

✓ RAG Workshop Notebook

TDWI 2025 – Hands-on Session

Dieses Notebook führt dich Schritt für Schritt durch den kompletten RAG-Workflow:

1. **Setup & Ziele** - Umgebung vorbereiten, Lernziele verstehen

Index Pipeline

2. 🛠️ **Daten laden & reinigen** – denn ohne Qualität läuft mal wieder nix
3. 🗂️ **Chunking** – den Elefanten in Scheiben schneiden
4. 🎨 **Embedding** – die Welt in Vektoren gießen

Vektorspeicher

5. 🗄️ **Vector Store** – Mathe statt Magie rund um FAISS, DuckDB, PostgreSQL

RAG Pipeline

6. 🔍 **Prompt pimpen** – Quellen smart einbauen
7. 🗨️ **Chat GPTiert** – Kontext drin, Halluzinationen draußen!?
8. **Evaluierung** - Ein paar einfache Checks

Die obigen Schritte werden anhand einer Beispiel-Wiki-Seite durchgespielt. Am Ende des Notebooks sind Zellen für eine

- Index Pipeline + Vektorspeicher
- RAG Pipeline mit jeweils vielen Wiki-Seiten.

Das Notebook dient dazu die Basiskonzepte aufzuzeigen ohne Anspruch auf Vollständigkeit.

Lernziele

Nach diesem Notebook kannst du ...

1. die Kernidee hinter Retrieval-Augmented Generation (RAG) erklären,
2. Beispiel-Webdaten laden und inspizieren,
3. Semantische Suche in Vektordatenbanken/indexes durchführen.

Du kannst das Notebook in Google Colab ausführen oder lokal auf deinem eigenen Notebook.

Voraussetzungen

Voraussetzungen

- Python-Kenntnisse
- Google Colab (frei verfügbare Notebook Umgebung <https://colab.research.google.com/>)
- HuggingFace-Account <https://huggingface.co/> und LLM wie Gemini <https://aistudio.google.com/apikey> (jeweils ohne Kosten)
- Optional OpenAI-Account (API Guthaben notwendig, kostenpflichtig)

Zusätzliche Voraussetzungen bei lokaler Ausführung

- Python ≥ 3.11
- Falls PostgreSQL pgvector genutzt werden soll: lokale Docker/podman Installation

Ablauf - grober Überblick

Abschnitt	Zeit	Inhalt
Setup (dieses Notebook)	10 min	Umgebung, Datenquellen, Key-Handling
Daten laden	10 min	Wiki Seite abrufen
Chunking	15 min	Text in Häppchen
Embedding	10 min	Vektorisierung & Kosten
Indexing	15 min	Vektor-Datenbanken
Augmentation & Retrieval	15 min	Daten anreichern und Chat
Evaluierung & Extras	5 min	Metriken, RAG-Chains

✓ 01 Setup

In diesem Abschnitt installieren wir Abhängigkeiten, importieren Libraries und definieren globale Konstanten.

✓ Schritt 01-01 – Python Pakete installieren

- In google colab requirements.txt zuerst nach content kopieren! (content ist Default-Verzeichnis und oberhalb von sample_data)
- Bei einem lokalen Notebook: requirements.txt liegt im Repository/Projekt-Rootverzeichnis, d.h. eine Ebene oberhalb von src/(dieses Notebook)

Der nachfolgende Installationsvorgang wird einige Minuten benötigen!

```
import sys
import subprocess
from pathlib import Path
import os, pathlib, sys

# prüft ob in Colab-Umgebung
is_colab = "google.colab" in sys.modules
req_path = Path("requirements.txt") if is_colab else Path("../") / "requirements.txt"
python_exe = sys.executable

# ist requirements.txt vorhanden?
if not req_path.exists():
    raise FileNotFoundError(
        f"Requirements-Datei nicht gefunden: {req_path.resolve()}\n"
        f"Bitte Datei hochladen in Colab oder lokal bereitstellen\n"
        f"Aktueller Ordner des Kernels: {Path().resolve()}"
    )

print("Installation gestartet ... wird einige Minuten benötigen!")
subprocess.check_call(
    [python_exe, "-m", "pip", "install", "-r", str(req_path), "--no-cache-dir"]
)

print("Arbeitsverzeichnis:", os.getcwd())
print("Python-Version   :", sys.version.split()[0])
print("Inhalt:", list(pathlib.Path().iterdir())[5])
```

↗ Installation gestartet ... wird einige Minuten benötigen!
 Arbeitsverzeichnis: /content
 Python-Version : 3.11.13
 Inhalt: [PosixPath('.config'), PosixPath('requirements.txt'), PosixPath('sample_data')]

✓ Schritt 01-02 – Import der Python Pakete und initiale Einstellungen

LangChain ist ein Open-Source-Framework, das Hilfsklassen und Bausteine bereitstellt, um Large-Language-Model-(LLM-)Anwendungen – vom einfachen Prompt-Template bis zum komplexen Retrieval-Augmented-Generation-(RAG-)Workflow – schnell zusammenzustecken.

Seit Ende 2024 wurden alle optionalen Integrationen (Loader, Embeddings, Tools u. a.) in das separate Paket **langchain-community** ausgelagert; das Kernpaket enthält nur noch die zentralen Abstraktionen, während langchain_community die Schnittstellen zu Dritt-Libs wie HuggingFace, OpenAI, Pinecone usw. bündelt. Beide Pakete werden gemeinsam gepflegt und folgen demselben API-Stil.

Vorteile

- Sehr große Auswahl an fertigen Komponenten (Loader, Splitter, Embeddings, Vektordatenbanken).
- Einheitliche Schnittstellen erleichtern Rapid Prototyping und späteren Austausch einzelner Bausteine.
- Aktive Community, laufend neue Integrationen, gute Tutorials und Beispiele.

Nachteile

- Häufige Breaking Changes; Upgrade kann Code-Anpassungen erfordern.
- Viele optionale Abhängigkeiten → voluminöse Installation und längere Import-Zeit.
- Abstraktionslage kann Debugging erschweren, weil interne Aufrufe in mehreren Schichten verborgen sind.

```
import warnings
import textwrap
import copy
import os

from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_community.document_transformers import BeautifulSoupTransformer

from langchain.text_splitter import CharacterTextSplitter, RecursiveCharacterTextSplitter
from langchain_text_splitters import HTMLSectionSplitter

from langchain_huggingface import HuggingFaceEmbeddings
from langchain_openai import OpenAIEmbeddings

import google.generativeai as genai
```

```
import faiss
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_community.vectorstores import FAISS
from langchain_community.vectorstores.faiss import DistanceStrategy
from langchain_community.vectorstores import DuckDB
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import Runnable

from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import numpy as np
from textwrap import shorten
import duckdb, pandas as pd

# stellt den globalen Warnfilter so ein, dass keine Warnungen mehr angezeigt werden.
# Praktisch für saubere Demo-Notebooks, aber gefährlich,
# denn wichtige Infos können unterdrückt werden.
warnings.filterwarnings("ignore")
os.environ["USER_AGENT"] = "TDWI-RAG-Workshop/1.0"
```

✓ Schritt 01-03 – Konfigurationen

Im Notebook wird nur eine Seite genutzt. Anhand dieser Seite werden die einzelnen Schritte gezeigt.

```
# Link zur HTML-Seite, die verarbeitet wird
# url = "https://en.wikipedia.org/wiki/Retrieval-augmented_generation"
url = "https://buckenhofer.com/2025/06/from-raw-text-to-ready-answers-a-technical-deep-dive-into-retrieval-augmented-generation-rag/"
```

✓ 02 🗡️ Daten laden & reinigen – denn ohne Qualität läuft mal wieder nix

In diesem Abschnitt laden wir die Wiki-Seite, zeigen die Metadaten und Nutzdaten an sowie entfernen html-Tags.

