

Inhaltsverzeichnis

1	Einleitung	2
1.1	LangChain	2
1.2	Hugging Face	2
1.3	Streamlit	2
1.4	DeepEval & Confident AI	3
2	Große Sprachmodelle	3
2.1	Training und Anpassung	4
2.1.1	Pre-Training	4
2.1.2	Fine-Tuning	4
2.1.3	Prompt Engineering	4
2.1.4	RAG	4
3	Retrieval Augmented Generation	5
3.1	Halluzination	5
3.2	Out Of Date	6
3.3	Keine Quellenangaben	6
3.4	Zusammenfassung	6
3.5	Workflow	7
4	Implementierung	8
4.1	Datensatz	8
4.2	Datenvorverarbeitung	9
4.3	Text Splitter	10
4.4	Embeddings-Modell	10
4.5	Chat-Modell	11
4.6	Vektor Datenbank	11
4.7	Retriever	12
4.8	Prompt-Templates	13
4.9	Chains	13
4.10	Frontend	14
5	Evaluation	14
5.1	Evaluierung des Retrievers	14
5.1.1	Contextual Precision Metric	14
5.1.2	Contextual Recall Metric	15
5.1.3	Contextual Relevancy Metric	15
5.2	Evaluierung der Generierung	15
5.2.1	Answer Relevancy Metric	15
5.2.2	Faithfulness Metric	15
5.3	Implementierung	16
6	Endprodukt	17
7	Vergleich zu anderen RAG Pipelines	18
8	Learnings und Ausblick	19
9	Anhang	21
9.1	Screenshots der Anwendung	21
9.2	Literaturverzeichnis	23

1 Einleitung

Im Rahmen des Wahlpflichtkurses “Artificial Intelligence and Adaptive Systems” ist eine Chatbot Anwendung unter Verwendung eines KI-Frameworks namens “Retrieval Augmented Generation” (RAG) entstanden. Umgesetzt wurde die Anwendung mit Python (v3.12), LangChain (v0.2.7), Hugging Face, Streamlit (v1.36.0) und DeepEval (v0.21.65). Zur Versionskontrolle wurde Git verwendet.

In den folgenden Kapiteln wird erklärt, um was genau es sich bei einer RAG Architektur handelt, wie das Projekt im Detail umgesetzt und welches Fazit am Ende gezogen wurde.

An dieser Stelle möchte ich darauf hinweisen, dass die ersten zwei Kapitel Große Sprachmodelle und Retrieval Augmented Generation größtenteils (unter Punkt 3.4 erläutere ich kurz, weshalb im Rahmen dieses Projektes kein Fine-Tuning durchgeführt wurde) übersprungen werden können, wenn die Funktionsweisen bereits bekannt sind.

1.1 LangChain

Bei [LangChain](#) handelt es sich um ein Open-Source-Framework für die Entwicklung von Anwendungen, die auf großen Sprachmodellen (siehe Kapitel 2) basieren. Durch die Verwendung dieses Frameworks wird die Komplexität der Integration und die Verwendung von großen Sprachmodellen vereinfacht (vgl. Kondrashchenko 2023). LangChain umfasst sieben Hauptmodule: Prompts, Models, Memory, Indexes, Chains, Agents und Callbacks (vgl. Kalra 2024) - auf die nähere Beschreibung der Module wird an dieser Stelle verzichtet.

In der entstandenen Chatbot Anwendung werden die Module Prompts (siehe Kapitel 4.8), Models (siehe Kapitel 4.4 und 4.5) und Chains (siehe Kapitel 4.9) verwendet.

1.2 Hugging Face

Bei [Hugging Face](#) handelt es sich um eine Plattform, die Tools und Ressourcen für maschinelles Lernen und natürliche Sprachverarbeitung (NLP) bereitstellt (vgl. Lutkevich 2023a). Die Plattform ist bekannt für ihre Transformers-Bibliothek, die vortrainierte Modelle für verschiedene NLP-Aufgaben bietet. Der Model Hub ermöglicht das Teilen und Finden von vortrainierten Modellen, und die Datasets-Bibliothek bietet Zugang zu vielen NLP-Datensätzen. Hugging Face fördert eine aktive Entwickler- und Forscher-Community, die die Entwicklung und den Einsatz von NLP-Modellen vorantreibt (vgl. ebd.).

1.3 Streamlit

[Streamlit](#) ist ein Open-Source-Framework, das die Erstellung interaktiver Webanwendungen für Datenwissenschaft und maschinelles Lernen mit minimalem Python-Code ermöglicht. Es bietet einfache Integration mit Python-Bibliotheken, interaktive Widgets, Echtzeit-Updates und einfache Möglichkeiten zum Teilen der Anwendungen (vgl. Streamlit Docs o. D.).

1.4 DeepEval & Confident AI

[DeepEval](#) ist ein Open-Source-Framework, das entwickelt wurde, um große Sprachmodelle (engl. Large Language Models (LLMs)) umfassend zu evaluieren. Es geht über traditionelle Metriken hinaus und integriert eine Vielzahl von Bewertungstechniken, um eine ganzheitliche Beurteilung der Leistung von LLMs sicherzustellen. DeepEval verwendet eine modulare Architektur, um die Ausgaben von LLMs ähnlich wie bei Pytest durch Unit Tests zu überprüfen. Dadurch können flexible und anpassbare Evaluierungsprotokolle erstellt werden, die auf spezifische Bedürfnisse und Kontexte zugeschnitten sind (vgl. [What is DeepEval? Features & Examples 2024](#)).

[Confident AI](#) wurde für LLM-Teams entwickelt, um LLM-Anwendungen von der Entwicklung bis zur Produktion qualitativ zu sichern (vgl. [Introduction | DeepEval - the Open-Source LLM Evaluation Framework 2024](#)). Es ist eine All-in-One-Plattform, die das volle Potenzial von DeepEval freisetzt, indem sie es u.a. folgendes ermöglicht:

- Kontinuierliche Evaluation von LLM-Anwendungen in der Produktion
- Debugging von LLM-Anwendungen während der Evaluierung
- Standardisierung und Zentralisierung von Evaluierungsdatensätzen in der Cloud

2 Große Sprachmodelle

Große Sprachmodelle, auch Large Language Models (LLMs) genannt, sind KI-Modelle für die Verarbeitung natürlicher Sprache. Sie basieren auf neuronalen Netzwerken, insbesondere der Transformer-Architektur, und sind mit riesigen Datensätzen vortrainiert (vgl. [Luber 2023](#)).

Die Transformer-Architektur besteht aus drei Hauptkomponenten: dem Encoder, den Aufmerksamkeitsmechanismen und dem Decoder (vgl. [Was sind Large Language Models \(LLMs\)? | Databricks o. D.](#)).

Der Encoder wandelt Text in numerische Tokens um und erstellt sinnvolle Repräsentationen, die semantisch ähnliche Wörter nahe beieinander platzieren. Aufmerksamkeitsmechanismen ermöglichen es dem Modell, sich auf relevante Teile des Textes zu konzentrieren, indem es Beziehungen zwischen den Tokens analysiert. Der Decoder wandelt die Tokens zurück in Wörter und prognostiziert dabei das nächste und übernächste Wort, basierend auf dem trainierten Wissen des Modells. Diese Architektur erlaubt LLMs, komplexe Aufgaben wie Fragenbeantwortung, Übersetzungen und semantische Analysen durchzuführen, indem sie den Kontext und die Struktur natürlicher Sprache verstehen und generieren können (vgl. [ebd.](#)).

2.1 Training und Anpassung

Das Training eines Large Language Models (LLMs) ist ein mehrstufiger Prozess, der darauf abzielt, das Modell mithilfe großer Mengen an Textdaten zu trainieren, damit es natürliche

Sprache verstehen, generieren und bearbeiten kann (vgl. Safar 2023). Der erste Schritt des Trainings ist das sogenannte Pre-Training, häufig gefolgt vom Fine-Tuning.

2.1.1 Pre-Training


Beim Pre-Training wird das LLM von Grund auf mit einem umfangreichen Corpus an Textdaten trainiert, ohne spezifische Anweisungen für bestimmte Aufgaben (vgl. Luber 2023). Das Modell lernt, indem es Muster, Beziehungen und Abhängigkeiten zwischen Wörtern und Sätzen in den Trainingsdaten erkennt. Der Fokus liegt darauf, die Modellparameter so zu optimieren, dass das Modell möglichst präzise Vorhersagen über die Wahrscheinlichkeit von Wörtern und Wortsequenzen treffen kann. Das Pre-Training ist ressourcenintensiv und erfordert leistungsfähige Rechner und viel Speicherplatz (vgl. ebd.).

2.1.2 Fine-Tuning

Nach dem Pre-Training erfolgt die Feinabstimmung des Modells für spezifische Aufgaben oder Anwendungsfälle. Hierbei werden die bereits vortrainierten Modellparameter weiter optimiert, indem das Modell mit spezifischen Daten für die gewünschte Aufgabe oder Domäne verfeinert wird. Dies kann durch überwachtes Lernen mit gelabelten Daten oder durch bestärkendes Lernen erfolgen, bei dem das Modell positive und negative Rückmeldungen erhält, um seine Leistung zu verbessern (vgl. ebd.).

Weitere Möglichkeiten, um ein LLM an spezifische Anforderungen anzupassen, bietet das Prompt Engineering sowie die Retrieval Augmented Generation (RAG) (vgl. Was sind Large Language Models (LLMs)? | Databricks o. D.).

2.1.3 Prompt Engineering

Ein Prompt ist für ein Sprachmodell eine Reihe von Anweisungen oder Eingaben, die ein Benutzer bereitstellt, um das Modell bei der Erstellung relevanter und zusammenhängender sprachbasierter Ausgaben zu führen. Prompt Engineering bezeichnet die Formulierung spezieller Prompts zur Steuerung des LLM-Verhaltens und macht eine schnelle und spontane Modellführung möglich (vgl. Prompts |  LangChain o. D.).

