TECHNISCHE UNIVERSITÄT
CHEMNITZ

Prototypische Realisierung eines Chatbots für eine Akzeptanzstudie im Hochschulkontext

Bericht zum Teamorientierten Praktikum

Fakultät für Informatik
Professur Technische Informatik

Eingereicht von: Alexander Aristarhov
Matrikel Nr.: 529327
Einreichungsdatum: 28.03.2021

Betreuer: Prof. Dr. W. Hardt
Dipl. Inf. R. Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Rahmen des Kurses *Teamorientiertes Projekt/Praktikum, Praktikum ESS* haben wir das Projekt *Chat Bot Entwicklung* erarbeitet. Wenn in dieser Arbeit von *uns* oder *wir* die Rede ist, sind damit die Gruppenmitglieder des Praktikums gemeint. Wir sind mein Gruppenpartner, Felix Finke, und ich, Alexander Aristarhov, als Verfasser dieser Arbeit.

Die exakte Problemdefinition lautet: *"Prototypische Realisierung eines Chatbots für eine Akzeptanzstudie im Hochschulkontext"*. Das Ziel ist es herauszufinden, ob ein Chatbot an einer Hochschule eine akzeptable Resonanz im Zusammenhang mit seinem Einsatzgebiet erhält. Dafür ist es nötig, dass ein Chat Bot in einer Testoberfläche realisiert wird, um im späteren Verlauf an einem Testpublikum einer Hochschule geprüft zu werden. Im besonderen wurde das Einsatzgebiet für einen FAQ Bereich des Kurses *Hauptseminar AUTOSAR Based Software Design* ausgewählt [vgl. 20, Seite 2].

Einzelne Ziele sind:

1. Eine Literaturrecherche zu den verfügbaren opensource Chatbot Frameworks.
   - Im Ergebnis der Literaturrecherche ein passendes Chatbot Framework auswählen.
2. Erstellung eines Konzepts zur Realisierung eines Chatbots.
   - Sich mit der Architektur des ausgewählten Frameworks vertraut machen und notwendige Schritte planen.
3. Implementierung eines ersten minimalen Chatbots.
   - Die Umgebung des Chatbot Frameworks aufsetzen und den Chatbot lauffähig machen.
4. Erweitern der Funktionalität bis zum FAQ Chatbot

Diese Arbeit soll die Umsetzung des Projekts erläutern. Dabei werden im besonderem die Punkte *Stand der Technik*, *Konzeption*, *Implementation* und *Evaluierung* behandelt. Es handelt sich bei diesem Projekt, wie auch in der Problemdefinition genannt, um einen Prototypen zum Zwecke der Ermittlung der Sinnhaftigkeit eines Chatbots im Hochschulkontext. Als Schlussfolgerung wird das Ergebnis des Projekts mit der Problemdefinition verglichen und bewertet.

2 Stand der Technik

Dieses Kapitel wird den Begriff *Chatbot* erläutern und dabei die Klassifikation sowie den Aufbau erklären. Weiterführend werden Kriterien für die Wahl eines passenden Chatbots ermittelt und in der Schlussfolgerung ein passendes Framework für dieses Projekt gewählt.

2.1 Grundlagen

Die beiden Wörter Plaudern (engl. *Chat*) und Roboter (engl. *Robot*, in Kurzform: *Bot*) bilden zusammen die englische Wortkombination *Chatbot*. Die einzelnen Aufgaben und Funktionen der beiden Wörter ergeben auch das Aufgabenfeld eines Chatbots. Der Chatbot soll einen Dialog mit dem menschlichen Benutzer führen (abgeleitet von *Chat*) und autonom reagieren (abgeleitet von *Robot*). Somit existiert eine Interaktion zwischen menschlichen Benutzer und dem Chatbot. Die Interaktion soll mittels natürlicher Sprache erfolgen. Die übertragenen Informationen werden vom Chatbot interpretiert und ausgewertet um anschließend geeignet zu reagieren. Die Antwort erfolgt ebenfalls in natürlicher Sprache. [vgl. 21, Seite 3-4].

Hierbei ist es wichtig ähnliche Begriffe, wie *Virtuelle Assistenten* und *Conversational Agents* (dt. Konversationsagent), von *klassischen Chatbots* abzugrenzen. Ein Virtueller Assistent hat immer die Aufgabe, dem Nutzer konkrete Aufgaben abzunehmen und den Umgang mit einem System zu erleichtern. Dabei muss der Virtuelle Assistent nicht zwangsläufig einen Dialog mit dem Benutzer führen. Der klassische Chatbot soll einen *einfachen* Dialog mit dem Nutzer führen und somit einen Gesprächspartner simulieren. Der Conversational Agent bildet eine Kombination aus klassischen Chatbot und Virtuellen Assistent, welcher mittels eines Dialoginterfaces mit einem Benutzer kommuniziert und gleichzeitig für konkrete Aufgaben Unterstützung liefert [vgl. 21, Seite 3-4].

In dieser Arbeit wird der Begriff *Chatbot* verwendet, dabei ist präziser ein *Conversational Agent* gemeint.

2.2 Klassifikation

Chatbots können sich anhand ihrer *Funktionsweise* und ihrem *Verwendungszweck* unterscheiden. Deshalb werden Chatbots auch anhand dieser beiden Kriterien besonders klassifiziert.

Mit dem Verwendungszweck sind die Einsatzmöglichkeiten und Aufgaben des Chatbots gemeint. Die Funktionsweise wird durch die Technik und Umsetzung des Chatbots beschrieben [vgl. 21, Seite 6].

2.2.1 Verwendungszweck

Beim Verwendungszweck wird in die Klassen *zielorientiert* und *nicht-zielorientiert* unterteilt.

