5 Dialoge und Intentionen

Unser Sprachassistent hat jetzt schon einige Fähigkeiten erlangt, die es theoretisch ermöglichen, mit einem Menschen zu kommunizieren. Versuchen Sie doch mal mit ein paar einfachen If-Statements eine Antwort auf Fragen wie *Wie spät ist es?* oder *Wie heißt du?* zu bekommen. Spätestens nach der fünften manuell geschriebenen Frage-Antwort-Kombination werden Sie merken, dass der Code immer komplexer wird. Weiterhin werden Sie wahrscheinlich feststellen, dass eine Frage auf viele verschiedene Arten formuliert werden kann. Alleine zur Frage nach der Uhrzeit fallen mir direkt folgende Variationen ein:

- Wie spät ist es?
- Wie viel Uhr ist es?
- Uhrzeit
- Wie spät haben wir es?
- Wie spät ist es jetzt?
- Ich wüsste gerne, wie spät es ist.

Dazu kommt, dass sich eine Frage auch noch parametrisieren lässt, z. B. *Wie spät ist es in Tokyo?* Andere lassen sogar mehrere Parameter zu, die alle einzeln verarbeitet werden müssen. Aber alles der Reihe nach. In diesem Kapitel werden wir uns erst einmal anschauen, wie wir eine Intention eines Benutzers deuten. Dabei verfolgen wir zwei Ansätze:

1. Einen auf Basis fest vorgegebener Muster, die zwar teilweise flexibel in der Erkennung sind, aber dennoch explizit angegeben werden müssen.
2. Einen zweiten auf Basis maschinellen Lernens, in dem wir einige Sätze und Entitäten vorgeben und mit Annotationen versehen und daraus ein Modell trainiert wird.

Dadurch bieten wir einerseits den zahlreichen Entwicklern, die für unseren herausragenden Open-Source-Sprachassistenten später weitere Intents entwickeln, zwei Wege, die Eingaben des Benutzers zu interpretieren. Andererseits, und dieses Ziel ist wohl im Moment noch eher das naheliegende, lernen Sie auch auf praktische Art und Weise den Unterschied zwischen regelbasierter und lernender KI kennen. Sie werden lernen zu entscheiden, wann maschinelles Lernen sinnvoll ist und wann Sie pragmatisch und zielorientiert zu einem klassischen Ansatz greifen sollten.

5.1 Intent Recognition – Die menschliche Intention verstehen lernen

Wir Menschen können uns über viele Kanäle äußern, akustisch durch Worte und Tonfall, durch Gesten und Mimik oder auch indem wir ... nichts tun. Der Kommunikationswissenschaftler Paul Watzlawick hat einmal fünf Axiome zur menschlichen Kommunikation formuliert (Watzlawik, Beavin, & Don, 2000):

- Man kann nicht nicht kommunizieren.
- Jede Kommunikation hat einen Inhalts- und einen Beziehungsaspekt.
- Kommunikation ist immer Ursache und Wirkung.
- Menschliche Kommunikation bedient sich analoger und digitaler Modalitäten.
- Kommunikation ist symmetrisch oder komplementär.

Auch wenn ich mich tatsächlich freue, etwas aus dem theoretischen Deutschunterricht aus der Oberstufe heute im Berufsleben tatsächlich noch im Kopf zu haben, kommt die ernüchternde Feststellung doch recht schnell, dass für die Kommunikation von Mensch und Maschine ganz andere Regeln gelten. Anders formuliert: Wir müssten einem Computer erst lehren, diese fünf Grundregeln zu beachten, sofern das in unserem Sinne wäre. Aber ist es das? Möchten wir, dass ein Sprachassistent gelangweilt wird, wenn wir länger nicht mit ihm sprechen? Oder soll er auf den gestressten Ton nach einem langen Arbeitstag damit reagieren, dass er eher beruhigenden Jazz spielt, statt, wie von uns gewünscht, die Nachrichten vorzulesen? Wahrscheinlich eher nicht. Unsere Erwartungshaltung an eine Maschine ist, dass sie tut, wozu wir sie auffordern. Die Gesetzmäßigkeiten der menschlichen Kommunikation gelten also nicht, auch wenn ich zugebe, dass es sicher einen gewissen Unterhaltungswert hätte, sie zu simulieren.

Wie aber sehen die Herausforderungen bei der Mensch-Maschine-Kommunikation aus? Tatsächlich werden wir uns nur mit einem kleinen Feld dieser doch recht umfangreichen Disziplin beschäftigen, nämlich dem Computer beizubringen, unsere Intentionen einzuordnen und zu interpretieren. Die Disziplin der *Intent Recognition* teilt sich also auf, wie in Bild 5.1 zu sehen ist.

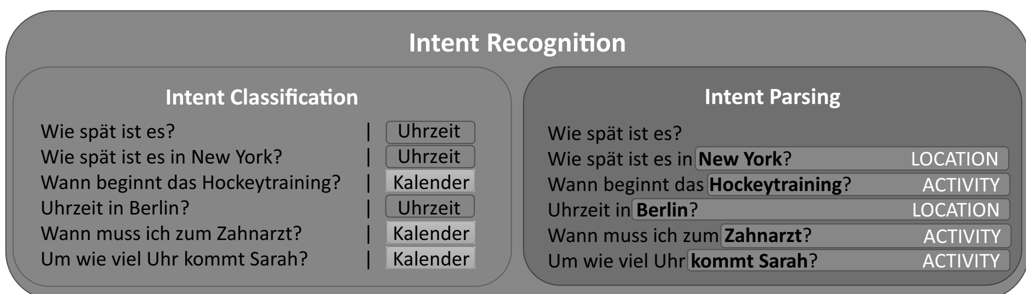


Bild 5.1 Die Erkennung von Intents gliedert sich auf in die Klassifizierung von Intentionen und in das syntaktische Verständnis des Befehls und seiner Parameter.

Unter vielen verschiedenen Funktionen, die unser Sprachassistent anbietet, muss per Klassifikation zuerst herausgefunden werden, welche am besten dafür geeignet ist, den Wunsch des Benutzers zu erfüllen. Im Beispiel in Bild 5.1 unterscheiden wir zwischen dem Intent *Uhrzeit* und dem Intent *Kalender*. Ersterer gibt die Uhrzeit in einer bestimmten Zeitzone aus, letzterer nennt den Zeitpunkt eines bestimmten Ereignisses im Kalender. Der Classifier muss nun mithilfe von einem passenden Set an Trainingsdaten so trainiert werden, dass er zwischen den beiden Intents entscheiden kann.

Ist dieses Ziel erreicht, können wir zum zweiten Schritt übergehen und den Satz über einen Parser analysieren, sodass wir etwaige Parameter extrahieren können. Im Falle der Urzeit wäre es zum Beispiel ein Ort oder eine Aktivität, in Bild 5.1 als *LOCATION* und *ACTIVITY* gekennzeichnet.

5.1.1 Intent Classifier am Beispiel

Da das alles sehr abstrakt klingt und das hier ja ein Praxisbuch ist, wollen wir uns kurz zur Untermauerung des Gelesenen eine Anwendung schreiben, die in der Lage ist, Texte, darunter auch Intents, zu klassifizieren.

Listing 5.1 Textklassifikation – Laden der Trainings- und Testdaten

```
1.  import spacy
2.  from spacy.util import minibatch, compounding
3.  import time, random, os
4.  from sklearn.metrics import classification_report
5.  import pandas as pd
6.  from sklearn.utils import shuffle
7.  import numpy as np

8.  def load():
9.    df_data = pd.read_csv(os.path.join(os.path.dirname(os.path.abspath(__file__)),
        "data.csv"))
10.
11.   df_data = shuffle(df_data)
12.   df_data.reset_index(inplace=True, drop=True)
13.
14.   train, test = np.split(df_data, [int(len(df_data)*0.8)])
15.
16.   train_set = {'sentences': train.text_de.tolist(),
        'intents': train.intent_index.tolist()}
17.   test_set = {'sentences': test.text_de.tolist(), 'intents': test.intent.tolist()}
18. return train_set, test_set
```

Sie sehen an den Imports, dass wir *SpaCy* und dessen Textklassifikations-Pipeline verwenden, um unsere Intentionserkennung zu trainieren. Weiterhin setzen wir *scikit-learn*, *pandas* und *NumPy* ein – drei Bibliotheken, auf die Sie immer wieder stoßen werden, wenn Sie sich mit maschinellem Lernen und der Manipulation und Analyse von Daten beschäftigen.

Listing 5.1 beginnt mit der Funktion *load*, die eine CSV-Datei (*Comma-separated Values*) einliest. Diese ist wie in Tabelle 5.1 zu sehen aufgebaut.

Tabelle 5.1 Trainings- und Testdaten für die Textklassifikation

text_en	text_de	text_fr	text_es	intent	intent_index
...	Buche Restaurant in Ulm	...	...	BookRestaurant	1
...	Füge Pink zu meiner Playlist hinzu	...	...	AddToPlaylist	0
...	Wie ist das Wetter in Landau	...	...	GetWeather	2
...	Spiele Musik von Queen	...	...	PlayMusic	3

Die Daten entstammen einem NLU (*Natural Language Understanding*) Benchmark von *Sonos*[1]. Ich habe sie jedoch bereits ein wenig umformatiert, sodass sie leichter auf unser Training anzuwenden sind. Im Beispiel *100_extras/100_04_intentclassifier* finden Sie das Skript *preprocessing.py*, das für diese Vorverarbeitung verantwortlich ist. Es fügt die einzelnen, mit Tags versehenen Satzteile zu ganzen Sätzen zusammen und übersetzt sie aus dem Englischen (*text_en*) ins Deutsche (*text_de*) und für alle Interessenten auch noch ins Spanische (*text_es*) und Französische (*text_fr*). Wir werden jedoch alleine mit der deutschen Übersetzung arbeiten. Dazu kommt noch das Label, das den eigentlichen Intent angibt, sowie eine numerische Repräsentation dieses Intents in *intent_index*.

TIPP: Wenn Sie sich schon ein wenig länger mit KI beschäftigen, werden Sie gemerkt haben, dass Fortschritte meist erst in Beispielen in englischer Sprache präsentiert werden. Das ist aus meiner Sicht in Ordnung, sprechen wir diese Sprache doch fast alle und haben somit eine gemeinsame Grundlage, um die Entwicklung kollaborativ voranzutreiben. Doch in diesem Fall macht es uns natürlich die Arbeit etwas schwer, denn wir wollen ja deutsche Texte klassifizieren können und nicht nur englische. Bei Modellen auf Basis numerischer Werte müssen wir vielleicht höchstens Metriken anpassen, aber nichts übersetzen. Zum Glück ist es aber heutzutage gar nicht mehr so schwer, Texte von einer Sprache in die andere zu übersetzen. Das Beispiel aus der Vorverarbeitung für das Beispiel der Textklassifikation möchte ich Ihnen nicht vorenthalten.

