Automatisierte Erkennung von Intents und Entities eines domänenspezifischen Befehls in menschlicher Sprache

Roland Berneder

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

© Copyright 2017 Roland Berneder

Diese Arbeit wird unter den Bedingungen der Creative Commons Lizenz *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0) veröffentlicht – siehe https://creativecommons.org/licenses/by-nc-nd/4.0/.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 26. Juni 2017

Roland Berneder

Inhaltsverzeichnis

Erklärung iii

Kurzfassung vii

Abstract viii

1 Einleitung 1
- 1.1 Problemstellung und Zielsetzung . . . 2
- 1.2 Gliederung der Arbeit . . . 3

2 State of the Art 4
- 2.1 Machine Learning . . . 4
  - 2.1.1 Einteilung . . . 5
  - 2.1.2 Terminologie . . . 6
  - 2.1.3 Supervised Machine Learning Algorithmen . . . 8
  - 2.1.4 Training und Evaluierung . . . 15
- 2.2 Natural Language Processing . . . 17
  - 2.2.1 Einsatz und Funktionsweise . . . 17
  - 2.2.2 Terminologie . . . 19
  - 2.2.3 Textaufbereitung . . . 21
  - 2.2.4 Feature Extraction . . . 23
- 2.3 Natural Language Processing Services . . . 26
  - 2.3.1 Microsoft LUIS . . . 26
  - 2.3.2 API.AI . . . 28
  - 2.3.3 WIT . . . 31
  - 2.3.4 ALEXA . . . 33
- 2.4 Aufgabenstellung . . . 34
- 2.5 Programmiersprache . . . 36

3 Testumgebung 37
- 3.1 Testdaten . . . 37
  - 3.1.1 Sprache . . . 37
  - 3.1.2 Szenario . . . 37
  - 3.1.3 Trainings- und Testkorpus . . . 39
- 3.2 Leistungsbeurteilung . . . 40
- 3.3 Testapplikation . . . 41

3.3.1 Technologie . . . 41
3.3.2 Format der Testbefehle . . . 42
3.3.3 Anbindung der NLPS . . . 42
3.3.4 Initialisierung . . . 43
3.3.5 Auswertung der Leistung . . . 43

4 Implementierung **45**
4.1 Prototypstruktur . . . 45
4.2 Aufbereitung der Trainingsdaten . . . 45
4.3 Textverarbeitung . . . 47
4.3.1 Wahl des ML-Algorithmus . . . 47
4.3.2 Wahl des Lemmatisierers . . . 48
4.3.3 Initialisierung . . . 50
4.3.4 Erkennung des Intents . . . 51
4.3.5 Entity Extraction . . . 52
4.4 Webserver . . . 54
4.5 Verwendung . . . 54

5 Testauswertung **56**
5.1 Trainingsdaten . . . 56
5.1.1 Intents . . . 56
5.1.2 Entities . . . 57
5.2 Testdaten . . . 57
5.2.1 Einfache Befehle . . . 58
5.2.2 Eingabequalität . . . 58
5.2.3 Komplexe Befehle . . . 58
5.2.4 Unbekannte Intents . . . 58
5.2.5 Synonyme, Listen und Wortvariationen . . . 59
5.2.6 Zahlen . . . 59
5.3 Gesamtleistung . . . 59

6 Resümee **62**
6.1 Resultat . . . 62
6.2 Ausblick . . . 63

A Übersicht der Machine Learning Decision Borders **64**

B Installationsanleitung **66**
B.1 NLPS-Prototyp . . . 66
B.2 Testapplikation . . . 66

C Trainingsdaten **67**

D Inhalt der CD-ROM **69**
D.1 Masterarbeit . . . 69
D.2 Literatur . . . 69
D.3 Quellcode . . . 69

D.4 Konfigurationsdaten der verwendeten NLPS 70
D.5 Testergebnisse 70

Quellenverzeichnis **71**
Literatur 71
Online-Quellen 73

Kurzfassung

In den letzten Jahren etablierten sich einige *Natural Language Processing Services*, deren Zweck es ist, die Kommunikation zwischen Menschen und digitalen Systemen zu vereinfachen. Ihre Aufgabe ist es, in menschlicher Sprache geschriebene Befehle zu interpretieren und in eine computerverständliche Form zu bringen. Dabei gibt es die Konvention, entsprechende Charakteristiken aus dem Befehl zu extrahieren. Zuerst wird die Domäne, beispielsweise „Pizzaservice“, ausgemacht. Als nächstes werden die Absicht, wie z. B. „bestellen“, und abschließend in dem Zusammenhang die Parameter, wie „eine“ und „Margherita“, erkannt. Einige dieser Systeme werden im Zuge dieser Arbeit auf ihre Bedienung und Nutzbarkeit in einem praxisnahen Testszenario getestet und deren Qualität und Leistung evaluiert. Außerdem wird ein eigener, domänenspezifischer Service implementiert und dabei die verwendeten Technologien im Rahmen der semantischen Texterkennung erläutert. Dieser wird den anderen Systemen in einer für diesen Zweck angefertigten Testapplikation gegenübergestellt. Schließlich erfolgt ein Resümee, in dem die Ergebnisse dargelegt werden und auf die Vorteile und Relevanz eines solchen, domänenspezifischen Spezialsystems eingegangen wird.

Abstract

Several *Natural Language Processing Services* were established in the last couple of years. They have been invented with the objective to ease communication between humans and digital systems. They translate orders, written in natural language, to a form computers can work with. For this, conventions have been introduced to extract certain characteristics. Initially the domain of the command is recognized, for example "pizza-service". Next the intent, for instance "order" and finally the corresponding parameters, like "one" and "margherita", are extracted. In the course of this thesis, the usability of these services is tested via a realistic scenario and their quality and performance are then evaluated. In addition to that a separate, domain specific service is implemented. The technology required for its semantic text interpretation is described in detail. This prototype is then compared to the other services via a test application, specifically developed for this purpose. Finally a conclusion is drawn where the results are discussed and the benefits and relevance of such a custom, domain specific system is evaluated.

Kapitel 1

Einleitung

Mit der Erfindung des Computers ging die Problemstellung einher, eine Möglichkeit zu finden, mit diesem Gerät kommunizieren zu können. Es liegt auf der Hand, dass Benutzereingaben möglichst einfach und intuitiv sein sollen. Daher wurde neben Peripheriegeräten wie Maus und Tastatur, auch nach benutzerfreundlicheren Eingabemöglichkeiten wie Touchscreens und letztlich der Text- und Spracherkennung gesucht.

Obwohl die Technologie der Spracherkennung erst in den letzten Jahren durch bekannte Vertreter wie *Siri*, *Alexa* oder *Now* immer mehr praktisch eingesetzt wurde, reichen die Wurzeln der Entwicklungen weit zurück bis in die fünfziger Jahre, wo Alan Turing mit seiner Idee, dem *Imitation Game* [27], den Grundstein für künstliche Intelligenz gelegt, und der Vision, sich auf menschliche Weise mit einem Computer unterhalten zu können, ihre Form gegeben hat. Gedämpft durch die nur spärlich vorhandenen Ressourcen, setzte sich die Entwicklung dieser sehr interdisziplinären, sowie rechen- und speicherintensiven Technologie nur träge in Gang. Mit der Verwendung von Hidden Markov Models 1975, durch James Baker, der zunehmenden Leistungsfähigkeit der Computer sowie durch das stetige Verbessern der digitalen Abbildung der Grammatik menschlicher Sprache, entstanden die ersten brauchbaren Programme, mit denen Spracherkennung möglich war [3]. Firmen und Produkte wie *Dragon Systems* und *HTK* sowie Open-Source-Projekte wie *CMU Sphinx* entstanden, die seither den Mechanismus der Spracherkennung weiterentwickeln.

Der typische Ablauf, wie Sprachbefehle in einem solchen System verarbeitet werden, wird in Abbildung 1.1 veranschaulicht. Der Sprachbefehl wird vom Kommunikationsgerät über das Mikrophon aufgenommen und dem *Speech-To-Text-Interpreter* übergeben. Dort wird das Audiosignal verarbeitet, in textuelle Form gebracht und der *Textinterpretation* übergeben, in der die Semantik und die Parameter des Befehls extrahiert, und in eine maschinenverständliche Form gebracht werden. Ein *Dialogmanager* verwertet diese Information und stellt zusammen mit der *Diskursanalyse* und der *Dialogstrategie* den Kontext der Unterhaltung her, die unter Umständen mehrstufig, also über mehrere Satzwechsel hinweg, verlaufen kann. Die *Anwendung*, die an das Sprachsteuerungssystem gekoppelt ist, liefert jene Informationen, die für die Erzeugung der Antwort über den *Dialogmanager* in der *Texterstellung* benötigt wird. Abschließend wird durch die *Text-To-Speech-Einheit* aus der textuellen Antwort ein Audiosignal erzeugt, das vom *Kommunikationsgerät* an den Anwender ausgegeben wird.

Durch die zunehmende Leistungsfähigkeit der Desktop- und Mobilgeräte in den ver-

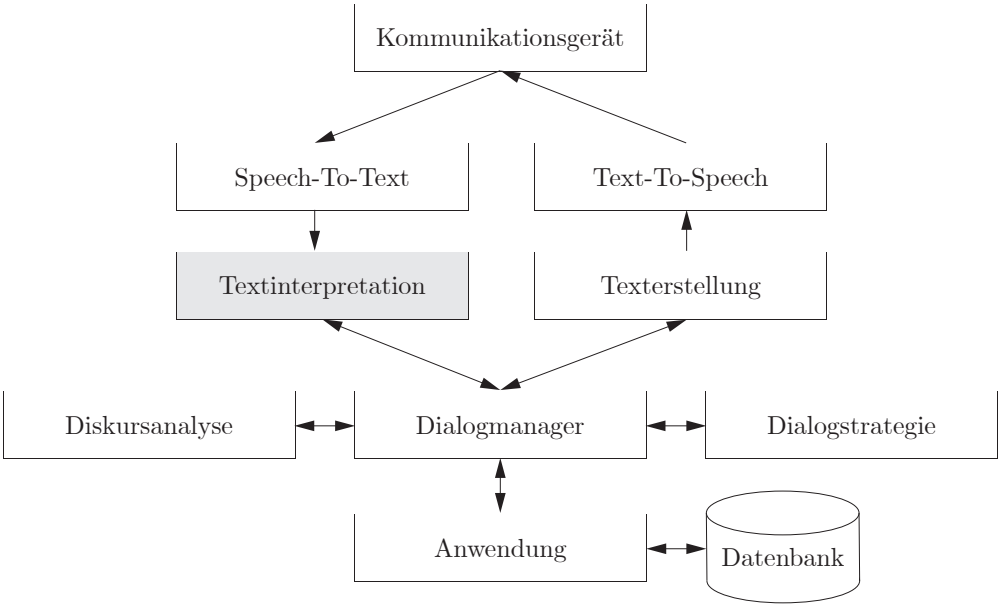


Abbildung 1.1: Typischer Ablauf einer Sprachsteuerungssoftware [12].

gangenen Jahren, erhielt die Technologie der Sprachsteuerung im Sektor der Endverbraucher ihren Einzug [13]. Mit steigender Akzeptanz der Benutzer und immer umfangreicher werdenden Funktionalitäten, breitet sich das Konzept der Interaktion mittels menschlicher Sprache auch auf soziale Medien und Onlineapplikationen aus. Gerade gewinnen sogenannte *Messenger Bots* stetig an Bedeutung. Diese automatisierten Sprach- und Textservices, auch *Natural Language Processing Services* (NLPS) genannt, integriert in soziale Kommunikationsplattformen oder in Onlineapplikationen, geben Auskünfte über allgemeine Inhalte, wie Wetter oder Aktien, oder sind an benutzerdefinierte Dienste, wie etwa Bestellservices für Essen, gekoppelt.

Diese Arbeit soll sich mit der Genauigkeit dieser NLPS auseinandersetzen und verfügbare Onlinedienste mit einer eigenen Implementierung basierend auf einem konkreten Anwendungsfall vergleichen. Der Rahmen der Arbeit beschränkt sich dabei auf den, in Abbildung 1.1 grau hinterlegten, Bereich *Textinterpretation* des Gebietes der Sprach- bzw. Textsteuerung.

1.1 Problemstellung und Zielsetzung

Anders als beim *Data-Mining* großer Mengen an Textdaten sind NLPS für Sprach- bzw. Textsteuerungen an wenige Wörter bzw. Sätze gebunden, was ein erhebliches Maß an Genauigkeit des Systems erfordert, um den eingegebenen Befehl korrekt interpretieren zu können.

So wie in menschlicher Sprache derselbe Inhalt mannigfaltig ausgedrückt werden kann, ist es auch auf der Ebene der Maschinensprache möglich, den extrahierten Inhalt auf verschiedene Arten zu übersetzen. Da diese Übersetzung von der, an den NLPS an-

gebundenen Anwendung verstanden werden muss, haben sich Konventionen etabliert, die im Kapitel 2 erklärt werden. Neben der Form, wie zwischen den jeweiligen Instanzen kommuniziert wird, sind auch die ausgetauschten Parameter von entscheidender Bedeutung. Beispielsweise wären die Begriffe „Paradeiser“ und „Tomate“ für Menschen zwar synonym, jedoch würde das bei Datenbankabfragen, sofern diese Spezialfälle dort nicht berücksichtigt werden, zu unzufriedenstellenden Ergebnissen führen. Auch die Komplexität des Satzaufbaus kann den NLPS vor eine Herausforderung stellen. Ein Beispiel wäre folgender Satz, der zwei Ortsnamen enthält, was zu Fehlinterpretationen führen kann: „Da es hier in Linz schneit, frage ich mich, wie das Wetter in Hagenberg ist.“

Es ist also leicht vorstellbar, dass in dieser Übersetzungskette zahlreiche potentielle Fehlerquellen lauern, die zu einem falschen Ergebnis führen können. Das Ziel dieser Arbeit ist, diese Übersetzungsgenauigkeit vorhandener, allgemein verwendbarer NLPS zu verifizieren, einen Prototyp eines eigenen, anwendungsspezifischen NLPS zu implementieren, eine Testapplikation zu schaffen, in der die Leistungen der angebundenen Services ausgewertet und verglichen werden können um zu prüfen, ob eine eigene Implementierung für einen speziellen Anwendungsfall eine bessere Leistungen erzielt, als allgemein verwendbare.

1.2 Gliederung der Arbeit

Zunächst wird im Kapitel 2 auf die Grundbausteine der Technologie des NLP eingegangen und gezeigt, wo diese ihre Anwendung finden. Daran anknüpfend wird auf die etablierten Konventionen eingegangen, die vorgeben, in welche Form textuelle Befehle überführt werden, um vom Computer weiterverarbeitet werden zu können. Dabei wird beleuchtet, aus welchen Elementen dieser Mechanismus, also die *Pipeline*, zusammengesetzt ist, wie Textdaten aufbereitet werden können, welche Algorithmen geeignet sind und wie diese konfiguriert und trainiert werden können. Darüber hinaus wird beschrieben, wie die Genauigkeit dieser Pipeline gemessen werden kann. Im Weiteren werden vier renommierte NLPS mit deren versprochenen Kompetenzen vorgestellt und gezeigt, wie diese eingestellt und trainiert werden. Danach folgt die Selektion der Programmiersprache und der verwendeten Module für den Prototyp. Im Kapitel 3 sind das Vokabular und das Testszenario beschrieben, auf das die zu testenden Systeme trainiert und geprüft werden sollen. Auf die konkrete Implementierung des Prototyps wird in Kapitel 4 eingegangen. Dieser wird mit den anderen NLPS in Kapitel 5 getestet, bevor die Ergebnisse im Kapitel 6 in das Resümee einfließen.

Kapitel 2

State of the Art

Dieses Kapitel widmet sich dem derzeitigen Stand der Technik hinsichtlich NLPS. Zunächst wird auf grundlegende, aus dem Gebiet der künstlichen Intelligenz entsprungene Mechanismen [28] eingegangen, die das Verarbeiten und Interpretieren von Textdaten in menschlicher Sprache überhaupt erst ermöglichen. Auf das umfangreiche Gebiet der NLPS, wichtige Begrifflichkeiten und Ausprägungen, sowie dessen Anwendung wird anschließend eingegangen. Dazu werden einige Verfahren vorgestellt, wie Text zunächst aufbereitet und anschließend die enthaltene Information extrahiert werden kann. Einige Vertreter der derzeitig verfügbaren NLPS, deren Vor- und Nachteile sowie deren Bedienbarkeit werden danach angeführt.

Da im Zuge dieser Arbeit ein selbstimplementierter Prototyp eines NLPS entwickelt wird, werden noch geeignete Programmiersprachen und dazugehörige Module beleuchtet, die für den geforderten Einsatz geeignet erscheinen.

2.1 Machine Learning

Entsprungen aus Teilgebieten der Wissenschaft über künstliche Intelligenz, wurde *Machine Learning* (ML) in den fünfziger Jahren für Aufgaben in dem Gebiet *Pattern Recognition* entwickelt [37]. Als ML-Algorithmen werden Mechanismen bezeichnet, die durch ihre „Lernfähigkeit“ auf verschiedene Problemstellungen angewandt werden können, ohne dafür explizit programmiert worden zu sein [28]. Durch seine vielseitige Anwendbarkeit und das enorme Potential wuchs das Fachgebiet ML in den vergangenen Jahrzehnten stark an. Es wurden zahlreiche Ausprägungen dieses Konzeptes entwickelt, wodurch nun eine Bandbreite an Problemstellungen bewältigt werden kann, die scheinbar nur durch die eigene Vorstellungskraft limitiert ist [20].

Allen ML-Algorithmen gemein ist der Ansatz, dass basierend auf einer Grundmenge an Daten, also dem „Erlernten“, Rückschlüsse auf noch unbekannte Daten gezogen werden können. Ein klassisches Anwendungsgebiet ist ein Spam-Filter eines Mail-Servers, der aufgrund eines großen Datensatzes von als unbedenklich oder Spam markierten E-Mails, eine automatische Spam-Filterung neuer Nachrichten durchführt. Wie wohl allgemein bekannt sein dürfte, passieren dabei auch Fehler, da ML-Algorithmen meist an eine statistische Unschärfe gebunden sind. Im Folgenden wird das Konzept von ML grob beschrieben und einige Begriffe erläutert. Ausführlichere Beschreibungen finden

sich in [20].

2.1.1 Einteilung von Machine Learning Algorithmen

Wie der Name erkennen lässt, werden ML-Algorithmen durch die Eigenschaft ausgemacht, zu „lernen“ im Stande zu sein. Genauer gesagt müssen sie trainiert werden. Dabei wird grundlegend in zwei verschiedene Anwendungsgebiete von ML unterschieden, die jeweils eine eigene Art des Trainings erfordern.

Supervised Machine Learning

Zur Kategorie supervised ML zählen all jene Algorithmen, die mit Datenpaaren trainiert werden. Diese Datenpaare bestehen aus dem zugrundeliegenden Input-Datensatz und dem dazugehörigen Ergebnis, wodurch der Algorithmus lernt, welches Ergebnis für den jeweiligen Input korrekt ist. Ein Beispiel ist etwa das Mark-Up und die Metainformationen eines E-Mails als Input, und einem daraus resultierenden Ergebnis, beispielsweise ob es sich dabei um ein Spam-E-Mail handelt oder nicht. Ein weiteres Beispiel ist das Herkunftsland, das Geschlecht und das Gehalt einer Person als Input und ihre Lebenserwartung als Output. Anhand der Trainingsdaten werden vom ML-Algorithmus Muster erkannt (Pattern Matching), die herangezogen werden, um das Ergebnis neuer Werte bestimmen zu können. Dabei ist natürlich wichtig, dass die Trainingsdaten auf das Wesentliche beschränkt sind, um dem ML-Algorithmus möglichst scharfe Unterscheidungsgrenzen zu liefern. Beispielsweise wird sich die Haarfarbe des Nachbarn kaum auf die Abschätzung der Lebenserwartung einer Person auswirken. Diese Information kann also, nach der Maxime „so viel wie nötig, so wenig wie möglich“, weggelassen werden.

Das Ergebnis eines supervised ML-Algorithmus ist also meist eine diskrete Zuweisung oder ein konkreter, kontinuierlicher Wert, der basierend auf den Trainingsdaten bestimmt wird.

Unsupervised Machine Learning

Obwohl *unsupervised* ML-Algorithmen ebenfalls mit einer Datenmenge trainiert werden, unterscheidet sich ihr Aufgabengebiet grundlegend. Während supervised ML-Algorithmen zum Ziel haben, basierend auf den Trainingsdaten Ergebnisse über neue Datensätze zu ermitteln, wird hier versucht, die Daten zu gruppieren (Clustering) oder wichtige Informationen daraus zu extrahieren. Die Vielfalt der Ergebnisse ist also, anders als bei supervised ML-Algorithmen, nicht (einfach) vorhersehbar und hängt stark von den eingestellten Parametern und den Trainingsdaten ab. Ein klassisches Anwendungsgebiet von unsupervised ML-Algorithmen ist das Extrahieren wichtiger Schlagwörter aus verschiedenen Zeitungsartikeln. Das Grundprinzip dabei ist trivial. Allgemeine Wörter, wie Pronomen oder Artikel, die häufig in allen Beiträgen vorkommen, werden wenig gewichtet, da sie wenig zur Unterscheidung bzw. Charakterisierung beitragen. Weniger häufige Wörter wie „Fußball“, „Politiker“ usw. sind viel aussagekräftiger über einen Artikel, da diese nur in gewissen Artikel-Kategorien (Clustern) vorkommen.

Unsupervised und supervised ML-Algorithmen können auch kombiniert bzw. seriell aneinandergekettet werden. Wird ein unsupervised ML-Algorithmus als Vorstufe eingesetzt, um die vorliegende Datenmenge aufzubereiten oder zu kategorisieren, kann des

Tabelle 2.1: Beispiel von Features und Labels.

Label	*Features*		
Lebenserwartung	Herkunftsland	Geschlecht	Gehalt
79	Österreich	Männlich	42.364
84	Österreich	Weiblich	35.023
79	Deutschland	Männlich	46.489
87	Japan	Weiblich	46.081

dadurch entstehende Ergebnis als Training für supervised ML-Algorithmen verwendet werden. Da im Zuge von NLPS vorwiegend supervised ML-Algorithmen ihre Anwendung finden, wie sich im Verlauf der Arbeit herausstellt, wird nur auf diese Art näher eingegangen. Doch zunächst sollen einige Begrifflichkeiten in diesem Zusammenhang näher erläutert werden.

2.1.2 Terminologie

Features und Labels

Um ML-Algorithmen trainieren zu können, erfordert es Trainingsdaten. In den vorangegangenen Beispielen wurde bereits verdeutlicht, dass die Datensätze, die beim Training als Input übergeben werden, aus mehreren Werten zusammengesetzt sein können. Tabelle 2.1 zeigt eine Liste, die die Lebenserwartung der Konstellation folgender Parameter gegenüberstellt: „Herkunftsland“, „Geschlecht“ und „Gehalt“.[1] Diese Parameter werden als *Features* bezeichnet, da sie die Grundlage bilden, auf deren Basis das Ergebnis ermittelt wird. Die Werte unter „Lebenserwartung“ bilden in diesem Fall die gesuchte Größe, die in der Fachsprache von ML als *Label* bezeichnet wird.

Die Zuweisung der Features und des Labels richtet sich also danach, welche Unbekannte durch den ML-Algorithmus gesucht wird. In diesem Beispiel könnten ebenso Lebenserwartung und Gehalt als Features verwendet und nach dem Label Geschlecht gesucht werden.

Dimensionen

Das Szenario, in dem das Gehalt als Feature 1 auf der X-Achse und die Lebenserwartung als Feature 2 auf der Y-Achse aufgetragen wird, ist in dem Diagramm in Abbildung 2.1 dargestellt. Die Datenpunkte sind als Symbole „männlich“ oder „weiblich“ eingetragen und repräsentieren die Labels. Bei diesem konkreten Beispiel handelt es sich um eine zwiedimensionale Trainingsdatenmenge, die einfach als Diagramm dargestellt werden kann. Obwohl es schwieriger visualisierbar ist, ist die Anzahl der Dimensionen, die von

[1] Die Werte der österreichischen Lebenserwartungen und der Gehälter lehnen sich an die Angaben von Statistik Austria an: http://statistik.at/web_de/statistiken/menschen_und_gesellschaft/gesundheit/gesundheitszustand/lebenserwartung_in_gesundheit/index.html bzw. http://statistik.at/web_de/statistiken/menschen_und_gesellschaft/soziales/gender-statistik/erwerbstaetigkeit/index.html

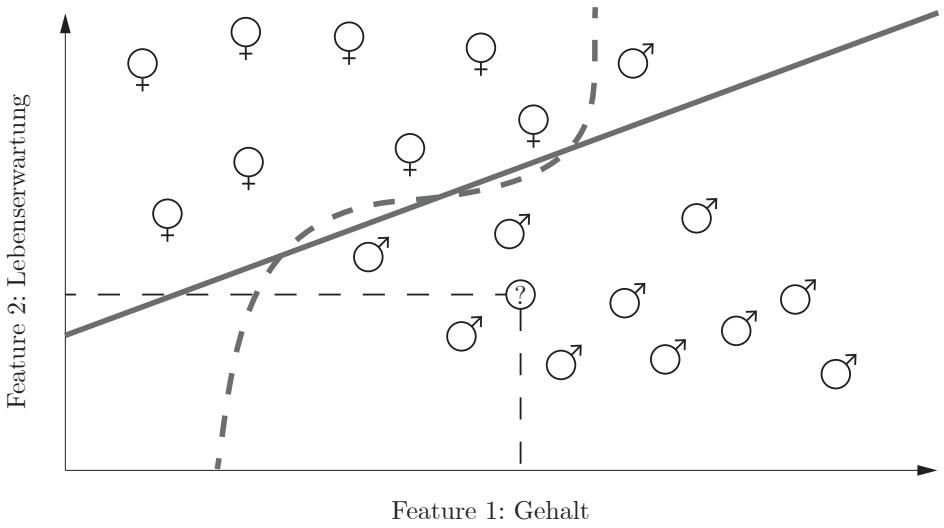


Abbildung 2.1: Szenario, in dem basierend auf Gehalt und Lebenserwartung nach dem Geschlecht der Person gesucht wird.

ML-Algorithmen verarbeitet werden können, theoretisch unbegrenzt. Wird ML im Umfeld von NLPS eingesetzt, ist dies unverzichtbar. Entsprechend der Anzahl an Features in der Datenmenge ergibt sich folglich ein n-dimensionaler *Feature-Raum*, der mit entsprechenden *Feature-Vektoren* gefüllt ist.

