FOM Hochschule für Oekonomie & Management

Hochschulzentrum Siegen

Seminararbeit

im Rahmen der Lehrveranstaltung
IT-Infrastruktur

über das Thema

Dialogbasierter Zugriff auf eine KI über eine Django REST API

von

Henning Beier

Betreuer : Daniel Bitzer
Matrikelnummer : 517370
Abgabedatum : 30. August 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
HTTP/S	Hypertext Transfer Protocol (Secure)
KI	Künstliche-Intelligenz
REST	Representational State Transfer
UUID	Universally Unique Identifier

1 Einleitung und Zielsetzung

In dieser Seminararbeit soll eine Künstliche-Intelligenz (KI)-Anwendung über eine *Representational State Transfer (REST) Application Programming Interface (API)* aufrufbar gemacht werden. Eine REST-API ist eine Programmierschnittstelle mit einem Architekturansatz, der speziell für Webservices entwickelt und konzipiert wurde. Eine KI-Anwendung muss dazu in einer Art eingebunden werden, die es erlaubt, auf diese selbst in geeigneter Weise über die Endpunkte einer API zuzugreifen. Das bedeutet, dass die KI-Anwendung als Webservice zur Verfügung gestellt wird und damit auch einen dialogbasierten Zugriff auf die KI-Anwendung und deren Funktionen selbst ermöglicht.

Verwendet werden dazu soll Django, ein quelloffenes Web-Framework, das weit verbreitet im produktiven Einsatz ist und ebenso wie viele KI-Anwendungen in der Programmiersprache Python geschrieben wurde. Dieses Zusammenführen und Bereitstellen als eine Lösung im Softwarebereich, bzw. hier als Dienst, stellt dabei einen typischen Arbeitsbereich der IT-Infrastruktur dar. Die KI-Anwendung muss also in eine geeignete IT-Infrastruktur eingebunden werden. Ein sicherer und belastbarer Webservice ist das angestrebte Ziel dieser Seminararbeit.

Die Seminararbeit mit allen Anlagen, dem Programmcode, allen Skripten und Quellen wird auch auf Github.com unter `https://github.com/haenno/ai-api/` zur Verfügung gestellt, um den Zugriff darauf zu erleichtern. Ebenso findet sich unter `https://ai-api.tstsrv.de/` eine laufende Instanz der fertigen Lösung zum einfachen Ausprobieren.

1.1 Praktische Relevanz und Problemstellung

Aktuell werden in vielen Bereichen neue Projekte und Lösungen mit KI entworfen, entwickelt und getestet. Dazu werden KI-Frameworks wie TensorFlow, PyTorch oder Keras benutzt. Für die vorgenannten, aber ebenso wie auch für andere KI-Frameworks hat sich Python als Programmiersprache durchgesetzt. Dies ist darin begründet, da sich mit Python große Datenmengen vergleichsweise einfach aufnehmen, verarbeiten und ausgeben lassen. Große Datenmengen sind für die KI-Entwicklung, häufig z. B. als Trainingsdatensätze, unverzichtbar.

Entwürfe und Testentwicklungen von KI-Anwendungen erfolgen insbesondere in frühen Phasen meist lokal in einer Entwicklungsumgebung. Ergebnisse zu präsentieren oder ein Austausch mit Dritten darüber ist damit nicht immer einfach möglich. Zumeist müssten

Dritte erst einen Python-Interpreter sowie umfangreiche weitere Abhängigkeiten, wie das verwendete KI-Framework selbst, installieren, um die KI-Anwendung überhaupt erst ausführen zu können. Das fällt nicht jedem Interessierten leicht. Auch gibt es durch die dynamische Entwicklung meist verschiedene Versionen der Software und Komponenten, die zu unbeabsichtigten und unerwarteten Problemen und Hürden führen können.

1.2 Methodik

Die Seminararbeit soll das gewünschte Ziel durch exploratives, vertikales Prototyping erreichen. Vertikal bedeutet dabei, dass von der einfachen Konsolen-KI-Anwendung bis hin zum dialogbasierten Aufruf über den Webserver alle Ebenen durchdrungen werden sollen. Dabei sollen alle Schritte, jede eingesetzte Software und Lösung gründlich betrachtet und alle Schritte und Entscheidungen nachvollziehbar dokumentiert werden.

Zuerst soll eine einfache KI-Anwendung, die als Konsolenanwendung lokal in einer Entwicklungsumgebung lauffähig ist, erstellt werden. Hier ist ein üblicher Prototyp einer KI-Anwendung, z. B. aus einem einfachen Tutorial, aus. Zum Erreichen des Ziels reicht es dabei aus, wenn nur wenige einfache Funktionen des verwendeten KI-Frameworks zum Einsatz kommen.

Im Anschluss daran soll ermittelt werden, auf welche Arten die Einbindung einer solchen KI-Anwendung in das Django-Framework möglich ist. Mögliche Lösungsansätze sollen ermittelt, betrachtet und bewertet werden.

Die schrittweise Einrichtung einer Django-Installation wird analog durchgeführt und alle gemachten Einstellungen, eingerichteten Erweiterungen und Anpassungen werden nachvollziehbar dokumentiert. Dies umfasst dabei auch die notwendigen Module für Bereitstellung der API sowie den Einsatz von Django in einem produktiven Umfeld.

Im Schlussschritt werden die Django-Installation und die für die Einbindung vorbereitete KI-Anwendung zusammengeführt. Der dann mögliche, dialogbasierte Zugriff auf die KI-Anwendung als Webservice, die Django-REST-API, soll durch den Einsatz eines dynamischen Frontends belegt werden.

2 Theorie: Begrifflichkeiten und Grundlagen

In diesem Abschnitt werden einige für das weitere Verständnis hilfreiche Begrifflichkeiten und Grundlagen kurz erklärt und eingeordnet. Dabei wird als Zielgruppe der Seminararbeit eine zumindest in IT-Grundlagen versierte Leserschaft angenommen.

2.1 REST-API

Die Abkürzung API bedeutet „Application Programming Interface“, ins Deutsche übersetzt also „Schnittstelle zur Programmierung von Anwendungen“. Eine API ist damit eine Schnittstelle zu einem Programm oder zu einer Programmierung und erlaubt meist anhand detaillierter Dokumentation den Zugriff auf die bzw. die Verarbeitung von Daten aus der Schnittstelle. Im heutigen Sprachgebrauch ist mit API meist der Zugriff auf einen Webservice gemeint. Grundsätzlich aber umfasst der Begriff API nicht nur die vorgenannten Webservices, die eine protokollorientierte Schnittstelle sind, sondern auch API anderer Typenklassen wie datei-, funktions- oder objektorientierte APIs.

Mit REST beschreibt Fielding im Jahr 2000 in seiner Arbeit *Architectural styles and the design of network-based software architectures* einen hybriden Architekturansatz aus der Vermischung verschiedener netzwerkbasierter Architekturansätze für Schnittstellen. Dieser Ansatz hat sich zwischenzeitlich durchgesetzt und wird vielfach in modernen Webservices verwendet. Fielding beschreibt in seiner Arbeit umfassend, wie eine zustandslose Kommunikation zwischen Client und Server über eine einheitliche Schnittstelle (API) mittels bekannter und bewährter *Hypertext Transfer Protocol (Secure) (HTTP/S)*-Methoden, wie GET, POST, PUT und DELETE, umgesetzt wird.

Heute werden REST-APIs vielfältig eingesetzt: ob zum Datenaustausch zwischen Unternehmen oder auch der Realisierung verschiedener Frontends bzw. Endgeräte für eine Anwendung. Aus der Praxis sind Schnittstellen dieser Art heute nicht mehr wegzudenken.

2.2 KI-Anwendungen und -Frameworks

Die Konrad-Adenauer-Stiftung e. V. (Wangermann, 2020) wie auch das Bundesministerium für Bildung und Forschung, Referat Künstliche Intelligenz (2022) selbst beschreiben in ihren jüngsten Veröffentlichungen in ausführlicher Breite, wie wichtig die Entwicklung von KI in allen Bereichen und auf allen Ebenen ist.

Dabei ist der praktische Einstieg in das Thema KI-Entwicklung im Vergleich zu anderen Forschungsfeldern ungleich leichter. Es werden für einen Start in den Themenbereich nicht viel mehr als ein Rechner und eine Internetverbindung benötigt. Die Entwickler der gängigen bzw. etwas weiter verbreiteten KI-Frameworks stellen umfangreiche Dokumentationen, die anhand praxisnaher Beispiele und nleitungen, den Einstieg erleichtern sollen.

Die KI-Frameworks TensorFlow, Keras und PyTorch haben sich über die letzten Jahre nicht nur auf dem Markt gehalten, sondern wurden von der Entwicklergemeinde deutlich mehr genutzt. Besonders die positive Entwicklung von PyTorch, hier mit einem Zuwachs von 468 % gemessen an Aktivitätsindikatoren auf Github.com, hebt sich hier noch stärker von den beiden anderen, vorgenannten mit jeweils etwa 50 % ab.

Abbildung 1: Entwicklung der KI-Framework-Nutzung auf Github.com

KI-Framework	Forks (10/2018)	Forks (08/2022)	Veränderung in %	Pull Requests (10/2018)	Pull Requests (08/2022)	Veränderung in %	Issues (10/2018)	Issues (08/2022)	Veränderung in %	Ges. (10/2018)	Ges.(08/2022)	Veränderung in %
TensorFlow	68.709	87.100	27%	8.941	21.654	142%	14.037	35.427	152%	91.687	144.181	57%
Keras	13.094	19.200	47%	3.292	5.491	67%	8.133	11.405	40%	24.519	36.096	47%
Microsoft Cognitive Toolkit	4.089	4.400	8%	458	555	21%	2.985	3.268	9%	7.532	8.223	9%
Torch	2.312	2.400	4%	509	511	0%	664	726	9%	3.485	3.637	4%
PyTorch	4.772	16.200	239%	7.282	56.052	670%	5.578	27.925	401%	17.632	100.177	468%
Caffe	15.777	19.000	20%	2.164	2.233	3%	4.401	4.781	9%	22.342	26.014	16%

Hinweise:
- Eigene Zählung am 27.08.2022 auf den in der Spalte URL genannten Seiten.
- Heise, 10/2018, Artikel "Vergleich von Machine-Learning-Frameworks", URL: https://www.heise.de/tests/Vergleich-von-Machine-Learning-Fr
- Forks, Pull Requests und Issues jeweils von Github.com, wo zutreffend Gesamtwerte (offene und geschlossene)

KI-Framework	URL
TensorFlow	https://github.com/tensorflow/tensorflow/
Keras	https://github.com/keras-team/keras
Microsoft Cognitive Toolkit	https://github.com/microsoft/CNTK
Torch	https://github.com/torch/torch7
PyTorch	https://github.com/pytorch/pytorch
Caffe	https://github.com/BVLC/caffe

Quelle: Eigene Darstellung

2.3 Django Web-Framework

Django ist ein quelloffenes und frei verfügbares Web-Framework, das nach eigenen Angaben schnell, vollständig, sicher, skalierbar und vielseitig ist (Django Software Foundation,

2022). Untermauert wird diese eigene Behauptung von Django durch bekannte Anwender und Nutzer wie Instagram, Pinterest und Delivery Hero (StackShare, Inc., 2022). Eine Studie bestätigt Django weiter eine große Gemeinschaft an Entwicklern, eine bedeutsame Anzahl an Fragen und Antworten bei Stackoverflow und Django wird auch als gute Lösung für größere Projekte betrachtet (Ghimire, 2020).