2.1.4 RAG

“Retrieval Augmented Generation (RAG) ist ein Architekturkonzept, mit dem sich die Effizienz von LLM-Anwendungen (Large Language Model) durch Nutzung kundenspezifischer Daten verbessern lässt. Zu diesem Zweck werden Daten und/oder Dokumente, die für eine Frage oder Aufgabe relevant sind, abgerufen und als Kontext für das LLM bereitgestellt.” (Was sind Large Language Models (LLMs)? | Databricks o. D.).

Das nächste Kapitel wird genauer auf RAG eingehen und erklären, weshalb sich dieses Konzept vor allem in Chatbots und Q&A-Systemen bewährt hat.

3 Retrieval Augmented Generation

Wie in der Einleitung bereits erwähnt, basiert die entstandene Chatbot Anwendung auf einer Retrieval Augmented Generation Architektur. Dabei handelt es sich um ein KI-Framework zum Abrufen von Fakten aus einer externen Wissensdatenbank, um so die Genauigkeit und Zuverlässigkeit generativer KI-Modelle zu verbessern (vgl. Martineau 2024). Doch weshalb ergibt die Verwendung eines solchen Frameworks, in Kombination mit Fine-Tuning, Sinn?

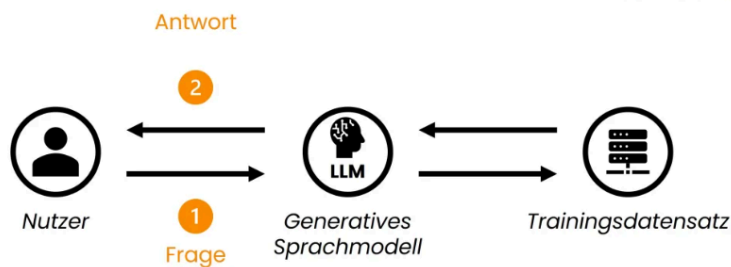


Abbildung 1 zeigt eine simplifizierte Darstellung eines Abfrageprozesses an ein Large Language Model (LLM).

Abbildung 1: Simplifizierte Darstellung eines Abfrageprozesses an ein LLM
(<https://ambersearch.de/de/retrieval-augmented-generation/>)

Der Nutzer gibt über ein User Interface eine Frage ein, die an ein generatives Sprachmodell weitergegeben wird. Basierend auf den Trainingsdaten generiert das Sprachmodell eine Antwort, die schließlich an den Nutzer zurückgegeben wird. Der alleinige Zugriff auf das parametrisierte Wissen des Large Language Models bringt einige Herausforderungen mit sich (vgl. Tran/Tran 2023) und im Folgenden wird auf drei genauer eingegangen.

3.1 Halluzination

KI-Halluzinationen treten auf, wenn Large Language Models falsche Informationen erzeugen, die von externen Fakten oder kontextueller Logik abweichen können. Diese Halluzinationen wirken oft plausibel, da LLMs darauf ausgelegt sind, kohärente Texte zu produzieren, jedoch fehlt ihnen ein Verständnis der zugrunde liegenden Realität. Ursachen für Halluzinationen können in der Datenqualität liegen, wie auch in den Methoden der Generierung und dem Eingabekontext (vgl. Lutkevich 2023). Es gibt mehrere Arten von KI-Halluzinationen, darunter:

- Satzwidrspruch: Hier widerspricht ein erzeugter Satz einem vorherigen Satz, z.B. indem er eine falsche Information einfügt.
- Widerspruch zur Eingabeaufforderung: Der erzeugte Satz steht im Widerspruch zur eigentlichen Aufforderung, z.B. indem er eine unpassende Antwort gibt.
- Faktischer Widerspruch: Hier wird eine fiktive Information als Tatsache dargestellt, z.B. durch die Erfindung von Details.
- Irrelevante oder zufällige Halluzinationen: Dabei werden zufällige Informationen erzeugt, die nichts mit der Eingabe oder Ausgabe zu tun haben.

RAG integriert einen Retrieval-Schritt, bei dem relevante Informationen aus externen Quellen abgerufen werden. Diese externen Daten dienen als Kontext für die generative Phase des Modells. Durch die Nutzung aktueller und verifizierter Informationen aus

zuverlässigen Quellen reduziert RAG die Wahrscheinlichkeit, dass das Modell falsche oder irreführende Inhalte generiert (vgl. Martineau 2024).

3.2 Out Of Date

Ein häufiges Problem bei generativen KI-Modellen ist die Verwendung veralteter Pre-Training-Daten. Dies führt dazu, dass die Modelle veraltete Informationen liefern können, insbesondere in dynamischen Umgebungen, in denen sich Fakten und Ereignisse schnell ändern (vgl. Kalra 2024b).

Die Implementierung von Retrieval-Augmented Generation (RAG) adressiert dieses Problem, indem sie sicherstellt, dass das Modell Zugang zu den aktuellsten und zuverlässigsten Informationen hat. Durch die Integration eines Retrievers, der bei jeder Anfrage relevante Daten aus externen Quellen abrufen, wird sichergestellt, dass die generierten Antworten auf aktuellen Informationen basieren. Dies reduziert die Wahrscheinlichkeit, dass das Modell veraltete oder ungenaue Daten verwendet, und erhöht somit die Genauigkeit und Vertrauenswürdigkeit der Antworten (vgl. ebd.).

3.3 Keine Quellenangaben

Ein weiteres Problem generativer KI-Systeme ist die Schwierigkeit, die Quellen und Grundlagen für die bereitgestellten Informationen nachzuvollziehen. In vielen Fällen geben diese Systeme Antworten oder erzeugen Inhalte, ohne klar anzugeben, woher die Informationen stammen oder wie sie generiert wurden.

RAG bietet hier eine Lösung, indem es den Benutzern ermöglicht, die Quellen der generierten Antworten nachzuvollziehen. Indem relevante Informationen aus externen Quellen abgerufen und als Kontext für die generativen Modelle verwendet werden, können Benutzer den Ursprung der gelieferten Informationen überprüfen und deren Vertrauenswürdigkeit beurteilen (vgl. Merritt 2024).

3.4 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass Fine-Tuning und RAG zwei Ansätze zur Verbesserung von LLMs darstellen. Fine-Tuning optimiert ein Modell durch Anpassung an spezifische Domänen mittels Training mit domänenspezifischen Daten (vgl. Tran/Tran 2023b). RAG verbessert die Genauigkeit durch Integration externer Informationsquellen. Beide Ansätze können und sollten kombiniert werden, um LLMs leistungsfähiger zu machen (vgl. Bhavsar o. D.).

Da die entstandene Chatbot-Anwendung keine Anforderungen stellt, die nicht gut von einem vortrainierten Modell erfüllt werden können und kein domänenspezifisches Wissen nötig ist, reicht die Implementierung einer RAG Architektur aus.

3.5 Workflow

Damit Daten aus einer externen Datenbank abgerufen werden können, muss es zunächst Daten in einer solchen Datenbank geben.

Abbildung 2 zeigt einen Workflow, unter der Verwendung von LangChain, für die Verarbeitung und Speicherung von Dokumenten in einer Vektor-Datenbank.

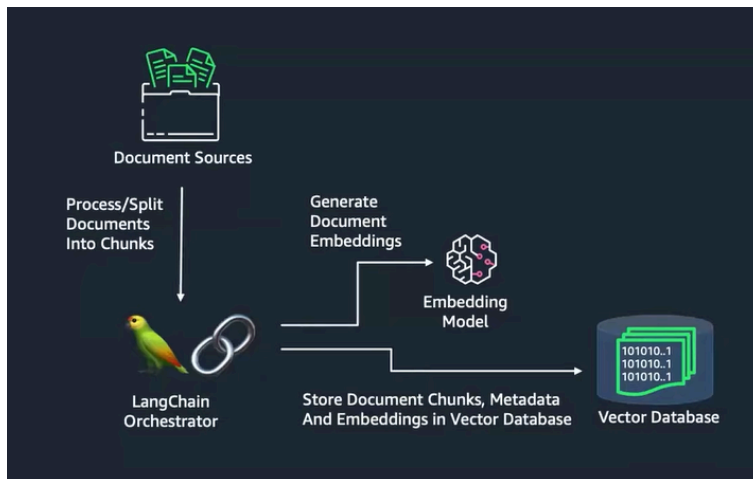


Abbildung 2: Vektor-Datenbank mit Daten

befüllen (https://www.youtube.com/watch?v=J_tCD_J6w3s)

Dafür werden die Dokumente zunächst in “Chunks” (Stücke) aufgeteilt, zu denen im nächsten Schritt mittels eines Embedding Models Embeddings (numerische Darstellungen von Informationen, die ihre semantische Bedeutung erfassen) generiert werden. Im letzten Schritt werden die Chunks mit ihren Metadaten sowie den zugehörigen Embeddings in der Vektor-Datenbank gespeichert. Der LangChain Orchestrator koordiniert und steuert die verschiedenen Schritte.

Sobald die Datenbank befüllt ist, können die dort vorhandenen Daten mittels RAG Framework abgerufen und dem LLM als Kontext mitgegeben werden.

Abbildung 3 zeigt, wie eine RAG Architektur in eine traditionelle generative Phase eines LLMs integriert wird und so die Antwort des LLMs auf einen User Prompt sehr viel zuverlässiger macht.