Klassifizierung und Regression

Supervised ML-Algorithmen lassen sich weiter unterteilen in Klassifizierungs- und Regressionstypen. Der Unterschied findet sich im erwarteten Ergebnis, also im Typ des Labels. Klassifizierungstypen weisen Input-Datensätze einem bekannten, diskreten Output-Wert zu, wie in dem vorangegangenen Beispiel, bei dem basierend auf den Features „Gehalt“ und „Lebenserwartung“ das Label „männlich“ oder „weiblich“ vergeben wird. Abbildung 2.1 zeigt also ein binäres Klassifizierungsproblem, bei dem der durch das Fragezeichen markierte Datenpunkt entweder den einen, oder den anderen Wert einnehmen kann. Auch mehrklassige (multiclass) Klassifizierungstypen sind möglich. Beispielsweise wenn durch die Features „Gehalt“ und „Geschlecht“ auf ein Herkunftsland rückgeschlossen werden soll.

Im Gegensatz zu Klassifizierungs- wird bei Regressionstypen nicht zwingend ein diskreter, sondern ein kontinuierlicher Wert ermittelt. Das Beispiel in Tabelle 2.1 zeigt einen Regressionstyp, bei dem das Label Lebenserwartung einen kontinuierlichen Wert annehmen kann. Eine Klassifizierung wäre in diesem Fall nicht zielführend.

Decision Border

Bei Klassifizierungstypen von ML-Algorithmen muss sich das System auf einen diskreten Wert bzw. eine konkrete Klasse, basierend auf den Trainingsdaten, festlegen. Demnach bilden sich Bereiche in denen Feature-Werte zu Label-Werten zugeordnet werden. Zwischen diesen Bereichen kommt es zu Grenzlinien, der sogenannten *Decision Border*, die in Abbildung 2.1 als dicke Linie eingezeichnet ist. Die Decision Border ist zwar hier als Gerade dargestellt, kann jedoch, abhängig vom eingesetzten ML-Algorithmus, auch beliebige Kurvenformen annehmen, wie die dicke, strichlierte Line in der Grafik zeigt. Die Form wirkt sich dabei auf die Genauigkeit des Systems aus, wie auch aus dem Diagramm hervorgeht. Führt der eingesetzte Algorithmus zu einer geraden Decision Border, ist in dem gezeigten Diagramm keine Möglichkeit gegeben, diese einzuzeichnen, ohne dabei Fehlzuweisungen in Kauf zu nehmen. Wie zu sehen ist, würde der nördlichste, männliche Datenpunkt falsch zugewiesen werden.

Regularisierung, Überanpassung und Generalisierung

Die Decision Border, die sich bei ML-Algorithmen ausbildet, ist also entscheidend dafür, welches Label ein neuer Datenpunkt erhält. Demnach soll eine möglichst ideale Form der Decision Border angestrebt werden, um die Genauigkeit des ML-Algorithmus zu optimieren. Durch *Regularisieren*, also durch entsprechendes Aufbereiten der Trainingsdaten, wie beispielsweise dem Wegfiltern unwichtiger Features oder durch Skalieren bzw. Gewichten der verwendeten Features, kann der ML-Algorithmus optimal trainiert und die Genauigkeit gesteigert werden. Eine *Überanpassung*, bei der die Decision Border zu sehr den Trainingsdaten angepasst wird, soll jedoch vermieden werden. In den Trainingsdaten können einige Werte enthalten sein, die sich vom Rest der Daten in derselben Klasse stark abheben, wie etwa mit dem nördlichsten, männlichen Datenpunkt in Abbildung 2.1 gezeigt. Grund für derartige Artefakte können u. a. auch Fehler in den Trainingsdaten sein, die vom ML-Algorithmus möglichst ignoriert werden sollen. Werden diese, wie die strichlierte Decision Border zeigt, stark gewichtet, kann es sein, dass der ML-Algorithmus auf neue Daten mangelhaft *generalisiert*, sich also bei der Klassifizierung neuer Punkte zu sehr an die Trainingsdaten hält und Fehlzuweisungen macht.

2.1.3 Supervised Machine Learning Algorithmen

In diesem Abschnitt werden einige der verfügbaren supervised ML-Algorithmen vorgestellt. Dabei wird auf die grundsätzliche Funktionsweise, die Stärken und Schwächen sowie mögliche Einsatzgebiete des jeweiligen Konzeptes eingegangen. Für weiterführende Informationen zu den Algorithmen eignet sich das Basiswerk [20]. In [10] wird die zugrundeliegende Mathematik näher beschrieben.

Zwar sind die zur Erklärung der Algorithmen dargestellten Diagramme, der einfacheren Verständlichkeit wegen, nur zweidimensional ausgeführt, jedoch ist die maximale Feature-Anzahl aller angeführten Algorithmen theoretisch unbegrenzt.

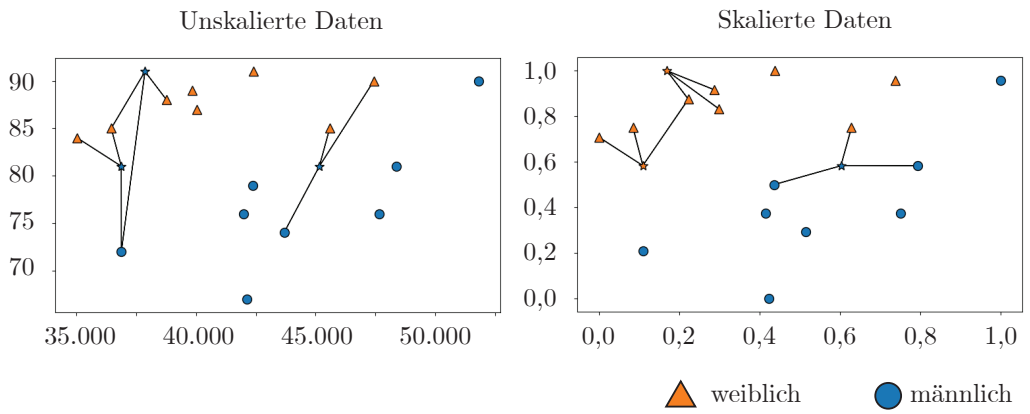


Abbildung 2.2: Klassifizierung neuer Punkte durch kNN bei $k = 3$. In dem linken Diagramm wird aufgrund der unskalierten Trainingsdaten falsch klassifiziert.

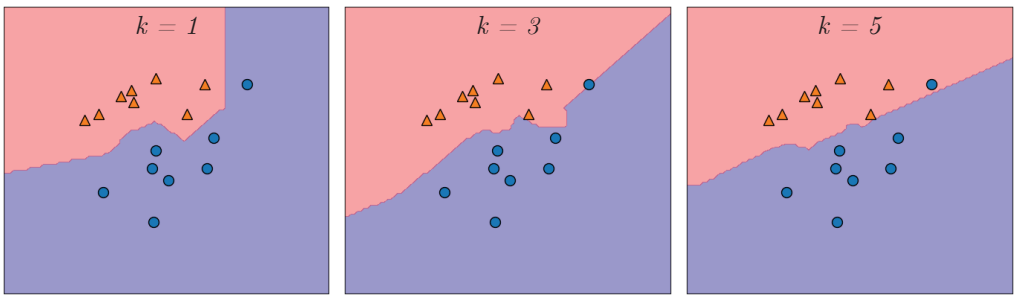


Abbildung 2.3: Decision Border eines kNN-Algorithmus mit unterschiedlichen Parameterwerten für k.

K-Nearest-Neighbors

Das Prinzip von *k-Nearest-Neighbors* (kNN) ist einfach zu verstehen. Der Algorithmus wird mit den entsprechenden Trainingsdaten trainiert, wodurch sich eine Punktewolke im Feature-Raum ergibt. Die Zuordnung bzw. die Klassifizierung neuer Punkte richtet sich nach deren nächsten Nachbarpunkten. Die resultierende Klasse entspricht somit der Mehrheitsklasse aus den k-nächsten Datensätzen. Zum Zweck der besseren Zuweisbarkeit eignen sich daher ungerade Zahlen für k, da es dabei immer zu einer eindeutigen Mehrheit kommt.

Die Größenordnung der einzelnen Features ist bei der Klassifizierung äußerst wichtig und sollte mit Bedacht gewählt werden, da kNN die Achsen des Feature-Raumes nicht automatisch skaliert. Abbildung 2.2 zeigt ein Beispiel, bei dem dieselben Daten, einmal unskaliert (links) und einmal skaliert, verwendet werden. Die drei Testpunkte sind als Stern eingezeichnet und mit der Farbe des vom kNN-Algorithmus gewählten Labels markiert. Die Linien, die von den Testpunkten ausgehen, markieren die Trainingspunkte, die dem Testpunkt am nächsten sind und für die Klassifizierung verwendet werden. Werden die Daten nicht skaliert, wie im linken Diagramm zu sehen ist, sind

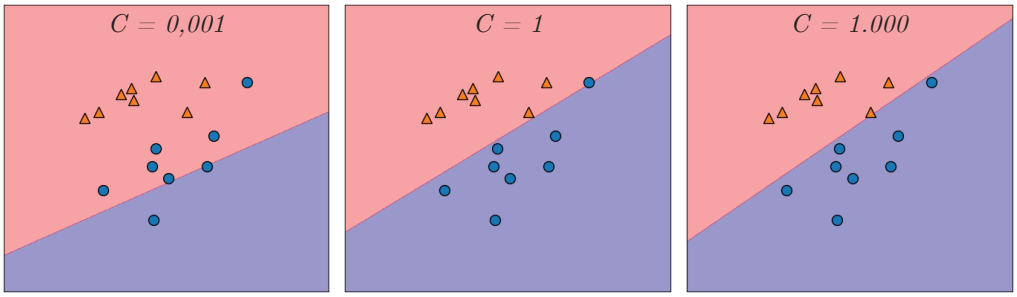


Abbildung 2.4: Decision Border eines LR-Algorithmus mit unterschiedlichen Parameterwerten für C.

die Werte der Lebenserwartung (y-Achse) um drei Zehnerpotenzen niedriger als die des Gehaltes und damit automatisch verhältnismäßig näher beisammen. Dadurch wirkt die Lebenserwartung auf die Klassifizierung viel schwerer Gewichtet als das Gehalt. Diese Gewichtung kann bewusst eingesetzt und gesteuert werden, was jedoch in diesem Beispiel nicht zielführend ist. Viel mehr sollten hier die Werte skaliert werden, um beiden Features dieselbe Wertigkeit zuzuweisen. Das Ergebnis ist in dem rechten Diagramm zu sehen, bei dem die Werte viel plausibler zu sein scheinen. Sind die Daten richtig gewichtet, kommt es auf die Einstellungen des kNN-Algorithmus an, wie neue Daten klassifiziert werden bzw. wie sich die Decision Border ausbildet. Abbildung 2.3 zeigt die Ausprägungen bei drei verschiedenen Einstellungen. Es ist zu sehen, dass sich die Kurve mit steigendem Wert für k zunehmend glättet und nicht mehr so sensibel auf einzelne Ausreißer reagiert. Das System generalisiert also besser und ist nicht mehr so sehr auf die Trainingsdaten „maßgeschneidert“. Durch geeignete Parameterwahl ist kNN also ein Generalist und kann für viele Datenkonstellationen verwendet werden. Bei Feature-Vektoren mit vielen Dimensionen oder bei lückenhaften Datensätzen, bei denen nicht alle Features im Vektor vorhanden sind, ist dieser Algorithmus weniger geeignet [7].

Linear Classifiers

Basierend auf dem Konzept der linearen Regression, bei der eine Punktewolke linear interpoliert wird, wird auch bei linearen Klassifikationen durch die Trainingsdaten eine Gerade, bzw. eine Ebene oder Hyperebene im mehrdimensionalen Feature-Raum, gezogen. Diese Trennung ist gleichzeitig die Decision Border, die zur Klassifizierung verwendet wird. Eine Implementierung dieses Konzeptes ist die *Logistic Regression* (LR). Eine Möglichkeit diesen Algorithmus zu konfigurieren, ist über den Parameter C, durch den die Steigung der Decision Border justiert, und für die jeweiligen Trainingsdaten optimiert werden kann. Bei hohen Werten für C versucht die LR, sich stark an die Trainingsdaten zu halten und reagiert sensibel auf Artefakte. Eine Überanpassung könnte entstehen und der ML-Algorithmus würde schlecht generalisieren. Daher ist das Ziel einen Wert zu finden, bei dem das System basierend auf den Trainingsdaten gut auf neue Daten generalisiert. In Abbildung 2.4 wird gezeigt, wie sich mit steigendem Wert für C, die Decision Border und somit die Genauigkeit des Algorithmus verändert. Wie zu sehen ist, eignet sich zum Antasten an den optimalen Wert ein Verändern auf einer

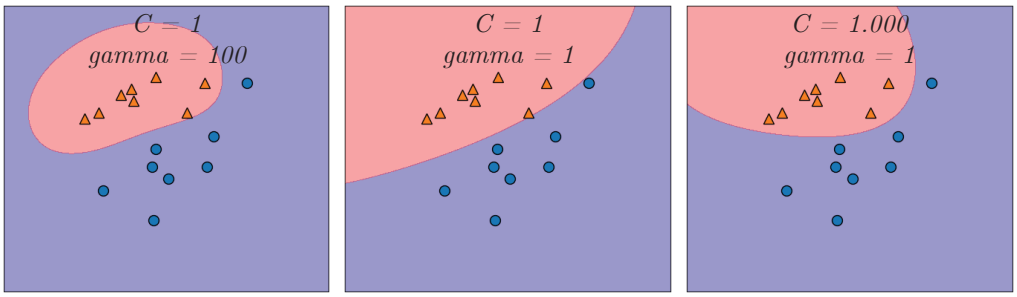


Abbildung 2.5: Decision Border eines SVM-Algorithmus mit unterschiedlichen Parameterwerten für C und *gamma*.

logarithmischen Skala [21].

Ein Vorteil von LR- gegenüber kNN-Algorithmen ist, dass sie sich auch für Daten eignen, bei denen nicht alle Features enthalten oder wichtig sind. Außerdem können sie auf sehr große Datenmengen skaliert werden [20]. Im Gegensatz zu kNN müssen die Daten bei der LR nicht skaliert werden, was die Datenaufbereitung etwas vereinfacht.

Support Vector Machines

Neben den linearen *Support Vector Machines* (SVM), die ähnliche Decision Borders erzeugen wie die LR, gibt es noch erweiterte Ausführungen, die es mittels mathematischer Tricks ermöglichen, durch lineare Geometrien komplexe (z. B. elliptische) Decision Borders zu erzeugen. Beispielsweise kann bei einem zweidimensionalen Feature-Raum eine neue Dimension aus dem Quadrat des ersten Features hinzugefügt werden, um so einen künstlichen, dreidimensionalen Feature-Raum zu schaffen. Unter Umständen ist es dann möglich, die Labels mittels einer Ebene zu trennen und so besser zu klassifizieren. Würde diese Ebene als Decision Border in den eigentlichen zweidimensionalen Feature-Raum projiziert, ergäbe das eine parabelförmige Kurve [10], wie Abbildung 2.5 zeigt. Bei noch weiter entwickelten SVM-Algorithmen kommt der sogenannte „radial basis function“-Kernel zum Einsatz, der gezielt jene Feature-Vektoren für die Berechnung der Decision Border verwendet, die sich an der Grenze der Klassen befinden. Diese werden als *Support-Vektoren* bezeichnet und geben der SVM ihren Namen. Mit dem Parameter *gamma* wird festgelegt, wie nahe sich Datenpunkte sein müssen, um als Gruppe erkannt und von Support-Vektoren umgeben zu werden. Für diesen Zweck sollten die Trainingsdaten, wie bei kNN, skaliert werden. Der Parameter C regelt simultan zur LR die Generalisierung der SVM.

Weil SVM für hohe Datenmengen und für Feature-Räume mit vielen Dimensionen geeignet sind [15], sind sie sehr vielseitig einsetzbar.

Decision Trees und Random Forest

Ein weiteres Konzept von ML bilden *Decision Trees* (DT), die die Trainingsdaten in eine baumförmige Datenstruktur bringen, woher sich der Name ableitet. Obwohl sich mittlerweile zahlreiche Ausprägungen dieses ML-Algorithmus entwickelt haben [4], sollen hier nur die grundsätzliche Funktionsweise sowie eine weiterentwickelte Methode vorgestellt

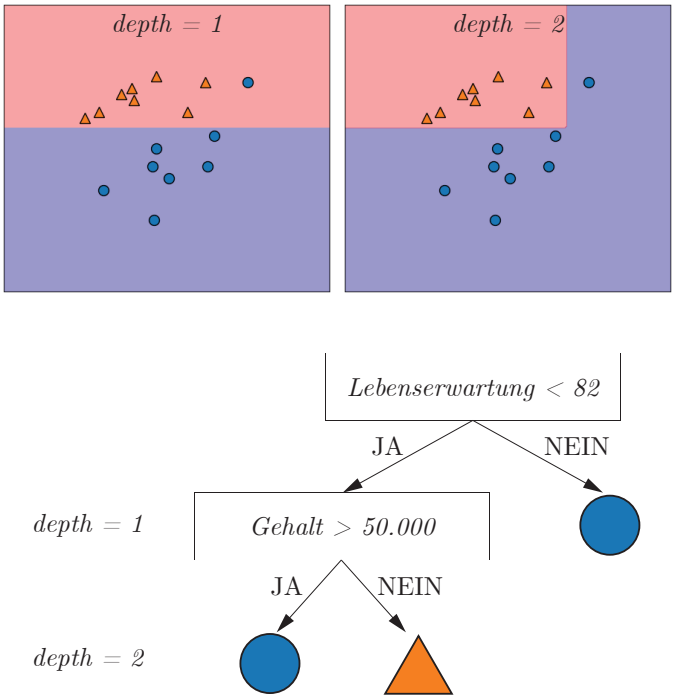


Abbildung 2.6: Beim Trainieren bzw. Erstellen des DT wird abwechselnd eine Achse (Dimension) dort unterteilt, wo die Labels der Datenpunkte am besten voneinander getrennt werden.

werden. Wie Abbildung 2.6 zeigt, ist das Training von DT ein iterativer Prozess, bei dem sich der Algorithmus mit jeder Stufe *depth* mehr den Trainingsdaten annähert. Mit jeder Iteration wird die Trainingsdatenmenge entlang der Feature-Achsen dort unterteilt, wo die Labels der Daten optimal von einander getrennt werden. Das linke Diagramm zeigt den DT bei $depth = 1$, wo die erste Grenze bei der Achse „Lebenserwartung“ eingezogen wird. Es entstehen zwei Felder. Da das untere Feld bereits eindeutig einem Label zugewiesen werden kann, entsteht an der Kante des Baums ein Blatt mit dem Label des Feldes. Das andere Feld ist hingegen noch nicht eindeutig zugewiesen und die maximal erlaubte Tiefe des Baumes ist in diesem Beispiel noch nicht erreicht. Daher wird das obere Feld erneut geteilt, jedoch diesmal an der Achse „Gehalt“. Da nun auch in dem Teilbaum beide entstanden Felder eindeutig zuweisbar sind, werden zwei Blätter erzeugt und der Baum ist fertiggestellt. Die Zuweisung neuer Datenpunkte ist demnach eine Abfolge von Verzweigungen in dem gebildeten Baum. Diese verhältnismäßig primitive Ausführung versucht jedes Artefakt zu berücksichtigen, wodurch unnötige „Label-Inseln“ entstehen können, die unverhältnismäßig große Bereiche abdecken. Deshalb neigt dieses Konzept zur Überanpassung an die Trainingsdaten.

Eine Ausprägung von DT, die dem entgegenwirkt, sind *Random Forests* (RF). Wie der Name bereits erahnen lässt, bestehen RF aus mehreren, zufällig erzeugten DT. Der Aufbau eines DT ist bis auf ein Detail dem vorhin beschriebenen Verfahren sehr ähnlich,

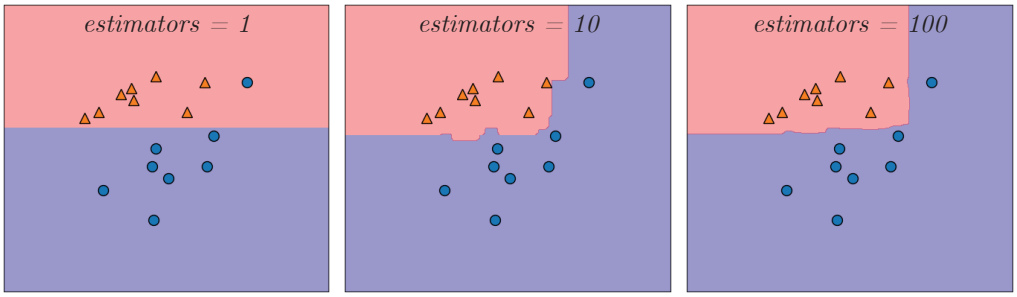


Abbildung 2.7: Decision Border eines RF-Algorithmus mit unterschiedlichen Parameterwerten für *estimators*.

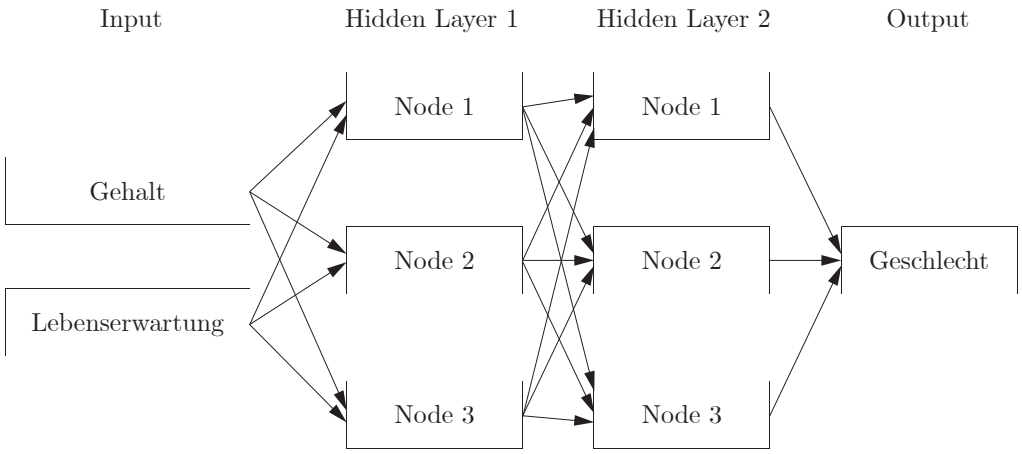


Abbildung 2.8: Aufbau eines NN mit zwei Hidden Layers zu je drei Neuronen.

jedoch wählen DT in RF aus den Trainingsdaten zufällig Datensätze aus. Dabei kommt es vor, dass einige Punkte mehrfach, andere hingegen gar nicht in den Trainingsdaten des DT vorkommen. Aus diesem Grund werden derartige DT „Estimators“ genannt. RF haben demnach einen Parameter *estimators*, der angibt, wie viele solcher DT in dem RF aufgebaut werden. Erfolgt eine Zuweisung eines neuen Datenpunktes, wird hier unter allen DT abgestimmt. Durch dieses Verfahren wird das Überanpassen eines einzigen DT reduziert, weshalb RF eine Verbesserung der Generalisierung aufweisen. Abbildung 2.7 zeigt, wie sich die Decision Border mit der Zahl der verwenedeten *estimators* ändert.

Die Funktionsweise von RF erfordert keine Skalierung der Trainingsdaten, kann jedoch parameterabhängig rechenintensiver sein als andere supervised ML-Algorithmen. Gerade bei einem Feature-Raum mit vielen Dimensionen und Feature-Vektoren ist der Rechenaufwand verhältnismäßig größer.

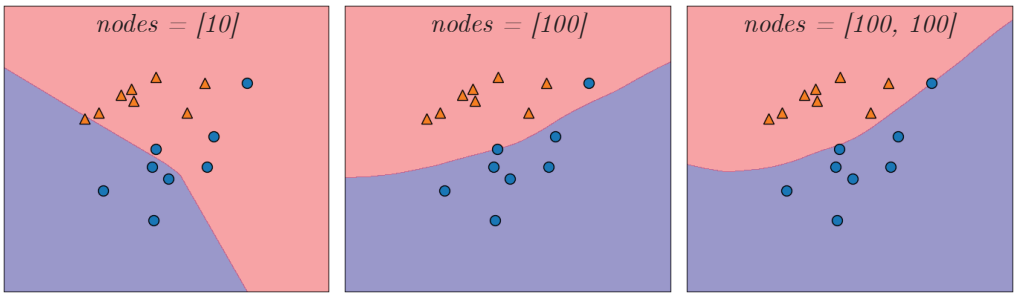


Abbildung 2.9: Decision Border eines NN-Algorithmus bei einem Hidden Layer mit 10 Nodes (links), bei einem Hidden Layer mit 100 Nodes (mittig) und bei zwei Hidden Layers mit jeweils 100 Nodes (rechts).

Neural Networks

Unter dem Namen *Deep Learning* erlangten *Neural Networks* (NN) große Bedeutung im Bereich der Text-, Sprach- und Bilderkennung [18]. Obwohl das Konzept dieses ML-Algorithmus grundsätzlich relativ leicht verständlich ist, führt es doch zu einem schwer vorhersehbaren bzw. nachvollziehbaren Verhalten. Wie Abbildung 2.8 zeigt, besteht der Aufbau eines NN aus den wesentlichen Komponenten: Input, Hidden Layers und Output. Die Feature-Vektoren werden in den *Nodes*, die auch *Neuronen* genannt werden, als gewichtete, lineare Funktionen verwendet um daraus eine nichtlineare Funktion zu errechnen [10]. Jedes Neuron hat dabei eine eigene Gewichtung und die Ergebnisse werden als gewichtete Inputs der Neuronen in dem nächsten Hidden Layer verwendet. Dies geschieht durch alle Hidden Layers hindurch, bis der Output-Node erreicht wird, wo die Teilergebnisse zusammengeführt werden und ein Gesamtergebnis errechnet wird. Über die Anzahl und die Einstellungen der Hidden Layers und deren Nodes lässt sich u. a. die Auflösung sowie Geometrie der Decision Border einstellen und somit die Anwendbarkeit auf komplexe Feature-Räume steuern. Beim Training von NN werden die Parameter der Nodes so verändert, dass die durch die *Cost Function* [22] ermittelte Fehlerquote so gering wie möglich gehalten wird.

Die Decision Border eines NN-Algorithmus ist in Abbildung 2.9 dargestellt. Es bestätigt sich, dass die Kurve feiner aufgelöst wird, je mehr Hidden Layers bzw. Neuronen verwendet werden. Die Form passt sich dadurch auch mehr den Trainingsdaten an. Das linke Diagramm zeigt, dass NN bei ungünstig gewählten Parametern eine fehlerhafte Decision Border ausbilden, deren Ursache viel schwieriger auszumachen ist, als es in verhältnismäßig einfacheren Algorithmen der Fall ist. Das rechte Diagramm zeigt die Decision Border eines viel größeren NN, die jedoch womöglich zur Überanpassung neigt. Die Variante, die wahrscheinlich am besten generalisiert, wird mittig dargestellt.

