

Chatbot

DIPLOMARBEIT

verfasst im Rahmen der

Reife- und Diplomprüfung

an der

**Höhere Lehranstalt für Informationstechnologie,
Ausbildungsschwerpunkt Medientechnik**

Eingereicht von:

Felix Dumfarth
Lukas Starka

Betreuer:

Thomas Stütz

Projektpartner:

Leonding, April 2022

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, April 2022

Felix Dumfarth & Lukas Starka

Zur Verbesserung der Lesbarkeit wurde in diesem Dokument auf eine geschlechtsneutrale Ausdrucksweise verzichtet. Alle verwendeten Formulierungen richten sich jedoch an alle Geschlechter.

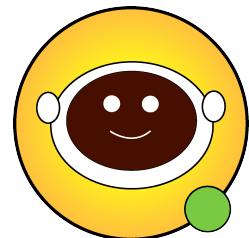
Abstract

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.



Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit dem Thema eines Chatbots, der mithilfe von Rasa erstellt worden ist und auf der Webseite der HTL Leonding zu finden sein soll. Dieser Chatbot soll interessierten Person einfach und schnell Antworten über die Schule liefern können und somit den Besuchern ein herumsuchen auf der Webseite sparen sowie anrufe und nachfragen beim Sekretariat.



Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Zahlreiche Organisationen bauen heutzutage auf eine starke Webpräsenz und wickeln darauf komplexe Produkt- oder Leistungsfunktionalitäten ab. Eine wichtige Rolle spielt dabei nicht nur der Kundenservice sondern auch die Kommunikation, denn die Nutzer erwarten eine möglichst zeitnahe Beratung. Die Bereitstellung von solchen Services ist derzeit mit hohen Kosten verbunden.

1.2 Istanzustand

Es gibt viele Personen die an der HTL Leonding interessiert sind und sich gerne schnell über die Schule informieren wollen.

1.3 Problemstellung

Auf der HTL Leonding Webseite muss jeder Besucher seine Informationen auf sehr vielen verschiedenen Unterseiten suchen oder beim Sekretariat anrufen.

1.4 Aufgabenstellung

Es soll ein Chatbot für die HTL Leonding Webseite erstellt werden, der Besuchern der Webseite einfach und schnell schulspezifische Fragen beantworten soll.

2 Technischer Hintergrund

2.1 Maschinelles Lernen

2.2 Deep Learning

2.3 Neuronale Netze

2.3.1 Convolutional Neural Networks

2.3.2 Recurrent Neural Networks

2.4 Word Vectors

2.5 Text Analysis

2.6 Natural Language Processing

3 Toolstack

3.1 Programmiersprachen

3.1.1 Java



Abbildung 1: Java Logo[?]

“Java ist eine objektorientierte Programmiersprache und eine eingetragene Marke des Unternehmens Sun Microsystems, welches 2010 von Oracle aufgekauft wurde. Die Programmiersprache ist ein Bestandteil der Java-Technologie – diese besteht grundsätzlich aus dem Java-Entwicklungswerkzeug (JDK) zum Erstellen von Java-Programmen und der Java-Laufzeitumgebung (JRE) zu deren Ausführung. Die Laufzeitumgebung selbst umfasst die virtuelle Maschine (JVM) und die mitgelieferten Bibliotheken. Java als Programmiersprache sollte nicht mit der Java-Technologie gleichgesetzt werden; Java-Laufzeitumgebungen führen Bytecode aus, der sowohl aus der Programmiersprache Java als auch aus anderen Programmiersprachen wie Groovy, Kotlin und Scala kompiliert werden kann. Im Prinzip könnte jede Programmiersprache als Grundlage für Java-Bytecode genutzt werden, meistens existieren aber keine entsprechenden Bytecode-Compiler.

Die Programmiersprache Java dient innerhalb der Java-Technologie vor allem zum Formulieren von Programmen. Diese liegen zunächst als reiner, menschenverständlicher

Text vor, dem sogenannten Quellcode. Dieser Quellcode ist nicht direkt ausführbar; erst der Java-Compiler, der Teil des Entwicklungswerkzeugs ist, übersetzt ihn in den maschinenverständlichen Java-Bytecode. Die Maschine, die diesen Bytecode ausführt, ist jedoch typischerweise virtuell – das heißt, der Code wird meist nicht direkt durch Hardware (etwa einen Mikroprozessor) ausgeführt, sondern durch entsprechende Software auf der Zielplattform.

Zweck dieser Virtualisierung ist Plattformunabhängigkeit: Das Programm soll ohne weitere Änderung auf jeder Rechnerarchitektur laufen können, wenn dort eine passende Laufzeitumgebung installiert ist. Oracle selbst bietet Laufzeitumgebungen für die Betriebssysteme Linux, macOS, Solaris und Windows an. Andere Hersteller lassen eigene Java-Laufzeitumgebungen für ihre Plattform zertifizieren. Auch in Autos, HiFi-Anlagen und anderen elektronischen Geräten wird Java verwendet.