Abbildung 2: Logo des Web-Frameworks Django

Quelle: Django Software Foundation,
https://www.djangoproject.com/community/logos/

Django lässt sich dabei einfach um Funktionen und Pakete erweitern. Der modulare Aufbau mit Datenmodellen, Sichten und Einstiegspunkten lässt zusammen mit dem Template-System und der einfachen Abstraktion für verschiedene Datenbanken sowie einer praktischen Administrationsoberfläche dabei dennoch eine schnelle Entwicklung von Webseiten zu.

3 Praktischer Teil: Vom Skript zum Webservice

Die Entwicklung erfolgt in verschiedenen Phasen, die aufeinander aufbauen. Erkenntnisse aus einer Phase bedingen darauf folgende Schritte. Ein Wechsel zwischen den Bereichen und ein wiederholtes Aufgreifen eines Teils der Lösung ist daher nicht als Zeichen von Unstrukturiertheit zu verstehen, sondern als für den Prozess des hier angewendeten, vertikalen Prototypings schlicht notwendig.

3.1 Vorbereitung einer KI-Anwendung

Als Beispiel für bereitzustellende KI-Anwendung wurde ein Chatbot gewählt. Eine Chatbotanwendung bietet sich an, da die Verarbeitung von menschlicher Sprache ein bedeutendes Forschungsfeld innerhalb der KI ist. Bereits seit Jahrzehnten wird in diesem Bereich intensiv Forschung betrieben: von Mitte der 1960er Jahre an mit z. B. ELIZA, einer Sprachverarbeitung, die im Wesentlichen auf Mustererkennung abzielt und vor allem durch den fehlenden Zugriff auf Informationen beschränkt war (Weizenbaum, 1983), bis hin zu jüngsten Entwicklungen, die auch durch Schlagzeilen in der Presse Aufmerksamkeit erlangten: Microsofts Chatbot Tay funktionierte technisch, wurde aber bereits nach wenigen Tagen wieder abgeschaltet, da sich ihr Charakter in unerwünschte Richtungen entwickelte: Sie verbreitete Hassbotschaften und leugnete den Holocaust (heise online, Andreas Wilkens, 2016). Noch mehr Aufmerksamkeit erlangte vor kurz Kurzem der Chatbot LaMDA von Google. Dies lag daran, da einer der Entwickler, zu der Überzeugung gelangte, dass der Chatbot ein Bewusstsein und sogar eine Seele entwickelt habe (heise online, Martin Holland, 2022).

Für die vorliegende Seminararbeit, wird jedoch nur eine sehr einfache Ausführung eines Chatbots herangezogen. Grundlage dafür ist ein Tutorial in dem der Autor Loeber Grundlagen über den Aufbau einer KI-Anwendung näher bringt (Loeber, 2020).

Speziell dieses Tutorial wurde gewählt, da es als KI-Framework PyTorch verwendet. PyTorch ist neben TensorFlow eines der aktuell führenden KI-Frameworks (siehe auch Abbildung 1), hat jedoch für diesen Anwendungsfall den Vorteil, dass es deutlich weniger Abhängigkeiten hat und kleiner in der Grundinstallation ist.

Zuerst wurde das gesamte Tutorial von mir durchgeführt und die KI-Anwendung lokal nachgebaut und getestet. Hierbei gab es keine nennenswerten Besonderheiten. Das vom Autor aufgebaute KI-Modell in der Datei `model.py` ist in diesem Fall ein künstliches, neuronales Netz mit drei Ebenen:

Abbildung 3: Programmcode: model.py aus Chatbot-Tutorial

```
1  import torch
2  import torch.nn as nn
3
4
5  class NeuralNet(nn.Module):
6      def __init__(self, input_size, hidden_size, num_classes):
7          super(NeuralNet, self).__init__()
8          self.l1 = nn.Linear(input_size, hidden_size)
9          self.l2 = nn.Linear(hidden_size, hidden_size)
10         self.l3 = nn.Linear(hidden_size, num_classes)
11         self.relu = nn.ReLU()
12
13     def forward(self, x):
14         out = self.l1(x)
15         out = self.relu(out)
16         out = self.l2(out)
17         out = self.relu(out)
18         out = self.l3(out)
19         # no activation and no softmax at the end
20         return out
```

Man erkennt hier in Zeile 14 die sichtbare Eingabeebene, folgend eine verborgene Ebene Zeile 16 sowie Ausgabeebene in Zeile 18. Da hier eine verborgene Ebene verwendet wird, spricht man von einem „Deep Learning“-Modell. Es handelt sich bei dieser künstlichen Intelligenz um ein künstliches neuronales Netz (KNN), das mit maschinellem Lernen (ML) ein mehrschichtiges bzw. tiefes Lernen (Deep Learning) durchführt. Mehr und tiefer soll in dieser Seminararbeit jedoch nicht darauf eingegangen werden, da es den Rahmen der Seminararbeit sprengen würde und nicht zum Thema selbst gehört. Deutlich wird aber, dass schon mit einer einfachen KI-Anwendung, also hier dem Einsatz des KI-Frameworks PyTorch, direkt wesentliche und grundlegende Anwendungsfälle und mögliche Anforderungen abgedeckt werden können. Deutlich komplexere Modelle sind hier nun also nur noch eine Frage der tatsächlichen Umsetzung und eine Umsetzung scheitert hier nicht mehr an einer fehlenden IT-Infrastruktur.

Mit Abschluss des Tutorials umfasst die Anwendung nicht nur die Datei `model.py` sondern auch eine `train.py` die es erlaubt, das KI-Modell anhand der Eingabedaten in der Datei `intents.json` zu trainieren. Die Eingabedaten sind dabei wie folgt aufgebaut:

Abbildung 4: Auszug der Eingabedatei für das KI-Modell „intents.json“

```
1  {
2    "intents": [
3      {
4      "tag"      : "Begrüßung",
5      "patterns" : ["Hi", "Hallo", "Guten Tag"],
6      "responses": ["Hallo auch!", "Schön dich zu sehen!", "Willkommen!"]
7      },
8      {
9      "tag"      : "Verabschiedung",
10     "patterns" : ["Tschüss", "Bis bald", "Bye"],
11     "responses": [
12        "Danke für das Gespräch — Tschüss!",
13        "Pass auf das die Tür zu geht. Nodda."
14     ]
15     }
16     (...)
17   ]
18 }
```

An der Struktur dieser Daten kann die Funktionsweise nachvollzogen werden: Die KI-Anwendung überprüft Eingaben und versucht diese anhand der erlernten Muster (Zeilen 5 und 10) einem Thema (Zeilen 4 und 9) zuzuordnen. Gelingt dies mit einer bestimmten Sicherheit, werden Antworten entsprechend dem Thema (Zeilen 6 und 11–13) zurückgegeben.

Der Vollständigkeit halber verweise ich noch kurz auf den Teil der KI-Anwendung, der die in dieser Eingabedatei vorhandene, natürliche Sprache verarbeitet. Bevor diese Daten für das Training und die Verwendung im Gespräch mit dem Chatbot verwendet werden können, müssen diese mittels Algorithmen der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) für die KI aufbereitet werden. Dazu werden die Sprachdaten tokenisiert, d. h. , also die Eingabe wird in Sätze und die Sätze werden in Wörter aufgeteilt. In diesem Tutorial geschieht dies mithilfe des Paketes „nltk“ (Natural Language Toolkit).

In einem nächsten Schritt werden diese Wörter dann auf ihren Wortstamm zurückgeführt. Dies war eine der ersten Stellen, an denen ich das Ergebnis des Tutorials angepasst habe. Ich habe ein entsprechendes Paket für die Verarbeitung deutscher Sprache gesucht und mit „HanTa“ gefunden (Wartena, 2019). Dieser Programmablauf erfolgt nicht nur beim Training des KI-Modells, sondern auch beim Gespräch mit Chatbot über die `chat.py`.

Weitere Anpassungen durch mich in dieser allerersten Entwicklungsstufe der KI-Anwendung direkt nach Abschluss des Tutorials waren das Hinzufügen einer Log-Funktion, die alle Eingaben und Antworten in Log-Dateien (hier als CSV) abspeichert. Ebenso erzeuge ich für jedes geführte Gespräch einen eindeutigen Identifikator (hier als

Universally Unique Identifier (UUID)), der es erlaubt, geführte Gespräche in den Log-Dateien anhand der UUID vollständig nachzuvollziehen.

Alle Einstellungen und Parameter der KI-Anwendung habe ich in der Datei `config.py` zusammengefasst. Abschließend habe ich die für das Starten der KI-Anwendung notwendigen Abhängigkeiten und Pakete für die Verwendung mit dem Python-Tool „Pipenv“ in der Datei „Pipfile“ zusammengefasst.

Der gesamte erste Entwicklungsstand, wie er bis hier beschrieben wurde, ist in den Anlagen zur Seminararbeit sowie auf Github.com im Unterordner `10-example-ai-app` dokumentiert. Die KI-Anwendung in diesem Entwicklungsstand kann ausgeführt und getestet werden, indem der Ordner `10-example-ai-app` mit einem Terminal betreten wird und unten stehende Befehle ausgeführt werden:

```
pipenv install
pipenv run python train.py
pipenv run python chat.py
```

Voraussetzung dafür sind ein installierter Python-Interpreter ab Version 3.9 und das Python-Tool „Pipenv“.