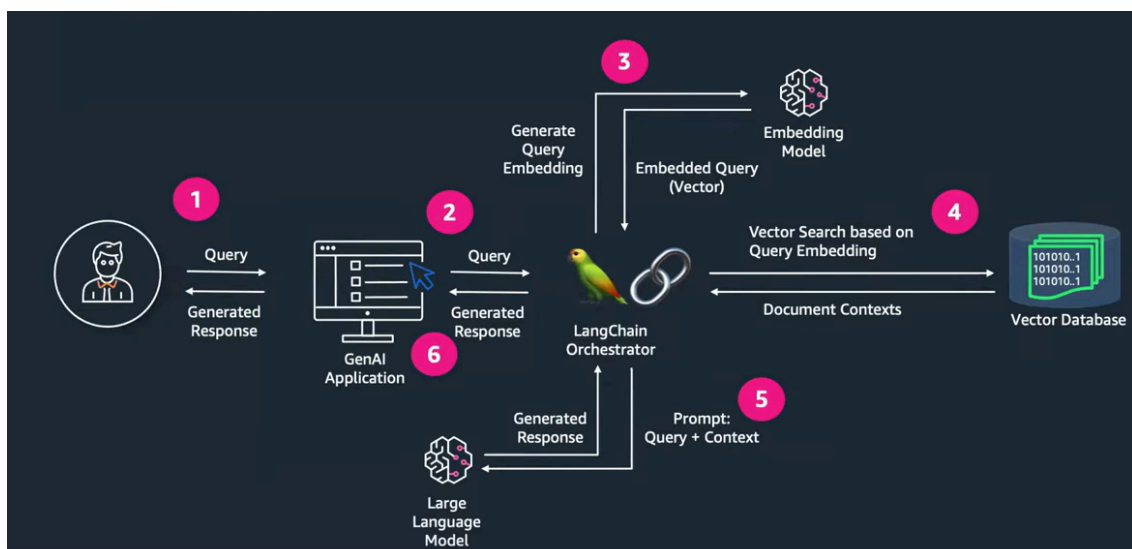


Abbildung 3: Workflow einer einfachen RAG Pipeline (https://www.youtube.com/watch?v=J_tCD_J6w3s)

Die Schritte eins und zwei stellen dar, wie ein Benutzer eine Anfrage (Query) an das System stellt, zu welcher im dritten Schritt mittels eines geeigneten Modells ein Embedding für die anschließende Vektorensuche generiert wird. Im vierten Schritt wird, basierend auf dem generierten Embedding, in der Vektordatenbank eine Vektorsuche durchgeführt, um relevante Inhalte zu der ursprünglichen Anfrage zu finden. Schritt fünf zeigt, dass die Anfrage mit dem gefundenen Kontext zu einem Prompt kombiniert und dieser dem LLM übergeben wird. Basierend auf dem Prompt generiert das LLM eine Antwort, welche schließlich an den Benutzer zurückgegeben wird. Alle diese Schritte werden vom LangChain Orchestrator koordiniert.

4 Implementierung

Die Implementierungsphase begann mit der Bearbeitung von Tutorials von LangChain zu der Entwicklung von [Chatbots](#) und [Q&A-Anwendungen mit RAG](#), gefolgt von der Recherche nach benötigten Komponenten:

- Datensatz, zur Befüllung der Vektor Datenbank
- Library zur Datenvorverarbeitung
- Text Splitter zur Aufteilung langer Texte in kleinere, verwertbare Segmente
- Ein Embeddings-Modell zur Umwandlung von Textdateien in Vektoren
- Ein Chat-Modell zur Generierung von Antworten
- Eine Vektor Datenbank
- Einen Retriever zur Suche und Rückgabe relevanter Dokumente aus der Vektor-Datenbank
- Geeignete Prompt-Templates zur strukturierten Formulierung von Anfragen an das Sprachmodell
- Geeignete Chains zur Verkettung verschiedener Verarbeitungsschritte im RAG-Prozess
- Ein Frontend

4.1 Datensatz

Nach einem Datensatz suchte ich auf [kaggle.com](https://www.kaggle.com), einer Online-Plattform für Datenwissenschaft und maschinelles Lernen, auf der u.a Daten entdeckt, geteilt und analysiert werden können.

Zunächst entschied ich mich für den Datensatz [Netflix Movies and TV Shows](#), da es in CSV Format vorliegt (wichtig im Schritt der Datenvorverarbeitung mit Pandas), als "well maintained" eingestuft wurde und auf der Plattform sehr beliebt ist.

Die Datenvorverarbeitung erwies sich aufgrund der vielen Spalten und der fehlenden Erfahrung als relativ arbeitsintensiv. Das tatsächliche Problem, auf welches ich erst am Ende des Projektes gestoßen bin, lag aber in der Evaluation. Der gewählte Datensatz besteht aus vielen Fakten, zu denen es keine Fragen gibt. Für den Schritt der Evaluation bedeutet das, dass ich mir händisch einen weiteren Datensatz mit passenden Fragen zu den Fakten hätte erstellen müssen.

Daher entschied ich mich am Ende des Projektes dazu, nochmal nach einem neuen Datensatz zu suchen und fand einen zu [Taylor Swift](#).

4.2 Datenvorverarbeitung

Die Datenvorverarbeitung ist ein zentraler Schritt zur Vorbereitung von Rohdaten für Analyse und maschinelles Lernen. Der Hauptzweck besteht darin, die Datenqualität zu verbessern, Diskrepanzen zu lösen sowie Korrektheit und Konsistenz sicherzustellen (vgl. Gryczka 2024). Durch sorgfältige Datenvorverarbeitung werden u.a. Vorhersagen in Machine-Learning-Modellen verbessert und die Interpretierbarkeit der Daten erhöht.

Obwohl die Datenvorverarbeitung zeitaufwendig ist und das Risiko von Datenverzerrungen oder Datenlecks birgt, bietet sie wesentliche, oben genannte, Vorteile (vgl. Gavrilova 2024). Die wichtigsten Schritte der Datenvorverarbeitung umfassen:

- Datenbereinigung: Dies beinhaltet die Erkennung und Behebung von Fehlern, Diskrepanzen und Ausreißern sowie die Förderung der Konsistenz durch Standardisierung und Angleichung der Daten.
- Datenintegration: Hierbei werden verschiedene Datenquellen zusammengeführt und fehlende Werte sowie Duplikate behandelt.
- Datentransformation: Diese Phase umfasst die Anpassung der Datenformate für die Analyse, die Skalierung von Merkmalen und die Kodierung kategorischer Variablen.
- Dimensionsreduktion: Zur Verringerung der Komplexität des Datensatzes wird der Fokus auf die wichtigsten Aspekte der Informationen gelegt.

Um die Daten für die entstehende Chatbot Anwendung zu laden und zu reinigen, wurde die Open-Source-Bibliothek [Pandas](#) verwendet.

Da für die entstandene Anwendung kein Fine-Tuning notwendig war, mussten auf dem ausgewählten Datensatz nicht alle oben genannten Schritte ausgeführt werden. Das Ziel war es lediglich, die Daten so aufzubereiten, dass der Retriever effizient und zuverlässig relevante Dokumente finden kann, sodass das LLM auf dieser Basis eine Antwort für den Benutzer generieren kann.

Folgende Schritte waren u.a. notwendig:

- Den Werten jeder Zeile wurde der entsprechende Spaltenname hinzugefügt, um den Kontext des Wertes zu bewahren
- NaN Werte wurden mit "unknown" ersetzt (später wurden solche Zeilen komplett gelöscht)
- Alle Spalten (ID, Question, Answer) wurden zu einer einzigen Spalte zusammengeführt, um den späteren Abruf zu vereinfachen
- Der Datensatz wurde in kleinere, sich überlappende Textblöcke aufgeteilt

4.3 Text Splitter

Beim Umgang mit langen Dokumenten kann es notwendig sein, den Text in kleinere, semantisch sinnvolle Abschnitte zu unterteilen, damit diese beispielsweise in das Kontextfenster des verwendeten Modells passen (vgl. Text Splitters | [🔗](#) LangChain o. D.). Mit Text Splittern ist genau dies möglich. Nachdem der Text in kleinere Abschnitte unterteilt wurde, werden diese kleinen Abschnitte zu größeren Einheiten kombiniert, bis eine bestimmte Größe (Chunk Size) erreicht ist, die durch eine spezifische Funktion gemessen wird. Sobald diese Größe erreicht ist, wird der Abschnitt als eigenständiges Textstück (Dokument) betrachtet und der Prozess beginnt von neuem, wobei eine gewisse Überlappung (Chunk Overlap) beibehalten wird, um den Kontext zwischen den Abschnitten zu wahren (vgl. ebd.). Dadurch kann der Text Splitter entlang zweier Achsen angepasst werden:

- der Art und Weise, wie der Text aufgeteilt wird
- der Methode zur Messung der Abschnittsgröße

In der Anwendung wird der [RecursiveCharacterTextSplitter](#) verwendet, welcher für allgemeinen Text empfohlen wird. Dieser Text Splitter wird durch eine Liste von Zeichen parametrisiert und versucht, den Text der Reihe nach an diesen Zeichen zu teilen, bis die Abschnitte klein genug sind.

An die Chunk Size (maximale Anzahl von Zeichen, die ein Chunk enthalten kann) und den Chunk Overlap (Anzahl der Zeichen, die zwischen zwei benachbarten Chunks überlappen sollten) habe ich mich mit Hilfe des Tools [ChunkViz](#) langsam und iterativ herangetastet.


4.4 Embeddings-Modell

Einbettungen (Embeddings) sind numerische Darstellungen von Informationen, die ihre semantische Bedeutung erfassen und Vergleiche zwischen verschiedenen Informationsstücken, wie z.B. Text ermöglichen (vgl. Was ist Einbettung? – Einbettungen in Machine Learning erklärt – AWS o. D.).

In einer RAG-Architektur wird ein Embeddings-Modell verwendet, um die semantische Ähnlichkeit zwischen Benutzerabfragen und relevanten Dokumenten zu erfassen. Für die Chatbot-Anwendung wurde das Sentence Transformers Modell [all-MiniLM-L6-v2](#) als Embeddings-Modell mittels des HuggingFaceEmbeddings Wrapper von LangChain verwendet. Bei diesem Modell handelt es sich um ein kleines aber leistungsstarkes Modell (384 Dimensionen), welches darauf optimiert ist, semantisch ähnliche Sätze im Vektorraum nahe beieinander und unähnliche Sätze weit voneinander entfernt zu platzieren. Eine Einschränkung des Modells besteht in der Begrenzung auf 256 Wortteile - längerer Text wird abgeschnitten (vgl. sentence-transformers/all-MiniLM-L6-v2 · Hugging Face 2001).