Um die Ausführungsgeschwindigkeit zu erhöhen, werden Konzepte wie die Just-in-time-Kompilierung und die Hotspot-Optimierung verwendet. In Bezug auf den eigentlichen Ausführungsvorgang kann die JVM den Bytecode also interpretieren, ihn bei Bedarf jedoch auch kompilieren und optimieren.”[?]

3.1.2 Python

3.1.3 Typescript

3.2 Technologien

3.2.1 Rasa

3.2.2 Angular

3.2.3 REST Service

3.3 Werkzeuge

3.3.1 IntelliJ IDEA

3.3.2 GitHub

3.3.3 Docker

4 Chatbots am Beispiel von Rasa

4.1 Allgemeines

4.1.1 Rasa Produkte

Der Rasa Stack wird in Rasa NLU und Rasa Core aufgeteilt. Diese sind so aufgebaut, dass sie unabhängig voneinander eingesetzt werden können. So besteht die Möglichkeit, nur einen Teil der Architektur auf Rasa aufzubauen und zusätzlich weitere Services mit einzubinden.

4.1.2 Rasa Core

Rasa Core ist verantwortlich für den Conversation Flow, Context-Handling, Bot-Responses und das Session Management. Dabei kann auf der Rasa NLU oder anderen Services aufgebaut werden, die die Intent Recognition und Entity Extraction übernehmen und die Ergebnisse dem Rasa Core zur Verfügung stellen.[?]

Rasa Core bezieht sich dabei auf die Hauptkomponente, die die Nachrichten erhält und darauf antwortet.[?]

Der Rasa Core hält für jede Session, also für jeden User, einen Tracker. Dieser enthält den momentanen Zustand der Konversationen, der jeweiligen User. Bekommt der Bot nun eine Nachricht, wird zuerst der Interpreter durchlaufen, welcher den Originaltext als Eingabe bekommt, und die Eingabe, den Intent und die extrahierten Entities zurückgibt. Zusammen mit dem aktuellen Zustand des Trackers entscheidet die Policy Komponente nun, welche Action, also Antwort des Bots, als nächstes ausgeführt werden soll. Diese Entscheidung wird nicht durch einfache Regeln getroffen, sondern genauso wie Intents oder Entities, auf der Grundlage von einem, mit Machine Learning, trainierten Model.[?, ?]

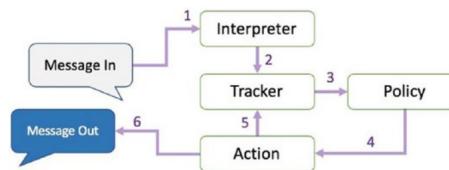


Abbildung 2: Rasa Core Aufbau [?]

4.1.3 Policies

Policies sind ein Teil von **Rasa Core** und der Assistent benutzt Policies um zu entscheiden, welche Action als nächstes ausgeführt werden soll. Es gibt machine-learning und rule-based policies.[?]

Hierbei kann man die Policies beispielsweise ändern. Dies macht man in der **config.yml** Datei.

Bei den Policies gibt es unterschiedliche Priorities, die dann zum Einsatz kommen, wenn mehrere Policies dieselbe Confidence vorhergesagt haben.[?]

TED Policy

Die TED Policy steht für Transformer Embedding Dialogue Policy und wird meistens standardmäßig verwendet.[?]

Bei jedem Dialog bekommt die TED Policy drei Informationen als Input. Die Nachricht des Users, die vorherige Action die vorhergesagt wurde und Slots und aktive Forms. Dann werden diese in den Dialogue Transformer Encoder gepackt und anschließend werden sogenannte Dense Layer verwendet. Danach wird die Ähnlichkeit zwischen den System Actions und dem Dialogue Embedding berechnet und zum Schluss werden noch CRF Algorithmen verwendet, um Entities zu erkennen.[?]

4.1.4 Rasa NLU

Rasa NLU hat grundsätzlich zwei Hauptaufgaben.

Zum einen wäre da die Intent Recognition und die Entity Recognition.[?, ?]

Die Intent Recognition, ist die Erkennung der Nutzer-Absichten. Dazu muss die NLU mit ausreichend Utterances, also Responses trainiert werden. Dabei gibt die NLU alle zugehörigen Intents geordnet nach dem Confidence Score zurück. Rasa verfügt demnach über ein Multi Intent Matching.[?, ?]

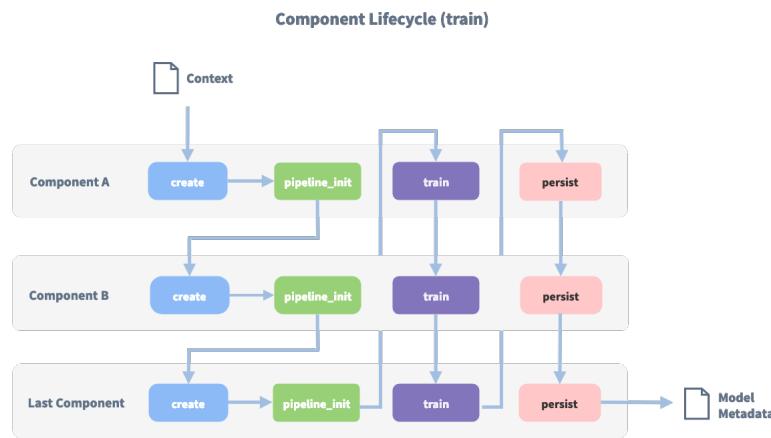


Abbildung 3: Component Lifecycle [?]

Außerdem gibt es noch die Entity Recognition, die dafür zuständig ist Entities, also wichtige Informationen, aus natürlicher Sprache zu extrahieren.[?]

Der Aufbau der NLU ist vollständig konfigurierbar mithilfe der sogenannten Pipeline.[?]

4.2 Pipeline

Rasa Open Source bietet bei der Initialisierung des Projekts eine Standard-NLU-Konfiguration.[?]

In Rasa Open Source werden eingehende Nachrichten von einer Reihe von Komponenten verarbeitet. Diese Komponenten werden nacheinander in einer sogenannten Processing Pipeline ausgeführt, die im config.yml File definiert ist. Wenn man eine NLU-Pipeline auswählt, kann man allerdings sein Model anpassen und an das Dataset verfeinern.[?]

In der Abbildung 3 werden die Komponenten und ihre Lifecycle abgebildet.

Bevor die erste Komponente mit der Create-Funktion erstellt wird, wird ein sogenannter Kontext erstellt (der nichts anderes als ein Python-dict ist). Dieser Kontext wird verwendet, um Informationen zwischen den Komponenten zu übergeben. Beispielsweise kann eine Komponente Merkmalsvektoren für die Trainingsdaten berechnen, diese im Kontext speichern und eine andere Komponente kann diese Merkmalsvektoren aus dem Kontext abrufen und eine Intent Klassifikation durchführen.[?, ?]

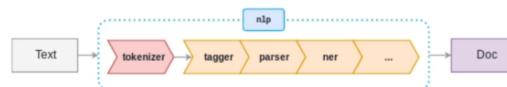


Abbildung 4: Component Lifecycle [?]

Zunächst wird der Kontext mit allen Konfigurationswerten gefüllt, die Pfeile im Bild zeigen die Aufrufreihenfolge und visualisieren den Pfad des übergebenen Kontexts. Nachdem alle Komponenten trainiert und beibehalten wurden, wird das finale context dictionary verwendet, um die Metadaten des Models beizubehalten.[?, ?]

Dies ist in der Grafik 4 zu erkennen.

4.2.1 Arten von NLU Pipelines

Es gibt verschiedene bereits konfigurierte Pipelines.[?]

Grundsätzlich ist zwischen Pipelines zu unterscheiden, indem man sich informiert, ob sie pre-trained word vectors verwenden oder nicht.

Beispiele für Pipelines mit pre-trained word vectors

Der Vorteil von Pipelines, die pre-trained word vectors verwendet ist, dass diese bereits aus der jeweiligen Sprache word vectors besitzen. Somit weiß das Model beispielsweise, dass Äpfel und Birnen ähnlich sind ohne, dass dies in den Intents irgendwo spezifiziert werden muss.[?, ?]

Die Vorteile hierbei sind also, dass das Model weniger Trainingsdaten benötigt, um eine gute Performance zu besitzen. Außerdem geht der ganze Trainingsprozess in der Regel schneller und die sogenannten Iteration Times sind kürzer als bei Modellen ohne pre-trained word vectors.[?, ?, ?]

```

1 language: "de"
2
3 pipeline: "spacy_sklearn"

```

Die SpaCy Pipeline verwendet pre-trained word vectors von GloVe oder fastText.[?, ?, ?]

Es gibt außerdem auch noch Pipelines von MITIE. Diese verwendet MITIE als Source für die word vectors. Ein Vorteil von MITIE ist, dass man hier auch seine eigenen word vectors trainieren kann, indem man einen Corpus von Wikipedia oder ähnlichen Seiten

verwendet. Allerdings wird MITIE meistens nicht empfohlen und es könnte auch sein, dass MITIE demnächst deprecated sein wird.[?, ?]

```
1 language: "de"
2
3 pipeline: "mitie_sklearn"
```

Beispiele für Pipelines ohne pre-trained word vectors

Der Vorteil von Pipelines ohne pre-trained word vectors ist, dass diese speziell auf den Fachbereich angepasst sind, für den man den Chatbot entwickelt.[?, ?]

Als Beispiel kann man die Wörter "balance" und "symmetry" aus dem Englischen sehen. Diese Wörter sind eng miteinander verwandt. Allerdings kann im Kontext von Banken das Wort "balance" auch mit "cash" verwandt sein. Bei einem pre-trained Model würden diese Wortvektoren nicht nah aneinander liegen, aber wenn man einen Chatbot hat der nur Intents besitzt, die mit Banken und Rechnungswesen zu tun haben, werden diese zwei Wörter "balance" und "cash" ohne pre-trained word vectors als ähnlich erkannt werden.[?]

Außerdem benutzen diese Pipelines kein sprach-spezifisches Model und somit kann man sie in allen Sprachen verwenden, die tokenisiert werden kann.[?]

```
1 language: "de"
2
3 pipeline: "tensorflow_embedding"
```

Der Bag-of-word-vectors Ansatz ist zwar sehr gut aber leider auch nicht perfekt. Ein Problem davon ist, dass er oftmals keine fachspezifischen Begriffe kennt und außerdem können Typos nicht als Wortvektoren gelernt werden. Außerdem ist das Problem bei pre-trained word vectors, dass zehntausende Vektoren gespeichert werden, die vermutlich nie verwendet werden.[?, ?]

Mit dem Tensorflow-Embedding macht man im Grunde genommen genau das Gegenteil. Diese Pipeline verwendet keine pre-trained vectors und sollte mit jeder Sprache verwendet werden können. Diese Pipeline lernt Embeddings für die Intents und für die Wörter und die Embeddings werden verwendet um die Ähnlichkeit zwischen dem Input und den Intents zu ermitteln.[?, ?]

Eine weitere Pipeline lautet wie folgt:

```
1 language: "de"
2
3 pipeline: "supervised_embedding"
```

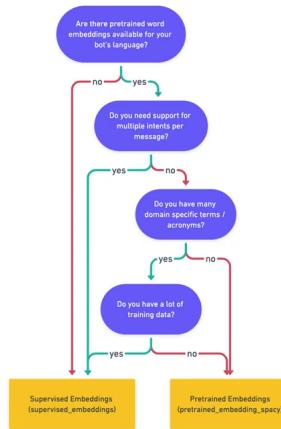


Abbildung 5: Pretrained oder Supervised Embeddings [?, ?]

Supervised Embeddings vs. Pretrained Embeddings

In der Grafik 5 sieht man ein Flussdiagramm, welches beschreibt, ob man Supervised Embeddings oder Pretrained Embeddings verwenden soll.

4.2.2 WhitespaceTokenizer

4.2.3 RegexFeaturizer

4.2.4 CRFEntityExtractor

4.2.5 DucklingHttpExtractor

4.2.6 LexicalSyntacticFeaturizer

4.2.7 CountVectorsFeaturizer

4.2.8 DIETClassifier

4.2.9 EntitySynonymMapper

4.2.10 ResponseSelector

4.2.11 FallbackClassifier

4.3 Welche Rolle spielen neuronale Netze in Rasa

Rasa selbst definiert 5 Stufen von AI.[?]

Level 1: Notifications



Abbildung 6: 5 Stufen von AI [?]

Auf Level 1 geht es darum simple Aufgaben zu erfüllen, wie man sie möglicherweise von seinem Smartphone kennt. Darunter fällt zum Beispiel das Anlegen von Terminen und anderen Benachrichtigungen, die man dann zu der eingestellten Uhrzeit bekommt.[?, ?, ?]

Level 2: "Chatbots" & FAQs

Auf Level 2 geht es darum, dass der Benutzer einfache Fragen stellen kann und anschließend der Chatbot auf diese antwortet. Diese Stufe findet man sehr oft, allerdings sind diese sehr fehleranfällig, weil hier nur eine Menge an Regeln angelegt wird, an die sich der Chatbot hält. Dies wird sehr oft in Form von FAQs genutzt und auch ein paar kleine Follow-up Fragen können hierbei schon definiert sein.[?, ?, ?]

Level 3: Contextual AI Assistants

Dieses Level wird derzeit von Rasa unterstützt. Zusätzlich zu normalen FAQs wird hierbei auch der Kontext beachtet. Es macht nämlich einen Unterschied, wann, wie und in welcher Situation ein Benutzer etwas gesagt hat. Bei Contextual Assistants wird der Kontext, also was bereits zuvor gesagt wurde, auch beachtet. Außerdem lenken sie Konversationen in die gewünschte Richtung und werden mit der Zeit immer besser, wenn man sie trainiert.[?, ?, ?]

4.4 Komponenten

In der sogenannten Domain werden alle Intents, Entities, Slots, Responses, Forms und Actions angegeben, die der Bot kennt. Diese ganzen Informationen befinden sich in der **domain.yml** Datei.[?]

4.4.1 Intents

Intents sind die Absichten hinter der Nachricht des Benutzers. Als Intents werden also alle möglichen Beispielsätze definiert, die ein Benutzer sagen könnte, um eine bestimmte Absicht auszudrücken.[?]

Intents werden in dem **nlu.yml** File wie folgt angegeben:

```

1  ## intent:<name des intents>
2  - <phrase 1>
3  - <phrase 2>
4  - <phrase 3>
```

4.4.2 Responses

Responses sind die Antworten, die vom Bot gegeben werden, wenn ein bestimmter Intent erkannt wurde.[?]

Responses fügt man in seinem **domain.yml** File wie folgt ein:

```

1  responses:
2    utter_greet:
3      - text: "Hey! How are you?"
4
5    utter_<name der response>:
6      - text: "<text>"
7      image: "<img link>"
8      ...
```

4.4.3 Stories

Stories werden als Trainingsdaten verwendet, die zum Trainieren des Models des Bots verwendet werden. Stories können dabei genutzt werden, um Models zu trainieren, bei denen auch unvorhersehbare Konversationspfade behandelt werden und unterscheiden sich in dieser Hinsicht von den Rules. [?]

Bei einer Story wird also die Unterhaltung zwischen einem Benutzer und dem Bot dargestellt. Dabei wird die Eingabe des Benutzers als Intent angegeben und die Antwort, mit der der Bot antworten soll, als Name der Action. [?]

Stories können wie folgt aussehen und sind im **stories.yml** File anzugeben:

```

1  stories:
2  - story: name der story
3    steps:
4      - intent: <name des intents>
5      - action: <name der action>
```

Checkpoints und OR-Statements

Man kann seine Stories außerdem mit Checkpoints und OR-Statements versehen. Bei diesen sollte man aber grundsätzlich aufpassen und sie nur bedacht verwenden, weil in den meisten Fällen die gewünschten Resultate besser mit **Rules** oder einem **ResponseSelector** zu erzielen sind. [?]

Checkpoints können genutzt werden, um seine Trainingsdaten zu vereinfachen, indem man einen Checkpoint in einer Story setzt und auf diesen in einer anderen Story wieder ansetzt. Von diesen sollte man allerdings nicht zu viele machen, weil sonst die Stories sehr leicht schwer zu lesen und unübersichtlich sind und außerdem die Trainingszeit dadurch erhöht wird. [?]

Man definiert einen Checkpoint am Ende seiner Story, wenn man diesen Teil der Story auch wieder als Voraussetzung für eine weitere Story setzt. In der nächsten Story beginnt man dann mit seinem Checkpoint, also dem Punkt auf den man anknüpfen möchte und die Teile der Konversation, die für diese Story ebenfalls vorausgesetzt werden sollen.

Im folgenden Beispiel werden Checkpoints von Stories verwendet, um an anderen Stories anzuknüpfen[?] :

```

1 stories:
2 - story: beginning_of_flow
3   steps:
4     - intent: greet
5     - action: action_ask_user_question
6     - checkpoint: check_asked_question
7
8   - story: handle_user_affirm
9     steps:
10    - checkpoint: check_asked_question
11    - intent: affirm
12    - action: action_handle_affirmation
13    - checkpoint: check_flow_finished
14
15  - story: handle_user_deny
16    steps:
17    - checkpoint: check_asked_question
18    - intent: deny
19    - action: action_handle_denial
20    - checkpoint: check_flow_finished
21
22  - story: finish_flow
23    steps:
24    - checkpoint: check_flow_finished
25    - intent: goodbye
26    - action: utter_goodbye

```

OR-Statements können dafür verwendet werden, wenn man auf mehrere Intents innerhalb einer Story gleich reagieren möchte. [?]

Man schreibt also anstelle von einem Intent in der Story ein **or** und gibt darunter alle Intents an, von denen einer eintreffen muss, damit die Story zutrifft.

```
1 stories:
```

```

2   - story:
3     steps:
4       # ... vorherige schritte
5       - action: utter_ask_confirm
6       - or:
7         - intent: affirm
8         - intent: thankyou
9         - action: action_handle_affirmation

```

4.4.4 Rules

Rules werden angegeben, um kleine Teile von Unterhaltungen anzugeben, die immer wieder gleich behandelt werden sollen. Diese sollten allerdings nicht allzu häufig verwendet werden, weil man nie alle Konversationen vorhersagen kann. Um Rules verwenden zu können, muss man die **RulePolicy** in der Policy Konfiguration eintragen. [?]

Um eine Rule zu verwenden, schreibt man folgendes in sein **rules.yml** File:

```

1   rules:
2
3   - rule: Say 'hello' whenever the user sends a message with intent 'greet'
4     steps:
5       - intent: greet
6       - action: utter_greet

```

4.4.5 Slots

Slots sind sozusagen das Gedächtnis des Bots. Diese sind als key-value Paare dargestellt und können dazu verwendet werden, damit Information, die der Benutzer bereitstellt, gespeichert werden können, ähnlich zu Entities. Diese Informationen können beispielsweise der Name des Benutzers sein oder Informationen, die für den generellen Kontext des Gesprächs wichtig sind. [?]

```

1   slots:
2     slot_name: <slot name>
3     type: <type>

```

4.4.6 Entities

Entities sind strukturierte Stücke von Informationen, die sich innerhalb der Nachricht eines Benutzers befinden. Solche Entities können beispielsweise ein Ort, ein Beruf oder ein Name sein.[?]

Um Entities zu erstellen schreibt man folgendes in sein **domain.yml** File:

```

1   entities:
2     - <entity name>
3     - <entity name>

```

Diese Entities müssen dann noch in den Intents angegeben werden, in denen sie vorkommen sollen. Dies macht man, indem man folgende Syntax bei den Trainingssätzen im **nlu.yml** File verwendet und ergänzt:

```
1 Hallo mein Name ist [Lukas](name).
2 Ich h tte gerne eine [gro e](size) [Pizza](meal)
```

Entity Roles

Entity Roles können sinnvoll in manchen Szenarien sein. Zum Beispiel bei folgendem Satz:

```
1 Buche einen Flug von [Linz](city) nach [London](city).
```

In diesem Fall sind sowohl Linz als auch London zwar richtig gekennzeichnet als city Entity, allerdings reicht diese Information noch nicht aus, damit der Chatbot richtig reagieren kann. Hierbei wäre es praktisch, wenn man noch angibt, welche dieser zwei Städte das Ziel und welche der Abflugsort ist. Dies macht man mit Entity Roles.[?]

Entity Groups

Entity Groups können genutzt werden, wenn man Entities miteinander gruppieren möchte.[?]

Dies kann zum Beispiel hier sinnvoll sein:

```
1 Ich h tte gerne eine gro e [Pizza](meal) mit [Pilzen](topping) und eine
[Salami](topping) [Pizza](meal).
```

Bei der Gruppe muss hier erkannt werden, welche zwei Entities zusammen gehören[?]:

```
1 Ich h tte gerne eine gro e [Pizza](meal) mit [Pilzen](topping) und eine
[Salami](topping) [Pizza](meal).
2 Group 1: [Pizza](meal) [Pilzen](topping)
3 Group 2: [Salami](topping) [Pizza](meal)
```

Nutzung von Entity Roles und Entity Groups

4.4.7 Actions

Es gibt 2 verschiedene Arten von Messages:

1. **Static Messages:** Diese sind unabhängig vom User Input und benötigen keinen Action Server[?]
2. **Dynamic Messages:** Diese sind abhängig vom User Input und benötigen einen Action Server[?]

Der Rasa Action Server führt sogenannte Custom Actions für einen Rasa Open Source Conversation assistent aus.

Wenn der Assistant eine gewisse Custom Action vorhersagt, sendet der Rasa Server einen POST request an den Actionserver mit einer JSON Payload mit dem Namen der vorhergesagten Action, der Conversation ID, den Inhalten des Trackers und den Inhalten der Domain.[?]

4.4.8 Forms

Um mehrere Informationen von einem Benutzer zu bekommen, eignen sich Forms. Um Forms zu verwenden, muss die **RulePolicy** in der Policy Konfiguration eingetragen sein.[?]

Wenn man ein Formular hinzuzufügen will, muss man dies in der forms Section in dem **domain.yml** File angeben.

```

1   forms:
2     restaurant_form:
3       required_slots:
4         cuisine:
5           - type: from_entity
6             entity: cuisine
7       num_people:
8         - type: from_entity
9           entity: number

```

4.4.9 Synonyms

Mithilfe von Synonymen kann man extrahierten Entities einen anderen Wert geben, als sie eigentlich vorher hatten, wenn diese in der Bedeutung gleich sind. Wenn man also mit verschiedenen Wörtern dasselbe meint, kann man sich Synonyms zur Hilfe nehmen.[?]

Ein Beispiel dafür wäre folgendes im **nlu.yml** File:

```

1   - synonym: Medientechnik
2     examples: |
3       - IT-Medientechnik
4       - IT Medientechnik
5       - Medientechnologie

```

4.5 Initialisieren

4.6 Trainieren

5 Implementierung

5.1 2D-Repräsentation

5.1.1 Frontend

Der Chatbot der HTL Leonding sollte auf der Schulhomepage als Chatblase angezeigt werden, und verschiedene Elemente wie Buttons und Links unterstützen.

Konzept

Während den Anfängen der Diplomarbeit wurde ein Konzept erstellt, um das mögliche Aussehen festzulegen. Lange Zeit wurde der Chatbot unter den Namen Leon geführt. Dies wurde jedoch im späteren Verlauf geändert und Leon wurde Teil des langjährigen Leonie Projektes der HTL Leonding. Ursprünglich war das Symbol des Chatbots, wie man am Konzept sehen kann, ein Bot. Durch den Wechsel in die Leonie Familie wurde dieses Logo jedoch gegen Leonie getauscht.

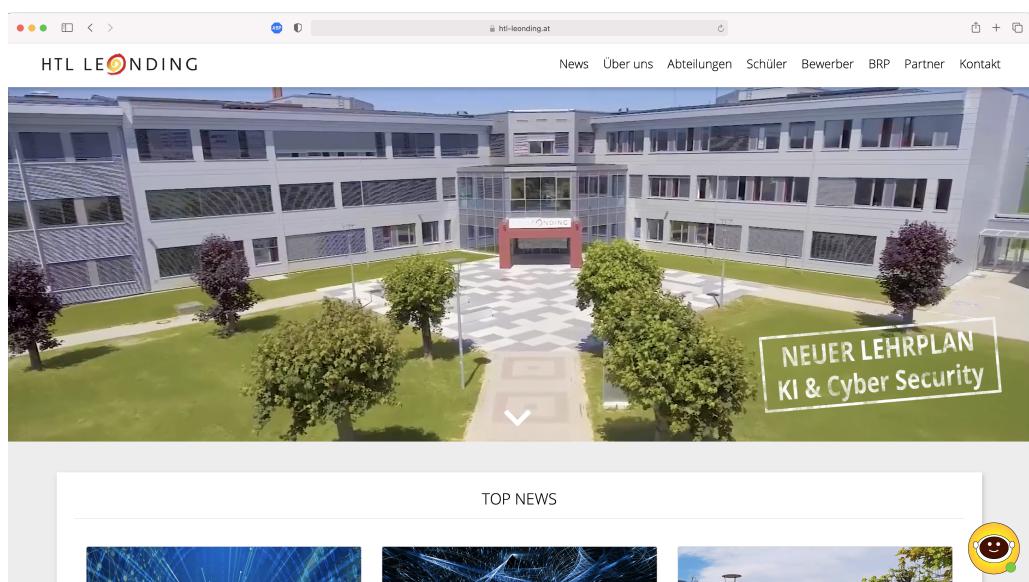


Abbildung 7: Konzept Chatbot geschlossen

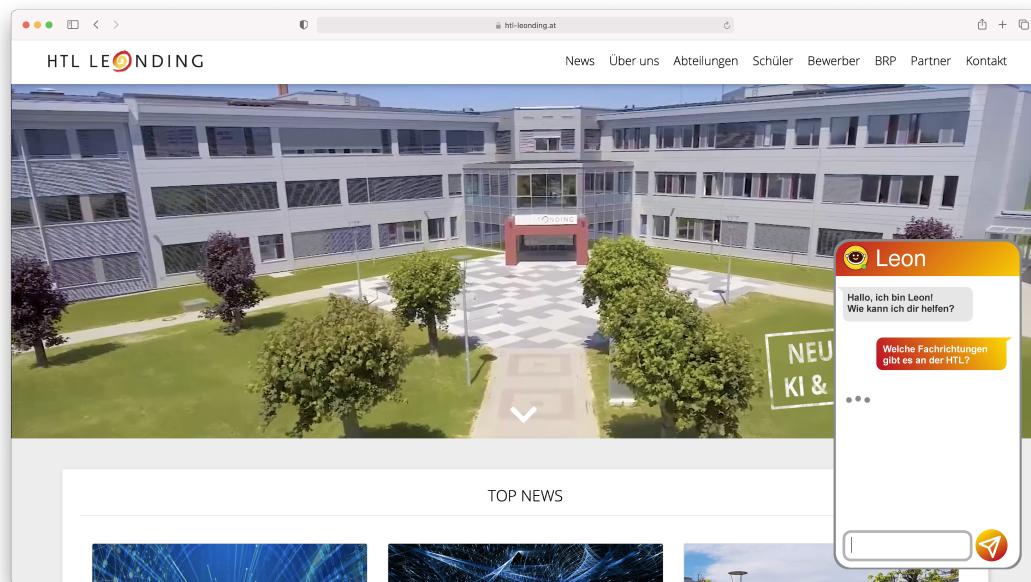


Abbildung 8: Konzept Chatbot geöffnet

Umsetzung

Umgesetzt wurde das Frontend mit Hilfe von Angular. Die Chatblase ist eine eigene Komponente, die durch CSS immer rechts unten fixiert ist. Die Farben des Chatbots sollten natürlich an die HTL Leonding erinnern, deshalb wurde ein Farbverlauf aus Farben des HTL Logos erstellt.

Jedoch begann der Chatbot sehr anders, zu Beginn wurde der Bot zuerst als ganze Seite entwickelt und nicht nur als Chatblase.

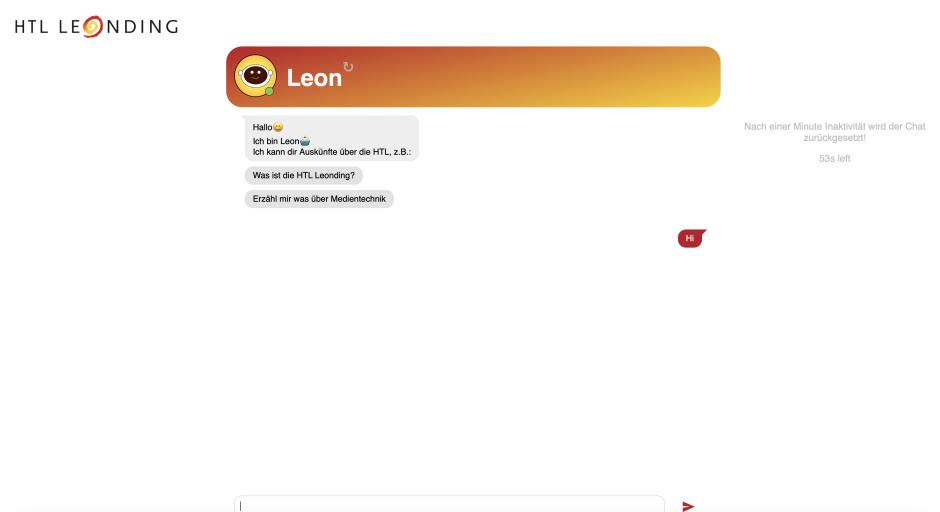


Abbildung 9: Chatbot auf einer ganzen Seite

Natürlich war dies nicht das entgültige Ziel so wurde der Bot schnell zur Chatblase umgewandelt. Zum testen war die HTL Leonding Seite mithilfe eines IFrame eingebunden und zusätzlich Chatblase.

Um das Gespräch in eine Richtung zu lenken wurden Buttons eingeführt, die nach fast jeder Antwort mögliche folge Fragen vorschlagen.

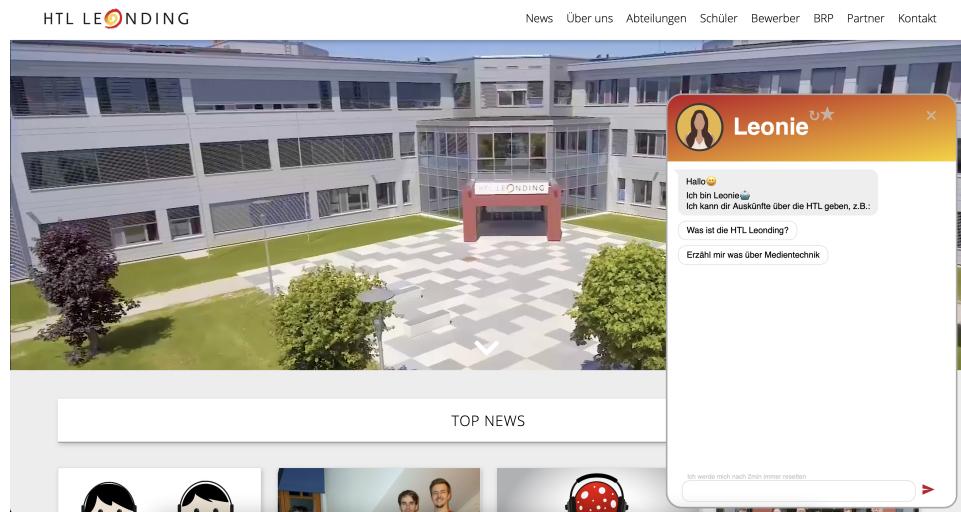


Abbildung 10: Chatbot

Um Bewertungen von echten Benutzern zu holen wurde außerdem eine Feedback Seite eingeführt, in dieser kann der Benutzer eine 1 bis 5 Sterne bewertung und einen Text absenden.

5.1.2 Dashboard

Um alle Gespräche und die Bewertungen der Benutzer anzuzeigen wurde ein Dashboard eingeführt, wo nur dies möglich war. Im Laufe der Diplomarbeit wurde dieses dann erweitert, um für den Leobot Conversation Cycle als Seite zu dienen.

Einerseits kann man sich die vergangenen Unterhaltungen ansehen, so wie die nlu.yml, stories.yml, rules.yml, domain.yml direkt im Monaco Editor bearbeiten und speichern.

5.1.3 Einbindung in Wordpress

Angular Elements

5.2 Schwierigkeiten

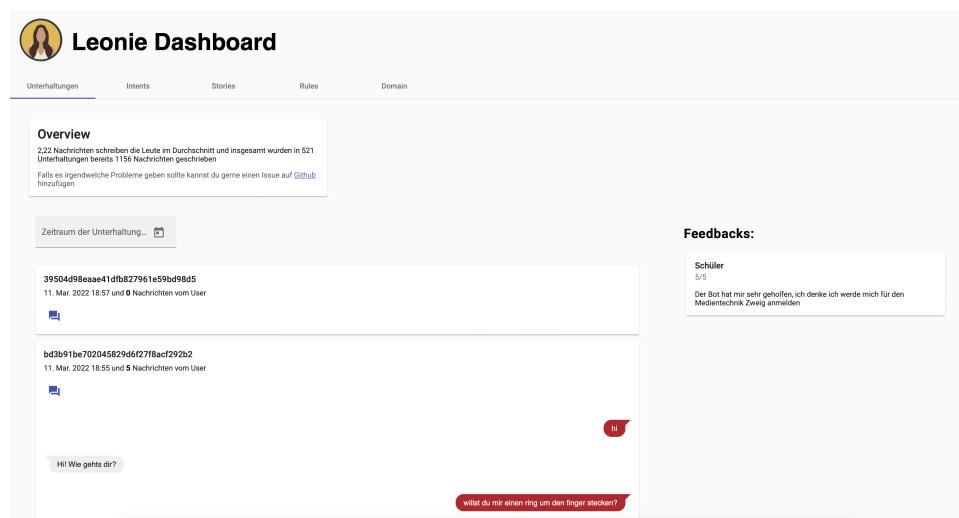


Abbildung 11: Dashboard

6 Evaluation

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Anhang