Nicht-Zielorientierte Chatbots sind Chatbots, welche keine expliziten Aufgaben erfüllen sondern mehr eine soziale Interaktion simulieren sollen. Das heißt sie führen einen Dialog mit einem Benutzer und es entsteht dabei ein besonders realistisches Gespräch.

Zielorientierte Chatbots haben die Aufgabe dem Benutzer bei einer konkreten Anfrage eine Antwort zu liefern oder zu helfen. Je nach Form der Hilfe kann man zielorientierte Chatbots in die Unterklassen *Informationsbot* und *Transaktionsbot* unterteilen. Ein Informationsbot soll auf Anfrage eines Benutzers oder nach einer bestimmter Konfiguration (z.B.: nach bestimmter Zeit) Antworten liefern. Die Antworten stellen meist eine Information in natürlicher Sprache dar. Ein Transaktionsbot soll eine komplexe Aktion für den Benutzer durchführen (z.B.: Bestellung einer Mahlzeit). Dabei gibt der Nutzer nur notwendige Informationen an und der Transaktionsbot führt im Kontext die gewollte Aktion durch.

Die Trennung dieser zielorientierter Unterklassen ist aber nicht immer vorhanden, da manche Chatbots beide Eigenschaften besitzen [vgl. 21, Seite 6].

2.2.2 Funktionsweise

Die grobe funktionale Einteilung von Chatbots besteht aus *regelbasierten*, *datenbasierten* und *formularbasierten* Funktionsweisen. Dabei vermischen sich diese Funktionsweisen in der Praxis, so dass ein Chatbot mehrere oder sogar alle Merkmale dieser Funktionsweisen haben kann.

Regelbasierte Chatbots bestehen aus fest definierten Regeln oder Mustern. Das heißt, dass alle Inputs und Outputs bekannt seien müssen. Wenn also vom Nutzer ein Input kommt, der nicht exakt so definiert ist, dann kann dieser Input auch nicht behandelt werden. Man kann diese Funktionalität mit weiteren Strategien wie *Schlüsselwörtern* oder *Textmustern* erweitern.

Datenbasierte Chatbots sind mit maschinellen Lernen (engl. *Machine Learning*) ausgestattet. Deshalb sind sie in der Lage Inputs zu verarbeiten, welche nicht exakt so definiert sind, weil dynamisch mittels großer Datenmengen gearbeitet wird. Somit sind die Inputs und Outputs im Vergleich zu *regelbasierten* Chatbots flexibler. Trotzdem kann es vorkommen, dass der Chatbot nicht in der Lage ist alle Inputs dynamisch zu interpretieren, weil im Voraus eine Menge an Daten (so genannte *Intents*) definiert werden müssen und diese begrenzen dann auch den Interpretationsfreiraum eines Chatbots.

Formularbasierte Chatbots besitzen eine besondere Funktionalität im Umgang mit strukturierten Daten. Dabei versucht der Chatbot anhand der Anfrage des Benutzers ein passendes Formular zu finden und dann schrittweise die jeweiligen Attribute abzufragen [vgl. 21, Seite 7].

2.3 Aufbau eines Chatbots

Die Anforderung aus Sicht des Benutzers ist einfach: Es wird eine Nachricht an den Chatbot gesendet, dieser verarbeitet den Inhalt der Nachricht und liefert eine passende Antwort. Nun werden aber die einzelnen Schritte etwas genauer behandelt.

Die gesendete Anfrage muss vom Chatbot *verstanden* werden, dafür wird *Natural Language Understanding* verwendet. Die Aufgaben davon sind, dass die Absicht (engl. *intent*) des Benutzers erkannt wird. Dafür sind folgende Schritte notwendig:

1. *Data cleaning*: Der Input wird von inhaltlich unwichtigen Inhalten *bereinigt* (z.B.: Tippfehler korrigieren, Satzzeichen entfernen, Groß- und Kleinschreibung anpassen, Stopplisten verwenden, ...)
2. *Formale Aufbereitung*: Es ist eine Struktur notwendig, um einen Umgang mit natürlicher Sprache zu ermöglichen. Dafür wird prinzipiell *Natural Language Processing* verwendet, welche das *Verstehen* (*Natural Language Understanding*) und *Erzeugen* (*Natural Language Generation*) natürlicher Sprache beinhaltet. Eine passende Methode wird für den Einsatzgebiet des Chatbots ermittelt.
3. *Intent Matching/Slot Filling*: Hier soll zu den aufbereiteten Daten eine passende Absicht gefunden werden (*Intent Matching*). Außerdem besteht die Möglichkeit, dass die Daten eine Information beinhalten, welche für einen Attribut bestimmt sind, dann muss diese Information zum Attribut adressiert werden (*Slot Filling*).
4. *Steuerung*: Nun wo die Absicht klar ist, kann die Steuerung über weitere Schritte entscheiden. Es können weitere Informationen aus einer Datenbank angefordert, eine einfach Antwort geliefert oder zum Kontext der Zustand des Dialogs verwaltet werden.
5. *Antwort*: Zum Schluss muss eine Antwort für den Benutzer generiert werden. Je nach Entscheidung der *Steuerung* kann es eine feste Antwort sein, möglicherweise mit Parametern oder sogar ganz dynamisch.

[vgl. 21, Seite 8-10]

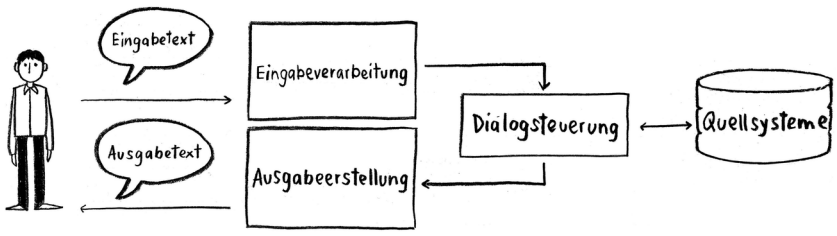

Abbildung 1: Vereinfachter Aufbau eines Chatbots

2.4 Wahl für dieses Projekt

Für den Einsatz eines Chatbots als Unterstützer im FAQ-Bereich eines Kurses, haben wir folgende Kriterien festgelegt:

- **Rechtslage**: Es ist notwendig, dass es sich um ein OpenSource-Framework handelt, weil sonst die kommerzielle Nutzung an der TU Chemnitz, ohne Rechte, nicht möglich wäre.
- **Umfang**: Da es sich um eine prototypische Realisierung handelt und das Einsatzgebiet auf den FAQ-Bereich begrenzt ist, muss das Framework nicht unnötig großen Umfang an Funktionen besitzen.
- **Kompatibilität**: Der Chatbot soll als eine Webapplikation erstellt und somit auch geeignet sein in eine Webseite integriert zu werden.

Um den Umfang des Chatbots etwas mehr zu spezifizieren, wollen wir uns auf die *datenbasierte* Funktionalität beschränken. Es wird davon ausgegangen, dass eine *formularbasierte* Funktionalität bei einem Chatbot in einem FAQ-Bereich keinen Zweck erfüllt, da auf einen interpretierten Input eine klare Antwort erwartet wird. Die *datenbasierte* Funktionalität impliziert, dass es sich um einen *zielorientierten* Chatbot handeln sollte.

Anhand dieser Kriterien und der Empfehlung der Praktikumsaufsicht ist die Wahl auf das *RASA-Framework* gefallen. Rasa ist ein OpenSource-Framework mit vermischten *datenbasierten* und *formularbasierten* funktionalen Elementen [vgl. 6]. Dabei ist dieser hohe Grad an Funktionalität kein besonders großer Nachteil, weil die ungebrauchten Funktionen einfach nicht genutzt werden, aber dafür ein Potential und Spielraum für die Entwicklung des Projekts bieten. Rasa hat außerdem eine breite Unterstützung an Schnittstellen und explizit die REST-Schnittstelle ist von Interesse bei einer Webapplikation [vgl. 16].

Schlussendlich wird eine graphische Oberfläche benötigt und dafür eignet sich ein weiteres Framework mit einer Chatintegration. Auch hier müssen die Kriterien erfüllt sein, deshalb ist die Wahl auf *Chatroom* von Scalableminds gefallen [vgl. 17]. *Chatroom* ist ein Framework welches ein Javascript erstellt und ist damit HTML kompatibel. Es eignet sich daher hervorragend für eine eigene Webseite.

In Kombination mit *Chatroom* sind alle oben genannten Kriterien erfüllt.

3 Konzeption

In dem Abschnitt *Wahl für dieses Projekt* ist die Wahl auf das Rasa-Framework gefallen. Deshalb ist die Konzeption eng mit den Eigenschaften und Aufbau des Rasa-Frameworks verbunden. Grundlegend gibt es drei Ebenen in der Architektur um dieses Projekt umzusetzen:

1. *Back-End*: Das Rasa-Framework benötigt eine *python* Umgebung.
2. *Rasa-Frame*: Das eigentliche Framework mit seiner internen Architektur, Funktionen und Schnittstellen.
3. *Front-End*: Eine Webapplikation mit graphischer Oberfläche.

3.1 Back-End

Beim Backend handelt es sich um eine leichtgewichtige *virtuelle Umgebung*. Da Rasa auf Python basiert, ist so eine Virtuelle Umgebung eine Lösung, damit das Projekt unabhängig von der Systemumgebung laufen kann. Außerdem werden lokale Pakete und Bibliotheken zur Verfügung gestellt [vgl. 2].

3.2 Rasa-Frame

Das Herzstück des Projekts ist das Rasa-Framework, welche alle Funktionen eines zielorientierten, datenbasierten Chatbots erfüllt. Um das Konzept von Rasa verständlich zu machen, werden die wichtigsten Bestandteile des Frameworks beschrieben:

- *Config*: In der Config sind die Bestandteile der *Pipeline* definiert, anhand welcher alle Daten verarbeitet werden und ein *Model* ergeben. Die *Policies* definieren die Regeln, wie das fertige *Model* auf den Input reagieren soll [vgl. 8].
- *Credentials*: Hier werden die Schnittstellen für Front-End-Applikationen definiert [vgl. 16].
- *Domain*: Die Domain listet alle *Intents*, *Entities*, *Slots*, *Responses*, *Forms* und *Actions* für den Chatbot auf. Man kann es als Inhaltsangabe der Daten betrachten [vgl. 5].
- *Endpoints*: Hier werden Schnittstellen für Services, Models und Datenbanken definiert, welche Rasa selbst verwenden soll.

- *Actions*: Diese dienen dazu besondere Funktionen auszuführen, wenn das *Model* anhand des Inputs eine *Action* vermutet. Das kann eine einfache Rückgabe (engl. *Response*), benutzerdefinierte *Action* oder ein Formular (engl. *Form*) sein [vgl. 3].
- *Data*: Hier werden die Daten für *Natural Language Understanding* (in Kurzform: *NLU*), Regeln (engl. *Rules*) und Geschichten (engl. *Stories*) beschrieben.
  - *NLU*: An dieser Stelle werden die möglichen Inputs anhand von Beispieltexten beschrieben. Jeder Beispieltext wird zu einem *Intent* zugeordnet. Dabei können diese Beispieltexte mit *Entities* ausgestattet werden um direkt wichtige Informationen auszulesen. Außerdem kann man pro *Intent* mehrere Beispieltexte als *Synonyme* beschreiben [vgl. 9]. Später soll hier pro *Intent* eine FAQ-Frage abgedeckt werden.
  - *Stories*: Als *Story* werden erwartete Abläufe im Chat definiert. Somit fällt es dem Chatbot einfacher seinen Kontext zu behalten [vgl. 12].
  - *Rules*: *Rules* beschreiben kurze Konversationen, welche immer oder unter einer bestimmten Bedingung gelten sollen (z. B. Begrüßung) [vgl. 11].
- *Models*: Hierbei handelt es sich um fertig erstellte Pakete. Diese sind abhängig von der *Config*, *Domain* und allen *Daten*. Wenn Rasa als Server gestartet wird, muss ein *Model* mit angegeben werden, welches dann als Grundlage der Kommunikation dient. *Models* eignen sich hervorragend als lieferbares fertiges Paket [vgl. 15].
- *Tests*: Rasa liefert ein eigenes Test-Modul. Man kann also ein *Model* mit Test-Dialogen überprüfen und bekommt als Ergebnis Graphiken, Tabellen und Log-Dateien [vgl. 14].

3.3 Front-End

Damit der Chatbot nun nutzbar ist, muss man eine graphische Oberfläche erschaffen und Eingaben des Benutzers an die REST-Schnittstelle von Rasa schicken [vgl. 16]. Dafür soll eine prototypische Webseite erstellt werden. Die Webseite kann dann auf einer eigenen Adresse unabhängig von Rasa laufen. Somit entsteht eine Client-Server-Architektur. Als Webapplikation soll *Chatroom* von Scalableminds dienen [vgl. 17].

3.4 Zusammenfassung

Mit den drei Ebenen der Architektur können die Anforderung des Projekts erfüllt werden. Als Back-End dient eine systemunabhängige Umgebung. Rasa-Frame liefert alle nötigen Funktionen und Klassen um die Daten zu verarbeiten und erstellt ein fertiges Model. Das Front-End dient als Oberfläche und als Schnittstelle zum Nutzer.

4 Implementation

4.1 System einrichten

4.1.1 Versionierungsystem

Der wichtigste erste Schritt ist die Einrichtung eines Versionierungsystems. Damit wird nicht nur Ordnung in ein Projekt gebracht, sondern auch Überblick über den Fortschritt, Protokoll zu allen Änderungen und die Möglichkeit produktiv im Team zu arbeiten. Deshalb haben wir ein Projekt auf GitHub eingerichtet. Dort befindet sich der gesamte Fortschritt unserer Arbeit. In der *README* gibt es eine Beschreibung zum Projekt und eine Anleitung wie man das Projekt verwenden kann. *Der aktuelle Zustand des Projekts kann HIER*[1] *auf GitHub betrachtet werden.*

4.1.2 Back-End

Bevor wir Rasa installieren und verwenden können, müssen wir eine sichere und unabhängige Umgebung schaffen. Da das Rasa-Framework auf *Python* basiert, brauchen wir eine Virtuelle Umgebung, welche auch Python unterstützt. Da ergeben sich zwei Kandidaten: ***Venv*** und ***Anaconda***.

Venv ist die eigene leichtgewichtige Virtuelle Umgebung von Python [siehe 2]. Diese wird mit *Python* direkt mitgeliefert. Die unterstützten Python Versionen sind *HIER*[2] auf der Webseite von Rasa zu finden. Notwendig ist auch der *Microsoft VC++ Compiler*[3]. Meistens wird `pip` als Package-Manager automatisch mit *Python* installiert, aber nicht unbedingt mit der aktuellsten Version. Also ist ein manuelles Aktualisieren in der Konsole empfohlen.

Anaconda ist ein Manager. Dieser Manager kann für eine virtuelle Umgebung, Pakete verwalten und dient auch als Python Distribution. Dabei besitzt Anaconda auch einen so genannten *Navigator*, um diese Umgebung auch graphisch in einer Applikation darzustellen und liefert außerdem eine eigene Kommandozeile (engl. *Prompt*) [siehe 1]. Auch hier ist der *Microsoft VC++ Compiler*[3] notwendig. Mit Anaconda ist es möglich die Python Version nachträglich im *Navigator* anzupassen. Ein weiterer Vorteil von Anaconda ist, dass man die fertig eingerichtete Umgebung einfach als Sammlung ausliefern kann. So braucht der zukünftige Nutzer sich nicht um die vollständige Installation kümmern.

Es spielt außerdem keine Rolle, welches Back-End verwendet wird, weil sie jeweils keinen Einfluss auf die Projektdaten haben.

Wir haben uns als Back-End für Venv entschieden.

[1] Projekt GitHub: https://github.com/PLAYLEX/RasaTUC
[2] Unterstützte Python Versionen: https://rasa.com/docs/rasa/installation
[3] MS VC++ Compiler: https://visualstudio.microsoft.com/de/visual-cpp-build-tools/

4.2 Rasa-Framework Einrichtung

4.2.1 Vorbereitung

Wichtig zu erwähnen ist es, dass bei dem bereitgestellten *Back-End* als Bibliotheken *uJson* und *TensorFlow* installiert sind. Eine besonders wichtige Rolle spielt *TensorFlow*, weil es sich hierbei um ein Framework handelt, welches Pipeline-artiges maschinelles Lernen anbietet [siehe 19]. Diese Funktion wird Grundlegend bei Rasa verwendet, um die entsprechenden Modelle zu erstellen.

Das FAQ wird in englischer Sprache bereitgestellt, deshalb lohnt es sich ein Sprachmodell für das System bereitzustellen, welches schon mit englischer Sprache trainiert ist. Bei einem Sprachmodell handelt es sich um ein trainiertes Modul mit sehr vielen Wörtern der jeweiligen Sprache und der Möglichkeit einen Satz in seine Bestandteile zu zerlegen. Das dient dann dazu die *Intents* für das System zu verstehen. Dafür haben wir uns für *SpaCy* entschieden, weil es mit Rasa kompatibel, frei zur kommerziellen Nutzung ist und umfangreiche Sprachmodule anbietet [vgl. 18].

Die Konfiguration vom Sprachmodell findet nach der Installation von Rasa statt.

4.2.2 Installation

Man navigiert in seine virtuelle Umgebung, denn jetzt wird das Projekt anhand des Rasa-Frameworks initialisiert. Wie schon in *Vorbereitung* erwähnt, wird ein Sprachmodell verwendet, dieses kann jetzt auch installiert werden:

```
pip install rasa[spacy]
python -m spacy download en_core_web_md
python -m spacy link en_core_web_md en
```

Mit dem ersten Befehl, werden die *Rasa Open Source* und *SpaCy* als Dependency in der Virtuellen Umgebung installiert. Jetzt muss SpaCy in Python das Modell laden:

```
import spacy
nlp = spacy.load("en_core_web_md")
exit()
```

Schlussendlich wird Rasa initialisiert:

```
rasa init
```

Wichtigste Befehle in Rasa sind:

- *rasa init*: Erstellt ein neues Projekt mit Beispieldaten ("Hello World").
- *rasa train*: Erstellt ein Modell, basierend auf der `config.yml` und Daten.

- *rasa run*: Startet den Rasa Server mit einem trainierten Modell.
- *rasa test*: Das trainierte Modell wird an allen Testdaten geprüft. [vgl. 4]

Damit ergibt sich folgende Projektstruktur:

```
/
└── rasa/
    ├── actions/
    │   └── actions.py ..... Optionale actions
    ├── data/
    │   ├── nlu.yml ..... Definition von Intents für den Chatbot
    │   ├── rules.yml ..... Regeln für das Verhalten vom Chatbot
    │   └── stories.yml ..... Erwartete Abläufe von Chats
    ├── models/ ..... Generierten Modelle
    │   └── ...
    ├── results/ ..... Ergebnisse vom letzten Test
    │   └── ...
    ├── test/ ..... Erstellte test-stories
    │   └── ...
    ├── config.yml ..... Konfiguration um die Modelle zu trainieren
    ├── credentials.yml ..... Registrierung fremder Applikationen
    ├── domain.yml ..... Initialisierung des Chatbots
    ├── endpoints.yml ..... Schnittstellen (API)
    └── rasa.log ..... Normales Rasa-Log-File
```

Wir haben jetzt *Rasa* mit *SpaCy* installiert und müssen die Konfiguration in `config.yml` anpassen:

```
pipeline:
  - name: "SpacyNLP"
    model: "en_core_web_md"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 200
  - name: FallbackClassifier
    threshold: 0.3
    ambiguity_threshold: 0.1
```

Nun ist das System vollständig aufgesetzt und bereit zur weiteren Konfiguration.

4.3 FAQ Integration

Dieses Projekt soll einen Chatbot realisieren, welcher auf FAQ Fragen antworten soll. Das bedeutet, dass für jede Antwort, die in unserem FAQ vorhanden ist auch eine entsprechende *Response* existieren muss. In der Regel wird eine *Response* nur dann vom Chatbot zurückgeliefert, wenn dazu ein entsprechendes *Intent* erkannt wurde. Also ist unser Ansatz, für jede Frage ein entsprechendes *Intent* zu schreiben.

4.3.1 Intents Definition

Die initiale Definition der *Intents* ist einfach. Wir definieren für jede Frage ein eindeutiges *Intent* in der `domain.yml`. Dazu werden entsprechend auch die *Responses* in der selben Datei definiert und direkt mit ihren Antworttexten versehen.

Nun werden die einzelnen *Intents* in der `nlu.yml` beschrieben. Das bedeutet, dass für jedes *Intent* Schlüsselwörter oder Sätze geschrieben werden, welche inhaltlich auf die passende Absicht des Nutzers zeigen kann. Am Anfang haben wir exakt den Satz genommen, der auch in dem FAQ beschrieben war. Das bedeutet immer ein Textbeispiel pro *Intent*. Damit jede Frage modular ist, wurde zu jeder Frage eine *Rule* in `rules.yml` geschrieben.

```
- intent: switch_topic
  examples: |
    - how can i switch my topic?
```

Abbildung 2: Intent mit nur einem Textbeispiel

4.3.2 Probleme mit Intents

Schnell war uns klar, dass ein einziger Textbeispiel pro *Intent* für fehlerhafte Erkennung der wahren Absicht des Nutzers sorgen wird. In *Abbildung 2* wird gezeigt, welches einzelne Textbeispiel für das *Intent 'switch topic'* beschrieben wird. Dabei wird genau dieser Satz gelernt und später vom Chatbot bzw. Model interpretiert. Besonders problematisch werden Synonyme: Was passiert also, wenn der Nutze statt *'switch' 'change'* schreibt? Oder statt *'topic' 'theme'* verwendet? Möglicherweise wird der *Intent* einfach nicht richtig erkannt oder es wird mit einem anderen *Intent* fehlinterpretiert, weil dort exakt das andere Wort verwendet wird. Eine weitere Fehlerquelle ist, wenn der Nutzer sehr wenig oder sehr viel in der Relation zum trainierten Textbeispiel schreibt. Es besteht also die Möglichkeit, dass der Nutzer nur ein Wort schreibt oder sogar mehrere Fragen am Stück. Dann findet das Model keine passende Antwort.

Unsere Test haben genau das vermutete Verhalten gezeigt. Im nächsten Abschnitt werden Lösungen und Lösungsansätze vorgestellt.

4.3.3 Lösungen für Intents

Länge der Eingabe: In der `config.yml` ist ein *FallbackClassifier* definiert. Dieser bestimmt eine untere Schranke für das Model, welche erreicht werden muss bei der Übereinstimmung des Nutzerinputs mit einem beliebigen *Intent*. Außerdem werden in der selben Datei die *policies* deklariert, welche beschreiben, wie sich das Model zu verhalten hat, falls die untere Schranke erreicht wird. Wir haben es so konfiguriert, dass eine *Rule* in der `rules.yml` existiert, um dem Nutzer eine Mitteilung zu senden. Wenn die Formulierung also undeutlich ist, wird der Nutzer informiert, dass er seine Eingabe umformulieren soll.

Für das Problem mit den Synonymen gibt es mehrere Lösungen.

Die erste Lösung ist es *Synonyms*, *Regular Expressions* und *Lookup tables* zu verwenden. Diese sind integrierte Bestandteile vom Rasa-Framework um Variablen mehrerer Bedeutung und Aufzählungen zusammenzufassen. *Synonyms* bestehen aus einem Beispielwort und einer Auflistung seiner Synonyme, wobei Synonyme auch aus mehreren Wörtern bestehen können (z. B. *Konto* und *Bank Konto*) [vgl. 13]. Bei *Regular Expression* werden Muster für Zahlen und Zeichenfolgen definiert. Das dient für die Eingabe von Nutzernummern (z. B. *Kontonummer*) [vgl. 10]. *Lookup Tables* dienen dazu eine Auflistung von Begriffen zu erstellen, welche zu einer Kategorie gehören (z. B. für *Bank* verschiedene Bankinstitute aufzählen) [vgl. 7].

Die zweite Lösung ist die Verwendung vieler verschiedener Textbeispiele pro *Intent*. Dabei können viele verschiedene Formulierungen, Längen und Versionen der Texte erstellt werden. Die Theorie ist, umso mehr Texte das Model kennt, umso besser kann es die Eingabe interpretieren.

Wir haben uns für den zweiten Ansatz entschieden, weil dort die Umsetzung am schnellsten ist. Unser Ziel ist es einen Prototypen zu erstellen, da ist Detailreichhaltigkeit und hohe Komplexität nicht notwendig. Wir haben uns also für jeden *Intent* viele verschiedene Textbeispiele ausgedacht. Um aber noch mehr Inhalte zu generieren, wurde eine Umfrage an die Teilnehmer des *Hauptseminar AUTOSAR Based Software Design* erstellt. Die Idee war, dass die Teilnehmer wahrscheinlich selbst schon mal das FAQ genutzt und somit schon die eine oder andere Frage im Kopf formuliert haben. In der Umfrage wurde in sechs Teilgebiete aufgeteilt: *Organisation*, *Topic Selection*, *Presentation*, *Report*, *Report Submission* und *Open Questions*. Die Teilnehmer sollten für jedes Teilgebiet Fragen formulieren und dazu die Antwort, die sie erwarten. Die Auswertung der Umfrage hat ergeben, dass es insgesamt 17 Teilnehmer gab und davon 4 Teilnehmer mindestens eine Antwort geliefert haben. Insgesamt haben wir 17 Antworten gesammelt. Diese Antworten haben wir einzeln ausgewertet und dann in die *Intents* integriert. Die Auswertungsdateien und weitere Details zu der Umfrage liegen *HIER*[4] auf dem GitHub des Projekts.

In kommenden Abschnitt wird die Umsetzung der Webapplikation behandelt.

[4] Umfragedaten: https://github.com/PLAYLEX/RasaTUC/tree/main/docs

4.4 Webapplikation

Wie schon im Abschnitt *Wahl für dieses Projekt* erklärt, wird als Front-End *Chatroom* von Scalableminds dienen. Die Struktur ist einfach: Das Framework besteht aus Quelldateien (z. B. *SCSS*) und erstellt daraus verwendbare JavaScript und CSS Dateien, welche ganz traditionell in eine Webseite eingebaut werden können. Dieses Framework kommuniziert über die REST-Schnittstelle von Rasa.

Hinweis: Chatroom bietet für den Browser 'Chrome' die Möglichkeit an, den Chatverlauf mit Spracheingabe und Sprachausgabe auszuschmücken. Diese Funktion wurde von uns deaktiviert.

4.4.1 Chatroom Option

Es gibt prinzipiell zwei Möglichkeiten, wie man dieses Framework verwenden kann:

Hosted Version ist die von Scalableminds bereitgestellte Applikation, dabei befinden sich die JavaScript und CSS Dateien auf einem bereitgestellten Server. Diese Dateien werden im HTML der eigenen Webseite auf ihre Internetadresse referenziert. Der Nachteil hier liegt darin, dass man als Webentwickler keine Kontrolle über die Verfügbarkeit dieses Ressourcen hat und auch eine Konfiguration nicht möglich ist.

Builded Version ist die selbst kompilierte Version. Das bedeutet, es werden die Quelldateien vom Webentwickler selbst kompiliert und man erhält eigene JavaScript und CSS Dateien. Diese können ganz einfach als Ressourcen für die eigene Webseite verwendet werden.

4.4.2 Chatroom Umsetzung

Wir haben uns für die Builded Version entschieden, weil diese uns unabhängig von externen Servern macht und liefert auch direkt mehr Sicherheit.

Nachdem das Framework gebaut und compiliert wurde, erhalten wir unsere JavaScript und CSS Dateien. Diese Ressourcen werden nun in unsere Webseite mit der Konfiguration der standardmäßigen REST-Schnittstelle von RASA eingebaut. In der Regel ist das der Port *5005*. Die Webseite selbst läuft absichtlich auf dem Port *8000*, damit sie unabhängig vom Rasa-Server ist. Unter diesem Umstand muss man aber dem Rasa-Server in die Startbedingung die Zugriffserlaubnis einer externen Webseite (unsere Webseite) erteilen. Dies geschieht mit dem Parameter `--cors "http://localhost:8000"`.

Den genauen Aufbau der Webseite kann man *HIER*[5] auf GitHub sehen.

[5] Webseite erstellen: https://github.com/PLAYLEX/RasaTUC

4.5 Weitere Probleme mit Intents

Im Test haben sich weitere Unregelmäßigkeiten und Auffälligkeiten mit den *Intents* ergeben. Unter Umständen kommt es vor, dass es für das Model zu Schwierigkeiten hat eine richtige Entscheidung zu treffen bei der Wahl der *Intents*. Besonders dann, wenn sich einige Fragen oder Begriffe überschneiden.

4.6 Weitere Lösungen für Intents

Deshalb ist uns die Idee gekommen, dass man einen besonderen *Intent* mit einer dazugehörigen *Action* erstellen kann. Der *Intent* soll ein Trigger sein, welchen der Nutzer auslösen kann, wenn seine letzte Chatantwort unerwartet oder komplett falsch war. Die *Action* erstellt eine übersichtliche Log-Datei mit der letzten Eingabe des Nutzers und der falschen Vermutung des Models. Somit kann dann ein menschlicher Administrator regelmäßig diese gemeldeten Fehler auswerten und als neues Textbeispiel in den richtigen *Intent* hinzufügen. So sollten immer mehr Einträge pro *Intent* entstehen und die Genauigkeit verbessern.

4.7 Rasa Ausführung

Um den Chatbot zu starten müssen folgende Schritte durchgeführt werden:

- In die virtuelle Umgebung wechseln.
- Im Rasa-Verzeichnis den Server mit folgendem Befehl starten:
  - `rasa run -vv --cors "http://localhost:8000"`
  - *Die Bedeutung der einzelnen Parameter können HIER*[6] *in der Rasa Dokumentation nachgeschlagen werden.*
- Jetzt wird die Webseite gestartet.
- Zum Schluss müssen die *Actions* als interner Server in Rasa gestartet werden.
  - `rasa run actions`

[6] CLI: https://rasa.com/docs/rasa/command-line-interface

5 Evaluierung

5.1 Testing

Die Lauffähigkeit des Projekts wurde jederzeit mit dem *Versionierungsystem* gewährleistet. Jede Änderung am Projekt wurde von allen Teilnehmern überwacht und abgenommen.

Klassisches *Black-Box-Testing* und *White-Box-Texting* wurde nicht betrieben, weil das Rasa-Framework keine klassischen Funktionen anbietet, welche in solcher Weise getestet werden können. Jedoch bietet Rasa eine eigene Testing-Schnittstelle: Ein fertiges Model in Rasa kann mit dem Befehl `rasa test` getestet werden. Alle Test-Stories im Verzeichnis `rasa/tests/` werden nacheinander ausgeführt. Dabei verhält sich der Test erstmal wie ein normaler Chatverlauf. In den Test-Stories wird als zusätzliche Information auch die Absicht (*Intent*) der simulierten Nutzereingabe mitgeliefert. Wenn also das Model diese simulierte Eingabe falsch interpretiert, wird das erkannt und registriert. Sobald die Test-Stories erfolgreich durchgelaufen sind, werden im Verzeichnis `rasa/results/` die Ergebnisse in Form von Graphiken, Matrizen und weiterverwendbaren JSON Dateien dargestellt. Explizit werden Berichte zu der korrekten Interpretation, Fehlerquote und historischen Verlauf von *Intents* und *Stories* erstellt.

5.2 Ergebnis

Unser Ziel war es einen prototypischen Chatbot zu erstellen und für ihn eine geeignete Oberfläche zu erschaffen. Der aktuelle Zustand sieht folgendermaßen aus: Wenn alle Serverinstanzen laufen, dann ist der Nutzer in der Lage über eine Webapplikation mit dem Chatbot zu kommunizieren. Jeder einzelne Kommunikation wird mit einer Session-ID versehen, damit jeder einzelne Chat auch nur seinen Kontext hat und seine Antwort erhält. Wenn der Nutzer Fragen im Rahmen des FAQ-Bereichs stellt, wird dem Nutzer mit hoher Wahrscheinlichkeit eine aussagekräftige Antwort geliefert. Falls die Antwort fehlerhaft ist, hat der Nutzer die Möglichkeit mit der Eingabe `!wrong!` den Chatbot über den Fehler aufmerksam zu machen. Im Hintergrund wird ein personalisiertes Log-File mit der fehlerhaften Nachricht versehen und dem Nutzer wird eine Entschuldigung als Antwort geliefert. Zwar erhält der Nutzer somit keine bessere Antwort, jedoch ist einem Administrator möglich mit diesem personalisieren Log-File bei regulären Wartungsarbeiten das Model zu verbessern. Prinzipiell ist es möglich parallel zum laufenden Server weiter an den Quellinformationen (z. B. *Intents*) zu arbeiten, weil der laufende Server sich nur am Model bedient. Diese Option erlaubt es auch dieses Model auszuliefern. Selbstverständlich kann dieses ausgelieferte Model trotzdem nur in Umgebungen verwendet werden, welche alle notwendigen Abhängigkeiten für einen lauffähigen Rasa-Server anbieten. Das bedeutet eine Rasa-Installation ist trotzdem notwendig und zusätzliche *Actions*, welche als Dienste genutzt werden, müssen auch vorhanden sein.

6 Zusammenfassung

In der *Einleitung* haben wir das Ziel als eine "prototypische Umsetzung eines Chatbots für eine Akzeptanzstudie im Hochschulkontext"definiert. Das spezifische Ziel war es die praktischen Möglichkeiten von Chatbots zu erforschen und einen Prototypen für einen FAQ-Bereich zu erstellen.

Grundlegend kam bei dieser Anforderung vom *Verwendungszweck* nur ein *zielorientierter* Chatbot in Frage, da dieser Chatbot ein klare Aufgabe hat dem Nutzer Fragen zu beantworten. Außerdem muss die *Funktionsweise datenbasiert* sein, weil der Chatbot dynamisch, anhand der Eingabe des Nutzers die richtige Absicht interpretieren und eine passende Antwort liefern soll.

Die *Wahl für dieses Projekt* ist dabei auf das OpenSource Rasa-Framework gefallen, weil es die Kriterien des *Umfangs*, der *Rechtslage* und der *Kompatibilität* erfüllt.

Das Konzept besteht Grundlegend aus drei Ebenen: *Back-End*: Beschreibt die virtuelle Umgebung für Rasa, *Rasa-Frame*: Das eigentliche Framework mit allen Funktionalitäten und *Front-End*: Als Webapplikation und Schnittstelle zum Nutzer.

Der Hauptbestandteil bei der *Implementation* war die *FAQ Integration*, weil jede FAQ-Frage als *Intent* aufgefasst und mit einer *Rule* versehen wurde. Daraus trainiert man ein Model, welches für den Chatbot als Referenz dient. Als *Webapplikation* wurde *Chatroom* von Scalableminds genutzt, weil dieser auch die Kriterien des *Umfangs*, der *Rechtslage* und der *Kompatibilität* erfüllt. Somit wird eine JavaScript und CSS erstellt, welche einfach in einer Webseite verwendet werden. Der Webclient kommuniziert über eine REST-Schnittstelle mit dem Rasa-Server. Jeder Verbindung wird eine eigene eindeutige Session-ID zugewiesen.

Wie zu erwarten war während der Entwicklung die Genauigkeit des Models bei der Interpretation der Eingabe des Nutzers ein Problem. Dieses Problem wurde mit der Verwendung vieler Textbeispiele pro *Intent* teilweise gelöst. Mit einer *Action* hat der Nutzer sogar die Möglichkeit eine falsche Antwort zu markieren.

Im Ausblick kann man weitere *Lösungen für Intents* ausarbeiten, in dem man *Synonyms*, *Regular Expressions* und *Lookup tables* von Rasa verwendet. Außerdem kann man die Webapplikation attraktiver gestalten und mit Hinweisen (z. B. für falsche Eingaben) versehen. Prinzipiell ist das *Ergebnis* in einem Zustand, dass ein Praxisversuch durchgeführt werden kann.

Literatur

[1] Inc. Anaconda. *Anaconda Individual Edition*. 2021. URL: https://docs.anaconda.com/anaconda/ (besucht am 20. 03. 2021).

[2] Python Software Foundation. *PEP 405 – Python Virtual Environments*. 2021. URL: https://www.python.org/dev/peps/pep-0405/#abstract (besucht am 11. 03. 2021).

[3] Rasa Technologies GmbH. *Actions*. 2021. URL: https://rasa.com/docs/rasa/actions (besucht am 11. 03. 2021).

[4] Rasa Technologies GmbH. *Command Line Interface*. 2021. URL: https://rasa.com/docs/rasa/command-line-interface (besucht am 22. 03. 2021).

[5] Rasa Technologies GmbH. *Domain*. 2021. URL: https://rasa.com/docs/rasa/domain (besucht am 11. 03. 2021).

[6] Rasa Technologies GmbH. *Introduction to Rasa Open Source*. 2021. URL: https://rasa.com/docs/rasa/ (besucht am 10. 03. 2021).

[7] Rasa Technologies GmbH. *Lookup Table*. 2021. URL: https://rasa.com/docs/rasa/training-data-format#lookup-tables (besucht am 23. 03. 2021).

[8] Rasa Technologies GmbH. *Model Configuration*. 2021. URL: https://rasa.com/docs/rasa/model-configuration (besucht am 11. 03. 2021).

[9] Rasa Technologies GmbH. *NLU Training Data*. 2021. URL: https://rasa.com/docs/rasa/nlu-training-data (besucht am 11. 03. 2021).

[10] Rasa Technologies GmbH. *Regular Expression*. 2021. URL: https://rasa.com/docs/rasa/training-data-format#regular-expressions (besucht am 23. 03. 2021).

[11] Rasa Technologies GmbH. *Rules*. 2021. URL: https://rasa.com/docs/rasa/rules (besucht am 11. 03. 2021).

[12] Rasa Technologies GmbH. *Stories*. 2021. URL: https://rasa.com/docs/rasa/stories (besucht am 11. 03. 2021).

[13] Rasa Technologies GmbH. *Synonyms*. 2021. URL: https://rasa.com/docs/rasa/training-data-format#synonyms (besucht am 23. 03. 2021).

[14] Rasa Technologies GmbH. *Testing Your Assistant*. 2021. URL: https://rasa.com/docs/rasa/testing-your-assistant (besucht am 11. 03. 2021).

[15] Rasa Technologies GmbH. *Tuning Your NLU Model*. 2021. URL: https://rasa.com/docs/rasa/tuning-your-model (besucht am 11. 03. 2021).

[16] Rasa Technologies GmbH. *Your Own Website*. 2021. URL: https://rasa.com/docs/rasa/connectors/your-own-website (besucht am 10. 03. 2021).

[17] scalable minds GmbH. *React-based Chatroom Component for Rasa Stack*. 2020. URL: https://github.com/scalableminds/chatroom (besucht am 11. 03. 2021).

[18] Matthew Honnibal und Ines Montani. *spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing*. 2017. URL: https://spacy.io/usage/spacy-101 (besucht am 21. 03. 2021).

[19] Martin Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45166.pdf (besucht am 21. 03. 2021).

[20] Dipl. Inf. René Schmidt. *Themen Team orientieres Praktikum*. 2020. URL: https://ce-praktika.slack.com/files/UDHSEL57U/F01CYF86MQE/themen.pdf (besucht am 08. 03. 2021).

[21] Toni Stucki, Sara D'Onofrio und Edy Portmann. „Theoretische Grundlagen zu Chatbots". In: *Chatbots gestalten mit Praxisbeispielen der Schweizerischen Post: HMD Best Paper Award 2018*. Wiesbaden: Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-28586-9. DOI: 10.1007/978-3-658-28586-9_2. URL: https://doi.org/10.1007/978-3-658-28586-9_2 (besucht am 10. 03. 2021).