin genauer untersucht werden mussten.

Wenn der Test erfolgreich implementiert und bestanden wurde, folgte die Phase des Refactorings. In dieser Phase wurde das verwendete Modell optimiert und falls notwendig ein Parametertuning für alle infrage kommenden Modelle ausgeführt. Hierdurch wurde zum einen evaluiert, welches der Modelle für die aktuelle Funktionalität am besten geeignet ist und zum anderen wurde eine vielversprechende Konfiguration für den nächsten Zyklus gefunden.
Hierbei war dies ein sehr umfangreicher Schritt, welcher nicht dem Grundgedanken des TDD entspricht, dass die einzelnen Zyklen aus vielen, aber schnellen Bearbeitungen bestehen. Deshalb und da ein vollständiges Parametertuning zu frühen Zeitpunkten und nach jedem Zyklus keinen ersichtlichen Vorteil bringt, wurde diese Phase nur in Verbindung mit Meilensteinen in vollem Umfang vollzogen. Dies ist an dieser Stelle sinnvoll, da mit diesen Meilensteinen jeweils eine hinreichend ausführliche Funktionalität abgeschlossen wurde, welche sich lohnte durch eine ausführliche Analyse der zur Verfügung stehenden Modelle zu optimieren.

Wie auch in dem vorgestellten Zyklus, wurde diese Phase in den anderen Zyklen verkürzt, indem nur das valide Modell aus der Green-Phase optimiert wurde, um somit eine gute Konfiguration für den nächsten Zyklus zu finden.

Nachdem die Anwendung die erste Anforderung, dass der Chatbot für die Reisekostenabrechnung genutzt werden kann, erfüllt und hierfür die ersten drei Funktionalitäten implementiert wurden, mussten nun die Testeigenschaften des ML berücksichtigt werden. Zu diesem Zweck wurde eine Test-Liste der Testeigenschaften erstellt, welche daraufhin mit dem TDD bearbeitet und implementiert wurde. Im Folgenden wird genauer auf die einzelnen Testeigenschaften des ML eingegangen und wie die Tests für deren Sicherstellung hinzugefügt wurden.

Für die Korrektheit wurde sichergestellt, dass die benötigten Voraussetzungen an die Metriken der Modelle eingehalten werden. Für den praktischen Anwendungsfall wurde vorausgesetzt, dass die Metriken Accuracy, Precision und F1_Score der beider Modelle über 90% liegen.

Für die Modell-Relevanz wurden keine weiteren Tests implementiert, da in der Vorbereitung die Auswahl der Modelle bereits eingeschränkt wurde. Somit wurde sichergestellt, dass das Modell die nötige Komplexität erfüllt. Bei gleichwertigen Modellen wurde jedoch der Indikator der Trainingszeit herangezogen, sodass ein kleineres und schneller trainierbares Modell präferiert wurde.

Für die Robustheit wurde beachtet, wie das Modell in der Anwendung verwendet wird. Da der Chatbot in einer geschlossenen Umgebung für den Nutzer verwendet wird, nimmt die absichtliche Fehlleitung der Prognose durch Adversarial Attacks eine untergeordnete Rolle ein. Für die Robustheit wurde daher eine sichere Vorhersage bei richtiger Benutzung in den Fokus gestellt. Das Modell soll eine gute Robustheit gegen Tippfehler aufweisen und trotz Tippfehlern eine gute Prognosesicherheit haben. Dies wurde im Sinne der feindlichen Frequenz durch einen Invarianztests umgesetzt. Hierfür wurden alle Daten, welche in einem CheckList-Test verwendet werden, gegen das beliebige Vertauschen von Zeichen getestet.

Für die Privatsphäre und Fairness wurde beachtet, woher die Datengrundlage stammt und ob schützenswerte Attribute in der Benutzung des Chatbots vorkommen. Da die Daten von Hand erstellt und durch interne Nutzung mithilfe von Rasa X ergänzt wurden, musste hierfür nur beachtet werden, welche Angaben der Nutzer über sich selbst macht. Da der Chatbot verwendet wird, um Reisekosten abzurechnen, wird eine Angabe über die Person, in Form einer E-Mail, benötigt, um die Abrechnung zuordnen zu können. Diese darf für die Fairness keine Auswirkungen auf die Prognose haben und für die Privatsphäre nicht weitergegeben werden. Die Fairness wurde in der praktischen Anwendung über das Prinzip der FTU realisiert. Hierbei wurde dies in der Konfiguration für den Slot berücksichtigt. Für die Privatsphäre wurden keine weiteren Tests implementiert, da der Chatbot nur mit dem zu Grunde liegende System kollaboriert, welches die Privatsphäre bereits gewährleistet.

Für die Effizienz wurde eine Voraussetzung an die Trainingszeit und Prognosezeit gesetzt. Daher darf die Trainingszeit nicht länger als eine Stunde betragen und die Prognose eine Zeitspanne von 5 Sekunden nicht überschreiten. Da die Anwendung Trainingszeiten von ca. 10 Minuten besitzt und die Prognosen eine sehr natürliche Reaktionszeit von ca. 3 Sekunden haben, wurde dies schon bei der Entwicklung des Chatbots eingehalten. Um das Effizienzkriterium zu sichern, wurden diese ebenfalls in je einem Test umgesetzt, welche in die Test-Suite aufgenommen wurden.

Für die Interpretierbarkeit wurden keine Tests umgesetzt, da in der Vorbereitung des Projekts hierfür keine weiteren Einschränkungen an die Auswahl der Modelle gesetzt werden wollte.

6 Ergebnis

Die Ergebnisse dieser Arbeit sind in zwei Teile unterteilt. Zum einen wird diskutiert, wie sich das TDD auf die Entwicklung ausgewirkt hat. Dabei wird berücksichtigt, wie das TDD aus der Sicht des Programmierers funktioniert hat, welche Vorteile es bei der Entwicklung hatte und wo die Unterschiede zum TDD der Softwareentwicklung ohne ML liegen. Zum anderen werden die Ergebnisse der Anwendung beschrieben. Hierbei wird auf verschiedene Vergleichsmetriken eingegangen, um die zwei untersuchten NLU-Modelle miteinander zu vergleichen und eine Vorstellung der Anwendung gegeben, um die Performanz der Anwendung zu veranschaulichen.

6.1 Auswirkungen des TDD auf die Entwicklung der Anwendung

Um die Auswirkungen des TDD auf die Entwicklung der Anwendung zu beschreiben, werden die Vorteile des TDD, wie sie in Kapitel 2.2 beschrieben sind, herangezogen und im Detail besprochen. Hierbei wird darauf eingegangen, welche Anpassungen und Verletzungen des TDD gemacht werden mussten, um die Entwicklung eines ML-Modells zu ermöglichen.