Für die Einbindung dieser KI-Anwendung muss diese nun aber in einer Form vorliegen, die es erlaubt, das Programm oder Programmteile an anderen Stellen einzubinden oder Teile davon aufzurufen. Die bisher vorliegende Datei- bzw. Skriptsammlung ist dazu so nicht nutzbar. Die KI-Anwendung muss daher zu einem Python-Modul umgebaut werden. Ein Python-Modul zeichnet sich dadurch aus, dass die notwendigen Dateien inkl. einer `__init.py__`-Datei in einem entsprechenden Dateiordner liegen. Der Aufruf kann dann aus anderen Python-Skripten mittels `import` **`NameDesModulOrdners`** aufgerufen werden. Programmfunktionen aus dem Modul können dann mit `NameDesModulOrdners.`**`FunktionsName`** aufgerufen und gestartet werden.

Ich habe die notwendigen Anpassungen an der KI-Anwendung vorgenommen, alle Inhalte in einer Datei zusammengetragen, gesamte Logik in passende Funktionen verpackt und aufrufbar über eine `main()`-Funktion gemacht.

Als weitere Möglichkeit zum Aufruf der KI-Anwendung wurde das Modul `argparse` in das Modul eingebunden. Damit ist der Aufruf von Programmteilen direkt über die Konsole möglich. Das Modul `argparse` stellt dabei Funktionen zur Verfügung, die an die Anwendung übergebene Parameter auf der Konsole parst, also verarbeitet. Außerdem werden automatisierte, nützliche Hilfetexte für die Kommandozeile erzeugt.

Hier im Detail ein Auszug aus der Datei `__init.py__` des Moduls `chatbot`:

Abbildung 5: Programmcode: Main()-Funktion aus KI-Anwendung

```
def main():
  parser = ArgumentParser(description='Allows the use of a chatbot with PyTorch.')

  parser.add_argument('-u', '--newuuid',  action='store_true',
            help='Get a new, unique id for a conversation.')

  parser.add_argument('-t', '--train',  action='store_true',
            help='Trains the NN model based on current data.')

  parser.add_argument('-c', '--chat',  action='store_true',
            help='Chat with the chatbot.')

  parser.add_argument('-i', '--textinput',  type=str,
            help='Give an text input to the chatbot and recive a answer. \
              Provide a UUID for the conversation with -cu or --conversationuuid \
                followed by the UUID.')

  parser.add_argument('-cu', '--conversationuuid',  type=str,
            help='Needed when using -i, --textinput. Provide a UUID to a
    conversation.')

  args = parser.parse_args()


  if vars(args)['newuuid']:
    print(str(get_new_uuid()))
    quit()

  if vars(args)['train']:
    train()
    quit()

  if vars(args)['chat']:
    chat()
    quit()

  # check for complete input for a 'singlechat' call
  complete_text_input = 0
  if vars(args)['textinput']:
    complete_text_input += 1
  if vars(args)['conversationuuid']:
    complete_text_input += 1

  if complete_text_input == 1:
    parser.error("Too few arguments: -cu/--conversationuuid and -i/--textinput are
    both needed.")

  if complete_text_input == 2:
    print(str(singlechat(str(args.conversationuuid), str(args.textinput))))
    quit()

  parser.print_help()


if __name__ == '__main__':
  main()
```

Anhand des strukturierten Aufbaus kann man die Programmlogik und die wesentlichen Aufrufe gut ablesen. Wird das Modul aufgerufen, springt Python in die `main()` -Funktion

(Zeile 53 und 54). In dieser Funktion wird zuerst das Modul `ArgumentParser` initialisiert (Zeile 2), im Anschluss werden die möglichen Aufrufe über Konsolenparameter konfiguriert (Zeile 4 bis 19). In Zeile 20 werden tatsächlich die beim Programmaufruf mitgegebenen Parameter verarbeitet (geparst). Entsprechend der mitgegebenen Parameter werden dann in den Zeilen 24 bis 48 die jeweiligen je nach Programmfunktionen gestartet und das Skript wird beendet. Fand keine Auswahl statt, wird durch Zeile 50 das hier zu sehende Hilfemenü auf der Konsole ausgegeben:

Abbildung 6: Hilfemenü der KI-Anwendung bei Konsolenaufruf

```
pipenv run python -m chatbot
usage: __main__.py [-h] [-u] [-t] [-c] [-i TEXTINPUT] [-cu CONVERSATIONUUID]

Allows the use of a chatbot with PyTorch.

options:
  -h, --help            show this help message and exit
  -u, --newuuid         Get a new, unique id for a conversation.
  -t, --train           Trains the NN model based on current data.
  -c, --chat            Chat with the chatbot.
  -i TEXTINPUT, --textinput TEXTINPUT
                        Give an text input to the chatbot and recive a answer.
    Provide a UUID for the conversation with
                        -cu or --conversationuuid followed by the UUID.
  -cu CONVERSATIONUUID, --conversationuuid CONVERSATIONUUID
                        Needed when using -i, --textinput. Provide a UUID to a
    conversation.
```

Neben dem Umbau zum Python-Modul und der Einbindung der Konsolenparameter, wurde noch das Logging weiter verfeinert und vervollständigt. Dieser Entwicklungsstand befindet sich im Unterordner `20-prepared-ai-app`. Der Inhalt ist ebenfalls wieder in Anlage der zu dieser Seminararbeit sowie auch auf Github.com zu finden. Der Aufruf kann wieder über die Konsole mit folgenden Befehlen erfolgen:

```
pipenv install
pipenv run python -m chatbot -t
pipenv run python -m chatbot
```

Dieser Aufruf führt dann zu dem oben sichtbaren Hilfemenü. Mit dem Befehl `pipenv run python -m chatbot -c` kann ein Chat auf der Konsole im direkten Dialog gestartet werden.

Die Vorbereitung der KI-Anwendung, des Chatbots kann damit an dieser Stelle abgeschlossen werden. Die Einbindung in ein anderes Python-Skript und auch der Aufruf über die Konsole sind möglich.

3.2 Einrichtung Django-Grundinstallation

Die Bereitstellung des Webservices als REST-API soll über Django erfolgen. Daher wird zuerst eine Grundinstallation von Django vorgenommen. Eine fertige Django-Grundinstallation sowie die Dokumentation über die Installation befindet sich in den Anlagen zur Seminararbeit und auf Github.com im Ordner `30-django-base-install` in der Datei `README.md`.

Bei dieser Installation wurden alle notwendigen Pakete und Abhängigkeiten berücksichtigt, die für die Einbindung der zuvor vorbereiteten KI-Anwendung notwendig waren. Auch einige weitere Pakete und Funktionen wurden vorausschauend bereits mit installiert. Neben zwingend für die Umsetzung notwendigen Paketen wie dem „Django Toolkit für REST API“, dem „Django REST framework“ wurden vor allem Pakete installiert, die Funktionen bereitstellen, die den Zugriff auf die API erleichtern.

Abbildung 7: Dateiinhalt der „Pipfile“ für die Django-Grundinstallation

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]

Django = "==4.1" # Django itself
djangorestframework = "==3.13.1" # Django toolkit for rest api
drf-yasg = "==1.21.3" # Swagger for Django / OpenAPI 2.0
django-cors-headers = "==3.13.0" # for cross site access from a frontend
daphne = "==3.0.2" # to serve django
whitenoise = "==6.2.0" # to serve static files
tzdata = "==2022.2" # to fix dependency error on some systems

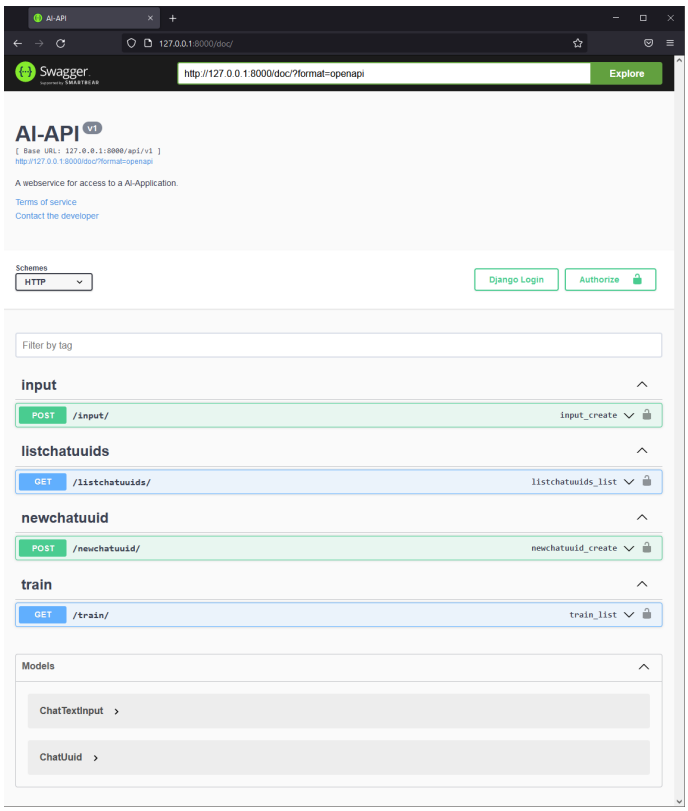
# for the chatbot module
numpy = "==1.23.2"
nltk = "==3.7"
HanTa = "==0.2.0" # 0.2.1 seems broken, fallback to 0.2.0
torch = "==1.12.1"

[dev-packages]

[requires]
python_version = "3"
```

Über das Paket `drf-yasg` bzw. „Yet another Swagger generator“ z. B. wird vollautomatisch eine Webseite generiert (Swagger), die der OpenAPI 2.0-Spezifikationen entspricht und Dritten damit eine vollständige Dokumentation der API bereitstellt und die Verwendung der API somit ermöglicht.

Abbildung 8: Bildschirmfoto von lokal laufender Swagger-Instanz



Quelle: Eigenes Bildschirmfoto

Der Installations- und Einrichtungsprozess ist bei Django gut zu dokumentieren. Die Schritte sind kleinteilig und führen, in der korrekten Abfolge ausgeführt, regelmäßig zum gleichen Ergebnis. Für diese Grundinstallation sind 14 Schritte erforderlich, die in einem Zeitraum von etwa 30 Minuten von einer Person mit IT-Kenntnissen gut abgearbeitet werden können. Es müssen z. B. verschiedene Konsolenbefehle, wie z. B. `pipenv run python manage.py createsuperuser` für die Anlage eines Administratoraccounts oder `pipenv run python -m chatbot -t` zum initialen Training des KI-Modells, eingegeben werden. Auch müssen verschiedene Dateien angelegt oder bestehenden Dateien Inhalte an der korrekten Stelle hinzugefügt werden. Dazu ein Auszug aus der Anleitung:

Abbildung 9: Auszug der „README.md“-Datei: Zeilenweise Konfiguration

```
1 # Pay attention to the order:
2 MIDDLEWARE = [
3     'corsheaders.middleware.CorsMiddleware', # add cors as the first line
4     ...
5     'django.middleware.security.SecurityMiddleware', # look out for this line and
      insert whitenoise below
6     'whitenoise.middleware.WhiteNoiseMiddleware', # for static files
7     ...
```

Neben diesen einfacheren Schritten war es aber auch notwendig, die KI-Anwendung selbst einzubinden. Dazu wird während des Installationsprozesses die KI-Anwendung aus dem Ordner `20-prepared-ai-app` in den Ordner der Django-Grundinstallation kopiert. Damit können Funktionen der KI-Anwendung als Modul an jeder Stelle von Django importiert und aufgerufen werden.

Dann müssen für die Funktionen der KI-Anwendung entsprechende Programmierungen in Django erstellt werden: Das Datenmodell muss abgebildet werden, die Sichten auf die Daten müssen erstellt und die entsprechenden Einstiegspunkte konfiguriert werden. Innerhalb von Django finden sich diese Programmierungen bzw. Konfigurationen in den Dateien `models.py`, `views.py` und `urls.py` wieder. Je nach Organisation finden sich diese Dateien an unterschiedlichen Stellen von Django wieder. Auch lassen sich die Inhalte auf verschiedene Hierarchien aufteilen und damit nach Programmteil getrennt hinterlegen. Das kann die Organisation von Anwendungen deutlich erleichtern.

In den in Django angelegten Datenmodellen erkennt mal z. B. schnell, dass diese vom Aufbau im Wesentlichen den bisherigen Log-Dateien entsprechen. Folgend sind die beiden Klassen zu den Datenmodellen zu sehen. Wie man erkennt, können diese auch mit Informationen angereichert werden, die im Webservice wichtige Hinweise zur korrekten Nutzung geben:

Abbildung 10: Auszug der „models.py“-Datei: das Django-Datenmodell

```
1  class ChatUuid(models.Model):
2      chatuuid = models.UUIDField(unique=True, default=uuid.uuid4,
3                                  editable=False, help_text="The UUID for a Chatbot
       conversation.")
4      created = models.DateTimeField(auto_now_add=True)
5      agreed = models.TextField(
6          blank=True, help_text="If user agreed to the service terms (http://...),
       send 'Einverstanden'.")
7
8      def __str__(self):
9          return str(self.chatuuid)
10
11 class ChatDialogue(models.Model):
12     chatuuid = models.UUIDField(blank=False)
13     created_at = models.DateTimeField(auto_now_add=True)
14     updated_at = models.DateTimeField(auto_now=True)
15     input = models.TextField(blank=False)
16     probability = models.FloatField(blank=True)
17     understood = models.BooleanField(blank=True)
18     output = models.TextField(blank=True)
19
20     def __str__(self):
21         return str(self.input)
```

So kann man durch die Angaben im Attribut `help_text = '...'` in den Zeilen 3 und 6 der Abbildung 10 hinterlegen, was für Nutzer der API relevant ist. Hier wird z. B.

der Klasse `ChatUuid` die Information mitgegeben, dass es sich dabei um eine UUID für ein Gespräch mit dem Chatbot handelt. Eine ChatUuid aber wird nur vergeben, so das Einverständnis zu den Nutzungsbedingungen erteilt wurde, hier indem der Parameter `agreed` mit dem Inhalt „Einverstanden“ übergeben wurde. Eine entsprechende Fehlermeldung wurde über einen Inhalt in der Datei `views.py` programmiert:

Abbildung 11: Auszug der „views.py“-Datei: die Django-Sichten

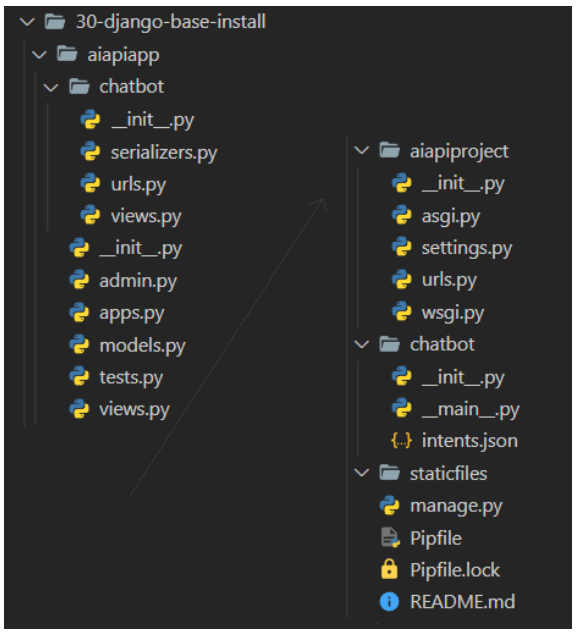
```
1  class ChatUuidNewAPIView(CreateAPIView):
2      model = ChatUuid
3      serializer_class = ChatUuidSerializer
4      queryset = ChatUuid.objects.all()
5
6      def post(self, request, *args, **kwargs):
7          request.data["chatuuid"] = get_new_uuid()
8
9          if request.data["agreed"] != "Einverstanden":
10             return Response(data="If 'agreed' is not 'Einverstanden' you can not
       use the services.",
11                             status=status.HTTP_400_BAD_REQUEST)
12
13         return self.create(request, *args, **kwargs)
```

In Abbildung 11 findet man eine Klasse des Typs `CreateAPIView`. Diese stellt über des „Djangao Rest framework“ einen Endpunkt für POST-Anfragen bereit. In der Zeile 6 sieht man entsprechend die Methode `def post(...):`. Die Programmlogik für die Fehlermeldung findet sich in der Zeile 9: Hier wird überprüft, ob der bereits bekannte Parameter `agreed` mit dem Inhalt „Einverstanden“ befüllt ist. Ist das nicht der Fall, wird über Zeile 10 eine Rückgabe mit Fehlermeldung und HTTP-Statuscode 400 (keine Verarbeitung aufgrund clientseitigem Fehler) zurückgegeben.

Nach dieser Systematik wurden für alle Funktionen der KI-Anwendung entsprechend Sichten und Einstiegspunkte programmiert, die der Installationsanleitung `30-django-base-installREADME.md` in den Anlagen und auf Github.com entnommen werden kann. Auch finden sich natürlich alle Programmierungen und Konfigurationen in dem Ordner zur Django-Grundinstallation selbst (`30-django-base-install`).

Den vollständigen Ordnerinhalt zeigt die folgende Abbildung:

Abbildung 12: Ordnerinhalt Django-Grundinstallation



Quelle: Eigenes Bildschirmfoto

Für die API wurden insgesamt vier Endpunkte, also die Zugriffspunkte auf die KI-Anwendung, erstellt. Zwei der Endpunkte, `input` und `newchatuuid`, sind dabei für das Gespräch mit dem Chatbot notwendig. Mittels einer POST-HTTP-Anfrage an `/api/v1/chatbot/input` können schon mit den beiden Pflichtparametern `input` und `chatuuid` Eingaben an die KI-Anwendung übergeben werden. Der Chatbot verarbeitet diese dann über den Modulaufruf aus einer Django-Sicht bzw. View heraus und gibt die Antwort im Feld `output` zusammen mit einigen anderen Feldern wie der Wahrscheinlichkeit des Verständnisses im direkten Dialog als Antwort auf die Anfrage selbst mit einem HTTP-Statuscode 200 (erfolgreich) zurück, wie die folgende Abbildung zeigt:

Abbildung 13: Antwort des Webservices auf Chatbot-Eingabe


```
{
  "id": 112,
  "chatuuid": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "created_at": "2022-08-28T20:31:48.580175Z",
  "updated_at": "2022-08-28T20:31:48.580175Z",
  "input": "Magst du Katzen?",
  "probability": 0.9999604225158691,
  "understood": true,
  "output": "≧◔◡◔≦"
}
```


Quelle: Eigenes Bildschirmfoto

Einen weiteren Endpunkt gibt es zum Training des KI-Modells, den `/api/v1/chatbot/train`. Dieser Endpunkt ist für die Verwendung mit einer GET-Methode angelegt. Der Aufruf jedoch wird nur erlaubt, wenn über eine HTTP-Basis-Authentifizierung ein Benutzername und das entsprechende Passwort eines Administratorkontos der Anfrage mitgegeben werden. Bei fehlenden Authentifizierungsdaten erfolgt die Antwort mit HTTP-Statuscode 403 (verboten) und die Fehlermeldung `{ "detail": Äuthentication credentials were not provided."}`. Werden falsche Angaben gemacht, erfolgt die Antwort mit gleichem Statuscode, jedoch mit der Fehlermeldung `{ "detail": Ïnvalid username/password."}`. Diese Fehlermeldungen werden dabei von Django bzw. dem „Django Rest framework“ automatisch erzeugt und übermittelt. Die einzige Einstellung dazu erfolgte in den Sichten, der Datei `views.py`, in Zeile 2 durch eine Angabe im Parameter `permission_classes`:

Abbildung 14: Auszug der „views.py“-Datei: Berechtigungen einer Klasse

```
1  class ChatTrainAPIView(views.APIView):
2      permission_classes = [permissions.IsAdminUser]
3
4      def get(self, request, format=None):
5          """
6          Returns the console output from the training process.
7          Only Admin Users can start the training.
8          """
9          trainoutput = train()
10
11         return Response(data=str(trainoutput), status=status.HTTP_200_OK)
```

Der Aufruf des `train`-Endpunkts dauert meist zumindest einige Sekunden. Es wird dadurch klar, dass die direkte, dialogbasierte Kommunikation vor allem durch Antwortzeiten beschränkte Grenzen hat. Anfragen, die in aller Regel länger als den Bruchteil einer Sekunde dauern, könnten über die API eine Rückmeldung geben, die anzeigt, dass die finale Antwort noch nicht in der nun erhaltenen Rückmeldung enthalten war. Es könnte dabei ein Token zur Nachfrage zum Bearbeitungsstand und Abholung des finalen Ergebnisses bei dem Webservice übergeben werden. Auch sehr lange laufende Verarbeitungen könnten damit für Nutzer der API transparent verarbeitet werden. Eine Umsetzung dieser Idee erfolgte hier jedoch aus Gründen des Umfangs der Seminararbeit nicht.

3.3 Zugriff mit einem Web-Frontend (Vue.js und Axios)

Für den einfachen Zugriff und um die Funktionalität abschließend zu bestätigen, soll auch ein Zugriff auf den neuen Webservice der KI-Anwendung über ein Web-Frontend durchgeführt werden. Dazu wurde eine einfache Webseite erstellt, die schrittweise die

dialogbasierte Kommunikation mit der KI-Anwendung, also dem Chatbot, aufbaut und durchführt. Die Auswahl des zum Einsatz kommenden Frameworks fiel hier auf „Vue.js“, das als leichtgewichtiges, quelloffenes und schnell einzurichtendes Framework ideal für diesen Anwendungsfall ist. Ein Chatdialog erfordert hier keinen besonders komplexen oder aufwendigen Aufbau. Lediglich HTTP-Anfragen müssen an eine REST-API gestellt werden können. Dazu kommt „Axios“ als HTTP-Bibliothek auf der Webseite zum Einsatz.

Die Erstellung der Webseite erfolgte durch Nutzung einer in den Funktionen leicht nachvollziehbare Beispielanwendung (bezkoder, 2021). Die dort erstellte Anwendung wurde gründlich gesichtet und konnte in der Funktionsweise nachvollzogen werden. Weitere Anpassungen und Optimierungen konnten darauf erfolgen, insbesondere der Funktionsumfang konnte noch weiter komprimiert werden. Dies war deshalb möglich, da in dem Anwendungsfall des Chatbots nur noch POST-Methoden zum Einsatz kommen.

Die erstellte Webseite befindet sich in den Anlagen und auf Github.com im Ordner `40-vue-axios-webpage`. Der Start der Webseite erfolgte nach Installation von „node.js“ mittels Eingabe von `npm install` und `npm run server` im Projektordner.

Die Verbindung zwischen dem Frontend (Webseite, Vue.js mit Axios) und Backend (Django, KI-Anwendung) wird in der Datei `http-common.js` eingestellt:

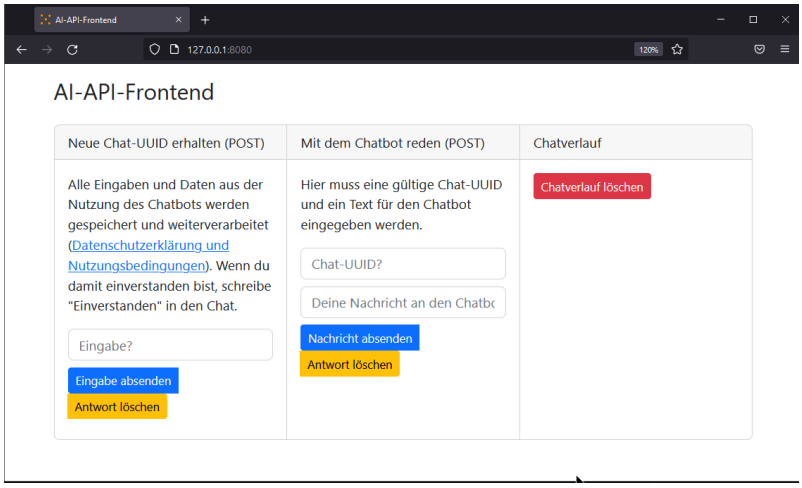
Abbildung 15: Programmcode: Verbindungskonfiguration Back- und Frontend

```
1  import axios from "axios";
2
3  export default axios.create({
4    baseURL: "http://127.0.0.1:8000/api/v1",
5    headers: {
6      "Content-type": "application/json",
7    },
8  });
```

Das Web-Frontend wurde passend zum Webservice entworfen. Damit das Durchführen von Tests und ein Eingriff in die Abläufe leicht möglich sind sowie zur leichten Fehlersuche, wurde die Webseite bewusst nicht als fertige Chatanwendung mit nur einer Eingabezeile und einem Ausgabefenster entworfen. Die Webseite bildet den mehrstufigen Verbindungsauf zwischen dem Client und Server. Zuerst muss eine Chat-UUID durch Übermittlung des Einverständnisses zu den Nutzungsbedingungen abgeholt werden. Dies ist in Abbildung 16 unten links zu sehen. Erst dann kann mit dieser Chat-UUID mit dem Chatbot gesprochen werden. Im mittleren Teil der Webseite können dann die

Chat-UUID sowie die Nachricht an den Chatbot eingetragen werden. Die Antwort wird dann im rechten Teil der Webseite dargestellt.

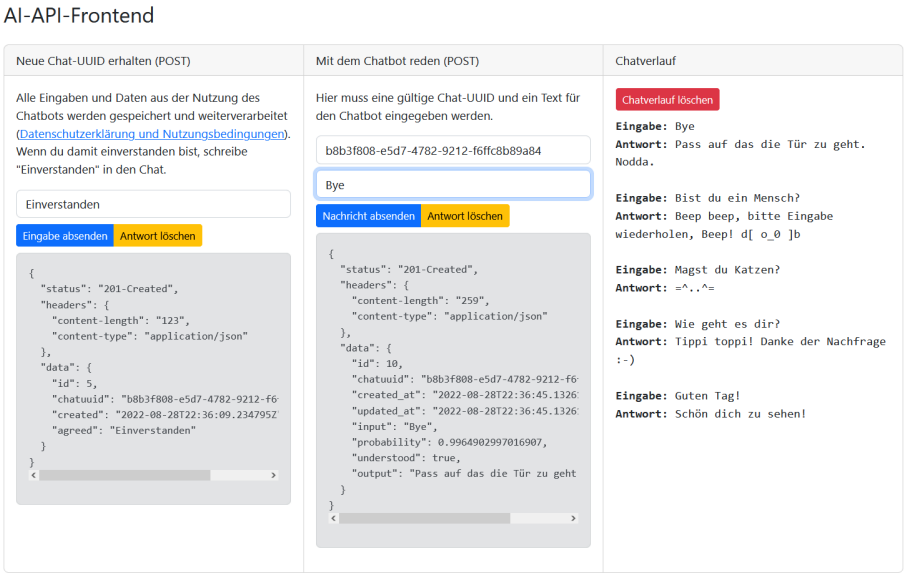
Abbildung 16: Bildschirmfoto: Mit Vue.js und Axios erstellte Webseite



Quelle: Eigenes Bildschirmfoto

In der Abbildung 17 ist das Ergebnis eines erfolgten Dialogs zu sehen. Die Antworten inkl. technischer Details werden jeweils unter der Aktion dargestellt. Sichtbar in den grauen Feldern der linken und mittleren Teile der Webseite. Im rechten Bereich wird der erfolgte Dialog mit dem Chatbot dargestellt.

Abbildung 17: Bildschirmfoto: Das Web-Frontend wurde für Dialog genutzt



Quelle: Eigenes Bildschirmfoto

3.4 Verteilung durch Containerisierung (Docker-Compose)

Zur einfacheren Verteilung der bisher erstellten Lösungen, gleich ob an Interessierte, zum fachlichen Austausch oder zu Zwecken der Veröffentlichung, muss das bisher erstellte (Django im Backend und Vue.js im Frontend) noch in Container überführt werden. Dadurch kann der Transport auf andere Systeme und IT-Umgebungen deutlich vereinfacht werden. Durch den Einsatz einer `docker-compose.yml`-Datei kann, nach Einrichtung, durch nur ein Kommando die gesamte, benötigte IT-Infrastruktur von Frontend und Backend gestartet werden.

In einem letzten Ordner `50-docker-deployment`, der ebenfalls wieder in den Anlagen zur Seminararbeit und auch auf Github.com zu finden ist, wurden zwei Unterordner erstellt. In diesen beiden Ordnern, Backend und Frontend, wurden die bisherigen Ergebnisse aus den Abschnitten zur Django-Grundinstallation und der Vue.js-Webseite übernommen. Ergänzt wurden diese Ordner jeweils durch passende `Dockerfile`-Dateien, in denen der Erstellungsprozess der Container anhand einer vorgegebenen Reihenfolge von Befehlen konfiguriert wurde.

Die Erstellung des Frontends folgt dabei einem 2-stufigen Plan, der zuerst über `npm run build` alle Inhalte für die Bereitstellung über „nginx“ erzeugt und dann im zweiten Schritt diese mit `CMD ["nginx", g", "daemon off;"]` als Webserver bereitstellt:

Abbildung 18: Programmcode: Dockerfile für das Frontend

```
# build stage
FROM node:18-bullseye as build-stage
WORKDIR /usr/src/app/ai_api_frontend
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
COPY --from=build-stage /usr/src/app/ai_api_frontend/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Die Erstellung des Containers für das Backend erfolgt auch über ein `Dockerfile`. Es fällt hier vielleicht auf, dass auch der `node:18-bullseye`-Container als Grundlage verwendet wird. Dieser wird hier gebraucht, da er schon Anwendung im Frontend findet und dadurch, dass er auch auf Debian basiert, sehr gut für das Backend verwendet werden kann.

Abbildung 19: Programmcode: Dockerfile für das Backend

```
# same base container as the frontend to save traffic and space
FROM node:18-bullseye
WORKDIR /usr/src/ai_api_backend
COPY . .
RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install pipenv -y
RUN pipenv install
RUN chmod +x start_api.sh
CMD ["sh", "start_api.sh"]
```

Neben dem `Dockerfile` für das Backend ist für den Start des Backends noch das Skript nötig, das die bei Django notwendigen Schritte ausführen kann. Dazu wurde die Datei `start_api.sh` erstellt und die notwendigen Abläufe (Datenbankmigrationen, statische Dateien einsammeln, KI-Modell trainieren usw.) wurden dementsprechend hinterlegt. Auch der Start des Backends selbst erfolgt über das Skript. Für die Bereitstellung des Webservices habe ich die Anwendung „daphne“ gewählt, da sie nicht unbedingt einen vorangestellten Proxy benötigt und auf eine modernere, asynchrone Schnittstelle setzt (ASGI).

Abbildung 20: Programmcode: „start_api.sh“-Skript zum Start des Backends

```
#!/bin/bash

echo 'Django startup...'
pipenv run python manage.py makemigrations
pipenv run python manage.py migrate --run-syncdb  # Apply database migrations
pipenv run python manage.py collectstatic --noinput  # Collect static files
pipenv run python -m chatbot -t  # train AI model

echo 'Starting Django in production mode (Daphne):'
exec pipenv run daphne -b 0.0.0.0 -p 8000 aiapiproject.asgi:application
```

Für das Bereitstellen der statischen Inhalte wurde schon in einem früheren Schritt das Django-Paket `whitenoise` installiert und konfiguriert, so dass hier im Übrigen keine weiteren Anpassungen mehr notwendig waren. Grundlage für die Erstellung dieser Containerisierung waren ein Teil der Vue.js-Dokumentation(You, n. d.), ein Artikel zu Docker-Compose und Vue.js (Bachina, 2020) sowie Inhalte aus den Vorlesungen zum Modul „IT-Infrastruktur“.

Aufgrund des beschränkten Umfangs dieser Seminararbeit muss an dieser Stelle auf weitere Ausführungen zu Docker-compose verzichtet werden. Ebenso kann auch keine Beschreibung der Produktivnahme dieser Lösung mit einem Proxy wie „Traefik“ und auch nicht die Konfiguration einer HTTPS-Verbindung mittels Zertifikaten von „Let's Encrypt“ erfolgen.

4 Fazit und Reflexion

Das gewünschte Ziel dieser Seminararbeit konnte erreicht werden: Eine KI-Anwendung konnte über eine REST-API, die über Django bereitgestellt wurde, erreicht werden. Auch war es möglich, die sich daraus ergebenden Vorzüge ddarzustellen und nutzbar zu : Die KI-Anwendung ist durch die Bereitstellung als Webservice deutlich leichter für Dritte zu erreichen. Auch ist die Lösung in ihrer Gesamtheit als ein Stück IT-Infrastruktur durch die erfolgreiche Containerisierung transportabel und verteilbar geworden.

Eine Herausforderung und Schwierigkeit bei der Entwicklung war es, die Grenze zwischen der API und der KI-Anwendung aufrecht zu erhalten. Bei dem Zusammenführen der Teile, also der KI-Anwendung in Django, verschwimmen und verwachsen Programmteile sehr schnell miteinander. Während der Umsetzung musste daher stets mit Bedacht und größter Sorgfalt vorgegangen werden, um die Funktionen der KI-Anwendung in die API einzubinden, ohne dabei Veränderungen an der KI-Anwendung selbst vorzunehmen. Eine Vermischung war hier aufgrund der Aufgabenstellung und Zielsetzung dringend zu vermeiden und war auch gelungen.

Die Vorstellung jedoch, die KI-Anwendung selbst vollständig in Django zu integrieren, war dabei stets als sehr naheliegend präsent. Alle Daten und Logs hätten statt in einer Datei, direkt über die Django-Datenmodelle und Abstraktion bequem in einer Datenbank gespeichert werden können. Für die Zukunft ist zu überlegen, ob eine solch strikte Trennung nicht den aufgezeigten Vorzügen gegenüber aufgegeben werden sollte.

Die Seminararbeit war für mich persönlich eine höchst interessante und spannende Aufgabe. Ich konnte durch die Umsetzung viele Details und Hintergründe zu den im Modul „IT-Infrastruktur“ vermittelten Grundlagen erlernen und auch mein Wissen um Funktionen in Python deutlich erweitern.

Anhang

Der Anhang beschränkt sich auf die Abbildung des Inhalts von zwei umfangreichen Dateien. Diese wurden ausgewählt, da sie wesentliche Elemente der Seminararbeit sind und auch ohne weiteren Kontext anderer Dateien oder Inhalte aufschlussreich sein können.

Alle übrigen Anlagen auf die in der Seminararbeit Bezug genommen wurde, wurden bei der Abgabe der Seminararbeit als Dateien mit eingereicht. Ebenso wurden alle Inhalte aber auch auf Github.com unter `https://github.com/haenno/ai-api` veröffentlicht.

Anhang 1: Notizen zur Django-Grundinstallation

Alle Einzelschritte die für die Django-Grundinstallation notwendig waren, wurden in der Datei `30-django-base-install\README.md` im Markdown-Format festgehalten. Diese Datei wird hier im Anhang aufgenommen und in voller Länge dargestellt, da damit der Umfang und Detailgrad der Installationsanleitung verdeutlicht wird. Diese Installationsanleitung ist dabei das Ergebnis intensiver Recherchen und umfangreicher Tests.

````
# django-base-install

The contents of this folder are based on my own notes <https://github.com/haenno/
    public-krams/blob/main/django-rest-notes.md> and the experience I made while
    developing a browser RPG adventure for a previous exam <https://github.com/
    tstsrv-de/rpg>.

## Useage

This folder provides a ready to use installation of Django with the AI-Application
    allready included. You can also make your own, clean install with the steps
    described in the next part. But if you just want to test it, take these steps:

1. Install Python and Pipenv.
2. Install depencys with ``pipenv install``.
3. Take the following steps to initialize and start Django:

    ```bash
    pipenv run python manage.py makemigrations
    pipenv run python manage.py migrate --run-syncdb
    pipenv run python manage.py createsuperuser
    pipenv run python manage.py collectstatic --noinput
    pipenv run python -m chatbot -t
    pipenv run daphne -b 0.0.0.0 -p 8000 aiapiproject.asgi:application
    ```

4. Open the backend on <http://127.0.0.1:8000>.
````

```
## The installation process

1. Install Python and Pipenv.
2. Create a ``Pipfile`` in the main folder with the following contents:

    ```toml
    [[source]]
    url = "https://pypi.org/simple"
    verify_ssl = true
    name = "pypi"

    [packages]

    Django = "==4.1" # Django itself
    djangorestframework = "==3.13.1" # Django toolkit for rest api
    drf-yasg = "==1.21.3" # Swagger for Django / OpenAPI 2.0 fits perfectly
    django-cors-headers = "==3.13.0" # for cross site access from a frontend
    daphne = "==3.0.2" # to serve django
    whitenoise = "==6.2.0" # to serve static files
    tzdata = "==2022.2" # to fix depency error on some systems

    # for the chatbot module
    numpy = "==1.23.2"
    nltk = "==3.7"
    HanTa = "==0.2.0" # 0.2.1 does not work well at this time, fallback to 0.2.0
    torch = "==1.12.1"

    [dev-packages]

    [requires]
    python_version = "3"

    ```

3. Then install the packeages with runnig ``pipenv install``.

4. Create the Django project with the command ``pipenv run django-admin
    startproject aiapiproject .`` in the same folder.

5. Next create the Django app with ``pipenv run django-admin startapp aiapiapp``.

6. Add and/or change lines in the `settings.py` in porject folder `aiapiproject`:

    ```python
    import os # for staticfiles

    # fill the list
    ALLOWED_HOSTS = ['localhost', '0.0.0.0', '127.0.0.1']

    # add the lines on the bottom to the list
    INSTALLED_APPS = [
        ...
        'aiapiapp', # the ai-app itself
        'rest_framework', # Django toolkit for a rest api
        'drf_yasg', # Swagger for Django / OpenAPI 2.0 will do just fine
        'corsheaders', # for configure acces from a frontend
```
```

```
    ]


    # Pay attention to the order:
    MIDDLEWARE = [
        'corsheaders.middleware.CorsMiddleware', # add cors as the first line
        ...
        'django.middleware.security.SecurityMiddleware', # look out for this line
    and insert whitenoise below
        'whitenoise.middleware.WhiteNoiseMiddleware', # for static files
        ...

    # add the following to the bottom of the file

    REST_FRAMEWORK = {
        # Use Django's standard `django.contrib.auth` permissions,
        # or allow read-only access for unauthenticated users.
        'DEFAULT_PERMISSION_CLASSES': [
            #'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly', #
    for later use
            'rest_framework.permissions.AllowAny',
        ]
    }

    STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles') # for staticfiles

    CORS_ORIGIN_ALLOW_ALL=True

    # for later use
    # CORS_ORIGIN_WHITELIST = [
    #     'http://localhost:8000',
    #     'http://127.0.0.1:8000'
    # ]

    ```
```

7. Add and/or change lines in the `urls.py` also in the porject folder `aiapiproject`:

```
    ```python
    from django.urls import include, re_path
    from django.http import HttpResponse # for index page

    # for swagger: start
    from rest_framework import permissions
    from drf_yasg.views import get_schema_view
    from drf_yasg import openapi
    schema_view = get_schema_view(
        openapi.Info(
            title="AI-API",
            default_version='v1',
            description="A webservice for access to a AI-Application.",
            terms_of_service="https://github.com/haenno/ai-api",
            contact=openapi.Contact(email="haenno@web.de"),
        ),
        public=True,
        permission_classes=(permissions.AllowAny,),
```

```
    )
    # for swagger: end

    urlpatterns = [
        path('', lambda request: HttpResponse('<html><body><p><h1>This is the
    backend of an \
            AI-API-Application</h1><b>Proceed to</b>         <a href="doc/">doc</a
    >, \
            <a href="redoc/">redoc</a>, <a href="spec.json">spec.json</a>         \
            or <a href="spec.yaml">spec.yaml</a> for more details to this API.<br>\
            <br>Check <a href="https://github.com/haenno/ai-api">github.com/haenno/
    ai-api</a>\
            for the project documentation.</p></body></html>'), name='index'),  #
    for index page
        path('admin/', admin.site.urls),
        re_path(r'^spec(?P<format>\.json|\.yaml)$',
                schema_view.without_ui(cache_timeout=0), name='schema-json'),  #
    for swagger
        path('doc/', schema_view.with_ui('swagger', cache_timeout=0),
            name='schema-swagger-ui'),  # for swagger
        path('redoc/', schema_view.with_ui('redoc', cache_timeout=0),
            name='schema-redoc'),  # for swagger
        path('api/v1/', include('aiapiapp.chatbot.urls')),
    ]

    ```
```

8. Create a folder ``staticfiles`` in the base directory (same as the file manage.py).

9. Create an API endpoint that creates and gives us a UUID. First, create a new folder in the APP directory `aiapiapp\chatbot\`.

10. Then create new files in that folder:
    1. A empty `__init__.py`.
    2. A file `serializers.py` with:

        ```python
        from rest_framework import serializers
        from aiapiapp.models import ChatUuid, ChatDialogue


        class ChatUuidSerializer(serializers.ModelSerializer):
            class Meta:
                model = ChatUuid
                fields = '__all__'


        class ChatTextInputSerializer(serializers.ModelSerializer):

            class Meta:
                model = ChatDialogue
                fields = '__all__'
        ```

    3. A file `urls.py` with:
        ```

```
    ```python
    from django.urls import path
    from aiapiapp.chatbot.views import ChatUuidNewAPIView, ChatUuidListAPIView,
 ChatInputAPIView, ChatTrainAPIView

    app_name = 'aiapiapp'

    urlpatterns = [
        path('listchatuuids/', ChatUuidListAPIView.as_view(), name='
listchatuuids'),
        path('newchatuuid/', ChatUuidNewAPIView.as_view(), name='newchatuuid'),
        path('input/', ChatInputAPIView.as_view(), name='input'),
        path('train/', ChatTrainAPIView.as_view(), name='train'),
    ]

    ```

4. A file `views.py` with:

    ```python
    from distutils.log import error
    import json
    import uuid
    from rest_framework.generics import ListAPIView, CreateAPIView
    from aiapiapp.models import ChatUuid, ChatDialogue
    from aiapiapp.chatbot.serializers import ChatUuidSerializer,
ChatTextInputSerializer
    from rest_framework import status, views
    from rest_framework.response import Response
    from rest_framework import authentication, permissions

    from chatbot import get_new_uuid, singlechat, train


    class ChatUuidListAPIView(ListAPIView):
        model = ChatUuid
        serializer_class = ChatUuidSerializer
        queryset = ChatUuid.objects.all().order_by('-created')


    class ChatUuidNewAPIView(CreateAPIView):
        model = ChatUuid
        serializer_class = ChatUuidSerializer
        queryset = ChatUuid.objects.all()

        def post(self, request, *args, **kwargs):
            request.data["chatuuid"] = get_new_uuid()

            if request.data["agreed"] != "Einverstanden":
                return Response(data="If 'agreed' is not 'Einverstanden' you
can not use the services.",
                                status=status.HTTP_400_BAD_REQUEST)

            return self.create(request, *args, **kwargs)


    class ChatInputAPIView(CreateAPIView):
```
```

```
            model = ChatDialogue
            serializer_class = ChatTextInputSerializer

            def post(self, request, *args, **kwargs):
                try:
                    textinput = str(request.data["input"])
                    if type(textinput) is not str and\
                            len(textinput) < 1:
                        raise error

                    chatuuid = uuid.UUID(request.data["chatuuid"])
                    if type(chatuuid) is not uuid.UUID and\
                            len(chatuuid) != 36:
                        raise error

                except:
                    return Response(data="Please provide a valid UUID for the
    conversation and a input text to answer to.",
                                    status=status.HTTP_400_BAD_REQUEST)

                answer = json.loads(singlechat(chatuuid, textinput))

                request.data["probability"] = float(answer["probability"])
                request.data["understood"] = bool(answer["understood"])
                request.data["output"] = str(answer["output"])

                return self.create(request, *args, **kwargs)


        class ChatTrainAPIView(views.APIView):
            permission_classes = [permissions.IsAdminUser]

            def get(self, request, format=None):
                """
                Returns the console output from the training process.
                Only Admin Users can start the training.
                """
                trainoutput = train()

                return Response(data=str(trainoutput), status=status.HTTP_200_OK)


        ```
```

11. Then some changes in the files in the APP directory ``aiapiapp``:
    1. In the `admin.py` add:

        ```python
        from aiapiapp.models import ChatUuid, ChatDialogue
        admin.site.register(ChatUuid)
        admin.site.register(ChatDialogue)

        ```

    2. In the `models.py` add:

        ```python

```
        import uuid
        class ChatUuid(models.Model):
            chatuuid = models.UUIDField(unique=True, default=uuid.uuid4,
                                        editable=False, help_text="The UUID for a
    Chatbot conversation.")
            created = models.DateTimeField(auto_now_add=True)
            agreed = models.TextField(
                blank=True, help_text="If user agreed to the service terms (http
    ://...), send 'Einverstanden'.")

            def __str__(self):
                return str(self.chatuuid)

        class ChatDialogue(models.Model):
            chatuuid = models.UUIDField(blank=False)
            created_at = models.DateTimeField(auto_now_add=True)
            updated_at = models.DateTimeField(auto_now=True)
            input = models.TextField(blank=False)
            probability = models.FloatField(blank=True)
            understood = models.BooleanField(blank=True)
            output = models.TextField(blank=True)

            def __str__(self):
                return str(self.input)
        ```
```

12. Then copy over the chatbot AI-Application module folder. Here it was the contents of ``chatbot``-Folder in ``20-prepared-ai-app``. Copy it in the folder, that contains the ``manage.py``-File.

13. Then initialize Django with the following coammnds. Note the superuser credentials!

    ```bash
    pipenv run python manage.py makemigrations
    pipenv run python manage.py migrate
    pipenv run python manage.py createsuperuser
    pipenv run python manage.py collectstatic --noinput
    pipenv run python -m chatbot -t
    ```

14. The finally start Django by running ``pipenv run daphne -b 0.0.0.0 -p 8000 aiapiproject.asgi:application`` and open <http://127.0.0.1:8000>.

License

MIT License: Copyright (c) 2022 Henning 'haenno' Beier, haenno@web.de, <https://github.com/haenno/ai-api>
    ```

Anhang 2: KI-Anwedung als Python-Modul

Die für die Einbindung in Django als Python-Modul vorbereitete KI-Anwendung findet sich in der Datei `20-prepared-ai-app\chatbot\_init__.py` die hier im Anhang der Seminararbeit aufgenommen wurde. Damit soll ein leichter Einblick in die Programmlogik ermöglicht werden.

```
try:
  from ast import arg
  from mailbox import linesep
  import nltk
  import torch
  import torch.nn as nn
  import json
  import numpy as np
  import uuid
  import random
  import csv
  import os

  from pickle import NONE
  from datetime import datetime
  from argparse import ArgumentParser
  from torch.utils.data import Dataset, DataLoader
  from HanTa import HanoverTagger as ht

except ModuleNotFoundError:
  print("Error: Install all needed modules first! See README.md for details...")
  quit()

# CPUGPU = 'cuda:0'
CPUGPU = 'cpu'

SRC_FILE = 'intents.json'
OUT_FILE = 'data.pth'
OPEN_QUESTIONS_FILE = 'to_answer.csv'
DIALOG_LOG_FILE = 'dialog_log.csv'
MIN_PROB = 0.95

nltk.download('punkt')  # run only once (uncomment once, to init a new system)
tagger = ht.HanoverTagger('morphmodel_ger.pgz')


class NeuralNet(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
      super(NeuralNet, self).__init__()

      self.l1 = nn.Linear(input_size, hidden_size).to(CPUGPU)
      self.l2 = nn.Linear(hidden_size, hidden_size).to(CPUGPU)
      self.l3 = nn.Linear(hidden_size, num_classes).to(CPUGPU)

      self.relu = nn.ReLU().to(CPUGPU)

```

```
    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)

        return out


def singlechat(conversation_uuid, text_input):
    """_summary_

    Args:
        conversation_uuid (_type_): _description_
        text_input (_type_): _description_

    Returns:
        _type_: _description_
    """

    device = torch.device(CPUGPU)
    data = torch.load(os.path.join(os.path.dirname(os.path.abspath(
        __file__)), OUT_FILE), map_location=torch.device(CPUGPU))

    input_size = data["input_size"]
    hidden_size = data["hidden_size"]
    output_size = data["output_size"]
    all_words = data['all_words']
    tags = data['tags']
    model_state = data["model_state"]

    model = NeuralNet(input_size, hidden_size, output_size).to(device)
    model.load_state_dict(model_state)
    model.eval()

    bot_name = "Chatbot"

    log_uuid = conversation_uuid

    log_dialog(log_uuid, 200)

    # sentence
    sentence = text_input

    log_dialog(log_uuid, 400, sentence)

    #sentence = tokenize(sentence)
    X = bag_of_words(sentence, all_words)
    X = X.reshape(1, X.shape[0])
    X = torch.from_numpy(X).to(device)

    output = model(X)
    _, predicted = torch.max(output, dim=1)

    tag = tags[predicted.item()]

```

```
104     probs = torch.softmax(output, dim=1).to(CPUGPU)
105     prob = probs[0][predicted.item()]
106 
107     if prob.item() > MIN_PROB:
108 
109         with open(os.path.join(os.path.dirname(os.path.abspath(__file__)), SRC_FILE),
        'r', encoding='utf-8') as json_data:
110             intents = json.load(json_data)
111 
112         for intent in intents['intents']:
113             if tag == intent["tag"]:
114                 answer = random.choice(intent['responses'])
115                 log_dialog(log_uuid, 600, answer, prob.item())
116                 return json.JSONEncoder().encode({
117                     'probability': float(prob.item()),
118                     'understood': bool(True),
119                     'output': str(answer),
120                 })
121     else:
122         log_dialog(log_uuid, 650, "", prob.item())
123 
124         with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
      OPEN_QUESTIONS_FILE), 'a', encoding='utf-8') as f:
125             f.write(str(log_uuid) + ";\"" + str(datetime.now()) +
126                     "\";\"" + sentence + "\"\n")
127         return json.JSONEncoder().encode({
128             'probability': float(prob.item()),
129             'understood': bool(False),
130             'output': str("Das habe ich nicht verstanden...")
131         })
132 
133 
134 def chat():
135 
136     device = torch.device(CPUGPU)
137     data = torch.load(os.path.join(os.path.dirname(os.path.abspath(
138         __file__)), OUT_FILE), map_location=torch.device(CPUGPU))
139 
140     input_size = data["input_size"]
141     hidden_size = data["hidden_size"]
142     output_size = data["output_size"]
143     all_words = data['all_words']
144     tags = data['tags']
145     model_state = data["model_state"]
146 
147     model = NeuralNet(input_size, hidden_size, output_size).to(device)
148     model.load_state_dict(model_state)
149     model.eval()
150 
151     bot_name = "Chatbot"
152     print("Lass quatschen -- Schreib einfach drauf los! (tippe 'ende' zum beenden)")
153 
154     log_uuid = get_new_uuid()
155 
156     log_dialog(log_uuid, 200)
157 
158     while True:
```

```
159         # sentence
160         sentence = input("            Du: ")
161 
162         log_dialog(log_uuid, 400, sentence)
163 
164         if sentence == "ende":
165             log_dialog(log_uuid, 900)
166             break
167 
168         #sentence = tokenize(sentence)
169         X = bag_of_words(sentence, all_words)
170         X = X.reshape(1, X.shape[0])
171         X = torch.from_numpy(X).to(device)
172 
173         output = model(X)
174         _, predicted = torch.max(output, dim=1)
175 
176         tag = tags[predicted.item()]
177 
178         probs = torch.softmax(output, dim=1).to(CPUGPU)
179         prob = probs[0][predicted.item()]
180         probstr = str(int(prob.item() * 100)) + " %"
181 
182         if prob.item() > MIN_PROB:
183 
184             with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
      SRC_FILE), 'r', encoding='utf-8') as json_data:
185                 intents = json.load(json_data)
186 
187             for intent in intents['intents']:
188                 if tag == intent["tag"]:
189                     answer = random.choice(intent['responses'])
190                     print(f"{bot_name} [{probstr}]: {answer}")
191                     log_dialog(log_uuid, 600, answer, prob.item())
192         else:
193             print(f"{bot_name} [{probstr}]: Das habe ich nicht verstanden...")
194             log_dialog(log_uuid, 650, "", prob.item())
195 
196             with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
      OPEN_QUESTIONS_FILE), 'a', encoding='utf-8') as f:
197                 f.write(str(log_uuid) + ";\"" + str(datetime.now()) +
198                         "\";\"" + sentence + "\"\n")
199 
200 
201 def log_dialog(log_uuid, type, log_str="", prob=None):
202     '''
203      50 = uuid created
204      51 = uuid created, no log file present
205     200 = start chat session
206     400 = user input
207     600 = chatbot answer, understood
208     650 = chatbot answer, unshure
209     900 = end chat session with keyword
210     '''
211 
212     with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
        DIALOG_LOG_FILE), 'a', encoding='utf-8') as f:
```

```
213         if prob:
214             f.write(str(log_uuid) + ";" + str(datetime.now()) + ";" +
215                     str(type) + ";\"" + log_str + "\";" + str(prob) + "\n")
216         else:
217             f.write(str(log_uuid) + ";" + str(datetime.now()) +
218                     ";" + str(type) + ";\"" + log_str + "\";\n")
219 
220 
221 def get_new_uuid():
222   while True:
223       new_uuid = str(uuid.uuid4())
224       found = False
225 
226       try:
227           with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
      DIALOG_LOG_FILE), 'r') as f:
228               my_content = csv.reader(f, delimiter=';')
229               for row in my_content:
230                   if new_uuid == row[0]:
231                       found = True
232                       break
233           if not found:
234               log_dialog(new_uuid, 50)
235               return new_uuid
236       except FileNotFoundError:
237           log_dialog(new_uuid, 51)
238           return new_uuid
239 
240 
241 def token_and_stem(text):
242   text = str(text)
243   bag_of_words = []
244   delimiters = ('.')
245   split_char = '+'
246 
247   tokenized_sentence = nltk.tokenize.word_tokenize(text, language='german')
248 
249   for word in tokenized_sentence:
250       stemmed_words = tagger.analyze(word, taglevel=2)
251       stemmed_words = stemmed_words[0]
252 
253       for char in delimiters:
254           stemmed_words = stemmed_words.replace(char, split_char)
255 
256       seperated_stemmed_words = stemmed_words.split(split_char)
257       for single_stemmed_word in seperated_stemmed_words:
258           if len(single_stemmed_word) > 1 and single_stemmed_word not in
      bag_of_words:
259               bag_of_words.append(single_stemmed_word)
260 
261   return bag_of_words
262 
263 
264 def token_and_stem_array(text):
265   #text = str(text)
266   new_text = ""
267   for word in text:
```

```
268         new_text = new_text + str(word) + " "
269     text = new_text
270     bag_of_words = []
271     delimiters = ('.')
272     split_char = '+'
273 
274     tokenized_sentence = nltk.tokenize.word_tokenize(text, language='german')
275 
276     for word in tokenized_sentence:
277         stemmed_words = tagger.analyze(word, taglevel=2)
278         stemmed_words = stemmed_words[0]
279 
280         for char in delimiters:
281             stemmed_words = stemmed_words.replace(char, split_char)
282 
283         seperated_stemmed_words = stemmed_words.split(split_char)
284 
285         for single_stemmed_word in seperated_stemmed_words:
286             if len(single_stemmed_word) > 1 and single_stemmed_word not in
      bag_of_words:
287                 bag_of_words.append(single_stemmed_word)
288     return bag_of_words
289 
290 
291 def bag_of_words(tokenized_sentence, all_words):
292 
293     tokenized_sentence = token_and_stem(tokenized_sentence)
294     bag = np.zeros(len(all_words), dtype=np.float32)
295 
296     for idx, w in enumerate(all_words):
297         if w in tokenized_sentence:
298             bag[idx] = 1
299 
300     return bag
301 
302 
303 def bag_of_words_array(tokenized_sentence, all_words):
304     tokenized_sentence = token_and_stem_array(tokenized_sentence)
305     bag = np.zeros(len(all_words), dtype=np.float32)
306 
307     for idx, w in enumerate(all_words):
308         if w in tokenized_sentence:
309             bag[idx] = 1
310 
311     return bag
312 
313 
314 def train():
315     trian_output = ""
316     with open(os.path.join(os.path.dirname(os.path.abspath(__file__)), SRC_FILE), 'r'
      , encoding='utf-8') as f:
317         dialogus = json.load(f)
318     trian_output = trian_output + \
319         str(f'training starts with contents from file {SRC_FILE}') + os.linesep
320 
321     all_words = []
322     tags = []
```

```
xy = []

for intent in dialogus['intents']:
    tag = intent['tag']
    tags.append(tag)
    for pattern in intent['patterns']:
        words = token_and_stem(pattern)
        all_words.extend(words)
        xy.append((words, tag))

# input()
x_train = []
y_train = []

for (patten_sentence, tag) in xy:
    bag = bag_of_words_array(patten_sentence, all_words)
    x_train.append(bag)

    label = tags.index(tag)
    y_train.append(label)

# input()
x_train = np.array(x_train)
y_train = np.array(y_train)

class ChatDataset(Dataset):
    def __init__(self):
        self.n_samples = len(x_train)
        self.x_data = x_train
        self.y_data = y_train

    # dataset[idx]
    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.n_samples

# Hyperparmeters
batch_size = 8
hidden_size = 8
output_size = len(tags)
input_size = len(x_train[0])  # 1st is all_words
learning_rate = 0.001
num_epochs = 1000

dataset = ChatDataset()
train_loader = DataLoader(
    dataset=dataset, batch_size=batch_size, shuffle=True, num_workers=0)

device = torch.device(CPUGPU)
trian_output = trian_output + str("device: %s" % device) + os.linesep
model = NeuralNet(input_size, hidden_size, output_size)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

```

```
380     for epoch in range(num_epochs):
381         for (words, labels) in train_loader:
382             words = words.to(device)
383             labels = labels.to(dtype=torch.long).to(device)
384 
385             # forward
386             outputs = model(words)
387             loss = criterion(outputs, labels)
388 
389             # backward
390             optimizer.zero_grad()
391             loss.backward()
392             optimizer.step()
393         if (epoch + 1) % 100 == 0:
394             trian_output = trian_output + \
395                 str(f'epoch {epoch+1}/{num_epochs}, loss={loss.item():.4f}') + os.
      linesep
396 
397     trian_output = trian_output + \
398         str(f'final loss, loss={loss.item():.4f}') + os.linesep
399 
400     # save nn
401     data = {
402         "model_state": model.state_dict(),
403         "input_size": input_size,
404         "hidden_size": hidden_size,
405         "output_size": output_size,
406         "all_words": all_words,
407         "tags": tags
408     }
409 
410     torch.save(data, os.path.join(os.path.dirname(
411         os.path.abspath(__file__)), OUT_FILE))
412 
413     trian_output = trian_output + \
414         str(f'training complete. file saved to {OUT_FILE}') + os.linesep
415     return trian_output
416 
417 
418 def main():
419     parser = ArgumentParser(
420         description='Allows the use of a chatbot with pyTorch.')
421 
422     parser.add_argument('-u', '--newuuid',  action='store_true',
423                         help='Get a new, unique id for a conversation.')
424 
425     parser.add_argument('-t', '--train',  action='store_true',
426                         help='Trains the NN model based on current data.')
427 
428     parser.add_argument('-c', '--chat',  action='store_true',
429                         help='Chat with the chatbot.')
430 
431     parser.add_argument('-i', '--textinput',  type=str,
432                         help='Give an text input to the chatbot and recive a answer.
      \
433                             Provide a UUID for the conversation with -cu or --
      conversationuuid \
```

```
                                followed by the UUID.')

    parser.add_argument('-cu', '--conversationuuid',  type=str,
                        help='Needed when using -i, --textinput. Provide a UUID to a
      conversation.')

    args = parser.parse_args()

    if vars(args)['newuuid']:
        print(str(get_new_uuid()))
        quit()

    if vars(args)['train']:
        print(str(train()))
        quit()

    if vars(args)['chat']:
        chat()
        quit()

    # check for complete input for a 'singlechat' call
    complete_text_input = 0
    if vars(args)['textinput']:
        complete_text_input += 1
    if vars(args)['conversationuuid']:
        complete_text_input += 1

    if complete_text_input == 1:
        parser.error(
            "Too few arguments: -cu/--conversationuuid and -i/--textinput are both
      needed.")

    if complete_text_input == 2:
        print(str(singlechat(str(args.conversationuuid), str(args.textinput))))
        quit()

    parser.print_help()


if __name__ == '__main__':
    main()
```

Literaturverzeichnis

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. Irvine: University of California.

Ghimire, D. (2020). Comparative study on Python web frameworks: Flask and Django.

Wangermann, T. (2020). KI in KMU: Rahmenbedingungen für den Transfer von KI-Anwendungen in kleine und mittlere Unternehmen.

Wartena, C. (2019). A probabilistic morphology model for German lemmatization. In *Proceedings of the 15th Conference on Natural Language Processing (KONVENS 2019)* (S. 40–49).

Weizenbaum, J. (1983). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *26*(1), 23–28.

Internetquellen

Bachina, B. (2020). *Vue.js - Local Development With Docker*. Verfügbar unter https://medium.com/bb-tutorials-and-thoughts/vue-js-local-development-with-docker-compose-275304534f7c [29. 08. 2022].

bezkoder (2021). *Vue Axios example with Rest API*. Verfügbar unter https://github.com/python-engineer/pytorch-chatbot [27. 08. 2022].

Bundesministerium für Bildung und Forschung, Referat Künstliche Intelligenz (2022). *Zwischenbericht zur KI-Strategie*. Verfügbar unter https://www.ki-strategie-deutschland.de/home.html?file=files/downloads/Zwischenbericht_KI-Strategie_Final.pdf&cid=806 [27. 08. 2022].

Django Software Foundation (2022). *Django overview*. Verfügbar unter https://www.djangoproject.com/start/overview/ [27. 08. 2022].

heise online, Andreas Wilkens (2016). *Microsofts Chatbot Tay nach rassistischen Entgleisungen abgeschaltet*. Verfügbar unter https://heise.de/-3151646 [16. 06. 2022].

heise online, Martin Holland (2022). *Hat Chatbot LaMDA ein Bewusstein entwickelt? Google beurlaubt Angestellten*. Verfügbar unter https://heise.de/-7138314 [16. 06. 2022].

Loeber, P. (2020). *Implementation of a Contextual Chatbot in PyTorch*. Verfügbar unter https://github.com/python-engineer/pytorch-chatbot [27. 08. 2022].

StackShare, Inc. (2022). *Django - Reviews, Pros and Cons, Companies using Django*. Verfügbar unter https://stackshare.io/django [27. 08. 2022].

You, E. (n. d.). *Vue.js - Local Development With Docker*. Verfügbar unter https://v2.vuejs.org/v2/cookbook/dockerize-vuejs-app.html [29. 08. 2022].

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde/Prüfungsstelle vorgelegen hat. Ich erkläre mich damit nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Wilnsdorf, 30. August 2022
(Ort, Datum)

(Eigenhändige Unterschrift)