1) Pakete installieren und OpenAI-API-Key laden

```
# Pakete installieren

# Schnittstelle zu OpenAI-Modellen
!pip -q install langchain_openai

# Zusätzliche LangChain-Komponenten (Loader, Vectorstores)
!pip -q install langchain-community

# PDF-Parser
!pip -q install pypdf

# In-Memory-Vektorstore
!pip -q install docarray

# Tokeenizer für OpenAI-Modelle
!pip -q install tiktoken

# OpenAI-API-Key laden
from google.colab import userdata
OPENAI_API_KEY = userdata.get('apikey_ab')

# Bricht ab, falls kein API-Key vorhanden ist
assert OPENAI_API_KEY
```

2) PDF aus GitHub laden

```
# Bibliothek für HTTP-Anfragen (Dateien, APIs etc.)
import requests

# LangChain-Loader für PDFs
# Nutzt pypdf, um Seiten auszulesen und in Dokument-
Objekte (Text + Metadaten) zu konvertieren
from langchain_community.document_loaders import PyPDFLoader

# GitHub Raw-Link zur PDF
url = "https://raw.githubusercontent.com/x8bean/Machine-Learning-Assignment/main/
Wissensquelle.pdf"

# Download + Speichern als temporäre Datei in Colab
pdf_path = "/content/tmp.pdf"

# Öffnet die Zieldatei im Schreib-/
Binärmodus ("wb"), lädt die PDF von der angegebenen URL herunter (mit 30 Sekunden
Timeout) und schreibt den Inhalt direkt in diese Datei
with open(pdf_path, "wb") as f:
    f.write(requests.get(url, timeout=30).content)

# PDF in LangChain-Dokumente laden
loader = PyPDFLoader(pdf_path)
page_docs = loader.load()

# Überprüfung, ob die PDF die Vorgabe von ≤ 10 Seiten erfüllt
assert len(page_docs) <= 10, f"PDF hat {len(page_docs)} Seiten (>10)."

# Anzahl der Seiten wird ausgegeben
```

```
print("Seiten geladen:", len(page_docs))
```

3) Chunking

```
# Teilt Texte rekursiv anhand von Trennzeichen (Absatz, Satz, Wort) in Chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Initialisiert den Textsplitter mit Regeln für Größe, Überlappung und Trennzeichen-Priorität
splitter = RecursiveCharacterTextSplitter(
    chunk_size=800, # Maximale Zeichen pro Chunk
    chunk_overlap=120, # Überlappung zwischen Chunks (verhindert Informationsverlust)
    separators=["\n\n", "\n", ".", " ", ""]
)

# Wendet das Chunking auf alle Seiten-
Dokumente an, erzeugt Liste von kleineren Document-Objekten
chunks = splitter.split_documents(page_docs)

# Anzahl der erzeugten Chunks wird ausgegeben
print("Chunks erstellt:", len(chunks))
```

4) Embedding

```
# Schnittstelle zu OpenAI-Embedding-
API, um Text in Embeddings zu verwandeln, die den Sinn des Textes darstellen
from langchain_openai import OpenAIEmbeddings

# In-Memory-Vektorstore, speichert Embedding-Vektoren und führt Ähnlichkeitssuche durch
from langchain_community.vectorstores import DocArrayInMemorySearch

# Erstellt Embedding-Funktion, die bei Aufruf OpenAI-API für Vektorisierung nutzt
embeddings = OpenAIEmbeddings(
    model="text-embedding-3-
small", # Modellauswahl für die Umwandlung von Text in Embeddings
    openai_api_key=OPENAI_API_KEY   # API-
Schlüssel für die Authentifizierung bei OpenAI
)

# Wandelt alle Chunks in Embeddings um und speichert sie im Vektorstore
vectorstore = DocArrayInMemorySearch.from_documents(chunks, embedding=embeddings)

# Erzeugt Retriever, der für eine Suchanfrage die Top-4 relevantesten Chunks ausgibt
retriever = vectorstore.as_retriever(
    search_type="mmr", # Maximal Marginal Relevance: bedeutet, dass nicht nur die
relevantesten Abschnitte ausgewählt werden, sondern auch möglichst unterschiedliche, um
Wiederholungen zu vermeiden
    search_kwargs={"k": 4, "fetch_k": 20} # k=4: gibt die 4 besten Treffer zurück (werden
später ins Prompt eingefügt); fetch_k=20: zieht zuerst die 20 besten Kandidaten, bevor
daraus die 4 ausgewählt werden
)
```

5) Implementierung des RAG-Systems

```
# Schnittstelle zu OpenAI-Chat-LLMs
from langchain_openai import ChatOpenAI

# Generiert strukturierte Prompts aus Template-Texten mit Platzhaltern
```

```
from langchain.prompts import ChatPromptTemplate

# Wandelt LLM-Ausgabe in reinen Text-String um
from langchain_core.output_parsers import StrOutputParser

# Um mehrere Verarbeitungsschritte miteinander zu verbinden:
# RunnableParallel: führt mehrere Aufgaben gleichzeitig aus
# RunnablePassthrough: gibt die Eingabe unverändert weiter
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

# Prompt-Template mit Platzhaltern für Kontext und Frage
template = """
Beantworte die Frage basierend auf dem Kontext.
Wenn Du die Frage nicht beantworten kannst, antworte "Ich weiß es nicht".

Context: {context}

Question: {question}
"""

# Erzeugt Prompt-Objekt, das bei Aufruf Platzhalter durch echte Werte ersetzt
prompt = ChatPromptTemplate.from_template(template)

# Initialisiert LLM mit Parametern
llm = ChatOpenAI(
    model="gpt-4o-mini", # Chat-Modellauswahl
    openai_api_key=OPENAI_API_KEY, # API-Schlüssel für die Authentifizierung bei OpenAI
    temperature=0.2, # Zufälligkeit der Ausgabe steuern
    max_tokens=200 # Antwortlänge begrenzen
)

# Erstellt Parser, der die reine Textausgabe extrahiert
parser = StrOutputParser()

# Kombiniert Retrieval und Frage in einer parallelen Struktur
setup = RunnableParallel(
    context=retriever, # Führt semantische Suche aus und liefert Chunks als Kontext
    question=RunnablePassthrough() # Leitet die Frage unverändert weiter
)

# Verknüpft alle Schritte: Retrieval --> Prompt --> LLM --> Ausgabe
chain = setup | prompt | llm | parser
```

6) Beispiel

```
# Beispielabfrage 1
print(chain.invoke("Wovon handelt das Dokument?"))

# Beispielabfrage 2
print(chain.invoke("Welche Unterschiede gab es bei den Zuschauerzahlen zwischen
Männern und Frauen während der WM 2014?"))
```