NN sind also in der Lage, sehr umfangreiche und komplexe Zusammenhänge zu erkennen. Die Features sollten auch hier skaliert werden um eine homogene Größenordnung und somit ausgeglichene Gewichtung in den Berechnungen zu ermöglichen. Nachteilig an NN ist die schwer nachzuvollziehende Funktionsweise und demnach das schwierige Training und Optimieren des Algorithmus.

2.1.4 Training und Evaluierung

Nachdem nun die Funktionsweisen der einzelnen supervised ML-Algorithmen dargelegt sind, wird hier auf eine zentrale Disziplin, das Training und die Evaluierung, eingegangen. Es werden dabei Techniken vorgestellt, mit Hilfe derer die Leistung (Trefferquote) bei ML bestimmt und die optimalen Parameter ermittelt werden können.

Datenaufbereitung

Grundsätzlich sollten die Daten, die zum Training eines ML-Algorithmus verwendet werden, gründlich gewählt und aufbereitet sein. Sind viele Features verfügbar, bedeutet das nicht zwingend, dass auch alle zur besseren Unterscheidbarkeit bzw. Klassifizierung der Labels beitragen. Bei der Bestimmung des Geschlechtes aufgrund des Gehalts und der Lebenserwartung einer Person, wäre das Feature der Haarfarbe des Nachbarn wohl belanglos und würde die Grenzen der Feature-Vektoren nur unscharf machen und die Klassifizierung erschweren. Neben dem Filtern der Daten müssen diese, falls der verwendete ML-Algorithmus dies erfordert, entsprechend aufbereitet und skaliert werden.

Cross-Validation

Die wohl interessantesten und für die Bewertung wichtigsten Kriterien eines ML-Systems sind seine Leistung, also die Trefferquote, und seine Generalisierung. Die Leistung gibt Aufschluss in welchem Verhältnis das System korrekt bzw. fehlerhaft zuweist. Es muss jedoch unterschieden werden, welche Daten bzw. welches System zur Bestimmung der Trefferquote verwendet wird.

Beim Training des ML-Algorithmus wird bekanntlich die Decision Border erzeugt, anhand der das System generalisiert, also in die Lage versetzt wird, Daten, die nicht in den Trainingsdaten enthalten sind, korrekte Labels zuzuweisen. Das Maß der Generalisierung kann demnach nicht mit den Trainingsdaten selbst ermittelt werden, da diese dem System bereits bekannt sind. Dazu sind also Testdaten erforderlich. Eine mögliche Vorgehensweise für die Ermittlung der Generalisierbarkeit ist, die Trainingsdaten in einen Trainingsblock mit ca. dreiviertel und einen Testblock mit dem Rest der Daten zu erstellen. Obwohl dadurch die Anzahl der verfügbaren Trainingsdaten und dadurch die Genauigkeit gesenkt wird, kann nun anhand der Testdaten die Generalisierung ermittelt werden. Dazu wird der ML-Algorithmus anhand des Trainingblocks trainiert und verglichen, wie viele Labels des Testblocks korrekt zugewiesen werden. Der Durchschnitt dieser Zuweisungen gibt die Leistung der Generalisierung an. Da bei diesem Verfahren jedoch nur ein Teil der Trainingsdaten verwendet wird, muss darauf geachtet werden, dass auch die Labels ordentlich verteilt werden, damit verhindert wird, dass zufällig eine ganze Klasse in die Testdaten geriet und somit nicht trainiert werden kann. Aus diesem Grund hat sich das *Cross-Validation*-Verfahren (CV) etabliert [17], bei dem die Trainingsdaten mehrmals unterschiedlich geteilt und der ML-Algorithmus getestet wird.

Eine Übersicht der Leistungen der vorgestellten supervised ML-Algorithmen ist in Tabelle 2.2 angeführt. Die verwendeten Systeme wurden dabei noch nicht optimiert, denn es soll nur gezeigt werden, wie die Ergebnisse der Testverfahren variieren. Als Trainingsgrundlage wurden Daten mit 1.000 Einträgen zu je zwei Labels und zwei Fea-

Tabelle 2.2: Gegenüberstellung der Genauigkeiten der verschiedenen supervised ML-Algorithmen im einfachen Test mit Trainings- und Testdaten sowie im Cross-Validation Test.

Algorithmus	*Genauigkeit, Testmethode*		
	Trainingsdaten, einfach	Testdaten, einfach	Testdaten, CV
kNN	95,1%	94,0%	92,7%
LR	86,1%	82,4%	85,3%
SVM	93,5%	94,0%	93,3%
RF	100,0%	95,6%	94,1%
NN	90,1%	86,8%	90,2%

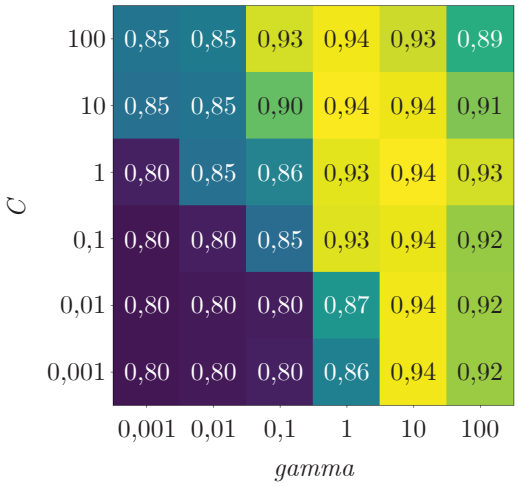


Abbildung 2.10: Beispiel einer GS-Wertetabelle basierend auf den Parametern C und *gamma* einer SVM.

tures verwendet.[2] Wie zu erkennen ist, sind die Algorithmen anhand der Trainingsdaten, auf die sie trainiert wurden, sehr optimistisch, jedoch bei Neudaten im Schnitt merklich ungenauer. Bei der CV, die die aussagekräftigsten Ergebnisse liefert, sind die Ergebnisse wieder etwas höher, was darauf schließen lassen könnte, dass beim Aufteilen zwischen Trainings- und Testdaten eine ungünstige Trennung gemacht wurde.

Parameteroptimierung

Da nun auf eine aussagekräftige Beurteilung der Leistung eingegangen wurde, ist es möglich, sich dem Thema Parameteroptimierung zu widmen. Neben einigen fortschrittlicheren Methoden, ist die *Grid Search* (GS) die wohl verbreitetste [5]. Dabei werden in eine Matrix verschiedene Parameterkombinationen eingetragen, der ML-Algorithmus darauf eingestellt, trainiert und mittels CV ausgewertet. Es entsteht eine Ergebnisma-

[2] Die Trainingsdaten wurden in Python mit der Funktion `make_moons` des Moduls `sklearn` erstellt.

trix, die beispielhaft in Abbildung 2.10 dargestellt ist. Die Einträge des CV-Ergebnisses sind entsprechend ihrer Größe farblich hinterlegt („Heatmap“), um die Bestwerte leichter erkennen zu können. Dies ist für die Wahl des Wertebereiches der Parameter wichtig, da „helle“ Ecken mit hohen Werten darauf schließen lassen, dass es sich lohnen würde, den Wertebereich in diese Richtung noch zu erweitern, da dahinter noch bessere Werte erzielt werden könnten.

Mit GS können jedoch nicht nur Parameterkonstellationen für einen ML-Algorithmus allein, sondern auch eine ganze Pipeline an Mechanismen getestet werden, um die optimale Leistung des gesamten Systems ermitteln zu können.

2.2 Natural Language Processing

Die Studien über *Natural Language Processing* (NLP) reichen zurück in die fünfziger Jahre. Es wird also schon sehr lange an einem System geforscht, mit dem ein Kommunizieren in menschlicher Sprache soweit möglich ist, dass ein Mensch nicht unterscheiden kann, ob er mit einem künstlichen System oder einem anderen Menschen kommuniziert. Alan Turing, ein Pionier auf dem Gebiet, drückte dies mit dem nach ihm benannten *Turing Test* aus [27].

Der Fachbereich NLP erstreckt sich über sehr viele Disziplinen und kann, wie in Abbildung 1.1 dargestellt, entsprechend aufgegliedert werden. In [12] ist der gesamte dargestellte Prozess beschrieben. Diese Arbeit beschäftigt sich jedoch ausschließlich mit dem Teilgebiet der Textinerpretation, für das der grundsätzliche Ablauf sowie einige Techniken in [8] erläutert sind.

2.2.1 Einsatz und Funktionsweise von NLP

Nicht nur die Disziplin NLP sondern auch die der Textinterpretation kann in verschiedene Teilgebiete unterteilt werden: in *Text Mining* (TM) und NLP für kurze Sätze [39]. Beide haben gemein, dass sie sehr ähnliche Grundmechanismen verwenden und versuchen, Informationen aus vorliegenden Texten zu extrahieren. Dazu müssen beide Systeme mit einer bestimmten Datenmenge trainiert werden, doch bereits hier weisen beide Systeme grundlegende Unterschiede auf.

Text Mining

Die Absicht, die bei TM verfolgt wird, ist basierend auf einem Text, der aus mehreren Sätzen besteht, Informationen zu extrahieren und so auf gewisse Charakteristiken rückzuschließen. Ein Anwendungsbeispiel wäre, Zeitungsartikeln Rubriken zuzuweisen, die nicht vorab definiert sind. Basierend auf der Ausgangsdatenmenge müssen also auch die Rubriken bzw. Kategorien gefunden werden, denen die Texte zugewiesen werden sollen. Rückschließend handelt es sich bei TM also um ein Kategorisierungsproblem bei dem die vorgestellten supervised ML-Algorithmen angewendet werden können. Diese Kategorisierung ist jedoch eines der letzten Glieder der Prozesskette und setzt entsprechende Verarbeitungsschritte voraus [25], die wie folgt grob unterteilt werden können:

- **Text Analysis und Information Extraction:** Bevor die Texte kategorisiert werden können, muss der Feature-Raum erzeugt werden. Anders als bei den Bei-

spielen, die im Zuge der ML Einführung erwähnt wurden, ist der Feature-Raum bei TM-Anwendungen nicht vorgegeben und bildet sich aus der vorliegenden Textdatenmenge. Dazu müssen zunächst die Texte aufbereitet und unnötige Informationen entfernt werden.

- **Clustering:** Basierend auf den übriggeblienen Wortmengen werden mittels unsupervised ML Cluster gebildet. Diese Cluster bzw. Gruppen haben als Label das Leitwort, also etwa den Rubriknamen, und als Daten die Features der korrespondierenden Texte [6].
- **Categorization:** Mittels der Ergebnisse aus dem Clustering kann zuletzt ein supervised ML-Algorithmus trainiert werden. Mit diesem können schlussendlich neue Texte automatisch den Rubriken zugewiesen werden.

TM ermöglicht also, automatisiert große Textmengen dynamischen Kategorien zuzuweisen und erleichtert dadurch anwendungsabhängig ihre Redaktion, Zensur, Analyse oder andere Aufgaben, die aufgrund der stetig wachsenden Datenmenge im Internet, nicht mehr von Menschen bewältigbar wären.

NLP für kurze Sätze

Gänzlich unterschiedliche Aufgabengebiete, wie etwa in dieser Arbeit, erfordern NLP für kurze Sätze. Dabei soll eine vergleichsweise kleine Textmenge, etwa ein Befehl, der aus einem einzigen Satz besteht, semantisch vom System erfasst und abgearbeitet werden [39]. Ein typisches Anwendungsgebiet dieser Systeme sind beispielsweise elektronische Assistenten, wie etwa *Siri*[3] oder *Alexa*[4], oder sogenannte *Messenger Bots*[5], die häufig in sozialen Kommunikationsplattformen wie *Facebook* usw. eingesetzt werden. Die Ausgangssituation ist meist grundlegend unterschiedlich zu TM, da bei TM die Textdaten oft bereits vorliegen und bei NLP für kurze Sätze die Trainingsdaten künstlich erzeugt werden müssen. Dennoch ähnelt die Prozesskette der Verarbeitung der des TM, jedoch erfolgt das Clustering meist nicht automatisch [9]:

- **Szenarien und Textbeispiele festlegen:** Zunächst sollte ein Umfang definiert werden, welche Befehle und Szenarien vom NLP-System erkannt und verarbeitet werden sollen. Dabei ist ein guter Ansatz erst Nutzerszenarien zu definieren und zu beschreiben, welche Interaktionsmöglichkeiten den Benutzern ermöglicht werden sollen. Das bedeutet, dass die Aufgaben bzw. Kategorien, z. B. „bestellen“, sowie wichtige Schlagwörter wie „Pizza“, definiert werden sollen, um anschließend zu den Aufgaben Szenarien ausarbeiten zu können. Beispielsweise „Ich möchte eine Pizza Diavolo bestellen.“. Basierend auf diesen Szenarien ergibt sich eine Sammlung an möglichen Befehlen, die ihren entsprechenden Kategorien bzw. Labels zugewiesen sind. Wichtig ist, dass eine entsprechend vielfältige Ausdrucksweise, also viele Möglichkeiten dasselbe auszusagen, vorhanden sind um ein möglichst gutes Training zu erzielen.
- **Textaufbereitung:** Die zusammengestellten Sätze werden anschließend ähnlich wie beim TM aufbereitet um aus der Textmenge entsprechende Features extrahie-

[3] https://www.apple.com/ios/siri/

[4] https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/b?ie=UTF8&node=9818047011

[5] https://www.forbes.com/sites/markfidelman/2016/05/19/10-facebook-messenger-bots-you-need-to-try-right-now/#25ef6713325a

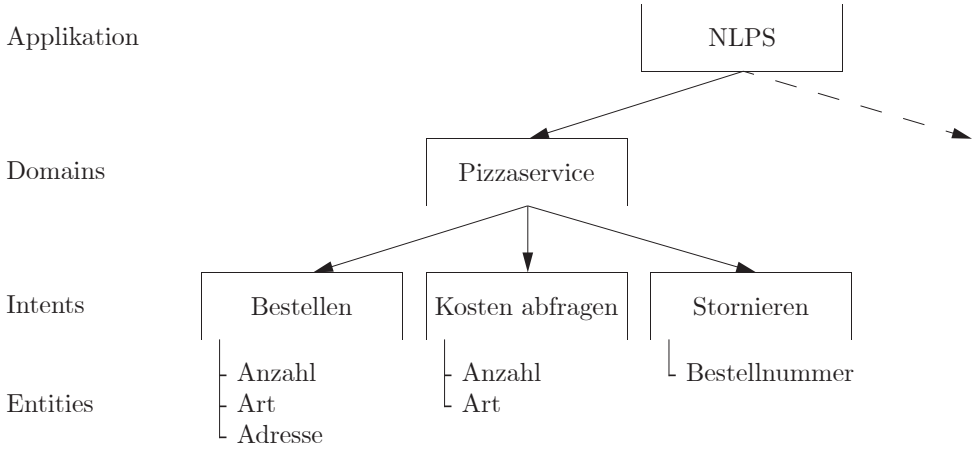


Abbildung 2.11: Hierarchie und Beispiele der Begriffe, die sich im Kontext mit NLP etabliert haben.

ren zu können. Da bei NLP in diesem Fall erheblich weniger Text der Verarbeitung zur Verfügung steht als beim TM, muss bei der Aufbereitung vorsichtig vorgegangen werden, damit nicht zu viele Informationen verloren gehen.

- **Categorization:** Ein geeigneter supervised ML-Algorithmus wird mit den Label und Feature Datenpaaren trainiert. Auch hier ist mittels CV eine Messung der Genauigkeit möglich, jedoch ist diese viel weniger aussagekräftig, da die realen Nutzerdaten aus völlig anderen Wörtern bzw. Features zusammengesetzt sein können als die Trainingsdaten und dennoch dasselbe aussagen.
- **Entity Extraction:** Nachdem in einem realen Nutzerszenario der eingegebene Befehl kategorisiert, also einem Vorhaben zugewiesen wurde, wird abschließend versucht, alle übrigen, erlernten Schlagwörter zu extrahieren um dem angehängten System möglichst viel Information des Befehls übermitteln zu können.

Natürlich sind die Einsatzzwecke von TM sowie NLP nicht an die hier angeführten gebunden und können sehr vielseitig verwendet werden. Der im Zuge dieser Arbeit entstehende Prototyp setzt jedoch auf das Prinzip NLP für kurze Sätze, weshalb im Weiteren intensiv darauf eingegangen wird.

2.2.2 Terminologie

Im Rahmen von NLP für Online-Applikationen wie Messenger Bots oder elektronische Assistenten haben sich im Kontext mit Verarbeitung kurzer Sätze und Befehle einige Begriffe etabliert [14], deren Hierarchie in Abbildung 2.11 dargestellt ist.

Domain

Viele verfügbare NLPS sind in der Lage Befehle und Fragen mehrerer Themengebiete zu verarbeiten. Ein berühmter Vertreter dieser Bots ist *Watson*[6] von *IBM*, der mit einer

[6] https://www.ibm.com/watson/

enormen Datensammlung von u. a. *Wikipedia* trainiert und 2011 in dem Wissensspiel *Jeopardy* auf verschiedene Themen befragt wurde und darauf entsprechend antworten musste. Diese Themengebiete werden im NLP-Jargon als *Domains* bezeichnet. Je nach Anwendung können sich Domains beispielsweise über „Wetter“, „Telefon“, „Pizzaservice“ u. v. m. erstrecken. Sie ergeben also die erste Ebene bzw. den ersten Zweig der Kategorisierung aus Abbildung 2.11, und schränken dadurch die Intents und Entities ein, die in dem Kontext erkannt werden können. Da es sich hierbei um ein Kategorisierungsproblem handelt, kann die Erkennung der Domain also mit einem ML-Algorithmus ermittelt werden, der auf die Domain-Findung trainiert ist.

Intent

Wurde der Befehl in eine Domain kategorisiert, also das Thema dieses Befehls definiert, muss im nächsten Schritt die Absicht, der *Intent*, die der Benutzer an den NLPS richtet, erkannt werden. So wie die Domains sind auch die Intents von kategorischer Natur, weshalb auch diese Information mittels eines domänenspezifischen, für die jeweilige Intent-Erkennung trainierten ML-Algorithmus erkannt werden kann. Abhängig von der Implementierung kann es also ein Netzwerk von ML-Algorithmen geben, die auf Aufgaben spezialisiert und ebenso baumförmig verzweigt sind, wie die Domains und Intents.

Entity

Entities, oder auch *Named Entities* genannt, sind der eigentliche Wortschatz im Zusammenhang mit dem NLPS bzw. mit dem Domain und Intent. Darin können genauso gut Eigennamen enthalten sein. Beispielsweise steht im Englischen das Wort „Apple“ kontextabhängig entweder für die Frucht oder den Firmennamen. Diese Mehrfachbedeutungen, Synonyme („Erde“ und „Welt“), Worthierarchien („Gala“, „Apfel“ und „Obst“) und Numerus („Tal“ und „Täler“) stellen, wie sich im Verlauf der Arbeit herausstellen wird, für NLPS eine besondere Herausforderung dar. Obwohl es naheliegend ist beschränken sich Entities jedoch nicht auf Nomen, sondern können grundsätzlich jeden Worttyp annehmen.

Durch die drei Ebenen, Domain, Intent und Entity, können Befehle üblicherweise soweit maschinenverständlich übersetzt werden, dass sie vom System ausgeführt werden können. Diese drei Ebenen können somit mit Elementen der Programmierung gleichgesetzt werden. Die Domain entspricht der Klasse (z. B. `PizzaService`), der Intent der Methode (z. B. `bestellen`) und die Entities stellen die Methodenparameter dar (z. B. „Diavolo“ für `pizza` und „1“ für `menge`). Eine beispielhafte Übersetzung des Satzes „Ich möchte bitte eine Diavolo bestellen.“ in ein JSON-Objekt ist hier angeführt:

```
{
  "domain": "pizzaservice",
  "intent": "bestellen",
  "entities": [
    "pizza": "diavolo",
    "menge": 1
  ]
}
```

2.2.3 Textaufbereitung

Die Qualität wie Domains, Intents und Entities mit einem NLPS aus einem Satz extrahiert werden können, hängt zunächst von der vorliegenden Eingabe ab. Im Idealfall liegt der Befehl in korrekter Rechtschreibung und Grammatik vor, und auch die Satzzeichen darin, sollten entsprechend gesetzt sein. Im Zuge dieser Arbeit wird davon ausgegangen, dass die eingegebenen Sätze keinerlei Fehler in Rechtschreibung, Grammatik oder Zeichensetzung enthalten, da deren Berücksichtigung und Korrektur weit außerhalb dieses Rahmens liegen. Zudem werden keine Abkürzungen („u.v.m," etc.) berücksichtigt.

Aus [16] sowie aus [8] sind im Folgenden einige Auszüge angeführt, wie Texte für die weitere Verarbeitung in einem NLPS aufbereitet werden können. Festzuhalten ist, dass nicht alle Aufbereitungsschritte immer sinnvoll sind und die Abfolge und Ausführung dieser Schritte stets auf die Implementierung des angehängten NLPS angepasst werden sollen.

Tokenization und Großschreibung

Befehle in menschlicher Sprache, die textuell vorliegen, können zunächst ganz grundlegend aufbereitet und in ihre einfachste Form gebracht werden. Dazu gehört, dass übermäßige Weißräume, also Absätze, Einrückungen etc., auf ein einziges Leerzeichen reduziert werden. Die Regeln der Grammatik bedingen, dass Satzzeichen direkt an Wörter angrenzen, wie beispielsweise zwischen zwei Gliedsätzen: „Ich möchte, dass . . . ". NLPS verwenden die Leerzeichen um die vorliegenden Zeichenketten in sogenannte *Tokens* aufzuteilen. Dieser Prozess heißt *Tokenization* und ergäbe in dem Fall u. a. den Token „möchte,", der sich vom Wort „möchte" aufgrund des Satzzeichens unterscheidet. Aus diesem Grund müssen die Satzzeichen vor der Bildung der Zeichenketten von den angrenzenden Wörtern durch ein Leerzeichen getrennt werden. Die dadurch entstehenden Tokens würden in diesem Pseudocode-Beispiel so aussehen:

```
tokenize("Ich möchte, dass ..."); // ["Ich", "möchte", ",", "dass", "..."]
```

Ist der Befehl in die Tokens aufgeteilt, können die Zeichen der Wörter in Kleinbuchstaben umgewandelt werden. So wird verhindert, dass Fehler von eigentlich kleingeschriebenen Wörtern, die evtl. wegen ihrer Satzstellung großgeschrieben werden müssen, verhindert werden.

Word-Stemming

Ist der Befehl allgemein aufbereitet und in die Zeichenketten bzw. Wörter aufgeteilt, können die Wörter vereinfacht werden. *Word-Stemming* [33] ist für die Vereinfachung von Wörtern eine Methode, mit der versucht wird, basierend auf definierten Regeln das Grundwort jedes Wortes zu bestimmen. Beispielsweise könnten Wörter wie „Autos" wegen der Endung „s" als Wort in Plural identifiziert und durch das Entfernen der Endung in die Singularform umgewandelt werden. Diese Regel würde jedoch bei dem Wort „Gras" zu „Gra" und somit zu einem Fehler führen. Anwendungsabhängig kann es sein, dass dieses Stemming trotz der Fehleranfälligkeit zur Verbesserung der Leistung beiträgt, da dadurch zumindest die Pluralform entfernt und somit die Vielfalt an Wörtern und später die Dimensionen im Feature-Raum reduziert werden.

Neben dem Entfernen der Pluralform gibt es noch mehr Anwendungen, die durch Regeln in Word-Stemming ermöglicht werden. Beispielsweise könnten Zeiten, Fälle oder auch Pronomen vereinfacht werden. Dafür sind bereits einige Word-Stemmer verfügbar [29].[7] Was also mit Word-Stemming versucht wird, ist den Feature-Raum durch Reduktion von Wortvielfalten zu verkleinern, sodass Sätze und Wörter in unterschiedlichen Zeiten, Fällen und Numerus besser verglichen werden können und dadurch die Leistung des NLPS verbessert wird.

Lemmatisierung

Viel fortschrittlicher als Word-Stemming ist die sogenannte *Lemmatisierung* [33].[8] Während beim Word-Stemming nach Regeln vorgegangen wird und so Fehler auftreten können, liegt der Lemmatisierung tatsächlich ein Lexikon zugrunde, auf das bei der Rückführung zur Grundform zurückgegriffen werden kann. Beispielsweise wird der Satz „Wir haben nur zwei Pizzen gegessen“ in die vereinfachte Wortsequenz „PRONOM hat nur zwei Pizza essen“ umgewandelt. Zu bemerken ist, dass dadurch die Verallgemeinerung der Wörter enorm ist. Das Subjekt „wir“ wird soweit reduziert, dass es mit „ich“ oder anderen Pronomen gleichgesetzt wird. Außerdem lässt sich erkennen, dass die durch die verwendete Zeit abgewandelten Wörter (aus „haben“ wird „hat“) zurückgeführt werden und auch der Plural korrekt in Singular verwandelt wird.

Durch Lemmatisierung vereinfachte Sätze sind also tendenziell weitaus weniger fehlerhaft als die, bei denen Word-Stemming verwendet wird, jedoch ist die Lemmatisierung ein viel rechenintensiverer Prozess, der bei der Verarbeitung großer Datenmengen womöglich zur Verlangsamung des Systems führt.

Stoppwortfilterung

Eine weitere, wichtige Möglichkeit Texte zu vereinfachen bzw. Feature-Räume zu reduzieren ist die *Stoppwortfilterung* [31]. Dabei werden jene Wörter aus dem vorliegenden Text entfernt, die tendenziell keine Relevanz über die Aussage des Textes haben. Derartige Wörter können beispielsweise Artikel, Konjunktionen oder Präpositionen sein, die vollkommen unabhängig von der Semantik in allen Texten vorkommen können und so keinen Beitrag zur Klassifikation leisten. Die Filterung selbst erfolgt über sogenannte Stoppwortlisten[9], die mit dem Text abgegelichen werden. Übereinstimmende Wörter werden dabei entfernt.

Tagging

Aus der hier vorgestellten Reihe der Textaufbereitungsverfahren ist das *Tagging* [11, 16], auch *CLAWS* (Constituent Likelihood Automatic Word-Tagging System) oder *POS-Tagging* (Part-of-Speech Tagging) genannt, das wohl fortschrittlichste. Dabei werden die Tokens des Satzes mit ihrer grammatikalischen Bedeutung markiert:

[7] Beispiele in Python: http://www.nltk.org/howto/stem.html

[8] Lemmatisierung ist ein Begriff der Lexikographie und drückt den Vorgang aus, ein abgewandeltes Wort in das Grundwort zurückzuführen.