4.5 Chat-Modell

Die Anwendung verwendet das LLM [Mistral-7B-Instruct-v0.2](#) mittels eines Wrappers von LangChain als Chat-Modell.

Bei einem Chat-Modell handelt es sich um eine spezialisierte Form eines LLM, das für den Zweck des Dialogs optimiert ist. Es verwendet Chat-Nachrichten als Eingaben und Ausgaben (vgl. Chat Models |  LangChain o. D.).

Zu Beginn der Implementierung wurde auf [Hugging Face](#) nach Modellen gesucht, die auf die Aufgabe der Textgenerierung (Text Generation) spezialisiert sind. Diese Aufgabe ist dafür gedacht, Texte basierend auf Prompts zu generieren (vgl. What is text generation? | IBM o. D.). Da Chatbot-Anwendungen Antworten zu eingegebenen Fragen generieren müssen, ist diese Aufgabe genau die richtige. Zum Zeitpunkt der Auswahl des Modells war das oben genannte das beliebteste. Außerdem verfügt dieses LLM über ein großes Kontextfenster (32k) und ist gut dokumentiert.

Folgende Parameter wurden übergeben, um das Verhalten und die Ausgabe des LLM zu beeinflussen:

- `max_new_tokens` = 512
- `top_k` = 50
- `temperature` = 0.3
- `repetition_penalty` = 1.1



`max_new_tokens` gibt die maximale Anzahl neuer Tokens an, die das Modell generieren kann. Eine höhere Anzahl ermöglicht längere und detailliertere Antworten, was für eine interaktive Konversation sinnvoll sein kann (vgl. Ai 2024).

Bei `top_k` handelt es sich um eine Technik, bei der das Modell bei der Generierung eines Tokens nur die wahrscheinlichsten `k` Tokens berücksichtigt (vgl. Saleem 2024). Mit `k` = 50 wird die Generierung relevanter Antworten gefördert.

Die `temperature` ist ein Parameter, der die Diversität der Generierung steuert. Eine hohe Temperatur führt zu unvorhersehbaren und kreativen Ergebnissen, während eine niedrige Temperatur zu deterministischeren und konservativeren Ausgaben führt (vgl. ebd.).

Die `repetition_penalty` steuert wie stark das Modell dazu neigt, bereits verwendete Tokens zu wiederholen. Ein Wert über 1 reduziert die Wahrscheinlichkeit von Wiederholungen und fördert so vielfältige und abwechslungsreiche Antworten (vgl. Ai 2024).

4.6 Vektor Datenbank

Eine der gängigsten Methoden zur Speicherung von und Suche in unstrukturierten Daten besteht darin, diese einzubetten und die resultierenden Vektoren mittels einer Vektor Datenbank zu speichern. Zum Zeitpunkt der Abfrage wird die Anfrage mittels eines Embedding Modells eingebettet und es werden diejenigen Vektoren aus der Datenbank abgerufen, die der eingebetteten Anfrage am ähnlichsten sind (vgl. Vector stores |   LangChain o. D.). Den beschriebenen grundlegenden Workflow einer Vektor Datenbank veranschaulicht die Abbildung 4.

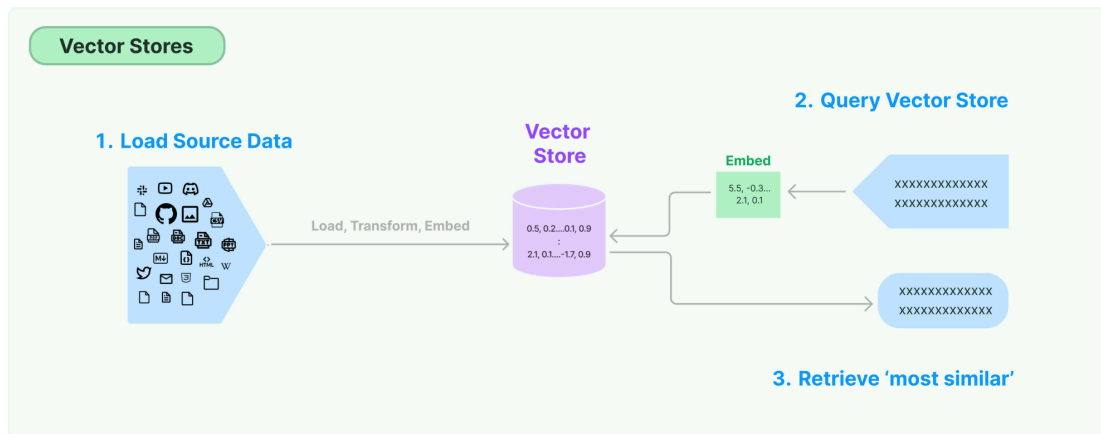


Abbildung 4: Grundlegender Workflow einer Vektor-Datenbank

(https://python.langchain.com/v0.1/docs/modules/data_connection/vectorstores/)

LangChain bietet verschiedene Integrationen, wovon sich für die Umsetzung der Chatbot-Anwendung zwei genauer angeschaut wurden: [FAISS](#) und [Chroma](#).

FAISS (Facebook AI Similarity Search), zeichnet sich durch eine schnelle Abrufgeschwindigkeit und hohe Genauigkeit aus, ist aber ressourcenintensiv. Chroma bietet mehr Flexibilität, eine einfachere Integration von Embedding-Modellen, ist für LLM-Anwendungen optimiert, jedoch schwerer skalierbar (vgl. FAISS vs Chroma: Vector Storage Battle 2024).

Obwohl sich Chroma für die kleine entstandene Anwendung besser eignen würde, wurde während der Entwicklung mit FAISS gearbeitet, da für die Verwendung von Chroma Microsoft Build Tools installiert werden müssen und FAISS sofort nutzbar ist.

4.7 Retriever

Ein Retriever ist eine Schnittstelle, die Dokumente (in Form einer Liste) anhand einer unstrukturierten Abfrage zurückgibt (vgl. Retrievers | [LangChain](#) o. D.).

LangChain bietet verschiedene Arten von Retrievern für unterschiedliche Anwendungsfälle. Für die entstandene Anwendung wurde ein vector store-backed Retriever verwendet. Diese Art von Retriever verwendet einen Vektorspeicher, um Dokumente abzurufen und dient als leichtgewichtiger Wrapper um die Vektorspeicherklasse, um diese an die Retriever-Schnittstelle anzupassen (vgl. Vector store-backed retriever | [LangChain](#) o. D.). Der vector store-backed Retriever nutzt die von einem Vektorspeicher implementierten Suchmethoden, wie z.B. die Ähnlichkeitssuche und Maximal Marginal Relevance (MMR), um die Texte im Vektorspeicher zu durchsuchen und abzufragen.

Der verwendete Retriever arbeitet mit Schwellenwert-basierter Suche (`similarity_score_threshold`), was bedeutet, dass nur Dokumente über einem bestimmten Ähnlichkeitswert zurückgegeben werden. Dieser Ähnlichkeitswert (`score_threshold`) wurde auf 0.5 eingestellt. Die Skala reicht von 0 bis 1, wobei 1 perfekte Übereinstimmung bedeutet (vgl. ebd.). Außerdem ist der Retriever so eingestellt, dass maximal vier Dokumente

zurückgegeben werden, um die Menge der Informationen, die verarbeitet werden müssen, zu begrenzen.

4.8 Prompt-Templates

Sprachmodelle benötigen Prompts, die aus Anweisungen (definieren die gewünschte Antwort oder das Verhalten des Modells), Kontext (bietet zusätzliche Informationen), Benutzereingaben und Ausgabeindikatoren (markiert den Beginn der Antwort des Modells) bestehen, um ihre Funktion zu erfüllen (vgl. Prompts | [LangChain](#) o. D.).

Ein Prompt-Template (Werkzeug von LangChain) strukturiert diese Elemente und ermöglicht systematisches Erzeugen, Teilen und Wiederverwenden von Prompts. Außerdem verbessern sie die Lesbarkeit im Code und trennen die Formatierung von der Modellauflogik (vgl. Sahota 2024).

Für Chat-Modelle bietet LangChain das [ChatPromptTemplate](#), das für die Formatierung von Listen von Chat-Nachrichten entwickelt wurde und verschiedene Nachrichtenklassen (AIMessage, HumanMessage etc.) unterstützt.

In der Chatbot-Anwendung werden das PromptTemplate (für die Umformulierung der Benutzereingabe) und das ChatPromptTemplate (für den Chatbot-Dialog) verwendet. Das ChatPromptTemplate berücksichtigt historische Nachrichten sowie den aktuellen Kontext.

4.9 Chains

Ein zentrales Element von LangChain sind die sogenannten Chains, welche die Integration von mehreren Komponenten ermöglichen. Beispiele sind die Kombination von Prompt-Templates mit LLMs, die Umsetzung von RAG-Architekturen, die Verbindung mehrerer LLMs in einer sequenziellen Reihenfolge und die Integration von LLMs mit beispielsweise Chat-Verläufen (vgl. Banjara 2024).

Für die Implementierung der entstandenen Chat-Anwendung wurden mehrere Chains (Übersicht [hier](#)) verwendet:

- LLM Chain: Diese Chain kombiniert ein LLM mit einem Prompt, indem der Prompt mit einem Eingabetext "ausgefüllt" wird und das LLM zur Generierung einer Antwort verwendet wird.
- StuffDocumentsChain: Hier werden mehrere Dokumente kombiniert und in den Kontext eines LLM integriert.
- RetrievalQAWithSourcesChain: Diese Chain führt eine Retrieval-Operation durch, wobei für den Eingabetext relevante Dokumente aus einer Datenbank abgerufen und anschließend kombiniert werden. Außerdem ist es mit dieser Chain möglich, die Quellen der für die Formulierung der Antwort verwendeten Dokumente zurückzugeben.

