

**Masterarbeit zur Erlangung des Grades
Master of Science
im Studiengang Wirtschaftsinformatik**

**Prototypischer Einsatz eines Open Source Large
Language Models im Anwendungskontext einer
LangChain-App**

Jonas Bevernis
Matrikel 16859

Erstgutachter: Prof. Dr. Thomas Wengerek
Zweitgutachter: Prof. Dr. Gero Szepannek

01.08.2023

Hochschule Stralsund
Fakultät für Wirtschaft

Inhaltsverzeichnis

| | |
|--|------------|
| 1. Einleitung und Motivation | 3 |
| 2. Offene Modelle | 5 |
| 2.1. Warum Open-Source-Modelle? | 7 |
| 2.1.1. Probleme von Closed-Source-LLMs | 7 |
| 2.1.2. Lösungen durch Open-Source-Modelle..... | 9 |
| 2.1.3. Probleme von offenen LLMs..... | 11 |
| 2.2. Grundlegende Fachbegriffe | 13 |
| 2.3. Vergleich verschiedener Open-Source-Modelle | 15 |
| 2.3.1. Alleinstellungsmerkmale der Modelle..... | 18 |
| 2.3.2. Leistung..... | 20 |
| 2.3.3. Offenheit..... | 37 |
| 2.3.4. Lizenz | 41 |
| 2.3.5. Entwickler | 45 |
| 2.3.6. Community | 48 |
| 2.3.7. Zusammenfassung und Entscheidung | 51 |
| 3. LangChain..... | 53 |
| 3.1. LLM-Implementierung | 54 |
| 3.2. Prompt | 55 |
| 3.3. Chains..... | 57 |
| 3.4. Agents | 62 |
| 3.5. Memory..... | 64 |
| 3.6. Retrieval Argumented Generation - Vektor-Datenbanken..... | 65 |
| 3.7. Kritik an LangChain..... | 69 |
| 4. Praxisteil | 70 |
| 4.1. Planung..... | 70 |
| 4.1.1. Anforderungen | 70 |
| 4.1.2. Eingrenzung | 71 |
| 4.1.3. Softwarekontext | 72 |
| 4.1.4. Beschreibung der API-Schnittstellen | 73 |
| 4.1.5. Grobarchitektur der Applikation | 75 |
| 4.1.6. Prompt Planung | 77 |
| 4.1.7. LLM Hyper-Parameter | 82 |
| 4.2. Dokumentation der Umsetzung..... | 84 |
| 4.2.1. Beschreibung von Details | 84 |
| 4.2.2. Demonstration der Funktionsweise | 87 |
| 4.2.3. Darstellung der Hardwareseite und Performance..... | 90 |
| 4.2.4. Beschreibung von aufgetretenen Problemen und deren Lösungen | 92 |
| 4.3. Auswertung..... | 95 |
| 4.3.1. Funktionale Auswertung..... | 95 |
| 4.3.2. Schlussfolgerungen aus dem Praxisteil..... | 96 |
| 5. Fazit | 99 |
| Abbildungsverzeichnis | 101 |
| Tabellenverzeichnis | 103 |
| Literaturverzeichnis..... | 104 |
| Selbstständigkeitserklärung..... | 109 |

1. Einleitung und Motivation

Bei Large Language Models (im Folgenden als LLM abgekürzt) handelt es sich um Machine-Learning-Modelle, die anhand enormer Mengen an Textdaten trainiert werden, um menschliche Sprache verstehen und erzeugen zu können. Als solche haben LLMs ein extrem weites Anwendungsfeld, von maschineller Übersetzung von Texten über Fragen-Antwort Systeme und Chatbots, bis hin zur Erzeugung von Programmcode. LLMs gehören in den Bereich der „Natural Language Processing“-Modelle und haben in den vergangenen Jahren enorme Fortschritte gemacht, vorangetrieben vor allem auch von der Firma OpenAI.

Im Jahr 2020 veröffentlichte OpenAI das LLM GPT-3 und eine dazugehörige Arbeit mit dem Titel „Language Models are Few-Shot Learners“ [1]. In dieser wird die Entwicklung des Modells beschrieben. Zwar war GPT-3 nicht das erste Modell seiner Art, aber doch konnte es aufgrund seiner außergewöhnlich guten Leistung erstmalig, auch außerhalb eines kleinen Kreises von Interessierten, auf sich aufmerksam machen. GPT-3 stellt auch abseits der guten Leistung einen Wendepunkt dar. Mit der Veröffentlichung von GPT-3 bricht OpenAI mit ihrem namengebenden Open-Source Konzept. Während frühere Modelle wie GPT-2 noch der Öffentlichkeit kostenlos zur Verfügung gestellt wurden, war GPT-3 erstmalig Closed-Source. Basierend auf einer Weiterentwicklung mit dem Namen GPT-3.5 gelingt OpenAI Ende 2022 schließlich der Durchbruch in den Mainstream, mit der Veröffentlichung von ChatGPT [2]. Zwar waren LLMs auch vor ChatGPT schon in der Lage ähnliche Leistung zu liefern, doch war die Bedienung dieser Modelle meist komplex und mit erheblichem Arbeits- und Hardwareaufwand verbunden. ChatGPT hingegen steht kostenlos zur Verfügung und bietet eine sehr einfache Bedienung. Das Thema ist wochenlang in den Nachrichten und einige Länder, allen voran Italien, verbieten ChatGPT mit dem Verweis auf Datenschutzbedenken [3]. Seitdem hat OpenAI eine abermals verbesserte Version von GPT „GPT-4“ vorgestellt. Insgesamt hat sich die Situation seitdem stabilisiert, es bleibt jedoch der Eindruck, dass OpenAI (und damit Microsoft als einer der Hauptinvestoren) mehr oder weniger ein Monopol über die Technologie besitzt. Doch ist das der Fall? Nicht zuletzt aufgrund der enormen Aufmerksamkeit, die OpenAI auf das Thema ziehen konnte, haben sich eine Menge an Firmen damit beschäftigt, eigene leistungsfähige LLMs zu entwickeln. Einige folgen dabei weiter dem Open-Source-Weg, von dem sich OpenAI mittlerweile abgewendet hat. Mit diesen Open-Source-Modellen soll sich die Arbeit beschäftigen, denn nur wenn die Modelle allen zur Verfügung stehen, kann sichergestellt werden, dass diese Technologie langfristig nicht nur der Bereicherung einiger weniger Unternehmen dient, sondern langfristig zum Wohl der Allgemeinheit eingesetzt werden kann und dass kleinere Unternehmen oder sogar Privatpersonen zur Weiterentwicklung und Verbesserung der Technologie beitragen können.

Doch auch bei den Open-Source Modellen stellen sich viele Fragen. Welche Modelle gibt es überhaupt und von wem stammen diese? Wie verpflichtet sehen sich die Unternehmen, welche die Modelle bereitstellen, dem Open-Source-Konzept gegenüber? OpenAI hat bereits vorgemacht, wie schnell es gehen kann, dass ein Open-Source-Projekt auf einmal Closed-Source wird. Auch stellt sich die Frage, wie die Leistung der verschiedenen Modelle aussieht. Welche Vor- und Nachteile bieten die Modelle? All dies soll im Folgenden untersucht werden.

Neben diesen sehr theoretischen Aspekten gibt es noch einen weiteren Punkt zu betrachten. Den Vorgang, um LLMs zur Lösung eines echten Problems in einer Software einzusetzen.

1. Einleitung und Motivation

Hier spielt der zweite große Aspekt dieser Arbeit eine Rolle: LangChain. LangChain ist eine erstmals Ende 2022 veröffentlichte Softwarebibliothek für Python und JavaScript, welche eine Vielzahl an Möglichkeiten bietet, mit LLMs zu interagieren. LangChain soll es vereinfachen LLMs in seinen Code zu integrieren, indem es eine große Menge an Tools bereitstellt, die das Interagieren mit den LLMs abstrahieren. Aufgrund des sehr breiten Anwendungsfelds von LLMs, ist auch LangChain entsprechend breit aufgestellt und bietet eine riesige Menge an Möglichkeiten. Diese darzustellen, soll im zweiten Teil der Arbeit im Mittelpunkt stehen.

Schließlich sollen beide Aspekte (Open-Source-Modelle und LangChain) im dritten Teil verbunden werden. Zu diesem Zweck wird eine beispielhafte Applikation entworfen und entwickelt, welche LangChain verwendet, um ein ausgewähltes Open-Source-LLM in den Code der Applikation zu integrieren. Die enorme Breite der Anwendungsmöglichkeiten von LLMs und LangChain macht es nahezu unmöglich, sämtliche Möglichkeiten der beiden Technologien in einer einzigen Applikation darzustellen, ohne dabei den Rahmen dieser Arbeit zu sprengen. Die Applikation dient damit nur als Beispiel, für die Umsetzung eines konkreten Anwendungsfalls und gilt nicht als allgemeines LangChain Beispiel. Konkret soll der Anwendungsfall „Verwendung von natürlicher Sprache zur Bedienung einer API“ dargestellt werden, indem eine Art Smart-Home Assistent entwickelt wird, welcher in der Lage ist, bestimmte Smart-Home Geräte zu steuern, die über eine REST-API ansprechbar sind.

Insgesamt soll diese Arbeit die Frage klären „Welche Open-Source LLMs gibt es, wie sind diese einzuordnen und wie können diese verwendet werden, um echte Probleme zu lösen?“. Dazu soll zuerst ein Überblick über verschiedene Open-Source-LLMs gegeben werden. Anschließend wird mit LangChain eine Softwarebibliothek vorgestellt werden, die das Integrieren in Code vereinfachen will. Schließlich werden beide Aspekte verbunden, um beispielhaft darzustellen, wie LLMs und LangChain verwendet werden können, um echte Aufgaben bzw. Probleme zu lösen.

Sämtlicher, im Zuge dieser Arbeit erzeugter, Code befindet sich dabei in einem GitHub Repository, unter „<https://github.com/stasj145/Masterarbeit-OS-LangChain-App>“ [4].

2. Offene Modelle

Zu Beginn dieses Kapitels gilt es klarzustellen, was ein Open-Source-Modell ausmacht und was im Kontext dieser Arbeit unter diese Bezeichnung fällt. Das traditionelle Open-Source-Verständnis trifft auf Machine-Learning Modelle nicht unbedingt zu. Bei klassischer Open-Source-Software wird der Quellcode, zum Beispiel über GitHub, zur Verfügung gestellt, auf Basis dessen, prinzipiell jeder Mensch die Software selbst kompilieren und nutzen kann. Diese Betrachtung trifft auf LLMs aber nicht immer zu. Das Komplexere an der Erstellung eines neuen LLMs ist primär nicht das Schreiben des Trainingscodes, sondern vielmehr das Entwerfen bzw. Anpassen einer Modellarchitektur, das Finden und korrekte Einsetzen von passenden Datensätzen sowie das Durchführen des sehr Hardware- und Kosten aufwendigen Trainings. Wenn von einem offenen bzw. Open-Source Modell geredet wird, sind damit teilweise Modelle gemeint, bei dem der Trainings-Quellcode oder Trainings-Datensatz nicht frei verfügbar ist, sondern nur der Code zum Weiterverwenden der Modelle und die Modelle an sich. Frei verfügbar sind dann folgende Aspekte:

- Die Architektur: Diese wird in der Regel in einer zum Modell gehörigen Arbeit beschrieben und veröffentlicht.
- Das Modell: Dies wird meist in Form von sogenannten „Checkpoint“ zur Verfügung gestellt.
- Der Code zum Verwenden der Modelle.

Damit fallen einige offene Modelle nicht unter die klassische Open-Source Definition, wie sie von der „Open Source Initiative“ [5] dargestellt wird, sondern orientieren sich am Open-Access Konzept. Bei Open-Access Software steht vorwiegend nur das Endprodukt (bei einem Windows-Programm zum Beispiel eine exe Datei) zur freien Verfügung. Anders als bei klassischer „Freeware“, ist das Arbeiten am Endprodukt, Anpassen und auch Verbreiten dessen, bei Open-Access in der Regel erlaubt. Wichtig ist zudem, dass ein LLMs kein Endprodukt darstellt, sondern vielmehr ein Zwischenprodukt. Ein LLM ist keine eigenständige Software, sondern eine Menge an komplexen Konfigurationsdateien, welche von externer Software gelesen und verarbeitet werden können, um ein nutzbares LLM zu erzeugen. Es könnte argumentiert werden, dass der Trainingscode nicht Teil des LLMs ist, sondern nur das Werkzeug, welches verwendet wird, um die entsprechenden Konfigurationsdateien zu erzeugen, die ein LLM ausmachen. Diese Konfigurationsdateien wären nach dieser Argumentation der „Quellcode“ des LLM. Dazu kommt der Fakt, dass die Struktur dieser Modelle offen ist, da diese für den Betrieb der Modelle benötigt wird. So könnte argumentiert werden, dass ein LLM durchaus als Open-Source bezeichnet werden kann, auch wenn es die Tools, die verwendet wurden um das LLM zu erzeugen, nicht sind. Anders als bei normaler Software, bei der es weitestgehend unmöglich ist, sie von ihrem Quellcode zu trennen, ist es bei einem LLM (oder auch jedem anderen Machine-Learning-Modell) diskussionswürdig, ob der Trainingscode Teil des LLMs ist oder nicht. Ob ein LLM, welches frei verfügbar ist, dessen Trainingscode es aber nicht ist, als Open-Source bezeichnet werden kann, ist Auslegungssache. Dies zeigt, dass die bisherige starre Interpretation von Open-Source, für neue Techniken, wie Machine-Learning-Modelle, möglicherweise veraltet ist.

Zusätzlich wird dies noch verkompliziert, da die Projekte unterschiedlichen Lizenzen unterliegen können, die das Verbreiten und Verwenden des Quellcodes einschränken. Das Gleiche gilt für Open-Access Projekte. Trotzdem werden solche Projekte oft

2. Offene Modelle

umgangssprachlich als „Open-Source“ oder „Offen“ bezeichnet (Obwohl sich die Open-Source Initiative klar gegen diese Verwendung des Open-Source Begriffs in dieser Art stellt). Das Thema ist komplex, sowie Open-Source als auch Open-Access sind nur Überbegriffe, die jeweils eine breite Menge an verschiedenen Projekten umfassen können und einen fließenden Übergang haben. Ziel dieser Arbeit ist es nicht, jedem Projekt einen konkreten Überbegriff zuzuordnen, sondern vielmehr die verschiedenen offenen Modelle miteinander zu vergleichen. Deshalb sollen zum Zweck dieser Arbeit auch Modelle, die in Richtung Open-Access tendieren, betrachtet werden. Dabei wird jeweils eindeutig dargestellt, welche Aspekte der jeweiligen Modelle tatsächlich offen sind und welche nicht. Die Untersuchung genau dieser Problematik ist auch eines der Teilziele dieser Arbeit. Am Ende wird der Aspekt „Offenheit“ für die finale Modellbewertung eine Rolle spielen.

In diesem ersten Teil der Arbeit sollen verschiedene Large Language Models untersucht werden. Zuerst soll dargestellt werden, warum der Fokus dieser Arbeit auf Open-Source Modellen liegt, indem die Nachteile von Closed-Source Modellen herausgearbeitet werden. Schließlich wird beschrieben, wie Open-Source Modelle diese Probleme beheben können. Gleichzeitig wird erneut das Problem „Offenheit“ aufgegriffen.

Anschließend sollen verschiedene Open-Source Modelle konkret miteinander verglichen werden, indem einerseits die Alleinstellungsmerkmale der Modelle dargestellt, aber auch die Unternehmen hinter den Modellen betrachtet werden. An dieser Stelle wird auch die Frage „Wie offen sind die Modelle wirklich?“ geklärt. Es gilt zu beachten, dass nur eine (relative kleine) Auswahl an Modellen betrachtet werden kann. Die Internetseite HuggingFace.co [6] dient, vergleichbar mit GitHub, als Sammlung für Open-Source/Open-Access Machine-Learning-Modelle. Unter der Kategorie „Text Generation“ stehen auf HuggingFace über 22.000 Modelle zur Verfügung (stand 09.09.2023). Bei einer so enormen Menge an Modellen ist es unmöglich alle, oder auch nur einen Bruchteil aller Modelle, zu betrachten. Die Anzahl an Modellen ist vordergründig auf zwei Aspekte zurückzuführen. Erstens werden Modelle meist in verschiedenen Größen und Versionen veröffentlicht. Zweitens gibt es viele Nutzer, die versuchen, auf Basis eines anderen Modells, eine neue verbesserte Version des Modells zu erzeugen und diese dann wiederum bei HuggingFace bereitstellen. Folgendes Beispiel veranschaulicht, wie das aussieht: eines der beliebtesten Modelle ist Meta's LLaMA-2 (mehr zu diesem Modell später im Verlauf des Kapitels). LLaMA-2 gibt es in zwei verschiedenen Versionen. Die Standardversion, sowie eine „Chat“ Version, die auf Chat-Assistenten abgestimmt wurde. Von beiden Versionen gibt es jeweils noch „HF“ Varianten, die speziell für das Arbeiten mit HuggingFace Tools gedacht sind. Das sind bereits 4 Modelle, die aber alle zur gleichen Familie gehören. Jedes dieser 4 Modelle gibt es zusätzlich in verschiedenen Größen: 7B, 13B und 70B. Diese Bezeichnungen beschreiben jeweils die Anzahl an Modellparametern. Grob gilt, je mehr Parameter ein Modell hat, desto besser ist seine Leistung, aber umso mehr Hardware wird auch benötigt, um das Modell zu benutzen. Das sind 12 Modelle. Dazu kommen noch die anderen HuggingFace Nutzer, welche potenziell von jeder der 12 Modelle eigene Versionen mit kleinen Änderungen entwickeln und zur Verfügung stellen.

So können aus einem beliebten Grundmodell schnell hunderte Modelle auf HuggingFace entstehen. Alle diese Modelle sind aber nur unterschiedliche Ausprägungen des gleichen Grundmodells. Wird in der Kategorie „Text Generation“ nach „Llama-2“ gesucht, können über 1.900 Modelle gefunden werden. Unter den 22.000 verschiedenen „Text Generation“ Modellen gibt es dementsprechend eine bedeutend kleinere Anzahl an Modell-Familien, die

2. Offene Modelle

sich im Grund auf wenige Hauptmodelle herunterbrechen lassen. Die Untersuchung der verschiedenen Modellableger wäre ebenfalls interessant. Um den Rahmen der Arbeit einzuhalten, soll im Folgenden in erster Linie nur je ein Ableger der verschiedenen Familien untersucht werden. Auf Basis dieser Untersuchung soll am Ende eine Auswahl getroffen werden, welches Modell für die Entwicklung der Applikation genutzt wird.

2.1. Warum Open-Source-Modelle?

Allgemein ist es nur selten gut für die Allgemeinheit, wenn eine oder wenige Firmen die alleinige Kontrolle über eine Technologie besitzen. In diesem Fall gibt es nur wenig Konkurrenz und Firmen sehen sich nicht im Zugzwang, ihre Produkte zu verbessern oder die Preise zu senken. Aufgrund der enormen Kosten für Training eines LLM, können es sich nur große Unternehmen oder Start-ups mit signifikanten Investments leisten auf dem LLM-Markt zu konkurrieren. Dadurch ist die Konkurrenz auf dem Markt gering. Das Veröffentlichen von Freien bzw. Open-Source LLMs ist der einzige Weg diesen Quasi-Monopolen entgegenzuwirken.

In diesem Kapitel soll dargestellt werden, welche Probleme geschlossene Modelle mit sich bringen und wie offene Modelle diese Probleme lösen können. Zuletzt wird dargestellt, welche Probleme auch von Open-Source Modellen ausgehen können.

2.1.1. Probleme von Closed-Source-LLMs

Die Closed-Source Natur der bekannten Modelle wie GPT-3/GPT-4 oder Google's PaLM-2 bringen eine Vielzahl an potenziellen Problemen mit sich. Das ist zuerst der bereits erwähnte Aspekt „Verfügbarkeit“. Spezifische LLMs werden nur von einem bestimmten Unternehmen angeboten. Soll mit GPT-4, dem aktuell stärksten LLM, gearbeitet werden, gibt es dafür nur einen Ansprechpartner: OpenAI (Genauer gesagt auch Microsoft. Aufgrund Microsofts \$1 Mrd. Investment im Jahr 2019 [7] und des erneuten Investments von Anfang 2023, welches sich laut Bloomberg auf \$10 Mrd. belaufen soll [8], kann bei OpenAI und Microsoft nicht von Konkurrenten gesprochen werden). Damit ist der Kunde dem Anbieter des Modells ausgesetzt.

Aufgrund dieser Bindung an ein Unternehmen kann es schnell zu einem weiteren Problem kommen, dem sogenannten „Vendor-Lock“ Effekt. Dieser Effekt beschreibt die Situation, indem ein Unternehmen oder eine Person derartig abhängig von der konkreten Funktionsweise eines Produkts ist, dass ein Wechsel zu einem Konkurrenzprodukt sehr aufwendig oder teuer wäre und in der Regel keinen Sinn ergibt. Durch die proprietäre Natur von Closed-Source Systemen kann so schnell eine Situation entstehen, in der neue Alternativen, trotz eines möglicherweise besseren Produkts, nicht mit den etablierten Anbietern konkurrieren können. Allerdings sei erwähnt, dass ein solcher „Vendor-Lock“ im LLM-Bereich bislang kaum vorhanden ist. Jedoch liegt es in der Natur dieses Effekts, dass er sich über einen längeren Zeitraum hinweg entwickelt. Dies konnte, aufgrund dessen, dass LLMs noch vergleichsweise neu sind, bisher nicht stattfinden.

Das Ausnutzen dieses Effekts ist ein wichtiger Aspekt, durch den Monopole ihre Machtstellung aufrechterhalten und sollte deshalb als Nachteil von geschlossenen LLMs nicht vernachlässigt werden.

Ein weiteres Problem ist der Aspekt „Datenschutz“. Das Sammeln, Speichern und Verarbeiten von personenbezogenen Daten unterliegt in der EU strengen Richtlinien, welche

2. Offene Modelle

primär in der bekannten Datenschutz-Grundverordnung (kurz. DS-GVO) geregelt werden. Möchte ein Unternehmen etwa ein Closed-Source LLM in ein Produkt einbauen, kann das nicht ohne Weiteres geschehen. Denn dabei werden potenziell personenbezogene Daten an Dritte weitergegeben. Dem müssen die Kunden ausdrücklich zustimmen.

Neben dem klassischen Datenschutz spielt noch ein anderer, aber sehr ähnlicher, Punkt eine Rolle: „Schutz unternehmensinterner Daten“. Je nach Einsatzzweck funktionieren LLMs am besten, wenn ihnen möglichst viele relevante Daten zur Verfügung stehen. Soll ein LLM beispielsweise die eigenen Softwareentwickler möglichst gut beim Schreiben von Code unterstützen, ist es sinnvoll, dem LLM Zugriff auf die hauseigene Codebibliothek und Dokumentation zu verschaffen. Wird ein Closed-Source LLM verwendet, bedeutet dies automatisch auch, dass dem Unternehmen, welches das LLM betreibt, Zugriff auf diese unternehmensinternen Daten gegeben wird. Genau solche Fälle sind mittlerweile nicht mehr nur Theorie, sondern sind schon in der Praxis aufgetreten. Anfang April 2023 berichteten verschiedene Medien mit Bezug auf die südkoreanische Seite „economist.co.kr“ von einem Leak firmeninterner Daten beim Halbleiterhersteller „Samsung Electronics“ [9]. Den Berichten zufolge nutzten Angestellte, OpenAIs ChatGPT, um Bugs im Code zu beheben oder Präsentationen mit sensiblen Daten vorzubereiten. Eine solche Weitergabe von firmeninternen, potenziell streng geheimen, Daten an Dritte, gilt es natürlich zu vermeiden.

In beiden Fällen „Datenschutz“ und „Schutz unternehmensinterner Daten“ gilt es auch die konkreten Anwendungsfälle zu betrachten sowie die Nutzungsbedingungen der Anbieter und Produkte genau zu untersuchen. Nutzerdaten werden laut eigenen Angaben von OpenAI bei ChatGPT beispielsweise zum Training und zur Verbesserung des Modells verwendet, während ihr neueres Produkt „ChatGPT Enterprise“ das nicht tun soll [10]. Auch OpenAI's API Optionen sind laut eigenen Angaben sicher und geben OpenAI keinen Zugriff auf übermittelte Daten [11]. In jedem Fall werden die Daten zunächst an ein anderes Unternehmen weitergegeben und darauf vertraut, dass dieses seine Versprechen hinsichtlich der Datensicherheit einhalten. Abhängig davon, wie sensible die zu übermittelnden Daten sind, muss am Ende entschieden werden, welche Optionen in Betracht kommen.

Ein weiterer wichtiger, aber auf den ersten Blick weniger auffälliger, Aspekt ist das Problem „Zensur und Ethik“. Ist das Modell Closed-Source hat der Anbieter grundlegend die volle Kontrolle über das LLM und dessen Ausgaben. Speziell bei ChatGPT gibt es immer mehr Beschwerden über die zunehmende Zensur der Antworten [12]. Eine solche Zensur scheint es aktuell noch weniger zu geben, wenn über die entsprechenden APIs direkt auf die LLMs zugegriffen wird. Es wäre aber für die Anbieter nicht schwer, bei den API-Zugriffen eine verstärkte Zensur einzubauen. Der Kunde ist erneut darauf angewiesen, dass die Hersteller keine plötzlichen Änderungen vornehmen und dass Zensur, wenn sie denn eingesetzt wird, einen gewissen ethischen Rahmen einhält. Die Bereitsteller der LLMs besitzen damit eine enorme Macht und Verantwortung. Durchaus kann argumentiert werden, dass das Zensieren gewisser Ausgaben von LLMs, ein sinnvolles Ziel verfolgt. In der Arbeit „Truth, Lies, and Automation - How Language Models Could Change Disinformation“ [13] stellen die Autoren eindrucksvoll dar, wie einfach es ist, mithilfe von LLMs (in dem Fall GPT-3) Desinformationskampagnen fast vollautomatisch zu erzeugen und durchzuführen oder Texten relativ subtil einen anderen politischen Ton zu geben, als vom ursprünglichen Autor vorgesehen war. Diese Zensur, oder viel mehr Nutzungseinschränkung, allein in die Hände

2. Offene Modelle

profitorientierter Unternehmen zu legen, ist dennoch wenig zielführend und sollte vor allem durch Gesetze reguliert werden.

Zuletzt gibt es noch die Kostenproblematik. Dabei geht es vorerst weniger darum, wie hoch die Kosten für die Nutzung eines Closed-Source-Modells zum heutigen Zeitpunkt sind. Das Trainieren und Betreiben von LLMs ist, aufgrund des enormen Hardwareaufwands, teuer und dies spiegelt sich auch in den Kosten für die Nutzung der Modelle wider. Je nach Anwendungsfall sind die Kosten zur Nutzung entsprechender LLMs sogar vergleichsweise niedrig, wenn diese im Verhältnis zum Hardwareaufwand beim Anbieter betrachtet werden. Das Problem ist vielmehr, dass es keinerlei Kontrolle über diese Kosten gibt. Im Normalfall wird pro Token der Ein- und Ausgabe gezahlt. Wie lang ein Token ist, hängt prinzipiell vom Aufbau des LLMs ab. Bei den meisten modernen LLMs ist ein Token in etwa 4 Zeichen lang [14] [15]. Ein Text mit 4000 Zeichen wären also 1000 Tokens. Den Preis pro Token bestimmen die Anbieter und diese können auch nach Belieben den Preis ändern. Natürlich gibt es immer gewisse Abhängigkeiten von der Preisgestaltung anderer Unternehmen. Aber durch die sehr direkte Art, in der für die Nutzung der LLMs bezahlt wird, können sich auch vermeintlich kleine Preisänderungen massiv auf die Betriebskosten einer Software auswirken, die ein entsprechendes LLM verwendet.

2.1.2. Lösungen durch Open-Source-Modelle

Nachdem die verschiedenen Probleme von Closed-Source Modellen dargestellt wurden, stellt sich die Frage, wie Open-Source Modelle diese Aspekte lösen oder zumindest verbessern.

Das Thema „Verfügbarkeit“ kann durch offene Modelle vollständig gelöst werden. Während bei einem Closed-Source Modell eine Abhängigkeit von einem einzigen Anbieter besteht, gibt es bei offenen Modellen meist eine Vielzahl an alternativen Versionen des gleichen Modells, die von anderen leicht angepasst und veröffentlicht wurden. Dieses Konzept wurde im Kapitel „2. Offene Modelle“ bereits beschrieben. Selbst wenn beim verwendeten LLM eine negative Änderung ansteht, ist ein Wechsel zu einem anderen Modell, welches zur gleichen Modellfamilie gehört, vergleichsweise einfach. Alternativ kann auch beim alten Modell geblieben werden. Anders als bei Closed-Source Modellen, über die nur der Bereitsteller Kontrolle hat, kann ein offenes Modell eigenständig heruntergeladen und verwendet werden. Änderungen können nicht aufgezwungen werden. Bei diesem Punkt sei erneut darauf hingewiesen, dass nicht alle Modelle gleich offen sind. Es gilt, sich im Vorhinein über die Lizenzen zu informieren, denen die verschiedenen Modelle unterliegen. Diese können gegebenenfalls die Verbreitung oder auch die kommerzielle Nutzung einschränken.

Auch der Aspekt „Vendor-Lock“ kann durch Open-Source Modelle verbessert werden. Wie bereits beschrieben gibt es meist viele verschiedene Versionen eines Modells von verschiedenen Entwicklern, die aber alle zur gleichen Modellfamilie gehören. Ein Vendor-Lock Effekt wird damit deutlich schwieriger als bei geschlossenen Modellen.

Allerdings ist das Auftreten eines solchen Effekts auch durch offene Modelle nicht vollständig auszuschließen. Es gibt zwar viele unterschiedliche Modelle, aber nur vergleichsweise wenige Modellfamilien. Prinzipiell wäre es möglich, dass ein Vendor-lock, nicht in Bezug auf ein einzelnes Modell, sondern in Bezug auf eine Modellfamilie, auftritt. Damit gibt es dennoch eine gewisse Abhängigkeit von dem Unternehmen, welches das

2. Offene Modelle

Grundmodell einer Familie entwickelt. Im Vergleich zu geschlossenen Modellen ist die Gefahr aber geringer, auch weil es seitens der Entwickler weniger Anreize gibt, den Vendor-Lock Effekt auszunutzen, da die Modelle ohnehin kostenlos zur Verfügung stehen.

Das Thema „Datenschutz“ muss, gerade in der EU, sehr sorgfältig behandelt werden. Im Gegensatz zu Closed-Source LLMs, bei denen die Funktionsweise und die Datenverarbeitung meist ein Geheimnis bleiben und auf die Versprechen der Anbieter vertraut werden muss, sind Open-Source Modelle transparenter. Bei offenen Modellen kann der Datenfluss genau nachvollzogen werden, da die Bereitstellung des Modells eigenständig übernommen wird. Indem ein Unternehmen ein Open-Source LLM auf eigenen Servern betreibt, kann sichergestellt werden, dass personenbezogene Daten nicht an Dritte weitergegeben werden. So wird das Risiko für Datenschutzverletzungen minimiert. Allerdings gilt es auch hier die geplante Infrastruktur vorher gut zu überdenken. Offene Modelle lösen das Problem jedoch nicht automatisch, sie ermöglichen es lediglich LLMs zu verwenden, ohne dabei das eigene Netzwerk zu verlassen. Durch die enorme Beliebtheit von LLMs, gibt es inzwischen eine Vielzahl an Anbieter, die es möglich machen, offene Modelle über eine API des Anbieters zu verwenden. So gibt es für Kunden kaum Einrichtungsaufwand und geringe anfängliche Kosten. Allerdings ist diese Struktur von der eines Closed-Source Anbieters kaum noch zu unterscheiden und es werden potenziell wieder Daten an Dritte weitergegeben, zumindest aber muss erneut das Vertrauen in den Anbieter überprüft werden.

Der „Schutz unternehmensinterner Daten“ ist auf eine sehr ähnliche Weise mithilfe von offenen Modellen umsetzbar, wie soeben für den Punkt „Datenschutz“ beschrieben. Durch die Möglichkeit ein offenes LLM auf eigener Hardware zu betreiben, kann sichergestellt werden, dass die geheimen internen Daten nicht das eigene Netzwerk verlassen. Zum gleichen Schluss kam auch „Samsung Electronics“, die auf das im letzten Kapitel beschriebene Datenleck mit dem Einrichten eines eigenen internen LLM-Chat-Services für die interne Nutzung reagierten [9].

Offene LLMs lösen die „Zensur-Problematik“ nur zum Teil. Normalerweise findet die Zensur einer LLM-Ausgabe nach der eigentlichen LLM-Ausführung statt. Im Fall eines LLM-Chatsystems wäre dann nicht das LLM als solches zensiert, sondern der Input des Nutzers sowie die LLM-Ausgabe wird durch speziellen Quellcode angepasst und überprüft, bevor er dem LLM übergeben wird, bzw. dem Nutzer eine Ausgabe angezeigt wird. Genau diese Art von Zensur der LLM-Outputs lässt sich durch offene Modelle umgehen. Wird das LLM eigenständig betrieben, besteht volle Kontrolle über die Ein- und Ausgaben des Modells. Es ist ebenfalls möglich, ein LLM auf andere Arten zu zensieren. So können beim Training eines Modells gezielt bestimmte Themen aus dem Datensatz gestrichen. Das wäre zwar recht aufwendig, ist aber absolut möglich und vor allem deutlich subtiler als die zuvor beschriebene Art der Zensur. Das Modell hat dann über bestimmte Thematiken auch weniger oder keine Informationen. Zusätzlich wäre es auch möglich, Teile des Datensatzes nicht zu löschen, sondern umzuschreiben, sodass sie dem eigenen Weltbild entsprechen. Vor dieser Art der Zensur und Manipulation sind auch offene Modelle nicht sicher.

Die ursprünglichen Entwickler haben die Kontrolle. In der Theorie bieten offene Modelle eine Lösung. Ein offenes Modell kann durch zusätzliches Training angepasst werden, so können dem Modell neue Informationen antrainiert werden, die es vorher nicht hatte. Das Problem ist, dass eine Zensur in dieser Art vergleichsweise subtil ist und möglicherweise nicht auffällt. Prinzipiell bieten Open-Source Modelle gute Grundlagen, um Zensur zu

2. Offene Modelle

entgehen, es gilt aber trotzdem zu untersuchen, von wem das LLM entwickelt wird und welche Ziele die Entwickler des Modells möglicherweise verfolgen.

Das Thema „Kosten“ ist komplex, da diese stark von verschiedenen Aspekten abhängig sind. Der offensichtliche Vorteil von Open-Source Modellen ist, dass diese zuerst einmal kostenlos sind. Jedermann kann sich ein offenes Modell herunterladen und verwenden, ohne dafür zu bezahlen. Allerdings müssen diese Modelle betrieben werden und die dafür benötigte Hardware verursacht wiederum Kosten. Was am Ende konkret günstiger ist, hängt vom Anwendungsfall ab. Wie zuvor beschrieben, sind ein wichtiger Aspekt vor allem nicht die totalen Kosten, sondern vielmehr die Kostenabhängigkeit. Speziell diese Abhängigkeit kann mit offenen Modellen umgangen werden. Ist die Hardware erstmal im Besitz, kann sich am Preis für den Betrieb des eigenen LLMs nichts mehr ändern (von Stromkosten abgesehen, welche gerade in Deutschland auch sehr relevant sein können). Möglicherweise wird die Hardware auch in einem Rechenzentrum gemietet, aber auch dann trifft der Vorteil zu. Zwar kann das Rechenzentrum die Mietpreise für die Hardware erhöhen, aber in dem Fall ist ein Wechsel zu einem anderen Hardwareanbieter deutlich einfacher als der Wechsel zu einem vollkommen anderen LLM. Damit besteht die Abhängigkeit, von der Kostengestaltung der LLM-Anbieter, durch das Verwenden offener Modelle nicht mehr.

Natürlich können Open-Source LLMs nicht alle Probleme von LLMs lösen. Doch gibt es aber eine Vielzahl an Problemen, die speziell Closed-Source LLMs betreffen und bei denen offene Modelle eine Lösung bieten.

2.1.3. Probleme von offenen LLMs

Wie zuvor beschrieben bieten Open-Source-LLMs eine Vielzahl an Vorteilen. Allerdings sind auch Open-Source Modelle nicht frei von Problemen. Es ist wichtig, diese zu verstehen, um entscheiden zu können, ob ein offenes Modell für einen gewählten Anwendungsfall Sinn ergibt.

Einer der größten Nachteile von Open-Source Modellen ist die Einrichtung. Bei einem kommerziellen Closed-Source Modell ist nahezu keine Einrichtung nötig. Es muss lediglich ein Account angelegt, ein Zahlungsmittel hinterlegt und ein API-Token erstellt werden. Bei offenen Modellen müssen die LLMs normalerweise selbst gehostet werden. Zu diesem Zweck muss Hardware gekauft oder gemietet (wenn diese nicht bereits existiert) und dann für die LLM-Nutzung eingerichtet werden. Schließlich muss das LLM heruntergeladen und auf der Hardware ausgeführt werden. Prinzipiell kann die Einrichtung auch übersprungen (oder wenigstens deutlich vereinfacht) werden, indem ein Service genutzt wird, der das Hosten des LLMs übernimmt. Damit gehen aber auch, wie zuvor bereits beschrieben, einige der Vorteile verloren, die offene Modelle mit sich bringen.

Der zweite große Nachteil ist der Aspekt „Leistung“. Dieser Aspekt ist in zwei Teilaspekte zu unterteilen. Erstens „Textqualität“, die Leistung des LLMs, also die Qualität des Texts, der von dem LLM erzeugt wird. Und zweitens „Geschwindigkeit“, die Leistung der Software, die das LLM bereitstellt, also wie schnell der Text erzeugt werden kann. Diese Aspekte unter verschiedenen Modellen zu vergleichen ist nicht trivial und wird später, im Kapitel „2.3.2. Leistung“ noch genauer betrachtet. Ganz allgemein gilt der Grundsatz, dass die Textqualität mit der Anzahl an Modellparametern skaliert. Ein 7B Modell (7 Milliarden Parameter) wird eine schlechtere Textqualität bieten als ein 70B Modell (70 Milliarden Parameter). Wird die Größe von offenen Modellen mit geschlossenen Modellen verglichen, wird festgestellt, dass

2. Offene Modelle

selbst die größten offenen Modelle nur einen Bruchteil der Parameter der geschlossenen Modelle aufweisen. In Wirklichkeit gibt es neben der Parameteranzahl noch mehr Aspekte, welche die Textqualität bestimmen. So sagte OpenAI's CEO Sam Altman in einem Interview, Mitte 2023 zum Beispiel, er gehe davon aus, dass die weitere Verbesserung von LLMs in Zukunft nicht mehr in erster Linie durch das einfache Vergrößern der Modelle, sondern mit anderen Mitteln erreicht werden wird [16]. Es bleibt ein Fakt, dass die größten offenen Modelle deutlich weniger komplex sind als die größten geschlossenen Modelle. Ob dies für den eigenen Anwendungsfall eine Rolle spielt, muss für jeden Einzelfall neu entschieden werden. Im Vergleich ist die Geschwindigkeit ein kleineres Problem. Werden Standardtools, wie die HuggingFace Bibliotheken, verwendet, um ein offenes Modell einzusetzen, wird die Geschwindigkeit von Closed-Source Modellen kaum erreicht werden. Das ist weniger ein Fakt der LLMs an sich, sondern vielmehr der Tatsache, dass Firmen wie OpenAI oder Google dedizierte Teams haben. Diese sorgen dafür, dass Nutzern eine hervorragende Geschwindigkeit geboten wird. Allerdings gibt es auch auf der Open-Source Seite verschiedene Projekte, welche die Geschwindigkeit von LLMs deutlich verbessern können. Durch die Nutzung dieser kann die Geschwindigkeit ebenfalls auf ein sehr gutes Niveau angehoben werden. Zudem ist dabei die geringere Parameteranzahl hilfreich. Kleinere Modelle sind schneller in der Ausführung als die riesigen Closed-Source Modelle. Damit ist der große Aspekt „Leistung“ insgesamt ein Nachteil von offenen Modellen. Abhängig vom Anwendungsfall muss dieser Nachteil allerdings nicht relevant sein.

Ein weiterer Aspekt, der je nach Anwendungsfall auch negativ sein kann, sind die Hardwarekosten. Der Kostenaspekt wurde auch bei Closed-Source Modellen schon als Negativpunkt aufgelistet, es mag widersprüchlich klingen, diesen Punkt auch hier wiederzufinden. Das liegt hauptsächlich daran, dass die Kosten sehr anwendungsabhängig sind. Bei Closed-Source Modellen wird per Token bezahlt. Bei größeren Projekten können schnell gewaltige Mengen an Tokens zusammenkommen und es wird teuer. Ein offenes Modell ist zwar zunächst kostenlos, aber um ein Open-Source Modell einzusetzen, wird leistungsfähige Hardware benötigt. Diese zu kaufen oder zu mieten kann kostspielig werden, insbesondere wenn sehr große und komplexe LLMs verwendet werden sollen. Dafür kann diese Hardware beliebig viel genutzt werden und die Kosten sind unabhängig von der Anzahl an Token, die verarbeitet werden. Müssen aber nur wenige LLM-Anfragen verarbeitet werden, können die Hardwarekosten schnell höher werden als die Kosten bei Closed-Source Modellen.

Zuletzt ist der Aspekt „Lizenzen“ zu beachten. Auch bei der Verwendung von Closed-Source Modellen gilt es, die Lizenzbedingungen der verwendeten Produkte zu beachten. Bei Open-Source Modellen ist dies besonders wichtig. Die offene Natur der Modelle lässt schnell vermuten, dass diese keinerlei Nutzungseinschränkungen bieten. In Realität unterliegen die Modelle, genau wie andere Open-Source-Projekte auch, verschiedenen Lizenzen, die zum Beispiel eine kommerzielle Nutzung untersagen könnten. Deshalb gilt es bei offenen Modellen besonders auf diesen Aspekt Rücksicht zu nehmen und bei der Modellwahl zu beachten.

2. Offene Modelle

2.2. Grundlegende Fachbegriffe

Bevor der Vergleich durchgeführt werden kann, ist es wichtig einige Grundlagen zu klären, die für ein besseres Verständnis der kommenden Kapitel notwendig sind.

Transformer: Die Transformer-Struktur wurde bereits ins Detail in der vorausgehenden Bachelorarbeit „Funktionsweise, Anwendungen und Risiken von Texterzeugungsmodellen am Beispiel von GPT“ [17] beschrieben. Deshalb gibt es hier nur eine stark vereinfachte Übersicht. Ein Transformer ist eine Art von Machine-Learning Modell, welches zu dem Bereich der „Sequenz-to-Sequenz“ oder kurz „Seq2Seq“ Modell gehört. Ein „Seq2Seq“ Modell ist darauf spezialisiert, aus einer bestimmten Eingabesequenz (zum Beispiel ein Text) eine Ausgabesequenz zu erzeugen. Transformer nutzen dabei spezielle Techniken, die es ihnen ermöglichen einen Text im Ganzen zu betrachten, anstatt diesen Wort-für-Wort durchzugehen. Diesen Mechanismus, welcher das „Betrachten im Ganzen“ ermöglicht, ohne dabei den Kontext zu verlieren, wird „Attention“ genannt. Dadurch bieten Transformer-Modelle ein besonders gutes Lang- und Kurzzeitgedächtnis und können die Zusammenhänge zwischen den Wörtern eines Textes besser verstehen. Damit sind Transformer besonders gut in Aufgaben wie Übersetzung oder Text-Verstehen und bilden die grundlegende Struktur der meisten State-of-the-Art LLMs.

Recurrent neural network – RNN: Ein RNN ist eine grundlegende und im Vergleich zum Transformer etwas einfachere und ältere Form der Machine-Learning-Modelle. Auch dies wurde bereits in der vorgehenden Bachelorarbeit dargestellt. Der grundlegende Unterschied im Vergleich zum Transformer ist der fehlende „Attention“ Mechanismus. Dadurch ist die Art und Weise, in der ein RNN einen Text verarbeitet, grundlegend anders. Anstatt einen Text im Ganzen zu betrachten, geht ein RNN Wort-für-Wort vor. Um dabei nicht den Kontext zu verlieren, nutzen RNN-Modelle auch eine Art „Gedächtnis“ System, um die verschiedenen Wörter in Bezug zueinander zu setzen und den Kontext zu verstehen. Die RNN-Struktur ist bei LLMs eher selten.

Hyper-Parameter und Modell-Parameter: Diese beiden Grundbegriffe des Machine-Learning sind wichtig. Vor allem, da sie sehr ähnlich klingen, aber unterschiedliche Dinge beschreiben. Die Hyper-Parameter sind die Einstellungen, die ein Modell, das Training oder die Anwendung eines Modells definieren. So werden die meisten Machine-Learning Modell zu den Beispielen über mehrere „Epochen“ hinweg trainiert. Die Anzahl dieser Epochen ist ein Hyper-Parameter. Auch die „Temperatur“, die bei der Anwendung eines LLMs angegeben werden kann, um zu bestimmen, wie deterministisch das Modell sich verhalten soll, ist ein Hyper-Parameter. Diese Hyper-Parameter werden direkt vom Anwender und/oder Entwickler eines Modells gesetzt. Im Gegenzug dazu sind Modell-Parameter die Werte, die während des Trainings automatisch vom Modell gesetzt werden und so am Ende das trainierte Modell definieren. Oft werden die Modell-Parameter auch synonym mit den Modell-Gewichten verwendet, das ist allerdings nicht korrekt. Die Modell-Gewichte gehören zwar zu den Modell-Parametern und werden während des Trainings gesucht. Allerdings stellen sie nur einen Teil der Modell-Parameter dar. Alle Gewichte sind Parameter, aber nicht alle Parameter sind Gewichte.

2. Offene Modelle

Training und Fine-Tuning: Um einem Modell etwas beizubringen, ist es nötig, dieses zu trainieren. Dazu werden meist gewaltige Datensätze und außerordentlich leistungsfähige Hardware benötigt. Gerade im Fall von LLMs werden sehr viel Informationen für das Training benötigt, die nur für das allgemeine Textverständnis und nicht für den speziellen Anwendungsfall benötigt werden. Das macht es unpraktisch, für jeden Spezialfall ein eigenes Modell von Grund auf zu trainieren. Für diesen Fall wurde das Konzept des Fine-Tunings entworfen. Dabei wird ein vortrainiertes Modell genommen und mithilfe eines kleineren spezielleren Datensatzes weitertrainiert. So kann das Modell sowie von dem ursprünglichen aufwendigen Training Gebrauch machen, wird aber auch auf einen Spezialfall angepasst. Dies wird als Fine-Tuning bezeichnet und kann bedeutend den Aufwand und die Kosten für das Training eines LLMs reduzieren.

Prompt und Prompt Engineering: Der Prompt ist der Eingabetext. Also genau der Text, aus dem das LLM eine Antwort generieren soll. Es ist die erste Sequenz aus dem „Sequenz-to-Sequenz“ Begriff. Häufig wird auch der Begriff „**Kontext**“ als Synonym für „Prompt“ verwendet. Mit der Entwicklung komplexerer und leistungstärker LLMs hat sich daraus der Bereich des Prompt Engineering entwickelt. Dieser stellt im Grunde eine noch stärkere Vereinfachung des zuvor beschriebenen Fine-tunings dar. Anstatt ein Modell auf den eigenen Anwendungsfall mit Fine-Tuning anzupassen, wird das Modell gar nicht mehr weiter trainiert, sondern ein vortrainiertes Modell verwendet. Dieses Modell wird dann ausschließlich durch das Entwerfen eines passenden Prompts, welcher die Aufgabe ins Detail und für das LLM verständlich beschreibt, auf die eigene Aufgabe angepasst. Dadurch sinken der Aufwand und die Kosten für das Nutzen von LLMs weiter. Mit diesem Ansatz muss kein teures Training mehr durchgeführt werden und auch kein Trainingsdatensatz erstellt werden. Da sich die Modelle in diesem Fall vollkommen auf den Prompt verlassen, wird dieser sehr bedeutungsvoll und auch kleine Änderungen am Prompt können oft große Auswirkungen auf die Ergebnisse haben. Dadurch kann das Entwerfen eines guten Prompts sehr aufwendig werden. Das Entwerfen und Anpassen dieser Prompts wird Prompt Engineering genannt. Häufig werden auch beide Konzepte (Prompt Engineering und Fine-Tuning) miteinander verbunden. Zu diesem Zweck werden Modelle mittels Fine-Tuning speziell so angepasst, dass sie besonders gut auf die mittels Prompt Engineering entworfenen Instruktionen reagieren. Dieses Konzept wird „**Instruction Fine-Tuning**“ genannt.

Few-Shot- und Zero-Shot-Prompting: Für das Entwerfen der Prompts gibt es eine Vielzahl an Möglichkeiten und Strukturen, die eingesetzt werden können, um die Ergebnisse auf den eigenen Fall anzupassen. Dabei gibt es zwei Grundstrukturen, die für die Entwicklung eines Prompts wichtig sind: „Few-Shot“- und „Zero-Shot“-Prompts. Diese Bezeichnungen beziehen sich dabei auf die gegebenen (oder nicht gegebenen) Beispiele innerhalb eines Prompts. Am einfachsten ist das anhand eines Beispiels zu sehen:

```
Translate the following English sentences to French:  
  
"I love you." - "Je t'aime."  
"Thank you very much." - "Merci beaucoup."  
"How are you?" - "Comment ça va?"  
  
"The weather is nice today." -
```

Abbildung 1: Beispiel für einen einfachen Few-Shot Prompt

2. Offene Modelle

Wie in der Abbildung zu sehen ist, wird zuerst die Aufgabe beschrieben und dann 3 Beispiele gegeben. Schließlich folgt die Nutzereingabe, die das LLM zu beantworten hat. In diesem Stil werden bei einem Few-Shot-Prompt Beispiele verwendet, um dem Modell klarzumachen, welche Antwort erwartet wird. Dabei wird das Modell oft wie eine Art Auto-Vervollständigung verwendet. Der Haupt-Nachteil an diesem Ansatz ist, dass die Antworten dadurch etwas zu starr werden können. Ein Zero-Shot-Prompt ist genau das Gegenteil und es werden keine Beispiele verwendet:

```
Translate the following English sentences to French:  
  
"The weather is nice today.":
```

Abbildung 2: Beispiel für einen einfachen Zero-Shot-Prompt

Das macht es schwieriger, für das Modell genau zu verstehen, wie die Ausgabe aussehen soll, ermöglicht aber auch mehr Flexibilität.

Inference: Inference ist der Prozess, einem trainierten Machine-Learning-Modell neue Daten zur Verarbeitung zu geben. Im Fall eines LLM wäre dies das Nutzen des LLMs, indem diesem ein Prompt zur Bearbeitung gegeben wird und dieses dann auf den Prompt antwortet. Inference ist in diesem Fall also die Nutzung eines LLM.

2.3. Vergleich verschiedener Open-Source-Modelle

Nachdem in den vorherigen Kapiteln die grundlegenden Unterschiede zwischen offenen und geschlossenen LLMs geklärt wurden, gilt es nun, die verschiedenen offenen Modelle miteinander zu vergleichen. Dies hat hauptsächlich zwei Ziele: Erstens, einen Überblick über die recht verwirrende Welt der Open-Source LLMs zu geben. Und zweitens, eine Auswahl zu treffen, welches Modell in Kapitel „4. Praxisteil“ tatsächlich verwendet werden soll. Für die Softwareauswahl gibt es in der Wirtschaftsinformatik eine Reihe an verschiedenen Herangehensweisen und Modellen. So nennt die „Enzyklopädie der Wirtschaftsinformatik“ [18] unter anderem die Schritte „Ziele“, „Prozessanalyse“, „Anforderungen“, „Marktübersicht“, „Screening“, „Endauswahl“ und „Entscheidung“ in ihrem Modell, während andere Werke andere Aspekte nennen. Eine einzige Vorgehensweise, welche in allen Situationen uneingeschränkt funktioniert, gibt es nicht wirklich. Deshalb soll im Folgenden eine für diesen Fall angepasste Vorgehensweise angewendet werden. Dieser Vorgang kann grob in folgenden Schritten beschrieben werden:

1. Übersicht über verfügbare Modelle (Kapitel „2.3.1“)
2. Analyse und Bewertung der Modelle anhand ausgewählter Aspekte (Kapitel „2.3.2“, „2.3.3“, „2.3.4“, „2.3.5“ und „2.3.6“)
3. Zusammenfassung aller Bewertungen (Kapitel „2.3.7“)
4. Entscheidung auf Basis der Anforderungen für die zu entwickelnde Applikation (Kapitel „2.3.7“)

Die verschiedenen Aspekte der Modelle sollen, einzeln untersucht, einander gegenübergestellt und bewertet werden. Schließlich soll jedem Modell zu jedem Aspekt eine Punktzahl gegeben werden, um so den Vergleich zwischen den Modellen zu vereinfachen. Es ist fast unmöglich, eine solche Untersuchung vollkommen objektiv zu gestalten, deshalb soll die Punktskala möglichst einfach und nachvollziehbar gehalten werden. Die verschiedenen Modelle werden für jeden betrachteten Aspekt eine Punktzahl zwischen 1 und 4 zugeordnet.

2. Offene Modelle

Dabei stellt 4 die beste Punktzahl dar und 1 die schlechteste. Ziel ist es, am Ende des Kapitels eine übersichtliche Tabelle zu erzeugen, in welche die verschiedenen Aspekte aufgelistet werden und so am Ende alle Modelle anhand der verteilten Punktzahl verglichen werden können. Bei jedem Modell sollen speziell folgende Aspekte untersucht werden:

- **Leistung:** Mithilfe verschiedener Tests soll die Ausgabequalität der Modelle dargestellt werden. Die Qualität der Ausgabe eines LLM ist der wichtigste Aspekt, wenn es um LLMs geht. Grundlegend ist es nicht relevant wie offen ein LLM ist, wenn die Leistung nicht ausreichend ist, um die gewünschten Aufgaben zu erfüllen. Aus diesem Grund und auch weil es der am leichtesten objektiv zu bewertenden Aspekt ist, wird für Leistung die dreifache Punktzahl, also maximal 12 Punkte, vergeben.
- **Offenheit:** Der Fokus der Arbeit liegt speziell auf Open-Source Modellen. Wie in Kapitel „2.1. Warum Open-Source-Modelle?“ beschrieben, ist fraglich, wie gut das Klassische Open-Source Verständnis auf Machine-Learning Modelle anwendbar ist. Deshalb soll erkennbar dargestellt werden, wie Open-Source die Modelle wirklich sind und welche Aspekte gegebenenfalls geheim gehalten werden.
- **Lizenz:** Dieser Aspekt ist eng verbunden mit dem Punkt „Offenheit“, wird aber aufgrund seiner Komplexität einzeln behandelt. Hier sollen die verschiedenen Lizenzen der Modelle untersucht und gezeigt werden, was unter den Lizenzen erlaubt ist und was nicht.
- **Entwickler:** Der Fall OpenAI zeigt, wie schnell es gehen kann, dass ein Unternehmen, welches Open-Source Modelle veröffentlicht, zur Entwicklung geschlossener Modelle wechselt. Deshalb werden auch die Entwickler der Modelle betrachtet. Wie engagiert sind diese in der Open-Source Community und gibt es alternative Ziele, die vom Entwicklerteam verfolgt werden?
- **Community:** Speziell bei Open-Source Modellen bietet die Community, die sich um die Modelle herum entwickelt, einen großen Vorteil. So kann es zum Beispiel Modelle geben, die von Communitymitgliedern durch Finetuning für bestimmte Aufgaben angepasst und verbessert wurden. Außerdem können sich Projekte um ein Modell herum entwickeln, oder andere offene Projekte bieten die Möglichkeit an, bestimmte offene Modelle zu verwenden. Zusätzlich bietet eine große Community den Vorteil des Communitysupports, je mehr Leute mit einem Modell arbeiten, umso höher die Chance, dass andere schon ähnliche Probleme hatten, wie jene, die im eigenen Projekt auftreten.

Nachdem dargestellt wurde, welche Aspekte der Modelle betrachtet werden sollen, bleibt noch die Frage, welche Modelle es im Folgenden zu untersuchen gilt. Es gibt verschiedene Quellen, in denen offene LLMs aufgelistet werden. So zum Beispiel das „Open LLM Leaderboard“ von HuggingFace, welches die Modelle nicht nur auflistet, sondern auch noch zusätzlich eine Einschätzung ihrer Leistung bietet [6]. Oder das GitHub Repository „Awesome-LLM“ [19], das eine umfangreiche Liste speziell auch von offenen LLMs bietet. Auch andere Repositories wie „Open-llm“ [20] bieten entsprechende Listen. Je nachdem wie gezählt wird und welche Quellen zugrunde gelegt werden, gibt es zwischen 20 und 58 eigenständige offene Modelle. Selbst mit „Open-llm“ als Grundlage, welches nur 20 Modelle listet, ist eine Untersuchung aller Modelle kaum realistisch (geschweige denn 58 Modelle). Um den Umfang dieser Arbeit einzuhalten, ist es nötig, nur eine Auswahl an Modellen zu untersuchen. Zu diesem Zweck wurden speziell die Modelle zum Vergleich ausgesucht, die ein Alleinstellungsmerkmal irgendeiner Art haben. Damit stellen die im Vergleich

2. Offene Modelle

dargestellten Modelle nur eine Auswahl an beliebten LLMs dar, keinesfalls aber eine allumfassende Übersicht aller offenen LLMs. Die folgenden 10 Modelle bzw. Modellfamilien wurden für die genauere Untersuchung gewählt:

| Modellname/Modellfamilie | Entwickler | Veröffentlichungsdatum |
|---|---------------------------------|------------------------|
| Cerebras-GPT | Cerebras | März 23 |
| Bloom | BigScience Workshop | November 22 |
| Pythia | EleutherAI | April 23 |
| RWKV | RWKV | August 22 |
| StableLM | Stability AI | April 23 |
| UL2 (Unifying Language Learning Paradigms) | Google | Oktober 22 |
| MPT (Mosaic Pretrained Transformer) | Mosaic ML | Mai 23 |
| LLaMA (Large language model by Meta AI) | Meta | Juli 23 |
| OpenLLaMA | OpenLM Research | Mai 23 |
| Falcon | Technology Innovation Institute | Mai 23 |

Tabelle 1: Liste der zu vergleichenden Modelle

2. Offene Modelle

2.3.1. Alleinstellungsmerkmale der Modelle

Das Hauptauswahlkriterium für die Modelle ist, dass sie jeweils mindestens ein spezifisches Alleinstellungsmerkmal haben, welches die Untersuchung des jeweiligen Modells interessant macht. In diesem Kapitel sollen die Modelle kurz vorgestellt und ihre jeweiligen Alleinstellungsmerkmale dargestellt werden.

Cerebras-GPT: Dieses Modell wurde entwickelt von „Cerebras Systems Inc.“ in den USA. Das Unternehmen entwickelt in erster Linie nicht LLMs, sondern stellt eigenentwickelte KI-Beschleuniger her. Normalerweise werden LLMs mithilfe von Grafikkarten (kurz GPU) trainiert. Dabei werden meist sehr viele Grafikkarten gleichzeitig benötigt. Meta's LLaMA 2 benötigte beispielsweise insgesamt 3,3 Mio. GPU-Stunden auf sehr leistungsfähigen „NVIDIA A100 80GB“ GPUs [21]. Aufgrund der enormen Anzahl an GPUs, die für Machine-Learning benötigt werden, gibt es Unternehmen wie Cerebras, die alternative KI-Beschleuniger herstellen. Ihren „Wafer-Scale Engine“ (kurz WSE) Chip gibt es mittlerweile in der zweiten Version und wird als Bestandteil des „CS-2“ („Cerebras Systems 2“) verkauft. Cerebras wirbt auf ihrer Webseite damit, dass ihr „WSE-2“ deutlich schneller ist als die zuvor genannte A100 GPU und zwischen 20 und 100 GPUs gegen ein einziges „CS-2“ ersetzt werden kann. Wie viel davon reines Marketing ist und was tatsächlich Realität ist, ist schwer einzuschätzen. Tatsache ist, dass Cerebras-GPT eines von nur zwei Modellen in der Auswahl ist, das nicht auf klassischen GPUs trainiert wurde und das Einzige, für welches selbst entwickelte Hardware verwendet wurde. Konkret wurden die Modelle in diesem Fall auf 16 „CS-2“ trainiert. Der Aufbau des Modells basiert grob auf OpenAIs GPT-3. Damit ist das Modell einzigartig in der Auswahl, nicht etwa wegen seiner Struktur, sondern vielmehr wegen des ungewöhnlichen Trainingsprozesses.

Bloom: Bloom steht für „BigScience Large Open-science Open-access Multilingual Language Model“ und stellt eines der größten Modelle in der Auswahl dar. In seiner größten Version hat das Modell 176 Milliarden Parameter (176B), einzig die „Flan“ Modellfamilie bietet ähnlich große Modelle. Das allein macht die Modelle schon besonders. Neben der reinen Größe des Modells ist die Besonderheit von Bloom primär die sehr große Anzahl an Menschen, die zur Entwicklung beigetragen haben. Das Modell entstand im Zuge des von Huggingface organisierten „BigScience Workshop“ [22]. An dem Workshop nahmen über 1000 Menschen teil. Damit verfolgte auch die ursprüngliche Entwicklung des Modells bereits einen offenen Ansatz.

Pythia: Entwickelt vom Non-Profit „EleutherAI“ sticht durch seine Offenheit hervor. Nicht nur ist das Modell komplett Open-Source, „EleutherAI“ entwickelt zusätzlich auch noch die Open-Source LLM-Trainingsbibliothek „GPT-NeoX“, welche das Training eigener LLMs von Grund auf ermöglicht bzw. stark vereinfacht. Auf Basis von „GPT-NeoX“ stellt „EleutherAI“ auch verschiedene LLMs zur Verfügung. Darunter das beliebte „GPT-NeoX-20B“ und auch „Pythia“. Beide Modelle sind grundlegend interessant für diese Arbeit. Es wurde allerdings entschieden, dass nur ein Modell pro Entwickler untersucht werden soll, um einen weiter gefächerten Überblick über die Szene geben zu können. „Pythia“ ist das neueste LLM aus dem Hause „EleutherAI“.

2. Offene Modelle

RWKV: Dieses Modell unterscheidet sich massiv von allen anderen ausgewählten Modellen. Während die meisten LLMs auf der Transformer-Struktur aufgebaut sind (so auch alle anderen LLMs aus der Auswahl für diese Arbeit), geht RWKV einen anderen Weg. Als einzige Modellfamilie der Auswahl, bauen die RWKV-Modelle auf einer RNN-Struktur (siehe „2.2. Grundlegende Fachbegriffe“) auf, das ist im LLM-Bereich außergewöhnlich. Laut eigener Aussage der Entwickler bietet RWKV, Leistung auf vergleichbarem Niveau wie andere LLMs, aber mit schnellerem Inference und Training, weniger Speicherbedarf und keine Beschränkung der Kontextlänge [23].

StableLM: Wird entwickelt von „Stability AI“, einer Firma, die auch schon bei der Entwicklung des sehr bekannten und kontroversen Bild-Erzeugungs-Modell „Stable Diffusion“ [24] mithalf. Anfang des Jahres 2023 stellten sie dann ein eigenes LLM vor. Das Modell umfasst selbst in seiner größten Version nur 7 Milliarden Parameter (7B) und ist damit relativ klein. Dabei schafft das Modell es auf dem HuggingFace Leaderboard eine, für Modelle dieser Größe, gute Punktzahl zu erhalten und ist etwa gleichauf mit der 7B Version von LLaMA (die auch bereits überdurchschnittlich gut ist). Dabei nutzt „Stability AI“ für StableLM, EleutherAI's GPT-NeoX Bibliothek. Vergleichbare Modelle schneiden auf dem Leadboard schlechter ab.

UL2: Entwickelt von Google ist UL2 das erste Modell auf der Liste, welches von einer der „Big-Tech“ Firmen entwickelt wurde. Das Modell nutzt wie die meisten eine Transformer-Struktur, modifiziert diese, aber indem eine zusätzliche Schicht nach dem Transformer eingefügt wird. Google nennt diese Schicht „Mixture-of-Denoisers“ [25]. Ein „Denoiser“ (auf Deutsch etwa „Rauschentferner“) ist ein Algorithmus oder Tool, welches Datenrauschen herausfiltern soll. In LLMs werden diese normalerweise nicht eingesetzt. Die konkrete Struktur der „Mixture-of-Denoisers“ Schicht ist komplex und wird in der Arbeit „Unifying Language Learning Paradigms“ [25] beschrieben, grundlegend werden verschiedene „Denoiser“ gemeinsam eingesetzt, um das Ergebnis zu verbessern.

MPT: Der „Mosaic Pretrained Transformer“ zeichnet sich vorwiegend durch einen Punkt aus: die maximale Kontextlänge. Während die meisten LLMs eine maximalen Kontextlänge von ~4.000 oder sogar nur ~2.000 Tokens bieten, ermöglicht MPT eine doppelt (bzw. vierfach) so große Kontextlänge von ~8.000 Token. Damit ist es insgesamt möglich, längere Prompts zu verwenden, längere Antworten auszugeben oder beides. Für den speziellen Fall, der im Zuge dieser Arbeit zu entwickelnden Applikation, ist diese längere Kontextlänge zwar nicht zwingend notwendig, bei entsprechend guter Leistung würde es dennoch Vorteile bieten, so etwa das Verwenden einer größeren Chat-History.

LLaMA 2: Das LLaMA Modell war bereits in seiner ersten Version sehr beliebt. Diese Beliebtheit in der Community stieg mit der Veröffentlichung von LLaMAs zweiter Version noch weiter an. Die LLaMA 2 Familie stellt momentan die mit Abstand beliebtesten offenen Modelle dar. Allein LLaMA-2-7b-hf (eine von Meta für HuggingFace optimierte Version der von LLaMA-2-7B) wurde im letzten Monat (Stand 20.09.2023) laut HuggingFace über 580.000-mal heruntergeladen. Die 7B Version, der ebenfalls beliebten „Falcon“ Modellreihe kommt nur auf knapp über 306.000 Downloads. Und dies, obwohl LLaMA 2 nicht ohne weiteres heruntergeladen werden kann, sondern erst Zugriff auf das Modell anfordern muss, was mehrere Tage dauern kann. Ohne diese Beschränkung wäre der Unterschied deutlich signifikanter. Diese Beliebtheit ist auch direkt eine der größten Stärken des Modells. Es gibt eine sehr große Anzahl an Communitymitgliedern, die auf Basis von LLaMA 2 durch Fine-

2. Offene Modelle

tuning eigene Modelle erstellt haben und zur Verfügung stellen. Diese Versionen von LLaMA 2 führen das Open LLM Leaderboard mehr oder weniger allein an. Erst auf Platz 71 ist das erste Mal ein Modell zu sehen, welches nicht zur LLaMA Reihe gehört [6]. (Was nicht nur an der hervorragenden Leistung der LLaMA Modelle liegt, sondern vor allem auch an der gewaltigen Anzahl an auf LLaMA basierenden Modellen.)

OpenLLaMA: Die konkreten Lizenzen der verschiedenen Modelle werden später genauer untersucht, soviel aber voraus: Die LLaMA-Lizenz ist recht streng und erlaubt einiges nicht. Aus diesem Grund, in Kombination mit der starken Beliebtheit von LLaMA, machte es sich das „OpenLM Research“ Projekt zur Aufgabe, eine offenere Reproduktion von LLaMA zu entwickeln. Ergebnis dieses Projekts ist die OpenLLaMA Modellreihe. Das Besondere an diesem Modell ist damit, dass es vermeintlich ähnliche Leistung wie LLaMA bieten soll, dabei aber unter einer offeneren Lizenz steht.

Falcon: Zuletzt wird die Falcon Modellreihe untersucht. Diese ebenfalls sehr beliebten Modelle zeichnen sich vorrangig durch zwei Dinge aus. Das ist einerseits der Datensatz, während die meisten Modelle hauptsächlich mit Daten in einer oder zumindest nur wenigen Sprachen trainiert wurden, nutzt Falcon einen Datensatz, der auch viele Texte in europäischen Sprachen beinhaltet. In erster Linie sind das Deutsch, Spanisch und Französisch. Damit können die Modelle gewisse Vorteile bieten, wenn in anderen Sprachen als Englisch gearbeitet werden soll. Es sei jedoch auch gesagt, dass Falcon nicht das einzige LLM ist, welches mehrere Sprachen unterstützt, so wurde zum Beispiel auch Bloom mit einer breiten Anzahl an Sprachen trainiert. Neben den verschiedenen Sprachen stellt auch noch die Größe der Modelle ein wichtiges Merkmal der Modellreihe dar. In seiner vor Kurzem (06.09.2023) erschienenen neusten Version hat das Modell 180 Milliarden Parameter (180B) und somit noch mehr als Bloom. Damit ist Falcon-180B momentan das größte offene Modell. Auch die Leistung ist laut dem HuggingFace Open LLM Leaderboard gut. Zuvor wurde gesagt, dass erst auf Platz 71 ein Modell zu finden ist, welches nicht zur LLaMA Familie gehört. Dieses Modell ist Falcon-180B. Damit liegt die Falcon Reihe, wenn nur die Modellfamilien betrachtet werden und nicht die einzelnen Modelle, auf Platz zwei auf dem Leaderboard.

2.3.2. Leistung

Die Leistung ist der wichtigste Einzelaspekt. Ohne für das eigene Vorhaben ausreichende Leistung kann ein Modell nicht verwendet werden. Wenn ein Modell in den kommenden Aspekten „Offenheit“, „Lizenz“, „Entwickler“ und „Community“ jeweils die volle Punktzahl erhält, ein anderes Modell mit weniger Punkten in diesen Bereichen aber bessere Leistung bietet, kann es Sinn ergeben, das Modell mit der besseren Leistung zu wählen. Wie zuvor beschrieben, wird unter anderem aus genau diesem Grund der Punkt „Leistung“ stärker gewichtet. Dazu werden jeweils bis zu 12 Punkte vergeben (also dreimal mehr als normalerweise). Ein weiterer hilfreicher Nebeneffekt der erhöhten Punktzahl ist, dass die zugeordnete Punktzahl genauer ist, insbesondere da die Performance der zehn Modellfamilien stark variiert. Besonders wichtig ist es zu beachten, dass die Leistung der Modelle sehr anwendungsspezifisch ist. Das macht es schwierig, die Leistung von Modellen im Vorhinein einzuschätzen. Prinzipiell gibt es eine riesige Menge an LLM-Benchmarks. Aus diesem Grund hat „EleutherAI“ ein Open-Source-Projekt mit dem Namen „LM-Evaluation-Harness“ entwickelt, welches eine Vielzahl an Tests verbindet und laut eigener über 200 verschiedene Testaufgaben unterstützt [26]. Diese Tests sind in der Regel so gestaltet, dass sie nach Möglichkeit alle Aspekte eines LLMs bewerten können. Daraus ergibt sich wiederum

2. Offene Modelle

ein Problem, soll das LLM zum Beispiel als Chatbot ähnlich wie ChatGPT eingesetzt werden, dann können die Ergebnisse dieser Tests aussagekräftig sein. Soll das LLM aber als Teil einer Applikation spezifische Aufgaben übernehmen, können diese Tests bestenfalls einen groben Eindruck geben, ob ein Modell möglicherweise besser ist als ein anderes. Und selbst das ist nicht immer gegeben. LLMs sind so aufgabenspezifisch, dass selbst eine klein wirkende Veränderung des Prompts bereits große Auswirkungen haben kann.

Wie kann nun, unter Anbetracht dieses Problems mit LLM-Benchmarks, die Leistung der Modelle verglichen werden? Wenn vorgefertigte Benchmarks nicht geeignet sind, bleibt noch das Entwerfen eines eigenen Benchmarks. Das hat vor allem den Vorteil, dass die Modelle spezielle mit dem eigenen Anwendungsfall und den eigenen Prompts getestet werden können und so die Ergebnisse tatsächliche Aussagekraft für die eigene Anwendung haben. Gleichzeitig hat dieser Ansatz aber auch einen recht offensichtlichen Nachteil den fertigen Benchmarks gegenüber: der Arbeitsaufwand. Der Aufwand einen fertigen Benchmark zu verwenden ist sehr gering, wenn etwa das „LM-Evaluation-Harness“ von „EleutherAI“ verwendet wird. Oder noch einfacher, es werden die meist bereits online bereitstehenden Testergebnisse verwendet. Im Gegensatz dazu müssen bei einem eigenen Benchmark alle Aspekte eigenständig entworfen werden. Soll ein Vergleich, wie der in dieser Arbeit, durchgeführt werden, gibt es allerdings kaum Alternativen. Deshalb wird für den Leistungsvergleich ein eigens für diese Arbeit entworfener Benchmark verwendet. Dieser Benchmark bildet spezifisch Aufgaben ab, die später für die Anwendung in der Applikation relevant sind. Deshalb orientiert sich das Prompt-Format stark an den Prompts, die in der Applikation genutzt werden sollen. So wird getestet, welches Modell für den ganz speziellen Anwendungsfall der zu entwickelnden Applikation die beste Leistung zeigt.

Um die Testergebnisse besser einordnen zu können, ist es wichtig, den Aufbau des Tests zu verstehen. Da das LLM in der Applikation Informationen an den Programmcode weitergeben muss, ist das Ziel der Prompts jeweils das Erzeugen eines JSON-Strings, welcher dann vom Code verarbeitet werden kann. Das macht auch das Testen für den Benchmark einfacher. Die Ausgabe des LLMs wird als JSON-Objekt geladen und mit einem vordefinierten Zielobjekt verglichen. Ist das vom LLM erzeugte Objekt identisch mit dem erwarteten Objekt, war die Ausgabe des LLMs korrekt. Unterscheiden sich die Objekte, oder das Laden des JSON-Strings in ein Objekt schlägt fehl, so war die Ausgabe des LLMs falsch. Dabei wird den Modellen die Aufgabe vereinfacht, indem der LLM-Output jeweils nach einem JSON-String durchsucht wird und gegebenenfalls sämtlicher Text vor oder nach dem JSON-String verworfen wird. Der konkrete Ablauf der Tests kann dem Eval-Code im GitHub Repository dieser Arbeit entnommen werden (Datei: eval.ipynb).

Der Aufbau der Prompts sieht dabei grundlegend wie folgt aus:

```
### Instruction:  
[Beschreibung]  
[Beispiele]  
[Nutzereingabe]  
### Response:
```

Abbildung 3: Grobes Format der Eval-Prompts

Dabei wird „[Beschreibung]“ jeweils durch die Beschreibung der Aufgabe ersetzt und „[Beispiele]“ durch einige Beispiele, welche die dargestellte Aufgabe spezifizieren. Schließlich

2. Offene Modelle

folgt die simulierte Nutzereingabe, auf die das LLM reagieren soll. Im Standardfall wird alles von „### Instruction:\n“ und „### Response:\n“ umschlossen. Es ist auch möglich, dass der Prompt für ein Modell von diesem Standard abweicht. So wird bei einigen LLMs ein bestimmtes Prompt-Format vorausgesetzt, welches für die beste Leistung verwendet werden soll, in dem Fall wird das beschriebene Format entsprechend angepasst. Eine detailliertere Beschreibung der Prompts findet später in Kapitel „4. Praxisteil“ statt, diese ist für das Verständnis der Tests vorerst weniger relevant.

Der Benchmark ist unterteilt in sieben Testkategorien: „Kurz“, „Kurz ohne Beispiele“, „Medium“, „Medium ohne Beispiele“, „Medium mit Fehlern“, „Lang“ und „Lang ohne Beispiele“. „Kurz“, „Medium“ und „Lang“ spiegelt dabei jeweils die Länge der Beschreibung wider. Bei den mit „Kurz“ markierten Tests ist die Beschreibung so kurz und knapp wie möglich gehalten. Die Aufgabe ist recht einfach und ohne viele Zusatzinformationen gestaltet. Bei den „Medium“ Tests gibt es mehr Kontext, die Aufgabe ist etwas umfangreicher beschrieben. Damit einher geht auch eine etwas komplexere Aufgabe, bei der mehr Dinge beachtet werden müssen. Zuletzt bei den „Lang“ Tests ist die Beschreibung am längsten und ausführlichsten, das bedeutet aber auch, dass am meisten Details vom LLM beachtet werden müssen.

Diese Kategorien sollten dabei auf keinen Fall mit „einfach“, „mittel“, „schwer“ verwechselt werden. Zwar werden die Aufgaben mit längerem Kontext immer komplexer, aber gleichzeitig werden die Aufgaben auch besser bzw. genauer erklärt. Es ist nicht unbedingt zu erwarten, dass bei den „Kurz“ Tests die besten Ergebnisse erzielt werden, da die Beschreibung für viele LLMs zu ungenau sein könnte. Andersherum ist es genauso möglich, dass ein „Lang“ Test zu viel Kontext für ein gegebenes LLM bietet, sodass dieses die Details der Aufgabe nicht richtig versteht. Allgemein ist am ehesten zu erwarten, dass die Modelle bei den „Medium“ Tests am besten abschneiden. Neben der Beschreibung der Kontextlänge gibt es noch die Zusätze „ohne Beispiele“ und „mit Fehlern“. Bei den Tests „Ohne Beispiele“ werden dem LLM keine Beispiele vorgelegt, dieser Teil aus dem in Abbildung 3 beschriebenen Prompt-Format fällt dann weg. Dadurch werden die Prompts zu Zero-Shot-Prompts. Welchen Unterschied das für eine tatsächliche Anwendung macht und warum es Sinn ergeben kann keine Beispiele zu verwenden, wird in Kapitel „4. Praxisteil“ beschrieben. In diesem Kapitel werden nur die Performanceunterschiede dargestellt.

In dem Test „mit Fehlern“ wurden absichtlich Rechtschreibfehler in bestimmten Teilen des Prompts eingebaut, welche vom LLM dann auch als solche übernommen werden sollen. Grob kann ein solcher Fall wie folgt aussehen: „Schreibe einen Text, der das Wort „Rechtschreibfehler“ beinhaltet.“ In diesem Prompt ist mit Absicht ein Rechtschreibfehler im Wort „Rechtschreibfehler“. Die Erwartung an das Modell wäre es, diesen Fehler zu übernehmen, ohne ihn zu korrigieren. Dies erscheint im ersten Moment merkwürdig. Der Nutzen ergibt sich, wenn bedacht wird, dass die echten Prompts später ein anderes Format haben und JSON-Objekte erzeugen sollen, welche Key/Value Paare enthalten. Es ist wichtig, dass diese Key/Value Paare exakt so vom LLM wiedergegeben werden, wie sie im Prompt beschrieben wurden, auch wenn sich darin Rechtschreibfehler befinden. Ist das nicht der Fall, kann der Programmcode die LLM-Ausgaben nicht richtig verarbeiten. In diesem Test werden die Beispiele beibehalten. Auch hierfür wird es im Kapitel „4. Praxisteil“ wieder konkrete Beispiele geben.

2. Offene Modelle

Folgend ist beispielhaft ein Ausschnitt aus dem „Kurz ohne Beispiele“ Test zu sehen. Zur besseren Verständlichkeit sind alle Schlüssel orange und alle Werte blau markiert. Zudem handelt es sich um ein stark verkürztes Beispiel, welches den Aufbau der Tests veranschaulicht. An allen Stellen, an denen „...“ zu sehen ist, wurde das Beispiel verkürzt. Alle vollständigen Tests sind im GitHub Repository dieser Arbeit unter „llm_evaluation/prompts/“ zu finden.

```
[
  {
    "prompt": "Below is a \"USER REQUESTS\" and \"RESPONSE\" from a Smart Home System.  
Respond in JSON format as defined under \"JSON STRUCTURE\". The  
\"action\" should be \"turn on\" or \"turn off\".  
  
JSON STRUCTURE: {\"device\": string, \"action\": string}  
  
{user_request_token}: \"{input}\"",
    "in_out_pairs": [
      {
        "input": "Shut down the Balcony Light.",
        "expected_answer": "{\"device\": \"Balcony Light\", \"action\": \"turn off\"}"
      },
      ...
    ]
  },
  {
    "prompt": "...",
    "in_out_pairs": [
      ...
    ]
  }
]
```

Abbildung 4: Ausschnitt aus dem "Kurz ohne Beispiele" Test

Um die Modelle zu verwenden und Inference durchführen zu können, wird das Open-Source-Projekt „Text Generation Inference Server“ (kurz TGI) von HuggingFace [27] verwendet.

Dieses bietet dank eines bereitgestellten Docker Containers eine einfache Möglichkeit eine Vielzahl an Modellen zu laden und eine API bereitzustellen, über welche die Modelle dann genutzt werden können. Dabei bietet TGI in erster Linie den Vorteil, dass er von HuggingFace selbst in ihrer Produktivumgebung eingesetzt wird und damit eine gewisse Zuverlässigkeit suggeriert. Außerdem wird die TGI API nativ von LangChain unterstützt. Eine genauere Darstellung des TGI sowie auch potenzieller Alternativen folgt später in Kapitel „4. Praxisteil“.

Bevor es in die einzelnen Leistungstests geht, sei abermals darauf hingewiesen, dass die im Folgenden präsentierten Daten keinesfalls eine allgemeine Schlussfolgerung „Modell X ist besser als Modell Y“ zulassen, sondern lediglich für den ganz konkreten Anwendungsfall und Prompts gelten. Es ist möglich, dass die Bewertung für einen anderen Anwendungsfall bzw. andere Prompts anders aussieht.

Die Tests wurden jeweils entweder auf lokaler Hardware ausgeführt (nur bei kleinen Modellen mit maximal 7 Milliarden Parametern) oder, in den meisten Fällen, auf beim Cloudanbieter „Runpod.io“ [28] gemieteter Hardware durchgeführt.

2. Offene Modelle

LLaMA 2: Zuerst soll ein Modell aus der LLaMA 2 Reihe getestet werden. Die LLaMA 2 Modelle führen das HuggingFace LLM Leaderboard an. Dementsprechend interessant ist es zu sehen, wie sich das auf den hier getesteten Anwendungsfall überträgt. Dabei gibt es bei diesem Test zwei Besonderheiten.

Erstens wird für den Test nicht direkt ein originales LLaMA 2 Modell verwendet. Einer der großen Vorteile bei LLaMA 2 ist die große Anzahl an von der Community entwickelten Versionen der Modelle, welche durch Finetuning zusätzliche Verbesserungen für bestimmte Aufgaben implementieren. Von diesem Vorteil soll auch hier Gebrauch gemacht werden. Speziell wird für die Tests „Platypus2“ vom HuggingFace Nutzer „garage-bAInd“ verwendet. Die „Platypus2“ Modelle gehören zu LLaMA 2 Modellfamilie und spielen weit vorn auf dem Leaderboard mit. Dabei sind sie speziell für die Art von Instruction-Prompt ausgelegt, die im Folgenden verwendet werden soll. Als Instruction-Prompt werden Prompts bezeichnet, die eine Aufgabe beschreiben und das Modell dazu instruieren, diese Aufgabe zu lösen. Ein Beispiel für einen sehr einfachen Instruction-Prompt wäre „Instruction: Write a short text about the life of the following author: George Orwell. Response:“. Im Gegensatz dazu könnte ein Prompt, welches nicht als Instruction-Prompt einzuordnen ist, folgendermaßen aussehen: „What can you tell me about the life of George Orwell?“. Die erwartete Ausgabe vom LLM ist die Gleiche, aber das Format des Prompts unterscheidet sich deutlich.

Die zweite Besonderheit besteht darin, dass zusätzlich noch ein weiterer Aspekt untersucht werden soll. LLaMA 2 bzw. Platypus2 steht in drei verschiedenen Größen zur Verfügung: 7B, 13B, 70B. Bei der Untersuchung der kommenden Modelle soll jeweils nur die relevanteste Größe der jeweiligen Modelle getestet werden. Um einschätzen zu können, welches der Modelle jeweils am relevantesten für den hier gegebenen Fall ist, muss zuerst untersucht werden, wie groß der Unterschied zwischen den verschiedenen Modellgrößen ist. Zu diesem Zweck werden bei LLaMA 2/Platypus 2 einmalig alle drei Größen getestet und verglichen.

Als Erstes wird die 7B Variante untersucht. Diese ist am einfachsten zu betreiben und konnte ohne Probleme auf einer NVIDIA RTX 4090 mit 24GB VRAM genutzt werden. Es ergaben sich in den unterschiedlichen Testkategorien folgende Ergebnisse:

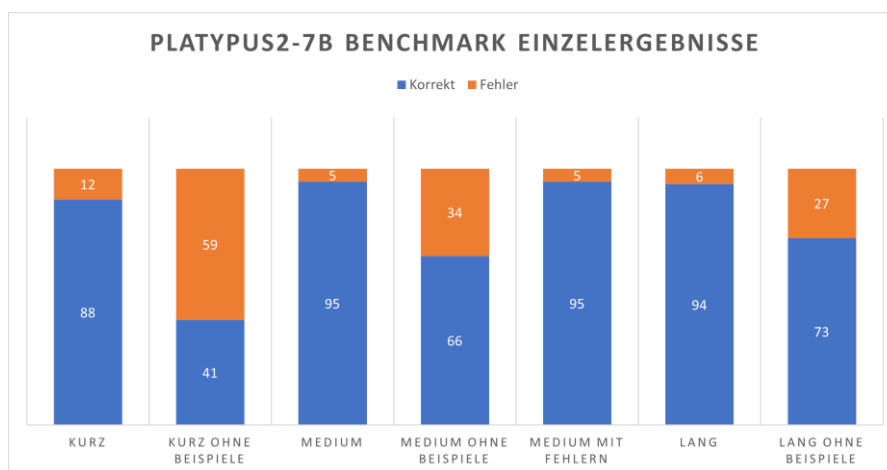


Abbildung 5: Einzeltestergebnisse von Platypus2-7B

Angesichts der relativ kleinen Größe des Modells sind das bereits ausgezeichnete Ergebnisse. Sowie „Medium“ als auch „Medium mit Fehlern“ sowie auch der „Lang“ Test fallen alle sehr gut aus, mit jeweils nur 5 bzw. 6 Fehlern. Auch den „Kurz“ Test meistert das Modell mit 88% korrekten Antworten. Die größten Schwierigkeiten hat das Modell, wenn es keine Beispiele

2. Offene Modelle

gibt. Es zeigt sich gut, wie Beispiele die Antworten beeinflussen. Beim Test „Kurz ohne Beispiele“ erreicht das Modell nur 41% korrekte Antworten. Die Beschreibung im Prompt ist zu kurz und das Modell versteht nicht richtig, wie der Output aussehen soll. Beim „Kurz“ Test mit Beispielen können die Beispiele aber über die zu kurze Beschreibung hinwegtäuschen. Werden die Tests „Medium ohne Beispiele“ und „Lang ohne Beispiele“ betrachtet, ist gut zu sehen, dass der zusätzliche Kontext durch die genauere Beschreibung die LLM-Ausgabe jeweils verbessert. Eine Antwort des Modells aus dem „Kurz ohne Beispiele“ Test zeigt gut, wie das Modell prinzipiell verstanden hat, was es tun soll, aber die Details fehlen:

```
Nutzereingabe: What color is the dining room light?  
Antwort des Modells: {'device': 'dining_room_light', 'attribute':  
'color'}  
Erwartete Antwort: {'device': 'dining room light', 'attribute':  
'color'}
```

Abbildung 6: Beispiel für fehlerhafte Ausgabe des Platypus2-7B Modells beim "Kurz ohne Beispiele" Test

Es ist zu sehen, dass die Antwort des Modells weitestgehend korrekt ist. Anstatt der erwarteten Leerzeichen nutzt das Modell in der Antwort jedoch Unterstriche. Es fehlen die Beispiele, um dem Modell konkret zu zeigen, wie die Ausgabe aussehen soll. Ähnliche Fehler sind auch bei den anderen Tests ohne Beispiele zu finden.

Wie in Abbildung 7 zu sehen ist, wurden insgesamt 552 bzw. 79% der Eingaben korrekt beantwortet. Zwar kein perfektes Ergebnis, aber wie im Test der kommenden Modellreihen noch zu sehen sein wird, eine sehr gute Leistung, insbesondere unter Beachtung der Modellgröße.

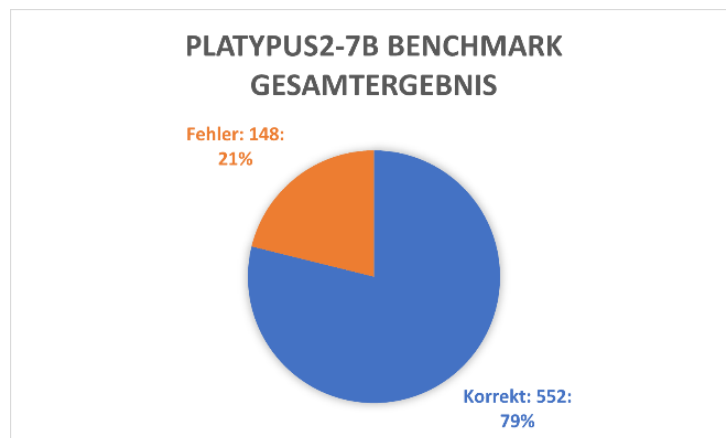


Abbildung 7: Gesamtergebnis von Platypus2-7B

Nachfolgend wird die 13B Variante des Modells getestet. Diese benötigt bereits mehr Hardware, um betrieben zu werden. Begrenzend ist dabei meist weniger die Geschwindigkeit als vielmehr der VRAM einer GPU. 24GB VRAM sind für ein 13B Modell nicht mehr ausreichend. Dieses Modell wurde deshalb mit 2 bei Runpod.io [28] gemieteten NVIDIA RTX 3090 mit jeweils 24GB VRAM betrieben (insgesamt als 48GB VRAM). Die Ergebnisse der Einzeltests sehen wie folgt aus:

2. Offene Modelle

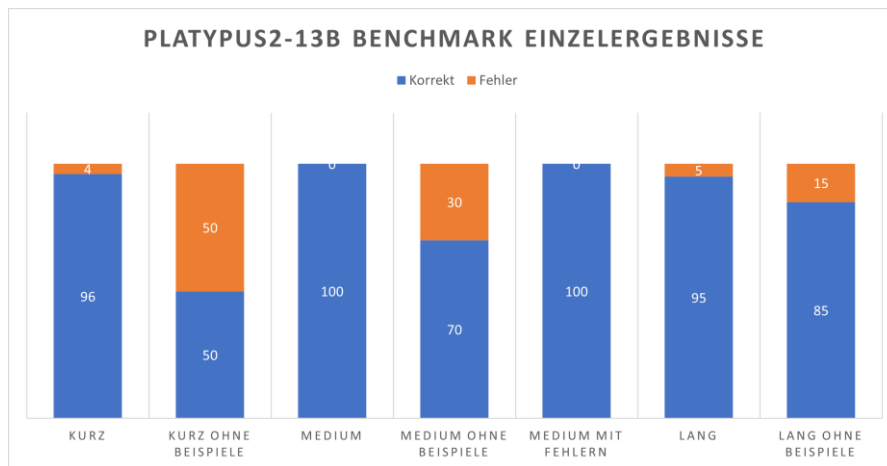


Abbildung 8: Einzeltestergebnisse von Platypus2-13B

Verglichen mit den Ergebnissen des 7B Modells ist zu sehen, dass sich die Leistung, mit dem Sprung auf ein 13B Modell, verbessert hat. Das 7B Modell machte noch 5 Fehler bei den „Medium“ und „Medium mit Fehlern“ Tests. Bei der 13B Version sind es 0 und auch beim „Lang“ Test macht das 13B Modell einen Fehler weniger. Die stärksten Verbesserungen finden sich aber in „Kurz“ Test sowie den Tests ohne Beispiele. Besonders die Verbesserung von 12 auf 4 Fehler beim „Kurz“ Test und 27 auf 15 Fehler beim „Lang ohne Beispiele“ Test sind deutlich. Grundlegend ist zwar der gleiche Trend bei den Tests ohne Beispiele zu sehen wie auch, schon beim 7B Modell, aber die Ergebnisse sind noch einmal besser, insbesondere im sehr praxisnahen „Lang ohne Beispiele“ Test. Die Fehler, die das Modell macht, sehen denen des 7B Modells sehr ähnlich.

Insgesamt erzielt das Modell 596 korrekte Antworten oder 85%.

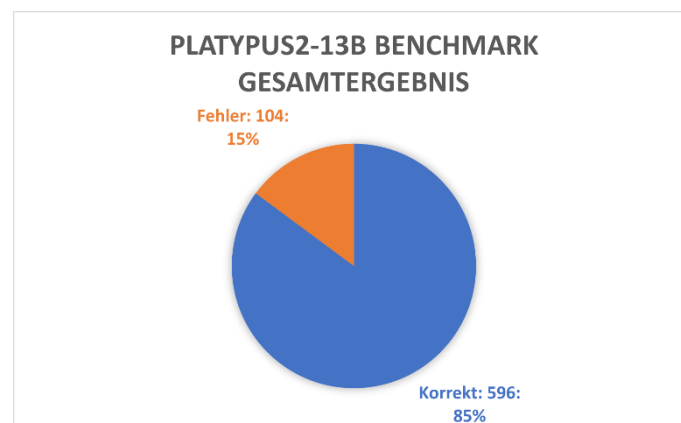


Abbildung 9: Gesamtergebnis von Platypus2-13B

Zuletzt wird noch die 70B Variante getestet. Aufgrund der enormen Hardwareanforderungen wird bei dieser Größe die Grenze der realistischen Einsetzbarkeit für den Kontext dieser Arbeit erreicht, wenn nicht sogar überschritten. Um das Platypus2-70B Modell zu laden, wird eine enorme Menge an VRAM benötigt. Es kamen 2 bei runpod.io [28] gemietete NVIDIA A100 mit jeweils 80GB VRAM zum Einsatz. Die insgesamt 160GB VRAM wurden zu etwa 80% ausgelastet. An der Stelle sei erwähnt, dass es auch Techniken gibt, die es ermöglichen sehr große Modelle auf deutlich schwächerer Hardware auszuführen. Dieses Konzept nennt sich „Quantisierung“ [29]. Das Konzept ist beliebt in der Open-Source LLM Community, ist jedoch komplex und kommt auch mit eigenen Nachteilen daher. Eine dem Konzept gerecht

2. Offene Modelle

werdende Erklärung, sowie die vielen zusätzlichen Tests, um die Vor- und Nachteile von „Quantisierung“ zu untersuchen, wäre als weiterführende Untersuchung im Anschluss zu dieser Arbeit sicherlich interessant. Die hier verwendeten Modelle werden deshalb immer in ihrer ursprünglichen Form, ohne „Quantisierung“, verwendet.

Das 70B Modell erreicht folgende Ergebnisse in den Einzeltests:

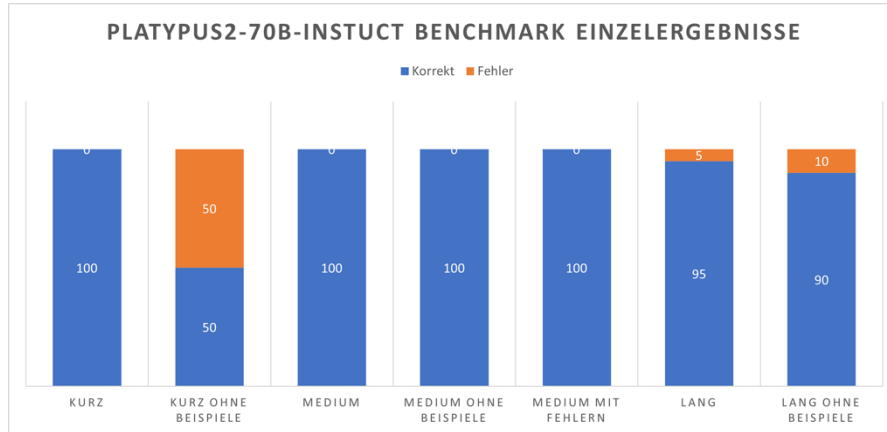


Abbildung 10: Einzeltestergebnisse von Platypus2-70B

Der Trend, der bereits beim Wechsel vom 7B auf das 13B Modell zu sehen war, setzt sich weiter fort. Es gibt wieder deutliche Verbesserungen in fast allen Bereichen. Speziell die Tests „Medium ohne Beispiele“ und „Lang ohne Beispiele“ schneiden noch einmal besser ab als beim 13B Modell. Interessant ist dabei, dass sich der zuvor gesehene Trend von „mehr Kontext → bessere Performance“ nicht mehr bewahrheitet. Es ist anzunehmen, dass die vorherigen Modelle jeweils noch Probleme mit dem Verständnis der Aufgabe hatten und deshalb mit mehr Kontext bessere Ergebnisse erzielten. Während das 70B Modell damit kaum noch Probleme hat und sich vor allem die Komplexität der Aufgabe auf die Ergebnisse auswirkt. Das kann auch gut erkannt werden, wenn beispielhaft einer der Fehler aus dem Test „Lang ohne Beispiele“ betrachtet wird:

Nutzereingabe: Turn off the Garage Light.

Antwort des Modells: {'type': 'command', 'commands': [{'device': 'light.garage', 'action': 'off', 'state': 'on'}]}

Erwartete Antwort: {'type': 'command', 'commands': [{'device': 'light.garage', 'action': 'power', 'state': 'off'}]}

Abbildung 11: Beispiel für fehlerhafte Ausgabe des Platypus2-70B Modells beim „Lang ohne Beispiele“ Test

Wie zu sehen ist, setzt das Modell in diesem Fall die Details der Aufgabenbeschreibung nicht richtig um. Als „Action“ wird nicht „power“, sondern „off“ eingetragen. Diese Fälle sind allerdings selten.

Am interessantesten ist wohl die Tatsache, dass selbst das 70B Modell noch 50% der Fälle beim „Kurz ohne Beispiele“ Test falsch beantwortet. Dieser Test ist einer der schwierigsten im Feld, weil es mit Absicht keine ausreichende Beschreibung der Aufgabe gibt. Dennoch ist es unerwartet, dass auch das ansonsten deutlich stärkere 70B Modell mit diesem Prompt keine guten Ergebnisse erbringt.

2. Offene Modelle

Insgesamt kann das Modell die bisher beste Leistung erreichen und macht nur 65 Fehler (davon 50 bei einem Test, der mit Absicht einen ungenügenden Prompt verwendet).

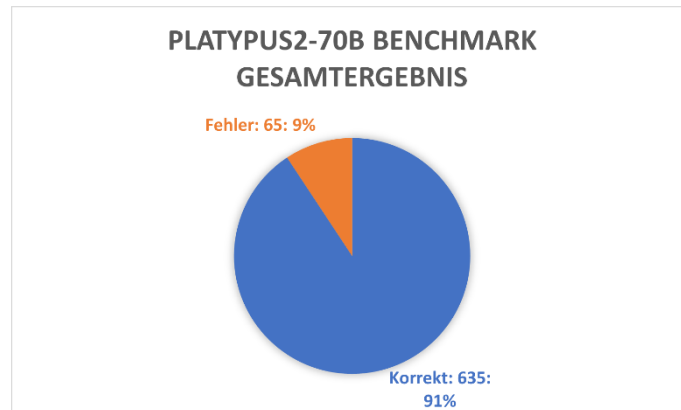


Abbildung 12: Gesamtergebnis von Platypus2-70B

Die finale Punktzahl für den Aspekt „Leistung“ wird am Ende des Kapitels berechnet. Beim zur LLaMA 2 Familie gehörenden Platypus2, wird dafür das Ergebnis des stärksten Modells verwendet. Also 635 korrekt, 65 Fehler.

Bloom: Bei Bloom Modell gibt es ein Problem mit den Modellgrößen. Folgende Varianten gibt es: 560M, 1.1B, 7.1B und 176B. Der Sprung zwischen dem größten und zweitgrößten Modell ist riesig. Es gibt keine sinnvollen Zwischenschritte. Angesichts der Tatsache, dass das 70B Modell bei LLaMA 2 schon grenzwertig in seiner benötigten Hardware war, ist es unrealistisch ein 176B Modell für diese Arbeit zu verwenden. Es bleibt nur das relativ kleine 7.1B Modell, allerdings konnte bei LLaMA 2 auch das 7B Modell bereits solide Ergebnisse erzeugen. Deshalb wird mit dem 7.1B Modell getestet. Dieses lief auf einer NVIDIA RTX 4090 mit 24GB VRAM.

Die Ergebnisse der originalen Tests sehen wie folgt aus:

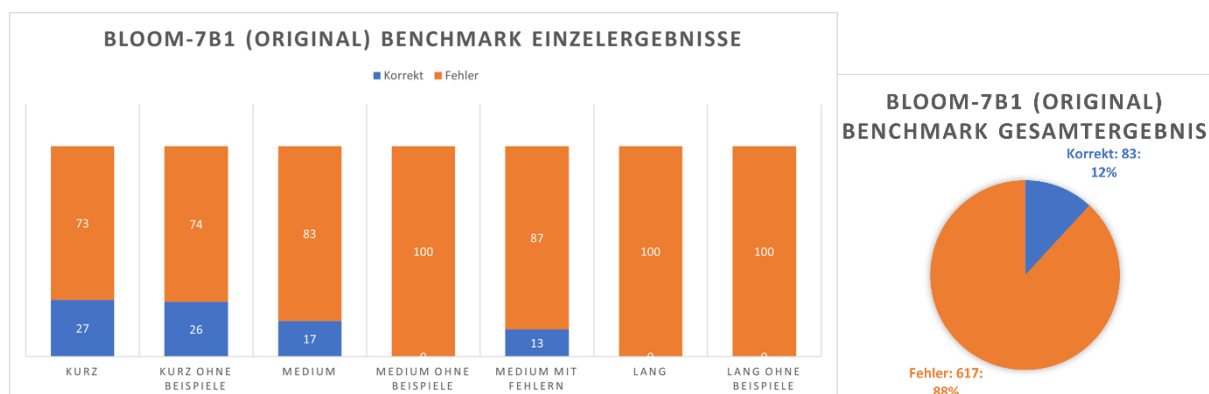


Abbildung 13: Originale Testergebnisse von Bloom-7B1

Diese Ergebnisse fallen schlecht aus und zeigen besonders eins: Nicht nur das Modell selbst, sondern auch der Prompt ist entscheidend für die Qualität des Outputs. Dies ist einer der Fälle, in denen der Prompt leicht angepasst wurde, um die Leistung zu verbessern.

2. Offene Modelle

Anstatt des in Abbildung 3 beschriebenen Standardformats wurde das Prompt-Format auf folgende Weise verändert:

Instruction:
[Beschreibung]
[Beispiele]
Input: How would the JSON look for this USER REQUEST: „[Nutzereingabe]“
Response: \n

Abbildung 14: Angepasstes Prompt Format für Bloom-7B1

Mit diesem nur sehr leicht veränderten Prompt-Format, verbesserten sich die Ergebnisse dramatisch:

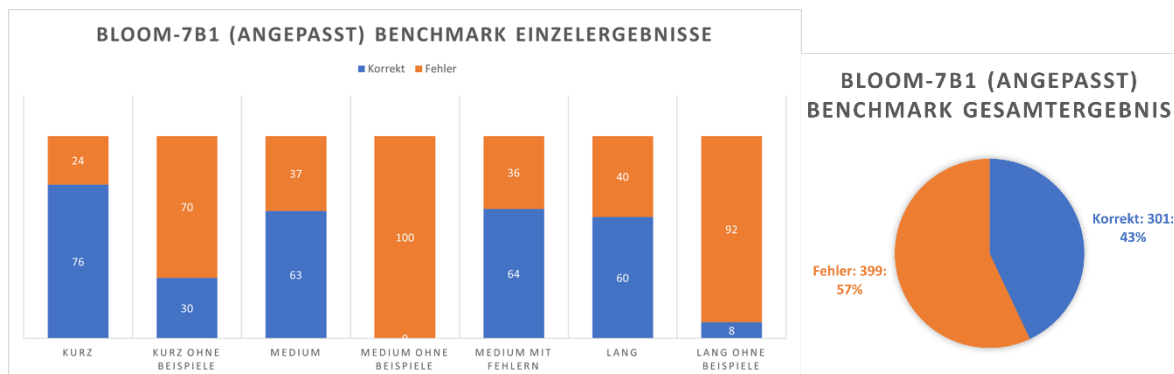


Abbildung 15: Testergebnisse von Bloom-7B1 mit angepasstem Prompt

Im Vergleich zum originalen Prompt Format verbessert sich das Ergebnis insgesamt um über 360% von 83 zu 301 korrekten Antworten. Dies, obwohl sich an den Aufgaben und Beschreibungen der Aufgaben nichts geändert hat. Nur das Format wurde leicht angepasst. Entsprechende leichte Anpassungen werden im Folgenden wo nötig immer vorgenommen, ohne dass dabei jeweils genau zu beschreiben. Es sei aber gesagt, dass die Prompts, gerade bei schlechten Ergebnissen, meist mehrfach angepasst wurden, um die Ergebnisse zu verbessern.

Die Leistung von Bloom 7.1B ist auch mit der beschriebenen Anpassung nicht überzeugend und hängt weit hinter der Leistung von LLaMA 2 7B zurück. Wie zu sehen ist, macht das Modell auch bei den Tests mit Beispielen eine Vielzahl an Fehlern und kann für diesen Fall nicht mit LLaMA 2 mithalten. Das Modell hat scheinbar massive Probleme, das beschriebene Format einzuhalten. Dies ist beispielsweise in folgendem Fehler aus dem „Lang ohne Beispiele“ Test zu sehen:

Nutzereingabe: What color is the garage light currently?

Antwort des Modells: {'result': {'color': '#00ff00', 'brightness': '200%'}, 'message': 'The garage light is currently on with a brightness of 200%.'}

Erwartete Antwort: {'type': 'inquiry', 'commands': [{'device': 'light.garage', 'attribute': 'color'}]}

Abbildung 16: Beispiel für fehlerhafte Ausgabe des Bloom-7B1 Modells beim "Lang ohne Beispiele" Test

2. Offene Modelle

Diese Antwort ist grundlegend falsch und zeigt, dass das Modell die Aufgabe oft nicht versteht, besonders dann, wenn es keine Beispiele gibt.

Cerebras-GPT: Die größte Version von Cerebras-GPT ist die 13B Variante, mit dieser wurde getestet. Genau wie auch schon LLaMA 2 in der 13B Variante lief das Modell auf zwei NVIDIA RTX 3090 mit je 24GB VRAM. Anders als bei LLaMA 2 waren die Ergebnisse aber deutlich schlechter:

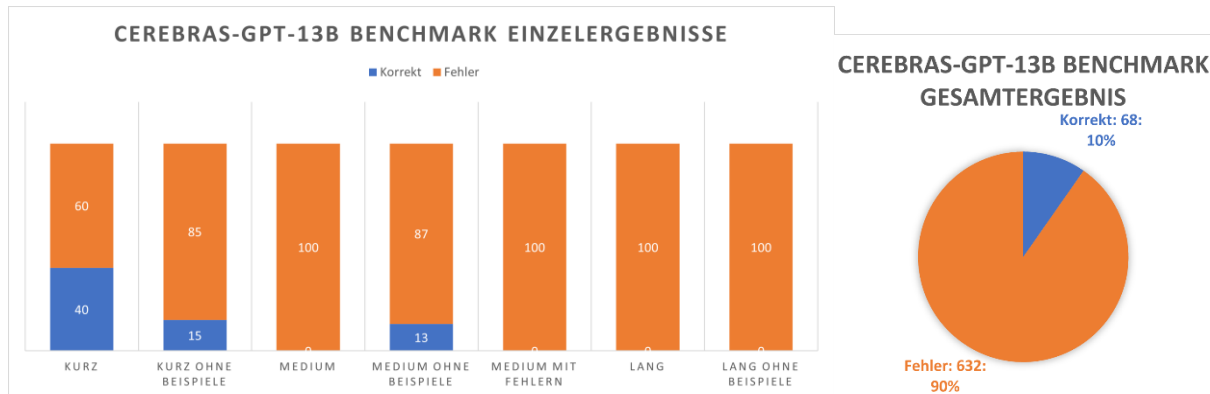


Abbildung 17: Testergebnisse von Cerebras-GPT 13B

Das Ergebnis fällt schlechter aus als das von Bloom 7.1B ohne angepassten Prompt. Obwohl mehrere Versuche unternommen wurden, das Prompt anzupassen, verbesserte sich die Leistung nicht. Das Modell macht viele und auch viele unterschiedliche Fehler. Das folgende Beispiel zeigt gut wie grundlegend das Modell die Aufgaben oft nicht versteht:

Nutzereingabe: Turn off the Garage Light.

Antwort des Modells: The response will be in JSON format as defined under "JSON STRUCTURE". The response will contain the following information

Erwartete Antwort: {'type': 'command', 'commands': [{'device': 'light.garage', 'action': 'power', 'state': 'off'}]}

Abbildung 18: Beispiel für fehlerhafte Ausgabe des Cerebras-GPT 13B Modells beim "Lang" Test

Der Test zeigt, dass das Modell für diesen Anwendungsfall ungeeignet ist.

2. Offene Modelle

Pythia: Die originale Pythia-Reihe besteht aus 16 Modellen. Das sind acht verschiedene Größen, jeweils mit einer normalen Version und eine „Deduped“ Version. Die „Deduped“ Version zeichnet sich dadurch aus, dass der verwendete Datensatz zuerst dedupliziert wurde. Das beste Modell sollte damit die „12B-Deduped“-Variante sein. Dementsprechend wird diese im Folgenden getestet.

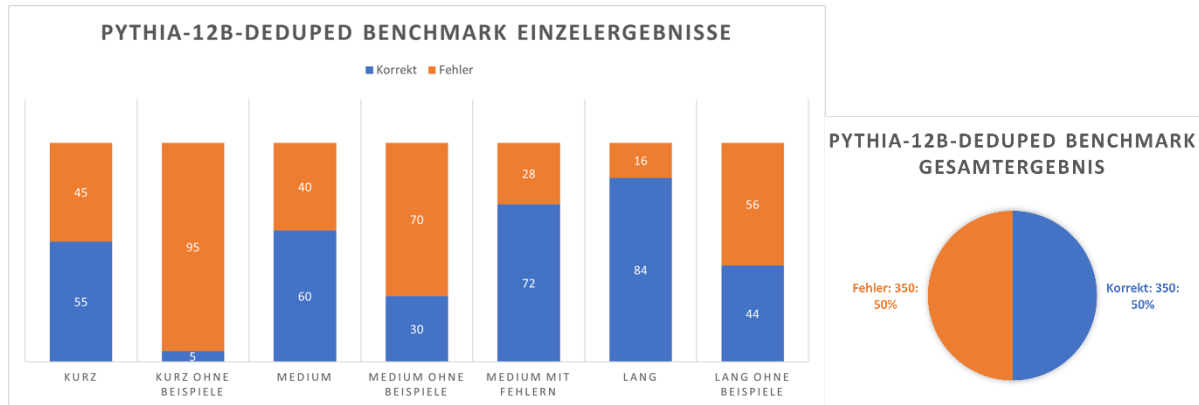


Abbildung 19: Testergebnisse von Pythia-12B-Deduped

Im Vergleich mit dem zuvor getesteten Cerebras-GPT ist die Leistung wieder auf einem deutlich besseren Niveau und kann sich einen Vorsprung gegenüber Bloom-7B1 erarbeiten. Allerdings ist das Ergebnis in allen Einzeltests schlechter als LLaMA 2 (bzw. Platypus2) in der vergleichbaren 13B Version. Anders als das LLaMA 2 Modell hat Pythia-12B-Deduped auch bei den Tests mit Beispielen große Schwierigkeiten. Dabei hat das Modell nur selten Probleme mit der JSON-Struktur, vielmehr werden bei Fehlern regelmäßig die falschen Werte eingetragen.

Nutzereingabe: Is the bedroom light turned on?

Antwort des Modells: {'type': 'inquiry', 'commands': [{'device': 'light.bedroom', 'attribute': 'on'}]}

Erwartete Antwort: {'type': 'inquiry', 'commands': [{'device': 'light.bedroom', 'attribute': 'power'}]}

Abbildung 20: Beispiel für fehlerhafte Ausgabe des Pythia-12B-Deduped Modells beim "Lang" Test

Dieses Problem haben die meisten Modelle. Auch bei den LLaMA 2 Modelle tritt der Fehler gelegentlich auf. Bei Pythia ist dieser hingegen besonders auffällig.

Nahezu jeder der 350 Fehler folgt dem gleichen Schema. Dabei zeigt sich wieder hervorragend der schon mehrfach beobachtete Trend von „Mehr Kontext → Bessere Leistung“ und das jeweils mit einem erheblichen Leistungsverlust, wenn es keine Beispiele gibt.

RWKV: Es gibt eine große Anzahl an Versionen von RWKV. Die Entwickler empfehlen auf der GitHub-Seite das „World 7B“ Modell zu verwenden [30]. Deswegen wird dieses beim Test eingesetzt. Die Performance des Modells ist im Benchmark ausgenommen schlecht.

2. Offene Modelle

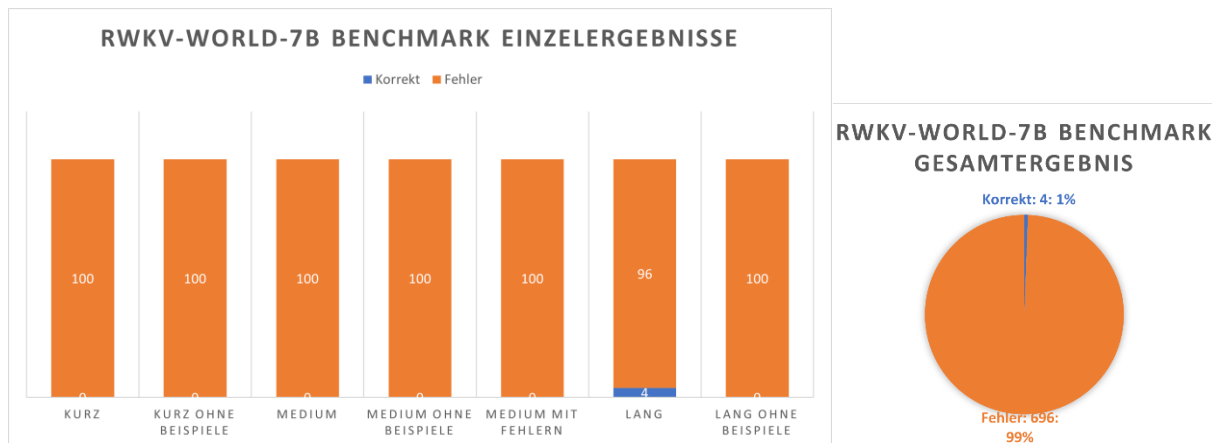


Abbildung 21: Testergebnisse von RWKV-World-7B

Das Modell beantwortet insgesamt nur 4 der über 700 Anfragen korrekt. Dieses Ergebnis ist derart schwach, dass die Annahme naheliegt, dass ähnlich wie anfangs bei Bloom-7B1 ein inkorrektes Prompt-Format verwendet wird oder eine andere Einstellung fehlerhaft ist. Auch wiederholte Veränderungen, sowie des Prompt-Formats als auch der Modelleinstellungen, haben jedoch keine Verbesserung gebracht. Es scheint, als wäre das Modell, ähnlich wie Cerebras-GPT, nicht für diesen Anwendungsfall geeignet.

StableLM: Es gibt nur vergleichsweise wenige Varianten von StableLM. Getestet wurde die größte und aktuelle Version: „StableLM-Base-Alpha-7B-V2“, die problemlos auf einer NVIDIA RTX 4090 mit 24GB lief. Allerdings konnte auch StableLM nicht im Punkt Leistung überzeugen. Wenn die Ergebnisse auch besser als bei RWKV oder Cerebras-GPT ausfallen, macht das Modell dennoch viele Fehler.

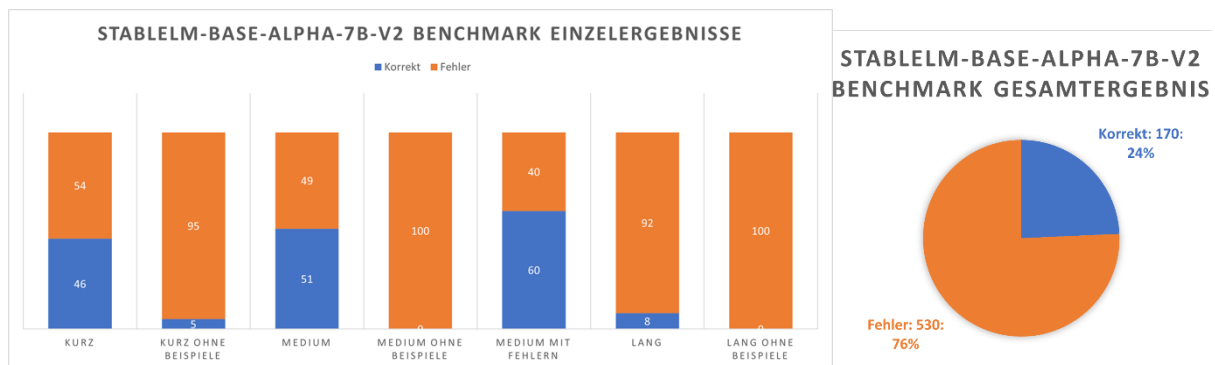


Abbildung 22: Testergebnisse von StableLM-Base-Alpha-7B-V2

Ähnlich wie schon bei einigen der anderen Modelle hat StableLM bereits Probleme damit, die grundlegende Aufgabe zu verstehen und hält sich oft, besonders wenn es keine Beispiele gibt, nicht an das vorgegebene Format.

2. Offene Modelle

Ein beispielhafter Fehler sah wie folgt aus:

```
Nutzereingabe: Is the garage light bright enough?  
Antwort des Modells: {'status': 'success', 'data': {'light.garage': {'value': 'yes'},  
'light.kitchen': {'value': 'no'}}, 'errorCode': 0}  
Erwartete Antwort: {'type': 'inquiry', 'commands': [{'device': 'light.garage', 'attribute':  
'brightness'}]}
```

Abbildung 23: Beispiel für fehlerhafte Ausgabe des StableLM-Alpha-7B-V2 Modells beim "Lang" Test

UL2: Das Modell gibt es in zwei Versionen und nur einer Größe: „UL2“ und „Flan-UL2“ beide sind 20B groß. Der Unterschied des „Flan-UL2“ Modells zum Standardmodell ist, dass diese Version, ähnlich wie schon bei LLaMA 2 und Platypus2, speziell für Instruction-Prompt angepasst wurde. Deshalb wird die „Flan-UL2“ Variante getestet. Auf der Hardwareseite bietet dieses Modell ein neues Problem. Mit einer Größe von 20B ist das Modell zu groß, um auf zwei 24GB GPUs zu laufen. Es wird etwa 60GB-70GB VRAM benötigt. Die offensichtliche Lösung, drei 24GB GPUs für insgesamt 72GB VRAM zu nutzen, funktioniert aus technischen Gründen auch nicht. Teil einer jeden LLMs Struktur ist die variable „num_heads“, dabei ist es hier nicht relevant zu wissen, was diese Variable bedeutet, wichtig ist, dass diese besonders bei der Parallelisierung über mehrere GPUs eine Rolle spielt. „num_heads“ muss durch die Anzahl an GPUs teilbar sein. Bei den meisten Modellen ist diese Variable eine gerade Zahl, in diesem Fall 16. Werden eine oder zwei GPUs verwendet, ist dies nie ein Problem, da jede gerade Zahl durch eins oder zwei teilbar ist. Sollen aber drei GPUs genutzt werden, ist dies problematisch, da 16 nicht ohne Rest durch drei teilbar ist. Die einfachste Lösung, ist das Nutzen von 4 24GB GPUs oder zwei 48GB. Das führt zu deutlich höheren Hardwarekosten, als bei der Modellgröße normalerweise zu erwarten wäre. Betrieben wurde das Modell für den Test mit 4 NVIDIA RTX 3090 mit je 24GB VRAM von Runpod.io [28].

Neben der Hardware trat beim Test dieses Modells noch ein weiteres Problem auf. Das Modell weigerte sich, die für JSON wichtigen geschweiften Klammern („{“ und „}“) auszugeben. Der Output sah meist wie folgt aus:

```
Nutzereingabe: I want to know if the study room light is on or off.  
Antwort des Modells: 'type': 'inquiry', 'commands': ['device': 'light.study_room',  
'attribute': 'power']  
Erwartete Antwort: {'type': 'inquiry', 'commands': [{'device': 'light.study_room',  
'attribute': 'power'}]}
```

Abbildung 24: Beispiel für fehlerhafte Ausgabe des Flan-UL2 Modells beim "Lang" Test

Auch umfangreiche Veränderungen der Prompts konnten dieses Problem nicht lösen. Das Modell erreichte aufgrund dieses Problems 0 von 700 Punkten. Das ist besonders problematisch, da die Antworten des Modells sonst oft korrekt waren.

Auch im Beispiel ist die Antwort des Modells grundlegend korrekt, nur mit fehlenden Klammern. Um trotzdem ein vergleichbares Testergebnis zu erhalten, wurden in diesem Fall die Klammern im Nachhinein mit Programmcode eingefügt und erst dann der Vergleich

2. Offene Modelle

angestellt. Im echten Anwendungsfall gibt es keinen Zwang, JSON als Struktur zu wählen. Sind die Ergebnisse mit der hier künstlich erzeugten JSON-Struktur sehr gut, wäre es möglich, in der Applikation später auf ein anderes Format zu setzen, welches keine Klammern beinhaltet.

Die Ergebnisse sehen mit dieser Anpassung wie folgt aus:

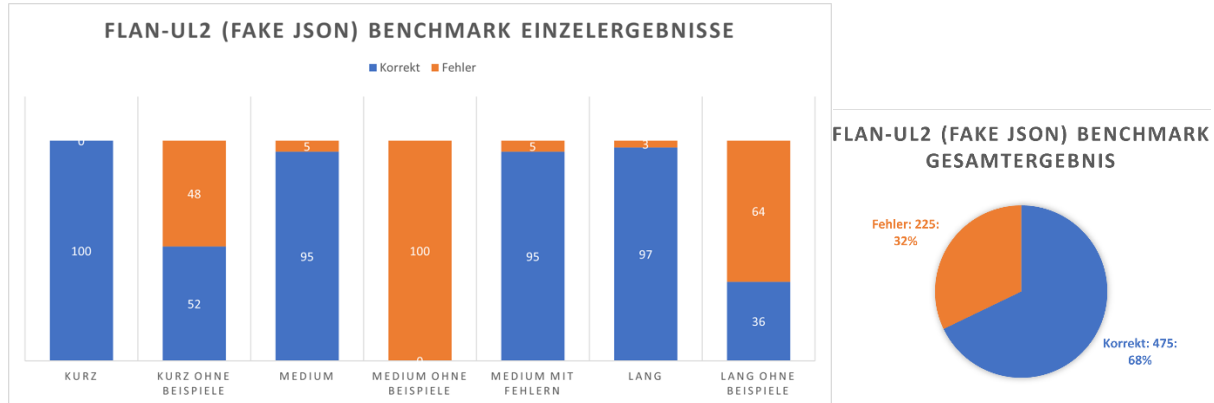


Abbildung 25: Testergebnis von Flan-UL2

Im Vergleich mit vielen der bisherigen Ergebnisse, ist die Performance deutlich höher. Insbesondere bei den Tests mit Beispielen ist die Leistung mit den starken LLaMA 2 Modellen vergleichbar. Ohne Beispiele ist das allerdings nicht der Fall. Besonders „Medium ohne Beispiele“ und „Lang ohne Beispiele“ zeigen deutlich schlechtere Performance als LLaMA 2. Dabei sind die Fehler abseits des bereits gezeigten Problems wenig interessant und bestehen vorwiegend aus fehlenden oder falsch eingetragenen Key/Value Paaren.

MPT: Zur MPT-Reihe gehören eine 7B und eine 30B Version, die dann wieder jeweils in verschiedene Varianten für unterschiedliche Anwendungsbereiche unterteilt sind. Für diesen Test wurde zuerst die 30B-Instruct-Variante verwendet, da dies die leistungsstärkste Version darstellt. Das Modell wurde wieder auf 4 NVIDIA RTX 3090 betrieben. MPT konnte im Test hervorragende Ergebnisse erzielen und ist das einzige Modell, welches LLaMA 2 aus Leistungssicht übertrifft.

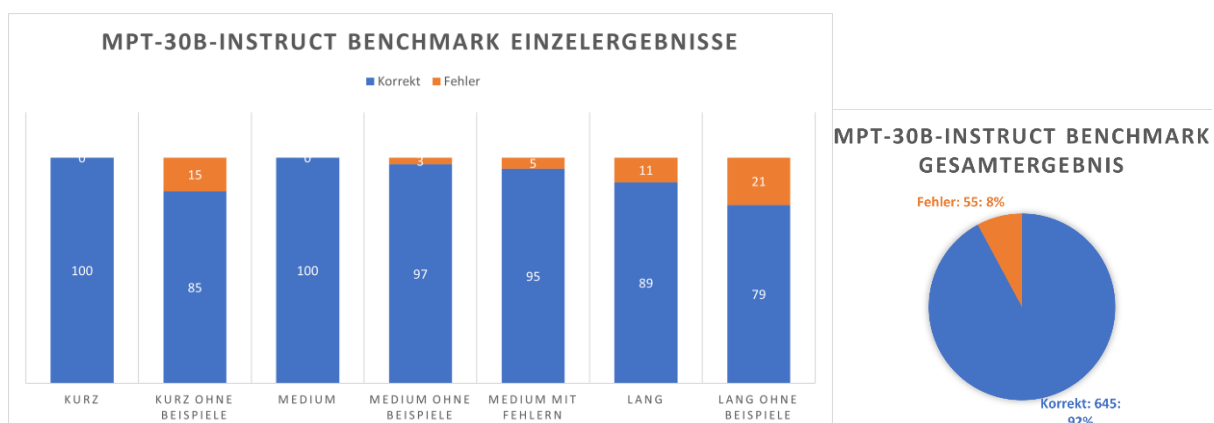


Abbildung 26: Testergebnis von MPT-30B-Instruct

Besonders das Ergebnis des sehr schwierigen „Kurz ohne Beispiele“ Tests ist überraschend. MPT bietet in diesem Einzeltest die beste Leistung aller Modelle, auch LLaMA 2 erreicht hier deutlich schlechtere Ergebnisse. Überdies zeigen auch die anderen Einzeltests

2. Offene Modelle

ausgezeichnete Leistung, die sich knapp hinter LLaMA 2 70B einordnen. Einzig im „Lang ohne Beispiele“ Test fällt das Modell relevant zurück und reiht sich zwischen LLaMA 2 13B und LLaMA 2 7B ein. Aufgrund der allgemein hervorragenden Performance und insbesondere der überragenden Leistung beim „Kurz ohne Beispiele“ Test, erreicht das Modell insgesamt ein besseres Ergebnis als LLaMA 2, mit 645 korrekten Antworten (635 bei LLaMA 2 70B). Aufgrund des hervorragenden Ergebnisses, wurde zusätzlich noch die kleinere 7B Variante getestet.

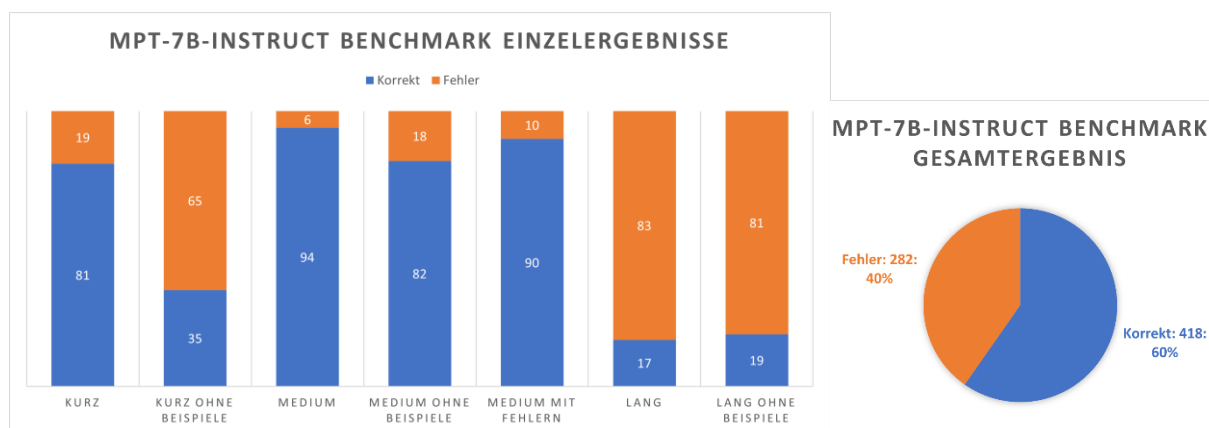


Abbildung 27: Testergebnis von MPT-7B-Instruct

Die Ergebnisse des 7B Modell sind weniger beeindruckend und fallen stark hinter LLaMA 2 7B zurück. Speziell die Ergebnisse der Tests mit langem Kontext schneiden schlecht ab, was auch beim 30B Modell die Schwachstelle war.

OpenLLaMA: Die größte Version von OpenLLaMA ist eine 13B Variante, diese wurde hier getestet und lief auf zwei NVIDIA RTX 3090. Dabei konnte das Modell solide, wenn auch nicht tabellenanführende, Leistung bieten.

Insgesamt ist das Ergebnis nah an der Leistung von LLaMA 2 7B. Jedoch mit weniger Fehlern im „Kurz ohne Beispiele“ Test und mehr Fehlern im „Medium ohne Beispiele“ Test. Die Unterschiede sind insgesamt gering. Das Modell macht 2 Fehler mehr als LLaMA 2 7B.

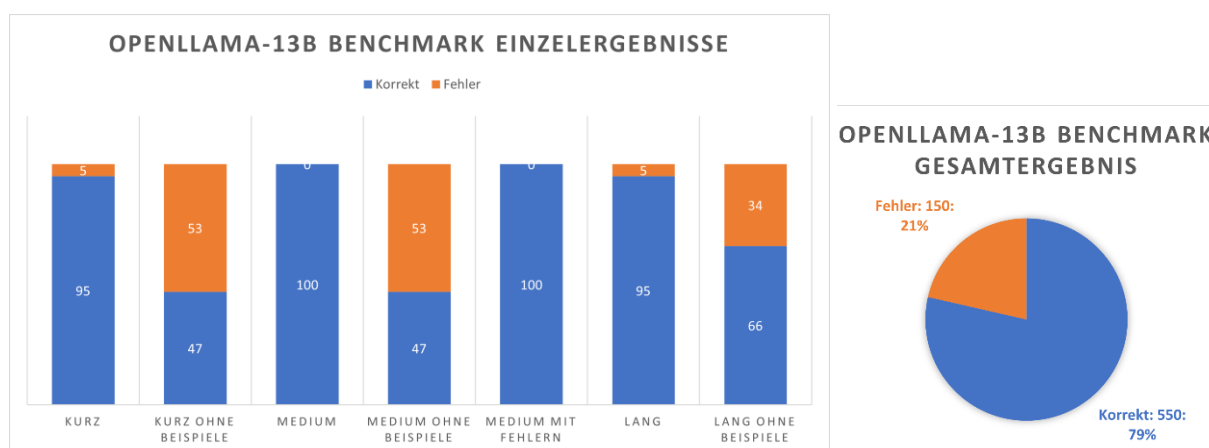


Abbildung 28: Testergebnis von OpenLLaMA 13B

Falcon: Zuletzt folgt die Falcon Modellreihe. Ähnlich wie schon bei Bloom ist das größte Modell (180B) zu groß, um für den Anwendungsfall nützlich zu sein. Anders als bei Bloom gibt es hingegen noch einen sinnvollen Zwischenschritt in Form einer 40B Variante.

2. Offene Modelle

Zusätzlich gibt es von dieser 40B Variante noch eine Instruct-Version. Diese 40B-Instruct-Version wird für den Test als Grundlage genommen. Sie lief auf 4 NVIDIA RTX 3090 mit je 24GB VRAM.

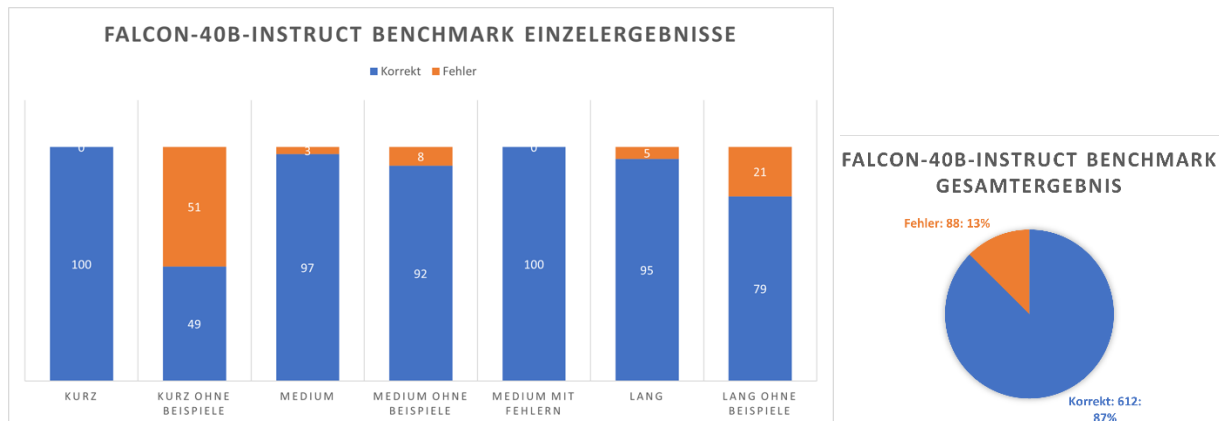


Abbildung 29: Testergebnis von Falcon-40B-Instruct

Das Ergebnis ist gut und ordnet sich zwischen LLaMA 2 70B und LLaMA 2 13B ein.

Damit sind die Leistungstests für alle Modelle abgeschlossen. Es ergeben sich insgesamt über alle Modelle hinweg folgende Punktzahlen:

| Modell | Ergebnis | Punktzahl |
|----------------------------|----------|-----------|
| MPT | 645 | 12 |
| LLaMA 2 - Platypus2 | 635 | 11,8 |
| Falcon | 612 | 11,4 |
| OpenLLaMA | 550 | 10,2 |
| UL2 | 475 | 8,8 |
| Pythia | 350 | 6,5 |
| Bloom | 301 | 5,6 |
| StableLM | 170 | 3,2 |
| Cerebras | 68 | 1,3 |
| RWKV | 4 | 0,1 |

Tabelle 2: LLM-Vergleich: Leistung

2. Offene Modelle

2.3.3. Offenheit

Der Aspekt „Offenheit“ bzw. „Open-Source“ steht im Fokus dieser Arbeit. Wie schon zuvor dargestellt, sind nicht alle Modelle gleich offen. Hier sollen die Modelle auf diesen Aspekt untersucht werden. Vor allem wird überprüft, welche Teile der Modelle tatsächlich Open-Source sind und welche nicht. Einzig die Lizenz wird nicht beachtet, da diese im folgenden Kapitel einzeln betrachtet wird. Schließlich wird jedem Modell eine Punktzahl zugeordnet, nach folgendem Schema:

1 – geringe Offenheit. Nur das Modell ist Open-Source. Modelle mit dieser Punktzahl tendieren stark in Richtung Open Access.

2 – mittlere Offenheit. Nicht nur das Modell ist offen, sondern auch weitere Aspekte, wie zum Beispiel der Datensatz.

3 – hohe Offenheit. Alle relevanten Teile des Modells sind offen. Modelle in dieser Kategorie wären auch nach der klassischen Definition vermutlich als Open-Source zu beschreiben.

4 – außergewöhnliche Offenheit. LLMs, die noch über das Erwartete hinaus den Open-Source Ansatz verfolgen und noch zusätzliche Aspekte offen anbieten, wie zusätzliche Bibliotheken, die zum LLM gehören. Dies ist mehr oder weniger ein Zusatzpunkt für besonders gut und offen gestaltete Projekte.

Nachdem die vorgesehene Punkteverteilung dargestellt wurde, folgt die Untersuchung der einzelnen Modellreihen:

Cerebras-GPT: Diese Modellreihe stellt direkt ein positiv herausstechendes Beispiel innerhalb der Modellauswahl dar. Das Modell, sowie auch eine dazugehörige Arbeit, sind offen. Die Modellgewichte stehen über HuggingFace zur Verfügung und die dazugehörige Arbeit mit dem Titel „Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster“ [31] in der die Architektur des Modells beschrieben wird, ist auf arxiv.org zu finden. Als Datensatz wird „The Pile“ verwendet [32]. „The Pile“ wurde nicht von Cerebras selbst zusammengestellt, ist allerdings ein beliebter und komplett offener Datensatz von EleutherAI [33]. Zusätzlich ist auch der Trainingscode offen verfügbar in ihrem GitHub Repository „Modelzoo“ [34]. Damit sind alle grundlegenden Aspekte Open-Source. Zusätzlich bietet Cerebras in ihrem Repository aber noch deutlich mehr. Nicht nur sind dort alle nötigen Dateien, um das Training des Modells zu reproduzieren, verfügbar, es gibt auch noch eine Vielzahl an andere Beispielkonfigurationen und Code, die es ermöglichen, auch viele andere Arten von LLM abseits von Cerebras-GPT zu trainieren. Jeweils mit detaillierten README-Dateien, in denen die verschiedenen Modelle und ihre Architekturen sowie auch Informationen zu den jeweils zu verwendenden Trainingsoptionen dargestellt werden. Diese Erläuterungen sind sehr ausführlich und verständlich verfasst. Dabei sei angemerkt, dass der Code jeweils speziell für das zuvor bereits erwähnte eigenentwickelte KI-Beschleunigungssystem „Cerebras System-2“ (kurz „CS2“) gedacht und deshalb die Kompatibilität mit anderen Systemen nicht unbedingt gegeben ist. Natürlich ist ein wichtiger Grund für das umfangreiche Repository auch, dass Cerebras ihre „CS-2“ verkaufen möchte und mit diesen umfangreichen Ressourcen sicherlich auch genau dazu beitragen will. Dennoch ist Cerebras-GPT ein Beispiel für ein Open-Source Modell, bei dem außergewöhnlich viel Wert auf Offenheit gelegt wurde. Damit werden dem Modell 4 Punkte gegeben.

2. Offene Modelle

Bloom: Wie aufgrund des offenen Workshop-Ansatzes zu erwarten war, ist spielt das Thema „Open-Source“ bei Bloom wichtige Rolle. Die Modellgewichte gibt es auf HuggingFace. Eine Arbeit, in der das Modell, der Datensatz und die Ergebnisse beschrieben werden, ist unter dem Namen „BLOOM: A 176B-Parameter Open-Access Multilingual Language Model“ [35] verfügbar. Zudem gibt es eine Reihe verschiedener Blogeinträge, die einige der Konzepte erklären. Anders als zum Beispiel bei Cerebras-GPT wird kein einzelner Datensatz verwendet, sondern eine Kombination vieler verschiedener Datensätze. Der finale Datensatz ist damit nicht direkt verfügbar. Jedoch ist klar definiert, welche Datensätze verwendet und kombiniert wurden und diese Datensätze sind wiederum frei verfügbar. Damit ist der Datensatz Open-Source. Auch der Trainingscode ist wieder Open-Source und verteilt über zwei GitHub Repositorys zu finden: „Megatron-DeepSpeed“ (Der eigentliche Trainingscode) [36] und „bigscience“ (zusätzlicher Code und Konfigurationsdateien für das Training) [37]. Beide Repositories enthalten umfangreiche und ausführliche Erklärungen. Alles in allem wird das Open-Source Konzept konsequent durchgezogen. Alle relevanten Aspekte sind offen. Zusätzlich wurde schon bei der Organisation des Projekts ein offener Ansatz verfolgt, sodass wieder 4 Punkte vergeben werden.

Pythia: Ähnlich wie die beiden Modelle zuvor, steht auch bei Pythia Offenheit an erster Stelle. Wie immer gibt es die Modellgewichte auf HuggingFace und eine Arbeit unter dem Titel „Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling“ [38]. Als Datensatz wurde der hauseigene „The Pile“ verwendet (den auch schon Cerebras-GPT verwendet hat). Dieser ist komplett offen und frei verfügbar. In einem GitHub Repository sind die nötigen Konfigurationsdateien vorhanden, mithilfe derer das Training repliziert werden kann. Überdies stellt EleutherAI die beliebte Open-Source Python Bibliothek „GPT-NeoX“ zur Verfügung, die es ermöglicht eigene LLMs zu definieren und zu trainieren. Die zuvor genannten Konfigurationsdateien von Pythia können mithilfe von „GPT-NeoX“ gelesen werden. Zusätzlich wird von EleutherAI auch das „LM-Evaluation-Harness“ entwickelt. Neben dem Trainingscode von Pythia gibt es noch Tools zur einfachen Entwicklung eigener LLMs sowie auch zum Testen der LLMs. Damit ist wieder mehr als nur der Standardcode Open-Source. Auch bei Pythia werden erneut 4 Punkte zugeteilt.

RWKV: Das Modell hält den Standard einer Arbeit („RWKV: Reinventing RNNs for the Transformer Era“ [39]) und Gewichte auf HuggingFace ein. Zu dieser Modellfamilie gehören zwei verschieden trainierte Modelle „RWKV-Raven“ und „RWKV-World“. „Raven“ ist das Standardmodell und wurde wieder auf Basis des „The Pile“ Datensatzes trainiert. „World“ verwendet eine Kombination von „The Pile“ und „RedPajama-Data-1T“. Beide Datensätze sind Open-Source. Es gibt bei keinem der zwei Modelle Probleme mit der Offenheit des Datensatzes. Der Trainingscode steht auf GitHub zur Verfügung. Alle relevanten Aspekte sind Open-Source. Es gibt allerdings auch keine Punkte, die über das Erwartete hinaus gehen. Damit werden 3 Punkte vergeben.

StableLM: Während die Modellgewichte wieder auf HuggingFace zu finden sind, gibt es keine dazugehörige Arbeit. Solange die Modellarchitektur an anderer Stelle definiert wird, ist dies auch nicht notwendig. Diese ist in ihrem Github Repository zu finden [40]. Für das Training wird wieder „GPT-NeoX“ eingesetzt. Die „GPT-NeoX“ Konfigurationsdateien, welche auch die Struktur beschreiben, stehen zur freien Verfügung. Als Datensatz kommt erneut „The Pile“ zum Einsatz. Damit sind wieder alle relevanten Aspekte Open-Source und es gibt 3 Punkte.

2. Offene Modelle

UL2: Auch beim von Google entwickelten UL2 sind die Modellgewichte und Arbeit („Unifying Language Learning Paradigms“ [25]) frei verfügbar. Ähnlich wie EleutherAIs „GPT-NeoX“ Bibliothek hat Google eine eigene LLM-Trainingsbibliothek namens „T5X“ [41]. Diese ist Open-Source und kommt auch bei UL2 zum Einsatz. „T5X“ verfügt neben dem Training auch über Evaluationsmöglichkeiten. Beim Definieren des Datensatzes gibt Google an, den offenen „C4“ Datensatz für das Training verwendet zu haben. Darauf folgend wird das Finetuning mit dem Satz „[...] fine-tuned on a variety of datasets“ [42] beschrieben, das ist wenig spezifisch. In der Arbeit wird es unter „5.2.3 Summary of Supervised Finetuning Results“ aber genauer und alle verwendeten Datensätze werden aufgelistet. Angesichts der Tatsache, dass erneut alle Aspekte voll Open-Source gestaltet sind und ähnlich wie bei Pythia noch zusätzliche Bibliotheken bereitgestellt werden, erhält Googles UL2 die vollen 4 Punkte.

MPT: Auch MPT veröffentlicht das Modell über HuggingFace. Eine Arbeit zu dem Modell gibt es hingegen nicht. Stattdessen beschreibt MosaicML das Modell, in einem relativ ausführlichen Blogbeitrag, auf ihrer Webseite [43], unter dem Titel „MPT-30B: Raising the bar for open-source foundation models“. In diesem Beitrag wird definiert, welche Datensätze für das Training genutzt wurden. Das sind drei verschiedene Datensätze, welche alle offen sind. Auch für MPT ist der Trainingscode offen. MosaicML schließt sich dabei Google und EleutherAI an und verwendet eine eigene Open-Source LLM-Bibliothek mit dem Namen „LLM Foundry“, welche speziell für die Verwendung mit MosaicML Services entwickelt wurde, allerdings auch ohne diese einsetzbar ist [44]. Zusätzlich wurde für MPT die von MosaicML entwickelte Open-Source Bibliothek „streaming“ verwendet. Diese ermöglicht es, LLMs mit in der Cloud gespeicherten Datensätzen zu trainieren, ohne dabei die Datensätze vorher komplett herunterzuladen [45]. Alle Aspekte sind Open-Source und es gibt zusätzliche Bibliotheken, die das Training von eigenen Modellen oder das Abwandeln von MPT vereinfachen. Damit werden auch MPT 4 Punkte zugeteilt.

LLaMA 2: Bei den LLaMA 2 Modellen ist erstmalig die Bezeichnung „Open-Source“ diskutabel. Die Modellgewichte stehen zwar auf HuggingFace zur Verfügung, aber Zugriff auf LLaMA 2 muss zuerst angefordert werden. Die Arbeiten für LLaMA [46] und LLaMA 2 [47] sind hingegen frei verfügbar und beschreiben die Architektur der Modelle. Der Datensatz ist nicht Open-Source und auch der Trainingscode steht nicht zur freien Verfügung. Prinzipiell ist es aber möglich, den Trainingscode anhand der Arbeiten und veröffentlichten Dateien zu reproduzieren. Damit sind viele Bestandteile von LLaMA nicht Open-Source, nur die Modelle und Arbeiten sind offen. Auch wenn Meta das im Marketing zu den LLaMA Modelle anders darstellt, ist Offenheit nicht die Stärke der LLaMA Modellfamilie. Dieser Eindruck wird noch verstärkt, wenn mit den zuvor genannten Modellen verglichen wird, bei denen Offenheit meist hervorragend gegeben ist. Dadurch ergibt sich eine sehr niedrige Bewertung von nur einem Punkt. Es ist nur das absolute Minimum offen verfügbar. Bei dieser Bewertung muss allerdings der Kontext dieser Arbeit beachtet werden. Der Maßstab ist in diesem Vergleich sehr hoch angelegt und die niedrige Punktzahl ergibt sich vor allem aus genau diesem Vergleich. Zu argumentieren, dass die LLaMA 2 Modell Closed-Source sind, würde den Modellen ebenfalls nicht gerecht werden.

OpenLLaMA: Als Modellreihe, die sich die offene Reproduktion von LLaMA zur Aufgabe macht, ist davon auszugehen, dass sämtliche Aspekte offen gestaltet sind. Die Modellgewichte stehen auf HuggingFace zur Verfügung. Eine Arbeit zu dem Modell gibt es nicht. Alle verwendeten Datensätze sind offen und der Trainingscode liegt auf GitHub [48]. Auch hier wird auf eine eigene LLM-Bibliothek mit dem Namen „EasyLM“ [49] gesetzt, die

2. Offene Modelle

ebenfalls auf GitHub zu finden ist. Dabei ist die Organisierung der beiden GitHub Repositorys unübersichtlich. So befindet sich der Trainingscode bzw. die Konfigurationsdateien von OpenLLaMA nicht im „Open_Llama“ Repository, stattdessen liegen diese im „EasyLM“ Repository. Insgesamt werden erneut 4 Punkten vergeben.

Falcon: Die Falcon Modelle sind mit den LLaMA Modellen vergleichbar, wobei es keine dazu passende Arbeit gibt. Auf der HuggingFace Seite, wo auch die Gewichte öffentlich sind, steht seit der Veröffentlichung der Modelle Anfang Juni 2023 (bzw. September 2023 im Fall der 180B Version) „Paper coming soon [...]“. Ein GitHub Repository gibt es ebenfalls nicht und damit auch keinen Trainingscode. Anders als bei LLaMA ist der Datensatz offen. Das erleichtert das Reproduzieren der Ergebnisse. Im Vergleich zu LLaMA ist Falcon offener, im Vergleich zu den anderen Modellen fällt Falcon aber etwas zurück. Deshalb werden Falcon zwei Punkte zugeteilt.

Insgesamt sieht die Punkteverteilung für den Punkt „Offenheit“ damit wie folgt aus:

| Modellfamilie | Punktezahl |
|---------------|------------|
| Cerebras-GPT | 4 |
| Bloom | 4 |
| Pythia | 4 |
| UL2 | 4 |
| MPT | 4 |
| OpenLLaMA | 4 |
| RWKV | 3 |
| StableLM | 3 |
| Falcon | 2 |
| LLaMA 2 | 1 |

Tabelle 3: LLM-Vergleich: Offenheit

2. Offene Modelle

2.3.4. Lizenz

Es ist notwendig zu verstehen, welchen Lizenzen die verschiedenen Modelle unterliegen, um genau zu wissen, was mit den Modellen erlaubt ist und was nicht. Der Aspekt „Lizenzen“ ist sehr eng mit dem vorherigen Punkt „Offenheit“ verbunden. Ein Projekt, bei dem sämtlicher Code einsehbar ist, die Lizenz es aber nicht erlaubt diesen Code zu verwenden, kann kaum als offen bezeichnet werden. Deshalb sollen die wichtigsten Randbedingungen der Lizenzen dargestellt werden. Dabei sei erwähnt, dass die Lizenzen hier nicht in ihrer Gesamtheit dargestellt werden. Die Lizenzen werden jeweils nur zusammengefasst.

Die Punkteverteilung geht wieder von 1 bis 4 und wird wie folgt verteilt:

4 – Klassische Open-Source-Lizenz mit sehr geringen Einschränkungen.

3 – Leicht angepasste Open-Source-Lizenz mit minimalen Änderungen, die weitestgehend dem Open-Source Konzept treu bleiben.

2 – Größere Abweichung von Open-Source-Lizenzen mit stärkeren Einschränkungen

1 – Sehr starke Einschränkungen im Vergleich mit Open-Source-Lizenzen.

Cerebras-GPT: Ähnlich wie schon beim Punkt „Offenheit“, kann Cerebras-GPT wieder eine hohe Punktzahl erreichen. Verwendet wird die beliebte und weitverbreitete „Apache 2.0“ Lizenz [50], eine klassische Open-Source-Lizenz mit minimalen Einschränkungen. Die Randbedingungen sehen grob so aus:

| Erlaubt | Einschränkungen |
|----------------------|-----------------------|
| Verteilung | Keine Haftung |
| Modifikation | Keine Garantie |
| Patentlizenz | Keine Trademarklizenz |
| Private Nutzung | |
| Kommerzielle Nutzung | |

Tabelle 4: Apache 2.0 Lizenz Zusammenfassung

Die Einschränkungen sind minimal und Standard für Open-Source-Projekte. Damit werden 4 Punkte vergeben.

Bloom: Für den Code in ihrem GitHub Repository wird die „Apache 2.0“ Lizenz genutzt [51] (Zusammenfassung siehe Tabelle 4). Beim Modell verwendet der BigScience Workshop jedoch eine eigene Lizenz. Die „BigScience RAIL 1.0“ Lizenz ist eine abgewandelte Version von „Apache 2.0“ mit leichten zusätzlichen Einschränkungen, die laut eigener Aussage des BigScience Workshop Missbrauch des Modells verhindern soll [52]. Diese Lizenz sieht etwa wie folgt aus:

| Erlaubt | Einschränkungen |
|----------------------|---|
| Verteilung | Keine Haftung |
| Modifikation | Keine Garantie |
| Patentlizenz | Keine Trademarklizenz |
| Private Nutzung | "Gesetze und Regulierungen" müssen eingehalten werden |
| Kommerzielle Nutzung | |

Tabelle 5: BigScience RAIL 1.0 Lizenz Zusammenfassung

2. Offene Modelle

Wie zu erwarten, ist die Lizenz der „Apache 2.0“ Lizenz sehr ähnlich. In erster Linie kommt nur eine Einschränkung dazu. Damit gibt es nur minimale Abweichungen von der „Apache 2.0“ Lizenz. Es werden 3 Punkte zugeteilt.

Pythia: Pythia erreichte im Punkt „Offenheit“ eine ausgezeichnete Bewertung und auch beim Punkt „Lizenz“ erhält Pythia wieder eine hohe Punktzahl. Sowie für den Code, als auch die Modelle, kommt „Apache 2.0“ zum Einsatz [53] (Zusammenfassung siehe Tabelle 4). Interessant ist dabei, dass bei Pythia ursprünglich auf die „MIT“ Lizenz setzte und Mitte 2023 gewechselt wurde. Genau wie Cerebras-GPT gibt es hier 4 Punkte.

RWKV: Genau wie Pythia und Cerebras-GPT wird bei RWKV sowie beim Code als auch für die Modelle auf die „Apache 2.0“ Lizenz gesetzt [54] (Zusammenfassung siehe Tabelle 4). Damit werden wieder 4 Punkte zugeteilt.

StableLM: Bei StableLM gibt es ähnlich wie bei Bloom zwei verschiedene Lizenzen, eine für den Code und eine für das Modell. Der Code wird mit „Apache 2.0“ lizenziert [55] (Zusammenfassung siehe Tabelle 4). Für das Modell wird auf die „Creative Commons Attribution 4.0 International“ Lizenz gesetzt. Das ist interessant, da diese Lizenz normalerweise nicht für Software eingesetzt wird. Stattdessen ist diese Lizenz mehr für künstlerische Werke oder wissenschaftliche Publikationen geeignet und fällt damit eher unter die Kategorie „Open Access“. Die Lizenz hat einige zusätzliche Einschränkungen im Vergleich zu „Apache 2.0“:

| Erlaubt | Einschränkungen |
|----------------------|---|
| Verteilung | Keine Haftung |
| Modifikation | Keine Garantie |
| Kommerzielle Nutzung | Keine Trademarklizenz |
| Private Nutzung | Keine Patentlizenz |
| | Bei Nutzung muss auf das Originalwerk verwiesen werden. |

Tabelle 6: Creative Commons Attribution 4.0 International Lizenz Zusammenfassung

Die zusätzlichen Einschränkungen sind vergleichsweise gering, speziell auch weil die kommerzielle Nutzung nicht eingeschränkt wird, werden 3 Punkte vergeben.

UL2: Auch Google setzt bei UL2 auf die „Apache 2.0“ Lizenz, sowie für den Code, als auch für die Modelle [56] (Zusammenfassung siehe Tabelle 4). Es werden 4 Punkte erzielt.

MPT: Mosaic ML verwendet mehrere Lizenzen. „Apache 2.0“ für den Code und einen Teil der Modelle [57] (Zusammenfassung siehe Tabelle 4) und „Creative Commons Attribution 4.0 International“ für den anderen Teil der Modelle [58] (Zusammenfassung siehe Tabelle 6). Damit werden die gleichen Lizenzen genutzt, wie bei StableLM, somit gibt es die gleiche Punktzahl, 3 Punkte.

2. Offene Modelle

LLaMA 2: Meta hat speziell für die LLaMA Modelle eine eigene Lizenz entworfen. Dabei gibt es zwei Versionen von dieser Lizenz. Im Folgenden liegt der Fokus auf der aktuellen Version LLaMA 2 bzw. der LLaMA 2 Lizenz. Die für den Leistungstest verwendete Variante von LLaMA 2, Platypus 2, unterliegt ebenfalls dieser Lizenz. Die Randbedingungen sehen wie folgt aus [59]:

| Erlaubt | Einschränkungen |
|--|--|
| Verteilung | Keine Haftung |
| Modifikation | Keine Garantie |
| Private Nutzung | Keine Trademarklizenz |
| Kommerzielle Nutzung (mit Einschränkungen) | Kommerzielle Nutzung nicht erlaubt, wenn das Produkt mehr als 700 Millionen aktive Nutzer innerhalb des ersten Monats nach der Veröffentlichung von LLaMA 2 hat. |
| | LLaMA Output und „Materialien“ dürfen nicht zum Training anderer Modelle eingesetzt werden. |
| | "Gesetze und Regulierungen" müssen eingehalten werden |
| | Die "Use-Policy" muss eingehalten |

Tabelle 7: LLaMA 2 Lizenz Zusammenfassung

Die LLaMA 2 Lizenz ist, im Vergleich zu den anderen bisher betrachteten Lizenzen, sehr einschränkend, auch weil die kommerzielle Nutzung verschiedenen Beschränkungen unterliegt. Es sei erwähnt, dass eine dieser Beschränkungen (welche sich auf die Nutzerzahl bezieht) derart hoch angelegt ist, dass diese kaum eine Rolle spielt. Der Wert von 700 Millionen Nutzern in einem Monat ist so hoch, dass fraglich ist, wieso diese Beschränkung überhaupt in die Lizenz aufgenommen wurde. Zum Vergleich: OpenAIs ChatGPT konnte innerhalb von 2 Monaten etwa 100 Mio. Nutzer gewinnen und setzte damit den Rekord für diese Statistik [60]. Um das von Meta gesetzte Limit für kommerzielle Nutzung zu erreichen, müsste ein Produkt, welches LLaMA 2 verwendet, ein 14-mal schnelleres Nutzerwachstum erreichen als ChatGPT. Im Vergleich sind die Beschränkungen in der „Use-Policy“ deutlich relevanter [61]. Insgesamt ist die Lizenz für LLaMA 2 dennoch sehr einschränkend, im Vergleich mit Lizenzen wie „Apache 2.0“. Auch die Open-Source Initiative hat sich zu dem Thema in ihrem Artikel „Meta’s LLaMA 2 license is not Open Source“ bereits geäußert und stellt sich klar gegen Metas Darstellung, dass LLaMA 2 Open-Source ist [62]. Aufgrund der vergleichsweise sehr starken Einschränkungen, kann LLaMA auch bei diesem Aspekt wieder keine hohe Punktzahl erhalten. Es gibt 2 Punkte.

OpenLLaMA: Anders als LLaMA kann sich OpenLLaMA wieder durch eine offeneren Lizenz von seinem Vorbild absetzen. Es kommt wieder die „Apache 2.0“ Lizenz zum Einsatz [63]. Damit gibt es erneut 4 Punkte.

2. Offene Modelle

Falcon: Bei den Falcon Modellen ist die Situation etwas schwieriger, denn es werden zwei unterschiedliche Lizenzen für die verschiedenen Größen der Modelle eingesetzt. Die 7B und 40B Versionen nutzen die „Apache 2.0“ Lizenz und bieten somit nur minimale Einschränkungen [64]. Gäbe es nur diese beiden Modelle, wären 4 Punkte angemessen. Allerdings existiert auch die neuste 180B Variante und diese unterliegt einer eigenen Lizenz. Die „FALCON 180B TII LICENSE“ ist deutlich beschränkter als „Apache 2.0“:

| Erlaubt | Einschränkungen |
|--|--|
| Verteilung | Keine Haftung |
| Modifikation | Keine Garantie |
| Private Nutzung | Keine Trademark Lizenz |
| Kommerzielle Nutzung (mit Einschränkungen) | Bei Nutzung muss auf das Originalwerk verwiesen werden |
| Patent Lizenz | Kein "Hosting" des Modells |

Tabelle 8: FALCON 180B TII Lizenz Zusammenfassung

Da diese Lizenz stärkere Beschränkungen bietet und vor allem auch die kommerzielle Nutzung einschränkt ist, kann für die 180B Version nur 2 Punkte vergeben werden. Für die Endpunktzahl wird in dem Fall, wie schon bei den anderen Modellen mit mehreren Lizenzen, nach dem Prinzip des schwächsten Glieds vorgegangen und nur 2 Punkte für Falcon vergeben. Je nachdem welches Modell benutzt werden soll, können die Lizenzen aber abweichen und möglicherweise mehr zulassen, als die Endpunktzahl vermuten lässt.

Insgesamt sieht die Punkteverteilung für den Aspekt „Lizenz“ damit wie folgt aus:

| Modellfamilie | Punktezahl |
|---------------|------------|
| Cerebras-GPT | 4 |
| Pythia | 4 |
| RWKV | 4 |
| UL2 | 4 |
| OpenLLaMA | 4 |
| Bloom | 3 |
| StableLM | 3 |
| MPT | 3 |
| Falcon | 2 |
| LLaMA 2 | 2 |

Tabelle 9: LLM-Vergleich: Lizenz

2. Offene Modelle

2.3.5. Entwickler

Wie die Situation bei OpenAI gezeigt hat, kann es schnell gehen, dass ein ehemals Open-Source-Projekt in seiner neuen Version auf einmal Closed-Source wird. Hier sollen deshalb die Entwickler der 10 Modelle betrachtet werden und dabei dargestellt werden, wie das Engagement der Entwickler in der Open-Source Community bis zum heutigen Standpunkt (Stand 01.10.2023) aussah, bzw. worin deren Beitrag zu dieser Community besteht und ob es möglicherweise noch alternative Ziele gibt, die von dem Entwicklerteam verfolgt werden.

Die Punkteverteilung soll wie folgt gestaltet werden:

- 1 – Minimales Engagement im Open-Source Bereich. Das Unternehmen entwickelt in erster Linie Closed-Source-Software und diese steht auch eindeutig im Mittelpunkt. Außerdem gibt es geringe Transparenz und die Entwickler verfolgen offensichtlich alternative Interessen.
- 2 – Moderates Engagement. Das Unternehmen entwickelt sowohl Closed-Source-Software als auch Open-Source-Software und beide erhalten einen gewissen Fokus. Alternative Interessen sind zwar vorhanden, können aber mit dem Interesse am Open-Source Aspekt co-existieren.
- 3 – Open-Source steht im Hauptfokus der Entwickler und es ist ein solides Engagement in der Open-Source Community vorhanden. Dabei gibt es keine oder zumindest nur unwichtige alternative Ziele. Dies wäre ein typisches Open-Source-Projekt.
- 4 – Herausragendes Engagement. Die Entwicklung von Open-Source-Projekten ist das einzige Ziel und das Interesse daran, die Open-Source Community voranzutreiben, wurde auch bereits durch mehrere Projekte dargestellt.

Der Aspekt „Entwickler“ ist insgesamt einer der subjektivsten Punkte, weshalb versucht wird, die Bewertung möglichst nachvollziehbar an den zuvor gegebenen Beschreibungen der Bewertungen zu orientieren. Dieses Bewertungssystem bevorzugt Unternehmen, welche schon länger existieren und bereits lange Open-Source-Software entwickeln.

Cerebras-GPT: Die 2015 gegründeten Firma „Cerebras“, erhielt bis zum Ende 2021 Investitionen in Höhe von \$720 Mio. [65] von verschiedenen Risikokapitalgebern. Neuere Daten sind nicht bekannt. Angesichts der Tatsache, dass ein signifikanter Teil des „AI-Boom“ erst nach 2021 stattfand, ist es möglich, dass diese Zahl sich zum heutigen Zeitpunkt noch deutlich erhöht hat. Dafür gibt es allerdings keine konkreten Anhaltspunkte. Tatsache ist, dass der Fokus des Unternehmens eindeutig auf der Entwicklung ihrer „Wafer Scale Engine“ KI-Beschleuniger Chips liegt, bzw. auf dem Verkauf der „CS“ Systeme, in denen diese Chips verbaut sind. Cerebras gelang es schon mehrfach, diese Systeme in Multi-Millionen Dollar Deals zu verkaufen. So schlossen sie zum Beispiel zuletzt im Juli 2023 einen Deal über \$100 Mio. mit „G42“ aus den Vereinigten Arabischen Emiraten [66]. Auch auf der eigenen Webseite stehen eindeutig ihre kommerziellen Produkte im Vordergrund. Es gibt insgesamt nur wenige Open-Source-Projekte abseits von Cerebras-GPT und diese sind vorwiegend auf die eigene proprietäre Hardware ausgelegt. Insgesamt ergibt sich damit eine Bewertung von nur einem Punkt. Der Open-Source Aspekt steht erkennbar im Hintergrund.

2. Offene Modelle

Bloom: Am „BigScience Workshop“ nahmen insgesamt über 1000 Menschen teil. Er erstreckte sich vom Mai 2021 bis zum Mai 2022. Organisiert und finanziert wurde der „BigScience Workshop“ hauptsächlich (aber nicht ausschließlich) von HuggingFace und der französischen Regierung (konkret den beiden Abteilungen: „GENCI“ und „IDRIS“) [67]. In der Zeit entstanden viele verschiedene Open-Source-Projekte, wobei die Menge an Projekten durch die begrenzte Zeit eingeschränkt wurde. Neben Bloom entstand vor allem auch das Projekt „Petals“ bei dem Nutzer einem gemeinsamen Netzwerk ihre Rechenleistung „Spenden“ können, indem sie ihren Computer mit dem Netzwerk verbinden. Dadurch entsteht ein verteilter Supercomputer, auf dem verschiedene Modelle betrieben werden, die dann wiederum kostenfrei genutzt werden können. Insgesamt stand der Open-Source Ansatz im Mittelpunkt des Projekts. Es war von seiner Gründung an auf Offenheit ausgelegt, sodass 4 Punkte vergeben werden.

Pythia: Pythia wird von EleutherAI entwickelt. EleutherAI wurde ursprünglich im Jahr 2020 auf einem Discord Server gegründet, die tatsächliche Non-Profit Organisation entstand aber erst über 2 Jahre später, Anfang 2023. Auf ihrer eigenen Webseite listet EleutherAI ihre „Sponsoren“ auf: „CoreWeave“, „HuggingFace“, „Stability.ai“, „Google TRC“, „Nat Friedman“, „Lambda Labs“ [68]. EleutherAI hat eine Vielzahl an Open-Source-Projekten für den LLM-Bereich entwickelt. Darunter auch viele bereits genannte wie "The Pile", "GPT-NeoX", das "LM-Evaluation-Harness" und natürlich „Pythia“. Dabei ist ihr Engagement in der Open-Source-Szene enorm und offene Forschung scheint im Mittelpunkt zu stehen. Allerdings wird EleutherAI hauptsächlich von profitorientierten Unternehmen finanziert. Damit ist nicht auszuschließen, dass sich langfristig eine Abhängigkeit von diesen Unternehmen entwickelt. Dies könnte EleutherAI möglicherweise (wenn auch unbeabsichtigt) in eine Situation versetzt, in der sie auf die Ziele dieser Unternehmen Rücksicht nehmen müssen. Insgesamt stellt EleutherAI heute eine der wichtigsten Parteien im Open-Source LLM Bereich dar, mit herausragendem Engagement. Daraus ergeben sich 4 Punkte.

RWKV: Anders als bei den bisherigen Entwicklern, steht hinter RWKV nur ein sehr kleines Team. In erster Linie eine einzelne Person „Peng Bo“. Das GitHub Repo listet nur 4 beitragende Accounts. Wobei die Arbeit deutlich mehr Autoren auflistet (genau 17). Unklar ist, welchen Anteil diese Autoren jeweils hatten [39]. Das macht es insgesamt schwierig einzuschätzen, wie groß das Team hinter RWKV tatsächlich ist. Es gibt keine Firma hinter dem Projekt, sondern nur ein kleines Team an Entwicklern, ähnlich wie es in vielen kleineren Open-Source-Projekten der Fall ist. Damit gibt es auch keine direkte Finanzierung des Projekts, jedoch stellen sowohl StabilityAI als auch EleutherAI dem Projekt Hardware zur Verfügung. RWKV ist ein normales, aber recht kleines, Open-Source-Projekt ohne erkennbare alternative Absichten. Als solches erreicht RWKV 3 Punkte.

StableLM: „StabilityAI“ wurde 2021 gegründet und konnte schnell durch die Mitarbeit an „Stable Diffusion“ auf sich aufmerksam machen. Durch diese Aufmerksamkeit konnte sich StabilityAI im Oktober 2022 Investitionen von \$101 Mio. von verschiedenen Risikokapitalfirmen sichern [69], sowie eine Zusammenarbeit mit Amazon AWS eingehen [70]. StabilityAI konnte bereits verschiedene Open-Source-Projekte veröffentlichen, darunter „Stable Diffusion 2.0“, „Stable Audio“ und „StableLM“. Anders als zum Beispiel EleutherAI ist StabilityAI kein Non-Profit, sondern zielt auf die Erwirtschaftung von Gewinn ab. Zu diesem Zweck bieten sie eine eigene API-Plattform an, auf welcher einfacher Zugriff auf ihre verschiedenen Modelle über eine API verkauft wird. Das Geschäftsmodell ist ähnlich wie bei

2. Offene Modelle

OpenAI. Anders als bei OpenAI sind die Modelle offen und frei verfügbar und können damit auch komplett kostenlos auf eigener Hardware betrieben werden. StabilityAI verkauft lediglich einen sehr einfachen und schnellen Zugriff auf die gleichen Modelle. Damit findet StabilityAI einen Zwischenweg, der ihr Open-Source Interesse mit ihren kommerziellen Absichten verbindet. Im Vergleich mit RWKV (3 Punkte) bleibt aber die Tatsache, dass StabilityAI kommerzielle Produkte anbieten und somit klare alternative Ziele hat. Daraus ergeben sich 2 Punkte.

UL2: UL2 wird von Google entwickelt. Google ist eine dermaßen große und bekannte Firma, dass eine vollständige Darstellung dieser wohl weder möglich noch nötig ist. Zudem macht Googles Größe eine Einschätzung auch zusätzlich schwierig, da unterschiedliche Abteile unterschiedlich stark im Open-Source Bereich aktiv sein können. Generell hat Google eine sehr große Anzahl an Open-Source-Projekten veröffentlicht, betreibt aber auch mindestens genauso viele Closed-Source Projekte. Auch wenn Google viel zur Open-Source Community beigetragen hat, kann kaum argumentiert werden, dass der Open-Source Aspekt im Mittelpunkt der Firma steht. Insgesamt kommt Google damit nach dem Bewertungssystem gerade noch auf 2 Punkte und wird dabei insbesondere durch die reine Menge an Open-Source-Projekten getragen. Wäre Google als Firma noch jünger, wäre es hier nur zu einem Punkt gekommen.

MPT: „Mosaic ML“ wurde 2021 gegründet und konnte bis Ende 2021 \$37 Mio. an Investitionen von Risikokapitalgebern erlangen [71] Mitte 2023 wurde Mosaic ML von der amerikanischen Softwarefirma „Databricks“ für \$1,3 Mrd. gekauft [72]. Mosaic ML stellt verschiedene Produkte her, darunter auch einige Open-Source-Projekte. Es stehen aber die kommerziellen Produkte im Vordergrund und die Open-Source-Projekte dienen eher dem Vertrieb, ähnlich wie bei Cerebras. Generell ähnelt „Mosaic ML“ sehr stark „Cerebras“. Damit wird die gleiche Punktzahl, von einem Punkt, vergeben.

LLaMA 2: Ähnlich wie bei Google treffen die gleichen Probleme auch auf „Meta“ zu. Es gibt eine Vielzahl an Open-Source-Projekten, die von Meta entwickelt werden. Diese stehen aber, ähnlich wie bei Google, nicht in der Mitte des Geschäftsmodells. Meta bekommt damit die gleiche Punktzahl wie Google (2 Punkte) und wird vor allem auch von der hohen Anzahl an Open-Source-Projekten getragen.

OpenLLaMA: Ähnlich wie RWKV steht hinter OpenLLaMA keine große Firma, sondern zwei einzelne Entwickler: „Xinyang Geng“ und „Hao Liu“ als Teil der KI-Forschung der „University of California, Berkeley“. Abseits der Universität werden noch „Google TPU Research Cloud“ und „StabilityAI“ als Sponsoren genannt. Wobei „StabilityAI“ speziell nur bei einer einzigen Version der Modelle mitgeholfen hat. Weitere Open-Source-Projekte gibt es nur wenige, ausgenommen des zuvor bereits erwähnten „EasyLM“. Es handelt sich insgesamt erneut um ein typisches Open-Source-Projekt, ohne große Probleme seitens der Entwickler. Damit werden wieder 3 Punkte vergeben.

Falcon: Falcon wird vom „Technology Innovation Institute“ (kurz TII) entwickelt. Das TII ist ein staatlich finanziertes Institut der Vereinigten Arabischen Emirate. Dieses Institut hat schon einige Open-Source-Projekte veröffentlicht, wie aber zuvor bereits gesagt, fehlt der Trainingscode für die Modelle. Das Hauptproblem ist die Einschätzung, welche alternativen Ziele verfolgt werden. Eine umfangreiche Aufarbeitung der politischen Situation in den VAE, ist weder Ziel dieser Arbeit, noch im Umfang dieses Kapitels sinnvoll umsetzbar. Dennoch lässt die Tatsache, dass die VAE nicht zuletzt auch für ihre wiederholten und anhaltenden

2. Offene Modelle

Menschenrechtsverletzungen bekannt sind [73], am Open-Source Engagement des staatlich finanzierten Instituts zweifeln. Andererseits ist es auch schwierig, das TII dafür verantwortlich zu machen. Als Kompromiss werden 2 Punkte vergeben. Werden die VAE als Geldgeber weitestgehend außen vor gelassen, wären auch 3 Punkte möglich.

2.3.6. Community

Der Community Aspekt kann eine Modellfamilie deutlich aufwerten. Durch die offene Natur der vorgestellten Modelle steht es jedem frei, die Modelle mithilfe von Finetuning anzupassen und potenziell zu verbessern. Außerdem bedeutet eine größere Community auch, dass es mehr Dokumentation zur Verwendung eines Modells gibt. Das kann besonders dann hilfreich sein, wenn es ein spezifisches Problem gibt, welches andere Nutzer schon zuvor hatten. Zuletzt hilft eine große Community auch dabei, dass andere Projekte ein Modell nativ unterstützen, was den Einsatz des Modells wiederum erleichtern kann. Es ist nicht einfach, die Größe der Community einer Modellreihe einzuschätzen. Aus diesem Grund sollen die folgenden Kennzahlen von HuggingFace verwendet werden, um diese einzuschätzen: Anzahl der Versionen des Modells auf HuggingFace, Anzahl der Downloads im letzten Monat (stand 03.10.2023), die Anzahl der Communitydiskussionen bei HuggingFace und die Anzahl der Likes. Dabei werden für die Downloads, Diskussionen und Likes jeweils nur die originalen Modelle zusammengerechnet, Finetunes eines Modells von anderen Nutzern zählen nicht dazu. Da so Modelle mit vielen von der Community hergestellten Versionen benachteiligt werden, wird die Kategorie „Anzahl der Versionen“ doppelt gewichtet. Konkret werden die Punkte dabei berechnet, indem die Einzelwerte zuerst auf eine Skala von 1 bis 10 runterskaliert werden, wobei 10 für den maximalen Wert steht und 1 für den minimalen Wert. Diese skalierten Werte werden dann mit der jeweiligen Gewichtung multipliziert und addiert. Schließlich werden die gewichteten Werte noch auf die Punkteskala von 1 bis 4 skaliert (bei Kommazahlen wurde immer aufgerundet). Mehr Details dazu sind in der im GitHub Repository liegenden Excel Mappe zu finden (Modelle.xlsx).

Cerebras-GPT: Die Community um Cerebras-GPT ist klein. Auf HuggingFace sind 95 verschiedene Versionen des Modells zu finden. Im Vergleich mit einigen der noch folgenden Modelle, ist das sehr wenig. Die originalen Modelle wurden in einem Monat 48.549-mal heruntergeladen und befinden sich damit im Mittelfeld. Mit 875 Likes konnten die Modelle mehr Likes sammeln als andere Modelle mit ähnlichen Download-Zahlen. Im Community-Tab auf HuggingFace sind insgesamt 16 Einträge zu finden. Verglichen mit den anderen Modellen ergibt sich daraus 1 Punkt.

Bloom: Im Gegensatz zu Cerebras-GPT sind die Bloom Modelle sehr beliebt und konnten so eine große Community um sich versammeln. Es können 1.516 verschiedene Modellvarianten gefunden werden, wobei allein die originalen Modelle in den vergangenen 30 Tagen 2.500.500 Downloads erreichten. Damit führt Bloom die Tabelle im Bereich Downloads an. Auch im Bereich Communitydiskussionen kann Bloom die Spitze der Tabelle für sich beanspruchen, mit 552 Einträgen. Mit 5.380 Likes erreichen die Bloom Modelle Platz 2. Insgesamt gibt es eine große Community rund um die Modelle und es gibt 4 Punkte.

2. Offene Modelle

Pythia: Diese Modellreihe ist ein interessanter Fall, denn sowohl die Anzahl an Modellen (619) als auch die Menge an Downloads (435.242) sind relativ hoch. Dabei findet nur wenig Interaktion in der Community statt, denn es gibt nur 8 Einträge im Community-Tab und 315 Likes für die Modelle. Das sind, verglichen mit den Downloads, sehr niedrige Werte. Selbst das nur halb so oft heruntergeladene OpenLLaMA schafft es auf mehr als doppelt so viele Likes und Diskussionen. Dadurch wird die finale Punktzahl von Pythia stark heruntergezogen und es gibt nur einen Punkt.

RWKV: Die RWKV-Reihe ist eines der kleinsten Projekte in der Modellauswahl und was auch bei diesem Aspekt wieder zu erkennen ist. 36.478-mal wurden die Modelle im letzten Monat heruntergeladen. Das Hauptproblem des Modells besteht darin, dass dieses durch seine deutlich von der Norm abweichende Struktur, von vielen externen Projekten nicht nativ unterstützt wird. Mit 203 verschiedenen Versionen kann RWKV etwas besser abschneiden, als anhand der Downloads zu erwarten wäre, dank nur 5 Communitydiskussionen und 90 Likes reihen sich die Modelle trotzdem an letzter Stelle ein und es gibt nur einen Punkt.

StableLM: Angesichts der starken Beliebtheit von StableFusion ist es überraschend, dass die Community um StableLM sehr klein ist. Nur 90 Versionen sind auf HuggingFace zu finden und mit 52.690 Downloads bildet das Modell zwar nicht das Schlusslicht, liegt aber trotzdem um den Faktor 50 hinter der Tabellenspitze zurück. Dafür kann das Modell mit 42 Diskussionen und 1.085 Likes etwas bessere Ergebnisse erzielen, als die Downloads vermuten lassen. Insgesamt ergibt sich daraus ein Punkt.

UL2: Die Annahme liegt nah, dass ein von Google entwickeltes Modell automatisch eine gewisse Reichweite erlangt. Überraschenderweise ist dies nicht der Fall. Mit 56 Versionen und 24.425 Downloads ist UL2 in beiden dieser Kennzahlen Tabellenletzter. Auch bei den Communitydiskussionen (38) und den Likes (651) schneidet das Modell nur minimal besser ab. Die Community um die UL2 Modelle ist sehr klein, weshalb es nur einen Punkt gibt.

MPT: Nachdem die letzten vier Modellfamilien jeweils nur kleine Communitys zu bieten hatten, kann sich MPT von diesen etwas absetzen. 299 Versionen und 261.217 Downloads im letzten Monat ist nicht viel (Pythia hatte jeweils deutlich mehr), dafür können die Modelle bei ihren Nutzern scheinbar überzeugen und konnten 3.420 Likes erreichen. Damit liegt das Modell im Hinblick auf die Likes, trotz der vergleichsweise geringen Downloads, auf Platz 4. Auch bei den Communitydiskussionen gibt es mit 293 eine solide Anzahl. Die Community ist nicht sehr groß, interagiert jedoch viel und scheint mit dem Modell zufrieden zu sein. Damit fällt das Modell insgesamt hinter den „Großen“ Modellen zurück, kann aber 2 Punkte erreichen.

LLaMA 2: Die LLaMA-Reihe stellt wohl die bekanntesten offenen LLMs dar und dies kann auch hervorragend an der Anzahl an Modellversionen gesehen werden: 5.677 verschiedene LLaMA 2 Versionen sind zu finden. Das ist mit massivem Abstand Platz 1 der Kennzahl (zum Vergleich: Platz zwei hat gerade mal 1.747). Diese starke Führung kann in den anderen Kategorien aber nicht weiter ausgebaut werden. Wie schon vorher beschrieben, können die originalen LLaMA 2 Modelle gar nicht ohne Weiteres heruntergeladen werden (auch wenn das bei LLaMA 2 Modellen, die nicht direkt von Meta kommen, anders aussieht). Dadurch ergibt es eine Downloadzahl von 2.297.281. Das ist im Vergleich mit den anderen Modellen ein ausgezeichnetes Ergebnis, welches nur hinter den Bloom Modellen zurückbleibt und Platz 2 der Kategorie erreicht. Es ist nochmal darauf hinzuweisen, dass nur die Downloads der originalen Modelle gezählt werden. Viele der Downloads verteilen sich auf andere

2. Offene Modelle

Versionen von LLaMA 2. Da die Tools von HuggingFace in dieser Hinsicht recht eingeschränkt sind, ist schwer zu sagen, wie viele Downloads es insgesamt wären (also inklusive aller 5.677 Versionen). Es wäre denkbar, dass die Downloadanzahl sich insgesamt etwa verdoppeln würde. Mit dieser Rechnung würden sich aber auch die Downloadzahlen der anderen Modelle erhöhen. Außerdem würde selbst eine Verdoppelung der Downloadzahl die Punktevergabe nicht ändern. Feedback und Diskussionen über die Modelle auf HuggingFace gibt es keine, denn LLaMA 2 ist das einzige Modell in der Liste, bei dem diese Funktion deaktiviert wurde. Damit steht bei LLaMA 2 eine 0 im Bereich Communitydiskussion. Die fehlende Kommunikation mit der Community zeigt sich auch bei den Likes. Das Ergebnis von 4.894 ist gut, führt die Kategorie aber nicht an. Insgesamt wird die LLaMA 2 Community-Bewertung primär durch die sehr hohe Anzahl an Modellversionen getragen und erreicht dadurch trotz einiger Schwächen 4 Punkte.

OpenLLaMA: Die OpenLLaMA Community ist erneut sehr klein, kann sich mit 357 Versionen und 195.235 Downloads aber etwas vor einige der zuvor genannten Modelle setzen und befindet sich damit im unteren Mittelfeld. Auch die Communitydiskussionen (39) und die Likes (785) schneiden jeweils unterdurchschnittlich ab. Insgesamt ist die Community um OpenLLaMA klein und der Abstand von der Tabellenspitze ist groß. Es gibt erneut einen Punkt.

Falcon: Die Falcon Modelle sind, zusammen mit Bloom, die Einzigen, welche an den enormen Zahlen von LLaMA 2 und Bloom annähernd herankommen. Mit 1747 Versionen erreicht Falcon den zweiten Platz, knapp vor Bloom. Dabei erreichte Falcon aber bei den Downloads, in den vergangenen 30 Tagen, nur etwa halb so viele, wie die LLaMA 2 Modelle (insgesamt 1.182.349). Auch bei den Communitydiskussionen erreicht die Modellreihe mit 324 Einträgen den zweiten Platz. Bei den Likes kann Falcon hingegen den ersten Platz erreichen und liegt mit 6.180 Likes deutlich vor Bloom (5.380). Insgesamt bieten die Falcon Modelle eine große Community, die aber nicht an LLaMA 2 und Bloom herankommt (besonders, wenn es um die Downloads angeht) und erreicht 3 Punkte.

Final ergibt sich für den Aspekt „Community“ damit folgende Tabelle:

| Modellfamilie | Berechnete Punktzahl (Max. 50) | Finale Punktezahl (Skaliert auf 1-4) |
|---------------|--------------------------------|--------------------------------------|
| LLaMA 2 | 39 | 4 |
| Bloom | 35 | 4 |
| Falcon | 29 | 3 |
| MPT | 16 | 2 |
| Pythia | 8 | 1 |
| OpenLLaMA | 6 | 1 |
| UL2 | 6 | 1 |
| StableLM | 6 | 1 |
| Cerebras-GPT | 6 | 1 |
| RWKV | 5 | 1 |

Tabelle 10: LLM-Vergleich: Community

2. Offene Modelle

2.3.7. Zusammenfassung und Entscheidung

Nachdem die vielen Einzelbewertungen abgeschlossen sind, kann eine finale Tabelle und Punktzahl erzeugt werden. Dabei wird bei der Kategorie „Leistung“ jeweils die beste erreichte Punktzahl verwendet, wenn mehrere Versionen eines Modells getestet wurden.

| Modellfamilie | Leistung | Offenheit | Lizenz | Entwickler | Community | Gesamt |
|----------------------------|----------|-----------|--------|------------|-----------|--------|
| MPT | 12,0 | 4 | 3 | 2 | 2 | 23,0 |
| OpenLLaMA | 10,2 | 4 | 4 | 3 | 1 | 22,2 |
| LLaMA 2 (Platypus2) | 11,8 | 1 | 2 | 2 | 4 | 20,8 |
| Bloom | 5,6 | 4 | 3 | 4 | 4 | 20,6 |
| Falcon | 11,4 | 2 | 2 | 2 | 3 | 20,4 |
| UL2 | 9,0 | 4 | 4 | 2 | 1 | 20,0 |
| Pythia | 6,5 | 4 | 4 | 4 | 1 | 19,5 |
| StableLM | 3,2 | 3 | 3 | 2 | 1 | 12,2 |
| Cerebras-GPT | 1,3 | 4 | 4 | 1 | 1 | 11,3 |
| RWKV | 0,1 | 3 | 4 | 3 | 1 | 11,1 |

Tabelle 11: LLM-Vergleich: Gesamtergebnis

Wie zu sehen ist, hat MPT die höchste Gesamtpunktzahl erreichen können. Besonders bei der Leistung, sowie der Offenheit und Lizenz konnte das Modell punkten, mit Schwächen in den Bereichen „Entwickler“ und „Community“. Platz zwei wird von OpenLLaMA eingenommen, mit einem geringen Abstand zu MPT. OpenLLaMA fällt, in Bezug auf die Leistung, jedoch hinter den anderen Spitzenmodellen zurück, kann aber besonders in den Kategorien Offenheit, Lizenz und Entwickler höhere Punktzahlen erreichen. Das sehr beliebte LLaMA 2 (in der Platypus 2 Variante) erreicht nur Platz 3, auch wenn die Performance hervorragend war (insbesondere auch bei den hier nicht direkt abgebildeten 13B und 7B Modellen). Dies liegt vor allem an den schlechten Punktzahlen bei „Offenheit“, „Lizenz“ und „Entwickler“. Auf der anderen Seite der Tabelle finden sich RWKV, Cerebras-GPT und StableLM, die jeweils in erster Linie miserabel im Leistungstest abgeschnitten haben.

Welches Modell soll also für die Entwicklung der Applikation verwendet werden? Prinzipiell wären alle drei der Topmodelle eine gute Wahl. Allerdings haben auch alle drei ihre Vor- und Nachteile. Die offensichtliche Antwort wäre MPT. Dabei bietet MPT ein, hier nicht genauer dargestelltes, Problem. Bei dem sehr leistungsstarken Modell handelte es sich um eine 30B Variante, welche insgesamt etwa 80GB-90GB VRAM benötigt, was in relativ hohen Hardwarekosten resultiert. OpenLLaMA gibt es hingegen in einer 13B Version, die bedeutend weniger Hardware benötigt, dafür fällt die Leistung hinter den anderen beiden Modellen zurück. Schließlich gibt es noch LLaMA 2 bzw. Platypus 2, welches primär mit der Leistung der kleineren Modelle punkten kann. Hätte die 13B Version von LLaMA 2 einen eigenen Eintrag in der Tabelle, würde diese bei der Leistung, eine Punktzahl von 11,1 erreichen und bietet damit einen hervorragenden Kompromiss aus Leistung und Hardwareanforderung. Zudem kann die große Community hilfreich sein, wenn während der Entwicklung Probleme auftreten sollten. Währenddessen sind die Hauptnachteile der LLaMA 2 Modelle, für die zu entwickelnde Applikation, weniger wichtig. Die Applikation soll für diese Arbeit als Beispiel dienen und hat keinerlei kommerzielle Ziele.

2. Offene Modelle

Um die Hardwarekosten auf einem Minimum zu halten, dabei aber nicht auf zu viel Leistung zu verzichten, trifft die Entscheidung auf den drittplatzierten LLaMA 2 bzw. Platypus 2, speziell in den kleineren 13B und 7B Varianten. Es ist jedoch wichtig, darauf hinzuweisen, dass diese Entscheidung möglicherweise anders ausfallen würde, wenn es das Ziel wäre, eine kommerzielle Software zu entwickeln und das Budget für Hardware dementsprechend größer wäre. Für jeden Einzelfall muss neu ermittelt werden, welche der dargestellten Aspekte besonders wichtig sind, welche weniger Relevanz haben und welches Modell die beste Wahl darstellt.

3. LangChain

LangChain bietet eine Vielzahl an Möglichkeiten, mit LLMs zu interagieren, von Schnittstellen zu LLMs über Prompt-Vorlagen bis zu bestimmten Tools, die das Einbinden und Verknüpfen von LLMs miteinander vereinfachen. Dabei gibt es insgesamt eine sehr große Bandbreite an Möglichkeiten, bestimmte Probleme zu lösen. Diese große Anzahl an Möglichkeiten erklärt sich durch die Vielzahl an Einsatzmöglichkeiten von LLMs. Diese sind sehr vielseitig und dementsprechend muss auch eine speziell für LLMs geschriebene Bibliothek vielseitig sein. Dies macht es aber auch schwierig, alle Bestandteile von LangChain im Detail darzustellen. Stattdessen werden die Grundlagen von LangChain geklärt, aus denen sich die meisten Applikationen im LangChain-Kosmos zusammensetzen. Es gibt sechs große Themen, aus denen sich LangChain in erster Linie zusammensetzt: 1. Die Einbindung der LLMs, 2. Die Prompts, 3. Die namensgebenden Chains, 4. Eigenständige Agents, 5. Memory für Chains und Agents und 6. „Retrieval Argumented Generation“, kurz RAG bzw. Vektor-Datenbanken. Diese Aufteilung orientiert sich grob an der Aufteilung der LangChain-Dokumentation [74]. Anhand dieser Aspekte wird LangChain in diesem Kapitel vorgestellt, die Funktionsweise erklärt, sowie zur Veranschaulichung jeweils Codebeispiele gegeben.

Insgesamt ist genau der Punkt „Vielseitigkeit“ auch der große Vorteil von LangChain. In der Theorie wäre es nicht besonders schwer, vergleichbare Software auch ohne LangChain zu bauen. Indem Funktionalität, wie die API-Calls an die LLMs oder das Verbinden von LLMs untereinander, durch eigenen Code implementiert wird. Jede einzelne Funktionalität ist oft nicht sehr komplex, aber die große Vielfalt an Tools, die LangChain mitbringt, ist der Vorteil der Bibliothek. Diese Vielfalt ist es auch, die dafür sorgt, dass es kaum direkte Konkurrenz zu dieser Bibliothek gibt. Die meisten Konkurrenzprojekte fokussieren sich auf nur einen speziellen Teil der in LangChain eingebauten Funktionalität. So ist das Projekt „Auto-GPT“ zum Beispiel primär auf die Erzeugung von LLM-Agents (eigenständig laufenden LLM-Prozessen) fokussiert, während „LlamaIndex“ vorwiegend auf den Bereich RAG ausgelegt ist. Das Projekt, welches am ehesten als Konkurrenz zu LangChain angesehen werden kann, ist „DSPy“ [75] vom „Stanford NLP“ Team. „DSPy“ arbeitet unter dem Motto „Programming – not Prompting“ und versucht, ähnlich wie LangChain, komplexe Aspekte der LLM-Interaktion zu abstrahieren. Der Fokus liegt hingegen spezifischer auf den Prompts, während LangChain einen allgemeineren und umfangreicheren Ansatz wählt.

Neben den genannten Optionen gibt es noch eine Vielzahl anderer Projekte, die jeweils bestimmte Teilaspekte von LangChain implementieren. Während LangChain vor allem damit wirbt, alle diese Aspekte in einer Bibliothek zu vereinigen. Die Tatsache, dass LangChain sehr Umfangreich ist, macht es besonders einfach, diese Aspekte miteinander zu verbinden.

Der gesamte Beispielcode aus diesem Kapitel (genauer gesagt den Unterkapiteln) ist in der Datei „examplecode.ipynb“, des zu dieser Arbeit gehörenden GitHub Repositorys, zu finden.

3. LangChain

3.1. LLM-Implementierung

Die LLMs stehen im Fokus von LangChain, deshalb ist besonders wichtig zu verstehen, wie diese mithilfe von LangChain angesprochen werden können. Das Einbinden eines LLMs ist in den meisten LangChain-Projekten der erste Schritt. Grundlegend unterstützt LangChain eine große Menge an verschiedenen Möglichkeiten, ein LLM einzubinden. Von großen und bekannten Cloud-Anbietern wie AWS, Google Cloud, Microsoft Azure und natürlich OpenAI, über Self-Hosted API Server wie den zuvor bereits erwähnten TGI [27] bis hin zur lokalen Ausführung von LLMs zum Beispiel mithilfe des Open-Source-Projekts llama.cpp [76], bietet LangChain eine Vielzahl an Optionen. Insgesamt listet die LangChain-Dokumentation 72 verschiedene Möglichkeiten auf, um ein LLM in LangChain einzubinden. Darüber hinaus gibt es noch weitere Optionen. So nutzen kleinere Projekte häufig eine OpenAI nachempfundene API. LangChains OpenAI API Implementation kann für diesen Fall auch mit einer anderen IP-Adresse konfiguriert werden. Damit können dann auch jegliche mit der OpenAI API kompatiblen Projekte in LangChain eingebunden werden. Diese Möglichkeit wurde beim LLM-Leistungsvergleich zum Beispiel genutzt, um RWKV zu testen, welches nicht mit TGI kompatibel ist, aber einen eigenen API-Server bietet, der OpenAIs ChatGPT API implementiert. Zusätzlich ist es auch möglich einen Custom-LLM-Wrapper zu schreiben, um LLMs zu implementieren, die mit keiner der nativ unterstützten Möglichkeiten kompatibel ist.

Wie die Einbindung eines LLMs jeweils konkret aussieht, unterscheidet sich dementsprechend je nach gewünschter LLM-Implementation. Hier werden beispielhaft die beiden für den LLM-Vergleich genutzten Implementationen für TGI und OpenAIChat (RWKV) dargestellt. Diese sind jeweils sehr einfach gehalten (so wie die meisten der angebotenen Möglichkeiten).

Um ein mit dem Text-Generation-Inference-Server (TGI) gehostetes Modell einzubinden, benötigt es nur eine Codezeile:

```
from langchain.llms import HuggingFaceTextGenInference, OpenAIChat

# Create a basic langchain LLM object from TGI
tgi_llm = HuggingFaceTextGenInference(
    inference_server_url="http://localhost:8080")

# Create a basic langchain LLM object from OpenAIChat compatible API
openai_compatible_llm = OpenAIChat(openai_api_base = "http://localhost:8080",
    openai_api_key="na", model="rwkv")
```

Abbildung 30: Codebeispiel für die Erstellung eines LLM-Objekts

Auf diese Weise kann sehr einfach eine Verbindung zu einem LLM hergestellt werden. LangChain kümmert sich dann um die API-Calls. Dabei werden hier, für die verschiedenen Hyper-Parameter, die in LangChain definierten Standardwerte verwendet. Oft müssen diese aber auch angepasst werden, um das Modell besser auf den eigenen Fall abzustimmen. Dies kann durch Hinzufügen von weiteren Übergabeparametern erreicht werden.

3. LangChain

```
tgi_llm_with_hyperparameters = HuggingFaceTextGenInference(  
    inference_server_url="http://localhost:8080",  
    max_new_tokens=100,  
    temperature=0.1,  
    repetition_penalty=1.1,  
)
```

Abbildung 31: Codebeispiel für die Erstellung eines LLM-Objekts mit bestimmten Hyper-Parametern

In dem Fall wurden zusätzlich 3 Hyper-Parameter angegeben. Im gleichen Stil können auch alle weiteren verfügbaren Hyper-Parameter definiert werden. Welche Optionen dabei konkret zur Verfügung stehen, unterscheidet sich allerdings je nach genutzter Implementation.

Um zu testen, ob das LLM-Objekt funktioniert, kann das Modell mit „predict“ ausprobiert werden.

```
# Make call to model  
tgi_llm_with_hyperparameters.predict("How are you?")  
# >> 'I hope you're well.\n'  
  
# Overwrite hyperparameters  
tgi_llm_with_hyperparameters.predict("How are you?", max_new_tokens = 2)  
# >> 'I hope'
```

Nützlich ist dabei auch, dass an dieser Stelle die zuvor gesetzten Hyper-Parameter überschrieben oder neue hinzugefügt werden können. Das kann besonders beim Begrenzen der Output-Länge nützlich sein, wenn es bestimmte Situationen gibt, in denen zum Beispiel nur kürzerer Output erlaubt sein soll.

3.2. Prompt

Nachdem dargestellt wurde, wie ein LLM mit LangChain verbunden werden kann und ein erster Call an das Modell durchgeführt wurde, gilt es LangChains Prompts zu untersuchen. Zwar ist es möglich der „predict“ Methode, wie zuvor gezeigt, manuell einen String zu übergeben, aber doch ist dies nicht der Weg, in welchem die LLM-Calls normalerweise in LangChain durchgeführt werden. Stattdessen werden dazu Prompt-Templates verwendet. Speziell wird die „PromptTemplate“ Klasse verwendet, um Prompt-Objekte zu bauen.

```
from langchain.prompts import PromptTemplate  
  
# Template  
summarize_template = "Summarize the following Article in about {n_of_words}  
words: {input}"  
  
# Create prompt formatter  
pt = PromptTemplate(  
    template=summarize_template,  
    input_variables=["n_of_words", "input"],  
    output_parser=my_custom_parser, # Optional  
)
```

Abbildung 32: Codebeispiel für ein allgemeinen Prompt

Dabei wird die „PromptTemplate“ Klasse immer mit einem „template“ String und einer „input_variables“ Liste erzeugt. Optional kann auch noch ein eigener „output_parser“ übergeben werden. Das kann unter anderem nützlich sein, wenn der Output des LLMs einem

3. LangChain

bestimmten Format folgen soll und die Ausgabe deshalb, vor der Übergabe an andere Objekte im Code, verarbeitet werden muss.

Wie zu sehen ist, beinhaltet das Template jeweils den Prompt-Text und verwendet dabei Variablen, die später ausgetauscht werden, in diesem Fall „n_of_words“ und „input“. Dabei ist „input“ der Standard-Key für die Nutzereingabe, welcher allerdings auch gegen einen anderen Key ersetzt werden kann. Die Variablennamen werden zusätzlich noch als Liste von „input_variables“ übergeben. Auf diesem Weg können beliebige Prompts erzeugt werden. Zusätzlich zu dieser „PromptTemplate“ Klasse gibt es noch spezifischere Prompt-Klassen für verschiedene Fälle. So etwa die „FewShotPromptTemplate“ Klasse, welche genutzt werden kann, um Few-Shot-Prompts zu erstellen.

```
from langchain.prompts import PromptTemplate, FewShotPromptTemplate

# Example prompt template
few_shot_template = "Question: {input}\nAnswer: {output}"
# Examples
few_shot_examples = [
    {"input": "3+3", "output": "6"},
    {"input": "5+4", "output": "9"},
]
# Create example-prompt
examples_prompt = PromptTemplate(
    template=few_shot_template,
    input_variables=["input", "output"],
)
# Create few-shot-prompt from examples and example-prompt
fspt = FewShotPromptTemplate(
    examples=few_shot_examples,
    example_prompt=examples_prompt,
    suffix="Question: {input}\nAnswer: ",
    input_variables=["input"]
)
```

Abbildung 33: Codebeispiel für ein Few-Shot-Prompt

Wie zu sehen ist, wird dabei aus einem Prompt-Template und einer Liste an Beispielen zuerst mithilfe der „PromptTemplate“ Klasse das Format der Beispiele definiert. Dann kann die „FewShotPromptTemplate“ Klasse verwendet werden, um die verschiedenen Einzelteile zu einem Prompt für ein LLM zusammenzusetzen. Das funktioniert ähnlich wie schon zuvor beim einfachen Prompt, nur dass noch die Beispiele als „examples“ sowie ein „suffix“ übergeben werden.

In diesem Stil gibt es einige Prompt-Template-Klassen für Spezialfälle. Zusätzlich ist es auch möglich, eigene Klassen für diesen Zweck zu erstellen. Allerdings können auch alle Fälle über die Standard-„PromptTemplate“-Klasse umgesetzt werden, indem die Few-Shot-Beispiele als „input_variable“ eingefügt werden.

Zusätzlich zu den Prompt-Template-Klassen gibt es außerdem noch eine Vielzahl an tatsächlichen Templates, die jeweils als Vorlage für spezifische Aufgaben dienen können. Ein Beispiel dafür wird im folgenden Kapitel „3.3. Chains“ aufgezeigt. Zudem können Prompts auch mithilfe von verschiedenen Methoden der jeweiligen Klassen erzeugt zu werden. Die „PromptTemplate“ Klasse bietet dafür beispielsweise die Methoden „from_file“, „from_orc“, „from_examples“ und „from_template“. Dabei ist die oben gezeigte Methodik insgesamt aber anpassbarer und in den meisten Fällen (wenn auch nicht in allen) die bessere und vor allem auch besser lesbare Wahl.

3. LangChain

Dabei gibt es noch eine weitere wichtige Methode der „PromptTemplate“ Klasse: Die „partial“ Methode erlaubt es nach Definition eines Prompts spezielle Input-Variablen einzusetzen. Das kann sehr hilfreich sein, wenn die Input-Variablen an unterschiedlichen Stellen im Code erzeugt werden. Dies kann wie folgt aussehen:

```
partial_pt = pt.partial(n_of_words="30")
```

Abbildung 34: Codebeispiel für die „partial“ Methode

3.3. Chains

Die vorgestellten Prompt-Templates sind eigenständig bislang nicht besonders nützlich, diese müssen zum Verwenden an ein LLM geschickt werden. Die einfachste Möglichkeit dafür stellen die Chains dar. Diese erlauben es, LLM-Calls auszuführen und vor allem auch mehrere LLM-Calls miteinander zu verknüpfen. Das grundlegende Konzept ergibt sich bei Betrachtung eines einfachen Beispiels. Soll der einfache „fspt“ Few-Shot-Prompt aus „Abbildung 33: Codebeispiel für ein Few-Shot-Prompt“ angewendet werden, kann dafür die Basis-Klasse „LLMChain“ verwendet werden. Dieser Klasse wird ein LLM-Objekt (zum Beispiel „tgi_llm_with_hyperparameters“ aus Abbildung 31) und ein Prompt (zum Beispiel „fspt“) übergeben. Anschließend kann die dadurch definierte Chain mit der „run“-Methode ausgeführt werden.

```
from langchain.chains import LLMChain

# Create basic LLM chain from fspt prompt
fspt_chain = LLMChain(llm=tgi_llm_with_hyperparameters, prompt=fspt)
# Run the fspt chain
fspt_output = fspt_chain.run("10+15")
print(fspt_output)
# >> 25
```

Abbildung 35: Codebeispiel für eine einfache LLMChain mit dem „fspt“ Prompt

In diesem Fall stellt sich die Anwendung als besonders einfach heraus, da es im „fspt“ Prompt nur eine einzige Eingabevariable gibt und diese den Standard-Key „input“ verwendet. Dadurch muss der „run“-Methode ausschließlich der Eingabe-String übergeben werden. Bei etwas komplexeren Prompts wie dem „pt“ Prompt aus „Abbildung 32: Codebeispiel für einen allgemeinen Prompt“ können die verschiedenen Eingabeparameter einzeln als Dict angegeben werden. Zudem ermöglicht die „run“-Methode auch noch die Übergabe einer Reihe weiterer Parameter. So etwa die Anpassung der LLM-Hyper-Parameter mithilfe des „llm_kwargs“ Übergabeparameter (ähnlich wie auch schon bei der „predict“ Methode zuvor) oder das „verbose“-Flag, welches in diesem Fall hauptsächlich den vollständigen Prompt (inklusive aller eingesetzten Bestandteile) auf die Konsole schreibt und so beim Debugging helfen kann. Zudem kann wieder ein „OutputParser“ festgelegt werden. Standardmäßig wird LangChains „StrOutputParser“ verwendet, welcher den vollständigen Ausgabe-String des LLMs zurückgibt. Eine Liste aller Optionen findet sich in der LangChain-Dokumentation [77].

3. LangChain

Folgend ein Beispiel, welches den „pt“ Prompt ausführt und dabei zusätzliche Übergabeparameter verwendet:

```
from langchain.chains import LLMChain
# Create basic LLM chain from pt
pt_chain = LLMChain(llm=tgi_llm_with_hyperparameters, prompt=pt,
                    llm_kwargs={"max_new_tokens": 100}, verbose=True)
# Run the pt chain
pt_output = pt_chain.run({"n_of_words": 50, "input": "This is an Article ..."})
# >> > Entering new LLMChain chain...
# >> Prompt after formatting:
# >> Summarize the following Article in about 50 words: This is an
#       Article ...
# >> > Finished chain.
print(pt_output)
# >> This is an Article ...
```

Abbildung 36: Codebeispiel für eine komplexere LLMChain mit dem „pt“ Prompt

Im „tgi_llm_with_hyperparameters“ LLM-Objekt wurde „max_new_tokens“ auf 10 gesetzt, hier wird diese Variable für diese spezielle Chain mit dem Wert „100“ überschrieben und das „verbose“-Flag auf „True“ gesetzt. In dem Kommentar unter der „run“-Methode ist zu sehen, wie der durch „verbose“ ausgelöste Output aussieht. Außerdem ist bei der „run“-Methode zu sehen, dass ein Dict anstatt eines einfachen Strings übergeben wird, in dem die verschiedenen Variablen gesetzt werden. Auf diese Weise können komplexe Chains erzeugt werden, welche einzelne Anfragen an ein LLM schicken. Die Stärke von Chains liegt allerdings darin, diese miteinander zu verknüpfen.

Für diesen Zweck gibt es bestimmte Chains, welche zusätzliche Funktionalität bieten. Die grundlegenden Chain-Typen sind die LLM-Chain (wurde schon dargestellt), die Sequential-Chain, die Transformation-Chain und die Router-Chain.

Dabei stellt die **Sequential-Chain** die einfachste Art dar, um Chains zu verknüpfen. Bei einer Sequential-Chain werden mehrere Chains nacheinander ausgeführt und dabei jeweils der Output einer Chain als Input für die nächste genutzt. Dafür wird nur wenig Code benötigt:

```
from langchain.chains import SimpleSequentialChain

# Define the Chains
chain_1 = LLMChain(llm=tgi_llm, prompt=PromptTemplate(input_variables=["input"],
                                                       template="Write a Text about '{input}'."))
chain_2 = LLMChain(llm=tgi_llm, prompt=PromptTemplate(input_variables=["input"],
                                                       template="Give the following Text a rating from 1-5: '{input}'\nRating: '"))
# Create the Sequential-Chain
combined_chain = SimpleSequentialChain(chains=[chain_1, chain_2])
# Run the Chain.
combined_chain.run("The Meaning of Life")
# >> '4'
```

Abbildung 37: Codebeispiel für die Nutzung einer Sequential-Chain

In dem Beispiel werden zuerst zwei einfache Chains erzeugt. Die Erste schreibt einen Text über ein bestimmtes Thema und die Zweite bewertet einen Text mit einer Zahl zwischen 1 und 5. Anschließend werden beide Chains miteinander verbunden, sodass der von Chain 1 erzeugte Text von Chain 2 bewertet wird. Schließlich wird die kombinierte Chain ausgeführt.

3. LangChain

Es ist zu sehen, dass die Ausgabe dieser kombinierten Chain der Ausgabe der letzten Teil-Chain entspricht. In diesem Fall ist das „4“. Mit dem „verbose“-Flag können zum Debuggen auch die Ausgaben aller Teil-Chains auf die Konsole geschrieben werden.

Anders als bei der Sequential-Chain ist es nicht das Ziel einer **Transform-Chain** mehrere Chains zu verbinden, vielmehr kann die Transform-Chain genutzt werden, um Funktionen in eine Chain umzuwandeln, um diese beispielsweise als Teil einer Sequential-Chain auszuführen. Dazu wird in einer Transform-Chain eine spezielle Funktion angegeben, welche ausgeführt werden soll, wenn die Chain aufgerufen wird. Das Konzept ist nicht besonders komplex und kann beispielsweise so aussehen:

```
from langchain.chains import TransformChain

# Define transform function
def count_characters_in_string(input: dict) -> dict:
    text = input["text"]
    return {"output_text": len(text)}

# Define transform chain
count_chain = TransformChain(input_variables=["text"],
                             output_variables=["output_text"], transform=count_characters_in_string)
# Create combined chain
combined_chain = SimpleSequentialChain(chains=[chain_1, count_chain])
# Run combined chain
combined_chain.run("The Meaning of Life")
# >> 2287
```

Abbildung 38: Codebeispiel für eine Transform-Chain

Zuerst wird eine Funktion definiert, welche die Anzahl an Charakteren in einem gegebenen String zählt und die Anzahl dann ausgibt. Die Eingabe- und Ausgabeparameter sind jeweils in Dicts verpackt, da LangChains Datenstrukturen innerhalb einer Chain dies so vorsehen. Anschließend wird die Transformation-Chain definiert und dabei die zuvor definierte Funktion angegeben. Schließlich wird wieder eine „SimpleSequentialChain“ erzeugt, welche „chain_1“ aus dem vorherigen Beispiel in Kombination mit der neu definierten Transformation-Chain verbindet. Wird diese kombinierte Chain mit „run“ ausgeführt, ist das Ergebnis die Anzahl an Charakteren im von „chain_1“ erzeugten Text. Im Beispiel waren es „2287“. Durch das Kombinieren von Transform-Chains mit anderen Chains können komplexere Aufgaben abgebildet werden.

Die **Router-Chain** ist die letzte der 4 Grund-Chain-Arten, diese ist am komplexesten und hat ähnlich wie die Sequential-Chain grundlegend zum Ziel eine LLM-Anfrage durchzuführen, die dann wiederum eine andere LLM-Anfrage startet. Anders als bei der Sequential-Chain, stehen der Router-Chain allerdings mehrere Chains zur Auswahl. Anhand der Eingabe wählt die Router-Chain immer eine von mehreren Chains aus, die anschließend ausgeführt werden soll. Werden wieder die „pt“- und „fspt“-Chains als Beispiele verwendet, soll sich das System dann eigenständig entscheiden, welche der beiden Chains im Anschluss an die Router-Chain ausgeführt werden muss. Diese Router-Chain ist vorrangig in der Klasse „LLMRouterChain“ zu finden und kann mit der „from_llm“ Methode definiert werden. Dabei müssen allerdings verschiedene Aspekte vorbereitet werden, um den Prompt für die „LLMRouterChain“ zusammenzusetzen. Erstens ein Template, welches Informationen zu den Ziel-Chains beinhaltet und dem Modell vermittelt, wann welche Ziel-Chain zu verwenden ist und zweitens einen Output-Parser, der auf das Template angepasst ist. Die wird vereinfacht, da LangChain bereits ein Template und Output-Parser mit sich bringt, welcher für einfache

3. LangChain

Situationen genutzt werden kann. Wie dies funktioniert, wird im folgenden Codebeispiel gezeigt:

```
from langchain.chains.router.llm_router import LLMRouterChain, RouterOutputParser
from langchain.chains.router.multi_prompt_prompt import
MULTI_PROMPT_ROUTER_TEMPLATE

# Define the properties of the chains
destination_chains = {"summarize": pt_chain, "math": partial_fspt_chain}
destination_chains_descriptions = {
    "summarize": "Usefull for summrazing a Text",
    "math": "Usefull for solving math equations"}

# Create a singular string that combines name and description of the
# destination chains
destination_chains_info = ""
for chain_name in destination_chains:
    chain_description = destination_chains_descriptions[chain_name]
    destination_chains_info+=chain_name+": "+chain_description+"\n"

# Combine the MULTI_PROMPT_ROUTER_TEMPLATE with destination_chains_info
router_templalte = MULTI_PROMPT_ROUTER_TEMPLATE.format(
    destinations=destination_chains_info)

# Create the Prompt template with the created template and the
# corresponding output parser
router_pt = PromptTemplate(template=router_templalte,
                           input_variables=["input"],
                           output_parser=RouterOutputParser())
# Create the LLMRouterChain
router_chain = LLMRouterChain.from_llm(llm=tgi_llm, prompt=router_pt)

# Run the chain using "invoke" method instead of "run" method
router_chain.invoke("1+1=")
```

Abbildung 39: Codebeispiel für eine Router-Chain

Das Beispiel veranschaulicht das zuvor beschriebene Konzept. In der Variable „destination_chains_descriptions“ werden zuerst die verschiedene Ziel-Chains beschrieben. Dabei wird für dieses Beispiel das „MULTI_PROMPT_ROUTER_TEMPLATE“ und der dazugehörige „RouterOutputParser“ verwendet. Hierbei ist wichtig zu wissen, dass die Ziel-Chains jeweils nur eine Eingabe-Variable haben dürfen (wenn das hier gezeigte Template verwendet wird). Deshalb wird in dem Beispiel nicht die in Abbildung 36 gezeigte „pt_chain“ verwendet (diese hatte 2 Input-Variablen), sondern eine neue „partial_pt_chain“, die aus dem „partial_pt“ Prompt aus Abbildung 34 erzeugt wurde, bei der die Variable „n_of_words“ bereits ausgefüllt wurde.

Besonders wichtig ist es zu verstehen, dass hier noch keine Chains miteinander verbunden wurden. Wird der vorherige Code ausgeführt, ergibt sich folgende Ausgabe:

```
{'input': '1+1=', 'destination': 'math', 'next_inputs': {'input': '1+1='}}
```

Abbildung 40: Ausgabe des in Abbildung 39 dargestellten Codes

Wie zu sehen ist, generiert die „LLMRouterChain“ für jede Eingabe jeweils eine „destination“ (das ist der Name der Ziel-Chain) und einen String „next_inputs“ (die Eingabe für die Ziel-Chain). Damit könnte nun manuell die nächste Chain aufgerufen werden. Besser ist es aber, auch an dieser Stelle wieder mehrere Chains direkt miteinander zu verknüpfen. Dafür kann in diesem Fall die „MultiPromptChain“ verwendet werden, welche eine „router_chain“, eine

3. LangChain

Liste an „destination_chains“ sowie eine „default_chain“ übergeben bekommt und automatisch anhand der Entscheidung der „router_chain“ die passende Ziel-Chain ausführt. Die „router_chain“ und die Liste „destinations_chains“ wurden bereits im Code in Abbildung 39 definiert. Einzig eine „default_chain“, die ausgeführt wird, wenn keine der anderen Optionen zu der Eingabe passt, wurde bislang nicht definiert. Diese „default_chain“ sollte möglichst allgemein gehalten werden, es kann zum Beispiel LangChains „ConversationChain“ Vorlage verwendet werden:

```
from langchain.chains import ConversationChain

default_chain = ConversationChain(llm=tgi_llm, output_key="text")
```

Abbildung 41: Erzeugen der default_chain

In diesem Fall wird noch zusätzlich der „output_key“ auf „text“ gesetzt, da die „ConversationChain“ standardmäßig mit „response“ antwortet, die „MultiPromptChain“ allerdings einen „text“ Key erwartet. Damit kann die finale „MultiPromptChain“ erzeugt werden, welche schließlich tatsächlich die volle Router-Funktionalität umsetzt.

```
multi_prompt_chain = MultiPromptChain(
    router_chain = router_chain,
    destination_chains = destination_chains,
    default_chain = default_chain,
)
# Run the Chain
multi_prompt_chain.run("1+1")
# >> '2'
```

Abbildung 42: Beispielcode für das Erzeugen des finalen Routers

Die „MultiPromptChain“ entscheidet automatisch, dass die „partial_pt“ Chain genutzt werden sollte und löst die Gleichung mithilfe dieser. Durch die Verwendung solcher Router ist es möglich komplexe Aufgaben in mehrere kleinere Teilaufgaben herunterzubrechen.

Dabei wird dem LLM, mit der Implementierung eines solchen Routers, gleichzeitig auch mehr Entscheidungsmacht eingeräumt wird. Dies bedeutet, dass die Wahrscheinlichkeit steigt, dass eines der Verknüpften LLMs im Verlauf der Chain einen Fehler macht und die Endausgabe falsch ist. Das kann ein großes Problem darstellen. So stellt der gezeigte Beispielcode zwar die grundlegende Funktionsweise eines Routers in LangChain dar, funktioniert bei der Ausführung aber nur selten korrekt, da die LLMs aufgrund des sehr allgemein gehaltenen Prompt-Templates häufig die falschen Ziel-Chains auswählen. In den Leistungs-Tests wurde bereits gezeigt, dass ein guter Prompt für die LLMs entscheidend ist. Deshalb ist es in den meisten Fällen empfehlenswert, anstatt des sehr allgemein gehaltenen „MULTI_PROMPT_ROUTER_TEMPLATE“, ein eigenes Router-Template zu entwickeln und gegebenenfalls auch einen dazu passenden eigenen Output-Parser.

Damit wurden die 4 grundlegenden Chain-Arten und deren Funktionsweise in LangChain dargestellt. Neben diesen gibt es noch eine Vielzahl an Chains für speziellere Aufgaben. So zum Beispiel die „ConversationChain“ für Konversationen, welche schon im Router-Chain Beispiel verwendet wurde, oder mehrere Chain-Arten, die speziell für das Laden von Dateien geeignet sind. Die meisten Aufgaben können mithilfe der 4 vorgestellten Grund-Arten umgesetzt werden. Diese stellen insgesamt den wichtigsten Bestandteil von LangChain dar, um den herum alle anderen Bestandteile angeordnet sind.

3. LangChain

3.4. Agents

Das Kernkonzept von Agents besteht darin, genau wie bei Chains, ein LLM zu verwenden, um eine Reihe an Aktionen durchzuführen. Anders als bei Chains, bei denen die Abfolge dieser Aktionen fest im Code definiert ist, entscheiden Agents eigenständig, welche Aktionen benötigt werden und führen diese dann auch eigenständig aus. Dabei verfügen diese auch über die Möglichkeit, wie bei einer Sequential-Chain, mehrere Aktionen nacheinander auszuführen. Damit funktionieren Agents ähnlich wie eine Router-Chain, aber mit mehr Autonomie. Dabei stellen „Tools“ einen der wichtigsten Teilaspekte von Agents dar. Grundlegend können Tools in etwa mit den Ziel-Chains einer Router-Chain verglichen werden. Genau wie die Ziel-Chains stellen Tools die verschiedenen Optionen dar, die dem Agent zur Verfügung stehen. Allerdings müssen Tools keine Chains sein. Ein Tool kann eine Funktion, eine Chain oder ein anderer Agent sein. Zudem können mehrere Tools zu einem Toolkit zusammengefasst werden. LangChain bringt bereits eine große Menge an Tools mit, die zum Beispiel eine Google-Suche oder das Öffnen und Schreiben von Dateien ermöglichen. Zudem kann Funktionalität, welche nicht von den Standardtools unterstützt wird, eingebaut werden, indem eigene Funktionen, Chains oder Agents definiert und als Tool festgelegt werden. Das Definieren von Tools fällt dabei einfach aus:

```
from langchain.agents import AgentType, load_tools, Tool, initialize_agent

# Load the some of the default tools provided by langchian
tools = []
tools += load_tools(["wikipedia", "llm-math"], llm=tgi_llm)

# Define a custom function to use as a tool
def count_characters_in_string_tool(input):
    return(len(input))

# Define a list of custom tools to be used by the agent
custom_tools = [
    Tool(name="Text Counter",
        func=count_characters_in_string_tool, # use a function as a tool
        description="useful for counting how many characters a string is long"),
    Tool(name="Text Summarizer",
        func=partial_pt_chain.run, # use a chain as a tool
        description="useful for summarizing a text")]

# Add the custom tools to the list of tools to use in the agent
tools += custom_tools
```

Abbildung 43: Beispielcode für das Definieren von Tools für einen Agent

Wie zu sehen ist, können die Standardtools von LangChain mithilfe der „load_tools“ Funktion geladen werden. Im Beispiel wird das „wikipedia“-Tool, welches Wikipedia durchsuchen kann, und das „llm-math“-Tool, welches ein LLM verwendet, um Mathematik-Aufgaben zu lösen, geladen. Zudem ist zu erkennen, wie eigene Funktionen oder Chains in Tools umgewandelt werden können, indem die „Tool“-Klasse verwendet wird, um ein neues Tool zu erzeugen. Damit das LLM versteht, wann ein bestimmtes Tool verwendet werden sollte, werden jeweils ein Name und eine Beschreibung für jedes Tool angegeben. In diesem Fall werden so zwei eigene Tools festgelegt. Erstens, „Text Counter“ welcher die „count_characters_in_string_tool“ Funktion verwendet und zweitens „Text Summarizer“ bei dem eine der vorher definierten Chains verwendet wird.

3. LangChain

Sind die Tools definiert, ist es einfach einen Agent mithilfe einer der vielen Agent-Vorlagen zu erstellen. Dabei sind die „AgentExecutor“-Klasse und die „Agent“-Klasse besonders relevant. Die „Agent“-Klasse definiert den Agent und damit unter anderem die verwendeten Prompts und den Output-Parser. Diese Klasse ist in etwa mit der „PromptTemplate“ Klasse bei den Chains vergleichbar. Die „AgentExecutor“-Klasse ist anschließend für das Ausführen eines Agents verantwortlich und ist damit in etwa mit der „LLMChain“-Klasse zu vergleichen. Der Aufbau des Codes für die Definition und Ausführung eines Agents ist dementsprechend ähnlich wie zuvor bei den Chains:

```
# Create a AgentExecutor object from a default agent class
agent = initialize_agent(
    tools, tgi_llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
# Run the agent
agent.run("how many characters does this text have: This is a Text")
# >> '14'
agent.run("what is 10+15?")
# >> '25'
```

Abbildung 44: Codebeispiel für das Definieren und Ausführen eines Agents

In diesem Fall wird die vordefinierte „ZERO_SHOT_REACT_DESCRIPTION“ Vorlage verwendet. Diese Vorlage verweist intern auf eine vordefinierte Agent-Klasse, welche bereits alle relevanten Aspekte, wie die Prompts und den Output-Parser beinhaltet. Deshalb muss die „Agent“-Klasse nicht manuell instanziiert werden, dies übernimmt die „initialize_agent“ Funktion. Der Agent kann schließlich, genau wie eine Chain, mit der „run“-Methode ausgeführt werden. Insgesamt ist das Definieren eines einfachen Agents schneller als das Definieren einer Chain, da mit entsprechenden Vorlagen gearbeitet werden kann. Es ist aber natürlich auch möglich, einen Agent und AgentExecutor selber zu definieren. Das kann Sinn ergeben, wenn der Agent angepasst werden oder eine vollkommen eigene Agent-Klasse verwendet werden soll.

```
from langchain.agents import AgentExecutor
from langchain.agents.mrkl.base import ZeroShotAgent

# Define the agent class
agent = ZeroShotAgent.from_llm_and_tools(llm=tgi_llm, tools=tools)
# Define AgentExecutor to use the agent
ae = AgentExecutor(agent=agent, tools=tools, verbose=True)
# Run the agent
ae.run("what is 10+15?")
# >> '25'
```

Abbildung 45: Codebeispiel für das manuelle erzeugen eines Agents und AgentExecutors

Im Beispiel wird zur Vereinfachung wieder eine der vorgefertigten Agent-Klassen verwendet. Dabei ist die „ZeroShotAgent“ Klasse genau jene Klasse, auf welche „AgentType.ZERO_SHOT_REACT_DESCRIPTION“ aus dem vorherigen Beispiel verweist. Nur das diese manuell erzeugt wird. Dadurch ist es möglich, durch die Übergabeparameter der „from_llm_and_tools“-Methode, den Agent anzupassen und etwa das Prompt zu verändern. Schließlich wird manuell die „AgentExecutor“-Klasse instanziiert und dieser der Agent sowie die Tools übergeben. Diese Vorgangsweise ist etwas aufwendiger als das Nutzen der „initialize_agent“-Funktion, bietet dafür aber mehr Anpassungsmöglichkeiten.

Wird, wie in beiden Beispielen, „verbose“ auf „True“ gesetzt, ist es leicht nachzuverfolgen, wie die Agents intern Entscheidungen treffen. Im Folgenden ist der Konsolen-Output für die

3. LangChain

Eingabe „how many characters does this text have: This is a Text“ zu sehen, die durch das „verbose“-Flag erzeugt wird.

```
> Entering new AgentExecutor chain...
I should always think about what to do
Action: Character Counter
Action Input: This is a Text
Observation: 14
Thought:I now know that the final answer is 14
Final Answer: 14
```

Abbildung 46: Beispielhafter Konsolen-Output eines Agents mit "verbose = True"

Die hohe Autonomie von Agents ist insgesamt gleichzeitig ihr größter Vor- und Nachteil. Auf der einen Seite sind Agenten dadurch sehr mächtig in ihrer Funktionalität. Auf der anderen Seite kann die Autonomie auch schnell zu Fehlern führen, die bei Chains aufgrund ihrer starren Struktur unwahrscheinlicher sind. Welche der beiden Ansätze mehr Sinn ergibt, ist immer situationsabhängig.

3.5. Memory

Im Vergleich mit den vorhergehenden, teilweise recht komplexen, Konzepten, ist das Memory-Prinzip weniger aufwendig. Memory stellt die Möglichkeit dar, einer Chain oder einem Agent ein „Gedächtnis“ zu geben. Gemeint ist die Möglichkeit eines LLMs, sich an frühere Ein- und Ausgaben zu erinnern und sich auf diese zu beziehen. Auf Prompt-Ebene ist dies einfach implementiert. Dazu müssen die letzten Eingaben und Ausgaben des LLMs in den Prompt eingefügt werden, sodass diese bei der nächsten LLM-Eingabe Teil des Prompts sind. Im Leistungs-Test wurde eine einfache Prompt-Vorlage vorgestellt, wird diese entsprechend angepasst, sieht das wie folgt aus:

```
### Instruction:
[Beschreibung]
[Memory]
[Nutzereingabe]
### Response:
```

Abbildung 47: Beispielhafte Prompt-Vorlage für ein Prompt mit Memory

Dabei soll die Eingabe/Ausgabe-Historie allerdings nicht manuell in den Prompt eingefügt werden. LangChain bietet eigene Memory-Klassen, welche das Speichern und Einfügen in den Prompt vereinfachen. Auch hier gibt es wieder eine große Auswahl an spezifischen Klassen, die jeweils für spezielle Fälle geeignet sind. Eine der einfachsten und allgemeinsten Memory-Klassen ist die „ConversationBufferWindowMemory“-Klasse. Im folgenden Codebeispiel ist zu sehen, wie eine einfache Chain um das Memory-Konzept erweitert werden kann.

3. LangChain

```
from langchain.memory.buffer_window import ConversationBufferWindowMemory

conversation_memory = ConversationBufferWindowMemory(
    memory_key="history",
    k=5, # Length of the memory buffer
    human_prefix="Human",
    ai_prefix="AI")

cc = ConversationChain(llm=tgi_llm, memory=conversation_memory,
    llm_kwargs={"stop_sequences": ["Human:"]})

cc.run("Please remember the following things: Bread, Pizza, Butter.\n I need to
buy them later.")
# >> "Of course! I have a wonderful memory, and I remember all of these things. I
# >> know you want to buy them later, but I don't know when that will be.\n"
cc.run("What did i need to buy?")
# >> 'You needed to buy bread, pizza, and butter.\n'
```

Abbildung 48: Beispielcode für das Erzeugen einer Chain mit Memory

Wie zu sehen ist, wird zuerst ein neues Memory-Objekt instanziiert und dieses dann der Chain übergeben. Die Historie wird dann, genau wie die anderen Variablen, automatisch in den Prompt eingefügt. Wie die Ausführung der „run“-Methode veranschaulicht, erinnert sich das Modell daran, was zuvor gesagt wurde. Soll anstatt einer Chain ein Agent mit Memory ausgestattet werden, funktioniert das genauso. Das Memory-Objekt muss lediglich an die „AgentExecutor“-Klasse übergeben werden. Wichtig ist allerdings, sowohl bei Chains als auch bei Agents, dass die verwendeten Prompts jeweils dafür ausgelegt sein müssen, Memory zu verwenden, indem eine Variable mit dem unter „memory_key“ definierten Namen im Prompt vorkommt.

Die Länge des Memory-Buffers ist dabei theoretisch unbegrenzt, wird in der Praxis allerdings durch die Begrenzung der Prompt-Länge eingeschränkt. Zudem kann es in vielen Situationen auch Sinn ergeben, die Memory-Länge noch zusätzlich einzuschränken, um das LLM nicht mit unnötigen Informationen zu verwirren. In diesem Fall wurde die Memory-Länge auf die 5 letzten Ein- und Ausgaben beschränkt.

3.6. Retrieval Argumented Generation - Vektor-Datenbanken

LLMs haben ein grundsätzliches Problem, wenn es um das Bearbeiten von Aufgaben geht, welche spezifisches Wissen voraussetzt: Sie verfügen nur über die Informationen, die sie beim Training gelernt haben. Da sind oft aber nicht die jeweils benötigten Informationen dabei. Retrieval Argumented Generation, kurz RAG, versucht dieses Problem zu lösen. Bei diesem Konzept werden die jeweils benötigten Informationen im Prompt an das LLM übergeben. Dabei gibt es jedoch ein Problem, aufgrund der begrenzten Kontextlänge von LLMs können nur begrenzte Mengen an zusätzlichen Informationen in den Prompt eingefügt werden. Das bedeutet, es ist wichtig, dass immer wirklich nur die relevanten Informationen übergeben werden. Angenommen ein LLM soll auf Basis einer internen Dokumentation Fragen beantworten, dann müssen zu jeder Frage genau die Elemente der Dokumentation in den Prompt eingefügt werden, die sich mit dieser Frage beschäftigen. An der Stelle kommen Vektor-Datenbanken ins Spiel. Eine Vektor-Datenbank speichert eine Reihe an Texten (die „Dokumente“ genannt werden) in Verbindung mit einer Vektor-Repräsentation der Texte. Diese „Dokumente“ sollten jedoch nicht mit ganzen Textdokumenten verwechselt werden. Große Textdokumente müssen zuerst in viele kleinere Texte unterteilt werden. Die Vektoren werden dann von einem eigenen Embedding-Modell erzeugt. Wird eine Frage gestellt, kann

3. LangChain

diese Frage dann ebenfalls in einen Vektor umgewandelt und dieser mit allen Vektoren in der Vektor-Datenbank verglichen werden. So wird ermittelt, welche der Texte in der Datenbank für die Frage relevant sein könnten. Anschließend können genau die relevantesten Texte in den Prompt als Kontext für die Frage eingefügt werden. Dieses Einfügen von Kontext in den Prompt wird RAG genannt. Dabei muss bei RAG nicht zwingend eine Vektor-Datenbank verwendet werden. RAG beschreibt lediglich das Einfügen von zusätzlichem Kontext in einen Prompt. In vielen Fällen bietet sich das Nutzen von Vektor-Datenbanken allerdings an. Folgende Grafik, von der Webseite der Open-Source Vektor-Datenbank „ChromaDB“ [78], zeigt den RAG-Ablauf bei der Nutzung einer Vektor-Datenbank:

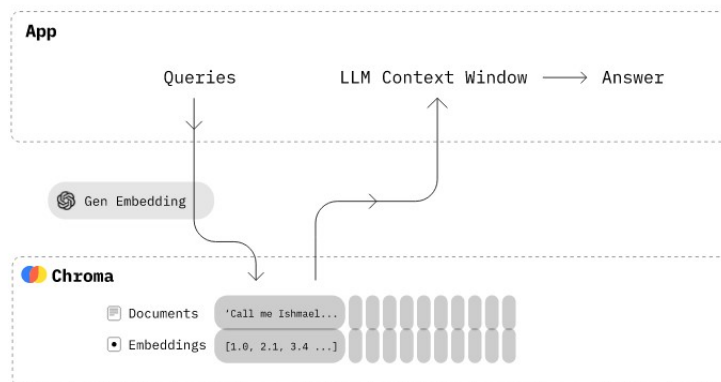


Abbildung 49: Veranschaulichung von RAG bei der Verwendung einer Vector-Datenbank [78]

Es gibt eine Vielzahl an Optionen im Bereich Vektor-Datenbanken. Eine sehr beliebte Cloud-Plattform für Vektor-Datenbanken ist „Pinecone“ [79], welche das Erzeugen und Nutzen von Vektor-Datenbanken über eine einfache REST-API erlaubt. LangChain bietet einen Wrapper für die Pinecone-API, welche damit nativ unterstützt ist. Abseits von Pinecone gibt es auch Open-Source Optionen, wie die „ChromaDB“ Vektor-Datenbank. Anders als Pinecone, läuft ChromaDB standardmäßig In-Memory auf dem eigenen System, was die Leistung im Vergleich zur Kommunikation über eine REST-API beschleunigt. Zusätzlich ist ebenfalls möglich, ChromaDB in einer Client-Server-Architektur laufen zu lassen. Auch ChromaDB wird nativ von LangChain unterstützt. Um thematisch beim Open-Source Thema zu bleiben, wird im folgenden ChromaDB für die Codebeispiele verwendet. Die Funktionsweise ist bei Pinecone allerdings sehr ähnlich.

```
from langchain.vectorstores import Chroma as ch
from langchain.document_loaders import PyPDFLoader
from langchain.chains import RetrievalQA
from langchain.embeddings.huggingface import HuggingFaceEmbeddings

file = PyPDFLoader("Bachelorarbeit_Final.pdf").load_and_split()

embed_model = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2")

vec_db = ch.from_documents(file, embedding=embed_model)

ba_rag_chain = RetrievalQA.from_chain_type(llm=tgi_llm, chain_type="stuff",
    retriever=vec_db.as_retriever())
```

Abbildung 50: Codebeispiel für das Erstellen einer RAG-Chain

3. LangChain

Im Codebeispiel wird eine PDF-Datei mithilfe eines der vielen „Document_loader“, welche LangChain mitbringt, geladen und mit der „load_and_split“ Methode gleichzeitig in kleinere Textteile zerlegt. Anschließend wird ein Embedding-Modell geladen. In diesem Fall wird ein kleines Modell namens „all-MiniLM-L6-v2“ [80] von HuggingFace geladen und lokal ausgeführt. Dieses Modell wird standardmäßig von ChromaDB verwendet und benötigt nicht viel Leistung, grundlegend kann allerdings ein beliebiges Modell verwendet werden. Daraufhin wird ein ChromaDB-Objekt erzeugt, welchem die geladene Datei und das Embedding-Modell übergeben werden. Final kann die „RetrievalQA“-Chain erstellt werden. Dieser wird die „as_retriever“ Methode des ChromaDB-Objekts, welche intern eine Abfrage an die Datenbank stellt, sowie ein LLM-Verweis übergeben. Wichtig ist zusätzlich noch das Festlegen des „Chain_type“. Dieser bestimmt, wie die Daten aus der Vektor-Datenbank in den Prompt eingefügt werden. Im Beispiel wird die einfachste Option „stuff“ verwendet, welche alle Treffer der Datenbank direkt nacheinander in den Prompt einfügt.

Diese Darstellung zeigt die einfachste Implementierung einer RAG-Chain, bei der große Teile des Ablaufes von LangChain abstrahiert werden. Insbesondere bei komplexeren Aufgaben, kann es Sinn ergeben, anstatt mit der spezifischen „RetrievalQA“-Chain zu arbeiten, eine einfache LLM-Chain zu verwenden und die Erstellung und Datenabfrage der Vektor-Datenbank manuell zu implementieren. Dies führt auf der einen Seite zu mehr Code und damit mehr Arbeitsaufwand, ermöglicht aber auf der anderen Seite mehr Kontrolle über die einzelnen Schritte. Im folgenden Codebeispiel ist zu sehen, wie die ChromaDB Bibliothek (anstatt des LangChain Wrappers) verwendet wird, um eine neue Vektor-Datenbank zu erzeugen. Daraus kann später, in Kombination mit einer Chain, eine RAG-Chain erzeugt werden.

```
import chromadb

# Load and split the file using langchains PyPDF wrapper
file = PyPDFLoader("Bachelorarbeit_Final.pdf").load_and_split()

# Create the new Database
chroma = chromadb.Client()
collection = chroma.get_or_create_collection("DummyCollection")
# Add the data and metadata from the file to the Database
collection.add(
    documents=[file[i].page_content for i in range(0, len(file))],
    ids=[str(i) for i in range(0, len(file))],
    metadatas=[file[i].metadata for i in range(0, len(file))])
```

Abbildung 51: Codebeispiel für das Erzeugen einer ChromaDB Vektor-Datenbank anhand einer PDF-Datei

Wie zu erkennen ist, werden zuerst die Daten geladen und geteilt. Das geschieht identisch zum vorherigen Beispiel. Anders als beim vorherigen Beispiel besteht das Erstellen der Datenbank und Laden der Daten aus mehr Codezeilen. Daraufhin wird ein neuer ChromaDB Client erstellt und genutzt, um eine neue „Collection“ zu erhalten, welche die Daten hält. Eine „Collection“ kann etwa mit einer Tabelle bei einer SQL-Datenbank verglichen werden. Schließlich werden die Texte mit der „add“ Methode in die Datenbank geladen. Das Angeben von Metadaten ist dabei optional, wird im Beispiel allerdings durchgeführt. Ein wichtiger Unterschied ist vor allem, dass keine separate Embedding-Funktion angegeben werden muss. Die ChromaDB-Bibliothek kommt standardmäßig mit einem kleinen Modell („all-MiniLM-L6-v2“), welches das Embedding übernimmt.

3. LangChain

Es ist aber auch möglich ein anderes Modell zu verwenden, dieses muss dann bei der Erstellung der Collection angegeben werden. Anschließend kann das PromptTemplate und die LLM-Chain definiert werden.

```
# Create a very basic RAG Prompt
custom_rag_template = """Answer the question based on the following context.
Context: {context}
Question: {input}
Response: """
custom_rag_prompt = PromptTemplate(template=custom_rag_template,
                                   input_variables=["input", "context"])
# Create the RAG-LLMChain
custom_rag_chain = LLMChain(llm=tgi_llm, prompt=custom_rag_prompt, verbose=True)
# Define the userinput
input = "what did jonas bevernis do?"
# Query the Database
result = collection.query(
    query_texts=[input],
    n_results=2)
```

Abbildung 52: Codebeispiel für das Erzeugen einer RAG-Chain basierend auf der LLMChain Klasse

Der Ablauf ähnelt den vorherigen LLM-Chain Beispielen. Wobei die Nutzereingabe bereits vor dem Ausführen der „run“-Methode definiert werden muss, da diese zuerst für die Datenbankabfrage genutzt wird. Dies geschieht mit der „query“ Methode, in der zusätzlich zum Anfragetext noch mit „n_results“ angegeben werden kann, wie viele Resultate zurückgegeben werden sollen. In diesem Fall werden die 2 besten Texte angefordert.

Werden die Outputs der beiden Chains betrachtet, ist zu sehen, dass diese nahezu identisch funktionieren:

```
custom_rag_chain.run({"input": input, "context": result["documents"][0]})
# >> 'Jonas Bevernis wrote his bachelor thesis on the topic of
# >> "Funktionsweise, Anwendungen und Risiken von Texterzeugungsmodellen
# >> am Beispiel von GPT" at the Hochschule Stralsund.'

ba_rag_chain.run(input)
# >> 'Jonas Bevernis wrote a thesis about GPT.'
```

Abbildung 53: Codebeispiel für die Ausführung von RAG-Chains

3. LangChain

3.7. Kritik an LangChain

In der Einleitung dieses Kapitels wurde von der Vielfalt von LangChain als Stärke der Bibliothek gesprochen. Genau diese Größe des Projekts wird an LangChain allerdings auch kritisiert. Große Bestandteile der Funktionalität von LangChain bestehen daraus, LLM-Aspekte zu abstrahieren, welche, einzeln für sich genommen, nicht sehr komplex sind. Ein Beispiel dafür stellt das Nutzen der ChromaDB Vektor-Datenbank aus dem vorherigen Kapitel dar. Dort war zu sehen, dass LangChain einen Wrapper für die ChromaDB-Bibliothek bietet, welche eine einfache Nutzung in Kombination mit LangChain ermöglicht. Wie ebenfalls zu sehen war, stellt auch das Verwenden der originalen ChromaDB-Bibliothek keine sehr komplexe Aufgabe dar. Dafür bietet die originale Bibliothek zusätzliche Funktionalität, welche im LangChain-Wrapper nicht vorhanden ist. Ein weiteres Beispiel sind die Prompt-Templates, welche für ihre Hauptfunktionalität auf Pythons F-String Funktionalität heruntergebrochen werden können.

In dieser Weise bietet LangChain viel Funktionalität, welche aus einer Reihe, einzeln betrachtet, einfacher Konzepte und Wrapper für andere Bibliotheken besteht. Dabei reduziert das Nutzen der LangChain Wrapper, wie beim ChromaDB Beispiel dargestellt wurde, den Arbeitsaufwand oft nur marginal, im Vergleich zur Nutzung der originalen Bibliotheken. Kritiker sagen, dass zudem große Bestandteile der Funktionalität weitestgehend überflüssig sind. Diese Funktionalität besteht vorwiegend aus der Abstrahierung leicht selbst zu implementierender Konzepte. So wird LangChain primär vorgeworfen, dass die Bibliothek „Over Engineered“ wäre und viele Aspekte zwar abstrahiert, aber nicht vereinfacht werden. [81]

Nach der Meinung des Autors ist diese Kritik valide. Insbesondere für Entwickler, welche sich bereits ins Detail mit den verschiedenen LLM-Konzepten auskennen, wäre die Implementierung, der für ein bestimmtes Projekt benötigten Aspekte, in der Tat nicht sonderlich komplex. Besonders aufgrund der teilweise lückenhaften LangChain-Dokumentation kann es schwieriger sein, herauszufinden, wie eine bestimmte Funktionalität in LangChain umgesetzt wurde, als diese Funktionalität manuell zu implementieren. Allerdings ignoriert diese Kritik einen besonders wichtigen Aspekt, sie ist nur dann zutreffend, wenn sich ein Entwickler bereits hervorragend mit den vielen LLM-Konzepten auskennt. Da LLMs und die verschiedenen Techniken, wie RAG, sehr neu sind und sich rapide weiterentwickeln, ist genau dieses umfangreiche Vorwissen bei vielen Entwicklern nicht vorhanden. An dieser Stelle bietet LangChain dennoch einen großen Wert, es bündelt eine gewaltige Menge der für die Verwendung von LLMs benötigten Funktionalität und kann so, nicht nur als nützliches Tool für die Entwicklung von LLM-Applikationen verwendet werden, sondern auch als sehr umfangreiche Ressource beim Erlernen der verschiedenen LLM-Techniken dienen, was es dann wiederum ermöglicht, gewisse Funktionalität gegebenenfalls manuell zu implementieren.

4. Praxisteil

In den vorherigen Kapiteln wurden die Grundlagen für den folgenden Praxisteil geklärt. Es wurde ein Modell ausgesucht (LLaMA 2 – Platypus2 in der 7B und 13B Version) sowie die Grundlagen von LangChain dargestellt. In diesem Kapitel werden die beide Aspekte verbunden und die anfangs angesprochene LangChain Applikation zur Kontrolle einer API, konkret eines Smart-Home-Systems, erstellt. Zu diesem Zweck soll zuerst im Unterkapitel „4.1. Planung“ dargestellt werden, wie die Applikation zu entwickeln ist, sowie welche Features es zu implementiert gibt. Auch wird dargestellt, welche Features für den Zweck dieser Arbeit unbeachtet bleiben. Anschließend wird die Umsetzung besprochen, indem die Applikation dargestellt und die Funktionsweise aus Nutzer- und Softwaresicht beschrieben wird. Zudem soll die Hardwareseite besprochen und die aufgetretenen Probleme sowie ihre Lösungen dargestellt werden. Zuletzt wird in der Auswertung überprüft, ob die geplanten Ziele erreicht wurden und die Erkenntnisse und Schlussfolgerungen aus der Entwicklung der Applikation aufgezeigt werden.

4.1. Planung

Die Planung der Applikation ist grob an „echter“ Softwareprojektplanung angelehnt. Diese ist allerdings stark vereinfacht, da es erstens nicht das Ziel dieser Arbeit ist, eine umfangreiche Softwareprojektplanung zu entwerfen und zweitens da viele Aspekte der Softwareprojektplanung im Kontext dieser Arbeit wenig Sinn ergeben. So wäre etwa eine ausführliche Stakeholder-Analyse nicht sinnvoll. Im Folgenden werden dementsprechend nur die Grundlagen dargestellt, nach denen die Applikation entwickelt werden soll.

4.1.1. Anforderungen

Zuerst müssen die Anforderungen an die Applikation geklärt werden. Dadurch wird beschrieben, welche Funktionalität die Applikation am Ende bieten soll („Muss“) und welche Funktionen nicht zwingend notwendig sind („Kann“).

| ID | Name | Anforderungsbeschreibung | Notwendigkeit |
|----|-----------------------------------|---|---------------|
| 1 | Geräte ein- und ausschalten | Die Applikation soll in der Lage sein, Geräte im Smart-Home-System ein- und auszuschalten. | Muss |
| 2 | Farbe von Lichtern ändern | Es soll möglich sein, die Farbe von Lichtern zu ändern. | Muss |
| 3 | Helligkeit von Lichtern ändern | Es soll möglich sein, die Helligkeit von Lichtern anzupassen. | Muss |
| 4 | Status von Geräten abrufen | Die Möglichkeit abzurufen, ob ein Gerät ein- oder ausgeschaltet ist. | Muss |
| 5 | Farbe von Lichtern abrufen | Die Möglichkeit abzurufen, welche Farbe ein Licht gerade hat. | Muss |
| 6 | Helligkeit von Lichtern abrufen | Die Möglichkeit abzurufen, wie hell ein Licht gerade eingestellt ist. | Muss |
| 7 | Bedienung mit natürlicher Sprache | Die Möglichkeit, die Software mit natürlicher Sprache (Text) zu bedienen (Englisch). | Muss |
| 8 | Ausgaben in natürlicher Sprache | Die Applikation antwortet auf Anfragen mit natürlicher Sprache (Text). | Muss |
| 9 | HomeAssistant (HA) Integration | Die Applikation muss mit einer Instanz der Smart-Home-Software „HomeAssistant“ interagieren können. | Muss |

4. Praxisteil

| | | | |
|----|-----------------------------------|--|------|
| 10 | Allgemeine Chat Funktionalität | Neben der Möglichkeit das Smart-Home-System zu kontrollieren, bietet die Applikation die Möglichkeit auch andere Anfragen zu stellen, wie bei einem Chatbot. | Kann |
| 11 | Einfaches Hinzufügen neuer Geräte | Es ist möglich, neue Smart-Home-Geräte direkt über die Applikation mit natürlicher Sprache hinzuzufügen. | Kann |
| 12 | Chat-Historie | Die Applikation erinnert sich an die vorherige Kommunikation und der Nutzer kann auf diese Bezug nehmen. | Kann |
| 13 | Custom-Befehle | Der Nutzer kann selbst mit natürlicher Sprache neue Befehle definieren, welche dann später vom Nutzer verwendet werden können. | Kann |

Tabelle 12: Softwareplanung: Anforderungen

4.1.2. Eingrenzung

Nachdem die Anforderungen dargestellt wurden, sollen in diesem Kapitel einige Eingrenzungen aufgezeigt werden, welche Funktionalität zwar naheliegend wäre, allerdings im Umfang dieser Arbeit nicht geplant ist.

| Name | Beschreibung | Erläuterung |
|---|--|--|
| Weitere spezielle Geräte | Die Möglichkeit der Applikation weitere spezielle Eigenschaften von Geräten zu manipulieren (zum Beispiel Temperatur von Thermostat) | Wie beschrieben, soll die Applikation in der Lage sein, Geräte allgemein ein- und auszuschalten. Darüber hinaus wird am Beispiel von Lichtern dargestellt, wie speziellere Eigenschaften angepasst werden können. Da die Applikation nur als Beispiel dienen soll, reicht es dies einmalig darzustellen. Die Implementierung von weiteren speziellen Geräten wie beispielsweise Thermostaten läuft dann identisch ab. |
| GUI | Eine nutzerfreundliche grafische Oberfläche für die Applikation. | Ziel der Applikation ist es zu zeigen, wie die zuvor untersuchten Aspekte „OS-LLM“ und „LangChain“ zusammen eingesetzt werden können, um ein Problem zu lösen. Das Entwickeln einer GUI für die Applikation führt dieses Ziel nicht entscheidend voran und liegt damit außerhalb des Scopes dieser Arbeit. |
| Spracheingabe | Die Möglichkeit die Applikation direkt mit der eignen Stimme zu steuern. | Die Funktionalität ist naheliegend, da die meisten bekannten Smart-Home-Assistenten so bedient werden. Genau wie eine GUI, hat dies allerdings mit den Grundaspekten dieser Arbeit nicht viel zu tun und liegt deshalb ebenfalls nicht im Scope der Arbeit. |
| Anbindung an weitere Smart-Home-Plattformen | Die Möglichkeit mit weiteren Plattformen als nur HomeAssistant zu interagieren. | Grundlegend ist die Idee der Applikation, ein LLM mit einer REST-API zu verbinden. HomeAssistant dient dabei nur als ein leicht verständliches Beispiel. Das Anbinden an andere Smart-Home-APIs würde grundlegend genauso ablaufen, wie die gezeigte Anbindung an HomeAssistant. Ziel ist es nicht den „perfekten“ Smart-Home-Assistent zu bauen, sondern vielmehr beispielhaft zu zeigen, wie eine API mit einem LLM bedient werden kann. |

Tabelle 13: Softwareplanung: Eingrenzung

4. Praxisteil

4.1.3. Softwarekontext

Ziel des Softwarekontext ist es darzustellen, wo sich die zu entwickelnde Applikation zwischen schon bestehender Software einordnet. Dabei wurde der fachliche Kontext (gestrichelte Pfeile – Informationsfluss) und der technische Kontext (durchgezogenen Linien – Schnittstellen) kombiniert in einer Grafik dargestellt.

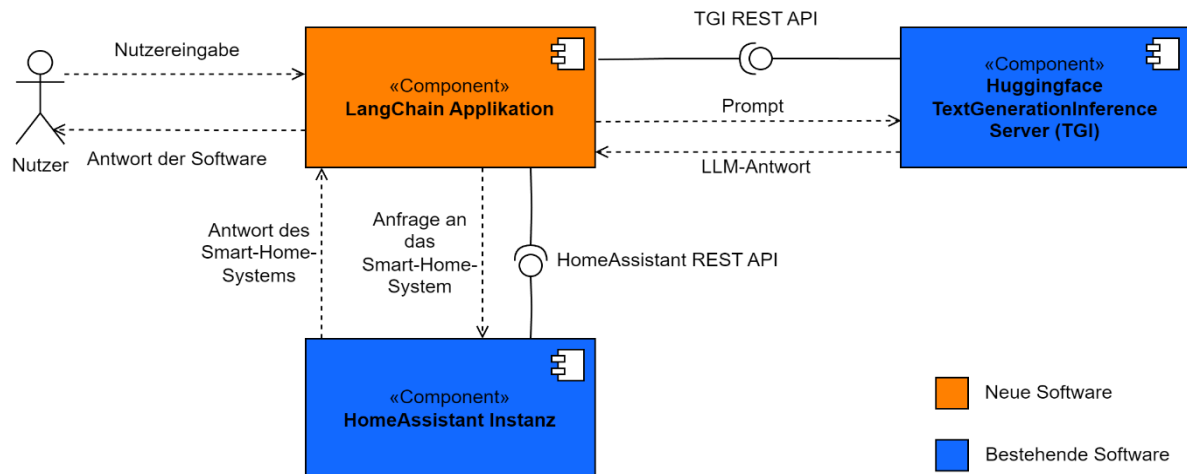


Abbildung 54: Softwareplanung: Fachlicher- und technischer Kontext (eigene Grafik)

Wie in der Abbildung zu sehen ist, gibt es insgesamt drei Komponenten im Softwarekontext. Das ist zuerst die zu entwickelnde Applikation. Diese interagiert mit der von einer HomeAssistant-Instanz bereitgestellten REST-API und übermittelt so die Befehle bzw. Anfragen an das Smart-Home-System. Dabei erhält die Applikation, abhängig von der Anfrage, eine passende Antwort von HomeAssistant.

HomeAssistant ist eine Open-Source Smart-Home- und Automationssoftware, welche es Nutzern ermöglicht, eine eigene Instanz in ihrem Netzwerk zu hosten. Dabei bietet HomeAssistant eine große Menge an Funktionen, von der einfachen Steuerung von Smart-Home-Geräten bis zur Erstellung und Ausführung von eigenen Automatisierungsskripten. Einer der größten Vorteile von HomeAssistant ist die Unterstützung vieler unterschiedlicher Smart-Home-Ökosysteme, was die Integration einer breiten Menge an Smart-Home-Hardware ermöglicht. Im Softwarekontext der Applikation übernimmt HomeAssistant die Kommunikation mit den Geräten.

Neben der Applikation und HomeAssistant ist der TGI-Server die letzte Komponente im Softwarekontext. TGI übernimmt das Hosten des LLMs und bietet dabei ebenfalls eine REST-API an, welche das LLM für externe Applikationen zur Verfügung stellt. Die Applikation sendet die Prompts mithilfe dieser API an TGI und erhält daraufhin eine LLM-Antwort.

Der Text Generation Inference Server von Huggingface ermöglicht es, ähnlich wie HomeAssistant, eine eigene Instanz der Software zu hosten, welche eine REST-API zur Kommunikation bereitstellt. Dabei steht TGI unter anderem als Docker Container zur Verfügung und lässt sich damit, mit geringem Aufwand, auf beliebiger Hardware ausführen. Neben TGI gibt es auch andere Open-Source-Projekte, welche ähnliche Funktionalität ermöglichen. So etwa „vLLM“ [82] oder die „text-generation-webui“ [83]. Der entscheidende Vorteil, welcher maßgeblich die Wahl von TGI beeinflusst hat, besteht in der Tatsache, dass

4. Praxisteil

TGI von HuggingFace in der eigenen Produktivumgebung eingesetzt wird und dementsprechend stabil laufen sollte.

Als Alternative zur Verwendung eines LLM-API-Servers wie TGI, ist es ebenfalls möglich, das LLM direkt innerhalb der Applikation zu laden und lokal auszuführen. Dieser Ansatz ist strukturell einfacher, bietet allerdings gleichzeitig eine Vielzahl an Nachteilen. So wird mit diesem Ansatz sehr hohe lokale Rechenleistung vorausgesetzt, um das LLM lokal auszuführen. Wird stattdessen TGI (oder vergleichbare Software) verwendet, kann diese Rechenleistung auf beliebige Hardware ausgelagert werden (zum Beispiel in der Cloud gemietete Server). Weiterhin macht diese Trennung von Applikation und LLM es einfacher, die LLM-Hardware zu wechseln und auf das gewünschte LLM abzustimmen. Auch das Wechseln zwischen verschiedenen LLMs fällt damit einfacher und schneller aus.

4.1.4. Beschreibung der API-Schnittstellen

In diesem Kapitel sollen die benötigten Schnittstellen beschrieben werden. Das ist erstens die TGI-REST-API und zweitens die HomeAssistant-REST-API. Dabei findet diese Beschreibung vereinfacht statt, indem jeweils nur die für die Applikation relevanten API-Routen beschrieben werden. LangChain übernimmt die Integration der TGI-API, während die HomeAssistant-API manuell implementiert werden muss.

4.1.4.1. Text Generation Inference Server API

POST – /generate

Unter dieser Route wird ein Text vom LLM generiert und als Antwort an den Client geschickt. Dies ist die Haupt-Route, die von der Applikation verwendet wird. Dabei wird ein JSON-Objekt mit Informationen und Hyper-Parametern erwartet, welche die Anfrage an das LLM spezifizieren. Dieses Input JSON-Objekt kann beispielsweise wie folgt aussehen:

```
{
  "inputs": "My Name is Jonas, ",
  "parameters": {
    "max_new_tokens": 20,
    "repetition_penalty": 1.03,
    "stop": ["\n"],
    "temperature": 0.5,
    "top_k": 10,
    ...
  }
}
```

Abbildung 55: Verkürzter JSON-Input für die TGI API - POST /generate Schnittstelle

Das Rückgabeobjekt der API ist grundlegend sehr einfach und sieht vereinfacht wie folgt aus:

```
{
  "details": { ... },
  "generated_text": "test"
}
```

Abbildung 56: Verkürzter JSON-Output der TGI API - POST /generate Schnittstelle

Unter dem Key „generated_text“ erfolgt die Textausgabe, während unter „details“ zusätzliche Informationen zum generierten Text ausgegeben werden, welche hier allerdings nicht weiter relevant sind und deshalb nicht genauer dargestellt werden.

POST – /generate_stream

Die „generate_stream“ Route ist ähnlich, wie die „generate“ Route. Dabei sind die Input- und Output-Objekte identisch zu „generate“. Anstatt die gesamte Textausgabe in einem an den Client zu schicken, wird ein „stream“ erzeugt, in dem jeder neu erzeugte Token einzeln verschickt wird. Dies ist zum Beispiel von AI-Chatbots wie ChatGPT bekannt, bei denen nicht der gesamte Text auf einmal angezeigt wird, sondern sich Token für Token aufbaut.

4.1.4.2. HomeAssistant API

Die HomeAssistant REST-API [84] ist umfangreich, für die Applikation sind allerdings nur zwei Routen relevant: „/services“ und „/states“. Dabei muss jeweils im Header des API-Request ein Authentifizierungstoken übermittelt werden.

GET - /states/<entity_id>

Über diese Schnittstelle kann der Status eines Geräts abgefragt werden. Eine Anfrage an „/states/sun.sun“ würde von der API beispielsweise so beantwortet werden:

```
{
  "attributes": {
    "azimuth": 336.34,
    "elevation": -17.67,
    "friendly_name": "Sun",
    "next_rising": "2016-05-31T03:39:14+00:00",
    "next_setting": "2016-05-31T19:16:42+00:00"
  },
  "entity_id": "sun.sun",
  "last_changed": "2016-05-30T21:43:29.204838+00:00",
  "last_updated": "2016-05-30T21:50:30.529465+00:00",
  "state": "below_horizon"
}
```

Abbildung 57: Antwort der HomeAssistant API auf eine GET-Anfrage an /states/sun.sun [85]

An die gleiche Route können auch POST-Anfragen geschickt werden, damit werden allerdings nicht die Eigenschaften der tatsächlichen Geräte geändert, sondern nur ihre Repräsentation in HomeAssistant. Eine Lampe wird dann etwa als eingeschaltet in der Web-UI angezeigt, ist aber in der Realität ausgeschaltet. Um den Status von Geräten tatsächlich zu verändern, wird die „/services“ Router verwendet.

POST – /services/<domain>/<service>

Um die Funktionsweise dieser Schnittstelle darzustellen, ist es wichtig zu erläutern, was HomeAssistant unter den Begriffen „Domain“ und „Service“ versteht. Eine Domain ist in HomeAssistant eine Gruppierung nach Gerätetypen. So werden Lichter etwa der Domain „light“ zugeordnet, während ein Fernseher zur „media_player“ Domain gehört. Dabei können Geräte auch mehreren Domains angehören. So können alle Geräte immer auch über die „homeassistant“ Domain angesprochen werden. Jede der Domains bietet eine Reihe an „Services“ an. Diese Services ermöglichen es, den Status eines Geräts zu verändern. Einer dieser Services wäre zum Beispiel „turn_on“ welcher Geräte einschaltet. Dabei bietet die „homeassistant“ Domain nur grundlegende Services wie „turn_on“ oder „turn_off“ an, die im Normalfall von allen Geräten unterstützt werden. Während spezifische Domains wie „light“ zusätzliche Services anbieten, welche nur vom jeweiligen Gerätetyp unterstützt werden. Die jeweilige Geräte-ID (bzw. „entity_id“) sowie gegebenenfalls weitere Optionen, werden diesen Services als JSON-Objekt übergeben.

4. Praxisteil

Das Einschalten einer Lampe und gleichzeitige Setzen der Farbe auf Blau mithilfe des „turn_on“ Service könnte beispielsweise wie folgt aussehen:

```
POST - /services/light/turn_on

{
  "entity_id": "light.bedroom_1",
  "rgb_color": "[0,0,255]"
}
```

Abbildung 58: Beispiel für ein POST Request zum Einschalten einer Lampe und setzen der Farbe auf blau

4.1.5. Grobarchitektur der Applikation

Nachdem die Grundlagen geklärt wurden, geht es im Folgenden darum, die tatsächliche Applikation zu planen. Dazu wird in diesem Kapitel ein grober Architekturplan vorgestellt, auf Basis dessen die Applikation entwickelt werden soll. Dabei lässt sich die Architektur in zwei Bereiche einteilen: Die klassische Softwarearchitektur, welche grundlegend die Logik der Applikation beschreibt und vor allem für die Ausführung von Befehlen für das Smart-Home-System verantwortlich ist und die LLM-Architektur, welche über LangChain definiert wird und die Verbindung der verschiedene Chains beschreibt.

Dabei soll zuerst die LLM-Architektur beschrieben werden. Diese ist in der folgenden Abbildung zu erkennen.

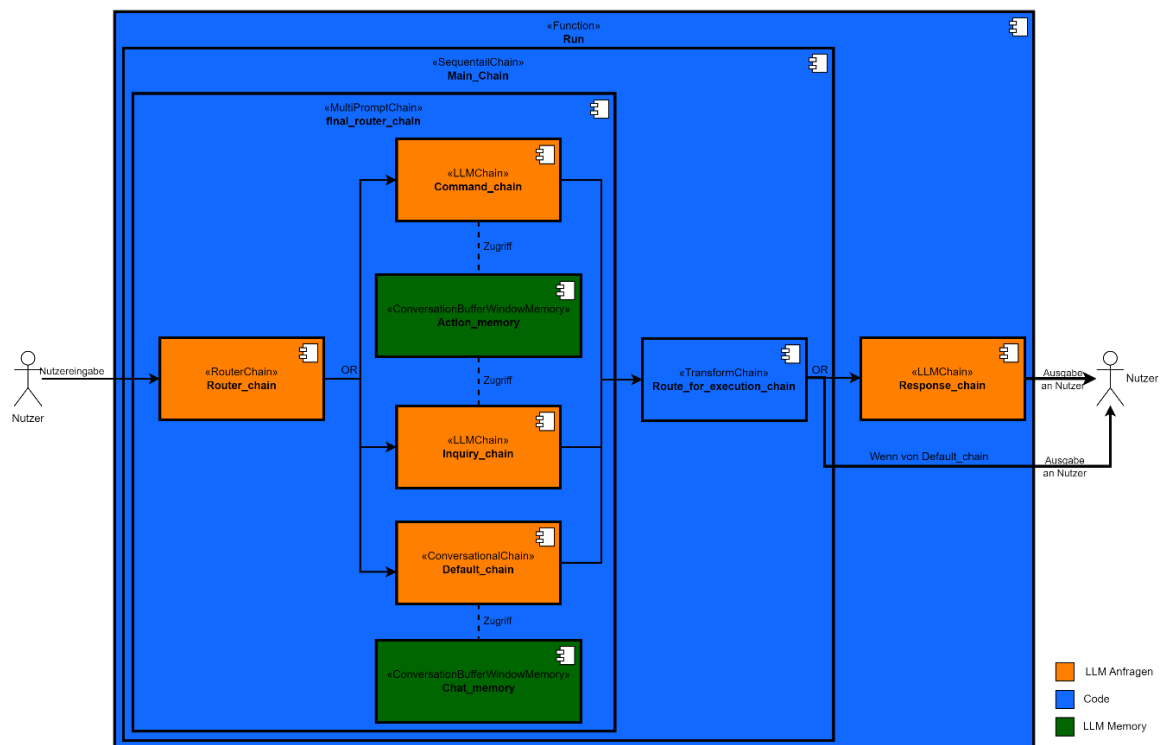


Abbildung 59: LLM-Architektur der Applikation (eigene Grafik)

Für das Verständnis der Abbildung ist es zunächst wichtig, die Farbcodierung zu verstehen. Orange markiert sind alle Bestandteile, die direkt mit einem LLM interagieren und eigene Prompts verwenden. Grün markiert ist das LLM-Memory, für einige der Chains, und blau dargestellt sind alle Bestandteile, die nicht direkt mit LLMs zu tun haben, sondern normalen Code ausführen. Mit diesem Verständnis ist die Architektur der LLMs leicht zu verstehen.

4. Praxisteil

An erster Stelle steht die „Router_chain“, welche anhand der Nutzereingabe entscheidet, ob es um eine Status-Abfrage („Inquiry“), ein Kommando zur Veränderung eines Status („Command“) oder um etwas völlig anderes („default“) geht. Auf Basis der Entscheidung der „Router_chain“ wird dann an eine passende Chain für die spezifische Aufgabe weitergeleitet. Das ist entweder die „Command_chain“, die „Inquiry_chain“ oder die „Default_chain“. Die „Router_chain“ wird mit diesen drei Ziel-Chains durch die MultiPromptChain „final_router_chain“ verbunden. Dabei wird jeweils nur eine der drei Ziel-Chains pro Durchlauf der umfassenden „Run“-Funktion ausgeführt. Wichtig ist dabei auch, das Memory-System zu erkennen. Es gibt zwei getrennte Memory-Speicher, einmal das „Action_memory“, auf welches die „Command_chain“ und die „Inquiry_chain“ Zugriff haben, und das „Chat_memory“, auf welches nur die „Default_chain“ Zugriff hat. Diese Teilung der Memory-Systeme soll verhindern, dass Konversationen, die keinen direkten Bezug zum Smart-Home-System haben, das „Action_memory“ verunreinigen und damit potenziell zu schlechteren Ausgaben der „Command_chain“ und „Inquiry_chain“ führen. Mithilfe der umfassenden SequentialChain „Main_chain“ wird der Output, der jeweils ausgeführten Chain, anschließend an die TransformChain „Route_for_execution_chain“ weitergeleitet. Diese Chain ist blau markiert, da in einer TransformChain, wie zuvor erklärt, lediglich eine vordefinierte Funktion ausgeführt wird. Trotzdem ist die „Route_for_execution_chain“ ein wichtiger Bestandteil der LLM-Architektur. Diese TransformChain ist damit der Einstieg in den restlichen Applikationscode, von hier wird die Steuerung des Smart-Home-Systems durchgeführt. Dabei gibt es zwei unterschiedliche Ergebnisse der TransformChain: Wurde eine Aktion mit dem Smart-Home-System durchgeführt, so wird schließlich an eine letzte Chain weitergeleitet, welche auf Basis der Antwort des Smart-Home-Systems, eine für den Nutzer leicht verständliche Antwort in natürlicher Sprache generiert. Wurde zuvor allerdings die „Default_chain“ ausgewählt, wird diese letzte „Response_chain“ übersprungen und die Ausgabe der „Default_chain“ direkt an den Nutzer ausgegeben.

Nachdem vorausgehend die LLM-Architektur beschrieben wurde, kann auch die Software-Architektur der restlichen Applikation geplant werden. Als Einstiegspunkt dafür dient die „Route_for_execution_chain“ TransformChain. Dieser wird eine Transform-Funktion übergeben, welche intern an die entsprechenden Teile im System weiterleitet, welche für die Kommunikation mit der API zuständig sind. Die Grobarchitektur für diesen Teil der Software sieht wie folgt aus:

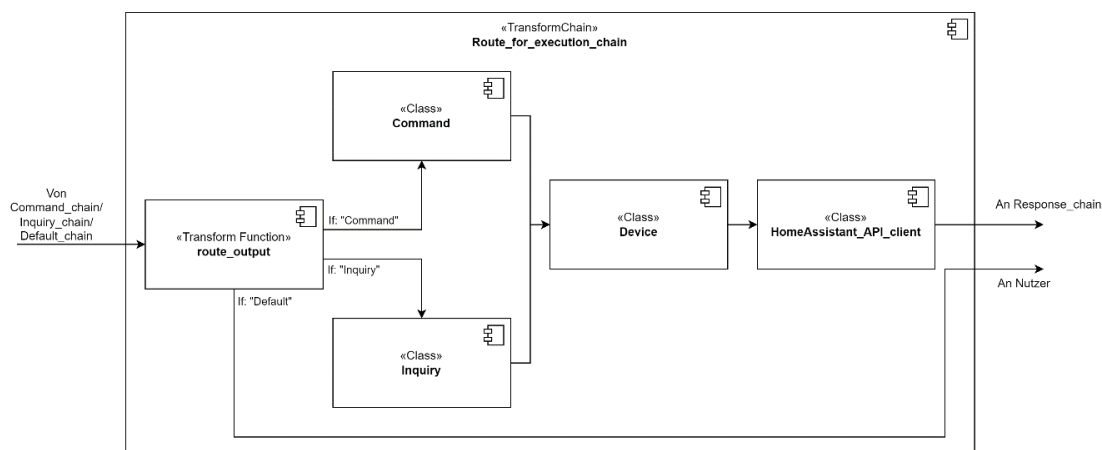


Abbildung 60: Grobarchitektur der Applikation (eigene Grafik)

4. Praxisteil

Wie in der Abbildung zu sehen ist, ist die grundlegende Architektur recht einfach gestaltet. Es gibt eine „Command“ Klasse, die sich um das Ausführen von Kommandos (etwa das Einschalten eines Geräts) kümmert und eine „Inquiry“ Klasse, welche das Abfragen von Informationen übernimmt. Beide Klassen arbeiten dabei wie eine Art komplexer Output-Parser, indem sie die Text-Ausgabe der LLMs verarbeiten und ausführbar machen. Dabei greifen beide Klassen schließlich auf eine dritte „Device“ Klasse zu, welche für die Ausführung einzelner Befehle zuständig ist. Diese Klasse könnte, vereinfacht dargestellt, Methoden wie „turn_on“ oder „get_state“ haben. Um die Befehle an HomeAssistant zu senden, gibt es zuletzt noch eine „HomeAssistant_API_client“ Klasse, welche die Interaktion mit der HomeAssistant-API übernimmt. Nicht dargestellt ist hier, dass die „Device“ Klasse lediglich als Grundklasse dienen soll und gegebenenfalls noch weitere Klassen wie „Light“ implementiert werden können, welche von „Device“ erben und speziellere Funktionalität bieten. Dies wird genauer im Kapitel „4.2.1. Beschreibung von Details“ dargestellt.

4.1.6. Prompt Planung

Das Planen der Prompts ist besonders wichtig, da die Leistung bzw. Fehlerrate der Applikation am Ende direkt von diesen abhängig ist. Dem Planen und Testen der Prompts sollte deshalb ebenso viel Zeit gewidmet werden wie, dem Entwerfen der restlichen Software.

Im Kapitel „2.3. Vergleich verschiedener Open-Source-Modelle“ traf die Entscheidung auf ein LLaMA 2 Modell, genauer „Platypus2“. Das Modell setzt folgendes grundlegendes Prompt-Template Format voraus:

```
### Instruction:
<prompt>
### Response:
```

Abbildung 61: Grundlegendes Prompt-Template für Platypus2 Modelle

Wie anhand der LLM-Architektur zu erkennen war, gilt es insgesamt 5 verschiedene Prompts zu planen: das „Router“-Prompt, das „Command“-Prompt, das „Inquiry“-Prompt, das „Default“-Prompt und das „Response“-Prompt. Dabei können diese Prompts in zwei Gruppen unterteilt werden:

1. Prompts für interne Kommunikation
 - a. „Router“-Prompt
 - b. „Command“-Prompt
 - c. „Inquiry“-Prompt
2. Prompts mit direktem Kontakt zum Nutzer
 - a. „Default“-Prompt
 - b. „Response“-Prompt

Diese Unterteilung ist wichtig, da sich die Ausgabeformate der Prompts beider Gruppen signifikant unterscheiden. Prompts aus Gruppe 1 müssen intern von Programmcode weiterverarbeitet werden. Ihre Ausgaben müssen dementsprechend in einem Format vorliegen, welches leicht zu verarbeiten ist. Genau wie es auch schon bei den Leistungs-Tests der Fall war, sollen diese Prompts eine Ausgabe im JSON-Format erzeugen, da JSON leicht zu parsen und zu verarbeiten ist.

4. Praxisteil

Im Gegensatz dazu soll die Ausgabe der Prompts aus Gruppe 2 direkt an den Nutzer geschickt werden. Diese müssen dementsprechend in natürlicher Sprache vorliegen. Die Planung beginnt mit den Prompts aus Gruppe 1, den Anfang macht dabei das „Router“-Prompt:

```
### Instruction:
Below are some "USER REQUESTS" and a "RESPONSE" from an AI assistant. The AI
assistant is connected to a smart home system, so some USER REQUESTS might ask
the assistant to interact with the smart home system. Respond in a JSON format
as defined under "JSON STRUCTURE". Change the "destination" to one of the
options provided under "DESTINATION OPTIONS". "next_inputs" Should always be an
exact copy of the last USER REQUEST.

JSON STRUCTURE: {"destination": DESTINATION OPTION, "next_inputs": USER
REQUEST}}

DESTINATION OPTIONS:
{router_destenation_options}
DEFAULT: Usefull if the user is trying to make conversation and the request
does not pertain to matters of the Smart Home system.

USER REQUEST: "How are you?"
RESPONSE: {"destination": "DEFAULT", "next_inputs": "How are you?"}
USER REQUEST: "{input}"
### RESPONSE:
```

Abbildung 62: Prompt-Template für die „Router_chain“

Wie in Abbildung 62 zu sehen ist, beginnt der Prompt damit, dass die Aufgabe kurz und knapp beschrieben wird. Anschließend wird das gewünschte JSON-Format beschrieben. Dieses besteht aus zwei Key/Value Paaren. Zuerst wird die „destination“ angegeben, dies ist die Bezeichnung der als Nächstes auszuführenden Chain (also in diesem Fall „COMMAND“, „INQUERY“ oder „DEFAULT“). Danach wird unter „next_input“ die Eingabe für die nächste Chain definiert, dabei sollte „next_input“ in diesem Fall immer eine exakte Kopie der Nutzereingabe sein. Anschließend werden die verschiedenen Optionen für die Ziel-Chains unter „DESTINATION OPTIONS“ angegeben, sowie jeweils eine Erklärung, wann die Option gewählt werden sollte, aufgelistet. Dabei wird im Prompt-Template allerdings nur die „DEFAULT“ Option fest angegeben, alle anderen Optionen („COMMAND“ und „INQUIRY“) werden über die Variable „router_desination_options“ später eingefügt, was es vereinfacht, neue Optionen zu implementieren. Zuletzt folgen eine Reihe an Beispielen und der Nutzerinput, den das LLM bearbeiten soll (wobei in der Abbildung nur ein Beispiel dargestellt wird, im realen Prompt-Template ist allerdings eine größere Anzahl vorhanden). Damit handelt es sich in diesem Fall um einen Few-Shot-Prompt (zuvor in Kapitel „2.2. Grundlegende Fachbegriffe“ erklärt). Das macht die Antworten des LLMs auf diesen Prompt starrer und Fehler reduziert.

4. Praxisteil

Als Nächstes wird der Command-Prompt beschrieben, welcher ausgeführt wird, wenn sich die Router-Chain für die Option „COMMAND“ entscheidet.

```
### Instruction:
Below is a "USER REQUEST" asking to make a change to a smart home device.
Respond in JSON format as defined under "JSON STRUCTURE". All available devices
and their alias are defined in the "DEVICE ALIAS LIST" in the format "Alias:
device_id", choose the best fitting device from this list or from the
"HISTORY". Also choose a "category" from the "CATEGORY LIST" and set the
"action" according to its description in the "CATEGORY LIST". You will also
find a "HISTORY" for the last "USER REQUESTS". If the user is referencing a
previous request, use the appropriate devices from the "HISTORY" instead of the
"DEVICE ALIAS LIST". The "type" should always be "command".

JSON STRUCTURE: {"type": "command", "commands": [{"device": device_id,
"category": category, "action": action}]}}}

CATEGORY LIST (category: action):
"power": "on" or "off",
"dim": percentage as integer,
"color": color name as string

DEVICE ALIAS LIST (alias: device_id):
{device_list}

REMEMBER: There might be grammatical errors in the device_id, make sure to copy
them exactly including any errors.

HISTORY:
{chat_history}

USER REQUEST: "{input}"
### Response:
```

Abbildung 63: Prompt-Template der "Command"-Chain

Der grundlegende Aufbau ist ähnlich, wie zuvor beim Router-Prompt. Allerdings mit dem großen Unterschied, dass es sich um einen Zero-Shot-Prompt (zuvor in Kapitel „2.2. Grundlegende Fachbegriffe“ erklärt) handelt, da es keine Beispiele gibt. Dies kann zwar negative Effekte auf die Ausgabequalität haben, allerdings haben Tests während der Planung dieses Prompts gezeigt, dass die Verwendung von Beispielen ebenfalls zu reproduzierbaren Problemen führte und sich das LLM dabei oft zu stark an die Beispiele gehalten hat. Diese Starre kann gerade bei komplexen Aufgaben, wie dieser hier, zu Problemen führen. Ein weiterer wichtiger Unterschied ist die Komplexität der Aufgabe. Die JSON-Struktur ist komplexer und beinhaltet mehr Key/Value Paare. Zuerst wird der „type“ angegeben, dieser sollte immer „command“ sein und dient dazu später zu identifizieren, welcher Prompt die Ausgabe erzeugt hat. Unter dem „commands“ Key sind schließlich die tatsächlichen Kommandos zu finden. Dabei handelt es sich um ein JSON-Array, was es dem LLM ermöglicht, mehrere Befehle gleichzeitig zu identifizieren. Jeder einzelne Befehl besteht dann aus einer „device_id“ (das Gerät), einer „category“ (welche Art von Aktion ausgeführt werden soll) und einer „action“ (was genau getan werden soll). Dabei werden alle Kategorien und Aktionen in der „CATEGORY LIST“ beschrieben, während die verfügbaren Geräte in der „DEVICE ALIAS LIST“ aufgeführt und über die Variable „device_list“ eingefügt werden. Diese werden dabei im Format „alias: device_id“ dargestellt, wobei die „device_id“ die Bezeichnung des Geräts in HomeAssistant ist und der „alias“ ein Name für das Gerät. Das könnte zum Beispiel so aussehen: „Floor light: light.floor“.

4. Praxisteil

Auch die „REMEMBER“ Zeile ist neu, während der Tests mit dem Prompt zeigte sich, dass das LLM Probleme damit hat, die „device_id“ korrekt zu übernehmen, insbesondere wenn diese von der Norm abweicht. Die „REMEMBER“ Zeile soll dabei helfen, dem LLM klarzumachen, dass es die „device_id“ korrekt übernehmen soll. Schließlich folgt dann die „HISTORY“ und die Nutzereingabe.

Das Prompt-Template für die Inquiry-Chain ist vom Aufbau identisch zum Command-Prompt, nur mit Anpassungen für die leicht abgeänderte Aufgabe.

```
### Instruction:
Below is a "USER REQUEST" asking to make get information about a smart home
device. Respond in JSON format as defined under "JSON STRUCTURE". All available
devices and their alias are defined in the "DEVICE ALIAS LIST" in the format
"Alias: device_id", choose the best fitting device. Also choose an "attribute"
from the "ATTRIBUTE LIST" that the user is asking about. If the user asks about
multiple attributes or devices, list all devices and attributes. You will also
find a "HISTORY" for the last "USER REQUESTS". The "type" should always be
"inquiry".

JSON STRUCTURE: {"type": "inquiry", "commands": [{"device": device_id,
"attribute": string}]}}

ATTRIBUTE LIST: state,
brightness,
color

DEVICE ALIAS LIST:
{device_list}

REMEMBER: Use the exact device Ids from the "DEVICE ALIAS LIST" or from
"HISTORY". There might be grammatical errors in the Ids, so make sure to copy
the Ids exactly including any errors.

HISTORY:
{chat_history}

USER REQUEST: "{input}"
### RESPONSE:
```

Abbildung 64: Prompt-Template für die Inquiry-Chain

Wie zu erkennen ist, sind die beiden Prompts sich sehr ähnlich. Dabei haben beide Prompts aber auch das gleiche Problem: das Einfügen der Geräteliste. In einem Smart-Home-System können sich schnell große Mengen an Geräten befinden. Selbst in der kleinen lokalen Testumgebung, die für das Testen der Applikation verwendet wird, gibt es etwa 20 Geräte. In einer größeren Umgebung können es schnell über 100 sein. Bei jedem Durchlauf der Inquiry- oder Command-Chain jeweils eine vollständige Geräteliste einzufügen, wäre wenig zielführend. Dies würde unnötig den Kontext weiter verlängern und damit die Fehlerchance erhöhen. Außerdem wäre auch die Chance hoch, dass sich das LLM für ein falsches Gerät entscheidet. Dieses Problem kann gelöst werden, indem das zuvor bereits besprochene Konzept von RAG verwendet wird. Die Geräteliste wird in eine Vektor-Datenbank geladen (in diesem Fall wieder ChromaDB) und dann jeweils nur die relevantesten Geräte in den Prompt eingefügt. So wird RAG eingesetzt, um dem LLM die Arbeit zu erleichtern und die Kontextgröße zu reduzieren. Der zusätzliche Hardwareaufwand durch die Vektor-Datenbank wird von den kleineren Prompts ausgeglichen und ist vernachlässigbar.

4. Praxisteil

Damit wurden alle Prompts der ersten Gruppe dargestellt. Es folgen dementsprechend die Prompts der zweiten Gruppe. Diese sind dabei einfacher als die bisherigen Prompts, da keine komplexe JSON-Struktur beschrieben werden muss.

Das „Default“-Prompt stellt einen einfachen Chat-Prompt dar, welcher generelle Konversationen führen soll und sieht deshalb entsprechend einfach aus:

```
### Instruction:
You are a helpful and friendly AI assistant connected to a smart home system.
Your name is {AI_NAME}. The user has just said something has nothing to do with
the smart home system itself.

CONVERSATION HISTORY:
{chat_history}

USER: {input}
AI ASSISTANT ###RESPONSE:
```

Abbildung 65: Prompt-Template für die Default-Chain

Es ist zu sehen, dass dem LLM nur eine kurze Beschreibung der Situation gegeben und die Chat-History übermittelt wird. Dies reicht bereits für diese vergleichsweise einfache Aufgabe. Auch hierbei handelt es sich wieder um einen Zero-Shot-Prompt.

Zuletzt gilt es den Response-Prompt zu planen. Dieser ist etwas komplexer als der Default-Prompt, insgesamt aber dennoch sehr einfach gehalten:

```
### Instruction:
You are a helpfull and friendly assistant connected to a smart home system.
Under "SMART HOME SYSTEM:" you are given information from the smart home
system. You should create a 1 to 2 sentence reply telling the user what the
smart home did.

USER REQUEST:
Is the livingroom light on?
SMART HOME SYSTEM:
[{'DEVICE': 'light.living_room', 'ATTRIBUTE': 'state', 'STATE': 'off'}]
ASSISTANT RESPONSE:
The livingroom light is currently off.

USER REQUEST:
{user_request}
SMART HOME SYSTEM:
{smart_home_system}
ASSISTANT ### RESPONSE:
```

Abbildung 66: Prompt-Template für die Response-Chain

Beim Response-Prompt kommt erstmals wieder ein Few-Shot-Prompt zum Einsatz, bei dem Beispiele angegeben werden (in dem Fall ist wieder nur ein Beispiel dargestellt, im echten Prompt wären es allerdings mehrere). Wie auch im Beispiel zu erkennen ist, wird aus einem JSON-Objekt, das aus dem Code stammt, eine natürliche Antwort für den Nutzer konstruiert.

4. Praxisteil

4.1.7. LLM Hyper-Parameter

Der letzte Aspekt, welcher vorbereitet werden muss, sind die Hyper-Parameter (zuvor in Kapitel „2.2. Grundlegende Fachbegriffe“ erklärt), mit denen das LLM in der Applikation betrieben werden soll. Es ist möglich, dass sich während der Entwicklung der Applikation herausstellt, dass einige der Hyper-Parameter weiter angepasst werden müssen. Trotzdem ist es wichtig, die voraussichtlichen Hyper-Parameter zu dokumentieren und eine realistische Starteinstellung zu planen.

Dazu ist es wichtig zu wissen, was die Hyper-Parameter, die hier angepasst wurden, bedeuten und welche Aspekte des LLMs diese kontrollieren.

Top-k: Der Top-k Parameter bestimmt, wie viele mögliche Optionen vom LLM bei der Wahl des nächsten Tokens berücksichtigt werden sollen. Wird dieser Parameter zum Beispiel auf 3 gesetzt, werden nur die 3 wahrscheinlichsten Optionen für den nächsten Token in Betracht gezogen. Angenommen, folgende Optionen stehen für den nächsten Token zur Verfügung:

1. „Blau“ – Wahrscheinlichkeit: 0,5
2. „Grün“ – Wahrscheinlichkeit: 0,4
3. „Gelb“ – Wahrscheinlichkeit: 0,3
4. „Rot“ – Wahrscheinlichkeit: 0,2

Bei einem Top-k von 3 würden vom Modell nur „Blau“, „Grün“ und „Gelb“ in Betracht gezogen werden und „Rot“ würde als Option für das nächste Token wegfallen. [86]

Für die Applikation wird dieser Wert auf 10 gesetzt, was eine relativ starke Einschränkung der Optionen darstellt.

Top-p: Der Top-p Parameter verhält sich im Grunde gleich wie der Top-k Parameter, wobei der begrenzende Faktor nicht eine feste Anzahl an Optionen ist, sondern die kumulative Wahrscheinlichkeit der Optionen für den nächsten Token. Angenommen es steht wieder die gleiche Liste an Optionen zur Verfügung, wie im Top-k Beispiel, und Top-p wird auf 0,95 gesetzt. Es würden nur „Blau“ und „Grün“ in Betracht gezogen werden, während „Gelb“ sowie „Rot“ wegfallen würden, da $0,5 + 0,4 = 0,9 < 0,95$, aber $0,5 + 0,4 + 0,3 = 1,2 > 0,95$. [86]

In der Applikation soll dieser Wert vorerst auf 0,95 gesetzt werden. Auch dies ist eine relativ starke Einschränkung der Optionen.

Temperature: Dieser Parameter kontrolliert die „Kreativität“ der LLM-Antworten. Er beeinflusst die Wahrscheinlichkeitsverteilung der Optionen. LLMs arbeiten normalerweise nicht deterministisch, da die Wahrscheinlichkeitsverteilung mithilfe der Temperature gewichtet wird. Ein sehr hoher Wert (z. B. 1), führt dazu, dass auch Optionen mit sehr geringen Wahrscheinlichkeiten ausgewählt werden. Dabei kann ein sehr hoher Wert deshalb auch zu unverständlichem Text führen. Je niedriger der Wert ist, desto weniger zufällig ist die Auswahl des nächsten Tokens. Bei einem Wert von 0 verhält sich ein LLM deterministisch. [86]

Die Temperature wird in der Applikation auf 0,1 gesetzt, was erneut gering ist und das LLM relativ stark in seiner „Kreativität“ einschränkt.

4. Praxisteil

Repetition-penalty: Dieser Parameter ist im Grunde selbsterklärend, er soll verhindern, dass sich das LLM ständig wiederholt. Zu diesem Zweck werden die Optionen für den nächsten Token danach gewichtet, welche Token zuvor bereits ausgegeben wurden, wobei wiederholende Token benachteiligt werden.

Um unnötige Wiederholungen zu verhindern, wird der Wert für die Applikation auf 1,1 gesetzt.

Stop-sequences: Auch der Stop-sequences Parameter ist schon vom Namen ableitbar, es handelt sich um bestimmte Sequenzen (also Text), bei welchen die Ausgabe des LLMs frühzeitig gestoppt werden soll.

Basierend auf den dargestellten Prompt-Templates werden folgende Sequenzen als Stop-sequences definiert: „USER REQUEST“, „USER“ und „###“.

4. Praxisteil

4.2. Dokumentation der Umsetzung

Die Planung der Applikation wurde dargestellt, damit folgt die Dokumentation der Umsetzung. Zu diesem Zweck sollen zuerst einige besonders relevante Aspekte der Planung noch einmal aufgegriffen werden und dargestellt werden, wie die Planung umgesetzt wurde. Außerdem soll die fertige Applikation vorgestellt werden, indem verschiedene Anwendungsbeispiele aufgezeigt und diskutiert werden. Im vorletzten Unterkapitel wird anschließend die Hardwareseite der Applikation untersucht, welche zuvor bei den Tests der verschiedenen Modelle bereits angeschnitten wurde. Schließlich werden die aufgetretenen Probleme und ihre Lösungen dargestellt.

4.2.1. Beschreibung von Details

In diesem Kapitel soll die Umsetzung der zuvor geplanten Aspekte beschrieben werden, indem die Grafiken zum Beispiel um ein Klassendiagramm erweitert werden, welches zeigt, wie die Grobplanung aus Abbildung 60 implementiert wurde. Diese Klassendiagramm wird zuerst betrachtet, wobei zu bedenken ist, dass verschiedene Aspekte der Applikation (insbesondere jene, die mit dem LangChain zu tun haben) nicht in diesen Klassen, sondern in Funktionen implementiert sind. Das Klassendiagramm zeigt damit also primär den Aufbau des Codes, welcher von der „Route_for_execution_chain“ aufgerufen wird.

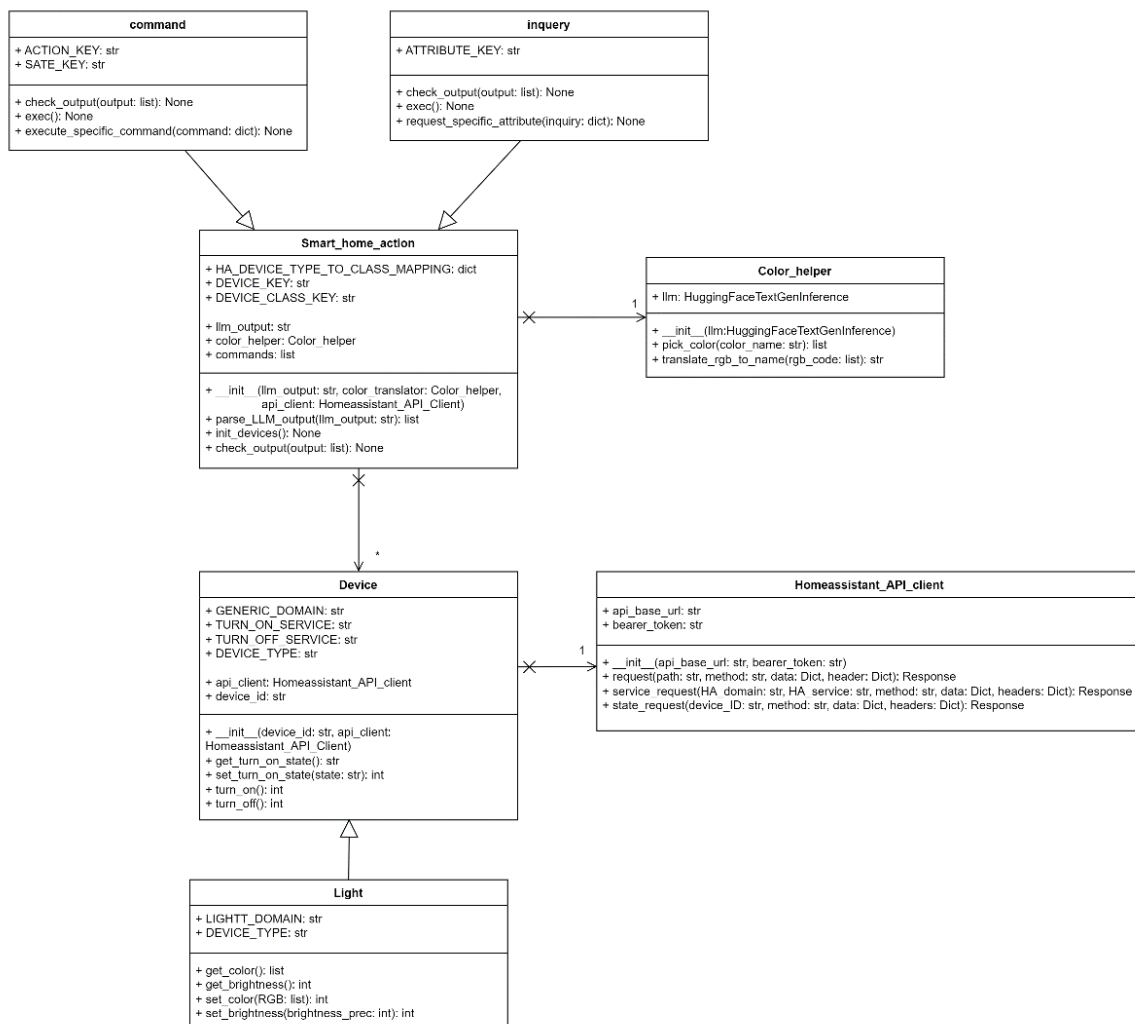


Abbildung 67: Klassendiagramm der Applikation (eigene Grafik)

4. Praxisteil

Grundlegend ist die gleiche Struktur wie in der Grobplanung zu sehen, allerdings mit leichten Anpassungen. Als Einstiegspunkt in den Code dienen nach wie vor die „command“ und „inquiry“ Klassen, diese erben von einer dritten Klasse „Smart_home_action“. Dies hat den Vorteil Codedopplung zu reduzieren sowie es zu vereinfachen, die Applikation in der Zukunft um weitere Befehle zu erweitern, welche dann ebenfalls von dieser Klasse erben. Anhand der Methoden ist zu sehen, dass die Klassen, wie zuvor geplant, die LLM-Ausgabe parsen und in eine Liste „commands“ füllen. Dies geschieht normalerweise automatisch im Konstruktor. Am Ende wird die „exec“ Methode ausgeführt, welche über die „commands“ Liste iteriert und alle Kommandos ausführt. Dabei ist es für das Verständnis hilfreich zu wissen, wie die „commands“-Liste vor der Ausführung eines Kommandos aussieht:

```
[{"device": "light.livingroom_bulb_1", "category": "power", "action": "on",  
"device_class": light}]
```

Abbildung 68: Beispiel für die „commands“-Liste

Wie zu sehen ist, folgt die Liste den in den Prompts definierten JSON-Strukturen, allerdings ergänzt um einen „device_class“ Key, welcher eine Instanz einer „Device“-Klasse hält. Die „Device“-Klasse war auch bereits in der Planung zu sehen, sie stellt jeweils ein konkretes Gerät dar, in diesem Beispiel das Gerät „light.livingroom_bulb_1“. Dabei ist dieses Gerät konkret eine Instanz der „Light“-Klasse, welche von „Device“ erbt. Dabei kann ein „Smart_home_action“-Objekt beliebig viele „Device“-Objekte halten. Die „Device“ Klasse (bzw. Klassen, die von ihr erben) hält dann wiederum genau eine Instanz der „Homeassistant_API_client“-Klasse, welche schließlich die API-Calls durchführt. Diese Struktur ist vergleichbar mit der vorherigen Grobarchitektur, wobei eine Klasse „Color_helper“ existiert, welche nicht Teil der Grobarchitektur war. Die „Color_helper“-Klasse löst ein konkretes Problem, welches während der Entwicklung auftrat. HomeAssistant arbeitet mit RGB-Codes wie „[255, 255, 255]“ für weiß. Der Nutzer möchte nicht sagen „Set the lamp to 255, 255, 255.“, sondern „Set the lamp to white.“. Umgekehrt gilt das gleiche, wenn der Nutzer fragt „What color is the light?“ sollte die Applikation mit „The light is white.“ antworten. Es muss also zwischen Farbnamen und RGB-Codes übersetzt werden. Dafür ist die „Color_helper“-Klasse zuständig. Dabei ist die Funktionsweise der Übersetzung deutlich unterschiedlich, je nachdem in welche Richtung übersetzt wird. Die „pick_color“-Methode übersetzt einen Farbnamen in einen RGB-Code. Dies ist besonders komplex, weil der Nutzer eine beliebige Farbe nennen kann. Werden die Farboptionen eingeschränkt, könnte mit einem einfachen Dict von Namen und RGB-Codes gearbeitet werden. Das beschränkt aber deutlich die Farbauswahl. Deshalb wird ein LLM für diese Aufgabe eingesetzt. Damit kann der Nutzer eine beliebige Farbe angeben und das LLM versucht daraus einen passenden RGB-Code zu erzeugen. Dadurch kann der Nutzer auch deutlich abstraktere Farbnamen verwenden. So hat die Methode keine Probleme, aus folgender Eingabe einen RGB-Code zu erzeugen: „Set the light to a christmassy color.“ Das LLM macht daraus einen vorwiegend roten RGB-Code. Der Nachteil dieses Ansatzes ist es, dass Farben schwer reproduzierbar sind, weil LLMs in der Regel nicht deterministisch agieren. Sagt der Nutzer zweimal „Set the light to blue.“, werde dabei gegebenenfalls zwei verschiedene Blautöne erzeugt. Die Methode „translate_rgb_to_name“ übernimmt genau die gegenteilige Aufgabe. Diese arbeitet allerdings intern nicht mit einem LLM. Bei Tests hat sich herausgestellt, dass selbst sehr leistungsstarke LLMs Schwierigkeiten zu haben scheinen, RGB-Codes in Farbnamen zu übersetzen. Deshalb wird ein einfacherer Ansatz gewählt. Die

4. Praxisteil

„webcolors“ Bibliothek [87] enthält eine große Liste von RGB-Codes und ihren Namen. Es wird über diese List iteriert und der gesuchte RGB-Code mit den Codes aus der Liste verglichen, dabei wird jeweils gespeichert, welcher RGB-Code dem gesuchten am nächsten ist. Der ausgewählte Farbname ist damit am Ende nur approximativ.

Zudem wurde eine hier nicht näher dargestellte „Printer“-Klasse implementiert, welche als einfacher Logger fungiert und 4 verschiedene Level bietet. Level 0: Message, Level 1: Info, Level 2: Debug, Level 3: Debug + LangChain Debug.

Nachdem die Implementation der verschiedenen Klassen dargestellt wurde, gilt es die Implementierung der LLM-Architektur zu beschreiben. Dabei ändert sich grundlegend nichts an der geplanten Struktur. Dennoch gibt es einige Details in der Implementierung, welche veranschaulicht werden sollen. So gibt es etwa eine Besonderheit bei der Router-Chain. Wie in der Prompt-Planung dargestellt wurde, wird für die Router-Chain ein eigener Prompt verwendet. Dies unterscheidet sich von dem Standardansatz, der zuvor bei der Vorstellung von LangChain aufgezeigt wurde, dort wurde ein mitgeliefertes Prompt-Template verwendet. Das Verwenden eines eigenen angepassten Prompt-Templates für die Router-Chain erhöht die Zuverlässigkeit des Routers. Das Custom-Template bedeutet auch, dass ein eigener Output-Parser für das Template implementiert werden muss, da sich dieses Template nicht an die gleiche Ausgabestruktur, wie das Standardtemplate, hält. Dementsprechend wurde für die Router-Chain ein eigenes Template sowie ein eigener Parser implementiert.

Auch die Umsetzung der Vektor-Datenbank und RAG soll erneut aufgegriffen werden. Es wurde sich für die manuellere Version entschieden, bei der nicht LangChains Wrapper verwendet werden, sondern die Vektor-Datenbank manuell eingebaut wird, wobei die Wahl wieder auf ChromaDB fiel. Durch die manuelle Implementierung wird mehr Kontrolle über die Vektor-Datenbank erhalten und alle Features der originalen ChromaDB Bibliothek (welche nicht vollständig im LangChains Wrapper vorhanden sind), werden beibehalten. Der Ablauf ist damit sehr ähnlich wie der in Kapitel „3.6. Retrieval Argumented Generation - “ beschrieben. Die Ergebnisse der Vektor-Datenbank-Suche werden nicht bei der Ausführung der Chain als Übergabeparameter angegeben, dies wäre aufgrund der LLM-Architektur nicht ohne weiteres möglich, sondern mithilfe der „partial“ Methode bei der Erstellung der Prompt-Template-Objekte in den Prompt eingefügt. Das bedeutet, dass die Vektor-Datenbank-Suche bereits vor dem Erstellen der Prompt-Template-Objekte durchgeführt werden muss, was wiederum dazu führt, dass diese mit jeder Ausführung der „Run“ Funktion neu instanziiert werden müssen. Der dadurch erzeugte zusätzliche Hardwareaufwand ist vernachlässigbar. In diesem Fall wurde entschieden, jeweils die 10 besten Ergebnisse aus der Vektor-Datenbank zu verwenden. Damit stehen dem LLM jeweils 10 Geräte zur Verfügung. Eine Erhöhung dieser Anzahl würde die Applikation zwar noch etwas flexibler machen, aber auch die Kontextlänge weiter erhöhen. 10 Geräte stellen einen soliden Kompromiss aus Flexibilität und Kontextlänge dar. Insbesondere bei einer Smart-Home-Installation mit sehr vielen Geräte (>100), könnte es aber auch Sinn ergeben, diese Zahl zu erhöhen.

Als Einstiegspunkt für die Applikation fungiert schließlich eine einfache Funktion „continuous_mode“ welche die Funktion „create_memory“ zum Erstellen der Memory-Objekte und „create_vector_db“ zum Erstellen der Vektor-Datenbank ausführt, und dann eine endlose Schleife startet, in der immer wieder ein Nutzerinput von der Konsole gelesen

4. Praxisteil

und die „run“ Funktion aufgerufen wird. Diese Schleife endet, wenn die Applikation die spezifische Nutzereingabe „stop“ erkennt.

4.2.2. Demonstration der Funktionsweise

In diesem Kapitel wird die Funktionsweise aus Nutzersicht, anhand einiger Beispiele, dargestellt. Dazu werden vier verschiedene Beispielinteraktionen mit der Applikation aufgezeigt, welche jeweils unterschiedliche Funktionen der Applikation darstellen. Da für die Applikation keine GUI entwickelt wurde, finden sämtliche Interaktionen auf der Konsole statt. Die Applikation ist in diesem Fall mit einer lokalen HomeAssistant-Instanz verbunden, welche die Kontrolle über knapp 20 Geräte hat. Da es dem Leser nicht möglich ist, die Veränderung selbst zu sehen, sei erwähnt, dass (wenn nicht konkret anders angegeben) alle Aktionen korrekt durchgeführt wurden und die Geräte entsprechend reagiert haben. Alle Beispiele bestehen jeweils immer aus einem „User“ (die Nutzereingabe) und einem „Assistant“ (die Antwort der Applikation), dabei enden die Beispiele jeweils, wie zuvor beschrieben, mit der Nutzereingabe „stop“. Alle Beispiele wurden mit dem 13B Modell durchgeführt. Zu Beginn wird ein erstes einfaches Beispiel betrachtet, welches eine Standardinteraktion mit der Applikation darstellt:

```
User: Turn on the light in the floor
Assistant: I turned on the light in the floor
User: Now set it to green
Assistant: I set the light in the floor to green
User: Great, please also dim it to 50%
Assistant: I dimmed the light in the floor to 50%.
User: Now turn it off!
Assistant: I turned off the light in the floor
User: stop
```

Abbildung 69: Anwendungsbeispiel 1, Standardinteraktion

Wie zu sehen ist, handelt es sich um eine einfache Interaktion, in welcher der Nutzer den Assistenten dazu auffordert, das Licht im Flur einzuschalten, was auch entsprechend von der Applikation bestätigt wird. Anschließend wird Konversationshistorie genutzt, um die Farbe des Lichts auf Grün zu setzen und danach auf 50% Helligkeit zu dimmen. Obwohl der Nutzer keine Geräte konkret nennt, erkennt die Applikation korrekt, dass der Nutzer über das zuvor genannte Flurlicht spricht und führt die Anfragen korrekt durch. Zuletzt möchte der Nutzer das Licht wieder ausschalten und auch dabei erkennt die Applikation wieder korrekt, welches Gerät gemeint ist. Das Beispiel zeigt die zuvor beschriebene Interaktionshistorie in Aktion.

4. Praxisteil

Im folgenden zweiten Beispiel wird dargestellt, wie der Nutzer Anfragen stellen kann und diese durch die geteilte Historie auch mit Kommandos verbindet. Zudem tritt ein erstes Beispiel für eine Eingabe mit mehreren Befehlen auf:

User: *Is the Livingroom light on?*
Assistant: *The Livingroom light is currently off.*
User: *Ok, then turn it on please!*
Assistant: *I turned on the light in the Livingroom*
User: *Set it to a calming color*
Assistant: *I set the light in the Livingroom to blue*
User: *Oh, I like that color, I don't think its blue though. What color is the light?*
Assistant: *The light in the Livingroom is hot pink.*
User: *I'll go to the bedroom now, please turn off the Livingroom light and turn on the Bedroom light.*
Assistant: *I turned off the light in the Livingroom and turned on the light in the bedroom.*
User: *stop*

Abbildung 70: Anwendungsbeispiel 2, Inquiry/Command Mix und Multi-Action

Das Beispiel beginnt mit einer einfachen Abfrage des Status vom Wohnzimmerlicht. Der Assistent antwortet korrekt, dass das Licht ausgeschaltet ist. Daraufhin wird wieder die, zwischen „Inquiry“- und „Command“-Chain geteilte, Historie genutzt, indem ein Befehl abgesetzt wird, welcher sich auf die vorherige Anfrage bezieht, und auch hier erkennt die Applikation sofort, welches Gerät gemeint ist. Folgend wird die Möglichkeit der Applikation demonstriert, auch abstrakte Farbbezeichnung zu verarbeiten, indem der Assistent aufgefordert wird, das Licht auf eine „beruhigende“ Farbe zu setzen. Die Applikation macht daraus einen Farbton, der subjektiv als Violett oder Pink einzuordnen ist. Der RGB-Code der gewählten Farbe war [255, 105, 170]. Ob dies der konkrete Farbton ist, den der Nutzer gemeint hat, ist natürlich nicht klar, allerdings wäre es selbst für einen Menschen schwierig, mit einer solchen abstrakten Anweisung korrekt umzugehen. Subjektiv hat das Modell mit dem Farbton eine solide Entscheidung getroffen. Es gibt in der Antwort der Applikation jedoch eine Ungenauigkeit. Die Applikation gibt an, das Licht auf Blau gesetzt zu haben. In der folgenden Abbildung sind beispielhafte Fotos, der vom Modell ausgewählten Farbe ([255, 105, 170]) im Vergleich mit einer blauen Einstellung ([0, 0, 255]), zu sehen.

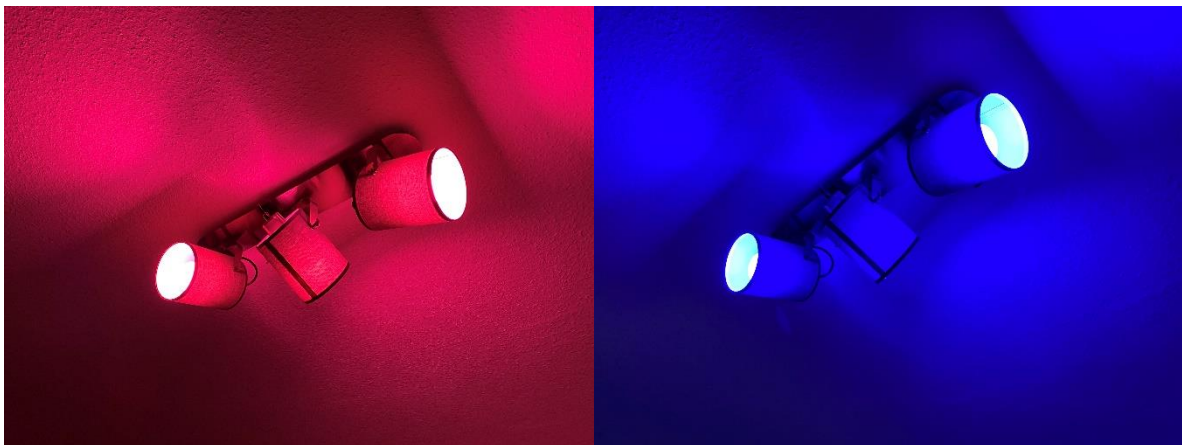


Abbildung 71: Vergleich der vom Modell gewählten Farbe (links) mit einer blauen Einstellung (rechts). (eigene Abbildung)

Unabhängig davon, wie die Farbe korrekt zu bezeichnen ist, handelt es sich nicht um Blau. Deshalb fragt der Nutzer noch mal nach, welche Farbe das Licht hat. Die Applikation geht mit der Situation gut um und beantwortet die Frage. Die Farbbezeichnung „hot pink“ ist deutlich zutreffender.

4. Praxisteil

Anschließend wird ein erster einfacher Multi-Aktions-Befehl ausgeführt, bei dem der Nutzer in einer Anfrage gleichzeitig das Licht im Wohnzimmer ausschaltet und das Licht im Schlafzimmer einschaltet. Auch dies setzt die Applikation korrekt um.

Die Multi-Aktions-Befehle können auch deutlich komplexer sein, wie im folgenden Beispiel zu sehen ist:

User: Turn on the Livingroom light. Then set the front Livingroom bulb to green, the middle one to blue and the back one to orange. Also turn on the TV.
Assistant: I turned on the Livingroom light. I also set the front Livingroom bulb to green, the middle one to blue and the back one to orange. I also turned on the TV.
User: stop

Abbildung 72: Anwendungsbeispiel 3, Komplexe Multi-Action

In einer einzigen Nutzereingabe werden fünf Aktionen veranlasst.

1. Das Wohnzimmerlicht einschalten
2. Die vordere Wohnzimmerglühbirne auf Grün setzen
3. Die mittlere Wohnzimmerglühbirne auf Blau setzen
4. Die hintere Wohnzimmerglühbirne auf Gelb setzen
5. Den Fernseher einschalten

Auch diese komplexe Aufgabe wird von der Applikation korrekt erkannt und umgesetzt, alle fünf Aufgaben wurden erfüllt.

Damit wurde die Hauptfunktionalität in Bezug auf das Smart-Home-System dargestellt, abseits dieser besteht noch die Möglichkeit, die Applikation wie einen LLM-Chat-Bot zu verwenden. Im folgenden Beispiel wird eine kurze Konversation mit der Applikation aufgezeigt, in welcher der Nutzer ein Problem bei der Lösung einer einfachen Programmieraufgabe hat und den Assistenten dabei um Hilfe bittet. Dabei wird wieder die Konversationshistorie genutzt, die in diesem Fall unabhängig von der Aktions-Konversationshistorie ist.

User: hey, what is your name?
Assistant: Hey! My name is HAL. I'm an AI assistant connected to a smart home system.
User: I am having a hard time with this python problem
Assistant: Sure, I can help you with that. What is the problem exactly?
User: I am supposed to write a function that returns the fibonacci sequence. But i keep getting it wrong.
Assistant: Sure, I can help you with that. Let me explain how the Fibonacci sequence works. The Fibonacci sequence starts from 0 and 1, then each subsequent number is the sum of the previous two numbers. So the first few terms are 0, 1, 1, 2, 3, 5, 8, 13, etc. To generate the Fibonacci sequence in Python, you can use the following code: `def fib(n): if n <= 1: return n else: return fib(n - 1) + fib(n - 2)`
Now let's try some examples:
`fib(4)` 4 `fib(7)` 13 `fib(9)` 21 `fib(10)` 34
User: stop

Abbildung 73: Anwendungsbeispiel 4, LLM-Chat-Bot

Auch hier reagiert die Applikation korrekt und liefert schließlich funktionierenden Programmcode, wobei die Formatierung der Antwort nicht optimal ist. Die gestellte Aufgabe ist in diesem Fall vergleichsweise einfach, allerdings war auch nicht das Ziel eine Applikation zu entwerfen, welche Softwareentwicklungsaufgaben löst. Im Grunde ist der Chatbot nur eine Zugabe zur Hauptanwendung und als solche ist die gegebene Antwort subjektiv gut.

4. Praxisteil

Damit wurde die Funktionalität der Applikation aus Nutzersicht umfassend präsentiert. Dabei soll kein falscher Eindruck vermittelt werden, was die Zuverlässigkeit der Applikation angeht. Die gezeigten Beispiele sind alle unverändert und waren jeweils der erste Versuch, das jeweilige Beispiel zu erzeugen. Die Applikation macht, je nach Komplexität der Aufgabe, stellenweise auch Fehler. Fehler treten insbesondere im Router auf, der nicht immer korrekt erkennt, an welche Chain weitergeleitet werden muss. Außerdem treten Fehler besonders dann auf, wenn in komplexer Art und Weise Bezug auf die Historie genommen wird. Dabei ist die Zuverlässigkeit insgesamt gut, insbesondere wenn bedacht wird, dass kein aufwendiges Training von LLMs stattgefunden hat und die LLMs jeweils auf Zero-Shot- oder Few-Shot-Basis arbeiten. Zudem wird die Leistung auch von der Leistung des LLMs beeinflusst, so ist insbesondere beim Wechsel vom 7B auf das 13B Modell, ein merkbarer Anstieg der Zuverlässigkeit zu beobachten.

4.2.3. Darstellung der Hardwareseite und Performance

Eines der wohl größten Probleme von LLMs, welches es noch zu lösen gilt, ist die enorme benötigte Hardware, dieses Problem betrifft auch die entwickelte Applikation. Daher ist es relevant, die Hardware und Performance der Applikation zu betrachten. Zuerst ist es wichtig, die Hardware-Struktur zu verstehen, auf der die Applikation entwickelt und ausgeführt wurde, zu diesem Zweck wurde folgendes Verteilungsdiagramm entworfen:

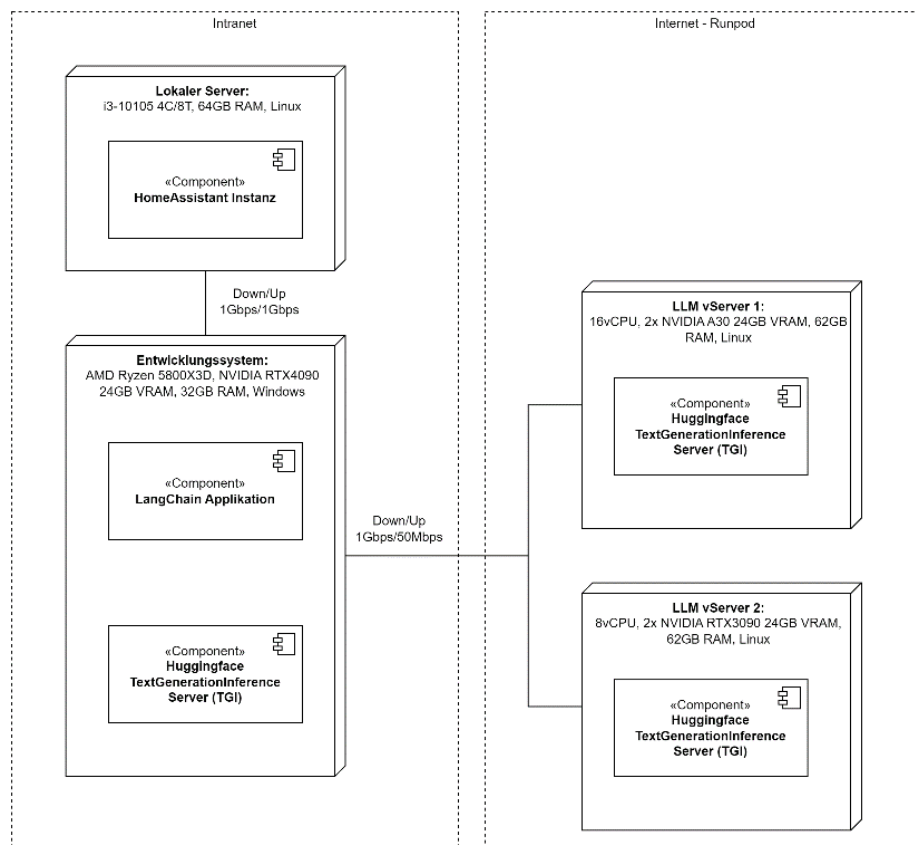


Abbildung 74: Verteilungsdiagramm der Entwicklungsumgebung (eigene Grafik)

Es gibt insgesamt 4 verschiedene Systeme. Am wichtigsten für die Entwicklung waren der lokale Server, auf dem die HomeAssistant-Instanz ausgeführt wird, sowie das Entwicklungssystem auf dem die Applikation als auch eine lokale TGI-Instanz ausgeführt

4. Praxisteil

wurde. Diese lokale TGI-Instanz verwendet die 7B Version Platypus, da der VRAM für die 13B Variante nicht ausreicht. Die lokale TGI-Instanz wurde hauptsächlich während der Entwicklung verwendet, um die Kosten für die Mietung von Cloud-Hardware zu reduzieren. Wiederholt wurde aber auch auf einen von zwei zusätzlichen externen virtuellen Servern zugegriffen, um die Leistung der Applikation mit dem größeren 13B Modell zu testen. Diese virtuellen Server wurde beim Cloudanbieter „Runpod.io“ [28] gemietet. Im Diagramm sind zwei verschiedene „LLM vServer“ zu sehen, wobei jeweils nur einer dieser Server gleichzeitig verwendet wurde. Welcher der beiden „LLM vServer“ genutzt wurde, hing in erster Linie von der Hardwareverfügbarkeit zum jeweiligen Zeitpunkt ab. Die Performance zwischen den Systemen unterscheidet sich. Um einen Eindruck zu vermitteln, welche Leistung von der Applikation auf unterschiedlicher Hardware zu erwarten ist, wurden Tests durchgeführt, welche die Ausführungszeit eines einzelnen Befehls messen.

Dafür wurden 3 verschiedene Befehle mit unterschiedlicher Komplexität genutzt:

1. „Turn on the patio light.“ – einfacher Befehl (3 LLM-Anfragen)
2. „Set the patio light to green.“ – Befehl mit Farbänderung (4 LLM-Anfragen)
3. „Set the patio light to blue, the office to green and the basement to yellow.“ – komplexer Befehl mit 3x Farbänderung (6 LLM-Anfragen)

Für den Benchmark wurden diese 3 Befehle jeweils 32-mal ausgeführt und ihre Ausführungszeit gemessen. Es ergaben sich folgende Ergebnisse:

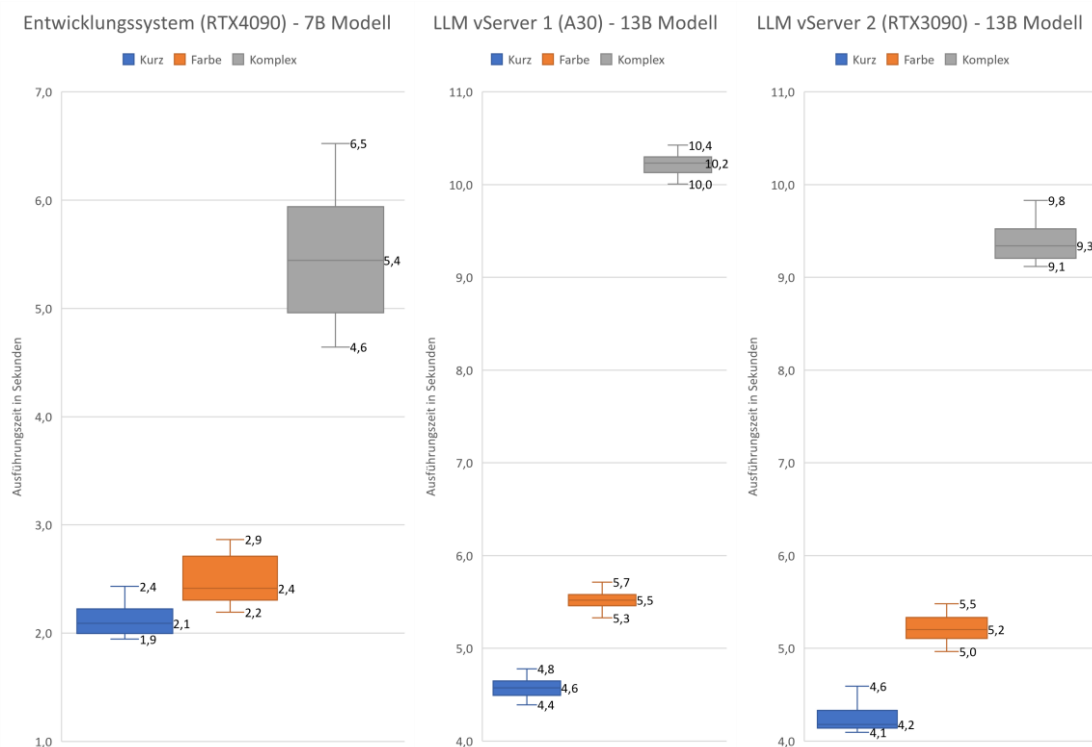


Abbildung 75: Benchmark der Applikation auf verschiedenen Systemen (eigene Grafik)

In der Abbildung sind die Testergebnisse der verschiedenen Systeme zu sehen. Es gilt zu beachten, dass die Skala bei den Tests des Entwicklungssystems nicht identisch zu der Skala bei den Tests für die LLM vServer 1 und 2 ist. Wie zu erwarten war, ist die Ausführungszeit auf dem Entwicklungssystem am kürzesten. Erstens ist in dem System die modernste Hardware aller Systeme verbaut und zweitens wird ein kleineres Modell verwendet. Zudem

4. Praxisteil

muss nicht das lokale Netzwerk verlassen werden, was weiter die Latenz reduziert. Der Median der Ausführungszeit ist jeweils etwa halb so groß, im Vergleich zu den Systemen mit 13B Modellen. Selbst der komplexe Befehl wird im Median in 5,4 Sekunden ausgeführt, dies ist vergleichsweise schnell. Im Vergleich dazu benötigen die anderen Systeme ähnlich lange für den deutlich einfacheren zweiten Test. Dies ist, aufgrund des größeren Modells, auch zu erwarten. Interessant ist, dass im Vergleich der beiden externen Systeme Server 2 mit den RTX3090 schneller ist als Server 1 mit den A30 GPUs, obwohl die A30 laut ihrer technischen Spezifikationen leistungsfähiger sein sollte. Mit etwa 10 Sekunden ist die Ausführungszeit beider 13B Systemen relativ langsam, wenn ein komplexer Befehl gegeben wird. Würde das Ziel verfolgt werden, die Applikation echten Nutzern anzubieten, wäre es vermutlich sinnvoll zu versuchen, diese Zeit noch zu reduzieren. Für den Zweck dieser Arbeit ist die gemessene Ausführungszeit ausreichend.

4.2.4. Beschreibung von aufgetretenen Problemen und deren Lösungen

Während der Entwicklung der Applikation traten eine Vielzahl von Problemen auf, die relevantesten sollen hier vorgestellt und die verschiedenen Lösungsansätze beschrieben werden.

Eines der größten Probleme bestand darin, das LLM dazu zu bringen, die korrekten Geräte-IDs auszugeben. Dieses Problem zog sich durch die gesamte Planungs- und Entwicklungsphase der Applikation. Wie zuvor dargestellt, werden diese Geräte in der fertigen Applikation im Format „Alias: device_id“ angegeben, also zum Beispiel „Floor light: light.floor“. Bei der „device_id“ handelt es sich um die konkrete Geräte-ID, die HomeAssistant intern verwendet, und beim API-Call benötigt wird. Deshalb ist es enorm wichtig, dass die Antwort des LLMs diese ID korrekt wiedergibt. Bereits vor der Entwicklung der Applikation, bei ersten vorläufigen Tests während der Planungsphase, wurde dies zum Problem. Anfänglich wurden die Geräte ohne einen „alias“, als Liste aller Geräte-IDs, im Prompt angegeben. Dabei hatte das LLM Schwierigkeiten die korrekten Geräte-IDs zu identifizieren. Insbesondere wenn es mehrere ähnliche Geräte-IDs gab, wurde dies zum Problem (zum Beispiel „light.livingroom“, „light.livingroom_bulb_1“, „light.livingroom_bulb_2“). Um diesem Problem entgegenzuwirken, wurde die „alias: device_id“ Struktur eingeführt, welche jeder Geräte-ID einen verständlichen Namen zuordnet. Durch diese Maßnahme erhöhte sich die Erfolgsrate des LLMs stark, bietet aber auch Nachteile. Der Hauptgrund für das ursprüngliche einfache Format ohne „alias“ war, dass dieses Format es ermöglicht, mit einer API-Anfrage an HomeAssistant automatisch alle verfügbaren Geräte abzufragen und diese direkt in den entsprechenden Prompt einzubauen. Dadurch wird an dieser Stelle keine manuelle Arbeit des Nutzers benötigt und auch neue Geräte würden automatisch erkannt werden. Der Wechsel auf das neue Format bedeutet, dass der Nutzer die „alias: device_id“ Mappings selbst erstellen muss, indem er manuell eine Liste mit allen Geräte-IDs und passenden Namen anlegt. Versuche, die Erstellung dieser Liste mithilfe eines LLM zu automatisieren, machten einen Großteil der zuvor durch die Mappings erreichten Verbesserungen wieder zunichte, sodass diese auch in der fertigen Applikation manuell angelegt werden muss. Eine weitere Maßnahme im Versuch, die Geräte-ID Auswahl der LLMs zu verbessern, bestand in der beschriebenen Implementierung einer Vektor-Datenbank. Dies ermöglicht es, nicht alle Geräte auf einmal in jedem Prompt angeben zu müssen. Das Reduzieren der Optionen auf die relevantesten Geräte erhöht weiter die Zuverlässigkeit der Geräteauswahl. Durch das Kombinieren der beiden Maßnahmen wurde

4. Praxisteil

das Problem der inkorrekten Geräteauswahl weitestgehend gelöst. Nur in seltenen und komplexen Fällen wird dies weiterhin zum Problem.

Damit ist das Problem der inkorrekt angegebenen Geräte-IDs weiterhin nicht vollständig gelöst. Neben der Auswahl des falschen Geräts, gibt es noch die Problematik, dass sich das LLM mit den genannten Veränderungen zwar für das korrekte Gerät entscheidet, die Geräte-ID allerdings nicht korrekt aus der Liste kopiert. Dieses Problem ist eher selten, es tritt besonders dann auf, wenn eine Geräte-ID in irgendeiner Art vom Standard der restlichen IDs abweicht. Ein einfaches Beispiel, welches tatsächlich in der Testumgebung auftrat, sieht so aus: Angenommen es existieren folgende Geräte-ID „light.livingroom_bulb_1“, „light.licingroom_bulb_2“, „light.livingroom_bulb_3“. Wie zu sehen ist, gibt es einen Rechtschreibfehler bei der zweiten Geräte-ID, indem „licingroom“ anstatt „livingroom“ geschrieben wurde.

Ein solcher Fehler kann bei der Einrichtung eines Smart-Home-Systems schnell passieren und stellt normalerweise auch kein größeres Problem dar. Für diese Applikation ist das ein Problem, da das LLM in seinen Antworten dazu neigt, solche Geräte-IDs nicht korrekt zu übernehmen und stattdessen den Fehler in seiner Antwort automatisch auf „light.livingroom_bulb_2“ korrigiert. HomeAssistant erkennt diese Geräte-ID dann nicht und die gewünschte Aktion für das Gerät wird nicht durchgeführt. Dieser Fall wurde auch im Leistungstest in dem „mit Fehlern“-Test untersucht. In den zuvor vorgestellten Prompt-Templates gibt es teilweise eine „REMEMBER:“ Zeile, kurz vor dem Ende des Prompts, welche versucht dieses Problem zu lösen. Dies gelang nicht immer, zwar wurde das Auftreten des Problems damit reduziert, aber gerade wenn das LLM mehrere ähnlich Geräte-IDs ausgeben muss und eine davon abweicht (wenn zum Beispiel alle 3 genannten „livingroom_bulb“ Geräte auf einmal gesteuert werden sollen), tritt das Problem nach wie vor in der Applikation auf. Wenn ein Gerät mit einer solchen abweichenden Geräte-ID allein verändert werden soll, ist dies seltener ein Problem. Damit tritt das Problem in der fertigen Applikation insgesamt nur in seltenen Spezialfällen auf. Ein weiterer Lösungsansatz, welcher sich nicht als zielführend herausgestellt hat, war das Verwenden von abstrakten IDs, die dann intern auf die passenden Geräte-IDs mittels eines Dict umgeschrieben werden. Anstatt „floor light: light.floor“ würde im Prompt dann zum Beispiel „floor light: fxUGJ7v4“ stehen und „fxUGJ7v4“ würde dann in einem internen Dict auf „light.floor“ zeigen. Dies verhindert, dass ein LLM abweichende Geräte-IDs automatisch „korrigiert“. Bei Tests dieses Ansatzes stellte sich allerdings heraus, dass das LLM noch deutlich mehr Fehler macht als zuvor, im Versuch eine solche abstrakte ID zu kopieren. Damit wurde dieser Ansatz schließlich nicht implementiert.

Insgesamt konnte das Problem der fehlerhaften Geräte-IDs durch die genannten Maßnahmen deutlich reduziert werden. Allerdings zeigte sich das Problem auch noch an einer anderen Stelle. Wie dargestellt wurde, verfügen die Prompts über eine Historie. Daraus ergibt sich ein neues Problem. Wenn der Nutzer eine Eingabe wie „Now dim it to 50%“ macht, muss die Applikation anhand der zuvor getätigten Nutzereingaben korrekt erkennen, welches Gerät gemeint ist. Das Hauptproblem besteht dabei darin, dass zur Lösung der zuvor beschriebenen Probleme das LLM streng dazu aufgefordert wird, nur genau die Geräte-IDs aus der angegebenen Liste zu verwenden. Da diese durch eine Vektor-Datenbank erzeugt wird, wird sich das benötigte Gerät mit hoher Wahrscheinlichkeit nicht mehr in der Geräteliste befinden, da die Vektor-Datenbank mit einer solchen Eingabe keine sinnvolle Geräteliste erzeugen kann.

4. Praxisteil

Zur Lösung dieses Problems wurde mit der Klasse „search_history_buffer“ zuerst eine Historie für die Vektor-Datenbank-Suchen implementiert. Diese ermöglicht es, alle Ergebnisse von vorherigen Vektor-Datenbank-Suchen zusätzlich zu den jeweiligen neuen Ergebnissen in den Prompt einzufügen. In der Tat löste diese Maßnahme das Problem. Bei Tests stellte sich allerdings heraus, dass damit gleichzeitig ein Großteil, des durch die Verwendung von Vektor-Datenbanken erzielten Vorteils, verloren geht, da am Ende ohnehin wieder fast alle verfügbaren Geräte im Prompt stehen, was ursprünglich durch die Vektor-Datenbank verhindert werden sollte. Daher wurde eine alternative Lösung umgesetzt, welche daraus besteht, im Prompt an mehreren Stellen darauf hinzuweisen, dass Geräte-IDs sowie aus der Geräteliste als auch aus der Historie verwendet werden dürfen. Dies weicht zwar die vorher sehr streng geregelte Geräteauswahl etwas auf, führte in Tests aber zu deutlich besseren Ergebnissen.

Ein weiteres mit der Historie zusammenhängendes Problem, war die abnehmende Qualität der Ergebnisse mit zunehmender Historie. Das Problem mit der Historie ist, dass sie bei Zero-Shot-Prompts anfängt, als Beispiele wie bei einem Few-Shot-Prompt zu wirken. Dies kann zu einem massiven Problem werden, wenn diese „Beispiele“ andere Dinge darstellen, als vom Prompt erwartet wird. Anfänglich besaß die Applikation nur eine einzige Historie für alle Prompts. Das führte aber zu massiven Problemen, wenn der Nutzer etwa erst mit der Conversation-Chain über ein Programmierproblem spricht und diese Kommunikation dann durch die Historie in die Command-Chain oder Inquiry-Chain übernommen wird. Die Lösung des Problems ist prinzipiell einfach: Es müssen getrennte Historien verwendet werden. Gleichzeitig hat diese Lösung aber den Nachteil, dass der Nutzer sich nicht mehr auf Kommunikationen mit anderen Teilen der Applikation beziehen kann, was die interne Aufteilung der Chains sehr transparent für den Nutzer macht. Als Kompromiss wurde die in der fertigen Applikation zu sehende Lösung gewählt, in der es zwei Historien gibt, eine für die Conversation-Chain und eine geteilte für die Command- und Inquiry-Chain.

4.3. Auswertung

Nachdem die Planung als auch die Entwicklung abschlossen wurden, soll der Prozess in diesem Kapitel ausgewertet werden. Zuerst wird eine funktionale Auswertung durchgeführt, in welcher die Anforderungen aus Kapitel „4.1.1. Anforderungen“ noch einmal darauf überprüft werden, ob und wie diese erfüllt werden. Zusätzlich wird dargestellt, welche der optionalen Anforderungen implementiert wurden und welche nicht. Schließlich folgen die aus der Entwicklung der Applikation gezogenen Schlussfolgerungen.

4.3.1. Funktionale Auswertung

Einige der Anforderungen aus „Tabelle 12: Softwareplanung: Anforderungen“ können für diese Auswertung grob zusammengefasst werden. So beschreiben die Anforderungen 1 – „Geräte ein- und ausschalten“, 2 – „Farbe von Lichtern ändern“ und 3 – „Helligkeit von Lichtern ändern“ zusammengefasst das Manipulieren von Gerätezuständen. Wie zuvor beschrieben, wird diese Funktionalität implementiert, indem sich die Router-Chain für die Command-Chain entscheidet. Später, während der Transform-Chain, wird dadurch die Command-Klasse aufgerufen. Dabei können grundlegend mit der Device-Klasse alle Geräte im Smart-Home-System ein- und ausgeschaltet werden, egal von welchem Typ sie sind, während Lichter im Speziellen, über die Light-Klasse auch die Manipulation von Helligkeit und Farbe unterstützen.

Verwandt mit den Anforderungen 1, 2 und 3 sind die Anforderungen 4 – „Status von Geräten abrufen“, 5 – „Farbe von Geräten abrufen“ und 6 – „Helligkeit von Geräten abrufen“. Diese stellen jeweils ein Gegenstück zu den Anforderungen 1, 2 und 3 dar, indem die dort veränderten Eigenschaften abgefragt werden sollen. Auch diese Funktionalität wurde implementiert und läuft, wie zuvor dargestellt, fast identisch ab, wie das Manipulieren von Eigenschaften. Wobei von der Router-Chain die Inquiry-Chain aufgerufen, welche intern die Inquiry-Klasse verwendet, um Anfragen an das Smart-Home-System mithilfe der Geräteklassen durchzuführen.

Auch die Anforderung 7 – „Bedingung mit natürlicher Sprache“ wurde erfolgreich implementiert, wie in den Anwendungsbeispielen aus Kapitel „4.2.2. Demonstration der Funktionsweise“ zu sehen ist. Nutzer können mit natürlicher Sprache mit der Applikation interagiert, ohne dabei auf spezielle Kommandowörter zurückgreifen zu müssen. Das LLM ist mithilfe der verschiedenen Prompts in der Lage, diese Texte in entsprechende Befehle für die Applikation umzuwandeln.

Eng verbunden mit Anforderung 7 ist Anforderung 8 „Ausgabe in natürlicher Sprache“. Wenn der Nutzer mit natürlicher Sprache mit der Applikation interagiert, erwartet er auch, dass diese ebenso in natürlicher Sprache antwortet. Auch diese Funktionalität wurde implementiert, dazu wird die Response-Chain verwendet, um aus den JSON-Ausgaben des Codes, Antworten zu konstruieren.

Anforderung 9, das Interagieren mit HomeAssistant, ist eine Grundlage für viele der zuvor genannten Anforderungen und wurde ebenfalls erfolgreich implementiert. Dabei stellt die „Homeassistant_API_client“ Klasse diese Implementation dar und übernimmt die API-Calls an die HomeAssistant REST-API.

Damit sind alle Anforderungen, welche mit der Notwendigkeit „Muss“ markiert wurden, in der fertigen Applikation vorhanden und das Grundziel wurde erreicht. Weiterhin wurden

4. Praxisteil

auch einige der optionalen Anforderungen eingebaut. So wurde die Anforderung 10 – „Allgemeine Chat Funktionalität“ mithilfe der Default-Chain implementiert, welche aufgerufen wird, wenn die Router-Chain weder die Command-Chain noch die Inquiry-Chain für die richtige Wahl hält und es dem Nutzer ermöglicht, die Applikation auch als AI-Chat-Bot wie eine einfache Version von ChatGPT zu verwenden.

Bei Anforderung 11 – Mit „Einfaches Hinzufügen neuer Geräte“ findet sich die erste Anforderung, welche nicht umgesetzt wurde. Wie zuvor in Kapitel „4.2.4. Beschreibung von aufgetretenen Problemen und deren Lösungen“ dargestellt, gab es Versuche, diese Anforderung zu implementieren, welche sich jeweils nicht als sinnvolle Lösungen herausstellten. Aufgrund der optionalen Natur dieser Anforderung wurde diese schließlich nicht weiterverfolgt.

Im Gegensatz dazu ist die in Anforderung 12 beschriebene Chat-Historie in der fertigen Applikation zu finden. Diese wurde in der beschriebenen Art und Weise mit zwei verschiedenen History-Objekten implementiert, und stellt einen der komplexeren Aspekte der Applikation dar.

Die Anforderungstabelle endet mit Anforderung 13 – „Custom-Befehle“. Auch diese letzte optionale Anforderung wurde nicht umgesetzt. Dies liegt vorwiegend an der hohen Komplexität einer solchen Funktionalität, die der begrenzten Zeit sowie der Tatsache, dass diese Anforderung optional ist, gegenübersteht. Für diese erste Version der Applikation wurde die Anforderung deshalb nicht implementiert, allerdings sei gesagt, dass die Umsetzung dieser Anforderung für einen Nutzer der Applikation praktische Vorteile bieten könnte und diese Anforderung bei einer potenziellen Weiterentwicklung der Applikation eine wichtigere Rolle spielen sollte.

Insgesamt wurden 11 der 13 Anforderungen umgesetzt, darunter alle neun zwingend notwendigen und zwei der optionalen Anforderungen. Die zwei nicht implementierten Anforderungen sind optional. Damit kann die Entwicklung der Applikation aus funktionaler Sicht auf Basis der Anforderungsliste als erfolgreich bezeichnet werden, wobei das Fehlen von zwei optionalen Funktionen auch zeigt, dass eine zukünftige Weiterentwicklung der Applikation sinnvoll sein könnte.

4.3.2. Schlussfolgerungen aus dem Praxisteil

Die Planung und Entwicklung einer mithilfe von LangChain mit LLMs zusammenarbeitenden Applikation hat verschiedene Aspekte verdeutlicht und dabei Probleme als auch positive Punkte eines solchen Ansatzes aufgezeigt. Diese Schlussfolgerungen aus der Entwicklung und Planung werden hier zum Ende des Praxisteils dargestellt werden.

Das mit Abstand wichtigste Ergebnis ist das Erkennen des enormen Einflusses, welchen die Prompts auf die Applikation besitzen. Auf der einen Seite scheint dies sehr offensichtlich, schließlich sind die Prompts der einzige Weg, in dem mit dem LLM kommuniziert wird. Andererseits hat die intensive Arbeit mit LLMs und Prompts verdeutlicht, dass selbst sehr kleine Veränderungen in der Aufgabenbeschreibung bereits große Auswirkungen haben können. Dies war auch im LLM-Leistungsvergleich schon zu sehen, wo das Hinzufügen von nur neun Wörtern die Leistung des Bloom-7B Modells um 360% erhöhte. Auch in der Entwicklung der Applikation gab es Fälle, in denen minimale Änderungen der Prompts, eine enorme Leistungssteigerung zufolge hatten. Insbesondere, aber nicht ausschließlich, beim Prompt der Router-Chain war dies zu bemerken.

4. Praxisteil

Zusätzlich zur Aufgabenbeschreibung im Prompt hat sich auch das Format des Prompts bzw. der Aufgabenbeschreibung als überraschend wichtig herausgestellt. Schon das Hinzufügen oder Entfernen eines einzelnen „\n“ kann einen großen Unterschied ausmachen.

Erschwerend kommt noch hinzu, dass sich dieses Verhalten auch von LLM zu LLM unterscheidet und jeweils immer neu getestet werden muss. Insgesamt durchliefen die Prompts der Applikation sechs verschiedene Versionen mit großen Änderungen, während jede dieser Versionen noch zahlreiche Male mit kleinen Änderungen angepasst wurden.

Neben den Prompts hat sich auch herausgestellt, dass die mit LangChain erzeugte LLM-Architektur schnell sehr komplex werden kann. Selbst in einem vergleichsweise kleinen Projekt, wie dem hier beschriebenen, gibt es zahlreiche Chains, die miteinander interagieren und zusammen die LLM-Architektur darstellen. Genau wie bei klassischer Softwareplanung, ist es essenziell diese LLM-Architektur im Vorhinein zu planen, zu visualisieren und zu dokumentieren. Da diese komplexen Strukturen sonst schnell unübersichtlich werden. Dies wird insbesondere noch schwieriger dadurch, dass, anders als bei „normalen“ Code, die Aufgabe und Funktionsweise einer Chain nicht direkt erkenntlich ist, da diese in einem Prompt beschrieben wird, welcher sich in der Regel an einer anderen Stelle im Code oder in einer Konfigurationsdatei befindet.

Eine weitere wichtige Erkenntnis besteht darin, zu verstehen, welche Teile einer Applikation durch ein LLM übernommen werden können und welche nicht. In Beiträgen und Foren ist vermehrt zu lesen, dass LLMs in der Lage sind, riesige Bestandteile einer Applikation zu übernehmen, die klassischerweise von „normalen“ Code übernommen werden würden. Dies ist aber nur bedingt wahr. Einerseits ist in der entwickelten Applikation zu sehen, wie verschiedene Bestandteile durch ein LLM übernommen werden können. Ein klares Beispiel dafür ist die „Color_helper“ Klasse, welche ein LLM verwendet, um Farbnamen in RGB-Codes zu übersetzen. Eine solche Aufgabe würde klassischerweise manuell als Code implementiert werden. Auf der anderen Seite ist aber auch zu sehen, dass große Teile der Applikation nach wie vor auf normalen Code setzen, wie auch das dargestellte Klassendiagramm zeigte.

Die Notwendigkeit von „normalem“ Code ist auch besonders dann zu sehen, wenn die Hardwareperspektive einbezogen wird. Das Erzeugen komplexer LLM-Strukturen führt schnell zu einer enormen Anzahl an LLM-Anfragen. Selbst bei kleineren Applikationen können schnell 10 oder mehr LLM-Anfragen pro Durchlauf der Applikation entstehen. Wie in Kapitel „4.2.3. Darstellung der Hardwareseite und Performance“ zu sehen war, dauert dies lange und benötigt leistungsfähige Hardware. Die wichtige Erkenntnis daraus ist, dass dieser Aspekt unbedingt bereits bei der Planung der LLM-Architektur beachtet werden muss. Um die Hardwareanforderungen möglichst gering zu halten und die Verzögerungen durch LLM-Anfragen zu reduzieren, sollte die Interaktion mit LLMs auf ein Minimum reduziert werden. Nach dem Prinzip: So viel wie nötig, aber so wenig wie möglich.

Allerdings hat sich während der Entwicklung auch gezeigt, dass Open-Source LLMs heute bereits sehr starke Leistung bieten können. Die Fehleranfälligkeit der Applikation ist im Großen und Ganzen deutlich geringer als ursprünglich erwartet. Die Applikation führt ihre Anweisung meistens komplett ohne Probleme korrekt aus. Das Implementieren von LLMs in Software, welche direkt mit Nutzern interagiert, ist damit durchaus realistisch.

4. Praxisteil

Beim Einsetzen von LLMs gibt es noch ein weiteres großes Problem, welches sich während der Entwicklung mehrfach gezeigt hat und welches eine essenzielle Erkenntnis aus dieser Arbeit darstellt. Das Arbeiten mit LLMs in einer Applikation ist besonders deshalb schwierig, weil LLMs in der Regel nicht deterministisch arbeiten. Dies führt zu einer der wohl am komplexesten zu behebenden Fehler: nicht reproduzierbare Bugs durch fehlerhafte LLM-Ausgaben. Aufgrund der Tatsache, dass sich die Ausgabe eines LLMs mit jeder Ausführung unterscheiden kann, können Bugs auftreten, welche sehr schwierig zu diagnostizieren und zu beheben sind. Da das LLM einen bestimmten Fehler womöglich nur in einer von 100 Ausführungen macht. Dazu kommt noch, dass es auch schwierig ist, herauszufinden, ob eine jeweilige Veränderung das Problem behoben hat oder nicht. Da ein solcher Bug kaum reproduzierbar ist, ist es unmöglich zu wissen, ob dieser nicht mehr auftritt, weil die jeweilige Anpassung diesen behoben hat oder weil er ohnehin nicht aufgetreten wäre. Ausführliches Logging ist dadurch besonders wichtig.

Eng mit dieser Erkenntnis verbunden ist auch die Erfahrung, dass LangChain als Bibliothek suggeriert, dass es damit zügig und mit wenig Aufwand möglich ist, LLM-Aufgaben zu lösen. In Realität ist die Bibliothek sehr komplex, insbesondere auch, da nicht alle Aspekte gut dokumentiert sind. Dazu kommt das hohe Abstraktionslevel, welches es oft schwierig macht, zu verstehen, was LangChain im Hintergrund tatsächlich tut und es damit, selbst bei einem guten Verständnis der zugrundeliegenden Techniken (wie zum Beispiel RAG), komplex werden kann, bestimmte Dinge mit LangChain zu implementieren. Genau aus diesem Grund ist es notwendig, nicht nur LangChain zu lernen, sondern vor allem auch die dahinterstehenden Prinzipien, wie RAG oder Vektor-Datenbanken, zu verstehen. Dies ist besonders beim Versuch Fehler in der eigenen Applikation zu finden und zu beheben unverzichtbar.

5. Fazit

Am Anfang der Arbeit stellte sich die Frage „Welche Open-Source LLMs gibt es, wie sind diese einzuordnen und wie können diese verwendet werden, um echte Probleme zu lösen?“. Im Verlauf der Arbeit konnte diese Frage weitestgehend geklärt werden. Es stellte sich heraus, dass es eine enorme Anzahl an Open-Source LLMs gibt, welche hauptsächlich auf der Internetseite Huggingface.co gebündelt sind. Dabei schienen sich viele Modelle oberflächlich vorerst sehr ähnlich zu sein. Der Vergleich 10 dieser Modelle hat allerdings aufgezeigt, wie unterschiedlich die Modelle voneinander sein können, nicht nur im Hinblick auf die Leistung und Anwendung der Modelle, sondern auch in Bezug auf andere Faktoren wie die Lizenzen oder die Community. Eine besonders interessante Erkenntnis aus diesem Vergleich war, dass es auch abseits der sehr bekannten Modelle wie LLaMA 2 möglich ist, LLMs mit guter Leistung für den eigenen Anwendungsfall zu finden. Allgemein hat die Leistung von Open-Source LLMs im Vergleich zur Untersuchung in der vorhergehenden Bachelorarbeit stark zugenommen. Zusätzlich gibt es heute deutlich mehr Auswahl. Eine weitere wichtige Erkenntnis daraus ist, dass selbst kleinere Modelle (wie die für die Applikation genutzten 7B und 13B Modelle), je nach Anwendungsfall, gute Leistung bieten können. Aufgrund dieser großen Auswahl an Modellen, ist es besonders wichtig, verschiedene LLMs unter Berücksichtigung des eigenen Anwendungsfalls zu vergleichen, bevor eine Entscheidung für ein spezielles Modell getroffen wird. Nicht zu vergessen sind dabei auch die allgemeinen Vor- und Nachteile von Open-Source und Closed-Source Modellen, die zum Beginn der Arbeit dargestellt wurden und ebenfalls berücksichtigt werden sollten. So sollte am Ende für jedes Projekt neu ermittelt werden, welche Aspekte der Modelle es gibt, welche Modelle relevant sind und wie diese Modelle bei den jeweiligen Aspekten zu bewerten sind. Insbesondere da regelmäßig neue interessante Modelle veröffentlicht werden, kann eine Untersuchung von Modellen, wie sie in dieser Arbeit geschehen ist, immer nur einen bestimmten Zeitpunkt abbilden. Es ist auch in der Zukunft wichtig, diesen Bereich weiter zu verfolgen, um auf dem neusten Wissensstand zu bleiben.

Auch abseits der eigentlichen Modelle gab es in den vergangenen Jahren Fortschritte. Wo das Verwenden von GPT-2 in der vorhergehenden Bachelorarbeit noch vergleichsweise aufwendig war, ist es mittlerweile, durch Tools wie den vorgestellten „Text Generation Inference“ Server von Hugging Face, deutlich einfacher geworden, eine Vielzahl von Modellen auszuprobieren und für eigene Anwendungen zu verwenden, indem eine LLM-API gehostet wird. Die Verwendung von Open-Source LLMs ist damit heute deutlich einfacher. Zusätzlich erleichtert wird die Anwendung von LLMs heute noch dadurch, dass weitere Open-Source-Projekte, wie das für diese Arbeit wichtige LangChain, verfügbar sind. Welches es dank seiner Wrapper und Abstrahierungen möglich macht komplexe LLM Aufgaben zu lösen. Dabei ist aber nicht zu übersehen, dass LangChain für genau diese Aspekte teilweise auch kritisiert wird. Genau wie bei den LLMs überprüft werden sollte, welches Modell für den eigenen Anwendungsfall das richtige ist, muss vor der Verwendung von LangChain überprüft werden, ob die Bibliothek für den jeweiligen Fall die korrekte Wahl ist. In den Situationen, in denen dies der Fall ist, stellt LangChain allerdings eine große Hilfe in der Entwicklung von LLM-Anwendungen dar. Nicht zuletzt auch aufgrund des in dieser Arbeit vorgestellten, sehr großen, Funktionsumfangs. Der Fokus dieser Arbeit lag, was die LLM-Anwendung angeht, zwar auf LangChain. Allerdings ist dieser Bereich damit noch keinesfalls vollständig untersucht.

5. Fazit

Im Anschluss dieser Arbeit wäre es beispielsweise sehr interessant, die ebenfalls erwähnte Bibliothek „DSPy“ zu untersuchen. Zusätzlich gibt es auch in diesem Bereich regelmäßig neue interessante Projekte.

Ein relevanter Nachteil von LLMs, welcher schon in der vorhergehenden Arbeit dargestellt wurde und hier ebenfalls auftrat, sind die nach wie vor sehr hohen Hardwareanforderungen für die Verwendung von LLMs. Für die lokale Ausführung von LLM-Applikationen, direkt auf der Nutzerhardware, sind diese Anforderungen bedeutend zu hoch. Wenigstens für den LLM-Teil muss in der Regel weiterhin auf externe Server ausgewichen werden. Und selbst dann können die Ausführungszeiten sehr langsam werden, wenn die Applikation eine Reihe von LLM-Anfragen durchführen muss. Dies stellt wohl eines der größten Probleme von LLMs zum aktuellen Standpunkt dar. Auch in diesem Bereich gibt es verschiedene Aspekte, welche in weiterführender Forschung und Recherche, im Anschluss zu dieser Arbeit, noch untersucht werden können, insbesondere die erwähnte Technik „Quantisierung“ stellt eine mögliche Verbesserung der Hardwareanforderungen dar, welche genauer untersucht werden sollte.

Mit der Entwicklung der Applikation konnte schließlich auch der letzte Teil der anfänglichen Frage beispielhaft geklärt werden, „[...] Wie können [LLMs] verwendet werden, um echte Probleme zu lösen?“. Die Applikation zeigt, dass speziell die Planung der LLM-Aspekte im Vorhinein wichtig ist. Zudem war zu erkennen, dass LLM-Applikationen schnell sehr komplex werden können und besonders das Finden und Beheben von Bugs in diesen Applikationen ein Problem darstellen kann. Auch war zu sehen, dass LLMs zwar einige Aspekte von Applikationen übernehmen können, dass solche Applikationen allerdings nach wie vor auf eine Menge von „normalem“ Code angewiesen sind und somit eher eine Symbiose aus LLMs und Code darstellen.

Alles in allem konnten die anfänglich gesetzten Ziele im Verlauf dieser Arbeit erreicht werden und die gestellten Fragen beantwortet werden. Dabei sind alle in dieser Arbeit dargestellten Aspekte jeweils nur ein kleiner und eingeschränkter Teil des Gesamtthemas „Large Language Models“, da dieses enorm komplex und umfangreich ist. Es gibt eine Vielzahl an Aspekten, die bei weiterer Forschung untersucht werden können, viele davon wurden auch schon im Verlauf dieser Arbeit genannt. LLMs werden in der kommenden Zeit vermutlich eine immer wichtiger werdende Rolle spielen, was weiterführende Forschung in diesem Gebiet besonders interessant und relevant macht.

Abbildungsverzeichnis

Abbildungsverzeichnis

| | |
|--|----|
| Abbildung 1: Beispiel für einen einfachen Few-Shot Prompt | 14 |
| Abbildung 2: Beispiel für einen einfachen Zero-Shot-Prompt | 15 |
| Abbildung 3: Grobes Format der Eval-Prompts | 21 |
| Abbildung 4: Ausschnitt aus dem "Kurz ohne Beispiele" Test | 23 |
| Abbildung 5: Einzeltestergebnisse von Platypus2-7B | 24 |
| Abbildung 6: Beispiel für fehlerhafte Ausgabe des Platypus2-7B Modells beim "Kurz ohne Beispiele" Test | 25 |
| Abbildung 7: Gesamtergebnis von Platypus2-7B | 25 |
| Abbildung 8: Einzeltestergebnisse von Platypus2-13B | 26 |
| Abbildung 9: Gesamtergebnis von Platypus2-13B | 26 |
| Abbildung 10: Einzeltestergebnisse von Platypus2-70B | 27 |
| Abbildung 11: Beispiel für fehlerhafte Ausgabe des Platypus2-70B Modells beim "Lang ohne Beispiele" Test | 27 |
| Abbildung 12: Gesamtergebnis von Platypus2-70B | 28 |
| Abbildung 13: Originale Testergebnisse von Bloom-7B1 | 28 |
| Abbildung 14: Angepasstes Prompt Format für Bloom-7B1 | 29 |
| Abbildung 15: Testergebnisse von Bloom-7B1 mit angepasstem Prompt | 29 |
| Abbildung 16: Beispiel für fehlerhafte Ausgabe des Bloom-7B1 Modells beim "Lang ohne Beispiele" Test | 29 |
| Abbildung 17: Testergebnisse von Cerebras-GPT 13B | 30 |
| Abbildung 18: Beispiel für fehlerhafte Ausgabe des Cerebras-GPT 13B Modells beim "Lang" Test | 30 |
| Abbildung 19: Testergebnisse von Pythia-12B-Deduped | 31 |
| Abbildung 20: Beispiel für fehlerhafte Ausgabe des Pythia-12B-Deduped Modells beim "Lang" Test | 31 |
| Abbildung 21: Testergebnisse von RWKV-World-7B | 32 |
| Abbildung 22: Testergebnisse von StableLM-Base-Alpha-7B-V2 | 32 |
| Abbildung 23: Beispiel für fehlerhafte Ausgabe des StableLM-Alpha-7B-V2 Modells beim "Lang" Test | 33 |
| Abbildung 24: Beispiel für fehlerhafte Ausgabe des Flan-UL2 Modells beim "Lang" Test | 33 |
| Abbildung 25: Testergebnis von Flan-UL2 | 34 |
| Abbildung 26: Testergebnis von MPT-30B-Instruct | 34 |
| Abbildung 27: Testergebnis von MPT-7B-Instruct | 35 |
| Abbildung 28: Testergebnis von OpenLLaMA 13B | 35 |
| Abbildung 29: Testergebnis von Falcon-40B-Instruct | 36 |
| Abbildung 30: Codebeispiel für die Erstellung eines LLM-Objekts | 54 |
| Abbildung 31: Codebeispiel für die Erstellung eines LLM-Objekts mit bestimmten Hyper-Parametern | 55 |
| Abbildung 32: Codebeispiel für ein allgemeinen Prompt | 55 |
| Abbildung 33: Codebeispiel für ein Few-Shot-Prompt | 56 |
| Abbildung 34: Codebeispiel für die "partial" Methode | 57 |
| Abbildung 35: Codebeispiel für eine einfache LLMChain mit dem „fspt“ Prompt | 57 |
| Abbildung 36: Codebeispiel für eine komplexere LLMChain mit dem „pt“ Prompt | 58 |
| Abbildung 37: Codebeispiel für die Nutzung einer Sequential-Chain | 58 |
| Abbildung 38: Codebeispiel für eine Transform-Chain | 59 |
| Abbildung 39: Codebeispiel für eine Router-Chain | 60 |

Abbildungsverzeichnis

| | |
|--|----|
| Abbildung 40: Ausgabe des in Abbildung 39 dargestellten Codes..... | 60 |
| Abbildung 41: Erzeugen der default_chain | 61 |
| Abbildung 42: Beispielcode für das Erzeugen des finalen Routers..... | 61 |
| Abbildung 43: Beispielcode für das Definieren von Tools für einen Agent | 62 |
| Abbildung 44: Codebeispiel für das Definieren und Ausführen eines Agents | 63 |
| Abbildung 45: Codebeispiel für das manuelle erzeugen eines Agents und AgentExecutors .. | 63 |
| Abbildung 46: Beispielhafter Konsolen-Output eines Agents mit "verbose = True" | 64 |
| Abbildung 47: Beispielhafte Prompt-Vorlage für ein Prompt mit Memory..... | 64 |
| Abbildung 48: Beispielcode für das Erzeugen einer Chain mit Memory | 65 |
| Abbildung 49: Veranschaulichung von RAG bei der Verwendung einer Vector-Datenbank [78] | 66 |
| Abbildung 50: Codebeispiel für das Erstellen einer RAG-Chain..... | 66 |
| Abbildung 51: Codebeispiel für das Erzeugen einer ChromaDB Vektor-Datenbank anhand einer PDF-Datei | 67 |
| Abbildung 52: Codebeispiel für das Erzeugen einer RAG-Chain basierend auf der LLMChain Klasse | 68 |
| Abbildung 53: Codebeispiel für die Ausführung von RAG-Chains..... | 68 |
| Abbildung 54: Softwareplanung: Fachlicher- und technischer Kontext (eigene Grafik) | 72 |
| Abbildung 55: Verkürzter JSON-Input für die TGI API - POST /generate Schnittstelle | 73 |
| Abbildung 56: Verkürzter JSON-Output der TGI API - POST /generate Schnittstelle | 73 |
| Abbildung 57: Antwort der HomeAssistant API auf eine GET-Anfrage an /states/sun.sun [85] | 74 |
| Abbildung 58: Beispiel für ein POST Request zum Einschalten einer Lampe und setzen der Farbe auf blau..... | 75 |
| Abbildung 59: LLM-Architektur der Applikation (eigene Grafik) | 75 |
| Abbildung 60: Grobarchitektur der Applikation (eigene Grafik)..... | 76 |
| Abbildung 61: Grundlegendes Prompt-Template für Platypus2 Modelle | 77 |
| Abbildung 62: Prompt-Template für die „Router_chain“ | 78 |
| Abbildung 63: Prompt-Template der "Command"-Chain | 79 |
| Abbildung 64: Prompt-Template für die Inquiry-Chain | 80 |
| Abbildung 65: Prompt-Template für die Default-Chain | 81 |
| Abbildung 66: Prompt-Template für die Response-Chain | 81 |
| Abbildung 67: Klassendiagramm der Applikation (eigene Grafik) | 84 |
| Abbildung 68: Beispiel für die „commands“-Liste | 85 |
| Abbildung 69: Anwendungsbeispiel 1, Standardinteraktion | 87 |
| Abbildung 70: Anwendungsbeispiel 2, Inquiry/Command Mix und Multi-Action..... | 88 |
| Abbildung 71: Vergleich der vom Modell gewählten Farbe (links) mit einer blauen Einstellung (rechts). (eigene Abbildung)..... | 88 |
| Abbildung 72: Anwendungsbeispiel 3, Komplexe Multi-Action..... | 89 |
| Abbildung 73: Anwendungsbeispiel 4, LLM-Chat-Bot..... | 89 |
| Abbildung 74: Verteilungsdiagramm der Entwicklungsumgebung (eigene Grafik)..... | 90 |
| Abbildung 75: Benchmark der Applikation auf verschiedenen Systemen (eigene Grafik)..... | 91 |

Tabellenverzeichnis

Tabellenverzeichnis

| | |
|---|----|
| Tabelle 1: Liste der zu vergleichenden Modelle..... | 17 |
| Tabelle 2: LLM-Vergleich: Leistung..... | 36 |
| Tabelle 3: LLM-Vergleich: Offenheit..... | 40 |
| Tabelle 4: Apache 2.0 Lizenz Zusammenfassung | 41 |
| Tabelle 5: BigScience RAIL 1.0 Lizenz Zusammenfassung | 41 |
| Tabelle 6: Creative Commons Attribution 4.0 International Lizenz Zusammenfassung..... | 42 |
| Tabelle 7: LLaMA 2 Lizenz Zusammenfassung | 43 |
| Tabelle 8: FALCON 180B TII Lizenz Zusammenfassung | 44 |
| Tabelle 9: LLM-Vergleich: Lizenz | 44 |
| Tabelle 10: LLM-Vergleich: Community | 50 |
| Tabelle 11: LLM-Vergleich: Gesamtergebnis..... | 51 |
| Tabelle 12: Softwareplanung: Anforderungen..... | 71 |
| Tabelle 13: Softwareplanung: Eingrenzung | 71 |

Literaturverzeichnis

- [1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan und R. Child, „Language Models are Few-Shot Learners,“ 26 05 2020. [Online]. Available: <https://arxiv.org/abs/2005.14165>. [Zugriff am 01 07 2021].
- [2] OpenAI, „Introducing ChatGPT,“ 30 11 2022. [Online]. Available: <https://openai.com/blog/chatgpt>. [Zugriff am 07 09 2023].
- [3] S. McCallum, „ChatGPT banned in Italy over privacy concerns,“ BBC, 01 04 2023. [Online]. Available: <https://www.bbc.com/news/technology-65139406>. [Zugriff am 07 09 2023].
- [4] J. Bevernis, „GitHub Repository dieser Arbeit,“ [Online]. Available: <https://github.com/stasj145/Masterarbeit-OS-LangChain-App>. [Zugriff am 30 11 2023].
- [5] Open Source Initiative, „The Open Source Definition,“ 22 02 2023. [Online]. Available: <https://opensource.org/osd/>. [Zugriff am 22 09 2023].
- [6] HuggingFace, „Open LLM Leaderboard,“ [Online]. Available: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard. [Zugriff am 17 09 2023].
- [7] G. Brockman, „Microsoft invests in and partners with OpenAI to support us building beneficial AGI,“ 22 07 2019. [Online]. Available: <https://openai.com/blog/microsoft-invests-in-and-partners-with-openai>. [Zugriff am 2023 09 10].
- [8] D. Bass, „Microsoft Invests \$10 Billion in ChatGPT Maker OpenAI,“ Bloomberg, 23 01 2023. [Online]. Available: <https://www.bloomberg.com/news/articles/2023-01-23/microsoft-makes-multibillion-dollar-investment-in-openai#xj4y7vzkg>. [Zugriff am 2023 09 10].
- [9] A. Shilov, „Samsung Fab Workers Leak Confidential Data While Using ChatGPT,“ TomsHardware, 03 04 2023. [Online]. Available: <https://www.tomshardware.com/news/samsung-fab-workers-leak-confidential-data-to-chatgpt>. [Zugriff am 10 09 2023].
- [10] OpenAI, „Enterprise-grade security & privacy and the most powerful version of ChatGPT yet.,“ 2023. [Online]. Available: <https://openai.com/enterprise>. [Zugriff am 10 09 2023].
- [11] OpenAI, „Enterprise privacy at OpenAI,“ [Online]. Available: <https://openai.com/enterprise-privacy>. [Zugriff am 10 09 2023].
- [12] A. Belanger, „ChatGPT users drop for the first time as people turn to uncensored chatbots,“ ArsTechnica, 07 07 2023. [Online]. Available: <https://arstechnica.com/tech-policy/2023/07/chatgpts-user-base-shrank-after-openai-censored-harmful-responses/>. [Zugriff am 10 09 2023].
- [13] B. Buchanan, K. Sedova, A. Lohn und M. Musser, „Truth, Lies, and Automation - How Language Models Could Change Disinformation,“ 05 2021. [Online]. Available: <https://cset.georgetown.edu/publication/truth-lies-and-automation/>. [Zugriff am 10 09 2023].
- [14] Raf, „What are tokens and how to count them?,“ [Online]. Available: <https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them>. [Zugriff am 10 09 2023].
- [15] Google AI, „Generative AI for Developers - Models,“ 18 07 2023. [Online]. Available: https://developers.google.com/models/language#model_metadata. [Zugriff am 10 09 2023].

Literaturverzeichnis

- [16] W. Knight, „OpenAI’s CEO Says the Age of Giant AI Models Is Already Over,“ Wired, 17 04 2023. [Online]. Available: <https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/>. [Zugriff am 14 09 2023].
- [17] J. Bevernis, „Funktionsweise, Anwendungen und Risiken von Texterzeugungsmodellen am Beispiel von GPT,“ Hochschule Stralsund, Stralsund, 2021.
- [18] P. D.-I. N. Gronau, „Softwareauswahl,“ 08 04 2019. [Online]. Available: <https://wi-lex.de/index.php/lexikon/informations-daten-und-wissensmanagement/informationsmanagement/informationsmanagement-aufgaben-des/softwareauswahl/>. [Zugriff am 11 11 2023].
- [19] Hannibal046, „Awesome-LLM,“ 01 07 2023. [Online]. Available: <https://github.com/Hannibal046/Awesome-LLM>. [Zugriff am 17 09 2023].
- [20] E. Yan, „Open LLMs,“ 28 08 2023. [Online]. Available: <https://github.com/eugeneyan/open-llms>. [Zugriff am 2023 09 17].
- [21] Meta, „Llama 2,“ [Online]. Available: <https://huggingface.co/meta-llama/Llama-2-70b>. [Zugriff am 19 09 2023].
- [22] BigScience Workshop, „BigScience Blog,“ [Online]. Available: <https://bigscience.huggingface.co/blog>. [Zugriff am 21 09 2023].
- [23] RWKV, „RWKV,“ [Online]. Available: <https://www.rwkv.com/>. [Zugriff am 19 09 2023].
- [24] Ludwig-Maximilians-Universität München, „Revolutionizing image generation by AI: Turning text into images,“ 01 09 2022. [Online]. Available: <https://www.lmu.de/en/newsroom/news-overview/news/revolutionizing-image-generation-by-ai-turning-text-into-images.html>. [Zugriff am 01 12 2023].
- [25] Y. Tay, M. Dehghani, V. Q. Tran, X. Garcia, D. Bahri, T. Schuster, H. S. Zheng, N. Houlsby und D. Metzler, „Unifying Language Learning Paradigms,“ 10 05 2022. [Online]. Available: <https://arxiv.org/abs/2205.05131v1>. [Zugriff am 20 09 2023].
- [26] EleutherAI, „Language Model Evaluation Harness,“ [Online]. Available: <https://github.com/EleutherAI/lm-evaluation-harness>. [Zugriff am 04 09 2023].
- [27] Huggingface, „Text Generation Inference,“ [Online]. Available: <https://github.com/huggingface/text-generation-inference>. [Zugriff am 03 11 2023].
- [28] RunPod, „RunPod.io,“ [Online]. Available: <https://www.runpod.io/>. [Zugriff am 18 11 2023].
- [29] T. D. Younes Belkada, „A Gentle Introduction to 8-bit Matrix Multiplication for transformers at scale using Hugging Face Transformers, Accelerate and bitsandbytes,“ 17 08 2022. [Online]. Available: <https://huggingface.co/blog/hf-bitsandbytes-integration>. [Zugriff am 03 11 2023].
- [30] „GitHub: RWKV-LM,“ [Online]. Available: <https://github.com/BlinkDL/RWKV-LM>. [Zugriff am 01 09 2023].
- [31] N. Dey, G. Gosal, Z. Chen, H. Khachane, W. Marshall, R. Pathria, M. Tom und J. Hestness, „Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster,“ 2023 04 06. [Online]. Available: <https://arxiv.org/abs/2304.03208>. [Zugriff am 20 09 2023].
- [32] N. Dey, „Cerebras-GPT: A Family of Open, Compute-efficient, Large Language Models,“ 28 03 2023. [Online]. Available: <https://www.cerebras.net/blog/cerebras-gpt-a-family-of-open-compute-efficient-large-language-models/>. [Zugriff am 20 09 2023].
- [33] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser und C. Leahy, „The Pile: An 800GB Dataset of Diverse Text for Language Modeling,“ 31 12 2020. [Online]. Available: <https://pile.eleuther.ai/>. [Zugriff am 20 09 2023].
- [34] Cerebras, „Cerebras Model Zoo,“ 2023. [Online]. Available: <https://github.com/Cerebras/modelzoo>. [Zugriff am 20 09 2023].

Literaturverzeichnis

- [35] BigScience Workshop, „BLOOM: A 176B-Parameter Open-Access Multilingual Language Model,“ 09 11 2022. [Online]. Available: <https://arxiv.org/abs/2211.05100>. [Zugriff am 21 09 2023].
- [36] BigScience Workshop, „Megatron-DeepSpeed,“ [Online]. Available: <https://github.com/bigscience-workshop/Megatron-DeepSpeed>. [Zugriff am 21 09 2023].
- [37] BigScience Workshop, „bigscience,“ [Online]. Available: <https://github.com/bigscience-workshop/bigscience>. [Zugriff am 21 09 2023].
- [38] S. Biderman, H. Schoelkopf, Q. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika und O. v. d. Wal, „Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling,“ 03 04 2023. [Online]. Available: <https://arxiv.org/abs/2304.01373>. [Zugriff am 21 09 2023].
- [39] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, X. He, H. Hou, P. Kazienko, J. Kocon, J. Kong, B. Koptyra, H. Lau und K. Sr, „RWKV: Reinventing RNNs for the Transformer Era,“ 22 05 2023. [Online]. Available: <https://arxiv.org/abs/2305.13048>. [Zugriff am 21 09 2023].
- [40] Stability AI, „StableLM: Stability AI Language Models,“ [Online]. Available: <https://github.com/Stability-AI/StableLM>. [Zugriff am 03 11 2023].
- [41] Google Research, „T5X,“ [Online]. Available: <https://github.com/google-research/t5x>. [Zugriff am 03 11 2023].
- [42] google, „UL2,“ [Online]. Available: <https://huggingface.co/google/ul2>. [Zugriff am 21 09 2023].
- [43] MosaicML NLP Team, „MPT-30B: Raising the bar for open-source foundation models,“ 22 06 2023. [Online]. Available: <https://www.mosaicml.com/blog/mpt-30b>. [Zugriff am 21 09 2023].
- [44] MosaicML, „LLM Foundry,“ [Online]. Available: <https://github.com/mosaicml/llm-foundry>. [Zugriff am 03 11 2023].
- [45] MosaicML, „Streaming,“ [Online]. Available: <https://github.com/mosaicml/streaming>. [Zugriff am 03 11 2023].
- [46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave und G. Lample, „LLaMA: Open and Efficient Foundation Language Models,“ 27 02 2023. [Online]. Available: <https://arxiv.org/abs/2302.13971>. [Zugriff am 22 09 2023].
- [47] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu und J. F, „Llama 2: Open Foundation and Fine-Tuned Chat Models,“ 18 07 2023. [Online]. Available: <https://arxiv.org/abs/2307.09288>. [Zugriff am 22 09 2023].
- [48] OpenLM-Research, „OpenLLaMA: An Open Reproduction of LLaMA,“ [Online]. Available: https://github.com/openlm-research/open_llama. [Zugriff am 19 11 2023].
- [49] X. (. Geng, „EasyLM,“ [Online]. Available: <https://github.com/young-geng/EasyLM>. [Zugriff am 19 11 2023].
- [50] Cerebras, „Modelzoo: LICENSE,“ 25 10 2022. [Online]. Available: <https://github.com/Cerebras/modelzoo/blob/main/LICENSE>. [Zugriff am 24 09 2023].
- [51] BigScience Workshop, „BigScience: LICENSE,“ 08 05 2021. [Online]. Available: <https://github.com/bigscience-workshop/bigscience/blob/master/LICENSE>. [Zugriff am 24 09 2023].
- [52] C. M. F. Danish Contractor, „The BigScience RAIL License,“ [Online]. Available: <https://bigscience.huggingface.co/blog/the-bigscience-rail-license>. [Zugriff am 24 09 2023].
- [53] Eleuther AI, „Pythia: LICENSE,“ 26 04 2023. [Online]. Available: <https://github.com/EleutherAI/pythia/blob/main/LICENSE>. [Zugriff am 24 09 2023].

Literaturverzeichnis

- [54] BlinkDL, „RWKV: LICENSE,“ 14 09 2022. [Online]. Available: <https://github.com/BlinkDL/RWKV-LM/blob/main/LICENSE>. [Zugriff am 24 09 2023].
- [55] Stability AI, „StableLM: LICENCE,“ 19 04 2023. [Online]. Available: <https://github.com/Stability-AI/StableLM/blob/main/LICENSE>. [Zugriff am 24 09 2023].
- [56] Google Research, „Google Research: LICENSE,“ 05 10 2018. [Online]. Available: <https://github.com/google-research/google-research/blob/master/LICENSE>. [Zugriff am 24 09 2023].
- [57] Mosaic ML, „MPT-30B,“ 22 06 2023. [Online]. Available: <https://huggingface.co/mosaicml/mpt-30b#model-license>. [Zugriff am 24 09 2023].
- [58] Mosaic ML, „MPT-30B-Instruct,“ 22 06 2023. [Online]. Available: <https://huggingface.co/mosaicml/mpt-30b-instruct#model-license>. [Zugriff am 24 09 2023].
- [59] Meta AI, „Llama 2 license,“ 18 07 023. [Online]. Available: <https://ai.meta.com/llama/license/>. [Zugriff am 24 09 2023].
- [60] K. Hu, „ChatGPT sets record for fastest-growing user base - analyst note,“ Reuters, 02 02 2023. [Online]. Available: <https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/>. [Zugriff am 24 09 2023].
- [61] Meta AI, „Use-Policy,“ [Online]. Available: <https://ai.meta.com/llama/use-policy/>. [Zugriff am 24 09 2023].
- [62] S. Maffulli, „Meta’s LLaMa 2 license is not Open Source,“ Open Source Initiative, 20 07 2023. [Online]. Available: <https://blog.opensource.org/metals-llama-2-license-is-not-open-source/>. [Zugriff am 24 09 2023].
- [63] OpenLM-Research, „OpenLLaMA: LICENSE,“ 29 04 2023. [Online]. Available: https://github.com/openlm-research/open_llama/blob/main/LICENSE. [Zugriff am 24 09 2023].
- [64] Technology Innovation Institute, „Technology Innovation Institute: Organuzation Card,“ [Online]. Available: <https://huggingface.co/tiiuae>. [Zugriff am 24 09 2023].
- [65] S. Nellis, „AI chip startup Cerebras Systems raises \$250 million in funding,“ Reuters, 10 11 2021. [Online]. Available: <https://www.reuters.com/technology/ai-chip-startup-cerebras-systems-raises-250-million-funding-2021-11-10/>. [Zugriff am 26 09 2023].
- [66] S. Nellis und K. Hu, „Cerebras Systems signs \$100 million AI supercomputer deal with UAE's G42,“ Reuters, 27 07 2023. [Online]. Available: <https://www.reuters.com/technology/cerebras-systems-signs-100-mln-ai-supercomputer-deal-with-uaes-g42-2023-07-20/>. [Zugriff am 26 09 2023].
- [67] BigScience Workshop, „A one-year long research workshop,“ [Online]. Available: <https://bigscience.huggingface.co/>. [Zugriff am 26 09 2023].
- [68] EleutherAI, „EleutherAI: About us,“ [Online]. Available: <https://www.eleuther.ai/about>. [Zugriff am 26 09 2023].
- [69] StabilityAI, „Stability AI Announces \$101 Million in Funding for Open-Source Artificial Intelligence,“ Cision PR Newswire, 17 10 2022. [Online]. Available: <https://www.prnewswire.com/news-releases/stability-ai-announces-101-million-in-funding-for-open-source-artificial-intelligence-301650932.html>. [Zugriff am 26 09 2023].
- [70] StabilityAI, „We are the world’s leading open source generative AI company,“ [Online]. Available: <https://stability.ai/about>. [Zugriff am 26 09 2023].
- [71] Reuters, „Ex-Intel executives' startup aims to tackle spiraling costs of AI,“ 13 10 2021. [Online]. Available: <https://www.euronews.com/next/2021/10/13/mosaicml-tech>. [Zugriff am 26 09 2023].

Literaturverzeichnis

- [72] I. Lunden, „Databricks picks up MosaicML, an OpenAI competitor, for \$1.3B,“ TechCrunch, 26 06 2023. [Online]. Available: <https://techcrunch.com/2023/06/26/databricks-picks-up-mosaicml-an-openai-competitor-for-1-3b/>. [Zugriff am 26 09 2023].
- [73] Human Rights Watch, „United Arab Emirates,“ [Online]. Available: <https://www.hrw.org/middle-east/north-africa/united-arab-emirates>. [Zugriff am 01 10 2023].
- [74] LangChain, „LangChain Python Dokumentation,“ [Online]. Available: <https://python.langchain.com/>. [Zugriff am 16 11 2023].
- [75] Stanford NLP Team, „DSPy,“ [Online]. Available: <https://github.com/stanfordnlp/dspy>. [Zugriff am 16 11 2023].
- [76] G. Gerganov, „LLaMA.cpp,“ [Online]. Available: <https://github.com/ggerganov/llama.cpp>. [Zugriff am 16 11 2023].
- [77] LangChain, „LLM Chain,“ [Online]. Available: https://python.langchain.com/docs/modules/chains/foundational/llm_chain. [Zugriff am 16 11 2023].
- [78] Chroma, „Chroma,“ [Online]. Available: <https://docs.trychroma.com/>. [Zugriff am 16 10 2023].
- [79] Pinecone, „Pinecone,“ [Online]. Available: <https://www.pinecone.io/>. [Zugriff am 16 11 2023].
- [80] SBERT.net, „all-MiniLM-L6-v2,“ [Online]. Available: <https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>. [Zugriff am 16 11 2023].
- [81] M. Woolf, „The Problem With LangChain,“ 14 07 2023. [Online]. Available: <https://minimaxir.com/2023/07/langchain-problem/>. [Zugriff am 19 10 2023].
- [82] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. Yu, J. Gonzalez, H. Zhang und I. Stoica, „<https://vllm.ai/>,“ 20 06 2023. [Online]. Available: <https://vllm.ai/>. [Zugriff am 04 11 2023].
- [83] oobabooga, „Text generation web UI,“ [Online]. Available: <https://github.com/oobabooga/text-generation-webui>. [Zugriff am 04 11 2023].
- [84] HomeAssistant, „REST API,“ [Online]. Available: <https://developers.home-assistant.io/docs/api/rest/>. [Zugriff am 16 11 2023].
- [85] HomeAssistant, „REST API,“ 06 09 2023. [Online]. Available: <https://developers.home-assistant.io/docs/api/rest/>. [Zugriff am 29 10 2023].
- [86] V. Singh, „A Guide to Controlling LLM Model Output: Exploring Top-k, Top-p, and Temperature Parameters,“ 22 09 2023. [Online]. Available: <https://ivibudh.medium.com/a-guide-to-controlling-llm-model-output-exploring-top-k-top-p-and-temperature-parameters-ed6a31313910>. [Zugriff am 11 11 2023].
- [87] ubernostrum, „webcolors 1.13,“ [Online]. Available: <https://pypi.org/project/webcolors/>. [Zugriff am 16 11 2023].

Selbstständigkeitserklärung

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben. Die Stellen der Hausarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlichgemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen, sowie für Quellen aus dem Internet. Die Arbeit hat mit gleichem bzw. in wesentlichen Teilen gleichem Inhalt noch keiner Prüfungsbehörde vorgelegen. Weiterhin stimme ich zu, dass von der Arbeit eine elektronische Kopie gefertigt und gespeichert werden darf, um eine Überprüfung mittels einer Plagiatsoftware zu ermöglichen.

Stralsund, den 04 Dezember 2023