[9] Beispiel für Python: http://www.nltk.org/howto/corpus.html#word-lists-and-lexicons

Satz 1:	„Ich möchte die Pizza Diavolo.“							
Satz 2:	„Wir möchten die Pizza Margherita.“							
Wörterbuch:	Ich	möchte	die	Pizza	Diavolo	Wir	möchten	Margherita
Vektor 1:	1	1	1	1	1	0	0	0
Vektor 2:	0	0	1	1	0	1	1	1

Abbildung 2.12: Beispiel eines BOW Aufbaus mit den entsprechenden Feature-Vektoren.

```
tokens = tokenize("Ich möchte, dass ...");
  // ["Ich", "möchte", ",", "dass", "..."]
pos_tag(tokens);
  // [["Ich", "PN"], ["möchte", "VB"], [",", "SZ"], ["dass", "KJ"], ["...", "SZ"]]
```

Das ergibt den Vorteil, dass weitere, wichtige Features aus dem Text extrahiert werden können und so nicht nur eine Wort-, sondern auch eine Grammatikverarbeitung durchführbar ist. So ist es also möglich beispielsweise das Verb „arbeiten“ vom Nomen „Arbeiten“ in Plural zu unterscheiden. Zudem ist es mit Tagging möglich, Sätze in deren Gliedsätze aufzuteilen und so gezielt auf die jeweiligen Satzfragmente einzugehen.

2.2.4 Feature Extraction

Ein elementares Glied in der NLP-Kette ist die *Feature Extraction* [14], bei der versucht wird, aus den vorliegenden Textdaten so viel Information wie möglich zu extrahieren und so einen möglichst qualitativen Feature-Raum zu erzeugen. Im Zuge der Leistungsoptimierung (Genauigkeit und Rechenintensität) ist dabei wichtig den Feature-Raum so groß wie nötig, aber so klein wie möglich zu halten, was bei Textdaten aufgrund ihrer enormen Vielfältigkeit eine große Herausforderung darstellt. Einige der vielen Techniken die dazu entwickelt wurden, sind im Folgenden angeführt.

Bag of Words

Eine eher triviale Methode ist das Ermitteln der Häufigkeit von Wörtern in der vorliegenden Textdatenmenge. Diese Technik startet mit einem leeren Wörterbuch, das den Feature-Raum repräsentiert. Beim Parsen der Textdaten wird bei jedem Wort geprüft, ob es bereits im Wörterbuch aufgenommen wurde. Ist dies nicht der Fall, wird ein neuer Eintrag, also eine neue Dimension im Feature-Raum, hinzugefügt. Gleichzeitig wird beim Feature-Vektor des gerade verarbeiteten Textes, bei dem Wort bzw. der Dimension der Wert erhöht. Schlussendlich gibt es ein großes Wörterbuch, den Feature-Raum, und so viele Feature-Vektoren wie Texte. Der Name *Bag of Words* (BOW) leitet sich übrigens von der Unordnung der Wörter im Feature-Vektor und Wörterbuch (wie ein Sack voll Wörter) ab. Abbildung 2.12 zeigt ein Beispiel anhand zweier Sätze. Darin ist zu erkennen, in welcher Reihenfolge sich das Wörterbuch aufbaut und wie zuletzt

Satz:	,,Ich möchte die Pizza Diavolo.“
Wörterbuch:	Ich, möchte, Ich möchte, die, möchte die, Pizza, die Pizza, Diavolo, Pizza Diavolo

Abbildung 2.13: Beispiel eines NG Aufbaus mit 1 als Unter- und 2 als Obergrenze.

die Feature-Vektoren der Sätze zusammengesetzt sind. Mit diesem Beispiel lassen sich bereits einige Probleme veranschaulichen.

Zunächst kann festgestellt werden, dass, ohne entsprechende Textaufbereitung, das Wort „möchte“ in Singular und Plural enthalten ist, was in den meisten Fällen zu Ungenauigkeiten führen kann. Zudem ist der Artikel „die“ im Wörterbuch vorhanden. Da beide Sätze den Artikel beinhalten, wird dieses Wort zur Klassifizierung bzw. Verarbeitung im NLPS keinen Beitrag leisten können und sollte z. B. mittels Stoppwortfilterung im Vorfeld entfernt werden. Ein weiteres Problem, das BOW mit sich bringt ist, dass keine Wortpaare und Eigennamen erkannt werden, wie beispielsweise „Fachhochschule Hagenberg“. Vor allem werden Wortpaare mit Verneinungen, wie „keine Pizza“, ignoriert. Schließlich ist noch ein, im Zuge des BOW Verfahrens inhärentes Problem feststellbar. Die Feature-Vektoren werden bei sehr umfangreichen Textmengen nur wenige Features enthalten, was bei der Wahl des ML-Algorithmus bedacht werden muss.

N-Gramme

Eine Weiterentwicklung von BOW sind *N-Gramme* (NG), mit denen versucht wird, Wortpaare und Eigennamen stärker zu gewichten. Das Prinzip ist dabei sehr ähnlich dem des BOW, jedoch werden bei NG Wörterbücher aufgebaut, die Einträge mit mehreren Wörtern erlauben. Bei NG wird dabei eine Unter- und Obergrenze definiert. Die Untergrenze legt fest aus wie vielen Wörtern die Wörterbucheinträge zumindest bestehen müssen, und die Obergrenze gibt die Maximalwortzahl an. Abbildung 2.13 zeigt ein Beispiel von NG mit 1 als Unter- und 2 als Obergrenze. Es ist leicht vorstellbar, dass bei NG und entsprechender Parameterwahl das Wörterbuch um ein Vielfaches größer wird. Auch hier eignet sich GS um die optimale Parameterpaarung zu finden und so das beste Verhältnis aus Rechenintensität und Genauigkeit des NLPS zu finden.

TF-IDF

Eine Möglichkeit die Token bzw. Tokenpaare zu gewichten, stellt *Term Frequency – Inverse Document Frequency* (TF-IDF) dar. Dabei wird beim Erstellen des Wörterbuchs nicht nur der Token selbst, sondern auch die Häufigkeit in der gesamten Textmenge eingetragen. Der Wert im Feature-Vektor für das jeweilige Wort ergibt sich dann aus dem Verhältnis zwischen der Häufigkeit des Tokens im Text, zur Gesamthäufigkeit in allen Textdaten. Auf diese Weise werden häufig vorkommende Wörter wenig gewichtet und wirken sich im NLPS nicht mehr so stark aus.

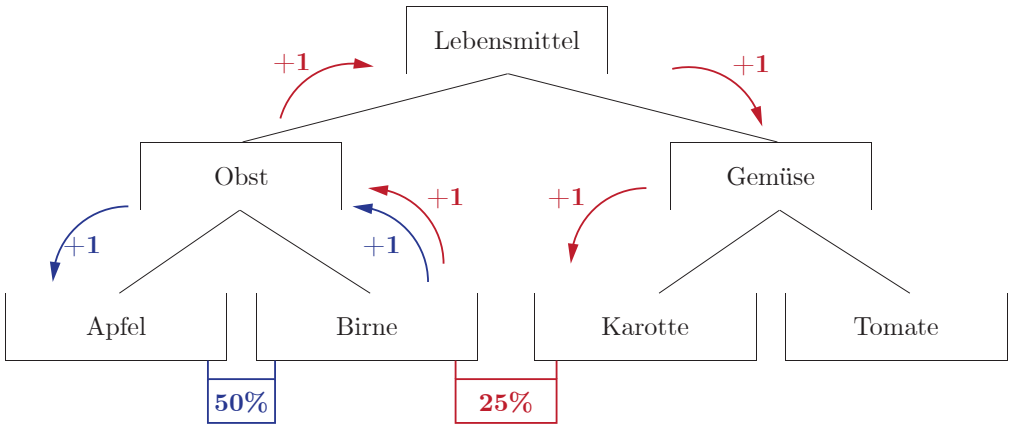


Abbildung 2.14: Vereinfachte Darstellung der Funktionsweise für die Ermittlung von Wortähnlichkeiten in WordNet.

WordNet und Wortähnlichkeiten

Obwohl das hier beschriebene Verfahren zwar nicht direkt für die Extraktion von Features verwendet werden kann, wird es im Zuge dieser Arbeit eingesetzt, um die Genauigkeit und Vielseitigkeit davon erheblich zu verbessern. Die Rede ist dabei von der Findung von Wortähnlichkeiten und Synonymen. Bei diesem Verfahren wird ein digitales Lexikon verwendet, beispielsweise WordNet[10] [19]. Dieses enthält nicht nur einfache Synonymlisten zu den Worteinträgen, sondern bildet auch deren Taxonomie ab. So ist es möglich nicht nur Synonyme, sondern auch deren prozentuelle Ähnlichkeit zu ermitteln und Wörter miteinander zu vergleichen. Dazu gibt es einige Vorgehensweisen[11] [23]. Eine davon, die „Path Similarity“, ist in Abbildung 2.14 sehr vereinfacht dargestellt. Dabei wird gezählt wie viele Kanten in der Struktur der Wörter passiert werden müssen, um die gesuchten Wörter zu verbinden. In diesem vereinfachten Beispiel sind die Wörter „Apfel“ und „Birne“ im Verhältnis eins zu zwei ähnlich, da sie einen gemeinsamen Elternknoten haben. Die Wörter „Birne“ und „Karotte“ erfordern einen längeren Weg und haben daher nur eine Ähnlichkeit von eins zu vier. Die tatsächliche Rechenweise sowie die Struktur des WordNets ist jedoch viel komplexer als in diesem Erklärungsbeispiel.

Chunking und Chinking

Um gezielt Worttypen und Grammatik für die Extraktion der Features verwenden zu können, bedienen sich die folgenden Methoden des Taggings der vorliegenden Texte.

Beim Chunking [1][26] werden Abfolgen von Satzfragmenten, also Gliedsätze, Wörter und Satzzeichen, in dem durch Tagging aufbereiteten Text gesucht. Das folgende Pseudocode-Beispiel zeigt, wie die Sequenz von Determinativ, gefolgt von zwei Nomen, gesucht wird.[12]

[10] https://wordnet.princeton.edu/

[11] Beispiel für Python: http://www.nltk.org/howto/wordnet.html#similarity

[12] Beispiel der verwendbaren WordNet Satzfregmente in Python: https://pythonprogramming.net/natural-language-toolkit-nltk-part-speech-tagging/

```
tokens = tokenize("Ich möchte eine Pizza Diavolo.");
tagged = pos_tag(tokens);
chunked = chunk(tokens, "/<DT.*>+<NN.*>+<NN.*>+/");
  // [["eine", "DT"], ["Pizza", "NN"], ["Diavolo", "NN"]]
```

Chinking [2] ist sozusagen der Gegenspieler zu Chunking und liefert nicht alle Treffer, sondern filtert die Treffer vom Ergebnis und liefert das, was übrig bleibt. Es kann also als erweiterte Stoppwortfilterung gesehen werden, in der nicht nur Wörter anhand einer Liste, sondern auch anhand ihrer Position im Satz bzw. ihrer Grammatik oder anhand angrenzender Wörter gefiltert werden können.

2.3 Natural Language Processing Services

Bis zu diesem Punkt wurde intensiv auf die Theorie über die Ansätze von ML und deren Potential und vielseitige Einsatzgebiete, das Konzept von NLPS, das Arbeiten mit und Aufbereiten von Text und die Möglichkeiten, Informationen aus Textdaten zu extrahieren, eingegangen. Nun sollen verfügbare Onlineservices vorgestellt werden, die diese Disziplinen zusammenführen, um einen vollwertigen NLPS aufzubauen. Im weiteren Verlauf dieser Arbeit wird zusätzlich ein selbstimplementierter Prototyp eines NLPS entwickelt, mit dem diese Services verglichen werden. Für diesen Zweck muss sichergestellt sein, dass alle NLPS für das Interpretieren kurzer Befehle in menschlicher Sprache ausgelegt sind und eine möglichst große und einheitliche Schnittmenge an interpretierten Attributen zurückliefern. Da der im Zuge dieser Arbeit entwickelte Prototyp speziell für eine Domäne entwickelt wird, müssen zumindest Intents und Entities in den NLPS trainierbar und in den Antworten vorhanden sein.

Vor allem in den letzten Jahren wurden einige dafür geeignete NLPS vorgestellt, die eine API für Entwickler bereitstellen [32]. Auch große Firmen wie *Google*, *Microsoft* und *Amazon* veröffentlichten entsprechende Softwares. Ein Auszug der verfügbaren NLPS wird im Folgenden vorgestellt.

2.3.1 Microsoft LUIS

Eine der größten Herausforderungen eines NLPS ist es, sehr allgemein und vielseitig einsetzbar und dabei möglichst treffsicher zu sein. Dies setzt gute Trainingsmöglichkeiten voraus, um den Service auf das Einsatzszenario, mit entsprechenden Intents und Entities, einstellen zu können. Die Software Language Understanding Intelligent Service[13] (LUIS) von *Microsoft* stellt dafür eine sehr anwenderfreundliche Möglichkeit des Trainings zur Verfügung. Die Benutzeroberfläche, die in Abbildung 2.15 dargestellt ist, wirkt sehr übersichtlich und das Training erfolgt intuitiv. Ein Weg das System zu trainieren ist, zunächst die Entities einzutragen. Dabei bietet LUIS grundsätzlich die Möglichkeit an, aus vorgefertigten Entities, wie Datum, Telefonnummern, Städtenamen etc., zu wählen, oder eigene Entites anzulegen. Dabei kann aus vier Entity-Typen gewählt werden [35]:

- **Simple:** Das System soll den Entity-Typ intelligent ermitteln. Dazu wird untersucht, wie die Entity in den Trainingsdaten eingesetzt wird.

[13] https://www.luis.ai

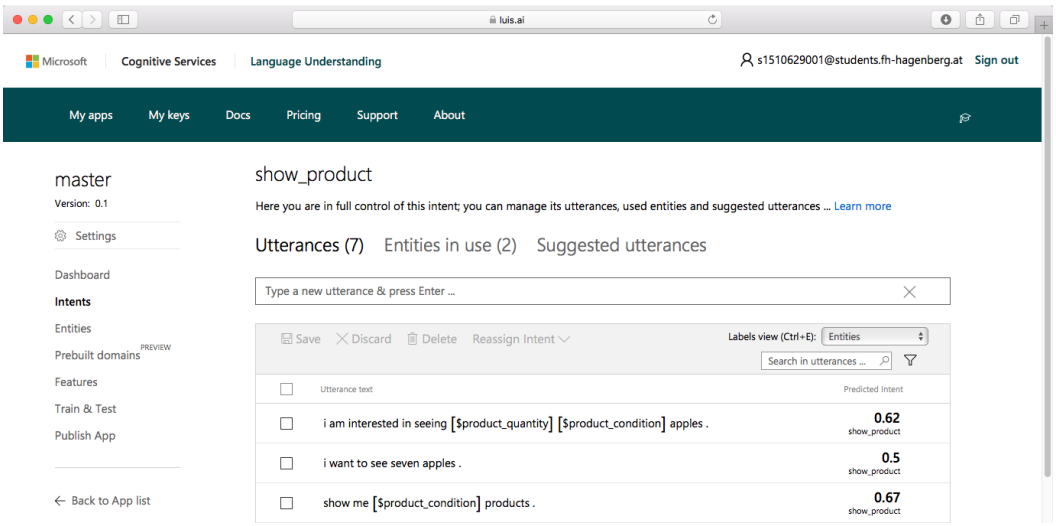


Abbildung 2.15: Benutzeroberfläche von LUIS.

- **List:** Falls alle Einträge dieser Entity bekannt sind, beispielsweise bei Produktnamen, können diese als Liste hinterlegt werden. In die entsprechende Tabelle werden in der ersten Spalte die Begriffe und in der zweiten Spalte deren Synonyme und Variationen, wie beispielsweise Plural oder durch den Fall veränderte Formen, eingetragen. Dabei ist wichtig zu wissen, dass LUIS keine intelligente Erkennung vornimmt, was bedeutet, dass tatsächlich alle möglichen Variationen in der Tabelle enthalten sein müssen.
- **Hierarchical:** Um zwischen den Entities Hierarchien aufbauen zu können, gibt es diesen Typ. Dabei können der Haupt-Entity bis zu 20 Kind-Entities untergeordnet werden, jedoch sind diese alle auf derselben Ebene. Die Werte der Entities werden auch hier intelligent zugeordnet.
- **Composite:** Damit können Entities vom Typ „List“ und „Simple“ als Kind-Elemente verknüpft werden, um eine fortgeschrittene Hierarchie aufzubauen. Diese ist jedoch auch auf zwei Ebenen begrenzt.

Sind die Entities eingetragen, können die entsprechenden Intents angelegt werden. Nach Festlegen des Intent-Namens, können in dem Eingabefeld der dargestellten Maske, die dazugehörigen Trainingsbefehle eingetragen werden. Diese Befehle sammeln sich in der darunterliegenden Liste, in der die Wörter ausgewählt und dann mit den bereits angelegten Entities verknüpft werden können. Wenn alle Entities und Intents angelegt sind, muss der NLPS noch mittels Knopfdruck trainiert werden und LUIS ist einsatzbereit. Eine beispielhafte Antwort von LUIS auf die Client-Anfrage mit dem Befehlstext „I want to order ten ripe apples.“ ist in folgendem JSON-Objekt angeführt:

```
{
  "query": "I want to order ten ripe apples.",
  "topScoringIntent": {
    "intent": "show_availability",
    "score": 0.347629845
  },
```

```
    "intents": [
      {
        "intent": "show_availability",
        "score": 0.347629845
      },
      {
        "intent": "order_product",
        "score": 0.205836952
      },
      {
        "intent": "show_seller",
        "score": 0.107168071
      },
      {
        "intent": "show_product",
        "score": 0.09204431
      },
      {
        "intent": "None",
        "score": 0.00381934969
      }
    ],
    "entities": [
      {
        "entity": "ripe",
        "type": "product_condition",
        "startIndex": 20,
        "endIndex": 23,
        "resolution": {
          "values": [
            "ripe"
          ]
        }
      }
    ]
  }
```


Aus Gründen der Vollständigkeit wird der angefragte Befehl in `query` wiederholt. Den Namen und die Trefferwahrscheinlichkeit des Intents ist in `topScoringIntent` enthalten. Um Fehlzuweisungen durch das angebundene System leichter korrigieren zu können, wird eine Übersicht an allen Intent-Wahrscheinlichkeiten unter `intents` bereitgestellt. Auch die `entities` werden übertragen, und enthalten dabei den Typ, den Wert und die Position des Wortes im angefragten Befehl.

Für den ganzen Vorgang gäbe es auch eine Programmschnittstelle, auf die jedoch in dieser Arbeit nicht weiter eingegangen wird. LUIS unterstützt 12 Sprachen, darunter Deutsch und Englisch, und hält 21 voreingestellte Domains, wie „Wetter“, „Sport“ und „Messaging“, bereit.

2.3.2 API.AI

Ein sehr vielseitiger NLPS ist API.AI[14], der von der Firma *Speaktoit* entwickelt und 2016 von *Google* gekauft wurde [34]. Auch dieses System ist intuitiv und ähnlich wie LUIS

[14] https://api.ai

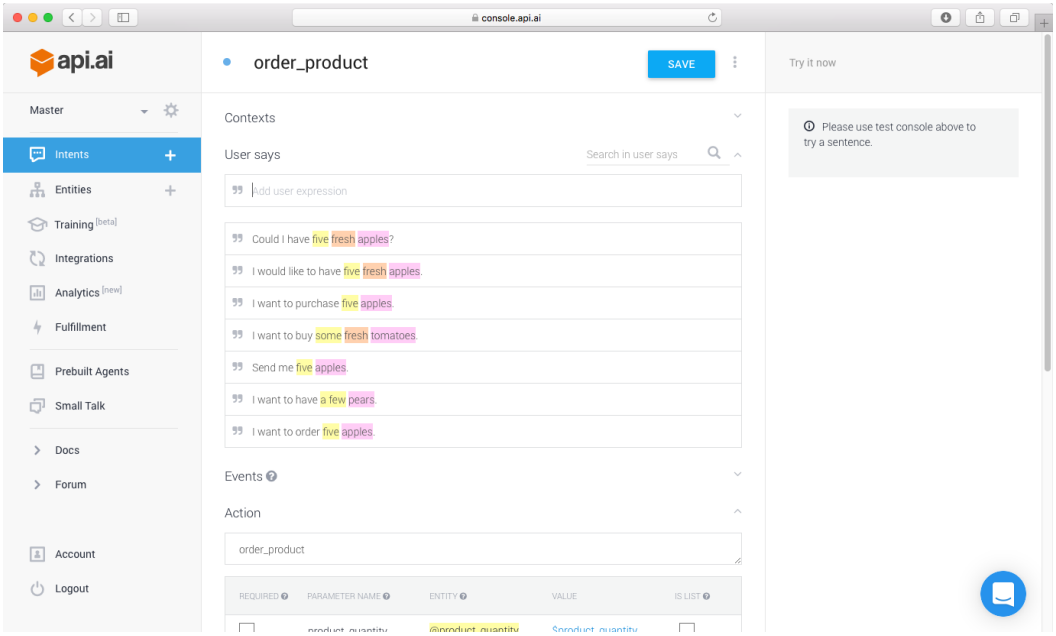


Abbildung 2.16: Benutzeroberfläche von API.AI.

bedienbar, und kann auf ähnliche Weise trainiert bzw. konfiguriert werden. Allerdings hält API.AI wesentlich mehr Funktionalitäten bereit.

Auch bei API.AI können zunächst Entities angelegt werden. Hier kann jedoch nicht aus verschiedenen Entity-Typen gewählt werden. Es gibt nur einen Typ, der „List“ von LUIS ähnelt, also auch Wörter und deren Synonyme in einer Tabelle festhält. Allerdings gibt es die Möglichkeit „Allow automated expansion“, mit der gesteuert werden kann, ob ähnliche Wörter bzw. Variationen des Wortes automatisch ergänzt werden dürfen. Dadurch entsteht ein Entity-Typ, der zwar ähnlich wie „Simple“ bei LUIS, jedoch viel besser konfigurierbar ist, da hier viele mögliche Entity-Wörter trainiert werden können. Anschließend können, analog zu LUIS, Intents definiert und mit Trainingssätzen trainiert werden, wie in Abbildung 2.16 dargestellt. Auch bei API.AI werden dabei die Wörter markiert und mit den Entities gekennzeichnet. Allerdings gibt es hier viel mehr Möglichkeiten bei der Entity- und Intent-Konfiguration:

- **Standardwerte für Entities:** Werden Entities im eingegebenen Text nicht erkannt, können dafür Standardwerte hinterlegt werden, die das NLPS in diesem Fall an das angebundene System übermittelt.
- **Erforderliche Entities:** Um beispielsweise eine Pizzabestellung abschließen zu können, sind zumindest die Daten des Pizzatyps, der Menge und der Anschrift des Kunden erforderlich. Nur wenn alle Informationen übermittelt wurden, ist die Bestellung durchführbar. In API.AI gibt es die Möglichkeit, Entities als erforderlich zu markieren. Werden diese erforderlichen Entities im eingegebenen Text nicht erkannt, wird der Benutzer mittels der verfügbaren Antworterzeugung aufgefordert, diese noch zu übermitteln.
- **Antworterzeugung:** In einem realen Szenario, bei dem Benutzer mit einem

NLPS bzw. Messenger Bot schreiben, wird eine Rückmeldung von diesem System erwartet, um sicher gehen zu können, dass dieses die Daten erhalten und verstanden hat. Dazu stellt API.AI ein Eingabefeld zur Verfügung, in dem einige mögliche Antworten in menschlicher Sprache eingegeben werden können. Es ist dabei möglich, die identifizierten Entity-Werte als Parameter zu verwenden.

- **Kontext:** Angenommen bei dem Pizzaservice-Beispiel würde der Typ der Pizza nicht erkannt, und über die Antworterzeugung dieser nachgefragt werden, dann wäre die Antwort des Benutzers wohl nur „Diavolo“. Damit das System den Zusammenhang mit der vorangegangenen Eingabe herstellen kann, ist es bei API.AI möglich, Kontexte zu setzen.
- **HTTP-Response:** Die Antwort des NLPS wird im JSON-Format an das angebundene System übermittelt. Dafür können hier die Variablennamen im JSON-Objekt bestimmt werden, sodass diese evtl. sprechender benannt sind.

Neben der umfangreichen Konfigurationsmöglichkeit stellt API.AI noch eine „Speech-To-Text“-Funktion zur Verfügung, sodass die Benutzereingabe auch akustisch erfolgen kann. Daher ist zu erkennen, dass API.AI nicht nur den Teil „Textinterpretation“ aus Abbildung 1.1 bereitstellt, sondern auch „Speech-To-Text“, Teile eines „Dialogmanagers“ und eine „Texterstellung“.

Auch API.AI hat eine Programmierschnittstelle, über die der sogenannte NLPS-Agent konfiguriert werden kann. Außerdem kann ein Webhook hinterlegt werden, der bei Benutzeranfragen automatisch vom NLPS benachrichtigt wird. Die Antwort von API.AI auf denselben Befehl wie im vorangegangenen Beispiel ist im folgenden JSON-Objekt dargestellt:


```
{
  "id": "07bcf490-1b81-4c3b-99cd-987c48f25522",
  "timestamp": "2017-05-22T06:13:22.352Z",
  "lang": "en",
  "result": {
    "source": "agent",
    "resolvedQuery": "I want to order ten ripe apples.",
    "action": "order_product",
    "actionIncomplete": false,
    "parameters": {
      "product_condition": "ripe",
      "product_name": "Apple",
      "product_quantity": "ten"
    },
    "contexts": [],
    "metadata": {
      "intentId": "d3dc79b5-00aa-4731-96a0-67b3f16e6ad3",
      "webhookUsed": "false",
      "webhookForSlotFillingUsed": "false",
      "intentName": "order_product"
    },
    "fulfillment": {
      "speech": "",
      "messages": [
        {
          "type": 0,
          "speech": ""
        }
```

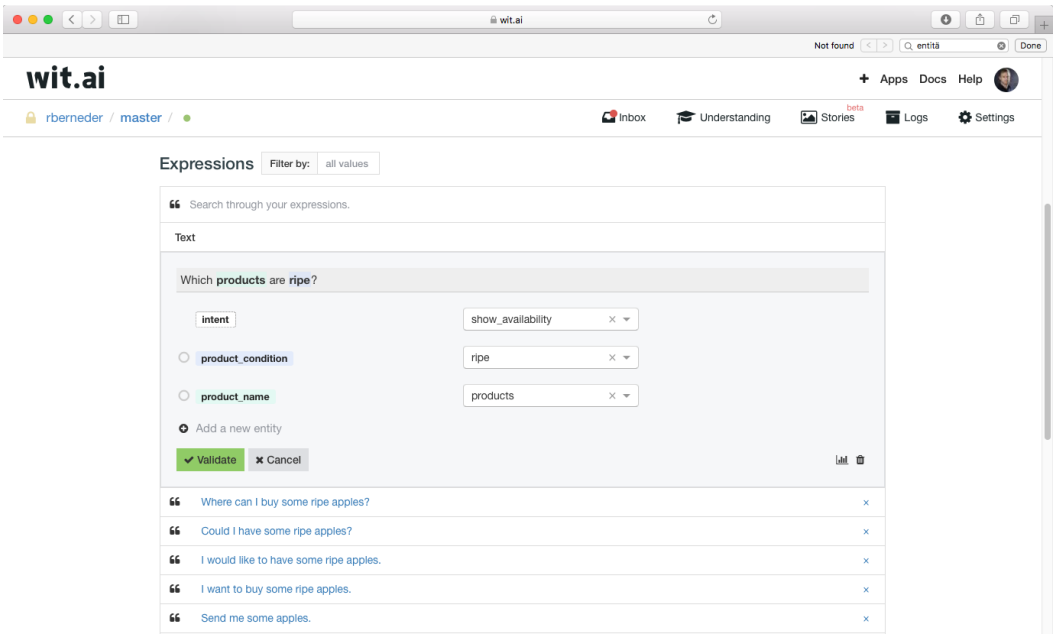


Abbildung 2.17: Benutzeroberfläche von WIT.

```
        ]
      },
      "score": 1.0
    },
    "status": {
      "code": 200,
      "errorType": "success"
    },
    "sessionId": "e8a9c997-6e87-2b8d-268c-681a24fa6633"
  }
```

Es fällt sofort auf, dass in diesem NLPS wesentlich mehr Informationen enthalten sind. Die `id`, die Sprache `lang` und der `timestamp` der Abfrage sowie die `sessionId` des Users können programmatisch weiterverwendet und z. B. zur Protokollierung verwendet werden. Der `status` erleichtert das Debugging und `result` enthält die eigentliche Payload. Unter anderem sind darin die Entities, der Intent und Informationen auf den Dialog- und Kontextstatus enthalten.