4.10 Frontend

Für die Erstellung des Frontends wurde Streamlit, ein Open-Source-Framework für die Erstellung von Webanwendungen, verwendet.

Für dieses Framework wurde sich aus mehreren Gründen entschieden:

- Schnelle Entwicklung mit reinem Python-Code
- Bietet eine Vielzahl von interaktiven Widgets
- Aktualisiert die Anwendung automatisch bei Änderungen im Code
- Einfache Bereitstellungsmöglichkeiten

5 Evaluation

Während der Entwicklung wurden verschiedene, einzelne Komponenten der Anwendung durch Trial-and-Error stetig angepasst. Beispielsweise wurden die Prompt-Templates mehrfach umformuliert und der Suchtyp des Retrievers wurde im Laufe der Zeit von 'similarity' auf 'similarity score threshold' umgestellt.

Am Ende des Projektes wurde für die Evaluation der gesamten RAG Pipeline, wie bereits in der Einleitung erwähnt, das Evaluations-Framework DeepEval verwendet.

DeepEval bietet mehr als 14 Plug-and-Use LLM-evaluierte Metriken (vgl. Quick introduction | DeepEval - the Open-Source LLM Evaluation Framework 2024), von denen in diesem Projekt fünf genutzt wurden: drei für die Evaluierung des Retrievers und zwei für die Bewertung der Generierung.

5.1 Evaluierung des Retrievers

Die drei verwendeten Metriken für die Bewertung des Retrievers decken alle wichtigen Hyperparameter (Embedding Modell, Anzahl der abzurufenden Dokumente (top-K), Temperatur, Prompt Template etc.) ab, die die Qualität des Abrufkontextes beeinflussen (vgl. RAG Evaluation | DeepEval - The Open-Source LLM Evaluation Framework 2024). Um sicherzustellen, dass die Generator Komponente relevante Daten vom Retriever bekommt, ist es sinnvoll, alle drei Metriken zu kombinieren (vgl. ebd.).

5.1.1 Contextual Precision Metric

Die Contextual Precision Metric verwendet ein LLM, um für jedes Dokument im zurückgegebenen Kontext des Retrievers zu bestimmen, ob es basierend auf der erwarteten Antwort relevant für die Eingabe ist, bevor die gewichtete kumulative Präzision als kontextueller Präzisionswert berechnet wird (vgl. Contextual precision | DeepEval - the Open-Source LLM Evaluation Framework 2024). Ein höherer kontextueller Präzisionswert deutet darauf hin, dass der Retriever in der Lage ist, relevante Dokumente im

zurückgegebenen Kontext höher einzustufen als andere (vgl. ebd.). Weitere Informationen finden sich [hier](#).

5.1.2 Contextual Recall Metric

Die Contextual Recall Metric bewertet die Qualität des Retrievers, indem sie untersucht, inwieweit der zurückgegebene Kontext mit der erwarteten Antwort übereinstimmt (vgl. Contextual recall | DeepEval - The Open-Source LLM Evaluation Framework 2024). Weitere Informationen finden sich [hier](#).

5.1.3 Contextual Relevancy Metric

Die Contextual Relevancy Metric bewertet die Qualität des Retrievers, indem sie die Gesamtrelevanz der Informationen misst, die im zurückgegebenen Kontext für eine gegebene Eingabe vorhanden sind (vgl. Contextual relevancy | DeepEval - the Open-Source LLM Evaluation Framework 2024). Weitere Informationen finden sich [hier](#).

5.2 Evaluierung der Generierung

Für die Bewertung generischer Generierungen bietet DeepEval zwei Metriken an, die im Folgenden genauer erklärt werden.

5.2.1 Answer Relevancy Metric

Die Answer Relevancy Metric misst die Qualität des Generators, indem sie bewertet, wie relevant die tatsächliche Antwort im Vergleich zu der erwarteten Antwort aus dem Datensatz ist (vgl. Answer relevancy | DeepEval - the Open-Source LLM Evaluation Framework 2024). Letztlich wird bewertet, wie gut die verwendeten(n) Prompt-Vorlage(n) das LLM anweisen, relevante und hilfreiche Outputs basierend auf dem Abrufkontext zu erzeugen (vgl. RAG Evaluation | DeepEval - The Open-Source LLM Evaluation Framework 2024). Weitere Informationen finden sich [hier](#).

5.2.2 Faithfulness Metric

Die Faithfulness Metric misst die Qualität des Generators, indem sie bewertet, ob die tatsächliche Ausgabe faktisch mit den Inhalten des Abrufkontexts übereinstimmt. Damit kann überprüft werden, ob das LLM Informationen ausgeben kann, die nicht halluzinieren (vgl. Faithfulness | DeepEval - The Open-Source LLM Evaluation Framework 2024). Weitere Informationen finden sich [hier](#).

5.3 Implementierung

Nachdem ich mich in das Testing-Framework eingelesen hatte, erstellte ich mir zunächst einen Datensatz aus einem kleinen Teil des ursprünglichen Datensatzes (ein Datensatz mit mehr als 50 Einträgen ist bei DeepEval kostenpflichtig). Die resultierende JSON Datei beinhaltet eine Liste mit Objekten mit folgenden Eigenschaften:

- question_id
- question
- expected_answer
- actual_answer
- context

Um die actual_answer sowie den context zu jeder Frage zu erhalten, wurden zunächst alle Fragen durch eine leicht abgewandelte Form des implementierten Chatbots verarbeitet. Die Chat-Historie wurde an dieser Stelle vernachlässigt.

Im zweiten Schritt wurden aus den JSON Objekten Testfälle der Klasse [LLMTestCase](#) (von DeepEval) erstellt und als Datensatz der Klasse [EvaluationDataset](#) (ebenfalls von DeepEval) auf ConfidentialAI gepusht.

Anschließend lud (pull) ich das eben erstellte EvaluationDataset von ConfidentialAI und testete alle Testfälle des Datensatzes, in einem parametrisierten [Pytest](#) (Python-Testframework) Test, gegen die oben erwähnten Metriken. Die Metriken wurden dabei mit ihren jeweiligen Default-Parametern verwendet.

Den eben beschriebenen Vorgang führte ich zweimal durch: einmal mit 20 Testfällen und einmal mit 34. Mit den jeweiligen Ergebnissen bin ich sehr zufrieden.

Im Testdurchlauf mit 20 Testfällen sind 75% aller Fälle erfolgreich gewesen. Abbildung 5 zeigt eine Übersicht der aggregierten Metriken mit ihren Durchschnittswerten.

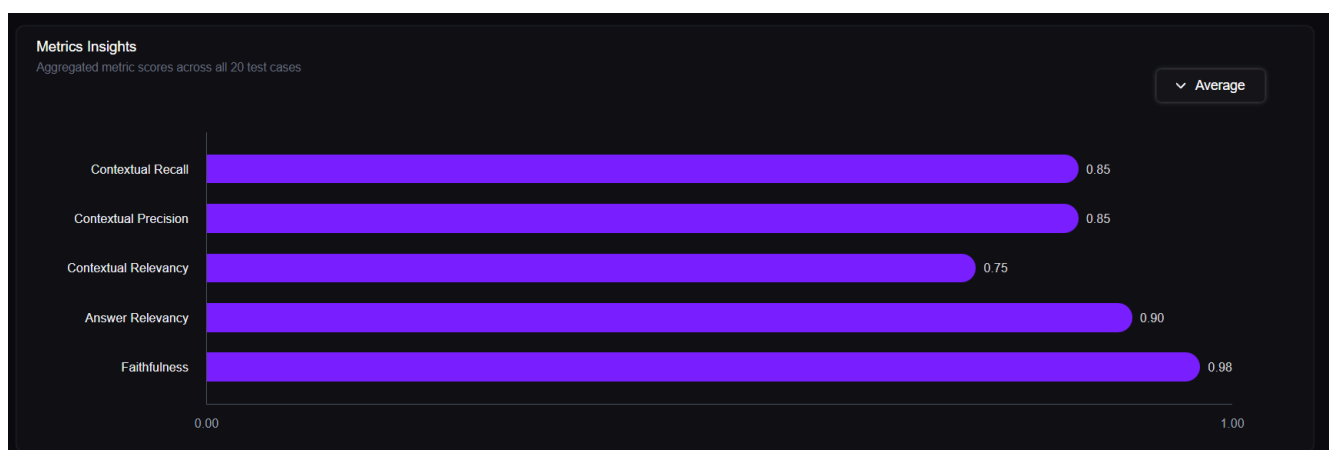


Abbildung 5: Ergebnisse für 20 Testfälle gegen fünf Metriken

Im zweiten Testdurchlauf, mit 34 Testfällen, sind 61.8% erfolgreich gewesen. Abbildung 6 zeigt ebenfalls eine Übersicht der aggregierten Metriken mit ihren Durchschnittswerten.

