Karlsruhe Institute of Technology

Synthese von Methodendefinitionen aus natürlichsprachlichen Äußerungen

Masterarbeit
von

Vanessa Steurer

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf Reussner
Betreuender Mitarbeiter:	Dipl.-Inform. Sebastian Weigelt
Zweiter betr. Mitarbeiter:	Dipl.-Inform. Hans-Peter Zorn

Bearbeitungszeit: 01.12.2018 – 18.06.2019

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 18.06.2019

(**Vanessa Steurer**)

Publikationsgenehmigung

Melder der Publikation
Hildegard Sauer

Institut für Programmstrukturen und Datenorganisation (IPD)
Lehrstuhl für Programmiersysteme
Leiter Prof. Dr. Walter F. Tichy

+49 721 608-43934
hildegard.sauer@kit.edu

Erklärung des Verfassers
Ich räume dem Karlsruher Institut für Technologie (KIT) dauerhaft ein einfaches Nutzungsrecht für die Bereitstellung einer elektronischen Fassung meiner Publikation auf dem zentralen Dokumentenserver des KIT ein.

Ich bin Inhaber aller Rechte an dem Werk; Ansprüche Dritter sind davon nicht berührt.

Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

Art der Abschlussarbeit:	Masterarbeit
Titel:	Synthese von Methodendefinitionen aus natürlichsprachlichen Äußerungen
Datum:	18.06.2019
Name:	Vanessa Steurer

Karlsruhe, 18.06.2019

(**Vanessa Steurer**)

Kurzfassung

Das Lehren neuen Wissens erfolgt beim Menschen schon von Kindesalter an durch Erklärungen in gesprochener Sprache. Bei computerbasierten Systemen hingegen wird diese Wissensvermittlung bisher durch die Definition von Konzepten und Prozeduren in einer spezifischen Programmiersprache realisiert. Für die Programmierung in natürlicher Sprache wird im Rahmen des PARSE-Projektes ein Ansatz zur Synthese von Methodendefinitionen aus natürlichsprachlichen Äußerungen vorgestellt. Dieser Ansatz ermöglicht es, natürlichsprachlich formulierte Funktionserweiterungen in ein Zielsystem zu integrieren. Um die dabei auftretende Vielfalt der Nutzerformulierungen behandeln zu können, werden verschiedene neuronale Netze trainiert. Diese Netze klassifizieren die Nutzereingabe hinsichtlich ihrer Intention und extrahieren verschiedene inhaltliche Bestandteile der beschriebenen, neuen Zielsystem-Funktionalität. Anschließend werden diese Bestandteile auf die Programmschnittstelle des Zielsystems abgebildet. Dazu werden verschiedene Sprachverarbeitungswerkzeuge sowie Kontext-Wissen und Synonyme eingesetzt. Für die Abbildung der Bestandteile auf bestehende Zielsystem-Funktionsaufrufe wird ein F_1-Wert von bis zu 90.9% erreicht. Die korrekte Abbildung vollständiger Funktionen inklusive aller benötigten Parameter erzielt ein F_1-Wert von bis zu 69.4%. Auch die Synthese eines aussagekräftigen Namens für diese neu definierte Funktionalität wird mit 90.4% Genauigkeit umgesetzt. Zusätzlich dazu ist das entwickelte Werkzeug in der Lage, neben den Definitionen neuen Wissens auch diktierte Anweisungsfolgen des Nutzers auf Skripte bestehend aus Zielsystem-Bibliotheksaufrufen abzubilden. Abschließend wurde festgestellt, dass in den meisten Fällen ein fehlendes Verständnis der Semantik einer Nutzereingabe zu Fehlern führt. Dieses Verständnis könnte zukünftig ausgebaut werden.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Grundlagen 3
2.1 Natürliche Sprache . . . 3
2.1.1 Syntax . . . 3
2.1.2 Semantik . . . 4
2.1.3 Pragmatik . . . 4
2.1.4 Satztypen . . . 4
2.1.5 Synonymie . . . 5
2.2 Verarbeitung natürlicher Sprache . . . 5
2.2.1 Werkzeuge . . . 6
2.2.2 Zeichenketten-Distanzmetriken . . . 9
2.3 Evaluationsmetriken . . . 11
2.4 Maschinelles Lernen . . . 12
2.4.1 Klassische Verfahren . . . 14
2.4.2 Verfahren auf Basis neuronaler Netze . . . 16
2.4.3 Wortrepräsentationen . . . 19
2.5 Microtasking-Plattform . . . 21
2.6 Programmsyntax . . . 21
2.7 Ontologien . . . 21

3 Projekt PARSE 23

4 Verwandte Arbeiten 25
4.1 Regelbasierte Ansätze . . . 25
4.2 Maschinelle Lernansätze . . . 29

5 Zieldefinition 35
5.1 Analyse der Problemstellung . . . 35
5.2 Zielstellung der Arbeit . . . 37

6 Vorstudie 39
6.1 Mögliche Datenquellen . . . 39
6.2 Konzeption einer Nutzerstudie . . . 40
6.3 Szenarien . . . 42
6.4 Datengrundlage . . . 45
6.5 Auswertung . . . 46
6.5.1 Nutzercharakteristiken . . . 46
6.5.2 Linguistische Analyse . . . 48

6.5.3 Zusammenfassung 51

7 Analyse **53**
7.1 Herausforderungen 53
7.2 Diskussion der verwandten Arbeiten 54
7.3 Konzeption von Teillösungen 55
7.3.1 Lehrsequenz-Erkennung 56
7.3.2 Aufbau von Methodendefinitionen 60
7.3.3 Ontologie-Abbildung 61

8 Entwurf **65**
8.1 Lehrsequenz-Erkennung 65
8.1.1 Detektion von Lehrsequenzen 66
8.1.2 Detektion von Deklaration und Beschreibung 72
8.1.3 Detektion von Aktionsbenennungen 75
8.2 Aufbau von Methodendefinitionen 78
8.3 Ontologie-Abbildung 79
8.3.1 Synthese einer Methodensignatur 81
8.3.2 Abbildung auf bestehende Individuen 83
8.3.3 Persistierung gefundener Abbildungen 92

9 Implementierung **95**
9.1 Lehrsequenz-Erkennung 95
9.1.1 Detektion von Lehrsequenzen 95
9.1.2 Detektion von Deklaration und Beschreibung 113
9.1.3 Detektion von Aktionsbenennungen 123
9.2 Aufbau von Methodendefinitionen 124
9.2.1 Klassifikatorintegration in Java 125
9.2.2 Zusammenführung der Klassifikationsergebnisse 127
9.2.3 Programmrepräsentation der Klassifikationsergebnisse 127
9.3 Ontologie-Abbildung 128
9.3.1 Synthese einer Methodensignatur 129
9.3.2 Abbildung auf bestehende Individuen 132
9.3.3 Persistierung der Ergebnisse 138
9.3.4 Evaluation der Abbildungssuche 139

10 Evaluation **155**
10.1 Datengrundlage 155
10.2 Ergebnisse 157
10.2.1 Synthese von Methodennamen 157
10.2.2 Abbildung auf bestehende Individuen 161
10.3 Bewertung der Ergebnisse 165

11 Erkenntnisse über eingesetzte Technologien **171**

12 Zusammenfassung und Ausblick **173**

Literaturverzeichnis **177**

Appendix **187**
A Nutzerstudie 187
A.1 Szenarien 187
A.2 Auszug aus den erhobenen Nutzereingaben 187
B Konfigurationsdatei der Implementierung 188

Abbildungsverzeichnis

2.1 Schematischer Vergleich von regelbasierten und maschinellen Ansätzen . . . 13
2.2 Entscheidungsbaum 14
2.3 Support Vector Machine 15
2.4 Naïve Bayes 16
2.8 Graphendarstellung einer exemplarischen Ontologie 22

3.1 Schematische Darstellung der Architektur von PARSE 24

5.1 Teilaufgaben der Arbeit 38

7.1 Hierarchie der Lehrsequenz-Bestandteile 58

8.1 Programmablauf des zu entwickelnden Agenten 66
8.2 Datenstruktur zur Programmrepräsentation der Lehrsequenz-Bestandteile . 79

9.2 Ausgabe des einfachen ANN-Netzes für Stufe 1 106
9.3 Kurvenverlauf der besten neuronalen Modellarchitektur `RNN4` für Stufe 1 . . 110
9.4 Ausgabe des einfachen `ANN1`-Netzes für Stufe 2 119
9.5 Ausgabe der besten neuronalen Modellarchitektur `RNN6` für Stufe 2 121
9.6 Ausgabe des besten RNN3-Netzes auf den Testdaten für Stufe 2 123
9.7 UML-Klassenhierarchie der ersten und zweiten Teilaufgabe der Arbeit . . . 125
9.8 Programmrepräsentation der Klassifikationsergebnisse 129
9.9 UML-Klassenhierarchie der dritten Teilaufgabe der Arbeit 130
9.10 Integration der Agentenausgaben in die PARSE-Graphdatenstruktur 140

Tabellenverzeichnis

2.1 Auszug des Penn-Treebank Wortarten-Tagset 6
2.2 Auszug des Penn-Treebank Phrasen-Tagset 7
2.3 Zugewiesene semantische Rollen 8

6.1 Anzahl erhobener Eingabesequenzen der Vorstudie 46
6.2 Anteil der Muttersprache Englisch unter den Einreichungen 46
6.3 Anteil der Programmierkenntnisse unter den Einreichungen 47
6.4 Dauer der Teilnahme (in Minuten) 47
6.5 Länge der Einreichungen (in Wörtern) 48
6.6 Auszug erfasster Aktionsbenennungen zu Szenario „Prepare coffee“ 50

7.1 Bestandteile einer Lehrsequenz 56

8.1 Erstelltes Trainingskorpus 67
8.2 Beispiele der Trainingskorpus-Vorverarbeitung 67
8.3 Verteilung der Stufe-1-Markierungen 69
8.4 Verteilung der Stufe-2-Markierungen 75
8.5 Programmrepräsentation der Lehrsequenz-Bestandteile 78
8.6 Struktur der Domänenontologie von PARSE 80
8.7 Ausprägungen des Ontologiekonzepts DataType 80

9.1 Auszug des Datensatzes aus der Vorstudie 96
9.2 Auszug des annotierten binären Klassifikationsdatensatzes 97
9.3 Struktur der binären Klassifikationsdatensatzes 97
9.4 Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 1 100
9.5 Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 1 100
9.6 Vergleich von N-Gramm-Konfigurationen des Entscheidungsbaums 102
9.7 Konfusionsmatrix des Entscheidungsbaums 102
9.8 Vergleich von N-Gramm-Konfigurationen des Entscheidungsbaums 103
9.9 Erzielte Genauigkeiten auf den Validierungsdaten der Stufe-1-Klassifikation mit klassischen Lernverfahren 103
9.10 Konfusionsmatrix des Entscheidungsbaums 104
9.11 Vergleich verschiedener Paketgrößen für Stufe 1 106
9.12 Unterschiedliche Netzschichten und ihre Parameter 108
9.13 Genauigkeiten auf den randomisierten Validierungsdaten verschiedener neuronaler Netze für Stufe 1 109
9.14 Genauigkeiten auf den randomisierten Validierungsdaten mit BERT für Stufe1 111
9.15 Genauigkeiten auf den szenarienbezogenen Validierungsdaten neuronaler Netze für Stufe 1 112
9.16 Test-Genauigkeiten der Stufe-1-Klassifikation 112
9.17 Auszug des annotierten Klassifikationsdatensatzes der Stufe 2 114
9.18 Struktur des Klassifikationsdatensatzes der Stufe 2 114

9.19 Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 2 115
9.20 Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 2 115
9.21 Vergleich der Genauigkeiten auf den Validierungsdaten verschiedener Paketgrößen für Stufe 2 119
9.22 Genauigkeiten auf den randomisierten Validierungsdaten verschiedener neuronaler Netze für Stufe 2 120
9.23 Genauigkeiten auf den szenarienbezogenen Validierungsdaten verschiedener neuronaler Netze für Stufe 2 122
9.24 Test-Genauigkeiten der Stufe-2-Klassifikation 123
9.25 Knotenattribute der erstellten Graphdatenstruktur 139
9.26 Umfang der Domänenontologie von PARSE 140
9.27 Datensatz für die Evaluation der Ontologie-Abbildung 143
9.28 Auszug synthetisierter Methodennamen der Ontologie-Evaluation 144
9.29 Synthetisierte Methodennamen der Ontologie-Evaluation 144
9.30 Wahrheitswerte gefundener Ontologie-Abbildungen 146
9.31 Ergebnisse der Abbildungssuche für Methoden-Individuen 147
9.32 Fehlerursachen für die Abbildungssuche von Methoden-Individuen 149
9.33 Durchschnittlicher Rang der Abbildungssuche für Methoden-Individuen . . 149
9.34 Ergebnisse der Abbildungssuche für vollständige Methodensignaturen . . . 150
9.35 Fehlerursachen für die Abbildungssuche von Methodensignaturen 153
9.36 Durchschnittlicher Rang der Abbildungssuche für vollständige Methodensignaturen 154

10.1 Erhobener Datensatz für die Ende-zu-Ende-Evaluation 157
10.2 Generierte Evaluationsgrundlage der Ende-zu-Ende-Evaluation 157
10.3 Ende-zu-Ende-Ergebnisse der Detektion von Lehrsequenzen 159
10.4 Auszug synthetisierter Methodennamen der Ende-zu-Ende-Evaluation . . . 159
10.5 Synthetisierte Methodennamen der Ende-zu-Ende-Evaluation 160
10.6 Ende-zu-Ende-Ergebnisse für Methoden-Individuen 162
10.7 Ende-zu-Ende-Ergebnisse für vollständige Methodensignaturen 163
10.8 Durchschnittlicher Rang der Abbildungssuche 164
10.9 Fehlerursachen für die Abbildungssuche von Methoden-Individuen 166

A.1 Szenarienindex der Vorstudie und Evaluationsstudie 187

1 Einleitung

1.1 Motivation

Intelligente Assistenten wie Amazon Alexa, Google Assistant oder Siri werden immer häufiger in den Alltag integriert. Sie beantworten Fragen oder erledigen vermehrt auch kleine Aufgaben, wie die Erstellung eines Kalendereintrags im persönlichen Terminplaner. Ihre zunehmende Beliebtheit verdanken sie vor Allem der Einfachheit ihrer Steuerung durch natürlichsprachliche Anweisungen. Eine bestehende Herausforderung der Schnittstellenentwicklung ist dabei die Komplexität der natürlichen Sprache.
Die Problemstellung, das Verständnis beliebiger Ausdrucksweisen und komplexer Anweisungen zu ermöglichen, wird als Programmieren in natürlicher Sprache bezeichnet. Damit jene freien Formulierungen auch in ausführbare Aktionen intelligenter Systeme umgesetzt werden können, ist die Abbildung menschlicher Sprachkonstrukte auf die Funktionen des Zielsystems erforderlich. Dabei soll es auch möglich sein, nicht nur bereits einprogrammierte Kommandos abzurufen, sondern auch neue Funktionalitäten hinzuzufügen. Diese Funktionserweiterung soll wiederum in natürlicher Sprache und überdies ohne Kenntnis von Programmierkonzepten ermöglicht werden. Ein weiterer wesentlicher Vorteil von lernfähigen Systemen liegt in der Möglichkeit, das Spektrum der vom System verstandenen Sprachausdrücke, zu ergänzen.
Da sich die Benutzer zunehmend an die Interaktion mit intelligenten Geräten mittels Sprache gewöhnen, wird ihr Interesse an der Programmierung solcher Geräte mittels natürlichsprachlicher Anweisungen mit großer Wahrscheinlichkeit zunehmen. Mögliche Anwendungsgebiete würde dieses Konzept dabei nicht nur in der im Projekt PARSE betrachteten Haushaltsroboter-Domäne finden. Auch beliebte Sprachassistenten wie Alexa müssen bisher für das Hinzufügen von nutzereigenen Fähigkeiten über Quelltext konfiguriert werden.

1.2 Zielsetzung

Das Ziel dieser Masterarbeit ist die Entwicklung eines Werkzeuges, welches in gesprochenen Eingaben Anweisungen für die Funktionserweiterung des Zielsystems detektiert und in Methodendeklarationen überführt.
Dafür soll ein Ansatz verfolgt werden, der die Intention des Nutzers, dem Zielsystem neue Funktionen beizubringen, anhand von syntaktischen und semantischen Mustern in der Spracheingabe prüft. Zunächst werden die be-schriebenen Arbeitsabläufe auf definierte Aktionen und zugehörige Aktionssequenzen untersucht. Aus den identifizierten Elementen wird eine Methodendeklaration und ein Methodenrumpf aus den Teilinstruktionen aufgebaut. Eine Teilinstruktion besteht dabei wiederum aus Funktionsnamen und möglichen

Parametern, welche in der Spracheingabe erkannt werden müssen. Um diese als neue Funktionalitäten in das Zielsystem integrieren zu können, müssen die extrahierten Teilinstruktionen auf bestehende Methoden der Programmierschnittstelle des Roboters abgebildet werden. Diese wird durch eine Ontologie modelliert. Zuletzt soll die Persistierung der zu lernenden Methoden für spätere erneute Verwendungen durch den Nutzer untersucht werden.

1.3 Aufbau der Arbeit

Diese Ausarbeitung wird zunächst auf die Grundlagen der natürlichen Sprachverarbeitung und des maschinellen Lernens eingehen. Weiterhin werden auch weitere Grundlagen zu Domänenontologien und Evaluationsmetriken vorgestellt. Anschließend werden wissenschaftliche Arbeiten verwandter Themengebiete dargelegt. Im darauf folgenden Abschnitt wird die Thematik der hiesigen Arbeit ausführlich analysiert und die Anwendbarkeit der verwandten Arbeiten bezüglich der Problematik dieser Arbeit diskutiert. Daraufhin wird aus den erörterten Lösungsansätzen der Analyse ein Entwurf konzipiert. Dieser Entwurf bildet die Basis der Implementierung. Um das Resultat des Lösungskonzeptes bewerten zu können, wird anschließend eine Evaluation des implementierten Werkzeuges durchgeführt. Im vorletzten Kapitel werden die in der Arbeit gewonnenen Erkenntnisse über verwendete Technologien vorgestellt. Abschließend werden im letzten Kapitel die erzielten Ergebnisse zusammengefasst und ein kurzer Ausblick auf weitere mögliche Forschungsthemen präsentiert.

2 Grundlagen

Dieses Kapitel dient dazu, einen Einblick in die verwendeten Grundlagen dieser Arbeit zu erhalten. Dafür werden im ersten Abschnitt 2.1 die Grundkonzepte natürlicher Sprache geklärt. In der Disziplin der *Programmierung in natürlicher Sprache* spielt vor allem die Komplexität dieser natürlichen Sprache eine bestehende Hürde. Deshalb werden in Abschnitt 2.2 die verwendeten Werkzeuge zur computergestützten Verarbeitung, verschiedene Distanzmetriken sowie genutzte lexikalische Datenbanken dieser Arbeit vorgestellt. Da die Problemstellung der Verarbeitung natürlicher Sprache häufig über maschinelles Lernen angegangen wird, wird in Abschnitt 2.4 eine Einführung in diese Thematik präsentiert. Weiterhin erfordert der Einsatz von Verfahren des maschinellen Lernens große Datenmengen. Deshalb wird in Abschnitt 2.5 kurz auf die mögliche Plattform Prolific zur Datenerhebung eingegangen. Im darauffolgenden Abschnitt 2.6 werden die in dieser Arbeit verwendeten Begriffe zur Beschreibung von Programmen kurz definiert. Zuletzt wird bei der Wissensrepräsentation im Bereich der Verarbeitung natürlicher Sprache oft auf Domänenontologien zurückgegriffen. Diese werden in Abschnitt 2.7 eingeführt.

2.1 Natürliche Sprache

Der Begriff der natürlichen Sprache beschreibt in der Sprachwissenschaft die gesprochene oder geschriebene Kommunikationsform der Menschen, die sich historisch herausgebildet hat. Sie steht im Kontrast zur künstlichen Sprache, welche alle Formen von Sprachen beinhaltet, die gezielt für ein spezifisches Fachgebiet entwickelt wurden. Ein Beispiel für eine künstliche Sprache sind die Programmiersprachen für die Interaktion mit einem computergestützten System. Natürliche Sprachen sind auf einem Regelwerk aufgebaut, das definiert, welche Zeichenfolge zugelassen sind. Diese Regeln werden auch als Syntax der Sprache bezeichnet. Weiterhin wird die Bedeutung dieser Zeichenfolgen durch die Semantik definiert. Die natürliche Sprache bringt zudem verschiedene Freiheiten mit sich, die die Disziplin der Verarbeitung natürlicher Sprache zu lösen versucht. Deshalb werden im folgenden verschiedene Werkzeuge zur Verarbeitung natürlicher Sprache vorgestellt.

2.1.1 Syntax

Die Syntax einer Sprache bezeichnet die Lehre vom Bau einer Texteinheit und gehört zur Grammatik natürlicher Sprachen. Sie definiert die Anordnung und Ausprägung von Zeichen und Wörtern innerhalb übergeordneter Einheiten wie Phrasen, Teilsätze oder Sätzen. Die Syntax definiert somit ein Regelwerk für die Struktur einer solchen Einheit [Eic95].

Eines der wichtigsten Konzepte dieses Regelwerks ist die dort definierte Satzgliedreihenfolge. In der englischen Sprache wird diese auf die Syntax von Subjekt-Prädikat-Objekt-Konstruktionen (SPO) festgelegt. Wird in Beispiel 2.1.1 die Wortebene als Einheit betrachtet, so ist die Struktur dieser Einheiten in den exemplarischen Eingaben syntaktisch korrekt. Betrachtet man stattdessen die beiden Eingaben auf Satzebene, so widerspricht dieser der korrekten Syntax (OSP).

Beispiel 2.1.1: Syntax

E1: „John drinks a coffee“
E2: „a coffee John drinks“

2.1.2 Semantik

Der Begriff der Semantik beschreibt in der Sprachwissenschaft die Beziehungen zwischen den Texteinheiten und deren Bedeutungen. Deshalb wird die Semantik auch als Bedeutungslehre bezeichnet. Eine Einheit kann hierbei wiederum eine Zeichenfolge, ein Wort, eine Phrase oder einem Satz entsprechen. Semantisch betrachtet ist eine Eingabe immer dann korrekt, wenn sie eine sinnhafte Bedeutung besitzt [Eic95]. Das Beispiel 2.1.2 präsentiert zwei Eingaben. Trotz dessen dass beide Eingaben syntaktisch korrekt sind, weist der zweite Satz eine inkorrekte Semantik auf. Die Eingabe lässt keine semantisch sinnhafte Interpretation erkennen.

Beispiel 2.1.2: Semantik

E1: „John drinks a coffee“
E2: „coffee drinks a John“

2.1.3 Pragmatik

Das dritte und letzte Gebiet der Zeichentheorie nach Morris [WM38] bildet neben der Syntax und der Semantik die Pragmatik. Die Pragmatik interpretiert die Bedeutung einer Texteinheit in Abhängigkeit des Kontextes. Das heißt, der Inhalt der Eingabe wird auf den Zweck einer Formulierung untersucht. In Beispiel 2.1.3 kann die erste Eingabe als warnender Ausruf oder Hinweis empfunden werden, jedoch auch als einfache Aussage in einer beschreibenden Erzählung. Genauso kann die Aussage der zweiten Eingabe als Empfehlung interpretiert werden. In einem anderen Kontext könnte sie auch als Befehl gedeutet werden.

Beispiel 2.1.3: Pragmatik

E1: „the coffee is hot“
E2: „you have to taste this coffee“

2.1.4 Satztypen

Der Satztyp oder auch die Satzart bezeichnet in der Sprachwissenschaft die Unterscheidung von Satzformen nach ihrem Handlungszweck (pragmatische Funktion) aber auch nach den formalen Aspekten. Ein solcher struktureller Aspekt ist die Stellung der finiten Verbform im Satz, das unter anderem über den Numerus (Singular, Plural), Genus (männlich, weiblich) oder Tempus (Zeitform) das Subjekt des Satzes spezifiziert. Aber auch die Satzzeichen geben Aufschluss über die Art des Satzes. Es werden im Englischen vier Satztypen unterschieden, die in Beispiel 2.1.4 durch Beispielssätze verdeutlicht werden sollen.

Der Aussagesatz, auch als Deklarativsatz bezeichnet, hat eine beschreibende Funktion. Mit dem Fragesatz oder Iterrogativsatz kann eine Frage geäußert werden. Diese Satzform wird gekennzeichnet durch ein Fragezeichen und häufig durch Fragewörter. Der Aufforderungssatz, auch Imperativsatz genannt, drückt eine Aufforderung oder eine Bitte aus. In den meisten Fällen steht die finite Verbform an erster Stelle des Satzes. Zuletzt wird unter dem Ausrufesatz oder Exklamativsatz ein Ausruf der Begeisterung oder der Überraschung verstanden. Sie werden meist gekennzeichnet durch ausdrucksstarke Adjektive. In dieser Arbeit wird jedoch die gesprochene Sprache behandelt, daher sind diese Satzzeichen nicht vorhanden.

Beispiel 2.1.4: Satztypen

Aussagesatz: „John likes Mary.“ → Kennzeichen: .
Fragesatz: „Does Mary like John?“
„Why does Mary like John?“ → Kennzeichen: ?
Aufforderungssatz: „Close the door!“
„Please close the door.“ → Kennzeichen: !, .
Ausrufesatz: „That was great!“ → Kennzeichen: !

2.1.5 Synonymie

Zwei unterschiedliche sprachliche Ausdrücke werden als synonym bezeichnet, falls sie die gleiche oder eine sehr ähnliche Bedeutung haben. Wird daher ein sprachlicher Ausdruck durch ein Synonym ersetzt, so ändert sich die Bedeutung des Satzes nicht. Jedoch gibt es Synonyme die keine perfekte Ersetzung repräsentieren, da sie eine unterschiedliche semantische Beziehung ausdrücken. Ein Synonym das ein Wort näher spezifiziert aus sein Ausdruck, wird auch als Hyponym bezeichnet. Dagegen wird eine Abstraktion eines Wortes durch eine Überordnung beziehungsweise das Hyperonym ausgedrückt wird. Weiterhin werden durch Meronyme eine weitere semantische Relation, die Teil-Ganzes-Beziehung, beschreiben. Das Gegenstück bildet das Holonym. Das Beispiel 2.1.5 soll diese unterschiedlichen Synonyme verdeutlichen. Synonyme können hierbei nicht nur für Nomen, sondern für fast alle Wortarten gefunden werden.

Beispiel 2.1.5: Synonyme

„money“ ↔ „cash, assets, buck“ (perfekte Synonyme)
„get“ ↔ „receive, gain, take in“ (perfekte Synonyme)
„happy“ ↔ „glad, pleased, thrilled“ (perfekte Synonyme)

„apple“ ↔ „fruit“ (Hyponym)
„fruit“ ↔ „apple“ (Hyperonym)
„walk“ ↔ „go“ (Hyponym)
„go“ ↔ „walk“ (Hyperonym)

„door knob“ ↔ „door“ (Meronym)
„door“ ↔ „door knob“ (Holonym)

2.2 Verarbeitung natürlicher Sprache

Die Disziplin des *Natural Language Processing*, kurz NLP, beschäftigt sich mit der rechnergestützten Verarbeitung von natürlicher Sprache. Ziel dieser Problemstellung ist es, sowohl gesprochene als auch geschriebene Sprache zu erkennen, zu analysieren und den Inhalt zur weiteren Verarbeitung zu extrahieren. Hierfür ist nicht nur das Verständnis

Tabelle 2.1: Auszug des Penn-Treebank Wortarten-Tagset

Markierung	Beschreibung
VB	Verb, Infinitiv
VBP	Verb, ungleich 3.Person Singular Präsens
DT	Artikel
NN	Substantiv, Singular
NNS	Substantiv, Plural
PRP	Personalpronomen
TO	Wort to
IN	Präposition od. untergeordnete Konjunktion

einzelner Wörtern und Sätze, sondern auch die Erfassung von komplexen Zusammenhängen und Sachverhalten notwendig.

2.2.1 Werkzeuge

Für die Verarbeitung natürlicher Sprache existieren verschiedene Werkzeuge, derartige Teilprobleme automatisiert zu lösen. Die für diese Arbeit relevanten Werkzeuge werden im Folgenden beschrieben.

2.2.1.1 Wortart-Markierung

Die Wortart-Markierung (engl. *part of speech tagging*) beschreibt die Zuordnung von Wortarten zu den Wörtern einer Eingabe. Um eine einheitliche Markierung dieser Wortarten sicherzustellen, haben sich unterschiedliche Auszeichnungssprachen etabliert, die jeweils einen unterschiedlichen Satz an Wortarten (engl. *tagsets*) umfassen. In dieser Arbeit wird der Wortart-Markierer im Penn-Treebank-Standard [MMS93] betrachtet. Das Beispiel 2.2.1 soll die Wortart-Markierung mit dem Penn-Tagset an einer exemplarischen Nutzereingabe E1 verdeutlichen. Die Bedeutungen der einzelnen Markierungen sind in Tabelle 2.1 erklärt. Aus der präsentierten Wortart-Markierung in Beispiel 2.2.1 lässt sich beispielsweise ablesen, dass es sich bei „cups“ um ein Substantiv im Plural handelt, während das Substantiv „coffee“ im Singular vorhanden ist.

Beispiel 2.2.1: Wortart-Markierung

E1: „to make coffee you have to locate the cups in the cupboard“

[TO to] [VB make] [NN coffee] [PRP you] [VBP have] [TO to]
[VB locate] [DT the] [NNS cups] [IN in] [DT the] [NN cupboard]

2.2.1.2 Phrasen-Erkennung

In der Phrasen-Erkennung (engl. *chunking*) werden die Wörter einer Eingabe in unteilbare Wortgruppen, auch als Phrasen bezeichnet, untergliedert. Dazu nutzt dieses Werkzeug die Informationen des Wortart-Markierers. Über die Phrasen einer Eingabe lassen sich weitere syntaktische Zusammenhänge zwischen den Wörtern abbilden. Wiederum soll mit dem nachfolgenden Beispiel 2.2.2 die Vorgehensweise eines Phrasenerkenners an der vorherigen Nutzereingabe E1 verdeutlicht werden. Die Tabelle 2.3 erläutert die hierbei vergebenen Markierungen. Obwohl die Phrasenmarkierungen häufig mehrere aufeinanderfolgende Wörter einer gleichen Wortgruppe zuweisen, werden die Markierungen wortweise vergeben. Um die Position eines Wortes innerhalb einer Wortgruppe weiter zu spezifizieren, hat sich das IOBES-Format als Standard etabliert. Hierbei wird pro Wort zusätzlich zur Wortart gespeichert, ob sich das Wort am Anfang (IOBES: B), am Ende (IOBES: E), innerhalb

Tabelle 2.2: Auszug des Penn-Treebank Phrasen-Tagset

Markierung	Beschreibung
VP	Verbalphrase
NP	Nominalphrase
PP	Prepositionalphrase

(IOBES-Tag: I) oder außerhalb (IOBES-Tag: O) einer Phrase befindet. Weiterhin spezifiziert die IOBES-Markierung S eine Phase, die nur aus einem einzelnen Wort besteht, beispielsweise der Name „Armar".

Beispiel 2.2.2: Phrasen-Erkennung

E1: „to make coffee you have to locate the cups in the cupboard"

[VP to make] [NP coffee] [NP you] [VP have to locate] [NP the cups]
[PP in] [NP the cupboard]

[B-VP to] [I-VP make] [B-NP coffee] [B-NP you] [B-VP have] [I-VP to]
[I-VP locate] [B-NP the] [B-NP cups] [B-PP in] [B-NP the] [I-NP cupboard]

2.2.1.3 Lemmatisierung

Eine weitere häufig eingesetzte Sprachverarbeitungstechnik bildet die Lemmatisierung. Diese bildet die morphologischen Varianten eines Wortes, zum Beispiel der Deklination oder Konjugation, auf ihren gemeinsamen Wortstamm, das Lemma, ab. Die Lemmatisierung kann hierbei auf Verben, als auch auf andere Wortarten wie Nomen oder Adverbien angewendet werden. Zur Verdeutlichung wird in Beispiel 2.2.3 eine exemplarische Nutzereingabe lemmatisiert.

Beispiel 2.2.3: Lemmatisierung

E1: „greeting means you have to look me into the eyes of the person and carefully shake my hand"

greeting → greet
means → mean
eyes → eye
carefully → careful

„greet mean you have to look me into the eye and careful shake my hand"

2.2.1.4 N-Gramme

Ein N-Gramm entsteht aus der Zerlegung einer Texteingabe in Fragmente, wobei N aufeinanderfolgende Fragmente als N-Gramm bezeichnet werden. Ein Fragment kann hierbei einem Buchstaben, Phonem oder Wort entsprechen. Wird beispielsweise wortweise segmentiert und jedes Wort einzeln in eine 1-elementige Menge untergliedert, so erhält man ein 1-Gramm. Dieser Spezialfall entspricht einer tokenisierten Texteingabe. Zerlegt man den Text in N-elementige Wortmengen aufeinanderfolgenden Wörtern, so erhält man ein N-Gramm. In Beispiel 2.2.4 wird die Bildung der Bi- und Tri-Gramme für die Beispieleingabe E1 veranschaulicht.

Tabelle 2.3: Zugewiesene semantische Rollen

Argument	spezifische Rolle	abstrakte Rolle
Arg0	Überbringer	AGENT
Arg1	überbrachter Gegenstand	THEME
Arg2	Ziel der Überbringung	DESTINATION
Arg0	Verursacher der Lokalisierung	AGENT
Arg1	lokalisierter Gegenstand	THEME

Beispiel 2.2.4: N-Gramme

E1: „to make coffee you have to locate the cups“

2-Gramm: [to make, make coffee, coffe you, you have, have to, to locate, locate the, the cups]

3-Gramm: [to make coffee, make coffe you, coffee you have, you have to, have to locate, to locate the, locate the cup]

2.2.1.5 Markierung semantischer Rollen

Eine weiterer Sprachverarbeitungsschritt bildet die Markierung von semantischen Rollen. Eine semantische Rolle beschreibt die Beziehung von Ereignissen und teilnehmenden Entitäten, wie beispielsweise Wert etwas getan hat oder was getan wurde. Dabei interpretieren sie die Prädikate einer Eingabe als Ereignisse und die Subjekte und Objekte des Verbs als Teilnehmer dieses Ereignisses, die jeweils unterschiedliche Rollen einnehmen können. Eine solche Rolle kann unterschiedlich abstrakt gewählt werden, wie beispielsweise „AGENT“ für den Handelnden, oder sehr spezifisch mit „BUYER“ für den Käufer. Bezogen auf den Sprachverarbeitungsschritt müssen für die Prädikate sowie ihre abhängigen Argumente eine Rolle vergeben werden. In Beispiel 2.2.5 wird eine beispielhafte Semantische-Rollen-Markierung zweier Nutzereingaben veranschaulicht. Die Zuordnung der Argumente auf die semantischen Rollen ist abhängig vom betrachteten Prädikat sowie dem genutzten Markierungssatz.

Beispiel 2.2.5: Semantische Rollen

E1: „Armar bring me the milk“

[Arg0 Armar] [Verb bring] [Arg1 me] [Arg1 milk]

E2: „you have to locate the cups“

[Arg0 you] [Verb locate] [Arg1 cups]

2.2.1.6 Korreferenzauflösung

Ein weitere Problemstellung der Verarbeitung natürlicher Sprache bildet die Auflösung von Korreferenzen. Eine Korreferenz beschreibt die Bezeichnung beziehungsweise Referenzierung einer einzelnen Entität durch zwei verschiedene sprachliche Ausdrücke. Die beiden linguistischen Ausdrücke beziehen sich auf die selbe Entität, das heißt sie haben den selben Referenten. Referenzen auf Entitäten, welche bereits zuvor in der Eingabesequenz eingeführt werden, werden auch als Anaphern bezeichnet. Für die Interpretation einer Eingabe muss folglich eine Beziehung zwischen diesen beiden Entitäten hergestellt werden. Das Beispiel 2.2.6 soll dieses Problem anhand zweier Nutzereingaben verdeutlichen. In der ersten

Nutzereingabe wird die Entität „milk“ durch das Personalpronomen „it“ referenziert. Um die zweite Anweisung „bring“ der ersten Nutzereingabe E1 von einem Zielsystem ausführen lassen zu können, ist eine Auflösung der Korreferenz nötig, da die Milch als zweites Argument der Methode eingesetzt werden muss. Auch in der zweiten Nutzereingabe E2 ist eine Korreferenzauflösung zwischen Eigenname „John“ und Possesivpronomen „him“ für ein Verständnis der Eingabe nötig.

Beispiel 2.2.6: Semantische Rollen

E1: „grab *the milk* and bring *it* to me“
„it“ ↔ „milk“

E2: „*John* is hungry bring *him* something to eat“
„him“ ↔ „John“

2.2.2 Zeichenketten-Distanzmetriken

Eine Zeichenketten-Distanzmetrik bezeichnet in der Information ein Verfahren zur Bestimmung des Abstands (engl. *inverse similarity*) zweier Zeichenketten. Diese Verfahren werden häufig dazu eingesetzt, um einen ungefähren Zeichenkettenabgleich oder eine Suche nach unscharfen Zeichenketten durchzuführen.

Die bekannteste Zeichenketten-Metrik ist die Levenshtein-Distanz, auch bekannt als Editier-Distanz. Sie gibt die Anzahl an Einfüge-, Ersetzungs- und Löschoperationen an, die nötig sind um eine Eingabe in eine andere umzuwandeln. Deshalb hat sie einen Wertebereich von 0, falls die Eingaben gleich sind, bis N, wobei N die Länge der längeren Zeichenkette entspricht. In Beispiel Beispiel 2.2.7 ist die Editier-Distanz der Wörter „kitten“ und „sitting“ gleich drei, da drei Operationen zur Transformation benötigt werden.

Beispiel 2.2.7: Levenshtein-Distanz

E1: „kitten“ ↔ „sitting“

1. kitten → sitten (Ersetzung: k → s)
2. sitten → sittin (Ersetzung: e → i)
3. sittin → sitting (Einfügen: g)

→ Levenshtein-Distanz = 3

Die Fuzzy-Distanz repräsentiert ähnlich zur Levenshtein-Distanz eine unscharfe (engl. *fuzzy*) Suche. Die Unterschiede zweier Zeichenfolgen werden unabhängig von ihrer Position gewichtet. Die betrachtete Fuzzy-Distanz, welche durch ein Modul des Projekts PARSE bereitgestellt wird, nutzt einerseits die Implementierung von Apache[1]. Diese vergibt für jedes identische Zeichen einer Zeichenkette einen Punkt. Bei aufeinanderfolgenden Übereinstimmungen werden jeweils zwei Bonuspunkte vergeben. Dieses Ergebnis entspricht einem Distanzmaß und liegt deshalb nicht in einem Wertebereich zwischen Null und Eins. Deshalb wurde die Implementierung der Fuzzy-Distanz im PARSE-Modul auf eine empirisch ermittelte Fuzzy-Ähnlichkeitsmetrik abgebildet: Die so ermittelte Fuzzy-Metrik ist in Gleichung 2.1 dargestellt.

$$\text{Fuzzy-Metrik}(s_1, s_2) = \frac{\text{Fuzzy-Distanz}(s_1, s_2)}{3 * \max(|s_1|, |s_2|) - 2} \tag{2.1}$$

[1]Implementierung: org.apache.commons.text.similarity.FuzzyScore

Beispiel 2.2.8: Fuzzy-Metrik

E1: „table“ ↔ „take“
Fuzzy-Distanz = 3 + 2 (Bonus)

→ Fuzzy-Metrik = $\frac{5}{3*\max(5,4)-2}$ = 0.38

E2: „red button“ ↔ „button“
Fuzzy-Distanz = 6 + 10 (Bonus)

→ Fuzzy-Metrik = $\frac{16}{3*\max(10,6)-2}$ = 0.57

Eine weitere häufig eingesetzte Zeichenkettenmetrik ist die Jaro-Winkler-Distanz. Dazu muss zunächst die Jaro-Distanz erklärt werden, welche die Basis dieser Metrik bildet. Die Jaro-Distanz aus Gleichung 2.2 berechnet sich aus der Länge der beiden Zeichenketten $s_1 unds_2$, der Anzahl m der übereinstimmenden Buchstaben (engl. *matching*) sowie die Hälfte der benötigten Transpositionen t (engl. *transpositions*). Zwei Buchstaben sind hierbei übereinstimmend, falls sie identisch sind und die Entfernung beider Buchstaben in den Zeichenketten nicht größer ist als m_d in Gleichung 2.3. Die Reihenfolge muss daher nicht identisch sein. Weiterhin bezeichnet eine Transposition die Umstellung von Buchstaben innerhalb einer Zeichenkette, sodass die zwei Zeichenketten identisch sind. Die Jaro-Distanz funktioniert sehr gut bei Zeichenketten mit geringen Abweichungen. Dies wird durch die Eingabe E1 in Beispiel 2.2.9 verdeutlicht, welche trotz zweier Tippfehler eine Ähnlichkeit von 0.84 erreicht. Durch die gegebene Definition der gemeinsamen Zeichen ergeben sich aber Probleme mit Zeichenketten, die einen längeren Zeichenblock als Präfix beinhalten. Dies soll durch die zweite Eingabe E2 des Beispiels verdeutlichen, welche eine Ähnlichkeit von 0.7 erzielt, obwohl der Unterschied nur in einer nachgelagerten Zeichenkette liegt.

$$\text{Jaro-Distanz}(s_1, s_2) = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) \tag{2.2}$$

$$m_d <= \left\lfloor \frac{\max(|s_1|, |s_2|)}{2} \right\rfloor - 1 \tag{2.3}$$

Beispiel 2.2.9: Jaro-Distanz

E1: „coffee“ ↔ „cofefee“

$|s_1| = 6$, $|s_2| = 7$
$m_d = \left\lfloor \frac{\max(6,7)}{2} \right\rfloor - 1 = 2 \rightarrow m = \{\text{c, o, f, f, e, e}\} = 6$
$t = 2$
→ Jaro-Distanz = $\frac{1}{3}(\frac{6}{6} + \frac{6}{7} + \frac{6-2}{6}) = 0.84$

E2: „cereals“ ↔ „cereal box“

$|s_1| = 7$, $|s_2| = 10$
$m_d = \left\lfloor \frac{\max(7,10)}{2} \right\rfloor - 1 = 4 \rightarrow m = \{\text{ c, e, r, e, a, l }\} = 6$
$t = 0$
→ Jaro-Distanz = $\frac{1}{3}(\frac{6}{7} + \frac{6}{10} + \frac{6-0}{6}) = 0.7$

Um die Jaro-Ähnlichkeit hinsichtlich dieses Problems zu verbessern, wurde die Jaro-Winkler-Distanz entwickelt. Bei zwei gegebenen Zeichenketten mit einem gemeinsamen Präfix p und einem Korrekturfaktor f, der die Korrektur der Ähnlichkeit in Abhängigkeit von

dem gemeinsamen Präfix darstellt, ergibt sich die Formel in Gleichung 2.4 für die Jaro-Winkler-Distanz. Die Jaro-Distanz wird hierbei mit JD abgekürzt, die Präfixlänge $|p|$ auf ein Maximum von vier begrenzt und ein Korrekturfaktor f standardisiert von 0.1 verwendet. Für das Beispiel 2.2.10 ergibt sich für die zweite Eingabe durch den gleichen Präfix beider Zeichenketten ein gesteigerter Ähnlichkeitswert von 0.82.

$$\text{Jaro-Winkler-Distanz}(s_1, s_2) = JD(s_1, s_2) + |p| * f * (1 - JD(s_1, s_2)) \tag{2.4}$$

Beispiel 2.2.10: Jaro-Winkler-Distanz

E1: „coffee“ ↔ „cofefee“

$JD(s_1, s_2) = 0.84$
$\rightarrow$ Jaro-Winkler-Distanz$(s_1, s_2) = 0.84 + 2 * 0.1 * (1 - 0.84) = 0.872$

E2: „cereals“ ↔ „cereal box“

$JD(s_1, s_2) = 0.7$
$\rightarrow$ Jaro-Winkler-Distanz$(s_1, s_2) = 0.7 + 4 * 0.1 * (1 - 0.7) = 0.82$

2.3 Evaluationsmetriken

Für die Beurteilung von Klassifikatoren werden häufig die vier Metriken Genauigkeit, Präzision, Ausbeute, und das F_1-Maß eingesetzt. Diese Metriken basieren auf den vier möglichen Wahrheitswerten, die ein Klassifikationsergebnis annehmen kann. Diese werden im Folgenden vorgestellt.
Zur Verdeutlichung dieser Wahrheitswerte wird in der Lehre häufig das Szenario eines medizinischen Labortests herangezogen, bei dem festgestellt werden soll, ob eine Person an einer bestimmten Krankheit erkrankt ist. Die Testergebnisse dieses Labor-Schnelltests werden anschließend mit der vorliegenden Gesundheitsakte des Patienten verglichen. Der Labortest stellt somit einen Klassifikator dar, dessen Diagnosegüte evaluiert werden soll. Dabei können vier mögliche Testresultate auftreten:

- richtig positives Ergebnis (engl. *true positive*, TP): Die Klassifikation entspricht dem erwarteten positiven Ergebnis
  Analogie: Der Patient ist krank, der Test hat dies richtig angezeigt
- richtig negatives Ergebnis (engl. *true negative*, TN): Die Klassifikation entspricht dem erwarteten negativen Ergebnis
  Analogie: Der Patient ist gesund, der Test hat dies richtig angezeigt
- falsch positives Ergebnis (engl. *false positive*, FP): Der Klassifikator hat fälschlicherweise ein positives Ergebnis klassifiziert.
  Analogie: Der Patient ist gesund, aber der Test hat ihn als krank eingestuft
- falsch negatives Ergebnis (engl. *false negative*, FN): Der Klassifikator hat fälschlicherweise ein negatives Ergebnis klassifiziert.
  Analogie: Der Patient ist krank, aber der Test hat ihn als gesund eingestuft

Eine erste Metrik zur Beurteilung von Klassifikatoren ist die Genauigkeit (engl. *accuracy*) aus Gleichung 2.5. Sie gibt den Anteil aller Elemente an, die korrekt klassifiziert werden.

$$\text{Genauigkeit} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \qquad \text{Genauigkeit} \in [0, 1] \tag{2.5}$$

Als zweite Metrik wird die Präzision (engl. *precision*) in Gleichung 2.6 vorgestellt. Mit

dieser Metrik kann bestimmt werden, wie häufig positive Ergebnisse korrekt klassifiziert werden. Dazu wird der Anteil an richtig positiven Ergebnissen unter allen positiv klassifizierten Ergebnissen bestimmt. Daraus folgt, dass bei einer hohen Präzision nur wenig Elemente fälschlicherweise als positiv klassifiziert werden. Bei einer Präzision von Null wird kein positiv klassifiziertes Element korrekt klassifiziert.

$$\text{Präzision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Präzision} \in [0, 1] \tag{2.6}$$

Eine weitere häufig eingesetzte Metrik ist die Ausbeute (engl. *recall*) aus Gleichung 2.7. Die Ausbeute misst den Anteil der gefundenen Elemente, die korrekt als positiv eingestuft werden, im Bezug auf die Menge aller tatsächlich positiven Elemente. Dementsprechend bedeutet eine hohe Ausbeute, dass viele der tatsächlich positiven Elemente auch als positiv klassifiziert werden. Eine niedrige Ausbeute weißt darauf hin, dass viele der eigentlich positiven Elemente nicht als solche erkannt werden.

$$\text{Ausbeute} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{Ausbeute} \in [0, 1] \tag{2.7}$$

Die Präzision und Ausbeute sollten jedoch immer in Kombination betrachtet werden, da sie allein nicht aussagekräftig genug für die Beurteilung eines Klassifikators sind. Eine hohe Ausbeute muss nicht mit einer hohen Präzision einhergehen. So kann ein Klassifikator, der jedes Element der Eingabe strikt mit positiv bewertet eine Ausbeute von 100% erreichen, währenddessen die Präzision im selben Szenario sehr niedrig ausfallen würde. Deshalb sollte allein aus einer hohen Ausbeute beziehungsweise Präzision nicht gefolgert werden, dass der Klassifikator gute Ergebnisse liefert. Deshalb wird mit dem F_1-Maß in Gleichung 2.8 eine weitere Metrik vorgestellt. Diese bildet das harmonische Mittel zwischen Präzision und Ausbeute und bewertet damit das Zusammenspiel von Präzision und Ausbeute.

$$\text{F}_1 = 2 * (\text{Präzision} * \text{Ausbeute})/(\text{Präzision} + \text{Ausbeute}) \qquad \text{F}_1 \in [0, 1] \tag{2.8}$$

2.4 Maschinelles Lernen

Der Begriff des maschinellen Lernens beschreibt das künstliche Generieren von Wissen aus Erfahrung. In der Definition von Mitchell [Mit97] wird ein solches System für maschinelles Lernen wie folgt beschrieben:

> *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.“*

Ziel ist es, den computerbasierten Systemen zu ermöglichen aus der Erfahrung zu lernen und die Welt als Hierarchie von Konzepten (engl. *hierachy of concepts*) zu verstehen [GBC16]. Ein solches Konzept ist durch seine Beziehung zu wiederum einfacheren Konzepten gekennzeichnet. Durch das Sammeln von Wissen aus Erfahrung, womit eine Menge an Beispieldaten gemeint ist, vermeidet der Ansatz des maschinellen Lernens die Notwendigkeit, dass Menschen das gesamte Wissen formal spezifizieren müssen.
Das maschinelle Lernen steht im Gegensatz zu den regelbasierten Ansätzen. Diese bilden die Realwelt für den Algorithmus durch händisch erstellte Wissensrepräsentationen ab. In Abbildung 2.1 werden diese beiden Ansätze schematisch gegenübergestellt, wobei die blauen Boxen jene Komponenten der Ansätze repräsentieren, die fähig sind, aus Daten zu lernen. Die Grafik ist hierbei angelehnt an die Darstellung aus dem Buch *Deep Learning* [GBC16] von Goodfellow.

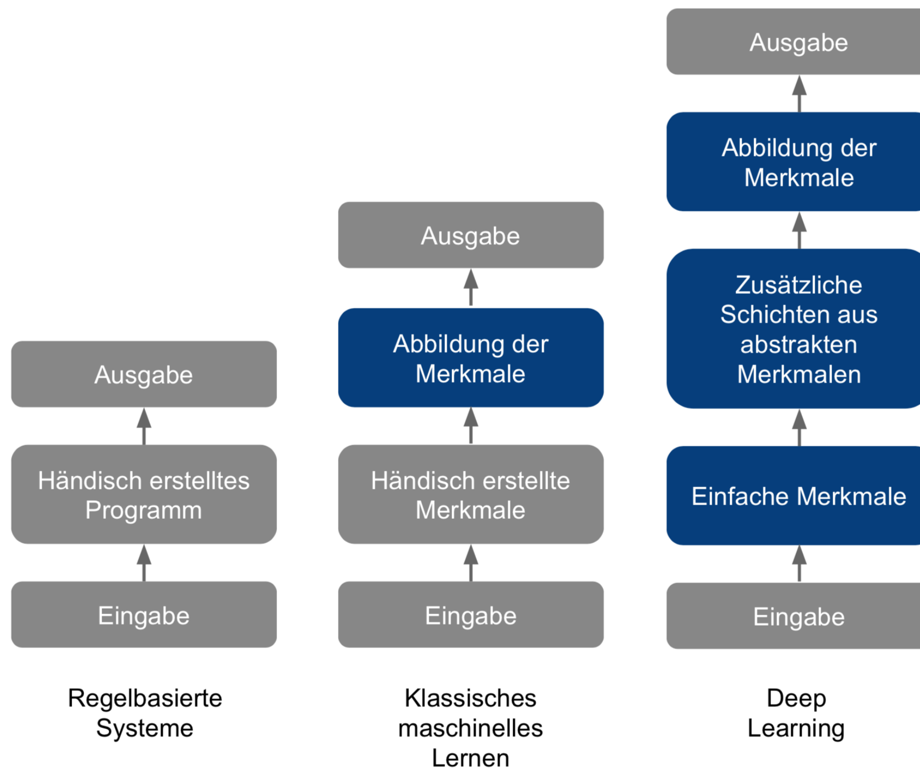


Abbildung 2.1: Schematischer Vergleich von regelbasierten und maschinellen Ansätzen

Ein maschineller Lernalgorithmus soll dazu eingesetzt werden, eine Menge von Aufgaben T (engl. *task*) zu lösen, die sonst zu schwer für ein fest verdrahtetes (engl. *hard-coded*), von Menschen geschriebenes Computerprogramm wären. Eines der bekanntesten Aufgabengebiete ist die Klassifikation. Hier soll das Computerprogramm erkennen, welcher Kategorie von Daten die Eingabe zuzuordnen ist. Ein Beispiel dafür ist das Erkennen von unerwünschten Mitteilungen (engl. *spam*) im Vergleich zu nicht Spam-Mitteilungen. Ein weiteres Aufgabengebiet ist die Regression. Hierbei soll das Computerprogramm einen numerischen Wert auf Basis von Eingabedaten hervorsagen. Diese Aufgabe ähnelt dem Klassifikationsproblem, jedoch unterscheiden sich die Ausgabeformate. Statt einer Reihe von Klassen werden die prognostizierten Werte ausgegeben. Ein Beispiel für eine Regressionsaufgabe ist die Vorhersage von zukünftigen Wetterdaten.

Das Performanzmaß P evaluiert die Fähigkeiten eines maschinellen Lernalgorithmus und ist abhängig von der Aufgabe T. Für Aufgaben wie der Klassifikation wird häufig die Genauigkeit (engl. *accuracy*, siehe Abschnitt 2.3) eines Modells gemessen. In vielen Fällen wird auch die Fehlerrate (engl. *error rate*) hinzugezogen. Wichtig ist, dass das Performanzmaß für die Beurteilung des maschinellen Lernalgorithmus auf Daten angewandt wird, die zuvor noch nicht gesehen wurden. Dafür wird ein unabhängiger Satz von Beispielen, den Testdaten, eingesetzt. Dieser soll die reale, unbekannte Welt simulieren.

Zuletzt wird für das maschinelle Lernen ein Satz aus Beispieldaten benötigt, die in der Definition von Mitchell [Mit97] als Erfahrung E bezeichnet werden. Aus diesen Beispieldaten, auch als Merkmale bezeichnet, soll mit Hilfe von maschinellem Lernen Wissen extrahiert werden. Dieser Lernprozess geschieht in der sogenannten Trainingsphase des Lernverfahrens. Im Optimalfall „erkennt“ das System Muster und Gesetzmäßigkeiten in den Daten und kann nach Beendigung des Trainings auf unbekannte Daten verallgemeinern. Dieser Lerntransfer auf neue Daten wird Generalisierung genannt. Die beim Trainingsprozess genutzte Erfahrung durch einen Algorithmus lässt sich in zwei Kategorien

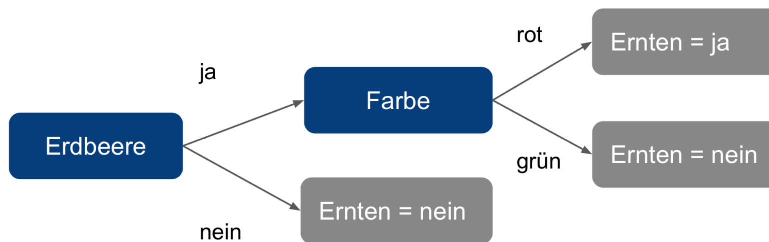


Abbildung 2.2: Entscheidungsbaum

teilen. Bei unüberwachten (engl. *unsupervised*) Lernverfahren verfügt (engl. *experiencing*) der Algorithmus über einen Datensatz aus verschiedenen Merkmalen. Es wird versucht, nützliche Eigenschaften zur Struktur der Daten zu erlernen, beispielsweise der Wahrscheinlichkeitsverteilung, die für die Erzeugung eines Datensatzes nötig ist, oder die Gruppierung (engl. *clustering*) der Daten in ähnliche Gruppen.
Dagegen erhalten überwachte (engl. *supervised*) Lernverfahren neben den eigentlichen Daten auch die erwarteten Zielgrößen (engl. *labels*). Der Algorithmus soll anschließend erlernen, diese Zielgrößen nach deren Beobachtung im Training vorherzusagen. Der Begriff des überwachten Lernens entstammt hierbei von der Sichtweise des Schülers, das von einem Lehrer demonstriert bekommt, was zu tun ist [GBC16]. Im Falle von maschinellem Lernen fehlt dieser Lehrer jedoch. Der Algorithmus muss das Lösen der Aufgabe T eigenständig erlernen. Am Beispiel des genannten Spam-Klassifikators bestünde ein Datensatz aus einer Menge von Mitteilungen sowie ihrer jeweiligen Kategorisierung in Spam und erwünschten Mitteilungen.

Eine weitere Unterteilung maschinellen Lernens ist in klassische Lernverfahren und Lernverfahren auf Basis neuronaler Netze. Diese beiden Varianten maschinellen Lernens werden in Abbildung 2.1 veranschaulicht und im Folgenden erläutert.

2.4.1 Klassische Verfahren

Klassische Lernverfahren werden häufig auch als symbolische Ansätze bezeichnet, da das Wissen, bestehend aus den Beispieldaten sowie den induzierten Regeln, explizit im Algorithmus repräsentiert ist. Die nachfolgend betrachteten Verfahren gehören zur Klasse der überwachten Lernverfahren.

2.4.1.1 Entscheidungsbaum

Entscheidungsbäume (engl. *decision trees*) stellen die aus den Trainingsdaten gewonnenen Informationen in Form eines Baumes dar. Ein solches Baumdiagramm veranschaulicht hierarchisch aufeinanderfolgende Entscheidungen, die die Daten auf bestimmte Eigenschaften testen und je nach Ausprägung in Untermengen aufteilen. In den Untermengen können die Daten bezüglich dieser Eigenschaft als ähnlich erachtet werden. Zur Klassifikation eines Datums wird der Baum von der Wurzel aus durchlaufen bis ein Blatt erreicht wird. Dieses Blatt repräsentiert die klassifizierte Klasse eines Datums.
Die Abbildung 2.2 zeigt einen einfachen Entscheidungsbaum, die jedem Datum einen Wert der binären Klassifikation bezüglich des Erntens von Erdbeeren zuordnet.

2.4.1.2 Random Forest

Random Forests erweitern das Verfahren eines Entscheidungsbäumes um eine Menge von Bäumen. Diese Mengen von Entscheidungsbäumen werden als Entscheidungswälder (engl. *forest*) bezeichnet. Das Verfahren ermittelt die Klassifikationen durch einen Mehrheitsentscheid einer Menge von Bäumen. Zusätzlich dazu werden diese Entscheidungswälder beim Random-Forest-Verfahren unter einer bestimmten Art von Zufälligkeit (engl. *random*) während des Lernprozesses erzeugt.

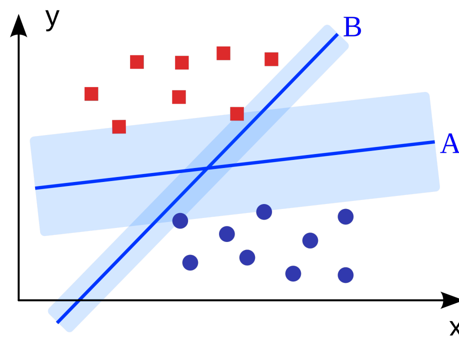


Abbildung 2.3: Support Vector Machine

2.4.1.3 Logistische Regression

Mit der logistischen Regression (engl. *logistic regression*) möchte man die Eintretenswahrscheinlichkeit eines Ereignisses (abhängige Variable) in Abhängigkeit verschiedener Einflussgrößen (unabhängige Variable) messen. Dabei repräsentiert sie einen Spezialfall der linearen Regression, bei der die abhängige Variable kategorisch statt kontinuierlich vertreten ist. Obwohl die Regression historisch Teil des Namens dieses Verfahren ist, wird die logistische Regression für Klassifikationsprobleme eingesetzt. Eine beispielhafte Problemstellung einer binären logistischen Regression ist die Klassifikation von nicht-erwünschten oder erwünschten Mitteilungen (engl. *spam*). Bei einer multinomialen logistischen Regression kann die Zielgröße mehrere Kategorien annehmen und behandelt Probleme wie beispielsweise die Klassifikation von Genres eines Buches.

2.4.1.4 Support Vector Machine

Die Support Vector Machine beziehungsweise Stützvektormaschine unterteilt eine Menge von Daten, repräsentiert durch einen Vektor in einem Vektorraum, derart, dass ein möglichst breiter Bereich um die Klassengrenzen frei von anderen Datenpunkten bleibt. Die Datentrennung wird in diesem Raum wird durch eine Hyperebene realisiert, die als Trennfläche fungiert und die Objekte in zwei Klassen teilt. Ziel ist es, den Abstand derjenigen Vektoren, die der Hyperebene am nächsten liegen, zu maximieren. Diese Vektoren werden Stützvektoren (engl. *support vectors*) genannt. Damit stellt die Support Vector Machine einen „Breiter-Rand-Klassifikator“ (engl. *large margin classifier*) dar. In Abbildung 2.3 wird eine beispielhafte Datenmenge betrachtet, welche durch zwei mögliche Hyperebenen verschiedener Randgrößen geteilt wird. Da die Hyperebene A die Klassen mit einen breiteren leeren Bereich trennt, stellt sie die bessere Lösung dar. Dieser leere Bereich soll anschließend ermöglichen, auch unbekannte Objekte möglichst zuverlässig als eine der beiden Klassen klassifizieren zu können.

2.4.1.5 Naïve Bayes

Der Naïve Bayes-Klassifikator ordnet jedes Datum anhand seiner Merkmale jener Klasse zu, zu welcher es mit der höchsten Wahrscheinlichkeit angehört. Das Verfahren erhielt seinen Namen aufgrund des Theorems von Bayes [BP63]. Formal handelt es sich um eine mathematische Funktion, die jedem Punkt eines Merkmalsraumes eine Klasse zuordnet.
Das Beispiel in Abbildung 2.4 zeigt eine abstrakte Darstellung des Naïve Bayes-Klassifikators. Als exemplarisches Problem soll der Klassifikator ein Buch verschiedenen Genres zuordnen. Da in den Beispieldaten das Fantasy-Genre am häufigsten vorkommt, wird das betrachtete Datum erst dahingegen eingeordnet. Bei Betrachtung der Merkmale des Eingabedokuments wird diese Einordnung jedoch angepasst. Im Beispiel enthält das Dokument das Wort „rot“. Dies wird als Indikator für das Genre der Romanzen interpretiert. Da das Dokument jedoch auch das Wort „Messer“ enthält, welches einen starken Indikator für das Krimi-Genre repräsentiert, wird das Dokument anhand dieser Merkmale näher an

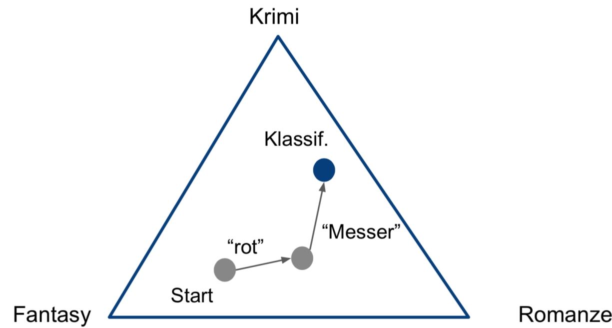


Abbildung 2.4: Naïve Bayes

das Genre der Krimibücher eingeordnet. Nach Betrachtung aller Merkmale prüft der Klassifikator, zu welcher Klasse der Datenpunkt im Merkmalsraum den geringsten Abstand besitzt. Diese Klasse wird als endgültige Klassifikation gewählt.

2.4.2 Verfahren auf Basis neuronaler Netze

Neuronale Netze zählen zu den nicht-symbolischen Ansätzen. Das Wissen ist implizit und versteckt (engl. *hidden*) repräsentiert. Daraus folgt, dass die erlernten Lösungswege der neuronalen Netze nicht direkt einsehbar sind und nur vermutet werden können. Besteht ein solches neuronales Netz intern aus mehreren Schichten, die wiederum eigene, abstrakte Repräsentationsformen der Merkmale modellieren, spricht man von tiefen neuronalen Netzen. Würde man die durch diese Netze modellierten Konzepte als Graph visualisieren, so würde dieser mehrere Hierarchieebenen (engl. *hierachy of concepts*) enthalten. Deshalb wird dieser Ansatz als *Deep Learning* [GBC16] bezeichnet. Neuronale Netze sind fähig, komplexe Zusammenhänge in den Eingaben zu erkennen, benötigen im Gegenzug jedoch deutlich größere Datenmengen für das maschinelle (Er-)Lernen als die klassischen Verfahren. Durch den Einsatz von Deep-Learning-Verfahren verspricht man sich, dass das aufwändige und komplexe Extrahieren von Merkmalen (engl. *feature engineering*), das für den Einsatz der klassischen Verfahren des maschinellen Lernens nötig ist, entfällt. Stattdessen sollen die neuronalen Netze diese Merkmale automatisch aus den Daten extrahieren. Dies wird in Abbildung 2.1 durch die dritte Spalte an Elementen visualisiert, wobei es auch auch neuronale Netze gibt, die nicht aus mehreren Schichten bestehen. In diesem Fall spricht man von einfachen künstlichen neuronalen Netzen ohne versteckte Netzschichten (siehe Abschnitt 2.4.2.1).

Künstliche neuronale Netze sind in Anlehnung an das menschliche Gehirn entstanden. Wie auch das menschliche Gehirn, bestehen sie aus sogenannten Neuronen und bilden in einem Verbund ein neuronales Netz. Dieses neuronale Netz ermöglicht es, Berechnungen durchzuführen, welche als Funktion ausgedrückt werden können. Um die Grundlagen eines neuronalen Netzes zu erläutern, muss zunächst das einzelne Neuron erklärt werden. Ein solches Neuron wird in Abbildung 2.5 visualisiert.
Das Neuron besteht aus einer Menge von Gewichten w_{ij}, die trainierbare Parameter darstellen und eine Menge an Eingaben x_i annehmen können. Bevor die Eingabewerte in das Neuron weitergeleitet werden, werden die einzelnen Werte mit dem jeweiligen Gewicht multipliziert. Die Übertragungsfunktion summiert diese anschließend zu einem Wert, der sogenannten Netzeingabe net_j, auf. Anschließend wird die Netzeingabe mit einem Schwellwert θ addiert, welcher ebenfalls einen trainierbaren Parameter des neuronalen Netzes darstellt. Daraufhin wird die vorverarbeitete Eingabe an eine sogenannte Aktivierungsfunktion φ übertragen. Diese gibt je nach Eingabe und zugrundeliegender Funktion eine bestimme Aktivierung o_j aus. Die beschriebene Berechnung lässt sich als folgende Funktion

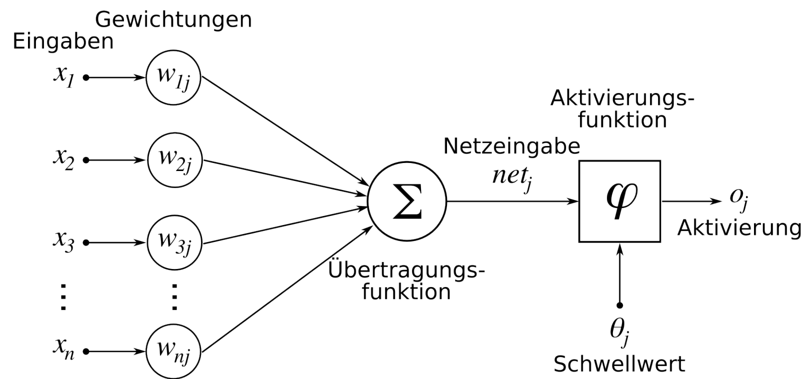


Abbildung 2.5: Künstliches Neuron[2]

in Gleichung 2.9 ausdrücken.

$$o_j = \varphi\left(\sum x_n * w_{nj} + \theta_j\right) \tag{2.9}$$

Ein neuronales Netz, das aus einem einzigen Neuron besteht, wird als Perzeptron bezeichnet. Ein Perzeption ist jedoch sehr beschränkt, da es nur sehr einfache Funktionen abbilden kann. Deshalb werden neuronale Netze aus einem Verbund dieser Neuronen aufgebaut, welche jeweils in verschiedenen Schichten (engl. *layer*) organisiert werden. Auf diese Weise wird die Berechnung komplexer Funktionen ermöglicht.

2.4.2.1 Künstliches neuronales Netz

Die einfachste Form eines solchen neuronalen Netzes ist das künstliche neuronale Netz (engl. *artificial neural network, ANN*). Dieses wird häufig auch als *Multilayer Perceptron* oder *Feedforward Netzwerk* bezeichnet wird. Das neuronale Netz besteht aus mehreren Schichten, die wiederum aus mehreren Neuronen aufgebaut sind. Die Neuronen innerhalb einer Schicht sind nicht miteinander verbunden, jedoch ist jedes Neuron einer Schicht mit allen Neuronen der nächsten Schicht verknüpft. Diese Eigenschaft wird auch als vollvernetzt bezeichnet und wird in Abbildung 2.6 visualisiert. Das Netz besteht weiterhin aus einer Eingabeschicht (*input layer*), welche die Daten empfängt, sowie einer Ausgabeschicht (*output layer*), die die verarbeiteten Daten in Form eines Ergebnisvektors ausgibt. Bei einer Klassifikation entspricht dieser Vektor den Wahrscheinlichkeiten für die jeweilige Klassenzugehörigkeit einer Eingabe. Zwischen diesen beiden Schichten kann eine beliebige Anzahl sogenannter versteckter Schichten (engl. *hidden layer*) eingesetzt werden. Diese Schichten werden als versteckt bezeichnet, da die Daten innerhalb dieser Schichten nicht einsehbar sind. Außerdem entsprechen diese Daten nicht mehr der ursprünglichen Netzeingabe, da sie intern in eine durch das Netz selbst erstellte, abstrakte Repräsentationsform umgewandelt werden. Mehrschichtige neuronale Netze, welche die intern berechneten Werte ausschließlich an die nachfolgende Schicht weitergeben, werden deshalb als vorwärtsgerichtete mehrschichtige neuronale Netze (engl. *feed forward network*) bezeichnet. Diese Netze repräsentieren eine mathematische Funktion, die einen Satz von Eingabedaten auf eine Menge an Ausgabewerte abbildet.

Trainingsvorgang eines neuronalen Netzes

Die beschriebene Berechnung in den einzelnen Neuronen nach Gleichung 2.9 wird als Vorwärtspropagierung (engl. *forward propagation*) bezeichnet. Dieses Vorgehen alleine würde jedoch zu falschen oder ungenauen Ergebnissen führen. Der Grund dafür ist, dass die

[2] Quelle: `https://goo.gl/UR1CJF`, zuletzt besucht am 17.06.19
[3] Quelle: `http://neuralnetworksanddeeplearning.com/chap1.html`, zuletzt besucht am 17.06.19

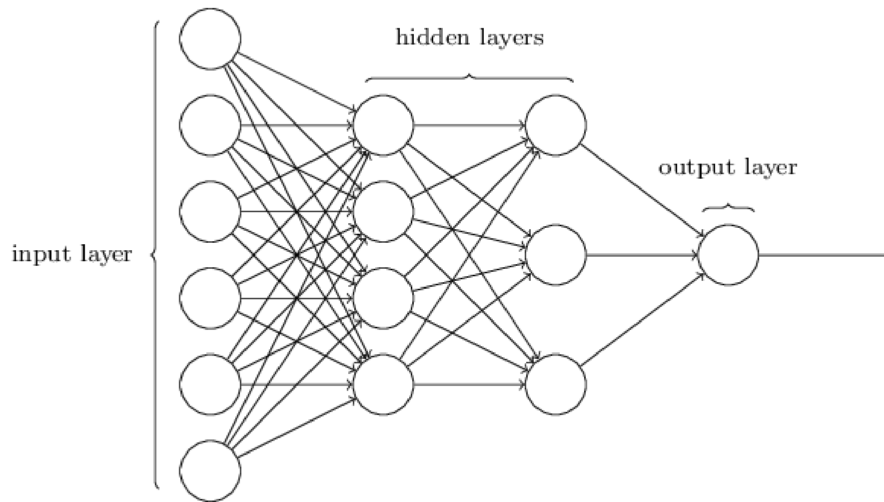


Abbildung 2.6: Künstliches neuronales Netz[3]

trainierbaren Parameter im neuronalen Netz auf die Problemstellung angepasst werden müssen. Diese Parameteroptimierung wird in dem sogenannten Rückwärtspropagierungsalgorithmus [RHW86] (engl. *backpropagation*) umgesetzt und geschieht sukzessive über mehrere Iterationen hinweg. Dieses Vorgehen wird als Training von neuronalen Netzen bezeichnet. In jeder Trainingsphase, auch als Epoche bekannt, werden die Parameter eines Netzes in Abhängigkeit von dem erzielten Fehler auf den Testdaten angepasst.

Eine Herausforderung beim Trainieren von neuronalen Netzwerken beziehungsweise des maschinellen Lernens im Allgemeinen bildet die gute Vorhersage des Modells auf neuen Daten. Diese Fähigkeit wird als Generalisierung bezeichnet. Hierzu ist einerseits die Größe des Datensatzes ausschlaggebend. Das Daumenmaß für die Größe eines Datensatzes für einen überwachten Lernansatz beträgt nach Goodfellow 5000 annotierte Daten pro Kategorie [GBC16]. Andererseits ist die Generalisierungsfähigkeit abhängig von der Dauer des Trainings. Hier kann der Effekt der Überanpassung (engl. *overfitting*) eintreten, bei dem die Gewichte des neuronalen Netzes zu sehr an die Trainingsdaten angepasst werden, sodass eine Vorhersage auf unbekannten Daten zu schlechten Ergebnissen führt.

2.4.2.2 Neuronales Netz mit langem Kurzzeitgedächtnis

Ein neuronales Netz mit langem Kurzzeitgedächtnis (engl. *long short term memory network*), kurz LSTM, ist eine Erweiterung eines rekurrentes neuronalen Netzes (engl. *recurrent neural network*, RNN). Im Gegensatz zu den vorwärtsgerichteten neuronalen Netzen aus Abschnitt 2.4.2.1, weisen rekurrente Netze zusätzliche Rückkopplungsschleifen auf. Sie stellen somit komplexere, erweiterte künstliche neuronale Netze dar. Mit Hilfe dieser Rückkopplungsschleifen ist es möglich, die Berechnungen eines Neurons nicht nur auf Basis der Neuronen in der vorherigen Schicht zu treffen, sondern ebenfalls basierend auf die Neuronen weiterer vorheriger Schichten.
LSTMs stellen spezifische rekurrente Netze dar, die bekannt dafür sind, dass sie wegen ihrer zugrundeliegenden Architektur langfristigere Abhängigkeiten in Daten erkennen können [HS97]. Bezogen auf textuelle Eingabedaten können LSTMs über mehrere Wörter oder gar Satzgrenzen hinweg Beziehungen modellieren. Dazu liest das Netz die Wörter der Eingabe nacheinander ein und klassifiziert diese abhängig von den Klassifikationen vorheriger Wörter. Jedoch wird in den Netzen der syntaktische und semantische Kontext nur unidirektional aufgebaut. Deshalb wurden bidirektionale Architekturen entworfen, die die Eingaben in beide Richtungen einlesen und somit bidirektionalen Kontext aufbauen können [SP97].

2.4.2.3 Geschaltete Rekurrente Einheit

Die geschaltete rekurrente Einheit (engl. *gated recurrent unit, GRU*) stellt eine Vereinfachung der LSTM-Architektur aus Abschnitt 2.4.2.2 dar. Einerseits wird die Anzahl an Neuronen beziehungsweise die sogenannten Schaltungen (engl. *gates*) reduziert. Andererseits wird auch die interne Speicherrepräsentation vereinfacht, sodass GRUs weniger rechenintensiv als LSTM-Netzwerke ausfallen. Weiterhin kann auch die häufig in Kombination mit LSTMs eingesetzte bidirektionale Schicht auf GRU-Architekturen angewendet werden.

2.4.2.4 Faltendes neuronales Netz

Ein faltendes neuronales Netz (engl. *convolutional neural network, CNN*) ist ein spezielles künstliches neuronales Netz (siehe Abschnitt 2.4.2.1), das bisweilen für die Bild- und Spracherkennung eingesetzt wurde. Jedoch werden sie immer häufiger auf Problemstellungen der Verarbeitung natürlicher Sprache [CW08], wie dem semantischen Zerteilen oder der Klassifizierung von Sätzen, angewendet.
Basis für die faltenden Netze bilden die Faltungsschichten, welche zur Mustererkennung in den Daten eingesetzt werden. Bezogen auf den Anwendungsfall der Bildverarbeitung können damit Merkmale wie Linien und Kanten extrahiert werden. Weiterhin werden sogenannte Pooling-Schichten verwendet, die die Eingabedaten verdichten und reduzieren. Zuletzt wird eine vollvermaschte Schicht eingesetzt, um die berechneten Werte des Netzes auf ein Klassifikationsergebnis abzubilden. Ein faltendes Netz kann hierbei aus mehreren Abfolgen dieser drei Schichten aufgebaut sein.

2.4.2.5 BERT

Bert [DCLT18a] ist ein von Google auf großen, nicht annotierten Datenmengen vortrainiertes Sprachmodell[4] (engl. *language model*). Die Basis von Bert bildet ein tiefes, bidirektionales neuronales Netz, das auf über drei Milliarden Wörtern aus dem Internet unüberwacht trainiert wurde. Zusätzlich dazu modelliert das Sprachmodell neben syntaktischen Zusammenhängen auch semantischen Kontext. Dieser Prozess wird von Google als zeitaufwändiges und ressourcenintensives Vortraining (engl. *pre-training*) bezeichnet. Anschließend soll Bert in einem Anpassungsschritt (engl. *fine-tuning*) auf kleineren Datenmengen für verschiedene Sprachverarbeitungsprobleme angepasst werden können. Als Beispiel werden hierfür die Problemstellung der Fragen-Beantwortung (engl. *question answering*) sowie die Stimmungsanalyse (engl. *sentiment analysis*) genannt.

2.4.3 Wortrepräsentationen

Die Aufgaben T (siehe Abschnitt 2.4), die durch maschinelles Lernen gelöst werden sollen, werden meist dadurch beschrieben, wie der Algorithmus ein Beispiel verarbeiten soll. Ein solches Beispiel besteht aus einer Sammlung von Merkmalen, die für den Algorithmus als Vektor repräsentiert werden müssen.
Um Wörter oder Texte als Eingabe für einen Algorithmus nutzen zu können, müssen diese auf numerische Werte abgebildet werden. Dafür wird zunächst ein Vokabular über das betrachtete Textkorpus benötigt, welches jedes einzigartige Wort, auch als Token (engl. *token*) bezeichnet, einer eindeutigen Zahl zuweist. Der entstandene Wortindex wird als Grundlage für die beiden folgenden Wortrepräsentationen genutzt.

2.4.3.1 Bag-of-Words

Das Bag-of-Words-Modell (BOW), erstmals vorgestellt von Salton [Sal89], ist der einfachste Ansatz, um textuelle Eingaben im Zahlenraum zu repräsentieren. Dafür wird eine Eingabe durch eine Menge von Wörtern, ungeachtet ihrer unterliegenden Grammatik oder

[4] Quelle: `https://github.com/google-research/bert`, zuletzt besucht am 17.06.19

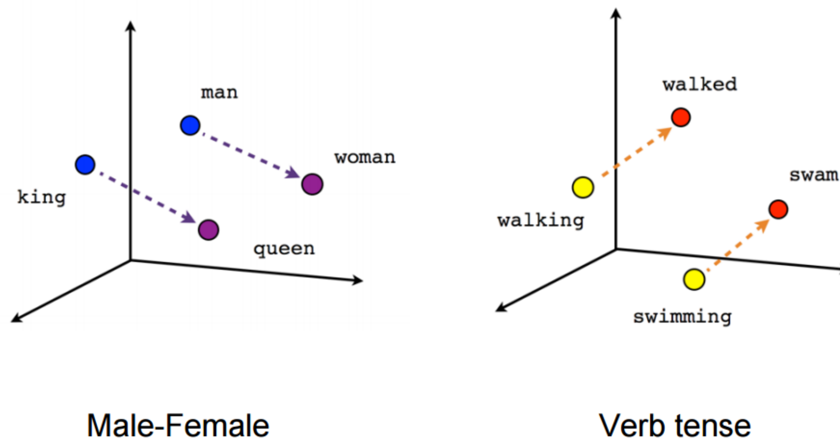


Abbildung 2.7: Zweidimensionale Wordvektorrepräsentationen[6]

Wortreihenfolge, als Vektor repräsentiert. Diese Wortmenge entspricht der numerischen Darstellung eines Vektors. Der Nachteil dieser Repräsentationsform ist, dass die Vektoren hochdimensional und dünn besetzt sind. Zusätzlich dazu lassen sich aus den Distanzen der Vektoren keine Information über linguistische Ähnlichkeit der Token im Vokabular ziehen [MCCD13]. Das Beispiel 2.4.1 zeigt eine Bag-Of-Words-Repräsentation einer exemplarischen Eingabe.

Beispiel 2.4.1: Bag-of-Words

E1: John likes to watch movies Mary likes movies too E2: John also likes to watch football games

	John	likes	to	watch	movies	Mary	too	also	football	games
E1	1	2	1	1	2	1	1	0	0	0
E2	1	1	1	1	0	0	0	1	1	1

2.4.3.2 Wortvektoren

Wortvektoren stellen niedrigdimensionale, dichte Vektoren dar, welche einen Token im mehrdimensionalen Raum repräsentieren. Diese Vektoren spannen visuell betrachtet einen Merkmalsraum auf. Somit bewältigen sie die Probleme des Bag-of-Word-Modells (siehe Abschnitt 2.4.3.1). Es gibt verschiedene Algorithmen, diese Wortvektorrepräsentationen zu erstellen, sie basieren jedoch alle auf dem Konzept von Firth „*you shall know a word by the company it keeps*“ [Fir68]. Überdies existieren unterschiedliche quelloffene (engl. *open source*) Wortvektor-Bibliotheken, die auf großen Datenbasen vortrainiert wurden. Eine dieser bekannten Sammlungen bildet die in dieser Arbeit eingesetzte Wortvektorbibliothek Fasttext[5].
Durch die Modellierung der Token im Vektorraum gibt die Kosinusdistanz zweier Vektoren Aufschluss über die Beziehung zweier Wörter. Überdies finden sich verwandte Wörter im Vektorraum als Nachbarn wieder. Dies soll das Beispiel aus Abbildung 2.7 verdeutlichen. Demnach gleicht der Abstand a der Vektoren der Wort „man“ und „woman“ dem Abstand b der Vektoren zu „king“ und „queen“. Somit ergibt die Addition von „king“ + a den Datenpunkt für das Wort „queen“ oder die Addition von „man“ + b den Datenpunkt für das Wort „queen“.

[5]Quelle: `https://github.com/facebookresearch/fastText`, zuletzt besucht am 17.06.19
[6]Quelle: `https://www.tensorflow.org/images/linear-relationships.png`, zuletzt besucht am 17.06.19

2.5 Microtasking-Plattform

Als *Microtasking-Plattform* werden auf Kleinstaufgaben (Microtasks) spezialisierte Internet-Marktplätze bezeichnet, bei denen qualifizierte Internetbenutzer bestimmte Aufgaben gegen Bezahlung bearbeiten. Die dabei stattfindende Auslagerung von Arbeitslasten ins Internet an eine Vielzahl von Internetnutzern wird als *Crowdsourcing* bezeichnet.
Eine der bekanntesten und ältesten Anbieter solchr Plattformen ist *Amazon Mechanical Turk*[7]. Eine Alternative dazu bildet *prolific.ac*[8], welcher besonders auf wissenschaftliche Aufgaben, wie beispielsweise akademischen Studien, ausgerichtet ist.

Innerhalb der Microtasking-Plattfornen gibt drei wichtige Rollen: Aufgabe (Microtask), Forscher (Researcher) und Arbeiter (Worker). Ein Microtask besteht aus einer nicht (effizient) durch den Computer automatisierbaren Aufgabe. Zu häufig angebotenen Aufgaben zählen das Erkennen von Objekten in Foto oder Videoaufnahmen, das Transkribieren von Audiodaten oder die Recherche von Informationen. Ein Forscher bietet eine Umfrage an, um Daten für seine Forschung zu gewinnen. Für ihn besteht die Möglichkeit, die Umfrage für eine bestimmte Zielgruppe, beispielsweise „männlich, über 40, Herkunft in Großbritannien“, näher einschränken. Ein qualifizierter Arbeiter bearbeitet die Aufgabenstellung der Umfrage. Bei erfolgreichem Abschluss erhält er eine Bezahlung.

2.6 Programmsyntax

Zur Verdeutlichung der in dieser Arbeit verwendeten Begriffe zur Beschreibung von Programmen wird in Quelltextausschnitt 2.1 ein Pseudocode präsentiert. Als Methode beziehungsweise Methodendefinition wird in dieser Arbeit die Definition einer neuen Funktionalität, beschrieben durch eine neue Methode (`myMethod`) verstanden. Diese kann dabei unterteilt werden in Methodenkopf und Methodenrumpf. Der Methodenkopf, in der Programmierung häufig auch als Methodensignatur bezeichnet, spezifiziert den Methodennamen (`myMethod`) sowie benötigte Parameter (`argument1, argument2`). Diese können, wie auch die Funktion selbst, verschiedenen Datentypen zugewiesen sein (`type1, type2`). Im Methodenrumpf wird der eigentliche Programmablauf, die sogenannte Prozedur der Methode, definiert. In Quelltextausschnitt 2.1 reicht dieser von Zeile 2 bis Zeile 6 und wird in den meisten Programmiersprachen durch Klammern umschlossen. Der Methodenrumpf kann lokale und globale Variablen (`variable1`) enthalten. Weiterhin kann er anderorts definierte Funktionen aufrufen (`functionCall1, functionCall2`).

```
1  function myMethod(argument1::type1, argument2::type2) {
2    var variable1 = 1
3
4    functionCall1(x, argument1)
5    var variable2 = functionCall2(argument2)
6  }
```

Quelltextausschnitt 2.1: Programmsyntax

2.7 Ontologien

Neben der philosiphischen Definition, in welcher eine Ontologie die „Lehre vom Sein“ und den damit verbundenen Erkenntissen beschreibt, wird sie auch in der Informationstechnik dazu genutzt Wissen zu darzustellen. In der Arbeit *A Situated Ontology for Practical NLP* beschreiben Mahesh et al. [MN96] die Ontologie als Wissensrepräsentation über die Welt beziehungsweise einer begrenzten Domäne. Sie enthält Informationen über einzelne

[7]Quelle: `https://www.mturk.com/`, zuletzt besucht am 17.06.19
[8]Quelle: `https://prolific.ac/`, zuletzt besucht am 17.06.19

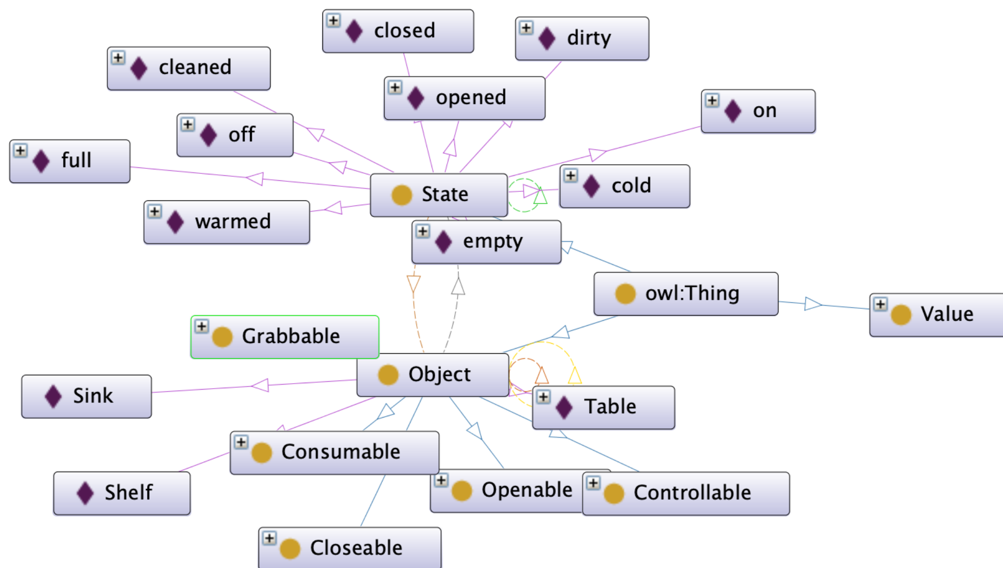


Abbildung 2.8: Graphendarstellung einer exemplarischen Ontologie

Wissenselemente, den sogenannten Individuen (engl. *instances*). Mittels dem Klassenbegriff (engl. *concept*) können Individuen in übergeordnete Kategorien eingeordnet werden. Weiterhin können Abhängigkeiten zwischen einzelnen Wissenselementen durch Relationen zwischen den Individuen ausgedrückt werden. Auf diese Weise entsteht eine hierarchische Struktur, die Taxonomie, welche eine Bedeutungsrepräsentation von Sprache ermöglicht. Zur Verdeutlichung der Wissensrepräsentation einer Domäne werden Ontologien häufig in Form eines gerichteten Graphen visualisiert. Die Klassen und Individuen bilden hierbei die Knoten des Graphen, währenddessen die Relationen durch Kanten zwischen den Knoten dargestellt werden. Abbildung 2.8 zeigt eine solche Graph-Visualisierung einer beispielhaften Ontologie. Die Individuen werden hierbei mit der Farbe Lila, die Klassen in gelb gekennzeichnet. Es ist eine große Anzahl an Relationen zwischen den Entitäten erkennbar, die wiederum durch unterschiedlichen Farben markiert sind.

3 Projekt PARSE

Das Projekt PARSE (Programming ARchitecture for Spoken Explanations) beschäftigt sich mit der Entwicklung einer Architektur zur Steuerung und Funktionserweiterung objektorientierter Zielsysteme mit Hilfe gesprochener, natürlicher Sprache (siehe Abbildung 3.1). Ein besonderer Fokus liegt dabei auf der modularen, agentenbasierten Architektur ProNat [WT15], welche die Integration beliebiger Eingabequellen und Zielsysteme ermöglichen soll.
Als exemplarisches Zielsystem wird der am KIT entwickelte humanoide Haushaltsroboter Armar-III [ARAS06] betrachtet. Liegt die Eingabe in gesprochener Form vor, wird diese mittels eines automatischen Spracherkenners (ASR) zunächst in Text überführt. Die gewonnenen Texte werden anschließend mit einer seichten Sprachverarbeitung (SNLP) analysiert und in eine Graphenrepräsentation überführt. Dieser Graph dient in den anschließenden Analyseschritten durch Sprachverständnisagenten (NLU) als gemeinsamen Datenstruktur und wird fortgehend mit extrahierten Informationen angereichert.

Die Architektur von PARSE ist dabei so konzipiert, dass die einzelnen Agenten unabhängig voneinander sind und somit eine parallele Verarbeitung der Eingabe möglich ist. Die Modellierung des Zielsystems wird mit einer Domänen-Ontologie umgesetzt und enthält die Attribute, Methoden und zugehörigen Parameter des Roboters. In der Nachverarbeitung wird diese Ontologie zusammen mit dem annotierten Graphen schließlich als Informationsquelle für die Generierung von ausführbarem Quellcode genutzt.

Es wurden bereits verschiedene Komponenten des Projekt PARSE aus Abbildung 3.1 umgesetzt. Zu Beginn wurde im Rahmen einer Arbeit von Günes [Gü15] ein Sprachkorpus aus Anweisungssequenzen für das betrachtete Zielsystem Armar-III erstellt. Anschließend wurde in einer weiteren Arbeit von Schneider [Sch15] ein Konzept entworfen, die Nutzereingaben sowie aller weiteren gemeinsamen Informationen der Komponenten im Projekt PARSE zu repräsentierten. Hierfür wurde eine Graphdatenstruktur als logische Repräsentationsform gewählt. In einer weiteren Arbeit von Paskaran [Pas15] wurden anschließend verschiedene automatische Spracherkenner gegeneinander evaluiert, um die Überführung von gesprochener Sprache in Textform für das Projekt PARSE zu integrieren. die Google Speech API[1] als Spracherkenner für das Projekt ausgewählt, dessen Ausgaben einer weiteren Arbeit von Scheu [Sch16] aufbereitet wurden. Anschließend wird in der Arbeit von Kocybik [Koc15] die eigentliche Transformation des generierten Textes in die entworfene Graphdatenstruktur durchgeführt. Weiterhin wurden Sprachvorverar-

[1] Quelle: `https://cloud.google.com/speech-to-text/`, zuletzt besucht am 17.06.19

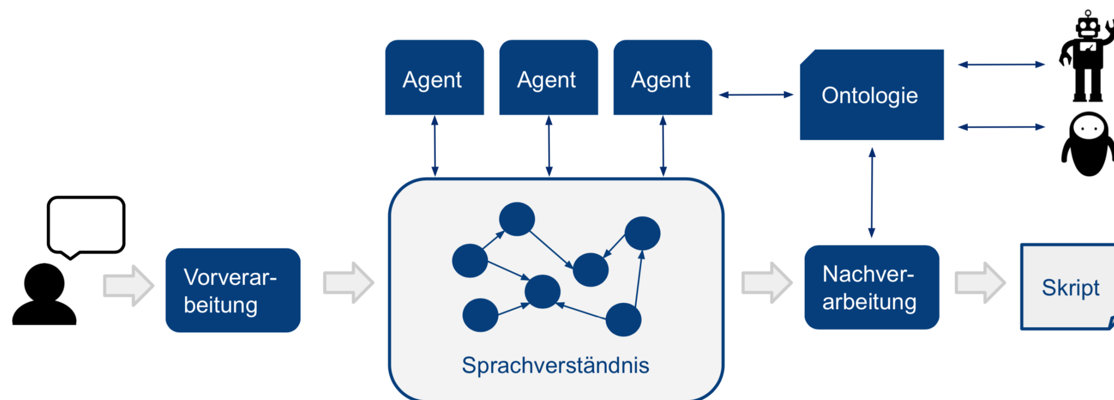


Abbildung 3.1: Schematische Darstellung der Architektur von PARSE

beitungswerkzeuge, wie beispielsweise die Wortart-Markierung integriert. Diese wird in PARSE durch eine Kombination aus dem Standford Part-of-Speech-Tagger [TKMS03] und SENNA [CWB+11] im Penn-Treebank-Standard umgesetzt und der Arbeit von [Kie16] weiter für das Projekt optimiert. Anschließend wurde in der Arbeit von Steurer [WHS18a] ein erster Semantikanalyse-Agent implementiert, welcher sich mit der Erkennung von Bedingungen sowie weiteren Kontrollstrukturen [WHS18b], in natürlichsprachlichen Anweisungen beschäftigt. Weiterhin wird die Analyse von Kontext sowie Korreferenzen durch einen weiteren Agenten in der Arbeit von Hey [Hey16] durchgeführt. Neben der Arbeit von Ou [Ou16], welche die Detektion von Aktionen in gesprochener Sprache implementiert, wird mit der Arbeit von Bitterwolf [Bit18] die Aktionsidentitätsflösung umgesetzt. Überdies wird PARSE durch die Arbeit von Schlereth [Sch17] um eine Dialogkomponente erweitert. Mit der Arbeit von Hochweiß [Hoc18] wird anschließend ein Ansatz zur Analyse und Korrektur von Disfluenzen in gesprochenen Anweisungen implementiert. Zuletzt wird in der Arbeit von Keim [Kei18] wird die Wortdisambiguierung sowie die Extraktion von Themen zur Domänenauswahl in das Projekt PARSE integriert. Schließlich wird in einem weiteren Semantikanalyse-Agenten von Hamann [Ham18] ein Ansatz für die Vollständigkeits- und Semantiküberprüfung für gesprochene Aussagen entwickelt. Als weitere ausstehende Arbeiten für das Projekt PARSE neben dieser Arbeit ist die Abbildung der Graphrepräsentation auf abstrakte Syntaxbäume mit dem Ziel der Abbildung der Inhalte auf Programmtext.

4 Verwandte Arbeiten

Die Disziplin der *Programmierung in natürlicher Sprache* beschäftigt sich damit, computerbasierte Systeme mit Hilfe menschlicher Äußerungen steuern und konfigurieren zu können. Insbesondere sollen dazu keine Programmierkenntnisse auf Seite der Nutzer nötig sein. Derartige Problemstellungen gewinnen in der aktuellen Forschung immer mehr an Relevanz. Bekannte Anwendungsgebiete sind hierbei das Bedienen von Navigationsgeräten, intelligenten Assistenten oder das Steuern von Robotern.

In diesem Kapitel werden wissenschaftliche Arbeiten vorgestellt, die sich mit verwandten Problemstellungen auseinandersetzen. Insbesondere liegt bei der Analyse der Arbeiten der Fokus auf Ansätzen, die natürlichsprachliche Erweiterungen einer Wissensbasis in ihre Betrachtung einbeziehen, wie es auch Zielsetzung in dieser Arbeit ist.
Für diese ausgewählten Arbeiten wird jeweils kurz darauf eingegangen, inwiefern sie mit dem Gesamtprojekt PARSE und der Problemstellung dieser Masterarbeit vergleichbar sind. Im nachfolgenden Analysekapitel in Abschnitt 7.2 wird anschließend diskutiert, wie sich die betrachteten Ansätze für diese Arbeit verwenden lassen. Die wissenschaftliche Arbeiten werden im Folgenden in regelbasierte und maschinelle Ansätze (siehe Abschnitt 2.4) unterteilt.

4.1 Regelbasierte Ansätze

Metafor

Die von Mihalcea, Liu und Liebermann entwickelte Anwendung *Metafor* [LL05] wandelt natürlichsprachliche Eingaben in Programmskelette der Sprache Python um. Metafor soll Programmieranfänger bei der Konzeption von Programmstrukturen unterstützen und zielt daher weniger auf eine direkte Ausführbarkeit des generierten Quelltextes ab. Die Identifikation dieser Programmstrukturen in geschriebener Sprache beschreiben die Entwickler in der Publikation *Natural Language Processing for Natural Language Programming (NLP for NLP)* [RML06]. Die wichtigsten Satzglieder, welche beispielsweise auf eine Klasse, Methode oder ein Attribut abgebildet werden könnten, werden aus der Texteingabe extrahiert. Der Ansatz beruht dabei auf dem Lokalisieren von charakteristischen Schlüsselwörtern sowie syntaktischen Strukturen wie Subjekt-Verb-Objekt-Konstruktionen, die auf die Verbalisierung von Programmkonstrukten hindeuten.

Die Problemstellung von Metafor ist es ebenfalls, eine natürlichsprachliche Eingabe auf Programmstrukturen abzubilden. Es werden Methodenrümpfe bestehend aus Bezeichnung und Parameter aufgebaut. Während die Anwendung jedoch eine Eingabe in Form einer

geschriebenen Geschichte betrachtet, werden in PARSE Anweisungsfolgen für ein Zielsystem per Sprachaufnahme diktiert. Zudem werden in Metafor die extrahierten Satzglieder direkt als Programmelemente, zum Beispiel einer Klasse, interpretiert und als solche in Quelltext modelliert. Die hiesige Problemstellung möchte gefundene Satzglieder jedoch auf bestehende Methoden der Programmschnittstelle von PARSE abbilden. Dies stellt einen zusätzlichen, komplexen Zwischenschritt dieser Arbeit dar. Einen interessanten Ansatz bildet jedoch die in der Publikation *NLP for NLP* vorgestellte Vorgehensweise für die Erkennung von Methodensignaturen.

SmartSynth

In der Arbeit *Synthesizing Smartphone Automation Scripts from Natural Language* [LGS13] von Le et al. wird die Anwendung *SmartSynth* präsentiert, welche Nutzern das Erstellen von Skripten für Mobiltelefone per natürlicher Sprache ermöglicht. Ein solches Skript könnte beispielsweise eingehende Textnachrichten während des Autofahrens (erkannt durch eine bestehende Bluetoothverbindung) vorlesen und eine automatische Antwort versenden. SmartSynth entwickelt dazu eine Modellsprache, welche die Nutzerwünsche auf eine für Mobiltelefone verständliche Skriptsprache übersetzt. Dazu wird der Inhalt der Spracheingabe zunächst auf Skript-Komponenten wie Schlüsselwörter und APIs abgebildet. Anschließend wird versucht, Datenflussrelationen zwischen den identifizierten Komponenten zu extrahieren. Werden Teile der Eingabe nicht verstanden, initiiert SmartSynth einen Dialog mit dem Benutzer, um mehr Informationen für die Verarbeitung zu erhalten. Im letzten Schritt konstruiert es das gewünschte Skript aus den erkannten Komponenten und Relationen.

SmartSynth betrachtet ein unterschiedliches Zielsystem, jedoch werden aus der Nutzereingabe ebenfalls Aktionen extrahiert und, wie in der hiesigen Arbeit geplant, auf bestehende Funktionalitäten der API abgebildet. Es können dabei nur Aufgaben in einem Skript synthetisiert werden, welche in einem einzigen Satz beschreibbar sind. Für die Abbildung nutzt SmartSynth unter anderem die Satzzeichen der textuellen Eingabe und Abhängigkeitsbäume. Da das Projekt PARSE auf komplexen, gesprochenen Anweisungen sowie einer seichten Sprachverarbeitung basiert, sind diese Informationen nicht nutzbar. Weiterhin interessant ist der Einsatz von Rückfragen durch die Anwendung, um unvollständige oder unklare Absichten zu bewältigen.

NLyze

Die Anwendung *NLyze* wird von Gulwani et al. in der Arbeit *Interactive Programming by Natural Language for SpreadSheet Data Analysis and Manipulation* [GM14] vorgestellt. Sie wurde als Zusatzmodul für das Tabellenkalkulationsprogramm Excel entworfen und stellt eine Schnittstelle für Nutzer zur Eingabe von natürlichsprachlichen Anweisungen bereit. Das System ist vorrangig auf erfahrene Nutzer ausgerichtet, welche ihre bereits existierenden, großen Datenbestände mit komplexen Operationen modifizieren möchten. Eine beispielhafte Anfrage könnte lauten „get the rows with salary bigger than 0 and sum them“. NLyze wandelt dafür die textuellen Spezifikationen des Nutzers in Operationen des Excel-Zielsystems um. Dabei werden unter anderem charakteristischen Schlüsselwörter für das Diktieren neuer Funktionen verwendet.

Die Problemstellung der Anwendung überschneidet sich mit dieser Arbeit. NLyze betrachtet Nutzereingaben in natürlicher Sprache und möchte diese als neue Funktionen für das System interpretieren. Für den regelbasierten Übersetzungsalgorithmus von Sprache in Systemkommandos werden Trainingsbeispiele zu einer Fragestellung gruppiert und auf ein einheitliches Muster abgebildet. Anschließend wird mit einem naive Bayes Klassifikator eine passendste Regel für das Kommando ausgewählt. Unterschiede zur hiesigen Arbeit

finden sich in der Beschränkung auf das Anwendungsgebiet von Tabellenkalkulationprogrammen sowie die Nutzung von textuellen Eingaben. Weiterhin ähneln die zu verarbeitenden Spracheingaben stark einer mit Schlüsselwörtern versehenen Formelsprache. Das Projekt PARSE möchte dagegen gesprochene und natürliche Eingaben verarbeiten zu können.

JustLingo

Das System *JustLingo* [WWVT15] von Wachtel et al. dient, ähnlich zur Anwendung *NLyze*, als Dialogsystemschnittstelle zur Zellenmanipulation von Tabellenkalkulationsprogrammen per natürlicher Sprache. Das System richtet sich vor Allem an Programmieranfänger, welche mit Hilfe von JustLingo die Funktionen des Programms über ein textuelles Dialogsystem vereinfacht inkrementell aufbauen und anschließend abrufen können. In einer ergänzenden Arbeit [Wie14] werden dazu Musterschablonen generiert, um neue mathematische Funktionen, beispielsweise der Mitternachtsformel, zur Anwendung hinzufügen zu können. Zur Interpretation der Nutzereingabe wird der Nutzer vom System aufgefordert, gezielte Informationen zu liefern.

Das System JustLingo möchte ebenfalls wie die hiesige Arbeit die Eingaben des Nutzers als neue Funktionen an das System interpretieren. Diese Eingaben erfolgen dabei ebenfalls in natürlicher, erklärender Weise. Die Anwendung setzt diese Wissensgenerierung jedoch vor allem durch einen informationsgenerierenden Dialog zwischen Nutzer und System um. Diese Arbeit soll mögliche Dialoginteraktionen beachten, jedoch nicht basierend auf eine Dialoginteraktion das Sprachverständnis aufbauen. Zuletzt fokussiert sich die Anwendung gezielt auf Tabellenkalkulationprogramme. Ziel von Projekt PARSE ist es dagegen, gesprochene Eingaben unabhängig von der Domäne verarbeiten zu können.

Instruction Based Learning System

Lauria et al. entwerfen in ihrer Arbeit *Mobile Robot Programming Using Natural Language* [LBKK02] eine Architektur für die Steuerung von Assistenzrobotern mit gesprochener Sprache. Betrachtet wird ein kleiner mobiler Roboter, der mittels natürlichsprachlichen Wegbeschreibungen durch eine Miniaturstadt gesteuert werden kann. Diese Befehle werden von dem verwendeten Spracherkennungswerkzeug Nuance direkt in logische Formen gebracht. Für den Systementwurf wird in einer separaten Studie [BLK+01] ein Sprachkorpus aus Wegbeschreibungen erstellt. Aus diesem wird anschließend eine Menge an primitiven Roboterfunktionen und zugehörigen symbolischen Namen zur Ansteuerung gewonnen. Die Ansteuerung des Robotersystem wird dadurch an den Sprachkorpus des Problemstellung angepasst. Weiterhin kann der Nutzer dem Roboter neue Befehle aus Komposition vorhandener Roboterfunktionen diktieren.

Es lassen sich viele Gemeinsamkeiten zum Projekt PARSE und dieser Arbeit erkennen. Ziel ist die Steuerung eines Zielsystems mit natürlich gesprochenen Instruktionssequenzen. Während die Architektur von PARSE auf einen modularen, domänen- und Zielsystemunabhängigen Entwurf ausgerichtet ist, liegt der Fokus der Architektur von Lauria auf dem laufzeiteffizienten, asynchronen Systementwurf von Sprachverarbeitung und Befehlsausführung. In Bezug zu dieser Arbeit wird ebenfalls ein Ansatz zur Funktionserweiterung des betrachteten Zielsystems beschrieben. Da die primitiven Roboterfunktionen mit symbolischen Namen aus dem Korpus gewonnen wurden, ist jedoch im Gegensatz zu dieser Arbeit eine direkte Übersetzung von Sprache möglich. Für jeden neuen Befehl wird ein Pythonskript mit Funktionaufrufen bestehender Roboterfunktionen erstellt. Neben der Quelltexterzeugung wird eine domänenspezifische Wissensrepräsentation erzeugt, welche die Wegbeschreibungen formal modelliert [LBK+01]. Mit dieser formalen Repräsentation kann überdies eine Konsistenz-Verifikation des generierten Quelltextes durchgeführt werden. Auch hier zeigt sich ein Unterschied zu Projekt PARSE, da das Sprachverständnis

dort unabhängig von der Domäne umgesetzt wird. Der korpusbasierte Ansatz hat zur Folge, dass bei fehlschlagender Verknüpfung auf bestehende Funktionen der Korpus um neue symbolische Eingaben sowie der Roboter um neue primitive Funktionen von Entwicklerseite erweitert werden muss. Als Alternative wird die Möglichkeit zur Rephrasierung der Nutzereingabe durch den Dialogkomponente angeboten.

Teaching Robots new Actions through Natural Language

Die Arbeit von She et al. [SCC+14] präsentiert einen Ansatz, einem Zielsystem neue übergeordneten Funktionalitäten (engl. *high-level functions*) per natürlichsprachlichen Kommandos beizubringen. Dazu wird ein Ansatz der Wissenserweiterung durch Dialog, ähnlich zu anderen Arbeiten [CRX+17], [WWVT15], [AKM16], verfolgt. Dazu muss der Nutzer eine Schritt-für-Schritt-Anleitung für den neuen Befehl aus einzelnen Instruktionen diktieren. Um die neue Funktion in der Wissensbasis des Systems zu persistieren, entwirft die Arbeit ein graphenbasiertes Rahmenwerk für Aktionsrepräsentationen. Die fest einprogrammierten primitiven Funktionen werden durch Vorbedingungen und Zielzuständen modelliert und direkt mit den direkten Systemansteuerungsfunktionen (engl. *low-level functions*) verknüpft. Neue übergeordnete Funktionen sind durch den Zielzustand der Umgebung nach dessen Ausführung modelliert und nicht durch die eigentliche, diktierte Instruktionssequenz. Eingeleitet wird dieses Lernkommando durch einen Dialog. Kann ein Sprachkommando nicht zugeordnet werden, so wird mittels einer Nachfrage geprüft, ob eine neue Funktion erlernt werden soll. Diese Einspeicherung muss jedoch streng nach Muster und nur unter Verwendung der bekannten primitiven Funktionen geschehen.

Im Bezug zu dieser Arbeit wird eine ähnliche Problemstellung betrachtet: Ein Zielsystem soll um neue Funktionen per Sprache erweitert werden können. Statt jedoch die Anwendung für den Umgang mit komplexen, natürlichen Nutzereingaben zu entwickeln, wird ein Dialogansatz verfolgt. Da der primitive Instruktionssatz des Roboters nur aus drei Funktionen besteht, können diese mit den Teilinstruktionen der Nutzerbeschreibung direkt verknüpft werden. Dabei wird davon ausgegangen, dass der Nutzer nur diese in seiner seriellen Beschreibung verwendet. Ausschließlich die Parameter der neuen Funktionalität, das heißt zum Beispiel Gegenstände oder Positionen, müssen auf die bestehende Wissensrepräsentation abgebildet werden.

PLOW

Die Arbeit *PLOW: A Collaborative Task Learning Agent* [ACF+07] von Allen et al. entwickelt einen webbasierten virtuellen Agenten, der durch Demonstration und Dialoginteraktion mit dem Nutzer neue, ausführbare Prozeduren erlernt. Es handelt sich bei diesen Prozeduren um Aufgaben, die innerhalb eines Webbrowsers ausführbar sind. Dazu gehört beispielsweise Informationsabruf im Internet, Flugbuchung oder das Ausfüllen von Formularen. Durch die Erweiterung der Demonstration um eine natürlichsprachliche Beschreibung der getätigten Aktionen, können auch Kontrollstrukturen umgesetzt werden. Anschließend werden die Erkenntnisse aus diesen beiden Informationsquellen genutzt, um die beschriebene Aktion zu generalisieren und in der Wissensrepräsentation abzuspeichern. Die Verknüpfung der Sprachelemente mit den HMTL-Elementen geschieht heuristisch. Anschließend können durch die Aufzeichnung der Demonstration die im Webbrowser aufgerufenen Funktionen eine Prozedur erstellt werden.

Neben einer textuellen Eingabemethode wird auch eine Sprachschnittstelle bereitgestellt. Ähnlich zu PARSE wird auch die Sprachverarbeitung wird mit einem domänenunabhängigen Zerteiler angegangen. Das Einprogrammieren neuer Funktionen ist jedoch mit sehr hohem Aufwand im Vergleich zur erreichten Aufgabenkomplexität verbunden. Weiterhin bleiben die Funktionen auf der Domäne der Webbrowser-Interaktion beschränkt.

Der Ansatz zur Elementverknüpfung sowie Unterscheidung zwischen Parametrisierung und Konstanten bilden für diese Arbeit interessante Vorgehensweisen.

4.2 Maschinelle Lernansätze

Voxelurn

In der Arbeit *Naturalizing a Programming Language* [WGLM17] von Manning et al. wird die entwickelte Anwendung *Voxelurn* beschrieben, in welcher Nutzer virtuelle dreidimensionale Objekte mit Hilfe einer Kommandozeilenumgebung aufbauen können. Instruktionen für das Programm können dabei textuell sowohl in der spezifischen Programmsyntax als auch in natürlichsprachlicher Form eingegeben werden. Ziel der Arbeit ist das „Naturalisieren“ der Programmiersprache, das heißt, das inkrementelle Erweitern der verwendeten Sprachbefehle um alternative, natürlichere Ausdrücke. Weiterhin können durch den Nutzer auch neue, komplexere Konzepte aus bekannten Funktionskandidaten hierarchisch aufgebaut werden. Sei beispielsweise der Befehl zum Bau eines Würfels und einer Pyramide bereits bekannt, könnte eine grafische Komposition dieser Objekte als Instruktion „Haus“ definiert werden, welcher wiederum, mehrfach ausgeführt, als Befehl „Häusersiedlung“ umgesetzt werden könnte. Intern wird, um Regeln für die Programmrepräsentation aus den Nutzereingaben zu abzuleiten, wird ein überwachtes logarithmisch-lineares Modell auf gesammelten Nutzereingaben trainiert.

Im Ansatz von Manning wird das System ebenfalls um neue Funktionalitäten erweitert. Voxelurn beschränkt sich dabei auf die Domäne des Aufbaus von 3D-Voxelwelten. Die grafische Nutzerumgebung wird zudem genutzt, um inferierte Objektkandidaten aus der Nutzereingabe zu visualisieren und dem Nutzer so eine Auswahlmöglichkeit zu bieten. So besteht die Möglichkeit bei Voxelurn grafisch per Versuch und Fehlschlag (engl. *trial and error*) vorzugehen. Diese Herangehensweise ist innerhalb von Projekt PARSE nicht umsetzbar oder erwünscht. Weiterhin bestehen die textuellen Befehle hauptsächlich aus kurzen, ungrammatischen Formelanweisungen, währenddessen in der hiesigen Arbeit gesprochene, natürliche Sprache betrachtet wird.

Hearthstone-Quelltexterzeugung

Die Arbeit *Abstract Syntax Networks for Code Generation and Semantic Parsing* [RSK17] von Rabinovich et al. beschäftigt sich mit der Generierung von Quelltexten aus Hearthstone-Spielkartentexten. Die auf einer Spielkarte beschriebene Funktionen, wie beispielsweise *„Verbündete Einheiten erhalten +1 Angriffskraft“*, wird anschließend als abstrakter Syntaxbaum mit diesem Inhalt erstellt. Dazu modelliert Rabinovich eine neuronale Encoder-Decoder-Netzarchitektur, bestehend aus mehreren modularen, rekursiv aufrufbaren Dekodiereinheiten für bestimmte Baumstrukturen. Für die Enkodiereinheit wird ein bidirektionales LSTM eingesetzt. Überdies wird der Attention-Mechanismus auf die Eingaben jedes Dekodierschrittes des neuronalen Netzes eingesetzt. Ausgabe des Modells ist eine Baumstruktur nach ASDL-Schemata [WAKS97], welche auf spezifische Quelltextfragmente abgebildet werden kann. Auf diese Weise ist es möglich, auch verschachtelte Funktionen zu übersetzen. Komplex sowie imperativ formulierte Sätze stellen jedoch ein großes Problem dar.

In Bezug auf das Projekt PARSE, bildet die Arbeit von Rabinovich englischsprachliche Texte auf Quelltext ab. Viele der Spielkarten besitzen jedoch eine sehr kurze, nicht umbedingt natürlichsprachlich formulierte und oft auch nur aus einem Wort bestehende Beschreibung. Der Ansatz zielt zudem nicht auf einen korrekt typisierten oder ausführbaren Quelltext aus. Überdies stellt das beschriebene Vorgehen ein Ende-zu-Ende-Ansatz dar. PARSE hingegen versucht die Abbildung von Sprachkommandos auf Quelltext in einer Verarbeitungsprozedur aus mehreren modularen Komponenten für Sprachverständnis aufzubauen.

Der neuronale Ansatz, um eine Eingabe auf eine Programm-Repräsentation abzubilden, ist für diese Arbeit interessant.

Almond

In der Arbeit *Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant* [CRX+17] von Campagna et al. wird die Chatanwendung *Almond* entwickelt, unterschiedliche Dienste per geschriebener Sprache zu steuern. Zu diesen Diensten zählen beispielsweise der digitale Kalender, Handyfunktionen wie Nachrichten und Telefonie oder die Steuerung intelligenter Lampen und Klimasensoren. Almond implementiert diese Dienste nicht selbst, sondern koppelt sie in einer gemeinsamen, per Textnachricht ansteuerbaren Schnittstelle. Es gibt drei Komponenten, wie ein Kommando an Almond aufgebaut sein kann: Systemaktion, Wissensanfrage und Auslöserspezifikation. Mit diesen kann der Nutzer eine Handlung vom System einfordern, eine Frage stellen und den gewünschten Zeitpunkt spezifizieren. Wird ein Kommando an Almond diktiert, so werden diese drei Komponenten durch Schlüsselwörter in der Eingabe unterschieden und anschließend verschiedene Prozeduren eingeleitet. Dies wird intern durch einen Zerteiler gelöst, welcher die Ähnlichkeit von Eingabe und Systemfunktionalitäten ermittelt. In einer Studie wird für jede der Systemfunktionen synonyme Formulierungen gesammelt, welche als Trainingsdaten für diese Abbildung dienen. Zusätzlich werden auch ein Datensatz zu den die eigentlichen Nutzerkommandos mit einem szenarienbasierten Vorgehen erhoben. Während neue Funktionalitäten nur von Entwicklerseite hinzugefügt werden können, ist es möglich, Paraphrasen zu bekannten Kommandos hinzuzufügen. Dazu kann der Nutzer eine neue Formulierung eingeben, das System in Lernmodus versetzen und die gewünschte Systemaktion per GUI auswählen.

Die Anwendung ähnelt Projekt PARSE insofern, dass per natürlicher Sprache Systemfunktionen angesprochen werden können. Allerdings liegt bei Almond der Fokus weniger auf einer uneingeschränkten Sprachschnittstelle, als in der Anbindung von Diensten unterschiedlicher Geräte und deren einheitliche Steuerung. Deshalb wird der Nutzer zusätzlich durch eine GUI mit einen Katalog über mögliche Formulierungen informiert. Wenn die Nutzereingabe nicht zugeordnet werden kann, startet Almond einen Dialog und fehlende Informationen zu füllen und ermittelte Kandidaten möglicher Aktionen und Anfragen zur Auswahl zu präsentieren. Bis auf den unterstützenden Dialog sind diese Herangehensweisen in PARSE nicht möglich umzusetzen. Das Vorgehen zur Datenerhebung von Campagna ist jedoch für diese Arbeit interessant.

Subevent-Detection

In *Detecting Subevent Structure for Event Coreference Resolution* [ALHM14] beschreiben Araki et al. eine Methode, Ereignis-Teilereignis-Beziehungen (engl. *subevent relations*) in Zeitungstexten zu detektieren. Eine Folge von Ereignissen (engl. *events*) kann oft in über- und untergeordnete Elemente aufgeteilt werden. Die Beziehung zwischen den Elementen dieser beiden Ebenen wird als Eltern-Kind-Relation (engl. *parent-child-relation*) bezeichnet. In der Arbeit werden derartige Beziehungen als hierarchische Cluster modelliert, wobei die Teilereignisse einem gemeinsamen Elternknoten, dem Hauptereignis, untergeordnet sind. Die Beziehung zwischen diesen Teilereignissen wird als Geschwister-Relation bezeichnet. Die Arbeit entwirft ein Klassifikator auf Basis einer logistischen Regression, welcher die Beziehung zweier Elemente des Datensatzes in eine von vier Klassen, beispielsweise der Eltern-Kind-Relation, einordnet. Als Eingabe dient ein großer Satz aus linguistischen Merkmalen der Eingabe. Anschließend entscheidet ein Bewertungsalgorithmus welcher der Knoten eines Clusters den Elternknoten bildet. Die detektierten Beziehungen können anschließend bei der Auflösung von Ereignis-Korreferenzen eingesetzt werden, die zwei Erwähnungen des selben Ereignisses miteinander verknüpft.

Trotz dass die Problemstellungen dieser Arbeiten sich auf den ersten Blick unterschieden, lassen sich gewisse Parallelen zwischen der Arbeit von Akari und der hiesigen Arbeit erkennen. Statt Lehrsequenzen und deren Inhalt detektieren zu wollen, werden zusammengehörige Ereignisse in eine hierarchische Struktur gebracht. Auf diese Weise können Teilschritte in Beziehung zu einem übergeordneten Ereignis gebracht werden. Dies könnte man wiederum als verwandten Ansatz der hiesigen Problemstellung ansehen, da auch ein Hauptereignis (hier Methodendefinition) bestehend aus Teilereignissen (hier Methodeninhalt) detektiert wird. Auch der logistische Regressionsansatz auf Basis linguistischer Merkmale ist interessant für diese Arbeit. Trotz der vielversprechenden Herangehensweise erreicht der Ansatz jedoch im Mittel nur 56% F1 für die Zuweisung von Teil- und Hauptereignis sowie 61% F1 für die Detektion von ebenbürdigen Schwester-Ereignissen.

Extracting Subevents

In der Arbeit *Extracting Subevents via an Effective Two-phase Approach* [BH16] von Badgett und Huang wird ein Ansatz vorgestellt, der automatisiert Teilereignisse (engl. *subevents*) aus Nachrichtenkorpora zum Thema Bürgerunruhen extrahiert. Es wird ein zweistufiger Ansatz zur Klassifikation von Teilereignissen verfolgt. In der ersten Phase werden die Sätze des Korpus binär klassifiziert, ob sie eine Beschreibung aus Teilereignissen enthalten. Eine solche Beschreibung wird als, durch Konjunktionen verbundene, Aneinanderreihung von Verb-Objekt-Konstruktionen definiert. Dazu zählen beispielsweise die Beschreibung *„burning tires“* oder *„taunted the police and throw stones“*. Um das Entscheidungsproblem zu lösen, wird ein künstliches neuronales Netz (ANN) mit Bootstrapping-Ansatz trainiert. Die Nomen und Verben werden mittels den vortrainierten Wortvektoren von *Stanford GloVe*[1] repräsentiert. Als Merkmale dienen initial 8 exemplarische Teilereignisse, die auf einen solchen Satz aus Teilereignis-Beschreibungen hindeuten. Damit können aus einem Datensatz von 230000 Zeitungsartikeln über Bürgerunruhen 5000 Sätze gefunden werden. Auf diesen wird ein neuronales Netz trainiert. Werden neue, weitere Sätze mit hoher Wahrscheinlichkeit dieser Gruppe von Beschreibungen zugeordnet, so werden die Trainingsdaten für die darauffolgende Bootstrapping-Iteration um diese erweitert. Nachdem in der ersten Stufe eine Menge von Sätzen als Beschreibung klassifiziert wurde, werden in der zweiten Stufe die enthaltenen Teilereignisse extrahiert. Dazu werden alle Konstrukte von Verb-Objekt-Konjunktionen ausgewählt, die mindestens zwei Mal im Korpus auftreten.

Die Arbeit von Badgett behandelt eine deutlich andere Problemstellung wie diese Arbeit. Sie möchte zu verschiedenen Artikeln eines übergeordneten Themas beziehungsweise Ereignisses, den Bürgerunruhen, häufige Teilereignisse gewinnen. Mit diesen Teilereignissen sollen anschließend auf unbekannten Daten wiederum Artikel zu diesem Thema gefunden werden können. Für die hiesige Arbeit ist der beschriebene neuronale Klassifikationsansatz interessant, da die Eingabedaten ebenfalls binär auf enthaltene Beschreibungen (hier Lehrsequenzen) klassifiziert werden könnten. Dazu behandelt Badgett ebenfalls natürlichsprachliche Texte, implementiert die Klassifikation jedoch stark auf eine Domäne beschränkt. So vermutet sie, dass die verwendeten Merkmale ihres Ansatzes beispielsweise für Zeitungsberichte über Sportthematiken nicht anwendbar sind.

Alternative Arbeiten von Jurafsky und Vogel [VJ10] interpretieren die betrachtete Abbildungsaufgabe als Planungsproblem. Für die Lösung nutzen sie lineare Strategieschätzer (engl. *linear policy estimators*), die mit Hilfe von Markov'schen Entscheidungsmodellen eine Abbildung natürlichsprachlicher Instruktionssequenzen auf Navigationsaktionen erlernen. Diese Aktionen werden bei derartigen Ansätzen als Zustandsübergänge der Umgebungselemente beobachtet. In der Arbeit von Branavan et al. [BZB10] wird das Szenario betrachtet, natürlichsprachliche Befehle aus Anleitungstexten auf GUI-Kommandosequenzen für das Betriebssystem Windows zu übersetzen. Für die Lösung dieser Problemstellung

[1]Quelle: `https://nlp.stanford.edu/projects/glove/`, zuletzt besucht am 17.06.19

wird ein Markov-Modell nach dem Prinzip des Lernens durch Belohnung (engl. *reinforcement learning*) trainiert. Hierbei konstruiert der Lerner wiederholt Kommandosequenzen für einen Anleitungstext, führt diese aus und beobachtet den Ausgang (Belohnung versus Bestrafung). Die Domäne der GUI-Interaktion unterscheidet sich jedoch stark von den komplexen Handlungen, die ein robotisches Zielsystem in einer realen 3D-Umgebung durchführen kann. Auch der Einsatz von Reinforcement Learning ist aufgrund einer fehlenden Simulationsumgebung im Rahmen der Arbeit nicht möglich.
In einer weiteren Arbeit von Tellex [TKD+11] wird die Problemstellung als probabilistisches Graphzustandsproblem angegangen. Eine Instruktionssequenz wird dazu abhängig von der unterliegenden hierarchischen und semantischen Struktur als Graph modelliert. Für dieses Planungsproblem wird ein Conditional Random Field-Modell trainiert, das die wahrscheinlichste Abbildung zwischen Satzbestandteilen der Aktionssequenz und zugehörigen Aktionen sowie Objekten vorhersagt. Diese Abbildung wird auf Basis einer Wissensrepräsentation getätigt, die Element-Eigenschaften, wie beispielsweise Name, Ort, Form und Beziehungen zu anderen Elementen enthält. Für das Training des Modells muss für jede Eingabe eine Musterlösung erstellt werden, welche die Inhalte mit den zugehörigen Elementen dieser Wissensrepräsentation verbindet. Ähnlich zu PARSE wird in der Arbeit von Tellex mit einem per Sprache steuerbaren Gabelstapler ein robotisches Zielsystem betrachtet, das in einer Lagerumgebung mit Objekten agiert. Auch die verwendete Wissensrepräsentation ist mit der Zielsystem-Ontologie von PARSE vergleichbar, wobei die Domäne der Küchenumgebung und der Funktionsumfang des betrachteten Roboters umfassender ausfällt.
Mei et al. präsentiert in seiner Arbeit [MBW15] einen neuronalen Ansatz, Instruktionen der 2D-Navigationsdomäne auf ihre entsprechenden Aktionssequenzen abzubilden. Er modelliert die Problemstellung als Sequenzmarkierungsproblem, das er mit einem Ende-zu-Ende-Lernverfahren angeht. Dazu trainiert er ein Encoder-Decoder-Modell mit langem Kurzzeitgedächtnis (LSTM) auf textuellen Daten über Wegbeschreibungen. Als weitere Eingabe dient eine Repräsentation des Weltzustandes, jedoch keine weiteren linguistischen Werkzeuge wie Zerteiler oder domänenspezifische Lexika. Trotz dass die betrachtete Navigationsumgebung mit Attributen wie Farben und einer kleine Elemente von Objekten bestückt ist, besteht sie nur aus geradlinigen, verbundenen Korridoren. Deshalb bleibt die mögliche Instruktionsmenge relativ begrenzt. Weiterhin erzielt das Modell bisher nur bei Betrachtung von einzelnen Sätzen eine zufriedenstellende Güte.
Auch der Ansatz von Suhr et al. [SA18] nutzt ein neuronales Lernmodell um Instruktionssequenzen auf Aktionen von Umgebungsmanipulationsbeschreibungen abzubilden. Es wird dazu als Sprachgenerierungsproblem (engl. *language generation*) betrachtet. Als Eingabe in einem Trainingsschritt dient ein Satz der Instruktionssequenz, der Umgebungszustand und die Historie aus den vergangenen Instruktionen und der zuvor vorhergesagten Aktion. In jedem Trainingsschritt gibt das Modell eine einzelne, zugeordnete Aktion aus. Diese wird im darauffolgenden Schritt ausgeführt und dient wiederum als Eingabe für den nächsten Schritt inklusive des neuen beobachteten Umgebungszustandes. Es werden drei unterschiedliche Szenarien betrachtet, die jedoch jeweils nur aus maximal 5 Aktionen sowie einer kleinen Menge aus möglichen Objekten, Positionen und Status zur Interaktion bereitstellen. Trotz diesen Begrenzungen entsteht ein Korpus aus Sprachbefehlen mit einer Vokabulargröße von 500 bis 800 unterschiedlichen Wörtern.
Zuletzt wird in der Arbeit von Landhäußer und Weigelt [LWT17] für die Tranformation textueller Kommandos in ausführbaren Quelltext ebenfalls eine Verknüpfung zwischen Sprache und Zielsystemfunktionalitäten benötigt. Die Hauptkomponente der Arbeit bildet die Domänenontologie des Zielsystems, welche die Programmierschnittstelle mit allen Klassen und Methoden repräsentiert. Nachdem im ersten Schritt die Eingabe linguistisch zerteilt wurde, wird ein unscharfer Abbildungsalgorithmus (engl. *fuzzy matching*) angewandt, um Verknüpfungen zwischen Sprache und Ontologie-Elementen zu finden.

Dieser prüft neben dem direkten Vergleich der Elementnamen mögliche Synonyme gewonnen aus WordNet. Weiterhin fließen auch die Relationen innerhalb der Domänenontologie in die Bestimmung einer Abbildung mit ein. Werden mehrere mögliche Kandidaten für eine Abbildung gefunden, wird mittels einer Bewertungsfunktion bestehend aus mehreren Metriken wie beispielsweise Teilwortübereinstimmung oder Wortlänge der beste Kandidat ausgewählt. Dieses heuristische Vorgehen wird in der verwandten Arbeit PLOW [ACF+07] verfolgt. Obwohl dieser Abbildungsalgorithmus einen eher einfachen Ansatz nutzt, spricht positiv für ihn, dass er ohne kostspielige, händisch erstellte Schablonen oder annotierte Datensätze auskommt. Weiterhin ist das beschriebene Vorgehen der Abbildungssuche unabhängig von der betreffende Domäne einsetzbar, erzielt jedoch eine etwas geringere Güte.

Viele der beschriebenen Ansätze erzielen in ihrer jeweiligen Domäne gute Ergebnisse. Für das Training von überwachten Modellen wird ein annotierter Datensatz benötigt, welcher die Zuordnung von Instruktionssequenzen zu erwarteten Aktionen enthält. Dieser Datensatz müsste bei einem Einsatz der Verfahren in der hiesigen Arbeit händisch erstellt werden. Weiterhin wird für die formale Repräsentationen der Eingabekommandos und deren Abbildung auf Aktionen domänenspezifische Lexika eingesetzt. Trotz der unterschiedlichen Vorgehensweisen werden die Modelle jedoch häufig auf eine spezifische Problemstellung angepasst und auf eingeschränkten Eingabetexten trainiert. Ziel des Projekt PARSE ist es, eine allgemeine Abbildung von Instruktionen auf Aktionssequenzen zu ermöglichen, um diese Verknüpfungen unabhängig von der Domäne und dem betrachteten Zielsystem herstellen zu können.

5 Zieldefinition

Ziel der Masterarbeit ist die Synthese von Methodendefinitionen in natürlichsprachlichen Äußerungen. Um das Verständnis dieser Problemstellung zu fördern und eine mögliche Lösungsstrategie zu erarbeiten, wird diese zunächst allgemein und abstrakt analysiert. Im Anschluss darauf wird die Zielstellung dieser Arbeit konkretisiert, in mehrere Teilaufgaben untergliedert sowie der Nutzen für das Gesamtprojekt klargestellt.

5.1 Analyse der Problemstellung

Ziel dieser Arbeit ist die Entwicklung eines Konzepts zur Synthese von Methodendefinitionen in gesprochener Sprache. Zur Veranschaulichung soll folgendes Szenario dienen:
Der Nutzer befindet sich in der Küche mit einem Haushaltsroboter, der mittels englischen Sprachbefehlen steuerbar ist. In diesem Szenario wird der am KIT entwickelte Roboter ARMAR-III [ARAS06] als Zielsystem betrachtet. Ein solcher Befehl an den Roboter kann dabei in natürlicher Sprache formuliert sein. Die Anweisung im folgenden Beispiel 5.1.1 fordert den Roboter beispielsweise auf, den Orangensaft aus dem Kühlschrank zu bringen. Die Sprachverarbeitung und das Sprachverständnis wird dabei von dem in Kapitel 3 vorgestellten System PARSE umgesetzt.

Beispiel 5.1.1: Befehl

„ get the orange juice from the fridge and bring it to me “

Weiterhin soll es nicht nur möglich sein, einem Roboterzielsystem bekannte Befehle zu erteilen, sondern auch neue Anweisungen beizubringen. Das Zielsystem soll damit um ihm noch unbekannte, auch komplexere, Funktionalitäten erweitert werden können. Diese Programmierung soll weiterhin intuitiv mittels natürlichen Sprachbefehlen, im Folgenden als *Lehrsequenz* definiert, möglich sein. Zur Verdeutlichung der Problemstellung wird wiederum eine exemplarische Lehrsequenz an den Küchenroboter ARMAR-III in Beispiel 5.1.2 betrachtet.

Definition 5.1.1: Lehrsequenz

Die natürlichsprachliche Funktionserweiterung eines Zielsystems wird als *Lehrsequenz* bezeichnet. Sie besteht aus dem verbalisierten Wunsch zur Wissenserweiterung des Systems sowie einer Inhaltsbeschreibung.

Beispiel 5.1.2: Lehrsequenz

„ emptying the dishwasher means you have to open the dishwasher grab every piece of dishware in it and put it into the cupboard then close the dishwasher “

In der obigen Spracheingabe beschreibt der Nutzer die Funktionalität *„Spülmaschine ausräumen“* mit dem Ziel, dem Roboter dieses neue Wissen beizubringen. Spracheingaben wie diese gilt es im Rahmen dieser Arbeit auf darin befindliche Funktionserweiterungen, hier genannt Lehrsequenzen, zu detektieren. Damit soll es möglich sein, zusätzliche, dem Roboter unbekannte, Funktionalitäten in Sprache zu definieren und persistieren.
Für die Lösung dieser Problemstellung muss untersucht werden, wie Menschen das Hinzufügen von neuen Kommandos verbalisieren. Dabei gilt es unter anderem zu unterschieden, welcher Teil der Eingabe die *Kommandodeklaration*, das heißt, die Verbalisierung einer Funktionserweiterung, enthält und welcher Teil die *Kommandobeschreibung* aus Teilaufgaben repräsentiert. Zur Verdeutlichung wird die obige Spracheingabe mit diesen beiden Teilen einer Lehrsequenz in Beispiel 5.1.3 markiert.

Definition 5.1.2: Kommandodeklaration

Als *Kommandodeklaration* wird der Teil einer Lehrsequenz bezeichnet, in welchem der Wunsch zur Funktionserweiterung sowie der Name der neuen Funktionalität geäußert wird.

Definition 5.1.3: Kommandobeschreibung

Als *Kommandobeschreibung* wird der Teil einer Lehrsequenz bezeichnet, in welcher der Inhalt aus den einzelnen Teilaufgaben der neuen Funktionalität beschrieben wird.

Beispiel 5.1.3: Markierte Lehrsequenz

[*emptying the dishwasher means you have to*]$_{\textbf{Dekl.}}$ [*open the dishwasher grab every piece of dishware in it and put it into the cupboard then close the dishwasher*]$_{\textbf{Beschr.}}$

Bezogen auf die exemplarische Spracheingabe, soll der Wortschatz des Zielsystems um das Kommando *„Spülmaschine ausräumen“* erweitert werden. Diese besteht unter anderem aus den Teilaufgaben *„Geschirr entnehmen“* sowie *„Geschirr in den Küchenschrank räumen“*. Die vom Nutzer erwartete Ergänzung der Zielsystemfunktionalitäten könnte in Pseudocode formuliert wie folgt aussehen:

Beispiel 5.1.4: Pseudocode der Lehrsequenz

```
1  # Kommandodeklaration
2  function empty_dishwasher():
3  # Kommandobeschreibung aus Teilaufgaben
4    open(dishwasher)
5    foreach piece_of_dishware in dishwasher:
6        grab(piece_of_dishware, dishwasher)
7        put(piece_of_dishware, cupboard)
8    close(dishwasher)
```

Die Logik des obigen Pseudocodes und die Art deren Integration in das Zielsystem gilt es zu analysieren. Die Kommandodeklaration entspricht in der Programmiersyntax dabei dem

Methodenkopf, wobei die Kommandobeschreibung den Methodenrumpf für das einzuprogrammierende Kommando bildet.

Zusammenfassend lässt sich die Arbeit zur Synthese von Methodendefinitionen in drei Teile gliedern. In einem ersten Schritt muss die Detektion der in der Spracheingabe befindlichen Lehrsequenzen und deren Bestandteile erarbeitet werden. Für den weiteren Teil der Arbeit sind dabei die vom Nutzer genannten Handlungsanweisungen an den Roboter interessant, welche einerseits die neue Funktionalität als auch die einzelnen Teilaufgaben spezifizieren. Diese werden in Definition 5.1.4 als *Aktionsbenennungen* bezeichnet und bestehen jeweils aus dem Namen einer Aktion und zugehörigen Entitäten. In diesem zweiten Schritt der Arbeit muss daraufhin die Zusammengehörigkeit dieser Namen und Entitäten in den Eingabesequenzen erkannt und zu Aktionsbenennungen zusammengesetzt werden. Diese stehen in semantischer Beziehung zum Verb und nehmen nach Jurafsky unter der Bezeichnung *Aktion* (engl. *event*) eine semantische Rolle ein. Er definiert eine Aktion in seinem Buch [JM00] wie folgt: *„[an event] describes [...] semantic relationships between a verb and noun arguments, roles like agent, instrument, or destination.“*

Definition 5.1.4: Aktionsbenennungen

Als *Aktionsbenennung* oder kurz *Aktion* wird die Bezeichnung einer Handlung mittels Namen und Entitäten definiert. Auf syntaktischer Ebene besteht eine Aktionsbenennung aus einem Verb sowie zugehörigen Nomen, die in einer semantischen Beziehung stehen.

Anschließend sollen die gewonnenen Aktionsbenennungen auf konkrete Funktionen und Parameter des Zielsystems abgebildet werden, um eine Einprogrammierung und spätere erneute Ausführung des neuen Kommandos zu gewährleisten. Hierbei findet die eigentliche *Synthese der Methodendefinitionen* statt, welche den dritten Teil der Arbeit bildet. Während die Eingaben im Sprachkontext eine Lehrsequenz darstellen, entsprechen sie in der Programmierumgebung den Methodendefinitionen. Für die Teilaufgabe der Synthese müssen die sprachlichen Elemente einer Lehrsequenz zunächst auf bekannte Programmkonstrukte abgebildet und anschließend zu Methodendefinitionen, bestehend aus Methodenkopf und Methodenrumpf, aufgebaut werden. Die verwendeten Begriffe zur Beschreibung von Programmen werden in den Grundlagen in Abschnitt 2.6 spezifiziert.

Definition 5.1.5: Synthese von Methodendefinitionen

Als *Synthese von Methodendefinitionen* wird hier die Zusammensetzung von Methodenkopf und Methodenrumpf zu einer neuen Funktion definiert.

Die Modellierung des Zielsystems wird innerhalb des Projekts PARSE mit einer Domänen-Ontologie (siehe Grundlagenkapitel, Abschnitt 2.7), die als Programmierschnittstelle (API) fungiert, umgesetzt. Die beschriebene Funktionalitätserweiterung soll dabei als Ergänzung der bestehenden Zielsystem-Ontologie umgesetzt werden. In Abbildung 5.1 wird die Zusammensetzung der Arbeit aus den beiden Teilaufgaben illustriert.

5.2 Zielstellung der Arbeit

Ziel der Arbeit ist es, dem Nutzer das Erweitern eines Zielsystems um neue Funktionalitäten per natürlichsprachlicher Eingabe zu ermöglichen. Dafür soll Werkzeug entwickelt werden, welches für die Rahmenarchitektur PARSE in Kapitel 3 als weiteren Agenten für Sprachverständnis eingesetzt wird. Dieser soll in zukünftigen Nutzereingaben befindliche

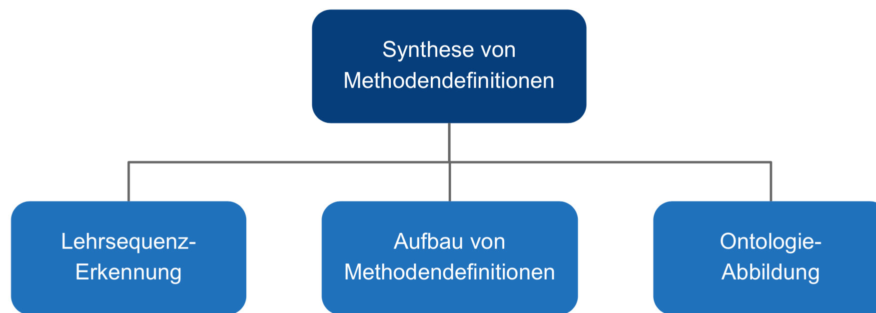


Abbildung 5.1: Teilaufgaben der Arbeit

Lehrsequenzen erkennen, Methodendefinitionen aufbauen und die beschriebenen Teilaufgaben (siehe Definition 5.1.1 bis Definition 5.1.5) auf bestehende Funktionen des Zielsystems abbilden. So soll es anschließend möglich sein, neu hinzugefügtes Wissen unter dem gewählten Definitionsnamen abzurufen. Dazu wird die Arbeit in drei Teile geteilt, welche in Abbildung 5.1 veranschaulicht werden. Betrachtet man die Architektur PARSE in Abbildung 3.1, so wird in dieser Arbeit die Komponente für Sprachverständnis um einen neuen Agenten ergänzt. Dieser modifiziert wiederum die Programmierschnittstelle des Roboters, hier modelliert als Domänen-Ontologie.

Diese Arbeit bildet innerhalb des Projekts PARSE somit einen ersten Schritt, mit natürlicher Sprache zu programmieren. Ein Vorteil einer solchen konversationellen Robotersystem-Schnittstelle gegenüber persönlichen Assistenten wie Google Assistant oder Amazon Alexa, ist, dass das System nicht auf die vorprogrammierten Funktionen der Entwickler beschränkt sein muss. Die Lernfähigkeit ermöglicht es dem Nutzer, neue Kommandos und komplexe Anweisungsfolgen zu definieren und auszuführen. Zweitens kann der Wortschatz des Systems, das heißt, die Menge an verstandenen natürlichsprachlichen Formulierungen, erweitert werden.
Neben der direkten Systemintegration neuer Funktionalitäten durch einen Anwendungsentwickler, gibt es auch alternative Herangehensweisen in der Forschung. Dazu zählen zum Einen das Lernen durch Observation von Expertendemonstrationen und deren Nachahmung (engl. *learning by imitation*), welche oft auch in Kombination mit dem Lernen durch Belohnung (engl. *reinforcement learning*, [KLM96]) eingesetzt werden. Eine andere Möglichkeit ist der Wissensaustausch durch einen vermehrten Dialog im Frage-Antwort-Katalog-Muster zwischen Mensch und Maschine. Für die Einprogrammierung von nutzerspezifischem Wissen gelten diese Varianten jedoch als nicht effektiv [BLK+01]. Einerseits ist es nicht möglich, die individuellen Anfragen unterschiedlicher Nutzer von Entwicklerseite vorab bei der Systementwicklung abzudecken, noch deren Komplexität beim Lernen durch Imitation abzudecken. Das Lernen durch Belohnung ist zudem ein langwieriger Prozess, welcher für die hiesige Arbeit untauglich ist. Überdies generieren diese Alternativen häufig keine Wissensrepräsentationen die der Nutzer anschließend abfragen kann [LBKK02].

Im Vergleich zu bisherigen Arbeiten soll die Synthese von Methodendefinitionen mit unbeschränkten, natürlichen Sprachkommandos ermöglicht werden. Zusätzlich dazu soll ein Ansatz entwickelt werden, welcher einerseits die Domänenunabhängigkeit zum Zielsystem gewährleistet und andererseits maschinelles Lernen für die Problemlösung einsetzt. Weiterhin muss das Werkzeug modular als Agent der Architektur PARSE entwickelt werden, welches bestehende Informationen integriert sowie gewonnenes Wissen anderen Agenten verfügbar macht.

6 Vorstudie

Die Problemstellung dieser Arbeit umfasst die Detektion von Lehrsequenzen (siehe Definition 5.1.1) in gesprochenen Anweisungen sowie die Identifikation zugehöriger Einzelbestandteile zur Synthese von Methodendefinitionen. Diese Fragestellung wurde in der Gesamtheit der hiesigen Anforderungen in der Forschung bisher nicht näher betrachtet. Dazu gehören die Betrachtung von natürlichsprachlichen Lehrsequenzen in gesprochener Sprache, die Möglichkeit zum Dialog sowie die Domäne der Roboterprogrammierung,

Da die Charakteristiken derartiger Nutzereingaben deshalb nicht einschätzbar sind, wird im Rahmen der Arbeit eine Vorstudie durchgeführt. Das Ziel dieser Vorstudie ist es, im Zuge einer syntaktischen und semantischen Analyse Erkenntnisse über die Struktur von möglichen Nutzereingaben zu erhalten. Es werden natürlichsprachliche Formulierungen betrachtet, die Lehrsequenzen beinhalten. Dabei handelt es sich um Äußerungen, in welcher Menschen einem Gegenüber etwas erklären und damit neues Wissen vermitteln wollen. Diese Daten sollen daraufhin auf mögliche linguistische Muster untersucht und für den Entwurf eines Lösungsansatz zur Synthese von Methodendefinitionen genutzt werden.

6.1 Mögliche Datenquellen

Als mögliche Datenquelle wird zunächst das bestehende Korpus [Gü15] des zugehörigen Projekt PARSE in Betracht gezogen. Die vorhandenen Aufnahmen bestehen aus natürlichen Spracheingaben in der Zieldomäne des Roboters Armar-III [ARAS06]. Jedoch enthalten sie keine Szenarien, in welchen der Roboter explizit um weitere Funktionalitäten erweitert wird.

Als weitere Datenquellen werden öffentliche Quellen und Korpora möglicher verwandter Themen betrachtet. Eine Quelle sind Rezepte in Kochbüchern. Die Texte enthalten eine Definition, den Rezeptnamen, sowie Anweisungsbeschreibungen, welche das eigentliche Kochrezept beinhalten. Rezepte bestehen häufig aus erklärenden Teilaufgaben für den Kochenden. Diese entsprechen im Bezug auf eine Lehrsequenz den dortigen Teilaufgaben in der Kommandobeschreibung (Definition 5.1.3). Jedoch gibt es auch Aspekte, welche gegen natürlichsprachlichen Formulierung einer Lehrsequenz sprechen. Einerseits sind Rezepte von Kochbüchern häufig stichpunktartig verfasst. Diese müssten ausgeschlossen werden. Weiterhin ist der Rezeptname in den meisten Fällen als einfache Überschrift repräsentiert. Die für diese Arbeit wichtigen Formulierungen, welche der Kommandodeklaration (siehe Definition 5.1.2) entsprechen, sind somit nicht enthalten. Deshalb werden Kochrezepte als Quelle für die Detektion von Lehrsequenzen ausgeschlossen.

Eine weitere mögliche Quelle sind Blogartikel zu sogenannten ”Do-it-yourself und ”How-To-Themen. In diesen werden ebenfalls ausführliche textuelle Erklärungen zu verschiedenen, häufig alltäglichen, Problemstellungen präsentiert. Ein öffentliches Korpus solcher Artikel gibt es bisher jedoch nicht. Weiterhin werden derartige Beiträge meist mit Bildern, Skizzen oder Videos angereichert, um die Inhalte zu verdeutlichen. Diese können mit Sprachverarbeitungstechniken jedoch nicht verarbeitet werden. Daher bietet die Verwendung von Blogartikeln als Quelle für diese Arbeit ebenfalls keine Option.

Eine weitere Quelle bieten Texte in Zeitungsartikeln. Diese werden in der Domäne des *Natural Language Processing* häufig als Datengrundlage verwendet. Besonders das mit Wortarten annotierte *Wall Street Journal* des Penn Treebank-Korpus [MMS93] wird oft als Maßstab zum Vergleich der entwickelten Ansätze verschiedener wissenschaftlicher Arbeiten genutzt. Gleiches gilt für das Reuters-Korpus [1] bestehend aus Nachrichtentexten, für welches ein Goldstandard für die Eigennamenerkennung und Dokumentenklassifikation existiert. Diese Korpora werden zum Beispiel in den Arbeiten von Manning [Man11], Collins [Col02] und Huang [SH16] für die Entwicklung von Ansätzen für die Wortartmarkierung sowie von Ratinov [RR09] und Devlin [DCLT18b] zur Eigennamenerkennung eingesetzt. Die Texte dieser Korpora nehmen in ihrer Art jedoch keine lehrende beziehungsweise erklärende Funktion für den Leser ein, sondern bieten eine reine sachliche Schilderung von Ereignissen. Daher können auch diese Textsammlungen nicht für die hiesige Problematik genutzt werden.

Da sich in öffentlichen Quellen keine geeigneten Daten zu natürlichsprachlichen Lehrsequenzen finden, müssen eigene Daten erhoben werden.

6.2 Konzeption einer Nutzerstudie

Öffentliche Quellen stellen bisher keinen geeigneten Datensatz zur Strukturanalyse von Lehrsequenzen zur Verfügung. Daher sollen im Rahmen einer Nutzerstudie eigene Daten zu dieser Problemstellung erhoben werden. Hierfür wird eine Gruppe von Probanden aufgefordert, sich eigene Lehrsequenzen zu einer bestimmten Szenerie auszudenken. Diese vom Nutzer eingereichten Eingaben werden schließlich als Datengrundlage verwendet.
Für den Aufbau der Nutzerstudie gilt es vorab verschiedene Randbedingungen zu definieren und gegeneinander abzuwägen, bevor konkrete Aufgaben, hier *Szenarien* genannt, entworfen werden können.

Die **Sprache** der Nutzerstudie wird auf Englisch festgelegt, da das Projekt PARSE und dessen Werkzeuge auf englische Eingaben spezialisiert sind.

Als Weiteres können für die Studie gesprochene oder geschriebene Nutzereingaben als **Eingabeformat** verwendet werden. Für geschriebene Eingaben spricht die Einfachheit der Aufnahme für den Nutzer: die geforderten Anweisungen können in textueller Form abgetippt werden. Stattdessen muss der Nutzer bei auditiven Eingabeformat die Anweisungen diktieren. Dazu sind ein Mikrofon und ein Aufnahmeprogramm für den Rechner nötig. Ein nächster wichtiger Punkt ist die bessere Verarbeitbarkeit textueller Abgaben. So können grobe Fehler in den Daten mit einer kurzen Durchsicht schnell erkannt werden. Weiterhin kann eine Studie mit textuellen Eingaben in Form einer Umfrage mit Texteingabefeldern aufgebaut werden. Zuletzt gibt es öffentliche Bibliotheken für die Beseitigung von einfachen Tippfehlern. Nachteile bei geschriebenen Eingaben sind, dass sie eher komplexer und länger sein können als gesprochene Anweisungen. Es besteht die Gefahr, dass durch die Abgabe in Textform die gewünschten Nutzereingaben eher deskriptiv anstatt einer Befehlsform ausfallen. Zudem ist gilt es mögliche Unterschiede zwischen geschriebener und gesprochener Sprache zu beachten und deren Vergleichbarkeit

[1] Quelle: `https://trec.nist.gov/data/reuters/reuters.html`, zuletzt besucht am 17.06.19

einzubeziehen, da die Agenten von PARSE bisher spezifisch für gesprochene Sprache entwickelt wurden.
Im Gegensatz dazu können die Daten im Rahmen der Nutzerstudie auch in gesprochener Form gesammelt werden. Der Lösungsansatz dieser Arbeit soll auf gesprochene Anweisungen ausgerichtet werden. Getätigte Eingaben können in den bestehenden PARSE Korpus aufgenommen werden. Für gesprochene Eingaben ist ein Mikrofon von guter Qualität nötig. Die Aufnahme der Nutzereingabe benötigt möglicherweise mehrere Versuche und ist daher aufwändiger. Außerdem müssen Audioaufnahmen zusätzlich transkribiert werden. Erst dann ist eine Durchsicht der Daten möglich. Mit einem automatischen Spracherkenner kann die zeitaufwändige manuelle Transkription der Sprachdaten zwar umgangen werden, jedoch sind die Ausgaben oft fehlerbehaftet. Daher können auch die Ausgaben eines Spracherkenners ohne zusätzliche manuelle Korrektur nicht als Alternative zu den Transkriptionen von gesprochenen Eingaben angesehen werden.

Die **Teilnahmebegrenzung** legt fest, dass ein Proband nur ein Mal an der Nutzerstudie teilnehmen darf. So wird verhindert, dass durch den Reihenfolge-Effekt (engl. *sequencing effect*)[Chr07] Lernerfolge erzielt werden, welche die Studienergebnisse beeinflussen. Zusätzlich wird von einem Probanden gefordert, die Aufgaben in einem Zug zu bearbeiten, um Historien-Effekt (engl. *history effect*)[Chr07] wie äußere Einflüsse, mögliche Recherchen sowie mehrfache Eingabeversuche zu vermeiden. Dies wird durch eine begrenzte Bearbeitungsdauer pro Teilnahme erreicht.

Eine letzte Randbedingung der Nutzerstudie bildet die Festlegung der **Eingabeprozedur**. Die Nutzereingaben können wie in den vergangenen Arbeiten des Projekt PARSE in einer lokalen Studie mit den Probanden vor Ort gesammelt werden. Hierbei besteht die Möglichkeit, eventuell aufkommende Fragen der Probanden bei der Nutzerstudie zu beantworten. Andererseits besteht die Gefahr, die Messergebnisse durch das Einwirken des Experimentators (engl. *experimenter effect*)[Chr07] zu beeinflussen. Eine Orientierung der Szenarienformulierung an den bereits erfolgreich durchgeführten Nutzerstudien von PARSE wäre ebenfalls möglich. Jedoch ist davon auszugehen, dass die Englischkenntnisse deutscher Teilnehmer nicht an das Niveau eines englischen Muttersprachlers herankommen. Vergangene Nutzerstudien des PARSE-Korpus haben gezeigt, dass derartige Eingaben in Aussprache und Grammatik häufig fehlerbehaftet sind. Da die Eingaben persönlich und vor Ort erfolgen müssen, ist zudem eine geringere Ausbeute an Teilnehmern zu erwarten. Eine digitale Nutzerstudie muss dagegen nicht terminlich organisiert und kann deshalb einer größeren Menge gleichzeitig zur Verfügung gestellt werden.
Die digitale Eingabeprozedur bietet weiterhin den Vorteil, dass der Teilnehmer die Nutzerstudie bequem von zu Hause durchführen kann. Durch den vereinfachten Zugriff kann eine weitaus größere Masse an Teilnehmern angesprochen und somit eine größere Menge an Daten gewonnen werden. Die Größe der Datengrundlage spielt bei der Analyse einen ausschlaggebende Rolle. Da die Nutzerstudie zudem nicht an eine lokale Eingabeprozedur gebunden ist und über das Internet verbreitet werden kann, besteht die Möglichkeit auch englische Muttersprachler als Teilnehmer für die Nutzerstudie zu gewinnen. Eine Hürde der digitalen Eingabeprozedur spielt vor allem die Verständlichkeit der Aufgabenstellungen innerhalb der Nutzerstudie für den Probanden. Diese müssen sehr präzise formuliert sein, um Missverständnisse zu vermeiden und die gewünschte Datengrundlage zu erzielen. Die dafür konzipierten Szenarientexte müssen daher in ihrem Aufbau für eine digitale Umfrage und für fremde Teilnehmer angepasst werden.

Anhand der obigen diskutierten Randbedingungen der Nutzerstudie wird das **textuelle Eingabeformat** in der **Sprache englisch** gewählt. Da die Datengrundlage eine erste Einsicht ermöglichen soll, ist die Verwendung von geschriebenen statt gesprochenen Eingaben vertretbar. Dies gilt auch für eine Verwendung dieser Daten als Eingabe für

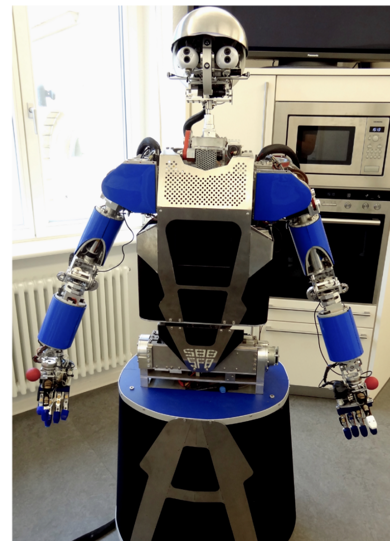
einen späteren Detektionsalgorithmus. Für die abschließende Evaluation (Kapitel 10) sollten jedoch gesprochene Eingaben verwendet werden. Die Datengrundlage soll im Rahmen einer **digitalen Umfrage** erhoben werden. Dafür wird der Dienst ***Google Forms***[2] von Google genutzt, welcher das Erfassen und Verwalten von Informationen erleichtert. Zudem ist es mit Google Forms möglich, eine **Teilnahmebegrenzung** für die Studie auf eine bestimmte Bearbeitungsdauer und **einmaliger Teilnahme pro Proband** festzulegen. Neben der Option, die Nutzerstudie lokal durchzuführen, besteht die Möglichkeit, das Internet als Studienplattform für mögliche Teilnehmer zu nutzen. Hierfür bietet sich der akademische **Crowdsourcing-Marktplatz *prolific.ac*** Abschnitt 2.5 an, welcher sich durch eine große Nutzerbasis sowie eine Spezialisierung auf Forschungsstudien auszeichnet. Deshalb wird eine **nicht-lokale Eingabeprozedur** gewählt. Ein weiterer Vorteil ist dabei, dass ein Großteil der Nutzer aus englischsprachlichen Ländern stammt und somit ein hoher Anteil an **englischen Muttersprachlern** zu erwarten ist.

6.3 Szenarien

Die Nutzerstudie wird als Umfrage gestaltet. Mittels kleiner Aufgaben soll der Nutzer dazu verleitet werden, sich eigene Lehrsequenzen zu einer bestimmten Szenerie auszudenken und diese textuell abzutippen. Dazu werden die Probanden zu Beginn kurz in das Themengebiet und das verfolgte Ziel der Nutzerstudie eingeführt. „*Teach a robot new skills with natural language instructions*“ lautet der veröffentlichte Titel der Studie. Bilder von dem exemplarisch betrachteten Roboter Armar-III [ARAS06] sowie dessen Interaktion mit der dargestellten Küchenumgebung sollen die Szenerie verdeutlichen. Einen Auszug der Nutzerstudie zur Einführung in die Studienumgebung wird in Beispiel 6.3.1 präsentiert.

Beispiel 6.3.1: Einführung in das Themengebiet der Nutzerstudie

Target system: Humanoid household robot (called Armar)
Setting: Kitchen
Goal: Teach the robot new skills
Your task: 4 natural language instructions for the robot to learn a specific skill



Die Studie enthält vier Aufgaben für den Nutzer, hier als *Szenarien* bezeichnet. In diesen Szenarien wird der Proband dazu aufgefordert, einem Robotersystem bestimmte neue

[2]Quelle: `https://docs.google.com/forms`, zuletzt besucht am 17.06.19

Funktionen beizubringen. Die Szenarien sind dabei so gestaltet, dass sie das Verwenden von Lehrsequenzen in den Formulierungen der Nutzereingaben hervorrufen sollen. Sie versetzen den Probanden in die Situation eines Lehrers, der dem Roboter als seinen Lehrling mit Hilfe von textuellen Eingaben erklären soll.
Vom Nutzer wird eine geschriebene Eingabe gefordert, die eine natürlichsprachliche Anweisungen für ein Szenario, beispielsweise dem Starten einer Spülmaschine, enthält. Zusätzlich soll diese Eingabe in Anweisungsform verfasst sein, wie als würde der Proband direkt zu dem Roboter-Zielsystem sprechen. Um für diese Vorstudie geeignete Daten zu produzieren, muss der Proband weiterhin seine Aussage als Lehrsequenz formulieren, die dem Roboter das Szenario unter Verwendung des Szenariennamens, einer expliziten Äußerung für eine Funktionserweiterung als auch die genannten Teilschritte erklärt.
Die hier geforderten Eigenschaften der Nutzereingabe sind komplex. Dies zeigt sich auch im Vergleich mit den Aufgaben anderer Studien, welche auf der Plattform *prolific.ac* angeboten werden. Deshalb wird zur Veranschaulichung der geforderten Aufgabenstellung ein Beispielszenario einschließlich exemplarischer Nutzereingabe präsentiert. Dieses Vorgehen wird auch in der verwandten Arbeit von Campagna [CRX+17] verfolgt. Statt vielen Paraphrasen des präsentierten Beispiels wurden damit vermehrt natürliche und freie Formulierungen des Szenarios in den Nutzereingaben gewonnen. Das Beispielszenario ist als Auszug der Nutzerstudie in Beispiel 6.3.3 zu sehen. In der Studie von Lauria [BLK+01] wird statt eines Beispielszenarios nur das gewünschte Ziel für die geforderten Wegbeschreibungen angegeben. Die Probanden waren in ihren Anweisungen an den dort betrachteten mobilen Roboter somit völlig frei. Jedoch beschränkt sich diese Freiheit anderweitig durch den geringen Funktionsumfang des Roboters, welcher in seiner Fortbewegung durch die modellierten Straßen der Miniaturstadt zusätzlich begrenzt ist. Ein anderer Ansatz wäre die Verwendung eines Videos zur Verdeutlichung der Problemstellung wie in den Arbeiten von Pane [PRM01] oder Tellex [TKD+11]. Dieses Video könnte die Interaktion durch Sprachbefehle eines Menschen mit dem betrachteten Zielsystem Armar-III präsentieren. Dies würde zwar ein leichteres Hineinversetzen des Probanden in die Zieldomäne der Roboterumgebung ermöglichen, jedoch würden die Eigenschaften der im Video verbalisierten Anweisungen durch die bewegten Bilder vermutlich in den Hintergrund geraten. Eine Auflistung der Sprachbefehle wäre daher zusätzlich nötig. Zudem enthalten bestehende, öffentliche Videos von Armar-III keine vom Mensch verbalisierten Lehrsequenzen, sondern ausschließlich Befehle zur sofortigen Ausführung. Aus diesen Gründen wird die Verwendung eines Videos zur Verdeutlichung der Aufgabenstellungen nicht umgesetzt. Stattdessen werden pro Szenario zu den Aufgabentexten entsprechende Bilder der Zielumgebung mitgeliefert.

Beispiel 6.3.2: Akzeptanzkriterien an die Eingaben

- Imagine the robot being a little child you want to teach something
- Do not forget to:
  a. **explicitly state** that you want to teach something (marked in red above)
  b. give the skill some name (marked in blue above)
  c. give a short description of the intermediate steps (marked in green above)
- Use **different wordings** as in the example and for each skill (if possible)

Beispiel 6.3.3: Beispielszenario mit exemplarischer Eingabe

New skill: Start the dishwasher
Intermediate steps:

- The dishwasher can be started by pressing the red button
- It is located in the kitchen
- It needs to be closed before starting it

Some exemplary instructions to teach this new skill:

- "Hi Armar, starting the dishwasher means you have to go to the dishwasher, close it and press the red button."
- "You have to close the dishwasher and press its red button for turning it on. That's how you start the dishwasher."

Bevor ein Studienteilnehmer seine Antworten zu den dargestellten Szenarien in die Texteingabefelder eintippt, wird er zusätzlich zum Beispiel nochmals darauf hingewiesen, welche Akzeptanzkriterien die Eingabe erfüllen muss. Das Beispiel 6.3.2 zeigt den entsprechenden Auszug der Nutzerstudie. Zu diesen Akzeptanzkriterien zählen zum Einen die Forderung, möglichst unterschiedliche Wortlaute in den vier Szenarien zu verwenden, sowie drei Anforderungen für die verbalisierten Lehrsequenzen. Letztere sind anschließend nochmals in deutsch gelistet. Neben der Forderung nach einem natürlichsprachlichen Inhalt der Nutzerabgabe, kann diese jedoch frei formuliert werden. Auch die Länge der Eingabesequenzen wird im Gegensatz zu anderen Studien [Mil77], [Bie83] nicht beschränkt.

Für die Erfüllung der Anforderungen muss eine Nutzereingabe drei Elemente aufweisen:

- einen Ausdruck, dass neues Wissen gelehrt wird
- einen Namen für das neue Kommando
- eine Beschreibung der Teilschritte des Kommandos

Um eine bestmögliche Verständlichkeit zu gewährleisten, werden in der gesamten Umfrage Fachbegriffe vermieden sowie einfache Formulierungen für die Szenarientexten verwendet. Zur besseren Übersichtlichkeit sind die Texte kurz, meist sogar stichpunktartig, verfasst. Auf diese Weise soll das präsentierte Vokabular gering gehalten werden, sodass der Proband nicht dazu beeinflusst wird, die Satzbestandteile der Szenarientexte geradewegs in seine Antwort zu übertragen [Mil77]. Farbliche Markierungen und Bilder sollen die

Aufgabenstellungen verdeutlichen. Dieses stichpunktartige Format für Studientexte sowie die Beigabe von Bildern wird auch in der Arbeit *Studying the language and structure in non-programmers' solutions to programming problems* von Pane et al. [PRM01] angewendet. Beeinflussungen der Studienergebnisse durch die vorgelegten Studiendokumente sollen somit verhindert werden.
Als Abschluss der Studie wird ein Teilnehmer der Umfrage befragt, ob dessen Muttersprache Englisch ist und ob Programmierkenntnisse vorhanden sind. So können später Aussagen über das Niveau der Sprach- beziehungsweise Programmierkenntnisse auf die Güte des Eingaben oder des Ergebnisses getroffen werden.

6.4 Datengrundlage

Mittels der Plattform *prolific.ac* sollen möglichst viele Aufnahmen akquiriert werden. In einem ersten Testlauf mit 10 Teilnehmern sind zunächst 10 Aufnahmen zu je 3 Szenarien, das heißt 30 Eingaben, entstanden. Anschließend wurde die Nutzerstudie mit weiteren 860 Teilnehmern durchgeführt. Hierbei wurde die Nutzerstudie um ein zusätzliches Szenario zu 4 Szenarien erweitert. So konnten in der zweiten Studie bei 860 Aufnahmen zu je 4 Szenarien insgesamt 3440 Eingaben gesammelt werden. Dies entspricht im Vergleich zu den verwandten Nutzerstudien zum Thema *Natural Language Programming* von Miller [Mil77], Biermann [Bie83] oder Pane et al. [PRM01] einer deutlich größeren Nutzerbasis, da diese eine Teilnehmeranzahl von 31 nicht überschritten.
Da die Aufnahmen im Gegensatz zu einer lokalen Studie nicht zur Laufzeit validiert und ein Korrekturwunsch seitens des Studienerstellers geäußert werden kann, muss dies innerhalb der digitalen Studie im Nachhinein geschehen. Eine Durchsicht der erhobenen Daten zeigt, dass einige Aufnahmen nicht valide sind. Dazu gehören Aufnahmen, die gezielt durch den Studienteilnehmer falsch getätigt wurden. Weiterhin gibt es auch einen geringen Satz an Teilnehmern, welche die Aufgabenstellung missverstanden und dadurch nicht verwertbare Daten eingereicht haben. Deshalb wurden nach einer ersten Durchsicht die *Einreichungen* von 75 Teilnehmern, das entspricht 11.6% der Eingaben, aus der Nutzerstudie aussortiert.

Definition 6.4.1: Einreichung

Als *Einreichung* werden alle Angaben eines Probanden innerhalb der Nutzerstudie bezeichnet. Diese kann entweder als Gesamtheit als valide Datenmenge akzeptiert oder von der Studie ausgeschlossen werden.

Dieses Aussortieren der Einreichungen entspricht einem Eingriff des Experimentators (engl *experimenter effect*) [Chr07] in die Studienergebnisse. Er lässt sich jedoch damit rechtfertigen, dass die Studie nicht der Untersuchung der Leistungsbereitschaft des Nutzers oder dessen Verständnis von der Aufgabenstellung dient. Stattdessen soll eine Datengrundlage möglicher Lehrsequenz-Formulierungen erhoben werden. Dies wird durch die Szenarientexte nachweislich erreicht. Da die erhobenen Daten mit den gewünschten Studienergebnissen übereinstimmen, ist die interne Validität (engl *internal validity*) [Chr07], der Studie gewährleistet. Im Falle der verwendeten Plattform *prolific.ac* werden alle Einreichungen dieser Teilnehmer abgelehnt und somit auch nicht entlohnt. Einen Auszug von Nutzereingaben dieser akzeptierten und abgelehnten Daten befindet sich im Anhang in Abschnitt A.2.2. Nach der Bereinigung des Datensatzes befinden sich noch insgesamt 3168 Eingaben in der Datengrundlage der Vorstudie.
Eine Übersicht über die gesammelten Daten ist in Tabelle 6.1 dargestellt.

Tabelle 6.1: Anzahl erhobener Eingabesequenzen der Vorstudie

	Testlauf	Nutzerstudie	Gesamt
Teilnehmer (Einreichungen)	10	860	870
Eingaben	30	3440	3470
Valide Eingaben	30	3138	3168

Tabelle 6.2: Anteil der Muttersprache Englisch unter den Einreichungen

	Akzeptiert	Abgelehnt	Gesamt
ENG	60%	51%	60%
N_ENG	40%	49%	40%

6.5 Auswertung

Nach der Filterung der Daten auf nicht valide Eingaben, werden sie anschließend auf inhaltlicher Ebene näher betrachtet. Dazu werden die Charakteristiken der Probanden sowie Besonderheiten in den eingereichten Nutzereingaben ausgewertet.

6.5.1 Nutzercharakteristiken

In diesem Abschnitt werden jene Daten der Nutzerstudie betrachtet, welche neben den primär geforderten Eingabesequenzen erfasst werden. Dazu gehören unter anderem die Angaben der Probanden bezüglich ihrer Muttersprache und Programmierkenntnissen, als auch weitere Charakteristiken, wie die benötigte Dauer für die Studie und die abgegebene Anzahl an Wörtern pro Einreichung (siehe Definition 6.4.1).

Als Erstes werden die Angaben der Probanden bezüglich ihrer **Muttersprache** betrachtet, wobei die Studie lediglich erfasst, ob ein Proband Englisch als Erstsprache (ENG) spricht, oder nicht (N_ENG). Aus Tabelle 6.2 geht in der Spalte *Gesamt* hervor, dass nur 60% aller Studienteilnehmer (591 versus 351 Probanden) Englisch als ihre Muttersprache angegeben haben. Dies entspricht in etwa den demographischen Statistiken[3] von *prolific.ac*. Demnach stammt ein Viertel der Menge möglicher Studienteilnehmer aus Großbritannien, weitere 30% kommen aus den USA. Die restlichen Teilnehmer gehören verschiedenen, vorwiegend nicht primär englischsprachigen Herkunftsländern an. Je nach Zeitverschiebung zu den unterschiedlichen Herkunftsländern spielt daher auch der Studienzeitpunkt eine Rolle für die Teilnahme möglicher Probanden. Da die hiesige Nutzerstudie über einen verteilten Zeitraum von etwa 5 Stunden durchgeführt wurde, ist somit eine vollständige Abdeckung eines Tages über die verschiedenen Zeitzonen nicht gewährleistet.
Um einen möglichen Einfluss der Erstsprache auf die eingereichten Studienabgaben der Probanden abzuschätzen, wird die Verteilung von Muttersprachlern über der Menge der akzeptierten sowie abgelehnten Einreichungen betrachtet. Tabelle 6.2 zeigt, dass die Menge der Englisch-Muttersprachler (ENG) bei akzeptierten Einreichungen um fast zehn Prozent höher ist als bei Englisch-Muttersprachlern (N_ENG). Dies könnte den positiven Einfluss der durch die Muttersprache verankerten Englischkenntnisse auf das Verständnis der geforderten Studienaufgaben zeigen. Die entsprechende Verteilung bei abgelehnten Einreichungen ist bei beiden Gruppen nahezu gleich hoch.

Als Nächstes werden die Angaben der Probanden zu ihren Programmierkenntnissen betrachtet. Dazu wird in der Studie erfasst, ob ein Proband Programmierkenntnisse besitzt (PROG), oder nicht (N_PROG). Die Spalte *Gesamt* der Tabelle 6.3 veranschaulicht, dass nur 30% aller Studienteilnehmer (261 versus 609 Probanden) Programmierkenntnisse besitzen. Dies war zu erwarten, da die Studienplattform *prolific.ac* auf das Crowdsourcing

[3] Quelle: `https://prolific.ac/demographics/`, zuletzt besucht am 17.06.19

Tabelle 6.3: Anteil der Programmierkenntnisse unter den Einreichungen

	Akzeptiert	Abgelehnt	Gesamt
PROG	28%	49%	30%
N_PROG	72%	51%	70%

Tabelle 6.4: Dauer der Teilnahme (in Minuten)

Eingabe	akzeptiert	abgelehnt
Mittelwert	9	17
Minimum	5	1
Maximum	25	25
Standardabweichung	6	6

(siehe Abschnitt 2.5) von allgemeinen Kleinstaufgaben ausgerichtet und nicht auf Programmieraufgaben spezialisiert ist. Die Werte der Tabelle zeigen zudem, dass diese 30% hauptsächlich in der Menge der abgelehnten Nutzereinreichungen vorzufinden sind. Eine Interpretation dieser Verteilung ist schwierig.
Einen strukturellen Unterschied der erfassten Nutzereingaben zwischen Probanden mit und ohne Programmierkenntnissen lässt sich ebenfalls schwer erkennen. Dieser Zusammenhang wird auch in vergangenen Arbeiten des Projekts PARSE bisher nicht näher untersucht. Es ist jedoch zu vermuten, dass sich kein bedeutender Unterschied dieser Nutzergruppen in den Eingabesequenzen abzeichnet. Das kann damit begründet werden, dass die in der Studie geforderten Eingaben in rein natürlicher Sprache erfolgen sollen, was für den Menschen ohnehin intuitiv ist. Eine weitere mögliche Ursache könnte sein, dass den Probanden derartige natürlichsprachliche Befehle an ein intelligentes Zielsystem durch Produkte wie Google Go oder Amazon Alexa bereits geläufig sein könnten.

Als weitere Charakteristik der Studienteilnehmer wird die benötigte Studiendauer für die Bearbeitung der Fragestellungen betrachtet. In Tabelle 6.4 lässt sich ablesen, dass der Mittelwert der Studiendauer bei akzeptierten Einreichungen bei neun Minuten liegt. Der Mittelwert bei abgelehnten Einreichungen ist deutlich höher. Einige erhaltene Einreichungen weisen eine sehr hohe Dauer auf. Dies sind vermutlich Teilnehmer, welche die Studie pausierten, abbrachen oder Verständnisprobleme hatten. Während das Minimum der Teilnahmedauer akzeptierter Einreichungen bei fünf Minuten liegt, beträgt das Minimum der abgelehnten Abgaben bei einer Minute. Eine Reihe von Nutzern möchten die Studie schnellstmöglich abschließen, um die mögliche Belohnung zu erhalten. Das Erarbeiten von validen Studienergebnissen benötigt aber mehr Zeit, welches sich in den Zahlen der akzeptierten Einreichungen widerspiegelt. Das Maximum der Studienteilnahmedauer wurde auf 25 Minuten festgelegt, nachdem eine Abgabe nicht mehr möglich ist.

Zuletzt wird die **Anzahl der Wörter** pro Einreichung betrachtet. Die Werte in Tabelle 6.5 sind dabei für alle vier Szenarien summiert. So werden bei akzeptierten Einreichungen durchschnittlich pro Szenario 47 Worte, bei abgelehnten Einreichungen im Mittel nur etwa 7 Worte pro Szenario abgeben. Die oben erwähnte Nutzergruppe mit mangelhaften Kurzantworten lässt sich in Tabelle 6.5 am Minima der abgelehnten Einreichungen von 4, das heißt mit einem Wort pro Szenario, erkennen. Das Maximum für akzeptierte sowie abgelehnte Einreichungen ist sehr hoch. Diese Werte sind jedoch als Ausreißer zu betrachten, bei welchen ein Proband besonders ausführlich in seiner Beschreibung der neuen Zielsystemfunktionalität war. Somit kann man behaupten, dass die Länge der Eingaben eine Einflussgröße ist und eine höhere Wortanzahl eher auf eine valide Einreichung schließen lässt. Dieser Zusammenhang findet sich auch bei den berechneten Minima und der Standardabweichung wieder.

Tabelle 6.5: Länge der Einreichungen (in Wörtern)

Eingabe	akzeptiert	abgelehnt
Mittelwert	140	30
Mininum	59	4
Maximum	946	761
Standardabweichung	66	29

Aus den letzten beiden Tabellen geht zusammenfassend hervor, dass akzeptierte Einreichungen allgemein eine längere Teilnahmedauer und eine höhere Anzahl an Wörtern aufweisen als abgelehnten Abgaben.

6.5.2 Linguistische Analyse

In diesem Abschnitt werden die in der Nutzerstudie gewonnenen Eingabesequenzen auf ihren Inhalt untersucht. Dabei soll die Frage geklärt werden, wie Menschen anderen Subjekten neues Wissen beibringen, das heißt, die Lehrsequenzen zu verbalisieren. Weiterhin sollen für die Teilaufgabe der Erkennung von Lehrsequenzen (siehe Abbildung 5.1) mögliche linguistische Merkmale in den einzelnen Einreichungen (siehe Definition 6.4.1) gefunden und analysiert werden. Dazu werden im Folgenden in Beispiel 6.5.1 eine Auswahl an akzeptierten Eingaben der Nutzerstudie diskutiert.

Beispiel 6.5.1: Eingaben der Nutzerstudie

E1: *„ Hi, Armar. We are going to learn how to make coffee. First, we must put a coffee cup under the dispenser. Then start the coffee machine by pressing the red button on the machine. “*

E2: *„ To greet somebody you have to look them in the eyes, waving your hand and saying hello. “*

E3: *„ If you want to greet someone make sure you look at the person you want to greet, wave your hand and say Hello. “*

E4: *„ Amar, coffee is a becerage that people drink, like water. In order to make a coffee, you have to locate the cups next to the machne, put one cup under the dispenser and lastly press the red Burton on the coffee machine. “*

E5: *„ Take the averages from the fridge, take the glasses from the kitchen counter, pour beverage to the glass and hand it over the user “*

E6: *„ Hi, Armar! First find a cup next to the machine. Then place the cup under the dispenser. Last, you have to push the red button. “*

E7: *„ Armar, when you greet him you look him in the eyes, wave your robot hand and say hello. “*

Die Eingaben E1 bis E7 bilden eine Stichprobe der gewonnenen Datengrundlage der Vorstudie. Sie zeigen sechs von verschiedenen Nutzern eingereichte Eingabesequenzen zu unterschiedlichen Szenarien. Es fällt auf, dass die Eingaben trotz ihr textuellen Ausprägung natürlichsprachlich und für die Zieldomäne zutreffend als mögliche gesprochene Kommandos des Nutzers an einen Roboter formuliert sind. Dies wurde durch mehrfache Hinweise in den Aufgabenstellungen der Vorstudie von den Probanden gefordert. Die Ausprägung der gewonnenen Eingabesequenzen bestätigen die in Abschnitt 6.2 geführte Argumentation, dass die textuellen Eingaben als Datengrundlage für einen Lösungsentwurf vertretbar sind.

In manchen Fällen (E1, E4, E6, E7) wird der Roboter zu Beginn der Eingabesequenz explizit mit Namen angesprochen, die restlichen Eingaben beginnen wiederum direkt mit der geforderten Äußerung. Die Erzählperspektive der einzelnen Eingaben variiert. Manche der Probanden richten die Instruktionen an den Roboter aus der Perspektive eines Lehrers oder Programmierers. So sind beispielsweise die Eingaben E2 und E3 sachlich und von der Rolle eines außenstehenden Betrachters der Szenerie formuliert. Im Gegensatz dazu steht die Nutzereingabe E1, in welcher der Proband sich selbst in die Situation miteinbezieht: *„We are going to learn“* oder *„I want you to ..“*. Diese Probanden versetzen sich in die Rolle des Endnutzers. Dieses Merkmal wird auch Pane et al. in seiner Arbeit zu *Studying the language and structure in non-programmiers' solutions to programming problems* [PRM01] identifiziert.
Zuletzt lässt sich erkennen, dass die Nutzereingaben in Präsens und überwiegend in der Imperativform formuliert sind (siehe Abschnitt 2.1.4). Nur wenige Eingaben des erhobenen Datensatzes sind im Passiv oder gar Konjunktiv ausgedrückt.

Bezogen auf die hiesige Problemstellung lässt sich in den Eingaben E1 bis E4 deutlich der Wunsch zur Funktionserweiterung des Roboters erkennen. In Eingabe E1 und E4 soll das Kaffeekochen, in E2 und E3 das Begrüßen einer Person erlernt werden. Diese Eingaben repräsentieren natürlichsprachliche Formulierungen einer typischen Lehrsequenz.
Einen charakteristischen Wortlaut bildet hierbei die Floskel in Eingabe E1 *„we are going to learn how to ...“*. Durch die verwendete Floskel wird eindeutig die Absicht des Probanden zur Funktionserweiterung des Zielsystems signalisiert. Daher bildet sie eine explizite Lehrsequenz. Die Eingaben E2, E3 und E4 sind ebenfalls als Lehrsequenz zu deuten, wobei hier der Wunsch des Nutzers, dem Roboter neues Wissen beizubringen, eher implizit formuliert wird. Bei impliziten Lehrsequenzen werden häufiger Wortlaute wie *„in order to ... you have to ...“*, *„to ... you have to ...“* und *„if you want to ...“* verwendet. Betrachtet man die Eingabe E4 genauer, fällt zudem auf, dass innerhalb der verbalisierten Lehrsequenzen nicht nur Sätze in Befehlsform, das heißt Imperativsätze, sondern auch Aussagesätze enthalten sein können. Die Unterschiede dieser Satztypen werden in Abschnitt 2.1.4 erläutert.
Die Eingaben E5 und E6 entsprechen keiner Lehrsequenz. Diese Eingaben enthalten weder einen expliziten noch impliziten Wunsch zur Funktionserweiterung. Sie bestehen aus einer reinen Anweisungsfolge, hier als *Befehlssequenz* bezeichnet. Für den Roboter ist dies als Skript zu interpretieren, dessen Teilaufgaben nacheinander und direkt ausgeführt werden sollen.

Definition 6.5.1: Befehlssequenz

Eine *Befehlssequenz* ist ein Kommando an ein Zielsystem mit dem Ziel der sofortigen Ausführung. Sie besteht aus einer Folge aus Imperativsätzen oder Aussagesätzen und bildet das Gegenstück zur *Lehrsequenz*.

Zuletzt zeigt die Eingabe E6, dass die Interpretation einer Eingabesequenz als Lehrsequenz oder Befehlssequenz nicht immer eindeutig ist. Im Falle von E6 ist unklar, ob der Nutzer dem Zielsystem dauerhaftes Wissen vermitteln oder stattdessen eine Anweisung befehligen möchte, deren Ausführung einmalig gültig ist. Dies lässt sich ohne Kontext der vorhergehenden Unterhaltung schwer bestimmen.

Eine strukturelle Betrachtung der Nutzereingaben ergibt, dass manche Satzkonstruktionen in der Datengrundlage wiederholt vorkommen. Dies betrifft vor allem ähnliche Wortlaute für die Lehrsequenzen. In der Nutzerstudie wird ein exemplarisches Szenario inklusive gegebener Lehrsequenz als Lösung präsentiert. Dieses birgt die Gefahr, dass sich die Studienteilnehmer in ihren Formulierungen zu sehr an den gegebenen Instruktionen dieses

Tabelle 6.6: Auszug erfasster Aktionsbenennungen zu Szenario „Prepare coffee“

Aktionsbenennungen		
bring someone a drink	bring drinks	bring beverages
bring one beverage	bring me some beverage	bring someone an orange juice
bring a glass of beverage	bring a cold drink for me	bring beverage to a human
bringing some beverage	serve a beverage	serve beverages
serve a drink	serve drinks	make a drink
make an orange juice	make some beverages	prepare a drink
get a drink	get a beverage	deliver a drink

Beispiels orientieren. Daher befinden sich in der Datensammlung, trotz der Forderung nach neuen Formulierungen (siehe Beispiel 6.3.2), viele Wortlaute der Lehrsequenzen aus dem Beispielszenario. Allerdings zeigt eine nähere Durchsicht dieser Eingaben, dass sie trotz der mehrfach verwendeten Lehrsequenz-Formulierungen sehr unterschiedlich zueinander sind. Dazu werden neben den charakteristischen Wortlauten die anderen Bestandteile einer Lehrsequenz näher untersucht. In der Analyse der Problemstellung in Kapitel 5 wird die Lehrsequenz hierfür in die *Kommandodeklaration* (einzuprogrammierendes Kommando) sowie die *Kommandobeschreibung* (Abfolge aus Teilaufgaben) unterteilt. Die beschriebenen Floskeln sind dabei charakteristisch für die Kommandodeklaration.
Eine weitere Komponente der Kommandodeklaration ist die vom Studienteilnehmer vergebene Benennung der zu erlernenden neuen Funktionalität. Diese enthält den Aktionsnamen inklusive der zugehörigen Entitäten und wurde als Aktionsbenennung definiert (siehe Definition 5.1.4). Um die Diversität der Kommandodeklarationen, speziell bei den gewählten Aktionsbenennungen der Kommandodeklaration in der Datengrundlage zu verdeutlichen, wird das Szenario „Prepare coffee“ der Studie exemplarisch betrachtet. Allein für die Aktionsbenennungen der zu lernenden Funktion sind viele verschiedene Varianten in den Nutzereingaben aufzufinden. Einen Auszug dieser gewählten Aktionsbenennungen wird in Tabelle 6.6 veranschaulicht.

Betrachtet man den zweiten Teil einer Lehrsequenz, die Kommandobeschreibung, so ist zeigt sich eine noch viel größere Variabilität der Eingaben. In der Kommandobeschreibung beschreibt der Studienteilnehmer die einzelnen Teilaufgaben der neuen Funktion. Sie fällt im Vergleich zur Kommandodeklaration daher meist ausführlicher aus, das heißt enthält durchschnittlich mehr Wörter. Weiterhin besteht sie aus mehreren Aktionsnamen und zugehörigen Entitäten, welche die Aktionsbenennungen der Kommandobeschreibung bilden. Im Szenario „Prepare coffee“ ergeben sich für die vorgegebenen Teilaufgaben deshalb zahlreiche weitere Kombinationsmöglichkeiten der einzelnen Aktionsbenennungen. Diese Variabilität ist auch bei den erfassten Eingaben der anderen Szenarios ähnlich stark aufzufinden.

Als letzte strukturelle Beobachtung ist zu nennen, dass die Kommandodeklaration und die Kommandobeschreibung in keiner festen Reihenfolge auftreten. Überdies sind innerhalb der beiden Blöcke neben den Imperativsätzen auch Aussagesätze (siehe Abschnitt 2.1.4) wiederzufinden, welche ebenfalls keiner speziellen Anordnungsvorschrift zu folgen scheinen. Sie können am Anfang und am Ende eines Blocks, aber auch zwischen den einzelnen Teilaufgaben der Kommandobeschreibung auftreten.

Anschließend werden die eingereichten Eingaben auf syntaktischer Ebene untersucht. Dabei fällt auf, dass **Schreibfehler** sehr häufig auftreten. Diese lassen sich in drei Gruppen aufteilen:

- Tippfehler (*typo*): Einzelne Buchstaben sind inkorrekt.
  Beispiel: *„machne“* statt *„machine“* (E4), *„becerage“* statt *„beverage“*(E4)

- Ersetzungsfehler: Ganze Wörter sind inkorrekt.
  Beispiel: *„Burton“* statt *„button“* (E4), *„average“* statt *„beverage“*(E5)
- Grammatikalische Fehler: Fehlende oder falsche Verwendung von Satzgliedern.
  Beispiel: *„pour beverage to the glass“* (E5) statt *„pour the beverage into the glass“*

Die auftretenden Tippfehler könnten als mögliche Flüchtigkeitsfehler zu interpretieren sein. Die überdies vorkommenden Ersetzungsfehler sind auf mögliche automatische Wortkorrekturen zurückzuführen. Dies zeigt das Beispiel *„average“* statt *„beverage“* in Eingabe E5. Zuletzt weisen die vorkommenden grammatikalischen Fehler auf weniger gute Englischkenntnisse des Probanden hin.
Während Tippfehler durch verschiedene Programmbibliotheken möglicherweise korrigiert werden könnten, werden die Ersetzungs- und grammatikalischen Fehler nicht automatisch zu erkennen und zu korrigieren sein. Besonders die letzten beiden Fehlergruppen könnten aufgrund der Fehlerfortpflanzung in der PARSE-Vorverbeitungskette daher zu fehlerhaften, irreführenden Eingaben und somit zu falschen Ergebnissen des zu entwickelnden Werkzeuges zur Synthese von Methodendefinitionen führen. Deshalb werden die Eingabesequenzen einer manuellen Korrektur unterzogen. Fehler in der Grammatik der formulierten Nutzereingabe werden jedoch nicht behoben.

6.5.3 Zusammenfassung

Im Rahmen dieser Arbeit dient die Vorstudie als Informationsquelle, um die menschliche Art und Weise der Wissensvermittlung an ein Subjekt zu untersuchen. Beim Menschen geschieht diese Vermittlung von Wissen durch einen Lehrer oder andere Bezugspersonen, welche die gewünschten Kenntnisse mittels schrittweiser Erklärungen an den Lehrling kommunizieren. Die Basis dieser Informationsübertragung bildet hierbei die natürliche Sprache.
Bezogen auf die Problemstellung dieser Arbeit wird der Mensch weiterhin als Lehrer, als Lehrling jedoch ein Zielsystem betrachtet. Die innerhalb der Vorstudie erfassten Äußerungen der Probanden deuten auf einen natürlichen, menschenähnlichen Umgang mit dem System hin. Ein Musterbeispiel dafür bildet die durchaus häufig verwendete Formulierung *„if you want to ...“*, bei welcher der betrachtete Roboter nicht nur als intelligentes, sondern als vermeintlich einem freien Willen folgenden System erachtet wird. Daher lässt die gewonnene Datengrundlage der Vorstudie grundsätzlich erkennen, wie Menschen im Gespräch neues Wissen vermitteln. Die externe Validität (engl. *external validity*) [Chr07] der erhobenen Datengrundlage, das heißt die Generalisierbarkeit der Studienergebnisse auf die reale Welt ist somit gewährleistet. Weiterhin zeigt die Studie, dass die Probanden natürlichsprachliche Äußerungen formulieren, die sich von Nutzereingaben in gesprochener Form nicht signifikant unterscheiden.
Die Arbeit konzentriert sich auf die Art und Weise der Eingaben, welche das Zielsystem um neue Funktionalitäten erweitern sollen. Der Fokus der Studie liegt hierbei auf den verbalisierten Lehrsequenzen und nicht der inhaltlich korrekten oder vollständigen Beschreibungen der spezifischen Teilaufgaben. Es fällt auf, dass der Mensch für die Verbalisierung dieser Lehrsequenzen bestimmte Wortlaute bevorzugt. Viele der verwendeten Formulierungen ähneln sich auf den ersten Blick stark.
Die zugrundeliegenden Satzkonstruktionen der Nutzereinreichungen, die dabei verwendeten Wörter und Art der Umschreibung der Aktionsbenennungen unterscheiden sich jedoch deutlich. Der Mittelwert und die Standardabweichung (siehe Tabelle 6.5) der verwendeten Wörter pro Einreichung bestätigen diese Variabilität der verschiedenen Eingaben. Diese Tatsache macht die Erkennung von Lehrsequenzen entgegen mancher zunächst ähnlichen Formulierungen zu einer beliebig komplexen Aufgabe.
Zudem werden innerhalb der Nutzerstudie entgegen der Aufgabenstellung auch viele Eingaben ohne explizit oder implizit formulierte Lehrsequenz abgegeben. Die betroffenen

Eingaben entsprechen einer Befehlssequenz (siehe Definition 6.5.1). Diese Einreichungen werden ebenfalls als valide Daten der Vorstudie gewertet und in die Datengrundlage aufgenommen. Für den hiesigen Lösungsansatz gilt es diese bei der Erkennung von Lehrsequenzen zusätzlich zu unterscheiden.
Zusätzlich zu diesen Problemstellungen stellen die Vielfalt der Formulierungen, geschaffen durch die Komplexität der natürlichen Sprache, die auftretenden Grammatikfehler und die mehrdeutigen Interpretationsmöglichkeiten einer Eingabe eine zusätzliche Herausforderung für diese Arbeit dar.

7 Analyse

In diesem Kapitel werden die zu bewältigenden Herausforderungen der Arbeit diskutiert, welche sich aus der Zieldefinition (Kapitel 5) und der Vorstudie (Kapitel 6) ergeben. Anschließend wird die Anwendbarkeit der Ansätze aus den verwandten Arbeiten (Kapitel 4) diskutiert. Den Hauptteil bildet die Konzeption von konkreten Ansätzen, die eine mögliche Lösungsstrategie der Teilprobleme dieser Arbeit darstellen. Dazu zählen beispielsweise die einzelnen Schritte zur Detektion von Lehrsequenzen sowie die Abbildung des Inhaltes der Nutzeranfrage auf die Roboterschnittstelle.

7.1 Herausforderungen

Aus der vorhergehenden Analyse der Problemstellung in Abschnitt 5.1 sowie der Auswertung der Vorstudie in Abschnitt 6.5 lassen sich verschiedene Herausforderungen bestimmen, die es im Zuge dieser Arbeit zu bewältigen gilt. Hierbei werden aus dem im Kapitel 5 der Zieldefinition vorgestellten Teilaufgaben der Arbeit (siehe Abbildung 5.1) drei übergeordnete Herausforderungen (H 1) bis (H 3) bestimmt. Diese werden im Folgenden kurz zusammengefasst. Anschließend werden diese übergeordneten Herausforderungen in weitere einzelne Schwierigkeiten unterteilt. Diese werden ebenfalls kurz erläutert. Ziel ist es, in der darauffolgenden Analyse auf diese einzelnen Herausforderungen einzugehen und sie für die Konzeption von Teillösungen hinzuzuziehen.

(H 1) Lehrsequenz-Erkennung: Die Nutzereingabe muss darauf untersucht werden, ob sie eine Funktionserweiterung des Zielsystems (Lehrsequenz, Definition 5.1.1) oder eine direkte Befehlsausführung (Befehlssequenz, Definition 6.5.1) repräsentiert. Dazu müssen die einzelnen Bestandteile der natürlichsprachlichen Eingabe, wie beispielsweise die Deklaration eines Kommandos (Definition 5.1.2), detektiert werden.

- (H 1.1) Uneingeschränkte, natürlichen Sprache: Die natürliche Sprache besitzt eine komplexe zugrundeliegende Grammatik. Dieses Regelwerk muss bei der Betrachtung von Lehrsequenzen untersucht werden, um mögliche syntaktische Muster abzuleiten

- (H 1.2) Keine Satztrennungszeichen[1]: Da keine Satzzeichen zur Verfügung stehen, sind die Enden der einzelnen Lehrsequenz-Bestandteile unbekannt

[1] In der Vorstudie werden zwar textuelle Eingaben inklusive Satzzeichen akquiriert, da jedoch ein Ansatz für gesprochene Sprache entwickelt werden soll, werden diese nicht genutzt.

- (H 1.3) Vielfalt der Formulierungen: Die Vorstudie zeigt, dass für verbalisierte Lehrsequenzen eine Vielzahl an Formulierungsmöglichkeiten expliziter und impliziter Art existieren
- (H 1.4) Mehrdeutigkeit der Formulierungen: Die Vorstudie zeigt auch, dass manche Formulierungen nicht eindeutig als Lehrsequenz zu interpretieren sind
- (H 1.5) Generalisierung: Für das Generalisieren der Detektion auf neue Argumente und Kontexte ist ein generischer Lösungsansatz nötig

(H 2) Aufbau von Methodendefinitionen: Die detektierten Bestandteile einer Lehrsequenz beziehungsweise Befehlssequenz sollen anschließend auf eine Entsprechung im Programmkontext abgebildet werden können. Dazu muss eine allgemeingültige Datenstruktur zur Darstellung dieser definiert werden.

- (H 2.1) Allgemeingültigkeit: Die Datenstruktur muss die Vielfalt der möglichen Formulierungen beziehungsweise Detektionsergebnisse abdecken. Dazu muss sie insbesondere zwischen Lehrsequenzen und Befehlssequenzen unterscheiden
- (H 2.2) Programmintegration: Die Datenstruktur muss die Inhalte der Nutzereingabe im Sprachkontext als auch eine Entsprechung auf Programmebene abbilden
- (H 2.3) Referenzauflösung: Mehrfach auftretende Instanzen der selben Entität der Nutzereingabe müssen identifiziert werden.
- (H 2.4) Fehlerbehaftete Ergebnisse: Fehlerbehaftete Detektionsergebnisse aus (H 1) müssen berücksichtigt werden

(H 3) Ontologie-Abbildung: Zuletzt müssen die extrahierten Programmelemente auf konkrete Konzepte in der Zielsystem-Ontologie abgebildet werden. Dabei gilt zu beachten, dass die natürliche Sprache Zusammenhänge oft unpräzise und unterspezifiziert im Gegensatz zu eindeutigen formalen Definitionen verfasst.

- (H 3.1) Keine Bezeichnervorschrift: Die Bezeichnung der Ontologiekonzepte folgen keiner Vorschrift
- (H 3.2) Fehlende Parameter: In der Nutzereingabe können fehlende Parameterangaben zu existierenden Zielsystem-Funktionen auftreten
- (H 3.3) Unbekannte Eingaben: In der Nutzereingabe können unbekannte, nicht auf die Zielsystem-Programmschnittstelle Elemente enthalten sein
- (H 3.4) Mehrere Kandidaten: Es darf nur eine eindeutige Ontologiekonzept-Abbildung existieren
- (H 3.5) Kontrollstrukturen: Formulierungen von Kontrollstrukturen in den Eingaben (wie Schleifen und Bedingungen) sollen in der Abbildung gespeichert werden können

Diese präsentierten Herausforderungen müssen in der folgenden Analyse und Lösungskonzeption der einzelnen Teilaufgaben in diesem Kapitel betrachtet werden. Dabei bildet die Herausforderung zur ersten Teilaufgabe (H 1) neben der umfassenden Vorstudie in Kapitel 6 den Hauptteil dieser Arbeit. Für die weiteren beiden Teilaufgaben wird ebenfalls ein Lösungsansatz entworfen, um eine vollständige Ende-zu-Ende-Implementierung und Evaluation der Synthese von Methodendefinitionen dieser Arbeit zu ermöglichen.

7.2 Diskussion der verwandten Arbeiten

Anschließend werden die in Kapitel 4 vorgestellten Ansätze der verwandten Arbeiten mit den festgelegten Zieldefinitionen aus Kapitel 5 sowie den oben präsentierten Heraus-

forderungen dieser Arbeit verglichen. Es muss geprüft werden, ob und wie die gewählten Ansätze auf die Problemstellung dieser Arbeit übertragbar sind.

Zunächst lässt sich festhalten, dass die betrachteten Arbeiten ihren Lösungsansatz für das Hinzufügen neuer Systemfunktionalitäten häufig auf spezifische Domänen richten, wie beispielsweise der Excel-Dokumentenmodifikation von Wiest [Wie14] und Gulwani [GM14], oder der Smartphone-Programmierung [LGS13] von Le. Ziel dieser Arbeit ist es, diese Funktionserweiterung unabhängig von der Domäne, anhand von linguistischen Merkmalen der Eingaben zu ermöglichen. In dieser Arbeit werden uneingeschränkte, natürlichsprachliche Nutzereingaben in gesprochener Sprache betrachtet. Demnach darf nicht von einer durch Satzzeichen segmentierten Eingabe ausgegangen werden, wie beispielsweise in den Arbeiten von Liu [LL05], Rabinovich [RSK17] und Akari [ALHM14]. Weiterhin stützen sich viele andere Arbeiten wie die von Le [LGS13], Gulvani [GM14] und Campagna [CRX+17], auf gezielte Schlüsselwörter für das Diktieren von neuen Funktionen. Ein regelbasiertes Herangehen kann in dieser Arbeit unterstützend einfließen, soll aber nicht Basis des zu entwickelnden Werkzeuges sein. Außerdem kann in vielen Systemen nur eine einzelne, kurze Instruktion behandelt werden, die in den meisten Fällen keine Kontrollstrukturen enthalten darf. Diese Einschränkungen der Nutzereingabe wird unter anderem in den vorgestellten verwandten Arbeiten von Lauria [LBKK02], She [SCC+14] und Wachtel [WWVT15] vorausgesetzt. Die Betrachtung von langen Instruktionsanweisungen aus komplexen Formulierungen und verbalisierten Kontrollstrukturen soll auch in dieser Arbeit behandelt werden. Um diese Einschränkungen auszugleichen, basieren viele Systeme auf den (textuellen oder gesprochenen) Dialog zwischen Nutzer und System. Auf diese Weise können gezielt benötigte Informationen abgefragt werden wie in dem Ansatz von Wachtel [WWVT15], She [SCC+14], Le [LGS13] und Azaria [AKM16]. Zudem wird der Nutzer in vielen Anwendungen durch eine grafischen Oberfläche in seinem Anliegen unterstützt. Dazu zählen die Menü-basierten Anwendungen von Manning [WGLM17] und Campagna [CRX+17]. Dem Nutzer kann anschließend die von ihm gewünschte Funktion aus einer Liste generierter Programmkandidaten auswählen. Im Gegensatz dazu stellt das Projekt PARSE keine grafische Nutzerschnittstelle zur Verfügung. Dem Gegenüber kann dadurch keine Auswahl an möglichen Systeminterpretationen der Eingabe präsentiert werden. Stattdessen soll eine Lösung erarbeitet werden, die bei erfolgreicher Abbildung des Inhaltes einer Nutzereingabe auf verschiedene Zielsystemfunktionen den treffendsten Kandidaten auswählt. Bei einer fehlerhaften Abbildung könnte, analog zur genannten Arbeit, durch Nachfragen oder Rekursion nicht erkannter Teilfunktionen vorgegangen werden. Diese Interaktion müsste von der Dialogkomponente des Projekts PARSE übernommen werden.

Zuletzt können auch wissenschaftliche Arbeiten anderer Themenbereiche interessante Ansätze für das Vorgehen in dieser Arbeit liefern. So sind einige Arbeiten aus den Bereichen der Klassifikation von Textbestandteilen deren Abbildung auf Wissensrepräsentationen für einzelne Teilaufgaben dieser Arbeit relevant. Auf diese spezifischen verwandten Arbeiten wird den entsprechenden Stellen dieser Arbeit in kurzer Form eingegangen.

7.3 Konzeption von Teillösungen

In diesem Abschnitt werden mögliche Lösungen für die drei im Kapitel 5 der Zieldefinition differenzierten Teilaufgaben der Arbeit konzipiert und diskutiert. Dazu gehören die Lehrsequenz-Erkennung, die Synthese von Aktionsbenennungen und die Ontologie-Abbildung (siehe Abbildung 5.1). Die im Kapitel 6 der Vorstudie gewonnenen Erkenntnisse sowie das Vorgehen verwandter Arbeiten fließen in diese Konzeption mit ein.

Tabelle 7.1: Bestandteile einer Lehrsequenz

Bestandteil	Bescheibung	Def.	Frage
Lehrsequenz	Nutzer verbalisiert Funktionserweiterung	5.1.1	1
Befehlssequenz	Nutzer diktiert Anweisungssequenz	6.5.1	1
Kommandodeklaration	Floskeln und Name der neuen Funktion	5.1.2	2
Kommandobeschreibung	Teilaufgaben der neuen Funktion	5.1.3	2
Aktionsname	Teilaufgabe / Aktion	5.1.4	3
Entität	Zugehörige Entitäten einer Aktion	5.1.4	3

7.3.1 Lehrsequenz-Erkennung

Mit der Lehrsequenz-Erkennung soll geprüft werden, ob der Nutzer eine Funktionserweiterung des Zielsystems durch seine Eingabe beabsichtigt. Wird ein solcher Erweiterungswunsch erkannt, sollen die Einzelheiten dieser beschriebenen, neuen Funktionalität, kurz Funktion, aus der Nutzereingabe extrahiert werden.

7.3.1.1 Bestandteile einer Lehrsequenz

Für die Detektion der Bestandteile einer Lehrsequenz müssen folgende drei Fragestellungen über eine betrachtete Eingabesequenz beantwortet werden:

1. Wird das Hinzufügen einer Funktion oder ein direkter Auruf beabsichtigt?
2. Wo wird der Wunsch zur Erweiterung verbalisiert und wo ihr Inhalt beschrieben?
3. Was sind die einzelnen Aktionen der neuen Funktion?

Die Antworten auf diese Fragestellungen entsprechen den Bestandteilen einer Lehrsequenz, welche es im Zuge dieser Teilaufgabe der Arbeit zu extrahieren gilt. Diese Lehrsequenz-Bestandteile wurden bereits in den vorherigen Kapiteln der Zieldefinition (Kapitel 5) und der Vorstudie (Kapitel 6) der Arbeit definiert. Die Tabelle 7.1 stellt diese nochmals durch eine kurze Beschreibung vor und präsentiert den Zusammenhang zu den drei Fragestellungen.

Die Detektion von Lehrsequenzen beinhaltet nicht nur die binäre Entscheidung, ob eine Lehrsequenz in der Eingabe vorhanden ist oder nicht. Für die weitere Problemlösung der Arbeit ist auch eine Erkennung der Kommandodeklaration und der Kommandobeschreibung, als auch die Bestimmung der jeweiligen Aktionsnamen und zugehörigen Entitäten nötig.
Damit entspricht die Lehrsequenz-Erkennung der Lösung eines klassischen Klassifikationsproblems: Für eine Eingabe wird ein diskretes Ergebnis, das heißt eine Zuordnung zu einer von sogenannten Klassen, erwartet. Die Problemstellung wird auch als Sequenzmarkierungsaufgabe (engl. *sequence labeling task*) bezeichnet, das heißt einer Musterkennungsaufgabe, bei der jedes Element der Eingabe in Abhängigkeit der anderen Markierungen der Sequenz kategorisiert werden muss. Bekannte Sequenzmarkierungsprobleme der Forschung sind die Wortartmarkierung (engl. *part of speech tagging*, siehe Abschnitt 2.2.1.1) und die Eigennamenerkennung (engl. *named entity recognition*).
Zuletzt lässt sich aus dem Vergleich zwischen den Ansätzen verwandter Arbeiten und der Problemstellung dieser Arbeit in Abschnitt 7.2 vermuten, dass ein regelbasierter Ansatz als Lösungsstrategie für diese Arbeit nicht ausreicht. Deshalb wird ein Ansatz basierend auf maschinellem Lernen für die Lösung des Klassifikationsproblems gewählt (siehe Abschnitt 2.4). Diese Ansätze verzeichnen in den letzen Jahren große Erfolge in der Forschung (siehe Abschnitt 2.4). So wenden beispielsweise die Arbeiten von Schmidhuber [HS97], Gers [GSC00] und Manning [Man11] maschinelles Lernen für die Probleme der Wortarten- beziehungsweise der Eigennamenerkennung an. Unter den drei Teilaufgaben der Arbeit

bildet die Detektion von Lehrsequenzen die größte und komplexeste Problemstellung. Aus diesem Grund wird sie im Folgenden weiter untergliedert.

7.3.1.2 Hierarchischer Ansatz

Die Lehrsequenz-Erkennung soll durch eine Klassifikation der einzelnen Bestandteile einer Lehrsequenz angegangen werden. Dabei wäre ein Ansatz, die Lehrsequenz-Erkennung in einem gemeinsamen Klassifikationsschritt zu lösen. Dies würde jedoch ein komplexes Entscheidungsproblem für den Klassifikator darstellen, da die einzelnen, jeweils unterschiedlichen Bestandteile einer Lehrsequenz gleichzeitig klassifiziert werden müssen. Dabei beeinflusst die Größe des Entscheidungsproblems die Trainingsdauer des eingesetzten Klassifikators. Außerdem wird bei steigender Problemgröße eine größere Menge an Beispieldaten für den Klassifikator benötigt. Überdies lässt sich an den drei präsentierten Fragestellungen über die Bestandteile einer Lehrsequenz erkennen, dass sie aufeinander aufbauen. Während die erste Fragestellung allgemein auf die Intention der Nutzereingabe ausgerichtet ist, bauen die beiden nächsten Fragen darauf auf, um den weiteren Inhalt der Eingabesequenz zu identifizieren. Die einzelnen Bestandteile einer Lehrsequenz stehen damit in einer hierarchischen Beziehung zueinander. Daher bietet sich ein stufenweises Vorgehen bei der Klassifikation an. Ein großer Vorteil einer stufenweisen Erkennung von Lehrsequenzen ist, dass durch die Aufspaltung feingranularere und weniger komplexe Klassifikationsprobleme entstehen. Diese Komplexität eines Entscheidungsproblems lässt sich an der Anzahl der zu klassifizierenden Klassen messen, welche in Folge einer Aufspaltung in mehrere Stufen verteilt werden. Dieser hierarchische Ansatz wird auch von der Arbeit von Cohen [CRM07] verfolgt. Dort beschreibt er, dass durch dieses Vorgehen das Erlernen von internen Strukturen der Daten für den Klassifikator aufgrund der einfacheren Teilprobleme verbessert wird.

Bezogen auf die hiesige Klassifikation entstehen entsprechend zu den drei Fragestellungen drei Stufen der Klassifikation. In Stufe 1 soll mittels einer binären Klassifikation entschieden werden, ob eine Eingabe eine verbalisierte Lehrsequenz enthält oder nicht (Befehlssequenz). Das daraus resultierende Ergebnis kann anschließend als Eingabe für die nachfolgende Stufe 2-Klassifikation dienen. Hier sollen die detektierten Lehrsequenzen aus Stufe 1 auf Kommandodeklarationen und -beschreibungen untersucht werden. Im Gegensatz dazu können Befehlssequenzen keine Kommandodeklarationen enthalten. Die Gesamtheit der Eingabe wird deshalb als Kommandobeschreibung interpretiert. Zuletzt soll in Stufe 3 für jeden dieser beiden Kommando-Einheiten klassifiziert werden, welche Elemente der Eingabe den Namen der Aktionen und welche die Entitäten bilden. Diese hierarchischen Beziehungen der Lehrsequenz-Bestandteile werden in Abbildung 7.1 dargestellt.

Hierbei dient als Eingabe einer Stufe ein relevanter Ausschnitt der Klassifikationsergebnisse vorheriger Stufen. Damit verringert sich nicht nur die Komplexität des ursprünglichen Klassifikationsproblems, sondern auch der betrachtete Datensatz je Stufe. Dies könnte einerseits zu einer möglichen verbesserten Klassifikationsgüte einzelner Stufen führen, da der Fokus auf gezielte Datenausschnitte präzisiert ist. Eine Trennung in unterschiedliche Stufen erleichtert zudem die Interpretierbarkeit der Klassifikationsergebnisse, da diese als Antwort auf die drei Fragestellungen zur Erkennung von Lehrsequenz-Bestandteilen zurückgeführt werden können. Andererseits könnte die verringerte Datenbasis je Stufe den Klassifikator in seinem Strukturerkennungsverhalten auch einschränken. Ein stufenbasierter Ansatz hat überdies den Nachteil, dass sich Fehler auf nachfolgende Klassifikationsstufen fortsetzen. Außerdem wird für die Evaluation des Klassifikators statt einem einzelnen Goldstandard nun eine Musterlösung pro Klassifikationsstufe je Eingabe benötigt.
Aufgrund der überwiegenden Vorteile wird ein hierarchischer Ansatz für die Teilaufgabe der Lehrsequenz-Erkennung angestrebt. Die enthaltenen Bestandteile einer Lehrsequenz entsprechen jedoch noch nicht den konkret zu erkennenden Klassen des Klassifikation-

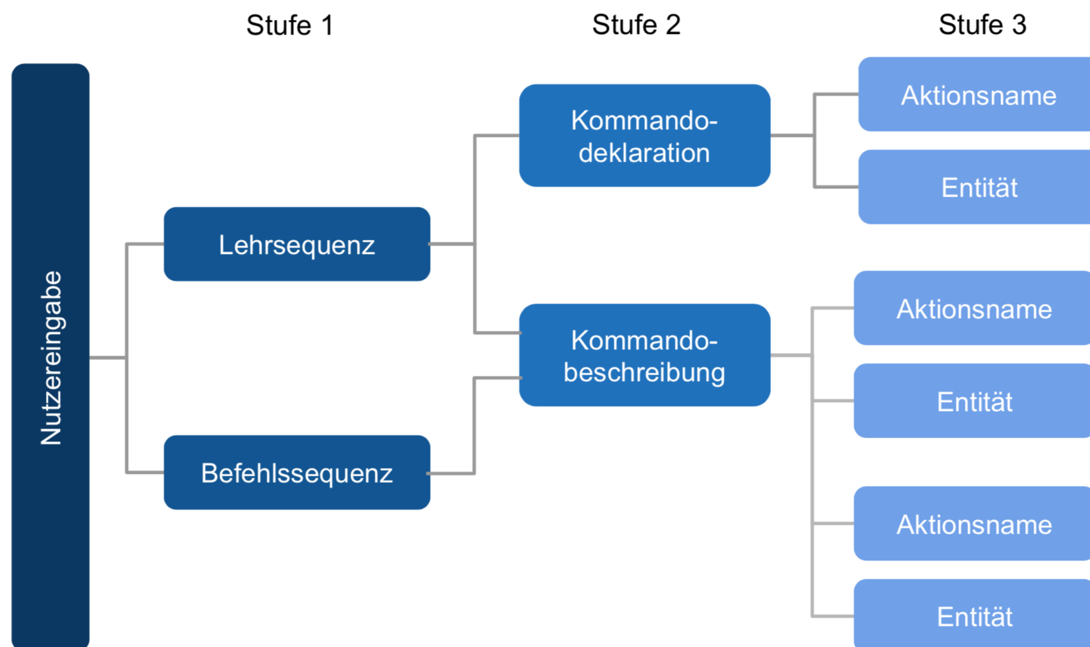


Abbildung 7.1: Hierarchie der Lehrsequenz-Bestandteile

sproblems. Diese werden im nachfolgenden Entwurfskapitel in Abschnitt 8.1 spezifiziert.

Weiterhin können zur Lösung des Klassifikationsproblems auch andere Agenten des Projekts PARSE (Kapitel 3) und deren, aus den Nutzereingaben extrahierte, Informationen miteinbezogen werden. Damit kann, anstatt alle Bestandteile der Lehrsequenz durch selbst modellierte Klassifikatoren zu gewinnen, auf bestehendes Wissen zurückgegriffen werden. Dies betrifft die Bestandteile einer Lehrsequenz zu Fragestellung drei, welche die Aktionsnamen und zugehörigen Entitäten der Teilaufgaben einer vom Nutzer neu beschriebenen Funktion enthalten. Diese Informationen werden bereits durch andere Sprachverarbeitungskomponenten von Projekt PARSE verarbeitet. Für die anderen beiden Stufen des hierarchischen Klassifikationsansatzes liegen keine in PARSE verwertbaren Informationen vor.
Deshalb wird für die ersten beiden Klassifikationsstufen ein Ansatz mit selbst modellierten maschinellen Klassifikationsverfahren gewählt, währenddessen die letzte Stufe unter Einbezug von Informationen eines bestehenden PARSE-Agenten gelöst werden soll.

7.3.1.3 Maschineller Klassifikationsansatz

Für die Durchführung einer Klassifikation gibt es, unabhängig von der Problemstellung dieser Arbeit, verschiedene Ansätze. Bei einem regelbasiertem Vorgehen wird versucht, das menschliche Verständnis für die Lösung des Klassifikationsproblems in Form von Regeln zu modellieren. Diese Regeln werden händisch von Experten erstellt und anschließend auf das Problem angewendet. Eine der bekanntesten Arbeiten nach diesem Ansatz sind die von Brill [Bri92], [Bri94], um das Klassifikationsproblem der Wortartmarkierung zu lösen. Der massive Aufwand für die Erstellung dieser Wissensbasis bildet jedoch einen großen Nachteil. Hinzukommend ist in den meisten Fällen nur eine eingeschränkte Abbildung der realen Welt auf ein Regelwerk möglich. Herausforderungen dabei sind, die Komplexität der natürlichen Sprache (H 1.2) und die damit verbundenen Vielfalt (H 1.3) und Mehrdeutigkeit (H 1.4) der Formulierungen abzudecken (siehe Abschnitt 7.1).
Ein maschineller Lernansatz soll diese Nachteile kompensieren. Es wird versucht, die genannte Wissensbasis mit Hilfe großer Datenmengen künstlich zu generieren (siehe Abschnitt 2.4). Voraussetzung für einen solchen Ansatz ist eine große, zum Problem passende

Datenbasis. Da für diese Arbeit im Rahmen der Vorstudie (siehe Kapitel 6) ein größerer, auf die Problemstellung angepasster Datensatz erhoben werden konnte, ist die genannte Voraussetzung erfüllt. Aufgrund der höheren Robustheit von Verfahren des maschinellen Lernens im Vergleich zu regelbasierten Ansätzen werden diese zur Lösung der hiesigen Klassifikationsprobleme eingesetzt.

Weiterhin lassen sich Verfahren des maschinellen Lernens in überwachte und unüberwachte Ansätze unterteilen (siehe Abschnitt 2.4). Unüberwachte Verfahren werden häufig für die Segmentierung von Daten in ähnliche Gruppen und für eine erste Datenvisualisierung eingesetzt. Hierfür müssen die erwarteten Zielgrößen bei unüberwachten Verfahren vorab nicht bekannt sein. Im Unterschied dazu sind bei überwachten Lernverfahren diese Zielgrößen, die sogenannten Klassen, bereits bekannt. Sie müssen dem Klassifikator zusätzlich zu den Eingabedaten mitgegeben werden. Anschließend erlernt der Klassifikator diese Zielgrößen aus dem Datensatz und kann sie auf unbekannten Daten vorhersagen. Bezogen auf das hiesige Klassifikationsproblem, ähneln die genannten Klassen der Einteilung einer Nutzereingabe in die gewünschten Bestandteile einer Lehrsequenz. Für die Detektion von Lehrsequenzen bietet sich daher der Einsatz von überwachten Lernverfahren an. Diese Klassen werden im nachfolgenden Entwurfskapitel in Abschnitt 8.1 definiert. Auch die Erstellung des dazu nötigen Goldstandards für das Training und die Evaluation des Klassifikators wird dort beschrieben.
Für die Lösung der ersten beiden Stufen der Lehrsequenz-Detektion wird daher ein maschineller Ansatz mit überwachten Lernverfahren verfolgt. Das *No Free Lunch Theorem* [WM97] besagt, dass kein universeller Algorithmus (hier Verfahren des maschinellen Lernens) für die Lösung eines Optimierungsproblems existiert. Das heißt, die Verallgemeinerung, dass ein Lernverfahren durchweg besser als ein anderes ist, ist falsch. Es spielen viele Faktoren, wie beispielsweise die Struktur und Größe der Daten sowie die Ausprägung der Modellparameter eine wichtige Rolle. Deshalb werden in dieser Arbeit viele verschiedene, sowohl symbolische Verfahren als auch neuronale Netze, in der Implementierung betrachtet. Diese sollen nacheinander auf das Problem angewandt und gegeneinander evaluiert werden. Für den Einsatz von überwachten Lernverfahren ist ein Goldstandard nötig, der für die erhobene Datenbasis der Vorstudie erstellt werden muss. Die Einzelheiten dieses Annotationsvorgangs werden im Entwurfs-Kapitel 8 festgelegt.

Eine weitere Unterteilung maschinellen Lernens ist in klassische Lernverfahren und Lernverfahren auf Basis neuronaler Netze (siehe Abschnitt 2.4.1). Mögliche Modelle für die Klassifikation sind einerseits klassische Verfahren wie Entscheidungsbäume in der Arbeit von Marquez [MR98] oder Logistische Regressionsmodelle [ALHM14] wie von Akari. Andere Ansätze betrachten auch beispielsweise Hidden-Markov-Modelle (HMM) [Kup92], [Rab90] oder Conditional Random Fields (CRF) [LMP01]. Sie gehören zu den symbolischen Ansätzen, in denen das erlernte Wissen explizit repräsentiert ist.
Bei nicht-symbolischen Ansätzen, wie den neuronale Netzen, wird zwar ein berechenbares Verhalten antrainiert, jedoch lässt sich das genaue Verhalten der Netze nicht vorhersagen. Es ist kein direkter Einblick in die Lösungswege möglich, das Wissen wird implizit repräsentiert. Da neuronale Netzarchitekturen zum aktuellen Stand der Forschung eine sehr hohe Güte in vielen Forschungsfragen erreichen, werden sie auch in der Disziplin der Sprachverarbeitung (engl. *Natural Language Processing*) eingesetzt. In der Arbeit von Collobert [CWB+11], Schmidhuber [HS97] und Gers [GSC00] werden erstmals Ansätze verwandter Klassifikationsprobleme wie die Wortartmarkierung oder die Eigennamenerkennung mit neuronalen Netzarchitekturen verfolgt. Für die Lösung der Klassifikationsprobleme der ersten beiden Stufen zur Detektion von Lehrsequenzen werden daher Modelle aus beiden Gruppen maschinellen Lernens betrachtet.

7.3.1.4 Klassifikation auf Grundlage von PARSE-Agenten

Zur Lösung der dritten Stufe der Lehrsequenz-Erkennung werden Informationen aus bestehenden Sprachverarbeitungskomponenten von Projekt PARSE genutzt. Gesucht werden die Bestandteile einer Lehrsequenz zur dritten Fragestellung, welche die Aktionsnamen und zugehörigen Entitäten einer vom Nutzer neuen Funktionalität enthalten.
Eine Möglichkeit, an diese Bestandteile einer Lehrsequenz zu gelangen, wäre der Einsatz des Aktionserkenner-Agenten aus der Arbeit von Ou [Ou16]. Dieser kann in den Eingabesequenzen befindliche Aktionen, bestehend aus den Rollen Akteur, Prädikat und Parametern, erkennen. Er speichert diese gewonnenen Informationen als Knotenattribute des Graphen, welcher aus den Wörtern der Eingabesequenz besteht. Die Beziehungen zwischen den Aktionen und Elementen einer Aktion werden durch Kanten in diesem Graphen repräsentiert. Die Evaluation des Werkzeugs innerhalb dieser Arbeit ergab eine Detektionsgüte von 78.3% Genauigkeit für Knoten (Rollen) sowie ein F_1-Maß von 75.5% für die einzelnen Kantentypen der Beziehungen.
Eine andere Möglichkeit bietet der Agent, welcher im Rahmen der Arbeit von Hey [Hey16] entworfen wurde. Dieses Werkzeug implementiert die Identifikation von Aktionen in der Nutzereingabe sowie die Erkennung von Beziehungen zwischen diesen als Argumente der Aktion. Diese detektieren Elemente der Nutzereingabe können im Bezug auf diese Arbeit direkt den Aktionsnamen und zugehörigen Entitäten der Lehrsequenz-Bestandteile zugeordnet werden. Die Beziehungen entsprechen den semantischen Rollen zweier Wörter (siehe Abschnitt 2.2.1.5) und werden mit Informationen aus verschiedenen lexikalischen Datenbanken und hinzugezogener Domänen- sowie Weltwissensontologien angereichert. Die Ergebnisse diesen Agenten werden anschließend als Kantenattribute dem PARSE-Graphen hinzugefügt. In der abschließenden Evaluation dieser Arbeit wurde eine Güte von 97.3% F_1-Maß für die Entitätenerkennung, 80.4% F_1 für die Aktionserkennung sowie 86.2% F_1 für die Beziehungsanreicherung ermittelt. Im Unterschied zum Aktionserkenner-Agenten ermittelt der Semantische-Rollen-Erkenner zusätzliche Informationen und kann überdies eine bessere Klassifikationsgüte vorweisen. Obwohl beide genannten Agenten für die dritte Stufe der Klassifikation verwendet werden können, wird daher der Agent aus der Arbeit von Hey [Hey16] gewählt.

7.3.2 Aufbau von Methodendefinitionen

In der zweiten Teilaufgabe der Arbeit gilt es die gewonnenen Ergebnisse aus der vorhergehenden Klassifikation weiterzuverarbeiten. Diese Ergebnisse bestehen aus Satzbestandteilen der Eingabesequenzen, welche durch den Klassifikator unterschiedlichen Klassen zugeordnet werden. Bisher repräsentieren die Klassifikationen einer Nutzereingabe die einzelnen Bestandteile einer Lehrsequenz (siehe Tabelle 7.1) auf Sprachebene. Für die spätere Erweiterung des Zielsystems der neu beschriebenen Funktionalität ist eine Repräsentation der Klassifikationsresultate im Programmkontext nötig.
Dazu müssen die Bestandteile einer Lehrsequenz, unter anderem bestehend aus Kommandodeklaration, Aktionsnamen und Entitäten, auf Programmelemente wie Methodenkopf, Methodenrumpf und Parametern abgebildet werden. Dies wird in Abschnitt 7.1 als Herausforderung (H 2.2) definiert. Anschließend sollen diese Programmbestandteile als Datenstruktur modelliert werden. Dabei soll beachtet werden, dass die Nutzereingabe sowohl einer Lehrsequenz als auch einer Befehlssequenz (siehe Definition 5.1.1 und Definition 6.5.1) entsprechen kann. Für die Ergebnisdarstellung innerhalb der Datenstruktur müssen deshalb für beide Fälle repräsentierbar sein (H 2.1). Zudem muss diese Datenstruktur allgemeingültig sein, das heißt, die Vielfalt der möglichen Formulierungen des Nutzers auf Sprachebene sowie deren Entsprechungen auf Programmebene müssen abgebildet werden können. Das mehrfache Auftreten von Instanzen der gleichen Entität in der Nutzereingabe, den sogenannten Korreferenzen (siehe Abschnitt 2.2.1.6), soll zudem identifiziert werden (H 2.3). Dies könnte mittels Informationen des Agenten für Korrefenzauflösung aus der Arbeit

von Hey [Hey16] umgesetzt werden. Eine weitere Herausforderung (H 2.4) ist der Umgang mit fehlerbehafteten Klassifikationsergebnissen. Diese könnten beispielsweise regelbasiert oder durch einen Mehrheitsentscheid verschiedener Modelle oder Verfahren zusätzlich verarbeitet werden.

Die zweite Teilaufgabe dieser Arbeit beinhaltet zudem die Anbindung des entwickelten Werkzeuges an das Projekt PARSE. Dazu muss der modellierte Klassifikationsansatz der ersten Teilaufgabe als neuer Agent für Sprachverständnis in PARSE integriert werden. Es soll damit eine Klassifikation neuer, unbekannter Eingabesequenzen durch die trainierten Modelle und verwendeten Agenteninformationen gewährleistet werden, welche anschließend in der Datenstruktur von PARSE gespeichert werden können. Für die Integration eines neuen Agenten in das Projekt PARSE müssen dessen Entwicklungskriterien übernommen werden. Es ergeben sich (unabhängig von dieser Teilaufgabe des Aufbaus von Methodendefinitionen) folgende Anforderungen:

(A1) Domänenunabhängigkeit: Das Werkzeug soll auch mit Eingabedaten anderer Domänen (wie neue Einsatzumgebung, neuer Roboter) umgehen können.

(A2) Modularität: Der Agent soll sich mit geringem Aufwand einzeln betrachten, testen und austauschen lassen.

(A3) Erweiterbarkeit: Der Agent soll mögliche zusätzliche Informationen (wie in Sprache erkannte Kontrollstrukturen) weiterer Agenten integrieren können.

(A4) Konfigurationsmöglichkeit: Durch das Setzen von Parametern soll es möglich sein, die Algorithmusausführung zu konfigurieren, um beispielsweise mögliche Erweiterungen oder andere Klassifikationsmodelle gezielt einfließen zu lassen.

7.3.3 Ontologie-Abbildung

Die in der Nutzereingabe beschriebene, neue Funktionalität soll später per Sprachkommando aufgerufen werden können. Dazu muss innerhalb der letzten Teilaufgabe der Arbeit die Programmschnittstelle (engl. *API*) des Zielsystems erweitert werden. Das Hinzufügen neuer Funktionen soll dabei über die Komposition von bereits einprogrammierten Zielsystemfunktionen möglich sein. Deshalb findet innerhalb der dritten Teilaufgabe dieser Arbeit die eigentliche Synthese von Methodendefinitionen statt. Hierfür ist es nötig, die Inhalte einer Nutzereingabe auf die Programmschnittstelle des Zielsystems abzubilden. In der Fachliteratur wird dazu häufig der Begriff des „Verankerns“ (engl. *grounding*) verwendet, um ein betrachtetes Element mit einer Entsprechung in der Weltmodellierung zu verknüpfen.

Die Programmschnittstelle des Zielsystems wird in das Projekt PARSE durch eine Domänen-Ontologie (siehe Abschnitt 2.7) repräsentiert. Um die vom Nutzer in der Eingabesequenz beschriebene, neue Zielsystemfunktionalität persistent abspeichern zu können, muss diese Ontologie daher um neue Elemente ergänzt werden. In der ersten Teilaufgabe der Lehrsequenz-Erkennung werden hierzu die Inhalte einer Nutzereingabe wie Aktionen und zugehörige Entitäten extrahiert. In der zweiten Teilaufgabe folgt die Repräsentation der Sprachinhalte in Programmkontext als Datenstruktur. Anschließend müssen die Elemente dieser Datenstruktur auf neue beziehungsweise bestehende Ontologie-Individuen abgebildet werden.

Ein einfacher Abgleich der jeweiligen Elementbezeichner wird in den meisten Fällen nicht ausreichen, um Übereinstimmungen zu finden. Dies liegt daran, dass es viele Möglichkeiten gibt, eine Funktionalität in der Ontologie sowie in Sprache zu benennen. Die Konjugation, der Numerus oder der Tempus von Wörtern sowie Synonyme erschweren einen einfachen namensbasierten Abgleich. Hinzu kommt, dass die Namensgebung der Ontologiekonzepte

keiner festen Vorschrift folgt (siehe Abschnitt 7.1, Herausforderung (H 3.1)). Dieses Problem der Abbildung wird in den Ansätzen der verwandten Arbeiten häufig dadurch umgangen, dass dem Nutzer der Funktionssatz des Zielsystems vorgegeben wird. Auf diese Weise ist in der Arbeit von Lauria [LBKK02] und She [SCC+14] eine direkte Abbildung zwischen Sprachelementen und Zielsystemfunktionen möglich. Eine andere Herangehensweise, wie beispielsweise in der Arbeit von Guadarrama [GRG+13] und Le [LGS13], ist der Einsatz von händisch erstellten Schablonen (engl. *templates*), um Informationen aus der Eingabe zu gewinnen. Diese werden meist in Verbindung mit Domänenlexika eingesetzt, in welcher die Verknüpfung zweier Elemente aus Sprach- und Zielsystemkontext fest definiert wird. Da das Projekt PARSE jedoch einen domänenunabhängigen Ansatz verfolgt, kann weder ein vordefinierter Instruktionssatz vorgegeben, noch Schablonen vorgefertigt werden.
Weiterhin gibt es auch einige Arbeiten, bei denen maschinelle Lernansätze basierend auf semantischen Zerteilern eingesetzt werden. Die Eingabesequenzen werden dazu in höhere, formale Repräsentationsformen transformiert. In der Anwendung von Azaria [AKM16], Zettlemoyer [ZC07] und Matuszek et al. [MHZF12] werden probabilistischen Zerteiler eingesetzt, um natürliche Sprachkommandos auf eine spezifische, Lambdakalkül-typisierte formale Sprache abzubilden. Anschließend werden überwachte Modelle für die Abbildung der formalen Sprache auf die eigentlichen Zielsystemfunktionen trainiert. Die Ambiguitäten der menschlichen Sprache, Referenzauflösung sowie fehlende Parameter werden in diesen Arbeiten nicht behandelt. Zuletzt sind Zerteiler-basierte Ansätze in der Regel auf Satzzeichen angewiesen. Deshalb ist auch diese Herangehensweise nicht als Lösungsstrategie die hiesige Arbeit geeignet.
Alternative Arbeiten von Mei [MBW15] oder Suhr [SA18] erlernen einen Ende-zu-Ende-Ansatz von Spracheingabe zu Aktionssequenzen mittels neuronalen Netzen. Als zusätzliche Informationsquelle zur Nutzereingabe dient eine Repräsentation des Weltzustandes. Weitere linguistischen Werkzeuge, wie Zerteiler oder domänenspezifische Lexika, sind jedoch nicht notwendig. Die betrachteten Zielsysteme bestehen jedoch aus einem geringen Instruktionssatz und weisen weniger komplexe Formulierungen als in dieser Arbeit auf. Hinzu kommt, dass die Modelle bisher nur bei Betrachtung einzelner Sätze eine zufriedenstellende Güte erzielen.
Zuletzt wird in der Arbeit von Landhäußer [LWT17] für die Transformation textueller Kommandos in ausführbaren Quelltext ebenfalls eine Verknüpfung zwischen Sprache und Zielsystemfunktionalitäten benötigt. Die Hauptkomponente der Arbeit bildet ebenfalls eine Domänenontologie, welche die API des Zielsystems repräsentiert. Für die Abbildung wird ein unscharfer Abbildungsalgorithmus (engl. *fuzzy matching*) vorgestellt. Dieser prüft neben dem direkten Vergleich der Elementnamen auch mögliche, aus WordNet gewonnene Synonyme. Weiterhin fließen die Relationen innerhalb der Domänenontologie in die Bestimmung einer Abbildung mit ein. Nachträglich wird durch eine Bewertungsfunktion, unter anderem bestehend aus Teilwortübereinstimmung und Wortlänge, der beste Kandidat ausgewählt. Dieses heuristische Vorgehen wird auch in der verwandten Arbeit von Allen [ACF+07] verfolgt. Obwohl der Abbildungsalgorithmus einen eher einfachen Ansatz nutzt, spricht positiv für ihn, dass er ohne aufwändig händisch erstellte Schablonen oder annotierte Datensätze auskommt. Weiterhin ist das beschriebene Vorgehen der Abbildungssuche unabhängig von der betreffende Domäne einsetzbar, erzielt jedoch eine etwas geringere Güte als die alternativen maschinellen Lernansätze.

Zusammenfassend erzielen viele der beschriebenen Ansätze in ihrer jeweiligen Domäne gute Ergebnisse. Für das Training von überwachten Modellen wird ein annotierter Datensatz benötigt, welcher die Zuordnung von Instruktionssequenzen zu erwarteten Aktionen enthält. Dieser Datensatz müsste bei einem Einsatz dieser Verfahren für diese Teilaufgabe händisch erstellt werden. Dies würde einen beträchtlichen zusätzlichen Zeitaufwand für den dritten Teil der Arbeit mit sich ziehen. Überdies werden trotz der unterschiedlichen

Vorgehensweisen die Modelle häufig auf eine spezifische Problemstellung angepasst und auf eingeschränkten Eingabetexten trainiert. Ziel des Projekt PARSE und dieser Arbeit ist es dagegen, eine allgemeine Abbildung zu ermöglichen und diese Verknüpfungen unabhängig von der Domäne und dem betrachteten Zielsystem herstellen zu können. Daher bietet sich ein Ansatz ähnlich der letzten beiden genannten Arbeiten an.

Für die Abbildung von Nutzereingabe auf Ontologie-Individuen könnten, zusätzlich zu den eigentlichen Bezeichnern, Synonyme (siehe Abschnitt 2.1.5) betrachtet werden. Um die Wahrscheinlichkeit für eine Übereinstimmung zweier Begriffe zu erhöhen, könnten die Bezeichner weiteren möglichen Sprachverarbeitungsvorgängen, wie beispielsweise der Bildung des Wortstamms (engl. *lemmatizing*, siehe Abschnitt 2.2.1.3), unterzogen werden. Ein weiterer Ansatz wäre die Verwendung von Wortvektoren. Dabei könnten die Wörter der Nutzereingabe sowie die Ontologieelement-Bezeichner durch Wortvektoren repräsentiert und anschließend mit einer Ähnlichkeitsmetrik auf Übereinstimmungen überprüft werden. Eine Schwierigkeit für die Ontologie-Abbildung ist die Herausforderung (H 3.4): Bei der Berechnung der Übereinstimmung zwischen den Elementen können mehrere Kandidaten als mögliche Abbildung in Frage kommen. Diese müssten mittels einer Bewertungsfunktion in einer Rangliste sortiert werden, um den besten Kandidaten zu identifizieren. Weitere Schwierigkeiten bilden nicht auffindbare Kandidaten (H 3.3) für eine Abbildung der Nutzereingaben-Elemente auf die Zielsystemschnittstelle (API). Kann keine Entsprechung in der API gefunden werden, muss die Anwendung das Fehlen von Informationen verdeutlichen. In einem nachfolgenden Schritt könnten diese Informationen anschließend durch eine Dialog-Interaktion oder einem rekursiven Aufruf des hiesigen Agenten erfragt werden. Damit zusammenhängend, sollen gefundene, mögliche Abbildungen mit nur niedrigen Konfidenzwerten aussortiert werden. Auf diese Weise sollen fälschliche Abbildungen vermieden werden. Zuletzt muss bei der Abbildung von Aktionen der Nutzereingabe auf bestehende Methoden der Zielsystem-Ontologie die Übereinstimmung der jeweils zugehörigen Entitäten beziehungsweise Parameter betrachtet werden. Hierbei muss für die abgebildeten Parameter überprüft werden, ob sie mit den Datentypen der gefundenen Methodenparameter übereinstimmen. Dabei kann es vorkommen, dass Parameterangaben der Nutzereingabe unvollständig sind (H 3.2). Wird keine passende Abbildung gefunden, muss dies ebenfalls gekennzeichnet werden. Diese fehlenden Parameter könnten anschließend wiederum durch eine Dialog-Interaktion behandelt werden.

Nachdem eine Entsprechung zwischen den Elementen einer Nutzereingabe und den Ontologie-Individuen gefunden wurde, müssen diese Abbildungsinformationen gespeichert werden. Wurde überdies die Nutzereingabe durch den Klassifikator als Lehrsequenz identifiziert, soll die beschriebene neue Funktion in das Zielsystem integriert werden. Eine Möglichkeit wäre hierfür die direkte Modifikation beziehungsweise Ergänzung der Domänen-Ontologie. Die Herausforderung (H 3.5) schreibt vor, dass eine Nutzereingabe verbalisierte Kontrollstrukturen enthalten kann, die ebenfalls im Zielsystem repräsentiert werden müssen. Die Detektion dieser Kontrollstrukturen, wie beispielsweise Bedingungen und Schleifen, wird durch weitere Agenten von PARSE in den Arbeiten von Weigelt [WHS18b] und Steurer [WHS18a] getätigt. Damit die vom Nutzer neu definierte Funktionalität der Lehrsequenz jedoch inklusive Kontrollstrukturen abgespeichert werden kann, müsste deren Integration im gleichen Zug mit der Ontologiemodifikation erfolgen. Da diese Kontrollstruktur-Informationen bisher ausschließlich im PARSE-Graphen gespeichert und noch keine Ontologierepräsentation entworfen wurde, muss die gemeinsame Integration nachgelagert geschehen. Daher wird von einer direkten Modifikation der Ontologie abgesehen. Deshalb muss für die Speicherung der Ergebnisse dieses Agenten eine Zwischenrepräsentation entworfen werden, welche die abgebildeten und neuen Ontologie-Elemente enthält. Ein Weg wäre es, diese Zwischenrepräsentation in die gemeinsamen Graphdatenstruktur von PARSE zu integrieren. Eine andere Möglichkeit bestünde darin, eine zweite

Ontologie ausschließlich bestehend aus den betreffenden Ontologie-Individuen zu erstellen und diese mit Referenzen auf die originale Domänen-Ontologie zu versehen. Mit beiden Zwischenrepräsentationen kann gewährleistet werden, dass die Kontrollstruktur-Agenten nach diesem Werkzeug aufgerufen und deren Informationen in einem Schritt der Nachverarbeitungskette (siehe Projekt PARSE, Kapitel 3) gemeinsam einfließen können. Da die anderen Agenten von PARSE wiederholend ihre extrahierten Informationen in der Graphdatenstruktur speichern, wird in dieser Arbeit ebenfalls die erste Variante bevorzugt.

8 Entwurf

In diesem Kapitel wird eine Lösungsstragie für die Problemstellung dieser Arbeit zur Synthese von Methodendefinitionen aus natürlichsprachlichen Äußerungen entworfen. Diese stützt sich auf die in Abschnitt 7.3 konzipierten Ansätze für Teillösungen. Weiterhin fließen auch die Erkenntnisse der Vorstudie aus Abschnitt 6.5 in den Lösungsentwurf ein. Die vorgestellten drei Teilaufgaben der Arbeit (siehe Abbildung 5.1) werden getrennt voneinander behandelt. Abbildung 8.1 stellt eine Beziehung zwischen den einzelnen Teilaufgaben der Arbeit her und illustriert einen ersten Programmablauf des zu implementierenden Agenten, der in den folgenden Abschnitten erklärt wird.

Beginnend mit der Eingabe einer Nutzereingabe wird diese in den Vorverarbeitungskomponente des Projekts PARSE (siehe Kapitel 3) mit einer seichten Sprachverarbeitung analysiert. Es folgen mögliche Ausführungen anderer Sprachverständnis-Agenten, welche die Nutzereingabe beziehungsweise die globale Graphdatenstruktur des Projekts PARSE, mit weiteren Informationen anreichern. Anschließend wird die Ausführung dieses Agent zur Synthese von Methodendefinitionen angestoßen. Hier beginnt der eigentliche Programmablauf des zu implementierenden Agenten.
Der Programmablauf wird analog zu den drei Teilaufgaben der Arbeit aus Abbildung 5.1 in drei Komponenten untergliedert, welche in Abbildung 8.1 durch drei Karteien visualisiert werden. Die erste Komponente bildet die der Lehrsequenz-Erkennung aus Abschnitt 7.3.1, welche wiederum in die dreistufige hierarchische Klassifikation unterteilt ist (siehe Abschnitt 7.3.1.2). Daraufhin folgt, mit der zweiten Komponente der Arbeit, die Teilaufgabe des Aufbaus von Methodendefinitionen aus Abschnitt 7.3.2. Zuletzt realisiert die dritte Komponente des Agenten die Ontologie-Abbildung aus Abschnitt 7.3.3, welche ebenfalls dreigeteilt ist. Anschließend kann die Ausgabe des Agenten wiederum in der Graphdatenstruktur gespeichert werden.

8.1 Lehrsequenz-Erkennung

In diesem Abschnitt wird ein Modellentwurf für die Detektion von Lehrsequenzen in textuellen Eingaben beschrieben. Mit diesem Modell sollen die einzelnen Bestandteile einer Lehrsequenz (siehe Tabelle 7.1) klassifiziert werden können. Da diese Lehrsequenz-Bestandteile in einer hierarchischen Beziehung zueinander stehen, wird für die Klassifikation ein stufenweiser Ansatz gewählt (siehe Abschnitt 7.3.1.2).
Im Folgenden werden die Entwürfe für die einzelnen Stufen der hierarchischen Klassifikation nacheinander vorgestellt. In den ersten beiden Abschnitten wird die Klassifikation

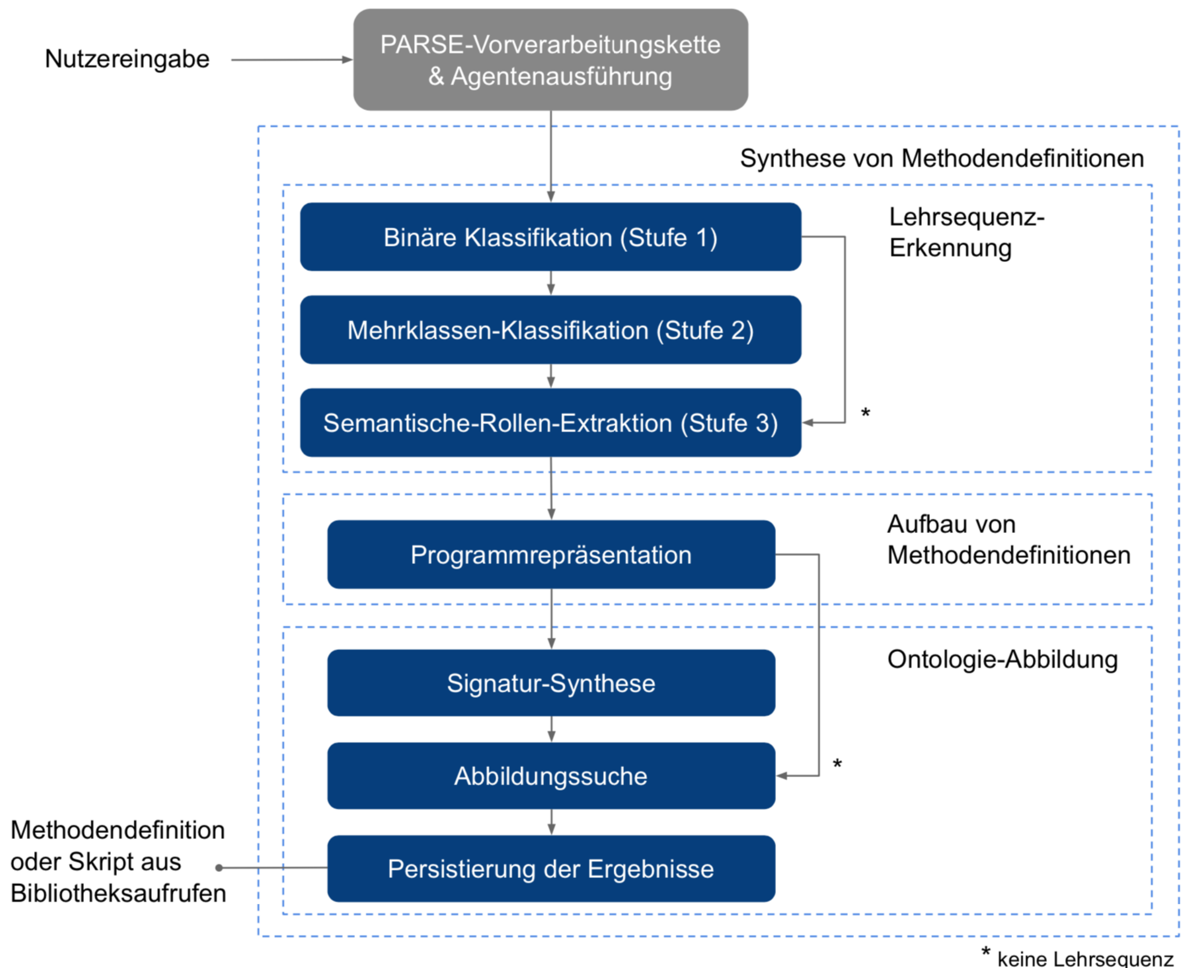


Abbildung 8.1: Programmablauf des zu entwickelnden Agenten

mit maschinellen Ansätzen vorgestellt, während im letzten Abschnitt ein Entwurf für die Klassifikation auf Basis von PARSE-Agenten für die dritte Stufe präsentiert wird.

8.1.1 Detektion von Lehrsequenzen

In der ersten Stufe der Klassifikation soll entschieden werden, ob eine Eingabesequenz ein Kommando zur Funktionserweiterung eines Zielsystems, das heißt eine Lehrsequenz, enthält (siehe Abschnitt 7.3.1). Dafür wird in Abschnitt 7.3.1.3 ein überwachter maschineller Lernansatz gewählt.

Das Training eines solchen Lernverfahrens benötigt Trainingsinstanzen, die aus der Datengrundlage generiert werden. Dafür orientiert sich diese Arbeit an dem Vorgehen nach Mihalcea. In ihrer wissenschaftlichen Arbeit [Mih07] nennt Mihalcea folgende drei Schritte zur Erstellung von Trainingsinstanzen:

1. Beschaffung und Vorverarbeitung des Korpus
2. Extraktion von Trainingsinstanzen aus dem Korpus
3. Vorverarbeitung der Trainingsinstanzen

Anschließend kann das Training des Klassifikators mit den generierten Trainingsinstanzen durchgeführt werden.

Tabelle 8.1: Erstelltes Trainingskorpus

	Eingaben
Vorstudie	3168
PARSE-Korpus	242
Gesamt	3410

Tabelle 8.2: Beispiele der Trainingskorpus-Vorverarbeitung

Wort	Ersetzung	Beschreibung
Hello	hello	Konvertierung in Kleinschreibung
don't	do not	Negationen ausschreiben
they're	they are	Abkürzungen ausschreiben
1. 2.	first second	Nummerierungen ausschreiben
1 2	eins zwei	Ziffern ausschreiben
a)	{ }	Aufzählungen löschen
!?	{ }	Satzzeichen löschen
ehm äh	{ }	Disfluenzen löschen

8.1.1.1 Beschaffung und Vorverarbeitung des Trainingskorpus

Im ersten Schritt muss der Korpus erstellt und vorverarbeitet werden. Dieser Korpus entspricht dem in der Vorstudie (siehe Kapitel 6) erhobenen Datensatz. Zusätzlich dazu werden alle vorhandenen Szenarientexte des bestehenden PARSE-Korpus als weitere Daten in das Korpus dieser Arbeit hinzugefügt. Die Zusammensetzung des vergrößerten Trainingskorpus wird in Tabelle 8.1 veranschaulicht.

Um die Texte des Trainingskorpus als Trainingsinstanzen für einen Klassifikator nutzen zu können, müssen diese vorverarbeitet werden. Eine Vorverarbeitung bereinigt und formatiert die Eingabetexte einheitlich. Die Nutzereingaben der Vorstudie sind dabei bereits von Schreibfehlern bereinigt, auftretende Grammatikfehler werden jedoch nicht behoben. In einem ersten Schritt werden zunächst Zeilenumbrüche und Sonderzeichen entfernt sowie Ziffern in Zahlwörter übersetzt. Weiterhin werden die englischen Abkürzungen für Negationen und Hilfsverben durch ihre Kontraktion, das heißt dem lautlichen Zusammenziehen zweier Wörter, ersetzt. Da der hiesige Ansatz für gesprochene Sprache entworfen wird, müssen auch die Satzzeichen entfernt werden. Dies wurde in Abschnitt 7.1 als Herausforderung (H 1.2) definiert. Zuletzt werden die im PARSE-Korpus vorkommenden Wortwiederholungen (*Reparationen*) und Verzögerungslaute (*Disfluenzen*) entfernt. Zu diesen Vorverarbeitungsschritten wird in Tabelle 8.2 jeweils ein Beispiel präsentiert, wobei das Symbol { } die Ersetzung mit der leeren Menge, das heißt, das Löschen eines Wortes, angibt.

8.1.1.2 Extraktion von Trainingsinstanzen

Nachdem durch die Vorverarbeitung des Korpus bereinigte Fließtexte erzeugt werden, müssen für das Training des Klassifikators anschließend Trainingsinstanzen extrahiert werden. Diese Trainingsinstanzen enthalten bei der Verwendung von überwachten Lernverfahren neben der Eingabe auch entsprechende Ausgabeklassen (engl. *label*), den Goldstandard. Überdies kann die Eingabe einer Trainingsinstanz zusätzlich zu den Fließtext-Sequenzen mit möglichen weiteren Merkmalen angereichert werden.

In der ersten Stufe der Klassifikation soll ermöglicht werden, bei einer gegebenen Eingabe zu bestimmen, ob sie einer Lehrsequenz entspricht. Falls nicht, ist die Eingabe der Gruppe der Befehlssequenzen (siehe Definition 6.5.1) zuzuordnen. Das Ergebnis entspricht daher einer binären Klassifikation pro Eingabesequenz, das heißt, es wird sequenzweise klassi-

fiziert.
Als Eingabesequenzen dienen alle im vorherigen Schritt vorverarbeiteten Trainingskorpustexte. Weitere Merkmale werden den Trainingsinstanzen dieser Klassifikationsstufe nicht hinzugefügt. Das binäre Entscheidungsproblem von Stufe 1 entspricht einer Klassifikation mit zwei Klassen. Aufgrund der sequenzweisen Klassifikation muss für den Goldstandard jeder Eingabesequenz eine Ausgabeklasse zugeordnet werden.
Die Trainingsinstanzen für die erste Stufe der Klassifikation werden in Definition 8.1.1 zusammengefasst.

Definition 8.1.1: Stufe 1 - Trainingsinstanzen

- Eingabesequenzen: gesamtes Trainingskorpus
- Ausgabeklassen: sequenzweise
  - Lehrsequenz TS
  - Befehlssequenz NO_TS

Für die Erstellung des Goldstandard wird der gesamte Korpus betrachtet. Dabei wird wie folgt vorgegangen: Enthält eine Eingabesequenz die in der Vorstudie geforderten Lehrsequenz-Anforderungen aus Definition **??**, wird sie der Klasse der Lehrsequenz (TS) zugewiesen. Zu diesen Anforderungen gehört zum Einen der Ausdruck, dass der Nutzer dem Zielsystem neues Wissen lehren möchte, zum Anderen einen Namen für das neue Kommando und zuletzt eine Beschreibung der zugehörigen Teilschritte. Trifft eine dieser Anforderungen nicht zu, wird eine Eingabesequenz als Befehlssequenz (NO_TS) markiert. Somit ergibt sich ein Goldstandard für Stufe 1, der pro Eingabesequenz eine Markierung aufweist. Beispiel 8.1.1 soll diesen händischen Annotationsschritt an exemplarischen Eingaben veranschaulichen.

Beispiel 8.1.1: Stufe 1 - Annotierte Eingaben

E1:
[armar coffee is a beverage that people like to drink in order to make coffee you have to locate the cups next to the machine put one cup under the dispenser and lastly press the red button on the coffee machine $]_{\textbf{TS}}$

E2:
[to greet a person look into the persons eyes wave your robot hand and say hello this is how you greet someone $]_{\textbf{TS}}$

E3:
[first find a cup next to the machine then place the cup under the dispenser last you have to push the red button $]_{\textbf{NO_TS}}$

E4:
[hi armar I want coffee to do this you need to put a cup unter the dispenser and press the button $]_{\textbf{NO_TS}}$

E5:
[when I enter the room greet me by saying hello $]_{\textbf{NO_TS}}$

Weiterhin ging durch die Auswertung der Vorstudie in Abschnitt 6.5.2 hervor, dass die Zuordnung von Eingaben zur Gruppe der Lehrsequenzen beziehungsweise der Befehlssequenzen nicht immer eindeutig möglich ist. Diese Schwierigkeit wird in den beschriebenen

Tabelle 8.3: Verteilung der Stufe-1-Markierungen

	Trainingsinstanzen
Lehrsequenzen (TS)	1988
Befehlssequenzen (NO_TS)	1422
Gesamt	3410

Herausforderungen der Teilaufgabe der Lehrsequenz-Detektion in Abschnitt 7.1 als (H 1.4) gelistet. Bezogen auf die Eingaben im obigen Beispiel 8.1.1 betrifft dies beispielsweise die Eingabe E4. Hier kann nicht klar gedeutet werden, ob der Nutzer diese Äußerung mit dem Ziel zur permanenten Funktionserweiterung (Lehrsequenz) oder zur sofortigen Ausführung (Skript) an den Roboter diktiert. Auch bedingte Äußerungen wie in Eingabe E5 lassen offen, ob sie einmalig erfüllt oder zukünftig abgespeichert werden soll. Für das Training des Klassifikators muss jede Trainingsinstanz jedoch mit einer eindeutigen Ausgabe-Markierung versehen werden. Daher müssen auch jene unklaren Fälle einer Ausgabeklasse im Goldstandard zugeordnet werden. Es wird einen Ansatz gewählt, welcher bei Unklarheit der korrekten Interpretation die betrachtete Eingabesequenz der Klasse der Befehlssequenzen zuordnet. Bezugnehmend auf das Beispiel 8.1.1 trifft dies auf die Eingabesequenzen E3 und E4 zu.

Schlussendlich ergeben sich für die Stufe-1-Klassifikation insgesamt 3410 Trainingsinstanzen. Die Klasse der Lehrsequenzen (TS) ist dabei mit 1988 Eingaben 58% anteilig im Trainingskorpus etwas höher vertreten als die Klasse der Befehlssequenzen mit 42%. Diese Daten werden in Tabelle 8.3 veranschaulicht.

8.1.1.3 Vorverarbeitung der Trainingsinstanzen

Im dritten Schritt zur Generierung von Trainingsinstanzen nach Mihalcea [Mih07] müssen die Eingaben und Ausgaben für den Klassifikator verständlich auf numerische Werte oder Vektoren abgebildet werden. Zunächst wird die Kodierung der Eingabemerkmale betrachtet. Bei Einsatz von maschinellem Lernen werden textuelle Eingaben üblicherweise als Menge von Vektoren dargestellt. Eine dieser Vektordarstellungen von Texteingaben ist das Bag-Of-Words-Modell [GH17]. Diese wird anhand eines Beispiels im Grundlagenkapitel in Abschnitt 2.4.3.1 erläutert. Eine Möglichkeit zur Vektorrepräsentation von Wörtern mit dem Bag-Of-Words-Modell besteht darin, die Anzahl der Vorkommen eines Wortes pro Eingabesequenz in diesem Vektor zu kodieren. Eine weitere ist die Kodierung der Wörter nach dem Tf-idf-Maß. Der Einsatz von Tf-idf-Gewichtungen wird jedoch in dieser Arbeit nicht verfolgt. Durch den Einsatz von Tf-idf-Gewichtungen wäre es möglich, dass wichtige Satzbestandteile, welche im Normalfall in die Kategorie der Stoppwörter eingeordnet fallen, eine geringe, irreführende Gewichtung erhalten könnten. Überdies ist die Deutung der Tf-idf-gewichteten Eingaben unklar, da die Datengrundlage dieser Arbeit auf einer Menge von ähnlichen Szenarientexten beruht. Die inverse Häufigkeit eines Wortes über diese Menge verwandter Texte ist daher nicht aussagekräftig. Deshalb wird die erste Variante gewählt. Zusammenfassend ist Bag-Of-Words-Modell eine sehr einfache Repräsentationsform, da ausschließlich die Häufigkeiten eines Wortes in der Eingabe betrachtet werden. Für eine binäre Klassifikation könnte dieser Ansatz jedoch ausreichen, da vom Modell möglicherweise einzelne Signalwörter in den Eingabesequenzen für die Entscheidung erlernt werden. Pro Eingabe verarbeitet mit dem Bag-Of-Words-Modell erhält der Klassifikator somit einen einzelnen Vektor, wobei die Vektordimension der Länge des Vokabulars entspricht. Ein Beispiel dafür wurde in den Grundlagen unter Abschnitt 2.4.3.1 präsentiert.

Als zweite Repräsentation von Eingabetexten werden trainierte Wortvektoren (siehe Abschnitt 2.4.3.2) verwendet. Im Unterschied zum Bag-Of-Words-Modell wird eine Eingabesequenz mit Wortvektoren nicht als Ganzes, sondern jedes Wort einzeln kodiert. Weiterhin

können statt der Häufigkeit des Wortes in der Eingabe mit Wortvektoren komplexere Informationen über das Wort selbst, wie beispielsweise Wortarten oder die Wortumgebungen, repräsentiert werden. Wortvektoren verfolgen das Ziel, die semantische Bedeutung eines Wortes auf einen hochdimensionalen geometrischen Zahlenraum abzubilden. Diese Wortvektoren werden daher auch *Word Embeddings* genannt und werden auf dem Vokabular trainiert. Ein anderer Ansatz ist es, bereits vortrainierte Wortvektor-Modelle zu nutzen. Letztere haben den Vorteil, auf einem sehr großen Vokabular erzeugt worden zu sein und somit tiefere semantische Informationen modellieren zu können. Für diese Arbeit werden die vortrainierten Wortvektoren von Facebook Research [1] namens *fasttext* gewählt. Diese haben neben anderen Wortvektor-Modellen, wie beispielsweise Stanford's GloVe oder Google's Word2Vec, einen weiteren Vorteil, da sie statt den Vektor eines Wortes direkt zu lernen, jedes Wort als N-Gramm (siehe Abschnitt 2.2.1.4) von Buchstaben kodieren. Dies ermöglicht die zusätzliche Erfassung von Bedeutungen kurzer Wörter, Suffixe und Präfixe. Mit diesem Ansatz erhält der Klassifikator somit für jedes Wort der Eingabesequenz einen Wortvektor. Die Länge der Vektoren muss beim manuellen Training selbst angegeben werden. Bei den vortrainierten fasttext-Vektoren ist die Länge auf 300 Dimensionen festgelegt.

Zuletzt müssen die Ausgabeklassen auf numerische Werte abgebildet werden. Dazu werden im binären Fall die beiden Klassen durch die Zahlen 0 und 1 repräsentiert.

8.1.1.4 Training des Klassifikators

Für das binäre Klassifikationsproblem von Stufe 1 wird, wie in Abschnitt 7.3.1.3 beschrieben, ein überwachter maschineller Lernansatz verfolgt. Dafür sollen verschiedene Modelle erstellt und die Ergebnisse miteinander verglichen werden. Ziel der Modellbildung ist es, dass für das binäre Klassifikationsproblem am besten passendste Modell zu finden. Das bedeutet, dass die Güte des trainierten Modells auf unbekannten Daten maximiert werden soll (Herausforderung (H 1.5) in Abschnitt 7.1). Dafür werden als Metriken die Genauigkeit, Präzision, Ausbeute und das F_1-Maß berechnet (siehe Abschnitt 2.3).

Im Zuge der Modellbildung soll die binäre Klassifikation zunächst mittels klassischem maschinellen Lernen, auch symbolische Verfahren genannt, implementiert werden (siehe Abschnitt 2.4.1). Es wird ein Portfolio aus verschiedenen Lernverfahren betrachtet, welche im Grundlagenkapitel in Abschnitt 2.4.1 kurz vorgestellt werden.

- Entscheidungsbaum (engl. *decision tree*, Abschnitt 2.4.1.1)
- Random Forest (engl. *random forest*, Abschnitt 2.4.1.2)
- Naïve Bayes (engl. *naive Bayes*, Abschnitt 2.4.1.5)
- Logistische Regression (engl. *logistic regression*, Abschnitt 2.4.1.3)
- Stützvektormaschine (engl. *support vector machine*, Abschnitt 2.4.1.4)

Weiterführend sollen neuronale Netze als nicht-symbolische Verfahren maschinellen Lernens für die Klassifikation genutzt werden. Dafür sollen wiederum verschiedene Modellarchitekturen für die Anwendung auf das binäre Klassifikationsproblem untersucht werden. Im Grundlagenkapitel in Abschnitt 2.4.2 werden diese neuronalen Ansätze ebenfalls kurz vorgestellt. Auf die Einzelheiten der untersuchten Architekturen und Parameter der eingesetzten Modelle wird im Folgenden Kapitel 9 der Implementierung eingegangen.

- Künstliches neuronales Netz (engl. *artificial neural network, ANN*, Abschnitt 2.4.2.1)
- Faltendes neuronales Netz (engl. *convolutional neural network, CNN*, Abschnitt 2.4.2.4)

[1] Quelle: `https://github.com/facebookresearch/fastText`, zuletzt besucht am 17.06.19

- Neuronales Netz mit langem Kurzzeitgedächtnis (engl. *long short term memory network, LSTM*, Abschnitt 2.4.2.2)
- Geschaltete rekurrente Einheit (engl. *gated recurrent unit, GRU*, Abschnitt 2.4.2.3)

Als weiteren Modellansatz soll das Werkzeug *BERT* [DCLT18a] von Google auf dem binären Klassifikationsproblem angewendet. Es handelt sich hierbei um ein von Google auf großen, nicht annotierten Datenmengen trainiertes Sprachmodel (engl. *language model*), das unter Anpassung von über 350 Millionen Modellparametern eines tiefen neuronalen Modells trainiert wurde (siehe Abschnitt 2.4.2.5). Dieses erweitert den Einsatz von vortrainierten Wortvektoren um ein zusätzliches Verständnis über Kontext und Sprachmodellierung. Anschließend kann das vortrainierte Modell in einem weniger ressourcenintensiven Anpassungsschritt (engl. *fine-tuning*) auf spezifische Klassifikationsprobleme mittels annotierten Daten angepasst werden.

Als Eingabe für die Klassifikationsmodelle dienen die zuvor generierten Trainingsinstanzen. Die für die Lernphase des Modells verwendeten Trainingsinstanzen werden als Trainingsdaten bezeichnet. Sie haben Einfluss auf die Gewichte des Klassifikators. Um die Güte verschiedener Modelle vergleichen zu können, muss neben den Trainingsdaten auch ein Anteil der Trainingsinstanzen für die Evaluierung des Modells zurückgehalten (engl. *holdout set*, [JWHT14]) werden. Die sogenannte Testmenge darf das Modell vorab im Training nicht gesehen haben. Auf diese Weise kann die Güte der Generalisierung, das heißt, die Fähigkeit des Modells zur Verallgemeinerung auf unbekannten Daten, evaluiert werden. Deshalb werden die Trainingsinstanzen in Trainingsdaten und Testdaten (engl. *train-test-split*) unterteilt. Die Generalisierung ist eine der in Abschnitt 7.1 genannten Herausforderungen (H 1.5), welche die Implementierung des binären Klassifikationsansatzes meistern muss.
Der Modellentwurf ist jedoch ein iterativer Prozess (siehe Abschnitt 7.3.1.3). Deshalb werden in dieser Arbeit viele verschiedene Lernverfahren auf das Problem angewandt und gegeneinander evaluiert.

Die erzielten Ergebnisse des trainierten Modells müssen kontinuierlich beobachtet werden, um den Einfluss von manueller Änderung verschiedener Parameter oder der Modellarchitektur (bei neuronalen Netzen) abzuschätzen. Auf diese Weise wird der Klassifikator auf das Problem angepasst. Würde diese Parameteroptimierung mit dem finalen Testdatensatz geschehen, wären die Resultate verfälscht, da der Klassifikator iterativ an die Testdaten angepasst werden würde. Dieses Verhalten beschreiben auch Russel et al. in ihrem Buch „Artificial Intelligence: A Modern Approach“, [RN09]. Daher wird ein dritter Datensatz, die Entwicklungs- oder auch Validierungsdaten genannt, aus den Trainingsinstanzen entnommen. Diese Validierungsdaten werden für das Anpassen der Parameter (engl. *tuning*), wie beispielsweise der Baumtiefe bei Entscheidungsbäumen oder die Anzahl versteckter Neuronen (engl. *hidden units*) bei neuronalen Netzen, genutzt. Der Testdatensatz ist dem trainierten Modell bis zur finalen Evaluation unbekannt. Das Modell kann somit unbeeinflusst (engl. *unbiased*) evaluiert werden. Es wird dabei häufig ein Verhältnis von 80:20 von Trainings- zu Testdaten nach dem Pareto-Prinzip angegeben. Von diesen Trainingsdaten werden anschließend weitere 20% für die Validierungsdaten empfohlen [LL99], [Col02]. Es ergibt sich ein Gesamtverhältnis von 64:20:16 der Train-Test-Validate-Aufteilung. Nachfolgend werden diese drei Datensätze mit Train, Test und Validate abgekürzt.
Statt einer Dreiteilung der Trainingsinstanzen ist auch der Einsatz von Kreuzvalidierung sehr verbreitet. Dabei wird die jeweilige Zuweisung der Datenteilungen in Test-, Trainings- und Validierungsdaten rotiert. Es werden daher mehrere Durchläufe benötigt. Auf diese Weise kann eine Aussage über die Güte auf unterschiedlichen Datensatztrennungen getroffen werden. Jedoch muss für jeden dieser Aufteilungen das Modell erneut trainiert und evaluiert werden, was die Kreuzvalidierung zu einem sehr zeitintensiven Vorgehen macht.

Wichtig ist, dass die zwei beziehungsweise drei aus den Trainingsinstanzen generierten Datensätze die Eigenschaften der Domäne jeweils gut repräsentieren. Sie sollten der gleichen Verteilung zugrunde liegen, um eine unbeeinflusste Aussage über die ungesehenen Testdaten – stellvertretend für die reale Welt – treffen zu können.
Für die Implementierung des Klassifikators werden wiederum unterschiedliche Train-Test-Validate-Aufteilungen untersucht:

Aufteilung 1: randomisiert

- Train: 64% aus randomisierten Trainingsinstanzen
- Test: 20% aus randomisierten Trainingsinstanzen
- Validate: 16% aus randomisierten Trainingsinstanzen

Aufteilung 2: szenarienbezogen

- Test: ca. 23% Trainingsinstanzen aus einem Szenario[2]
- Train: 77% aus randomisierten Trainingsinstanzen der restlichen drei Szenarien
- Validate: 20% aus randomisierten Trainingsinstanzen der restlichen drei Szenarien

Für die erste Aufteilung werden die Trainingsinstanzen randomisiert in Train-, Test- und Validate-Datensätze unterteilt. In Aufteilung 2 geschieht diese Trennung szenarienbezogen, das heißt, es werden die vier in der Vorstudie in Abschnitt 6.3 definierten, unterschiedlichen Nutzeraufgaben („Szenarien“) zur Datenerfassung getrennt betrachtet. Dafür wird eines der Szenarien gänzlich als Testdatensatz verwendet. Der Trainings- und Validierungsdatensatz wird anschließend aus den verbliebenen drei Szenarien randomisiert gebildet. Da die vier Szenarien verschiedene Aufgabenstellungen an den Nutzer stellen, konnte innerhalb der Nutzerstudie somit eine Gewinnung möglichst unterschiedlicher Daten ermöglicht werden. Wird nun einer dieser Szenarien im Testdatensatz isoliert, hat dies zur Folge, dass das „Thema“ der Testdaten disjunkt zu denen der Trainings- und Validierungsdaten ist. Damit lässt sich anschließend prüfen, ob der Klassifikator bei guten Testresultaten nicht nur auf vorher ungesehenen Daten, sondern auch mit Daten einer komplett neuen Thematik umgehen kann. Dies unterstreicht die Robustheit des Klassifikators, erschwert das Klassifikationsproblem jedoch zusätzlich. Die Implementierung des beschriebenen Klassifikatoransatzes der ersten Stufe wird im nachfolgenden Implementierungs-Kapitel 9 präsentiert.

8.1.2 Detektion von Deklaration und Beschreibung

In der zweiten Stufe der Klassifikation soll kategorisiert werden, welcher Bestandteil einer bereits erkannten Lehrsequenz zur Kommandodeklaration und welcher zur Kommandobeschreibung (siehe Tabelle 7.1) gehört. Dafür wird in Abschnitt 7.3.1.3 ebenfalls ein überwachter maschineller Lernansatz gewählt. Anschließend wird die Generierung von Trainingsinstanzen, analog zum beschriebenen Vorgehen nach Mihalcea [Mih07] in der ersten Stufe der Klassifikation (Abschnitt 8.1.1), durchgeführt.
Wurde in der binären Klassifikation der ersten Stufe keine Lehrsequenz erkannt, so wird dieser Schritt zur Detektion von Deklaration und Beschreibung übersprungen. Dieser Umstand veranschaulicht der Programmablauf des Agenten in Abbildung 8.1 in der ersten Kartei zu Teilaufgabe eins dieser Arbeit: Bei Detektion einer Lehrsequenz, wird mit der zweiten Stufe der Klassifikation fortgeführt, was durch die Kante ausgehend vom ersten zum zweiten Block verdeutlicht wird. Wird dagegen eine Befehlssequenz erkannt, führt

[2] Die Nutzereingaben eines Szenarios entsprechen etwa einem Viertel des Vorstudiendatensatzes. Nach Hinzunahme des PARSE-Datensatzes ist der Anteil etwas geringer.)

eine weitere Kante vom ersten Block direkt zur dritten Stufe der Klassifikation, die zweite Klassifikationsstufe wird folglich übersprungen.

8.1.2.1 Beschaffung und Vorverarbeitung des Trainingskorpus

Für die Generierung von Trainingsinstanzen muss das betrachtete Korpus vorverarbeitet werden. Da dieser Vorverarbeitungsschritt unabhängig zu den bestehenden Klassifikationsproblemen ist, kann das vorverarbeitete Trainingskorpus der Stufe 1 aus Abschnitt 8.1.1.1 für die zweite Stufe der Klassifikation wiederverwendet werden.

8.1.2.2 Extraktion von Trainingsinstanzen

Die zweite Klassifikationsstufe soll die in ein einer Lehrsequenz befindlichen Satzbestandteile der Kommandodeklaration sowie der Kommandobeschreibung klassifizieren. Da dem konzipierten Lösungsansatz für die Detektion von Lehrsequenzen eine hierarchischer Ansatz zugrunde liegt, folgt daraus, dass die Ergebnisse der vorherigen Klassifikationsstufe in die darauffolgende miteinfließen.
Für das in Stufe 2 betrachtete Klassifikationsproblem sind jedoch nur die in Stufe 1 als Lehrsequenz markierten Eingabesequenzen relevant. Diese Beziehung wird in Abbildung 7.1 veranschaulicht. Deshalb wird als Eingabesequenz für die Trainingsinstanzen dieser Stufe die Menge aller als Lehrsequenz (Stufe 1, Klasse TS) klassifizierten Eingaben verwendet. Bezogen auf das präsentierte Beispiel 8.1.1 der vorherigen Klassifikationsstufe, betrifft dies nur die Eingaben E1 und E2. Für die Erstellung der Ausgabeklassen wird das Entscheidungsproblem der Stufe 2 näher betrachtet: Im Vergleich zur vorherigen Stufe entspricht die Problemstellung keiner binären Klassifikation. Die linguistische Auswertung der Vorstudie in Abschnitt 6.5.2 hat gezeigt, dass eine Nutzereingabe neben den betrachteten Imperativsätzen auch Aussagesätze, Fragesätze und Ausrufesätze enthalten kann, wobei die Unterschiede dieser Satztypen in Abschnitt 2.1.4 verdeutlicht werden. Daher wird neben den Klassen für die Kommandodeklaration und Kommandobeschreibung eine weitere Klasse für sonstige Äußerungen benötigt. Das Entscheidungsproblem ist deshalb eine Mehrklassen-Klassifikation mit drei Klassen. Zudem wird die Eingabesequenz nicht mehr als Ganzes, sondern jedes Wort und dessen Zuordnung zu den Klassen einzeln betrachtet. Deshalb wird wortweise markiert. Diese Problemstellung wird in der Praxis auch als Sequenzmarkierungsverfahren (engl. *sequence labeling*) bezeichnet.
Die Extraktion der Trainingsinstanzen der zweiten Klassifikationsstufe wird in Definition 8.1.2 zusammengefasst. Eine Trainingsinstanz entspricht hierbei einer Nutzereingabe, wobei bei Betrachtung des Sequenzmarkierungsproblems der zweiten Klassifikationsstufe genau genommen die einzelnen Wörter einer Nutzereingabe einzeln betrachtet werden. Der definierte Begriff einer Trainingsinstanz wird jedoch weiterhin für die Beschreibung einer Eingabesequenz inklusive der annotierten Ausgabeklassen verwendet.

Definition 8.1.2: Stufe 2 - Trainingsinstanzen

- Eingabesequenzen: Lehrsequenzen (Klasse TS) aus Stufe 1
- Ausgabeklassen: wortweise
  - Kommandodeklaration DECL
  - Kommandobeschreibung SPEC
  - Sonstige Äußerungen ELSE

Die benötigten Ausgabeklassen für das Training des Klassifikators werden wiederum händisch erstellt. Dabei wird wie folgt vorgegangen: Alle in Stufe 2 betrachteten Eingabesequenzen wurden in Stufe 1 als Lehrsequenz (TS) klassifiziert. Daraus folgt, dass die

Nutzereingabe eine Funktionserweiterung des Zielsystems beschreibt. Die Eingabesequenz enthält somit eine Beschreibung einer neuen Funktion sowie den Ausdruck, dass der Nutzer neues Wissen vermitteln möchte. Letzteres wurde in der Auswertung der Vorstudie als Floskel bezeichnet. Für die Ausgabeklassen der Stufe 2 bedeutet dies, dass der Satzbestandteil der Eingabesequenz, welcher den Wunsch zur Funktionserweiterung sowie den Namen der neuen Funktionalität beinhaltet, als Kommandodeklaration (DESC) markiert wird. Der Satzbestandteil, welcher die Teilschritte der neuen Funktion beschreibt, wird der Klasse der Kommandobeschreibung (SPEC) zugewiesen. Die restlichen Ausdrücke der Eingabesequenz, welche noch keine Zuordnung besitzen, werden mit der Klasse Sonstige Äußerung (ELSE) annotiert. Somit ergibt sich ein Goldstandard von Stufe-2-Ausgabeklassen, der pro Eingabesequenz aus mehreren zusammenhängenden Blöcken aus einheitlichen Markierungen besteht, wobei mindestens ein DECL-Block und ein SPEC-Block vertreten ist. Der beschriebene Annotationsschritt für Stufe 2 soll anhand von Beispiel 8.1.2 verdeutlicht werden.

Beispiel 8.1.2: Stufe 2 - Annotierte Lehrsequenzen

E1:
[armar coffee is a beverage that people like to drink]$_{\textbf{ELSE}}$ [in order to make coffee you have to]$_{\textbf{DECL}}$ [locate the cups next to the machine put one cup under the dispenser and lastly press the red button on the coffee machine]$_{\textbf{SPEC}}$

E2:
[to greet a person]$_{\textbf{DECL}}$ [look into the persons eyes wave your robot hand and say hello]$_{\textbf{SPEC}}$ [this is how you greet someone]$_{\textbf{DECL}}$

Es ergeben sich für die Stufe-2-Klassifikation 1988 Trainingsinstanzen mit insgesamt 74934 Wörtern. Durch den hierarchischen Klassifikationsansatz und die damit verbundene Verwendung einer Teilmenge der Stufe-1-Ausgaben (hier Klasse TS), dezimiert sich die Menge der verfügbaren Trainingsinstanzen für Stufe 2. So sind statt 3410 Trainingsinstanzen in Stufe 1 nur 1988 Trainingsinstanzen für die zweite Klassifikationsstufe verfügbar. Betrachtet man die Verteilung der einzelnen Klassen von Stufe 2, so fällt auf, dass die Klasse der Kommandobeschreibungen (SPEC) mit 76% sehr hoch im Vergleich zu den anderen beiden Klassen mit 21% und 3% vertreten ist. Diese ungleiche Verteilung der Klassen muss in der Implementierung mit einer besonderen Behandlung beachtet werden, da diese oft mit Problemen einhergehen. Eines davon ist das einseitige Training des Klassifikators auf den Beispielen, die mit einer Mehrheit vertreten sind, ein anderes ist das Genauigkeits-Paradoxon (engl. *accuracy paradox*), bei dem sehr gute Genauigkeitswerte trotz einer Menge an Fehlklassifikationen erreicht werden können. Auf diese Probleme wird im nachfolgenden Kapitel der Implementierung detaillierter eingegangen (siehe Abschnitt 9.1.2.3). Diese Schwierigkeit ist Herausforderung (H 1.7) in Abschnitt 7.1 gelistet. Tabelle 8.4 illustriert diese Verteilung der Trainingsinstanzen für Stufe 2.

8.1.2.3 Vorverarbeitung der Trainingsinstanzen

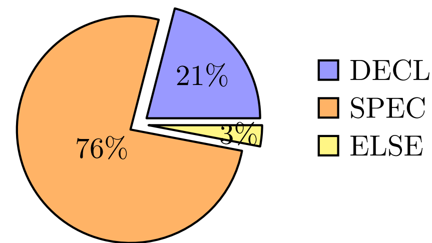
Für die Verwendung von textuellen Merkmalen als Eingabe für einen Klassifikator, müssen diese wie bereits in Stufe 1 in Abschnitt 8.1.1.3 beschrieben, auf numerische Werte abgebildet werden.

Aufgrund des komplexeren Klassifikationsproblems der zweiten Stufe, wird auf die Textrepräsentation mittels dem Bag-Of-Words-Modell verzichtet. Stattdessen werden ausschließlich die vortrainierten Worteinbettungen (engl. *word embeddings*) von Facebook fast-

[3]Betrachtet werden alle Stufe-2-Eingaben, es kommt jedoch nicht in allen eine ELSE-Markierung vor.

Tabelle 8.4: Verteilung der Stufe-2-Markierungen

	Trainingsinstanzen	Wörter
Stufe 1: Gesamt	3410	119948
Stufe 1: NO_TS	1422	45014
Stufe 1: TS	1988	74934
Stufe 2: Gesamt	1988	74934
Stufe 2: DECL	1988	15559
Stufe 2: SPEC	1988	57156
Stufe 2: ELSE	1988[3]	2219



text verwendet. Aufgrund des selben Vokabulars der Trainingsinstanzen muss keine weitere Anpassung dieser Wortvektorrepräsentation vorgenommen werden.

Zuletzt müssen die Ausgabeklassen auf numerische Werte abgebildet werden. Dazu werden die Klassen durchnummeriert als Zahl repräsentiert. Die Ausgabe einer Trainingsinstanz besteht anschließend aus einer Liste von Zahlen gleicher Länge wie die Anzahl der Wörter in der Eingabesequenz.

8.1.2.4 Training des Klassifikators

Für die Lösung des Mehrklassen-Klassifikationsproblems von Stufe 2 wird ebenfalls ein überwachter, maschineller Lernansatz verfolgt. Dabei sollen verschiedene Modellarchitekturen für einen Klassifikator erstellt und deren Ergebnisse gegeneinander evaluiert werden. Ziel der Modellbildung ist es wiederum, ein auf das Entscheidungsproblem angepassten Klassifikator zu trainieren. Dieser soll anschließend fähig sein, auf ungesehenen Daten Klassifikationsergebnisse zu erzeugen, die sich nicht signifikant von der Klassifikationsgüte auf den Validierungsdaten unterscheiden. Diese Fähigkeit der Klassifikators wird als Generalisierung bezeichnet, welche auch bereits mit der Herausforderung (H 1.6) in Abschnitt 7.1 definiert wurde.

Als Modelle für das Mehrklassen-Sequenzmarkierungs-Problem sollen wiederum verschiedene neuronale Netzarchitekturen getestet werden. Deshalb werden, wie auch schon im Entwurf der ersten Klassifikationsstufe in Abschnitt 8.1.1.4, künstliche neuronale Netzarchitekturen wie ANNs, LSTMs und GRUs (siehe Abschnitt 2.4.2) betrachtet. In der Forschung hat sich gezeigt, dass insbesondere bidirektionale Ansätze gute Ergebnisse bei Sequenzmarkierungsproblemen erzielen [Man11], [CWB+11], [SH16]. Daher sollen diese in der Implementierung des Klassifikators gezielt eingesetzt werden. Weiterhin soll, wie schon im binären Fall, das Werkzeug *BERT* [DCLT18a] (siehe Abschnitt 2.4.2.5) auf dem Mehrklassen-Klassifikationsproblem angewendet werden.

Als Eingabe für die verschiedenen Verfahren des maschinellen Lernens dienen die zuvor generierten Trainingsinstanzen. Anschließend folgt die Test-Train-Validate-Trennung der Trainingsinstanzen analog zur ersten Stufe der Klassifikation in die randomisierte und die szenarienbezogene Aufteilung 1 und 2, beschrieben in Abschnitt 8.1.1.4.

Die Implementierung des beschriebenen Klassifikatoransatzes der zweiten Stufe wird in Kapitel 9 vorgestellt.

8.1.3 Detektion von Aktionsbenennungen

In der dritten Stufe der Klassifikation sollen die in der Eingabesequenz enthaltenen Aktionsnamen und zugehörige Entitäten, die sogenannten Aktionsbenennungen (siehe Definition 5.1.4) kategorisiert werden. Damit entspricht Entscheidungsproblem der dritten Stufe wiederum einer wortweisen Mehrklassen-Klassifikation. Es wird ebenfalls ein Klassifikationsansatz entworfen, jedoch wird im Gegensatz zu den vorherigen Klassifikationsstufen

kein maschineller Lernansatz auf Basis annotierter Daten verfolgt. Stattdessen wird ein heuristischer Ansatz verfolgt, der die Ergebnisse des bestehenden Semantischen-Rollen-Erkenners verwendet.

8.1.3.1 Extraktion von Klassifikationsdaten

Die dritte Klassifikationsstufe soll die in einer Lehrsequenz befindlichen Aktionen und Entitäten klassifizieren. Dieses Vorgehen wurde in Abschnitt 7.3.1.4 erläutert. Als Grundlage dieser Klassifikation dienen semantische Rollen, welche die Beziehungen zwischen Aktionen und zugehörigen Entitäten beschreiben. Ein Beispiel für diese semantische Rollen-Markierungen wird in den Grundlagen in Abschnitt 2.2.1.5 präsentiert. Diese werden durch den Semantischen-Rollen-Erkenner von Projekt PARSE, kurz SRL-Agenten, bereitgestellt.

Der SRL-Agent ermittelt die semantischen Rollen auf Basis einer Eingabesequenz. Bezogen auf den hierarchischen Klassifikationsansatz dieser Teilaufgabe gibt zwei Möglichkeiten, diese Rollen für aus der Eingabesequenz zu gewinnen. Eine Variante wäre es, die Klassifikationsergebnisse aus der vorherigen Stufe wiederzuverwenden und sie als Eingabe für den Semantischen-Rollen-Erkenner zu nutzen. Dies entspräche einem weiteren hierarchischen Vorgehen der Klassifikation. Als Eingabe würden in diesem Fall die Satzteile der Nutzereingabe, unterteilt in die zuvor klassifizierten Lehrsequenz-Bestandteile der Kommandodeklaration (Klasse DECL), Kommandobeschreibung (Klasse SPEC) und sonstigen Äußerungen (ELSE) dienen. Beispielsweise würden jene Wörter der Nutzereingabe, die in der zweiten Stufe der Klassifikation als Kommandodeklaration klassifiziert wurden, als separate Eingabe für den Semantischen-Rollen-Erkenner verwendet werden. Der SRL-Agent würde seine Aktions- und Entitätenannotationen nur auf diesen Satzteilen der Klasse DECL detektieren. Analog könnte der als Kommandobeschreibung klassifizierte Satzteil der Nutzereingabe mit dem SRL-Agenten betrachtet und wiederum separat klassifiziert werden. Die unterschiedlichen Stufe-2-klassifizierten Satzbestandteile der Nutzereingabe könnten somit getrennt voneinander mit dem SRL-Agenten untersucht werden. Ein solcher Lehrsequenz-Bestandteil besteht jedoch nicht aus einem einzigen zusammenhängenden Block von Satzbestandteilen. Aus der Analyse der Vorstudie ging hervor, dass mehrere getrennte Blöcke auf Sprachebene existieren können. Diese Aufspaltung der Eingabesequenz wird durch Fehlklassifikationen der einzelnen Wörter in der zweiten Stufe zusätzlich verstärkt. Im Optimalfall bräuchte man mehrere zusammenhängende Blöcke einer der jeweiligen Klassen (DECL, SPEC, ELSE), die daraufhin nach ihren zugrundeliegenden semantischen Rollen untersucht werden könnten. Es muss jedoch auch mit fehlerbehafteten Klassifikationsresultaten gerechnet werden. Eine Detektion solcher schachbrettartiger Klassifikationsausgaben wäre mit dem Semantischen-Rollen-Erkenner nicht möglich, da die intern verwendeten Werkzeuge auf ganze (Teil-)Sätze angewiesen sind.
Die andere Variante wäre es, die Klassifikationsergebnisse der ersten Stufe als Eingabe für den SRL-Agenten zu verwenden. Die Eingabe bestünde dann aus jenen Nutzereingaben, die als Lehrsequenz (Klasse TS) detektiert wurden. Als Eingabe werden somit keine Teilsätze, sondern die vollen Nutzereingaben als Eingabe betrachtet. Dies ist für den Einsatz des SRL-Agenten von Vorteil, da er auf vollständigen Nutzereingaben trainiert wurde. Die Ausgabe dieser Klassifikationsvariante wäre somit unabhängig von den Klassifikationsergebnissen der zweiten Stufe. Deshalb müssten in einem zusätzlichen Schritt die durch den SRL-Agenten extrahierten semantischen Rollen in einem Nachverarbeitungsschritt auf die drei Klassen der zweiten Klassifikationsstufe zurückgeführt werden. Aufgrund der besser zu erwartenden Klassifikationsergebnisse des SRL-Agenten bei Eingabe von vollen Nutzereingaben wird die zweite Variante gewählt.

Die Ausgabeklassen werden aus der Implementierung des SRL-Agenten gewonnen. Ein Wort kann dabei mehrere und auch unterschiedliche Rollen von den umgebenden Wörtern zugewiesen bekommen. Dies wurde bereits in den Grundlagen in Abschnitt 2.2.1.5 er-

läutert. Der SRL-Agent vergibt den Verben einer Eingabesequenz die semantische Rollenmarkierung V, die zugehörigen Argumente werden mit den Rollen A_i annotiert. Die Rolle A_0 steht normalerweise für den Handelnden *lat. Proto-Agent)* und Arg_1 für den Behandelten *lat. Proto-Patient)* des beschriebenen Ereignisses. Weitere mögliche Rollenmarkierungen sind in der Arbeit von Collobert [CWB+11] nachzulesen. Die Extraktion von Klassifikationsdaten der dritten Stufe wird in Definition 8.1.3 zusammengefasst.
Im Bezug auf diese Arbeit sollen in der dritten Stufe der Klassifikation die Aktionsbenennungen, bestehend aus Aktionsnamen und Aktionsentitäten (siehe Definition Definition 5.1.4), aus der Nutzereingabe extrahiert werden. Die Ausgabeklassen des SRL-Agenten sind auf diese beiden Gruppen direkt abbildbar. Die markierten Verben (Rolle V) der Nutzereingabe können als Aktionsname interpretiert werden, während die Argumente des SRL-Agenten (Rolle A_i) den Aktionsentitäten eines Nutzerkommandos entsprechen. Deshalb liefert der SRL-Agent die passenden Klassifikationsergebnisse für die dritte Klassifikationsstufe dieser Arbeit.

Definition 8.1.3: Stufe 3 - Eingabedaten

- Eingabesequenzen: Lehrsequenzen (Klasse TS) aus Stufe 1
- Ausgabeklassen: wortweise
  - Aktionsname V
  - Argument A_i

Da in dieser Klassifikationsstufe kein eigener Klassifikator trainiert wird, muss für diese Stufe kein Goldstandard erstellt werden. Zur Verdeutlichung der Ausgabeklassen wird in Beispiel 8.1.3 das Annotationsverhalten des Semantischen-Rollen-Erkenners an exemplarischen Nutzereingaben veranschaulicht. Während im dargestellten Beispiel die einzelnen Wörter annotiert sind, werden die tatsächlichen Annotationen des SRL-Agenten anhand neu eingefügter Kanten zwischen diesen Wörtern in den PARSE-Graph hinzugefügt. Neben der Klasse V werden unterschiedliche Markierungen für die Argumente, hier Entitäten genannt, vergeben. Diese annotieren dabei nicht nur die Wortgruppen der Nominalphrasen, sondern unter anderem auch zugehörige Präpositionen oder Adverbien (siehe Abschnitt 2.2.1.2). Zusammengehörige semantische Satzbestandteile werden dabei mit der selben Markierung, beispielsweise A1, versehen.
Wichtig für die zweite Teilaufgabe sind jedoch vor Allem die durch den SRL-Agenten annotierten Satzbestandteile. Diese semantisch zusammengehörigen Gruppierungen repräsentieren die Aktionsbenennungen der Nutzereingabe und müssen für jedes Hauptverb (Rolle V) einer Eingabesequenz extrahiert werden. Diese zu extrahierenden Aktionsbenennungen werden exemplarisch für Eingabe E1 in Beispiel 8.1.4 gelistet. Dabei soll die Trennung der Wörter in die einzelnen semantischen Rollen in der Ausgabe erhalten bleiben. Zudem fällt auf, dass in der zweiten Gruppierung zwei Wörter mit der Rolle V markiert sind. Hier kann jedoch durch die zusätzliche Rollenmarkierung AM-MOD das Hilfsverb *„have“* vom eigentlichen Hauptverb *„locate“* unterschieden werden.

Beispiel 8.1.3: Stufe 3 - Annotierte Lehrsequenz

E1 (gekürzt):
in order to [make $]_{\mathbf{V1_1}}$ [coffee $]_{\mathbf{A1_1}}$ [you $]_{\mathbf{A0_2}}$ [have $]_{\mathbf{V,\ AM\text{-}MOD_2}}$ to [locate $]_{\mathbf{V_2}}$ [the cups $]_{\mathbf{A1_2}}$ [next $]_{\mathbf{AM\text{-}TMP_2}}$ [to the machine $]_{\mathbf{A4_2}}$ and [press $]_{\mathbf{V_3}}$ [the red button $]_{\mathbf{A1_3}}$ [on the coffee machine $]_{\mathbf{AM\text{-}LOC_3}}$

wobei der Rollenindex i eine zum Hauptverb $\mathbf{V_i}$ gehörige SRL-Gruppierung angibt

Tabelle 8.5: Programmrepräsentation der Lehrsequenz-Bestandteile

Lehrsequenz-Bestandteil	Kl.stufe	Programm-Bestandteil
Lehrsequenz	1	Methode
Befehlssequenz	1	Methodenrumpf
Kommandodeklaration	2	Methodenkopf (Signatur)
Kommandobeschreibung	2	Methodenrumpf (Prozedur)
Sonstige Äußerungen	2	Methodenkopf bzw. Methodenrumpf
Aktionsname	3	Methodenname bzw. Funktionsname
Entität	3	Methodenparameter bzw. Funktionsparameter

Beispiel 8.1.4: Stufe 3 - Extrahierte Aktionsbenennungen

$\mathbf{V_1}$ make: coffee
$\mathbf{V_2}$ have locate: you, the cups, next, to the machine
$\mathbf{V_3}$ press: the red button, on the coffee machine

8.2 Aufbau von Methodendefinitionen

In der zweiten Teilaufgabe des Aufbaus von Methodendefinitionen dieser Arbeit müssen die Klassifikationsergebnisse der ersten Komponente dieses Agenten als Programmbestandteile repräsentiert und in einer allgemeingültigen Datenstruktur modelliert werden (siehe Abschnitt 7.3.2).

Im ersten Schritt werden die Klassifikationsergebnisse betrachtet. Auf der sprachlichen Ebene gibt es die Lehrsequenzen. Diese entsprechen auf der Ebene der Programmrepräsentation dem Hinzufügen einer neuen *Methode* zur bestehenden Programmschnittstelle des Zielsystems. Weiterhin können die Kommandodeklaration einer Lehrsequenz auf den *Methodenkopf* eines Programms sowie die Kommandobeschreibungen auf den *Methodenrumpf* abgebildet werden. Die einzelnen Teilaufgaben der vom Nutzer beschriebenen Funktionalität entsprechen verschiedenen *Methodenaufrufen* im Rumpf der neu definierten Methode. Die Aktionsnamen repräsentieren den *Methodennamen* der neu hinzugefügten Methode beziehungsweise den *Funktionsnamen* der im Rumpf aufgerufenen Funktionen. Zuletzt lassen sich die Entitäten auf die *Parameter* einer Funktion abbilden.
Wird in der ersten Stufe der Klassifikation eine Befehlssequenz statt einer Lehrsequenz erkannt, so entspricht das auf Ebene der Programmrepräsentation dem Aufrufen eines Skriptes bestehend aus einer Sequenz von Funktionsaufrufen. Deshalb kann eine Befehlssequenz auf eine Menge von Funktionsnamen und Parametern abgebildet werden. Dieses Befehlssequenz-Skript ist vergleichbar mit dem definierten *Methodenrumpf* einer Lehrsequenz-Methode, jedoch ohne die Definition eines Methodenkopfes. Die beschriebene Zuordnung von Lehrsequenz-Bestandteile auf Programmelemente wird in Tabelle 8.5 illustriert. Außerdem wird in Abbildung 8.2 die entworfene Datenstruktur veranschaulicht. Die Grafik veranschaulicht die verschiedenen Oberkonzepte durch eine Hierarchie. Die Beziehungen zwischen Lehrsequenz-Bestandteil und Programm-Bestandteil werden durch das Symbol $\leftrightarrow$ gekennzeichnet. Zudem ist eine Definition der verwendeten Begriffe der Programmsyntax in den Grundlagen in Abschnitt 2.6 zu finden.

Für eine Programmrepräsentation der Klassifikationsergebnisse wird eine Datenstruktur modelliert. Dazu müssen die Ergebnisse der drei Klassifikationsstufen, bestehend aus annotierten Satzbestandteilen einer Nutzereingabe, zunächst vereinigt werden. Die gewonnen Aktionen und Entitäten aus der dritten Klassifikationsstufe (siehe Abschnitt 8.1.2) müssen hierzu den Klassen der zweiten Klassifikationsstufe (siehe Abschnitt 8.1.3), das

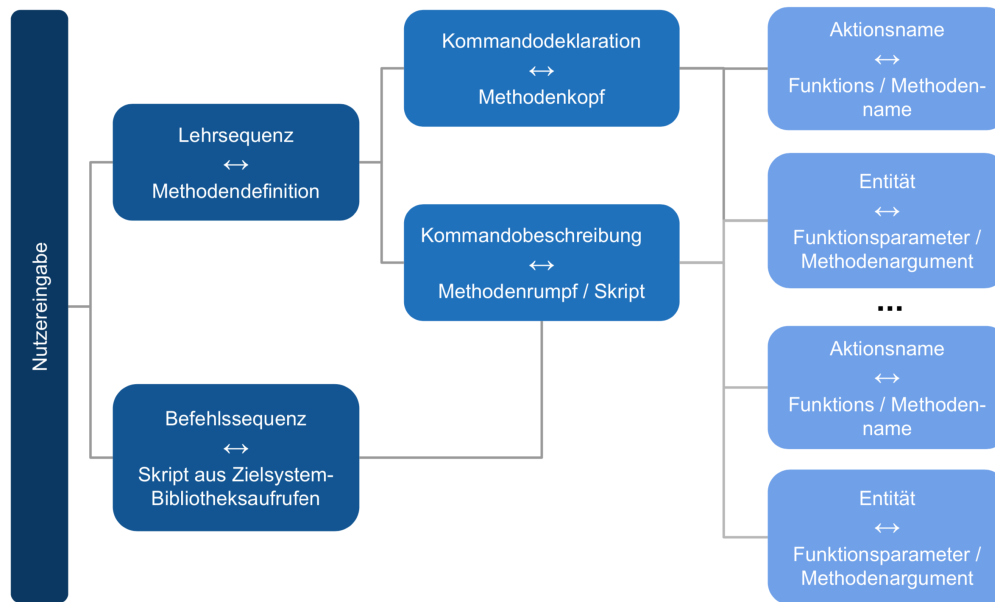


Abbildung 8.2: Datenstruktur zur Programmrepräsentation der Lehrsequenz-Bestandteile

heißt dem Teil der Kommandobeschreibung beziehungsweise Kommandobeschreibung, zugeordnet werden. Anschließend müssen die genannten Herausforderungen (H 2) in Abschnitt 7.1 erfüllt, das heißt, sämtliche Fälle möglicher Methodendefinitionen abgedeckt werden. Da eine Nutzereingabe sowohl eine Definition einer neuen Funktion (Lehrsequenz) als auch eines sofort ausführbaren Kommandos (Befehlssequenz) enthalten kann, müssen beide Fälle von der modellierten Datenstruktur abgebildet werden können. Dazu muss der Programmablauf des Agenten an dieser Stelle abweichen. Wie bereits oben angedeutet, wird dazu im Falle einer Befehlssequenz die zweite Stufe der Klassifikation übersprungen. Der Inhalt der Nutzereingabe wird direkt als Sequenz von Methodenaufrufen, das heißt als Kommandobeschreibung beziehungsweise *Methodenrumpf* interpretiert. Nach der binären Klassifikation der ersten Stufe kann zur Detektion von Aktionen und Entitäten deshalb direkt mit der dritten Klassifikationsstufe fortgefahren werden. Dieses Vorgehen wird in Abbildung 8.1 durch die Abzweigung der Ausführung in der ersten Kartei visualisiert. Weiterhin muss die Datenstruktur zur Modellierung der Programmrepräsentation einer Nutzereingabe auch eine beliebige Anzahl an Teilaufgaben und zugehörigen Entitäten unterstützen. Dies wird in den Anforderungen an den Agenten in Abschnitt 7.3.2 als Anforderung an die Modularität (A2) definiert.

Zuletzt muss das entworfene Werkzeug als Agent in Projekt PARSE (siehe Abbildung 3.1) integriert werden. Diese Integration wurde in Abschnitt 7.3.2 bereits beschrieben. Dazu soll eine im PARSE-Graph modellierte Nutzereingabe eingelesen und anschließend mit den zuvor beschriebenen Ansätzen maschinellen Lernens klassifiziert werden können. Diese Klassifikationsergebnisse werden darauffolgend, wie oben beschrieben, als repräsentierte Programmelemente in einer Datenstruktur gespeichert.

8.3 Ontologie-Abbildung

In der dritten Teilaufgabe der Ontologie-Abbildung dieser Arbeit müssen die extrahierten Bestandteile der Nutzereingabe auf die Zielsystem-Ontologie abgebildet werden. Dazu wird in diesem Abschnitt ein Vorgehen entworfen, wie die im vorherigen Schritt modellierten Programmbestandteile einer klassifizierten Nutzereingabe auf konkrete Funktionen eines Zielsystems abgebildet werden können.

Dazu wird zunächst die in dieser Arbeit eingebundene Ontologie vorgestellt. Die Ontolo-

Tabelle 8.6: Struktur der Domänenontologie von PARSE

Konzept	Beschreibung	Beispiel
System	ansprechbare Systeme	`Armar, Robot`
Object	auftretende Objekte	`Bottle, Table`
Method	ausführbare Aktionen	`bring, turnOff`
State	annehmbare Zustände	`closed, dirty`
DataType	Datentypen	`boolean, double`
Parameter	Argumente der Methoden	`bring.what, bring.where`
Value	Werte der Parameter	`Bottle, Table`

Tabelle 8.7: Ausprägungen des Ontologiekonzepts DataType

Datentyp	Beschreibung	Beispiel	Parameter
Object	auftretende Objekte	`Bottle`	`get.what`
Grabable	Greifbare Objekte	`Bottle`	`bring.what`
Drinkable	Trinkbare Objekte	`Coffee`	`pour.what`
Locatable	Anvisierbare Objekte	`Coffee`	`bring.where`
Speed	Geschwindigkeitswerte	`Slow`	`move.howFast`
int	Integer	`three`	`what.howLong`
String	Zeichenkette	`hello`	`say.what`

gie modelliert die Domäne des Haushaltsroboters ARMAR-III [ARAS06] und spezifiziert dessen Funktionsumfang sowie die Küchenumgebung, welche zur Interaktion bereitsteht. Die Struktur der Ontologie wird in Tabelle 8.6 dargestellt. In der Ontologie befinden sich manuell erstellte Informationen über die Aktionen, die das Zielsystem ausführen kann, sowie auftretenden Elemente der Umgebung und deren mögliche Zustände. Die Klasse `Method` stellt beispielsweise das Konzept für Aktionen, `Parameter` das Konzept für die Argumente dieser Aktionen dar. So kann mittels der Aktion `bring` und deren Parameter `bring.what` und `bring.where` ein Objekt der Umgebung, wie beispielsweise eine Flasche (`Bottle`), zu einem Zielort, beispielsweise dem Tisch (`Table`), gebracht werden. Die Objekte sowie die Aktionen und deren Argumente sind zusätzlich mit einschränkenden Datentypen versehen. Diese Einschränkungen beschreiben Beziehungen zwischen den Individuen. Sie definieren durch die zulässigen Werte der Argumente einer Aktion, wie Objekte miteinander interagieren können und welche Zustände ein Objekt annehmen kann. In Tabelle 8.7 wird ein Auszug dieser Datentypen präsentiert. Es lässt sich erkennen, dass die Objekte dort entsprechend ihrer Eigenschaften weiter unterteilt und spezifiziert werden. So bildet beispielsweise die Datentypklasse `Grabable` eine Schnittstelle für greifbare Objekte, während die Klasse `Drinkable` die Objekte bündelt, welche die Eigenschaften einer Flüssigkeiten besitzen. Da Individuen einer Ontologie mehreren Klassen angehören können, können auf diese Weise auch Objekte repräsentiert werden, welche mehrere Objekteigenschaften erfüllen. Dies ist beispielsweise für das Objekt-Individuum `Coffee` der Fall, welches sowohl greifbar (Grabable), als auch verzehrbar (Drinkable) ist. Zuletzt ist anzumerken, dass es primitive Datentypen gibt, die keine Repräsentation in der Ontologie besitzen. Dazu zählen beispielsweise Zahlenrepräsentationen wie `int` und `double` oder auch die Modellierung von Zeichenketten mit `String`. Diese sind im Gegensatz zu den Konzepten anderer Datentypen, beispielsweise dem Objekt `Table`, nicht als eigenständiges Individuum in der Ontologie modelliert.

Für die Umsetzung der gewünschten Abbildung von Programmbestandteilen auf Ontologie-Individuen müssen zwei Fälle unterschieden werden. Im Falle einer Lehrsequenz wird neues Wissen in Form einer neuen Methode deklariert. Ziel ist es, das Zielsystem zu erweitern. Für

die Beschreibung dieser neuen Methode wird auf bestehende Funktionalitäten zurückgegriffen. Deshalb muss einerseits für die Ergänzung des Zielsystems eine neue Methode synthetisiert, sowie andererseits eine Abbildung der Eingabe auf bestehende Elemente umgesetzt werden. Da die Zielsystem-API in PARSE als Ontologie modelliert ist, entspricht dies der Synthese einer Signatur für die neue Methode für den ersten Schritt. Im zweiten Schritt müssen die extrahierten Programmbestandteilen auf Ontologie-Individuen abgebildet werden. Dieses Vorgehen wird auch im Programmablauf des Agenten in Abbildung 8.1 innerhalb der letzten Kartei zu Teilaufgabe drei dieser Arbeit veranschaulicht: Auf die Signatur-Synthese folgt die Abbildungssuche. Im Falle einer detektierten Befehlssequenz wird keine Funktionserweiterung des Zielsystem beschrieben. Die Programmrepräsentation entspricht einer Sequenz aus Funktionsaufrufen (siehe Abschnitt 8.2), weshalb nur der zweite Schritt der Ontologie-Abbildung notwendig ist. Bezogen auf die Abbildung 8.1 spaltet sich der Programmablauf an dieser Stelle. Die zweite Kante ausgehend von der Programmrepräsentation in der zweiten Kartei führt direkt zum zweiten Block der Abbildungssuche in der letzten Kartei. Der erste Block der Signatur-Synthese wird folglich übersprungen.
Zusammenfassend ist es nicht das Ziel dieser Teilaufgabe, neues Wissen durch die Ergänzung einer Zielsystem-Ontologie zu modellieren. Stattdessen sollen die Bestandteile der Nutzereingabe, die zuvor als Instruktionsnamen und Instruktionsparameter extrahiert wurden in der Wissensrepräsentation wiedergefunden werden. Dies entspricht einer Abbildung *extrahierter* Instruktionen (siehe Definition 8.3.1) auf bestehende Individuen der Ontologie sowie den Aufbau einer Komposition aus diesen Individuen. Die konzipierte Vorgehensweise dieser beiden Schritte wird im Folgenden vorgestellt.

Definition 8.3.1: Extrahierte Instruktionen

Die in der ersten Teilaufgabe dieser Arbeit klassifizierten und anschließend in der zweiten Teilaufgabe modellierten Instruktionsnamen und Instruktionsparameter werden als *extrahierte* Instruktionen bezeichnet.

8.3.1 Synthese einer Methodensignatur

Für das Hinzufügen einer in der Nutzereingabe beschriebenen Funktionalität muss die Roboter-API ergänzt werden. Dies kann in Form eines neuen Methoden-Individuums (`Method`) für die modellierte Domänenontologie umgesetzt werden. Für die Erstellung eines solchen Methoden-Individuums muss eine Methodensignatur definiert werden, die als Bezeichner für die neue Funktionalität dient.

Für die Synthese einer Signatur werden ein Name sowie Argumente für die neue Funktionalität benötigt. Diese werden innerhalb der Nutzereingabe in Form der Kommandodeklaration mitgeliefert, welche im ersten Schritt der Arbeit klassifiziert wird. Sie bilden den Programmbestandteil der Signatur aus der zweiten Teilaufgabe bestehend aus Methodennamen und benötigten Methodenparametern (siehe Tabelle 8.5). Werden innerhalb einer Kommandodeklaration mehrere Signaturen identifiziert, so sind diese als gleichbedeutende, synonyme Definitionen der selben Funktion zu betrachten.
Das Beispiel 8.3.1 soll dies an einer exemplarischen Nutzereingabe verdeutlichen. Die als Kommandodeklaration detektierten Satzteile enthalten mehrere extrahierte Instruktionen I1 bis I3 (Definition 8.3.1). Die extrahierten Namen und Parameter werden im Beispiel als Instruktionsnamen und Instruktionsparameter bezeichnet, um sie von dem primär gesuchten Methodenkopf der neuen Roboterfunktionalität zu unterscheiden. Im dritten Schritt werden zwei mögliche Kandidaten K1 und K2 für diese neue, zu erstellende Methode generiert. Da diese Kandidaten der Kommandodeklaration als gleichbedeutend betrachtet werden, muss zuletzt ein Kandidat ausgewählt werden. Ziel ist es, aus diesem Kandidaten eine Methodensignatur für ein neues Methoden-Individuum zu synthetisieren, welches die

neue Funktionalität der Nutzereingabe kapselt. Im darauffolgenden Abschnitt 8.3.2 kann anschließend der Inhalt dieser Funktionalität entsprechend der Nutzereingabe durch die Abbildung der Funktionsaufrufe auf weitere Ontologie-Individuen umgesetzt werden.

Definition 8.3.2: Bereinigte Instruktionen

Die vorverarbeiteten *extrahierten* Instruktionen werden als *bereinigt* bezeichnet.

Um die beiden möglichen Kandidaten für Methodenname und Parameter zu erhalten, müssen die Instruktionen *bereinigt* (Definition 8.3.2) werden. So soll sichergestellt werden, dass der Bezeichner neuer Ontologie-Individuen aussagekräftig definiert sind. Dazu müssen die Instruktionsnamen zunächst lemmatisiert sowie Hilfsverben entfernt werden. Das Lemma beziehungsweise der Wortstamm eines Eingabewortes (siehe Abschnitt 2.2.1.3) lässt sich aus dem zugehörigen Knoten des PARSE-Graphen gewinnen. Analog dazu lässt sich die Ausprägung eines Wortes als Hilfsverb über die erkannte Wortart in den Knotenattributen finden. Zuletzt müssen überflüssige Instruktionen entfernt werden. Bezogen auf das Beispiel 8.3.1 stellt die extrahierte Instruktion I2 keinen sinnvollen Bezeichner dar. Deshalb müssen die Instruktionen in einem weiteren Verarbeitungsschritt von den Floskeln der Lehrsequenzen (siehe Vorstudien-Abschnitt 6.5.2) bereinigt werden.
Ähnlich zu den Instruktionsnamen müssen auch die extrahierten Instruktionsparameter bereinigt werden. Wichtig für die Spezifikation eines Parameters ist das Substantiv. Da die Bezeichner der betrachteten Ontologie-Individuen im Singular definiert sind, werden auch die Substantive lemmatisiert, um sie ebenfalls auf ihren Wortstamm im Singular abzubilden. Durch Adjektive oder Adverbien kann der Parameter zusätzlich näher beschrieben werden, deshalb werden auch diese für die Synthese eines Methodennamens betrachtet. Deshalb könnte die Filterung der Entitäten durch eine Liste mit relevanten Wortarten sowie nicht erwünschten Stoppwörtern umgesetzt werden. Schlussendlich erhält man die beiden gewünschten Kandidaten K1 und K2 für die zu synthetisierende Methodensignatur. Zuletzt kann die gewünschte Methodensignatur für die neue Funktionalität durch Konkatenation des Methodennamens und der Parameter erreicht werden. Dies wird in Beispiel 8.3.1 im letzten Schritt bei Wahl des Kandidaten K1 mit der generierten Methodensignatur M1 veranschaulicht.

Beispiel 8.3.1: Kandidaten für die Methodensignatur

Nutzereingabe
„preparing a cup of coffee means you have to put a coffee mug under the dispenser and then press the red button on the coffee machine thats how you make some coffee“

Kommandodeklaration
[preparing a cup of coffee means you have to]
[thats how you make some coffee]

Extrahierte Instruktionsnamen und Instruktionsparameter
I1 [preparing] [a cup of coffee]
I2 [means, have] [you]
I3 [make] [thats, how, you, some coffe]

Bereinigte Methodennamen und Methodenparameter (Kandidaten)
K1 [prepare] [cup coffee]
K2 [make] [coffee]

Synthetisierte Methodensignatur
M1 prepareCupCoffee M2 prepareCupCoffee(CupCoffee::Object)

Als erweiterten Schritt könnte geprüft werden, ob ein Kandidat nicht bereits als Methoden-Individuum in der Ontologie existiert. Dazu muss nach möglichen Abbildungen des Methodennamens sowie der Parameter auf bestehende Individuen gesucht werden. Auf diese Weise könnte man einerseits eine wiederholte Definition einer bereits existierenden Funktion vermeiden. Die dafür benötigte Abbildung auf bestehende Individuen wird im folgenden Abschnitt 8.3.2 erläutert.
Des Weiteren können die Methodenparameter mit bereits vorhandenen Ontologie-Elemente verknüpft werden, um die erstellte Methodensignatur mit bestehenden Ontologie-Parametern anzureichern. Für Beispiel 8.3.1 hieße dies, dass statt der Methodensignatur M1 `prepareCupCoffee` die Signatur M2 `prepareCupCoffee(CupCoffee)` eingespeichert wird, angenommen `CupCoffee` sei ein Individuum der Ontologie. Von einer reduzierten Form vom M2 ohne den konkatenierten Funktionsparameter im neuen Methodennamen (hier `prepare(CupCoffee)`) wird abgesehen, da dies zu zu vielen Namenskonflikten mit bestehenden Methoden-Individuen der Ontologie führen könnte. Diese Methodendefinition wäre beispielsweise von der primitiven Ontologie-Methode `prepare(prepare.what)::Method` nicht zu unterscheiden.

8.3.2 Abbildung auf bestehende Individuen

Für Interpretation der in der Nutzereingabe beschriebenen Aktionen müssen die in der zweiten Teilaufgabe extrahierten Programmrepräsentationen auf die Roboter-API abgebildet werden. Dies wird durch eine Abbildung der einzelnen Funktionsnamen und Funktionsparameter auf konkrete Individuen der Domänenontologie umgesetzt. Ziel ist es, aus den gewonnenen Abbildungskandidaten eine Sequenz aus bestehenden Methoden-Individuen (`Method`) inklusive deren benötigten Argumenten (`Parameter`) aufzubauen.

Im Falle einer Befehlssequenz entspricht dies einem Skript aus Bibliotheksaufrufen der Zielsystem-Programmierschnittstelle. Bei einer klassifizierten Lehrsequenz repräsentiert sie den Inhalt der neu definierten Methode und bildet den Methodenrumpf. Für Letztere wurde im vorherigen Abschnitt 8.3.1 bereits die Synthese der Signatur beschrieben, die den Methodenkopf der neuen Methodendefinition repräsentiert. Die genannten Aktionen werden in der Nutzereingabe in Form der Kommandobeschreibung mitgeliefert und in der ersten Teilaufgabe der Arbeit klassifiziert. In der zweiten Teilaufgabe werden sie anschließend dem Programmbestandteil des Funktionsaufrufs zugeordnet, welcher aus Funktionsnamen und benötigten Funktionsparametern besteht (siehe Tabelle 8.5). Werden deshalb in einer Kommandobeschreibung mehrere Aktionen identifiziert, so sind diese, im Gegensatz zu den Elementen einer Kommandodeklaration, als eigenständige und unterschiedliche Funktionsaufrufe zu betrachten.
Das Beispiel Beispiel 8.3.2 soll diesen Sachverhalt an einer exemplarischen Nutzereingabe veranschaulichen. Die detektierte Kommandobeschreibung enthält zwei Funktionsaufrufe I1 und I2. Die extrahierten Namen und Parameter (siehe Definition 8.3.1) dieser Aufrufe werden im Beispiel als Instruktionsnamen und Instruktionsparameter bezeichnet, um sie von den primär gesuchten Kandidaten für die Funktionsaufrufe der neuen Roboterfunktionalität zu unterscheiden. Im dritten Schritt werden die verbliebenen Kandidaten K1 und K2 präsentiert. Da die Kandidaten als eigenständig betrachtet werden, muss für jeden dieser Kandidaten anschließend eine Abbildung auf bestehende Ontologie-Individuen getätigt werden. Ziel ist es, eine Sequenz aus Ontologie-Methoden inklusive gesetzten Parametern zu finden, die den Inhalt der beschriebenen Nutzenfunktion widerspiegelt.

Beispiel 8.3.2: Vorverarbeitung der Elemente für den Methodenrumpf

Nutzereingabe
„preparing a cup of coffee means you have to put a coffee mug under the dispenser and then press the red button on the coffee machine thats how you make some coffee"

Kommandobeschreibung
[put a coffee mug under the dispenser and then press the red button on the coffee machine]

Extrahierte Instruktionsnamen und Instruktionsparameter
I1 [put] [a coffee mug, under, the dispenser]
I2 [press] [then, the red button, on the coffee machine]

Bereinigte Funktionsnamen und Funktionsparameter (Kandidaten)
K1 [put] [coffee mug, dispenser]
K2 [press] [red button, coffee machine]

Für das Erhalten der beiden Kandidaten für Funktionsname und Parameter müssen die Instruktionen wiederum vorverarbeitet, das heißt nach Definition 8.3.2 bereinigt, werden. Ziel ist es, das Abbildungsproblem durch den Einsatz von Sprachverarbeitungstechniken zu vereinfachen (Herausforderung (H 3.1) in Abschnitt 7.1). Dazu sollen die Instruktionsnamen analog zum Vorgehen im vorherigen Abschnitt 8.3.1 lemmatisiert und auftretende Hilfsverben entfernt werden. Auch die Verarbeitung der Instruktionsparameter soll gleich erfolgen. Dazu gehört das Filtern von Wortarten sowie das Entfernen von Stoppwörtern. Da die Bezeichner der betrachteten Ontologie-Individuen im Singular definiert sind, werden auch die Substantive lemmatisiert, um die Wahrscheinlichkeit für eine Übereinstimmung der Bezeichner basierend auf die Bezeichner-Distanzmetriken zu erhöhen. Schlussendlich erhält man eine Menge von Kandidaten, für welche im nachfolgenden Schritt eine Abbildung auf Ontologie-Individuen gesucht wird. Dabei wird die folgende Abbildungssuche in die namensbasierte Abbildungssuche, das heißt auf Basis von Bezeichner-Distanzmetriken gefundene Abbildungen in Abschnitt 8.3.2.1, sowie typbasierte Abbildungssuche, das heißt auf Basis ähnlicher Datentypen gefundene Abbildungen in Abschnitt 8.3.2.2, untergliedert. Diese gefundenen Abbildungskandidaten sollen im letzten Schritt mit einer Bewertung versehen werden. Damit verfolgt der Ansatz der Ontologie-Abbildung einen bekanntes Vorgehen, häufig bezeichnet als Informationsgewinnung (engl. *information retrival*). Es werden zunächst so viele Informationen wie möglich (hier Kandidaten) für eine Abbildung herangezogen, damit anschließend entsprechend einer definierten Bewertungsfunktion eine Rangliste aus diesen Ergebnissen aufgestellt werden kann. Für die Abbildungssuche werden die vier folgenden Bezeichnungen aus Definition 8.3.3, Definition 8.3.4, Definition 8.3.5 und Definition 8.3.6 eingeführt, um unterschiedliche Stadien der Abbildung zu beschreiben.

Definition 8.3.3: Zutreffende Bezeichner-Abbildung

Eine mittels der namensbasierten Abbildungssuche gefundene Zuordnung zwischen Programmbestandteil-Bezeichner und Individuums-Bezeichner wird als *zutreffend* (engl. *matched*) bezeichnet.
Hierfür muss eine Ähnlichkeit der Bezeichner auf Basis von Zeichenketten-Distanzmetriken erkannt werden.

Definition 8.3.4: Abzubildendes Methodenargument

Ein gefordertes Argument eines Methoden-Individuums wird als *abzubildendes* Ontologie-Individuum bezeichnet.

Definition 8.3.5: Typ-sichere Parameter-Abbildung

Eine mittels der typbasierten Abbildungssuche gefundene Zuordnung zwischen einem *zutreffenden* Funktionsparameter und einem *abzubildenden* Ontologie-Argument einer Methode wird als *typ-sicher* abgebildetes Parameter-Individuum bezeichnet.
Hierfür müssen die Datentypen des Parameters und des abzubildenden Arguments übereinstimmen.

Definition 8.3.6: Vollständige Abbildung

Sind alle *abzubildenden* Argumente eines Methoden-Individuums *typ-sicher* abgebildet, wird die Methoden-Abbildung als *vollständig* bezeichnet.
Hierfür müssen alle Datentypen der abzubildenden Argumente und der zutreffenden Funktionsparameter übereinstimmen.

8.3.2.1 Namensbasierte Abbildungssuche

Die Diskussion über verschiedene Ansätze verwandter Themenstellungen in Abschnitt 7.3.3 hat ergeben, dass sich für die Problemstellung dieser Arbeit unter Betrachtung der Voraussetzungen ein Ansatz auf Basis der Ähnlichkeit von Element-Bezeichnern anbietet. Darunter wird der Vergleich der Elemente zweier Gruppen auf Basis von Zeichenketten-Vergleichsmetriken verstanden. Dieser Ansatz wird im Folgenden als namensbasierte Abbildungssuche bezeichnet. Bezogen auf diese Problemstellung heißt das, dass ein extrahierter Programmbestandteil (siehe Definition 8.3.1) der Nutzereingabe, beispielsweise ein Funktionsname, mit den Bezeichnern der Ontologie-Individuen abgeglichen werden muss. Zusätzlich dazu werden verschiedene Vorverarbeitungstechniken für die Verarbeitung von natürlicher Sprache eingesetzt, um diesen Ansatz weiter zu optimieren.

Für den Vergleich von Bezeichnern bieten sich Zeichenketten-vergleichende (engl. *string matching*) Verfahren (siehe Abschnitt 8.3.2.1) an, um die Ähnlichkeit zweier Bezeichner zu bestimmen. Häufig bilden diese Verfahren die Ähnlichkeit zweier Zeichenketten auf einen Zahlenbereich zwischen Null und Eins ab. Desto höher der Wert, desto ähnlicher sind sich zwei Element-Bezeichner. Bei einer Abbildungssuche können dabei folgende drei Fälle auftreten:

- 1-zu-1: 1 Programmbestandteil auf 1 Ontologie-Individuum
- 1-zu-0: 1 Programmbestandteil auf 0 Ontologie-Individuen
- 1-zu-N: 1 Programmbestandteil auf N Ontologie-Individuen

Bei einer *1-zu-1-Abbildung* wird zu einem Programmbestandteil ein einziges zutreffendes Ontologie-Individuum (Definition 8.3.3) gefunden. Es tritt jedoch häufiger auf, dass entweder keine Ähnlichkeit zwischen den Bezeichnern (*1-zu-0-Abbildung*) erkannt oder dass eine Menge von zutreffenden Bezeichnern *1-zu-N-Abbildung* gefunden wird. Während im letzten Fall alle N Individuen in die Betrachtung parallel miteinbezogen werden können, ist im zweiten Fall keine Zuordnung mehr möglich. Er muss als fehlender Funktionsaufruf beziehungsweise fehlender Funktionsparameter behandelt werden. Weiterhin darf aus einer gefundenen Zuordnung, die den höchsten Ähnlichkeitswert besitzt, nicht gefolgert werden, dass diese automatisch eine gültige Abbildung zwischen Programmbestandteil und Ontologie-Individuum darstellt. Diese Problematik muss in einer weiteren Lösungsstrategie angegangen werden, welche in Abschnitt 8.3.2.2 beschrieben wird.

Bezogen auf den Aufbau von Methodendefinitionen kann der namensbasierte Ansatz sowohl für das Finden von Abbildungen der Funktionsnamen auf Ontologie-Methoden als auch

für die Zuordnung der Funktionsparameter auf Ontologie-Parameter verfolgt werden. Die Suche der Abbildungen soll hierbei für Methoden und Parameter getrennt und unabhängig voneinander umgesetzt werden. In Beispiel 8.3.3 wird die beschriebene namensbasierte Abbildungssuche für die exemplarische Funktion F1 durchgeführt. Für den bereinigten Funktionsnamen (Definition 8.3.2) *put* von F1 werden im Beispiel die drei Individuen M1 bis M3 ihrem Bezeichner nach als zutreffend betrachtet. An ihrem zugewiesenen Ähnlichkeitswert ist abzulesen, dass das Methoden-Individuum `put::Method` mit 1.0 eine perfekte Übereinstimmung darstellt, wobei im Beispiel der Suffix nach `::` das Konzept der Ontologie angibt (siehe Tabelle 8.6). Weiterhin könnte es für die Suche nach Entsprechungen von Funktionsnamen innerhalb der Methoden-Individuen hilfreich sein, die Bezeichner der bereinigten Funktionsparameter in diese miteinzubeziehen. Auf diese Weise könnte das Individuum `putDown::Method` von dem Individuum `put::Method` dank eines zusätzlichen Funktionsparameters *down* unterschieden werden.
Bei der Abbildungssuche für die Funktionsparameter muss zusätzlich beachtet werden, dass für jeden der extrahierten Parameter einzeln eine Zuordnung in der Ontologie gesucht werden muss. Dies wird in Beispiel 8.3.3 durch P1 und P2 verdeutlicht. Außerdem muss nicht nach dem Ontologie-Konzept `Method`, sondern nach den Konzepten `Object`, `Value` und `State` gesucht werden, da diese Ontologie-Individuen die Elemente der Zielsystem-Umgebung modellieren, mit welchen interagiert werden kann. Diese Ontologie-Individuen können daraufhin wiederum als mögliche Argumente einer Ontologie-Methode innerhalb des Konzeptes `Parameter` genutzt werden. Im Beispiel 8.3.3 werden dazu für den Funktionsparameter P1 vier zutreffende Individuen, für den Funktionsparameter P2 zwei zutreffende Individuen gefunden. Weiterhin besteht ein Funktionsparameter meist aus mehreren Wörtern. Deshalb könnte es von Vorteil sein, auch Permutationen der Wörter eines Funktionsparameters für den Vergleich der Bezeichner einzusetzen.

Beispiel 8.3.3: Namensbasierte Abbildungssuche

Bereinigter Funktionsname und Funktionsparameter
F1 [put] [coffee mug, dispenser]

Suche nach zutreffenden Methoden-Individuen
M [put]
M1 Wert: 1.0 `put::Method`
M2 Wert: 0.8 `cut::Method`
M3 Wert: 0.6 `putDown::Method`

Suche nach zutreffenden Parameter-Individuen
P1 [coffee mug]
P1.1 Wert: 0.9 `CoffeeCup::Object`
P1.2 Wert: 0.8 `Coffee::Object`
P1.3 Wert: 0.8 `Coffee::Value`
P1.4 Wert: 0.3 `cold::State`
P2 [dispenser]
P2.1 Wert: 0.6 `CoffeeMachine.Dispenser::Object`
P2.2 Wert: 0.3 `Dishes::Object`

Das Beispiel 8.3.3 soll überdies verdeutlichen, dass nicht nur gültige Abbildungen von Funktionsnamen und Funktionsparametern auf bestehende Ontologie-Individuen getätigt werden. Um robustere Ergebnisse bei Einsatz von Zeichenketten-Vergleichsverfahren zu erhalten, bietet es sich daher an, die Ergebnisse verschiedener Verfahren innerhalb eines kombinierten Ansatzes zu nutzen. Beispielsweise könnte ein Ensemble-Ansatz mit den zwei Distanzmetriken *Jaro-Winkler* und *Fuzzy* eingesetzt werden. Diese werden in den

Grundlagen in Abschnitt 2.3 mit ihren zugrundeliegenden mathematischen Formeln sowie anhand von Beispielen erläutert. Durch den unterschiedlichen Fokus der zugrundeliegenden Bezeichner-Ähnlichkeisberechnungen dieser beiden Metriken könnte eine Kombination die Vorteile beider Ansätze vereinigen. Diese werden im Folgenden kurz erklärt. Die Jaro-Winkler-Ähnlichkeitsmetrik (siehe Gleichung 2.4) eignet sich für den Vergleich kurzer Zeichenketten und achtet besonders auf den Präfix zweier Eingaben [CRF03]. Diese Eigenschaften passen zur hiesigen Problemstellung der Ontologie-Abbildung, da die Ontologie-Bezeichner sowie die extrahierten Programmbestandteile überwiegend aus kurzen Zeichenfolgen bestehen. Weiterhin könnte die präfixorientierte Suche einen Vorteil darstellen, da bei der Benamung häufig die spezifischere Entität zuerst genannt wird. Dies zeigen die Beispiele `AppleJuice` und `OrangeJuice` sowie `waveHand` und `turnAround`. Eine präfixorientierte Abbildungssuche würde den Fokus eines Bezeichner-Vergleichs auf den vorderen Teil der Zeichenkette verstärken. Im Gegensatz zur Jaro-Winkler-Metrik wird der Fuzzy-Algorithmus (siehe Gleichung 2.1) eingesetzt, um eine bestimmte Zeichenkette in einer längeren Eingabe oder gar einem Text zu finden. Dabei muss bei der Suche nicht die exakte Zeichenfolge einer Eingabe vorhanden sein, da auch leicht abweichende, unscharfe (engl. *fuzzy*) Zeichenketten abgeglichen werden können. Der Fuzzy-Algorithmus beruht auf der Levenshtein-Distanz, welche angibt, wie viele Operationen nötig sind, um eine Zeichenkette aus einer anderen herzuleiten. Die Vergleichsmetrik gewichtet die Unterschiede zweier Zeichenfolgen unabhängig von ihrer Position und stellt einen zweiten Ansatz dar, um robustere Ergebnisse zu erhalten. Mit beiden Verfahren könnten für die betrachteten Programmbestandteilen zutreffende Ontologie-Individuen gesucht, jedoch nur die Schnittmenge aus beiden Ergebnismengen weiterverwendet werden. Das Projekt PARSE stellt mit dem `OntologyConnector`-Modul bereits eine Schnittstelle zur Domänenontologie von PARSE als auch für den Einsatz von möglichen Zeichenketten-Vergleichsverfahren zur Verfügung. Daher muss das Modul in das zu entwickelnde Werkzeug dieser Arbeit eingebunden werden.
Zuletzt wäre es möglich, neben den Funktionsnamen und Funktionsparametern aus der Nutzereingabe, auch die Synonyme (siehe Abschnitt 2.1.5) dieser Programmbestandteil-Bezeichner für die Abbildungssuche zu nutzen. Dazu werden in der Arbeit von Hey [Hey16] die Knoten des PARSE-Graph wortweise mit Synonymen angereichert. Auf diese Weise könnte die Wahrscheinlichkeit, eine zutreffende Abbildung zu finden, erhöht werden. Allerdings enthalten diese Synonyme bedeutungsgleiche Wörter, welche nicht selten irreführend sein könnten. So wird das Verb *look* zu *see* als verwandt betrachtet, jedoch beispielsweise auch das Verb *wait*. Letzteres Synonym könnte zu einer fälschlichen Zuordnung führen. Deshalb muss bei Einsatz von Synonymen eine geringere Gewichtung der Ähnlichkeitswerte aus den Zeichenketten-Vergleichsverfahren in Betracht gezogen werden. Weiterhin müssen Korreferenzen in der Nutzereingabe (siehe Abschnitt 2.2.1.6) betrachtet werden, um Referenzen in der Nutzereingabe auf Objekte und Subjekte der Umgebung aufzulösen. Dies wurde als Herausforderung (H 2.3) in Abschnitt 7.1 definiert. Für die Beispieleingabe „go to the fridge and open it“ muss zur korrekten Abbildung der zweiten Instruktion das Personalpronomen „it“ mit dem Objekt „fridge“ gleichgesetzt werden, um es daraufhin als Argument der Methode `open` übergeben zu können. Hier kann ebenfalls die Arbeit von Hey [Hey16] herangezogen werden, in welcher der PARSE-Graphen mit sich gegenseitig referenzierenden Entitäten kennzeichnet.

8.3.2.2 Typbasierte Abbildungssuche

Da eine Abbildung basierend auf dem Vergleich von Element-Bezeichnern fehleranfällig ist und überdies nicht gültig sein muss, muss die Korrektheit einer gefundenen Zuordnung anderweitig verifiziert werden. Dies kann mit einer Überprüfung der Datentypen beider Elemente umgesetzt werden.

Für jeden extrahierten Funktionsnamen wird dazu die Menge zutreffender Methoden-

Abbil-
dungen (Definition 8.3.3) betrachtet. Diese zutreffenden Methoden-Individuen können wiederum eine Menge an Argumenten fordern. Die Beziehung zwischen Methoden und ihren Argumenten ist in der Domänenontologie durch Relationen eines Methoden-Individuums zu ihren abzubildenden Methodenargumenten (Definition 8.3.4) und deren möglichen annehmbaren Werten und Datentypen modelliert. Für jedes abzubildende Argument einer Methode wird deshalb nach einer Besetzung in der Menge der durch den namensbasierten Ansatz gefundenen, zutreffenden Funktionsparameter gesucht. Wird eine Übereinstimmung der Datentypen zwischen einer zutreffenden Parameter-Abbildung und einem gesuchten Methoden-Argument gefunden, bedeutet dies, dass der abgebildete Funktionsparameter als typ-sicherer Wert (Definition 8.3.5) für das Argument des betrachteten Methoden-Individuums gilt. Eine mit der typbasierten Suche gefundene Abbildung besteht daher aus einer zutreffenden Abbildung eines Funktionsnamens auf ein Methoden-Individuum sowie aus einer Menge von typ-sicher abgebildeten Funktionsparametern auf die Methoden-Argumente.
Eine Methode wird als vollständig abgebildet (Definition 8.3.6) interpretiert, falls für jedes benötigte Argument eines Methoden-Individuums ein typ-sicheres Ontologie-Individuum gefunden wird. Kann für ein betrachtetes Methoden-Individuum nur eine Teilmenge der geforderten Argumente abgebildet werden, so wird diese Methodensignatur als unvollständiges Ergebnis der Abbildungssuche ebenfalls behalten. Das gefundene Teilergebnis wird nicht verworfen. Gibt es mehrere Methoden-Individuen mit einer Menge typ-sicherer beziehungsweise vollständig abgebildeter Argumente, muss eine Rangfolge nach der besseren Übereinstimmung aufgestellt werden. Dazu ist eine Bewertungsfunktion nötig, die neben der Anzahl typ-sicher abgebildeter Parameter auch eine Bestrafung bei fehlenden Parametern oder nicht abbildbaren Programmbestandteilen umsetzt. Eine solche Funktion wird in Abschnitt 8.3.2.3 entworfen und erklärt.

Um die beschriebene Lösungsstrategie der Ontologie-Abbildung zu verdeutlichen, wird der konzipierte Ansatz in Beispiel 8.3.4 anhand einer exemplarischen Eingabe erläutert.
Eine Aktion der Nutzereingabe kann mehrere Entitäten als Argumente besitzen. Dies wird in der zweiten Teilaufgabe dieser Arbeit durch die Funktionsnamen inklusive den zugehörigen Funktionsparametern modelliert. In der zuvor beschriebenen namensbasierten Abbildung der dritten Teilaufgabe (Abschnitt 8.3.2.1) wird anschließend für jede dieser extrahierten Elemente eine Abbildungssuche auf Basis von Zeichenkettenvergleichen durchgeführt. Dabei wird wiederum je Element eine Kandidaten-Menge an Ontologie-Individuen generiert. Diese Menge besteht aus den zutreffenden Ontologie-Kandidaten (Definition 8.3.3) für die Funktionsnamen als auch für die Funktionsparameter. In Beispiel 8.3.4 wird dieser Schritt durch die Kandidaten M1, P1 und P2 veranschaulicht. Anschließend müssen pro Methoden-Kandidat alle möglichen Kombinationen mit den gefundenen Parameter-Kandidaten gebildet werden. So erhält man mit Hilfe des kartesischen Produktes für jeden der drei Methoden-Kandidaten für M1 sowie der drei Funktionsparameter-Kandidaten von P1 und den zwei weiteren Kandidaten von P2 eine Kombinationsmenge von 6 Elementen. Diese Kandidaten-Kombinationen der zutreffenden Ontologie-Individuen werden im Beispiel durch K1 bis K6 repräsentiert. Anschließend werden diese Funktionsaufrufs-Kandidaten als Eingabe für die typbasierte Abbildungssuche verwendet. Dazu muss jedoch wiederum ein kartesische Produkt, diesmal zwischen den Funktionsaufrufs-Kandidaten und den benötigten Argumenten des betrachteten Methoden-Individuums, gebildet werden. Für das Beispiel 8.3.4 heißt das, dass aufgrund der zwei zu besetzenden Argumente `put.what` und `put.where` des Methoden-Individuums `put::Method` 12 Datentyp-Vergleiche getätigt werden müssen. Dies wird mit den Funktionsaufruf-Kandidaten M1K1 bis M1K6 dargestellt, wobei der erste Kandidat M1K1 zwei von zwei Typprüfungen besteht, der zweite Kandidat beispielsweise nur einen Erfolg bei der Typprüfung des ersten

Methodenarguments verzeichnet. Das Ergebnis der typbasierten Suche ist im letzten Abschnitt des Beispiels präsentiert. Bei einer Übereinstimmung der Datentypen, das heißt einer typ-sicheren Parameter-Abbildung, muss diese Zuordnung gespeichert und anschließend bewertet werden. Zuletzt muss das beschriebene Vorgehen für jeden zutreffenden Methoden-Kandidaten (Definition 8.3.3) als auch für jede extrahierte, bereinigte Instruktion (Definition 8.3.2) der Nutzereingabe durchgeführt werden.

Beispiel 8.3.4: Typbasierte Abbildungssuche

zutreffende Methoden-Kandidaten:
M1 [put] : [`put::Method`, `cut::Method`, `putDown::Method`]

zutreffende Parameter-Kandidaten:
P1 [coffee mug] : [`CoffeeCup::Object`, `Coffee::Value`, `cold::State`]
P2 [dispenser] : [`CoffeeMachine.Dispenser::Locateable`, `Dishes::Object`]

Bilden von Parameter-Kandidaten-Kombinationen
K1 [`CoffeeCup::Object`, `CoffeeMachine.Dispenser::Locateable`]
K2 [`CoffeeCup::Object`, `Dishes::Object`]
K3 [`Coffee::Value`, `CoffeeMachine.Dispenser::Locateable`]
K4 ... K6

Betrachtetes Methoden-Individuum
M1 [`put::Method`] : [`put.what::Object`, `put.where::Locateable`]

Vergleich der Datentypen
M1K1 [`put.what::Object` ↔ `CoffeeCup::Object`] Erfolg
[`put.where::Locateable` ↔ `CoffeeMachine.Disp.::Locateable`] Erfolg
M1K2 [`put.what::Object` ↔ `CoffeeCup::Object`] Erfolg
[`put.where::Locateable` ↔ `Dishes::Object`] Misserfolg
M1K3 ... M1K6

Suche nach typ-sicheren Funktionsaufruf-Kandidaten
M1K1 [`put.what : CoffeeCup, put.where : CoffeeMachine.Dispenser`]
M1K2 [`put.what : CoffeeCup`]
M1K3 ... M1K6

8.3.2.3 Bewertung der Abbildungen

Im letzten Schritt der Ontologie-Abbildung müssen die gefundenen Funktionsaufruf-Kandidaten aus Abschnitt 8.3.2.2 bewertet werden. Das Ergebnis der Ontologie-Abbildung ist eine Menge aus bewerteten Kandidaten pro Funktionsaufruf. Gibt es mehrere Kandidaten für den selben Funktionsaufruf kann anhand der Bewertungsfunktion eine Rangfolge der Abbildungen aufgestellt werden.
Andererseits kann dank der Bewertungsfunktion ein Schwellwert (engl. *threshold*) definiert werden, welcher Abbildungen mit geringem Wert filtert. Dabei muss davon ausgegangen werden, dass ein Nutzer in seiner Eingabesequenz nicht immer korrekterweise alle benötigten Entitäten einer Aktion, das heißt Argumente einer Methode, mitliefert. Dies wurde bereits in Abschnitt 7.1 mit Herausforderung (H 3.2) definiert. Dieses Nutzerverhalten wird auch in der Studie von Malmaud et al. [MWCM14] und Misra [KMSLS15] beobachtet. Letztere beschreiben ihre Nutzerinstruktionen im Kontext der Küchenrezept-Domäne nicht nur als unvollständig, sondern auch als mehrdeutig. Deshalb sollen auch zutreffende Methoden-Kandidaten (Definition 8.3.3) mit nicht vollständig besetzten oder gar leeren Argumentlisten zur Bewertung zugelassen werden. Des Weiteren wird dieses Vorgehen stark durch die Tatsache bekräftigt, dass ein namensbasierter Ansatz für die

Abbildungssuche, trotz Ensemble-Vergleich der Bezeichnermetriken, Synonymen, Korreferenzauflösungen und Vorverarbeitung der Zeichenfolgen, fehleranfällig ist. Würde man nur vollständig abgebildete Methoden-Kandidaten (Definition 8.3.6) in die Bewertung mitaufnehmen, so würde sich die Ergebnismenge der Ontologie-Abbildung erheblich verkleinern.

In Gleichung 8.1 wird die Bewertungsfunktion S definiert. In der Gewichtung der Funktionsaufrufe sollen einerseits die zutreffenden Methoden-Kandidaten M_T (Definition 8.3.3) sowie die typ-sicher abgebildeten Parameter-Kandidaten P_V (Definition 8.3.5) berücksichtigt werden. Überdies soll auch die Abdeckung abgebildeter Methodenargumente (Definition 8.3.4) mit dem Gewicht P_{AA} sowie die Abdeckung extrahierter Programmbestandteile (Definition 8.3.1) mit dem Gewicht P_{AE} miteinbezogen werden.
Falls alle betrachteten Elemente eines Funktionsaufruf-Kandidaten gleich hoch bewertet werden würden, so hätte die Anzahl der Argumente einer Methode Einfluss auf ihre Bewertung. Methoden mit mehr abzubildenden Argumenten beziehungsweise typ-sicher abgebildeten Funktionsparametern bekämen eine höhere Bewertung als Methoden, die weniger Argumente fordern. Es muss daher sichergestellt werden, dass die Gewichtung für das Methoden-Individuum ähnlich stark einfließt, als die Gewichtung aller gefundener Parameter eines Funktionsaufrufs. Dazu soll die Bewertung für Methoden und für Parameter jeweils getrennt berechnet und mit einem festen Faktor in die Gesamtbewertung des Funktionsaufruf-Kandidaten miteinfließen. Dieser Faktor wird empirisch ermittelt. Es wird festgelegt, die Erkennung des richtigen Zielsystem-Bibliotheksaufrufs aus einer Nutzereingabe, das heißt die Abbildung auf das passende Methoden-Individuum wichtiger ist, als die Erkennung zutreffender Parameter-Individuen. Deshalb wird für die Gewichtung des Methoden-Kandidaten der Faktor 0.6, für die Gewichtung der typ-sicher abgebildeten Parameter-Menge der Faktor 0.4 gewählt. Es ergibt sich die Bewertungsformel in Gleichung 8.1, wobei diese zusätzlich den Subtrahenden P_{AE} und den Gewichtungsfaktor P_{AA} für die Parameter-Menge enthält.

$$\begin{aligned} S &= 0.6 * \text{zutreffender Methoden-Kandidat} \\ &\quad + 0.4 * \text{typ-sichere Parameter-Kandidaten} * \text{Abdeckung abzubildender} \\ &\qquad \text{Methodenargumente} \qquad -0.3 * \text{Abdeckung extrahi} \\ &= 0.6 * M_T + 0.4 * P_V * P_{AA} - 0.3 * P_{AE} \end{aligned} \tag{8.1}$$

Können nicht alle extrahierten Funktionsparameter (Definition 8.3.1) eines Funktionsaufrufs auf die Argumente eines Methoden-Kandidaten typ-sicher abgebildet werden, so muss für diesen Kandidaten eine Bestrafung einberechnet werden. Dies soll durch die exemplarische Formulierung „bring the glass to the table" in Beispiel 8.3.5 verdeutlicht werden. Für die beiden Funktionsparameter „the glass" und „the table" seien zutreffende Parameter-Kandidaten (Definition 8.3.3) durch den namensbasierten Ansatz gefunden worden. Beide der genannten Funktionsaufruf-Kandidaten K1 und K2 repräsentieren korrekte Abbildungen. Dennoch kann für `bring1::Method` einer der extrahierten Funktionsparameter („the table") nicht typ-sicher abgebildet werden, da in diesem Fall weniger abzubildende Methoden-Argumente (Definition 8.3.4) benötigt werden. Hierfür wird ein Strafterm eingeführt, der die Abdeckung der aus der Nutzereingabe extrahierten Funktionsparameter in die Bewertungsfunktion integriert. Dies wird durch den Subtrahenden P_{AE} in Gleichung 8.1 erreicht. So kann gewährleistet werden, dass der Kandidat K2 aus Beispiel 8.3.5 eine höhere Bewertung als K1 erhält. Weiterhin soll verhindert werden, dass dieser Strafterm zu hoch in die Gewichtung der Bewertungsfunktion miteinfließt. Es wird ein empirisch ermittelter Faktor von 0.3 festgelegt.

Beispiel 8.3.5: Abdeckung extrahierter Funktionsparameter

E1: „bring the glass to the table“ $\rightarrow E_P = 2$

K1: `bring1(bring1.what)`
K2: `bring2(bring2.what, bring2.where)`

Ziel: $S(\text{K2}) > S(\text{K1})$

Der Strafterm P_{AE} wird dabei wie folgt definiert: In Gleichung 8.3 wird zunächst die Differenz aus extrahierten Funktionsparametern E_P und abzubildenden Methoden-Argumenten M_P gebildet. Diese wird anschließend mit der Anzahl der extrahierten Funktionsparameter E_P normiert, sodass der Strafterm P_{AE} aus Gleichung 8.1 einen Wert von maximal Eins annimmt. Zusätzlich dazu wird das Minimum von P_{AE} auf Null gesetzt. Negative Werte wären möglich, falls ein Methoden-Individuum mehr Argumente benötigt, als in der Nutzereingabe angegeben beziehungsweise extrahiert werden konnten, also für M_P größer E_P. Dieser Fall soll anschließend mit der Parameter-Kandidaten-Gewichtung P_V abgedeckt werden.

$$P_{AE} = \frac{E_P - M_P}{E_P} \tag{8.2}$$

Es folgt die Definition der Gewichte für den zutreffenden Methoden-Kandidat M_T und die typ-sicheren Parameter-Kandidaten P_V aus Gleichung 8.1: Für den Methoden-Kandidaten M_T wird der Ähnlichkeitswert des Zeichenketten-Vergleichverfahrens aus der namensbasierten Abbildungssuche (siehe Abschnitt 8.3.2.1) für das gefundene, zutreffende Methoden-Individuum gewählt. Analog dazu wird für die mit der typbasierten Abbildungssuche gefundenen typ-sicheren Parameter-Kandidaten P_V die Summe der einzelnen Ähnlichkeitswerte eines zutreffenden Funktionsparameter-Kandidaten eingesetzt.
Um zusätzlich die Abdeckung der abzubildenden Argumente der Ontologie-Methode einfließen zu lassen, wird die Gewichtung P_{AA} eingeführt. Mit dieser Gewichtung soll erreicht werden, dass Funktionsaufruf-Kandidaten mit mehreren besetzten typ-sicheren Abbildungen auf die abzubildenden Methodenargumente besser bewertet werden, als typ-sichere Abbildungen, die nicht alle Methodenargumente besetzen. Im Optimalfall werden alle Argumente besetzt, der Kandidat gilt als vollständig abgebildet (Definition 8.3.6). Das Beispiel 8.3.6 soll diesen Sachverhalt an der exemplarischen Nutzeranweisung „fill the glass“ verdeutlicht werden. Es sei angenommen, dass für den Funktionsparameter „the glass“ ein typ-sicherer Parameter-Kandidat durch die typbasierte Suche gefunden worden ist. Weiterhin sei die Bewertung M_T für beider Methoden-Kandidaten `fill::Method` und `find::Method` auf Basis der Zeichenketten-Ähnlichkeitswerte gleich hoch. In diesem Fall soll der zweite Kandidat eine höhere Bewertung erhalten, da er von 1 von 1 abzubildende Methodenargumente mit typ-sicheren Parameter-Kandidaten besetzt. Der Funktionsaufruf gilt in diesem Fall sogar als vollständig abgebildet.

Beispiel 8.3.6: Abdeckung abzubildender Methodenargumente

E1: „fill the glass“ $\rightarrow E_P = 1$

K1: `fill(fill.what, fill.where)` $\rightarrow M_P = 2,\ P_V = 1$
K2: `find(find.what)` $\rightarrow M_P = 1,\ P_V = 1$

Ziel: $S(\text{K2}) > S(\text{K1})$

Um dieses Bewertungsverhalten zu erhalten, wird die Gewichtung P_{AA} aus Gleichung 8.1 eingeführt und in Gleichung 8.3 definiert. Dazu soll die Abdeckung der abzubildenden Argumente einer betrachteten Ontologie-Methode durch ein Verhältnis aus der Anzahl typ-

sicher abgebildeter Parameter-Individuen P_V und der Anzahl abzubildender Methodenargumente M_P gebildet werden. Können nicht alle Argumente des Methoden-Kandidaten durch einen Parameter-Kandidaten besetzt, das heißt typ-sicher abgebildetet werden, ist P_V kleiner M_P. Dadurch wird der Wert des Gewichts P_{AA} kleiner Eins. Das Maximum von P_{AA} erreicht den Wert Eins, sofern exakt alle Argumente der Methode vollständig gesetzt werden konnten, das heißt P_V gleich M_P. Besitzt eine Ontologie-Methode keine Argumente, so ist der Faktor P_{AA} Null. Dies führt dazu, dass der mittlere Teil der Bewertungsfunktion aus Gleichung 8.1 Null ergibt. Die Bewertung eines solchen Funktionsaufruf-Kandidaten bildet sich in diesem Fall aus dem Gewicht des Methoden-Kandidaten M_T und dem Strafterm P_{AE}.

$$P_{AA} = \frac{P_V}{M_P} \tag{8.3}$$

In Beispiel 8.3.7 wird die definierte Bewertungsfunktion auf die zuvor per namens- und typbasierter Abbildungssuche gefundenen Kandidaten (siehe Beispiel 8.3.4) angewendet. Dafür wird der Funktionsname `put` sowie zwei als zugehörig extrahierte Funktionsparameter betrachtet. Aus dem Funktionsnamen werden durch die namensbasierte Abbildungssuche drei zutreffende Methoden-Kandidaten gewonnen. Die darauffolgende typbasierte Suche liefert im dritten Schritt eine Menge bestehend aus vier Funktionsaufruf-Kandidaten. Anschließend können die einzelnen Werte pro Kandidat in die Bewertungsfunktion eingesetzt werden. Für den Kandidaten M1K1 ergibt sich die höchste Bewertung von 1.25. Er repräsentiert auch die tatsächliche korrekte Zuordnung von Programmbestandteilen zu Ontologie-Individuen in diesem Beispiel. Insbesondere erhält der Kandidat M1K2 eine schlechtere Bewertung, da er trotz der korrekt gefundenen Ontologie-Methode `put` eine nicht vollständige Argumentliste dieses Methoden-Individuums aufweist.

Beispiel 8.3.7: Bewertung von Abbildungen

Funktionsname und Funktionsparameter
F1 [put : coffee mug, dispenser] $\rightarrow E_P = 2$

Methoden-Kandidaten
M1 `put` ($M_T = 1.0$) : [`put.what`, `put.where`] $\rightarrow M_P = 2$
M2 `cut` ($M_T = 0.8$) : [`cut.what`] $\rightarrow M_P = 1$
M3 `putDown` ($M_T = 0.6$) : [ ] $\rightarrow M_P = 0$

Funktionsaufruf-Kandidaten
M1K1 [`put.what` : `CoffeeCup` ($P_V = 0.9$),
`put.where` : `CoffeeMachine.Dispenser` ($P_V = 0.6$)] $\rightarrow P_V = 2$
M1K2 [`put.what` : `CoffeeCup` ($P_V = 0.9$)] $\rightarrow P_V = 1$
M2K1 [`cut.what` : `CoffeeCup` ($P_V = 0.9$)] $\rightarrow P_V = 1$
M3K1 [`putDown`] $\rightarrow P_V = 0$

Bewertung der Funktionsaufruf-Kandidaten
```
    S = 0.5*MT + 0.5*(PV         *PV/MP) - 0.3*(EP - MP)/EP)
SM1K1 = 0.5*1.0 + 0.5*((0.9 + 0.6)*(2/2)) - 0.3*((2-2)/2) = 1.25
SM1K2 = 0.5*1.0 + 0.5*((0.9)      *(1/2)) - 0.3*((2-2)/2) = 0.725
SM2K1 = 0.5*0.8 + 0.5*((0.9)      *(1/1)) - 0.3*((2-1)/2) = 0.55
SM3K1 = 0.5*0.6 + 0.5*( 0.0       * 0   ) - 0.3*((2-1)/2) = 0.15
```

8.3.3 Persistierung gefundener Abbildungen

Der letzte Schritt des entwickelten Werkzeuges in dieser Arbeit muss es sein, die gewonnen Ergebnisse für weitere Agenten des PARSE-Projektes abzuspeichern und zugreifbar zu machen. Die gefundenen Ontologie-Abbildungen stellen hierbei nicht nur eine Verknüp-

fung zwischen Sprache und Wissensrepräsentation her, vielmehr entspricht das Ergebnisprodukt dieser Arbeit einem Skript aus Zielsystem-Bibliotheksaufrufen beziehungsweise einer Ontologie-Funktionserweiterung.

Wie bereits in der zugehörigen Analyse in Abschnitt 7.3.3 beschrieben, wird von einer direkten Modifikation der Domänenontologie abgesehen. Der Grund dafür ist, dass dieses Ergebnisprodukt in einem Nachverarbeitungsschnitt der PARSE-Verarbeitungskette um zusätzliche Kontrollstrukturen weiter ergänzt werden können soll. Deshalb müssen die Ergebnisse dieser Arbeit in der globalen Graphdatenstruktur des Projekts PARSE zwischengespeichert werden. Dies wurde auch bereits in Abschnitt 7.1 in der Herausforderung (H 3.5) für diese Arbeit definiert. Ein weiterer Vorteil dieser Vorgehensweise ist, dass zusätzlich zur erhaltenen Sequenz aus Funktionsaufrufen bestehend aus den Ontologie-Individuen, die ursprüngliche Verbindung zu den zugehörigen Satzbestandteilen der Nutzereingabe erhalten bleiben kann. Dies kann dadurch erreicht werden, indem die Graphknoten, welche die Wörter der Nutzereingabe beinhalten, mit den Abbildungsinformationen verknüpft werden. Um zusätzlich dazu die, in Abschnitt 8.2 innerhalb einer Datenstruktur modellierten, Ergebnisse der Sprache-zu-Programmrepräsentation nicht zu verlieren, wird eine daran angelehnte Graphdatenstruktur entworfen. Diese soll anschließend in den PARSE-Graph integriert und mit den zugehörigen Satzbestandteilen der Nutzereingabe verknüpft werden.

Anstatt nur die beste gefundene Abbildungen eines jeden Funktionsaufrufs zu speichernwerden die besten drei Kandidaten ausgewählt. Diese sollen dabei unabhängig von ihrer Bewertung, das heißt ohne Aussortierung mit einem festgelegten Schwellwert, in den Graphen integriert werden. So kann eine größere Ergebnismenge präsentiert werden, die mit einer höheren Wahrscheinlichkeit eine typ-sichere Abbildung enthält. Dazu ist es denkbar, das andere Agenten die präsentierten Informationen weiterverarbeiten und aussortieren könnten. Für die Persistierung sind deshalb folgende Informationen aus dem Ergebnisprodukt dieser Arbeit relevant:

- Methodenkopf
  - synthetisierter Methodenname
  - typ-sicher abgebildete Ontologie-Individuen als Parameter (Definition 8.3.5)
- Methodenrumpf / Skript
  - Nummer eines Funktionsaufrufs
  - zutreffendes Methoden-Individuum eines Funktionsaufrufs (Definition 8.3.3)
  - Paare aus Methodenargument und typ-sicher abgebildetem Ontologie-Individ.
  - Bewertung des abgebildeten Funktionsaufrufs
  - Rang des abgebildeten Funktionsaufrufs (Top 3)
- zugehörige Satzglieder

Im nachfolgenden Kapitel der Implementierung soll anschließend eine Graphdatenstruktur erstellt werden, welche diese Informationen in Form von Knoten- und Kantenattributen enthält. Es gilt zu beachten, dass eine beliebige Anzahl an Funktionsaufrufen modelliert werden können muss. Überdies müssen einerseits Informationen zu unterschiedlichen Aktionen und andererseits Informationen zur selben Aktion bestehend aus einer Rangfolge der drei best-bewertetsten Kandidaten gespeichert werden können. Diese müssen wiederum jeweils unterschiedliche Anzahlen an Parametern annehmen können. Hierfür

würde sich eine Menge von Knoten für die Ontologie-Parameter sowie für die Ontologie-Methoden anbieten, welche die obigen Werte in Form von Knotenattributen halten. Um die Parameter eines Funktionsaufrufs zu kapseln, könnte ein gemeinsamer, übergeordneter Eltern-Knotentyp pro Funktionsaufruf erstellt werden. Zuletzt könnte die Verknüpfung der Abbildungen mit den Satzgliedern durch eine Kante von Methoden- beziehungsweise Parameterknoten zu den Wortknoten des Graphen umgesetzt werden.

9 Implementierung

In diesem Kapitel wird die Implementierung der einzelnen Teilaufgaben (siehe Abbildung 5.1) dieser Arbeit vorgestellt. Es soll ein Werkzeug entwickelt werden, welches die Synthese von Methodendefinitionen in natürlichsprachlichen Äußerungen ermöglicht. Dazu werden die Sprachverständniskomponenten des PARSE-Projekts um einen den Agent `MethodSynthesizer` erweitert. Als Grundlage für die Implementierung dient hierbei das entworfene Vorgehen aus Kapitel 8. Auch der konzipierte Programmablauf wurde bereits zu Beginn dieses Kapitels in Abbildung 8.1 illustriert. Er gliedert die unterschiedlichen Implementierungsschritte in die drei Teilaufgaben der Lehrsequenz-Erkennung aus Abschnitt 8.1, des Aufbaus von Methodendefinitionen aus Abschnitt 8.2 und der Ontologie-Abbildung aus Abschnitt 8.3. Diese Schritte werden im Folgenden beschrieben.

9.1 Lehrsequenz-Erkennung

Die erste Teilaufgabe der Lehrsequenz-Erkennung dieser Arbeit beschäftigt sich mit der Klassifikation von Lehrsequenzen in natürlichsprachlichen Eingaben. Die Implementierung des in Abschnitt 8.1 konzipierten Lösungsansatzes wird in diesem Abschnitt vorgestellt. Sie bildet den ersten Teil der Implementierung, welcher im Programmablauf des Agenten in Abbildung 8.1 durch die erste Kartei visualisiert wird.

Die Beschreibung des implementierten Ansatzes wird analog zum Entwurf in drei Abschnitte für die drei Stufen der hierarchischen Klassifikation untergliedert (Abschnitt 7.3.1.2). Als Datengrundlage für die Implementierung des Klassifikationsansatzes dient der unverarbeitete, aus der Vorstudie gewonnene, Datensatz. In Tabelle 9.1 wird dieser für drei exemplarische Eingabesequenzen abgebildet.

9.1.1 Detektion von Lehrsequenzen

Für die Detektion von Lehrsequenzen wird ein binärer Klassifikationsansatz auf Basis maschinellen Lernens implementiert. Dieser soll erkennen, ob der Nutzer in der Eingabe eine Lehrsequenz (siehe Definition 5.1.1) verbalisiert oder stattdessen Sequenz von Befehlen zur sofortigen Ausführung diktiert. Im ersten Abschnitt werden hierzu die Programmierumgebung sowie benötigte Bibliotheken und Werkzeuge der Implementierung vorgestellt. Die Implementierung des Klassifikators erfolgt unabhängig vom Projekt PARSE und wird erst später in den Agenten integriert. Im zweiten und dritten Abschnitt wird die Generierung und Vorverarbeitung von Trainingsinstanzen aus den Daten nach dem

Tabelle 9.1: Auszug des Datensatzes aus der Vorstudie

	original_input	scenario
2178	To ring a beverage, open the fridge and select one of te beverages inside, pour it into one of the glasses on the kitchen counter and hand the glass over to the person.	3
2509	collect cutlery from cupboard, bring them to the table and place down neatly	4
999	You have to place the cup under the dispenser and press the red button to make coffee.	2

Konzept aus Abschnitt 8.1.1 vorgestellt. Daraufhin wird ein Referenzwert als Ausgangssituation des binären Klassifikationsproblems berechnet. Anschließend wird im darauffolgenden Absatz die Modellbildung klassischer Lernverfahren sowie Verfahren, die neuronalen Netzen verwenden, beschrieben. Diese implementierten Modelle werden anhand der erzielten Modellgüte auf den Validierungsdatensätzen iterativ erweitert, um eine Verbesserung zu erzielen. Im Absatz der Modellevaluation werden die entworfenen Modelle abschließend auf dem unbekannten Testdatensatz ausgewertet.

9.1.1.1 Programmierumgebung

Für die Implementierung der maschinellen Lernansätze wird eine Python-Umgebung (Version 3.6) mit den folgenden, für Datenverarbeitung und Datenanalyse optimierten, Modulen erstellt:

- *scikit-learn*: Bibliothek mit Werkzeugen zur Datenverarbeitung sowie für maschinelles Lernen
- *Jupyter notebook*: Grafische Ausführungsumgebung für die Programmiersprachen Julia, Python und R
- *pandas*: Bibliothek mit Datenstrukturen und Operatoren für den vereinfachten Zugriff auf tabellenbasierte Datensätze
- *keras*: Bibliothek für den Einsatz von Lernverfahren auf Basis neuronaler Netze
- *tensorflow*: Rahmenarchitektur zur datenstromorientierten Programmierung von neuronalen Netzen als Unterbau (engl. *backend*) zu Keras

9.1.1.2 Generierung von Trainingsinstanzen

Die Trainingsinstanzen für die binäre Klassifikation werden nach dem in Abschnitt 8.1.1 beschriebenen Vorgehen aus dem Datensatz der Vorstudie erstellt. Im Schritt der Korpusvorverarbeitung werden die Schreibfehler in den Nutzereingaben des Datensatzes korrigiert, Grammatikfehler werden jedoch nicht behoben. Zusätzlich dazu werden in Abschnitt 8.1.1.1 weitere Vorverarbeitungsschritte, wie beispielsweise das Entfernen von Satzzeichen und Disfluenzen, sowie das Ausschreiben von Ziffern festgelegt. Eine Auflistung aller getätigten Schritte wird dort bereits in Tabelle 8.2 präsentiert. Die Vorverarbeitung des Korpus wird mit der Sprachverarbeitungs-Softwarebibliothek *SpaCy*[1] umgesetzt.

Anschließend werden die Trainingsinstanzen für Stufe 1 nach der Definition 8.1.1 aus Abschnitt 8.1.1.2 entworfen. Zur Erstellung der Ausgabeklassen wird das Annotationswerkzeug *Gate*[2] verwendet, welches das aufwändige manuelle Erstellen der Ausgabeklassen visuell unterstützt.

[1] Quelle: `https://spacy.io/`, zuletzt besucht am 17.06.19
[2] Quelle: `https://gate.ac.uk/wiki/jape-repository/annotations.html`, zuletzt besucht am 17.06.19

Tabelle 9.2: Auszug des annotierten binären Klassifikationsdatensatzes

	input	scenario	output
2178	to bing a beverage open the fridge and select one of the beverages inside pour it into one of the glasses on the kitchen counter and hand the glass over to the person	3	1
2509	collect cutlery from cupboard bring them to the table and place down neatly	4	0
999	you have to place the cup under the dispenser and press the red button to make coffee	2	1

Tabelle 9.3: Struktur der binären Klassifikationsdatensatzes

Spaltenname	Beschreibung
input	Eingabesequenz
scenario	Szenario der Vorstudie, nummeriert von 0 bis 4[3]
output	Ausgabeklasse
case	Markierung bei unklarer Interpretation der Eingabesequenz

Der annotierte Datensatz wird anschließend als CSV-Datei in `study_binary_labeled.csv` gespeichert, wobei die Spaltenbezeichnungen in Tabelle 9.3 erklärt werden. Die Spalte `case` dient der reinen Dokumentation unklarer Fälle bei Interpretationen der Eingabesequenzen. Dies wurde in der Auswertung der Vorstudie in Abschnitt 6.5.2 erläutert sowie deren Umgang als Herausforderung (H 1.4) in Abschnitt 8.1.1.2 festgelegt. Für die nachfolgend implementierten Modelle werden ausschließlich die Spalten `input` und `output` betrachtet. Die Tabelle 9.2 präsentiert drei exemplarische Trainingsinstanzen aus dem Datensatz nach diesem Vorverarbeitungsschritt. Die vergebenen Ausgabeklassen und deren Bedeutung ist in Abschnitt 8.1.1.2 in Tabelle 7.1 nachzulesen.

9.1.1.3 Vorverarbeitung der Trainingsinstanzen

Für den Klassifikator müssen die Eingabemerkmale und Ausgabeklassen der Trainingsinstanzen in ein für den Klassifikator verständliches Format überführt werden. Diese Vorverarbeitungsschritte der Trainingsinstanzen werden im Folgenden beschrieben.

Als erstes werden die Eingabemerkmale verarbeitet. Dazu werden die in Abschnitt 8.1.1.3 vorgestellten numerischen Textrepräsentationen Bag-Of-Words und Wortvektoren erstellt. Als erstes müssen die Eingaben vorab tokenisiert, das heißt, der Fließtext in Folgen von Worteinheiten (engl. *token*) zerlegt werden. In vielen Ansätzen zur Klassifikation von Texten mittels Verfahren des maschinellen Lernens wird nach dem Schritt der Tokenisierung das Löschen von Stopwörtern vorgenommen. Es wird jedoch entschieden, diese Stopwörter nicht durch die übliche Stopwortlisten zu entfernen, da beispielsweise die als Stopwort zählenden Präpositionen *to* oder *by* häufig als Signalwort für Lehrsequenzen gelten könnten (siehe Abschnitt 6.5.2). Anschließend kann die tokenisierte Eingabe optional lemmatisiert werden. Dabei werden die morphologischen Varianten eines Wortes, zum Beispiel der Deklination oder Konjugation, auf ihren gemeinsamen Wortstamm zurückgeführt. Ein Beispiel für eine solchen Verarbeitungsschritt natürlicher Sprache wird in den Grundlagen in Abschnitt 2.2.1.3 präsentiert. Für diesen Sprachverarbeitungsschritt wird wiederum die Programmbibliothek *SpaCy* verwendet. Zuletzt wird das Vokabular erstellt, welches einen Wortindex über alle unterschiedlichen Wörter des Datensatzes bildet. Jedes Wort einer Eingabesequenz kann somit einer eindeutigen Zahl im Korpus zugeordnet werden. Die beschriebenen Vorverarbeitungsschritte der Trainingsinstanzen werden in Beispiel 9.1.1 dargestellt.

Beispiel 9.1.1: Vorverarbeitung der Trainingsinstanzen

Eingabesequenz
start the coffee machine by pressing the button this is how you make coffee

Tokenisierung
start, the, coffee, machine, by, pressing, the, button, this, is, how, you, make, coffee

Lemmatisierung (optional)
start, the, coffee, machine, by, **press**, the, button, this, **be**, how, you, make, coffee

Vokabularbildung
{the : 1, coffee : 2, start : 3, make : 4, ... }

Anschließend können die beiden Wortrepräsentationen erstellt werden. Für die binäre Klassifikation soll zunächst das einfache Repräsentationsmodell Bag-Of-Words (siehe Abschnitt 2.4.3.1) eingesetzt werden. Mit Hilfe des erstellen Vokabulars kann eine Eingabesequenz durch einen numerischen Vektor repräsentiert werden. In Abschnitt 8.1.1.3 wird entschieden, die der Anzahl an Vorkommen eines Wortes pro Eingabesequenz in einem Vektor zu kodieren. Es entsteht eine Matrix-Repräsentation, deren Spalten die Indizes des Wortindexes und deren Reihen die Anzahl des Tokens pro Eingabe bilden. Die Reihen dieser Matrix entsprechen dabei einem Bag-Of-Words-Modell. Eine Eingabesequenz wird mit dieser Repräsentationsform schließlich durch einen Vektor entsprechend der Länge des Vokabulars numerisch repräsentiert. Zusammenfassend werden in Beispiel 9.1.2 die beschriebenen Schritte zur Erstellung der Bag-Of-Words anhand einer exemplarischen Eingabe veranschaulicht.

Beispiel 9.1.2: Erstellung der Bag-Of-Words-Repräsentation

Eingabesequenzen
E1: press the button this is how you make the coffee
E2: hi make coffee please

Vokabular
{the : 1, coffee : 2, make : 3, hi : 4, press : 5 ... }

wobei das Vokabular hier nur über E1 und E2 (statt dem Korpus) gebildet ist

Bag-Of-Words

Wort	the	coffee	make	hello	press	button	please	this	be	how	you
Index	1	2	3	4	5	6	7	8	9	10	11
E1	2	1	1	0	1	1	0	1	1	1	1
E2	0	1	1	1	0	0	1	0	0	0	0

wobei die Zahlen den ungewichteten Wortanzahlen einer Eingabesequenz entsprechen

Je nach Größe des Korpus wächst das Vokabular und somit auch die Länge der Bag-Of-Words-Vektoren. Außerdem ist die erstellte Matrix häufig dünn besetzt. Diese Aspekte machen den Einsatz von Bag-Of-Words-Modellen ineffizient. Daher wird mit den Wortvektoren (siehe Abschnitt 2.4.3.2) eine zweite Repräsentationsform von Texteingaben eingesetzt. Hierbei wird jedes Wort auf einen dichten Vektor, das sogenannte Word Embedding, abgebildet. Für die Verwendung von Wortvektoren wird der bekannte Wortvektoralgorithmus CBOW [MCCD13] auf das Vokabular angewendet.
Zusätzlich erfolgt der Einsatz von bereits vortrainierten Fasttext-Wortvektoren. Dafür wird der Wortvektordatensatz `wiki-news-300d-1M`[4] verwendet. Er enthält eine Million 300-

[4]Quelle: `https://fasttext.cc/docs/en/english-vectors.html`, zuletzt aufgerufen am 30.05.2019

dimensionale Wortvektoren die auf Wikipedia und Nachrichtendatensätzen bestehend aus 16 Millionen unterschiedlichen Tokens trainiert wurden. Es ist daher anzunehmen, dass den vortrainierten Wortvektoren ein komplexeres Sprachmodell (engl. *language model*) zugrundeliegt und somit mehr semantisches Wissen repräsentieren, als die auf dem begrenzten Datensatz der Vorstudie selbst trainierten Wortvektoren. Die berechnete Übereinstimmung dieser Fasttext-Wörter mit dem hiesigen Korpus liegt bei 99,6%, wobei die einzigen unbekannten Wörter Eigennamen wie „armar“ oder „appollinaris“ sind.
Durch das erstellte Vokabular kann eine Eingabe durch eine Folge von numerischen Werten, den Wortindizes, repräsentiert werden. Bei Verwendung von Wortvektoren referenziert dieser Index anschließend nicht mehr ein Wort, sondern den entsprechenden Wortvektor. Eine Eingabesequenz wird mit dem Wortvektor-Modell schließlich durch eine Folge von Vektoren entsprechend der Länge der Sequenz numerisch repräsentiert.
Dies hat zur Folge, dass die Eingaberepräsentationen je Satz unterschiedlich lang sind. Da die verwendeten Klassifikatoren ausschließlich Eingaben gleicher Länge verarbeiten können, müssen diese deshalb vorab auf eine einheitliche Länge aufgefüllt werden. Hierfür wird die Länge der Eingaben, das heißt die Wortanzahl einer Sequenz, auf 135 Wörter festgelegt. Bei Betrachtung der Wortanzahlen der Eingaben aus dem erhobenen Datensatz fiel auf, dass über 99.9% der Daten eine Sequenzlänge von unter 135 Wörtern aufweisen. Daher wird dies als obere Schranke der Eingabelänge gewählt. Eingabesequenzen, die diese Länge überschreiten, werden gekürzt. Für das Auffüllen von Einheiten wird standardmäßig die Zahl 0 als Symbol gewählt. Für die Wortvektorrepräsentation wird jedes Element der aufgefüllten, indizierten Eingabe wiederum durch einen 300-dimensionalen Vektor ersetzt. Im einen Fall sind die eigens, aus den Korpusdaten erstellten Wortvektoren, andererseits sind dies die vortrainierten Fasttext-Vektoren. Somit verwendet das neuronale Netz für das Training eine 135-elementige Menge aus Vektoren pro Eingabesequenz. Zusammenfassend werden in Beispiel 9.1.3 die beschriebenen Schritte zur Erstellung der Wortvektorrepräsentationen anhand einer exemplarischen Eingabe veranschaulicht.

Beispiel 9.1.3: Erstellung der Wortvektor-Repräsentation

Eingabesequenz
place a cup under the coffee dispenser before starting it you can find cups next to the machine start the machine by pressing the red button this is how you prepare coffee

Wortindizierung
23, 6, 17, 44, 2, 8, 41, 115, 139, 12, 4, 64, 76, 96, 36, 3, 2, 13, 62, 2, 13, 85, 98, 2, 37, 34, 84, 32, 54, 4, 58, 8

wobei 8 der Index im Vokabular für *coffee* ist

Padding
23, 6, 17, 44, 2, 8, 41, 115, 139, 12, 4, 64, 76, 96, 36, 3, 2, 13, 62, 2, 13, 85, 98, 2, 37, 34, 84, 32, 54, 4, 58, 8, 0

wobei die Eingabelänge vor dem Padding 33, nach dem Padding 135 ist

Wortvektoren
$\mathrm{v} = (\ \mathrm{v}_1\ \mathrm{v}_2\ \mathrm{v}_3\ \ldots\ \mathrm{v}_{135}\ )^{\mathrm{T}}$

wobei v_i einem 300-dimensionalen Vektor pro Wort entspricht

Als zweites müssen zur Vorverarbeitung der Trainingsinstanzen auch die Ausgabeklassen auf numerische Werte abgebildet werden. In Fall einer binären Klassifikation ist dies ein-

Tabelle 9.4: Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 1

	randomisiert	szenarienbezogen
Gesamt (Klasse TS + NO_TS)	3410	3410
Train	2182	2092
Validate	546	524
Test	682	794

Tabelle 9.5: Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 1

	randomisiert	szenarienbezogen
Train + Validate	2728	2616
Klasse TS	1562	1431
Klasse NO_TS	1166	1185
Genauigkeit	57,3%	54,7%

fach, da pro Eingabesequenz nur ein Ausgabewert entsprechend einer der beiden Klassen klassifiziert werden muss. Deshalb kann die Ausprägung der beiden Klassen durch die Zahl 0 beziehungsweise 1 repräsentiert werden.

Nach der Vorverarbeitung der Trainingsinstanzen kann das Training des Klassifikators begonnen werden. Dafür werden diese in Train-, Validate- und Test-Datensätze aufgeteilt. Die trainierten Modelle werden anschließend für die in Abschnitt 8.1.1.4 beschrieben randomisierte sowie die szenarienbezogene Aufteilung getrennt voneinander evaluiert. Die Größe der Trainingsinstanzen nach der Aufteilung der beiden Train-Validate-Test-Varianten wird in Tabelle 9.4 dargestellt. Für die szenarienbezogene Aufteilung wird das Szenario *Greet someone* (siehe Abschnitt A.1) exemplarisch als Testdatensatz ausgewählt.

9.1.1.4 Referenzwert

Bevor das binäre Klassifikationsproblem mit maschinellen Lernverfahren umgesetzt wird, muss die Ausgangssituation auf dem Datensatz bestimmt werden. Dieser Status Quo, der sogenannte Referenzwert (engl. *baseline*), dient als Vergleichswert für kommende Modellauswertungen, um die Verbesserung der Resultate verschiedener Modellarchitekturen zu messen. Es gibt zwei verbreitete Ansätze, diesen Referenzwert zu berechnen: die Zufallsvorhersage und der Zero-Rule-Algorithmus. Während im ersten Ansatz eine zufällige Klasse für jede Testinstanz vorhergesagt wird, entscheidet der Zero-Rule-Algorithmus auf Basis der häufigsten Klasse (engl. *majority class*) im Trainingsdatensatz. Das bedeutet, falls ein Datensatz eines binären Entscheidungsproblems 80 Trainingsinstanzen der Klasse „0“ und 20 Instanzen der Klasse „1“ enthält, wird der Zero-R-Klassifikator die häufiger vergebene Klasse „0“ für alle Testdaten hervorsagen und somit eine Referenzwert-Genauigkeit 80% erreichen. Der Zero-Rule-Algorithmus bildet daher einen besseren Ansatz zur Berechnung des Referenzwertes und wird im Folgenden als Ausgangslage für kommende Modellauswertungen der binären Klassifikation verwendet.
Es ergeben sich die folgende Genauigkeitswerte in Tabelle 9.5, wobei für die Berechnung der Referenzwert-Genauigkeit Trainings- und Validierungsdatensatz zusammen betrachtet werden. Weiterhin lässt sich erkennen, dass beide Klassen des Klassifikationsproblems von Stufe 1 relativ ausgeglichen im Datensatz repräsentiert sind. Die Klasse der Lehrsequenzen (TS) ist in beiden Train-Test-Aufteilungen mit 57,3% beziehungsweise 54,7% häufiger im Datensatz enthalten. Diese Verteilung entspricht den erreichten Genauigkeitswerten mit dem Zero-Rule-Algorithmus.

9.1.1.5 Modellbildung: Klassische Lernverfahren

Als erster Ansatz des binären Klassifikationsproblems werden klassische maschinelle Verfahren betrachtet (siehe Abschnitt 2.4.1). Dazu wird ein Entscheidungsbaum (siehe Abschnitt 2.4.1.1) als erstes Modell implementiert.
Der Quelltext für dieses Modell wird beispielhaft für die Implementierung aller folgenden klassischen Lernverfahren in Quelltextausschnitt 9.1 präsentiert. Die Eingabedaten werden hierbei in der Praxis oft mit `X`, die Ausgabeklassen mit `y` angegeben. Für die klassischen Lernverfahren werden die Eingabesequenzen durch Bag-Of-Words-Vektoren (siehe Abschnitt 2.4.3.1) repräsentiert. Als Eingabe dienen zum einen lemmatisierte sowie nicht-lemmatisiert Eingabesequenzen (siehe Abschnitt 2.2.1.3). Diese werden in Zeile 5 bis 9 mit dem `CountVectorizer` der Machine Learning Bibliothek *sklearn* erstellt. Anschließend erfolgt die Instanziierung des eigentlichen Lernverfahrens, dem Entscheidungsbaum `DecisionTreeClassifier`. Mit der Methode `Model.fit` in Zeile 15 wird das Modell auf den Trainingsdaten trainiert. Anschließend kann mit `Model.predict` ab Zeile 20 auf dem Validierungsdatensatz die Genauigkeit nach Gleichung 2.5 der Klassifikation evaluiert werden.

```
1  from sklearn.feature_extraction.text import CountVectorizer
2  from sklearn.preprocessing import LabelEncoder
3  from sklearn.tree import DecisionTreeClassifier
4  from sklearn.metrics import accuracy_score
5
6  # Bag-Of-Words Repraesentation der Eingabe
7  vect = CountVectorizer(ngram_range=(1, 1)).fit(X_train)
8  X_train = vect.transform(X_train)
9  X_val = vect.transform(X_val)
10
11 # Numerische Repraesentation der Ausgabe
12 le = LabelEncoder().fit(["TS", "NO_TS"])
13 y_train = le.transform(y_train)
14
15 # Training auf Trainingsdaten
16 dt = DecisionTreeClassifier()
17 dt.fit(X_train, y_train)
18
19 # Vorhersage auf Validierungsdaten
20 y_pred = dt.predict(X_val)
21 acc = accuracy_score(y_val, y_pred)
```

Quelltextausschnitt 9.1: Quelltext zur Stufe-1-Klassifikation mittels Entscheidungsbaum

Randomisierte Train-Validate-Test-Aufteilung

Als erstes wird der implementierte Entscheidungsbaum auf den randomisierten Datensätzen (siehe Tabelle 9.4) angewandt. Es ergibt sich eine gemessene Genauigkeit von 81.9% auf den Validierungsdaten mit tokenisierten Eingaben. Die Genauigkeit mit lemmatisierten Eingabesequenzen ist etwas höher. Sie beträgt 82.2%. Im Vergleich zum gemessenen Referenzwert von 57.3% in Tabelle 9.5 ist dies eine Verbesserung von 24.9 Prozentpunkten.
In Quelltextausschnitt 9.1 in Zeile 7 kann mit dem Parameter `ngram_range` des `CountVectorizers` zusätzlich die Vokabularbildung von einzelnen Wörtern, das heißt Unigrammen, auf N-Gramme (siehe Abschnitt 2.2.1.4) erweitert werden. Daher wird das Modell für `ngram_range` von 2 bis 5, die zusätzlich zu den Unigrammen in das Vokabular aufgenommen werden, getestet. Die Vokabulargröße wächst dabei exponentiell. Bei Unigrammen sind es 1073 Einträge im Vokabular, mit 1- bis 5-Grammen schon 97829 Einträge. Tabelle 9.6 veranschaulicht die erreichten Genauigkeiten auf den Validierungsdaten unterschiedlicher Konfigurationen. Es lässt sich erkennen, dass bei Verwendung einer `ngram_range` von 1

Tabelle 9.6: Vergleich von N-Gramm-Konfigurationen des Entscheidungsbaums

N-Gram	1	2	3	4	5
Genauigkeit	82.2	88.5	88.1	89.0	88.9
Vokabulargröße	1073	9836	29819	59908	97829

Tabelle 9.7: Konfusionsmatrix des Entscheidungsbaums

		Vorhersage	
		NO_TS	TS
Klasse	NO_TS	209	30
	TS	33	274

bis 4 die besten Ergebnisse erzielt werden. Dieses Modell verbessert das unparametrisierte Modell des Entscheidungsbaumes um weitere 7 Prozentpunkte von 82.2% auf 89.2%. Zusammenfassend zeigt Ergebnis bei Verwendung eines Entscheidungsbaumes eine deutliche Verbesserung zum Referenzwert auf. Bei Betrachtung der Konfusionsmatrix (engl. *confusion matrix*) des Ergebnisses der bisher besten Modellkonfiguration in Tabelle 9.7, zeigt sich, dass die Anzahl der fälschlich zugewiesenen Klassen mit 33 beziehungsweise 30 von 546 Validierungsinstanzen etwa 11% beträgt.

Szenarienbezogene Train-Validate-Test-Aufteilung

Als nächstes wird das Modell mit der szenarienbezogene Datensatzaufteilung (siehe Tabelle 9.4) trainiert. Hierbei erzielt der Entscheidungsbaum bei Verwendung eines Vokabulars aus reinen lemmatisierten Unigrammen 86.8% Genauigkeit auf dem Validierungsdatensatz. Die Genauigkeit auf den Validierungsdaten auf tokenisierten Eingaben ist wiederum geringfügig schlechter mit 86.2%. Bei unterschiedlicher Konfiguration des `ngram_range`-Parameters ergibt sich diesmal die beste Genauigkeit mit 92.2% bei Verwendung von 1- bis 3-Grammen. Diese Genauigkeitswerte sind in Tabelle 9.8 aufgetragen.

Zusammenfassend zeigt Ergebnis bei Verwendung eines Entscheidungsbaumes wiederum eine deutliche Verbesserung zum Referenzwert auf. Die zusätzliche Lemmatisierung der Eingaben scheint überdies zu einer geringfügigen Verbesserung der Modellergebnisse zu führen. Die Ergebnisse fallen zudem etwas besser aus als bei der randomisierten Verteilung. Trotz dieser besseren Ergebnisse auf den Validierungsdaten ist in der Modellevaluierungsphase eine deutlich schlechtere Genauigkeit als bei der randomisierten Train-Validate-Test-Aufteilung zu erwarten. Das liegt daran, dass im Falle der szenarienbezogenen Datentrennung der Testdatensatz zusätzlich disjunkt zum Szenario der Trainings- und Validierungsdaten ist.
Bei Betrachtung der Konfusionsmatrix des Ergebnisses der bisher besten Modellkonfiguration in Tabelle 9.10 werden wiederum nur 8.5% der Validierungsinstanzen fälschlich klassifiziert. Diese Fehlklassifikationen gilt es im Folgenden unter Einsatz von anderen Lernverfahren zu verringern. Deshalb werden neben dem Entscheidungsbaum auch weitere Verfahren maschinellen Lernens, wie die Modelle Naïve Bayes, Logistische Regression, Random Forest und Support-Vektor-Machines, auf das binäre Entscheidungsproblem angewandt. Diese werden in den Grundlagen in Abschnitt 2.4.1 unter der Kategorie der klassischen Verfahren des maschinellen Lernens vorgestellt. Die erzielten Genauigkeitswerte der einzelnen Modelle sind in Tabelle 9.9, wiederum für beide Aufteilungen der Trainingsinstanzen, aufgetragen. Die besten N-Gramm-Konfigurationen des Vokabulars beider Train-Validate-Test-Aufteilungen werden für die folgenden Modelle übernommen. Es fällt auf, dass für beide Trainingsdaten-Trennungen die Logistische Regression besonders gut abschneidet. In der randomisierten Verteilung beträgt die erzielte Genauigkeit auf dem Validierungsdatensatz 92.7%, auf dem szenarienbasierten Datensatz sogar 92.9%. Ob diese

Tabelle 9.8: Vergleich von N-Gramm-Konfigurationen des Entscheidungsbaums

N-Gram	1	2	3	4	5
Genauigkeit	86.3	91.8	92.2	92.2	92.0

Tabelle 9.9: Erzielte Genauigkeiten auf den Validierungsdaten der Stufe-1-Klassifikation mit klassischen Lernverfahren

	randomisiert	szenarienbezogen
Referenzwert	57.3	54.7
Entscheidungsbaum	89.0	92.2
Random Forest	92.1	90.1
Support Vektor Machines	84.8	86.1
Naïve Bayes	77.1	80.9
Logistische Regression	92.7	92.9

Genauigkeit auch auf den Testdatensätzen erreicht wird, wird in der Modellevaluation in Abschnitt 9.1.1.7 geklärt. Für diese Evaluation wird das beste Modell der Logistischen Regression ausgewählt.

9.1.1.6 Modellbildung: Verfahren auf Basis neuronaler Netze

Als zweite Lösungsmethode des binären Klassifikationsproblems mittels maschinellem Lernen werden neuronale Netze betrachtet. Dazu wird ein einfaches künstliches neuronales Netz (ANN, siehe Abschnitt 2.4.2.1) als erstes Modell implementiert. Im vorherigen Abschnitt hat das logistische Regressionsmodell als bester Klassifikator abgeschnitten. Daher wird versucht, diesen Ansatz auf ein neuronales Netz zu erweitern. Die Struktur einer logistischen Regression kann dabei als neuronales Netz ohne verdeckte Schicht (engl. *hidden layer*) und einem Ausgangsknoten angesehen werden. Die Abbildung 9.1 soll diese Beziehung veranschaulichen.

Die Implementierung des auf neuronale Netzarchitekturen abgebildeten Regressionsmodells wird in Quelltextausschnitt 9.2 vorgestellt. Die erste und einzige Schicht ab Zeile 8 besteht aus einer vollvermaschten dichten Schicht, dem sogenannten *Dense-Layer*. Als Eingabe wird die mit dem `CountVectorizer` verarbeiteten Trainingsinstanzen, das heißt Vektoren aus Wortanzahlen, verwendet. Es werden zwei Varianten betrachtet: lemmatisierte und nicht-lemmatisierte Eingabesequenzen (siehe Abschnitt 2.2.1.3). Zur Bildung dieser Vektoren benötigt die Schicht als Eingabedimension die Größe des Vokabulars. Weiterhin muss die Ausgabedimension der Netzschicht angegeben werden. Diese soll einen einzelnen Wert, welche für die vorhergesagte Klasse steht, zurückgeben und wird daher auf 1 gesetzt. Außerdem muss die Aktivierungsfunktion des Ausgabeknotens für die binäre Klassifikation auf die mathematische Sigmoid-Funktion gesetzt werden. Diese bildet Eingaben auf kontinuierliche Werte zwischen 0 und 1 ab, wobei Werte näher bei 0 für die Klasse NO_TS, Werte nahe an 1 auf die Klasse TS abgebildet werden.
Anschließend erfolgt die Modellkonfiguration mit `model.compile`, mit Hilfe der die im Training verwendete Fehlerfunktion (engl. *loss function*) und der Optimierer für das Anpassen der zu lernenden Gewichte definiert wird. Für binäre Klassifikationsprobleme wird die dazu bereitgestellte logarithmische Fehlerfunktion `binary_crossentropy` und als Optimierer der häufig eingesetzte Adam-Optimierer empfohlen [KB14]. Außerdem soll für die spätere Modellevaluation die Genauigkeit als Metrik gemessen werden.
Mit dem Aufruf von `model.fit` in Zeile 14 kann das Training des Klassifikators be-

[5] Quelle: `https://towardsdatascience.com/@rickykim78`, Blogartikel *Another twitter sentiment analysis part 9*, zuletzt besucht am 17.06.19

Tabelle 9.10: Konfusionsmatrix des Entscheidungsbaums

		Vorhersage	
		NO_TS	TS
Klasse	NO_TS	230	20
	TS	21	251

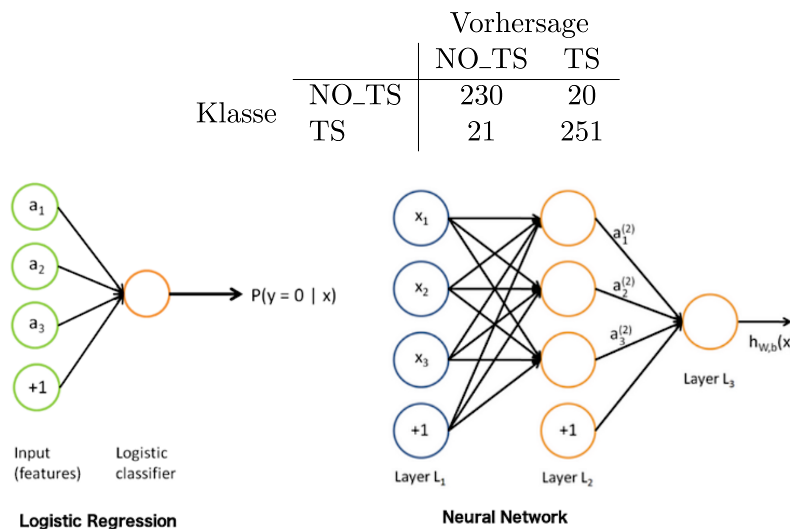


Abbildung 9.1: Vergleich der logistische Regression mit einem neuronalem Netz[5]

gonnen werden. Dazu muss die Anzahl der Trainingsiterationen, den sogenannten *Epochen*, angeben werden. Innerhalb einer Epoche wird eine gewisse Anzahl an Trainingsinstanzen für die interne Modellierung der Gewichte der Knoten eines Netzes betrachtet. Nach jeder dieser Epochen kann anschließend überprüft werden, wie sich die Genauigkeit und der Fehler geändert hat. Ein weiterer wichtiger Parameter ist die *Paketgröße* (engl. *batch size*). Diese gibt an, wie viele Trainingsinstanzen in einer Epoche für das Training betrachtet werden sollen. Desto größer die Paketgröße gewählt wird, desto besser ist die Approximation, desto länger benötigt jedoch auch das Training. Als erste Konfiguration wird die Epochenanzahl auf 200 gesetzt, wobei für die Paketgröße verschiedene Werte untersucht werden. Weiterhin muss der Parameter `validation_data` in Zeile 18 mit dem Validierungsdatensatz befüllt werden. Nach Abschluss des Trainings kann mittels `model.evaluate` die Güte des Modells auf den Validierungsdaten bestimmt werden.

```
1  from keras.models import Sequential
2  from keras.layers import Dense
3
4  # Modellbildung
5  model = Sequential()
6  model.add(Dense(output_dim=1,
7                  input_dim=vocab_size,
8                  activation='sigmoid'))
9  # Modellkonfiguration
10 model.compile(optimizer='adam',
11               loss='binary_crossentropy',
12               metrics=['accuracy'])
13 # Training
14 history = model.fit(X_train, y_train,
15               epochs=200,
16               validation_data=[X_val, y_val],
17               batch_size=32)
18 # Modellevaluation
19 loss, accuracy = model.evaluate(X_val, y_val)
```

Quelltextausschnitt 9.2: Quelltext zur Stufe-1-Klassifikation mittels ANN

Randomisierte Train-Validate-Test-Aufteilung

Die Ausgaben des Modells für jede Trainingsepoche der randomisierten Train-Validate-Test-Trennung werden in der folgenden Abbildung 9.2 dargestellt. Es werden zunächst vorab lemmatisierte Eingabesequenzen in das Modell eingespeist sowie eine Paketgröße von 100 betrachtet. Die Ausgabe ist wie folgt zu lesen: *loss* und *acc* geben den Fehler und die Genauigkeit je Epoche auf den Trainingsdaten an, *val_loss* und *val_acc* werden auf den Validierungsdaten berechnet. Während dem Training werden in jeder Epoche die gesamten Trainingsdaten betrachtet und die Gewichte des neuronalen Netzes angepasst, abhängig von dem Fehler auf den Trainingsdaten. Das Ergebnis einer Epoche ist daher ein Satz an modifizierten Parametern beziehungsweise Gewichten, welcher mit einer bestimmten Güte die Fähigkeit besitzt, auf ungesehenen Validierungsdaten zu generalisieren (siehe Abschnitt 2.4.2.1). Diese Güte der Generalisierungsfähigkeit spiegelt sich in der erzielten Validierungsgenauigkeit wider. Je nach Ausgang einer weiteren Trainingsepoche kann sich diese Genauigkeit verbessern oder verschlechtern. Deshalb muss die Trainingsepoche mit der höchsten Validierungsgenauigkeit gewählt werden, da sie dem Modellzustand mit dem geringsten Validierungsfehler auf den ungesehenen Daten entspricht. Deshalb betrachtet man für die Modelloptimierung bei neuronalen Netzen im Gegensatz zur Klassifikation mittels klassischer Lernverfahren neben der Validierungsgenauigkeit zusätzlich den Fehler auf den Validierungsdaten.
Zusätzlich zu den Trainingsausgaben wird in Abbildung 9.2 ein Kurvenverlauf erstellt. Auf der X-Achse werden die Anzahl der Epochen, auf der Y-Achse die erzielte Genauigkeit in der linken Grafik beziehungsweise der Evaluationsfehler in der rechten Grafik aufgetragen. Hierbei werden die Werte für die Trainingsdaten in blau und für die Validierungsdaten in rot visualisiert. Der Verlauf der Genauigkeit steigt in den ersten Epochen sehr stark an. Dies bedeutet, dass das Netz viele neue Informationen aus den Daten lernt. Ungefähr ab der 20. Epoche nimmt die Steigung der Kurve ab. Dies könnte damit begründet werden, dass das Netz schon ein Grundgerüst von Konzepten (siehe Abschnitt 2.4) zur Modellierung der Daten aufgebaut hat, diese mit neuen Ausprägungen in den Daten dennoch kontinuierlich anpasst. Die Genauigkeit steigt weiterhin. Anschließend konvergiert die Genauigkeit auf den Trainingsdaten auf 100% ab etwa der 50. Epoche. Die Validierungsgenauigkeit erreicht ab diesem Zeitpunkt einen Wert von über 90%, schwankt jedoch noch gering. Betrachtet man die gemessenen Fehler in der rechten Grafik, so verhalten diese sich passend zu den erreichten Genauigkeitswerten. Bei einer geringen Genauigkeit ist der Fehler hoch. Er sinkt in den ersten 20 Epochen stark ab und konvergiert dann ebenfalls. Ziel ist es, das Modell im dem Zustand auszuwählen, bei der die Gewichte das beste Ergebnis auf den Validierungsdaten erreichen. Das betrachtete Modell erreicht in Epoche 98 bis 101 die Genauigkeit auf den Validierungsdaten von 90.1%. Als bestes Ergebnis wird jedoch die Epoche 99 gewählt, da dort der Validierungsfehler am geringsten ist. Diese Epoche lässt sich im Kurvenverlauf an dem Zeitpunkt erkennen, an welchem der Validierungsfehler wieder ansteigt. In diesem Fall soll das Modelltraining abgebrochen werden und der Zustand des trainierten Modells, das heißt die angepassten Gewichte, gespeichert werden.
Bricht man das Training nicht ab, wie es im Kurvenverlauf in Abbildung 9.2 zu sehen ist, so kann es zu dem Effekt der Überanpassung (engl. *overfitting*, siehe Abschnitt 2.4.2.1) führen. Hierbei passt sich das Netz zu sehr auf die konkreten Trainingsdaten an, es lernt sie sozusagen auswendig. Die Folge daraus ist, dass das Netz weniger gut auf den unbekannten Daten generalisieren kann. Dies ist an einem steigenden Validierungsfehler nach der 100. Epoche im rechten Bild zu erkennen, während der Fehler auf den Trainingsdaten für eine längere Trainingszeit gegen Null geht.

Anschließend wird das gleiche Modell mit unterschiedlichen Paketgrößen konfiguriert. Die Ergebnisse werden Tabelle 9.11 dargestellt. Zusätzlich dazu werden auch die vorab nicht-lemmatisierten Eingaben betrachtet.

```
Train on 2182 samples, validate on 546 samples
Epoch 1/200
4s - loss: 0.6175 - acc: 0.6334 - val_loss: 0.5607 - val_acc: 0.6465
Epoch 2/200
                                ...
Epoch 98/200
1s - loss: 0.0101 - acc: 1.0000 - val_loss: 0.2647 - val_acc: 0.9009
Epoch 99/200
1s - loss: 0.0100 - acc: 1.0000 - val_loss: 0.2645 - val_acc: 0.9009
Epoch 100/200
1s - loss: 0.0098 - acc: 1.0000 - val_loss: 0.2646 - val_acc: 0.9009
Epoch 101/200
1s - loss: 0.0096 - acc: 1.0000 - val_loss: 0.2649 - val_acc: 0.9009
```

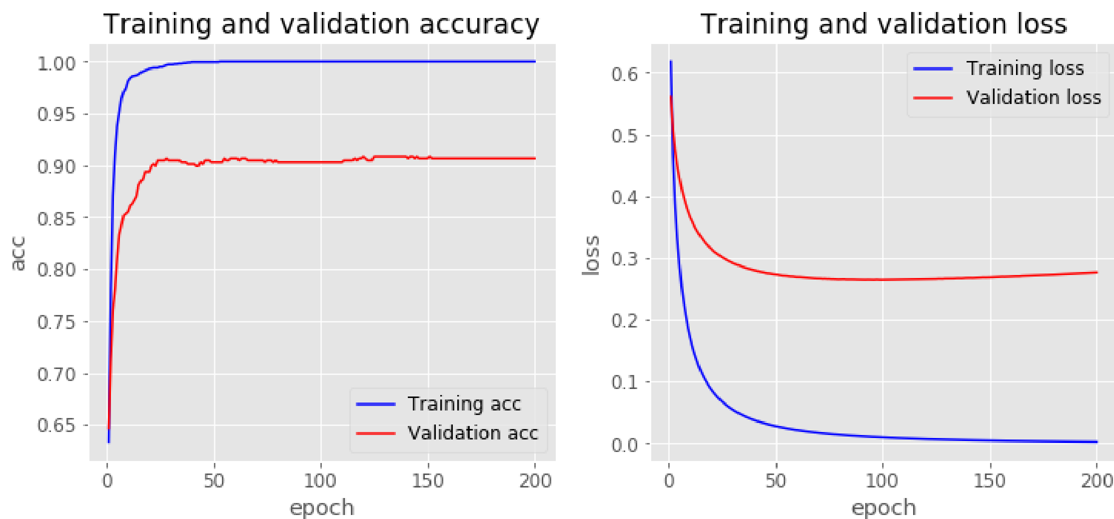


Abbildung 9.2: Ausgabe des einfachen ANN-Netzes für Stufe 1

Tabelle 9.11: Vergleich verschiedener Paketgrößen für Stufe 1

Paketgröße	32	64	100	200	500
Tokenisierte Eingaben	90.0	90.0	90.0	88.8	85.9
Lemmatisierte Eingaben	90.2	90.2	90.3	89.4	86.5

Es zeigt sich, dass die Resultate der Modelle, deren Eingaben im Vorverarbeitungsschritt zusätzlich lemmatisiert werden, geringfügig besser ausfallen als mit ausschließlich tokenisierten Eingabesequenzen. Dieses Verhalten wurde auch bei den trainierten Modellen der klassischen Lernverfahren beobachtet. Für den Parameter der Paketgröße lässt sich kein Einfluss auf das Ergebnis erkennen. Es lässt sich jedoch beobachten, dass bei steigender Paketgröße und gleichbleibender maximaler Epochenanzahl von 200 die Genauigkeit wieder sinkt. Dies liegt daran, dass das Modelltraining noch mehr Epochen benötigen würde, um auf das vermutlich gleiche Ergebnis zu kommen. Insgesamt erreicht das neuronale Netz mit einer Paketgröße von 100 sowie lemmatisierten Eingaben die besten Ergebnisse mit 90.3% Genauigkeit auf den Validierungsdaten. Deshalb werden diese Konfigurationen für die zukünftigen Modellerweiterungen weiterverfolgt. Im Vergleich zum Referenzwert von 57.3% wird mit dieser Konfiguration eine Verbesserung von über 40 Prozentpunkte erreicht. Auch im Vergleich zu den klassischen Lernverfahren wird bereits mit einem sehr einfachen neuronalen Modell eine sehr gute Genauigkeit (neuronal 90.3% versus 92.9% klassisch) erzielt.

Um die Güte des neuronalen Netzes zu verbessern, wird die Netzarchitektur erweitert. In

Quelltextausschnitt 9.3 wird die Implementierung dieses erweiterten Modells vorgestellt. Die *Embedding*-Schicht (engl. *layer*) ab Zeile 5 bildet die erste Schicht des Netzes und übernimmt die Transformation der Wörter einer Eingabesequenz in Folgen von Wortvektoren. Im Gegensatz zum vorherigen Modell wird die Eingabe nicht mit einem `CountVectorizer`, sondern mit einer `Embedding-Layer` transformiert. Die Eingabe pro Sequenz besteht daher nicht mehr aus einem einzelnen Vektor aus Worthäufigkeiten, welcher der Länge des Vokabulars entspricht. Stattdessen ist die Eingabe ein kurzer, auf eine einheitliche Länge aufgefüllter Vektor (engl. *padded vector*) aus Wortindizes des Vokabulars. Für die Bildung dieses Vokabulars werden ausschließlich lemmatisierte Eingabesequenzen betrachtet, da diese sowohl in den klassischen als auch in den neuronalen Modellen zu besseren Ergebnissen führen. Überdies wird bei den neuronalen Modellen auf das Hinzufügen von N-Grammen zum Vokabular verzichtet, da diese Umgebungsinformationen implizit in den Wortvektoren enthalten sind (siehe Abschnitt 2.4.3.2). Die Embeddingschicht benötigt als Parameter neben der Eingabe-Vokabulargröße die maximale Eingabesequenzlänge sowie die Ausgabe-Vektorlänge. Während die ersten beiden Parameter durch die Vokabularbildung und das Padding festgelegt sind, kann die Wortvektorlänge `embedding_dim` frei definiert werden. Da als zweite Implementierungsvariante die vortrainierten Fasttext-Wortvektoren eingesetzt werden und deren Länge auf 300 Dimensionen festgelegt ist, wird die Vektordimension für die manuell erstellten Wortvektoren ebenfalls auf 300 festgelegt. Für den Einsatz dieser vortrainierten Wortvektoren muss in der Embeddingschicht mit dem zusätzlichen Parameter `weights` die Fasttext-Embedding-Matrix mitgegeben werden.
Zwischen Embeddingschicht und Ausgabeschicht (Zeile 10 bis 15) liegt das Potential für Modellerweiterungen. Hier werden nacheinander folgende unterschiedliche Netzschichten und deren Kombinationen eingefügt, die im Grundlagenkapitel unter Abschnitt 2.4.2 näher erklärt werden. Der Parameter u steht dabei für die Anzahl an Neuronen (engl. *units*), k für die Größe des Filterfensters beim faltenden neuronalen Netz (engl. *convolutional*, CNN) und der Parameter p in der Dropout-Schicht für den Prozentsatz an Neuronen, die während der Berechnung eines Schrittes ignoriert werden sollen.

In Quelltextausschnitt 9.3 ist eine Kombination aus bidirektionalem LSTM, sowie einer Dropout- und mehreren Dense-Schichten implementiert (siehe Tabelle 9.12). Diese Modelle lesen die Eingabesequenzen bidirektional, das heißt von beiden Seiten, ein. Deshalb können diese komplexere Beziehungen aus den Eingabesequenzen modellieren. Es handelt sich dabei nicht mehr um ein künstliches neuronales Netz (ANN), sondern um ein rekurrentes Netz (RNN). Das beschriebene Modell besteht aus `2 + n` Schichten, wobei `n` die Anzahl der zusätzlich hinzugefügten Schichten angibt. Es werden hierbei nur die Schichten gezählt, welche Eingaben von vorherigen Schichten aufnehmen um damit Knotengewichte zu lernen. Die Eingabeschicht (*Input-Layer*) wird daher nicht mitgezählt. Sie ist im Quelltext implizit durch `model = Sequential()` repräsentiert.
Zuletzt folgt analog zum ersten neuronalen Modell eine vollvermaschte, dichte Schicht, das *Dense-Layer*. Diese repräsentiert die eindimensionale Ausgabeschicht (*Output-Layer*), durch welche ein binäres Klassifikationsergebnis mittels der Sigmoid-Funktion erzeugt werden kann.
Die Konfiguration des Modells mit `model.compile` wird aus dem vorherigen einfachen Modellansatz übernommen. Das Modell wird daher mit 200 Epochen als maximale Trainingsdauer und 100 als Paketgröße initialisiert. Anschließend wird das Training des Klassifikators auf dem Trainingsdatensatz begonnen. Dank der Datensatzteilung in zusätzliche Validierungsdaten kann der Lernvorgang nach jeder Epoche evaluiert werden. Falls sich in den Modellergebnissen keine positive Entwicklung bezüglich des Leistungsmaßes ableiten lässt, so kann der Trainingsprozess vorzeitig abgebrochen werden. Als Leistungsmaß wird in der Praxis häufig der Validierungsfehler betrachtet [Pre97]. Sinkt dieser bei 6 aufeinan-

Tabelle 9.12: Unterschiedliche Netzschichten und ihre Parameter

Name	Abkürzung	Beschreibung
Flatten	`Flat`	Netzschicht, die die mehrdimensionalen Tensor-Eingaben der Embeddingschicht zu einem eindimensionalen, flachen Vektor transformiert.
GlobalMaxPooling1D	`GMax`	Netzschicht zur Dimensionsreduktion und Prävention vor Überanpassung (Regularisierung). Behält nur die Aktivität des aktivsten („maximalen“) Neurons für die weiteren Berechnungsschritte.
LSTM	`LSTM(`u`)`	Netzschicht, die langfristigere Abhängigkeiten in Daten erkennen können (siehe Abschnitt 2.4.2.2).
Bidirektionales LSTM	`BiLSTM(`u`)`	Analog zum LSTM, mit dem Unterschied, dass die Eingabe von beiden Seiten eingelesen und verarbeitet werden.
GRU	`GRU(`u`)`	Vereinfachtes Modell eines LSTMs. Weniger ressourcenintensiv (siehe Abschnitt 2.4.2.3).
Bidirektionales GRU	`BiGRU(`u`)`	Analog zu GRU, mit dem Unterschied, dass die Eingaben von beiden Seiten eingelesen und verarbeitet werden.
Convolutional	`Conv(`u, k`)`	Netzschicht, bei der die Aktivität der Neurons wird über eine diskrete Faltung (engl. *convolution*) berechnet wird (siehe Abschnitt 2.4.2.4). Wird häufig zur Merkmalsextraktion eingesetzt.
Dense	`D(`n`)`	Vollvermaschte Netzschicht, die häufig zur Klassifikation eingesetzt wird. Die Anzahl der Neuronen n korrespondiert üblicherweise mit der Anzahl an Klassen. Kann verschiedene Aktivierungsfunktionen für die Abbildung der Berechnungen auf die Ausgabeklassen nutzen.
Dropout	`DO(`p`)`	Netzschicht, die einen festgelegten Prozentsatz p an Neuronen im Netzwerk für den kommenden Berechnungsschritt zufällig ausschaltet (engl. *dropout*). Wird zur Überanpassungs-Prävention (Regularisierung) eingesetzt.

derfolgenden Epochen nicht, so wird das Training abgebrochen um Ressourcen zu sparen. Zusätzlich kann durch den frühzeitigen Abbruch des Trainings sichergestellt werden, den Klassifikator nicht zu sehr an die Trainingsdaten anzupassen (engl. *overfitting*), um auf unbekannten Daten besser generalisieren zu können (siehe Abschnitt 2.4). Die Gewichte zum Zeitpunkt des geringsten Validierungsfehlers werden gespeichert. Zuletzt wird das Modell mit diesen optimalen Gewichten auf dem Validierungsdatensatz evaluiert. Die letzten beiden Logiken werden durch die Keras-Funktionen `EarlyStopping` und `ModelCheckpoint` implementiert. Der Quelltext der beschriebenen Implementierung wird in Quelltextausschnitt 9.3 präsentiert.

```
1   embedding_dim = 300
2   maximum_sequence_len = 135
3
4   model = Sequential()
5   model.add(Embedding(input_dim=vocab_size,
6                       output_dim=embedding_dim,
7                       input_length=maximum_sequence_len,
8                       weights=[fasttext_embedding_matrix],
```

Tabelle 9.13: Genauigkeiten auf den randomisierten Validierungsdaten verschiedener neuronaler Netze für Stufe 1

Name	Erweiterung	Ep	SWv	VWv
ANN1	-	99	90.3	90.3
ANN2	`Flat, D(100)`	12	91.6	84.6
ANN3	`GMax, D(100)`	11	89.9	87.9
CNN1	`Conv(128, 5), Max(2),Conv(128, 5), GMax, D(10)`	7	95.2	95.4
RNN1	`GRU(128), D(100)`	12	56.2	56.2
RNN2	`BiGRU(32), DO(0.2), D(64), DO(0.2)`	12	94.7	95.2
RNN3	`LSTM(128), D(100)`	12	56.2	56.2
RNN4	`BiLSTM(128), D(64)`	5	95.1	95.6
RNN5	`BiLSTM(128), D(100), DO(0.3), D(50)`	10	93.6	94.5

```
9                            output_dim=embedding_dim))
10  # Beginn Netzerweiterungen
11  model.add(Bidirectional(LSTM(128)))
12  model.add(Dense(100, activation='relu'))
13  model.add(Dropout(0.3))
14  model.add(Dense(50, activation='relu'))
15  # Ende Netzerweiterungen
16  model.add(Dense(1, activation='sigmoid'))
17
18  model.compile(optimizer='adam',
19                           loss='binary_crossentropy',
20                           metrics=['accuracy'])
21
22  history = model.fit(X_train, y_train,
23                           epochs=200,
24                           validation_data=[X_val, y_val],
25                           batch_size=100,
26                           callbacks=[early_stopping, model_checkpoint])
27  loss, accuracy = model.evaluate(X_val, y_val)
```

Quelltextausschnitt 9.3: Quelltext zur Stufe-1-Klassifikation mittels erweitertem ANN

Die Ergebnisse der besten erweiterten Netze auf der randomisierten Datensatzaufteilung sind in Tabelle 9.13 aufgeführt. In den ersten beiden Spalten der Tabelle sind die Modelle mit Namen und Erweiterungsschichten aufgetragen. Das erste, einfache künstliche neuronale Netz befindet sich beispielsweise in Zeile 1 mit dem Namen `ANN1`. In den Spalten 3 und 4 sind die Epoche (Ep) zum Zeitpunkt des geringsten Validierungsfehlers (`val_loss`) sowie die Genauigkeit auf den Validierungsdaten mit selbst trainierten Wortvektoren (SWv) aufgetragen. Die Modellresultate mit den vortrainierten Wortvektoren (VWv) von Fasttext sind in Spalte 5 aufgeführt.

Die besten Ergebnisse der randomisierten Train-Validate-Test-Aufteilung werden mit dem Modell `RNN4` und den vortrainierten Wortvektoren erreicht. Die Genauigkeit des bidirektionales LSTM beträgt 95.6% auf den Validierungsdaten. Ohne die Fasttext-Vektoren wird eine etwas schlechtere Genauigkeit von 95.1% erreicht. Im Vergleich zum einfachen neuronalen Netz entspricht dies einer Verbesserung von weiteren 5.5 Prozentpunkten.
Der zugehörige Kurvenverlauf des Modells `RNN4` wird in Abbildung 9.3 veranschaulicht. Auf der X-Achse wird wiederum die Anzahl der Epochen sowie auf der Y-Achse die erzielte Genauigkeit in der linken Grafik beziehungsweise der Evaluationsfehler in der rechten Grafik aufgetragen. Hierbei werden die Werte für die Trainingsdaten in blau und für die Validierungsdaten in rot visualisiert. Der Kurvenverlauf des Modelltrainings von `RNN4` fällt

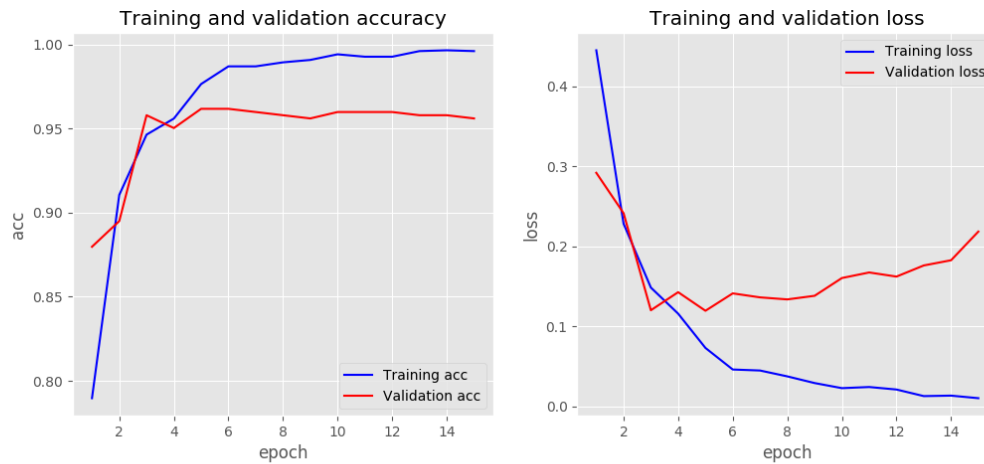


Abbildung 9.3: Kurvenverlauf der besten neuronalen Modellarchitektur `RNN4` für Stufe 1

im Gegensatz zum Kurvenverlauf des einfachen Modells `ANN1` aus Abbildung 9.2, weniger glatt, sondern eher schwankend aus. Dies liegt einerseits daran, dass der Ausschnitt nur die ersten 15 Epochen zeigt, statt einen Kurvenverlauf über 200 Epochen in der vorherigen Abbildung. Andererseits ist dies auf die unterschiedliche Modellarchitektur von `RNN4` zurückzuführen. Das Modell erreicht den geringsten Validierungsfehler bereits in der 5. Epoche. Die Konvergenz der Genauigkeit tritt hierbei deutlich schneller ein als mit Modell `ANN1` (99. Epoche) und erreicht dabei einen höheren Genauigkeitswert mit mehr als 5 Prozentpunkten Unterschied. Daraus lässt sich schließen, dass die modellierte Netzarchitektur auf Basis eines bidirektionalen LSTMs die Trainingsdaten mit einer geringeren Anzahl an Epochen auf interne Konzepte (siehe Abschnitt 2.4) abbilden zu können scheint. Die komplexere Modellarchitektur scheint die Trainingsdaten sozusagen besser zu verstehen und auf unbekannten Daten besser generalisieren zu können. Weiterhin zeigt der Kurvenverlauf in Abbildung 9.3 ab Epoche fünf, dem Zeitpunkt des geringsten Validierungsfehlers, einen Anstieg dieses Fehlers sowie eine gleichzeitige Verringerung des Fehlers auf den Trainingsdaten. Dies deutet wiederum den Effekt der Überanpassung (siehe Abschnitt 2.4.2.1) auf die Trainingsdaten hin. Würde das Training an dieser Stelle mit der aktuellen Modellkonfiguration nicht abgebrochen werden, so würde das Modell auf den Trainingsdaten immer bessere Werte erzielen (da es sie auswendig lernt), jedoch auf unbekannten Daten immer schlechter generalisieren können. Dieses Verhalten spiegelt sich in den erzielten Genauigkeiten auf der linken Seite wider.
Die Modellarchitektur auf Basis des zweifach-faltenden neuronalen Netzes `CNN1` (siehe Tabelle 9.12 und Abschnitt 2.4.2.4) schneidet ebenfalls sehr gut mit 95.2% auf selbst trainierten und 95.4% Genauigkeit auf Fasttext-Vektoren ab. Weiterhin zeigen die Ergebnisse, dass die bidirektionale Architekturen deutlich besser abschneiden als die unidirektionalen Varianten. An den Modellresultaten lässt sich auch ablesen, dass der Einsatz von vortrainierten Wortvektoren im Falle der binären Klassifikation nur in manchen Fällen eine Verbesserung der Genauigkeit erbringt. Außerdem fällt auf, dass das Training schon bei einer vergleichsweise geringen Epochenzahl, das heißt sehr schnell, das Minima des Validierungsfehlers erreicht. Das Training wird an dieser Stelle jeweils vorzeitig abgebrochen, um die beste Modellkonfigurationen zu speichern.

Als letzten Klassifikator wird das Werkzeug *BERT* [DCLT18a] von Google auf dem binären Klassifikationsproblem angewendet (siehe Abschnitt 2.4.2.5). Dieser Klassifikator nutzt das vortrainierte Modell und muss in einem Anpassungsschritt (engl. *fine-tuning*) auf das spezifische Klassifikationsproblem mittels annotierten Daten angepasst werden. Dabei wird die

Tabelle 9.14: Genauigkeiten auf den randomisierten Validierungsdaten mit BERT für Stufe1

	Epochen	Genauigkeit
BERT1	5	99.5
BERT2	10	99.6
BERT3	300	99.6

Paketgröße auf 32 begrenzt, um Speicherüberlauf-Fehler (engl. *out of memory error*) zu verhindern. In Tabelle 9.14 werden die Ergebnisse von BERT auf den Validierungsdaten nach dem Training mit unterschiedlich langen Epochen veranschaulicht. Es wird im besten Fall eine Validierungs-Genauigkeit von 99.6% mit dem Modell `BERT3` erzielt. Diese steigert die Genauigkeit der besten, eigens aufgebauten Modells `RNN4` um weitere 4.2 Prozentpunkte. Das auf einer sehr großen Menge von Textdaten allgemein trainierte Sprachmodell BERT scheint hier positiven Einfluss zu nehmen.

Aus den modellierten Netzarchitekturen muss für die Evaluation auf den Testdaten das beste Modell ausgewählt werden. Nach dem *Ockham'schen Rasiermesser-Prinzip*[BEHW87] soll bei Modellansätzen mit vergleichbar guten Ergebnissen immer das einfachere Modell gewählt werden. Deshalb wird aus den selbst entworfenen Modellen die Architektur `RNN4` mit Wortvektoren für die Modellevaluation der randomisierten Train-Validate-Test-Datensätze ausgewählt. Zusätzlich dazu wird das BERT-Modell `BERT3` in der späteren Evaluation betrachtet.

Szenarienbezogene Train-Validate-Test-Aufteilung

Die obigen Modellentwürfe werden anschließend auf die szenarienbezogene Trennung der Daten angewandt. Die Ergebnisse unter Verwendung der gleichen Modelle wie in der randomisierten Datensatzaufteilung werden zusammenfassend in Tabelle 9.15 aufgeführt. Die Modellresultate werden sowohl für selbst trainierte Wortvektoren (SWv) als auch für die vortrainierten Wortvektorrepräsentationen (VWv) mittels Fasttext präsentiert.
Das beste Ergebnis wird mit der modellierten Architektur `CNN1` erzielt. Mit selbst trainierten Wortvektoren wird eine Genauigkeit von 97.3% auf den Validierungsdaten erreicht. Das Modell `CNN1` erreicht mit den Fasttext-Wortvektoren sogar eine Genauigkeit von 97.7%. Da auch das bidirektionale LSTM `RNN4` ähnlich gute Ergebnisse erzielt, werden diese beiden Modelle für die Modellevaluation gewählt. Das vortrainierte BERT-Modell `BERT3` schlägt mit 99.2% Genauigkeit wiederum alle selbst modellierten Netzarchitekturen. Daher wird auch dieses für die folgende Evaluation auf den Testdatensätzen ausgesucht.

9.1.1.7 Modellevaluation

Nach der Modellierung von verschiedenen Verfahren des maschinellen Lernens auf Basis von klassischen und neuronalen Ansätzen folgt deren Evaluation auf den bisher im Training ungesehenen Testdatensätzen. Dazu wird eine Vergleichstabelle (siehe Tabelle 9.16) erstellt, welche alle im vorherigen Abschnitt vorgestellten Modellarchitekturen und deren erzielten Ergebnisse auf dem Testdatensatz für beide Train-Test-Validate-Aufteilungen enthält. Die unterschiedlichen Schichten, welche in diesen Modellarchitekturen eingesetzt wurden, wurden in Tabelle 9.12 bereits erläutert.
Bei der randomisierten Aufteilung der Trainingsinstanzen erzielt das Modell `BERT3` die höchste Genauigkeit auf dem Testdatensatz mit 98.8%. Unter den selbst trainierten Modellen erreicht die Architektur `CNN1` mit Fasttext-Wortvektoren die höchste Genauigkeit mit 96.6%. Das `BERT3`-Modell steigert die Genauigkeit des besten selbst trainierten Modells `CNN1` um weitere 2.9 Prozentpunkte. Das bidirektionale Modell `RNN4` erreicht ein etwas schlechteres Ergebnis als das faltende Netz mit 95.9% Testgenauigkeit. Weiterhin fällt auf,

Tabelle 9.15: Genauigkeiten auf den szenarienbezogenen Validierungsdaten neuronaler Netze für Stufe 1

Name	Erweiterung	Ep	SWv	VWv
ANN1	-	164	90.7	90.7
ANN2	Flat, D(100)	8	90.5	87.4
ANN3	GMax, D(100)	9	89.3	91.8
CNN1	Conv(128, 5), Max(2),Conv(128, 5), GMax, D(10)	8	97.3	97.7
RNN1	GRU(128), D(100)	5	51.9	51.9
RNN2	BiGRU(32), DO(0.2), D(64), DO(0.2)	10	95.4	95.8
RNN3	LSTM(128), D(100)	8	51.9	51.9
RNN4	BiLSTM(128), D(64)	9	96.0	96.2
RNN5	BiLSTM(128), D(100), DO(0.3), D(50)	8	93.7	95.4
BERT1	-	5	-	99.1
BERT2	-	10	-	99.2
BERT3	-	300	-	99.2

Tabelle 9.16: Test-Genauigkeiten der Stufe-1-Klassifikation

	Randomisiert		Szenarienbezogen	
	Validate	Test	Validate	Test
Referenzwert	57.3	-	54.7	-
Random Forest	92.1	90.2	90.1	37.4
Logistische Regression	92.7	94.7	92.9	71.9
CNN1 + VWv	95.5	96.6	97.7	86.2
RNN4 + VWv	95.6	95.9	96.2	91.9
BERT3	99.6	98.8	99.2	97.7

dass die erzielten Genauigkeiten auf den Validierungsdaten im Vergleich zur abschließenden Evaluation auf den ungesehenen Testdaten ähnlich oder gar besser ausfallen.

Für die szenarienbezogene Datentrennung wird eine beste Testdatensatz-Genauigkeit von 97.7% mit dem `BERT3`-Modell erreicht. Dies entspricht einer Menge von nur 21 Fehlklassifikationen von insgesamt 794 Testinstanzen. Das beste selbst trainierte Modell `RNN4` bei Einsatz der Fasttext-Vektoren fällt mit 91.9% Genauigkeit etwas schlechter aus. Im Gegensatz zur randomisierten Datenverteilung schneidet das faltende Netz mit 86.2% deutlich schlechter als das bidirektionale Modell ab. Es scheint auf das unbekannte Szenario dieser Datentrennung weniger gut generalisieren zu können und ist daher weniger robust. Bei Vergleich der erreichten Genauigkeit auf den Validierungs- beziehungsweise den Testdaten fällt auf, dass letztere deutlich schlechter ausfallen. Die Tendenz einer schlechteren Testgenauigkeit fällt bei den neuronalen Netzen im Gegensatz zu klassischen Lernansätzen geringfügig schlechter aus. Sie beträgt insgesamt etwa 8 Prozentpunkte Unterschied zur Genauigkeit auf den Validierungsdaten. Das lässt sich darauf zurückführen, dass bei der szenarienbezogenen Testdaten-Trennung ein für die Trainingsdaten unbekanntes Szenario behandelt wird. Dementsprechend ist die erreichte Genauigkeit der konzipierten Modelle sehr zufriedenstellend.

Abschließend lässt sich aus den Modellergebnissen folgern, dass maschinelles Lernen eine gute Lösungsstrategie für das binäre Entscheidungsproblem darstellt. Die Genauigkeit der Modelle fällt dabei sehr unterschiedlich aus. Generell lässt sich eine Tendenz erkennen, dass neuronale Netze den klassischen Verfahren überlegen sind. Auch der Einsatz von vortrainierten Wortvektor-Modellen wirkt sich positiv auf die Klassifikation aus. Dies trifft für beide Train-Validate-Test-Aufteilungen zu. Insbesondere die bidirektionalen neuronalen

Netze erzielen gute Ergebnisse. Die Detektion von Lehrsequenzen innerhalb der ersten Klassifikationsstufe ist somit zufriedenstellend möglich.

Mögliche Gründe für die Fehlklassifikationen beziehungsweise die geminderte Generalisierbarkeit auf den Testdaten lassen sich einerseits auf die Komplexität der natürlichen Sprache (H 1.1) sowie die Vielfalt der Formulierungen (H 1.3) zurückführen. Diese wurden in Abschnitt 7.1 als Herausforderungen für diese Teilaufgabe ermittelt. Diese Herausforderungen erschweren es dem Klassifikator möglicherweise, gemeinsame Strukturen in den sich unterscheidenden Trainingsdaten zu finden. Die Herausforderung (H 1.4) spricht mit der Mehrdeutigkeit der Formulierungen das Vorhandensein von nicht eindeutigen Grenzfällen des binären Entscheidungsproblems in den Trainingsdaten an. Diese könnten den Klassifikator zusätzlich verwirren.

9.1.2 Detektion von Deklaration und Beschreibung

Für die Detektion von Deklaration und Beschreibung wird im Entwurfskapitel in Abschnitt 8.1.2 ein Mehrklassen-Klassifikationsansatz gewählt. Ziel der zweiten Stufe der hierarchischen Klassifikation ist es, die Kommandodeklaration (Definition 5.1.2, Klasse `DECL`) sowie Kommandobeschreibung (Definition 5.1.3, Klasse `SPEC`) detektiert werden. Auf Sprachebene untergliedern diese Lehrsequenz-Bestandteile den Namen der beschriebenen Funktion von ihrem Inhalt. Die folgende Implementierung wird dabei analog zum Vorgehen im vorherigen Abschnitt 9.1.1 der Detektion von Lehrsequenzen beschrieben. Die vorgestellte Programmierumgebung aus Abschnitt 9.1.1.1 wird übernommen. Anschließend folgt die Generierung und Vorverarbeitung der Trainingsinstanzen. Nach der Berechnung des Referenzwertes werden im Absatz der Modellbildung Verfahren auf Basis neuronaler Netze auf den Trainingsdaten trainiert. Diese entworfenen Modelle werden abschließend in der Modellevaluation auf einem unbekannten Testdatensatz ausgewertet.

9.1.2.1 Generierung von Trainingsinstanzen

Die Generierung der Trainingsinstanzen für die zweite Stufe der Klassifikation wird auf Basis der Definition 8.1.2 durchgeführt. Als Eingabe für das Training der maschinellen Lernverfahren dienen die im Goldstandard von Stufe 1 als Lehrsequenz (Klasse TS) klassifizierten Trainingsinstanzen. Daher halbiert sich der Datensatz etwa für die zweite Klassifikationsstufe (siehe Tabelle 8.4). Die Ausgabeklassen für Stufe 2 werden wiederum mit dem Annotationswerkzeug *Gate* getätigt. Der annotierte Datensatz wird anschließend als CSV-Datei in `study_multi_labeled.csv` gespeichert, wobei die Spaltenbezeichnungen in Tabelle 9.18 erklärt werden. Die Spalte `mclass_case` dient der reinen Dokumentation unklarer Fälle bei Interpretationen der Eingabesequenzen. Dies wurde in der Auswertung der Vorstudie in Abschnitt 6.5.2 erläutert sowie deren Umgang als Herausforderung (H 1.4) im Entwurfskapitel in Abschnitt 8.1.2.2 definiert. Für die nachfolgend implementierten Modelle werden ausschließlich die Spalten `preprocessed` und `mclass_output` betrachtet. Die Tabelle 9.17 präsentiert zwei exemplarische Trainingsinstanzen aus dem Datensatz nach diesem Vorverarbeitungsschritt. Die vergebenen Ausgabeklassen und deren Bedeutung ist in Abschnitt 8.1.2.2 in Tabelle 7.1 nachzulesen.

9.1.2.2 Vorverarbeitung der Trainingsinstanzen

Da der Klassifikationsalgorithmus eine numerische Eingabe benötigt, müssen die Eingabemerkmale und Ausgabeklassen der Trainingsinstanzen vorverarbeitet werden. Diese Vorverarbeitungsschritte werden im Folgenden kurz beschrieben.

Als erstes werden die Eingabemerkmale verarbeitet. Dazu wird die in Abschnitt 8.1.2.3 vorgestellte numerische Textrepräsentation der Wortvektoren (siehe Abschnitt 2.4.3.2) verwendet. Der Bag-Of-Words-Ansatz (siehe Abschnitt 2.4.3.1) aus dem binären Entscheidungsproblem wird für die Merklassen-Klassifikation aufgrund seines geringeren Informa-

Tabelle 9.17: Auszug des annotierten Klassifikationsdatensatzes der Stufe 2

	preprocessed	scenario	binary_output	mclass_output
999	you have to place the cup under the dispenser and press the red button to make coffee	2	1	[SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, SPEC, DECL, DECL, DECL]
2509	collect cutlery from cupboard bring them to the table and place down neatly	4	0	-

Tabelle 9.18: Struktur des Klassifikationsdatensatzes der Stufe 2

Spaltenname	Beschreibung
preprocessed	Vorverarbeitete Eingabesequenz der Stufe 1
scenario	Szenario der Vorstudie, nummeriert von 0 bis 4[6]
binary_output	Ausgabeklasse der Stufe 1
mclass_output	Ausgabeklasse der Stufe 2
mclass_case	Markierung bei unklarer Interpretation der Eingabesequenz

tionsgehaltes übersprungen. Da die Trainingsinstanzen der ersten Stufe mit Klasse TS als Trainingsinstanzen der Stufe 2 übernommen werden, müssen diese nicht nochmals vorverarbeitet werden. Sie sind daher nach dem Vorgehen in Abschnitt 9.1.1.3 zunächst tokenisiert und lemmatisiert sowie auf ein Vokabular abgebildet (siehe Beispiel 9.1.1) worden. Anschließend wurden die Eingabesequenzen auf eine Sequenzlänge von 135 Wörtern aufgefüllt beziehungsweise begrenzt und Wortvektoren erstellt (siehe Beispiel 9.1.3).

Als zweites müssen die Ausgabeklassen auf numerische Werte abgebildet werden. Bei der Mehrklassen-Klassifikation wird pro Wort der Eingabesequenz eine Klasse erwartet. Eine Ausgabeklasse besteht daher aus einer Liste von numerischen Werten je Klassenausprägung. Diese Ausgaben müssen analog zu den Eingabemerkmalen auf eine einheitliche Länge von 135 aufgefüllt werden. Da mit der Zahl 0 aufgefüllt wird, darf die überdies Zahl 0 nicht als Klassenrepräsentation verwendet werden. Da im Entwurf in Definition 8.1.2 3 Klassen für die zweite Klassifikationsstufe definiert sind, wird die Ausprägung dieser Klassen von 1 bis 3 repräsentiert.

Nach der Vorverarbeitung der Trainingsinstanzen kann das Training des neuronalen Mehrklassen-Klassifikators begonnen werden. Dafür werden diese Instanzen wiederum in die randomisierte und szenarienbezogene Train-, Validate- und Testdatentrennung (siehe Abschnitt 8.1.1.4) aufgeteilt und anschließend getrennt voneinander evaluiert. Die Größe der Trainingsinstanzen nach der Aufteilung der beiden Train-Validate-Test-Varianten wird in Tabelle 9.19 dargestellt. Für die szenarienbezogene Aufteilung wird wiederum das Szenario *Greet someone* exemplarisch als Testdatensatz ausgewählt.

9.1.2.3 Referenzwert

Bevor die Lösung des Mehrklassen-Klassifikationsproblems mit maschinellen Lernverfahren angegangen wird, wird die Ausgangssituation auf dem Datensatz bestimmt. Dazu wird analog zu Abschnitt 9.1.2.3 der Referenzwert nach dem Zero-Rule-Algorithmus bestimmt. Es ergeben sich die folgende Genauigkeitswerte in Tabelle 9.20. Darin lässt sich erkennen, dass die drei Klassen des Klassifikationsproblems von Stufe 2 sehr unbalanciert im Datensatz verteilt sind. Die Klasse der Kommandobeschreibung (SPEC) ist in beiden Train-Test-

Tabelle 9.19: Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 2

	randomisiert	szenarienbezogen
Stufe 1 (Klasse TS + NO_TS)	3410	3410
Stufe 2 (Klasse TS)	1988	1988
Train	1272	1145
Validate	318	286
Test	398	557

Tabelle 9.20: Train-Validate-Test-Aufteilung der Trainingsinstanzen von Stufe 2

	randomisiert	szenarienbezogen
Train + Validate	59274	54614
Klasse DECL	12500	11635
Klasse SPEC	44997	41346
Klasse ELSE	1777	1633
Genauigkeit	75.9%	75.7%

Aufteilungen mit 75.9% beziehungsweise 75.7% mehr als dreifach so häufig im Datensatz enthalten als die Klasse der Kommandodeklaration (DECL). Gegenüber der Klasse der sonstigen Äußerungen (ELSE) ist sie sogar über 25-fach häufiger vertreten.

Diese ungleiche Verteilung der Klassen führt beim Training von Klassifikatoren zu Problemen. Es liegen unterschiedlich viele Ausgabeklassen für die drei Klassen vor, deren Detektion ist jedoch gleich wichtig. Besonders die Minderheitsklasse (engl. *minority class*) DECL ist für die darauffolgende dritte Stufe der hierarchischen Klassifikation von gleicher Bedeutung wie die häufigste Klasse SPEC. Deshalb müssen Techniken zur Klassenbalancierung vorgenommen werden.
Es gibt verschiedene Ansätze, die Klassenverteilung zu beeinflussen. Die erste Methode einer weiteren Datenerhebung zu den unterrepräsentierten Klassen innerhalb einer weiteren Vorstudie wird ausgeschlossen. Eine andere Möglichkeit ist die Datensatzverminderung (engl. *undersampling*), bei der die überrepräsentierte Klasse (hier SPEC) auf die gleiche Größe der kleinsten bzw. der wichtigsten unterrepräsentierten Klasse (hier DECL) geschrumpft wird. Dies ist jedoch keine Option, da für neuronale Netze die Größe des Datensatzes ausschlaggebend für die erzielte Genauigkeit im Training des Modells ist und daher nicht noch weiter verkleinert werden soll. Auch die Datensatzerhöhung (engl. *oversampling*), bei welcher die Instanzen der Minderheitsklasse auf die Größe der häufigsten Klasse kopiert wird, wird ausgeschlossen. So würden zu viele Duplikate im Datensatz entstehen, das ein „Auswendiglernen“ der Sätze zur Folge haben könnte. Das Hinzufügen von synthetischen Daten wird wegen des für die Logik zusätzlich verbundenen Aufwandes verworfen. Eine weitere Methode ist die Änderung der Gütemetrik. Die Genauigkeit ist aufgrund ihrer Nicht-Beachtung der falsch positiven Klassifikationen nicht aussagekräftig. So spiegelt die Referenzwert-Genauigkeit von 75% die Klassenverteilung wieder und ergibt sich aus der guten Klassifikation der häufigsten Klasse. Eine ähnlich gute Detektion der anderen Klassen ist daraus nicht folgerbar. Dies ist die Aussage des Genauigkeits-Paradoxon (engl. *accuracy paradox*) [VAPM14]. Deshalb kann neben der Genauigkeit auch die Konfusionsmatrix und die Pro-Klassen-Genauigkeit betrachtet werden. Zuletzt gibt es die Möglichkeit der Klassengewichtung (engl. *class weight balacing*). Anstatt Zeit und Ressourcen für eine zusätzliche Datenerhebung oder Datenbearbeitung zu verbrauchen, kann innerhalb des Modelltrainings der Klassifikationsfehler so angepasst werden, dass die Klassen gleichermaßen in die Fehlergewichtung mit einfließen. Deshalb wird neben der Beachtung anderer Metriken dieser Klassengewichtungs-Ansatz gewählt.

9.1.2.4 Modellbildung: Verfahren auf Basis neuronaler Netze

Für das Entscheidungsproblem der Mehrklassen-Klassifikation werden neuronale Netze betrachtet. Im Vergleich zu dem binären Klassifikationsproblem der ersten Stufe in Abschnitt 9.1.1.6 handelt es sich in dieser Stufe um ein Sequenzmarkierungsproblem (siehe Abschnitt 8.1.2.4). Das bedeutet, dass zum Einen für eine einzelne Eingabe mehrere, das heißt eine Sequenz von Klassifikationen getätigt werden müssen und zum Anderen, dass insbesondere diese einzelnen Klassifikationen zu einer Eingabe voneinander abhängig zu entscheiden sind. Trotz des Wissens, dass die bidirektionalen Netzarchitekturen als Musterbeispiel für diese Probleme eingesetzt werden, wird zunächst ein einfaches künstliches neuronales Netz (ANN, siehe Abschnitt 2.4.2.1) als erstes Modell implementiert. In Quelltextausschnitt 9.4 wird der Quelltext für dieses Modell exemplarisch für alle weiteren Netzimplementierungen vorgestellt. Die verwendeten Schichten (engl. *layer*), welche bereits aus der binären neuronalen Klassifikation in Abschnitt 9.1.2.4 vorgestellt wurden, werden an dieser Stelle nicht erneut erläutert. Stattdessen wird detailliert auf die neuen verwendeten Schichten eingegangen, welche für die Lösung des Sequenzmarkierungsproblems eingesetzt werden.

Das einfache Modell besteht aus 5 Schichten. Als erste versteckte Schicht wird wiederum das *Embedding-Layer* verwendet. Die Eingabesequenzen wurden bei der Überführung in Wortvektor-Repräsentationen in Abschnitt 9.1.2.2 auf eine maximale Sequenzlänge von 135 Wörtern aufgefüllt, das heißt mit einem Padding versehen (siehe Beispiel 9.1.3). Da in dieser Klassifikationsstufe wortweise klassifiziert wird, muss das Padding einer aufgefüllten Eingabesequenz bei der Klassifikation der Ausgabeklassen übersprungen werden. Dies kann durch die Parameterinitialisierung `mask_zero = True` in der Embeddingschicht bewerkstelligt werden. Weiterhin liegt zwischen der Embeddingschicht und Ausgabeschicht (Zeile 11 bis 13) das Potential für Modellerweiterungen. Dort werden im späteren unterschiedliche Netzschichten und deren Kombinationen eingefügt.
Zuletzt folgt in Zeile 14 eine neue Schicht, das *TimeDistributed-Layer*. Im Gegensatz zur binären Klassifikation aus Abschnitt 9.1.1.6, bei welcher für eine Eingabe eine einzige Klassifikation vorhergesagt werden musste, ist für das Sequenzmarkierungsproblem dieser Stufe eine Menge an Klassifikationen pro Eingabe nötig. Ziel ist es, für jedes Wort (beziehungsweise Wortvektor) einer Eingabe einzeln genau eine Klassifikation durchzuführen und auszugeben. Eingabe- und Ausgabelänge des Netzes sollen daher gleich sein. Dieser Ansatz wird auch als *Many-to-Many*-Netzarchitektur bezeichnet. Dafür ist es nötig, dass die Netz-Ausgabeschicht jedes der n Elemente der Eingabe einzeln auf eine Klasse abbildet. Dies wurde ursprünglich mit einem *Dense-Layer* und einer Sigmoid-Funktion erreicht, welche für eine Eingabesequenz aus n Wörtern jedoch nur ein einziges statt n Klassifikationsergebnissen ausgeben kann. Das *TimeDistributed-Layer* löst dieses Problem: Diese Schicht ermöglicht es, als Hülle (engl. *wrapper*) für eine weitere neuronale Netzschicht zu fungieren. Es wendet diese gekapselte Schicht auf jedes der n Elemente (Wortvektor) der Eingabe getrennt an. Die Werte werden sozusagen zeitversetzt (engl. *time-distributed*) in die gekapselte Schicht eingegeben und die Berechnungen in einer Ergebnismenge gesammelt. Somit kann dadurch, dass TimeDistributed-Layer in der modellierten Architektur aus Quelltextausschnitt 9.4 mit einem Dense-Layer kombiniert wird, unabhängig von der Sequenzlänge eine dedizierte Berechnung für jedes Element der Eingabe durchgeführt werden.
Da für die zweite Stufe der Klassifikation nach Abschnitt 8.1.2.2 drei Klassen DECL, SPEC, ELSE klassifiziert werden müssen, ergeben sich mit der Ziffer 0, mit der das Padding markiert wird, vier mögliche Ausgabeklassen. Deshalb wird ein Dense-Layer mit vier Ausgabeneuronen benötigt. Als Aktivierungsfunktion wird die mathematische Softmax-Funktion verwendet. Diese bildet die Werte eines Eingabevektors auf den Wertebereich zwischen Null und Eins ab, sodass sich die einzelnen Komponenten auf Eins aufsummieren. Die Ausgabe des TimeDistributed-Dense-Layers entspricht damit einer Menge aus n Klas-

sifikationen pro Eingabe, wobei eine einzelne Klassifikation wiederum aus einem Vektor von 4 Werten zwischen Null und Eins besteht.
Wird dem TimeDistributed-Layer ein (bidirektionales) LSTM (siehe Abschnitt 2.4.2.2) vorgeschaltet muss zusätzlich der Parameter `return_sequences = True` der LSTM-Schicht konfiguriert werden. Dieser bewirkt, dass statt einem einzelnen, zusammengefassten Vektor pro gesamter Eingabe, ein Vektor für jeden Zeitschritt sowie Element der Eingabe zurückgegeben wird. Die Länge der LSTM-Schicht-Ausgabe entspricht daher der Länge der Eingabesequenz. Auf diese Weise kann erreicht werden, dass durch das modellierte TimeDistributed-Layer in jedem Zeitschritt die pro-Wort-Ausgaben der LSTM-Schicht durch ein Dense-Layer verarbeitet werden können. Obwohl es auch möglich wäre, ein Netz zu modellieren, dass in einem Zeitschritt alle n Klassifikationen für eine Eingabe auf einen Schlag vorhersagt beziehungsweise ausgibt, ist die Verwendung eines TimeDistributed-Layers von Vorteil. Zum Einen ermöglicht diese Schicht es, das Sequenzmarkierungsproblem so zu erlernen, wie es definiert ist: ein Eingabewort auf eine Ausgabeklasse abzubilden und das für jeden Zeitschritt getrennt. Zum Anderen vereinfacht der Einsatz des TimeDistributed-Layers das neuronale Netz, da es weitaus weniger Gewichte und damit weniger Trainingszeit benötigt.
Mit dem Aufruf der Keras-Funktion `model.summary` in Zeile 16 wird der Aufbau des Netzes abgerufen. Dabei zeigt sich, dass schon bei geringer Netzgröße eine große Menge an Parametern (wiederum fast 800000) zu trainieren sind. Anschließend erfolgt die Modellkonfiguration mit `model.compile`. Für Mehrklassen-Klassifikationsprobleme wird die dazu bereitgestellte logarithmische Fehlerfunktion `categorial_crossentropy` zur Berechung der kategorischen Kreuzentropie und als Optimierer der häufig eingesetzte Rmsprob-Optimierer empfohlen [Gra13]. Außerdem soll für die spätere Modellevaluation die Metrik Genauigkeit gemessen werden.
Mit dem Aufruf von `model.fit` in Zeile 21 wird das Training des Klassifikators angestoßen. Als erste Konfiguration wird die Epochenanzahl auf 100 gesetzt. Für den Parameter der Paketgröße werden wie zuvor verschiedene Werte untersucht. Weiterhin wird die Klassenbalancierung mittels dem Parameter `sample_weight` umgesetzt. Dieser beeinflusst die Wichtigkeit der Klassen während des Trainings. Dazu wird die Gewichtung im Hinblick auf einen Ausgleich der unterschiedlich verteilten Klassen gewählt. Der Parameter wird in Zeile 4 als Pseudodarstellung gelistet und der `model.fit`-Methode übergeben. Für die tatsächliche Implementierung muss statt einem Faktor pro Klasse, ein Faktor pro Ausgabeklasse mitgegeben werden.
Nach Abschluss des Trainings wird die Genauigkeit des Modells zur besten Epoche mittels `model.evaluate` auf den Validierungsdaten bestimmt.

```
1   embedding_dim = 300
2   maximum_sequence_len = 135
3   # Pseudodarstellung der Klassenbalancierung
4   sample_weights = {'PAD':0, 'DECL':1.6, 'SPEC':0.44, 'ELSE':10.4}
5
6   model = Sequential()
7   model.add(Embedding(input_dim=vocab_size,
8                       output_dim=embedding_dim,
9                       input_length=maximum_sequence_len,
10                      mask_zero=True))
11  # Beginn Netzerweiterungen
12
13  # Ende Netzerweiterungen
14  model.add(TimeDistributed(Dense(4, activation="softmax")))
15
16  model.summary()
```

```
17    model.compile(optimizer='rmsprop',
18                  loss='categorical_crossentropy',
19                  metrics=['accuracy'])
20
21    history = model.fit(X_train, y_train,
22                  epochs=100,
23                  validation_data=[X_val, y_val, sample_weights],
24                  sample_weight=sample_weights,
25                  batch_size=32)
26    loss, accuracy = model.evaluate(X_val, y_val)
```

Quelltextausschnitt 9.4: Quelltext zur Stufe-2-Klassifikation mittels ANN

Randomisierte Train-Validate-Test-Aufteilung

Das entworfene Modell wird zunächst auf die randomisierte Train-Validate-Test-Trennung angewandt. Dabei wird in Tabelle 9.21 der Vergleich verschiedener Paketgrößen-Konfigurationen und lemmatisierten beziehungsweise tokenisierten Eingabedaten präsentiert.
Es zeigt sich, dass die Modellresultate, mit und ohne Lemmatisierung in der Vorverarbeitung sich kaum unterscheiden. Es wird daher die Variante ohne zusätzliche Lemmatisierung gewählt, da diese möglicherweise weniger Informationsverlust im Hinblick auf die Eingaberepräsentation durch Wortvektoren beinhaltet. Betrachtet man die Genauigkeit auf den Validierungsdaten in Abhängigkeit von der Paketgröße, so lässt sich ebenfalls kein großer Einfluss auf die Genauigkeit erkennen. Da die benötigte Trainingsdauer bei größeren Paketgrößen steigt, wird eine Paketgröße von 32 bevorzugt.
Insgesamt erreicht das erste, sehr einfache Netz `ANN1` mit einer Paketgröße von 32 und tokenisierten Eingaben die besten Ergebnisse von 85.2% Genauigkeit. Dies entspricht im Vergleich zum Referenzwert von 75.9% einer Verbesserung von fast 10 Prozentpunkten. Wichtig ist die Setzung des Embeddingparameters `mask_zero`. Dieser führt dazu, dass das Padding maskiert, das heißt, die Auffüllung der Eingabesequenzen auf eine einheitliche Länge mit 0, ignoriert wird. Ohne diesen Parameter würde die Klassifikation der Paddingmarkierungen ebenfalls in die Genauigkeitsberechnung miteinbezogen werden. Es ergäbe sich eine fehlleitende Genauigkeit auf den Validierungsdaten von 95.5% statt 85.2%, die zu Stande kommt, da das Netz das Padding immer korrekt klassifiziert.

Weiterhin wird der Einfluss der ungleichmäßig verteilten Klassen auf das Modellergebnis untersucht. Dafür wird im zugehörigen Klassifikationsbericht des betrachteten Modells `ANN1` in Abbildung 9.4 die Detektionsresultate einmal mit und ohne Klassenbalancierung aufgetragen. Dieser Bericht berechnet die Metriken Präzision (engl. *precision*), Ausbeute (engl. *recall*) sowie das F_1-Maß (engl. *f_1-score*) für jede der drei Klassen. Diese Metriken werden in den Grundlagen in Abschnitt 2.3 erläutert. In der letzten Spalte ist mit der Metrik Support (engl. *support*) die Anzahl der vorhandenen Klassenmarkierungen im Goldstandard angegeben. Es lässt sich erkennen, dass die Klasse der Kommandodeklarationen (DECL) mit 67% vergleichsweise schlechter detektiert wird als die Klasse der Kommandobeschreibungen (SPEC) mit 91% F_1-Maß. Dies liegt an der Ungleichverteilung der Ausgabeklassen in den Trainingsdaten und den daraus folgenden unterrepräsentierten Trainingsbeispielen für die Klasse DECL. Die Klasse der Sonstigen Äußerungen (ELSE) erreicht trotz der sehr geringen Anzahl an Markierungen ein hohes F_1-Maß von 82%. Dies kann mit dem vergleichsweise einfachen Klassifikationsproblem dieser Klasse begründet werden, welche inhaltlich fast ausschließlich aus sehr ähnlichen Begrüßungen besteht.
Um den Einfluss der ungleichen Klassenverteilung zu kompensieren, wird der Parameter `sample_weight` gesetzt. Der Fokus liegt dabei auf der unterrepräsentierten und für die zweite Klassifikationsstufe wichtige Klasse der Kommandodeklaration (DECL). Der Vergleich der Klassifikationsergebnisse im zweiten Teil von Abbildung 9.4 zeigt, dass die Ausbeute der unterrepräsentierten DECL-Klasse von 71% auf 91% steigt, die Präzision im Gegenzug

Tabelle 9.21: Vergleich der Genauigkeiten auf den Validierungsdaten verschiedener Paketgrößen für Stufe 2

Paketgröße	32	64	100	300
Tokenisierte Eingaben	85.1	85.0	85.1	84.9
Lemmatisierte Eingaben	85.2	85.1	85.1	85.2

```
Ohne Klassenbalancierung:
     precision    recall  f1-score   support
DECL      0.65      0.71      0.67      2482
SPEC      0.89      0.93      0.91      9272
ELSE      0.96      0.72      0.82       325

Mit Klassenbalancierung:
     precision    recall  f1-score   support
DECL      0.54      0.91      0.68      2482
SPEC      0.97      0.77      0.86      9272
ELSE      0.51      0.86      0.64       325
```

Abbildung 9.4: Ausgabe des einfachen `ANN1`-Netzes für Stufe 2

jedoch sinkt. So kommt es zu einem ähnlich hohen F_1-Maß von 68% wie ohne Klassenbalancierung. Die Werte der häufigsten Klasse SPEC verhalten sich gegensätzlich. Überdies sinkt die Genauigkeit auf den Validierungsdaten von 85.2% auf 80.1% mit Klassenbalancierung. Es wird daher keine Verbesserung der Genauigkeit mit der Balancierung erreicht, sondern ausschließlich die Verteilung der korrekt detektierten Klassen verschoben. Fehlende beziehungsweise falsche Markierungen müssen bei beiden Varianten daher mittels einer Heuristik im Anschluss ausgebessert werden. Daher fällt die Entscheidung darauf, präzisiere statt einer höheren Ausbeute an Kommandodeklarationen-Markierungen zu finden. Deshalb wird für die folgenden Modelle keine Klassenbalancierung verwendet.

Um die Klassifikationsgenauigkeit des ersten, einfachen Modells `ANN1` weiter zu steigern, werden verschiedene Netzarchitekturen für die Lösung des Mehrklassen-Klassifikationsproblems untersucht. Die hierbei betrachteten Schichten wurden im vorherigen Tabelle 9.12 vorgestellt. Diese Modellerweiterungen werden im Implementierungsausschnitt in Quelltextausschnitt 9.4 zwischen Embedding-Layer und Ausgabeschicht in Zeile 13 bis 15 eingefügt. Dabei werden als Eingabe zusätzlich zu den selbst trainierten Wortvektor-Repräsentationen (SWv) wiederum die vortrainierten Wortvektoren von Fasttext (VWv) verwendet. Außerdem werden die besten ermittelten Konfiguration aus dem vorherigen Modell übernommen. Dazu gehören tokenisierte Eingabesequenzen, eine Paketgröße von 32 sowie keinen Einsatz von Klassenbalancierungsmethoden. Die Modellresultate auf den Validierungsdaten bei randomisierter Train-Validate-Test-Trennung werden in Tabelle 9.22 veranschaulicht.
Als erste Erweiterung wird in Netz `ANN2` die Architektur des Netzes `ANN1` um eine vollvermaschte Schicht aus 50 Neuronen erweitert. Anschließend werden mit den Netzen `RNN1` und `RNN2` zwei verschiedene Architekturen mit jeweils einer LSTM-Schicht getestet. Mit der Architektur von Netz `RNN3` wird das einfache Netz `ANN1` um eine bidirektionale LSTM-Schicht ergänzt. Dies entspricht einer der am häufigsten eingesetzten Modellarchitekturen für Sequenzmarkierungsprobleme. Dies belegen auch die Arbeiten von Schmidhuber [HS97], Gers [GSC00] und Manning [Man11] für die Probleme der Wortarten- beziehungsweise der Eigennamenerkennung. Auch die verwandte Arbeit von Rabinovich [RSK17] nutzt dieses Vorgehen. Diese bidirektionale Architektur wird in den Netzen `RNN5`, `RNN6` und `RNN7` durch verschiedene Kombinationen aus Dense- und Dropout-Schichten weiter ergänzt. Zuletzt wird in `RNN4` eine bidirektionale GRU-Schicht als Erweiterung für die Netzarchitektur des Klassifikators getestet. Die verwendeten Netzschichten wurden in **??** bereits vorgestellt.

Tabelle 9.22: Genauigkeiten auf den randomisierten Validierungsdaten verschiedener neuronaler Netze für Stufe 2

Name	Erweiterung	SWv	VWv
ANN1	`-`	85.1	85.1
ANN2	`D(50)`	85.3	85.3
RNN1	`LSTM(128)`	97.4	97.8
RNN2	`LSTM(128), D(64)`	97.3	97.7
RNN3	`BiLSTM(128)`	98.6	98.7
RNN4	`BiGRU(128)`	98.4	98.5
RNN5	`BiLSTM(128), D(100), DO(0.3), D(50)`	98.2	98.2
RNN6	`BiLSTM(128), DO(0.2)`	98.5	98.8
RNN7	`BiLSTM(256), DO(0.2)`	98.6	98.7

Das beste Ergebnis auf den Validierungsdaten aus Tabelle 9.22 wird mit dem Netz `RNN6` mit 98.8% Genauigkeit und vortrainierten Wortvektoren erreicht. Dies entspricht einer weiteren Verbesserung des simplen `ANN1` um 13.7 Prozentpunkte. Das Netz basiert auf einer bidirektionalen LSTM-Netzarchitektur. Auch die anderen Bi-LSTM-Netze erreichen eine ähnlich gute Genauigkeit. Die erfolgreiche Klassifikation mit dieser Architektur wiederholt sich hierbei zu den Ergebnissen der binären Klassifikation in Stufe 1 (siehe Tabelle 9.13). Die LSTM-Modelle ohne bidirektionale Schicht erzielen etwas schlechtere Ergebnisse. Das einfache Modell `ANN2` mit zusätzlicher vollvermaschten Schicht (Dense-Layer) schneidet deutlich schlechter ab. Daraus lässt sich vermuten, dass dies die Komplexität des Klassifikationsproblems in seiner einfacheren Modellarchitektur nicht abbilden kann. Weiterhin führt die Verwendung der vortrainierten im Vergleich zur Verwendung der selbst trainierten Wortvektorrepräsentationen in allen Fällen zu einem gleich bis besseren Ergebnis. Dieses Verhalten war auch schon in der ersten Klassifikationsstufe zu erkennen (siehe Abschnitt 9.1.1.6).

Anschließend wird der Klassifikationsbericht zum besten Modell `RNN6` in Abbildung 9.5 betrachtet. Der trainierte Klassifikator erreicht für alle drei Klassen gute Klassifikationswerte. Diese sind an der F_1-Metrik abzulesen, wobei diese mit der Klasse der Kommandobeschreibung (SPEC) ihren Bestwert mit 99% Erkennungsgenauigkeit erreicht. Im Vergleich zu den Klassifikationsresultaten des einfachen Netzes in Abbildung 9.4, bessert sich besonders die Detektion der Klasse der Kommandodeklaration (DECL). Es wird eine Steigerung des F_1-Maßes von 67% auf 98% erzielt. Diese Besserung ist ausschlaggebend für die darauffolgende Teilaufgabe der Arbeit zum Aufbau von Methodendefinitionen, da diese Klasse im Datensatz unterrepräsentiert und damit eine gute Erkennung in den Eingaben besonders wichtig ist.

Zusätzlich zu den Trainingsausgaben wird der Kurvenverlauf des besten Netzes `RNN6` in **??** präsentiert. Auf der X-Achse wird wiederum die Anzahl der Epochen, auf der Y-Achse die erzielte Genauigkeit (links) beziehungsweise der Evaluationsfehler (rechts) aufgetragen. Die Validierungskurven schwanken etwas in ihrem Verlauf. Auch die Trainingskurven lassen mehrere kleinere Schwankungen erkennen, die in den vorherigen Kurvenverläufen nicht aufgetreten sind. Dies könnte darauf zurückzuführen sein, dass das betrachtete Modell eine Dropout-Schicht beinhaltet. Die Dropout-Schichten stellen eine Methode zur Verringerung von Überanpassungseffekten (engl. *overfitting*) dar. Dies wird dadurch umgesetzt, dass beim Training des Netzwerks ein festgelegter (hier 20%) Prozentsatz der Neuronen in jedem Layer des Netzes zufällig ausschaltet (engl. *dropout*) und für den kommenden Berechnungsschritt nicht berücksichtigt werden. Dadurch verliert das trainierte Netz in jedem Zustand einen Teil seiner Informationen, was dazu führen könnte, dass kurzfristig die Genauigkeit sinkt sowie der Fehler steigt. Weiterhin lässt sich erkennen, dass die Kurven

	precision	recall	f1-score	support
DECL	0.97	0.98	0.98	2482
SPEC	0.99	0.99	0.99	9272
ELSE	0.97	0.90	0.93	325

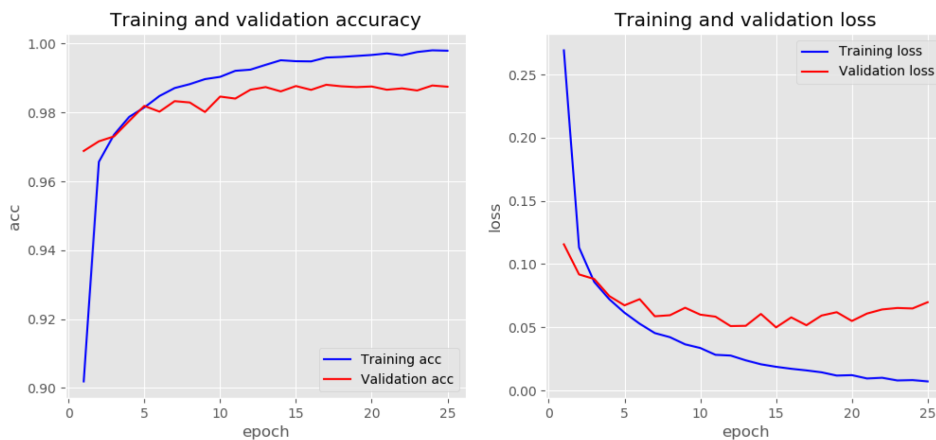


Abbildung 9.5: Ausgabe der besten neuronalen Modellarchitektur `RNN6` für Stufe 2

der Genauigkeit als auch des Fehlers nah beieinander liegen. Dies ist ein gutes Zeichen, da sich daraus schließen lässt, dass zu diesem Zeitpunkt wenig bis keine Überanpassung des Modells an die Trainingsdaten stattgefunden hat. Das Modell generalisiert weiterhin gut auf den Validierungsdaten. Zusätzlich dazu wird Zeitpunkt des geringsten Validierungsfehlers in Epoche 15 erreicht. An dieser Stelle wird das Modelltraining bei einer erreichten Validierungsgenauigkeit von 98.8% abgebrochen.

Für das binäre Entscheidungsproblem der ersten Stufe der Klassifikation in Abschnitt 9.1.1.7 wurde zusätzlich zu den selbst modellierten Modellen das Werkzeug BERT [DCLT18a] von Google angewendet (siehe Abschnitt 2.4.2.5). Jedoch steht zum Zeitpunkt dieser Arbeit für BERT noch keine öffentliche Schnittstelle zur Klassifikation von Mehrklassen-Entscheidungsproblemen zur Verfügung. Deshalb wird das Werkzeug nicht für die zweite Klassifikationsstufe herangezogen.
Anschließend wird aus den trainierten Netze für die anschließende Evaluation auf den Testdaten das beste Modell ausgewählt. Dies betrifft für die randomisierte Train-Validate-Test-Datensatztrennung das trainierte neuronale Netz `RNN6` mit vortrainierten Wortvektoren.

Szenarienbezogene Aufteilung

Die im vorherigen Absatz definierten Modellentwürfe werden anschließend auf die szenarienbezogene Trennung der Daten angewendet. Die daraus resultierenden Ergebnisse auf den Validierungsdaten werden in Tabelle 9.23 zusammengefasst. Dabei werden die Modelle sowohl für selbst trainierte Wortvektoren (SWv) als auch für die vortrainierten Wortvektoren von Fasttext (VWv) getestet.
Die beste Genauigkeit von 98.2% wird mit mehreren Netzen erzielt, unter Einsatz von vortrainierten als auch mit selbst trainierten Wortvektoren erzielt. Betrachtet man die Netzarchitektur dieser besten Modelle `RNN3`, `RNN6` und `RNN7`, so lässt sich eine Gemeinsamkeit in der Verwendung einer bidirektionalen-Netzschicht erkennen. Der positive Einfluss des Einsatzes von vortrainierten Wortvektoren lässt sich in diesen Modellergebnissen ebenfalls erkennen, jedoch fällt er nicht so stark aus wie zuvor bei der randomisierten Datentrennung in Tabelle 9.22 Für die nachfolgende Evaluation auf unbekannten Testdaten werden diese drei besten Modelle herangezogen. Ein Kurvenverlauf wird an dieser Stelle nicht nochmals diskutiert, da sich dieser dem präsentierten Verlauf des besten Modells `RNN6` aus

Tabelle 9.23: Genauigkeiten auf den szenarienbezogenen Validierungsdaten verschiedener neuronaler Netze für Stufe 2

Name	Erweiterung	SWv	VWv
ANN1	`-`	85.0	85.1
ANN2	`D(50)`	85.1	85.1
RNN1	`LSTM(128)`	97.3	97.3
RNN2	`LSTM(128), D(64)`	97.0	97.1
RNN3	`BiLSTM(128)`	98.2	97.9
RNN4	`BiGRU(128)`	97.6	98.1
RNN5	`BiLSTM(128), D(100), DO(0.3), D(50)`	97.8	98.1
RNN6	`BiLSTM(128), DO(0.2)`	98.2	98.1
RNN7	`BiLSTM(256), DO(0.2)`	98.1	98.2

der randomisierten Datentrennung in Abbildung 9.5 sehr ähnelt.

9.1.2.5 Modellevaluation

Im Anschluss auf den Entwurf und das Training der neuronalen Netze, werden diese auf unbekannten Testdaten evaluiert. Für die Modellevaluation werden dazu die im vorherigen Schritt ausgewählten Architekturen betrachtet, die auf den Validierungsdaten die besten Ergebnisse erzielen. In Tabelle 9.24 werden die Testresultate getrennt für beide Train-Test-Split-Aufteilungen präsentiert. Im oberen Tabellenabschnitt werden hierzu die ausgewählten Netze aus der randomisierten Modellevaluation präsentiert (siehe Tabelle 9.22), währenddessen im unteren Tabellenabschnitt die drei gleich guten Modelle der szenarienbezogenen Datentrennung (siehe Tabelle 9.23) aufgeführt sind.
Das beste Modell `RNN6 + VWv` aus der randomisierten Train-Test-Validate-Aufteilung erzielte eine Test-Genauigkeit von 98.8%. Dies entspricht dem Wert aus dem Modelltraining. Das Netz ist somit im Stande, auf ungesehenen Testdaten gut zu generalisieren. Selbiges Modell auf Basis von selbst trainierten Wortvektoren (`RNN6 + VWv`) schneidet auf den Testdaten geringfügig schlechter ab mit 98.4% Genauigkeit.
Bei der szenarienbezogenen Datenaufteilung erreichten gleich drei Modell `RNN3 + SWv`, `RNN6 + SWv` und `RNN7 + VWv` die beste Validierungsgenauigkeit von 98.2%. Auf den Testdaten schneidet jedoch das komplexeste Modell `RNN7 + VWv` sowie unter Einsatz von vortrainierten Wortvektoren am besten ab mit 97.5% Genauigkeit. Die Testgenauigkeit fällt im Vergleich zu den Genauigkeiten auf den Validierungsdaten somit geringfügig ab. Diese Tendenz der geringfügig schlechteren Testgenauigkeit bei der szenarienbezogenen Datentrennung wiederholt sich zum binären Entscheidungsproblem der ersten Klassifikationsstufe (siehe Abschnitt 9.1.1.7). Der Grund dafür liegt wiederum an dem, für das Modell unbekannte Testszenario und den damit verbundenen neuen Formulierungen und verwendeten Wörtern. Hinzu kommt, dass in der szenarienbezogenen Datentrennung das Vokabular eines Szenarios nicht in das Modelltraining mit einfließt. Daher wirkt sich besonders in diesem Fall der Einsatz von vortrainierten Wortvektoren positiv aus, da diese mögliche, zusätzliche Informationen zu den unbekannten Wörtern mit einfließen lassen können. Deshalb zeichnet sich die Steigerung bei Vergleich der Verwendung von selbst trainierten versus vortrainierten Wortvektoren noch deutlicher aus, als bei der randomisierten Verteilung.

Zusammenfassend führt auch hier die Verwendung von vortrainierten Wortvektorrepräsentationen zu einer gesteigerten Genauigkeit. Alle genannten Modelle, die für die Modellevaluation in Betracht gezogen wurden, basieren auf einer bidirektionalen Architektur und LSTM-Schichten (siehe Abschnitt 2.4.2.2). Die Verschlechterung der Testgenauigkeiten bei der randomisierten im Vergleich zur szenarienbezogenen Datentrennung hält sich mit durchschnittlich 1.8 Prozentpunkten pro Modell sehr in Grenzen. Daher ist das erzielte

Tabelle 9.24: Test-Genauigkeiten der Stufe-2-Klassifikation

	Randomisiert		Szenarienbezogen	
	Validate	Test	Validate	Test
Referenzwert	75.9	-	75.7	-
RNN6 + VWv	98.8	98.8	98.1	97.5
RNN3 + SWv	98.6	98.3	98.2	96.0
RNN6 + SWv	98.5	98.4	98.2	95.8
RNN7 + VWv	98.7	98.5	98.2	97.5

```
Randomisierte Verteilung:
     precision    recall  f1-score   support
DECL      0.96      0.97      0.97      3036
SPEC      0.99      0.99      0.99     11507
ELSE      0.99      0.87      0.93       458

Szenarienbezogene Verteilung:
     precision    recall  f1-score   support
DECL      0.95      0.95      0.95      3904
SPEC      0.98      0.99      0.99     15298
ELSE      0.97      0.85      0.91       586
```

Abbildung 9.6: Ausgabe des besten RNN3-Netzes auf den Testdaten für Stufe 2

Ergebnis als sehr gut zu betrachten. Zuletzt zeigen die zugehörigen Klassifikationsberichte in beiden Datenaufteilungen in Abbildung 9.6 ebenfalls durchweg gute F_1-Werte mit über 91% über alle Klassen. Da die Klasse der sonstigen Äußerungen (ELSE) in der nachfolgenden Teilaufgabe der Arbeit nicht von Bedeutung ist, wird unter Ausschluss dieser Klassenmarkierung sogar ein F_1-Wert von mindestens 95% für die Detektion der Lehrsequenz-Bestandteile erreicht.

Mögliche Gründe für die Fehlklassifikationen beziehungsweise die geminderte Generalisierbarkeit auf den Testdaten lassen sich, wie auch schon in Abschnitt 9.1.1.7, auf die Komplexität der natürlichen Sprache (H 1.1) sowie die Vielfalt der Formulierungen (H 1.3) zurückführen. Diese Herausforderungen wurden in Abschnitt 7.1 ermittelt und erschweren es dem Klassifikator vermutlich, gemeinsame Strukturen in den sich unterscheidenden Trainingsdaten zu finden. Möglicherweise entspricht auch die Komplexität des Entscheidungsproblems nicht der Größe der Datenbasis. In diesem Falle könnte eine Erweiterung der Datenbasis zu einer Verbesserung der Klassifikationen führen. Zudem spricht die Herausforderung (H 1.4) das Vorhandensein von nicht eindeutigen Grenzfällen des Mehrklassen-Entscheidungsproblems in den Trainingsdaten an. Diese könnten den Klassifikator zusätzlich verwirren, was wiederum zu Fehlklassifikationen führt.

9.1.3 Detektion von Aktionsbenennungen

Für die Detektion von Aktionsbenennungen wird im Entwurfskapitel in Abschnitt 8.1.3 der Ansatz gewählt, extrahierte Informationen eines bestehenden PARSE-Agenten zu nutzen. Dazu werden die semantischen Rollen des SRL-Agenten weiterverwendet, um die Aktionsnamen und zugehörigen Entitäten in der Nutzereingabe zu detektieren.

Der Semantische-Rollen-Erkenner (siehe Abschnitt 2.2.1.5) ist in Projekt PARSE als `SRLabeler`-Agent modelliert. Er nutzt die im PARSE-Graph angereicherten Informationen als Eingabe. Dazu gehören beispielsweise die Wortgruppen und Eigennamen durch den Zerteiler *Senna* [CWB+11], sowie die Instruktionsnummer-Annotationen aus der Arbeit von Kocybik [Koc15], welche den Bezugsrahmen einzelner Aktionen in der Eingabe markieren. Die Ausgabe des `SRLabeler` wird anschließend wieder im Graphen in Form von Kante-

nattributen gespeichert, wobei für den hiesigen Ansatz nur das Kantenziel, nicht die Kantenquelle betrachtet wird. Das Kantenziel entspricht dabei einer Rolle, die einem Knoten beziehungsweise dem betreffenden Wort der Nutzereingabe zugewiesen wurde. Um diese Annotationen zu erhalten, muss der SRL-Agenten im hiesigen Programmablauf des zuvor neu erstellten `MethodSynthesizer`-Agenten ausgeführt werden.
Für die anschließende Extraktion der semantischen Rollen aus dem Graphen wird die Klasse `SRLExtractor` erstellt. Sie durchsucht den PARSE-Graphen nach vorhandenen Annotationen des SRL-Agenten und extrahiert die zugehörigen Graphknoten. Ein Wort kann dabei mehrere und auch unterschiedliche Rollen von den umgebenden Wörtern zugewiesen bekommen. Wie bereits im Entwurf beschrieben, sollen jene Satzbestandteile der Nutzereingabe gewonnen werden, deren semantischen Rollen zum selben Hauptverb gehören. Diese zusammengehörigen Satzbestandteile eines Hauptverbs lassen sich im Kontext der Lehrsequenzen als Aktionsbenennung (siehe Definition 5.1.4), bestehend aus Aktionsnamen und zugehörigen Entitäten, interpretieren. Durch den `SRLabeler`-Agenten entsteht daher für jedes Hauptverb der Nutzereingabe eine Gruppierung von SRL-Annotationen. Neben den Hauptverben können auch weitere modifizierende Verben, wie Hilfsverben oder Modalverben, in der Nutzereingabe auftreten (siehe Beispiel Beispiel 8.1.4). Deshalb wird eine Liste von Graphknoten pro Gruppierung, jeweils befüllt mit den Haupt- und den dazugehörigen modifizierenden Verben, benötigt.
Im ersten Schritt müssen diese Verben (SRL-Rolle V) in der Nutzereingabe gefunden werden. Dazu stellt die Klasse des `SRLExtractors` die Methode `getMainAndModifierVSrlNodes` bereit. Um im Späteren zurückschließen zu können, welches dieser Verben das Hauptverb einer Gruppierung bildet, wird statt einer einfachen Liste die Datenstruktur einer `HashMap` gewählt. Diese stellt eine Abbildung (engl. *map*) zwischen allen Hauptverben einer Nutzereingabe sowie zusätzlich möglichen, modifizierenden Verben je Gruppierung her. Die genannten Verben bilden die Aktionsnamen eines Nutzerkommandos. In der Implementierung wird überdies beachtet, dass das Hauptverb einer Aktionsbenennung immer an erster Stelle eines Aktionsnamen steht. Die zweite wichtige Methode des `SRLExtractors` ist die Funktion `getSrlChunksFromMainVSrlNodes`. Diese liefert den gegebenen Hauptverben alle zugehörigen SRL-Knoten, welche die Gruppierung (engl. *chunk*) bilden. Auf Sprachebene bilden diese zugehörigen Knoten die Aktionsentitäten eines Nutzerkommandos, das heißt die Parameter der Funktionen (siehe Entwurf Abschnitt 8.1.3.1). Dabei soll die Trennung der Wörter in die einzelnen semantischen Rollen des SRL-Agenten in der Ausgabe erhalten bleiben. Es wird daher pro Hauptverb der Nutzereingabe eine Liste aus Graphknoten (`List<List<INode>> srlChunkList`), aufgeteilt in ihre annotierten SRL-Rollen, erstellt. Sie bilden die Aktionsbenennungen und damit die Ergebnisse der dritten und letzten Klassifikationsstufe.

9.2 Aufbau von Methodendefinitionen

In diesem Abschnitt wird die Implementierung des in Abschnitt 8.2 konzipierten Lösungsansatzes zum Aufbau von Methodendefinitionen vorgestellt. Sie bildet den zweiten Teil der Implementierung, welcher im Programmablauf des Agenten in Abbildung 8.1 durch die zweite Kartei visualisiert wird. Ziel ist es, die Klassifikation neuer Nutzereingaben zu ermöglichen und deren Ergebnisse auf Programmebene abzubilden.

Für den Aufbau von Methodendefinitionen ist eine Integration der Ergebnisse aus den drei hierarchischen Klassifikationsstufen in den Programmablauf des zuvor erstellten PARSE-Agenten `MethodSynthesizer` nötig. Abbildung 9.7 illustriert die Klassenmodellierung als UML-Diagramm, deren Implementierung nachfolgend beschrieben wird. Die Klassen lassen sich in drei Gruppierungen aufteilen, welche innerhalb der Implementierung durch getrennte Pakete modelliert sind. Das erste Paket `commandClassification` implementiert die Klassifikatorintegration der zuvor trainierten Modelle aus Abschnitt 9.1 in Java. Das

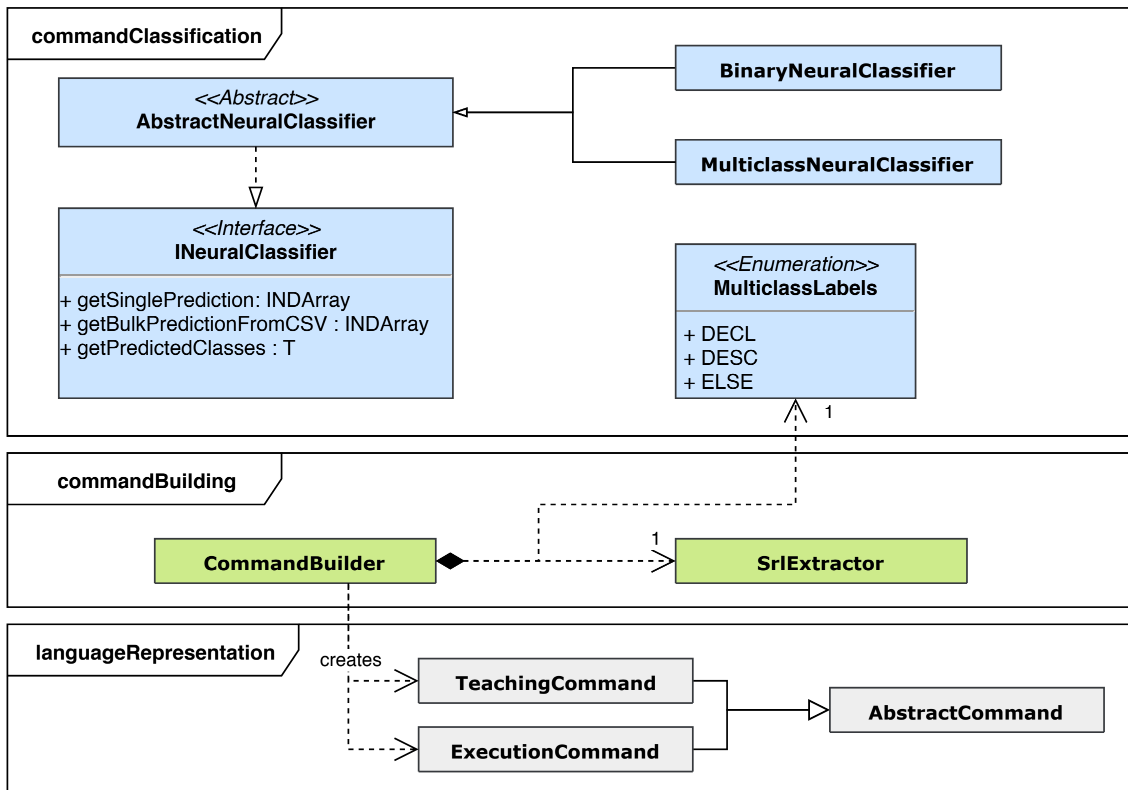


Abbildung 9.7: UML-Klassenhierarchie der ersten und zweiten Teilaufgabe der Arbeit

zweite Paket `commandBuilding` realisiert die in Abschnitt 8.2 beschriebene Datenstruktur zur Programmrepräsentation der Klassifikationsergebnisse im Programmkontext um. Dazu müssen die Ergebnisse der unterschiedlichen Stufen des hierarchischen Klassifikationsansatzes (siehe Abschnitt 7.3.1.2) zunächst zusammengefügt werden. Dies wird ebenfalls in diesem Paket umgesetzt. Zuletzt befinden sich benötigten Klassen für diese Datenstruktur im dritten Paket `languageRepresentation`.

9.2.1 Klassifikatorintegration in Java

Die Implementierung der ersten beiden Klassifikationsstufen zur Detektion von Lehrsequenzen (siehe Abschnitt 9.1.1 und Abschnitt 9.1.2) erfolgte unabhängig von Projekt PARSE in einer Pythonumgebung. Um die trainierten Klassifikatoren im Rahmen von PARSE dennoch nutzen zu können, müssen diese als Modelle in Java geladen werden. Die dafür benötigten Klassen werden in Abbildung 9.7 im Paket `commandClassification` dargestellt.

Dazu wird die Programmbibliothek *Deeplearning4J* genutzt, welche mit Hilfe von *Keras* eine Programmierschnittstelle zur Sprache Python zur Verfügung stellt. Auf diese Weise ist ein Import der künstlichen neuronalen Netze in eine Java-Umgebung möglich. Für den Modellimport werden dabei nur die besten, aus der Modellevaluierung in Abschnitt 9.1.1.7 und 9.1.2.5 hervorgehenden Klassifikatoren betrachtet. Diese können mittels der Keras-Funktion `myModel.save` als serialisiertes HDF5-Datenformat abgespeichert werden. Dieses Format wird insbesondere in wissenschaftlichen Anwendungen für die Speicherung großer Datenmengen verwendet. Anschließend können die Modelle in Java mittels dem Aufruf `KerasModelImport.importKerasSequentialModelAndWeights(myModel)` inkludiert werden. Da zum Zeitpunkt dieser Arbeit keine Java-Anbindung zur Integration des trainierten BERT-Sprachmodells zur Verfügung steht, werden nur die selbst trainierten neuronalen Keras-Modelle aus Abschnitt 9.1.2 eingesetzt.

Um anschließend Nutzereingaben mit diesen trainierten Modellen klassifizieren zu können, wird die Klasse `AbstractNeuralClassifier` und das Interface `INeuralClassifier` erstellt. Letzeres stellt die öffentlichen Methoden `getSinglePrediction` und `getBulkPredictionFromCSV` bereit, um Klassifikationen auf einer einzelnen beziehungsweise einer Sammlung von Eingabesequenzen durchzuführen. Die eigentliche Implementierung der Modellklassifikation wird innerhalb der zwei Unterklassen `BinaryNeuralClassifier` und `MulticlassNeuralClassifier` der Klasse `AbstractNeuralClassifier` durchgeführt. Sie erweitern die Oberklasse um spezifische Umsetzungen für die binäre beziehungsweise die Mehrklassen-Klassifikation der ersten beiden Klassifikationsstufen. Dort wird die individuell nötige Datenvorverarbeitung und Interpretation der Klassifikatorausgaben implementiert. Weiterhin wird über die Konfigurationsdatei `MethodSynthesizer.conf` die Möglichkeit zum Austausch der eingesetzten neuronalen Modelle bereitgestellt. Diese Möglichkeit zur Konfiguration des Agenten wurde in der Analyse in Abschnitt 7.3.2 als Anforderung (A4) an die Implementierung definiert. Im Anhang in Abschnitt B wird diese Konfigurationsdatei präsentiert.
Der Programmablauf des Agenten sieht dabei wie folgt aus: Zunächst wird die binäre Klassifikation auf der Eingabesequenz mittels `BinaryNeuralClassifier.getSinglePrediction` angestoßen. Diese entspricht der ersten Stufe der Klassifikationshierarchie. Wird eine Lehrsequenz (Klasse TS) erkannt, folgt die Mehrklassen-Klassifikation der zweiten Stufe mit selbiger Methode auf einer Instanz des `MulticlassNeuralClassifiers`. Die numerischen Ausgaben der zweiten Klassifikation werden als Enum-Klasse `MulticlassLabels` mit ihren entsprechenden Klassenmarkierungen (DECL, SPEC, ELSE) modelliert. Die Klassifikationsergebnisse werden somit als Liste dieser Enum-Elemente (`List<MulticlassLabels> mclassResults`) dargestellt. Wird in der Eingabe durch die erste Stufe keine Lehrsequenz (Klasse NO_TS) klassifiziert, so liegt eine Befehlssequenz vor. Hierbei wird die zweite Klassifikationsstufe übersprungen, die Nutzereingabe vollständig als Kommandobeschreibung (Klasse SPEC) interpretiert und direkt mit der dritten Stufe fortgefahren. Dieses Vorgehen wurde im Entwurfskapitel unter Abschnitt 8.2 erläutert und wird im Programmablauf des Agenten in Abbildung 8.1 durch die Aufspaltung des Verlaufs in der ersten Kartei dargestellt. Für den Fall, dass der Klassifikator der ersten Stufe die Nutzereingabe als Befehlssequenz klassifiziert, der Klassifikator der zweiten Stufe dennoch Kommandodeklarationen in der Eingabe erkennt, wird ein Sonderfall implementiert. Dieses Problem wurde in Abschnitt 7.1 mit der Herausforderung (H 2.4) definiert. Hierbei ist wichtig, dass sich eine klassifizierte Befehlssequenz (Klasse NO_TS) und gleichzeitig klassifizierte Kommandodeklarationen (Klasse DECL) widersprechen, da letztere Teile der Nutzereingabe als Lehrsequenz-Bestandteile interpretieren. Deshalb wird ab einer bestimmten Anzahl an detektierten Kommandodeklarationen, welche empirisch ermittelt und auf 6 festgelegt wurden, die Eingabe unabhängig von der binären Klassifikation als Lehrsequenz interpretiert.

Anschließend wird mit der Klasse des `SRLExtractors` die dritte Klassifikationsstufe (siehe Abschnitt 9.1.3) angestoßen. Sie extrahiert die vorhandenen Annotationen des Semantischen-Rollen-Erkenners aus dem PARSE-Graphen. Um sicherzustellen, dass SRL-Annotationen vorhanden sind, wird in der Initialisierung des `MethodSynthesizer`-Agenten deren Existenz überprüft und bei Bedarf die Ausführung des `SRLabelers` angestoßen. Der `SRLExtractor` liefert durch den Aufruf der Methode `getSrlChunksFromMainVSrlNodes` eine Menge von Listen (`List<List<INode>> srlChunkList`), jeweils befüllt mit Graphknoten der selben Gruppierung semantischer Rollen. Diese abhängigen Knoten lassen sich als eine Aktionsbenennung (siehe Definition 5.1.4) interpretieren und bilden die Ergebnisse der letzten Klassifikationsstufe. Die genannten Klassen werden in Abbildung 9.7 in der Komponente zur ersten Teilaufgabe dieser Arbeit visualisiert.

9.2.2 Zusammenführung der Klassifikationsergebnisse

Nachdem im ersten Schritt die Eingabe von Eingabesequenzen und deren Klassifikation ermöglicht wurde, bildet die Zusammenführung dieser Klassifikationsergebnisse den nächsten Schritt. Ziel ist es, die durch den `SRLExtractor` gewonnenen Aktionsbenennungen den Klassen aus der zweiten Klassifikationsstufe zuzuweisen. Die dafür benötigten Klassen werden in Abbildung 9.7 in den Paketen `commandClassification` und `commandBuilding` dargestellt.

Die durch den `SRLExtractor` gewonnene Menge von Listen zusammengehöriger Graphknoten (`List<List<INode>> srlChunkList`), den Aktionsbenennungen, wird im Folgenden genauer betrachtet. Für jeden Graphknoten aus `srlChunkList` wird geprüft, welche Klassenmarkierung durch den Klassifikator zugewiesen wurde. Dazu dient der Index des Graphknoten in der Nutzereingabe als Index in den Klassifikationsergebnissen `List<MulticlassLabels> mclassResults`. Es soll eine Liste aus `MulticlassLabels` erstellt werden, deren Elemente aus einer Stufe-2-Klassenmarkierung für jede vom SRL-Agenten extrahierte Aktionsbenennung besteht. Dazu wird diejenige Klassenmarkierung, welche am häufigsten für die Graphknoten jeder inneren Liste von `srlChunkList` vergeben wurde, als Klassenmarkierung für die gesamte Liste (hier Aktionsbenennung) gesetzt. Tritt der Fall auf, dass für die Knoten einer Aktionsbenennung eine ausgeglichene Verteilung von Stufe-3-Klassifikationsergebnissen vorliegt, so wird die Umgebung der Aktionsbenennung betrachtet: Falls vorhanden, wird die Klassenmarkierung des Vorgängerwortes übernommen, ansonsten wird die zugewiesene Klasse des Wortes im Satz nach der Aktionsbenennung gewählt. Durch den Mehrheitsentscheid soll gewährleistet werden, dass mögliche Klassifikationsfehler weniger stark einfließen können. Mit diesem Vorgehen können die Klassifikationsergebnisse der zweiten und dritten Klassifikationsstufe in der `List<MulticlassLabels> labeledChunkList` zusammengeführt werden. In Beispiel 9.2.1 wird diese Implementierung an einer exemplarischen Eingabe veranschaulicht. In der ersten Spalte der Tabelle sind die Stufen der hierarchischen Klassifikation angegeben, in allen weiteren Spalten werden die Klassifikationsresultate der einzelnen Stufen gezeigt. Da die binäre Klassifikation der ersten Stufe nicht wortweise, sondern pro ganzer Nutzereingabe durchgeführt wird, ist das Ergebnis hier sinnbildlich für das erste Wort der Eingabesequenz angegeben. Die Implementierung des beschriebenen Programmablaufs wird innerhalb der Klasse `CommandBuilder` durchgeführt. Diese wird in Abbildung 9.7 in der zweiten Komponente dargestellt ist.

Beispiel 9.2.1: Zusammenführung der Klassifikationsergebnisse

```
mclassResults = [ SPEC, DECL, DECL, SPEC, DECL, SPEC, SPEC, SPEC ]
srlChunkList  = [ [ "V", "A1"], ["V", "A1", "A1"] ]

Stufe   in    order  to    make  coffee  locate  the   cups
    1   TS    -      -     -     -       -       -     -
    2   SPEC  DECL   DECL  DECL  DECL    DECL    SPEC  SPEC
    3   -     -      -     V     A1      V       A1    A1

labeledChunkList = [ DECL, SPEC ]
```

9.2.3 Programmrepräsentation der Klassifikationsergebnisse

Nachdem im vorherigen Schritt die Klassifikationsergebnisse zusammengeführten wurden, bildet die Erstellung einer Programmrepräsentation dieser Ergebnisse den nächsten Schritt dieser zweiten Teilaufgabe der Arbeit. Dabei müssen die Klassifikationsresultate als Programmelemente (siehe Tabelle 8.5) innerhalb einer Datenstruktur modelliert werden.

Für die Modellierung dieser Programmelemente wird eine Menge von Klassen erstellt und im Paket `languageRepresentation` gesammelt abgelegt. Dazu gehört die Repräsentation einer Lehrsequenz mit der Klasse `TeachingCommand` sowie die Klasse `ExecutionCommand` zur Abbildung einer Befehlssequenz. Die abstrakte Klasse `AbstractCommand` stellt das Oberkonzept der Datenstruktur zur Repräsentation der Klassifikationsergebnisse der ersten Teilaufgabe dieser Arbeit (siehe Abschnitt 8.2) dar. In der Herausforderung (H 2.1) in Abschnitt 7.1 wird eine Allgemeingültigkeit der Datenstruktur gefordert, die mit der folgenden Implementierung erfüllt werden soll. Für die Modellierung der Aktionsbenennungen wird die abstrakte Klasse `AbstractInstruction` für die Aktionsnamen, und die abstrakte Klasse `AbstractInstructionParameters` für die Aktionsentitäten erstellt. Diese Aktionsbenennungen können wiederum jeweils in den Blöcken der Kommandodeklaration (Klasse DECL), Kommandobeschreibungen (Klasse SPEC) und sonstigen Äußerungen (Klasse ELSE) auftreten. Daher werden für beide abstrakte Klassen jeweils drei weitere Klassen erstellt, die diese erweitern und die Elemente der `MulticlassLabels` (DECL, SPEC, ELSE) repräsentieren. Somit entstehen beispielsweise für die Repräsentation der Kommandodeklaration (Klasse DECL) die Klassen `DeclarationInstruction` und `DeclarationParameters`. Die Abbildung 9.8 veranschaulicht die Beziehungen zwischen den einzelnen Klassen als UML-Diagramm.
Abschließend entsteht somit eine Datenstruktur `TeachingCommand` für die Programmrepräsentation der Lehrsequenzen sowie eine Datenstruktur `ExecutionCommand` für die Befehlssequenzen. Deshalb werden diese beiden Klassen wiederum von einer weiteren abstrakten Klasse `AbstractCommand` vereinigt. Für die Befüllung der `TeachingCommand`-Datenstruktur wird zusätzlich eine Liste aus zusammengeführten Klassifikationsergebnissen pro erkannter Aktionsbenennung (`List<MulticlassLabels> labeledSrlChunkList`) verwendet. Anschließend wird für jede enthaltene Aktionsbenennung eine Instanz der Unterklassen von `AbstractInstruction` und `AbstractInstructionParameters`, je nach vergebener Klasse aus `MulticlassLabels` erstellt. Da eine Befehlssequenz als Skript aus Funktionsaufrufen interpretiert wird (siehe Abschnitt 8.2), kann die Datenstruktur `ExecutionCommand` direkt mit Instanzen der Klasse `DescriptionInstruction` für die Funktionsnamen sowie `DescriptionInstructionParameters` für die Funktionsparameter befüllt. Die einzelnen erstellten Klassen halten dabei jeweils eine Liste aus Graphknoten `List<INode> nodes` und einen Bezeichner `String name`, die die entsprechend klassifizierten Satzbestandteile der Nutzereingabe enthalten.
Das beschriebene Vorgehen zum Aufbau der Programmrepräsentation wird in der Klasse `CommandBuilder` implementiert, welche in Abbildung 9.7 in der zweiten Komponente dargestellt ist.

9.3 Ontologie-Abbildung

In diesem Abschnitt wird die Implementierung des in Abschnitt 8.3 konzipierten Lösungsansatzes zur Abbildung von Programmelementen auf Ontologiekonzepte vorgestellt. Sie bildet den dritten Teil der Implementierung, welcher im Programmablauf des Agenten in Abbildung 8.1 durch die letzte Kartei visualisiert wird. Ziel ist es, die im zweiten Teilschritt extrahierten Programmbestandteile wie Funktionsnamen und Funktionsparameter auf konkrete Individuen der Zielsystem-Ontologie abzubilden. Hierfür wird die vorgestellte Ontologie aus Tabelle 8.6 betrachtet, die durch das PARSE-Modul des `OntologyConnector` angebunden werden kann.

Die Programmrepräsentation der Nutzereingabe wird mit den Zielsystem-Funktionalitäten verknüpft. Auf diese Weise kann im Falle einer Lehrsequenz eine neue Funktionalität sowie im Falle einer Befehlssequenz ein Skript aus Funktionsaufrufen synthetisiert werden. Diese unterschiedlichen Ausgaben des Agenten äußern sich auch im Programmablauf des Agenten in Abbildung 8.1, in dem bei einer detektieren Befehlssequenz der Schritt der Signatur-

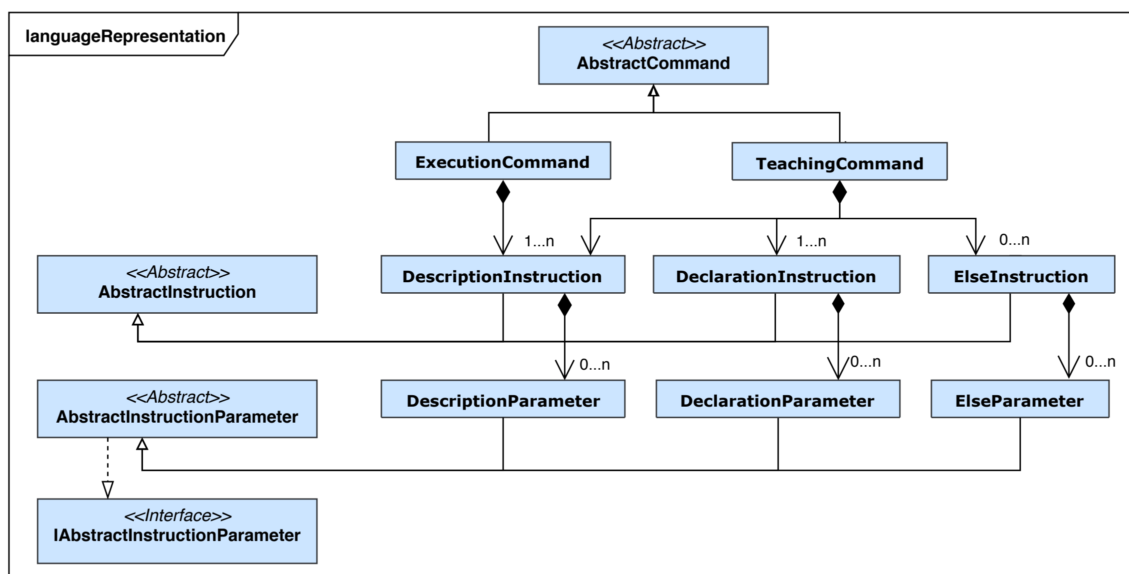


Abbildung 9.8: Programmrepräsentation der Klassifikationsergebnisse

Synthese zur die Erstellung einer Methodendefinition übersprungen wird.
In den nachfolgenden Abschnitten wird der Ansatz zur Synthese einer neuen Methodensignatur aus Abschnitt 8.3.1 sowie die Implementierung der Abbildungssuche aus Abschnitt 8.3.2 als weitere Komponente des entworfenen PARSE-Agenten `MethodSynthesizer` beschrieben. Anschließend wird die Umsetzung der Rangfolgeerstellung gefundener Abbildungen nach Abschnitt 8.3.2.3 sowie die Persistierung gefundener Kandidaten nach Abschnitt 8.3.3 im PARSE-Graphen vorgestellt. Die Abbildung 9.9 illustriert die Klassenmodellierung dieser Implementierung als UML-Diagramm, wobei die `OntologyMapper`-Klasse den Eingangspunkt für die Ontologie-Abbildung darstellt. Die Implementierung gliedert die benötigten Klassen in zwei Pakete auf. In dem Paket `ontologyMapping` wird die Logik der implementierten Signatur-Synthese sowie der Abbildungssuche umgesetzt. Das Paket `commandRepresentation` beeinhaltet dagegen Klassen, die für die Repräsentation gefundener Ontologie-Abbildungen nötig sind. Hierbei bildet stellt die Klasse `CommandCandidate` das Oberkonzept dieser Repräsentationen dar. Zuletzt erfolgt eine Evaluation der implementierten Ontologie-Abbildung im letzten Abschnitt.

9.3.1 Synthese einer Methodensignatur

Im diesem Abschnitt wird die Implementierung der Synthese einer Methodensignatur beschrieben, wobei der Entwurf aus Abschnitt 8.3.1 maßgeblich als Grundlage dieser Implementierung dient. Falls der Nutzer in seiner Eingabesequenz eine neue Funktionalität für das Zielsystem definiert, muss für diese eine Methodensignatur synthetisiert werden. Diese kann wiederum als Bezeichner für ein neues Methoden-Individuum verwendet werden, welches später die neue Funktionalität in der Zielsystem-Ontologie modelliert. Im Programmablauf des Agenten in Abbildung 8.1 wird dieser Schritt im ersten Block der Signatur-Synthese innerhalb der dritten Kartei dargestellt. Diese repräsentiert die dritte Teilaufgabe der Ontologie-Abbildung dieser Arbeit.

Für die Synthese einer Methodensignatur muss daher folgender Programmablauf vorausgesetzt werden: In der Eingabesequenz wird eine Funktionserweiterung des Zielsystems durch den Nutzer verbalisiert. Nach dem Entwurf in Abschnitt 8.1 werden daraufhin in der ersten Komponente des implementierten Agenten eine Lehrsequenz und deren Bestandteile (siehe Tabelle 7.1, Herausforderung (H 1)) klassifiziert. Daraufhin wird in der zweiten Komponente dieses Agenten die Datenstruktur `TeachingCommand` erstellt (siehe

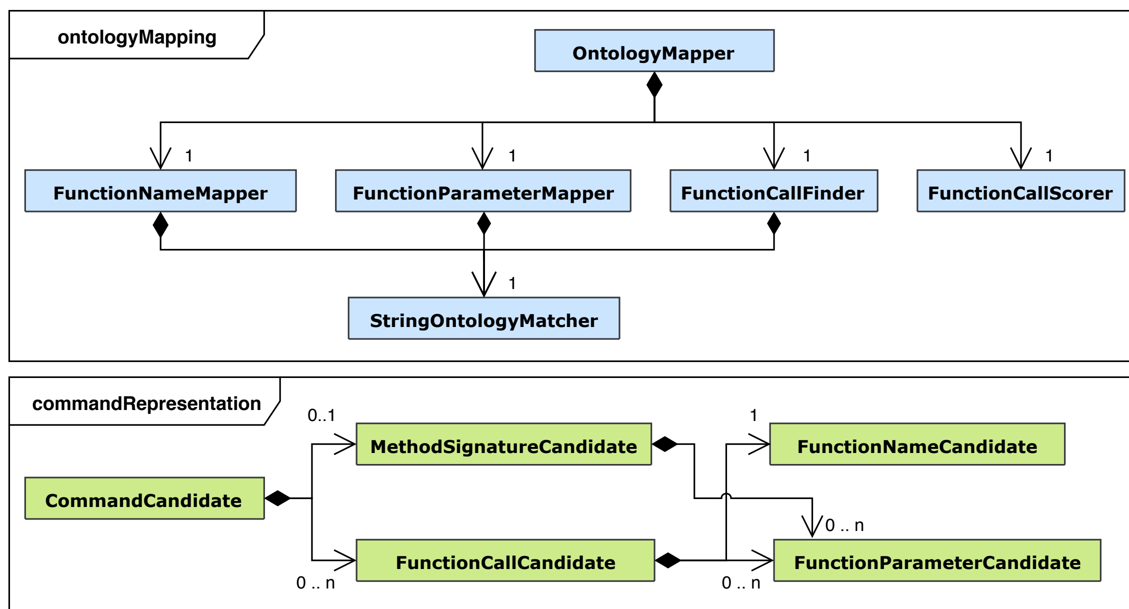


Abbildung 9.9: UML-Klassenhierarchie der dritten Teilaufgabe der Arbeit

Abschnitt 8.2, Herausforderung (H 2)), welche die Lehrsequenz als Methodendefinition auf Programmebene modelliert. Diese Datenstruktur besteht aus modellierten Klassen zur Repräsentation der Programmbestandteile des Methodenkopfes und des Methodenrumpfes und wurde in Abschnitt 9.2.3 vorgestellt. Das UML-Klassendiagramm in Abbildung 9.8 präsentiert den Aufbau dieser Datenstruktur. Relevant für die Implementierung der Synthese von Methodensignaturen ist ausschließlich der Teil der Datenstruktur, der die Programmbestandteile eines Methodenkopfes modelliert. Zu diesen Programmbestandteilen gehören die Instruktionen der Kommandodeklaration (`DeclarationInstruction`, Definition 5.1.2) sowie die zugehörigen Parameter (`DeclarationParameter`). Aus diesen Elementen muss schließlich eine Methodensignatur synthetisiert werden.

Die Implementierung der Synthese von Methodensignaturen wird in der Klasse `OntologyMapper` durch die Methode `buildMethodSignature` umgesetzt. Für die Modellierung einer Methodensignatur wird die Klasse `MethodSignatureCandidate` erstellt. Sie bildet einen Teil der Modellierung der Agentenergebnisse, welche in der UML-Klassenhierarchie in Abbildung 9.9 im Paket `commandRepresentation` vorgestellt wird. Der Rückgabewert der Methode entspricht dementsprechend einer Instanz der Klasse `MethodSignatureCandidate`. Für die Synthese einer Methodensignatur müssen ein Name sowie benötigte Argumente definiert werden. Der Name wird aus den Instruktionen der Kommandodeklaration (`DeclarationInstruction`) gebildet. Wichtig hierbei ist, dass bei Existenz mehrerer Instruktionen, diese als gleichbedeutende Deklarationen der selben Funktion betrachtet werden (siehe Abschnitt 8.3.1). Die Argumente werden aus den Parametern dieser Instruktionen (`DeclarationParameter`) gewonnen.
Im ersten Schritt der Implementierung werden die Deklarationsinstruktionen vorverarbeitet. Neben der im Entwurf in Abschnitt 8.3.1 beschriebenen Vorverarbeitung werden auch Stoppwörter entfernt. Um die Stoppwortliste anpassen zu können, wird diese in die Konfigurationsdatei aus Abschnitt 9.2.1 unter dem Attribut `STOP_WORDS` integriert. Im Anhang in Abschnitt B wird diese Konfigurationsdatei präsentiert. Diese Möglichkeit zur Konfiguration des Agenten wurde in der Analyse in Abschnitt 7.3.2 als Anforderung (A4) an die Implementierung definiert. Nach dieser Bereinigung der Instruktionen erhält man die *bereinigte* Instruktionsmenge aus Definition Definition 8.3.2. Das beschriebene Vorgehen wird Zeile 1 in Algorithmus 1 in Form von Pseudocode dargestellt. Anschließend werden

Algorithmus 1 Programmablauf der Signatur-Synthese

```
1: function BUILDMETHODSIGNATURE(declarationInstructions)
2:     clearedInstructions ← NM.PREPROCESSNAMES(declarationInstructions)
3:     paramMatches = [ ]
4:     for decl in clearedInstructions do
5:         clearedParams ← PM.PREPROCESSPARAMETERS(decl)
6:         paramMatches.addAll(PM.FINDPARAMCANDIDATES(clearedParams))
7:     end for
8:     return methodSignature ← CREATEMETHODNAME(anyDecl, paramMatches)
9: end function
```

die Parameter dieser bereinigten Menge an Deklarationsinstruktionen ebenfalls vorverarbeitet. Auch hier lässt sich mit der Konfigurationsdatei die Filterung der Wortarten über den Wert `PARAMETER_POS` konfigurieren (siehe Abschnitt B). Die unterschiedlichen Vorverarbeitungsschritte werden durch die beiden Klassen `FunctionNameMapper` und `FunctionParameterMapper` jeweils getrennt für die Instruktionen sowie Instruktionsparameter implementiert. Es folgt eine namensbasierte Abbildungssuche (siehe Abschnitt 8.3.2.1) für die Instruktionsparameter durch den Funktionsaufruf von `findParamCandidates` (Zeile 6). Ziel ist es, für die extrahierten Instruktionsparameter (Definition Definition 8.3.1) eine Entsprechung in der Zielsystem-Ontologie zu finden, um sie als Argumente der synthetisierten Methodensignatur übergeben zu können. Hierbei wird in der namensbasierten Abbildungssuche eine Zuordnung auf Basis von Bezeichnerähnlichkeiten durchgeführt. Eine typbasierte Abbildungssuche (siehe Abschnitt 8.3.2.2) aus Basis des Vergleichs von Datentypen ist nicht möglich, da die zu synthetisierende Methodensignatur zu diesem Zeitpunkt noch nicht als Methoden-Individuum in der Ontologie zu finden ist. Die Implementierung dieser Abbildungssuchen wird im nachfolgenden Abschnitt 9.3.2 beschrieben. Die gefundenen Parameter-Abbildungen werden daraufhin in einer gemeinsamen Liste über alle Deklarationsinstruktionen gespeichert (Zeile 6). Dieses Vorgehen soll die Wahrscheinlichkeit für das Finden von Abbildungen auf die Ontologie erhöhen und lässt sich für die Synthese der Methodensignatur nur deswegen über alle Instruktionen hinweg durchführen, da diese wie bereits genannt, als Synonyme betrachtet werden können. Zuletzt wird in Zeile 8 von Algorithmus 1 die eigentliche Methodensignatur synthetisiert. Der Methodenname wird aus einer der synonymen Deklarationsinstruktionen und durch Konkatenation der Bezeichner der Instruktionsparameter gebildet. Im letzten Schritt werden auf die Ontologie abgebildete Parameter als Argument der synthetisierten Methodensignatur übergeben. Beispielsweise wird aus der Instruktion „make“ und dem Parameter „some cereals“ der Methodenname `makeSomeCereals(Cereals::Object)` erstellt, falls das Objekt-Individuum in der Ontologie per namensbasierter Abbildungssuche gefunden werden konnte. In Abschnitt 8.3.1 wurde dazu bereits ein ausführliches Beispiel 8.3.1 präsentiert. Hierbei muss zusätzlich sichergestellt werden, dass aufgrund der gemeinsamen Liste aus Parameter-Abbildungen keine Duplikate von Individuen-Abbildungen als Argument übergeben werden. Dies gilt auch für die konkatenierten Parameterbezeichner im Methodennamen.

Im Gegensatz zur Erstellung des Methodenrumpfes, bei dem für jede Instruktion eine Rangfolge an Kandidaten zurückgegeben wird, wird für den Methodenkopf nur eine einzige Methodensignaturen synthetisiert. Dies hat zur Folge, dass für die namensbasierte Abbildungssuche zum Finden von Parameter-Abbildungen für die Argumente der Methodensignatur nur jene mit höchstem Bezeichner-Ähnlichkeitswert ausgewählt werden. Überdies wird entschieden, für die Argumente der Methodensignatur nur Parameter-Abbildungen mit einem Ähnlichkeitswert von 0.95 und höher zuzulassen. Dies soll verhindern, dass Methodensignaturen mit falschen Ontologie-Individuen besetzt werden. Die Bezeichner

der extrahierten Instruktionsparameter werden unabhängig davon immer an den synthetisierten Methodennamen angehängt. Diese zuletzt genannten Aspekte werden in der Implementierungsbeschreibung der namensbasierten Abbildungssuche in Abschnitt 9.3.2 nochmals angesprochen.

Zusammenfassend wird die auf diese Weise modellierte Methodensignatur nicht als tatsächliches neues Methoden-Individuum der Zielsystem-Ontologie in dieser Arbeit modelliert und hinzugefügt. Stattdessen werden die Ergebnisse nach dem Entwurf in Abschnitt 8.3.3 in der gemeinsamen Graphdatenstruktur des Projekt PARSE persistiert.

9.3.2 Abbildung auf bestehende Individuen

In diesem Abschnitt wird die Implementierung der Abbildung auf bestehende Individuen beschrieben, deren konzipierter Ansatz in Abschnitt 8.3.2 vorgestellt wurde. Ziel ist es, die extrahierten Bestandteile einer Nutzereingabe auf Zielsystem-Bibliotheksaufrufe abzubilden. Im Programmablauf des Agenten in Abbildung 8.1 wird dieser Schritt im Block der Abbildungssuche der dritten Kartei entsprechend der dritten Teilaufgabe der Ontologie-Abbildung dargestellt.

Nach der Klassifikation der Lehrsequenz-Bestandteile innerhalb der ersten Komponente des implementierten Agenten aus Abschnitt 9.1 wird in der zweiten Komponente daraufhin die Datenstruktur `AbstractCommand` für die Programmrepräsentation dieser Klassifikationsergebnisse erstellt. Diese Datenstruktur modelliert im Falle einer verbalisierten Lehrsequenz durch die Klasse `TeachingCommand` eine Methodendefinition. Im zweiten Fall wird durch die Klasse `ExecutionCommand` die Befehlssequenz als Skript modelliert. Beide Klassen erweitern die abstrakte Oberklasse `AbstractCommand` (siehe Abbildung 9.8). Relevant für die Implementierung der Abbildung auf bestehende Individuen ist ausschließlich der Teil der Datenstruktur, der den Methodenrumpf beziehungsweise das Ausführungsskript modelliert. In beiden Fällen bestehen diese Teile aus einer Sequenz von Funktionsaufrufen. Ein Funktionsaufruf lässt sich wiederum untergliedern in eine Instruktion der Kommandobeschreibung (`DescriptionInstruction`, Definition 5.1.3) sowie zugehörige Parameter (`DescriptionParameter`). Für diese Beschreibungsinstruktionen und Parameter müssen schließlich Abbildungen auf bestehende Individuen der Zielsystem-Ontologie gefunden werden.

Die Implementierung der Abbildung auf bestehende Ontologie-Individuen wird in der Klasse `OntologyMapper` durch die Methode `buildScriptOfFunctionCalls` umgesetzt. Für die Modellierung der Kandidaten eines Funktionsaufrufs wird die Klasse `FunctionCallCandidate` und für die Kandidaten der Funktionsnamen und Funktionsparameter jeweils die Klassen `FunctionNameCandidate` und `FunctionParameterCandidate` erstellt. Diese Klassenmodellierung ist im UML-Diagramm in Abbildung 9.9 modelliert im Paket `ontologyMapping` wiederzufinden. Hierbei deutet die Benennung der Klassen an, dass mehrere Kandidaten für jeden Bestandteil der Nutzereingabe erstellt werden. Nach dem entworfenen Ansatz aus Abschnitt 8.3.2 wird statt einer einzigen möglichen Abbildung eine Menge an möglichen Abbildungskandidaten zurückgegeben und mit einer Bewertungsfunktion aus Abschnitt 8.3.2.3 in eine Rangfolge überführt. Der vereinfachte Programmablauf dieser Abbildungssuche wird in Algorithmus 2 dargestellt.
Ähnlich zur Synthese von Methodensignaturen in Algorithmus 1 ist ebenfalls eine Bereinigung der Instruktionen (`DescriptionInstruction`) und Parameter (`DescriptionParameter`) nötig. Diese Vorverarbeitungsschritte sowie die folgende namens- und typbasierte Abbildungssuche wird in den Klassen `FunctionNameMapper` (NM) und `FunctionParameterMapper` (PM) implementiert. Damit diese Klassen sowohl für Instanzen der modellierten Kommandobeschreibung (`DescriptionInstruction`) als auch der Kommandodeklaration (`DeclarationInstruction`) annehmen können, werden die abstrakten Oberk-

Algorithmus 2 Programmablauf der Erstellung von Bibliotheksaufruf-Sequenzen

```
 1: function buildScriptOfFunctionCalls(descriptionInstructions)
 2:     clearedInstructions ← NM.preprocessNames(descriptionInstructions)
 3:     functionCalls = [ ]
 4:     for desc in clearedInstructions do
 5:         clearedParams ← PM.preprocessParameters(desc)
 6:         nameMatches ← NM.findNameCandidates(desc)
 7:         paramMatches ← PM.findParamCandidates(clearedParams)
 8:         callCandidates ← FCF.findCallCandidates(nameMatches, paramMatches)
 9:         scoredCallCandidates ← FCS.calculateScores(callCandidates)
10:         functionCalls.addAll(FCS.getTopN(scoredCallCandidates))
11:     end for
12:     return functionCalls
13: end function
```

lassen `AbstractInstruction` und `AbstractInstructionParameter` für die Implementierung verwendet. Anschließend muss für jede dieser Instruktionen separat eine Abbildungssuche durchgeführt werden (Zeile 4 bis 8), die zuletzt in einer Liste von bewerteten Funktionsaufruf-Kandidaten (Zeile 10) gespeichert und zurückgegeben wird. Dazu werden die zusätzlichen Hilfsklassen `FunctionCallFinder` (FCF) und `FunctionCallScorer` (FCS) erstellt. Diese Klassen werden ebenfalls im Klassendiagramm von Abbildung 9.9 dargestellt.

9.3.2.1 Namensbasierte Abbildungssuche

Die Implementierung der namensbasierten Abbildungssuche aus Abschnitt 8.3.2.1 wird durch die Klassen `FunctionNameMapper` und `FunctionParameterMapper` umgesetzt. Ziel ist es, für die extrahierten Instruktionsparameter (Definition Definition 8.3.1) eine Entsprechung in der Ontologie zu finden, um sie auf Bibliotheksaufrufe des Zielsystems abbilden zu können. Die namensbasierte Abbildungssuche trifft diese Zuordnungsentscheidungen auf Basis von Bezeichnerähnlichkeiten. In Abschnitt 8.3.2.1 wurde dazu bereits ein ausführliches Beispiel 8.3.3 präsentiert.

Das Projekt PARSE stellt mit dem `OntologyConnector`-Modul bereits eine Schnittstelle zur Domänenontologie von PARSE (siehe Tabelle 8.6) als auch für den Einsatz von möglichen Zeichenketten-Vergleichsverfahren zur Verfügung. Im Entwurf werden dafür die Distanzmetriken Jaro-Winkler und Fuzzy gewählt (siehe Abschnitt 2.2.2). Die Einbindung dieses Moduls sowie die Implementierung des Ensemble-Ansatzes dieser beiden Metriken wird in der Klasse `StringOntologyMatcher` (SM) implementiert. Dazu wird für die Repräsentation der unterschiedlichen Individuen der Ontologie auf das verfügbare Interface `IIndividual` zurückgegriffen. Weiterhin wird im Entwurf in Abschnitt 8.3.2.1 die Einbindung von Korreferenzinformationen (siehe Abschnitt 2.2.1.6) und Synonymen (siehe Abschnitt 2.1.5) beschrieben. Diese Anforderung an die Erweiterbarkeit des Agenten wurde in der Analyse in Abschnitt 7.3.2 als Anforderung (A3) definiert. Diese werden durch den `ContextAnalyzer`-Agenten des Projekts PARSE bereitgestellt. Falls diese Informationen im aktuell betrachteten PARSE-Graph durch eine vorherige Agentenausführung annotiert sind, so wird dies der `OntologyMapper`-Klasse als Einstiegspunkt der Ontologie-Abbildung mit dem boolschen Attribute `useContext` mitgeteilt.

In Algorithmus 3 wird ein grober Programmablauf der Implementierung der namensbasierten Abbildungssuche für Funktionsnamen dargestellt, welche durch die Klasse `FunctionNameMapper` bereitgestellt wird. Dafür wird für jede extrahierte Instruktion (`DescriptionInstruction`) die Methode `findFunctionNameCandidates` aufgerufen. Nach der bere-

its angesprochenen Vorverarbeitung der Funktionsnamen werden weitere Zeichenketten-Kombinationen durch Konkatenation der extrahierten Parameter erstellt (Zeile 3). Beispielsweise wird aus der Formulierung „take the milk out“ mit den entsprechend extrahierten Instruktionen und Parametern die Kombinationsmenge [„take“, „take out“, „take milk out“] gebildet. Anschließend wird mit der Methode `getMethodEnsembleMatches` der Klasse `StringOntologyMatcher` der entworfene Ensemble-Ansatz aus den Zeichenketten-Vergleichsverfahren umgesetzt. Das Beispiel 8.3.2 aus Abschnitt 8.3.2 verdeutlicht dieses Vorgehen an einer Reihe von exemplarischen Funktionsnamen. Es werden die empirisch ermittelten unteren Schranken für die Bezeichner-Ähnlichkeitswerte von 0.4 für Jaro-Winkler sowie 0.15 für die Fuzzy-Metrik festgelegt. Alle Ontologie-Methoden deren ermittelter Ähnlichkeitswert auf Basis des Vergleichs von Individuen-Bezeichner und den kombinierten Funktionsnamen niedriger als die festgelegten Schranken ausfällt, werden aussortiert. Anschließend werden für den Ensemble-Ansatz nur jene Ontologie-Methoden behalten, die mit beiden Distanzmetriken gefunden werden konnten. Zusätzlich dazu wird eine weitere Schranke von 0.4 des gemittelten Ähnlichkeitswertes beider Metriken festgelegt, um die Menge an Ontologie-Individuen weiter auszudünnen. Die so gefundenen zutreffenden Abbildungen werden in einer Liste aus Instanzen der Klasse `FunctionNameCandidate` gespeichert. Diese Klasse hält die Attribute des zutreffend abgebildeten Methodenindividuums (Definition 8.3.3), den ermittelten Ensemble-Ähnlichkeitswert sowie die zugehörige bereinigte Instruktion (Definition 8.3.1). Selbiges Vorgehen wird für die Abbildungssuche auf Basis von Synonymen dieser Funktionsnamen durchgeführt (Zeile 4), mit dem Unterschied, dass nur nahezu perfekte Übereinstimmungen mit dem Bezeichner eines Ontologie-Individuums in die Menge der Kandidaten aufgenommen werden. Als untere Schranke für den Ensemble-Ähnlichkeitswert dieser perfekten namensbasierten Zuordnungen wird ein Wert von 0.9 festgelegt. So wird beispielsweise für das Methoden-Individuum `locate::Method` das ermittelte Synonym „locate“ mit einem Ähnlichkeitswert von 1.0 zugelassen, während das Synonym „locomote“ wegen einer zu geringen Bezeichner-Übereinstimmung abgelehnt wird.

Für die namensbasierte Abbildungssuche der Funktionsparameter wird ein ähnliches Vorgehen in der Klasse `FunctionParamMapper` implementiert. Dies wird im vereinfachten Programmablauf in Algorithmus 4 dargestellt. Ein Parameter-Kandidat wird durch die Klasse `FunctionParameterCandidate` repräsentiert. Diese hält analog zur Klasse der Funktionsnamen-Kandidaten das zutreffend abgebildete Parameter-Individuum, den Ähnlichkeitswert sowie den zugehörigen extrahierten Parameter. Im Gegensatz zur Abbildung von Funktionsnamen ist der Rückgabetyp für die Funktionsparametersuche eine Liste aus Listen von Kandidaten (Zeile 2). Der Grund dafür ist, dass für jeden der extrahierten Parameter einer Instruktion eine getrennte Menge Parameter-Abbildungskandidaten gesammelt wird (Zeile 3). Außerdem müssen die Parameter unterschiedlich vorverarbeitet werden. Statt der Verben stehen nun die Entitäten im Fokus (siehe Abschnitt 8.3.2.1). Anschließend werden die Wörter jedes vorverarbeiteten Funktionsparameter-Bezeichners permutiert. So wird beispielhaft aus dem extrahierten Funktionsparameter „white cup“ wird die Permutationsmenge [„white cup“, „cup white“] gebildet. Mit der Methode `getParameterEnsembleMatches` der Klasse `StringOntologyMatcher` wird der entworfene Ensemble-Ansatz für namensbasierte Parameter-Abbildungen umgesetzt. Die Schranken für die Ähnlichkeitsmetriken sowie den Ensemble-Ansatz werden von oben übernommen und als globale Konstanten in der `StringOntologyMatcher` definiert. Da die Parameterbezeichner in den meisten Fällen aus mehreren Wörtern bestehen, wird die Möglichkeit nach der namensbasierten Suche von Teil-Zeichenketten (engl. *substrings*) implementiert. Für den exemplarischen Parameter `coffee machine dispenser` werden dadurch drei getrennte Zeichenketten-Vergleiche mit jedem Teilwort des Bezeichners und den Ontologie-Individuen vorgenommen, deren Ähnlichkeitswerte für das Ergebnis gemittelt werden. Analog dazu wird die na-

Algorithmus 3 Namensbasierte Abbildungssuche für Funktionsnamen auf O.-Methoden

```
1: function findFunctionNameCandidates(instruction)
2:     nameCandidates = [ ]
3:     combinedNames ← concatNames(instruction.getClearedFunctionName(),
                               instruction.getClearedParamNames())
4:     nameCandidates.addAll(SM.getMethodEnsembleMatches(combinedNames))
5:     nameCandidates.addAll(SM.getPerfectEnsembleMatches(synonyms))
6:     return nameCandidates
7: end function
```

Algorithmus 4 Namensbasierte Abbildungssuche für Funktionsparameter auf Ontologie-Individuen

```
1: function findFunctionParameterCandidates(instructionParams)
2:     paramCandidates = [ ][ ]
3:     for param in instructionParams do
4:         candidatesPerParam = [ ]
5:         permutedParamNames ← permuteWords(param.getClearedParamName())
6:         candidatesPerParam.addAll(SM.getParameterEnsembleMatches(
                                  permutedParamNames)())
7:         candidatesPerParam.addAll(SM.getPerfectEnsembleMatches(synonyms))
8:         paramCandidates.add(candidatesPerParam)
9:     end for
10:    return paramCandidates
11: end function
```

mensbasierte Suche mittels Synonymen der Instruktionsparameter durchgeführt (Zeile 7). Weiterhin werden an dieser Stelle auch die Korreferenzinformationen (siehe Abschnitt 2.2.1.6) des `ContextAnalyzer`-Agenten eingesetzt. Konnte dieser eine Korreferenz auflösen, so wird beispielhafte für die Formulierung „go to the fridge open it“ statt dem Personalpronomen „it“ das referenzierte Objekt „fridge“ als Eingabe für die Bezeichner-Vergleichsmetriken im Falle der zweiten Instruktion gewählt. Zuletzt werden die ermittelten Kandidaten (`FunctionParameterCandidate`) pro Funktionsparameter in einer Liste gespeichert und zurückgegeben (Zeile 10). Die beschriebenen Klassen sind im UML-Diagramm in Abbildung 9.9 im Paket `ontologyMapping` modelliert.

9.3.2.2 Typbasierte Abbildungssuche

Die Implementierung der typbasierten Abbildungssuche aus Abschnitt 8.3.2.2) wird durch die Klasse `FunctionCallFinder` (FCF) im gleichen Zug mit der Bewertung der typ-sicher abgebildeten Funktionsaufrufe (Definition 8.3.5) durch die Bewertungsfunktion aus Abschnitt 8.3.2.3 umgesetzt. Ziel ist es, die gefundenen, zutreffenden Funktionsnamen und Funktionsparameter (Definition 8.3.3) aus der namensbasierten Suche auf Basis des Vergleichs mit den Datentypen der Ontologie-Individuen zu validieren. Diese Datentypen der Ontologie-Individuen werden in Tabelle 8.7 vorgestellt.

In Algorithmus 5 wird ein vereinfachter Programmablauf der Implementierung der typbasierten Abbildungssuche dargestellt. In der zuvor beschriebenen namensbasierten Abbildungssuche wird für jede der extrahierten Funktionsnamen und Funktionsparameter eine Abbildungssuche per Zeichenkettenvergleich durchgeführt. Dabei wird für jedes Element eine Kandidaten-Menge an Ontologie-Individuen generiert. Diese besteht zum Einen aus den Ontologie-Kandidaten für die Funktionsnamen, modelliert durch eine Liste von `FunctionNameCandidate`-Instanzen (`nameMatches`, Zeile 1). Zum Anderen aus den Kandidaten für die Funktionsparameter, repräsentiert durch eine Liste von Listen aus `FunctionParam-`

`Candidate`-Instanzen, (`paramMatches`, Zeile 1). Anschließend bildet die Methode `findCallCandidates` aus der Klasse `FunctionCallFinder` (FCF, Zeile 1) für jede mögliche Kombination dieser beiden Kandidatenmengen, jeweils pro Instruktion der Nutzereingabe, eine Menge aus Funktionsaufruf-Kandidaten `FunctionCallCandidate`. Diese Klasse modelliert einen Funktionsaufruf-Kandidaten durch die Attribute eines Methoden-Kandidaten (`FunctionNameCandidate`), einer Liste aus Ontologie-Individuen für die Funktionsparameter (`FunctionParamCandidate`), einer Bewertung sowie einem Schlüssel-Wert-Speicher (engl. *key-value store*) aus Zuordnungen von Parameter-Kandidat auf Methodenargument. Die letzten beiden Attribute werden zu diesem Zeitpunkt noch nicht gesetzt. In Abschnitt 8.3.2.3 wurde dazu bereits in detailliertes Beispiel 8.3.4 präsentiert. Anschließend können diese als Funktionsaufruf-Kandidaten (`callCandidates`, Zeile 4) in der typbasierten Abbildungssuche im nächsten Schritt behandelt werden, deren Implementierung in der Klasse `FunctionCallScorer` (FCS) gekapselt wird. Dazu muss jedoch wiederum ein kartesische Produkt, diesmal zwischen den zutreffenden Parameter-Kandidaten (engl. *matched*, Definition 8.3.3) und den abzubildenden Argumenten des betrachteten Methoden-Individuums (engl. *needed*, Definition 8.3.4), gebildet werden. Diese Logik wird in Algorithmus 5 durch den Schlüssel-Wert-Speicher `matchedtoNeededParamCombinations` in Zeile 7 dargestellt.
Die Methode `checkDataType` bildet die Hauptmethode für die Implementierung der typbasierten Abbildungssuche (Zeile 9). Für für jedes Schlüssel-Wert-Paar aus `matchedtoNeededParamCombinations`, bestehend aus abgebildetem Parameter-Individuum und abzubildendem Methoden-Argument, vergleicht diese Methode die Datentypen. Bei einer Übereinstimmung der Datentypen muss dies als typ-sichere Zuordnung (Definition 8.3.5) gespeichert und anschließend bewertet werden (`validMatch` aus Zeile 11 und 14). Daraus folgt, dass ein Funktionsparameter, welcher die Typprüfung nicht besteht, trotz einer gefundenen Parameter-Abbildung aus der namensbasierten Suche, nicht in die Ergebnismenge aufgenommen wird. Die berechnete Bewertung sowie die typ-sichere Abbildung der Parameter-Kandidaten auf Ontologie-Argumente werden anschließend in den entsprechenden Attributen der betrachteten Instanz der Klasse `FunctionCallCandidate` gesetzt (Zeile 15 und 16). Nach der Speicherung dieses typ-sicheren Funktionsaufruf-Kandidaten muss dieses Vorgehen für jeden weiteren Funktionsaufruf sowie jede weitere Instruktion der Nutzereingabe wiederholt werden. Die beschriebenen Klassen sind im UML-Diagramm in Abbildung 9.9 im Paket `ontologyMapping` modelliert.

Überdies gibt es eine Reihe von Sonderfällen, die von der Implementierung behandelt werden müssen. Die Abbildung von primitiven Datentypen, wie Zeichenketten oder Zahlen, stellt ein Problem dar (siehe Tabelle 8.7). Diese werden nicht als Individuen in der Ontologie modelliert, da es nicht möglich beziehungsweise zielführend wäre, alle möglichen Zahlen und Zeichenketten durch konkrete Individuen zu repräsentieren. Deshalb können diese auch nicht durch die namensbasierte Abbildungssuche gefunden und somit innerhalb der typbasierten Abbildungssuche behandelt werden. Deshalb muss für Methoden-Individuen mit abzubildenden Argumenten aus primitiven Datentypen (Definition 8.3.4) eine Sonderbehandlung implementiert werden. Trifft der Algorithmus 5 bei der Erstellung des Schlüssel-Wert-Speichers `matchedtoNeededParamCombinations` aus Zeile 7 auf primitive Datentypen der geforderten Methodenargumente, so werden zusätzliche Schlüssel-Wert-Kombinationen mit den ursprünglich extrahierten (statt abgebildeten) Instruktionsparametern (Definition 8.3.1) hinzugefügt.
Weiterhin gibt es den Fall, dass für eine extrahierte Instruktion keine Methoden-Abbildung durch die namensbasierte Abbildungssuche gefunden werden konnte. Dieser fehlende Funktionsaufruf muss im Ergebnis des implementierten Agenten trotzdem ersichtlich sein. Daher wird der Liste aus Funktionsaufrufs-Kandidaten (`FunctionCallCandidate`) pro Instruktion ein leeres Element hinzugefügt. Die Sonderfälle zu leeren abzubildenden Argumentlis-

Algorithmus 5 Typbasierte Abbildungssuche für Funktionsparameter

```
 1: callCandidates ← FCF.findCallCandidates(nameMatches, paramMatches)
 2: function FCS.calculateCombinedScores(callCandidates)
 3:     validFunctionCalls = [ ]
 4:     for functionCall in callCandidates do
 5:         for ontologyMethod in nameMatches do
 6:             validMatches = [ ]
 7:             matchedtoNeededParamCombinations ← findCombinations(
                        ontologyMethod.getArguments(), paramMatches)
 8:             for paramPair in matchedtoNeededParamCombinations do
 9:                 validMatch ← checkDataType(paramPair)
10:                 if validMatch then
11:                     validMatches.add(paramPair)
12:                 end if
13:             end for
14:             score ← calculateFunctionCallScore(validMatches)
15:             functionCall.setScore(score)
16:             functionCall.setValidOntologyParametersMap(paramPair)
17:             validFunctionCalls.add(functionCall)
18:         end for
19:     end for
20:     return validFunctionCalls
21: end function
```

ten (Definition 8.3.4) der Methoden-Individuen sowie fehlende typ-sichere Parameter-Abbildungen werden im darauffolgenden Abschnitt 9.3.2.3 kurz diskutiert.

9.3.2.3 Bewertung der Abbildungskandidaten

Die Bewertung der gefundenen Ontologie-Abbildungskandidaten wird durch die Methode `calculateFunctionCallScore` in der bereits vorgestellten Klasse `FunctionCallScorer` implementiert (siehe Abbildung 9.9). Diese Methode ist Teil der typbasierten Prüfung und wird in Algorithmus 5 in Zeile 14 aufgerufen. Sie implementiert die konzipierte Bewertungsfunktion Gleichung 8.1 aus Abschnitt 8.3.2.3. Für die Bewertung der Funktionsaufruf-Kandidaten benötigt sie Informationen über die extrahierten (Definition 8.3.1) und zutreffend abgebildeten Funktionsnamen und Parameter (Definition 8.3.3). Weiterhin fließen die Anzahlen der abzubildenden Methodenargumenten (Definition 8.3.4) und typ-sicheren Parameter-Abbildungen (Definition 8.3.5) in die Bewertung mit ein. Die genannten Informationen werden als Attribute der Klasse des Funktionsaufruf-Kandidaten `FunctionCallCandidate` modelliert. In Abschnitt 8.3.2.3 wurde dazu bereits ein detailliertes Beispiel 8.3.7 präsentiert. Zusätzlich dazu wurde ein Bonus für perfekte erzielte Bezeichner-Ähnlichkeitswerte (>0.9 Distanzmetrik-Wert) bei der namensbasierten Abbildung von Funktionsnamen auf Methoden-Abbildungen vergeben. Dieser Bonus wurde empirisch ermittelt und auf den Faktor 1.5 festgelegt.
Zuletzt muss aus den berechneten Werten für die einzelnen Kandidaten einer Instruktion eine Rangfolge erstellt werden. Dies wird durch die Methode `getTopNCandidates` umgesetzt. Die Anzahl durch den Agenten zurückgegebener Kandidaten pro Instruktion lässt sich über die Konfigurationsdatei über den Wert `TOP_N` festlegen. In Anhang unter Abschnitt B wird diese Konfigurationsdatei präsentiert. Diese Möglichkeit zur Konfiguration des Agenten wurde in der Analyse in Abschnitt 7.3.2 als Anforderung (A4) an die Implementierung definiert.

Die Implementierung stellt zudem sicher, dass für Methoden-Individuen deren abzubilden-

den Argumente (Definition 8.3.4) nicht besetzt werden konnten, trotzdem eine Bewertung erhalten. Dieser Sonderfall tritt ein, falls keine Parameter-Abbildungen mit dem namensbasierten Ansatz gefunden werden konnten. Auch Funktionsparameter, für die keine typ-sichere Parameter-Abbildung (Definition 8.3.5) gefunden werden konnte, werden durch den Ansatz der typbasierten Suche nicht in die endgültige Kandidatenliste der typ-sicheren Funktionsaufrufe übernommen. Dennoch wird durch den implementierten Programmablauf sowie die konzipierte Bewertungsfunktion eine Bewertung auf Basis der abgebildeten Methode erhält. Dies gilt auch für Methoden mit leeren Argumentlisten.

9.3.3 Persistierung der Ergebnisse

Die Ergebnisse des implementierten Agenten müssen anschließend in der gemeinsamen Graphdatenstruktur des Projekts PARSE persistiert werden. Im Programmablauf des Agenten in Abbildung 8.1 wird dieser Schritt im letzten Block der dritten Kartei entsprechend der dritten Teilaufgabe der Ontologie-Abbildung dargestellt. In Abschnitt 8.3.3 wurde entschieden, für die Speicherung dieser Ergebnisse wiederum eine Graphdatenstruktur zu modellieren, die anschließend in den PARSE-Graph integriert wird. Die Abbildung 9.10 visualisiert die beschriebene Integration der Agentenausgaben in die Graphdatenstruktur von PARSE anhand einem Auschnitt eines Beispielsatzes „preparing coffee means grabbing a cup and (...)“. Die bestehenden PARSE-Knoten gehören dem Graphknoten-Typ `token` an und sind in der Grafik in der letzten Ebene zu finden.

Zunächst wird ein neuer Graphknoten-Typ `commandMapper` (grau) erstellt. Die Wurzel der neuen Graphdatenstruktur bildet ein Knoten diesen Typs mit dem booleschen Attribut `isTeachingSequence`, das angibt, ob es sich bei der verbalisierten Nutzereingabe um eine Lehrsequenz handelt. Anschließend wird die Unterteilung des Kommandos in Methodenkopf (Kommandodeklaration) und Methodenrumpf beziehungsweise Skript aus Zielsystem-Bibliotheksaufrufen (Kommandobeschreibung) aus der Programmrepräsentations-Datenstruktur aus Abschnitt 8.2 übernommen. Dafür werden zwei neue Knotentypen `declaration` (blau) und `description` (grün und orange) erstellt. Beide Knoten halten keine Attribute und dienen der reinen Gliederung der folgenden Daten. Jedoch ist es dadurch möglich, eine Anfrage an den PARSE-Graphen zu richten, welche beispielsweise alle Knoten des Typs für den Methodenrumpf zurückgibt. Zusätzlich dazu werden die einzelnen abgebildeten Funktionsaufrufe über einen weiteren Knotentyp `functionCall` gekapselt. Diese Knoten geben über das Integer-Attribut `number` den Index an, der die Reihenfolge des Funktionsaufrufs entsprechend dessen Auftreten in der Eingabesequenz beschreibt. In Abbildung 9.10 wird für die Beispieleingabe für den Methodenkopf ein Funktionsaufruf (blau), für den Methodenrumpf zwei Funktionsaufrufe (grün und orange) erstellt.

Schließlich müssen die einzelnen abgebildeten Funktionsnamen und Funktionsparameter modelliert werden. Dazu werden wiederum zwei neue Knoten des Typs `functionName` beziehungsweise `functionParameter` erstellt. Beide halten über die Attribute `ontologyMethod` beziehungsweise `ontologyParameter` vom Datentyp String den eindeutigen Bezeichner der Zielsystem-Ontologie für das entsprechende, abgebildete Individuum. Zusätzlich dazu besitzt der `functionName`-Knoten das Integer-Attribut `topN`. Dieses gibt an, an welchem Platz der Rangliste der betrachtete Funktionsaufruf steht. Über einen `functionCall`-Knoten werden daher nicht nur Funktionsname und Funktionsparameter eines einzelnen Kandidaten gekapselt, sondern die Menge aller Kandidaten für diesen Funktionsaufruf. Um die Größe dieser Menge je nach unterschiedlichen Anforderungen an den Agenten anpassbar zu machen, wird er in eine globale Konfigurationsdatei unter einem gleichnamigen Parameter `topN` integriert. Weiterhin speichern die `functionName`-Knoten über das Attribut `topNScore` vom Datentyp Double den über die Bewertungsfunktion in Abschnitt 8.3.2.3 berechneten Wert des gefundenen Funktionsaufrufs. Im Gegensatz dazu halten die `functionParameter`-Knoten neben dem Identifikator (ID) des abge-

Tabelle 9.25: Knotenattribute der erstellten Graphdatenstruktur

Attribut	Knotentyp	Datentyp	Beschreibung
isTeachingSequence	commandMapper	boolean	Angabe, ob Lehrsequenz
number	functionCall	int	Nummer eines Funktionsaufrufs
ontologyMethod	functionName	String	ID der Methoden-Individuums
topN	functionName	int	Nummer eines Funktionsaufrufs
topNScore	functionName	double	Nummer eines Funktionsaufrufs
ontologyParam	functionParameter	String	ID des Ontologie-Individuums
methodParameter-ToFill	functionParameter	String	ID des Parameter-Individuums
value	token	String	Wort der Eingabesequenz

bildeten Ontologie-Individuums `ontologyParameter` nur ein weiteres Attribut. Dieses Attribut gibt unter dem Namen `methodParameterToFill` den Identifikator des Parameter-Individuums der Ontologie-Methode an, in das der abgebildete Parameter „eingesetzt" werden soll. Tabelle 9.25 verdeutlicht diesen Zusammenhang mittels einer Auflistung der erstellten Knotenattribute nochmals. Bezogen auf das Graphmodellierungs-Beispiel in Abbildung 9.10 wird für die Funktionsaufrufe jeweils ein `functionName`-Knoten und ein `functionParameter`-Knoten erstellt. Die Identifikatoren der abgebildeten Ontologie-Individuen werden in der Grafik beispielhaft mit dem Kürzel `onto#<<Bezeichner>>` angegeben. Überdies werden für die zweite Instruktion „grabbing a cup" beispielhaft zwei Kandidaten gefunden, welche durch das Attribut `topN=1` beziehungsweise `topN=2` gekennzeichnet sind. Zuletzt müssen die erstellten Knoten mit Kanten verbunden und in den PARSE-Graphen integriert werden. Dazu wird ein neuer Kanten-Typ mit dem Namen `commandMapper` erstellt. Dieser besitzt keine weiteren Attribute und dient der Bildung einer Hierarchie der modellierten Knoten. Dazu wird der Wurzelknoten mit den beiden Knoten `declaration` und `description` für die Unterscheidung zwischen Methodenkopf und Methodenrumpf verbunden. Diese können jeweils aus einer Menge von Funktionsaufrufen, das heißt `functionCall`-Knoten, bestehen. Deshalb werden entsprechende Kanten von den Ausgangsknoten zu diesen `functionCall`-Knoten eingefügt. Ein Funktionsaufruf gliedert sich wiederum in einen einzigen Funktionsnamen (`functionName`-Knoten) und eine Menge von `functionParameter`-Knoten, die als seine Kindknoten durch entsprechende Kanten modelliert werden. Zuletzt soll eine Verbindung zwischen extrahierten Lehrsequenz-Bestandteilen und der ursprünglichen Nutzereingabe hergestellt werden. Diese Nutzereingabe wird in der globalen Graphdatenstruktur von PARSE durch eine Sequenz von Knoten des Typs `token` mit dem Attribut `value` modelliert, welches letztendlich das betrachtete Wort der Nutzereingabe angibt. Deshalb werden die erstellten Knoten für Funktionsname und Funktionsparameter mit ihren jeweilig zugehörigen Wörtern der Eingabesequenz durch weitere Kanten verbunden.

9.3.4 Evaluation der Abbildungssuche

Um eine Aussage über die Güte der implementierten Ontologie-Abbildung treffen zu können, wird diese Komponente evaluiert. Bei Eingabe einer Nutzereingabe soll geprüft werden, ob der entworfene Lösungsansatz zur dritten Teilaufgabe aus Abschnitt 8.3 die extrahierten Programmbestandteile auf ihre korrekten Entsprechungen in der Zielsystem-Programmschnittstelle abbildet. Dafür muss untersucht werden, ob die in der Nutzereingabe verbalisierten Zielsystem-Funktionalitäten auf die korrekten Individuen der betrachteten Ontologie abgebildet worden sind. Diese Überprüfung geschieht durch einen händischen Abgleich der Nutzereingaben-Bestandteile mit den ermittelten Abbildungen auf konkrete Ontologie-Individuen.

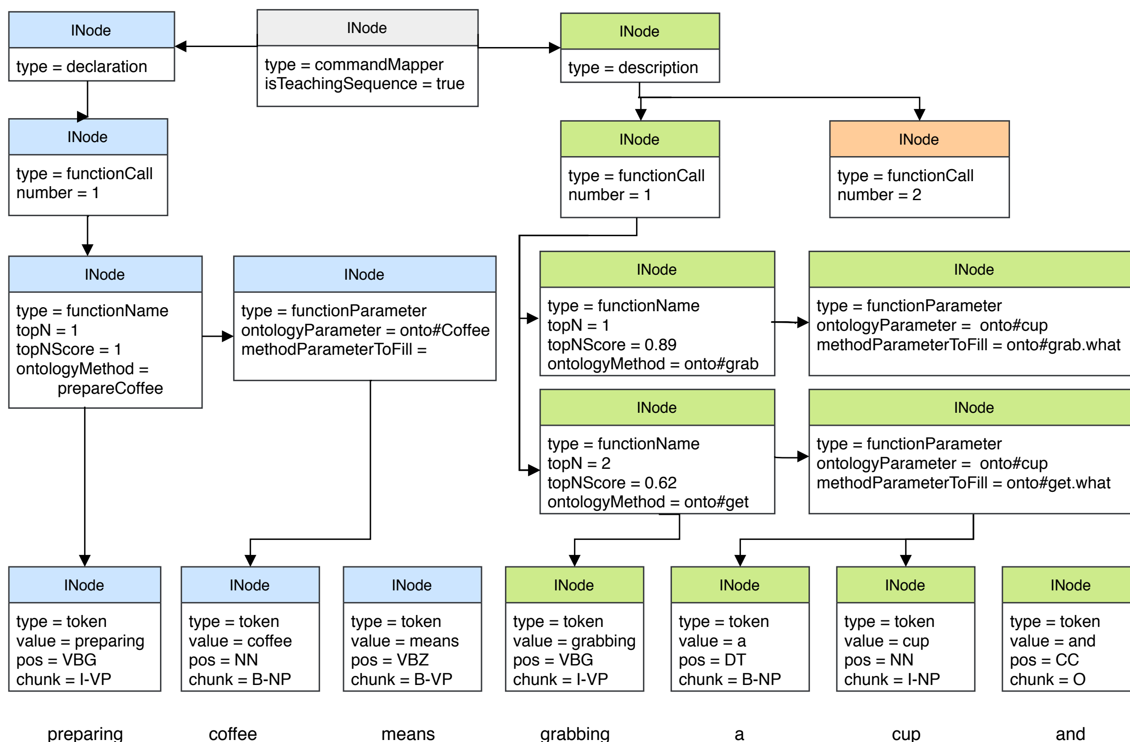


Abbildung 9.10: Integration der Agentenausgaben in die PARSE-Graphdatenstruktur

Tabelle 9.26: Umfang der Domänenontologie von PARSE

Individuum	Beispiel	Anzahl
Zielsysteme (`System`)	`Armar, Robot`	3
Objekte (`Object`)	`Bottle, Table`	53
Methoden (`Method`)	`bring, turnOff`	79
Zustände (`State`)	`closed, dirty`	10
Datentypen (`DataType`)	`boolean, Drinkable`	18
Methodenargumente (`Parameter`)	`bring.what, bring.where`	43
Werte (`Value`)	`Bottle, Table`	93
Gesamt		299

Im Folgenden wird zunächst die in der Evaluation betrachtete Domänenontologie vorgestellt. Die Struktur der Ontologie wurde bereits in Tabelle 8.6 in Abschnitt 8.3 erläutert. Interessant für die Evaluation der dritten Teilaufgabe ist jedoch vor allem der Umfang der betrachteten Ontologie, das heißt die Anzahl an unterschiedlichen Individuen der einzelnen Klassen (siehe Abschnitt 2.7). Diese werden in Tabelle 9.26 aufgeführt. Insgesamt enthält die Ontologie 299 Individuen. Darunter befinden sich 79 unterschiedliche Methoden-Individuen, welche für die Abbildung der Instruktionen auf Methoden-Individuen in Frage kommen. Diese fordern wiederum 43 unterschiedliche, abzubildende Methodenargumente (Definition 8.3.4). Weiterhin müssen für die Abbildung von Instruktionsparametern auf diese Argumente 293 Individuen betrachtet werden, die sich nach dem Entwurf aus Abschnitt 8.3 aus der Menge von Objekten, Werten und Zuständen zusammensetzt. Hierfür müssen innerhalb der typbasierten Abbildungssuche aus Abschnitt 9.3.2.2 auf weitere 18 verschiedene Datentypen geprüft werden. Diese wurden in Tabelle 8.7 bereits vorgestellt und erläutert. Zuletzt werden neben den annotierten Synonymen im PARSE-Graphen auch Synonyme über die `SameIndividual`-Relation in der Ontologie dargestellt. Dies betrifft 39 der vorhandenen Individuen.

Die Evaluation der Ontologie-Abbildung wird getrennt von den anderen Teilaufgaben durchgeführt, um die die Güte der Ergebnisse dieser Komponente unabhängig beurteilen zu können. Dieses Vorgehen wurde bereits in Abschnitt 9.1.1.7 und Abschnitt 9.1.2.5 zur separaten Evaluation der entworfenen Klassifikationsmodelle zur Detektion von Lehrsequenzen in der ersten Teilaufgabe dieser Arbeit verfolgt. Für die isolierte Bewertung der dritten Teilaufgabe muss eine, vom Rest dieser Arbeit entkoppelte, Eingabe für die implementierte Komponente erzeugt werden.
Die Ontologie-Abbildung kann jedoch nicht gesondert ausgeführt werden. Sie basiert auf der Erstellung der Programmrepräsentations-Datenstruktur der Nutzereingabe aus der zweiten Teilaufgabe, welche wiederum auf Basis der Klassifikationsergebnisse im ersten Schritt erstellt wird (siehe Herausforderungen (H 1) und (H 2) in Abschnitt 7.1). Deshalb muss sichergestellt werden, dass die Ausgaben der ersten beiden Komponenten bei Eingabe einer Nutzereingabe in den hiesigen Agenten ihrem korrekten, erwarteten Ergebnis entsprechen. Für die erste Teilaufgabe kann dies gewährleistet werden, indem den beiden selbst trainierten Klassifikatoren eine bereits bekannte Nutzereingabe übergeben wird. Die Ausgabe eines Modells auf bekannten Eingabedaten entspricht der beim Training gesehenen Musterlösung. Auf diese Weise wird erreicht, dass die Ausgabe der Klassifikatoren der ersten und zweiten Klassifikationsstufe (siehe Abschnitt 8.1.1 und 8.1.2) der Musterlösung der jeweiligen Eingabesequenz aus dem erstellten Goldstandard (siehe Abschnitt 8.1.2.2) entspricht. Dies gilt jedoch nicht für die dritte Stufe der Klassifikation (siehe Abschnitt 8.1.3). Diese nutzt einen externen Agenten von PARSE zur Annotation von semantischen Rollen und wurde nicht auf bekannten Eingabesequenzen trainiert. Daraus folgt, dass die Ergebnisse dieser dritten Stufe von den erwarteten, korrekten Ergebnissen abweichen können. Dies wirkt sich anschließend auf die zweite Teilaufgabe dieser Arbeit aus, welche den Aufbau von Methodendefinitionen übernimmt. Die zweite Komponente erhält die Klassifikationen und semantischen Rollen aus dem ersten Schritt als Eingabe. Aus diesen Klassifikationen wird daraufhin eine allgemeingültige Datenstruktur erstellt, welche die klassifizierten Sprachbestandteile mit konkreten Programmbestandteilen verknüpft. Schlussendlich lässt sich durch die Wiederverwendung der erhobenen Nutzereingaben aus der Vorstudie (Kapitel 6) eine gewisse Vorhersagbarkeit der vorherigen beiden Komponentenausgaben des implementierten Agenten erzielen, jedoch nicht gänzlich sicherstellen. Deshalb müssen die Ergebnisse einer getrennten Evaluation der Ontologie-Abbildung auf ihren Ursprung und auf mögliche Folgefehler herrührend aus den vorherigen Komponenten untersucht werden.

Für die Evaluation der Ontologie-Abbildung werden zwei Ergebnisse unabhängig voneinander bewertet: Das Finden eines Namens für das neu zu erstellende Methoden-Individuum im Falle einer Lehrsequenz (siehe Abschnitt 9.3.1) sowie die Abbildung auf bestehende Ontologie-Individuen im darauffolgenden Abschnitt. Für die Evaluation der Ontologie-Abbildung werden jeweils 25 Eingabesequenzen zu jedem der vier Szenarien (siehe Abschnitt 6.3) der innerhalb der Vorstudie erhobenen Datenbasis zufällig gezogen. Werden in einer Nutzereingabe Zielsystem-Funktionalitäten angesprochen, welche der Programmschnittstelle völlig fremd sind, wird sie aussortiert. Anschließend wird, wiederum zufällig, ein Ersatz gezogen. Dies betrifft jedoch nicht Synonyme oder semantisch ähnliche Formulierungen von Aktionen, sondern nur gänzlich im Funktionsumfang des Zielsystems nicht enthaltene Befehle.
Anschließend wird für jede dieser 100 Eingabesequenzen eine Musterlösung für die Abbildungssuche erstellt. Dieser Schnitt wird exemplarisch an einer Nutzereingabe in Beispiel 9.3.1 veranschaulicht. Die Musterlösung besteht aus einer Sequenz von Ontologie-Methoden inklusive ihrer benötigten Parameter. Im Falle einer verbalisierten Lehrsequenz wird zusätzlich ein synthetisierter, neuer Methodenname erwartet. Im Beispiel soll dies durch das Schlüsselwort `new` verdeutlicht werden. Mit dem Symbol `::` wird, wie bereits im

Entwurfskapitel in Abschnitt 8.3, der Datentyp des Ontologie-Individuums angegeben.

Beispiel 9.3.1: Musterlösung der Abbildungssuche

„in order to make coffee you have to locate the cups in the cupboard and press the red button on the coffee machine“

```
new makeCoffee():
        locate( locate.what ↔ Cup::IObject,
                locate.where ↔ Cupboard::Object )::Method
        press( press.what ↔ CoffeeMachine.RedButton::Object )::Method
```

Bei der Erstellung der Musterlösungen fiel auf, dass fast alle gesammelten Eingabesequenzen zum vierten Szenario *Set the Table for Two* (siehe Abschnitt A.1) nicht auf die Ontologie abbildbar sind. Der Grund dafür sind nicht vorhandene Ontologie-Methoden, die eine variable Anzahl an Parametern annehmen können. Die Anweisung „*grab the forks knifes and plates*“ spricht beispielsweise das Ontologie-Individuum `grab::Method` an. Dieses fordert jedoch nur ein einzelnes zu greifendes Objekt mit dem Parameter-Individuum `grab.what::Grabable` an und kann somit keine Liste an beliebig vielen Elementen annehmen. Das selbe Problem ergibt sich bei den anderen Ontologie-Methoden. Eine Möglichkeit, dieses Problem zu lösen, wäre es, bei dieser konkreten Formulierung drei statt nur einem Funktionsaufruf der Methode *grab*, jeweils mit einem der aufgezählten Objekte als Parameter zu bilden. Aus einer natürlichsprachlichen Aufzählung von Elementen in der Nutzereingabe kann jedoch nicht immer ein mehrfacher Aufruf der benötigten Methode geschlossen werden. Dies verdeutlicht die Formulierung „*grab the forks knifes are not needed*“. Deshalb stellt die Erkennung von mehrfach nötigen Funktionsaufrufen kein triviales Problem dar und muss gesondert behandelt werden. Innerhalb von Projekt PARSE ist geplant, dies zukünftig mittels des `ActionAnalyzer`-Agenten umzusetzen. Zum Zeitpunkt dieser Arbeit ist dieser jedoch noch nicht abgeschlossen. Aus diesem Grund wird entschieden, das vierte Szenario von der Evaluation der Ontologie-Abbildung auszuschließen.

Zusammenfassend ergeben sich als Grundlage für die Evaluation 75 zufällig gewählte Eingabesequenzen aus den Szenarien eins bis drei. Der Anteil an Lehrsequenzen liegt im Vergleich zu verbalisierten Befehlssequenzen in den betrachteten Nutzereingaben bei etwa 68%. Dies weicht mit fast 10 Prozentpunkten Unterschied von der Verteilung ab, welche bei Erstellung des Goldstandards gemessen wurde (siehe Tabelle 8.3). Außerdem werden insgesamt 297 Funktionsaufrufe über alle drei Szenarien gezählt. Daraus ergibt sich eine durchschnittliche Anzahl von 3.96 diktieren Funktionsaufrufen pro Eingabesequenz. Die Tabelle 9.27 veranschaulicht diese Werte.
Zuletzt ist noch darauf hinzuweisen, dass die diktierten Nutzereingaben keine sinnvollen, korrekt ausführbare Prozeduren beschreiben müssen. In der Teilaufgabe der Ontologie-Abbildung ist es das Ziel, die einzelnen Funktionsaufrufe auf entsprechende, erwartete Elemente der Zielsystem-Ontologie abzubilden. Es wird nicht gefordert, eine ausführbare Sequenz aus Funktionsaufrufen zu erstellen, welche die vom Nutzer gewünschte Problemstellung löst. Die hier erstellten Skripte stellen ein Zwischenergebnis dar. Diese gilt es in einem Nachverarbeitungsschritt des Projekts PARSE auf ihre Sinnhaftigkeit zu prüfen und mit weiteren Informationen wie Kontrollstrukturen anzureichern.

9.3.4.1 Bewertung der Synthese von Methodennamen

Der erste Teil der Ausgabe des Agenten entspricht einem synthetisierten Methodennamen für die neue Methodendefinition. Diese Ausgabe wird nur im Falle einer verbalisierten, und innerhalb der ersten Komponente des Agenten, detektierten Lehrsequenz erstellt. Dieser

Tabelle 9.27: Datensatz für die Evaluation der Ontologie-Abbildung

	Anzahl	Lehrsequenzen	Befehlssequenzen	Funktionsaufrufe
Szenario 1	25	18	7	77
Szenario 2	25	19	6	97
Szenario 3	25	14	10	123
Gesamt	75	52	23	297

Umstand wird auch durch den präsentierten Programmablauf in Abbildung 8.1 veranschaulicht: Im Falle einer Befehlssequenz führt die zweite Kante, ausgehend von der Programmrepräsentation in der zweiten Kartei direkt zum zweiten Block der Abbildungssuche in der letzten Kartei. Der erste Block der Individuen-Erstellung wird für Befehlssequenzen folglich übersprungen. Deshalb wird für die Bewertung der Synthese von Methodennamen ausschließlich die Menge aller Lehrsequenzen unter den Nutzereingaben betrachtet. Für jede dieser Eingaben wird nach dem implementierten Vorgehen aus Abschnitt 9.3.1 ein Methodenname synthetisiert. Aus Tabelle 9.27 geht hervor, dass 52 Lehrsequenzen im Evaluationsdatensatz vorhanden sind. Da diese Nutzereingaben dem Klassifikator bereits durch das Training bekannt sind, kann daraus geschlossen werden, dass die Klassifikationen der erwarteten Musterlösung entsprechen. Dies bestätigt auch die Auswertung der Agentenausgaben, die für alle, bis auf eine einzige Eingabe, mit den erwarteten Werten übereinstimmen. Diese fehlerhafte Eingabe lässt sich jedoch nicht auf ein falsches Klassifikationsergebnis zurückführen. Die Eingabe wird mittels der binären Klassifikation korrekt als Lehrsequenz erkannt. Auch die Mehrklassen-Klassifikation detektiert die Lehrsequenz-Bestandteile erwartungsgemäß. Jedoch kann bei Zusammenführung der Klassifikationsergebnisse mit den semantischen Rollen aus der dritten Stufe der Lehrsequenz-Detektion (siehe Abschnitt 9.2.2) keine Instruktion für die Kommandodeklaration (Definition 5.1.2) gefunden werden. Dieser Fehler ist jedoch auch durch eine Eingabe von bekannten Nutzereingabe nicht beeinflussbar, da sie abhängig von den Annotationen des Semantischen-Rollen-Erkenner-Agenten ist.

Bei einer detektierten Lehrsequenz kann daraufhin die Synthese eines Methodennamens für jede dieser vom Nutzer neu diktierten Funktionalitäten durchgeführt werden. Hierbei gibt es für die Bildung eines passenden Bezeichners nicht nur eine einzige korrekte Lösung. Daher kann die Evaluation der synthetisierten Methodennamen nicht anhand der bisherigen Metriken wie Genauigkeit und Präzision gemessen werden. Zunächst wird dazu in Tabelle 9.28 eine Auswahl der synthetisierten Methodennamen zu den Szenarien 1 „Greet Someone“, 2 „Prepare Coffee“ und 3 „Bring some Beverage“ präsentiert. Zusätzlich präsentiert Tabelle 9.29 weitere Einzelheiten dieser Methodennamen. Die dritte Spalte gibt die Anzahl als unpassend eingestufter Namen pro Szenario an. Dort ist abzulesen, dass für das erste Szenario 4 von 18 synthetisierten Namen als unpassend eingestuft werden. Für Szenario 2 sind jeweils 2 Namen. Die Bewertung des synthetisierten Methodennamens als unpassend wird zum Einen vergeben, falls der gewählte Name als Oberbegriff für die diktierte Methode gänzlich ungeeignet ist. Ein Beispiel dafür ist der synthetisierte Name `askSpeaker` für die Methodendefinition von „Bring some Beverage“. Dieser Fall tritt insgesamt 8 Mal auf. Zum Anderen beinhalten die restlichen, dieser als unpassend eingestuften Bezeichner zusätzliche, angehängte Wörter, die den Methodennamen jedoch nicht weiter präzisieren. Ein Beispiel dafür bilden die Methodennamen `preparePressCoffee` und `prepareCoffeeFriend`.

Weiterhin wird neben der Synthese des Methodennamens auch nach abbildbaren Individuen in der Ontologie gesucht, die zusätzlich zum synthetisierten Methodennamen in Form einer Parameterliste mitgegeben wird. Für das zweite Szenario kann in allen Fällen der Funktionsparameter „coffee“ identifiziert und auf das Ontologie-Individuum `Coffee::`

Tabelle 9.28: Auszug synthetisierter Methodennamen der Ontologie-Evaluation

Szenario	Beispiel
1	greetSomeone, greetSomebody, greetSomeoneProperly, involvePeople, greetFriendFriend, greetSomeoneRobot
2	prepareCoffee, makeCoffee, makeSomeCoffee, prepareCoffeeFriend, preparePressCoffee
3	bringBeverage, bringBeverage, bringBeverageSomeone, askSpeaker, bringBeverageUser, bringBeverageSpeaker

Tabelle 9.29: Synthetisierte Methodennamen der Ontologie-Evaluation

Szenario	Lehrsequenzen	Unpassend	Beispiel	Parameter	Beispiel
1	18	4	`greetFriendFriend`	1	`Robot`
2	19	2	`preparePressCoffee`	19	`Coffee`
3	14	2	`askSpeaker`	13	`Beverage`

`Object` abgebildet werden. Auch im dritten Szenario „Bring some Beverage“ wird in 13 von 14 Fällen das Ontologie-Individuum `Beverage::Object` gefunden. Im ersten Szenario wird der Funktionsparameter „Someone“ zwar identifiziert, die Bezeichner-Ähnlichkeit zum entsprechenden Ontologie-Individuum `Speaker::Object` reicht jedoch nicht aus, um eine konfidente Abbildung zu erzeugen. Deshalb wird nur in einem Fall ein Funktionsparameter der Nutzereingabe auf das (falsche) Individuum `Robot::Object` abgebildet.

Zuletzt werden die Fehlerursachen für unpassend synthetisierte Methodennamen betrachtet. Dazu werden in Beispiel 9.3.2 eine Auswahl dieser Fehler mit ihrem synthetisierten Methodennamen präsentiert. Dazu wird in der jeweilig ersten Zeile die betrachtete Eingabesequenz präsentiert. Die darauffolgenden Zeilen geben die, innerhalb der zweiten Teilaufgabe der Arbeit gebildeten, Funktionsnamen und Funktionsparameter an. Zuletzt wird in der jeweils letzten Zeile der synthetisierte Methodenname angegeben. Die Eingabesequenz E1 in Beispiel 9.3.2 des Szenario „Bring some Beverage“ ist ein Beispiel für einen gänzlich unpassenden Methodennamen.

Beispiel 9.3.2: Synthese von unpassenden Methodennamen

E1: „if i ask you to get me a drink get a glass from (...) then give me the drink“
ask : [to, you, i]
get : [a drink, me]
→ `askSpeaker`

E2: „hi my friend preparing coffee means you have to start (...)“
prepare : [coffee means, hi my friend]
→ `prepareCoffeeFriend`

E3: „(...) start the coffee machine by pressing the button thats how you prepare coffee“
prepare press : [coffee, how, you]
→ `preparePressCoffee`

Dieser Fehler lässt sich darauf zurückführen, dass zwei Instruktionen als potentieller Methodenkopf erkannt werden. Da der Entwurf in Abschnitt 8.3.1 mehrere dieser Methodenkopf-Instruktionen als synonyme Definitionen eines Methodennamens betrachtet, wird in diesem Fall die unpassende beider Instruktionen ausgewählt. Die Eingabesequenz E2 gehört einer weiteren Fehlergruppe an. Hier werden den Methodennamen unpassende, zusätzliche

Wörter angehängt. Diese sind Teil der extrahierten Funktionsparameter, aus welchem die Methodennamen nach dem implementierten Vorgehen in Abschnitt 9.3.1 gebildet werden. Für die Eingabesequenz E2 wird der Methodenparameter „hi my friend“ gefunden. Dieser wird, von bestimmten Wortarten bereinigt, an den synthetisierten Methodennamen angehängt. Die selbe Fehlerursache lässt sich auch im dritten synthetisierten Methodennamen erkennen, wobei hier zwei Verben als mögliche Methodennamen gefunden werden. Als Lösung bietet sich eine zusätzliche Einschränkung der Wortarten für die Erstellung der Funktionsparameter oder eine Sperrliste (engl. *black list*) von Stoppwörtern an.

9.3.4.2 Bewertung der Abbildung auf bestehende Individuen

Für die Bewertung der Abbildung auf bestehende Individuen wird jeder Funktionsaufruf (siehe Tabelle 8.5) einzeln betrachtet und bewertet. Dabei werden zwei Varianten unterschieden. In der ersten Variante wird die implementierte Abbildungssuche auf Korrektheit der gefundenen Verknüpfungen zwischen Programmbestandteilen und Methoden-Individuen untersucht. In der zweiten Variante wird die Abbildungssuche zur Auffindung von vollständigen Methodensignaturen, das heißt korrekten Methoden-Individuen mit richtiger Besetzung aller benötigten Parameter-Individuen, bei Eingabe der extrahierten Programmbestandteile, überprüft.
Die Implementierung der Abbildungssuche aus Abschnitt 9.3.2 gibt eine Rangfolge aus gefundenen Ontologie-Kandidaten zurück. Deshalb wird bewertet, wie häufig sich die gesuchten Ontologie-Elemente an erster Stelle dieser Rangfolge befinden (Top-1-Ergebnisse). Anschließend werden die gefundenen Ontologie-Abbildungen unter Betrachtung der ersten drei Rangfolgeplätze verglichen. Diese Ergebnisse werden als Top-3-Ergebnisse bezeichnet.

Zur Bewertung der Ergebnisse werden die Metriken Präzision, Ausbeute und F_1-Maß berechnet. Diese basieren auf den vier Wahrheitswerten richtig positiv (engl. *true positive*, TP), richtig negativ (engl. *true negative*, TN), falsch positiv (engl. *false positive*, FP) und falsch negativ (engl. *false negative*, FN). Die Wahrheitswerte sowie die Aussagen inklusive der unterliegenden mathematischen Formeln der Metriken werden im Grundlagenkapitel in Abschnitt 2.3 erläutert.
Die Vergabe dieser Wahrheitswerte an unterschiedlich auftretende Ergebnisse der Abbildungssuche geschieht dabei wie folgt: Eine Übereinstimmung des Funktionsaufrufs der Musterlösung mit dem gefundenen Methoden-Individuum beziehungsweise der Signatur wird als richtig positiv gewertet. Wird keine Abbildung gefunden, wird ein falsch negativ klassifiziert. Wird eine zusätzliche Abbildung gefunden, die nicht in der Musterlösung gefordert wird, wird ein falsch positiv gewertet. Zuletzt wird ein falsch abgebildetes, aber gefordertes Methoden-Individuum nicht nur als falsch positiv, sondern auch als falsch negativ klassifiziert. Gleichermaßen wird bei einer *fehlerhaften* Abbildung einer Signatur (siehe Definition 9.3.1) vorgegangen.

Definition 9.3.1: Fehlerhafte Signatur-Abbildungen

Eine Signatur-Abbildung wird als *fehlerhaft* bezeichnet, falls entweder das Methoden-Individuum falsch abgebildet oder mindestens einer der benötigten Parameter-Individuen falsch abgebildet ist oder fehlt.

Im Gegensatz zur Bewertung der Abbildungssuche bei Betrachtung des besten gefundenen Kandidaten müssen für die Bewertung bei Betrachtung der besten drei Kandidaten zusätzliche falsch positive Wahrheitswerte hinzugefügt werden. Denn einerseits steigt die Ausbeute, da die Wahrscheinlichkeit, die korrekte Abbildung unter den drei betrachteten Kandidaten zu finden, steigt. Da es jedoch nur eine korrekte Abbildung geben kann, werden zugleich aber auch neue, fehlerhafte Abbildungen gefunden. Die Präzision fällt auf maximal 1/N, wobei N der Anzahl an Plätzen, das heißt drei, entspricht. Für jeden der

Tabelle 9.30: Wahrheitswerte gefundener Ontologie-Abbildungen

Fall	Werte Top-1	Werte Top-3
Korrektes Methoden-Individuum / Signatur	TP	TP + FP + FP
Keine Abbildung	FN	FN
Zusätzliches Methoden-Individuum / Signatur	FP	FP + FP + FP
Falsches M.-Individuum / fehlerhafte Signatur	FP + FN	FP + FN + FP + FP

genannten Fälle müssen daher bei Betrachtung der Top-3-Ergebnisse zwei falsch positive Wahrheitswerte dazu addiert werden. Wird keine Abbildung gefunden (FN), so bleibt dieser Wert bei Betrachtung der Top-3-Ergebnisse erhalten.
Die Bewertung der Abbildungen wird in Tabelle 9.30 zusammengefasst. Ein konkretes Beispiel dazu folgt in den beiden Absätzen zur Methoden- und Signaturbewertung.

Methoden-Individuen

In diesem Abschnitt wird die Güte der implementierten Abbildungssuche anhand gefundener Abbildungen auf korrekte Methoden-Individuen gemessen. Das nachfolgende Beispiel 9.3.3 soll die Zuordnung der Wahrheitswerte aus der obigen Tabelle 9.30 zu exemplarischen Ausgaben des implementierten Ansatzes verdeutlichen. Es wird eine Eingabesequenz mit zwei verbalisierten Funktionsaufrufen *„locate“* und *„press“* betrachtet. Innerhalb der Abbildungssuche werden zwei Kandidaten für den ersten, sowie drei weitere für den zweiten Funktionsaufruf gefunden. Anschließend werden für diese Evaluation ausschließlich die gefundenen Methoden-Individuen mit der erwarteten Musterlösung verglichen. Die gefundenen Parameter-Individuen werden dabei ignoriert. Evaluiert man die gefundenen Top-1-Ergebnisse, so wird in Beispiel 9.3.3 für den ersten Funktionsaufruf ein richtig positiv (TP) vergeben. Für den zweiten Funktionsaufruf befindet sich das erwartete Methoden-Individuum jedoch an zweiter Stelle der Rangfolge. An erster Stelle steht ein falsches Methoden-Individuum. Deshalb müssen für die Top-1-Ergebnisse die Wahrheitswerte falsch negativ (FN) und falsch positiv (FP) nach Tabelle 9.30 berechnet werden. Werden die Top-3-Ergebnisse betrachtet, so kann für beide Funktionsaufrufe ein richtig positiv (TP) gewertet werden. Wichtig ist, dass diese Ergebnisse trotz der gefundenen, falschen Parameter als korrekt anerkannt werden.

Beispiel 9.3.3: Musterlösung der Abbildungssuche v. Methoden-Individuen

„(...) locate the cups in the cupboard and press the red button on the coffee machine“

function call 1 of 2
top 1: score (1.26) `locate::Method`;
    `locate.what` ↔ `Cup::Object`
    `locate.where` ↔ `Cupboard::Object`
top 2: score (0.81) `locate::Method`;
    `locate.what` ↔ `Cup::Object`
    `locate.where` ↔ `CoffeeMachine::Object`

function call 2 of 2
top 1: score (0.98) `put::Method`;
    `put.what` ↔ `CoffeeMachine::Object`
top 2: score (0.97) `push::Method`;
    `push.what` ↔ `Dishwasher.RedButton::Pushable`
top 3: score (0.64) (...)

function call 1 of 2 → TP (top 1); TP (top 3)
function call 2 of 2 → FN + FP (top 1); TP (top 3)

Tabelle 9.31: Ergebnisse der Abbildungssuche für Methoden-Individuen

	Szenario	Präzision	Ausbeute	F_1	Ausbeute*	F_1*
Top-1	Szenario 1	76.3	58.4	66.2	77.6	76.9
	Szenario 2	78.3	74.2	76.2	85.7	81.8
	Szenario 3	84.7	81.3	83.0	89.3	87.0
	Gesamt	80.7	73.1	76.7	85.4	83.0
Top-3	Szenario 1	29.7	66.7	41.1	88.1	44.4
	Szenario 2	28.7	82.3	42.6	95.2	44.1
	Szenario 3	29.5	84.6	43.8	92.9	44.8
	Gesamt	29.3	79.1	42.8	92.5	44.5

Für die 75 Nutzereingaben des Evaluationsdatensatzes (siehe Tabelle 9.27) ergeben sich die Ergebnisse in Tabelle 9.31. Die Tabelle gliedert die Evaluation in Top-1- und Top-3-Ergebnisse, wobei die Szenarien jeweils getrennt, als auch zusammengefasst in der jeweiligen vierten Zeile präsentiert sind. Es werden die Metriken Präzision, Ausbeute und F_1 betrachtet (siehe Abschnitt 2.3).
Aus der Tabelle lässt sich ablesen, dass bei Betrachtung der Top-1-Ergebnisse, die Abbildungssuche für Methoden-Individuen über alle Eingaben einen zufriedenstellenden Wert für das F_1-Maß von 76.7% erreicht. Die Präzision ist mit 80.7% etwas höher als die Ausbeute mit 73.1%. Bei Untersuchung der Fehlerursachen fällt auf, dass die geringe Ausbeute auf einen wiederkehrenden Fehler zurückzuführen ist. Der in der ersten Teilaufgabe integrierte Semantische-Rollen-Erkenner erkennt fälschlicherweise manche Verben nicht als Aktionen (siehe Abschnitt 8.1.3) an. Dies führt dazu, dass eine komplette Instruktion inklusive ihrer Funktionsparameter (siehe Abschnitt 8.2) in der aufgebauten Datenstruktur aus Teilaufgabe zwei fehlt. Aufgrund dieser fehlenden Funktionsnamen kann in der dritten Teilaufgabe der Ontologie-Abbildung wiederum keine Abbildungssuche (siehe Abschnitt 8.3) durchgeführt werden. Das heißt, die entsprechenden Methoden-Individuen können nicht gefunden werden. Deshalb werden mehrere falsch negative Wahrheitswerte gezählt und die Ausbeute somit verschlechtert. Die Behebung des Fehlers stellt sich als komplex heraus und ist ohne Eingriff in den betroffenen Agenten nicht möglich. Da diese Fehlerursache bei den 297 gezählten Funktionsaufrufen der erstellten Musterlösungen mit insgesamt 43 mal sehr häufig auftritt, werden Ausbeute und F_1-Maß unter Ausschluss dieser Fehlerursache nochmals berechnet. Die Ergebnisse sind in den letzten beiden Spalten der Tabelle unter Ausbeute* und F_1* aufgetragen. Die Präzision wird durch diese Fehlerausschluss nicht verändert. Somit ergibt sich eine verbesserte, gesamte Ausbeute* von 85.4% und ein sehr gutes F_1*-Maß von 83%. Dies bedeutet, dass 85.4% aller, auf dem ersten Platz platzierten Methoden-Individuen den erwarteten Methoden-Individuen entsprechen.
Unter Betrachtung der ersten drei Plätze steigt die Ausbeute dieser gefundenen Methoden-Abbildungen weiter auf sehr gute 92.5% (ohne den ausgegrenzten Fehler des Semantischen-Rollen-Erkenners). Im Gegenzug sinkt die Präzision durch die Hinzuzählung von falsch positiven Wahrheitswerten für jeden weiteren betrachteten Platz der Rangfolge (siehe Tabelle 9.30) auf 29.3%.

Zuletzt werden verschiedene weitere Fehlerursachen betrachtet. Diese werden in Tabelle 9.32 mit einer Auswahl aus exemplarischen Beispieleingaben dargestellt und im Folgenden erläutert. Einerseits ist der namensbasierte Ansatz auf Basis von Bezeichner-Vergleich anfällig für Fehler. In manchen Fällen werden kürzere Wortsequenzen höher bewertet, wie die Tabelle mit Fall eins präsentiert. Dies führt anschließend zu falsch abgebildeten Methoden-Individuen.
Eine weitere, häufigere Ursache sind fehlende Synonyme (siehe Abschnitt 2.1.5). Trotz der

Integration von Synonymen sowie der Ontologie-Spezifikationen von identischen Ontologie-Methoden mittels `SameIndividuals` (siehe Abschnitt 9.3.2.1), werden manche Formulierungen aus den Nutzereingaben nicht abgedeckt. Zusätzlich führt die Beschränkung der eingesetzten Synonyme auf ausschließlich solche mit perfekter Bezeichner-Ähnlichkeit (>0.9 von 1.0) in wenigen Fällen zur Aussortierung eines sinnvollen Synonyms. Diese Fehlerursache ist im zweiten Fall in Tabelle 9.32 dargestellt. Fehlende Synonyme führen dazu, dass manche Funktionsnamen nicht als solche erkannt werden. So ist beispielsweise die Abbildung von „hit the button“ auf das Methoden-Individuum `push::Method` nicht auffindbar. In anderen Fällen führen fehlende Synonyme dazu, dass auf ein falsches Methoden-Individuum der Ontologie abgebildet wird. So auch mit der Beispielanweisung „make eye contact“, welche auf das falsche Individuum `make::Method` abgebildet wird.
Eine andere Ursache für falsche Abbildungen ergeben sich aus Fehlern des Semantischen-Rollen-Erkenners (SRL), die jedoch auf neue Gründe und nicht den oben explizit ausgeschlossenen Fehlerfall zurückzuführen sind. In Fall drei wird beispielsweise die Anweisung „bring the *selected beverage*“ vorgestellt, bei welcher nicht nur der Funktionsaufruf `bring::Method`, sondern auch `select::Method` (in kursiv) als Funktionsaufruf erkannt wird. Dieser zusätzliche, jedoch nicht erwartete Funktionsaufruf wird durch den SRL in der zweiten Teilaufgabe als Aktion markiert. Der Fehler produziert daher einen weiteren falsch negatives Ergebnis. Ein anderer, jedoch ähnlicher Fehler zeigt sich bei der Anweisung „start to wave your hands“. Hier muss aus den Markierungen des Semantischen-Rollen-Erkenners zwischen Hauptverb und modifizierendem Verb unterschieden werden. Dies wird in der ersten Teilaufgabe in Abschnitt 8.3 umgesetzt, führt jedoch manchmal zu Fehlern.
Der vierte Fall repräsentiert die häufigste Fehlerursache, welche sich bei der Abbildungssuche ereignet. In 297 gezählten Funktionsaufrufen der Musterlösung tritt diese Fehlerursache 16 mal auf. Häufig werden in Formulierungen zusätzliche Funktionsaufrufe erkannt, die semantisch betrachtet, keine vom Zielsystem auszuführende Aktion beschreiben. Mit „bring it to the person *who asked you*“ soll das Zielsystem ein Objekt befördern, jedoch wird auch der hintere Teil der Anweisung (in kursiv) als zweiter Funktionsaufruf `ask::Method` erkannt. Auch mit „*make sure* you close it“ soll der Roboter nur eine einzige Aktion, hier das Schließen des Kühlschranks, ausführen. Es wird jedoch sowohl die korrekte Abbildung `close::Method` als auch die falsche Abbildung `make::Method` ausgegeben. Weitere falsch interpretierte Anweisungen sind solche, bei denen Aktionen anderer Subjekte, beispielsweise dem Nutzer oder eines Objekts, beschrieben werden. Dazu präsentiert Tabelle 9.32 zwei weitere Beispiel in Fall vier. Hier werden ebenfalls zusätzliche, das heißt, falsch positive Abbildungen gefunden-
Eine letzte Fehlerursache schildert der fünfte Fall. In manchen Nutzereingaben werden dem Zielsystem zusätzlich zu einem Befehl eine kurze, aus einer Aktion bestehende, Erklärung mitgegeben. Mit der Anweisung „press the button *to start it*“ befiehlt der Nutzer das Drücken eines Knopfes. Dass dadurch eine Maschine gestartet wird, ist eine Nebeninformation. Diese werden durch die Abbildungssuche jedoch als zusätzliche Funktionsaufrufe detektiert. In 297 gezählten Funktionsaufrufen der Musterlösung tritt diese Fehlerursache 15 mal auf. Da diese Kurzerklärungen jedoch synonyme Duplikate von richtig positiven Funktionsaufrufen repräsentieren, wird entschieden, diese als korrekt zu werten, jedoch gesondert für die Evaluation zu markieren.

In Tabelle 9.33 wird eine Statistik über die erstellte Rangliste der Abbildungssuche präsentiert. In der zuvor besprochenen Tabelle Tabelle 9.31 ist zu erkennen, dass die Betrachtung der ersten drei Ranglistenplätze pro Funktionsaufruf die Ausbeute der Abbildungssuche von 85.4% auf 92.5% (unter Ausschluss des spezifischen SRL-Fehlers) steigert. Die Tabelle 9.33 zeigt dazu, dass die korrekte, mit der Musterlösung übereinstimmende, Abbildung sich durchschnittlich über alle Evaluationseingaben auf Platz 1.20 befindet. Die „Ausreisser“ dieser ermittelten Abbildungen sind in seltenen Fällen auf dem zweiten oder dritten

Tabelle 9.32: Fehlerursachen für die Abbildungssuche von Methoden-Individuen

Ursache	Eingabe	Musterlösung	Ergebnis	W.werte
1 Bezeichner	„lower one of your hands“	`lowerLeftHand`	`lowerHead`	FP + FN
2 Synonyme	„get close the table“	`approach`	`get`	FP + FN
	„make eye contact“	`lookAt`	`make`	FP + FN
	„hit the button“	`push`	-	FN
3 SRL	„start to wave your hands“	`wave`	`start`	FP + FN
	„bring the *selected beverage*“	-	`select`	FP
4 Semantik	„bring it to the person *who asked you*“	-	`ask`	FP
	„*make sure* you close it“	-	`make`	FP
	„when I walk into the room“	-	`walk`	FP
	„the machine fills the cup“	-	`fill`	FP
5 Erklärung	„start it by *pressing the red button*“	-	`push`	TP
	„press the button *to start it*“	-	`start`	TP

Tabelle 9.33: Durchschnittlicher Rang der Abbildungssuche für Methoden-Individuen

	Ranglistenplatz
Szenario 1	1.31
Szenario 2	1.16
Szenario 3	1.16
Gesamt	1.20

Platz der Rangliste zu finden. Dies unterstreicht die Korrektheit der in Abschnitt 8.3.2.3 aufgestellten Bewertungsfunktion für Abbildungskandidaten, da sie den erwarteten Kandidaten an erste Stelle stellt.

Vollständige Methodensignaturen

In diesem Abschnitt wird die Güte der implementierten Abbildungssuche anhand der Abbildung auf das korrekte Methoden-Individuum sowie aller benötigten Parameter-Individuen, auch als Methodensignatur bezeichnet, gemessen. Das nachfolgende Beispiel 9.3.4 soll die Zuordnung der Wahrheitswerte aus der obigen Tabelle 9.30 zu exemplarischen Ausgaben des implementierten Ansatzes verdeutlichen. Es wird die selbe Eingabesequenz wie im vorherigen Beispiel 9.3.3 zur Abbildung von Methoden-Individuen gewählt, jedoch auf die Unterschiede in der Bewertung eingegangen. Für diese Evaluation müssen neben den Methoden-Individuen auch die gefundenen Parameter-Individuen mit der erwarteten Musterlösung verglichen werden. Evaluiert man die gefundenen Top-1-Ergebnisse, so ist in Beispiel 9.3.4 der erste Funktionsaufruf richtig positiv. Das Methoden-Individuum als auch beide Parameter-Individuen des Funktionsaufrufs `locate::Method` stimmen mit den erwarteten Individuen der Ontologie überein. Für den zweiten Funktionsaufruf trifft dies nicht zu. Zwar befindet sich die korrekte Abbildung des Methoden-Individuums an zweiter Stelle, jedoch ist der abgebildete Parameter falsch. Deshalb muss dies als *fehlerhafte* Methodensignatur (siehe Definition 9.3.1) als falsch negativ und falsch positiv gewertet werden. Selbiges gilt für die Bewertung der Top-3-Ergebnisse.

Tabelle 9.34: Ergebnisse der Abbildungssuche für vollständige Methodensignaturen

		Präzision	Ausbeute[7]	F_1	Ausbeute*	F_1*
Top-1	Szenario 1	58.3	46.1	51.5	61.4	59.8
	Szenario 2	67.4	62.0	64.6	71.3	69.3
	Szenario 3	67.2	64.5	65.8	70.8	69.0
	Gesamt	65.3	59.0	62.0	68.9	67.0
Top-3	Szenario 1	25.9	54.4	35.1	71.7	38.1
	Szenario 2	25.3	70.7	37.2	81.4	38.6
	Szenario 3	23.9	67.7	35.4	74.3	36.2
	Gesamt	24.5	65.2	35.7	76.6	37.1

Beispiel 9.3.4: Musterlösung der Abbildungssuche von M.signaturen

„(...) locate the cups in the cupboard and press the red button on the coffee machine“

function call 1 of 2
top 1: score (1.26) `locate::Method`;
`locate.what` ↔ `Cup::Object`
`locate.where` ↔ `Cupboard::Object`
top 2: score (0.81) (...)

function call 2 of 2
top 1: score (0.98) `put::Method`;
`put.what` ↔ `CoffeeMachine::Object`
top 2: score (0.97) `push::Method`;
`push.what` ↔ `Dishwasher.RedButton::Pushable`

function call 1 of 2 → TP (top 1); TP (top 3)
function call 2 of 2 → FN + FP (top 1); FN + FP (top 3)

Für die 75 Nutzereingaben des zuvor generierten Evaluationsdatensatzes (siehe Tabelle 9.27) ergeben sich die Ergebnisse in Tabelle 9.34. Die Tabelle gliedert sich wie zuvor in Top-1- und Top-3-Ergebnisse; auch die selben Metriken Präzision, Ausbeute und F_1 werden betrachtet (siehe Abschnitt 2.3).
Aus der Tabelle lässt sich ablesen, das bei Betrachtung der Top-1-Ergebnisse, die Abbildungssuche für Methodensignaturen über alle Eingaben ein F_1-Maß von 62% Güte erreicht. Die Präzision ist mit 65.3% etwas höher als die Ausbeute mit 59.0%. Wiederum lässt sich die geringe Ausbeute auf den spezifischen Erkennungsfehler des Semantischen-Rollen-Erkenners aus dem vorherigen Paragraphen zurückführen. Deshalb werden Ausbeute und F_1-Maß unter Ausschluss dieser Fehlerursache nochmals berechnet. Somit ergeben sich eine verbesserte Ausbeute* von 68.9% und F_1* mit 67%. Die Präzision wird durch diese Fehlerausschluss nicht verändert. Dies bedeutet, dass 68.9% aller, auf dem ersten Platz platzierten Methoden-Individuen inklusive ihren Parameter-Individuen den erwarteten Methodensignaturen entsprechen. Unter Betrachtung der ersten drei Plätze steigt die Ausbeute* dieser gefundenen Methoden-Abbildungen weiter auf 76.6%. Im Gegenzug sinkt die Präzision nach der Berechnung in Tabelle 9.30 wiederum stark auf 24.5%.

Zuletzt werden die Fehlerursachen betrachtet. Diese werden in Tabelle 9.35 mit einer Auswahl aus exemplarischen Beispieleingaben dargestellt und im Folgenden erläutert. Das Symbol `?` steht hierbei für ein fehlendes, nicht abgebildetes Ontologie-Individuum. Die fünf Fehlerfälle, welche bei der Bewertung der einzelnen Methoden-Individuen aus Tabel-

le 9.32 im vorherigen Paragraphen aufgetreten sind, sind auch bei Betrachtung vollständiger Methodensignaturen wiederzufinden. Es wird daher nur auf Ergänzungen zu diesen Fehlerursachen eingegangen. Außerdem wird eine zusätzliche Spalte eingefügt, welche die extrahierten Funktionsparameter aus Basis des Semantischen-Rollen-Erkenners in der dritten Stufe der ersten Teilaufgabe (siehe Abschnitt 8.1.3) angibt. Alle vorgestellten Fehler führen bei Betrachtung vollständiger Methodensignaturen zu einer Bewertung mit den beiden Wahrheitswerten falsch positiv und falsch negativ.
Der erste Fall zu falsch gefundenen Bezeichnern erweitert sich auf die Parameter-Individuen der Ontologie. Auch hier werden kürzere Bezeichner höher bewertet. Dies führt dazu, dass beispielsweise das Parameter-Individuum `Dishes::Object` häufig höher in der Rangfolge als der erwartete Funktionsparameter `Dispenser::Object` eingeordnet wird. Somit erhält die eigentlich korrekte Methodensignatur eine geringere Bewertung nach der Bewertungsfunktion aus Abschnitt 8.3.2.3 als die, mit ähnlicher-bewertetem, falschen Funktionsparameter. Die fehlerhafte Methodensignatur (siehe Definition 9.3.1) rückt an erste Stelle und produziert für die Top-1-Ergebnisse einen falsch positiven und falsch negativen Wahrheitswert.
Wie auch schon bei der Evaluation der Methoden-Abbildung im vorherigen Paragraph, sind manche Fehler auf Synonyme (siehe Abschnitt 2.1.5) zurückzuführen. Die Fehlerursache erweitert sich bei zusätzlicher Betrachtung der Parameter-Individuen, falls durch den Nutzer unbekannte Synonyme verwendet oder durch WordNet falsche Synonyme in Betracht gezogen werden. Die Folge sind falsch abgebildete Parameter-Individuen.
Die dritte Fehlerursache ist auf fehlende oder falsch extrahierte Aktionsentitäten zurückzuführen, die sich aus der semantischen Rollen-Klassifikation des SRL-Agenten ergeben. In vielen Fällen wird ein in Beziehung stehender Parameter nicht mit einer semantischen Rolle gekennzeichnet. Das Fehlen von semantische Rollen führt wiederum dazu, dass die entsprechenden Funktionsparameter nicht extrahiert werden und somit nicht für die Abbildungssuche als Eingabe dienen können. Dies wird in der dritten Spalte der Tabelle dadurch visualisiert, dass das Element in der Parameterliste fehlt. Die Folge sind fehlende Parameter-Individuen und somit eine unvollständige Methodensignatur. Ein Grund für die Fehler des Semantischen-Rollen-Erkenners könnte auf Grammatikfehler zurückzuführen sein, wie es in der Beispielanweisung „take mug then (...)“ der Fall ist. Dazu präsentiert Tabelle 9.35 zwei weitere Beispiel in Fall drei.
Auch die Menge der Semantikfehler in Fall vier erweitert sich bei zusätzlicher Evaluation der Parameter-Individuen. Die Anweisung „put the cup down“ benötigt Wissen über den Kontext der Unterhaltung, um darauf zu schließen, dass der Becher auf den Tisch gestellt werden soll (`put(Cup,Table)`). Da diese Information nicht mitgeliefert wird, wird keine vollständige Methodensignatur gebildet.
Eine zusätzliche, neue Fehlerursache zeigt der sechste Fall. In natürlicher Sprache werden statt einer expliziten Erwähnung des Subjektes oder Objektes sehr häufig Personal- oder Possessivpronomen verwendet. Dies verdeutlicht die Anweisung „grab a cup and put *it* under the dispenser“. Die Lösung ist der Einsatz von Korreferenzen (siehe Abschnitt 2.2.1.6), die der integrierte `ContextAnalyzer`-Agent aufzulösen versucht. Jedoch findet dieser nicht immer die korrekte Korreferenz. Diese führt zu falschen Parameter-Abbildungen und somit zu falsch positiv und falsch negativen Ergebnissen. Zusätzlich dazu referiert der Nutzer die Objekte der Umgebung häufig unterspezifiziert. Mit „close the fridge door (...) open the door“ kann nur mit Hilfe des Kontext auf die korrekte Tür abgebildet werden. Auch dafür wird der `ContextAnalyzer`-Agent eingesetzt. Werden jedoch keine solche Objektreferenzen gefunden, wird der Name allein (hier „door“) in die namensbasierte Abbildungssuche eingegeben. In diesem Fall werden dabei eine Menge von Ontologie-Individuen gefunden, die Türen repräsentieren. Wie bereits in Fehler-Fall eins erläutert, wird der ähnlichste, in diesem Fall der kürzeste Bezeichner gewählt. So kommt es zu einer fehlerhaften Methodensignatur.

Zuletzt sind manche Abbildungsfehler darauf zurückzuführen, dass der Nutzer benötigte, erwartete Informationen nicht explizit angibt, sie vom korrekten Methoden-Individuum der Ontologie jedoch gefordert werden. Dies wurde bereits in den Herausforderungen dieser Arbeit in Abschnitt 7.1 als Herausforderung (H 3.2) definiert. Dies ist beispielsweise für die Anweisung „pour the coffee“ der Fall. Das entsprechende Methoden-Individuum benötigt zwei Parameter. Da nur ein Parameter zu extrahieren ist, führt dies zu einem fehlenden Funktionsparameter und damit zu einer unvollständigen Methodensignatur. Zuletzt ist wichtig, dass die genannten Fehlerursachen zu den bereits genannten Ursachen bei getrennter Betrachtung der Methoden-Individuen zusätzlich hinzukommen. Deshalb fallen die Evaluationsergebnisse dieser zweiten Evaluationsvariante auch etwas schlechter aus als die erstere.

Tabelle 9.35: Fehlerursachen für die Abbildungssuche von Methodensignaturen

Ursache	Eingabe	Parameter	Musterlösung	Ergebnis
1 Bezeichner	„open the door“	[the door]	`open(Cupboard.Door)`	`open(Fridge.Door)`
	„put it under the dispenser“	[the dispenser, it]	`put(Cup,Dispenser)`	`put(Cup,Dishes)`
2 Synonyme	„open the refrigerator“	[the refrigerator]	`open(Fridge.Door)`	`open(?)`
3 SRL	„look him in the eye“	[him]	`lookAt(Eye)`	`lookAt(?)`
	„put a glass on the counter“	[a glass on the counter]	`put(Glass,Counter)`	`put(Glass,?)`
	„take mug then“	[then]	`take(CoffeeMug)`	`take(?)`
4 Semantik	„put the cup down“	[the cup, down]	`put(Cup,Table)`	`put(Cup,?)`
6 Korref.	„grab it“	[it]	`grab(Cup)`	`grab(Kitchen)`
	„give it to the user“	[it, to the user]	`give(Glass,User)`	`give(Table,User)`
7 Benutzung	„pour the coffee“	[the coffee]	`pour(Coffee,Cup)`	`pour(Coffee,?)`

Tabelle 9.36: Durchschnittlicher Rang der Abbildungssuche für vollständige Methodensignaturen

	Ranglistenplatz
Szenario 1	1.47
Szenario 2	1.19
Szenario 3	1.20
Gesamt	1.22

Außerdem wird wiederum eine Statistik über die erstellte Rangliste der Abbildungssuche aufgestellt. In der zuvor präsentierten Tabelle Tabelle 9.34 ist zu erkennen, dass die Betrachtung der ersten drei Ranglistenplätze pro Funktionsaufruf die Ausbeute der Abbildungssuche von 68.9% auf 76.6% (unter Ausschluss des spezifischen SRL-Fehlers) steigert. Aus Tabelle 9.36 lässt sich ablesen, dass die korrekte Abbildung über alle Evaluationseingaben durchschnittlich auf Platz 1.22 steht. Es fällt jedoch auf, dass in manchen Fällen die Methoden-Individuen mit weniger Parametern den Vorrang auf Platz eins haben. So werden für die Anweisung „take the glass from the cupboard“ beide der Abbildungen `take(Glass)` und `take(Glass,Cupboard)` als korrekt angesehen. Wertvoller für das Skript wäre die letztere, diese ist jedoch (erst) auf Platz zwei der Rangliste aufzufinden. Dieser Umstand könnte jedoch mit einer Modifikation der Bewertungsfunktion nachträglich gelöst werden.

Zusammenfassend lässt sich sagen, dass das erzielte F_1*-Maß von 67% für eine vollständige Methodensignatur (siehe Tabelle 9.34) an erster Stelle der Abbildungsrangliste, bestehend aus der korrekten Abbildung des Methoden-Individuums sowie der korrekten Abbildung jedes einzelnen Parameter-Individuums, eine zufriedenstellenden, guten Wert darstellt. Weiterhin steigert sich die Ausbeute bei Betrachtung der ersten drei Ranglistenplätze von 68.9% auf gute 76.6%. Potential für Verbesserungen lässt sich an den präsentierten Fehlerursachen erkennen. Manche dieser Fehler, wie beispielsweise die Auflösung von Korreferenzen oder ein Verständnis von Kontext und semantischen Zusammenhängen, stellen jedoch komplexe Probleme dar, die in weiteren Arbeiten des Projekts PARSE behandelt werden könnten.

10 Evaluation

In diesem Kapitel soll das implementierte Werkzeug zur Synthese von Methodendefinitionen als Ganzes evaluiert werden. Eine getrennte Evaluation einzelner Komponenten dieses Werkzeuges wurde bereits in vorherigen Kapiteln durchgeführt. Die Abschnitte 9.1.1.7 und 9.1.2.5 präsentieren dazu eine Evaluation der ersten Teilaufgabe zur Detektion von Lehrsequenzen. Weiterhin wird in Abschnitt 9.3.4 die Ontologie-Abbildung der dritten Teilaufgabe eigenständig evaluiert. Zusätzlich dazu führt dieses Kapitel eine Ende-zu-Ende-Evaluation des implementierten Agenten durch. Ziel ist es, die Ausgaben des entwickelten Agenten bei Eingabe von neuen, natürlichsprachlichen Äußerungen zu testen.
Dazu wird im nachfolgenden Abschnitt die Erhebung einer neuen Datengrundlage für die Evaluation beschrieben. Anschließend werden die Ergebnisse der Ende-zu-Ende-Evaluation vorgestellt, wobei eine Auswahl an Beispieleingaben und die zugehörige Ausgabe des Agenten präsentiert werden. Im darauffolgenden Abschnitt wird auf bestehende Fehler des implementierten Agenten und deren Ursachen eingegangen. Zuletzt werden die Evaluationsergebnisse diskutiert und ermittelte Einschränkungen des Werkzeuges angesprochen.

10.1 Datengrundlage

In der Vorstudie wurde das bestehende PARSE-Korpus bereits um neue, auf die zu behandelnde Problemstellung angepasste, Nutzereingaben erweitert. Um jedoch eine unbeeinflusste Evaluationsgrundlage zu erhalten, müssen für die Ende-zu-Ende-Evaluation erneut neue Eingabesequenzen erhoben werden. Anstatt jedoch die Szenarien aus der Vorstudie (siehe Abschnitt 6.3) für die Generierung neuer Eingabesequenzen wiederzuverwenden, soll die Datengrundlage für die Ende-zu-Ende-Evaluation auf Basis neuer Szenarientexte durchgeführt werden. Dies hat den Vorteil, dass der implementierte Agent nicht nur mit neuen, unbekannten Nutzereingaben getestet wird, zusätzlich dazu sollen die in den Nutzereingaben diktierte Aufgabenstellung an das Zielsystem ebenfalls neu konzipiert werden. Diese neuen Szenarien sollen bisher nicht in den Nutzereingaben behandelte Funktionalitäten des Zielsystems und neue Objekte der Umgebung ansprechen. So kann mit dieser Evaluation nicht nur gezeigt werden, dass der entwickelte Agent mit unbekannten Nutzereingaben umgehen kann. Vielmehr kann bei zufriedenstellenden Evaluationsergebnissen belegt werden, dass der Agent unabhängig von den ihn bekannten Aufgabenstellungen der Szenarien gute Ergebnisse erzielt. Deshalb werden, analog zu den Kriterien im Kapitel der Vorstudie in Abschnitt 6.3, die zwei neuen Szenarien „Start the Dishwasher“ und „Prepare Cereals“ entworfen. Diese Szenarien weisen die selbe Struktur der bisherigen Szenarientexte auf und sollen beim Probanden das Verbalisieren von Lehrse-

quenzen hervorrufen. Für die Unterscheidung zu den vier Szenarien der Vorstudie wurde die dortige Nummerierung fortgeführt. Deshalb erhalten die beiden neuen Szenarien der Ende-zu-Ende-Evaluation die Nummerierung 5 und 6. Eine vollständige Auflistung sowie der Inhalt der Szenarien wird im Anhang unter Abschnitt A.1 aufgeführt.
Für die Generierung neuer Daten für diese Evaluation wäre es eine Möglichkeit, das bestehende Korpus um neue Aufnahmen in gesprochener Form zu erweitern. Auf diese Weise wäre ein realitätsnähere Aufnahmeumgebung als in der Vorstudie gewährleistet, da die Anweisungen in Sprache diktiert werden müssten. Überdies müssen diese Sprachaufnahmen lokal vor Ort getätigt werden. Durch die Notwendigkeit einer lokalen Aufnahme können, wegen des hohen Aufwandes für Proband und Studienersteller, jedoch nur wenige Sprachaufnahmen erhoben werden. Als zweite Möglichkeit könnten, wie bereits in der Vorstudie, textuelle Nutzereingaben gesammelt werden. Dies stellt durch die textuelle Form der Eingabe ein weniger realitätsnahe Testumgebung dar. Ein Vorteil dieser Variante ist jedoch, dass einerseits durch die Nutzung einer internationalen Studienplattform eine große Menge an Eingabetexten schnell und mit weniger Aufwand verbunden erhoben werden kann. Überdies ist der Anteil an englischen Muttersprachlern unter den Probanden sehr wahrscheinlich höher als bei einer lokalen Studie. Eine ausführlichere Diskussion weiterer Vor- und Nachteile beider Varianten wurde bereits in Abschnitt 6.2 für die Vorstudie durchgeführt. Die Auswertung des in der Vorstudie erhobenen Datensatzes in Abschnitt 6.5.3 zeigte zudem, dass die erfassten Äußerungen der Probanden natürlichsprachlich formuliert sind und sich von Aufnahmen in imperativer, gesprochener Form nicht signifikant unterscheiden. Deshalb wird für die Aufnahme einer neuen Datengrundlage für die Ende-zu-Ende-Evaluation die zweite Variante mittels der *Proflic.ac* (siehe Abschnitt 2.5) gewählt.

Für die Ende-zu-Ende-Evaluation werden 110 Teilnehmer für die Studie gewonnen. Aus den gewonnenen 110 Einreichungen (siehe Definition 6.4.1) werden neun ungültige Einreichungen aussortiert, die keine oder nur ungenügende Eingaben aufweisen. Schließlich ergibt sich der Datensatz in Tabelle 10.1 mit 101 Eingabesequenzen je Szenario.
Für die Ende-zu-Ende-Evaluation wird die Größe der Evaluationsgrundlage auf 50 Eingabesequenzen je Szenario festgelegt. Daraufhin werden jeweils 50 Nutzereingaben zu jedem der beiden Szenarien aus der erhobenen Datenbasis zufällig gezogen. Werden in einer Nutzereingabe jedoch Zielsystem-Funktionalitäten angesprochen, welche der Programmschnittstelle völlig fremd sind, wird diese aussortiert. Dies betrifft jedoch nicht synonyme oder semantisch ähnliche Formulierungen von Aktionen, sondern nur gänzlich im Funktionsumfang des Zielsystems nicht enthaltene Befehle. Anschließend wird, wiederum zufällig, ein Ersatz für die aussortieren Nutzereingaben gezogen, bis die Evaluationsgrundlage die festgelegten 50 Eingaben je Szenario enthält. Zuletzt werden die Nutzereingaben, wie auch schon in der Datenvorverarbeitung der Vorstudie in Abschnitt 6.5.2 von Schreibfehlern, jedoch nicht von Grammatikfehlern bereinigt, um die Vergleichbarkeit mit gesprochenen Nutzereingaben zeigen zu können. Die so generierte Evaluationsgrundlage wird in Tabelle 10.2 dargestellt.
Die Tabelle lässt erkennen, dass der Anteil an verbalisierter Lehrsequenzen in diesem Datensatz mit 78% größer ausfällt als im Datensatz der Vorstudie mit 59% (siehe Tabelle 8.3). Eine Befehlssequenz (siehe Definition 6.5.1) wird im Programmkontext als Skript aus Funktionsaufrufen (siehe Abschnitt 8.3.2) interpretiert. Dieses Skript wird bei der Programmrepräsentation einer Lehrsequenz (siehe Definition 5.1.1) ebenfalls erstellt, um den Methodenrumpf einer neuen Methodendefinition zu modellieren. Zusätzlich dazu muss im Falle einer Lehrsequenz ein Methodenkopf synthetisiert werden (siehe Abschnitt 8.3.1). Deshalb kann bei Betrachtung der Programmrepräsentation die Abbildung einer Befehlssequenz mit der Abbildung des Methodenrumpfes der Lehrsequenzen verglichen werden, weshalb wiederum eine ungleiche Verteilung von Lehr- und Befehlssequenzen im Evalua-

Tabelle 10.1: Erhobener Datensatz für die Ende-zu-Ende-Evaluation

	Anzahl
Teilnehmer (Einreichungen)	110
Valide Einreichungen	101
Eingaben	202

Tabelle 10.2: Generierte Evaluationsgrundlage der Ende-zu-Ende-Evaluation

	Anzahl	Lehrsequenzen	Befehlssequenzen	Funktionsaufrufe
Szenario 5	50	44	6	158
Szenario 6	50	34	16	315
Gesamt	100	78	22	473

tionsdatensatz die Aussagekraft dieser Evaluation nicht beeinträchtigt. Weiterhin wird in Tabelle 10.2 die Anzahl diktierter Aktionen, hier im Programmkontext bereits als Funktionsaufrufe (siehe Tabelle 8.5) bezeichnet, angegeben. Es fällt auf, dass die Probanden für die Erklärung der in den Szenarien geforderten Aufgabe insgesamt 473 Funktionsaufrufe, das heißt durchschnittlich 4.73 Instruktionen pro Eingabe, an das Zielsystem diktieren. Im Vergleich zu den Szenarientexte der Vorstudie enthalten die beiden neuen Szenarien im Durchschnitt 1.76 mehr Instruktionen pro Nutzereingabe (siehe Tabelle 9.27). Dabei fällt auf, dass im sechsten Szenario mit durchschnittlich 6.30 Funktionsaufrufen mehr als doppelt so viele Instruktionen an das Zielsystem gerichtet werden, als in Szenario 5 mit durchschnittlich 3.16 Funktionsaufrufen pro Eingabesequenz. Das lässt darauf schließen, dass diesem Szenario eine komplexere Aufgabenstellung zugrundeliegt, die der Nutzer dem Zielsystem beibringen muss.

10.2 Ergebnisse

Für die Bewertung der Güte des implementierten Werkzeuges zur Synthese von Methodendefinitionen muss für jede Eingabesequenz eine Musterlösung für die erwartete Ausgabe des Agenten händisch erstellt werden. Diese Ausgabe entspricht den abgebildeten Ontologie-Individuen aus der letzten Teilaufgabe der Arbeit (siehe Abschnitt 8.3). Die Ende-zu-Ende-Evaluation wird analog zur Evaluation der Ontologie-Abbildung in Abschnitt 9.3.4 durchgeführt, mit dem Unterschied, dass statt den bekannten Nutzereingaben der Vorstudie, die für das Training der Klassifikatoren der ersten Teilaufgabe eingesetzt wurden, der neu erhobene Evaluationsdatensatz verwendet wird. Es wird ebenfalls die gleiche, dort verwendete Domänenontologie aus Tabelle 9.26 herangezogen.
Für die Evaluation werden zwei Ergebnisse unabhängig voneinander bewertet: Das Finden eines Namens für die neu zu erstellende Methodendefinition im Falle einer Lehrsequenz (siehe Abschnitt 9.3.1) sowie die Abbildung der verbalisierten Funktionsaufrufe auf bestehende Ontologie-Individuen im darauffolgenden Abschnitt. Zur Bewertung der Ergebnisse werden die Metriken Präzision, Ausbeute und F_1-Maß berechnet. Diese basieren auf den vier Wahrheitswerten richtig positiv (engl. *true positive*, TP), richtig negativ (engl. *true negative*, TN), falsch positiv (engl. *false positive*, FP) und falsch negativ (engl. *false negative*, FN). Die Wahrheitswerte sowie die eingesetzten Metriken werden im Grundlagenkapitel in Abschnitt 2.3 erläutert.

10.2.1 Synthese von Methodennamen

Der erste Teil der Ausgabe des Agenten entspricht einem synthetisierten Methodennamen für die neue Methodendefinition der Nutzereingabe. Diese Ausgabe wird nur im Falle einer verbalisierten Lehrsequenz erstellt. Deshalb sind die Ausgaben dieses Schritts sehr

abhängig von der Güte der Klassifikationen, die in der ersten Komponente des Agenten durchgeführt werden. Der im Entwurfskapitel in Abbildung 8.1 visualisierte Programmablauf verdeutlicht diese Beziehungen. Die drei Teilaufgaben dieser Arbeit, repräsentiert durch die drei Karteien dieser Grafik bilden eine aufeinander aufbauende Kette von Komponenten. Die Synthese von Methodennamen wurde auch bereits in der isolierten Evaluation der Ontologie-Abbildung in Abschnitt 9.3.4.1 durchgeführt. Um für die separate Evaluation die Beeinflussung der Ergebnisse durch vorherige Komponenten zu verhindern, wurde ein Ausschnitt der Datengrundlage aus der Vorstudie wiederverwendet. Diese Nutzereingaben sind den Klassifikatoren der ersten Komponente bekannt und somit entspricht deren Klassifikationsergebnis der erwarteten Musterlösung. Im Unterschied dazu werden für die Ende-zu-Ende-Evaluation keine dem Klassifikator bekannten Daten verwendet. Es wird die generierte Evaluationsgrundlage aus dem vorherigen Abschnitt 10.1 eingesetzt. Diese enthält dabei nicht nur neue Nutzereingaben zu den bereits vorhandenen, bekannten Szenarien der Vorstudie, sondern neue Nutzereingaben zu zwei gänzlich neu konzipierten Szenarien für das betrachtete Zielsystem. Im ersten Teil dieses Abschnitts wird daher auf die Ergebnisse der ersten beiden Komponenten dieses Agenten eingegangen. Im Anschluss darauf folgt die Auswertung der synthetisierten Methodennamen.

Die nachfolgende Tabelle 10.3 präsentiert die Ergebnisse der Lehrsequenz-Erkennung aus der ersten Teilaufgabe dieser Arbeit. Diese Ergebnisse sind der binären Klassifikation der ersten Stufe der Klassifikationshierarchie aus Abschnitt 8.1.1 zuzuordnen, wurden aber nach der Implementierung in Abschnitt 9.3.1 um eine zusätzliche Überprüfung der klassifizierten Lehrsequenz-Bestandteile der zweiten Stufe (siehe Tabelle 7.1) erweitert. Falls demnach in der ersten, binären Stufe eine Befehlssequenz detektiert wird, aber der Klassifikator der zweiten Stufe dennoch Lehrsequenz-Bestandteile in der Nutzereingabe findet, so wird das Ergebnis als Lehrsequenz interpretiert. Aus diesem Vorgehen ergibt sich für den Evaluationsdatensatz bestehend aus 100 Nutzereingaben eine Genauigkeit von 85% hinsichtlich der korrekten Klassifikation einer Eingabe zur Klasse der Lehrsequenzen oder der Befehlssequenzen. Weiterhin lässt die Tabelle 10.3 erkennen, dass die Präzision mit 93.2% höher ausfällt als die Ausbeute mit 87.2%. Dies bedeutet dass im Verhältnis zu den korrekten Klassifikationen weniger falsch positive (FP) Klassifikationen gefunden werden, als falsch negative (FN). Daraus folgt wiederum, dass die erste Komponente des Agenten weniger Fehler dabei macht, Nutzereingaben fälschlicherweise Lehrsequenz zu klassifizieren, als die Intention des Nutzerkommandos hinsichtlich einer Funktionserweiterung des Zielsystems nicht zu erkennen. Der Agent erzielt für die erste Komponente einen F_1-Wert von 90.1%.
Überdies können die Klassifikationsergebnisse aus der zweiten Zeile der präsentierten Tabelle 10.3 direkt mit den gemessenen Testergebnissen des trainierten binären Klassifikators der ersten Teilaufgabe aus Abschnitt 9.1.1.7 verglichen werden. Dort wurde mittels der „szenarienbezogenen Datensatzaufteilung“ eines der Szenarien vom Training zurückgehalten, sodass anschließend die Klassifikatorgüte auf einem unbekannten Testdatensatz evaluiert werden konnte. Dieses Vorgehen gleicht der Vorgehensweise dieser Ende-zu-Ende-Evaluation, da der Agent ebenfalls auf unbekannten Szenarientexten getestet wird. Bei der isolierten Evaluation der binären Klassifikatoren aus Abschnitt 9.1.1.7 erreicht das in den Agenten integrierte Modell eine Genauigkeit von 91.9% (siehe Tabelle 9.16). Eine weitere Verbesserung mit 97.7% konnte mit dem angepassten BERT-Modell von Google erzielt werden, das zum Zeitpunkt dieser Arbeit jedoch noch nicht in die Java-Umgebung dieser Anwendung exportiert werden konnte (siehe Abschnitt 9.2.1). Vergleicht man diese erzielten Genauigkeiten mit den binären Klassifikatorausgaben des Agenten bei der Ende-zu-Ende-Evaluation, so zeigt sich eine Verschlechterung von knapp 10 Prozentpunkten auf 82%. Dies könnte darauf zurückzuführen sein, dass die neuen Szenarientexte komplexere Aufgabenstellungen behandeln als jede der Vorstudie. Diese Vermutung wird dadurch un-

Tabelle 10.3: Ende-zu-Ende-Ergebnisse der Detektion von Lehrsequenzen

Variante	Genauigkeit	Präzision	Ausbeute	F_1
Binäre + Mehrkl.-Klf.	85.0	93.2	87.2	90.1
Binäre Klf.	82.0	98.4	77.9	87.0

Tabelle 10.4: Auszug synthetisierter Methodennamen der Ende-zu-Ende-Evaluation

Szenario	Beispiel
5	startDishwasher, startDishwasherHello, startDishwasherHelloFirst, turnOnDishwasher, useDishwasher, pushTheBlueButton
6	prepareCerealFirstThing, prepareSomeCereal, prepareABowlOfCereal, makeCereal, makeCereals, makeSomeCereals, makeBowlOfCereal, prepareCereals

terstrichen, dass die Anzahl diktierter Instruktionen je Nutzereingabe von durchschnittlich 2.97 (siehe Tabelle 9.27) auf durchschnittlich 4.73 Instruktionen pro Eingabesequenz in den beiden neuen Szenarien (siehe Tabelle 10.2) steigt.
Dennoch lässt sich erkennen, dass die Kombination der beiden Klassifikatorenergebnisse der ersten Teilaufgabe hinsichtlich der binären Entscheidung über die Detektion von Lehrsequenzen eine zusätzliche Verbesserung des Ergebnisses bewirkt. Dies entspricht der Lösungsstrategie für mögliche Fehlklassifikationen aus Abschnitt 9.2.1 und wurde in der Herausforderung (H 2.4) aus Abschnitt 7.1 definiert. Die Genauigkeit kann durch die Ergebniskombination von 82 auf 85% gesteigert werden. Auch der F_1-Wert erhöht sich von 87 auf 90.1%.

Konnte der Agent eine Lehrsequenz in der Nutzereingabe detektieren, so folgt im zweiten Schritt die Synthese eines Methodennamens für jede dieser vom Nutzer neu diktierten Funktionalitäten. Hierbei gibt es für die Bildung eines passenden Bezeichners nicht nur eine einzige korrekte Lösung (siehe Herausforderung (H 3.1) in Abschnitt 7.1). Daher wird die Evaluation der synthetisierten Methodennamen, wie bereits in der Ontologie-Abbildungsevaluation in Abschnitt 9.3.4.1, nicht anhand von Metriken gemessen. Stattdessen wird in Tabelle 10.4 eine Auswahl an synthetisierten Methodennamen zu den Szenarien 5 „Start the Dishwasher“ und 6 „Prepare some Cereals“ präsentiert. In fünf beziehungsweise zwei Fällen kann der synthetisierte Name als unpassend eingestuft werden. Dies wird in der dritten Spalte von Tabelle 10.5 angegeben. Dies ist zum Einen der Fall, falls der gewählte Name als Oberbegriff für die diktierte Methode gänzlich ungeeignet ist. Ein Beispiel dafür ist der synthetisierte Name `pushTheBlueButton` für die Methodendefinition von „Start the Dishwasher“. Dieser Fall tritt nur zwei Mal auf. Zum Anderen beinhalten die restlichen fünf Methodennamen zusätzliche, angehängte Wörter, die den Methodennamen jedoch nicht weiter präzisieren. Ein Beispiel dafür bilden die Methodennamen `startDishwasherHello` und `prepareCerealFirstThing`. Diese Suffixe könnte durch einen Filter von Wortarten oder eine Sperrliste (engl. *black list*) von Wörtern entfernt werden.
Neben der Synthese des Methodennamens wird auch nach zusätzlich abbildbaren Individuen in der Ontologie gesucht, die in Form einer Parameterliste mitgegeben werden kann. Für das erste Szenario werden dafür in 36 von 39 Fällen der Funktionsparameter „dishwasher“ identifiziert und auf das Ontologie-Individuum `Dishwasher::Object` abgebildet. Im zweiten Szenario wird der Funktionsparameter „Cereals“ zwar identifiziert, aber durch eine Bezeichner-Ähnlichkeit von weniger als 90% nicht auf das Individuum `Cereal::Object` abgebildet werden. Um dieses Problem zu lösen, wäre eine Senkung der Schranke für perfekte Bezeichner-Ähnlichkeiten denkbar.

Szenario	Lehrsequenzen	Unpassend	Parameter
5	39	5	36
6	34	2	0

Tabelle 10.5: Synthetisierte Methodennamen der Ende-zu-Ende-Evaluation

Fehlerursachen

Anschließend werden die Gründe für auftretende Fehler in der Synthese von Methodennamen beziehungsweise den darunterliegenden Klassifikationen der ersten Komponente untersucht. Einerseits lassen sich die Fehlklassifikationen der trainierten Modelle aus der ersten Teilaufgabe höchstwahrscheinlich auf die unbekannten Eingabesequenzen zurückführen. Diese Klassifikatoren werden wegen der ihnen zugrundeliegenden neuen Szenarien nicht nur mit neuen Zielsystem-Aktionen, sondern auch neuen Objekte der Umgebung konfrontiert. Andererseits ist die Lehrsequenz-Klassifikation, wie auch als Herausforderung (H 1.4) in Abschnitt 7.1 bereits definiert, nicht immer eindeutig. Dies sollen die in Beispiel 10.2.1 präsentierten Grenzfälle verdeutlichen. Für jede der Nutzereingaben N1 bis N3 ist unklar, ob der Nutzer dem Zielsystem dauerhaftes Wissen vermitteln (Lehrsequenz) oder die Ausführung der Anweisungen befehligen (Befehlssequenz) möchte. Im Entwurf in Abschnitt 8.1.1.2 wurde entschieden, diese unklaren Äußerungen der Gruppe der Befehlssequenzen zuzuordnen. Der trainierte Klassifikator detektiert jedoch jede der vier präsentierten Nutzereingaben als Lehrsequenz. Dieses Fehlverhalten wurde auch bereits in der isolierten Klassifikatorevaluation in Abschnitt 9.1.2.5 diskutiert. Für die Evaluation der Synthese von Methodennamen führt dieser Fehlerfall zu einer Bewertung mit Wahrheitswert falsch positiv (FP).
Eine falsch positiv detektierte Lehrsequenz führt wiederum zur Synthese eines nicht erwarteten Methodennamens, basierend auf einem extrahierten Funktionsnamen und zugehörigen Funktionsparametern (siehe Tabelle 8.5). Für die obige Synthesen-Evaluation in Tabelle 10.3 wird ein falsch positiv klassifiziert, die Präzision sinkt. Gleichzeitig heißt dies für die nachfolgende Evaluation der Ontologie-Individuen in Abschnitt 9.3.4.2, dass diese, als Methodenkopf interpretierten Elemente, wiederum nicht von der Abbildungssuche für den Methodenrumpf behandelt werden. Damit müssen sie in der nachfolgenden Evaluation als fehlend, das heißt als falsch negativ, bewertet werden. Im Gegensatz dazu führt eine falsch negativ detektierte Lehrsequenz dazu, dass ein erwarteter Methodenname nicht synthetisiert wird. Dies führt nicht nur zu einer Verschlechterung der Ausbeute der Synthesen-Evaluation in Tabelle 10.3, sondern auch zur Interpretation des Funktionsnamen und Funktionsparameters als Funktionsaufruf für die nachfolgende Abbildungssuche. Das hat wiederum zur Folge, dass in der nachfolgenden Evaluation der Ontologie-Individuen in Abschnitt 9.3.4.2 ein falsch positives Ergebnis klassifiziert wird.

Beispiel 10.2.1: Unklare Klassifikation von Lehrsequenzen

N1: „hi robot i would like you to start the dishwasher the dishwasher can be turned on by the blue button being pressed twice and then closing the door“

N2: „hey armar we are going to start the dishwasher so what we have to do is first make sure the dishwasher is closed and then press the button to start the dishwasher“

N1: „hi robot i want you to prepare some cereal you do this by pouring cereal into the (...) then you pour the milk into the bowl“

Überdies lassen sich weitere, nicht auf die binäre Lehrsequenz-Fehlklassifikation der ersten Komponente zurückzuführende, Fehlerursachen erkennen. Diese wurden auch bereits

in der isolierten Evaluation der Ontologie-Abbildung in Abschnitt 9.3.4.1 angesprochen. In Beispiel 10.2.2 wird eine Auswahl dieser Fehler mit ihrem synthetisierten Methodennamen vorgestellt. Dazu wird in der jeweilig ersten Zeile die betrachtete Eingabesequenz präsentiert. Die zweiten Zeile gibt die Funktionsnamen und Funktionsparameter an, die innerhalb der zweiten Teilaufgabe der Arbeit gebildeten wurden. Zuletzt wird in der dritten Zeile der synthetisierte Methodenname angegeben. Der synthetisierte Methodenname `pushTheBlueButton` zur ersten Eingabesequenz E1 des Szenario 5 „Start the Dishwasher“ ist gänzlich ungeeignet. Der Fehler lässt sich darauf zurückführen, dass die Eingabesequenz innerhalb der binären Klassifikation der ersten Komponente zwar korrekt als Lehrsequenz erkannt wurde, die darauffolgende Mehrklassen-Klassifikation jedoch die falschen Instruktion als Methodenkopf (in kursiv) klassifiziert. Eine weitere Fehlerursache zeigt sich bei Betrachtung der zweiten Eingabesequenz E2. Hier wird dem synthetisierten Namen ein unpassender Suffix „FirstHello“ angehängt. Dieser Suffix ist Teil der extrahierten Funktionsparameter, aus welchem die Methodennamen nach dem implementierten Vorgehen in Abschnitt 9.3.1 gebildet werden. Die selbe Fehlerursache lässt sich auch im zweiten synthetisierten Methodennamen erkennen.

Beispiel 10.2.2: Synthese von unpassenden Methodennamen

E1: „hello armar you have to *push the blue button* in order to turn on the dishwasher and then you must (...)“
push : [the blue button, in order to, hello armar you]
→ `pushTheBlueButton`

E2: „hello armar to start the dishwasher first you will have to (...)“
start : [the dishwasher, first, hello armar]
→ `startDishwasherFirstHello`

E3: „hi i am going to show you how to prepare cereal the first thing to do it put (...)“
prepare show : [cereal the first thing to]
→ `prepareCerealFirstThing`

10.2.2 Abbildung auf bestehende Individuen

Der zweite Teil der Ausgabe des Agenten entspricht einer Sequenz aus auf die Zielsystem-Ontologie abgebildeten Funktionsaufrufen. Im Falle einer Lehrsequenz bildet diese den Methodenrumpf, bei einer verbalisierten Befehlssequenz wird diese als Skript interpretiert. Für die Bewertung der abgebildeten Ontologie-Individuen wird jeder Funktionsaufruf der Eingabesequenz einzeln betrachtet. Dabei werden, wie auch schon in der isolierten Evaluation der Ontologie-Abbildung in Abschnitt 9.3.4.2, zwei Varianten unterschieden. In der ersten Variante werden die abgebildeten Methoden-Individuen, ohne Betrachtung der gefundenen Parameter, untersucht. In der zweiten Variante werden die Ausgaben des Werkzeuges auf korrekt abgebildete, vollständige Methodensignaturen überprüft. Neben der korrekten Abbildung des Methoden-Individuums müssen hierzu zusätzlich alle benötigten Parameter-Individuen mit der erwarteten Musterlösung übereinstimmen. Die Musterlösung wird hierbei aus der betrachteten Domänenontologie aus Tabelle 9.26 erstellt. Diese wurde bereits in der separaten Ontologie-Evaluation in Abschnitt 9.3.4.2 ausführlich beschrieben.
Die Implementierung der Abbildungssuche aus Abschnitt 9.3.2 gibt eine Rangfolge aus gefundenen Individuen-Kandidaten eines Funktionsaufrufs zurück. Deshalb wird im Folgenden einerseits bewertet, wie häufig sich die gesuchten Ontologie-Elemente an erster Stelle dieser Rangfolge befinden. Diese werden als Top-1-Ergebnisse bezeichnet. Als Zweites werden die gefundenen Ontologie-Abbildungen unter Betrachtung der ersten drei Rangfolgenplätze verglichen. Diese Ergebnisse werden als Top-3-Ergebnisse bezeichnet.

Tabelle 10.6: Ende-zu-Ende-Ergebnisse für Methoden-Individuen

		Präzision	Ausbeute	F_1	Ausbeute*	F_1*
Top-1	Szenario 5	82.3	85.4	83.9	94.4	87.9
	Szenario 6	92.0	87.6	89.8	92.9	92.5
	Gesamt	88.6	86.9	87.7	93.4	90.9
Top-3	Szenario 5	28.6	86.8	43.1	95.8	44.1
	Szenario 6	32.2	92.4	47.7	98.0	48.5
	Gesamt	31.0	90.5	46.1	97.3	47.0

Für die Vergabe der Wahrheitswerte an die unterschiedlich auftretenden Ausgaben des Werkzeuges wird das Bewertungsschema der isolierten Abbildungs-Evaluation aus Tabelle 9.30 befolgt. Bei Übereinstimmung des Funktionsaufrufs der Musterlösung mit dem gefundenen Methoden-Individuum beziehungsweise der Methodensignatur, wird ein richtig positiv gewertet. Wird keine Abbildung gefunden, wird ein falsch negativ klassifiziert. Wird eine zusätzliche Abbildung gefunden, die nicht in der Musterlösung gefordert wird, wird es als falsch positiv gewertet. Zuletzt wird ein falsch abgebildetes, aber gefordertes Methoden-Individuum nicht nur als falsch positiv, sondern auch als falsch negativ klassifiziert. Dies wird auch bei einer fehlerhaften Methodensignatur (siehe Definition 9.3.1) gewertet. Nach diesem Bewertungsschema können anschließend die Ausgaben des Agenten mit den händisch erstellten Musterlösungen verglichen werden.

10.2.2.1 Methoden-Individuen

In diesem Abschnitt wird die Güte des implementierten Agenten bei Eingabe der neu erhobenen, natürlichsprachlichen Nutzereingaben aus der Evaluationsgrundlage anhand der ersten Evaluationsvariante gemessen. Dazu werden die gefundenen Abbildungen auf korrekte Methoden-Individuen untersucht. In Tabelle 10.6 werden die Ergebnisse veranschaulicht, wobei die Tabelle in Top-1- und Top-3-Ergebnisse aufgegliedert ist. Die Szenarien werden jeweils getrennt, als auch gemeinsam evaluiert und zusammengefasst in der jeweiligen dritten Zeile präsentiert. Es werden die Metriken Präzision, Ausbeute und F_1 betrachtet (siehe Abschnitt 2.3).

Aus der Tabelle 10.6 lässt sich ablesen, dass bei Betrachtung der Top-1-Ergebnisse, die Ausgaben des Agenten für die Abbildung von Methoden-Individuen über alle Eingaben einen guten F_1-Wert von 87.7% erreicht. Die Präzision ist mit 88.6% gering höher als die Ausbeute mit 86.9%. Bei einer ersten Untersuchung der Fehlerursachen lies sich die geringe Ausbeute, wie bereits bei isolierten Evaluation der dritten Teilaufgabe in Abschnitt 9.3.4.2, auf einen wiederkehrenden Fehler des semantischen Rollen-Erkenners zurückführen. Dabei detektiert der semantische Rollen-Erkenner fälschlicherweise manche Verben nicht als Aktionen (siehe Abschnitt 8.1.3). Dies führt dazu, dass ein kompletter Funktionsaufruf in der aufgebauten Datenstruktur aus Teilaufgabe zwei (siehe Abschnitt 8.2) fehlt. Aufgrund dieses fehlenden Funktionsaufrufs kann in der dritten Teilaufgabe der Ontologie-Abbildung wiederum keine Abbildungssuche (siehe Abschnitt 8.3) durchgeführt werden. Deshalb können die entsprechenden Methoden-Individuen nicht detektiert werden, wodurch mehrere falsch negative Wahrheitswerte für das Nicht-Auffinden der Methoden-Individuen gezählt werden und somit die Ausbeute sinkt. Diese Fehlerursache tritt bei den 473 gezählten Funktionsaufrufen insgesamt 33 mal auf. Deshalb werden Ausbeute und F_1-Maß unter Ausschluss dieser Fehlerursache nochmals berechnet und in den letzten beiden Spalten der Tabelle unter Ausbeute* und F_1* aufgetragen. Die Präzision wird durch diesen Fehlerausschluss nicht verändert. Somit ergibt sich eine verbesserte, gesamte Ausbeute* von 93.4% und ein F_1*-Maß von 90.9% für die Top-1-Ergebnisse. Dies bedeutet, dass 93.4% aller auf dem ersten Platz eingeordneten Methoden-Individuen der erstellten Musterlösung

Tabelle 10.7: Ende-zu-Ende-Ergebnisse für vollständige Methodensignaturen

		Präzision	Ausbeute	F_1	Ausbeute*	F_1*
Top-1	Szenario 5	58.9	64.9	61.7	72.2	64.9
	Szenario 6	71.1	67.9	69.5	72.1	71.6
	Gesamt	66.8	67.0	66.9	72.1	69.4
Top-3	Szenario 5	20.0	66.2	30.8	73.7	31.5
	Szenario 6	25.3	72.4	37.5	76.8	38.1
	Gesamt	23.5	70.4	35.2	75.8	35.8

entsprechen.
Bei Betrachtung der ersten drei Ranglistenplätze für einzelnen gefundenen Funktionsaufrufe der Agentenausgabe steigt die Ausbeute* weiter auf 97.3%. Im Gegenzug sinkt die Präzision durch das Hinzuzählen von falsch positiven Wahrheitswerten für jeden weiteren betrachteten Platz der Rangfolge (siehe Tabelle 9.30) auf 31.0% bei einem Maximalwert von 33%.

10.2.2.2 Vollständige Methodensignaturen

In diesem Abschnitt wird die Güte des implementierten Agenten bei Eingabe neuer, natürlichsprachlicher Nutzereingaben anhand der zweiten Evaluationsvariante gemessen. In der zweiten Variante werden die Ausgaben auf korrekte Methoden-Individuen inklusive vollständiger, korrekt abgebildeter Parameter-Individuen, den Methodensignaturen, untersucht. Tabelle 10.7 präsentiert die erzielten Ergebnisse, wobei die Tabelle wieder in Top-1- und Top-3-Ergebnisse und die einzelnen Szenarien aufgeteilt ist. Auch diesmal werden die Metriken Präzision, Ausbeute und F_1 betrachtet (siehe Abschnitt 2.3).

Die Tabelle 10.7 veranschaulicht, dass bei Betrachtung der Top-1-Ergebnisse, die Ausgaben des Agenten für die Abbildung vollständiger Methodensignaturen über alle Eingaben einen F_1-Wert von 66.9% erzielt. Die Präzision fällt mit 66.8% fast ähnlich hoch aus wie die Ausbeute mit 67%. Wie auch schon im vorherigen Abschnitt 10.2.2.1 ist die geringe Ausbeute teilweise auf einen Fehler des semantischen Rollen-Erkenners zurückzuführen. Deshalb werden mit Ausbeute* und F_1* die Metrik des Agenten unter Ausschluss des Fehlers nochmals berechnet. Es wird eine verbesserte Ausbeute von 72.1% für die Top-1-Ergebnisse erreicht. Somit steigt auch der erzielte F_1*-Wert auf 69.4% um weitere 2.5 Prozentpunkte. Dies bedeutet, dass 69.4% alle durch den implementierten Agenten gefundenen Ontologie-Individuen den erwarteten, vollständigen Methodensignaturen der Musterlösung entsprechen. Die Betrachtung der Top-3-Ergebnisse ergibt eine weitere Steigerung der Ausbeute* auf 75.8%. Die Präzision sinkt dagegen auf 23.5%, was zu einem gemeinsamen F_1*-Wert von 35.8 führt.
Zuletzt werden die Daten in Tabelle 10.8 betrachtet. Diese gibt den durchschnittlichen platzierten Rang eines korrekten Methoden-Individuums beziehungsweise der erwarteten Methodensignaturen an. Die Ranglistenwerte zeigen, dass der durch die Bewertungsfunktion in Gleichung 8.1 errechnete Platz durchschnittlich bei 1.05 für die Methoden-Individuen, sowie bei 1.10 für vollständige Methodensignaturen liegt. Der Einfluss des sechsten Szenarios auf den Durchschnitt der erreichten Ranglistenplätze fällt hierbei höher aus. Grund dafür ist, dass das sechste Szenario fast doppelt so viele verbalisierte Instruktionen enthält wie das fünfte (siehe Tabelle 10.2). Das unterstreicht die Güte der konzipierten Bewertungsfunktion, da sie das gesuchte Ontologie-Element in fast allen Fällen auf dem ersten Platz platziert. Es fällt jedoch, wie auch schon bei der isolierten Abbildungsevaluation in Abschnitt 9.3.4.2, auf, dass in manchen Fällen von Methoden-Individuen, die mit unterschiedlicher Parameteranzahl in der Ontologie vorkommen, das Individuum mit der kürzeren Parameterliste eine höhere Bewertung erhält. So werden beispielsweise für

Tabelle 10.8: Durchschnittlicher Rang der Abbildungssuche

	Methoden-Individuen	vollst. Methodensignaturen
Szenario 5	1.04	1.04
Szenario 6	1.06	1.11
Gesamt	1.05	1.10

die Anweisung „take the milk from the fridge" beide der Abbildungen `take(Milk)` und `take(Milk,Fridge)` gefunden und beide als korrekt angesehen. Wertvoller für das erstellte Skript des Agenten wäre die letztere, da sie mehr Informationen enthält. Diese ist jedoch erst an zweiter Stelle der Rangliste, nach der ersten Alternative mit nur einem Parameter, aufzufinden.

10.2.2.3 Fehlerursachen

In diesem Abschnitt werden die Ursachen der in der vorherigen Evaluation beobachteten Fehler der Abbildung auf bestehende Ontologie-Individuen diskutiert. Die Tabelle 10.9 präsentiert eine Auswahl dieser Fehler, wobei das Symbol `?` ein fehlendes, nicht abgebildetes Ontologie-Individuum repräsentiert. Dazu dient die erste Spalte der Identifikation des Fehlerfalls, die nächsten beiden Spalten geben eine konkrete Eingabesequenz und die dazu extrahierten Funktionsparameter an. In der vierten Spalte wird die erwartete Musterlösung angegeben und die letzte Spalte enthält die tatsächliche Ausgabe des Agenten. Da viele der Fehler bereits aus der separaten Ontologie-Abbildungsevaluation in Abschnitt 9.3.4.2 bekannt sind, wird im Folgenden in verkürzter Form auf diese eingegangen. Eine neue Fehlerursache, der innerhalb dieser Ende-zu-Ende-Evaluation hinzukommt ist der letzte Fall 8. Dort führen Fehlklassifikationen der in der ersten Komponente des Agenten integrierten Klassifikatoren dazu, dass nachfolgende Funktionsaufrufe nicht oder falsch erkannt werden und somit nicht korrekt abgebildet werden können. In Tabelle 10.9 wird dazu beispielhaft die Formulierung „*to make cereals*" genannt, welche als Lehrsequenz interpretiert und damit für die Synthese eines Methodennamens verwendet werden sollte. In diesem Fall wurde jedoch keine Lehrsequenz klassifiziert, es wird eine Abbildungssuche getätigt und die fehlerhafte Methodensignatur (siehe Definition 9.3.1) `make(Cereals)` gefunden. Im zweiten Beispiel dieses Fehlerfalls verhält es sich genau gegensätzlich. Es wird eine Lehrsequenz detektiert, jedoch der falsche Funktionsaufruf als Lehrsequenz-Bestandteil (in kursiv) zugewiesen. Deshalb fehlt die Abbildung dieses Funktionsaufrufs auf die Ontologie (hier `push(Button)`) und führt zu einer Verschlechterung der Ergebnisse. Mit 15 Nutzereingaben ist dieser Fehlerfall die zweithäufigste Fehlerursache des implementierten Agenten. Da sich dieser Fehlerfall ebenfalls auf die Synthese von Methodennamen auswirkt, wurde er bereits im vorherigen Abschnitt 10.2.1 ausführlich erklärt.
Die nachfolgenden Fehlerursachen sind unabhängig von der ersten Komponente des implementierten Agenten. Im ersten Fall wird aufgrund des namensbasierte Ansatzes auf Basis von Bezeichner-Vergleichen kürzere Wortsequenzen mit einer höheren Bewertung versehen. Dies trifft auch für das in Beispiel Tabelle 10.9 präsentierte Ontologie-Individuum `Fridge.Door::Object` im Vergleich zu `Cupboard.Door::Object` bei Vergleich mit dem Funktionsparameter „the door" zu.
Im zweiten Fall führen unbekannte Synonyme (siehe Abschnitt 2.1.5) zu fehlerhaften Abbildungen von Methoden-Abbildungen beziehungsweise Methodensignaturen. Ist ein Synonym wie beispielsweise „click" für die Ontologie-Methode `push::Method` nicht bekannt, so kommt es zu einer fehlenden Abbildung oder einer falschen Abbildung, wie es im zweiten Beispiel der Fall ist.
Eine weitere Fehlerursache wird durch den dritten Fehlerfall repräsentiert. Diese ergeben sich aus den Fehlern des PARSE-Agenten für semantischen Rollen (SRL), die jedoch auf neue Gründe und nicht den oben explizit ausgeschlossenen Fehlerfall zurückzuführen sind.

Ein Beispiel dafür bildet die erste Formulierung „the milk is found in the fridge“. Hier konnte zwar der korrekte Funktionsname „find“, jedoch nicht beide der benötigten Funktionsparameter gefunden werden. Auch im zweiten Beispiel wird in der Formulierung „pour milk onto cereals“ nur einer der beiden Funktionsparameter als solcher annotiert. Eine mögliche Ursache für die Fehler des integrierten Agenten könnten Grammatikfehler sein, die im Gegensatz zu Schreibfehlern nicht in den Eingabesequenzen behoben wurden.
Die vierte Fehlerursache ergibt sich aus semantisch falsch interpretierten Nutzereingaben. In den 473 gezählten Funktionsaufrufen tritt dieser Fehler 39 mal auf und repräsentiert damit die häufigste Fehlerursache dieses Agenten. Häufig werden in Formulierungen zusätzliche Funktionsaufrufe erkannt, die semantisch betrachtet, keine vom Zielsystem auszuführende Aktion beschreiben. Mit der Formulierung „press the button so *it starts washing*“ soll das Zielsystem einen Knopf drücken, jedoch wird auch der hintere Teil der Anweisung (in kursiv) als zweiter Funktionsaufruf auf die Ontologie-Methode `wash::Method` abgebildet. Auch mit der Anweisung „*make sure* the door is closed“ soll das Schließen einer Tür befehligt werden. Der Agent interpretiert jedoch den ersten Teil (in kursiv) ebenfalls als Funktionsaufruf und bildet ihn auf die fehlerhafte Methodensignatur `make(Fridge.Door)` ab.
Die Fehler des fünften Falls treten auf, wenn der Nutzer dem Zielsystem innerhalb einer Anweisung zusätzliche Kurzerklärungen mitteilt. Beispielsweise wird mit der Anweisung „press the button *to start it*“ das Drücken eines Knopfes befehligt, dass dadurch wiederum eine Maschine gestartet wird, ist eine Nebeninformation. Diese Nebeninformation wird in 17 Fällen durch den Agenten jedoch als zusätzlicher Funktionsaufruf detektiert. Da diese Kurzerklärungen jedoch synonyme Duplikate von richtig positiven Funktionsaufrufen repräsentieren, werden diese als korrekt gewertet. Treten bei der Ontologie-Abbildung dieser zusätzlichen Funktionsaufrufe Fehler auf, so werden diese, wie auch bei den anderen genannten Fehlerfällen, mit den entsprechenden Wahrheitswerten gewertet.
Der sechste Fehlerfall wird durch falsche oder fehlende Auflösung von Korreferenzen (siehe Abschnitt 2.2.1.6) ausgelöst. Im Beispielsatz „put it on the table“ muss das Personalpronomen *it* mit Hilfe von Kontextwissen über die gesamte Nutzereingabe auf das Nomen „bowl“ abgebildet werden. Dies wird durch den integrierten PARSE-Agenten für Kontextauflösungen umgesetzt. Kommt dieser jedoch zu einem falschen Ergebnis führt dies wiederum zu falschen Parameter-Abbildungen.
Der siebte und bereits aus der Abbildungsevaluation in Abschnitt 9.3.4.2 bekannte Fehlerfall lässt sich auf Formulierungsfehler des Nutzers selbst zurückführen. Häufig kommt es dazu, dass ein Nutzer falsche Wörter zur Beschreibung einer Zielsystemfunktionalität verwendet. Ähnlich oft unterschlägt der Nutzer, vermutlich unwissentlich, wichtige benötigte Informationen. Dies wurde bereits als Herausforderung (H 3.2) in Abschnitt 7.1 definiert. Die Anweisung „pour the milk“ benötigt für eine vollständige Abbildung auf die Zielsystem-Ontologie nicht nur den Parameter `pour.what`, sondern auch `pour.where`. Da dieser nicht angegeben wurde, führt dies zu einer fehlerhaften Methodensignatur. Die gleiche Fehlerursache ergibt sich für die Beispielformulierung „carrying to the table“, bei welcher der Nutzer die Information unterschlägt, welcher Gegenstand zum Tisch getragen werden muss.

10.3 Bewertung der Ergebnisse

Insgesamt werden innerhalb der Ende-zu-Ende-Evaluation des Agenten für einen unbekannten, neu erhobenen Datensatz gute Ergebnisse erzielt. Für die Detektion von Lehrsequenzen in der ersten Komponente der Arbeit auf Basis von trainierten neuronalen Netzen wird ein sehr guter F_1-Wert von 90.1% auf unbekannten Testdaten erreicht (siehe Tabelle 10.6, Herausforderung (H 1) in Abschnitt 7.1). Daraufhin werden die klassifizierten Sprachbestandteile der Nutzereingaben auf ihre Repräsentation im Programmkontext abgebildet (siehe Tabelle 8.5, Herausforderung (H 2)). Dies geschieht innerhalb der zweiten Komponente des Agenten, wodurch die Nutzereingabe nun entweder als Methodendefinition oder

Tabelle 10.9: Fehlerursachen für die Abbildungssuche von Methoden-Individuen

Ursache	Eingabe	Parameter	Musterlösung	Ergebnis
1 Bezeichner	„close the door“	[the door]	`close(Cupboard.Door)`	`close(Fridge.Door)`
	„go to the dishwasher“	[the dishwasher]	`go(Dishwasher)`	`go(Dishes)`
2 Synonyme	„click the button“	[the button]	`push(Button)`	-
	„load the dishwasher“	[the dishwasher]	`fill(Dishwasher, Dishes)`	`lowerHead`
	„pour a portion into a bowl“	[a portion, into a bowl]	`pour(Milk,Bowl)`	`put(Popcorn,Bowl)`
3 SRL	„the milk is found in the fridge“	[in the fridge]	`find(Milk,Fridge)`	`find(?,Fridge)`
	„pour milk onto cereals“	[milk onto cereals]	`pour(Milk,Cereals)`	`pour(Milk,?)`
4 Semantik	„press the button so *it starts washing*“	[ ]	-	`wash(?)`
	„*make sure* the door is closed“	[sure, the door]	-	`make(Fridge.Door)`
5 Erklärung	„press the button *to start it*“	[it]	-	`start(?)`
	„press the button *to turn it on*“	[it, on]	-	`turnOn(Kitchen)`
6 Korref.	„put it on the table“	[it, table]	`put(Bowl,Table)`	`put(Kitchen,Table)`
	„put some in the bowl“	[some, the bowl]	`put(Milk,Bowl)`	`put(?,Bowl)`
7 Benutzung	„pour the milk“	[the milk]	`pour(Milk,Bowl)`	`pour(Milk,?)`
	„carrying to the table“	[the table]	`carry(Bowl,Table)`	`carry(?,Table)`
8 Klassif.	„to make cereals“	[ ]	-	`make(Cereals)`
	„*push the button* to turn on the dishwasher“	[dishwasher]	`push(Button)`	`turnOn(Dishwasher)`

als Skript aus Zielsystem-Bibliotheksaufrufen interpretiert werden kann. Zuletzt werden mit der letzten Komponente die Elemente dieser Programmrepräsentation mit Elementen der Zielsystem-Ontologie verknüpft (Herausforderung (H 3)). Hierbei wird für die korrekte Abbildung einzelner Zielsystem-Funktionsaufrufe ein F_1-Wert[1] von 90.9% erzielt. Für die korrekte Abbildung dieser Funktionsaufrufe mit vollständiger Parameterliste auf die Zielsystem-Programmschnittstelle erreicht der entwickelte Agent eine gutes F_1-Maß von 69.4% (siehe Tabelle 10.7). Weiterhin wird in über 90.4% der Fälle ein aussagekräftiger Methodenname im Falle einer Lehrsequenz synthetisiert.
Es lässt sich zudem feststellen, dass die gemessene Ausbeute die Präzision für das Finden von Zielsystem-Abbildungen in allen Fällen übersteigt. Ein Ansatz, der die Ausbeute gegenüber der Präzision bevorzugt, ist erwünscht, da im Projektkontext zunächst möglichst viele Zielsystem-Abbildungen gefunden werden sollen, um in der Nachverarbeitung der PARSE-Verarbeitungskette (siehe Kapitel 3) überflüssige Instruktionen auszufiltern. Dies steht im Gegensatz zum gewählten Ansatz für die Lehrsequenz-Erkennung der ersten Komponente. Dort wird die Präzision als wichtiger betrachtet, um möglichst genaue Ergebnisse für die Klassifikation der Nutzereingaben zu erreichen. Somit wird mit dem konzipierten Ansatz dieser Arbeit zur Synthese von Methodendefinitionen ein zufriedenstellendes Ergebnis erreicht.

In Beispiel 10.3.1 werden hierzu zwei erfolgreiche Abbildungen von exemplarischen Eingabesequenzen auf Skripte bestehend aus Zielsystem-Bibliotheksaufrufen präsentiert. Das darauffolgende Beispiel 10.3.2 zeigt anschließend jeweils zwei unterschiedliche Nutzereingaben, die der entwickelte Agent erfolgreich in Methodendefinitionen transformiert. Diese korrekt transformierten Beispieleingaben zeigen auf, dass der Agent in der Lage ist, freie Formulierungen in natürlicher Sprache zu verarbeiten (Herausforderung (H 1.1) in Abschnitt 7.1). Hierbei unterscheiden sich die formulierten Eingabesequenzen eines Szenarios deutlich (Herausforderung (H 1.3)). Sie weisen somit eine unterschiedliche Syntax (siehe Abschnitt 2.1.1) auf. Dennoch beschreiben die Eingabesequenzen eines Szenarios eine semantisch gleiche Aufgabenstellung (siehe Abschnitt 2.1.2). Somit ist der implementierte Ansatz in der Lage, die Semantik der Nutzereingaben trotz syntaktischer Unterschiede erfolgreich auf die gleiche Methodendefinition beziehungsweise das gleiche Skript abzubilden (Herausforderung (H 1.5) und (H 3.4)). Dies soll durch die Präsentation unterschiedlicher Nutzereingaben in Beispiel 10.3.2 verdeutlicht werden. Weiterhin ist es dem Agenten möglich, lange Anweisungssequenzen zu verstehen, die eine große Anzahl an unterschiedlichsten Objekten und Aktionen der Umgebung referenzieren.

Beispiel 10.3.1: Skripte aus Zielsystem-Bibliotheksaufrufen

Szenario 5

„hi robot close the dishwasher and press the blue button two times“

```
close(Dishwasher.Door)
press(Dishwasher.Button)
```

Szenario 6

„carefully take the milk out of the fridge take it to the kitchen table pour the cereals into the bowl first and then carefully pour the milk into bowl“

```
retrieve(Milk, Fridge)
take(Milk, Table)
pour(Cereal, Bowl)
pour(Milk, Bowl)
```

[1] Entspricht dem F1*-Wert, bei welchem der spezifische Verb-Fehlklassifikation (siehe Abschnitt 10.2.2.1) des integrierten semantischen Rollen-Erkenners ausgeschlossen wird.

Beispiel 10.3.2: Synthetisierte Methodendefinitionen

Szenario 5

„hi armar when you need to start the dishwasher you have to first close the dishwasher door and then press the blue button“

„hi armar i am going to show you how to use the dishwasher you have to close the door to the machine when it is closed press the blue button“

```
new startDishwasher(Dishwasher):
     close(Dishwasher.Door)
     press(Dishwasher.BlueButton)
```

Szenario 6

„hi armar time to learn how to prepare cereals go to the fridge and open the door to remove milk and place it on the table fill the bowl with cereal and then pour the milk over the cereal“

„preparing cereals means you have to go to the fridge open the door get the milk place the milk on the table next to the bowl and the cereal box after that you have to fill the bowl with cereal and then pour the milk in to the bowl“

```
new prepareCereals():
     go(Fridge)
     open(Fridge.Door)
     retrieve(Milk, Fridge)
     place(Milk, Table)
     fill(Cereal, Bowl)
     pour(Milk, Bowl)
```

Es existieren jedoch auch Fälle, bei denen der Ansatz auf Probleme trifft. Diese Fehler wurden bereits in Abschnitt 10.2.2.3 angesprochen und in Tabelle 10.9 anhand von Beispielen visualisiert. Im Folgenden sollen diese kurz zusammengefasst werden. Einerseits ist es wichtig, zu nennen, dass die diktierten Nutzereingaben keine sinnvollen, korrekt ausführbaren Prozeduren beschreiben müssen. Stattdessen wird das Ziel verfolgt, die einzelnen Bestandteile einer Lehrsequenz auf entsprechende Elemente der Zielsystem-API abzubilden. Damit wird nicht gefordert, eine ausführbare Sequenz aus Funktionsaufrufen zu erstellen, welche die vom Nutzer gewünschte Problemstellung löst. Die erstellten Agentenausgaben stellen ein Zwischenergebnis dar, die es in einem Nachverarbeitungsschritt des Projekts PARSE auf ihre Sinnhaftigkeit zu prüfen und mit weiteren Informationen anzureichern gilt. Diese Arbeit kann anschließend genutzt werden, eine Wissensrepräsentation der synthetisierten Methodendefinitionen in der Programmierschnittstelle des Zielsystems aufzubauen.

Neben der Fehlerursache, dass der Nutzer wichtige Informationen unterschlägt, bildet das größte Problem der Agentenausführung das fehlende Verständnis von semantischen Zusammenhängen. Diese sind nicht nur zur korrekten Auflösung von Korreferenzen und semantischen Rollen wichtig, sondern auch für die Erkennung von Funktionen, deren Reihenfolge und gegenseitige Referenzierung oder der Identifikation von Akteuren dieser auszuführenden Funktion. Deshalb müssen die Ausbeute-orientierten Agentenausgaben in einem Nachverarbeitungsschritt der PARSE-Verarbeitungskette unter Anderem von diesen Funktionsduplikaten bereinigt werden. Auch die Integration von Schleifen und weiteren Kontrollstrukturen wird noch nicht unterstützt. Jedoch ist ermöglicht der implementierte Ansatz deren Integration (Herausforderung (H 3.5) in Abschnitt 7.1). Diese Anforderung an die Erweiterbarkeit des Agenten wurde in der Analyse in Abschnitt 7.3.2

als Anforderung (A3) definiert. Weiterhin bietet die Programmierschnittstelle noch keine Behandlung von variablen Parameterlisten sowie einer Spezifikation von Multiplizitäten eines Parameters an. Es wird jedoch davon ausgegangen, dass der entwickelte Agent, falls diese Eigenschaften in der Ontologie in Form von zusätzlichen Individuen und Relationen hinzugefügt wurden, eine Abbildung ohne weitere Modifikation der Implementierung ermöglicht. Zusätzlich dazu ist der namensbasierte Ansatz auf Basis von Bezeichner-Ähnlichkeiten ausbaufähig, da die verwendeten Distanzmetriken trotz eines Ensemble-Ansatzes (siehe Abschnitt 8.3.2.1), vorverarbeiteten Eingaben, Synonymen und Korreferenzauflösungen fehleranfällig sind.

Zuletzt lässt sich durch die Evaluation mit der in Abschnitt 10.1 erhobenen Datengrundlage auf Basis neuer Szenarien zeigen, dass der Ansatz des implementierten Agenten szenarienunabhängig ist. Diese Anforderung an die Domänenunabhängigkeit des Agenten wurde in der Analyse in Abschnitt 7.3.2 als Anforderung (A1) definiert. Dies kann auch dadurch belegt werden, dass einerseits die Verteilung der Daten aus der Vorstudie aus Tabelle 8.3 mit der Datenbasis der Ende-zu-Ende-Evaluation in Tabelle 10.2 sich stark ähnelt. Andererseits ist erkennbar, dass in den separaten Evaluationen der einzelnen Komponenten auftretenden Fehlerfälle mit denen der hiesigen Ende-zu-Ende-Evaluation übereinstimmen. Auch die Anteile des Auftretens dieser Fälle ist vergleichbar. Daraus lässt sich folgern, dass sich für den implementierten Agenten bei Eingabe von Nutzereingaben aus unbekannten Szenarien nicht signifikant mehr Fehler oder Fehler anderer Art ergeben. Somit kann eine Szenarienunabhängigkeit des Ansatzes belegt werden. Eine Szenarienunabhängigkeit des Agenten würde wiederum bedeuten, dass auch Nutzereingaben, die ein anderes Zielsystem sowie eine unterschiedliche Zielsystem-Umgebung ansprechen, behandelt werden können.

11 Erkenntnisse über eingesetzte Technologien

Zuletzt werden die im Laufe der Arbeit gesammelten, interessanten Einsichten über verwendete Technologien kurz festgehalten. Diese wurden projektspezifisch erhoben, lassen sich in den meisten Fällen jedoch auf andere Kontexte und ähnliche Problemstellungen übertragen. Diese Erkenntnisse lassen sich einerseits in Erkenntnisse zur Thematik des maschinellen Lernens analog zum ersten Teil dieser Arbeit sowie andererseits in Problemstellungen zum Thema der Verarbeitung natürlicher Sprache und Softwaretechnik im zweiten Teil unterteilen.

Generell lässt sich die Erkenntnis ziehen, dass die Menge, Qualität und Diversität einer Datengrundlage viel entscheidender sind als eine zusätzliche Modellkonfiguration. Auch der Leiter der Tesla-Abteilung für künstliche Intelligenz (engl. *artificial intelligence, AI*) Andrej Karparthy, äußert eine ähnliche Feststellung:

> *„As far as I'm aware adding more data is pretty much the only guaranteed way to monotonically improve the performance of a well-configured neural network almost indefinitely."*

Für die Lösung von textuellen Entscheidungsproblemen auf Basis neuronaler Netze lassen sich weitere Feststellungen aus dieser Arbeit ableiten. Dazu gehört die Erkenntnis, dass vor Allem die Vorverarbeitung der Daten eine wichtige Rolle spielt. Innerhalb der hiesigen Klassifikationsprobleme konnten mit einer tokenisierten Eingabe durchweg bessere Ergebnisse erzielt werden, als mit zusätzlich auf den Wortstamm abgebildeten (lemmatisierten) Textdaten. Dies geht in Einklang damit, dass eine Vergrößerung des Vokabulars die Güte der Modelle weiter verbessert, wobei das betrachtete Vokabular aus weit unter 1000000 Wörtern besteht. Außerdem führte der Einsatz von Wortvektoren im Gegensatz zu rein vokabularisierten Datensätzen in allen Fällen zu einer Steigerung der Modellgüte. Der Vergleich der vortrainierten Wortvektor-Ansätze Word2Vec, Fasttext und GloVe ergab in den überwiegenden Fällen eine beste Modellgüte bei Einsatz des Fasttext-Wortvektormodells. Weiterhin ergab der Vergleich unterschiedlicher neuronaler Netzarchitekturen auf die beiden textuellen Klassifikationsprobleme dieser Arbeit in allen Fällen eine Überlegenheit der bidirektionalen LSTM-Architekturen. Auch die faltenden neuronalen Netze (engl. *convolutional neural net*) konnten im hiesigen binären Klassifikationsproblem gleich gute, in manchen Fällen sogar die besten Werte, erzielen. Zudem belegen die Klassifikationsergebnisse der beiden besten Modelle mit jeweils über 95.9% Genauigkeit auf unbekannten, ran-

domisierten Testdatensätzen, dass der Einsatz von neuronalen Netzen zur Lösung von komplexen (Sequenzmarkierungs-)Problemen textueller Daten vielversprechend ist. Überdies wurden diese Ergebnisse ohne jeglichen Einsatz der populären Attention-Mechanismen erzielt.
Die Konfiguration der Hyperparameter modellierter neuronaler Netzarchitekturen ergab, dass die Anpassung der Lernrate sowie ein frühzeitiger Abbruch des Modelltrainings unter Beobachtung des ermittelten Validierungsfehlers, die Findung des besten Modells erheblich beschleunigte. Außerdem müssen für die Lösung textueller Sequenzmarkierungsprobleme durch maschinelles Lernen besonders für variable Eingabelängen weitere Einzelheiten beachtet werden. Einerseits wird in den meisten Fällen eine Auffüllung dieser Sequenzen auf eine einheitliche Länge benötigt. Da ein neuronales Netz, statt bei Ende der ursprünglichen Sequenz seine Prädiktionen abzubrechen, für jedes Element der aufgefüllten Sequenz eine Prädiktion liefert, müssen diese bei der nachfolgenden Evaluation herausgerechnet werden. Außerdem führte im Zuge der Modellbildung dieser Arbeit die Begrenzung der Sequenzlänge, ausgehend von der maximalen Länge der erhobenen Eingabesequenzen auf eine Sequenzlänge basierend auf dem 99%-Quantil, zu einer zusätzlichen Verbesserung der Ergebnisse.
Zuletzt ist die Java-Integration von Keras- beziehungsweise Tensorflow-Modellen mittels der Bibliothek Deeplearning4j nur begrenzt möglich. Generell wird eine einfache und abstrahierte Einbindung trainierter Modelle bereitgestellt, wenn das Ziel einer Datenklassifikation auf neuen, unbekannten Daten verfolgt wird. Jedoch wird zum Zeitpunkt dieser Arbeit nicht für alle der unterschiedlichen Schichten eine Integration bereitgestellt. Dies gilt auch das BERT-Sprachmodell von Google, für das zu diesem Zeitpunkt noch keine Java-Anbindung zur Verfügung steht.

Aus den weiteren implementierten Komponenten dieser Arbeit lassen sich ebenfalls Erkenntnisse ziehen. Zum Einen über den Einsatz von Bezeichner-vergleichbasierten (engl. *string matching*) Verfahren. Diese bieten eine valide Vorgehensweise, um einen domänenunabhängigen Ansatz für das Finden von Abbildungen zwischen zwei Mengen zu finden. Dafür mussten jedoch eine Reihe an Vor- und Nachverarbeitungsschritten durchgeführt werden, um die besagte zufriedenstellende Güte der Ergebnisse zu erhalten. Andererseits liefert dieser Ansatz besonders für den Vergleich langer Zeichenketten oder für Zeichenketten bestehend aus mehreren Wörtern häufig nicht nachvollziehbare Ergebnisse.
Weiterhin führte die Integration von Synonymen einerseits zu einigen zusätzlichen Detektionen, welche ohne diese nicht möglich gewesen wären. Jedoch wird in vielen Fällen auch ein im betrachteten Kontext gänzlich falsches Synonym gefunden, das wiederum zu falschen Ergebnissen der entwickelten Anwendung führen würde. Deshalb ist es wichtig, bei einem Einsatz von Synonym-Informationen die gefundenen Synonyme mit einer geringeren Konfidenz zu bewerten, als die tatsächlich gefundenen Satzbestandteile.
Zuletzt - und das sollte eigentlich selbstverständlich sein - erweist sich die Verwendung von Protokollierungswerkzeugen (engl. *logging tools*), regelmäßig platziert in den einzelnen implementierten Klassen, als sehr hilfreich. Auf diese Weise kann der Programmablauf einer entwickelten Anwendung, sei es zu reinen Informationszwecken oder der Fehlerbehebung, übersichtlich verfolgt werden.

12 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Ansatz entwickelt, mit dem die Synthese von Methodendefinitionen aus natürlichsprachlichen Äußerungen ermöglicht wird. Ziel ist es, einem Zielsystem neue und komplexe Funktionalitäten durch Nutzereingaben in natürlicher Sprache beizubringen. Mit dem entwickelten Ansatz dieser Arbeit soll es anschließend möglich sein, diese neuen Funktionalitäten aus der Programmierschnittstelle des betrachteten Zielsystems per Sprachkommando aufzurufen. Ein solches Zielsystem soll mit Realwelt-Objekten interagieren können, wie beispielsweise ein Roboter in einer Küchenumgebung. Für die Lösung dieser Problemstellung gehört zum Einen die Erkennung von verbalisierten Funktionserweiterungen eines Zielsystems in den Nutzereingaben. Anschließend muss die Extraktion des Inhaltes dieser Funktionserweiterungen sowie deren Abbildung auf bestehende Zielsystem-Funktionalitäten ermöglicht werden. Im letzten Schritt müssen diese abgebildeten Zielsystem-Bibliotheksaufrufe als eine neue Methodendefinition gespeichert werden. Zusätzlich dazu wird auch die Erstellung eines Ausführungsskriptes aus Zielsystem-Bibliotheksaufrufen im Falle von diktierten, natürlichsprachlichen Anweisungsfolgen statt einer diktierten Funktionserweiterung unterstützt.

Zunächst wurde innerhalb einer Vorstudie eine Datenbasis von 3410 textuellen Eingabesequenzen über die wissenschaftliche Crowd-Sourcing-Platform Prolific.ac erhoben. Im Anschluss wurde auf Basis einer ausführlichen Analyse dieser Daten ein Ansatz für eine mehrstufige Klassifikation von Nutzerintention und Inhalten einer Eingabesequenz entworfen. Beginnend mit einer binären Klassifikation wurden verschiedene neuronale Netze trainiert, um die Fragestellung zu beantworten, ob der Nutzer das Hinzufügen einer neuen Zielsystem-Funktionalität oder eine direkte Ausführung der diktierten Anweisungen beabsichtigt. Hierbei konnte in der separaten Klassifikatorevaluation eine sehr gute Genauigkeit von 95.9% mit einem bidirektionalen LSTM beziehungsweise von 98.8% mit dem BERT-Sprachmodell[1] auf unbekannten, randomisierten Testdaten erreicht werden. In der zweiten Stufe der Klassifikation sollte daraufhin erkannt werden, wo in einer Nutzereingabe der Wunsch zur Funktionserweiterung verbalisiert und wo ihr Inhalt beschrieben wird. Dieses Mehrklassen-Entscheidungsproblem wurde ebenfalls mittels neuronaler Netze gelöst. Die Modellevaluation ergab eine sehr gute Genauigkeit von 98.8%, wiederum mit einem bidirektionalen LSTM auf unbekannten, randomisierten Testdaten. Zuletzt wurde für die Extraktion der einzelnen Aktionen einer neuen Funktionalität ein bestehender PARSE-Agent für die Detektion von semantischen Rollen in die hiesige Arbeit integriert. Daraufhin wur-

[1]Quelle: `https://github.com/google-research/bert`, zuletzt besucht am 17.06.19

den die klassifizierten Sprachbestandteile einer Nutzereingabe, bestehend aus Aktionen und Entitäten, auf ihre Repräsentationen im Programmkontext, das heißt Funktionsnamen und Funktionsparameter abgebildet. Diese Programmrepräsentation modelliert eine Nutzereingabe entweder als Methodendefinition oder als Skript.
Im Anschluss darauf wurden die Elemente dieser Programmrepräsentation mit Elementen der Zielsystem-Programmierschnittstelle, hier modelliert durch eine Ontologie, verknüpft. Es wurde eine Kombination aus namensbasierter und Datentyp-basierter Suche implementiert, welche Abbildungen zwischen den extrahierten Programmbestandteilen auf bestehende Individuen der Zielsystem-Ontologie findet. Dabei wurden unter Anderen ein Ensemble aus Bezeichner-Vergleichsmetriken, verschiedene Sprachverarbeitungswerkzeuge, Synonymen sowie Korreferenz-Informationen, unter Anderem bereitgestellt durch weitere PARSE-Agenten, integriert. Der entwickelte Ansatz ermittelt eine Rangfolge solcher Abbildungskandidaten pro extrahierter Aktion der Nutzereingabe. Anschließend wurde die implementierte Abbildungssuche isoliert, das heißt unabhängig von den Klassifikations-Komponenten dieser Arbeit, evaluiert. Dabei wurde gemessen, dass die Abbildung einzelner Zielsystem-Funktionen sehr zuverlässig mit einem F_1-Wert von bis zu 83% ausfällt. Für die korrekte Abbildung dieser Funktionen inklusive vollständiger Argumentliste auf die Zielsystem-Programmschnittstelle erreicht der entwickelte Agent einen zufriedenstellenden F_1-Wert von 67%. Zusätzlich dazu wurde im Falle einer erkannten Funktionserweiterung durch den Nutzer, ein aussagekräftiger Methodenname für die neue Funktionalität synthetisiert. Hierfür konnte eine ebenfalls gute Genauigkeit von bis zu 84.3% gemessen werden. Im letzten Schritt der Implementierung wurden die Anwendungsausgaben dieser Arbeit in der Graphdatenstruktur des Projekts PARSE gespeichert.
Schließlich musste die entwickelte Anwendung in einer abschließenden Ende-zu-Ende-Evaluation getestet werden. Dazu wurden wiederum jeweils 100 neue Eingabesequenzen mit Prolific.ac zu zwei neuen Szenarien erhoben. Eine Evaluation der binären Klassifikationsergebnisse konnte einen F_1-Wert von 87% (Genauigkeit 82%) und mit einem zusätzlichen kombinatorischen Ansatz einen F_1-Wert von 90.1% (Genauigkeit 85%) erreichen. Dies bedeutet, dass eine sehr zuverlässige Erkennung der Nutzerintention bezüglich der Absicht einer Funktionserweiterung oder einer direkten Skriptausführung möglich ist. Die Evaluation der Anwendungsausgaben, welche den Ontologie-Abbildungen aus der letzten Komponente entsprechen, ergab einen sehr guten F_1-Wert von bis zu 90.9% für die korrekten Zielsystem-Bibliotheksaufrufe. Für die korrekte Abbildung vollständiger Funktionen, das heißt inklusive korrekt abgebildeter Parameter, wird ein zufriedenstellender F_1-Wert von bis zu 69.4% erreicht. Auch für die Synthese von Methodennamen wird in 90.4% der Fälle ein geeigneter Name für die neue Zielsystem-Funktionalität generiert. Im Vergleich zur separaten Ontologie-Abbildungsevaluation und trotz der hinzukommenden, möglichen Fehlklassifikationen aus den Klassifikationskomponenten, fallen die Ergebnisse ähnlich gut oder teilweise sogar besser aus. Dies unterstreicht die Robustheit der modellierten Klassifikatoren. Bei Eingabe von gänzlich neuen Nutzereingaben sind nicht signifikant mehr Fehler oder Fehler anderer Art messbar. Deshalb kann daraus eine Unabhängigkeit des implementierten Ansatzes von den betrachteten Nutzereingaben gefolgert werden.

Trotz dieser guten Ergebnisse lassen sich noch einige Punkte zur Verbesserung und vor Allem zur Ergänzung der vorgestellten Arbeit für die Synthese von Methodendefinitionen bestimmen. Einerseits sollte die Abbildung von variablen Parameterlisten sowie Erkennung von Multiplizitäten von Objekten aus der Umgebung möglich sein. Diese Eigenschaften werden von der betrachteten Zielsystem-Programmierschnittstelle noch nicht unterstützt und müssten erneut mit der entwickelten Anwendung getestet werden. Außerdem wären zusätzlich zu den bereits integrierten Objekt-Korreferenzen auch Referenzen zwischen zwei Funktionen hilfreich, um semantisch gleiche Aktionen innerhalb einer Nutzereingabe zu identifizieren. In einer weiterführenden Arbeit wäre es außerdem denkbar, bestehende Am-

biguitäten und fehlende oder nicht abbildbare Elemente durch gezielte Nachfragen einer Dialogkomponente an den Nutzer zu lösen. Das Gewinnen von Informationen durch einen eingeleiteten Klarstellungsdialog ist ein häufig verfolgter Ansatz der Programmierung in natürlicher Sprache [Gri75], um das computerbasierte Inferieren von Informationen zusätzlich zu unterstützen. Zudem könnte durch den Einsatz einer solchen Dialogkomponente vor der Einprogrammierung einer erkannten Funktionserweiterung die synthetisierte Funktionalität durch das Zielsystem natürlichsprachlich wiederholt und eine Bestätigung vom Nutzer abgewartet werden. Des Weiteren könnte bei wiederholter Ausführung der selben Anweisungsfolge der Nutzerwunsch nach einer Einprogrammierung dieser Funktionalität durch das Zielsystem erfragt werden.

Zuletzt stellt die Ausgabe der Anwendung ein Zwischenergebnis dar. Für die Persistierung der gewonnenen Methodendefinitionen als Funktionserweiterung müssen diese in einem Nachverarbeitungsschritt um Kontrollstrukturen wie Schleifen und Bedingungen ergänzt werden. Zusätzlich dazu müssen die diktierten Anweisungen auf ihre Sinnhaftigkeit geprüft werden. Um dieses Zwischenergebnis schließlich in ausführbaren Quelltext zu transformieren, müssen weitere Schritte umgesetzt werden. Es ist jedoch anzunehmen, dass sich die Lösung dieser Schritte nach Green und Petre [GP96] aufgrund der Nähe der modellierten Problemwelt zur letztendlichen Programmwelt (engl. *closeness of mapping*) einfacher gestaltet.

Literaturverzeichnis

[ACF+07] Allen, James ; Chambers, Nathanael ; Ferguson, George ; Galescu, Lucian ; Jung, Hyuckchul ; Swift, Mary ; Taysom, William: PLOW: A Collaborative Task Learning Agent. In: *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*. Vancouver, British Columbia, Canada : AAAI Press, 2007 (AAAI'07). – ISBN 978–1–57735–323–2, S. 1514–1519 (zitiert auf den Seiten 28, 33 und 62).

[AKM16] Azaria, Amos ; Krishnamurthy, Jayant ; Mitchell, Tom M.: Instructable Intelligent Personal Agent. (2016), S. 9 (zitiert auf den Seiten 28, 55 und 62).

[ALHM14] Araki, Jun ; Liu, Zhengzhong ; Hovy, Eduard H. ; Mitamura, Teruko: Detecting Subevent Structure for Event Coreference Resolution. In: *LREC*, 2014, S. 4553–4558 (zitiert auf den Seiten 30, 55 und 59).

[ARAS06] Asfour, T. ; Regenstein, K. ; Azad, P. ; Schröder, J.: ARMAR-III: An integrated humanoid platform for sensory-motor control. In: *Humanoid Robots, 2006 6th IEEE-RAS International Conference on* IEEE, 2006, S. 169–175 (zitiert auf den Seiten 23, 35, 39, 42 und 80).

[BEHW87] Blumer, Alselm ; Ehrenfeucht, Andrzej ; Haussler, David ; Warmuth, Manfred K.: Occam's Razor. In: *Inf. Process. Lett.* 24 (1987), Nr. 6, 377–380. http://dx.doi.org/10.1016/0020-0190(87)90114-1. – DOI 10.1016/0020–0190(87)90114–1. – ISSN 0020–0190 (zitiert auf Seite 111).

[BH16] Badgett, Allison ; Huang, Ruihong: Extracting Subevents via an Effective Two-Phase Approach. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas : Association for Computational Linguistics, November 2016, S. 906–911 (zitiert auf Seite 31).

[Bie83] Biermann, Sigmon Ballard: An experimental study of natural language programming. In: *International Journal of Man-Machine Studies* (1983), Nr. 18, 71-87. http://dx.doi.org/10.1016/S0020-7373(83)80005-4. – DOI 10.1016/S0020–7373(83)80005–4 (zitiert auf den Seiten 44 und 45).

[Bit18] Bitterwolf, Christian: *Bestimmung von Aktionsidentität in gesprochener Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, 2018. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/bitterwolf_ba (zitiert auf Seite 24).

[BLK+01] Bugmann, Guido ; Lauria, Stanislao ; Kyriacou, Theocharis ; Klein, Ewan ; Bos, Johan: Using Verbal Instructions for Route Learning: Instruction Analysis. In: *In Proc. of Torwards Intelligent Mobile Robots Conf*, 2001, S. 20–5 (zitiert auf den Seiten 27, 38 und 43).

[BP63] Bayes, Mr. ; Price, Mr.: An Essay towards Solving a Problem in the Doctrine of Chances. In: *Philosophical Transactions* 53 (1763), 370–418. http:

//dx.doi.org/10.1098/rstl.1763.0053. – DOI 10.1098/rstl.1763.0053 (zitiert auf Seite 15).

[Bri92] Brill, Eric: A simple rule-based part of speech tagger. In: *Proceedings of the third conference on Applied natural language processing* Association for Computational Linguistics, 1992, S. 152–155 (zitiert auf Seite 58).

[Bri94] Brill, Eric: Some Advances in Transformation-Based Part of Speech Tagging. In: *CoRR* abs/cmp-lg/9406010 (1994) (zitiert auf Seite 58).

[BZB10] Branavan, S. R. K. ; Zettlemoyer, Luke S. ; Barzilay, Regina: Reading between the Lines: Learning to Map High-Level Instructions to Commands. In: *ACL*, The Association for Computer Linguistics, 2010, S. 1268–1277 (zitiert auf Seite 31).

[Chr07] Christensen, Larry B.: *Experimental Methodology 10th ed.* http://lib.ugent.be/catalog/rug01:001036809. Version: 2007 (zitiert auf den Seiten 41, 45 und 51).

[Col02] Collins, Michael: Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In: *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2002 (EMNLP '02), 1–8 (zitiert auf den Seiten 40 und 71).

[CRF03] Cohen, William W. ; Ravikumar, Pradeep ; Fienberg, Stephen E.: A Comparison of String Distance Metrics for Name-matching Tasks. In: *Proceedings of the 2003 International Conference on Information Integration on the Web*, AAAI Press, 2003 (IIWEB'03), 73–78 (zitiert auf Seite 87).

[CRM07] Cohen, Shahar ; Rokach, Lior ; Maimon, Oded: Decision-tree Instance-space Decomposition with Grouped Gain-ratio. In: *Inf. Sci.* 177 (2007), September, Nr. 17, 3592–3612. http://dx.doi.org/10.1016/j.ins.2007.01.016. – DOI 10.1016/j.ins.2007.01.016. – ISSN 0020–0255 (zitiert auf Seite 57).

[CRX+17] Campagna, Giovanni ; Ramesh, Rakesh ; Xu, Silei ; Fischer, Michael ; Lam, Monica S.: Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In: *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. Perth, Australia : ACM Press, 2017. – ISBN 978–1–4503–4913–0, S. 341–350 (zitiert auf den Seiten 28, 30, 43 und 55).

[CW08] Collobert, Ronan ; Weston, Jason: A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In: *Proceedings of the 25th International Conference on Machine Learning*. New York, NY, USA : ACM, 2008 (ICML '08). – ISBN 978–1–60558–205–4, 160–167 (zitiert auf Seite 19).

[CWB+11] Collobert, Ronan ; Weston, Jason ; Bottou, Léon ; Karlen, Michael ; Kavukcuoglu, Koray ; Kuksa, Pavel: Natural Language Processing (Almost) from Scratch. In: *J. Mach. Learn. Res.* 12 (2011), November, 2493–2537. http://dl.acm.org/citation.cfm?id=1953048.2078186. – ISSN 1532–4435 (zitiert auf den Seiten 24, 59, 75, 77 und 123).

[DCLT18a] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *arXiv preprint arXiv:1810.04805* (2018) (zitiert auf den Seiten 19, 71, 75, 110 und 121).

[DCLT18b] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018 (zitiert auf Seite 40).

[Eic95] Eichinger, Ludwig M.: *Hadumod Bußmann: Lexikon der Sprachwissenschaft, Stuttgart, Kröner 1990, 904 S., (Kröners Taschenausgabe ; 452).* 1995 (zitiert auf den Seiten 3 und 4).

[Fir68] Firth, John G.: A Synopsis of Linguistic Theory 1930-1955ïn Studies in Linguistic Analysis, 1968 (zitiert auf Seite 20).

[Gü15] Günes, Zeynep: *Aufbau eines Sprachkorpus zur Programmierung autonomer Roboter mittels natürlicher Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, 2015. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/guenes_ba (zitiert auf den Seiten 23 und 39).

[GBC16] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org (zitiert auf den Seiten 12, 14, 16 und 18).

[GH17] Goldberg, Yoav ; Hirst, Graeme: *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017. – ISBN 1627052984, 9781627052986 (zitiert auf Seite 69).

[GM14] Gulwani, Sumit ; Marron, Mark: NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data - SIGMOD '14*. Snowbird, Utah, USA : ACM Press, 2014. – ISBN 978–1–4503–2376–5, S. 803–814 (zitiert auf den Seiten 26 und 55).

[GP96] Green, Thomas ; Petre, Marian: Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. In: *Journal of Visual Languages* 7 (1996), 06, S. 131–. http://dx.doi.org/10.1006/jvlc.1996.0009. – DOI 10.1006/jvlc.1996.0009 (zitiert auf Seite 175).

[Gra13] Graves, Alex: Generating Sequences With Recurrent Neural Networks. In: *arXiv:1308.0850 [cs]* (2013), August (zitiert auf Seite 117).

[GRG+13] Guadarrama, Sergio ; Riano, Lorenzo ; Golland, Dave ; Go, Daniel ; Jia, Yangqing ; Klein, Dan ; Abbeel, Pieter ; Darrell, Trevor u. a.: Grounding spatial relations for human-robot interaction. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* IEEE, 2013, S. 1640–1647 (zitiert auf Seite 62).

[Gri75] Grice, H. P.: Logic and Conversation. Version: 1975. http://www.ucl.ac.uk/ls/studypacks/Grice-Logic.pdf. In: Cole, Peter (Hrsg.) ; Morgan, Jerry L. (Hrsg.): *Syntax and Semantics: Vol. 3: Speech Acts.* New York : Academic Press, 1975, 41-58 (zitiert auf Seite 175).

[GSC00] Gers, Felix ; Schmidhuber, Jürgen ; Cummins, Fred: Learning to Forget: Continual Prediction with LSTM. In: *Neural computation* 12 (2000), 10, S. 2451–71. http://dx.doi.org/10.1162/089976600300015015. – DOI 10.1162/089976600300015015 (zitiert auf den Seiten 56, 59 und 119).

[Ham18] Hamann, Daniel: *Vollständigkeits- und Semantiküberprüfung für gesprochene Aussagen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, August 2018. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/hamann_ba (zitiert auf Seite 24).

[Hey16] Hey, Tobias: *Kontext- und Korreferenzanalyse für gesprochene Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master's Thesis, 2016. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/hey_ma (zitiert auf den Seiten 24, 60, 61 und 87).

[Hoc18] Hochweiss, Robert: *Analyse und Korrektur von Disfluenzen in gesprochener Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, Mai 2018. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/hochweiss_ba (zitiert auf Seite 24).

[HS97] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-Term Memory. In: *Neural Comput.* 9 (1997), November, Nr. 8, 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735. – DOI 10.1162/neco.1997.9.8.1735. – ISSN 0899–7667 (zitiert auf den Seiten 18, 56, 59 und 119).

[JM00] Jurafsky, Daniel ; Martin, James H.: *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2000. – ISBN 0130950696 (zitiert auf Seite 37).

[JWHT14] James, Gareth ; Witten, Daniela ; Hastie, Trevor ; Tibshirani, Robert: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. – ISBN 1461471370, 9781461471370 (zitiert auf Seite 71).

[KB14] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *CoRR* abs/1412.6980 (2014). http://arxiv.org/abs/1412.6980 (zitiert auf Seite 103).

[Kei18] Keim, Jan: *Themenextraktion zur Domänenauswahl für Programmierung in natürlicher Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Diplomarbeit, 2018. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/keim_ma (zitiert auf Seite 24).

[Kie16] Kiesel, Viktor: *Optimierung von POS-Tagger-Ergebnissen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, April 2016. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/kiesel_ba (zitiert auf Seite 24).

[KLM96] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W.: Reinforcement learning: A survey. In: *Journal of artificial intelligence research* 4 (1996), S. 237–285 (zitiert auf Seite 38).

[KMSLS15] Kumar Misra, Dipendra ; Sung, Jaeyong ; Lee, Kevin ; Saxena, Ashutosh: Tell Me Dave: Context-Sensitive Grounding of Natural Language to Manipulation Instructions. In: *The International Journal of Robotics Research* 35 (2015), 11. http://dx.doi.org/10.1177/0278364915602060. – DOI 10.1177/0278364915602060 (zitiert auf Seite 89).

[Koc15] Kocybik, Markus: *Projektion von gesprochener Sprache auf eine Handlungsrepräsentation*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, Juli 2015. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/kocybik_ba (zitiert auf den Seiten 23 und 123).

[Kup92] Kupiec, Julian: Robust part-of-speech tagging using a hidden Markov model. In: *Computer Speech and Language* 6 (1992), Nr. 3, 225 - 242. http://dx.doi.org/https://doi.org/10.1016/0885-2308(92)90019-Z. – DOI https://doi.org/10.1016/0885–2308(92)90019–Z. – ISSN 0885–2308 (zitiert auf Seite 59).

[LBK+01] Lauria, Stanislao ; Bugmann, Guido ; Kyriacou, Theocharis ; Bos, Johan ; Klein, Ewan: Personal Robot Training via Natural-Language Instructions. In: *IEEE Intelligent Systems* 16 (2001), S. 38–45 (zitiert auf Seite 27).

[LBKK02] Lauria, Stanislao ; Bugmann, Guido ; Kyriacou, Theocharis ; Klein, Ewan: Mobile Robot Programming Using Natural Language. In: *Robotics and Autonomous Systems* 38 (2002), März, Nr. 3-4, S. 171–181. `http://dx.doi.org/10.1016/S0921-8890(02)00166-5`. – DOI 10.1016/S0921–8890(02)00166–5. – ISSN 09218890 (zitiert auf den Seiten 27, 38, 55 und 62).

[LGS13] Le, Vu ; Gulwani, Sumit ; Su, Zhendong: SmartSynth: synthesizing smartphone automation scripts from natural language. In: *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, 2013, 193–206 (zitiert auf den Seiten 26, 55 und 62).

[LL99] L. Lohr, Sharon: Sampling: Design and Analysis, 1999 (zitiert auf Seite 71).

[LL05] Liu, Hugo ; Lieberman, Henry: Metafor: Visualizing Stories As Code. In: *Proceedings of the 10th International Conference on Intelligent User Interfaces*, 2005 (IUI '05). – ISBN 1–58113–894–6, 305–307 (zitiert auf den Seiten 25 und 55).

[LMP01] Lafferty, John ; McCallum, Andrew ; Pereira, Fernando C.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001) (zitiert auf Seite 59).

[LWT17] Landhäusser, Mathias ; Weigelt, Sebastian ; Tichy, Walter F.: NLCI: A Natural Language Command Interpreter. In: *Automated Software Engineering* 24 (2017), Dezember, Nr. 4, S. 839–861. `http://dx.doi.org/10.1007/s10515-016-0202-1`. – DOI 10.1007/s10515–016–0202–1. – ISSN 0928–8910, 1573–7535 (zitiert auf den Seiten 32 und 62).

[Man11] Manning, Christopher D.: Part-of-speech Tagging from 97Linguistics? In: *Proceedings of the 12th International Conference on Computational Linguistics and Intelligent Text Processing - Volume Part I.* Berlin, Heidelberg : Springer-Verlag, 2011 (CICLing'11). – ISBN 978–3–642–19399–6, 171–189 (zitiert auf den Seiten 40, 56, 75 und 119).

[MBW15] Mei, Hongyuan ; Bansal, Mohit ; Walter, Matthew R.: Listen, Attend, and Walk: Neural Mapping of Navigational Instructions to Action Sequences. In: *arXiv:1506.04089 [cs]* (2015), Juni (zitiert auf den Seiten 32 und 62).

[MCCD13] Mikolov, Tomas ; Corrado, G.s ; Chen, Kai ; Dean, Jeffrey: Efficient Estimation of Word Representations in Vector Space, 2013, S. 1–12 (zitiert auf den Seiten 20 und 98).

[MHZF12] Matuszek, Cynthia ; Herbst, Evan ; Zettlemoyer, Luke S. ; Fox, Dieter: Learning to Parse Natural Language Commands to a Robot Control System. In: *ISER*, 2012 (zitiert auf Seite 62).

[Mih07] Mihalcea, R.: Using wikipedia for automatic word sense disambiguation. In: *Proceedings of NAACL HLT* Bd. 2007, 2007, S. 196–203 (zitiert auf den Seiten 66, 69 und 72).

[Mil77] Miller, Lance: Natural Language Programming. In: *SIGART Bull.* (1977), 02, Nr. 61, 32–32. `http://dx.doi.org/10.1145/1045283.1045299`. – DOI

10.1145/1045283.1045299. – ISSN 0163–5719 (zitiert auf den Seiten 44 und 45).

[Mit97] MITCHELL, Thomas M.: *Machine Learning*. 1. New York, NY, USA : McGraw-Hill, Inc., 1997. – ISBN 0070428077, 9780070428072 (zitiert auf den Seiten 12 und 13).

[MMS93] MARCUS, Mitchell P. ; MARCINKIEWICZ, Mary A. ; SANTORINI, Beatrice: Building a Large Annotated Corpus of English: The Penn Treebank. In: *Comput. Linguist.* 19 (1993), Juni, Nr. 2, 313–330. http://dl.acm.org/citation.cfm?id=972470.972475. – ISSN 0891–2017 (zitiert auf den Seiten 6 und 40).

[MN96] MAHESH, Kavi ; NIRENBURG, Sergei: A Situated Ontology for Practical NLP. In: *Workshop On Basic Ontological Issues in Knowledge Sharing* (1996), 11 (zitiert auf Seite 21).

[MR98] MÀRQUEZ, Lluís ; RODRÍGUEZ, Horacio: Part-of-speech tagging using decision trees. In: *European Conference on Machine Learning* Springer, 1998, S. 25–36 (zitiert auf Seite 59).

[MWCM14] MALMAUD, Jonathan ; WAGNER, Earl ; CHANG, Nancy ; MURPHY, Kevin: Cooking with Semantics, 2014, S. 33–38 (zitiert auf Seite 89).

[Ou16] OU, Yue: *Erkennung von Aktionen in gesprochener Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, September 2016. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/ou_ba (zitiert auf den Seiten 24 und 60).

[Pas15] PASKARAN, Dinesh: *Evaluation unterschiedlicher Spracherkennungssysteme in der Domäne Humanoide Robotik*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, November 2015. https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/paskaran_ba (zitiert auf Seite 23).

[Pre97] PRECHELT, Lutz: Early Stopping - but when? In: *Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2*, Springer-Verlag, 1997, S. 55–69 (zitiert auf Seite 107).

[PRM01] PANE, John F. ; RATANAMAHATANA, Chotirat '. ; MYERS, Brad A.: Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. In: *International Journal of Human-Computer Studies* 54 (2001), 02, Nr. 2, S. 237–264. http://dx.doi.org/10.1006/ijhc.2000.0410. – DOI 10.1006/ijhc.2000.0410. – ISSN 10715819 (zitiert auf den Seiten 43, 45 und 49).

[Rab90] RABINER, Lawrence R.: Readings in Speech Recognition. Version: 1990. http://dl.acm.org/citation.cfm?id=108235.108253. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1990. – ISBN 1–55860–124–4, Kapitel A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, 267–296 (zitiert auf Seite 59).

[RHW86] RUMELHART, D. E. ; HINTON, G. E. ; WILLIAMS, R. J.: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. Version: 1986. http://dl.acm.org/citation.cfm?id=104279.104293. Cambridge, MA, USA : MIT Press, 1986. – ISBN 0–262–68053–X, Kapitel Learning Internal Representations by Error Propagation, 318–362 (zitiert auf Seite 18).

[RML06] Rada Mihalcea, Hugo L. ; Lieberman, Henry: NLP (Natural Language Processing) for NLP (Natural Language Programming). In: *Computational Linguistics and Intelligent Text Processing, 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006, Proceedings*, 2006, 319–330 (zitiert auf Seite 25).

[RN09] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach.* 3rd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2009. – ISBN 0136042597, 9780136042594 (zitiert auf Seite 71).

[RR09] Ratinov, Lev ; Roth, Dan: Design Challenges and Misconceptions in Named Entity Recognition. In: *Proceedings of the Thirteenth Conference on Computational Natural Language Learning.* Stroudsburg, PA, USA : Association for Computational Linguistics, 2009 (CoNLL '09). – ISBN 978–1–932432–29–9, 147–155 (zitiert auf Seite 40).

[RSK17] Rabinovich, Maxim ; Stern, Mitchell ; Klein, Dan: Abstract Syntax Networks for Code Generation and Semantic Parsing. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 2017, S. 1139–1149 (zitiert auf den Seiten 29, 55 und 119).

[SA18] Suhr, Alane ; Artzi, Yoav: Situated Mapping of Sequential Instructions to Actions with Single-Step Reward Observation. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Melbourne, Australia : Association for Computational Linguistics, Juli 2018, S. 2072–2082 (zitiert auf den Seiten 32 und 62).

[Sal89] Salton, Gerard: *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1989. – ISBN 0–201–12227–8 (zitiert auf Seite 19).

[SCC+14] She, Lanbo ; Cheng, Yu ; Chai, Joyce Y. ; Jia, Yunyi ; Yang, Shaohua ; Xi, Ning: Teaching Robots New Actions through Natural Language Instructions. In: *The 23rd IEEE International Symposium on Robot and Human Interactive Communication.* Edinburgh, UK : IEEE, August 2014. – ISBN 978–1–4799–6765–0 978–1–4799–6763–6, S. 868–873 (zitiert auf den Seiten 28, 55 und 62).

[Sch15] Schneider, Michael: *Entwurf einer Handlungsrepräsentation für gesprochene Sprache*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, Mai 2015. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/schneider_ba` (zitiert auf Seite 23).

[Sch16] Scheu, Sven: *Aufbereitung von Spracherkennerausgaben*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master's Thesis, Juli 2016. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/scheu_ma` (zitiert auf Seite 23).

[Sch17] Schlereth, Mario: *Entwicklung eines Dialogagenten für dialogbasierte Programmierung*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Master's Thesis, 2017. `https://code.ipd.kit.edu/weigelt/parse/wikis/Theses/schlereth_ma` (zitiert auf Seite 24).

[SH16] Shen, Yatian ; Huang, Xuanjing: Attention-Based Convolutional Neural Network for Semantic Relation Extraction. In: *Proceedings of COLING 2016,*

the 26th International Conference on Computational Linguistics: Technical Papers. Osaka, Japan : The COLING 2016 Organizing Committee, Dezember 2016, 2526–2536 (zitiert auf den Seiten 40 und 75).

[SP97] Schuster, M. ; Paliwal, K.K.: Bidirectional Recurrent Neural Networks. In: *Trans. Sig. Proc.* 45 (1997), November, Nr. 11, 2673–2681. http://dx.doi.org/10.1109/78.650093. – DOI 10.1109/78.650093. – ISSN 1053–587X (zitiert auf Seite 18).

[TKD+11] Tellex, Stefanie ; Kollar, Thomas ; Dickerson, Steven ; Walter, Matthew R. ; Banerjee, Ashis G. ; Teller, Seth J. ; Roy, Nicholas: Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In: *AAAI*, 2011 (zitiert auf den Seiten 32 und 43).

[TKMS03] Toutanova, Kristina ; Klein, Dan ; Manning, Christopher D. ; Singer, Yoram: Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2003 (NAACL '03), 173–180 (zitiert auf Seite 24).

[VAPM14] Valverde-Albacete, Francisco J. ; Peláez-Moreno, Carmen: 100Information Transfer Factor Explains the Accuracy Paradox. In: *PLOS ONE* 9 (2014), 01, Nr. 1, 1-10. http://dx.doi.org/10.1371/journal.pone.0084217. – DOI 10.1371/journal.pone.0084217 (zitiert auf Seite 115).

[VJ10] Vogel, Adam ; Jurafsky, Dan: Learning to Follow Navigational Directions. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2010 (ACL '10), 806–814 (zitiert auf Seite 31).

[WAKS97] Wang, Daniel C. ; Appel, Andrew W. ; Korn, Jeff L. ; Serra, Christopher S.: The Zephyr Abstract Syntax Description Language. In: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. Berkeley, CA, USA : USENIX Association, 1997 (DSL'97), 17–17 (zitiert auf Seite 29).

[WGLM17] Wang, Sida I. ; Ginn, Samuel ; Liang, Percy ; Manning, Christopher D.: Naturalizing a Programming Language via Interactive Learning, Association for Computational Linguistics, 2017, S. 929–938 (zitiert auf den Seiten 29 und 55).

[WHS18a] Weigelt, S. ; Hey, T. ; Steurer, V.: Detection of Conditionals in Spoken Utterances. In: *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, 2018, S. 85–92 (zitiert auf den Seiten 24 und 63).

[WHS18b] Weigelt, Sebastian ; Hey, Tobias ; Steurer, Vanessa: Detection of Control Structures in Spoken Utterances. In: *International journal of semantic computing* 12 (2018), Nr. 3, S. 335–360. http://dx.doi.org/10.1142/S1793351X18400159. – DOI 10.1142/S1793351X18400159. – ISSN 1793–351X, 1793–7108 (zitiert auf den Seiten 24 und 63).

[Wie14] Wiest, Fabian: *Interaktive Mustergenerierung als Funktionserweiterung des natürlichsprachlichen Dialogsystems JustLingo*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, 2014. https://ps.ipd.kit.edu/176_241.php (zitiert auf den Seiten 27 und 55).

[WM38] W. MORRIS, Charles: *Foundations of the Theory of Signs.* Chicago : University of Chicago Press, 1938. – ISBN 978–0226575773 (zitiert auf Seite 4).

[WM97] WOLPERT, D. H. ; MACREADY, W. G.: No Free Lunch Theorems for Optimization. In: *Trans. Evol. Comp* 1 (1997), Nr. 1, 67–82. `http://dx.doi.org/10.1109/4235.585893`. – DOI 10.1109/4235.585893. – ISSN 1089–778X (zitiert auf Seite 59).

[WT15] WEIGELT, S. ; TICHY, W.F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on* Bd. 2, 2015, S. 819–820. – `http://ps.ipd.kit.edu/163_422.php` (zitiert auf Seite 23).

[WWVT15] WACHTEL, Alexander ; WEIGELT, Sebastian ; VOIGT, Philipp ; TICHY, Walter F.: Prototyp einer natürlichsprachlichen Schnittstelle für Tabellenkalkulation. In: *Multikonferenz Software Engineering & Management 2015 : 1Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015 in Dresden, Germany. Hrsg.: U. Aßmann* Bd. 239, Gesellschaft für Informatik (GI), Bonn, 2015 (GI-Edition : lecture notes in informatics). – ISBN 978–3–88579–633–6, S. 1617–5468 (zitiert auf den Seiten 27, 28 und 55).

[ZC07] ZETTLEMOYER, Luke S. ; COLLINS, Michael: Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In: *EMNLP-CoNLL*, 2007 (zitiert auf Seite 62).

Appendix

A Nutzerstudie

Der Sprachkorpus des PARSE-Projektes wurde um sechs weitere Szenarien ergänzt. Diese wurden dahingehend entworfen, verabliserte Funktionserweiterungen in den Nutzereingaben hervorzurufen.

A.1 Szenarien

Die Szenarien 1 bis 4 wurden hierbei für die Vorstudie dieser Arbeit konzipiert. Szenarien 5 und 6 wurden für die Ende-zu-Ende-Evaluation erstellt. In Szenario 0 werden alle bereits verfügbaren Nutzereingaben des PARSE-Korpus gesammelt.

Tabelle A.1: Szenarienindex der Vorstudie und Evaluationsstudie

Nummer	Szenario	Bescheibung der neuen Aufgabe
0	PARSE	Alle bestehenden Texte aus dem PARSE-Korpus
1	Greet Someone	Begrüßung von Personen
2	Prepare Coffee	Zubereiten eines Kaffes mit der Kaffemaschine
3	Bring some Beverage	Einschenken und Servieren eines Getränks
4	Set the Table for Two	Decken des Tischs für zwei Personen
5	Start the Dishwasher	Schließen und Einschalten der Spülmaschine
6	Prepare Cereals	Zubereiten eines Müslis aus Zutaten

A.2 Auszug aus den erhobenen Nutzereingaben

Im Folgenden wird ein kleiner Auszug aus den erhobenen Nutzereingaben der Vorstudie präsentiert. Diese werden in valide Nutzereingaben sowie in invalide Nutzereingaben untergliedert. Letztere wurden für das Training der Klassifikatoren nicht in Betracht gezogen.

A.2.1 Valide Nutzereingaben

„Hi Armar, were going to lane how to set the table for two people to eat“

„Hi Armar, starting the coffee machine means that you take the cup next to the machine and place them under the dispenser. Once you have done that, you press the red button.“

„To greet a person you look him or her in the eyes and then wave robot hand and say 'hello'“

„Hey Armar, look at our new guest! You can greet him by saying hello looking him in the eyes and waving your hand.“

„a) Looking someone in the eyes means to look directly into their eyes. b) To wave, you need to raise your hand and then rotate the hand side to side for a few seconds before stopping c)After putting your hand back down you need to say hello while still looking into their eyes“

„Hi Amar, to set the table for 2 you must get 2 plates, 2 glasses, 2 knives and 2 forks out of the cupboard and then take them to the kitchen table“

„put cup into the machine, and press the red button ... That's how we make a drink”“

„Look at the person in front of you. Extend one of your hands towards the person. Say 'Hello' to that person.“

A.2.2 Abgelehnte Nutzereingaben

„I will tell him and show him“

„first show how to open the fridge and point out where certain items do in the fridge and also what the shelves/containers are used for and what goes in them“

„Clean fridge“

„Hello metal ass, what can you do for me? Pretend you have cellotape stuck to your hand, smile and say HELLO!“

„Hi, how are you? give me your hand“

„Bring the bank from a fridge. Open a bank's top bu clockwise. Pour it inside the glass. And trasfer it to user“

„hey armar, my old friend... some romantic dinner tonight! I've meet a supernice girl with incredible curves so please get out of here, we want to get alone, but before that i really appreciate if you set the table with 2 plates, 2 glasses, 2 knives, 2 forks and maybe some nice candle (told you, it's a tete-a-tete, you know what i mean! and i know you have a crush for her...).“

B Konfigurationsdatei der Implementierung

Die Konfigurationsdatei für die Synthese von Methodendefinitionen in natürlichsprachlichen Äußerungen enthält alle für den implementierten `MethodSynthesizer`-Agenten einstellbaren Parameter.

```
1  ############## binary classification (step 1) ##############
2  # defines the tokenizer for the models, trained on whole vocab
3  VOCAB=models/tok_full_vocab_tokenized.json
4
5  # defines the model used for binary classification: with or without
       pretrained fasttext word vectors
6  # binary_RNN4_oWv_tok_fulldata.best.hdf5  OR
       binary_RNN4_fasttextWv_tok_fulldata.best.hdf5
7  BIN_MODEL=models/binary_RNN4_fasttextWv_tok_fulldata.best.hdf5
8
9  ############## multiclass classification (step 2) ##############
10 # defines the model used for multiclass classification: with or
       without pretrained fasttext word vectors
```

```
11 # mclass_RNN1_oWv_tok_full.best.hdf5  OR
       mclass_RNN1_fasttextWv_tok_full.best.hdf5
12 MCLASS_MODEL=models/mclass_RNN1_oWv_tok_full.best.hdf5
13
14 # use prediction per DL4J (true) OR per python script (false)
15 USE_INTERAL_MCLASS_MODEL=false
16
17 ############## ontology mapper ##############
18 # return topN function call candidates
19 TOP_N=10
20
21 # defines the stopwords and modal verbs to skip for function and
       parameter names
22 STOP_WORDS=armar, learn, how, you, that, this, teach, explain, thank
       , would, need, have, must, like, can, should, want, means, mean,
       next, then, show, do, now
23
24 # defines pos tags for extracting parameter names
25 PARAMETER_POS=CD, JJ, JJR, JJS, NN, NNS, NNP, NNPS, PDT, PP, PP$, RB
       , RB, RBR, RBS, RP
```

Quelltextausschnitt 12.1: Konfigurationsdatei des Agenten

Anhang Anhang