API.AI ist für 15 Sprachen verfügbar, darunter Deutsch und Englisch, enthält 10 voreingestellte Domains, die zuschaltbar sind, und hat vorgefertigte Schnittstellen für die Integration in 15 verschiedenen Social Media Plattformen bzw. Messengers.

2.3.3 WIT

Bei WIT[15] werden Entities ähnlich verwaltet wie bei den anderen vorgestellten NLPS, wie in Abbildung 2.17 gezeigt. Allerdings gibt es bei den Intents ein eigenes Konzept. Die Intents müssen hier nicht zwingend verwendet werden und können als „Traits“ einem

[15] https://wit.ai

Trainingsbefehl hinzugefügt werden. Dazu wird eine Liste an möglichen Werten angelegt, in diesem Fall sind das die Intent-Namen, und ein konkreter Wert für den eingegebenen Trainingssatz ausgewählt. Dieses System ermöglicht eine viel flexiblere Handhabung der Intents, da theoretisch mehrere zuweisbar sind. Entities sind dagegen sehr ähnlich zu pflegen wie bei API.AI, da auch hier eine Liste an Wörtern und deren Synonymen geführt wird und eingestellt werden kann, ob diese Wörter- bzw. Synonymliste automatisch vom System erweitert werden darf. Die Kernkompetenz bei WIT liegt auf dem Dialogsystem, mit dem mehrstufige Unterhaltungen vorkonfiguriert werden können. Bei diesen „Stories“, wie dieses Konzept bei WIT genannt wird, können alternierend User- und Bot-Aussagen eingegeben werden. Dabei ist es möglich, Bedingungen zu setzen, die den Bot in der Konversation zurück- bzw. nach vor springen lassen, beispielsweise, wenn noch nicht genug Daten eingegeben wurden, um die gewünschte Aktion auszuführen. Demnach ist auch WIT mit mehr Komponenten, die in Abbildung 1.1 gezeigt sind, ausgestattet. Auf den Beispielbefehl liefert WIT eine etwas spärlichere, im Zuge dieser Arbeit aber völlig ausreichende Antwort:


```
{
  "msg_id": "0idHrXCCSNB1SzSo4",
  "_text": "I want to order ten ripe apples.",
  "entities": {
    "product_condition": [
      {
        "confidence": 0.98214110157072,
        "value": "ripe",
        "type": "value"
      }
    ],
    "product_name": [
      {
        "confidence": 0.99586552997596,
        "value": "apples",
        "type": "value"
      }
    ],
    "intent": [
      {
        "confidence": 0.94913300262228,
        "value": "order_product"
      }
    ]
  }
}
```


Zur Zuweisbarkeit und programmatischen Verwendung wird die `msg_id` übermittelt. Der Ausgangstext und die erkannten `entities` werden ebenso angehängt. Zu erkennen ist, dass es, wie angemerkt, keinen eigenen Eintrag für den Intent gibt, sondern dieser aufgrund des Konzeptes von WIT unter den Entities aufscheint.

WIT kann 53 Sprachen verarbeiten, darunter auch Deutsch und Englisch, verfügt jedoch nicht über voreingestellte Domains und hat keine vorbereiteten Anbindungsmöglichkeiten an Social Media Systeme.

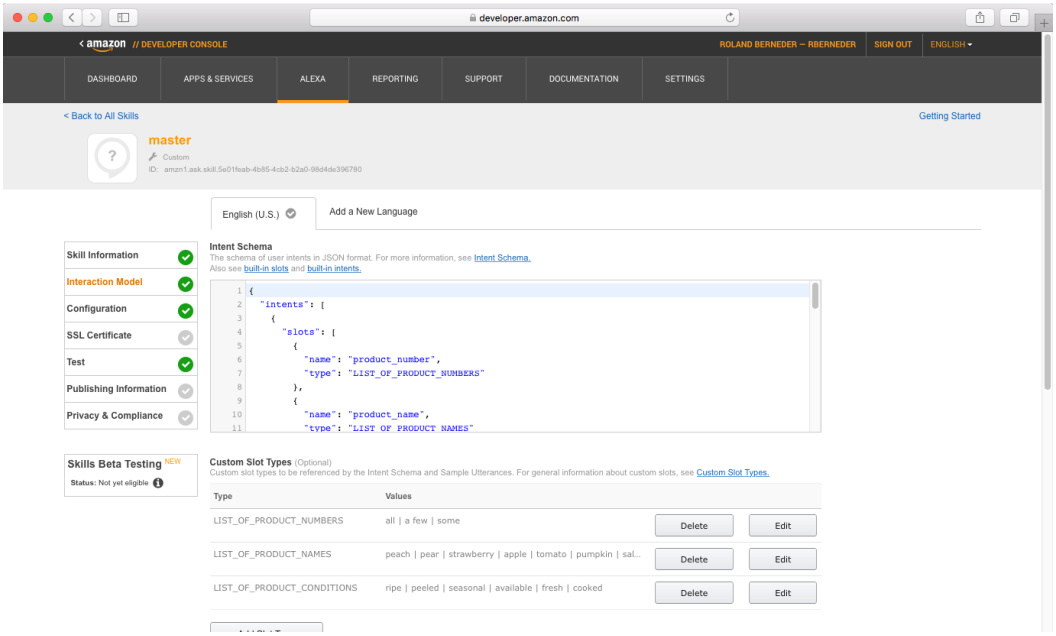


Abbildung 2.18: Wenig benutzerfreundliche und unübersichtliche Oberfläche von ALEXA.

2.3.4 ALEXA

ALEXA[16] von *Amazon* ist zwar kein reiner NLPS, weil der Service nur im Zuge von *Amazon Echo* erreichbar ist. Da jedoch ein Entwickler-Webhook verbunden werden kann, dem die Intents und Entities bei Abfragen übermittelt werden, soll auch ALEXA in dieser Arbeit bewertet werden.

Gleich am Beginn der Entwicklung eines sogenannten *Alexa Skill*, also einer NLP-Instanz, lässt sich feststellen, dass ALEXA noch keine ausgereifte Benutzeroberfläche bietet, wie Abbildung 2.18 zeigt. In drei kleine Textboxen müssen Intents, Entities und Trainingsbefehle im JSON-Format bzw. in der entsprechenden, von *Amazon* vorgegebenen Konvention eingetragen werden. In Verbindung mit den Produkten von *Amazon* gibt es zwar eine Speech-To-Text-Einheit, jedoch ist hier kein Erzeugen eines Dialogs möglich. Die Liste an Funktionen und Einstellungen ist hier also verhältnismäßig kurz. Wird eine Anfrage mit einem beispielhaften Befehl von ALEXA verarbeitet und an den Webhook übermittelt, sieht dieses JSON-Objekt wie folgt aus:

```
{
  "type": "IntentRequest",
  "requestId": "EdwRequestId.305d2c23-d781-44ad-884f-2a6d8630e2ac",
  "locale": "en-US",
  "timestamp": "2017-05-23T08:15:29Z",
  "intent": {
    "name": "order_product",
    "slots": {
      "product_condition": {
```

[16] https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/getting-started-guide

```
          "name": "product_condition",
          "value": "ripe"
        },
        "product_name": {
          "name": "product_name",
          "value": "apples."
        },
        "product_number": {
          "name": "product_number"
        }
      }
    }
  }
```

Das Ergebnis ist also sehr ähnlich dem von WIT, jedoch werden hier die Trefferwahrscheinlichkeiten nicht ausgegeben.

ALEXA unterstützt derzeit nur Deutsch und Englisch, verfügt nicht über voreingestellte Domains und hat, da es eine prorprietäre Software von *Amazon* ist, die für die firmeneigenen Produkte verwendet werden soll, keine Anbindungsmöglichkeit an Social Media Systemen.

2.4 Aufgabenstellung

NLPS wurden im Zuge der Entwicklungen elektronischer Assistenten und der Verbesserung der Benutzerfreundlichkeit von Applikationen geschaffen. Nun stellt sich aber die Frage, wie gut sich diese NLPS in einem realistischen, praxisnahen Szenario einsetzen lassen. Dabei sind einige Faktoren, die für diese Beurteilung herangezogen werden, zu berücksichtigen:

- **Einstellbarkeit:** Da NLPS für die Textverarbeitung eingesetzt werden, wird von ihnen erwartet, mit den enormen Eingabemöglichkeiten und Textvariationen der Benutzer umgehen und daraus richtige Ergebnisse ermitteln zu können. Demnach ist es naheliegend, dass möglichst viel Logik hinsichtlich der Textinterpretation auf den NLPS ausgelagert wird. Dies macht es erforderlich, das System gemäß der erwarteten Ausgabe einstellen zu können. Daher ist die Flexibilität der Einstellbarkeit ein wichtiges Kriterium bei der Beurteilung eines NLPS. Beispielsweise könnte die Bildung einer mehrstufigen Hierarchie von Entities für einen Anwendungsfall mit entsprechender Taxonomie wichtig sein.
- **Interpretationsqualität:** Ist der NLPS eingestellt, ist im nächsten Schritt die Interpretationsqualität basierend auf der zur Verfügung stehenden Trainingsdatenmenge interessant. Dies ist wohl der schwierigste Teil an einem NLPS, da es zahlreiche Herausforderungen zu meistern gilt:
  - *Kontext und Eigennamen:* Die Bedeutung eines Wortes hängt nicht immer nur vom Wort alleine, sondern oft auch vom Zusammenhang ab. Ein Beispiel dafür ist der Satz „Ich brauche die Kohle um mir ein Auto zu kaufen.“, bei dem mit dem Wort „Kohle“ Geld und nicht der Brennstoff gemeint ist. Auch Eigennamen erschweren die richtige Erkennung der Entities. Im Englischen darf beispielsweise im Satz „Apple has a new product.“ keine Frucht erkannt werden.

  - *Eingabequalität des Benutzers:* Gerade bei der digitalen Texteingabe kommen die Großschreibung und das Verwenden der Satzzeichen häufig aus Faulheit zu kurz. Allgemein können in der Eingabe Grammatik- und Rechtschreibfehler enthalten sein. Natürlich ist die Qualität der Eingabe aber für den NLPS von großer Bedeutung. Während Rechtschreibfehler in Wörtern verhältnismäßig gut korrigiert werden können [24], sind Grammatikfehler wesentlich komplexer zu handhaben.

    Auch Sonderzeichen sowie verwendete Emoticons oder Zeichenfehler müssen vom NLPS verarbeitet werden können.
  - *Komplexität des Befehls:* Für den Fall, dass der eingegebene Befehl grammatikalisch und rechtschreiblich korrekt vorliegt, kann der Satz komplex aufgebaut und schwer zu interpretieren sein. Doppelte Verneinungen und lange Sätze mit mehreren Gliedsätzen, stellen hier eine besondere Herausforderung dar.
  - *Falscheingaben:* Neben bekannten Intents und Entities, ist es auch möglich, dass Benutzer Befehle eingeben, die dem NLPS fremd sind, beispielsweise wenn ein Pizzabestellservice nach dem Wetter gefragt wird. Der NLPS muss mit dieser Situation umgehen können und darf dabei nicht einfach den wahrscheinlichsten Intent auswählen, der ihm bekannt ist.

- **Ergebnisqualität:** Sind die Intents und Entities aus dem zu interpretierenden Befehl extrahiert, sollen diese dem angebundenen System übermittelt werden. Um dort eine möglichst einfache Weiterverarbeitung zu gewährleisten, sollten die übertragenen Daten entsprechend aufbereitet sein.
  - *Synonyme:* Viele NLPS sind in der Lage die Liste der bekannten Entities bzw. deren Synonyme intelligent zu erweitern. Das bedeutet, dass beispielsweise „Paradeiser“ als Entity erkannt wird, weil „Tomate“ vom angebundenen System über die API in die Entity-Liste des NLPS eingetragen wurde. Es wäre also wünschenswert, dass diesem System jenes Wort übertragen wird, das es auch kennt. In dem Fall also „Tomate“.
  - *Wortvariationen:* Bedingt durch die Grammatikregeln kann es zu einigen Variationen der erkannten Wörter kommen. So können die im Befehl gewählte Zeit, der Fall und der Numerus zu einer Vielfalt an Wortvariationen führen. Diese Variationen sollten um die Weiterverarbeitung zu vereinfachen vom NLPS in die Grundform gebracht werden.
  - *Zahlen:* Auch Zahlen können im Befehl vorkommen. Diese müssen vom NLPS sowohl in Zahl- als auch in ausgeschriebener Textform erkannt werden. Vor der Übermittlung sollte jedoch die Zahlform gebildet werden, da mit dieser im angebundenen System gerechnet werden kann.

Es gibt also einige Qualitätsmerkmale, die in realistischen Szenarien wichtig bzw. für Entwickler eines Systems, das NLPS verwendet, wünschenswert sind. Es soll in dieser Arbeit überprüft werden, wie sich die vorgestellten NLPS anhand dieser Gesichtspunkte bewähren und verglichen werden, ob ein selbstimplementiertes System eine bessere Trefferquote erzielt. Einen derartigen Prototyp zu entwickeln ist aufgrund der derzeitigen Forschungs- bzw. Literatursituation kein leichtes Unterfangen. Über die Jahrzehnte

sammelten sich zwar viele wissenschaftliche Arbeiten und Bücher zum Thema ML und NLP, speziell zum Thema TM, jedoch gibt es nur spärliche Informationen über Systeme, die sich mit der Interpretation von kurzen Befehlen auseinandersetzen und auf einen NLPS abzielen. Der wissenschaftliche Beitrag dieser Arbeit ist demnach, bestehende NLPS auf Ihre Einsetzbarkeit zu prüfen, ein eigenes System zu implementieren, dieses zu dokumentieren und die Ergebnisse gegenüberzustellen und auszuwerten.

2.5 Programmiersprache

Für die Implementierung eines Prototyps ist zu Beginn die Programmiersprache zu wählen. Für Java sind Pakete wie *Apache OpenNLP* oder *Stanford CoreNLP*, die zur grundlegenden Textverarbeitung geeignet sind, sowie *Apache Lucene* für die Tokenization, verfügbar. Letzteres gibt es auch in C. Ein vielversprechendes System auf C++ ist *Vowpal Wabbit*, das ein Command-Line Interface anbietet. Auch R stellt mit dem Skript-Interface, den einfachen Möglichkeiten Daten zu analysieren, und dem *NLP*-Package eine sehr interessante Wahl dar. Wegen der im Folgenden beschriebenen, umfangreichen Module, seiner einfachen Möglichkeiten der Datenmanipulation und nicht zuletzt wegen der Empfehlungen [8, 38], wird für das Projekt jedoch die Programmiersprache Python in der Version 3.6.0 verwendet.

Im Zuge der Entwicklung stellten sich folgende Module als besonders hilfreich heraus:

- `mglearn` ist ein Modul, das von Andreas Müller entwickelt wurde und zahlreiche Funktionen bereitstellt um Daten zu visualisieren. Gerade für Arbeiten mit ML bietet es Hilfestellung.
- `numpy` verwaltet Arrays, erweitert die Python-nativen Funktionen und erleichtert die Datenmanipulation enorm. Iterieren, Filtern, Konkatenieren u. v. m. ist dadurch sehr einfach möglich.
- `sklearn` stellt ein sehr umfangreiches Modul dar, in dem u. a. ML-Algorithmen, Pipeline-Bildung, GS und Feature Extraction Mechanismen enthalten sind.
- `nltk` und `wordnet` machen die Textverarbeitung, Tagging, Tokenization, Ähnlichkeitsvergleiche unter Wörtern etc. möglich.
- `spacy` ist, wie `nltk`, ein weiterer Vertreter der Textverarbeitungsmodule. Gerade die Lemmatisierung funktionierte in den getesteten Szenarien bei `spacy` deutlich besser.
- `sanic` stellt einen minimalistischen Webserver bereit und wird benötigt, um den NLPS via Webbrowser erreichbar zu machen.

Kapitel 3

Testumgebung

Für die Beurteilung der Leistung der zu testenden NLPS ist zunächst ein Szenario erforderlich, das die Grundlage des Trainings und der Tests bilden wird. Wie sich die Leistungsbewertung zusammensetzt, wird im Anschluss erläutert. Zur automatisierten Durchführung dieser Bewertung wird eine Testapplikation geschaffen, an die alle NLPS angebunden werden.

3.1 Testdaten

3.1.1 Sprache

Eine maßgebliche Entscheidung bei der Erstellung von Testdaten ist zu Beginn die Sprachwahl. Obwohl sich bei ML die Sprache wenig auswirkt, hat sie bei der Feature-Extraction und im NLP eine wesentliche Bedeutung. Den zugrundeliegenden Mechanismen bei der Erkennung des Textes müssen die Regeln der Grammatik und die Lexik bekannt sein. Um ein vergleichbares Ergebnis zu erhalten muss bei allen NLPS die Sprache gleich sein. Bei LUIS ist die Sprache Englisch am weitesten entwickelt und beinhaltet die meisten Spezialwörter [36]. Von API.AI und ALEXA gibt es keine Stellungnahme, welche Sprache hier die besten Ergebnisse erzielt. Nach Angaben von WIT sind zwar alle Sprachen sehr gut verwendbar, jedoch wird hier auch Englisch empfohlen, da deren *Linguistic Parser* für diese Sprache am weitesten fortgeschritten ist.[1] Da also Englisch die am besten implementierte Sprache ist, wird das Testszenario in Englisch durchgeführt.

3.1.2 Szenario

Für eine qualitative Bewertung der NLPS muss das gewählte Szenario einige Anforderungen erfüllen:

- **Messbarkeit:** Ganz allgemein müssen die Ergebnisse aus dem Textverarbeitungsprozess eindeutig und messbar sein. Dafür müssen aus den verwendeten Trainings- und Testsätzen die Intents und Entities klar und unverwechselbar hervorgehen um keinen Raum für Interpretationsfreiheit zu lassen.

[1] Aus dem E-Mail von laurent@witai-2.intercom-mail.com am 26.05.2017.

- **Trainingsdatenumfang:** Da NLPS auf ML-Algorithmen aufbauen, und deren Leistung entsprechend vom Umfang und der Qualität der Trainingsdaten abhängig sind, muss ein Szenario gewählt werden, bei dem ausreichend viele Satzvariationen für die verschiedenen Intents erzeugt werden können.
- **Intents:** Die Schwierigkeit die Befehle in die Intents zu kategorisieren ist geknüpft an die Anzahl der zuweisbaren Intents. Um ein anspruchsvolles Szenario zu schaffen, sollten also mehrere Intents angelegt werden.
- **Entities:** Um die NLPS ausgiebig zu prüfen, sollten in den Entities Hierarchien und Synonyme gebildet werden. Außerdem stellen Entities, die sich aus mehreren Wörtern zusammensetzen, ebenso wie Entities die mit Eigennamen verwechselt werden könnten, eine Herausforderung dar. Wenn die verwendeten Entities Nomen sind, sollten auch Adjektive und Verben als Entities eingesetzt werden, um die Extraktion auf mehrere Worttypen zu prüfen. Dies gilt ebenso für Zahlen.

Das Szenario, das diese Anforderungen erfüllt und in dieser Arbeit verwendet wird, handelt von einer Obst- und Gemüseplattform, bei der eine entsprechende Produkthierarchie gebildet werden kann. Bei dieser Plattform sollen zahlreiche Vertreter von Obst und Gemüse mit einigen Synonymen und Gattungen hinterlegt sein. Diese Produkte sollen verschiedene Zustände haben können, wie beispielsweise „ripe“ oder „cooked“. Außerdem sollen unkonkrete Mengenangaben, wie „a few“, aber auch Zahlen möglich sein, die entweder ausgeschrieben, oder als Ziffern angegeben werden können.

Intents

Dieser Online-Service soll Benutzern die Möglichkeit geben, sich über die darin vorhandenen Produkte, ihre Verfügbarkeit und ihre Verkäufer zu informieren, und die Produkte auch kaufen zu können. Aus diesem Grund sind folgende Intents definiert:

- `show_product` wird bei Befehlen erkannt, in denen die Benutzer ein bestimmtes Produkt angezeigt bekommen wollen.
- `show_availability` ist mit Benutzereingaben verbunden, in denen abgefragt wird, ob ein bestimmtes Produkt verfügbar ist.
- `order_product` wird bei Bestellungen erkannt.
- `show_seller` soll aus Anfragen über den Lieferanten extrahiert werden.
- `unknown_intent` ist der Fallback-Intent, der dann ausgegeben werden soll, wenn keiner der anderen zuweisbar ist. Dies ist beispielsweise dann der Fall, wenn das System mit falschen Benutzereingaben konfrontiert ist. Dazu gehören z. B. fehlerhafte Rechtschreibung oder Syntax, aber auch Befehle, die dem NLPS fremd sind, wie etwa Anfragen über das Wetter.

Entities

Die Entities der Obst- und Gemüseplattform gliedern sich wie folgt:

- `product_name` ist die Entity-Kategorie die die Namen der Produkte und deren Synonyme umfasst. Auch die Namen der Produktkategorien, also „fruits“ und „vegetables“, sollen in diesem Szenario enthalten sein. Das bedeutet also, dass diese Liste nur Nomen beinhalten wird, was die Auswertung evtl. erleichtern könnte.

Tabelle 3.1: Auszug der zum Training verwendeten Befehle.

Intent	*Trainingssätze*
`show_product`	Show me ripe apples. I want to see fresh pears. I want to see some oranges.
`order_product`	I want to have ten apples. I want to buy seven fresh pears. I want to order some oranges.
`show_seller`	Who sells apples? Who has fresh pears? Where can I get oranges?
`show_availability`	When can I get apples? When are pears ripe? Do they have fresh oranges today?

- `product_condition` enthält eine Liste an Adjektiven und Verben, die die Produktbeschreibung bzw. die Anfrage verfeinern. Beispielsweise „available“ oder „fried“.
- `product_quantity` ist eine Entity-Kategorie die keinen festen Typ hat. Es sind zwar einige konkrete Wörter in diese Liste eingetragen, wie etwa „some“, jedoch sollen auch Zahlen, die die Produkte quantifizieren, als `product_quantity` erkannt werden können.

Bei der Definition der Entities lässt sich bereits erkennen, wie wichtig es ist, den verwendeten NLPS genau konfigurieren zu können. Wie aus diesen Entities hervorgeht, enthält `product_name` nur Nomen und `product_condition` nur Adjektive und Verben. Wären diese Einstellungen bei den vorgestellten NLPS möglich, könnte die Genauigkeit womöglich verbessert werden. Daher lässt sich bereits jetzt ein Vorteil eines selbstimplementierten Systems erahnen, da dieses beliebig konfigurierbar ist und so gezielter Entity-Extraction betrieben werden kann.

3.1.3 Trainings- und Testkorpus

Um die NLPS entsprechend zu trainieren, werden zu jedem Intent und den verwendeten Entity-Typen Trainingssätze, nach Möglichkeit in großer Menge, benötigt. Je Intent werden gleichviele Befehle angelegt, die zum Training verwendet werden. Ein Auszug davon ist in Tabelle 3.1 aufgelistet. Die vollständige Liste der verwendeten Trainingsbefehle und Entities ist in Anhang C enthalten. Neben den Befehlen für das Training, müssen auch dem NLPS unbekannte Befehle erzeugt werden, um diese mit Neudaten testen zu können. Basierend auf den Anforderungen, beschrieben auf Seite 34f, eignet sich dabei folgende Gliederung dieser Befehle:

- **Trainingsdaten:** Abhängig vom gewählten ML-Algorithmus und den Trainingsdaten kann es sein, dass der NLPS beim Interpretieren dieser Befehle fehlerhafte Ergebnisse liefert, obwohl die eigentlich korrekten vorliegen. Um dies zu evaluieren, werden alle Trainingssätze getestet.

- **Einfache Befehle:** Von den Trainingsdaten geringfügig abgewandelte Sätze sollten von den NLPS am einfachsten erkannt werden. Hier wird die beste Leistung erwartet.
- **Komplexe Befehle:** Sätze, die aus mehreren Gliedsätzen und Entities bestehen, Verneinungen und andere fordernde Elemente enthalten.
- **Eingabequalität:** Sätze mit unbrauchbaren Zeichen, wie mehrere Leer- oder Sonderzeichen, sollten von NLPS gut verarbeitet werden.
- **Unbekannte Intents:** Befehle, die nicht in der Domäne der Obst- und Gemüseplattform liegen.
- **Synonyme und Wortvariationen:** Mit diesen Testdaten soll geprüft werden, ob die Synonyme der Entities korrekt verwendet werden. Auch durch Plural, Fälle oder Zeiten veränderte Wörter sollen richtig erkannt werden.
- **Zahlen:** Testsätze mit ausgeschriebenen oder durch Ziffern ausgedrückte Zahlen sollen korrekt interpretiert werden. Auch unkonkrete Mengenangaben, wie etwa „some“, werden geprüft.

3.2 Leistungsbeurteilung

Prämisse für das Training der zu testenden NLPS ist, dass die Lexik des angebundenen Systems vollständig verwendet wird, und demnach dem NLPS alle relevanten Wörter bekannt sind. Dies ist die Basis dafür, dass ein praxisnahes Bewerten dieser Services möglich ist, denn die extrahierten Begriffe müssen sich mit jenen decken, die das angebundene System auch verarbeiten kann. Zur eigentlichen Verifizierung, ob diese Parameter korrekt sind, müssen den Testbefehlen in der Testapplikation auch die entsprechenden Lösungen hinterlegt sein, die dann abgeglichen werden können. Die Bewertung erfolgt in drei Kategorien, die sowohl individuell, aber auch zusammengeführt bewertet werden:

- **Intent:** Ganz profan ist die Bewertung des Intents. Dieser muss gleich dem in der Lösung hinterlegten Intent sein. Er kann also entweder richtig oder falsch sein.
- **Entity-Name:** Ohne den extrahierten Wert zunächst zu prüfen, soll beurteilt werden, ob der NLPS die Entity-Klasse richtig erkannt hat, selbst wenn der Wert womöglich nicht korrekt ist.