Die im Kapitel 2.2.2 beschriebenen Voraussetzungen für das TDD konnten bei der Implementierung zum Großteil eingehalten werden. Im Detail wurde eingehalten, dass dynamisch programmiert wurde, da die Tests und die Auswertung der Tests Feedback geben, ob und bei einer Analyse an welchen Stellen das Modell angepasst werden muss. Des Weiteren wurde eingehalten, dass die Tests selbst geschrieben wurden und durch das Ausführen des Jupyter Notebooks konnte direktes Feedback darüber geben, ob eine Änderung die gewünschten Auswirkungen hatte oder nicht. Hierbei ist zu berücksichtigen, dass die Tests ein Training des Modells erfordern und es somit zu einer Verzögerung kommt.
Lediglich die Voraussetzung, dass die Anwendung aus lose gekoppelten Komponenten besteht, konnte nicht eingehalten werden, da die Anwendung des Chatbots durch die Verkettung der Modelle und ML-Modelle im Allgemeinen klare Abhängigkeiten aufweisen. Dies machte sich bei den Tests bemerkbar, da zum einen ein Fehler sowohl bei der Erkennung der Intentionen auftreten konnte als auch bei der Klassifizierung, welche Aktion zurückgegeben wird. Zum anderen hatte das iterative Hinzufügen der Trainingsdaten, insbesondere bei den NLU-Daten, die Auswirkung, dass zuvor bestandene Tests in einer späteren Iteration wieder fehlschlagen konnten. Dies konnte jedoch durch eine genauere Differenzierung der Daten behoben werden.

Im Folgenden wird beschrieben, ob die im Kapitel 2.2.3 beschriebenen Vorteile des TDD auch für die Implementierung der Conversational AI zutrafen. Hierfür werden diese wieder in den folgenden Paragraphen beschrieben:

Sorgenfreies Testen: Die Test-Suite gibt durch die Sicherstellung einer minimalen Funktionalität eine gute untere Schranke für die Performanz des Modells. Der Programmierer muss hierbei jedoch beachten, dass dies nur eine untere Schranke der Performanz sicherstellt, da nicht alle möglichen Inputs getestet werden können. Eine bedachte Wahl der Tests ist daher ausschlaggebend, wie aussagekräftig die Test-Suite über die Performanz des Modells ist.

Sicherheit im Fall der Unsicherheit: Unter Beachtung der minimalen Funktionalität trifft dieser Punkt ebenfalls zu. Auch mit der Verwendung von ML gibt das TDD bei der Implementierung Sicherheit darüber, welche Funktionalitäten bereits umgesetzt wurden und überprüft diese im Fall der Unsicherheit.

Schwierigkeit variabel: Durch die Komplexität des ML kann die Wahl der Patterns nicht so weit vereinfacht werden, da hier nur zwischen zwei Patterns gewählt werden kann. Trotzdem ist die Schwierigkeit der Umsetzung durch die schrittweise Vereinfachung der Funktionalitäten variabel an das Können des Programmierers anpassbar.

Keine Suche nach dem Startpunkt: Dieser Vorteil bleibt auch bei der Entwicklung eines Modells bestehen, auch wenn mehr Vorwissen vom Programmierer gefordert wird. In der praktischen Anwendung hat der Chatbot z. B. mehrere Modelle, welche berücksichtigt werden müssen und in der richtigen Reihenfolge und im richtigen Setup anzuwenden sind. Hierfür gibt die Aufteilung der Funktionalitäten einen guten Aufschluss darüber, in welcher Reihenfolge die Anwendung zu implementieren ist, wenn das entsprechende Vorwissen angeeignet wurde.

Arbeiten in einem Team: Der Vorteil, dass durch die Fokussierung auf Tests die Kommunikation zwischen Programmierern und Nicht-Programmieren erleichtert wird, bleibt auch bei der Verwendung von ML bestehen. Da durch diese Fokussierung und Kommunikation über Tests sogar das Vorwissen über das ML außer Acht gelassen werden kann, ist es vorstellbar, dass auch die ML-Entwickler besser mit Nicht-ML-Verständigen kommunizieren können, was die Kommunikation in gemischten Projekten und mit den Kunden verbessern könnte.

Zusammenfassend kann gesagt werden, dass die Vorgehensweise des TDD auch für die Entwicklung einer Conversational AI geeignet und vorteilhaft ist. Dies wird im Wesentlichen darauf begründet, dass das TDD trotz des ML eine sehr gute gute gedankliche Struktur der Anwendung liefert und eine gute Kommunikation über die Funktionen der Anwendung fördert. Gegen die Verwendung spricht der Punkt, dass das TDD mit ML nicht so weit vereinfachen ist. Dabei muss jedoch berücksichtigt werden, dass ML auch ohne die Verwendung von TDD eine schwierige Aufgabe in der Softwareentwicklung darstellt, da ein mathematisch

komplexes Vorwissen notwendig ist und viele Eigenschaften des ML berücksichtigt werden müssen.

6.2 Die Performanz der Anwendung

In diesem Abschnitt wird kurz auf die Performanz der Anwendung eingegangen und die NLU-Modelle genauer untersucht. Da der Fokus der Arbeit auf der Anwendung des TDD liegt, wurde keine ausführliche Untersuchung der ML-Modelle gemacht, jedoch wird in diesem Abschnitt kurz vorgestellt, wie das entwickelte Modell in der Anwendung funktioniert. Um die Performanz des Modells zu beschreiben, wird im Folgenden auf bewährte Methoden des ML zurückgegriffen und ein Beispiel aus der Anwendung vorgestellt.

Um ein Klassifikationsmodell zu untersuchen, wird im ML oft auf verschiedene Metriken wie Accuracy, Precision, Recall und den F1_Score zurückgegriffen. Diese Metriken basieren dabei auf der Konfusionsmatrix. Die Konfusionsmatrix ist eine Matrix, welche die Vorhersage des Modells den wahren Labeln gegenüberstellt. In der folgenden Abbildung 17 wird die Konfusionsmatrix des DIETClassifiers und des SklearnIntentClassifier gegenübergestellt:

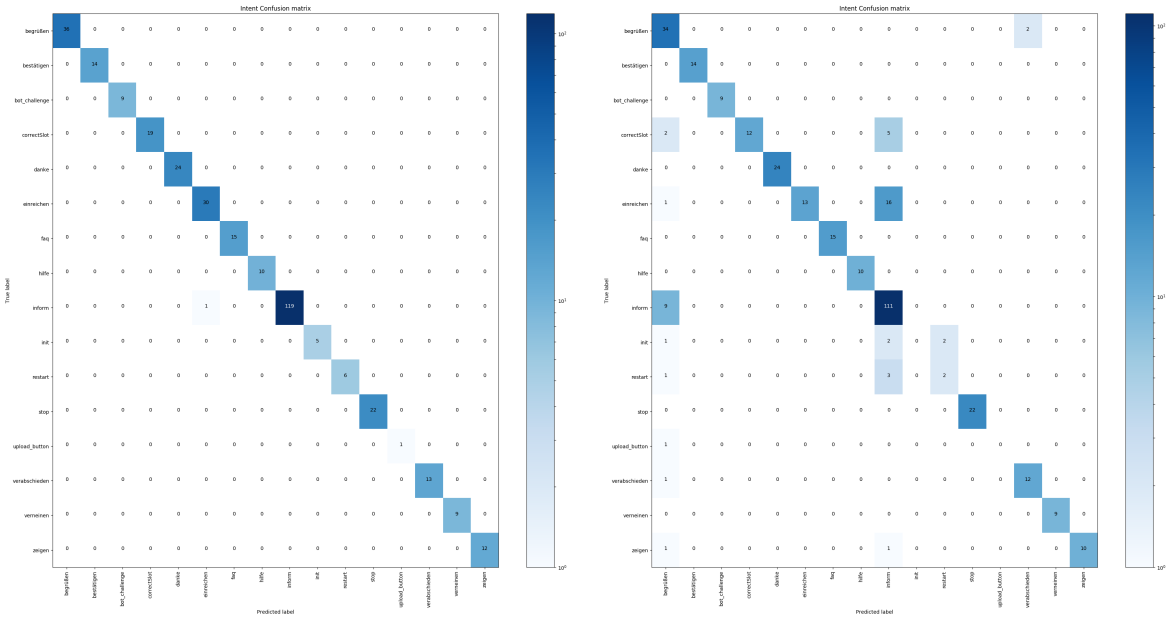

Abbildung 17: Gegenüberstellung der Konfusionsmatrizen des DIETClassifiers (links) und des SklearnIntentClassifiers (rechts)

Wie an dieser Gegenüberstellung erkennbar ist, sind beide Klassifikatoren in der Lage, die meisten Intentionen gut zu klassifizieren. Es ist jedoch auch erkennbar, dass der DIET-Klassifikator weniger Fehlklassifikationen vorweist als der Sklearn-Klassifikator. Für eine genauere Betrachtung der Konfusionsmatrizen sind diese im Appendix F vergrößert dargestellt.

Bei genauerer Betrachtung der Fehler der Modelle ist erkennbar, dass der Sklearn-Klassifikator, abgesehen davon, dass er mehr Fehler macht, auch deutlich gravierendere. Hierunter fallen z. B. sowohl die Fehlklassifikationen von „hey“ als Verabschiedung als auch die Fehlklassifikation von „ciao“ als Begrüßung. Diese und weitere Fehlklassifizierungen von Begrüßungen zusammen mit einigen Verwechslungen zwischen den Intentionen „einreichen“ und „informieren“ erschweren eine flüssige Konversation mit dem Chatbot (vgl. Abbildung 19).
Um neben der Klassifizierung auch die Sicherheit der Vorhersage zu berücksichtigen, kann ein Histogramm betrachtet werden, welches die richtigen und falschen Klassifizierungen mit dessen Sicherheit in der Vorhersage gegenüberstellt:

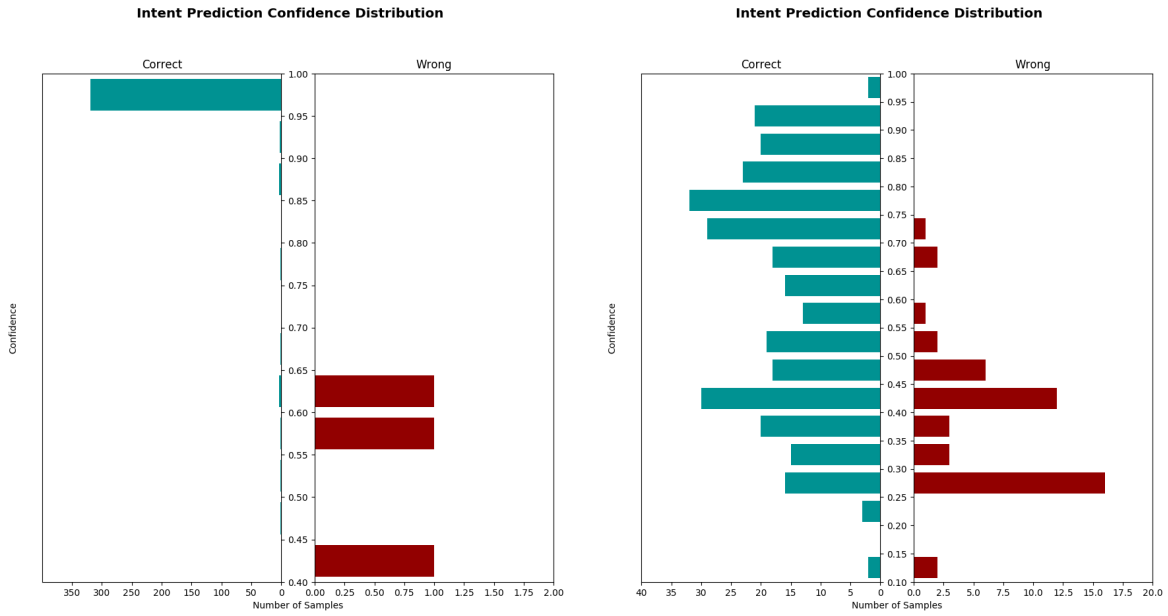

Abbildung 18: Gegenüberstellung der Histogramme des DIETClassifiers (links) und des SklearnIntentClassifiers (rechts)

Hieran ist erkennbar, dass der DIET-Klassifikator im linken Histogramm neben weniger Fehlklassifikationen auch eine höhere Sicherheit der Vorhersage liefert als der Sklearn-Klassifikator.

Um abschließend die Auswirkungen der schwächeren Klassifikation auf die Konversationen zu analysieren, wird im Folgenden eine zentrale Teststory betrachtet. Diese Story des Assistenten beschreibt den Happy-Path mit dem korrekten Ausfüllen der Form mit anschließender Korrektur eines Slots.
Um diese Story zu testen, können die in Rasa vorhandenen Funktionen verwendet werden. Diese testen eine gegebene Story und erstellen eine Datei mit den Ergebnissen, wobei angegeben wird, an welcher Stelle das Modell eine andere Aktion vorhersagt. Da diese Ergebnis-Datei den Verlauf der Konversation nicht gut veranschaulicht, wurde diese Konversation für die Veranschaulichung im Rasa X Client ausgeführt und in den folgenden Abbildungen veran-

schaulicht. Weil diese Konversation zu umfangreich für eine übersichtliche Veranschaulichung ist, wird zunächst der schematische Konversationsverlauf in Listing 16 vorgestellt und im Anschluss daran die Fehlschläge der Vorhersage in Abbildung 19 aufgezeigt. Der gesamte Konversationsverlauf wird in Appendix G veranschaulicht.

```
- story: Test happy path with correction
  steps:
  - intent: init
  user: |- init
  - action: utter_hallo
  - intent: begrüßen
  user: |- hey
  - action: utter_helfen
  - intent: einreichen
  user: |- ich muss etwas abrechnen
  - action: beleg_form
  ... Alle Slots werden belegt ...
  - action: utter_zeig_Daten
  - action: utter_fragen_korrekt
  - intent: verneinen
  user: |- nein
  - action: utter_slot_korrigieren
  - intent: correctSlot
  user: |- Projektzugehörigkeit
  - action: action_slot_reset
  - action: beleg_form
  - requested_slot: Projektzugehörigkeit
  - intent: inform
  user: |- [Lidl](Projektzugehörigkeit)
  - Projektzugehörigkeit: Lidl
  - action: utter_zeig_Daten
  - action: utter_fragen_korrekt
  - intent: bestätigen
  user: |- ja
  - action: utter_beleg_erstellt
  - intent: bestätigen
  user: |- ja
  - action: utter_fertig
  - intent: danke
  user: |- Danke
  - action: utter_danke
  - intent: verabschieden
  user: |- bye
  - action: utter_bye