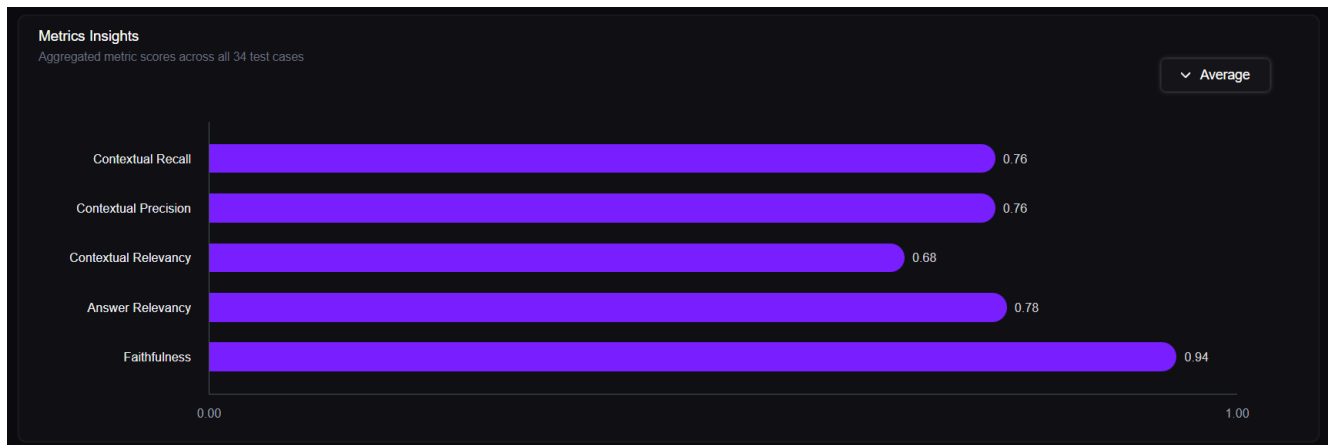


Abbildung 6: Ergebnisse für 34 Testfälle gegen fünf Metriken

Was durch die beiden Grafiken deutlich wird, ist, dass der Retriever nicht optimal konfiguriert ist: Bei der Auswahl relevanter Informationen für gegebene Eingaben gibt es Verbesserungspotential. Zwei Ansatzpunkte sind die Chunk Size des Text Splitters sowie die Anpassung der Top-K.

Es wäre möglich, dass die Chunks zu groß sind und dadurch irrelevante Informationen enthalten oder zu klein, wodurch wichtiger Kontext fehlen könnte.

Weiterhin könnte die Anzahl der zurückgegebenen Ergebnisse nicht ideal sein: Eventuell werden zu viele Ergebnisse zurückgegeben, was irrelevante Informationen einschließt oder zu wenige, sodass relevante Informationen fehlen.






6 Endprodukt

Innerhalb des Kurses “Artificial Intelligence and Adaptive Systems” ist eine funktionsfähige Chatbot-Anwendung mit Retrieval-Augmented Generation (RAG) Pipeline entstanden, die Fragen zu Taylor Swift beantworten kann.

Die Funktionsweise des Bots ist so gestaltet, dass er nur Fragen beantwortet, zu denen entsprechende Dokumente vorhanden sind. Gibt es zu einer Frage keine passenden Dokumente, so bekommt der Benutzer eine Antwort wie z.B. “Ich weiß es nicht”.

Folgendes habe ich aus zeitlichen Gründen nicht mehr geschafft zu implementieren beziehungsweise zu verbessern:

- Logging statt print-Statements
- Datenvorverarbeitung
 - Fehleranfällige Metadaten-Verarbeitung
 - Datenvorverarbeitung nur in Bezug auf das Test-Set, nicht auf den gesamten Datensatz
- Indexierung der Vektor-Datenbank (siehe [hier](#))
 - Vermeidung von Duplikaten
 - Vermeidet die Neuberechnung von Embeddings für unveränderte Inhalte
 - Effizientere Suche
- Verwendung eines fortgeschritteneren Retrievers

- Parent Document Retriever: teilt Dokumente in kleine Stücke, speichert sie und ruft bei Bedarf die größeren, zusammenhängenden Originaldokumente anhand der übergeordneten IDs ab (vgl. Parent Document Retriever |  LangChain o. D.)
- Contextual Compression Retriever: komprimiert Dokumente kontextbezogen, um nur relevante Informationen basierend auf einer Anfrage zurückzugeben (vgl. Contextual compression |   LangChain o. D.)
- Verwendung eines Re-Rankers zur Verfeinerung der Ergebnisse des Retrievers
- Besseres Memory Management bezüglich Nachrichtenverlauf
 - Beispielsweise durch Trimming älterer Nachrichten zur Reduzierung des Speicherverbrauchs (vgl. Memory management |   LangChain o. D.)
- Gegenüberstellung verschiedener Modelle und Datensätze
- Testing nur an der Oberfläche

Insgesamt bin ich mit dem Ergebnis zufrieden, wobei zu betonen ist, dass ich keine Vorerfahrung und nicht viel Zeit zur Verfügung hatte. Somit wurde während des gesamten Implementierungsprozesses lediglich an der Oberfläche gekratzt, was viel Raum für Verbesserung lässt.

7 Vergleich zu anderen RAG Pipelines

Am Ende meines Projektes wollte ich über die zahlreichen, absolvierten Tutorials hinweg wissen, wie andere RAG Pipelines aufgebaut sind.

Auf GitHub gibt es ein [Repository](#) von [deepset](#), welches in der README auf diverse Google Colab-Notebooks verweist. Haystack ist ein Open-Source-Framework zum Erstellen von Anwendungen mit LLMs, RAG Pipelines und uvm. (vgl. Haystack Introduction o. D.). Deepset ist die Firma hinter Haystack (vgl. deepset o. D.).

Der grundsätzliche Aufbau der dort implementierten RAG Pipelines ist in den jeweiligen Notebooks sehr ähnlich zu meinem Aufbau. Zu beachten ist, dass es sich bei all diesen Implementierungen um Beispiele handelt und um keine Architekturen, die produktiv eingesetzt werden.

Weiterhin stieß ich bei meiner Recherche auf die Open-Source RAG Engine [RAGFlow](#), welche produktiv eingesetzt werden kann und vermutlich auch wird. Ein großer Unterschied zu meiner Implementierung, auf den ich genauer eingehen möchte, ist die Verwendung eines Re-Rankers.

Eine Re-Ranker Komponente durchsucht die vom Retriever zurückgegebenen Dokumente und ordnet sie basierend auf ihrer Relevanz (vgl. Solanki 2023). Die Verwendung dieser Komponente ist deshalb sinnvoll, weil der Retriever irrelevante Dokumente hoch einstufen könnte, während relevante Dokumente niedrig eingestuft werden. Entsprechend sind nicht alle zurückgegebenen Top-k-Dokumente relevant und nicht alle relevanten Dokumente befinden sich unter den Top-k. Der Re-Ranker verfeinert die Ergebnisse des Retrievers und bringt die relevantesten Dokumente nach oben (vgl. ebd.).

8 Learnings und Ausblick

Im Rahmen dieses Projektes habe ich zum ersten Mal eine Chatbot-Anwendung von Grund auf selbst implementiert, sowie eine RAG Pipeline aufgebaut. Dadurch habe ich viel gelernt und der Ausblick ist entsprechend weitreichend. Um den Umfang dieses Kapitels angemessen zu halten, möchte ich nur auf ein paar Punkte eingehen.

Ein zentrales Learning aus diesem Projekt ist die entscheidende Rolle des Prompt Engineering.

Anfänglich gab der Retriever eine Vielzahl irrelevanter Dokumente zurück, was mich zunächst dazu veranlasste, die Daten besser vorzuverarbeiten. Durch zusätzliches Experimentieren mit Parametern wie Top-k und dem threshold gelang es mir, nur relevante Dokumente aus der Vektor-Datenbank zu holen. Trotzdem blieben die generierten Antworten nur teilweise korrekt, was mich irritierte. Mein erster Lösungsansatz bestand in der Verwendung eines anderen LLM, was jedoch keine Verbesserungen brachte. Anschließend probierte ich andere Embedding-Modelle wie Cohere aus, doch auch das blieb ohne Erfolg. Letztendlich bestand die Lösung in der Anpassung der Prompt-Templates, um dem Modell bessere Anweisungen zu geben und somit relevantere Antworten zu erhalten.

Ein weiteres wesentliches Learning ist die Bedeutung der sorgfältigen Auswahl des Datensatzes. Der ursprünglich gewählte Datensatz "Netflix Movies and TV Shows" erwies sich aufgrund fehlender Fragen zu den Fakten als ungeeignet für die Evaluierung. Dies führte dazu, dass ich am Ende des Projektes einen neuen Datensatz suchen und die Datenvorverarbeitung anpassen musste.

Des Weiteren traten bei dem Versuch, aus dem Chatverlauf und einer neuen Frage eine alleinstehende Frage zu formulieren, mehrere Schwierigkeiten auf. Anstatt die ursprüngliche Frage umzuformulieren, wurde häufig ein Kommentar hinzugefügt. Beispielsweise wurde auf der Frage "Who is Taylor Swift?" die Formulierung "What is Taylor Swift's identity? (unchanged as the user prompt does not reference the chat history)".

Außerdem führte ein nicht vollständig relevanter Kontext zu unpassenden Antworten, welche dann wiederum die Formulierung der alleinstehenden Frage beeinflussten.

Ein weiteres Problem bestand darin, dass statt einer neu formulierten Frage mehrere Varianten generiert wurden. Beispielsweise wurde aus "Who is Taylor Swift?" der Prompt "What is Taylor Swift's identity (assuming the user is unfamiliar with her) or, Who is the singer named Taylor Swift? (assuming some context, but the user's question does not directly reference the chat history)".

Meine Vermutung ist, dass sowohl der Retriever als auch das Prompt-Template, welches verwendet wurde, um das LLM anzuweisen, die Frage neu zu formulieren, verbessert werden müsste, um den Prozess zur Generierung alleinstehender Fragen zu optimieren.

Aus zeitlichen Gründen habe ich mich letztendlich für einen anderen Ansatz entschieden. Anstatt den User Prompt basierend auf dem Chatverlauf umzuformulieren, wird er nun grundsätzlich vor dem Retrieval-Schritt durch ein LLM überarbeitet. Ziel ist es, die Anfrage so präzise wie möglich zu gestalten - jedoch ohne die Berücksichtigung des vorherigen Kontextes.

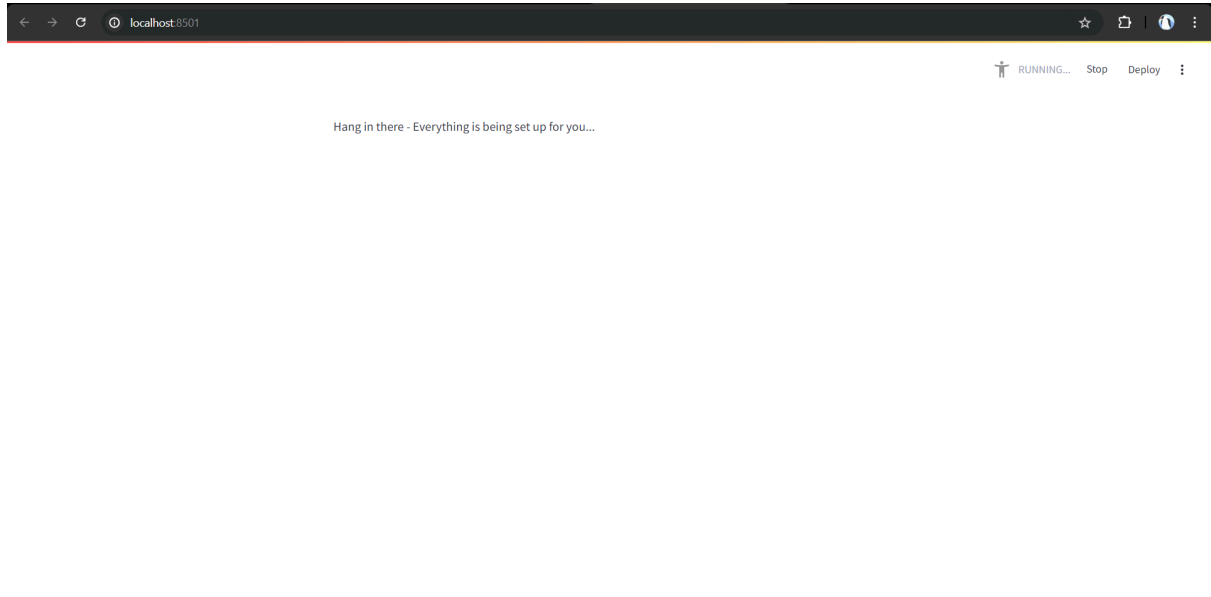
Ein letztes Learning, welches ich erwähnen möchte, ist das Folgende:

Zu Beginn des Projektes legte ich mir eine zweite csv Datei an, in die ich nur einen Bruchteil des heruntergeladenen Datensatzes kopierte. Dies erleichterte mir den Einstieg und gab mir die Möglichkeit, in jedem Schritt besser nachvollziehen zu können, was gerade passiert. Auf diesem Datensatz baute ich mein gesamtes QA-System auf - auch die Datenvorverarbeitung ist auf diese Daten zugeschnitten. Beispielsweise schaute ich mir in den knapp 100 Zeilen jede an, in der das "Frage"-Feld leer war und suchte in Bezug auf die Antworten nach einem Muster, nach dem ich genau diese Zeilen identifizieren und löschen konnte. Erst ganz am Ende des Projektes fiel mir auf, dass ich die ganze Zeit über nur mit einem Bruchteil des ursprünglichen Datensatzes gearbeitet und schlicht vergessen hatte, das originale Set zu verwenden. Also befüllte ich die Vektor-Datenbank mit dem vollständigen Datensatz und musste feststellen, dass die Qualität meiner RAG-Pipeline erheblich schlechter wurde. Meine Vermutung ist, dass die implementierte Datenvorverarbeitung für den kleinen Teil des Datensatzes ausreichend funktioniert, der gesamte Datensatz davon jedoch nicht abgedeckt wird.

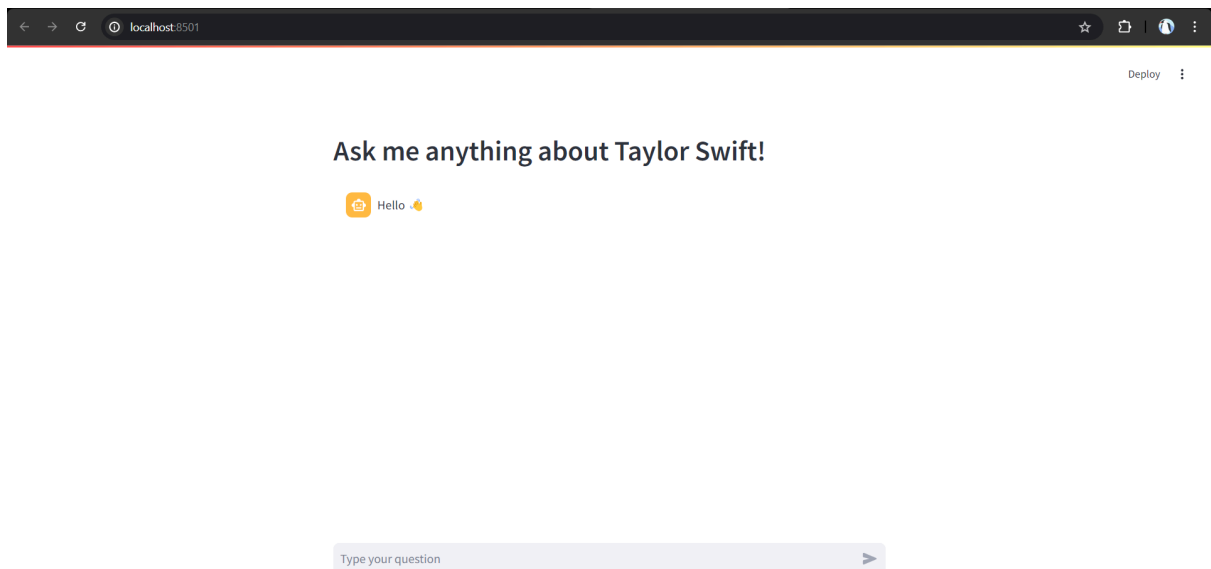
Die oben aufgeführten Erkenntnisse und die Liste (in Kapitel 6) der nicht umgesetzten Funktionen zeigen deutlich, dass die entstandene Anwendung grundsätzlich funktionsfähig ist, jedoch erhebliches Verbesserungspotential besteht. Im Rahmen dieses Projekts konnte ich nur einen ersten Einblick gewinnen und die Grundlagen schaffen. Es gibt noch viele Möglichkeiten zur Weiterentwicklung und Optimierung der Anwendung, die aufgrund der zeitlichen Begrenzung nicht realisiert werden konnten.

9 Anhang

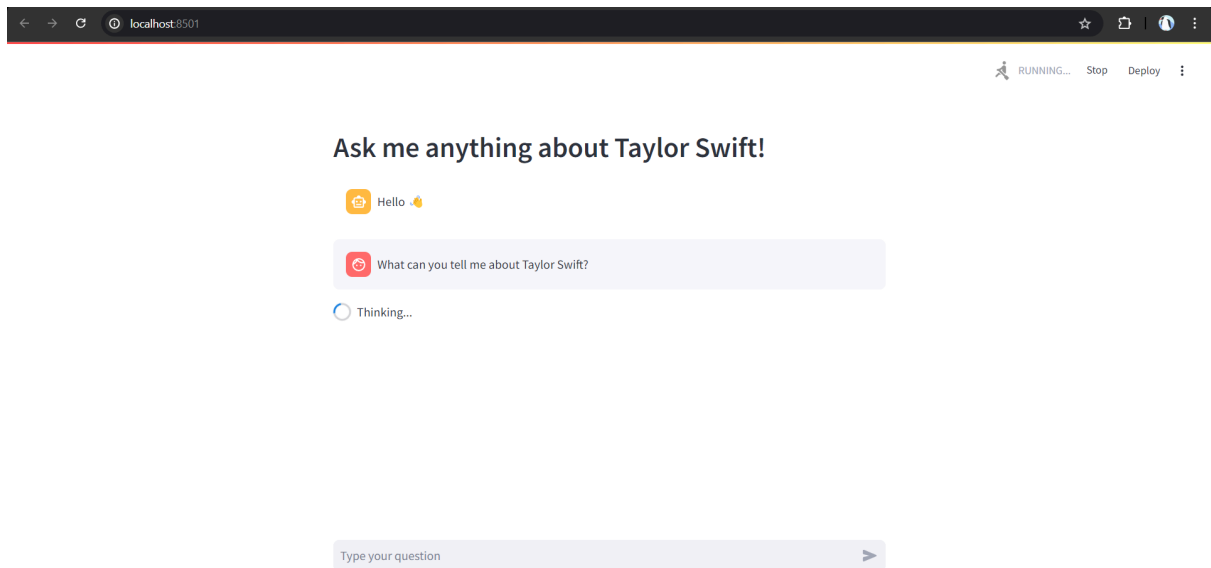
9.1 Screenshots der Anwendung



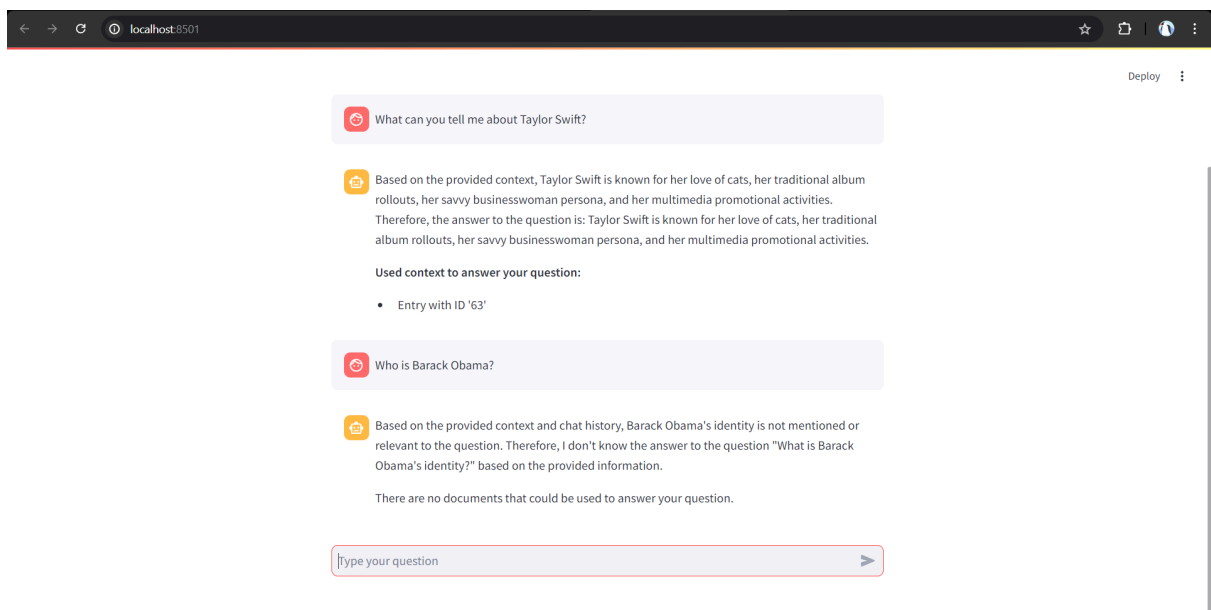
Der obige Screenshot zeigt die Ansicht, während die Anwendung gestartet wird.



Der obige Screenshot zeigt die Ansicht, wenn die Anwendung erfolgreich gestartet wurde und bereit für Anfragen ist.




Der obige Screenshot zeigt einen Spinner mit dem Label 'Thinking...', während relevante Dokumente aus dem Vektor-Store geholt und die Antwort generiert werden.



Der obige Screenshot zeigt das Verhalten des Chatbots, wenn der Retriever zu einer Frage keine passenden Dokumente zurückgegeben hat.

9.2 Literaturverzeichnis



Auf alle nachfolgenden Quellen wurde zuletzt am 13.07.2024 zugegriffen.



1. Ai, Novita (2024): What are large language model settings: temperature, Top P and Max tokens, in: *Medium*, 29.04.2024, [online]
https://medium.com/@marketing_novita.ai/what-are-large-language-model-settings-temperature-top-p-and-max-tokens-a482d8d817b2.
2. Answer relevancy | DeepEval - the Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/metrics-answer-relevancy>.
3. Banjara, Babina (2024): A Comprehensive Guide to Using Chains in Langchain, Analytics Vidhya, [online]
<https://www.analyticsvidhya.com/blog/2023/10/a-comprehensive-guide-to-using-chains-in-langchain/>.
4. Bhavsar, Pratik (o. D.): RAG vs Fine-Tuning vs both: A Guide for Optimizing LLM Performance - Galileo, Galileo, [online]
<https://www.rungalileo.io/blog/optimizing-llm-performance-rag-vs-finetune-vs-both>.
5. Chat Models |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/model_io/chat/.
6. ChunkViz (o. D.): [online] <https://chunkviz.up.railway.app/>.
7. Contextual compression |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/.
8. Contextual precision | DeepEval - the Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/metrics-contextual-precision>.
9. Contextual recall | DeepEval - The Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/metrics-contextual-recall>.

10. Contextual relevancy | DeepEval - the Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/metrics-contextual-relevancy>.
11. deepset (o. D.): GitHub, [online] <https://github.com/deepset-ai>.
12. FAISS vs Chroma: Vector Storage Battle (2024): [online] <https://myscale.com/blog/faiss-vs-chroma-vector-storage-battle/>.
13. Faithfulness | DeepEval - The Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/metrics-faithfulness>.
14. Find Open Datasets and Machine Learning Projects | Kaggle (o. D.): [online] <https://www.kaggle.com/datasets>.
15. Gavrilova, Yulia (2024): How to Preprocess Data in Python, Serokell Software Development Company, [online] <https://serokell.io/blog/data-preprocessing-in-python>.
16. Generative AI on AWS (2023): Retrieval-Augmented Generation (RAG) using LangChain and Pinecone - The RAG Special Episode, [YouTube] https://www.youtube.com/watch?v=J_tCD_J6w3s.
17. Gryczka, Katarzyna (2024): Data preprocessing: a comprehensive step-by-step guide, in: *Technology & Software Development Blog | Future Processing*, 31.01.2024, [online] <https://www.future-processing.com/blog/data-preprocessing-a-comprehensive-step-by-step-guide/>.
18. Haystack Introduction (o. D.): Haystack Documentation, [online] <https://docs.haystack.deepset.ai/docs/intro>.
19. Introduction | DeepEval - the Open-Source LLM Evaluation Framework (2024): [online] <https://docs.confident-ai.com/docs/confident-ai-introduction>.
20. Kalra, Karan (2024a): LangChain: A Complete Guide & Tutorial, Nanonets Intelligent Automation, And Business Process AI Blog, [online] <https://nanonets.com/blog/langchain/>.
21. Kalra, Karan (2024b): Retrieval-Augmented Generation & RAG Workflows, Nanonets Intelligent Automation, And Business Process AI Blog, [online]

<https://nanonets.com/blog/retrieval-augmented-generation/#what-problem-do-they-solve>.

22. Kondrashchenko, Iryna (2023): First Steps in LangChain: The Ultimate Guide for Beginners (part 1), in: *Medium*, 18.06.2023, [online]
<https://medium.com/@iryna230520/first-steps-in-langchain-the-ultimate-guide-for-beginners-part-1-2baf5a4e1b81>.
23. Luber, Stefan (2023): Was ist ein Large Language Model?, in: *BigData-Insider*, 11.09.2023, [online]
<https://www.bigdata-insider.de/was-ist-ein-large-language-model-a-d735d93bbc24d3c4091de8ce25aa36e8/>.
24. Lutkevich, Ben (2023a): Hugging face, WhatIs, [online]
<https://www.techtarget.com/whatis/definition/Hugging-Face>.
25. Lutkevich, Ben (2023b): KI-Halluzination, ComputerWeekly.de, [online]
<https://www.computerweekly.com/de/definition/KI-Halluzination>.
26. Martineau, Kim (2024): What is retrieval-augmented generation?, IBM Research, [online] <https://research.ibm.com/blog/retrieval-augmented-generation-RAG>.
27. Memory management | 🦜🔗 LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/use_cases/chatbots/memory_management/.
28. Merritt, Rick (2024): What Is Retrieval-Augmented Generation aka RAG | NVIDIA Blogs, NVIDIA Blog, [online]
<https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/>.
29. Parent Document Retriever | 🔗 LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/parent_document_retriever/.
30. Prompts | 🔗 LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/model_io/prompts/.
31. Prompts | 🔗 LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/model_io/prompts/.

32. Quick introduction | DeepEval - the Open-Source LLM Evaluation Framework (2024):
[online] <https://docs.confident-ai.com/docs/getting-started>.
33. RAG Evaluation | DeepEval - The Open-Source LLM Evaluation Framework (2024):
[online] <https://docs.confident-ai.com/docs/guides-rag-evaluation>.
34. Retrievers |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/.
35. Safar, Milad (2023): Large Language Models – Grundlagen KI-getriebener Kommunikation, Weissenberg, [online]
<https://weissenberg-group.de/was-ist-ein-large-language-model/>.
36. Sahota, Harpreet (2024): Introduction to Prompt Templates in LangChain, Comet,
[online]
<https://www.comet.com/site/blog/introduction-to-prompt-templates-in-langchain/>.
37. Saleem, Ayesha (2024): How to tune LLM Parameters for optimal performance | Data Science Dojo, Data Science Dojo, [online]
<https://datasciencedojo.com/blog/tuning-optimizing-llm-parameters/>.
38. sentence-transformers/all-MiniLM-L6-v2 · Hugging Face (2001): [online]
<https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>.
39. Solanki, Shivam (2023): Improving RAG (Retrieval Augmented Generation) Answer Quality with Re-ranker, in: *Medium*, 27.08.2023, [online]
<https://medium.com/towards-generative-ai/improving-rag-retrieval-augmented-generation-answer-quality-with-re-ranker-55a19931325>.
40. Streamlit Docs (o. D.): [online] <https://docs.streamlit.io/>.
41. Text Splitters |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/.
42. Tran, Hoang/Hoang Tran (2023a): Which is better, retrieval augmentation (RAG) or fine-tuning? Both., Snorkel AI, [online]
<https://snorkel.ai/which-is-better-retrieval-augmentation-rag-or-fine-tuning-both/>.

43. Tran, Hoang/Hoang Tran (2023b): Which is better, retrieval augmentation (RAG) or fine-tuning? Both., Snorkel AI, [online]
<https://snorkel.ai/which-is-better-retrieval-augmentation-rag-or-fine-tuning-both/>.
44. Vector store-backed retriever |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/vectorstore/.
45. Vector stores |  LangChain (o. D.): [online]
https://python.langchain.com/v0.1/docs/modules/data_connection/vectorstores/.
46. Was ist Einbettung? – Einbettungen in Machine Learning erklärt – AWS (o. D.): Amazon Web Services, Inc., [online]
<https://aws.amazon.com/de/what-is/embeddings-in-machine-learning/#:~:text=Embedding%20models%20are%20algorithms%20trained,models%20used%20in%20ML%20applications.>
47. Was sind Large Language Models (LLMs)? | Databricks (o. D.): Databricks, [online]
<https://www.databricks.com/de/glossary/large-language-models-llm>.
48. What is DeepEval? Features & Examples (2024): Deepchecks, [online]
<https://deepchecks.com/glossary/deepeval/#:~:text=DeepEval%20is%20an%20open%2Dsource,holistic%20assessment%20of%20LLM%20performance>.
49. What is text generation? | IBM (o. D.): [online]
<https://www.ibm.com/topics/text-generation>.