  Sind in dem Befehl mehrere Entities vorhanden, so wird der prozentuelle Anteil ermittelt, wie viele davon korrekt interpretiert wurden.
- **Entity-Wert:** Bei der Verifizierung der Entity-Werte müssen einige Entscheidungen getroffen werden. Da in den Trainingsdaten alle Wörter der Entities in deren Stammform eingetragen sind, wäre es wünschenswert, dass auch die NLPS diese Stammform bilden, um dem angeschlossenen System den Abgleich zu vereinfachen. Wie sich zeigen wird, ist dies jedoch bei keinem der vorgestellten NLPS der Fall, weshalb in dieser Arbeit die extrahierten Wörter als korrekt gewertet werden, auch wenn sie nicht in der Stammform übermittelt werden.

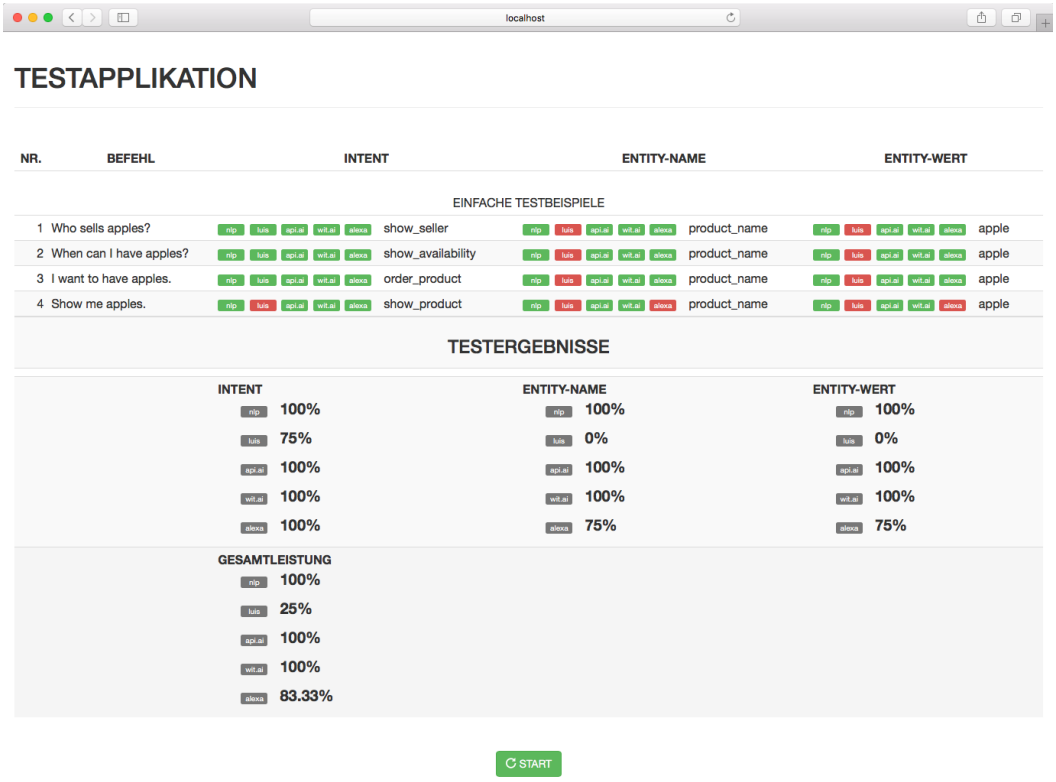


Abbildung 3.1: Screenshot der Testapplikation.

3.3 Testapplikation

Um den Prozess, die Genauigkeit der NLPS zu ermitteln, zu beschleunigen, wird eine Testapplikation verwendet. Diese ist mit den Testbefehlen samt Lösung bestückt, die von den NLPS übersetzt werden müssen. Dafür werden asynchron HTTP-Anfragen an die Schnittstellen der Services geschickt. Nach Eintreffen jeder Antwort wird diese mit der Lösung verifiziert, das Ergebnis im Interface, das in Abbildung 3.1 dargestellt ist, eingetragen und die Genauigkeitswerte für den Service in der Ergebnistabelle aktualisiert. Eine Installationsanleitung dieser Testapplikation ist in Anhang B beschrieben.

3.3.1 Technologie

Server

Da der Großteil der Logik in der Weboberfläche, also im JavaScript, passiert, ist das Anforderungsprofil an den Webserver überschaubar. Es müssen nur die über HTTP angefragten Daten ausgehändigt werden. Daher wird serverseitig eine Node-Instanz verwendet, auf der das Paket ExpressJS installiert ist, welches für das Routing zuständig ist und auf Port 3000 am `localhost` läuft. Die Weboberfläche kann, nach dem Starten mit `npm start` im Verzeichnis der Applikation, unter der URL http://localhost:3000 aufgerufen werden.

Weboberfläche

Zur Vereinfachung der Programmierung der Weboberfläche sind folgende Libraries im Einsatz:

- **jQuery:** In der Version 3.2.0 ist jQuery für die Unterstützung von *Twitter Bootstrap*, für die Vereinfachung der AJAX-Aufrufe sowie der DOM-Manipulation im Einsatz.
- ***Twitter Bootstrap:*** Das Styling sowie das JS, speziell die Tooltip-Mechanik, wird von Bootstrap verwendet.

3.3.2 Format der Testbefehle

Aus Gründen der Übersichtlichkeit und Wartbarkeit sind die Testbefehle in eine eigene Datei als JSON-Format ausgelagert. Wie das nachfolgende Beispiel zeigt, sind die Tests dabei in die Kategorien organisiert, enthalten den Befehl und den dazugehörigen Intent und die Entities.

```
[
  "title": "Trainingsdaten",
  "tests": [
    {
      "command": "Show me ripe apples.",
      "intent": "show_product",
      "entities": [
        {
          "name": "product_condition",
          "value": "ripe"
        },
        {
          "name": "product_name",
          "value": "apple"
        }
      ]
    }
  ]
]
```

3.3.3 Anbindung der NLPS

Um eine möglichst (pseudo-) generische Abwicklung zu ermöglichen, gibt es für jeden zu testenden NLPS ein JavsScript-Objekt. Dieses setzt sich zumindest aus der Eigenschaft `agent`, die zur Identifizierbarkeit den Namen des NLPS enthält, und der Methode `performTest`, die das `testEntry`-Objekt als Parameter erwartet, zusammen. Das folgende Codebeispiel zeigt den asynchronen Ansatz anhand der Rückgabe eines `Promise`-Objektes, das die Callback-Funktion erst ausführt, wenn das Ergebnis des NLPS vorliegt. So können alle Anfragen parallel verarbeitet werden.

```
nlp.performTest = function(testEntry) {
  return new Promise(function (resolve, reject) {
    $.ajax({
      "method": "POST",
      "url": "http://localhost:4000",
```

```
        "data": JSON.stringify({"command": testEntry.command})
      }).done(function(data) {
        var result = {
          "agent": nlp.agent,
          "request": testEntry,
          "response": data
        }
        resolve(result);
      });
    });
  }
```

Es können jedoch nicht alle NLPS mit einem einfachen HTTP-Request angefragt werden und erfordern eine spezielle Anbindung:

- **LUIS:** In der kostenfreien Version von LUIS sind die Aufrufe bei 1.000 Einheiten pro Monat gedeckelt und es kann nur eine Anfrage pro Sekunde verarbeitet werden. Daher werden in der Weboberfläche mit `performTest` die Tests in einem Array gesammelt und im Sekundentakt asynchron abgerufen. Dies funktioniert dafür über eine einfache URL, in der die APP-ID, ein Zugangsschlüssel und der URL-kodierte Befehl als GET-Parameter angehängt werden.
- **API.AI:** Etwas komfortabler und sicherer funktioniert die Abfrage bei API.AI. Ein zusätzlich geladener JavaScript-Code von API.AI liefert bei Übergabe des `accessToken` ein API.AI-Objekt, das mit der Methode `textRequest` und dem Befehls-String als Parameter ein Promise Objekt zurückgibt, das die `then`-Callback-Funktion bei Fertigstellung mit dem resultierenden Ergebnis aufruft.
- **WIT:** Mit einem POST-Request, der in der Payload ein JSON-Objekt erfordert, wird eine Abfrage bei WIT ausgeführt. Dieses JSON-Objekt enthält den `access_token` und unter dem Key `q` den Befehl, der verarbeitet werden soll.
- **ALEXA:** Weil ALEXA keine externe Anbindung ermöglicht, gibt es in der Weboberfläche einen Workaround. Die Testbefehle wurden dafür im Vorfeld in der Eingabemaske von ALEXA eingegeben und die Antwort manuell in ein JSON-Objekt kopiert, das offline am Node-Server gespeichert wird. Bei der automatisierten Abfrage wird dann diese Datei agerufen und mittels des Testbefehls das Ergebnis zur Auswertung in die Weboberfläche geladen.

3.3.4 Initialisierung

Die HTML-Datei, in der die Weboberfläche initialisiert wird, enthält zu Beginn noch keine Testdaten. Es werden erst die verwendeten NLPS-Objekte registriert, um dem System mitzuteilen, welche Services zu testen sind. Anschließend werden die Testdaten vom Server nachgeladen, über JavaScript in HTML umgewandelt und in die Liste der Testbefehle eingefügt. Gleichzeitig wird JavaScript-seitig ein Auswertungsobjekt erzeugt, das zu jedem Service die Trefferquoten aufzeichnet und das Ergebnis in die Ergebnisliste einträgt.

3.3.5 Auswertung der Leistung

Kommt das verarbeitete Ergebnis vom NLPS zurück, wird es mittels JavaScript verifiziert. Dazu wird im ersten Schritt der Intent verglichen. Stimmt dieser überein, wird der

Intent des Einzeltests mit 100% richtig bewertet. Die Entities der Antwort werden alle einzeln geprüft. Zunächst wird dabei verifiziert, ob der Entity-Name auch in der Lösung existiert. Zusätzlich wird sichergestellt, dass der dazugehörige Entity-Wert ausgefüllt ist. Wäre er leer, würde der Entity-Name als falsch gewertet werden. Der Entity-Wert des Entity-Namens wird mit dem korrespondierenden Lösungswert verglichen. Dabei wird zuerst geprüft, ob es sich bei beiden um eine Zahl handelt. Setzt sich der vom NLPS gelieferte Wert aus Zahlen und anderen Zeichen zusammen, wird das Ergebnis als falsch gewertet. Sind Lösung und der Entity-Wert vom Typ Zahl und gleich, dann wird es als korrekt angesehen. Ist der Entity-Wert eine Zahl, die ausgeschrieben vorliegt, wird diese, sofern sie sich mit dem Lösungswert deckt, kulanterweise als gültig gewertet. Ist der Wert aber keine Zahl, müssen sich die Buchstaben des Wortes zu 70% decken um als richtig gewertet zu werden. Dies ist jedoch ein riskanter Workaround, der aber sehr gut funktioniert.

Zur manuellen Überprüfung werden alle Ergebnisse in den Tooltip geschrieben, der bei Maus-Hover angezeigt wird.

Kapitel 4

Implementierung

Die vorgestellten NLPS sollen anhand der beschriebenen Testdaten untereinander, aber auch mit einem selbstimplementierten NLPS verglichen werden. In diesem Kapitel wird die Implementierung dieses Prototyps dargelegt und gezeigt, wie die Disziplinen von ML, Textaufbereitung und Feature-Extraktion zu einem funktionierenden, domänenspezifischen NLPS zusammengeführt werden. Zunächst wird dazu auf die Übersicht der Aufgaben eingegangen, die anschließend einzeln näher beschrieben werden. Abschließend wird gezeigt, wie das fertige System gestartet und verwendet wird.

4.1 Prototypstruktur

Der NLPS-Prototyp setzt sich aus drei wesentlichen Aufgabengebieten zusammen:

- **Aufbereitung der Trainingsdaten:** Das Zusammensetzen und Bereitstellen der Trainingsbefehle ist in der Klasse `CommandCompiler` realisiert.
- **Textverarbeitung:** Die Klasse `LanguageProcessingService` erledigt den Kern der Arbeit. Sie benützt die Klasse `CommandCompiler`, enthält die Instanz des ML-Algorithmus, verwendet ausgewählte NLP-Module und ist für die Textverarbeitung und die Intent- und Entity-Erkennung zuständig.
- **Webserver:** Um den NLPS im Webbrowser verwenden zu können, gibt es die Klasse `WebService`, die entsprechend die Verbindung zwischen Client und Prototyp herstellt.

Für ausgereifte Softwarelösungen müssten diese Aufgaben und Klassen zwar in viel feinere Strukturen überführt werden, jedoch ist die Aufgliederung im Zuge dieses prototypischen Versuchs ausreichend, da die Leistung davon nicht beeinträchtigt wird. Dessen etwas vereinfachtes UML-Diagramm der Implementierung ist in Abbildung 4.1 dargestellt.

4.2 Aufbereitung der Trainingsdaten

Wie in den weiteren Abschnitten hervorgeht, wird zur Erkennung des Intents ML verwendet. Um den ML-Algorithmus adäquat trainieren zu können, sollen die Trainingsbefehle mit Bedacht aufbereitet werden. Aus den Trainingsdaten in Anhang C geht hervor,

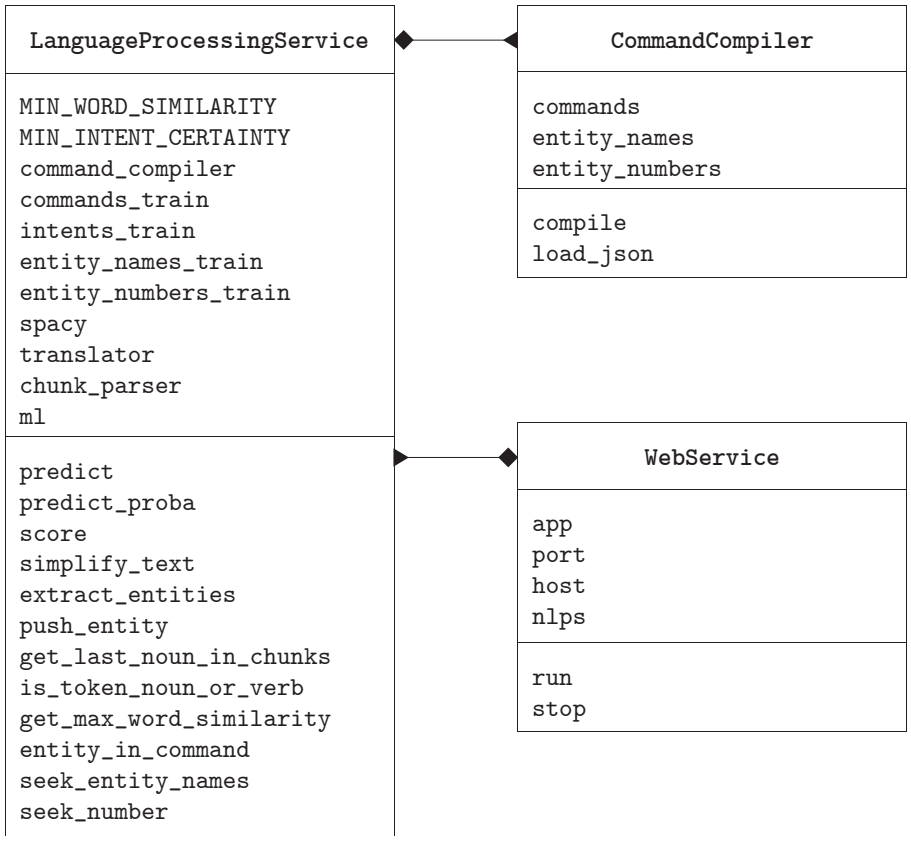


Abbildung 4.1: UML-Diagramm des NLPS-Prototyps.

dass es für jeden Intent sieben Trainingsbefehle, sechs Einträge für `product_condition`, 31 für `product_name` und vier Einträge für `product_quantity` gibt. Um den verwendeten ML-Algorithmus mit allen bekannten Wortkombinationen zu trainieren, sind in den Rohbefehlen Platzhalter eingebaut:

```
[
  {
    "intent": "order_product",
    "command": "I want to buy %product_quantity% %product_condition%
                %product_name%."
  }
]
```

In der Klasse `CommandCompiler` werden diese Platzhalter durch die Einträge der Entities ersetzt und so alle Kombinationen erzeugt. Dadurch ergibt sich eine große Menge sehr ähnlicher Trainingsbefehle. Die genaue Anzahl ist in Tabelle 4.1 angegeben.

Tabelle 4.1: Anzahl an Trainingsbefehlen die durch die Kombination der Entities mit den Rohbefehlen entstehen.

Intent	*Trainingsbefehle*
`show_product`	1.252
`order_product`	2.728
`show_availability`	657
`show_seller`	1.550

4.3 Textverarbeitung

4.3.1 Wahl des ML-Algorithmus

Mit den zusammengestellten Trainingsdaten kann nun mit der Wahl der eingesetzten Technologien begonnen werden. Dazu werden alle vorgestellten supervised ML-Algorithmen mit verschiedenen Parameterkombinationen getestet und die Ergebnisse evaluiert.

Evaluierungsaufbau

Das Evaluierungsszenario soll den Einsatz im fertigen NLPS simulieren, weshalb mit den realen Trainingsdaten gearbeitet wird, welche mit dem TF-IDF-Verfahren aufbereitet werden, wie Zeile 1 in dem folgenden Code-Auszug zeigt:

```
1 pipe = make_pipeline(TfidfVectorizer(min_df=1), KNeighborsClassifier())
2 param_grid = {
3   "kneighborsclassifier__n_neighbors": [1, 3, 9, 21, 51, 101],
4   "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]
5 }
6 grid = GridSearchCV(pipe, param_grid, cv=5)
7 grid.fit(commands_train, intents_train)
8 print("pipeKNN score: {:.3f}, parameters: {}".format(grid.best_score_,
9       grid.best_params_))
```

Die Pipeline der Aufbereitung und des ML-Prozesses lässt sich bequem mit der Funktion `make_pipeline` realisieren. Dieser Funktion werden die Objekte für TF-IDF und ML in der Reihenfolge, in der sie aufgerufen werden sollen, übergeben. Dies wird durch die durchgängige Schnittstellenkonvention im Paket `sklearn` ermöglicht. Ist die Pipeline erstellt, kann auch diese mittels GS auf vorgegebene Parameterkombinationen getestet werden. Die Auswertung der Leistung der gerade getesteten Einstellungen wird mittels CV ermittelt. Dazu sind fünf Iterationen eingestellt, was in diesem Fall ausreichend ist. Siehe dazu Zeile 2 bis 6. In Zeile 7 wird die eigentliche GS durchgeführt und die Parameter ermittelt, die in Zeile 8 bzw. 9 ausgegeben werden.

Auswertung

In der Tabelle 4.2 sind die Parameter aufgelistet, die für die Ermittlung des besten ML-Algorithmus getestet werden. Die Einstellungen für die `ngram_range` des TF-IDF-

Tabelle 4.2: Alle Kombinationen der Parameter der ML-Algorithmen, die evaluiert werden.

Algorithmus	*Parametertests*
TF-IDF	`ngram_range: [(1, 1), (1, 2), (1, 3)]`
kNN	`n_neighbors: [1, 3, 9, 21, 51, 101]`
LR	`C: [0.001, 0.01, 0.1, 1, 10, 100]`
SVM	`C: [0.001, 0.01, 0.1, 1, 10, 100]`, `gamma: [0.001, 0.01, 0.1, 1, 10, 100]`
RF	`n_estimators: [1, 10, 100]`
NN	`hidden_layer_sizes: [[10], [100], [10, 10], [100, 100]]`

Verfahrens sind dabei für alle Tests gleich. Die Tupels geben an, wie groß die NG sein dürfen. Beispielsweise ergeben sich bei dem Bereich `(1, 3)` alle Kombinationen von NG mit der Länge 1, 2 und 3. Die maximale Leistung der ML-Algorithmen und die dazugehörigen Parameterkonstellationen sind in Tabelle 4.3 aufgelistet. Es ist zu sehen, dass kNN mit der Genauigkeit von 89,7% mit Abstand zurückliegt, was wohl auf die „lückenhaften“ Feature-Vektoren zurückzuführen ist. Wie in Kapitel 2 beschrieben, ist kNN nicht in der Lage, Datensätze adäquat zu verarbeiten, die nicht alle Features des Feature-Raumes enthalten. Interessant ist, dass der Algorithmus den vollen Bereich der NG ausnutzt um so mehr Features zur Auswertung zur Verfügung zu haben. Der Wertebereich der NG wird jedoch absichtlich nicht weiter erhöht, da dadurch das Risiko der Überanpassung steigt. Der nächstbeste Algorithmus ist RF mit 94,9% bei zehn zufälligen DT. Wie auch alle übrigen Algorithmen führen hier NG mit jeweils nur einem Wort zum besten Ergebnis. Da sich die RF jedoch zufällig bilden, kann dieses Ergebnis mit denselben Parametern leicht variieren. Auf dem dritten Platz befindet sich mit 96,5% der NN-Algorithmus. Es gibt jedoch noch viele weitere, weit aufwändigere Trainings- und Optimierungsmöglichkeiten, als das hier verwendete Verfahren, was aber weit den Rahmen dieser Arbeit und des Prototyps übersteigt [18]. Mit einem sehr guten Wert von 97,5% stellen sich die beiden linearen ML-Algorithmen LR und SVM als die für den Prototyp geeignetsten heraus. Beide verwenden dabei einen Wert von 10 für `C`, was bedeutet, dass sie, wie die Abbildungen 2.4 und 2.5 zeigen, moderat auf Artefakte reagieren und so ein gutes Generalisieren versprechen. Der Parameter `gamma` bei dem SVM-Algorithmus wurde mit dem Wert 0,1 ermittelt, wodurch sich nicht so leicht Klasseninseln bilden und deshalb ebenfalls gut generalisiert werden kann.

Da die LR also die gleiche Leistung erzielt, jedoch bedeutend weniger rechenintensiv ist, fällt die Wahl des verwendeten ML-Algorithmus für den Prototyp auf die LR.

4.3.2 Wahl des Lemmatisierers

Für das Lemmatisieren bieten sich in Python das Modul `spacy` und `wordnet` in `nltk` an. Zweites liefert jedoch deutlich unbrauchbarere Ergebnisse, wie der folgende Test anhand

Tabelle 4.3: Auswertung der GS mit den besten Parameterkombinationen.

ML-Algorithmus	*Leistung*	*Parameter*
kNN	89,7%	`ngram_range: (1, 3)` `n_neighbors: 9`
LR	97,5%	`ngram_range: (1, 1)` `C: 10`

SVM	97,5%	`ngram_range: (1, 1)`
		`C: 10`
		`gamma: 0.1`

RF	94,9%	`ngram_range: (1, 1)` `n_estimators: 10`
NN	96,5%	`ngram_range: (1, 1)` `hidden_layer_sizes: [100]`

dreier einfacher Testsätze zeigt:

```
For sentence >I want to see ripe apples<:
  WN: >I want to see ripe apple<
  SP: >-PRON- want to see ripe apple<

For sentence >We have seen freshly riped berries<:
  WN: >We have seen freshly riped berry<
  SP: >-PRON- have see freshly rip berry<

For sentence >Show me who has sold freshly harvested pineapples<:
  WN: >Show me who ha sold freshly harvested pineapple<
  SP: >show -PRON- who have sell freshly harvest pineapple<
```

Es ist zu erkennen, dass beim ersten Satz, mit Ausnahme des Pronomens, beide gleich lemmatisieren und dabei fehlerfrei arbeiten. Im zweiten Testbefehl wird bereits eine deutliche Diskrepanz der Qualität und die unterschiedliche Verarbeitung des Pronomens erkennbar. Während `spacy` nicht zwischen den Personen unterscheidet, was der NLPS später in diesem Szenario auch nicht machen soll, differenziert `nltk` diese sehr wohl. Das Wort „freshly“, also ein Adverb, wird von beiden nicht verändert, was in Ordnung ist. Das Wort „riped“ wird von `nltk` nicht verändert, was für die weitere Textverarbeitung sehr ungünstig ist, da es sich vom Stammwort „ripe“ unterscheidet. Von `spacy` wird zumindest der Wortstamm gebildet, jedoch auf einer falschen Grundlage. Es wird das englische Wort „reif“ mit „reißen“ verwechselt, was auch zu einem unbrauchbaren Ergebnis führt. Den Plural lemmatisieren beide einwandfrei. Beim dritten Testsatz machen sich eklatante Unterschiede bemerkbar. Während `spacy` ein einwandfreies Ergebnis liefert, scheitert `nltk` an der Korrektur der Zeit und macht aus dem Wort „has“ ein unbekanntes Wort „ha“.

Aus diesen Tests zeigt sich, dass im Zuge dieses Szenarios `spacy` für die Aufgabe des Lemmatisierens die bessere Wahl ist.

4.3.3 Initialisierung

Allgemeine Prozessbeschreibung

Zum Starten des NLPS wird die Datei `main.py` ausgeführt. Dadurch wird eine Instanz des NLPS initialisiert, die wiederum ein Objekt des `CommandCompiler` erzeugt. Das `LanguageProcessingService`-Objekt wird dann dem `WebService` bei der Initialisierung übergeben, der abschließend die HTTP-Requests entgegennimmt.

Textaufbereitung

Die zur Textverarbeitung entwickelte Klasse `LanguageProcessingService` wird mit folgendem Code initialisiert:

```
1 def __init__(self):
2     self.command_compiler = CommandCompiler()
3     self.commands_train, self.intents_train, self.entity_names_train, self.
      entity_numbers_train = self.command_compiler.compile()
4
5     self.spacy = spacy.load("en")
6
7     chunk = r"""CHUNK: {<JJ.*>*<NN.*>+}"""
8     self.chunk_parser = nltk.RegexpParser(chunk)
9
10    self.translator = "".maketrans("", "", string.punctuation)
11    self.simplify_all_commands()
12
13    pipe = make_pipeline(TfidfVectorizer(min_df=1), LogisticRegression())
14    param_grid = {
15        "logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
16        "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]
17    }
18    self.ml = GridSearchCV(pipe, param_grid, cv=5)
19    self.ml.fit(self.commands_train, self.intents_train)
```

Der erste Schritt bei der Initialisierung ist das Aufbereiten der Trainingsbefehle. Dazu wird der von der Klasse `CommandCompiler` bereitgestellte Text in `commands_train`, `intents_train`, `entity_names_train` und `entity_numbers_train` aufgeteilt und zwischengespeichert, wie in den Programmzeilen 2 und 3 zu sehen ist. Für die Verarbeitung der Textdaten werden ein paar Module initialisiert:

- `spacy` wird für die Lemmatisierung und die Ermittlung von Wortähnlichkeiten benötigt. Dafür wird die englische Sprache in Zeile 5 geladen.
- `nltk` muss nicht instanziert werden und wird deshalb mit einem `import`-Statement einfach geladen.
- `chunk_parser` wird in Zeile 7 und 8 aus dem Modul `nltk` instanziert und ist für das Chunking zuständig, von dem bei der Entity-Extraction Gebrauch gemacht wird.
- `translator` wird aus der Python-nativen Funktion `maketrans` in Zeile 10 erzeugt und dient der Vereinfachung der zu verarbeitenden Befehle. In diesem Fall werden davon alle Satzzeichen entfernt und durch Leerzeichen ersetzt.