✓ Schritt 02-01 – Abrufen der HTML-Seite

Die HTML-Seite wird gelesen in die Variable `html_page_ori`. Diese Variable enthält

- Metadaten
- Nutzdaten ("HTML-Seite") inklusive html-Tags. Die Nutzdaten müssen im weiteren Verlauf bereinigt werden ("Cleansing").

```
# AsyncHtmlLoader (Langchain) lädt HTML-Seiten parallel ein und liefert sie als Document-Objekte zurück
HTMLloader = AsyncHtmlLoader (url)
```

```
html_page_ori = HTMLloader.load()
```

```
🔄 Fetching pages: 100%|#####| 1/1 [00:01<00:00, 1.45s/it]
```

✓ Schritt 02-02 – Metadaten prüfen/anzeigen

```
html_page = copy.deepcopy(html_page_ori)
print(html_page[0].metadata)
```

```
🔄 {'source': 'https://buckenhofer.com/2025/06/from-raw-text-to-ready-answers-a-technical-deep-dive-into-retrieval-augmented-generation-rag/'}
```

✓ Schritt 02-03 – Nutzdaten prüfen/anzeigen

```
# Es werden die ersten 1000 Zeichen angezeigt
# und mit Hilfe von textwrap "umgebrochen" statt einer langen Zeile
# html Tags sind noch enthalten
html_page = copy.deepcopy(html_page_ori)
content = html_page[0].page_content[:1000]
wrapped = textwrap.fill(content, width=80)
```

```
print(wrapped)
```

```
🔄 <!DOCTYPE html> <!--[if IE 6]> <html id="ie6" lang="en" prefix="og:
https://ogp.me/ns#"> <![endif]--> <!--[if IE 7]> <html id="ie7" lang="en"
prefix="og: https://ogp.me/ns#"> <![endif]--> <!--[if IE 8]> <html id="ie8"
lang="en" prefix="og: https://ogp.me/ns#"> <![endif]--> <!--[if !(IE 6) | !(IE
7) | !(IE 8) ]><!--> <html lang="en" prefix="og: https://ogp.me/ns#">
<!--<![endif]--> <head> <meta charset="UTF-8" />
```

```
<meta http-equiv="X-UA-Compatible" content="IE=edge"> <link
rel="pingback" href="https://buckenhofer.com/xmlrpc.php" />
<!--[if lt IE 9]> <script src="https://buckenhofer.com/wp-
content/themes/Extra/scripts/ext/html5.js" type="text/javascript"></script>
<![endif]--> <script type="text/javascript">
document.documentElement.className = 'js'; </script> <!--
Search Engine Optimization by Rank Math PRO - https://rankmath.com/ -->
<title>From Raw Text to Ready Answers – A Technical Deep-Dive into Retrieval-
Augmented Generation (RAG) - data.KISS</title> <meta name
```

▼ Schritt 02-04 – Cleansing

Das Dokument beinhaltet html-Tags, die bereinigt werden müssen. Dazu wird `transform_documents` aufgerufen. `transform_documents` ändert die Originaldaten, daher wird zuerst eine Kopie angelegt. Durch das Anlegen der Kopie können Zellen im Notebook mehrfach aufgerufen werden und ist somit fehlertoleranter (insb. `transform_documents` könnte sonst nur ein Mal aufgerufen werden oder es muss wieder von vorne begonnen werden und die Wiki-Seite muss abgerufen werden).

`BeautifulSoupTransformer` ist ein `LangChain`-Helper, der HTML-Dokumente mit `Beautiful Soup` parst und den Inhalt nach vorgegebenen Regeln ausdünn. Durch `copy.deepcopy(html_page_ori)` wird zuerst eine vollständige Kopie der geladenen Document-Liste erzeugt, damit das Original unverändert bleibt. `transform_documents(..., tags_to_extract=["p"])` entfernt anschließend alles außer den p-Tags: Jeder zurückgegebene Document enthält nur noch den reinen Paragraph-Text, Metadaten wie die URL bleiben erhalten.

```
soup_transformer = BeautifulSoupTransformer()

html_page = copy.deepcopy(html_page_ori)
html_page_transformed = soup_transformer.transform_documents(html_page, tags_to_extract=["p"])
```

▼ Schrittm 02-05 – Metadaten prüfen/anzeigen

```
print(html_page_transformed[0].metadata)
```

```
{'source': 'https://buckenhofer.com/2025/06/from-raw-text-to-ready-answers-a-technical-deep-dive-into-retrieval-augmented-generator'}
```

▼ Schritt 02-06 – Nutzdaten prüfen/anzeigen

```
content = html_page_transformed[0].page_content[:5000]
wrapped = textwrap.fill(content, width=80)
```

```
print(wrapped)
```

```
Retrieval-Augmented Generation (RAG) fixes that. This recipe solves three
chronic LLM pain points: Freshness – swap in new documents; no expensive re-
training. Transparency & traceability – show the user where each fact came from
and reduce hallucinations. Parameter efficiency – you don't need to finetune the
LLM on every domain. The idea crystallised in the paper “ Retrieval-Augmented
Generation for Knowledge-Intensive NLP Tasks
(https://arxiv.org/abs/2005.11401?utm_source=chatgpt.com) ” by Patrick Lewis et
al., 2020. Since then it has become a de-facto blueprint for building
production-grade, knowledge-grounded chatbots and assistants. The diagram below
captures both flows: the lower left block is the index pipeline , the upper
right block is the RAG pipeline where retrieval meets generation. Your documents
go through an embedding model that converts each text chunk into a vector. Those
vectors are written to a vector store (FAISS, pgvector, DuckDB, etc.) where
“distance = semantic similarity”. My blog article “ Vector database – what, why,
and how (https://buckenhofer.com/2024/05/vector-database-what-why-and-how/) ”
describes basics for storing vectors: Document → Loading & Cleansing →
Chunking → Embedding Model → [0.11, 0.42, ...] → Vector Store The query
pipeline uses the knowledge base created in the vector store by the index
pipeline to answer questions. A user question arrives. Retriever does a
similarity search in the vector store. Retrieved passages are augmented onto the
prompt. The LLM reads this enriched prompt and generates the final answer.
Interested to experiment and learn RAG? My github repository for TDWI 2025
(https://github.com/abuckenhofer/tdwi2025) conference contains a working example
in a notebook. The notebook is structured into the following sections: Before a
single vector is born, your raw corpus must be typed, de-noised, and traceable .
Type detection & parsing – Use multi-format loaders so PDFs, HTML, Markdown, and
plain text all arrive as uniform text objects. Boiler-plate stripping –
Navigation bars, cookie banners, and “back to top” links poison embeddings
because they appear verbatim across pages; strip them early. Section labeling –
```

context but limit recall (only a few can fit in the prompt); tiny chunks boost recall but explode prompt length. Overlap matters – Introduce 10-20 % overlap so sentences that cross boundaries aren't torn apart, especially for legal or medical texts with heavy pronoun use. Domain-specific splitters Section-aware splitting keeps <h2> to <h6> blocks intact in manuals or docs. Token-based splitting for code ensures you respect language syntax rather than character counts. Sliding windows – For extremely inter-linked prose, a sliding window with higher overlap (25-50 %) can preserve co-reference chains without ballooning the total number of chunks. Here, text turns into mathematics. Model choice – Open-source sentence transformers give you on-prem freedom; vendor APIs (e.g. OpenAI) trade cost for turnkey quality and speed. Dimensionality & normalisation – Most modern models output 384- to 1024-dimensional vectors. ℓ^2 -normalising them turns cosine similarity into a simple dot product, lowering computation cost downstream. Batching & streaming – Stream chunks in batches to avoid exhausting GPU RAM; persist vectors incrementally so failed runs can resume without re-embedding everything. Quantisation options – 4-bit or 8-bit quantisation preserves ~99 % of accuracy while halving memory. Product quantisation later in the vector DB can shrink storage even more. Where those vectors actually live determines latency, recall, and scalability. FAISS – Ideal for single-node or GPU-accelerated use cases; supports HNSW graphs for high-recall ANN and IVF-PQ for compres

✓ 03 ✂ Chunking – den Elefanten in Scheiben schneiden

Theorie: Warum Chunking?

Large-Language-Modelle haben ein Kontextlimit (z. B. 128k Tokens bei GPT-4o). Ein langer Artikel muss daher in kleinere Teile („Chunks“) zerlegt werden. Die Wahl der Chunk-Größe und des Overlaps beeinflusst die Qualität der Ergebnisse. Das Kontextlimit wird immer größer. Trotzdem ist ein Chunking sinnvoll, denn je größer der Text (Chunk) ist, umso mehr gehen relevante, spezifische Inhalte verloren.

Es gibt eine Vielzahl von Chunking-Möglichkeiten:

- **Fixed Length** (z. B. 1000 Zeichen) – simpel, aber ignoriert Satzgrenzen
- **Token-basiert** – präziser, braucht tiktoken
- **Rekursives Splitten** – behält semantische Einheiten (Absätze, Sätze)

✓ Schritt 03-01 – Anzahl Zeichen der Webseite

```
print(len(html_page_transformed[0].page_content))
```

↩ 9042

✓ Schritt 03-02 – Fixed Length Chunking

Einfaches Chunking anhand fixer Größen mit/ohne Überlappung von Zeichen.

Führe die Schritte 03-01 bis 03-03 mit verschiedenen Überlappungen aus (chunk_overlap=...), siehe 03-01.

```
text_splitter = CharacterTextSplitter(
    separator=" ",          # Zeichen für das Trennen des Textes
    chunk_size=1000,        # Anzahl Zeichen im Chunk
    chunk_overlap=0,        # Anzahl überlappende Zeichen; wiederhole die nächsten Schritte mit anderen Werten, z.B. 0 oder 50
)
```

```
text_chunks=text_splitter.create_documents([html_page_transformed[0].page_content])
```

```
print(f"Anzahl Chunks, die erzeugt wurden: {len(text_chunks)}")
```

↩ Anzahl Chunks, die erzeugt wurden: 10

✓ Schritt 03-03 – Chunks prüfen/anzeigen (1 von 2)

Es werden nun 2 Chunks miteinander verglichen.

```
# vergleiche die letzten Zeichen von Chunk x mit den ersten Zeichen von Chunk x+1
```

```
content = text_chunks[0].page_content
wrapped = textwrap.fill(content, width=100)
```

```
print(wrapped)
```

↩ Select Page Monday, June 23, 2025 | RAG (<https://buckenhofer.com/category/rag/>) , VectorDB (<https://buckenhofer.com/category/vectordb/>) | 0 (<https://buckenhofer.com/2025/06/from-raw-text-to-ready-answers-a-technical-deep-dive-into-retrieval-augmented-generation-rag/#comments>) | Large language models (LLMs) are astonishing pattern-completion engines, but they are also static

archives: everything they “know” is baked into billions of parameters frozen at training time. Once the world changes—or your sensitive company manuals that are not public—the model is stuck in yesterday’s reality. Retrieval-Augmented Generation (RAG) fixes that. This recipe solves three chronic LLM pain points: Freshness – swap in new documents; no expensive re-training. Transparency & traceability – show the user where each fact came from and reduce hallucinations. Parameter efficiency – you don’t need to finetune the LLM on every domain. The idea crystallised in the paper “ Retrieval-Augmented Generation for

✓ Schritt 03-03 – Chunks prüfen/anzeigen (2 von 2)

```
content = text_chunks[1].page_content
wrapped = textwrap.fill(content, width=100)
```

```
print(wrapped)
```

🔗 Knowledge-Intensive NLP Tasks (https://arxiv.org/abs/2005.11401?utm_source=chatgpt.com) ” by Patrick Lewis et al., 2020. Since then it has become a de-facto blueprint for building production-grade, knowledge-grounded chatbots and assistants. The image below shows the pipelines used for RAG. Your documents go through an embedding model that converts each text chunk into a vector. Those vectors are written to a vector store (FAISS, pgvector, DuckDB, etc.) where “distance = semantic similarity”. My blog article “ Vector database – what, why, and how (<https://buckenhofer.com/2024/05/vector-database-what-why-and-how/>) ” describes basics for storing vectors: Document → Loading & Cleansing → Chunking → Embedding Model → [0.11, 0.42, ...] → Vector Store The query pipeline uses the knowledge base created in the vector store by the index pipeline to answer questions. A user question arrives. Retriever does a similarity search in the vector store. Retrieved passages are augmented onto the

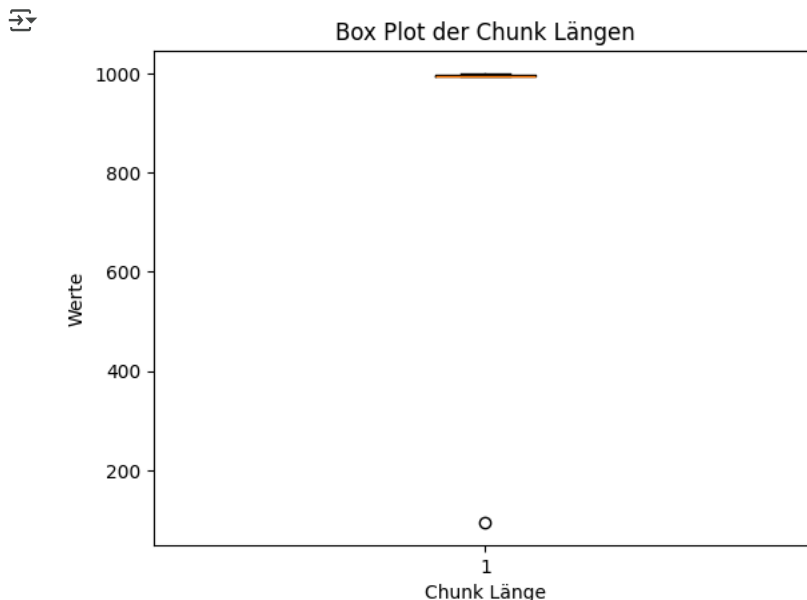
✓ Schritt 03-04 - Verteilung der Anzahl Zeichen von Minimum bis Maximum

```
data = [len(doc.page_content) for doc in text_chunks]
```

```
plt.boxplot(data)
plt.title('Box Plot der Chunk Längen')
plt.xlabel('Chunk Länge')
plt.ylabel('Werte')
```

```
plt.show()
```

```
print(f"Median Chunk Länge: {round(np.median(data),2)}")
print(f"Durchschnittliche Chunk Länge: {round(np.mean(data),2)}")
print(f"Minimum Chunk Länge: {round(np.min(data),2)}")
print(f"Max Chunk Länge: {round(np.max(data),2)}")
```



```
Median Chunk Länge: 995.5
Durchschnittliche Chunk Länge: 906.3
Minimum Chunk Länge: 95
Max Chunk Länge: 1000
```

✓ Schritt 03-05 – Rekursives Splitten

Verbessertes Chunking anhand thematischer Einheiten, z.B. Überschriften oder Tabellen in einer html-Seite.

sections_to_split_on legt fest, an welchen HTML-Elementen der Text zerlegt werden soll: Immer wenn im Quell-HTML ein h1, h2, h3 oder table auftaucht, beginnt der HTMLSectionSplitter einen neuen Abschnitt.

Mit `HTMLSectionSplitter(sections_to_split_on)` erstellst du daher einen Chunker, der strukturbewusst entlang dieser Tags splittet, statt einfach blind nach Zeichenlänge.

`copy.deepcopy(html_page_ori)` schützt das Original vor Veränderungen, anschließend wird in `split_text(html_page[0].page_content)` nur der kopierte HTML-String verarbeitet.

Das Ergebnis `split_content` ist eine Liste von Textsegmenten, die jeweils genau den Inhalt eines Headers bzw. einer Tabelle (plus folgenden Fließtext bis zum nächsten Split-Tag) enthalten – optimal für ein verbessertes Chunking vor dem Embedding.

Führe die nachfolgenden Schritte durch und vergleiche die Ergebnisse wenn `("p","P")` auskommentiert ist oder nicht.

```
sections_to_split_on = [
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    #("table", "Table"),
    #("div", "Div"),
    #("img", "Image"),
    #("p", "P"),
]

splitter = HTMLSectionSplitter(sections_to_split_on)

html_page = copy.deepcopy(html_page_ori)
split_content = splitter.split_text(html_page[0].page_content)
```

✓ Schritt 03-06 - Anzahl Chunks prüfen/anzeigen

```
len(split_content)
```

➡ 19

✓ Schritt 03-07 - die ersten 5 Chunks prüfen/anzeigen

```
#split_content[:5]
for i, doc in enumerate(split_content[:5], 1):
    header = next(iter(doc.metadata.values()), "-")
    preview = shorten(doc.page_content.replace("\n", " "), 120)
    print(f"{i:02d}. {header}: {preview}")
```

➡ 01. #TITLE#: Header #top-header Secondary Nav #et-info .et-extra-social-icons .et-top-search cart /.container /#top-header Main [..
02. From Raw Text to Ready Answers – A Technical Deep-Dive into Retrieval-Augmented Generation (RAG): From Raw Text to Ready Answers
03. The Two Pipelines That Matter: The Two Pipelines That Matter The diagram below captures both flows: the lower left block is the
04. Index Pipeline – Make knowledge searchable: Index Pipeline – Make knowledge searchable Your documents go through an embedding mc
05. RAG (Query-Time) Pipeline – Answer with context: RAG (Query-Time) Pipeline – Answer with context The query pipeline uses the kn

✓ Schritt 03-08 – Chunks per Abschnitt prüfen/anzeigen

Abschnitte sind z.B. Überschriften 1., 2., 3. Ebene, Tabellen oder andere html-Abschnitte. Wie viele Chunks je Abschnitt sind vorhanden?

```
from collections import Counter

class_counter = Counter()

for doc in split_content:
    document_class = next(iter(doc.metadata.keys()))
    class_counter[document_class] += 1

final_chunks = split_content

print(class_counter)
```

➡ Counter({'Header 2': 10, 'Header 1': 6, 'Header 3': 3})

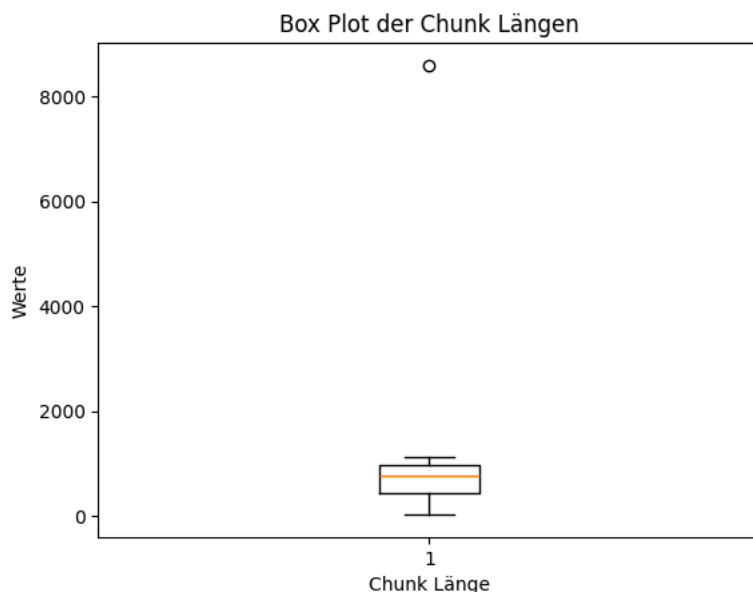
✓ Schritt 03-09 - Verteilung der Anzahl Zeichen von Minimum bis Maximum

```
data = [len(doc.page_content) for doc in final_chunks]

plt.boxplot(data)
plt.title('Box Plot der Chunk Längen')
plt.xlabel('Chunk Länge')
plt.ylabel('Werte')
```

```
plt.show()
```

```
print(f"Median Chunk Länge: {round(np.median(data),2)}")
print(f"Durchschnittliche Chunk Länge: {round(np.mean(data),2)}")
print(f"Minimum Chunk Länge: {round(np.min(data),2)}")
print(f"Max Chunk Länge: {round(np.max(data),2)}")
```



```
Median Chunk Länge: 761.0
Durchschnittliche Chunk Länge: 1101.58
Minimum Chunk Länge: 35
Max Chunk Länge: 8589
```

✓ 04 🧩 Embedding – die Welt in Vektoren gießen

✓ Schritt 04-01 – API Keys einlesen

Die API Keys in Google Colab in Secrets hinzufügen oder bei einem lokalen Notebook in die Datei .env

Der OpenAI Key ist optional (zumindest einen dummy Wert setzen). Die API Keys für Huggingface und Gemini sind frei (ohne Kosten). Die Keys haben die folgenden Namen:

- OPENAI_API_KEY
- HF
- GEMINI

Für OpenAI: Einen API Key unter <https://platform.openai.com/settings/organization/api-keys> erzeugen. Achtung: der Zugriff erzeugt Kosten. Ein ChatGPT Abo reicht nicht, es muss ein API-Guthaben vorhanden sein.

Für GEMINI: Einen API Key unter <https://aistudio.google.com/apikey> erzeugen.

```
if is_colab:
    from google.colab import userdata
    api_key_openai = userdata.get('OPENAI_API_KEY')
    api_key_hf = userdata.get("HF")
    api_key_gemini = userdata.get("GEMINI")
else:
    from pathlib import Path
    from dotenv import load_dotenv, find_dotenv
    import os
    load_dotenv(find_dotenv())
    api_key_openai = os.getenv("OPENAI_API_KEY")
    api_key_hf = os.getenv("HF")
    api_key_gemini = os.getenv("GEMINI")
```

✓ Schritt 04-02 – Open Source Embedding Modell Hugging Face

logout() entfernt eventuell veraltete Hugging-Face-Tokens aus der lokalen Cache-Datei, damit nach einem fehlgeschlagenen Auth-Versuch keine 401-Fehler mehr auftreten.

login(api_key_hf) hinterlegt anschließend deinen aktuellen API-Key, sodass der Transformer-Checkpoint später automatisch aus dem Hub geladen werden kann.

Mit HuggingFaceEmbeddings(model_name="sentence-transformers/all-mpnet-base-v2") erstellst du einen LangChain-Wrapper, der das Sentence-Transformer-Modell all-mpnet-base-v2 intern als Encoder nutzt.

embed_documents([...]) feuert den Encoder auf jede Text-Chunk-Zeile aus final_chunks und liefert für jeden Chunk einen hochdimensionalen Vektor zurück – die Basis für semantische Suche oder Retrieval-Augmented-Generation.

Der nachfolgende Codeabschnitt kann einige Minuten dauern und mit einer Warnung beginnen: Not logged in!

WARNING:huggingface_hub.login:Not logged in!

```
from huggingface_hub import logout, login
logout()
print("Login...")
login(api_key_hf)

print("Erstelle Embeddings")
embeddings_os = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-mpnet-base-v2"
)

hf_embeddings_os = embeddings_os.embed_documents([chunk.page_content for chunk in final_chunks])
print("Embeddings mit all-mpnet-base-v2 erstellt")
```

```
↗ Login...
  Erstelle Embeddings
  Embeddings mit all-mpnet-base-v2 erstellt
```

✓ Schritt 04-03 - Embeddings prüfen/anzeigen

```
print(f"Länge des Embedding Vektors ist {len(hf_embeddings_os[0])}")
print(f"Das Embedding Objekt ist ein Array mit Anzahl Zeilen (Anzahl Chunks) X Vektorlänge {len(hf_embeddings_os)} X {len(hf_embeddings_o[0])}")
print(f"Der erste Vektor: {hf_embeddings_os[0]}")
```

```
↗ Länge des Embedding Vektors ist 768
  Das Embedding Objekt ist ein Array mit Anzahl Zeilen (Anzahl Chunks) X Vektorlänge 19 X 768
  Der erste Vektor: [0.07044349610805511, -0.033378567546606064, -0.04742196947336197, 0.027081340551376343, 0.03717686980962753, 0.04...
```

✓ Schritt 04-04 – OpenAI Embedding Modell

Optionaler Schritt: zum Vergleich werden Embeddings mit OpenAI erzeugt.

```
embeddings = OpenAIEmbeddings(model="text-embedding-3-small",
                               openai_api_key=api_key_openai )

openai_embeddings = embeddings.embed_documents([chunk.page_content for chunk in final_chunks])

print("Embeddings mit OpenAI text-embedding-3-small erstellt")
```

```
↗ Embeddings mit OpenAI text-embedding-3-small erstellt
```

✓ Schritt 04-05 - Embeddings prüfen/anzeigen

Optionaler Schritt für OpenAI.

```
print(f"Länge des Embedding Vektors ist {len(openai_embeddings[0])}")
print(f"Das Embedding Objekt ist ein Array mit Anzahl Zeilen (Anzahl Chunks) X Vektorlänge {len(openai_embeddings)} X {len(openai_embeddings[0])}")
print(f"Der erste Vektor: {openai_embeddings[0]}")
```

```
↗ Länge des Embedding Vektors ist 1536
  Das Embedding Objekt ist ein Array mit Anzahl Zeilen (Anzahl Chunks) X Vektorlänge 19 X 1536
  Der erste Vektor: [0.009730181656777859, 0.008005942218005657, 0.02222522348165512, -0.01271531917154789, 0.006327276583760977, 0.00...
```

✓ 05 📖 Vector Store – Mathe statt Magie rund um FAISS, DuckDB, PostgreSQL

Speicherung der Vektoren in einer Datenbank (z.B. PostgreSQL pgvector, Qdrant, Chroma, Milvus, Oracle, etc) oder einem Vektroindex (z.B. FAISS "Facebook AI Similarity Search"). Eine Datenbank kann neben den Vektoren auch die Nutzdaten speichern

✓ Schritt 05-01 FAISS konfigurieren und Vektoren erzeugen

`faiss.IndexFlatIP(...)` erzeugt zunächst einen flachen FAISS-Index, der rein das Skalarprodukt auswertet; die Vektordimension wird aus dem ersten Embedding abgeleitet.

Der LangChain-Wrapper `FAISS(...)` verbindet diesen Index mit der Embedder-Funktion (`embeddings_os`), einem kleinen In-Memory-Docstore und einem leeren Mapping `index_to_docstore_id`, sodass jede gespeicherte Vektorposition wieder auf ihr Ursprungs-Dokument zurückgeführt werden kann.

Mit `distance_strategy=DistanceStrategy.COSINE` teilt man LangChain mit, die Ähnlichkeit als Kosinus zu interpretieren – dafür sollten die Dokument-Vektoren vor dem Einfügen (automatisch oder manuell) auf Einheitslänge normiert werden, weil `IndexFlatIP` selbst nur das Dot-Product speichert.

`add_documents(final_chunks)` lässt LangChain jeden Chunk einbetten, optional normalisieren und anschließend sowohl im FAISS-Index als auch im Docstore ablegen; damit ist der semantische Suchraum für spätere Queries vollständig eingerichtet. Das `add_documents()`-Aufruf liefert die internen Doc-IDs, die LangChain beim Einfügen erzeugt. Diese sind in der Anzeige zu sehen.

Warum Docstore? Für den reinen FAISS-Algorithmus brauchst du nur den Index – dort liegen die Vektorkoordinaten. Der docstore ist lediglich eine Komfortschicht von LangChain: Er merkt sich, welches Document (inkl. Metadata) zu welcher Index-Position gehört. Methoden wie `similarity_search()` oder `as_retriever()` holen darüber automatisch den Originaltext zurück. Verzichtest du auf den Docstore, musst du nach einer Vektorabfrage die Treffer-IDs selbst in ein externes Mapping auf deine Dokumente auflösen – oder gleich ganz ohne LangChain-Wrapper direkt mit FAISS arbeiten.

Warum `normalize=true`? wenn doch Cosinus ohne Normierung funktioniert? FAISS-Index `IndexFlatIP` kennt aber nur das reine Skalarprodukt (inner product) und berechnet beim Suchen keinerlei Normen. Damit sein Ergebnis dennoch exakt dem Cosinus entspricht, müssen alle Vektoren schon beim Einfügen auf Einheitslänge normalisiert werden

```
index = faiss.IndexFlatIP(len(hf_embeddings_os[0]))
```

```
vector_store = FAISS(
    embedding_function = embeddings_os,
    index               = index,
    docstore            = InMemoryDocstore(),
    index_to_docstore_id= {},
    normalize_l2=True,
    distance_strategy   = DistanceStrategy.COSINE
)
```

```
vector_store.add_documents(final_chunks)
```

```
→ ['6a0aec2f-be55-4eed-a315-fc51c6f62d46',
   '69dbada2-103a-4ee2-b302-a2c153d79680',
   '99b5c7b8-10e0-43f0-9b57-c327c399140a',
   '74a4016b-9681-4906-a01c-7c4cf56740c5',
   'b24faec8-f451-4bf8-934d-41190c17e9d8',
   'ac2ea835-83fa-4f19-86a1-cef2c23a327b',
   '35259941-2db3-42f0-9663-751ad6b71427',
   '79695042-d87c-4899-8e25-9637ac4914a4',
   '9f8eae99-bfad-416a-98dc-0aeb3b5dbaba',
   '565f0d60-6f89-4ea7-a671-6251be8027b4',
   '848da56d-f13e-407b-88c7-ff9f39c01082',
   '2c887881-8adb-471f-9268-3491d8c66fdb',
   '8ff41c4f-845a-43f7-8228-47d4530d694a',
   '5ac905aa-5199-4e2e-ba3e-75a8110ac991',
   '7aa2ad77-c765-4f6b-8560-2bd350adab67',
   '0db43c8a-3374-492e-919a-852e507eb0c0',
   '00086264-917e-4001-bcb4-d66595f9596c',
   '2c26b57a-8ad8-425c-b6c2-4e4307b08bff',
   '490cd24f-76d3-4638-9111-45ff4c6ce185']
```

✓ Schritt 05-02 - FAISS Index speichern

Der eingebaute `InMemoryDocstore` lebt nur so lange wie der Python-Prozess; speicherst du lediglich den Roh-Index mit `faiss.write_index`, überdauern nur die Vektordaten – die Texte sind danach weg. Nutzt du dagegen den LangChain-Wrapper und rufst `vector_store.save_local("pfad")` auf, legt er zwei Dateien an: `index.faiss` für die Vektoren und `docstore.pkl` für Docs & Mapping. Beim nächsten Start kannst du alles mit `FAISS.load_local("pfad", embeddings)` vollständig wiederherstellen, inklusive Docstore.

```
vector_store.save_local(folder_path="Data", index_name="Wiki_index")
```

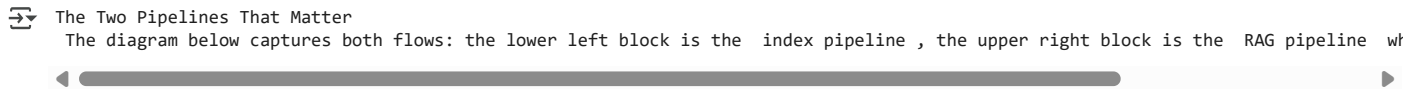
✓ Schritt 05-03 - Ähnlichkeitssuche im FAISS Index

In der Vektordatenbank/Vektorindex wird eine Ähnlichkeitssuche durchgeführt und das ähnlichste Dokument wird angezeigt.

```
query = "Which pipelines are used in RAG?"
```

```
docs = vector_store.similarity_search(query)
```

```
print(docs[0].page_content)
```



✓ Schritt 05-04 DuckDB konfigurieren und Vektoren erzeugen

DuckDB ist eine spaltenorientierte, in-process OLAP-Datenbank („SQLite für Analytics“). Sie läuft ohne Server, liest direkt Parquet/CSV/JSON, beschleunigt größere Abfragen dank Vektor-Ausführung und lässt sich nahtlos in Python, R oder C++ einbetten – ideal, wenn man analytische SQL-Power in einem einzigen Datei-Container benötigt.

Der Code baut einen Vektor-Store auf DuckDB-Basis:

- DuckDB.from_documents(...) bettet jeden Chunk mit embeddings_os ein und legt die resultierenden Vektoren samt Original-Text in einer neuen DuckDB-Tabelle ab.
- connection sorgt dafür, dass alles sofort in eine Datei geschrieben wird und nach Kernel-Neustarts weiterverwendbar ist.

```
conn = duckdb.connect("Data/duckdb_vectors.duckdb")
```

```
vector_store_duckDB = DuckDB.from_documents(
    documents = final_chunks,
    embedding = embeddings_os,
    connection = conn,
)
```

```
conn.close()
```

✓ Schritt 05-05 Tabelle in DuckDB anzeigen

Die Nutzdaten (Texte + Metadaten) wandern zusammen mit den Vektoren in dieselbe .duckdb-Datei. LangChain legt dort intern eine Tabelle embeddings für Metadaten und Nutzdaten an. Diese wird nun mittels SQL abgefragt.

Die Ausgabe zeigt, dass zum Beispiel wichtige Metadaten wie Titel und URL fehlen. Solche Metadaten sollten hinzugefügt werden, z.B. durch weitere Spalten (titel, url) in der Tabelle oder alternativ im metadata-Feld.

```
con = duckdb.connect("Data/duckdb_vectors.duckdb")
```

```
# Schema inspizieren
print(con.execute("DESCRIBE embeddings").fetchdf())
```

```
# Erste Einträge ansehen
df = con.execute("""
    SELECT id,
           left(text, 60) AS preview,
           embedding,
           metadata
    FROM embeddings
    LIMIT 5
    """).fetchdf()
```

```
display(df)
con.close()
```

12/19

Retrieved passages are augmented onto the prompt.

The LLM reads this enriched prompt and generates the final answer.

Ergebnis #3

Ähnlichkeit (similarity): 0.3996

Inhalt:

Quick Recap – Why Your RAG Pipeline Matters

Cleaning → Chunking → Embedding → Vector Storage → Retrieval → Prompt Augmentation. Those six steps turn raw, ever-changing document

✓ Schritt 05-07 PostgreSQL konfigurieren und Vektoren erzeugen

Für diesen Schritt muss eine lokale PostgreSQL mit pgvector existieren!

```
docker run -d -p 5432:5432 -e POSTGRES_PASSWORD=... --name pgvector pgvector/pgvector:pg17
```

pgvector-Extension muss im Container einmalig aktiviert werden: `docker exec -it pgvector psql -U postgres -c "CREATE EXTENSION IF NOT EXISTS vector;"`

```
pip install pgvector psycopg2-binary
```

```

Collecting pgvector
  Downloading pgvector-0.4.1-py3-none-any.whl.metadata (18 kB)
Collecting psycopg2-binary
  Downloading psycopg2_binary-2.9.10-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.9 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from pgvector) (2.0.2)
Downloading pgvector-0.4.1-py3-none-any.whl (27 kB)
Downloading psycopg2_binary-2.9.10-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.0 MB)
    3.0/3.0 MB 40.4 MB/s eta 0:00:00
Installing collected packages: psycopg2-binary, pgvector
Successfully installed pgvector-0.4.1 psycopg2-binary-2.9.10

```

```

from langchain_community.vectorstores import PGVector
from langchain_community.vectorstores.pgvector import DistanceStrategy

# Verbindungs-String zu deinem Docker-Container
CONN_STR = "postgresql+psycopg2://postgres:password@localhost:5432/postgres"

# Vector-Store aus Dokument-Chunks erstellen
vector_store_pg = PGVector.from_documents(
    documents = final_chunks,
    embedding = embeddings_os,
    connection_string= CONN_STR,
    distance_strategy= DistanceStrategy.COSINE
)

# Query wie gewohnt
docs = vector_store_pg.similarity_search("What are key stages?", k=3)
print(docs[0].page_content)

```

✓ 06 🌀 Prompt pimpen – Quellen smart einbauen

Der Prompt wird angereichert mit dem Inhalt der Ähnlichkeitssuche.

✓ Schritt 06-01 - Daten für Prompt abfragen (aus Vektorfordatenbank/index holen)

```

# Abfrage definieren
query = "Which pipelines are used in RAG?"

# Ähnlichkeitssuche durchführen
retrieved_docs = vector_store_duckDB.similarity_search(query, k=2) # Top-2 Chunks erhalten

# Ergebnisse anzeigen

import textwrap

for i, doc in enumerate(retrieved_docs):
    print(textwrap.fill(f"\nRetrieved Chunk {i+1}:\n{doc.page_content}",width=100))
    print("\n\n")

```

Retrieved Chunk 1: The Two Pipelines That Matter The diagram below captures both flows: the lower left block is the index pipeline , the upper right block is the RAG pipeline where retrieval meets generation.

Retrieved Chunk 2: RAG (Query-Time) Pipeline – Answer with context The query pipeline uses the knowledge base created in the vector store by the index pipeline to answer questions. A user question arrives. Retriever does a similarity search in the vector store. Retrieved passages are augmented onto the prompt. The LLM reads this enriched prompt and generates the final answer.

✓ Schritt 06-02 Prompt erstellen

Der Code fasst zunächst den Inhalt der beiden relevantesten Dokument-Chunks (retrieved_docs[0] und [1]) zu einem einzigen retrieved_context-String zusammen. Anschließend wird daraus ein augmented_prompt gebaut, der dem LLM klar vorgibt, nur auf Basis dieses Kontexts die gestellte Frage (query) zu beantworten und bei fehlender Information ehrlich mit „I don't know“ zu reagieren. Mit textwrap.fill wird der Prompt schließlich formatiert, damit er beim Ausdrucken im Notebook leserlich umbricht.

```
# Die ersten beiden Dokumente werden verwendet
retrieved_context=retrieved_docs[0].page_content + retrieved_docs[1].page_content

# Prompt zusammenstellen
augmented_prompt=f"""

Given the context below answer the question.

Question: {query}

Context : {retrieved_context}

Remember to answer only based on the context provided and not from any other source.

If the question cannot be answered based on the provided context, say I don't know.

"""

print(textwrap.fill(augmented_prompt,width=150))
```

➡ Given the context below answer the question. Question: Which pipelines are used in RAG? Context : The Two Pipelines That Matter below captures both flows: the lower left block is the index pipeline , the upper right block is the RAG pipeline where retrieval generation.RAG (Query-Time) Pipeline – Answer with context The query pipeline uses the knowledge base created in the vector : pipeline to answer questions. A user question arrives. Retriever does a similarity search in the vector store. are augmented onto the prompt. The LLM reads this enriched prompt and generates the final answer. Remember to answer or context provided and not from any other source. If the question cannot be answered based on the provided context, say I don't know

✓ 07 🌟 Chat GPTiert – Kontext drin, Halluzinationen draußen!?

Der Prompt ist erstellt und die Anfrage kann nun an das LLM gesendet werden.

✓ Schritt 07-01 - Prompt an öffentliche gehostetes GEMINI Modell senden und Antwort anzeigen

genai.configure(api_key=...) legt den Google-API-Key fest, damit nachfolgende Aufrufe authentifiziert sind. GenerativeModel("gemini-1.5-flash") instanziiert die schnelle Gemini-1.5-Flash-Variante. Die Klasse GeminiRunnable verpackt das Modell in ein LangChain-kompatibles Runnable: invoke() wandelt jeden Input in einen String, ruft model.generate_content(prompt) auf und gibt nur den reinen Antworttext zurück. Mit gemini_llm.invoke(augmented_prompt) wird genau dieser Ablauf ausgeführt und die Antwort für das zuvor gebaute Retrieval-Prompt ausgegeben.

```
genai.configure(api_key=api_key_gemini)

model = generativeai.GenerativeModel("gemini-1.5-flash")

class GeminiRunnable(Runnable):
    def invoke(self, input, config=None):
        prompt = input.to_string() if hasattr(input, "to_string") else str(input)
        response = model.generate_content(prompt)
        return response.text

gemini_llm = GeminiRunnable()

response_gemini = gemini_llm.invoke(augmented_prompt)
print(response_gemini)
```

➦ Based on the provided text, there are two pipelines: the index pipeline and the RAG pipeline (also referred to as the query pipeline)

✓ Schritt 07-02 - Prompt an OpenAI senden und Antwort anzeigen

ChatOpenAI legt eine Chat-Schnittstelle zu OpenAI fest; hier wird das kompakte Modell „gpt-4o-mini“ genutzt und mit temperature=0 deterministisch gemacht. Mit max_tokens=None übernimmt die API ihr Standardlimit, timeout=None wartet unbegrenzt, und max_retries=2 erlaubt zwei automatische Wiederholungen bei Netz-/Rate-Limit-Fehlern. messages enthält genau eine Rolle-Inhalts-Kombination – der Retrieval-Prompt steht als „human“-Nachricht darin – und wird via llm.invoke(messages) an das Modell geschickt. Die Methode liefert ein AIMessage-Objekt zurück (ai_msg), dessen .content den finalen Antworttext des LLM enthält.

```
from langchain_openai import ChatOpenAI
```

```
# LLM Anfrage konfigurieren
llm = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0,
    max_tokens=None,
    timeout=None,
    max_retries=2,
    openai_api_key=api_key_openai
)
```

```
messages=[("human", augmented_prompt)]
```

```
ai_msg = llm.invoke(messages)
```

```
# Antwort aus dem Response-Objekt extrahieren
response_openai=ai_msg.content
```

```
print(response_openai)
```

➦ The pipelines used in RAG are the index pipeline and the RAG (Query-Time) pipeline. The index pipeline creates a knowledge base in t

✓ 08 Evaluierung - Ein paar einfache Checks

✓ Schritt 08-01 Einfache Evaluierung (1)

Evaluierung durch Vergleich von erwarteten Schlüsselwörtern in der Antwort. Vergleich mit Antwort von GEMINI.

```
ground_truth = {
    "Which pipelines are used in RAG?": ["Index Pipeline", "RAG Pipeline"],
}
```

```
treffer = sum([1 for wort in ground_truth["Which pipelines are used in RAG?"] if wort.lower() in response_gemini.lower()])
print(f"Antwort: {response_gemini}\nTreffer: {treffer}/{len(ground_truth['Which pipelines are used in RAG?'])}")
```

➦ Antwort: Based on the provided text, there are two pipelines: the index pipeline and the RAG pipeline (also referred to as the query

Treffer: 2/2

✓ Schritt 08-02 Einfache Evaluierung (2)

Evaluierung durch Vergleich von erwarteten Schlüsselwörtern in der Antwort. Vergleich mit Antwort von OpenAI.

```
ground_truth = {
    "Which pipelines are used in RAG?": ["Index Pipeline", "RAG Pipeline"],
}
```

```
treffer = sum([1 for wort in ground_truth["Which pipelines are used in RAG?"] if wort.lower() in response_openai.lower()])
print(f"Antwort: {response_openai}\nTreffer: {treffer}/{len(ground_truth['Which pipelines are used in RAG?'])}")
```

➦ Antwort: The pipelines used in RAG are the index pipeline and the RAG (Query-Time) pipeline. The index pipeline creates a knowledge
Treffer: 2/2

✓ Schritt 08-03 Evaluierung durch LLM (1)

Evaluierung der Antwort von Gemini durch OpenAI LLM

- correctness: Ist die Antwort sachlich korrekt?
- conciseness: Ist sie knapp & auf den Punkt?
- relevance: Beantwortet sie die Frage?
- coherence: Klingt sie flüssig & verständlich?

```
from langchain.evaluation import load_evaluator
from langchain_openai import ChatOpenAI

# correctness, relevance, conciseness, coherence
### Schritt 08-03 Evaluierung durch LLM (1)
evaluator = load_evaluator("labeled_criteria", criteria="conciseness", llm=ChatOpenAI(openai_api_key=api_key_openai))

ergebnis = evaluator.evaluate_strings(
    input="Which pipelines are used in RAG?",
    prediction=response_gemini,
    reference="Index Pipeline, RAG (Query-Time) pipeline"
)
print(ergebnis["score"], ergebnis["reasoning"])
```

🔄 1 Step by step reasoning:

1. Does the submission directly address the question asked?
 - Yes, the submission mentions the two pipelines used in RAG.
2. Is the submission concise and to the point?
 - The submission is concise and clearly states the two pipelines mentioned in the reference text.

Therefore, the submission meets the criteria.

✓ Schritt 08-04 Evaluierung durch LLM (2)

Evaluierung der Antwort von OpenAI durch OpenAI LLM

- correctness: Ist die Antwort sachlich korrekt?
- conciseness: Ist sie knapp & auf den Punkt?
- relevance: Beantwortet sie die Frage?
- coherence: Klingt sie flüssig & verständlich?

```
from langchain.evaluation import load_evaluator
from langchain_openai import ChatOpenAI

evaluator = load_evaluator("labeled_criteria", criteria="conciseness", llm=ChatOpenAI(openai_api_key=api_key_openai))

ergebnis = evaluator.evaluate_strings(
    input="Which pipelines are used in RAG?",
    prediction=response_openai,
    reference="Index Pipeline, RAG (Query-Time) pipeline"
)
print(ergebnis["score"], ergebnis["reasoning"])
```

🔄 1 1. The submission is concise and to the point:

- The submission mentions two pipelines used in RAG: the index pipeline and the RAG (Query-Time) pipeline.
- It clearly describes the purpose of each pipeline: the index pipeline creates a knowledge base in the vector store, while the RAG

Based on the step by step reasoning, the submission meets the criteria for conciseness.

Y

✓ Gesamtablauf mit Index-Pipeline und RAG-Pipeline

✓ Index-Pipeline inkl. Speicherung für viele Dokumente

```
# -----
# 0) VORBEREITUNG: mehrere URLs in eine Liste schreiben
# -----
urls = [
    "https://en.wikipedia.org/wiki/Retrieval-augmented_generation",
    "https://en.wikipedia.org/wiki/Vector_database",
```



```

"https://en.wikipedia.org/wiki/Hierarchical_navigable_small_world",
"https://en.wikipedia.org/wiki/Model_Context_Protocol",
"https://en.wikipedia.org/wiki/Multi-agent_system",
# ... füge hier beliebig weitere Links an
]

# -----
# 1) ALLE HTML-SEITEN PARALLEL LADEN
# -----
HTMLloader = AsyncHTMLLoader(urls)          # <- akzeptiert jetzt eine Liste
html_pages_ori = HTMLloader.load()          # Liste von Document-Objekten

# -----
# 2) SECTIONS SPLITTEN & CHUNKS BILDEN
# -----
sections_to_split_on = [
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("table", "Table"),
]

splitter = HTMLSectionSplitter(sections_to_split_on)

text_splitter = RecursiveCharacterTextSplitter(
    separators=[" ", "\n", "."],
    chunk_size=5000,
    chunk_overlap=50,
)

all_chunks = []          # sammelt alle Chunks aller Dokumente

for doc in html_pages_ori:
    doc_copy = copy.deepcopy(doc)
    split_content = splitter.split_text(doc_copy.page_content)
    chunks = text_splitter.split_documents(split_content)
    all_chunks.extend(chunks)

print(f"Gesamtzahl erzeugter Chunks: {len(all_chunks)}")

# -----
# 3) HUGGINGFACE EMBEDDINGS
# -----
logout()
print("Starting login...")
login(api_key_hf)

embeddings_os = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-mpnet-base-v2"
)

# -----
# 4) VEKTORDB AUFBAUEN UND DOKUMENTE HINZUFÜGEN
# -----
print("Start Einfügen in VektorDB")
conn = duckdb.connect("Data/duckdb_vectors.duckdb")

# Der folgende Befehl entfernt alle Zeilen
# Befehl entfernen, wenn keine Daten gelöscht werden sollen
conn.execute("DELETE FROM embeddings")

vector_store_duckDB = DuckDB.from_documents(
    documents = all_chunks,
    embedding = embeddings_os,
    connection = conn,
)

conn.close()
print("Alle Dokumente wurden indiziert!")

🔄 Fetching pages: 100%|#####| 5/5 [00:00<00:00, 13.58it/s]
Gesamtzahl erzeugter Chunks: 66
Starting login...
Start Einfügen in VektorDB
Alle Dokumente wurden indiziert!

```

✓ DuckDB abfragen

```
con = duckdb.connect("Data/duckdb_vectors.duckdb")
```

```
# Schema inspizieren
print(con.execute("DESCRIBE embeddings").fetchdf())

df = con.execute("""
    SELECT count(*) as cnt
    FROM embeddings
""").fetchdf()
print("\nAnzahl Datensätze:")
display(df)

# Erste Einträge ansehen
df = con.execute("""
    SELECT id,
           left(text, 60) AS preview,
           embedding,
           metadata
    FROM embeddings
    LIMIT 5
""").fetchdf()
print("\nBeispiel-Datensätze:")
display(df)
con.close()
```

	column_name	column_type	null	key	default	extra
0	id	VARCHAR	NO	PRI	None	None
1	text	VARCHAR	YES	None	None	None
2	embedding	FLOAT[]	YES	None	None	None
3	metadata	VARCHAR	YES	None	None	None

Anzahl Datensätze:

	cnt
0	66

Beispiel-Datensätze:

	id	preview	embedding	metadata
0	33c03f05-274e-4fe4-a41d-c6db6d185155	Jump to content \n \n \n \n \n \n \n \n Main m...	[0.02118021, -0.06125822, -0.022656694, 0.0098...	{"Header 1": "#TITLE#"}
1	8561b070-865b-4c73-b908-a1307a6700e1	Contents \n move to sidebar \n hide \n \n \n \n ...	[0.060398534, 0.011601106, -0.029227823, 0.037...	{"Header 2": "Contents"}
2	3aff8785-b267-41d0-ae86-8ef07ee6de05	Retrieval-augmented generation \n \n \n \n 12 ...	[0.071465515, -0.0012724106, -0.043260608, -0.0...	{"Header 1": "Retrieval-augmented generation"}
3	40f6eb88-d9fc-4856-84e7-c2ff7c9fca8b	RAG and LLM Limitations \n [edit] \n \n LLMs...	[0.035701912, -0.0039028309, -0.006321461, 0.0...	{"Header 2": "RAG and LLM Limitations"}
4	c8e6df03-4351-48d1-bb7c-4eaa61f4e997	Process \n [edit] \n \n Retrieval-augmented ...	[0.052486185, -0.04693809, -0.02241138, 0.0072...	{"Header 2": "Process"}

Nächste Schritte: [Code mit df generieren](#) [Empfohlene Diagramme ansehen](#) [New interactive sheet](#)

✓ RAG Pipeline

```
query = "What is a vector database?"
top_k = 1

# -----
# 1) Verbindung & Vector-Store laden
# -----
# Embedding-Modell **muss** identisch zu dem im Indexschritt sein
embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-mpnet-base-v2"
)

# Verbindung zur bestehenden DB-Datei
conn = duckdb.connect("Data/duckdb_vectors.duckdb")

# Vector-Store-Objekt erstellen
vector_store = DuckDB(
    connection = conn,
    embedding = embeddings,
)

# -----
# 2) Retrieval
# -----
retrieved_docs = vector_store.similarity_search(query, k=top_k) # :contentReference[oaicite:1]{index=1}
```

```

#for i, doc in enumerate(retrieved_docs, start=1):
#    print(textwrap.fill(f"\nRetrieved Chunk {i}:\n{doc.page_content}", width=100))
#    print("\n" + "-"*110)

retrieved_context = "\n\n".join(d.page_content for d in retrieved_docs)

# -----
# 3) Prompt für Gemini bauen
# -----
augmented_prompt = f"""
Given the following context, answer the question.

Question: {query}

Context:
{retrieved_context}

Answer ONLY from the context provided. If the answer is not in the context, say "I don't know".
"""

#print(textwrap.fill(augmented_prompt, width=120))

# -----
# 4) Gemini-Aufruf
# -----
genai.configure(api_key=api_key_gemini)
model = genai.GenerativeModel("gemini-1.5-flash")

class GeminiRunnable(Runnable):
    def invoke(self, input, config=None):
        prompt_str = str(input)
        response = model.generate_content(prompt_str)
        return response.text

gemini_llm = GeminiRunnable()

response = gemini_llm.invoke(augmented_prompt)
print("\nAntwort:\n" + textwrap.fill(response, width=100))

# -----
# 5) Aufräumen
# -----
conn.close()


```

↔

Antwort:
A vector database, vector store, or vector search engine is a database that uses the vector space