Listing 5.2 Maschinelle Übersetzung mit Transformers

```
1.  from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
2.  import torch
3.
4.  device = „cuda:0" if torch.cuda.is_available() else „cpu"
5.  tokenizer_en_de = AutoTokenizer.from_pretrained
      („Helsinki-NLP/opus-mt-en-de")
6.  model_en_de = AutoModelForSeq2SeqLM.from_pretrained
      („Helsinki-NLP/opus-mt-en-de").to(device)
7.
8.  text = „This example has to be translated."
9.
10. with tokenizer_en_de.as_target_tokenizer():
```

[1] *https://github.com/sonos/nlu-benchmark*

```
11.   tokenized_text = tokenizer_en_de(text, return_tensors='pt').to(device)
12.   translation = model_en_de.generate(**tokenized_text)
      return tokenizer_en_de.batch_decode(translation,
        skip_special_tokens=True)[0]
```

Listing 5.2 lädt jeweils ein Modell für einen *Tokenizer*, der die zu übersetzende Zeichenkette in ihre Einzelteile zerlegt, sowie ein *Sequence-To-Sequence*-Modell, das, wie der Name vermuten lässt, Datenstrukturen einer Sequenz in eine andere Sequenz transformieren kann.

Die bereits angesprochene Bibliothek *transformers* von *Hugging Face* ist hier eine große Erleichterung, denn neben der Klasse, die uns Initialisierung und Benutzung einfach machen, hostet das Unternehmen auch die entsprechenden Modelle. In unserem Fall ist das *opus-mt-en-de* der Universität Helsinki. Indem wir dessen Namen bei der Initialisierung von *AutoTokenizer* und *AutoModelForSeq2SeqLM* angeben, werden diese vom *Hugging Face Model Hub*[2] heruntergeladen. Dort finden Sie zahlreiche vortrainierte Modelle für verschiedenste Aufgaben aus dem Speech- und NLP-Bereich. Die Universität Helsinki bietet dort kostenfrei 1334 Modelle an, die jeweils von einer in eine andere Sprache übersetzen (alleine für die Übersetzung ins Deutsche bzw. aus dem Deutschen sind dort 52 Modelle zu finden). Die Qualität der Übersetzung ist bis auf einige wenige Ausnahmen, die vorrangig durch Eigennamen hervorgerufen werden, hervorragend. Sie können sich davon aber ein eigenes Bild machen, indem Sie sich die Trainingsdaten in *data.csv* anschauen.

In Listing 5.2 wird nun der Text in Zeile 11 in einzelne Tokens zerlegt und in Batches decodiert, also aus der internen Repräsentation zurück in Text umgewandelt. Ein besonderes Augenmerk soll noch auf Zeile 3 und die Aufrufe `to(device)` gelegt werden. Wir prüfen hier, ob *CUDA* auf dem System installiert ist, sprich ob wir die Inference der Modelle auf einer GPU rechnen können. Wenn *device* also ein *CUDA*-Device referenziert, kann das *Sequence-to-Sequence*-Modell auf die GPU übertragen werden. Um es anzuwenden, müssen sich auch die Tokens im GPU-Speicher befinden. Dieser Transfer geschieht in Zeile 11. Sollte keine GPU zu Verfügung stehen, wird das Device *cpu* verwendet und alle Operationen finden eben auf der CPU statt. ■

Sind die Daten in ein sogenanntes Pandas *DataFrame* geladen, mischen wir sie in Listing 5.1 in Zeile 11 durch. Das hat den Sinn und Zweck, dass wir beim Aufteilen der Daten in Zeile 14 nicht nur Texte eines Intents in den Testdaten haben. Das wäre nämlich sehr wahrscheinlich, da die Daten nach Intent geordnet in der CSV-Datei abgespeichert sind. Wir erstellen in Zeile 12 einen neuen Index für das *DataFrame* und nutzen dann `numpy.split()`, um 80 % der Daten für das Training und den Rest für die Tests zurückzulegen. Haben Sie bereits das ein oder andere Machine-Learning-Buch gelesen, werden Sie wohl die Methode `train_test_split()` von *scikit-learn* verwendet haben, die eben auch diese Aufgabe erfüllt. Mit *numpy.split* können wir einen Datensatz aber auch in mehrere Gruppen aufteilen, falls wir z. B. noch einen Validierungsdatensatz benötigen. `train_test_split()` ist jedoch auf zwei Gruppen begrenzt.

[2] https://huggingface.co/models

In Zeile 16 und 17 konstruieren wir ein *Dictionary* (zu erkennen an den geschweiften Klammern) und fügen einen Eintrag mit dem Index *sentences* und einen mit dem Index *intents* hinzu. Nun ziehen wir einzelne Spalten aus dem *DataFrame* und konvertieren sie in eine Liste. Für den Trainingsdatensatz benutzen wir die deutschen Sätze und die numerischen Indizes der Intents. Für den Testdatensatz benutzen wir ebenfalls die deutschen Sätze, jedoch die Namen der Intents. Am Ende liefern wir beide *Dictionaries* per Return zurück.

Es folgt eine Methode zur Evaluierung des Modells, zu sehen in Listing 5.3.

Listing 5.3 Textklassifikation – Evaluierung der Trainingsergebnisse

```
1.  def evaluate(tokenizer, textcat, test_texts, test_cats):
2.    docs = (tokenizer(text) for text in test_texts)
3.    preds = []
4.    for i, doc in enumerate(textcat.pipe(docs)):
5.      scores = sorted(doc.cats.items(), key = lambda x: x[1],reverse=True)
6.      catList=[]
7.      for score in scores:
8.        catList.append(score[0])
9.      preds.append(catList[0])
10.
11.   labels = ["AddToPlaylist", "BookRestaurant", "GetWeather", "PlayMusic",
12.     "RateBook", "SearchCreativeWork", "SearchScreeningEvent"]
13.
14.   print(classification_report(test_cats, preds, labels=labels))
```

Die Sätze aus den Testdaten werden einzeln in die Textklassifikations-Pipeline eingespeist und die resultierenden *Scores* gesammelt, der Größe nach sortiert, der beste selektiert und dessen Kategorie als Zeichenkette in der Liste *preds* gespeichert. Alle Testtexte, die jeweiligen Vorhersagen und die dazugehören Labels werden der Funktion `classification_report()` übergeben, die diese auswertet und textuell aufbereitet, sodass wir sie über die Konsole ausgeben können. Bild 5.2 zeigt diese Ausgabe einmal nach der ersten Iteration des Trainings und einmal nach der zehnten.

Wie viel Arbeit *scikit-learn* uns mit `classification_report()` abnimmt, sieht man, wenn man sich die Berechnung der in Bild 5.2 aufgelisteten Metriken ansieht. Was aber sagen diese Werte genau aus? Es ist zu sehen, dass Precision, Recall F1-Score und Support für jede Klasse im Modell abgebildet werden.

- **Precision (Genauigkeit):** Wenn das Modell einen Text als Intent der Kategorie *AddToPlaylist* klassifiziert, wie oft hat es damit recht?
- **Recall (Trefferquote):** Wenn der Intent wirklich der Klasse *AddToPlaylist* angehört, wie oft ist die Klassifikation richtig?
- **F1-Score:** Ist der gewichtete Durchschnitt zwischen Precision und Recall. Dieser sagt aus, wie gut der Classifier für diese eine bestimmte Klasse im Vergleich zu allen anderen Klassen ist.
- **Support:** Der Support gibt an, wie oft ein Intent der jeweiligen Klasse in den Testdaten vorkommt.

Die Berechnung von Precision, Recall und F1-Score ist nicht allzu kompliziert und kann durch drei einfache Formeln abgebildet werden.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \qquad \text{Formel 5.1}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \qquad \text{Formel 5.2}$$

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \qquad \text{Formel 5.3}$$

Was aber sind *True Positive*, *False Positive*, *True Negative* und *False Negative*? Die folgende Liste liefert die Erklärung:

- **True Positive (TP):** Wie viele der Intents wurden korrekterweise einer Klasse zugeordnet?
- **False Positives (FP):** Wie viele der Intents wurden fälschlicherweise einer Klasse zugeordnet?
- **True Negatives (TN):** Wie viele Intents wurden korrekterweise einer Klasse nicht zugeordnet?
- **False Negatives (FN):** Wie viele Intents wurden fälschlicherweise einer Klasse nicht zugeordnet?

```
                      precision    recall  f1-score   support

       AddToPlaylist       0.86      0.92      0.89       365
      BookRestaurant       0.95      0.89      0.92       387
          GetWeather       0.88      0.96      0.92       384
           PlayMusic       0.85      0.93      0.89       426
            RateBook       0.91      0.92      0.92       385
  SearchCreativeWork       0.73      0.71      0.72       388
SearchScreeningEvent       0.86      0.71      0.78       422

            accuracy                           0.86      2757
           macro avg       0.86      0.86      0.86      2757
        weighted avg       0.86      0.86      0.86      2757

Iteration 0: 56.271087114000004 Sekunden

                      precision    recall  f1-score   support

       AddToPlaylist       0.99      0.98      0.98       390
      BookRestaurant       0.98      0.98      0.98       366
          GetWeather       0.98      0.98      0.98       388
           PlayMusic       0.95      0.96      0.95       435
            RateBook       0.98      0.99      0.99       353
  SearchCreativeWork       0.92      0.92      0.92       446
SearchScreeningEvent       0.94      0.92      0.93       379

            accuracy                           0.96      2757
           macro avg       0.96      0.96      0.96      2757
        weighted avg       0.96      0.96      0.96      2757

Iteration 9: 46.11511875100001 Sekunden

Text: buche ein restaurant für heute abend
{'AddToPlaylist': 1.7029149645964026e-08, 'BookRestaurant': 0.9999778270721436,
'GetWeather': 6.506423687824281e-06, 'PlayMusic': 1.4323073028186042e-14, 'RateB
ook': 2.100799889603877e-07, 'SearchCreativeWork': 1.1409509448334632e-10, 'Sear
chScreeningEvent': 1.5424508092110045e-05}

Text: füge alle musik von blink 182 meiner playlist hinzu
{'AddToPlaylist': 0.9999954700469971, 'BookRestaurant': 1.4475273424352508e-08,
'GetWeather': 1.0257154792138837e-14, 'PlayMusic': 1.7102250922107487e-06, 'RateB
ook': 2.860326503650867 4e-06, 'SearchCreativeWork': 3.8697521631547716e-08, 'Sea
rchScreeningEvent': 9.429575176245208e-13}
```

Bild 5.2 Trainingsfortschritt des Testklassifikators nach einer und nach zehn Iterationen und eine Klassifikation für zwei unbekannte Sätze auf Basis des fertig trainierten Modells. Die jeweils wahrscheinlichsten Klassen sind rot unterstrichen.

Zum Glück müssen wir auch diese Auswertung nicht selbstständig vornehmen, gibt es dafür doch die sogenannte *Confusion Matrix*, die diese vier Werte für jede Kombination aus tatsächlicher Klasse (Bild 5.3, *y*-Achse) eines Intents und vorhergesagter Klasse (Bild 5.3, *x*-Achse) eines Intents abbildet. In der allerersten Zeile sehen wir, dass 387 Datensätze der Kategorie *AddToPlaylist* richtig zugeordnet wurden. Der Wert rechts davon, die 3, sagt weiterhin aus, dass 3 Intents der Klasse *BookRestaurant* fälschlicherweise *AddToPlaylist* zugeordnet wurden, wohingegen kein *GetWeather*-Intent der Trainingsdaten der Klasse *AddToPlaylist* zugeordnet wurde usw. Es handelt sich hierbei um die *False Positives*, zu sehen in dem blauen Rahmen in Bild 5.3. Demgegenüber stehen die *False Negatives*, die durch den roten Rahmen markiert sind. 2 Intents der Kategorie *AddToPlaylist* sind fälschlicherweise PlayMusic zugeordnet worden.

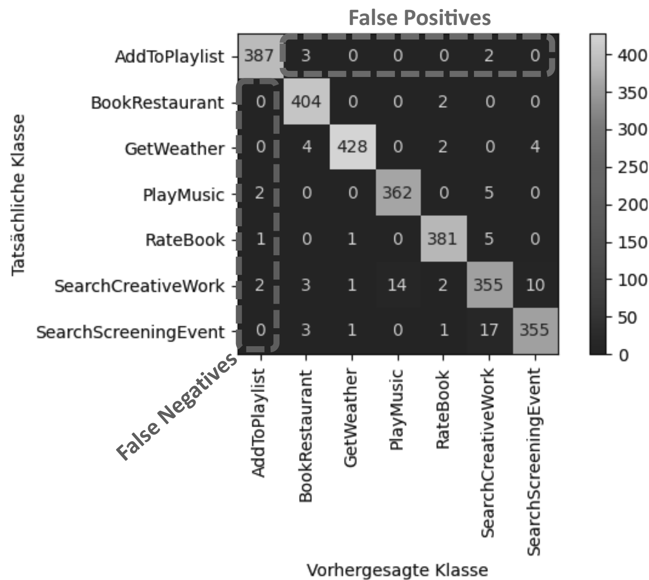


Bild 5.3 Confusion Matrix nach dem 10. Schritt des Trainings des Text Classifiers

HINWEIS: Um es mal mit den Worten eines Fernsehkochs zu sagen: Die *Confusion Matrix* aus Bild 5.3 können Sie mit nur wenigen Handgriffen und Zutaten ruck, zuck selber zaubern. Fügen Sie einfach das nachstehende Listing vor Zeile 11 von Listing 5.3 ein (Imports und `rcParams.update` gerne an den Kopf der Datei).

Listing 5.4 Textklassifikation – Plotten der Confusion Matrix

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams.update({'figure.autolayout': True})

cm = confusion_matrix(test_cats, preds, labels=labels)
ConfusionMatrixDisplay(cm, display_labels=labels).plot()
```

```
plt.xticks(rotation=90)
plt.xlabel(„Vorhergesagte Klasse")
plt.ylabel(„Tatsächliche Klasse")
plt.show()
```

Ich nutze die Möglichkeiten der Visualisierung sehr gerne, hilft die *Heatmap* doch schnell zu verstehen, wie gut ein Modell ist. Wenn Sie das Beispiel ausführen, müssen Sie das Fenster jeder *Confusion Matrix* schließen, bevor das Training weitergeht. Achten Sie dabei mal auf die positive Veränderung von der ersten Iteration bis hin zur zehnten! ■

Unser Classifier ist also recht treffsicher, denn wir haben in Summe kaum *False Positives* oder *False Negatives*. Die größte Anzahl Ausreißer finden wir dort, wo 17 Intents *SearchCreativeWork* zugeordnet wurden, obwohl sie zu *SearchScreeningEvent* gehören.

Nach unserem Ausflug in die Evaluierung unseres Modells können wir uns endlich anschauen, wie das Modell trainiert wird. Listing 5.5 zeigt die Methode `train()`, in der das erste Mal zu sehen ist, dass wir die Klassifikation auf SpaCy aufsetzen. Wir laden zuerst das vortrainierte Modell *de_core_news_sm* und entfernen alle Pipelines außer der für die Textklassifikation (siehe Zeile 4). Von Zeile 9 bis 15 fügen wir nun manuell alle Labels hinzu, die unsere Klassen repräsentieren.

Listing 5.5 Textklassifikation – Training des Modells

```
1.  def train(train_data, iterations, test_texts, test_cats, dropout = 0.3):
2.    nlp = spacy.load("de_core_news_sm")
3.
4.    textcat = nlp.create_pipe("textcat", config={"exclusive_classes": True,
5.      "architecture": "ensemble"})
6.    nlp.add_pipe(textcat, last=True)
7.
8.    # Festlegen der Intents als Labels für den Classifier
9.    textcat.add_label("AddToPlaylist")
10.   textcat.add_label("BookRestaurant")
11.   textcat.add_label("GetWeather")
12.   textcat.add_label("PlayMusic")
13.   textcat.add_label("RateBook")
14.   textcat.add_label("SearchCreativeWork")
15.   textcat.add_label("SearchScreeningEvent")
16.
17.   # Nur die Pipeline für den Text Classifier wird benötigt.
18.   pipe_exceptions = ["textcat", "trf_wordpiecer", "trf_tok2vec"]
19.   other_pipes = [pipe for pipe in nlp.pipe_names if pipe not in
20.     pipe_exceptions]
21.   with nlp.disable_pipes(*other_pipes):
22.     optimizer = nlp.begin_training()
23.
24.     print("Beginne Training ...")
25.     batch_sizes = compounding(16.0, 64.0, 1.5)
```

```
26.     for i in range(iterations):
27.       print('Iteration: '+str(i))
28.       start_time = time.perf_counter()
29.       losses = {}
30.       random.shuffle(train_data)
31.       batches = minibatch(train_data, size=batch_sizes)
32.       for batch in batches:
33.         texts, annotations = zip(*batch)
34.         nlp.update(texts, annotations, sgd=optimizer, drop=dropout,
35.           losses=losses)
36.       with textcat.model.use_params(optimizer.averages):
37.         evaluate(nlp.tokenizer, textcat, test_texts, test_cats)
38.
39.       print ("Iteration " + str(i) + ": " +str(time.perf_counter() -
40.         start_time)+  " Sekunden")
41.
42.     with nlp.use_params(optimizer.averages):
43.       modelName = "ensemble_model"
44.       filepath = os.path.join(os.getcwd(), modelName)
45.       nlp.to_disk(filepath)
46.   return nlp
```

Es folgt das Training ab Zeile 22. Wir legen zu Beginn die *batch_sizes* fest, die bestimmen, wie viele Daten in jeder Iteration in das Training eingespeist werden. Hier wird keine feste Größe verwendet, sondern der Generator *compounding*, der bei jedem Aufruf einen interpolierten Wert zwischen 16 und 64 zurückgibt. Die sequenzielle Erhöhung der Batch Size hat den Effekt, dass das Training massiv beschleunigt wird; wie immer gilt: Probieren Sie es gerne mal aus.

Nach dem Erstellen der tatsächlichen Batches in Zeile 31 führen wir ein Update des Modells aus Zeile 34 aus und evaluieren dieses im Nachgang in Zeile 37 über die Funktion `evaluate()`, die wir ja bereits kennengelernt haben.

Nun haben wir die Methoden für die Evaluation, das Datenladen und das Training beisammen. Im letzten Schritt müssen wir diese nur noch miteinander verknüpfen. Wir beginnen in Listing 5.6 damit, dass wir die Daten laden und die Spalten *sentences* und *intents* aus dem Trainingsdatensatz in separate Listen laden (siehe die Zeilen 3 bis 5). Nun erstellen wir ein etwas kryptisches Objekt, nämlich die *cat_list*, um die jeder Satz der Trainingsdaten ergänzt wird. Statt aber einfach jede Klasse mit Namen zu übergeben, benötigt *SpaCy* für das Training ein *Dictionary*, in dem jede Klasse namentlich berücksichtigt ist und wenn diese Klasse für den Datensatz relevant ist, wird sie mit einer 1 versehen. Wenn die Klasse nicht relevant ist, wird sie mit einer 0 versehen. Dieses Konstrukt erstellen wir von Zeile 7 bis 17. In Zeile 20 wird dann jeder Text der Trainingsdaten mit dem passenden Dictionary *cat_list* gezippt, also in ein *Tuple* gepackt. Die Liste all dieser Tuples machen letztendlich unsere Trainingsdaten aus.

Es folgt die vergleichsweise einfache Konstruktion der Testdaten, die lediglich darauf basiert, dass *test_texts* und *test_cats* aus dem Objekt *test_set* ausgelesen wird. *test_cats* enthält in diesem Fall die Namen der Klassen, nicht den *intent_index* wie in den Trainingsdaten! Das hatten wir ja in der Methode `load()` gesehen.

Listing 5.6 Textklassifikation – Aufruf von Datenladen, Training und Evaluation in der Hauptmethode

```
1.  if __name__ == '__main__':
2.
3.    train_set, test_set = load()
4.    train_texts = train_set['sentences']
5.    train_cats = train_set['intents']
6.
7.    final_train_cats=[]
8.    for cat in train_cats:
9.
10.     cat_list = {'AddToPlaylist': 1 if cat == 0 else 0,
11.       'BookRestaurant': 1 if cat == 1 else 0,
12.       'GetWeather': 1 if cat == 2 else 0,
13.       'PlayMusic': 1 if cat == 3 else 0,
14.       'RateBook': 1 if cat == 4 else 0,
15.       'SearchCreativeWork': 1 if cat == 5 else 0,
16.       'SearchScreeningEvent': 1 if cat == 6 else 0,
17.     }
18.     final_train_cats.append(cat_list)
19.
20.   training_data = list(zip(train_texts, [{"cats": cats} for cats in
21.     final_train_cats]))
22.
23.   test_texts = test_set['sentences']
24.   test_cats = test_set['intents']
25.
26.   nlp = train(training_data, 10, test_texts, test_cats, "ensemble")
27.
28.   test1 = "Buche ein Restaurant für heute Abend".lower()
29.   test2 = "Füge alle Musik von Blink 182 meiner Playlist hinzu".lower()
30.   nlp2 = spacy.load(os.path.join(os.getcwd(), "ensemble_model"))
31.   doc2 = nlp2(test1)
32.   print("Text: "+ test1)
33.   print(doc2.cats)
34.
35.   doc3 = nlp2(test2)
36.   print("Text: "+ test2)
37.   print(doc3.cats)
```

In Zeile 26 kommt es zum Aufruf des eigentlichen Trainings. Die Parameter *training_data*, *test_texts*, *test_cats* sollen uns nicht mehr überraschen. Die 10 gibt die 10 Iterationen an, die wir trainieren wollen. Das *ensemble* gibt Auskunft über die Modellarchitektur, Alternativen sind hier z. B. *bow*- oder *simple_cnn*-Architekturen, ersetzen Sie also *ensemble* mal durch diese Begriffe. *SpaCy* führt für diese Architekturen eine vollständige Liste in der Dokumentation[3].

Ist das Training abgeschlossen, sind wir bereit, Validierungssätze zu definieren, hier *test1* und *test2*. Das eben gespeicherte Modell wird in Zeile 30 geladen und auf die zwei Sätze angewandt. Indem wir *doc2.cats* ausgeben, erfahren wir die ermittelten Klassen. Wie schon in der *Confusion Matrix* in Bild 5.3 zu sehen, treffen wir die Kategorien sehr gut. Und das, obwohl wir unsere Test- und Trainingsdaten zuvor durch einen maschinellen Übersetzer erzeugt haben.

[3] *https://spacy.io/api/architectures*

5.1.2 Intent Parsing am Beispiel

Nun können wir also erkennen, ob ein Satz eine bestimmte Intention hat. Wie aber nehmen wir diesen nun so auseinander, dass wir die wichtigen Parameter darin identifizieren und extrahieren können? Zu wissen, dass der Computer einen speziellen Ort suchen soll, ist ja schön und gut, aber zu wissen was für einen Ort genau, ist obligatorisch. Und wenn wir mehrere Parameter haben, müssen wir diese ja auch noch einem Objekt zuordnen! Schauen wir uns mal ein Beispiel an.

Bild 5.4 zeigt einen sogenannten *Dependency Visualizer* auf Basis von *SpaCy*, der syntaktische Zusammenhänge zwischen einzelnen Begriffen hervorhebt. Die Labels der Klassen sind jedoch von uns festgelegt, oder richtig ausgedrückt: antrainiert. Sie sehen, dass wir so z. B. bei der Ortssuche identifizieren können, was gesucht werden soll: einen Urlaubsort. Dieser Urlaubsort wiederum muss laut Anfrage eine gewisse Eigenschaft aufweisen, nämlich sehr sonnig zu sein. Schauen wir uns mal an, wie die Trainingsdaten für so einen Fall beschaffen sein müssen.

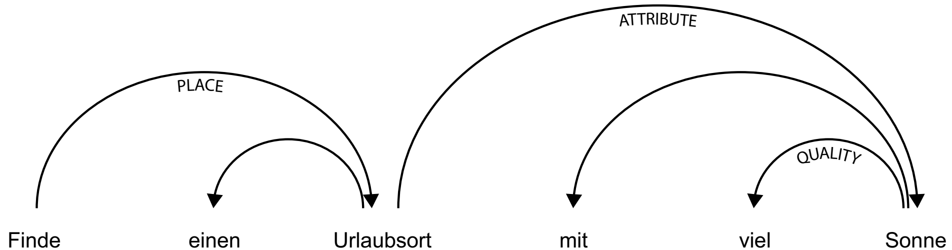


Bild 5.4 Dependency Visualizer mit eigenen Labels für einen Intent zur Ortssuche

Listing 5.7 Trainingsdaten für den Intent Parser

```
TRAIN_DATA = [
  (
    "Finde einen Urlaubsort mit viel Sonne",
    {
      "heads": [0, 2, 0, 5, 5, 2],
      "deps": ["ROOT", "-", "PLACE", "-", "QUALITY", "ATTRIBUTE"],
    }
  ),
  ...
]
```

Listing 5.7 zeigt einen exemplarischen Datensatz aus dem Beispiel *100_extras/100_05_ intentparser*, das wir uns nun anschauen wollen, um damit zu experimentieren. Der Datensatz weist zu Beginn einen einfachen Satz ohne Satzzeichen auf. Wir trainieren das Modell ohne Satzzeichen, da es uns um den Kontext geht, in dem die Wörter stehen. Es macht kaum einen Unterschied, wenn ein Ausrufezeichen oder ein Punkt am Satzende steht. Ob es sich um eine Frage handelt, erkennt das Modell in der Regel am Satzbau.

TIPP: Wie ist es denn mit Groß- und Kleinschreibung in Trainingsdaten? Im Englischen werden Sätze häufig komplett kleingeschrieben, da es dort kaum zum Kontext beiträgt, ob ein Satz Großbuchstaben enthält, denn es werden nur Satzanfänge und Eigennamen großgeschrieben. Im Deutschen ist es etwas anders. Ein Beispiel:

Der Junge sieht dir Ungeheuer ähnlich.

Der Junge sieht dir ungeheuer ähnlich.

Was uns vielleicht zum Schmunzeln bringt, kann zu falschen Assoziationen in unserem Modell führen und unseren Sprachassistenten entsprechend beeinflussen. Wir passen als Konsequenz die Groß- und Kleinschreibung im Deutschen nicht an. ■

Jedem Satz ist eine numerische Liste mit dem Namen *heads* zugeordnet. Jedes Token hat ein *Head-Element*, das von *SpaCy* als syntaktisches Elternelement bezeichnet wird und über dessen Index in *heads* referenziert wird. Diese Zahl darin referenziert ein anderes Wort, auf das sich das ursprüngliche Wort bezieht. Die sich aus Listing 5.7 ergebenden Abhängigkeiten sind in Tabelle 5.2 dargestellt. Das Elternelement von *Urlaubsort* ist beispielsweise *Finde*, das von *viel* ist *Sonne*. Wenn Sie in naher Zukunft selber Testdaten erstellen, werden Sie sich überlegen, welche Regeln Sie dafür anwenden können. Tatsächlich ist der Gedanke des hierarchisch übergeordneten Wortes für mich immer am passendsten gewesen.

Tabelle 5.2 Tabellarische Darstellung eines Trainingsdatensatzes für das Intent Parsing

Wort	Head	Label
Finde	Finde	ROOT
einen	Urlaubsort	–
Urlaubsort	Finde	PLACE
mit	Sonne	–
viel	Sonne	QUALITY
Sonne	Urlaubsort	ATTRIBUTE

Wenn Sie jedoch viele hundert Datensätze taggen müssen, empfehle ich Ihnen, auf SpaCy zurückzugreifen, denn über folgendes Listing bekommen Sie die *heads* anhand eines vortrainierten Modells relativ einfach ausgegeben.

```
doc = nlp("Finde einen Urlaubsort mit viel Sonne")
for token in doc:
  print(token.text, token.head.text)
```

Wichtig ist noch, dass zirkuläre Referenzen verboten sind, weswegen beispielsweise *Finde* → *Urlaubsort* und gleichzeitig *Urlaubsort* → *Finde* eine unzulässige Zuweisung in den *Head-Elementen* ist.

Beschäftigen wir uns nun damit, dass wir die Beziehungen, die wir zwischen Wort und Head gebildet haben, beschreiben oder besser: mit Labels versehen. Die Spalte *Label* in Tabelle 5.2 bildet dieses Labeling ab und kennzeichnet *Finde* als das *Wurzelelement*, *Urlaubsort* als einen *Ort*, *viel* als qualitatives Merkmal und *Sonne* als *Attribut*.

Nun, da wir wissen, wie wir unsere Trainingsdaten für das Intent Parsing vorbereiten müssen, können wir das Training starten. Wir erzeugen zunächst eine leere Pipeline für die deutsche Sprache in Zeile 10 und fügen einen Parser hinzu. Dann holen wir aus jedem Eintrag aus *TRAINING_DATA* (hier nicht abgebildet) den Text und die Annotationen und fügen die Labels dem Parser hinzu (Zeile 15), sodass dieser weiß, welche Worte wie gelabelt werden sollen.

Listing 5.8 Training und Evaluierung des Intent Parsings

```
1.  import random
2.  from pathlib import Path
3.  import spacy
4.  from spacy.util import minibatch, compounding
5.
6.  if __name__ == "__main__":
7.    iterations=15
8.    nlp = spacy.blank("de")
9.
10.   parser = nlp.create_pipe("parser")
11.   nlp.add_pipe(parser, first=True)
12.
13.   for text, annotations in TRAIN_DATA:
14.     for dep in annotations.get("deps", []):
15.       parser.add_label(dep)
16.
17.   pipe_exceptions = ["parser", "trf_wordpiecer", "trf_tok2vec"]
18.   other_pipes = [pipe for pipe in nlp.pipe_names if pipe not in
19.     pipe_exceptions]
20.   with nlp.disable_pipes(*other_pipes):
21.     optimizer = nlp.begin_training()
22.     for itn in range(iterations):
23.       random.shuffle(TRAIN_DATA)
24.       losses = {}
25.       batches = minibatch(TRAIN_DATA, size=compounding(4.0, 32.0, 1.001))
26.       for batch in batches:
27.         texts, annotations = zip(*batch)
28.         nlp.update(texts, annotations, sgd=optimizer, losses=losses)
29.       print("Losses", losses)
30.
31.   texts = [
32.     "finde ein hotel mit gutem internet",
33.     "suche das günstiges fitnessstudio nahe der arbeit",
34.     "suche das beste restaurant in berlin",
35.   ]
36.
37.   docs = nlp.pipe(texts)
38.   for doc in docs:
39.     print(doc.text)
40.     print([(t.text, t.dep_, t.head.text) for t in doc if t.dep_ != "-"])
```

Wie auch beim Intent Classifier schließen wir sämtliche Schritte aus der Pipeline aus, die wir nicht benötigen (Zeilen 17–20) und beginnen das Training für 15 Iterationen. Das Beispiel ist sehr ähnlich zu dem vorigen, wie Sie sehen werden. Nach dem letzten Schritt wenden wir den Parser auf drei unbekannte Sätze an und schauen, wie diese mit Labels versehen werden (siehe Bild 5.5).

```
Anaconda Prompt
Losses {'parser': 23.245105244219303}
Losses {'parser': 16.897597763687372}
Losses {'parser': 28.236346662044525}
Losses {'parser': 17.39258297532797}
Losses {'parser': 12.862866576761007}
Losses {'parser': 11.736745685338974}
Losses {'parser': 16.71676853299141}
Losses {'parser': 19.161401491612196}
Losses {'parser': 19.698628412326798}
Losses {'parser': 18.332283061856288}
Losses {'parser': 17.236523714902432}
Losses {'parser': 14.35519179056746}
Losses {'parser': 10.194273205335776}
Losses {'parser': 8.32290686437969}
Losses {'parser': 6.307664709598612}
finde ein hotel mit gutem internet
[('finde', 'ROOT', 'finde'), ('hotel', 'PLACE', 'finde'), ('gutem', 'ATTRIBUTE', 'internet'), ('internet', 'ATTRIBUTE',
'hotel')]
suche das günstiges fitnessstudio nahe der arbeit
[('suche', 'ROOT', 'suche'), ('günstiges', 'PLACE', 'suche'), ('fitnessstudio', 'QUALITY', 'arbeit'), ('nahe', 'QUALITY'
, 'arbeit'), ('arbeit', 'ATTRIBUTE', 'günstiges')]
suche das beste restaurant in berlin
[('suche', 'ROOT', 'suche'), ('beste', 'PLACE', 'suche'), ('restaurant', 'QUALITY', 'berlin'), ('berlin', 'ATTRIBUTE', '
beste')]
```

Bild 5.5 Das Loss reduziert sich mit jeder Trainingsiteration, sodass nach der fünfzehnten Trainingsiteration Parameter und Beziehungen innerhalb der drei Testsätze richtig erkannt und zugeordnet werden.

Mit `nlp.to_disk(output_dir)` kann das Modell übrigens auch wie im Beispiel in Abschnitt 5.1.1 gespeichert werden. Nun haben wir gesehen, wie wir Intents klassifizieren und wie wir sie dann auf mögliche Parameter untersuchen. Sollte also die Aufgabe sein, eine Suche nach einem Ort zu implementieren, können wir nun herausfinden, ob der Benutzer eine solche Ortssuche aufrufen soll und nach welchen Orten mit welchen Eigenschaften wir etwa in *Google Maps* nachschlagen müssen.

Da wir nun der Aufgabe gewachsen wären, genau das umzusetzen, möchte ich Sie erneut nüchtern damit konfrontieren, dass es Frameworks gibt, die das bereits wesentlich besser machen als wir und auf die wir aufbauen werden. Dennoch war die Arbeit nicht umsonst, haben wir doch nun ein grundlegendes Verständnis von den Anforderungen, die wir an ein solches Framework stellen müssen. Die Auswahl möglicher Kandidaten ist Teil des folgenden Abschnitts.

5.2 Auswahl eines Chatbot-Frameworks

Das Erste, was Ihnen sicher bei dem Titel auffallen wird, ist, dass dort Chatbot steht. Dieser Begriff erweckt bei manchen sicher erst mal Assoziationen zu Anwendungen wie IKEAs *LATTJO* oder anderen, eher unpersönlichen Dialog Engines diverser Websites, die darauf programmiert sind, bei Fragen weiterzuhelfen, wie etwa über Preise Auskunft zu geben oder zu berichten, wo im Shop man Einbauschränke findet. Die Herausforderung, wie wir ja bereits in Abschnitt 5.1.1 und Abschnitt 5.1.2 gesehen haben, ist, auf unvorhergesehene Sätze richtig zu reagieren. Nicht gemeint ist irgendeine Art von Visualisierungswerkzeug, dass es uns erlaubt, Texte in eine dafür vorgesehene Box einzugeben und Dialoge anzuzeigen. Da wir einen Sprachassistenten entwickeln, haben wir dafür sowieso eher wenig Verwendung.

Wir werden nun die Auswahl zweier Frameworks treffen. Das erste soll uns helfen, relativ statischen, wenig dynamischen Text gut zu erkennen und auf Basis von *Regular Expressions* dessen Intention zu klassifizieren und zu analysieren. Ein Beispiel dafür wäre etwa die Frage nach der Uhrzeit. Das zweite Framework hingegen wird auf maschinellem Lernen basieren und anhand von Trainingsdaten entscheiden, welche Intention der Benutzer hat und was mögliche Parameter sind. Hier wäre etwa die Ortssuche zu nennen, die wir im vorigen Abschnitt exemplarisch behandelt haben.

Nun gibt es viele verschiedene Bibliotheken, die diese Arbeit sehr gut machen. Die bekanntesten habe ich Ihnen in Tabelle 5.3 zusammengestellt. Hierbei war eine der Anforderungen, dass diese Frameworks offline funktionieren müssen, also keine Daten herausgeben dürfen, die irgendwo anders verarbeitet werden. Damit fallen zwar eine Hand voll vielversprechender Kandidaten raus, wie etwa *Dialogflow* von Google, aber wir bleiben dafür unseren Prinzipien treu und werden – so viel sei schon mal gesagt – dennoch sehr gute Ergebnisse erzielen. Weitere Anforderungen waren, dass eines der beiden zu nutzenden Frameworks anhand von Trainingsdaten die Klassifikation und das Intent Parsing erlernen kann – und dass es die deutsche Sprache unterstützt.

Rasa NLU ist die erste Bibliothek, die sich bei der Recherche finden lässt, ist es doch eines der größten Vertreter am Markt und wird Open Source und kommerziell vertrieben. Hinter Rasa steht ein großes Team von Experten, die Open-Source-Version hat beinahe 13.000 Sterne auf *Github*. Es gibt hunderte Anleitungen, hunderte Videos, die dessen Funktion erklären – aber das ist leider auch nötig, denn das Framework ist so komplex, dass es mittlerweile schon ein eigenes Buch dazu gibt (Kong, Wang, & Nichol, 2021).

Tabelle 5.3 Vergleich bekannter Chatbot-Bibliotheken in Python

Name	Machine Learning	Intent-Behandlung	Sprachen
Rasa NLU	Ja	Ja	Mehrere
ChatterBot	Ja	Nein	Mehrere
ChatbotAI	Nein	Ja	Deutsch, Englisch
Snips-NLU	Ja	Ja	Mehrere
Transformers	Ja	Nein	Mehrere
DeepPavlov	Ja	Ja	Mehrere

ChatterBot ist eine mittlerweile auch sehr weitverbreitete, offene Dialog Engine, die auf Basis vergangener Konversationen angelernt wird und sich mit dem daraus resultierenden Modell mit Benutzern unterhalten kann. Neue Antworten von Gesprächspartnern fließen direkt wieder ins nächste Training ein, sodass der Bot automatisch dazulernt. *ChatterBot* ist ebenfalls mit 11.500 Sternen auf *Github* überaus beliebt. Es kann allerdings noch keine Intents behandeln, was uns lediglich erlauben würde, mit dem Benutzer unseres Sprachassistenten nette Gespräche zu führen, nicht aber zu verstehen, was er oder sie möchte. In diesem Fall fehlt also konkret die NLU-Komponente.

ChatbotAI ist ein eher unauffälligeres Framework mit rund 300 Sternen. Da es einen Ansatz verfolgt, der auf Regular Expressions und nicht auf Machine Learning basiert, hat es im Moment auch nicht die Sichtbarkeit, die andere Bibliotheken haben. Dazu kommt, dass es neben der Sprache Englisch nur noch Hebräisch, Deutsch und brasilianisches Portugiesisch unterstützt. Allerdings sind beliebige Sprachen sehr einfach hinzuzufügen (ich hatte damals die Unterstützung für Deutsch beigesteuert), sodass die Funktionalität in der Hinsicht erweiterbar ist. Der Vorteil dieses Frameworks ist, dass es sehr einfach zu verstehen, leichtgewichtig und dennoch sehr mächtig ist.

Snips-NLU ist eine Bibliothek, die alle unsere Anforderungen erfüllt und dazu noch sehr leicht zu verstehen und zu implementieren ist. Allerdings ist *Snips.co* im November 2019 von *Sonos* aufgekauft worden, wahrscheinlich um dort eine Art Smart Speaker zu entwickeln. Als Konsequenz erfährt *Snips-NLU* keine Weiterentwicklung mehr, doch obwohl es lediglich in Version 0.20.2 veröffentlicht und auf Open Source umgestellt wurde, erfüllt es alle Anforderungen an eine NLU-Bibliothek, die wir für einen Sprachassistenten benötigen würden.

Transformers von *Hugging Face* haben wir ja bereits kennengelernt, z. B. als wir unsere Trainingsdaten in den vorherigen Abschnitten maschinell übersetzt haben. Da sich deren schlaue Leute in Paris und New York nicht nur mit Übersetzungen beschäftigen, sondern sich im Großen und Ganzen NLP und NLU widmen, existieren dort auch Modelle und Module zur Bereitstellung eines Chatbots. Jedoch ist das Framework nicht auf das Führen von Dialogen ausgelegt, das müssten wir umständlich nachimplementieren. Zwar gibt es fertige Modelle wie etwa DialoGPT von Microsoft, das einen ähnlichen Aufbau wie GPT-2 hat, ebenfalls autoregressiv ist, jedoch auf Dialogen von Reddit trainiert ist. Allerdings ist das Modell weder auf das Parsen von Intents trainiert, noch auf das Identifizieren deren Parameter.

Zuletzt sei noch *DeepPavlov* genannt, das auch eine beeindruckende Funktionalität hat und mit etwa 5000 Sternen auf *Github* eine gewisse Beliebtheit aufweist. Allerdings basiert es auf Deep Learning (*Tensorflow* oder *PyTorch*). Das macht es uns etwas schwer, es auf kleinen skalierten Geräten, etwa einem Raspberry Pi, zu betreiben, braucht die Anwendung der Modelle doch in der Regel eine GPU mit entsprechend Speicher, um den Benutzer nicht zu lange auf eine Antwort warten zu lassen.

HINWEIS: Nach dieser sicherlich nicht erschöpfenden Liste stellt sich natürlich noch die Frage, warum wir nicht zu *NLTK* oder *SpaCy* greifen, schließlich sind das ja die Werkzeuge überhaupt, wenn es um die Analyse von Texten geht. Nun ja, die Stärken dieser beiden Bibliotheken liegen tatsächlich in der Analyse, etwa in der Aufteilung in einzelne Tokens, dem POS-Tagging oder der Entitätsextraktion. Das heißt, beide bieten zwar die Funktionen, die vielen *Dialog Engines* zugrunde liegen, jedoch haben sie keine *high-level*-Features für die eigentliche Mensch-Maschine-Interaktion. ■

Die Auswahl, die ich nach einigen Tests getroffen habe, ist auf *ChatbotAI* und *Snips-NLU* gefallen. Erstens aufgrund der genauen Abbildung der Funktionen, die wir für unseren Sprachassistenten benötigen. Wir möchten keine langen, möglichst glaubhaften Gespräche führen, sondern Befehle klassifizieren und verarbeiten können. Beide Bibliotheken sind nicht mit Funktionen überladen, die wir überhaupt nicht benötigen, und lassen sich sehr gut auf verschiedene Sprachen und Usecases anpassen. Ein weiterer Grund war für mich auch die Möglichkeit, anhand dieser Frameworks zu unterrichten. Wir werden mit *ChatbotAI* einen Durchstich, wie man so schön sagt, machen. So, dass wir schnell die ersten Funktionen auf die Straße bringen und wir erste Interaktionen per Sprache zulassen. Für anspruchsvollere Intents, die zuvor nicht von *ChatbotAI* erkannt wurden, werden wir dann *Snips-NLU* einsetzen. Bild 5.6 verdeutlicht den Prozess noch einmal und zeigt auch gleich, wie wir einzelne Interaktionen in *ChatbotAI* (links) und mit *Snips-NLU* (rechts) definieren können; dazu aber gleich mehr.

- **Primärer Prozess** (*ChatbotAI*), um
  - fest vorgegebene Eingaben zu erkennen,
  - Intentionen zu verstehen und Aktionen aufzurufen.

```
{% block %}
    {% client %}(wer ist|was ist|kennst du) (?P<query>.*){% endclient %}
    {% response %}{% call whatIs: %query %}{% endresponse %}
{% endblock %}
```

Wurde der Befehl nicht erkannt …

- **Sekundärer Prozess** (*Snips-NLU*), um
  - unbekannte Sätze über ein vortrainiertes Modell zu erkennen,
  - flexibel Intentionen zu verstehen, Schlüsselwörter zu extrahieren und Aktionen aufzurufen.

```
{
    'input': 'wie spät ist es in amerika',
    'intent': {
        'intentName': 'getTime',
        'probability': 0.7193185567015047
    },
    'slots': [{
            'rawValue': 'amerika',
            'value': {
                'kind': 'Custom',
                'value': 'amerika'
            },
            'entity': 'country',
            'slotName': 'country'
        }
    ]
}
```

Bild 5.6 Zwei sequenzielle Frameworks ermöglichen eine statische und eine dynamische Erkennung von Befehlen.

Eine Analogie, um zu erklären, warum wir hier zweigleisig fahren, ist, eine Klausur in der Schule oder im Studium zu schreiben. Wenn Sie die Frage bereits kennen, müssen Sie sie nicht großartig interpretieren, sondern die Antwort lediglich abspulen. Diese ist dann meistens zu einhundert Prozent korrekt. Wenn die Frage aber neu und unbekannt ist, muss sie erst interpretiert werden und Sie müssen sich Gedanken machen, wie die konkrete Fragestellung überhaupt lautet, gegebenenfalls an den Parametern der Frage oder anhand ähnlicher Fragen, die Sie bereits anders formuliert gehört haben. Hier kann es sein, dass Sie nicht vollständig richtigliegen, wenn Sie nicht genug gelernt haben oder schlechtes Lernmaterial zur Hand hatten. Aber Sie sind dennoch besser, als wenn Sie eine gelernte Antwort geben, die überhaupt nicht richtig ist.

Sehen Sie den Vergleich der beiden Methoden auch gerne als Experiment, mit dem Sie sich für Ihren ganz individuellen Anwendungsfall ein Bild machen können, ob eine regelbasierte

oder eine Implementierung auf Basis von Machine Learning bessere Ergebnisse erzielt. Im realen Leben würden Sie sich wahrscheinlich irgendwann für eines der beiden entscheiden, denn mehrere Frameworks zu betreiben, bedeutet auch mehr Wartungsaufwände z. B. für Updates oder Betrieb und womöglich auch mehr Lizenzkosten.

In Bild 5.7 sehen Sie eine Auflistung der Fragestellungen zu den beiden unterschiedlichen Ansätzen nach den Themen Verwendungszweck, Implementierungsaufwand, Güte und Performance.

Für welche Zwecke sind starre und für welche flexible Parser hilfreich?	Wie kompliziert sind die jeweiligen Vorgehen umzusetzen?	Wie gut sind beide Ansätze zu warten?	Wie ist deren Performance?
• Ist ein eindeutiger Befehl zu erwarten? • Werden längere Konversationen geführt? • Kommt es zu Smalltalk?	• Wie genau muss die Konversation vorbestimmt werden? • Wie schwer ist es, eine weitere Sprache hinzuzufügen?	• Wie können Trainingsdaten oder Dialoge erweitert werden? • Wie prüfe ich die Güte meines Chatbots?	• Wie lange dauert es, eine Antwort zu generieren? • Wie lange dauert es, den Dialog vorzubereiten?

Bild 5.7 Fragestellung zu dem Experiment starres vs. dynamisches Chat-Framework

Die erste Fragestellung zielt auf den Verwendungszweck des Frameworks in Ihrer Anwendung ab. Gibt es eine klar formulierbare Aufgabe, womöglich eine Art Befehl, der sich in ein bis drei Wörter fassen lässt? Denkbare Beispiele wären *Start/Stopp* für einen Timer, *Uhrzeit* oder *Licht aus/an*. Demgegenüber stehen längere Konversationen oder gar Smalltalk, der so variabel ist, dass er sich überhaupt nicht vorhersagen lässt.

Hinsichtlich der Komplexität der Umsetzung wird im zweiten Block von Bild 5.7 hinterfragt, wie genau Sie eine Konversation vorbestimmen müssen und wie komplex der Prozess der Entwicklung initial ist und wie sich auch eine Mehrsprachigkeit abbilden lässt. Reicht es etwa, eine Hand voll Sätze zu formulieren, die dem Framework genügen, um den Dialog eindeutig zu klassifizieren und zu analysieren? Oder müssen Sie, wie im Beispiel in Abschnitt 5.1.1, mehrere tausend Trainingsdatensätze formulieren?

Die dritte Frage beschäftigt sich mit der Wartung Ihrer Anwendung. Ist sie nachträglich erweiterbar? Oder sind die Regeln so starr auszuformulieren, dass jede Erweiterung hunderte von Entwicklerstunden verschlingt? Und wie stelle ich fest, dass Intentionen richtig erkannt werden und sich nicht mit anderen überschneiden? Sie sollten sich ebenso fragen, ob Ihr Sprachassistent auch nach dem ersten Release noch kontinuierlich weiterentwickelt wird, oder ob er nur alle paar Monate oder Jahre in einer neuen Version veröffentlicht wird.

Zu guter Letzt stellt sich natürlich auch noch die Frage nach der Performance. Wie schnell liefert ein Assistent eine Antwort? Wie lange dauert es, die Eingabe zu verarbeiten, die Eingabe zu verstehen und die Ausgabe zu generieren? Der Mensch hat in der Regel recht wenig Geduld. Fragen Sie sich mal, wie lange Sie bereit sind zu warten, bis eine Website lädt. Oder wie anstrengend ein Gespräch mit einem Menschen sein könnte, der mehrere Sekunden benötigt, um zu antworten.

5.2.1 Intents auf Basis regulärer Ausdrücke

Machen wir uns endlich ans Eingemachte, fehlt einem ersten Assistenten doch nur noch die Fähigkeit, die Benutzereingabe zu verarbeiten und passend zu reagieren. Legen Sie dazu ein neues Beispiel mit dem Namen *07_dialoge_und_intents* an und fügen Sie den benötigten Ordner und das Anaconda Environment hinzu. Die Abhängigkeiten, die Sie über *pip* vorinstallieren sollten, können Sie wie immer dem letzten Kapitel entnehmen.

Wir beginnen ganz seicht, indem wir zwei Intents für das Framework *ChatbotAI* in der *main.py* definieren. Intents werden hier als Calls bezeichnet und müssen mit der Annotation `@register_call` versehen werden, die als Parameter den Namen des Calls mitgegeben bekommt, in Listing 5.9 *time* und *stop*. Eine solche Annotation weist darauf hin, dass eine Methode folgt. Schauen wir uns zuerst `getTime()` an. Diese Methode ruft in Zeile 6 zuerst das aktuelle Datum inklusive der Uhrzeit ab und konstruiert in den Zeilen 7-8 einen String, um die Uhrzeit in Worte zu fassen. Ein paar Worte zu den Parametern: *session_id* kann übergeben werden, um mehrere verschiedene Sitzungen zu handhaben, falls z. B. mehrere Gesprächspartner mit dem Chatbot interagieren. In unserem Fall werden wir aber immer die Session *general* verwenden, da wir Benutzer und komplexere Dialoge selber verwalten. Die Variable *dummy* mag Ihnen etwas merkwürdig vorkommen. Zu Recht, hat sie doch keine weitere Funktion in der Logik des Intents. Jedoch ist Vorgabe von ChatbotAI, dass ein Call immer einen Parameter aufweisen muss. Und diesem muss dazu noch ein Wert zugewiesen sein, denn Python schreibt vor, dass einem Parameter mit einem Default-Wert (in unserem Fall *session_id*) kein Parameter folgen darf, der keinen Default-Wert besitzt. Demnach weisen wir *dummy* den Wert 0 zu, wir referenzieren aber weder die Variable noch deren Inhalt.

Listing 5.9 Definition zweier Calls für eine Zeitabfrage und zum Unterbrechen des Assistenten

```
1.  from chatbot import Chat, register_call
2.  from datetime import datetime
3.
4.  @register_call("time")
5.  def getTime(session_id = "general", dummy=0):
6.    now = datetime.now()
7.    return "Es ist " + str(now.hour) + " Uhr und " + str(now.minute) + "
8.      Minuten."
9.
10. @register_call("stop")
11. def stop(session_id = "general", dummy=0):
12.   if va.tts.is_busy():
13.     va.tts.stop()
14.     return "okay ich bin still"
15.   return "Ich sage doch gar nichts"
```

Die Methode `stop()` wird ebenfalls mit `@register_call` annotiert und erhält *session_id* und *dummy* als Parameter. Sie dient dazu, den Sprachassistenten jeder Zeit zu unterbrechen, wenn wir *stop* sagen. Zu sehen ist in den Zeilen 12 und 13, dass geprüft wird, ob derzeit ein Text gesprochen wird. Die Methode `is_busy()` müssen wir allerdings in *TTS.py* noch nachtragen. Definieren Sie diese innerhalb der Klasse `Voice` wie in Listing 5.10 zu sehen.

Listing 5.10 Prüfen, ob derzeit gesprochen wird oder nicht

```
def is_busy(self):
  if self.process:
    return self.process.is_alive()
```

Die Logik ist simpel: Ist der Prozess noch aktiv, wird *True* zurückgeliefert, andernfalls *False*. Sie können *TTS.py* nun wieder schließen und zur *main.py* zurückkehren.

Abhängig davon, ob der Sprachassistent tatsächlich gesprochen hat oder nicht, geben wir eine entsprechende Rückmeldung in Form einer Zeichenkette, die den Befehl aus Sicht des Assistenten kommentiert. Zur Erinnerung: *va* definieren wir global bei der Initialisierung der Klasse *VoiceAssistent* am Ende der Datei.

Um *ChatbotAI* verwenden zu können, sollten Sie die Bibliothek über `pip install ChatbotAI` in Ihr Environment installieren. Kümmern wir uns nun darum, dass das Framework weiß, wann es mit welchem Call auf die Benutzereingaben reagieren soll. *ChatbotAI* organisiert sich in sogenannten Templates. Legen Sie im Projektordner einen Ordner *dialogs* und darin eine Datei mit dem Namen *dialogs.template* an. Dessen Inhalt sollte dem aus Listing 5.11 entsprechen.

Listing 5.11 Template für die Intents stop und time

```
1.  {% block %}
2.      {% client %}(wie spät ist es|wie viel uhr ist es){% endclient %}
3.      {% response %}{% call time: 0 %}{% endresponse %}
4.  {% endblock %}
5.
6.  {% block %}
7.      {% client %}(stop|halt|stopp|schweigefuchs){% endclient %}
8.      {% response %}{% call stop: 0 %}{% endresponse %}
9.  {% endblock %}
```

Ein Dialog wird in einem sogenannten *Block* organisiert. Man erkennt schnell, dass der erste der Wiedergabe der Uhrzeit und der zweite dem Unterbrechen des Assistenten dient. Der mit *client* markierte Text kennzeichnet die Eingabe eines Benutzers in Form einer *Regular Expression*. Der Intent wird also aufgerufen, wenn der Benutzer entweder *wie spät ist es* oder *wie viel uhr ist es* sagt.

Die Response kann entweder ein Text sein, der sofort zurückgeliefert wird, oder aber auch ein Funktionsaufruf, durch den die Funktion selber einen Rückgabewert generiert. In unserem Fall nutzen wir Letzteres und rufen im ersten Block die mit *time* annotierte Methode und im zweiten Block die mit *stop* annotierte Methode auf. Einem Call können wir verschiedene Parameter mitgeben. Einer ist, wie eben schon angesprochen, obligatorisch, weswegen wir hier jeweils eine 0 für den Parameter *dummy* übergeben.

Lassen Sie uns nun zurück in die *main.py* gehen und dort am Ende der Methode `__init__()` das Dialog-Framework initialisieren.

Listing 5.12 Initialisierung von ChatbotAI

```
1.  logger.info("Initialisiere Chatbot...")
2.  dialog_template_path = './dialogs/dialogs.template'
3.  if os.path.isfile(dialog_template_path):
4.    self.chat = Chat(dialog_template_path)
5.  else:
6.    logger.error('Dialogdatei konnte nicht in {} gefunden werden.',
        dialog_template_path)
7.    sys.exit(1)
8.  logger.info('Chatbot aus {} initialisiert.', dialog_template_path)
```

Wir legen zuerst den Pfad des Templates fest, das wir soeben formuliert haben, und prüfen, ob die Datei existiert oder nicht. Da wir ja später lediglich über Sprache arbeiten und unsere Benutzer nicht sehen können, ob das Log irgendeine Fehlermeldung ausgibt, müssen wir in vielen Situationen besonders genau auf Fehler prüfen, diese in manchen Fällen akustisch ausgeben und zumindest sauber loggen. Wird das Template nicht gefunden, beenden wir den Assistenten, denn wir haben keine Möglichkeit, dessen Funktion auf irgendeine Art und Weise wiederherzustellen. `sys.exit()` beendet die Applikation und weist mit dem Parameter 1 darauf hin, dass die Anwendung mit einem Fehler-Code geschlossen wurde – 0 hingegen kennzeichnet eine fehlerfreie Beendigung.

Damit haben wir schon alles beisammen, um die letzten zwei Zeilen in die While-Schleife unserer Methode `run()` zu integrieren. Konkret geht es nur um die Zeilen 4–6 aus Listing 5.13, in denen wir `self.chat.respond()` aufrufen und den verstandenen Satz des Benutzers als Parameter übergeben.

Listing 5.13 Generieren einer Antwort über ChatbotAI

```
1.  sentence = recResult['text']
2.  logger.debug('Ich habe verstanden "{}"', sentence)
3.
4.  output = self.chat.respond(sentence)
5.  logger.debug('Output ist "{}", output)
6.  self.tts.say(output)
7.
8.  self.is_listening = False
9.  self.current_speaker = None
```

Das Resultat *output* ist eine Zeichenkette, die wir über `tts.say()` von unserer Text-To-Speech-Engine sprechen lassen können. Probieren Sie den Assistenten gerne mal ausführlich aus, nachdem Sie *chatbot* über *pip* installiert haben. Dabei geht es vor allem darum, ein Gefühl dafür zu bekommen, wie *VOSK* die Befehle interpretiert, die die Benutzer geben. Versuchen Sie es zunächst mit diesen Sätzen:

- *Wie spät ist es?*
- *Stopp!*

Diese entsprechen nun eins zu eins den Templates aus Listing 5.11 und sollten vom Framework mühelos erkannt werden. Entspricht der gesprochene Befehl einem der Muster in der

dialogs.template, wird die Funktion aufgerufen, die per *call* referenziert wird. Versuchen Sie es nun mit *Uhrzeit!*, werden Sie keinen Erfolg haben. Natürlich, passt der Befehl doch auf keinen regulären Ausdruck im Template.

Die ersten zwei Intents zu programmieren, ging also recht leicht von der Hand. Jedoch haben wir ein paar wesentliche Punkte noch nicht bedacht: Was passiert, wenn ein Befehl nicht korrekt interpretiert werden kann? Im Moment bekommen wir in diesem Fall noch eine englischsprachige Fehlermeldung von `chat.respond()` zurückgeliefert. Weiterhin organisieren wir noch unsere beiden Intents in einem Template. Wie aber sieht es aus, wenn Benutzer mehrere Intents nachträglich hinzufügen wollen? Damit beschäftigen wir uns gleich, lassen Sie uns zunächst aber mal schauen, wie wir unser zweites Framework auf Basis maschinellen Lernens in die Anwendung integrieren.

5.2.2 Intents auf Basis maschinellen Lernens

In diesem Abschnitt werden Sie lernen, die Bibliothek *Snips-NLU* einzusetzen, um auf Befehle eines Nutzers zu reagieren, die auf andere Art und Weise formuliert wurden, als wir während der Implementierung vorgesehen hatten. Diese Fähigkeit bekommen wir allerdings nicht geschenkt, denn wir müssen, wie schon in den Machine-Learning-Beispielen, zuvor Trainingsdaten erstellen – das wiederum allerdings nicht in übermäßig großem Umfang, wie Sie gleich sehen werden.

Sie finden das aktuelle Beispiel im Ordner *08_dialogs_and_intents_ml*. Richten Sie sich einen Projektordner und ein *Conda Environment* ein und kopieren Sie den Code des gesamten letzten Kapitels in diesen neuen Ordner.

HINWEIS: Snips-NLU baut auf der Sprache RUST auf und erwartet, dass diese auf dem ausführenden System installiert ist. Sie finden die Installationsdatei hier: *https://www.rust-lang.org*

Dann löschen Sie zuerst den Ordner *dialogs*, in dem wir unser Template für *ChatbotAI* abgelegt haben; dieses verwenden wir in diesem Beispiel nicht. Erstellen Sie stattdessen einen Ordner mit dem Namen *dialog-datasets* und darin die Datei *stop_dataset.yaml*. Die Endung deutet darauf hin, dass wir einen alten Bekannten, nämlich das YAML-Format wiedersehen und damit arbeiten werden. Befüllen Sie nun die Datei mit dem Inhalt von Listing 5.14. Darin finden Sie die Definition eines Intents (Snips-NLU verwendet intern dieselbe Bezeichnung wie wir) mit dem Namen *stop*. Nun übergeben wir in *utterances* einige Beispielausdrücke, die diesen Intent charakterisieren.

Listing 5.14 Definition des Stop-Intent

```
1.  type: intent
2.  name: stop
3.  utterances:
4.    - stop
5.    - halt
```

```
6.    - sei still
7.    - sei ruhig
8.    - abbrechen
9.    - schweigefuchs
```

Sie sehen, dass wir kaum mehr Tipparbeit zu leisten haben als bei dem Framework, das auf Regular Expressions basiert. Nun gut, hier haben wir bisher weder einen Funktionsaufruf noch einen Parameter definiert.

Schauen wir uns gleich noch ein Beispiel an. Dabei wird es zugunsten der Vergleichbarkeit um die Zeitansage gehen, wie auch schon in Abschnitt 5.2.1. Legen Sie ebenfalls im Ordner *dialog-datasets* die Datei *time_dataset.yaml* an und befüllen Sie diese gemäß Listing 5.15.

Listing 5.15 Definition der Entität country und des Time-Intents

```
1.  type: entity
2.  name: country
3.  automatically_extensible: true
4.  use_synonyms: false
5.  matching_strictness: 0.8
6.  values:
7.    - deutschland
8.    - england
9.
10. type: intent
11. name: getTime
12. utterances:
13.   - Wie spät ist es in [place:country](deutschland)
14.   - Wie viel Uhr ist es in [place:country](deutschland)
15.   - Uhrzeit in [place:country](deutschland)
16.   - Wie spät ist es
17.   - Wie viel Uhr ist es
18.   - Uhrzeit
```

Hier lernen Sie einen zweiten Typ kennen, *Entity*. Eine Entität kann ein beliebiges Objekt sein, in diesem Fall definieren wir ein Land. Eine Entität kann mehrere Eigenschaften haben. Die erste, *automatically_extensible*, erlaubt dem Parser, eigene Werte für diese Entität anzunehmen. Falls die Eigenschaft auf *False* steht, werden nur die Werte erlaubt, die wir in der Liste *values* übergeben. Es folgt der Parameter *use_synonyms*, der es erlaubt, Synonyme für die diese Entität zu ermitteln und auch diese als zulässiges Element zu vermerken. In Zeile 5 von Listing 5.15 wird zuletzt noch *matching_strictness* gesetzt. Es gibt vor, wie streng das Framework beim Erkennen der Entität sein soll. Nun folgt eine Liste von Werten, die die Entität exemplarisch oder ausschließlich annehmen kann.

TIPP: *Snips-NLU* verfügt über eine recht große Liste von vordefinierten Entitäten, darunter etwa Geldbeträge, Zahlen, Datums- und Zeitangaben, Städte, Länder, Regionen etc. Eine Übersicht finden Sie in der entsprechenden Dokumentation unter *https://snips-nlu.readthedocs.io/en/latest/builtin_entities.html*.

Es folgt die Definition des Intents *getTime* ab Zeile 10. In der Liste *utterances* sehen Sie nun, dass wir an bestimmten Stellen unsere Entität vom Typ *country* referenzieren. Der in eckigen Klammern stehende Ausdruck *place:country* bedeutet, dass eine Variable mit Namen *place* und vom Typ *country* an dieser Stelle zu finden sein soll. Für eine solche Variable (einen *Slot* im *Snips-NLU*-Jargon) wird auch immer ein Beispielwert angegeben, hier *deutschland*.

HINWEIS: Wie findet man eigentlich Synonyme für ein bestimmtes Wort? Das ist tatsächlich nicht schwer umzusetzen, wie Listing 5.16 zeigt. Wir setzen dabei auf Wortfelder (*Synsets*), die ähnliche Wörter gruppieren und so Sammlungen von Synonymen bilden.

Listing 5.16 Synonyme von good über Wordnet Synsets

```
from nltk.corpus import wordnet

for syn in wordnet.synsets("good"):
  for name in syn.lemma_names():
    print(name)
```

Das *n*, *v*, *a* oder *r* hinter dem Synonym deutet auf die Wortart hin, ist es also ein Nomen, ein Verb, ein Adjektiv oder ein Adverb. Die Nummer am Ende der Ausgabe indiziert verschiedene Bedeutungsgruppen, sodass Synonyme mit einer gleichen Bedeutung eine gleiche Nummer haben. Wordnet ist eine fantastische Bibliothek, die neben Synonymen auch Erklärungen zu einzelnen Wörtern vorhält oder Ähnlichkeiten zwischen Wörtern bewerten kann. Schauen Sie sich mal das offizielle Howto an *https://www.nltk.org/howto/wordnet.html*. ■

Die letzten drei Beispielsätze in *utterances* enthalten keine Ortsangabe. Das Framework wird den Parameter in diesem Fall nicht befüllen.

Fahren wir nun fort, indem wir Snips-NLU initialisieren. Löschen Sie den Initialisierungspart von *ChatbotAI* aus der Methode `__init__()` und ersetzen Sie diesen durch Listing 5.17.

Listing 5.17 Initialisierung von Snips-NLU

```
1.  logger.info("Initialisiere Chatbot...")
2.  self.nlu_engine = None
3.  dataset = Dataset.from_yaml_files("de", ['./dialog-datasets/stop_dataset.yaml',
4.    './dialog-datasets/time_dataset.yaml'])
5.  nlu_engine = SnipsNLUEngine(config=CONFIG_DE)
6.  self.nlu_engine = nlu_engine.fit(dataset)
7.
8.  if not self.nlu_engine:
9.    logger.error("Konnte Dialog Engine nicht laden.")
10.   sys.exit(1)
11. else:
12.   logger.debug("Dialog Metadaten: {}.", self.nlu_engine.dataset_metadata)
```

Darin Initialisieren wir *SnipsNLUEngine* auf Basis der deutschen Sprache (zu sehen an *CONFIG_DE* in Zeile 5). Es folgt der Aufruf der Methode `fit()`, die das Fine Tuning auf Basis unserer beiden YAML-Dateien, die wir zuvor erstellt haben, durchführt. Die Zeilen 8 bis 12 bestehen lediglich aus Error-Handling und Logging.

Damit Klassen und Module bekannt sind, fügen Sie noch die Imports aus Listing 5.18 hinzu. Die Bibliothek *pytz* wird uns gleich helfen, Zeitzonen zu verarbeiten.

Listing 5.18 Imports für die Verwendung von Snips-NLU, Ein- und Ausgabe und Verarbeitung von Zeitzonen

```
import io
import pytz
from snips_nlu import SnipsNLUEngine
from snips_nlu.default_configs import CONFIG_DE
from snips_nlu.dataset import Dataset
```

Entfernen Sie nun die Funktionen *stop* und *getTime*, also alle, die wir mit *register_call* annotiert haben und ersetzen Sie sie durch die in Listing 5.19. Der Hintergrund ist einmal natürlich der Austausch des Dialog-Frameworks, aber auch die Erweiterung der Methode *getTime* um eine Zeitzone. Diese ermitteln wir händisch über ein Dictionary, das erst mal nur Einträge für fünf Länder beinhaltet. Natürlich verfügen Amerika und China aufgrund ihrer Größe über mehrere Zeitzonen, aber das soll uns hier für einen ersten Entwurf erst mal nicht interessieren.

Listing 5.19 Neuimplementierung der Intents getTime und stop

```
1.  def getTime(place):
2.    country_timezone_map = {
3.      "deutschland": pytz.timezone('Europe/Berlin'),
4.      "england": pytz.timezone('Europe/London'),
5.      "frankreich": pytz.timezone('Europe/Paris'),
6.      "amerika": pytz.timezone('America/New_York'),
7.      "china": pytz.timezone('Asia/Shanghai')
8.    }
9.
10.   now = datetime.now()
11.   timezone = country_timezone_map.get(place.lower())
12.   if timezone:
13.     now = datetime.now(timezone)
14.     return "Es ist " + str(now.hour) + " Uhr und " + str(now.minute) +
15.       "Minuten in " + place.capitalize() + "."
16.   return "Es ist " + str(now.hour) + " Uhr und " + str(now.minute) +
17.     "Minuten."
18.
19. def stop():
20.   if va.tts.is_busy():
21.     va.tts.stop()
22.     return "okay ich bin still"
23.   return "Ich sage doch gar nichts"
```

Sie sehen, dass die Anwendungslogik sich nicht großartig ändert. Die Methode `stop()` sieht sogar so aus wie vorher, bloß dass wir die Annotation entfernt haben.

Letztendlich müssen wir, wie auch bei *ChatbotAI*, das Parsing in die Methode `run()` der Klasse *VoiceAssistant* integrieren. Löschen Sie dazu die Zeile `output = self.chat.respond(sentence)` und ersetzen Sie sie durch den Inhalt von Listing 5.20.

Listing 5.20 Parsen und Verarbeiten des Sprachbefehls

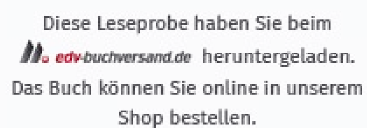
```
1.  parsing = self.nlu_engine.parse(sentence)
2.  output = ""
3.
4.  if parsing["intent"]["intentName"] == "getTime":
5.    if len(parsing["slots"]) == 0:
6.      output = getTime("Germany")
7.    elif parsing["slots"][0]["entity"] == "country":
8.      output = getTime(parsing["slots"][0]["rawValue"])
9.  elif parsing["intent"]["intentName"] == "stop":
10.   stop()
11. self.tts.say(output)
```

Das Parsen des verstandenen Satzes geschieht in Zeile 1 und liefert ein Objekt zurück, das als JSON-Objekt adressiert werden kann. Darin befinden sich unter dem Index *intent* zum Beispiel der *intentName*, der den Namen beinhaltet, der in der YAML-Datei festgelegt wurde. Auf diesen prüfen wir in Zeile 4 und Zeile 9 und rufen entsprechend die Funktion `getTime()` oder `stop()` auf. Im ersten Fall lesen wir die sogenannten *Slots* (Zeile 7) aus, die die Parameter beinhalten, die wir in Form einer Entität angelegt haben, also *country*. Ist ein solcher *Slot* befüllt, holen wir dessen Wert und übergeben ihn der Funktion `getTime()`. Die Funktion `stop()` benötigt keine Parameter und kann einfach aufgerufen werden, ohne dass die *Slots* Beachtung finden.

Bevor wir unsere nächste Version testen können, müssen wir noch *snips-nlu* und *pytz* über *pip* installieren. Dazu kommt noch eine Besonderheit: *Snips-NLU* benötigt ein deutsches Sprachpaket, das allerdings als *root* installiert werden muss. Auf Windows klicken Sie dazu einfach auf den Windows-Button unten links, tippen *Anaconda Prompt* und sobald das Icon im Startmenü erscheint, machen Sie einen Rechtsklick darauf und wählen *Als Administrator ausführen*. Dann aktivieren Sie wie gewohnt die Umgebung und führen den Befehl `python -m snips_nlu download de` aus. Vorher muss *Snips-NLU* zwingend installiert sein.

Beim Testen unseres Assistenten werden Sie feststellen, dass das Framework sehr gut darin ist, Intents zu erkennen und das, obwohl es im Vergleich zu unserem *Intent Classifier* aus Abschnitt 5.1.1 kaum Trainingsdaten bekommen hat.

Nun haben Sie beide Frameworks kennengelernt und gesehen, wie diese integriert werden können. Für mich persönlich ist die Definition von Intents in beiden Fällen ähnlich simpel, bloß dass die Formatierung für den Machine-Learning-Ansatz ein wenig sauberer ist und dass jedem Start des Assistenten ein Training vorausgeht, das jedoch in wenigen Sekunden durchgeführt ist.



Diese Leseprobe haben Sie beim edv-buchversand.de heruntergeladen.
Das Buch können Sie online in unserem Shop bestellen.

Hier zum Shop