Nachdem diese Systeme aufgesetzt sind, werden in Zeile 11 alle Trainingsbefehle vereinfacht um die Dimensionalität und Vielfalt der Wortvariationen in dem Befehl weitest-

gehend zu reduzieren. Dies ist notwendig, damit die Übereinstimmung zweier Sätze, die z. B. in unterschiedlichen Zeiten geschrieben sind, möglichst groß ist. Die genauen Verarbeitungsschritte dieser Vereinfachung werden im Zuge der Intenterkennung beschrieben.

Sind alle Trainingsdaten in ihre vereinfachte Form überführt, kann damit das ML-System aufgesetzt und trainiert werden. Dies erfolgt auf bekannte Weise von Zeile 13 bis 19. Zu bemerken ist, dass hier, obwohl die idealen Parameterwerte bereits im Zuge der Wahl des ML-Algorithmus ermittelt wurden, diese erneut durch die GS festgelegt werden. Der Grund dafür ist, dass Änderungen der Trainingsdaten, welche in einem realen Szenario häufig vorkommen können, auch die optimalen Parameter verändern können. Um also stets die beste Leistung erzielen zu können, wird dieser Prozess bei jeder Initialisierung durchlaufen.

4.3.4 Erkennung des Intents

Die Aufgabe der Intent-Erkennung, den übergebenen Befehl einem von vier Intents, oder dem `unknown_intent`, zuzuweisen, ist ein reines Kategorisierungsproblem, das mit überschaubarem Aufwand und ML gelöst werden kann. Dennoch lohnt sich ein Blick in [30], in der ein interessanter Ansatz, generisch Intents bzw. Funktionen zu verschachteln, verfolgt wird. Dies führt jedoch in dem hier verwendeten Szenario zu weit.

Wird der `LanguageProcessingService`-Instanz ein Befehl zur Interpretation übergeben, ist der erste Schritt, den Intent zu bestimmen. Dazu wird dieser, genau wie die Trainingssätze, zunächst durch die Methode `simplify_text` vereinfacht:

```
1 def simplify_text(self, command=""):
2     command_result = command.lower()
3     command_result = " ".join(command_result.split())
4     command_result = command_result.translate(self.translator)
5     doc_spacy = self.spacy(command_result)
6     command_result = " ".join([token.lemma_ for token in doc_spacy])
7     return command_result
```

Als erstes wird in Zeile 2 der Befehl in Kleinbuchstaben überführt, um fehlerhafte oder unterlassene Großschreibung auszuschließen. Anschließend werden in Zeile 3 alle überflüssigen Leerzeichen und in Zeile 4 alle Satzzeichen entfernt. In Zeile 5 und 6 erfolgt schlussendlich das Lemmatisieren mit `spacy`. So entsteht sowohl aus „Who wants to sell apples to me?“ als auch aus „Who wanted to sell apples to us?“ das gleiche Ergebnis: „who want to sell ripe apple to -PRON-“. Eine Zuweisung mit ML ist damit erheblich leichter. Die Methode `predict` der Klasse `LanguageProcessingService` dient als Schnittstelle nach außen, veranlasst die Interpretation und gibt Intent und Entities zurück:

```
1 def predict(self, command):
2     simplified_command = self.simplify_text(command)
3     if self.predict_proba(command).max() < self.MIN_INTENT_CERTAINTY:
4         intent = np.array(["unknown_intent"])
5     else:
6         intent = self.ml.predict([simplified_command])
7     entities = self.extract_entities(simplified_command, command)
8     return intent, entities
```

Nach der Vereinfachung des Befehls in Zeile 2, wird als nächstes mit `predict_`

`proba` ermittelt, mit welcher Wahrscheinlichkeit der passendste Intent ausgewählt wird.[1] Ist die „Zuversicht“ unterhalb des Mindestmaßes `MIN_INTENT_CERTAINTY`, so wird der `unknown_intent` zurückgeliefert. Ist die Wahrscheinlichkeit höher, wird von dem ML-Algorithmus bestimmte Intent ausgewählt. Dieser Mindestwert ist beliebig gewählt und in dem Prototyp auf 70% angesetzt, was eine durchschnittliche Gegenwahrscheinlichkeit von 10% je Intent, bzw. eine maximale Gegenwahrscheinlichkeit von 30% für einen Intent bedeutet.

Vom Intent unabhängig werden die Entities extrahiert. Beide Ergebnisse werden abschließend als Rückgabewert dem Aufrufer übermittelt.

4.3.5 Entity Extraction

Allgemeine Prozessbeschreibung

Es gibt drei Entity-Klassen, nach denen gesucht wird: der `product_name`, die `product_condition` und die `product_quantity`. Die ersten beiden werden im zu prüfenden Befehl, der in originaler und vereinfachter Form vorliegt, gesucht. Dabei wird nicht nur die Zeichnfolge verglichen, sondern auch auf grammatikalische Regeln geachtet. Jedes gefundene Wort wird zum Array der Entities hinzugefügt. Die `product_quantity` ist ein Spezialfall und kann aus einer unkonkreten Mengenangabe „some“, aus einer ausgeschriebenen Zahl „twenty four“ oder aus einer Ziffernfolge „24“ bestehen. Daher wird zunächst nach den vordefinierten, trainierten Mengenangaben gesucht. Bleibt diese Suche erfolglos, wird nach den Ziffern, und bei erneutem Fehlschlagen nach ausgeschriebenen Zahlen gesucht, und bei einem Treffer den Entities angehängt.

Erkennung bekannter Wörter

Für Demonstrationszwecke zur Erkennung bekannter Wörter wird folgender Satz verwendet: „I want to have twenty four ripe, orange carrots.“ Darin lässt sich u. a. der Fallstrick ausmachen, dass „orange“ hier ein Adjektiv und damit nicht die Frucht gemeint ist. Um dies entsprechend zu berücksichtigen, erfolgt eine Grammatikerkennung, wie im Code-Auszug der Methode `seek_entity_names` zu sehen ist:

```
1  tokens = nltk.word_tokenize(command)
2  tagged = nltk.pos_tag(tokens)
3  chunks = self.chunk_parser.parse(tagged)
4  relevant_nouns = self.get_last_noun_in_chunks(chunks)
5
6  entities = []
7  for product_name in self.entity_names_train["product_name"]:
8      simplified_product_name = self.simplify_text(product_name)
9      if self.entity_in_command(simplified_product_name, simplified_command)
10         and (simplified_product_name in relevant_nouns
11             or " " in simplified_product_name)
12         or self.get_max_word_similarity(simplified_product_name, simplified_command,
13             "nv", relevant_nouns) >= self.MIN_WORD_SIMILARITY:
14         entities = self.push_entity("product_name", product_name, entities)
```

[1] Bei `predict_proba` wird absichtlich der unvereinfachte Befehl übergeben, da dies die Konvention innerhalb `sklearn` erfordert. Der Text wird innerhalb der Methode jedoch wieder vereinfacht.

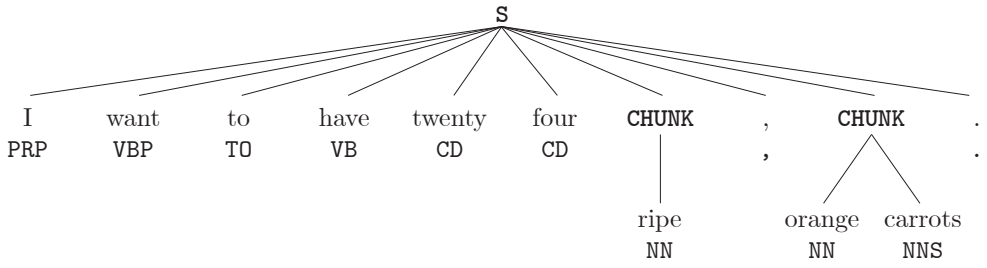


Abbildung 4.2: Das Ergebnis des Satzes „I want to have twenty four ripe, orange carrots.“ durch Chunking.

Zur Erkennung der Grammatik muss der Rohbefehl, wie in Zeile 1 bis 3 gezeigt, zunächst durch Tagging, danach durch Tokenization und letztlich durch Chunking aufbereitet werden. Der `chunk_parser` wurde mit `r"CHUNK: <JJ.*><NN.*>+"` initialisiert und sucht also nach Mustern, wo kein oder mehrere Adjektive vor einem oder mehreren Nomen stehen um gezielt nach `product_name` und `product_condition` zu suchen. Das Ergebnis des Chunkings an dem Beispielsatz ist in Abbildung 4.2 zu sehen. Daraus lässt sich erkennen, dass das Tagging hier nicht einwandfrei funktioniert hat, da „ripe“ und „orange“ in dem Fall keine Nomen sind. Produktnamen können jedoch nicht vor einem Nomen stehen bzw. ohne Satzzeichen getrennt aneinander gereiht sein. Daher wird in Zeile 4 die rekursive Methode `get_last_noun_in_chunks` aufgerufen, die die Baumstruktur nach den letzten Nomen in den Chunks durchsucht und diese zurückliefert. Davon ist das Ergebnis „ripe“ und „carrots“. Das Wort „orange“ wird korrekt entfernt.

Nun, da die Kandidaten für `product_name` feststehen, wird in Zeile 6 ein leeres Entity-Array erzeugt und in Zeile 7 über alle trainierten Produktnamen iteriert und davon in Zeile 8 für Textvergleiche die vereinfachte Form gebildet. Die Verzweigung ab Zeile 9 prüft die folgenden Kriterien, die zur erfolgreichen Erkennung des Namens erfüllt werden müssen:

- Ist der vereinfachte Produktname in dem vereinfachten Befehl vorhanden, mit der Bedingung, dass:
  - der Name auch in dem Array der relevanten Nomen vorkommt,
  - oder der Name sich aus mehreren Wörtern zusammensetzt?[2]
- Ist der Name nicht direkt enthalten, wird gesucht, ob der Befehl ein verwandtes Wort, wie „filbert“ statt „hazelnut“, enthält. Dabei muss die Mindestähnlichkeit von 95%, die beliebig festgelegt ist, gegeben sein. Dabei werden von dem zu prüfenden Wort alle `wordnet`-bekannten Synonyme, sogenannte *synsets*, mittels der vorgestellten Ähnlichkeitsprüfung verglichen. Da für `product_condition` relevante Adjektive nicht mit `wordnet` verglichen werden können, wird dafür `spacy` verwendet.

Die Treffer werden mit `push_entity` an das Ergebnis-Array im Key-Value-Format angehängt. Im Falle mehrerer Treffer werden diese mit Beistrich getrennt eingetragen.

[2]Diese Prüfung ist notwendig, da das Chunking und die Nomensuche Namen mit mehreren Wörtern eliminieren.

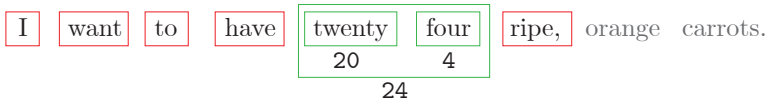


Abbildung 4.3: Bei der Erkennung von Zahlen wird bei jedem Wort und den Folgewörtern geprüft, ob es sich um eine Zahl handelt.

Die trainierten Wörter für `product_condition` werden auf sehr ähnliche Weise geprüft. Allerdings entfällt dabei der Abgleich mit den relevanten Nomen.

Erkennung von Zahlen

Wie einleitend bereits beschrieben, werden zunächst mittels einfachem Pattern-Matching die bekannten, unkonkreten Mengenangaben gesucht. Erkannte Ausdrücke werden ebenfalls mit `push_entity` dem Ergebnis-Array angehängt. Andernfalls wird nach Zahlen in textueller und Ziffernform gesucht. Dabei wird, wie in Abbildung 4.3 dargestellt, jedes Wort auf eine Zahl geprüft, und bei einem Erfolg untersucht, ob die folgenden Wörter ebenfalls zur Zahl gehören, woraufhin diese ggf. zusammengerechnet werden. Der Code zur Umrechnung der einzelnen Wörter ist in der Methode `seek_number` implementiert und prüft zunächst, ob die Zahl in Ziffern geschrieben ist. Falls nicht, wird das Wort anhand einer Hashtabelle, in der die Zahlenwörter den Wertigkeiten gegenübergestellt sind, verglichen und der Zahlenwert ermittelt. Das Ergebnis wird entsprechend zurückgegeben.

Wurden bereits Zahlenwörter, wie „twenty“ und „four“ im Beispiel, gefunden und zusammengerechnet, bricht die Suche ab, sobald ein Wort folgt, das keinen Zahlenwert repräsentiert. Es wird also automatisch die erste gefundene Zahl erkannt und nicht weiter gesucht.

4.4 Webserver

Die `LanguageProcessingService`-Instanz wird gemeinsam mit dem `host` und dem `port` dem `WebService` zur Initialisierung übergeben. Dieser hält neben dem NLPS- auch das `Sanic`-Objekt, welches die Aufgaben des Webservers übernimmt und die HTTP-Requests verarbeitet. Mit der Methode `run()` wird das `WebService`-Objekt und damit der Webserver gestartet, wodurch der gesamte NLPS einsatzbereit ist, und standardmäßig POST-Requests an die URL http://localhost:4000 verarbeitet. Die Anfragen werden entsprechend dem NLPS übergeben und dessen Antwort in ein adäquates JSON-Objekt umgewandelt und dem Aufrufer zurückgeschickt.

4.5 Verwendung

Mit dem Befehl `python main.py` wird der Service in der Konsole gestartet. Danach wartet der Webserver auf die POST-Anfragen des Clients, die folgende JSON-Konvention erfüllen müssen:

```
{
  "command": "I want to have twenty four ripe, orange carrots."
}
```

Nach Verarbeitung des Befehls wird dem Client schlussendlich die übersetzte Antwort zurückgeschickt:

```
{
  "requested_command": "I want to have twenty four ripe, orange carrots.",
  "result": {
    "intent": ["order_product"],
    "entities": [
      {
        "name": "product_name",
        "value":"carrot"
      },
      {
        "name": "product_condition",
        "value": "ripe"
      },
      {
        "name": "product_quantity",
        "value": 24
      }
    ]
  }
}
```

Kapitel 5

Testauswertung

Der fertiggestellte Prototyp sowie die angebundenen NLPS sind trainiert und an die Testapplikation angebunden. In diesem Kapitel werden die Tests durchgeführt und die Leistung, sowie die Qualität und Nutzbarkeit der Ergebnisse beurteilt. Eine ausführliche Liste der Testdaten und der Antworten der NLPS ist im Projektverzeichnis auf der beiliegenden CD-ROM enthalten. Siehe dazu Anhang D. Entsprechend der Kategorien, definiert auf Seite 39f, werden mit allen fünf angebundenen NLPS Tests durchgeführt und nach den auf Seite 40 beschriebenen Regeln beurteilt.

5.1 Trainingsdaten

5.1.1 Intents

Das Prinzip der Generalisierung bedingt den Kompromiss, dass die Leistung bei den Trainingsdaten abnimmt. Deshalb werden alle 28 Trainingsbefehle ausgewertet, um zu verifizieren, wie sehr sich dieser Effekt messen lässt. Dafür wurden zufällige Entities ausgewählt, die in den Platzhaltern für `product_name`, `product_condition` und `product_quantity` eingesetzt sind. Wie der Tabelle 5.1 zu entnehmen ist, weisen sowohl der Prototyp, als auch drei der NLPS, die Trainingsbefehle fehlerfrei den Intents zu. Einzig LUIS schneidet mit einem äußerst unbrauchbaren Ergebnis von nur 64% ab. Dies könnte darin begründet sein, dass *Microsoft* hier zu stark generalisiert und so eine Unteranpassung erzeugt. Es werden über alle Trainingsbefehle hinweg die Intents verwechselt, jedoch kein `unknown_intent` zurückgeliefert. Es lassen sich auch keine Ge-

Tabelle 5.1: Testergebnisse der Trainingsdaten.

NLPS	*Intent*	*Entity-Name*	*Entity-Wert*	*Gesamt*
Prototyp	100%	96%	96%	98%
LUIS	64%	37%	37%	46%
API.AI	100%	100%	100%	100%
WIT	100%	82%	82%	88%
ALEXA	100%	76%	76%	84%

meinsamkeiten feststellen, die auf dieses Fehlverhalten Rückschlüsse ziehen ließen.

5.1.2 Entities

Verglichen zu der Erkennung des Intents weisen, bis auf API.AI, der auch hier perfekt abschneidet, die NLPS bei den Entities spürbare Mängel auf. Erneut liefert LUIS mit nur 37% für die Erkennung der Entity-Namen und der Entity-Werte, ein unbrauchbares Ergebnis. Hier fällt auf, dass LUIS offenbar keine automatische Erkennung des Plurals der trainierten Wörter beherrscht. Alle Wörter die exakt so im Testbefehl enthalten sind, wie sie trainiert wurden, werden korrekt erkannt. Allerdings werden keine Zahlen erkannt, die in ausgeschriebener Form vorliegen. WIT liefert, wie API.AI bei fast allen Entities perfekte Ergebnisse, jedoch ist auch dieser Service offenbar nicht in der Lage, ausgeschriebene Zahlen als Mengenangaben zu erkennen. Ernüchternd ist leider auch, dass die extrahierten Produktnamen nicht in den Singular überführt werden, weshalb das angebundene System dies berücksichtigen oder korrigieren muss. ALEXA liefert seine Ergebnisse in einer noch schlechter aufbereiteten Form. Hier werden nicht einmal die Satzzeichen entfernt, die an das erkannte Wort gesetzt sind. Auch ALEXA ist nicht in der Lage die Zahlen in ausgeschriebener Form zu erkennen. Unkonkrete Mengenangaben werden aber fehlerfrei extrahiert. API.AI erzielt bei der Erkennung der Entities nur deshalb alle Punkte, weil ausgeschriebene, also nicht in Ziffern ausgedrückte Zahlen, als richtig gewertet werden. Von den Rückgabewörtern werden die Satzzeichen entfernt und der Singular gebildet.

Die extrahierten Werte des Prototyps sind für die weitere Verarbeitbarkeit am besten aufbereitet. Die Stammform der Wörter ist gebildet, Satzzeichen entfernt und ausgeschriebene Zahlen in Ziffern umgewandelt. Dennoch werden nur 98% erreicht. Es wurden in den Tests zweimal die Werte für `product_name` falsch extrahiert. Der erste Fehler ist auf das Modul `nltk` zurückzuführen, da in dem Satz „Are apples currently ripe?“ das Wort „apples“ als Verb erkannt wurde, und somit vom Prototyp nicht unter den relevanten Nomen gefunden werden konnte. Siehe dazu Zeile 1 und 2 im nachstehenden Code-Auszug.

```
1 print(nltk.pos_tag(nltk.word_tokenize("Are apples currently ripe?")))
2 # [("Are", "NNP"), ("apples", "VBZ"), ("currently", "RB"), ("ripe", "VBN"), ("?",
      ".")]
3
4 print(nltk.pos_tag(nltk.word_tokenize("Do they have ripe apples today?")))
5 # [("Do", "VBP"), ("they", "PRP"), ("have", "VBP"), ("ripe", "JJ"), ("apples", "NNS
      "), ("today", "NN"), ("?", ".")]
```

Der zweite Fehler tritt im Satz „Do they have ripe apples today?“ auf und hängt mit der Implementierung des Prototyps zusammen. Wie aus Zeile 3 und 4 hervorgeht, ist das Letzte Wort der Nomenkette „today“, weshalb das eigentliche Subjekt „apples“ nicht erkannt werden kann.

5.2 Testdaten

Weitere 31 Tests, jedoch mit Sätzen, die dem NLPS noch nicht bekannt waren, wurden verteilt auf die folgenden Kategorien durchgeführt. Die resultierenden Leistungen sind in Tabelle 5.3 aufgelistet.

5.2.1 Einfache Befehle

Die hier verwendeten Sätze sind sehr kurz und für Menschen leicht verständlich, wie beispielsweise „Who sells apples?“. Die Intents werden dabei von allen NLPS richtig erkannt, außer von LUIS. Beim Satz „Show me apples.“ wurde der Intent `show_seller` eingesetzt. Womöglich deshalb, weil in den Trainingsdaten dieses Intents ebenfalls das Wort „show“ vorkommt. Die Entities wurden vom Prototyp, API.AI und WIT korrekt, in bekannter Qualität, extrahiert. Da in allen Testsätzen der Produktname in der Mehrzahl vorkommt, kann LUIS keinen einzigen erkennen. ALEXA scheitert ebenfalls bei jedem zweiten Satz. Warum z. B. aus dem Satz „I want to have apples.“ der Produktname nicht extrahiert werden kann, ist fragwürdig und ohne Einblicke in die Implementierung leider nicht erklärbar.

5.2.2 Eingabequalität

In dieser Kategorie wurden exakt vier Tests durchgeführt:

- „ Where can I buy ripe apples ?“: Der Intent wurde von allen Systemen korrekt erkannt. Aufgrund der Mehrzahl liefert LUIS nur „ripe“ als `product_condition` zurück, die jedoch API.AI fehlerhaft als `product_quantity` erkannte. Es scheint also, als ob API.AI mit zu vielen Leerzeichen im Befehl an Grenzen stößt.
- „Show me ripeapples.“: Obwohl die Intents richtig erkannt werden, wirken sich fehlende Leerzeichen hingegen eklatant auf die Leistung der Entity-Extraction aus. Einzig LUIS ist in der Lage, die `product_condition` „ripe“ zu erkennen.
- „WHEN IS GOLDEN DELICIOUS RIPE?“ und „when is golden delicious ripe?“: Die Großschreibung wirkt sich bei keinem Service negativ auf die Leistung aus. Beide Sätze werden von allen korrekt interpretiert.

5.2.3 Komplexe Befehle

Unter den Tests im Zuge komplexer Befehle finden sich lange, umständlich formulierte Sätze, wie beispielsweise „Although it is not their season, I would really like to have orange carrots by lunch.“. Bei dieser Länge gibt es deutliche Leistungsrückgänge der Intent-Erkennung, bei der erstmals auch der `unknown_intent` auftritt. Der Prototyp ist bei diesem Satz jedoch der einzige Service, der sowohl Intent, als auch die Entities korrekt erkennt, während alle anderen als Produktname „orange“ extrahiert haben. Obwohl API.AI allgemein bei diesen langen Sätzen eine schlechte Leistung erbringt, ist dieser NLPS als einziger in der Lage, Sätze mit Verneinungen, wie „I don’t want to see ripe apples, but fresh pears.“, korrekt zu erkennen.

5.2.4 Unbekannte Intents

Eine Schwierigkeit bei NLPS ist, zu erkennen, wenn ein Befehl aus einer anderen Domain stammt und nicht vom angebotenen Service verarbeitet werden kann. Einen derartigen Befehl nicht abzufangen, sondern zu verarbeiten und zusammenhanglose Ergebnisse zurückzuliefern, wäre für einen Client irritierend. Daher ist das Erkennen vom `unknown_intent` wohl ebenso wichtig wie das Erkennen der Intents korrekter Befehle.

Allgemein ist WIT hier der beste NLPS und der Prototyp, mit einem Fehler bei der Intent-Erkennung, auf Platz zwei. API.AI und LUIS schneiden bei diesem Test sehr schlecht ab, und bei ALEXA scheint es kein adäquates Verhalten auf unbrauchbare Eingaben zu geben. Ein getesteter Fall ist der unvollständige Befehl „Ripe apples.“, der vom Prototyp und WIT korrekt als unbekannt interpretiert wird. Die beiden, aber auch API.AI und ALEXA erkennen die Entities die darin enthalten sind. Der Satz „Apple is a company located in Silicon Valley.“ wird zwar vom Prototyp als unbrauchbar erkannt, jedoch wird das Wort „Apple“ als Produktname erkannt. Auch LUIS macht diesen Fehler. Alle anderen erkennen korrekter Weise keine Entities. Getestet wurde auch der deutsche Satz „Zeig mir reife Äpfel.“, bei dem der Intent nur von API.AI und ALEXA falsch zugewiesen wurde, was jedoch die Frage aufwirft, wie dies mit Wörtern, die es in der englischen Sprache nicht gibt, möglich sein kann.

5.2.5 Synonyme, Listen und Wortvariationen

Es wird evaluiert, ob nicht trainierte Synonyme der bekannten Produktnamen erkannt werden. Dies ist wichtig, da das an den NLPS angebundene System diese Synonyme nicht kennen würde und daher das Wort nicht verarbeiten könnte. Nur der Prototyp ist dank der Wortvergleiche durch `nltk` in der Lage, diese Aufgabe zu bewältigen und erkennt aus den Sätzen „Show me filberts.“ und „Show me butternuts.“ korrekt die Wörter „hazelnut“ und „walnut“. Listen an Entities, wie aus dem Satz „Show me ripe apples, pears and tomatoes.“ werden ebenfalls nur vom Prototyp vollständig extrahiert. WIT extrahiert „apples, pear“, API.AI nur „tomato“, und die beiden übrigen erkennen keine Namen. Bekannte Synonyme, wie „Berner Rose“, werden, außer von ALEXA und API.AI, sehr gut extrahiert.

5.2.6 Zahlen

Der wohl schwierigste Testbefehl dieser Reihe ist „I want to have twenty four ripe, orange carrots.“, der vom Prototyp fehlerfrei erkannt wird. Während API.AI zwar Zahlen in Ziffernform erkennt, werden Zahlen, die in mehreren Wörtern geschrieben sind, nicht erkannt. Alle übrigen Systeme erkennen weder Zahlen in Ziffern- noch in ausgeschriebener Form.

5.3 Gesamtleistung

Zusammenfassend ergeben sich die Gesamtleistungen, die in Tabelle 5.2 aufgelistet sind. Unter den fünf getesteten NLPS, weist der Prototyp in jeder Disziplin die höchste Leistung auf und erzielt gesamt 94%. Auf Platz zwei, mit gleicher Leistung der Intent-Erkennung, jedoch mit deutlich geringerer Trefferquote bei der Extraktion der Entities, ist WIT mit einem Gesamtwert von 85%. Ähnlich gute Ergebnisse erzielte auf Platz drei API.AI mit gesamt 80%. Vorletzter ist ALEXA mit 75% und deutlich zurück liegt LUIS mit 48% als letzter.