```

Listing 16: Der schematische Konversationsverlauf des Tests

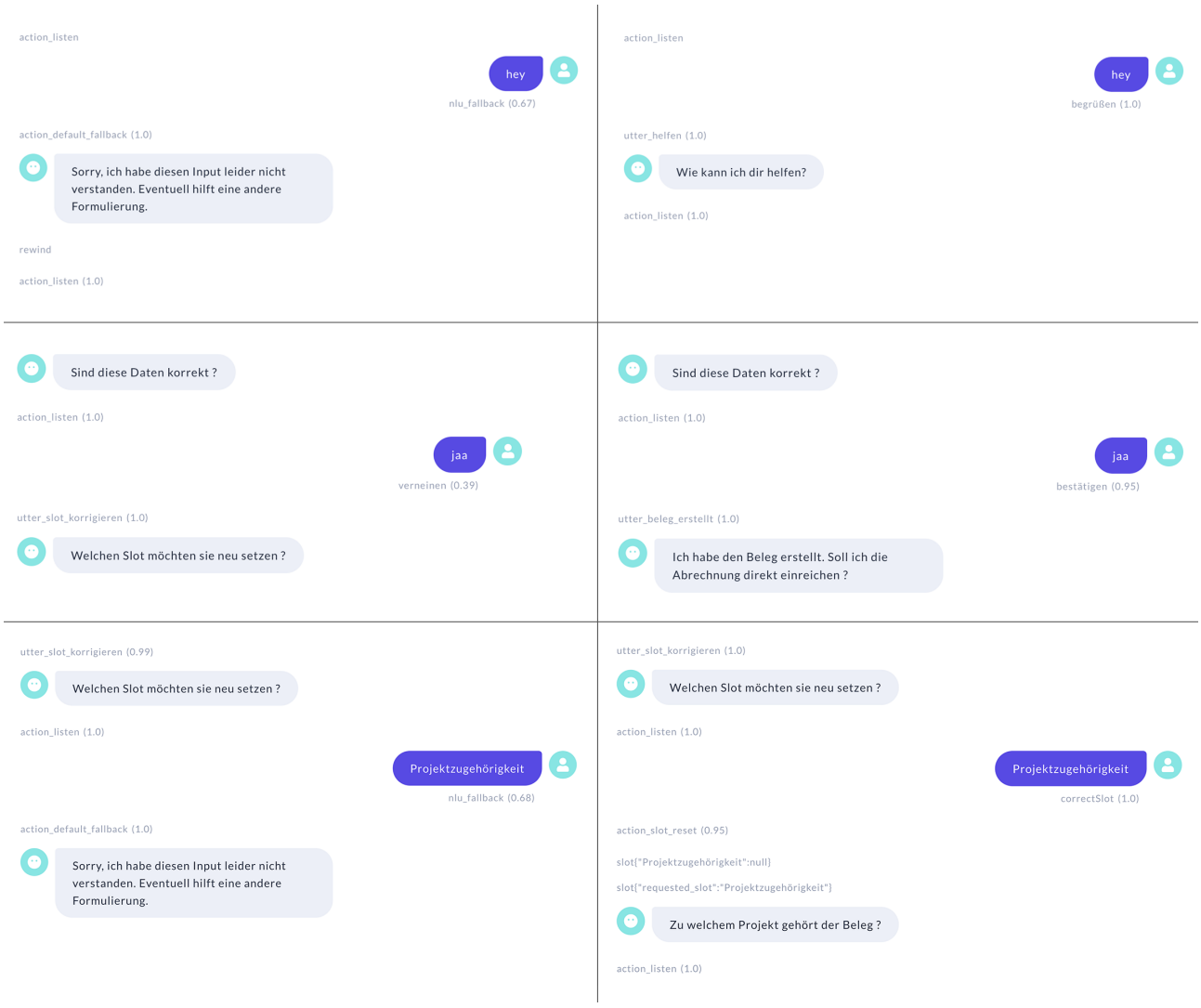

Abbildung 19: Gegenüberstellung der Fehlklassifikationen des SklearnIntentClassifier (links) zu den richtigen Klassifikationen durch den DIETClassifier (rechts)

Die drei aufgezeigten Fehlklassifikationen des Sklearn-Klassifikators im Bereich der Begrüßungen, Bestätigungen und dem Korrigieren von Slots zeigen, dass, obwohl der Klassifikator keine schlechte Klassifizierung in der Konfusionsmatrix aufzeigt, eine Konversation mit diesem Modell schwierig sein kann. Dieser Umstand muss für ein Modell in der Produktion unter allen Umständen vermieden werden, da der Nutzer hierdurch dem Chatbot gegenüber eine ablehnende Haltung einnimmt (vgl. [4]).
Um genau diese Problematik zu beheben, wurden die zusätzlichen Tests durch das CheckList-Framework angelegt, wodurch der Sklearn-Klassifikator nicht als valides Modell zugelassen werden würde. Zum einen erreicht es nicht die Mindestanforderung an die Korrektheit und

zum anderen werden deutlich schlechtere Werte bei der Auswertung der Tests erreicht, wodurch nahegelegt wird, dass dieses Modell nicht eigenständig als Modell in der Produktion geeignet ist.
Da der DIET-Klassifikator sowohl die Mindestanforderungen erfüllt als auch die Test-Suite einwandfrei besteht (vgl. Abbildung 16), wurde sich in diesem Anwendungsfall für den DIET-Klassifikator entschieden.

7 Ausblick und Fazit

7.1 Fazit

In dieser Arbeit wurde die agile Softwareentwicklung mit dem TDD auf eine Anwendung im Bereich des NLP angewendet. Hierfür wurden die Konzepte des TDD, das Testen von ML und die Conversational AI erläutert und daraufhin auf die Entwicklung einer Conversational AI für die Reisekostenabrechnung angewendet. Hierbei wurde der Fokus darauf gelegt, die Konzepte des TDD an die Entwicklung der ML-Modelle anzupassen und diese zu dokumentieren.
Um nun ein Fazit über diese Arbeit zu ziehen, wird noch einmal genauer auf die Entwicklung mit dem TDD geschaut. Das TDD bietet in der normalen Softwareentwicklung essenzielle Vorteile, welche die Implementierung, Kommunikation und Organisation betreffen. Während der Umsetzung dieser Arbeit wurden diese Vorteile ebenfalls als Vorteile bei der Implementierung eines ML-Modells festgestellt. Besonders hervorzuheben sind hierbei die Vorteile, dass das TDD sowohl eine gute gedankliche Struktur der Implementierung liefert als auch bei der Umsetzung der einzelnen Schritte unterstützt. Des Weiteren ist hervorzuheben, dass TDD bei der Entwicklung von ML besonders vorteilhaft für die langfristige Instandhaltung ist. Hierfür wird betont, dass die alleinige Entwicklung von ML-Modellen in der Praxis nicht ausreichend ist. Das TDD bietet hierbei den Vorteil, dass jegliche Voraussetzungen an die Anwendung bereits mit einem Test gesichert sind. Somit kann die, an die Entwicklung anschließende, dauerhafte Instandhaltung von ML-Modellen auf diese Tests zurückgreifen und durch wenige Ergänzungen eine dauerhafte Performanz des Modells in der Entwicklung sicherstellen.
Um auch die wesentlichen Unterschiede zu betrachten, muss berücksichtigt werden, dass einige Tests im Kontext des ML nur eine minimale Funktionalität gewährleisten. Dies ist ein Unterschied zum klassischen TDD, da hierbei die Tests mit deren Implementierung die gesamte Funktionalität sichern. Dies ist durch die statistische Natur und die Notwendigkeit der stetigen Anpassung des ML zu begründen, da dadurch nicht jeder möglicherweise auftretende Testfall getestet werden kann. Hierbei kann es sogar kritisch sein, zu viele Testfälle zu prüfen, da diese Testfälle zu einem Ausschluss aller Modelle zu einem späteren Zeitpunkt führen können. Daher muss darauf geachtet werden, dass die Tests möglichst genau die gewollte Funktionalität gewährleisten und somit das Grundgerüst und immer geltende Fälle für das Modell sicherstellen. Hierdurch wird auch eine Dokumentation der Tests wichtig, da erkennbar sein muss, wie ein Test eine Funktionalität sicherstellt.
Ein weiterer Unterschied zum klassischen TDD besteht darin, dass bei der Entwicklung im ML die Voraussetzung der lose gekoppelten Komponenten verletzt wird. In der praktischen Umsetzung dieser Arbeit betraf dies im Wesentlichen die Trainingsdaten des NLU-Modells. Hierbei musste beachtet werden, dass die verschiedenen Intentionen keine zu ähnlichen Trai-

ningsdaten aufwiesen, da dadurch die Vorhersage einer der Intentionen geschwächt werden und somit die erstellten Tests zu diesen Intentionen fehlschlagen konnten.

Alles in allem wurde in dieser Arbeit das TDD erfolgreich angewendet, um eine Conversational AI zu implementieren. Hierbei waren die Vorteile dieses Vorgehens sowohl zahlreicher als auch von größerem Nutzen als die beschriebenen Nachteile. Vor allem sind hierzu die Vorteile der Kommunikation und Strukturierung verantwortlich, da diese die Arbeitsweise sowohl in einem Team als auch des einzelnen Entwicklers positiv beeinflussen.

7.2 Ausblick

Abschließend kann für diese Arbeit gesagt werden, dass die Vorteile des TDD für die Entwicklung der Conversational AI sehr nützlich sind und über die Nachteile überwiegen. Hierbei wurde sich in dieser Arbeit auf die Entwicklung einer Conversational AI fokussiert, wodurch die Fragestellung nahegelegt wird, ob das TDD auch in anderen Anwendungsbereichen des ML, insbesondere mit tabellarischen Daten, vergleichbare Vorteile bietet. Hierbei ist dies vor allem interessant, da sich die Generierung der Tests durch die numerische Natur der Daten von den in dieser Arbeit erstellten Tests unterscheidet. Dies wird an dieser Stelle für zukünftige Forschungsmöglichkeiten nahegelegt.
Des Weiteren wäre für die Anwendung interessant, wie sich diese auf reale Nutzer auswirkt. Hierfür wurde die in dieser Arbeit implementierte Anwendung geschaffen, welche im Anschluss an diese Arbeit mit dem Reisekosten-Gorilla verbunden werden kann.

Literatur

[1] Z. Carmon, K. Wertenbroch, H. Yang, and R. Schrift, "Designing ai systems that customers won't hate," MIT Sloan Management Review, 12 2019.

[2] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," IEEE Transactions on Software Engineering, 2020.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in neural information processing systems, pp. 5998–6008, 2017.

[4] X. Luo, S. Tong, Z. Fang, and Z. Qu, "Frontiers: Machines vs. humans: The impact of artificial intelligence chatbot disclosure on customer purchases," Marketing Science, vol. 38, no. 6, pp. 937–947, 2019.

[5] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," Medical image analysis, vol. 42, pp. 60–88, 2017.

[6] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," Journal of Field Robotics, vol. 37, no. 3, pp. 362–386, 2020.

[7] W. Khan, A. Daud, J. A. Nasir, and T. Amjad, "A survey on the state-of-the-art machine learning models in the context of nlp," Kuwait journal of Science, vol. 43, no. 4, 2016.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," IEEE transactions on software engineering, vol. 41, no. 5, pp. 507–525, 2014.

[9] J. Zhang, E. Barr, B. Guedj, M. Harman, and J. Shawe-Taylor, "Perturbed model validation: A new framework to validate model relevance," 2019.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," arXiv preprint arXiv:1412.6572, 2014.

[11] L. Visengeriyeva, A. Kammer, I. Bär, A. Kniesz, and M. Plöd, "MLOps Principles." `https://ml-ops.org/content/mlops-principles`, October 2020. [Online; Stand: 6.11.2020].

[12] M. J. Kusner, J. R. Loftus, C. Russell, and R. Silva, "Counterfactual fairness," arXiv preprint arXiv:1703.06856, 2017.

[13] C. Molnar, Interpretable machine learning. Lulu. com, 2020.

[14] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in Proceedings of the 31st international conference on neural information processing systems, pp. 4768–4777, 2017.

[15] K. Beck, Test-driven development: by example. Addison-Wesley Professional, 2003.

[16] S. Hundertmark, "Wozu brauchen wir Conversational AI," 01 2021.

[17] F. Koetter, M. Blohm, J. Drawehn, M. Kochanowski, J. Goetzer, D. Graziotin, and S. Wagner, "Conversational agents for insurance companies: from theory to practice," in International Conference on Agents and Artificial Intelligence, pp. 338–362, Springer, 2019.

[18] A. Nichol, "The next generation of AI assistants in enterprise," 08 2018.

[19] P. Meyer, "Conversational AI for Builders: The 4 Levels of Complexity Scale," 01 2021.

[20] M. T. Ribeiro, T. Wu, C. Guestrin, and S. Singh, "Beyond accuracy: Behavioral testing of nlp models with checklist," arXiv preprint arXiv:2005.04118, 2020.

[21] "Top 20 Best Chatbot Builders For Big and Small Business - Chatbots.org."

[22] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," IEEE transactions on Neural Networks, vol. 13, no. 2, pp. 415–425, 2002.

[23] T. Bunk, D. Varshneya, V. Vlasov, and A. Nichol, "Diet: Lightweight language understanding for dialogue systems," arXiv preprint arXiv:2004.09936, 2020.

[24] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[25] V. Vlasov, J. E. Mosig, and A. Nichol, "Dialogue transformers," arXiv preprint arXiv:1910.00486, 2019.

Anhang

Appendix A

Die Verwendung der Obvious Implementation im klassischen TDD:
Angenommen, es würde der folgende Test für den Flächeninhalt eines Quadrats vorliegen:

```
1  public void testArea() {
2    Square s = new Square(2.0);
3    assertEquals(4.0, s.area());
4  }
5
```

Listing 17: Test für die Area-Funktion

So würde der Programmierer direkt die folgende Klasse umsetzen:

```
1  class Square {
2    private double edge;
3
4  Square(double edge) {
5    this.edge = edge;
6    }
7
8  public double area() {
9    return edge * edge;
10   }
11 }
12
```

Listing 18: Die Klasse „Square“

Appendix B

Die Verwendung des „Fake-it“-Patterns im klassischen TDD:
Wir wollen die Funktion *area*() aus dem Beispiel eines Quadrats testen. Hierzu wurde der folgende Test implementiert:

```
1  public void testArea() {
2    Square s = new Square(2.0);
3    assertEquals(4.0, s.area());
4  }
5
```

Listing 19: Test für die Area-Funktion

Wenn nun die Funktionalität gefälscht wird, wird die Konstante der Lösung zurückgegeben:

```
1  public double area() {
2    return 4.0;
3  }
4
```

Listing 20: Temporäre Funktion für die Area-Funktion

Nun ist die Funktionalität gefälscht und der Test wird bestanden. Dies bedeutet es müssen nun die Duplikate im Code ersetzt werden, um auf die richtige Implementierung zu kommen. Dies ist in diesem Beispiel die 4.0, welche zurückgegeben wird. Hierbei ist auffällig, dass diese 4 durch die Multiplikation von $2 * 2$ ersetzt werden kann:

```
1  public double area() {
2    return 2.0 * 2.0;
3  }
4
```

Listing 21: Temporäre erweiterte Funktion für die Addition

Dies ist nach wie vor noch nicht die gewünschte Funktionalität, erkennbar an den noch vorhandenen Duplikationen der 2. Wenn man diese nun durch das Attribut *edge* ersetzt, welches im Test der 2 entspricht, erhält man die gewünschte Funktion:

```
1  public double area() {
2    return edge * edge;
3  }
4
```

Listing 22: Finale Funktion für die Area-Funktion

Appendix C

Die Verwendung des Patterns der Triangulation im klassischen TDD:
Dieses Pattern startet ähnlich wie das „Fake-it“-Pattern, wobei die simpelste Lösung als Ausgangspunkt dient. Der einfachste Fall in dem Beispiel eines Quadrats ist ein theoretisches Quadrat mit einer Kantenlänge von 0.
Es wird also mit dem folgenden Test gestartet:

```
1  public void testArea() {
2    Square s = new Square(0.0);
3    assertEquals(0.0, s.area());
4  }
5
```

Listing 23: Test für die Area-Funktion

Hierfür wird die Lösung wieder hart codiert:

```
1  public double area() {
2    return 0.0;
3  }
4
```

Listing 24: Temporäre Funktion für die Area-Funktion

Nun wird jedoch durch einen weiteren Test ein weiterer Standpunkt erzeugt, welcher dann eine Abstraktion der Funktion erfordert:

```
1  public void testArea2() {
2    Square s = new Square(1.0);
3    assertEquals(1.0, s.area());
4  }
5
```

Listing 25: Zweiter Test für die Area-Funktion, um die Abstraktion zu forcieren

Diese Tests legen nun die Notwendigkeit nahe, die Funktion so zu abstrahieren, dass diese den Wert von *edge* zurückgibt, da in beiden Fällen der Input der Funktion der Lösung entspricht.

```
1  public double area() {
2    return edge;
3  }
4
```

Listing 26: Angepasste Funktion für die Area-Funktion

Durch einen weiteren Test, welcher wieder fehlschlägt, wird dann jedoch klar, dass die richtige Funktionalität durch $edge * edge$ erreicht wird:

```
1  public void testArea3() {
2    Square s = new Square(2.0);
3    assertEquals(4.0, s.area());
4  }
5
```

Listing 27: Weiterer Test für die Area-Funktion, um erneut eine Abstraktion zu forcieren

```
1  public double area() {
2    return edge * edge;
3  }
4
```

Listing 28: Finale Funktion für die Area-Funktion

Appendix D

Ergänzung des nächsten Tests zum Beispiel aus 2.2.1:
Ein passender Test für die *times*()-Funktion ist:

```
1  public void testTimesAdvanced() {
2    fiveDollar = new Dollar(5);
3    fiveDollar.times(2)
4    assertEquals(new Dollar(10), fiveDollar.amount)
5    fiveDollar.times(3)
6    assertEquals(new Dollar(15), fiveDollar.amount)
7  }
8
```

Listing 29: Fortgeschrittener Test der Multiplikation

Die Struktur dieses Tests passt nicht, da die Lösung zu diesem Problem die Veränderung der Rückgabe der *times*()-Funktion ist. Die Lösung ist, den Rückgabewert der *times*()-Funktion auf ein neues Objekt der Klasse zu setzen. Hierzu kann das Pattern der Obvious Implementation verwendet werden, um den Test umzuschreiben und zu implementieren:

```
1  public void testTimesAdvanced() {
2    fiveDollar = new Dollar(5);
3    Dollar product = fiveDollar.times(2);
4    assertEquals(new Dollar(10), product.amount)
5    product = fiveDollar.times(3);
6    assertEquals(new Dollar(15), product.amount)
7  }
8
```

Listing 30: Test für multiple Aufrufe der Multiplikation

Die entsprechende Implementierung im Code wäre:

```
1  class Dollar{
2    private int amount;
3
4    Dollar(int amount){
5      this.amount = amount
6    }
7
8    public Dollar times(int multiplier) {
9      return new Dollar(amount * multiplier);
10   }
11 }
12
```

Listing 31: Die finaleKlasse „Dollar“

Appendix E

Die ergänzenden Tabellen der Funktionalitäten:

Benötigte Funktionalität:
Funktionalität: Intentionen, um den Task zu erfüllen: „einreichen“ als Trigger für das Formular, „informieren“ als Antworten des Nutzers Herunterbrechen: Intention „einreichen“ hinzufügen, um das Formular auszulösen. Intention „informieren“ hinzufügen, welche die Antworten des Nutzers auf die gestellten Fragen des Bots klassifiziert.

Das Erstellen der Test-Liste
Testliste: MFTs für Intentionen: einreichen informieren inkl. Abtesten der Entitäten End-to-End-Tests für Konversationen: „Hallo“ - „Hi, kann ich helfen?“ - „Ja, ich will etwas abrechnen“ - „Platzhalter für das Formular wurde ausgeführt, bye“ - „Tschüss“

Benötigte Funktionalität:
Funktionalität: Das Formular: Form für einen Beleg Herunterbrechen: Hinzufügen der Form für die Slots: *Quittung, Anlass, Kategorie, Betrag, Mehrwertsteuer, Verrechenbarkeit, Projektzugehörigkeit, Zahlungsart, Email* Ergänzen der entsprechenden Antworten und Anpassen der Dateien Auslösen des Formulars durch die Intention „einreichen“ Hinzufügen der Möglichkeit einer schnelleren Abrechnung durch Angabe mehrerer Entitäten

Das Erstellen der Test-Liste
Testliste: Die Intention „einreichen“ löst das Formular aus Die Form füllt alle Slots vollständig aus (Erkennen der Entitäten ist bereits gesichert) End-to-End-Tests für Konversationen: „Hallo“ - „Hi, kann ich helfen?“ - „Ja, ich will etwas abrechnen“ - Das Formular wird ausgeführt - „Tschüss“

Benötigte Funktionalität: Funktionalität: Korrekturen ermöglichen Herunterbrechen: Intention „korrigieren“ hinzufügen „korrigieren“ löst eine Custom-Action aus, welche einen Slot zurücksetzt Nach dem Ausfüllen der Form kann bei der Frage, ob die Daten korrekt sind, durch Verneinung die Custom-Action ausgeführt werden.
Das Erstellen der Test-Liste
Testliste: Intention „korrigieren“ löscht einen Slot. Custom-Action zum Extrahieren des Slots aus der Message und Resetten des Slots End-to-End-Tests für Konversationen: „Hallo“ - „Hi, kann ich helfen?“ - „Ja, ich will etwas abrechnen“ - Das Formular wird ausgeführt - „Stimmen diese Daten?“ - „Nein“ - „Slot korrigieren“ - „Stimmen diese Daten?“ - „Ja“ - „Tschüss“

Benötigte Funktionalität: Funktionalität: FAQs zu Mappe, Beleg und Ausgabe beantworten Herunterbrechen: Retrieval-Intention FAQ für die Erklärung einer Mappe, eines Belegs und einer Ausgabe
Das Erstellen der Test-Liste
Testliste: MFTs für die Intentionen: faq/Mappe faq/Beleg faq/Ausgabe End-to-End-Tests für Konversationen: „Hallo“ - „Hi, kann ich helfen?“ - „Was ist eine Mappe?“ - Beantworten - „Danke“ - „Gerne“ - „Ciao“ - „Tschüss“ „Hallo“ - „Hi, kann ich helfen?“ - „Was ist eine Ausgabe?“ - Beantworten - „Danke“ - „Gerne“ - „Ciao“ - „Tschüss“ „Hallo“ - „Hi, kann ich helfen?“ - „Was ist ein Beleg?“ - Beantworten - „Danke“ - „Gerne“ - „Ciao“ - „Tschüss“

Appendix F

Vergrößerung der Konfusionsmatrizen des DIET-Klassifikators und des Sklearn-Klassifikators:

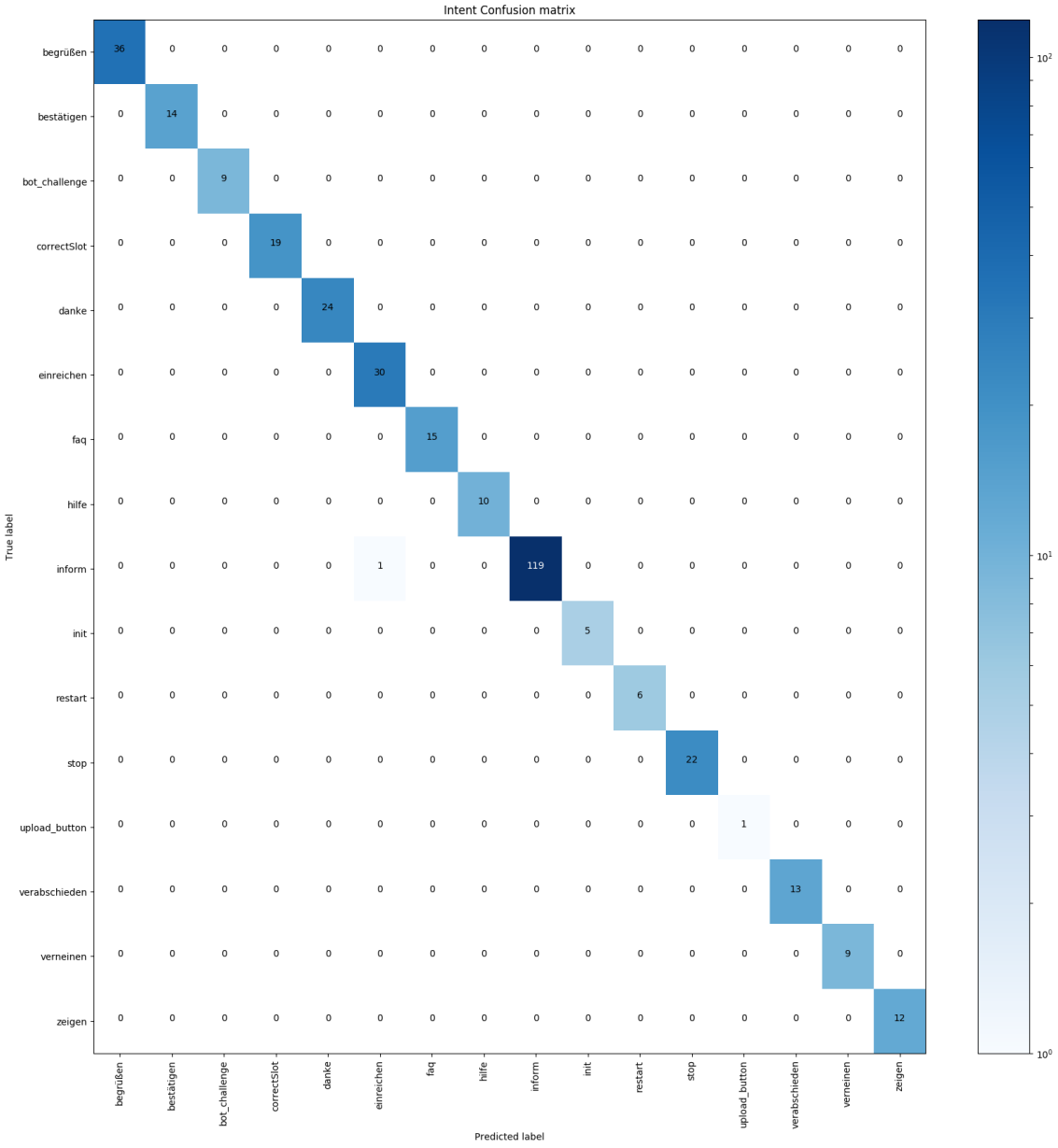

Abbildung 20: Konfusionsmatrix des DIETClassifiers

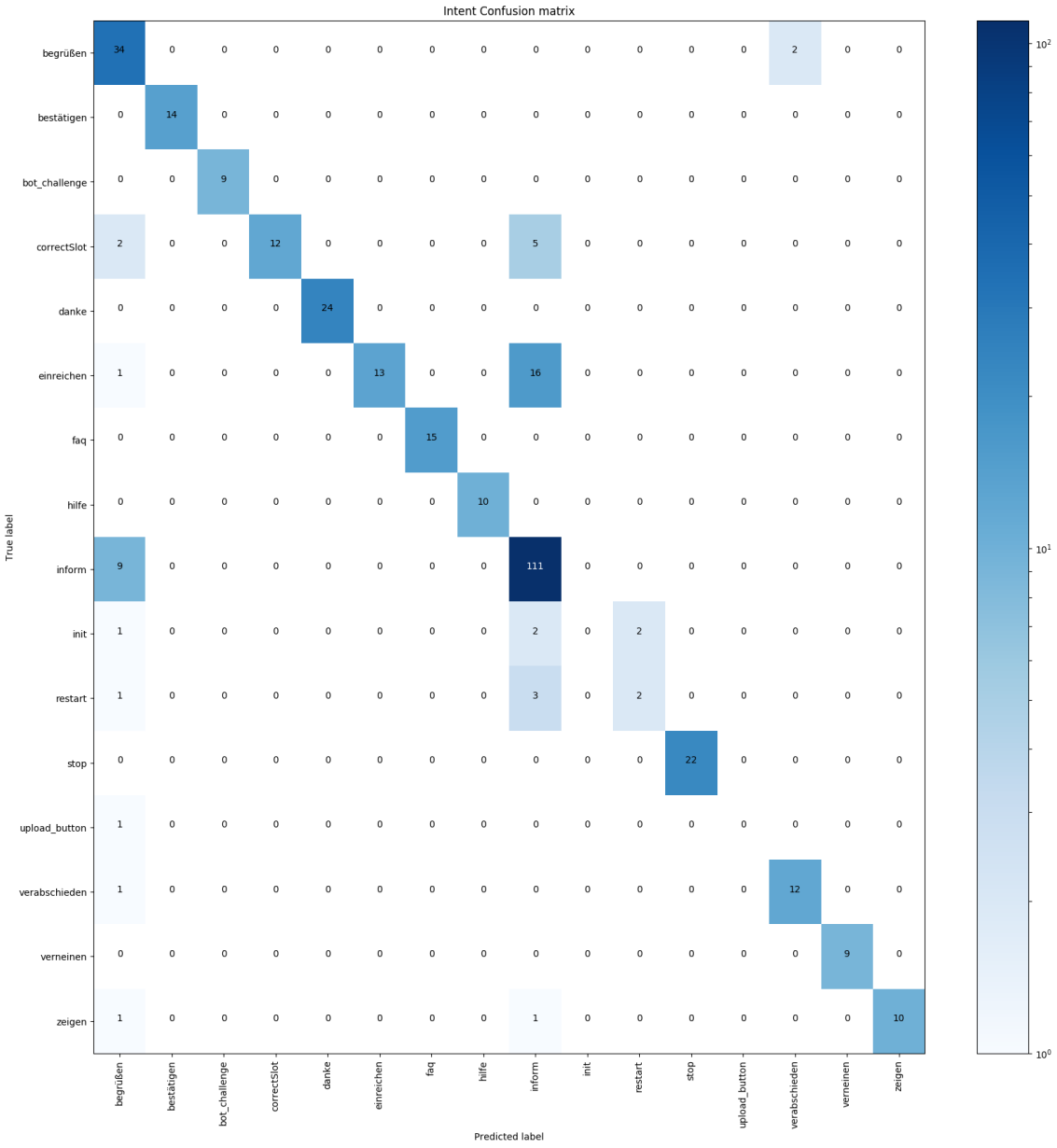

Abbildung 21: Konfusionsmatrix des SklearnIntentClassifiers

Appendix G

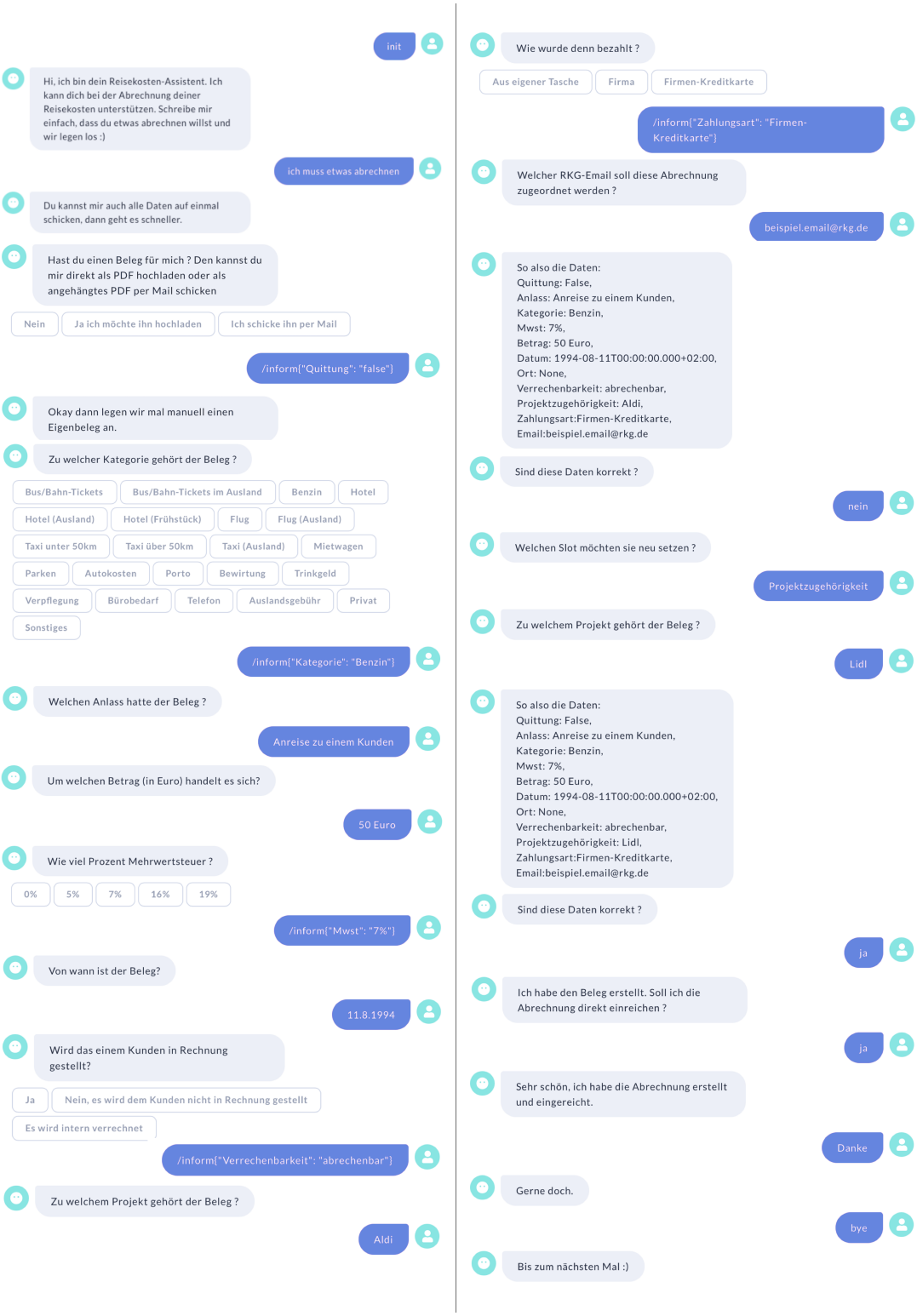

Abbildung 22: Die gesamte Konversation der DIET-Konfiguration durch Rasa X