Da die Testsätze beliebig gewählt wurden und für eine aussagekräftige, quantitative Analyse Daten realer User in großer Zahl erforderlich sind, ist diese Leistungsbestimmung mit einer gewissen Unschärfe behaftet. Dennoch kann davon ausgehend eine

Tabelle 5.2: Gesamtvergleich der Leistung zwischen den getesteten NLPS.

NLPS	*Intent*	*Entity-Name*	*Entity-Wert*	*Gesamt*
Prototyp	97%	94%	91%	94%
LUIS	61%	43%	38%	48%
API.AI	88%	79%	74%	80%
WIT	97%	82%	75%	85%
ALEXA	86%	72%	66%	75%

Einschätzung gemacht werden, in welcher Größenordnung sich die finalen Leistungen bewegen.

Tabelle 5.3: Testergebnisse der Testdaten.

NLPS	*Intent*	*Entity-Name*	*Entity-Wert*	*Gesamt*
Einfache Befehle				
Prototyp	100%	100%	100%	100%
LUIS	75%	0%	0%	25%
API.AI	100%	88%	88%	92%
WIT	100%	100%	100%	100%
ALEXA	100%	50%	50%	67%
Eingabequalität				
Prototyp	100%	75%	75%	83%
LUIS	75%	75%	75%	75%
API.AI	100%	63%	63%	75%
WIT	100%	75%	75%	83%
ALEXA	100%	75%	75%	83%
Komplexe Befehle				
Prototyp	83%	100%	75%	86%
LUIS	50%	53%	28%	44%
API.AI	33%	72%	44%	50%
WIT	67%	100%	58%	75%
ALEXA	50%	83%	41%	58%
Unbekannte Intents				
Prototyp	80%	70%	70%	73%
LUIS	20%	60%	60%	47%
API.AI	40%	50%	50%	47%
WIT	100%	90%	90%	93%
ALEXA	0%	90%	90%	60%
Synonyme, Listen und Wortvariationen				
Prototyp	100%	95%	95%	97%
LUIS	71%	57%	43%	57%
API.AI	100%	24%	10%	44%
WIT	100%	71%	50%	74%
ALEXA	100%	50%	36%	62%
Zahlen				
Prototyp	100%	100%	100%	100%
LUIS	60%	40%	33%	44%
API.AI	100%	80%	73%	84%
WIT	100%	67%	60%	76%
ALEXA	100%	67%	60%	76%

Kapitel 6

Resümee

In den letzten Jahren haben sich einige Natural Language Processing Services etabliert, die vereinzelt weitgehende Möglichkeiten der Textverarbeitung und des Dialogmanagements bereitstellen. Im Zuge dieser Arbeit wurde die Textinerpretationsfähigkeit dieser Systeme eruiert und ein eigener Prototyp entwickelt, der mit ihnen verglichen wurde. Im Vordergrund stand dabei die Nutzbarkeit dieser Services in der Praxis, weshalb ein Testszenario mit vier Intents und drei Entity-Klassen entstand. Damit konnten nicht nur Erkenntnisse über das Training, sondern auch über die Genauigkeit und die Qualität der Ergebnisse gewonnen werden. Dabei lag der Fokus auf der Frage, ob bestehende, für allgemeine Nutzungsszenarien geschaffene Services eine zufriedenstellende Leistung erbringen, oder eine eigene, zweckoptimierte Implementierung geeigneter ist. Im Zuge der Entwicklung des Prototyps wurde umfangreich auf zugrundeliegende Technologien eingegangen, und zahlreiche Techniken vorgestellt, die für die Aufbereitung sowie die Interpretation von Texten in menschlicher Sprache geeignet sind. Zur Überprüfung der Leistung und Analyse der Service-Antworten wurde eine Testapplikation eingerichtet, die an alle Systeme angebunden ist und die Trefferquote automatisch ermittelt.

6.1 Resultat

Die Tests zeigten, dass die bestehenden Online-Services, zumindest bei dem gewählten Testszenario mit dem entsprechenden Trainings- und Testkorpus, deutlich schlechtere Leistungen erzielten, als der domänenspezifisch, zweckoptimert entwickelte Prototyp. Die verwendeten Systeme wiesen vereinzelt, selbst bei einfachen Befehlen, schwer nachvollziehbare Fehler auf, in denen Intents falsch und Entities fehlerhaft oder gar nicht erkannt wurden. Aufgrund der mangelhaften Einstellbarkeiten der Services, ließ die Qualität der Ergebnisse zu wünschen übrig. Wäre es beispielsweise möglich, eine Entity als Zahl (in Ziffern) zu definieren, deren Worttyp (Nomen, Adjektiv etc.) festzulegen oder den Numerus zu definieren (z. B. immer Singular), würde das die Qualität der Antwort erheblich steigern, was den Mehraufwand und die Fehlerquote bei der Weiterverarbeitung drastisch reduzieren würde. Auch das automatische Erkennen von Wortähnlichkeiten und damit das Erweitern der Entity-Synonyme, wurde in den Tests vermisst. Das detailierte Fazit zu den Systemen im Rahmen des Testszenarios setzt sich wie folgt zusammen:

- **LUIS:** Obwohl der Service vielversprechend wirkte, da Entities genauer einstellbar waren als bei anderen Systemen, war die Gesamtleistung von 48% unbrauchbar. Die Entities wurden kaum erkannt, und im User-Interface traten häufig Fehler beim Speichern und Trainieren auf. LUIS wirkte insgesamt unausgereift und unbrauchbar.
- **API.AI:** Für die Textinterpretation gab es weniger Einstellbarkeiten als beispielsweise bei LUIS, jedoch waren die Ergebnisse dennoch insgesamt zufriedenstellend. API.AI war der einzige unter den Probanden, der Zahlen erkannte, wies generell eine gute Qualität auf und erzielte eine Gesamtleistung von 80%.
- **WIT:** Mit seinem minimalistischen User-Interface zur Eingabe der Trainingsdaten erweckte WIT zunächst keinen guten Eindruck. Durch die ausgezeichnete Erkennung des Intents und der Entities erwies sich dieses System aber als bestes unter den getesteten. Die Mischung aus einfacher Wartbarkeit und guter Leistung von 85% machen daraus einen nützlichen Service.
- **ALEXA:** Auch wenn ALEXA kein klassischer NLPS ist, wurde er dennoch in die Testreihe aufgenommen. Das praktisch nicht vorhandene User-Interface erschwerte die Wartbarkeit und das Zurechtfinden im System. Mit einer dürftigen Entity-Erkennung und einer Gesamtleistung von 75% ist ALEXA als NLPS eher unbrauchbar.
- **Prototyp:** Die Leistung von 94% und die Qualität der Ergebnisse aus der Entity-Erkennung zeigten deutlich, dass ein domänenspezifischer NLPS nicht nur eine sehr hohe Trefferquote aufweist, sondern auch die Weiterverarbeitung des Ergebnisses enorm vereinfacht.

Das Fazit der gesamten Untersuchungen und Testdurchläufe ist, dass einige der derzeit verfügbaren NLPS zwar brauchbar sind, jedoch die Qualität der Ergebnisse zu wünschen übrig lässt. Die Weiterverarbeitung der Antworten erfordert teilweise erst eine Verarbeitung mittels NLPS, beispielsweise beim Übersetzen von textuellen Zahlen in Ziffern. Aus diesem Grund erwies sich der Aufwand der Entwicklung eines domänenspezifischen Systems, das automatisch Intents und Entities aus Befehlen in menschlicher Sprache erkennt, als gerechtfertigt, da damit das Szenario bestmöglich verarbeitet und die Qualität maximiert werden konnte.

6.2 Ausblick

Obwohl der Prototyp bereits auf hochentwickelte Grammatiksysteme zurückgreift, sind die Regeln und das Verhalten nur rudimentär implementiert. Diese könnten z. B. mittels Chunking-Regeln soweit verbessert werden, dass Gliedsätze oder einfache und mehrfache Verneinungen erkannt werden können. Für eine verbesserte Erkennung von unbrauchbaren Befehlen könnte eine Identifikation der Sprache der eigentlichen Verarbeitung vorgeschalten sein. Ebenso wäre die Vorverarbeitung von Abkürzungen sinnvoll, damit der Befehl zur Interpretation in einwandfreiem, ausgeschriebenem Zustand vorliegt. Es finden sich somit noch zahlreiche Potentiale, um aus dem Prototyp einen ausgereiften Service zu machen.

Anhang A

Übersicht der Machine Learning Decision Borders

Zum Zweck der besseren Vergleichbarkeit sind in Abbildung A.1 auf der nächsten Seite noch einmal die Decision Borders aller vorgestellten ML-Algorithmen übersichtlich zusammengefasst.

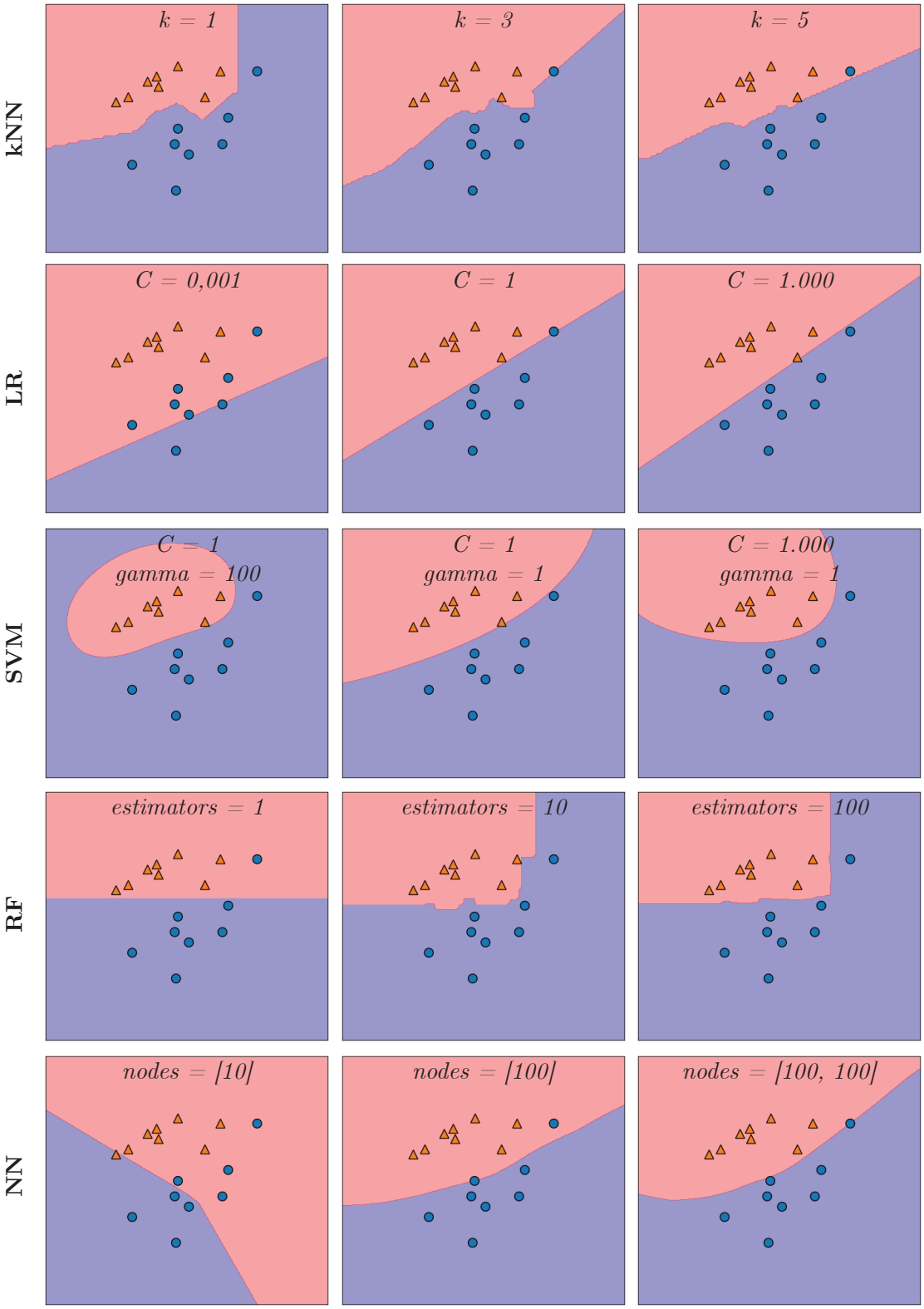


Abbildung A.1: Übersicht der ML-Decision Borders.

Anhang B

Installationsanleitung

B.1 NLPS-Prototyp

1. Python[1] muss in der Version 3.6.0 (oder höher) am Betriebssystem installiert sein.
2. Für das Python-Projekt des NLPS-Prototyps müssen die folgenden Module installiert sein: `mglearn`[2], `numpy`[3], `sklearn`[4], `nltk`[5], `spacy`[6] und `sanic`[7].
3. Im Verzeichnis des Prototyps, in dem sich die Datei `main.py` befindet, wird der NLPS-Prototyp in der Konsole mit `python main.py` ausgeführt. Der Webservice ist dann unter http://localhost:4000 erreichbar.

B.2 Testapplikation

1. NodeJS[8] in der Version 6.10.3 (oder höher) und `npm`[9] in der Version 5.0.1 (oder höher) müssen am Betriebssystem installiert sein.
2. Im Verzeichnis der Testapplikation, in dem sich die Datei `package.json` befindet, werden die Abhängigkeiten des Projektes mit `npm install` installiert.
3. Gestartet kann die Testapplikation mit `npm start` werden. Im Browser ist diese dann unter http://localhost:3000 erreichbar.

[1] https://www.python.org/downloads/
[2] https://github.com/amueller/mglearn
[3] https://www.scipy.org/install.html
[4] http://scikit-learn.org/stable/install.html
[5] http://www.nltk.org/data.html
[6] https://spacy.io/docs/usage/
[7] https://github.com/channelcat/sanic
[8] https://nodejs.org/en/download/
[9] https://docs.npmjs.com/getting-started/installing-node

Anhang C

Trainingsdaten

In Tabelle C.1 sind die Entities aufgelistet, die in die Rohbefehle der Tabelle C.2 auf der nächsten Seite eingesetzt werden.

Table C.1: Werte der Entities, die in die Rohbefehle eingesetzt werden. In Klammer stehende Werte repräsentieren entsprechende Synonyme.

Entity	*Wert*
`product_name`	**fruit** apple (Jonagold, Gala, Berner Rose, Fuji, Granny Smith, Macintosh, Pink Lady, Golden Delicious, Winesap), orange (blood orange), pear (Williams Christ, Concorde), banana, hazelnut, walnut, cranberry, goji, kiwifruit, lemon, pineapple (pine-apple), peach, strawberry **vegetable** tomato, carrot, pumpkin
`product_condition`	ripe, fresh, seasonal, available, cooked, peeled
`product_quantity`	a few, all, some

Table C.2: Rohbefehle, bei denen die Platzhalter durch die entsprechenden Entities ersetzt werden. In `N` werden die Einträge von `product_name`, in `C` die von `product_condition` und in `Q` jene von `product_quantity` eingesetzt.

Intent	*Rohbefehl*
`show_product`	Show me `Q N`. I want to see `C` products. Show me `C` products. I want to see `Q N`. Give me `Q N`. Display `Q N`. I am interested in seeing `Q C N`.
`order_product`	I want to order `Q N`. I want to have `Q N`. I want to purchase `Q N`. Send me `Q N`. I want to buy `Q C N`. I would like to have `Q C N`. Could I have `Q C N`?
`show_seller`	Where can I buy `Q C N`? Who sells `N`? Who offers `C N`? Who has `C N`? Where can I get `N`? Show me the location where I can buy `Q C N`. Where are `C N` sold?
`show_availability`	When can I get `N`? When are `N C`? In which month can I have `N`? Which products are `C`? Are `N` currently `C`? Do they have `C N` today? When are `N` in the shelves?

Anhang D

Inhalt der CD-ROM

D.1 Masterarbeit

Pfad: /

Berneder_Roland_2017.pdf Masterarbeit

D.2 Literatur

Pfad: /literatur

e-mail_von_WIT.pdf . E-Mail von WIT bzgl. Reifegrad der verwendeten Sprachmodelle

Pfad: /literatur/online-quellen

*.htm . . . . . Kopien der besuchten, referenzierten Websites

D.3 Quellcode

Pfad: /nlps-prototyp

main.py . . . . . Auszuführende Datei um den NLPS zu starten

snippet*.py . . . . . Code-Fragmente, die für die Wahl der NLPS Technologien verwendet wurden

Pfad: /nlps-prototyp/thesis_examples

*.py . . . . . Code-Fragmente, die zur Erzeugung der Grafiken in der Masterarbeit verwendet wurden

Pfad: /testapplikation

app.js . . . . . Auszuführende Datei um die Testapplikation zu starten

D.4 Konfigurationsdaten der verwendeten NLPS

Pfad: /nlps-konfigurationen

*.zip	Konfigurationsdaten mit denen die jeweiligen NLPS-Agenten wiederhergestellt werden können

D.5 Testergebnisse

Pfad: /nlps-ergebnisse

results*.json	HTTP-Responses auf die Anfragen der Testbefehle
testapplikation.jpg . . .	Screenshot der Testapplikation mit den Ergebnissen der NLPS

Quellenverzeichnis

Literatur

[1] Steven Abney. „Partial Parsing via Finite-State Cascades“. *Natural Language Engineering* 2.04 (1996), S. 337–344 (siehe S. 25).

[2] Steven Abney. „Part-of-Speech Tagging and Partial Parsing“. In: *Corpus-Based Methods in Language and Speech Processing*. Hrsg. von Steve Young und Gerrit Bloothooft. Houten, Niederlande: Springer, 1997, S. 118–136 (siehe S. 26).

[3] Janet Baker u. a. „Historical Development and Future Directions in Speech Recognition and Understanding“. In: *Proceedings of the MINDS Report of the Speech Understanding Working Group*. (Chantilly, Virginia, USA). Feb. 2007 (siehe S. 1).

[4] Rodrigo C. Barros u. a. „A Survey of Evolutionary Algorithms for Decision-Tree Induction“. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.3 (2012), S. 291–312 (siehe S. 11).

[5] James Bergstra und Yoshua Bengio. „Random Search for Hyper-Parameter Optimization“. *Journal of Machine Learning Research* 13 (2012), S. 281–305 (siehe S. 16).

[6] Michael W. Berry und Malu Castellanos. *Survey of Text Mining II. Clustering, Classification, and Retrieval*. London, UK: Springer, 2008 (siehe S. 18).

[7] Kevin S. Beyer u. a. „When is “Nearest Neighbor” Meaningful?“ In: *Proceedings of the 7th International Conference on Database Theory*. (Jerusalem, Israel). London, UK: Springer, 1999, S. 217–235 (siehe S. 10).

[8] Steven Bird, Ewan Klein und Edward Loper. *Natural Language Processing with Python. Analyzing Text with the Natural Language Toolkit*. Sebastopol, California, USA: O'Reilly Media, 2009 (siehe S. 17, 21, 36).

[9] Ronan Collobert u. a. „Natural Language Processing (Almost) from Scratch“. *Journal of Machine Learning Research* 12 (2011), S. 2493–2537 (siehe S. 18).

[10] Jerome Friedman, Trevor Hastie und Robert Tibshirani. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. New York, NY, USA: Springer, 2001 (siehe S. 8, 11, 14).

[11] Roger Garside. „The CLAWS Word-Tagging System“. *The Computational Analysis of English: A Corpus-Based Approach* (1987), S. 30–41 (siehe S. 22).

[12] Xuedong Huang, Alex Acero und Hsiao-Wuen Hon. *Spoken Language Processing. A Guide to Theory, Algorithm, and System Development*. Upper Saddle River, New Jersey, USA: Prentice Hall, 2001 (siehe S. 2, 17).

[13] Xuedong Huang, James Baker und Raj Reddy. „A Historical Perspective of Speech Recognition“. *Communications of the ACM* 57.1 (2014), S. 94–103 (siehe S. 2).

[14] Peter Jackson und Isabelle Moulinier. *Natural Language Processing for Online Applications. Text Retrieval, Extraction and Categorization*. 5. Aufl. Amsterdam, Niederlande: John Benjamins Publishing, 2007 (siehe S. 19, 23).

[15] Thorsten Joachims. „Text Categorization with Support Vector Machines. Learning with Many Relevant Features“. In: *Proceedings of the 10th European Conference on Machine Learning*. (Chemnitz, Deutschland). Hrsg. von Claire Nedellec und Celine Rouveirol. Springer, Apr. 1998, S. 137–142 (siehe S. 11).

[16] Daniel Jurafsky und James H. Martin. *Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, New Jersey, USA: Prentice Hall, 2000 (siehe S. 21, 22).

[17] Ron Kohavi. „A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection“. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. (Montreal, Canada). San Francisco, California, USA: Morgan Kaufmann Publishers, 1995, S. 1137–1143 (siehe S. 15).

[18] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. „Deep Learning“. *Nature* 521.7553 (2015), S. 436–444 (siehe S. 14, 48).

[19] George A. Miller u. a. „Introduction to WordNet. An On-Line Lexical Database“. *International Journal of Lexicography* 3.4 (1990), S. 235–244 (siehe S. 25).

[20] Andreas C. Müller und Sarah Guido. *Introduction to Machine Learning with Python. A Guide for Data Scientists*. 2. Aufl. Sebastopol, California, USA: O'Reilly Media, 2017 (siehe S. 4, 5, 8, 11).

[21] Andrew Y. Ng. „Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance“. In: *Proceedings of the 21st International Conference on Machine Learning*. (Banff, Canada). 2004, S. 78–85 (siehe S. 11).

[22] June H. Park u. a. „Economic Load Dispatch for Piecewise Quadratic Cost Function Using Hop“. *IEEE Transactions on Power Systems* 8.3 (1993), S. 1030–1038 (siehe S. 14).

[23] Ted Pedersen, Siddharth Patwardhan und Jason Michelizzi. „WordNet::Similarity. Measuring the Relatedness of Concepts“. In: *Proceedings of Demonstration Papers at Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*. (Boston, Mass., USA). 2004, S. 38–41 (siehe S. 25).

[24] Octavian Popescu und Ngoc Phuoc An Vo. „Fast and Accurate Misspelling Correction in Large Corpora.“ In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. (Doha, Qatar). Okt. 2014, S. 1634–1642 (siehe S. 35).

[25] Ah-Hwee Tan. „Text Mining. The State of the Art and the Challenges“. In: *Proceedings of the Pacific Asia Knowledge Discovery of Data-Mining*. (Beijing, China). Hrsg. von Ning Zhong und Lizhu Zhou. Bd. 8. Apr. 1999, S. 65–70 (siehe S. 17).

[26] Erik F. Tjong Kim Sang und Sabine Buchholz. „Introduction to the CoNLL-2000 shared Task: Chunking“. In: *Proceedings of the 2nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning*. (Lisbon, Portugal). Hrsg. von Claire Nedellec. 2000, S. 127–132 (siehe S. 25).

[27] Alan M. Turing. „Computing Machinery and Intelligence“. *Mind* 59.236 (1950), S. 433–460 (siehe S. 1, 17).

[28] Leslie G. Valiant. „A Theory of the Learnable“. *Communications of the ACM* 27.11 (1984), S. 1134–1142 (siehe S. 4).

[29] S. Vijayarani, J. Ilamathi und Nithya. „Preprocessing Techniques for Text Mining – An Overview“. *International Journal of Computer Science & Communication Networks* 5.1 (2015), S. 7–16 (siehe S. 22).

[30] Juan Xu, Qi Zhang und Xuanjing Huang. „Understanding the Semantic Intent of Domain-Specific Natural Language Query“. In: *Proceedings of the International Joint Conference on Natural Language Processing*. (Nagoya, Japan). Okt. 2013, S. 552–560 (siehe S. 51).

[31] Zhou Yao und Cao Ze-Wen. „Research on the Construction and Filter Method of Stop-Word List in Text Preprocessing“. In: *Proceedings of the Conference on Intelligent Computation Technology and Automation*. (Shenzhen, China). März 2011, S. 217–221 (siehe S. 22).

Online-Quellen

[32] Pavlo Bashmakov. *Advanced Natural Language Processing Tools for Bot Makers – LUIS, Wit.ai, Api.ai and Others (UPDATED)*. URL: https://stanfy.com/blog/advanced-natural-language-processing-tools-for-bot-makers/ (besucht am 18. 05. 2017) (siehe S. 26).

[33] *Dive Into NLTK, Part IV: Stemming and Lemmatization*. URL: http://textminingonline.com/dive-into-nltk-part-iv-stemming-and-lemmatization (besucht am 16. 05. 2017) (siehe S. 21, 22).

[34] Scott Huffman. *Making Conversational Interfaces Easier to Build*. URL: https://developers.googleblog.com/2016/09/making-conversational-interfaces-easier-to-build.html (besucht am 04. 06. 2017) (siehe S. 28).

[35] Denise Mak. *Entities in LUIS*. URL: https://docs.microsoft.com/en-us/azure/cognitive-services/luis/luis-concept-entity-types (besucht am 20. 05. 2017) (siehe S. 26).

[36] Denise Mak und Hanna Carol. *Localization Support in LUIS Apps*. URL: https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-concept-language-support (besucht am 24. 05. 2017) (siehe S. 37).

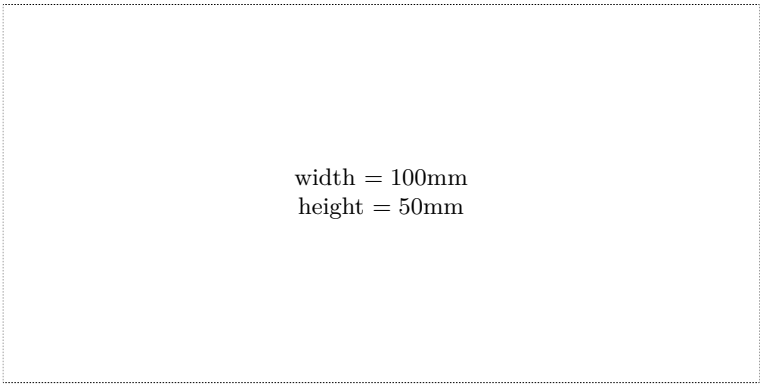
[37] Bernard Marr. *A Short History of Machine Learning – Every Manager Should Read*. URL: https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#390aef2515e7 (besucht am 02. 05. 2017) (siehe S. 4).

[38] Joseph Misiti. *What Programming Languages Are Suitable for Natural Language Processing?* URL: https://www.quora.com/What-programming-languages-are-suitable-for-natural-language-processing (besucht am 23. 05. 2017) (siehe S. 36).

[39] Luca Scagliarini und Marco Varone. *Natural Language Processing and Text Mining*. URL: http://www.expertsystem.com/natural-language-processing-and-text-mining/ (besucht am 15. 05. 2017) (siehe S. 17, 18).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —