Conversational AI. Dialogsysteme, Chatbots, Assistenten

Veranstalter: Christoph Ringlstetter

Sitzung 3: Einführung zu Langchain

Was machen wir denn heute.

- Orga. Referate, Zulassung, Termine.
- Besprechung Langchain Konzepte Buch Kap 2,3,4
- Besprechung Langchain Erster Überblick Implementierungen

Basis der folgenendn Ausführungen das Langchain Buch und die Tutorials

https://python.langchain.com/docs/tutorials/

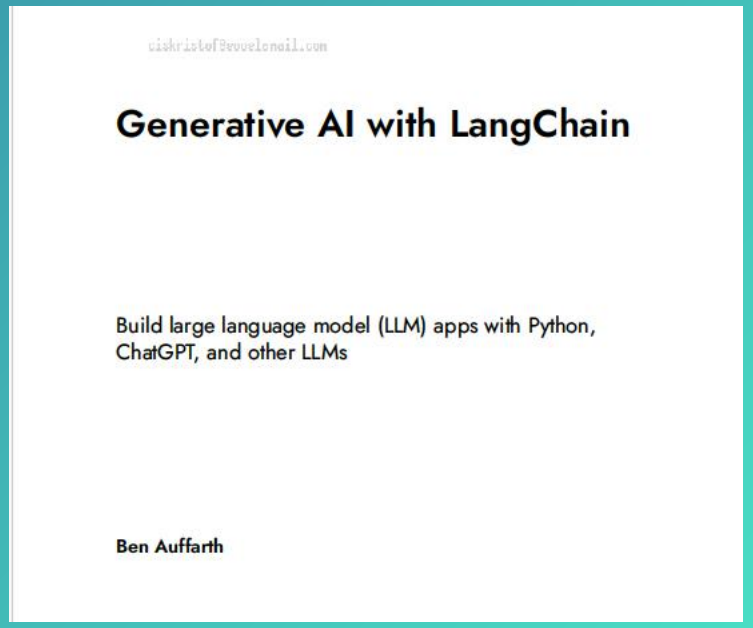
Basics

[Build a Simple LLM Application with LCEL]

[Build a Chatbot]

[Build vector stores and retrievers]

[Build an Agent]



Langchain was ist es?

Open Source Python Framework zur Entwicklung von LLM basierten Applikationen

Basis: LLM Benutzung + konversationeller Kontext + Persistenz: Agents und Memory

Support für Action Plans und Strategien --> robust, performant

Support für Memory und externes Wissen → weniger Halunzination

modular, Pipelines von Services (chains)

zielorientierte Interaktion mit Agenten

statefulness, open source + community support

Kap 2 im Buch

- Stochastic Parrots vs zielgerichtete Conversational AI
- Key Komponenten von LangChain
- Wie funktioniert es erste Beispiele
- Vergleich mit anderen Frameworks

Langchain ein Framework zur Entwicklung von LLM Applikationen

Serviceframework um Einschränkungen von bloßen LLMs zu überwinden

- externes Wissen fehlt
- fehlerhafte Antworten
- fehlerhaftes Schließen
- kann keine Aktionen auslösen: Änderung der Umwelt

Langchain um dynamische, datengetriebene Applikationen LLM agnostisch zu bauen: jenseits einfacher LLM API calls.

Konzepte: Chains, Action Plan generation, Memory.

Langchain zur Entwicklung von LLM Applikationen

Going beyond stochastic parots: „no true comprehension of the meaning behind the words“ Emily Bender: on the dangers of stochastic parots. 2021

Antworten verhindern die: falsch, irrelevant, unethisch, logisch sinnlos sind.

Für bloße LLMs auch Scaling wird nicht alles Lösen.
Compositionality Gap: Paper Measuring the Compositionality Gap: Ofir Press et al.
: müssen mit “wahrem Verständnis“ anreichern: prompting, chain of thought, retrieval+grounding -> Lernprozess im Umgang mit den Modellen.

Langchain zur Entwicklung von LLM Applikationen Limitierung der Modelle

Wissen veraltet -- CEO von Firma Y ist X

Keine Aktionen möglich – lösche meinen Urlaub

Halluzinationsrisiko – X verursacht durch Y

Bias, Diskrimination – Kreditvergabe, geschlechtszuordnungen

Transparenz der Ausgabe – Warum hast du das gesagt?

Kontextualität – Beziehe dich auf die Äusserung am Anfang unsers Gespr.

Verbindung zu externen Sourcen -- wie ist das Wetter.

Cutoff Problem: I dont know…

Einfache Berechnungen: 333 * 5 = falsch

Langchain zur Entwicklung von LLM Applikationen – Verbesserungen implementiert in Langchain

Retrieval Augementation: externe Daten hinzufügen, weniger Halluzinieren
Chaining: über Python Rechnen, über Websuche Aktualität verbesserung
Prompt Engineering: kritischer Kontext um Antworten anzupassen
Monitoring, Filtering, Reviews: Probleme erkennen, Filter: Listen, Klassifikatoren, Prinzipien, Menschl Judges
Memory: Konversationen versch. Sessions speichern. Kontext.
FineTuning: Model auf den Zweck adaptieren.
Insgesamt: raw model size allein kann die Unzulänglichkeiten nicht lösen

Langchain zur Entwicklung von LLM Applikationen – LLM App

Client → Frontend → Backend → Database

Traditionelle Software Applikation:

Client Layer: Benutzerinteraktion

Frontend Layer: Präsentation und Business Logik

Backend Layer: Programm Logik, APIs, Berechnungen

Database Layer: Speichern und Retrieval

alles was man an Funktionalität braucht muss explizit programmiert werden.

Langchain zur Entwicklung von LLM Applikationen – LLM App

Benutze ein LLM in einer Applikation um natürlich-sprachliche Prompts zu verstehen und um applikationsgerechte Antworten als Output zu generieren

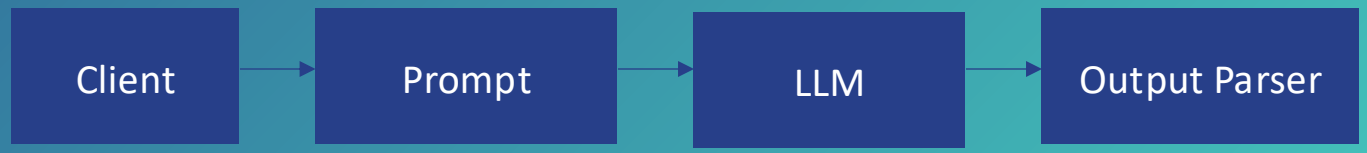
Client Layer

Prompt Engineering Layer

LMM Backend

Output Parsing

Optional: Integration mit externen Services via Function APIs: augment LLM



Langchain zur Entwicklung von LLM Applikationen – Anwendungen

Chatbots, virtuelle Assistenten

Intelligente Suchmaschinen

Automatische Content Erzeugung

Questions/Answers

Sentiment, Sumarization, Anonymization, Classification: NLP Tasks

Datenanalyse

Codegeneration

Task orientierte Dialogsysteme, Action Bots

→ Decision Making

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Chains: modulare Komponenten werden in wiederverwendbare pipelines zusammengeschaltet: mehrfache LLM Calls, Datenextraktion, Datenanalyse => chatbotartige soziale Interaktion

Chainelemente:

z.B. das Promttemplate: schicken einr formatierten Anweisungen an ein LLM

utility chains: LLMMath, SQLDatabaseChain

OpenAIModerationChain: um bestimmte Vorgehensweisen (policies) abzusichern.

Langchain zur Entwicklung von LLM Applikationen – Komponenten

LLMCheckerChain: statements verifizieren um die Anzahl falscher Antworten zu reduzieren: Selbstreflektionstechnik siehe nachher.
PAPER: SELFREFINE: https://arxiv.org/pdf/2303.17651
Iterative Refinement with Self-Feedback: Aman Madaan et al.

Router Chains: e.g. für Agenten um autonome Entscheidungen zu implementieren

Langchain zur Entwicklung von LLM Applikationen – Komponenten

CHAINS: Vorteile

modular

kompsitionell

lesbar

wartbar

wiederverwertbar

geeignet zur Toolintegration

produktiv: für Menschen gut verständlich planbare Schritte (am besten stateless, single responsibility input/output klar)

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Agenten: Systeme die dynamisch mit Benutzern und der Umgebung interagieren. Autonom, fähig Aktionen durchzuführen um Ziele und Aufgaben zu erledigen: die Umwelt verändern.

Agenten orchestrieren Chains ⇔ Chains integrieren Low-Level Module

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Vorteile von Agenten:

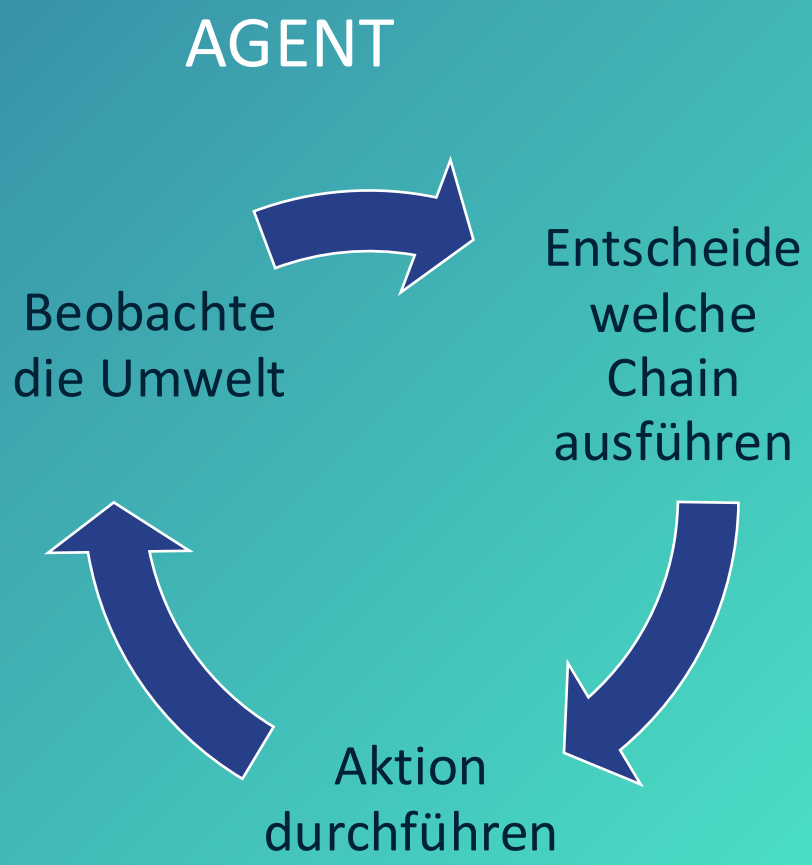
zielorienterte Ausführung von Schritten

dynamisches Antworten

stateful

robust

kompositionell



Langchain zur Entwicklung von LLM Applikationen – Komponenten

Memory: Persistenzzustand zwischen den Executions (Ausführungsschritte) einer Chain oder eines Agenten.

Z.B. Speichern des Chathistory Inhalts

=> Koheränz und Relevanz über die Zeit.

Fakten, Beziehungen, Ergebnisse – Deduktionen

Multistep Goals: Agent muss den Fortschritt tracken braucht also auch eine History. Möglichkeit der Rückkehr.

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Optionen von Memory Implementierung:

Conversation Buffer Memory: alle Nachrichten in der History. Latenz und Kosten gehen hoch – in jedem neuen Prompt alles Alte wieder drin.

Window Memory. Nur die letzten X Schritte werden wiederholt..

Knowledge Graph Memory: Zusammenfassung der Konversation als Knowledge Graph zur Integration in Prompts

Entity Memory: Persistenz von Zustand und Fakten in einer Datenbank.

Timemachine.

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Datenbankoptionen für Memory Implementierung. SQL, NOSQL

REDIS für HighPerformance Caching

AWS Dynamo für Cloud Infrastruktur

Spezialserver: Remembral, Motörhead für bestimmte Applikationen optimiert.

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Tools: Funktionen die der Agent aufruft um Aktionen auszuführen

Entscheidung wird dabei durch das LLM als Reasoning Engine emuliert

Beispiele: MT, Calculator, Maps, Weather, Stocks, Slides, Tables, Search Engines, Knowledge Graps (SPARQL) , Wikipedia, Shopping, Image Gen, Domainknowledge Tools (Pubchem).

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Zusammenfassung.

Langchain erlaubt die Implementation von fortgeschrittenen LLM Applikationen:

Modulare Komponenten verbinden LLMs mit Daten und Services: grundlegende Chatbotfähigkeiten, komplexes Reasoning, Persistenz (Memory)

Chains z.B.: Dokumente laden -> Embeddings für Retrieval -> Querying LLMs -> Parsen des Outputs -> Schreiben auf Memory.

Chains matchen Module auf Applikationsziele.

Agenten für wiederholte Aktionen. Beobachten, Planen, Updaten, Persistieren.

Langchain zur Entwicklung von LLM Applikationen – Komponenten

LLMs und Chat: connect, query interfaces: async, stream, batch

Document Loaders: Daten in Dokumentenformaten einlesen: text, video

Document Transformers: Manipulieren, split, filter, translate: Datenadaption

Text Embeddings: Vektorrepräsentation, Suche, Retrieval

Retrievers: Dokumente auf Grundlage der Query zurückgeben

Tools: Interfaces für Agenten um mit externen Sytemen zu interagieren

Agenten: Goal Driven, planen auf der Grundlage Beobachten der Umwelt

Toolkits: Initialisieren. Gruppieren.

Memory: Persistente Information, Workflows, read/write Session Daten

Callbacks: Monitoring von Chains. Z.b. Kosten OpenAI

Langchain zur Entwicklung von LLM Applikationen – Komponenten

Vektor Stores: Text Embeddings repräsentieren die Dokumente “semantisch“.
Stores für große Dokumente/Menten
Dokument Chunking: Dann zum LLM gesandt, LLM verarbeitet Vektoren direkt – achtung synchrone Embeddingtechnologie. Addon zum Input (Query) RAG Prompt
??? Später klären.
Wie genau wird der Dokumen pass in den Prompt für das LLM gemacht.
Integration für Vector Storage: alle möglichen Datenbanken. Alibaba Cloud Opensearch, Mata AI Annoy Library for Approximate Nearest Neighbor, Elastic, Chroma, Mong, PGVector Scikitlearn Voctorstore KNN Search → CH5 Eigene Vorlesung?

Langchain zur Entwicklung von LLM Applikationen – LANGCHAIN INTRO

Andere Frameworks.

Transformer Optimus
Deep Set AI/Haystack
langchain.ai/Langchain
microsoft /autogen
run llama/llama index

(data source: GitHub star history, https://star-history.com/):

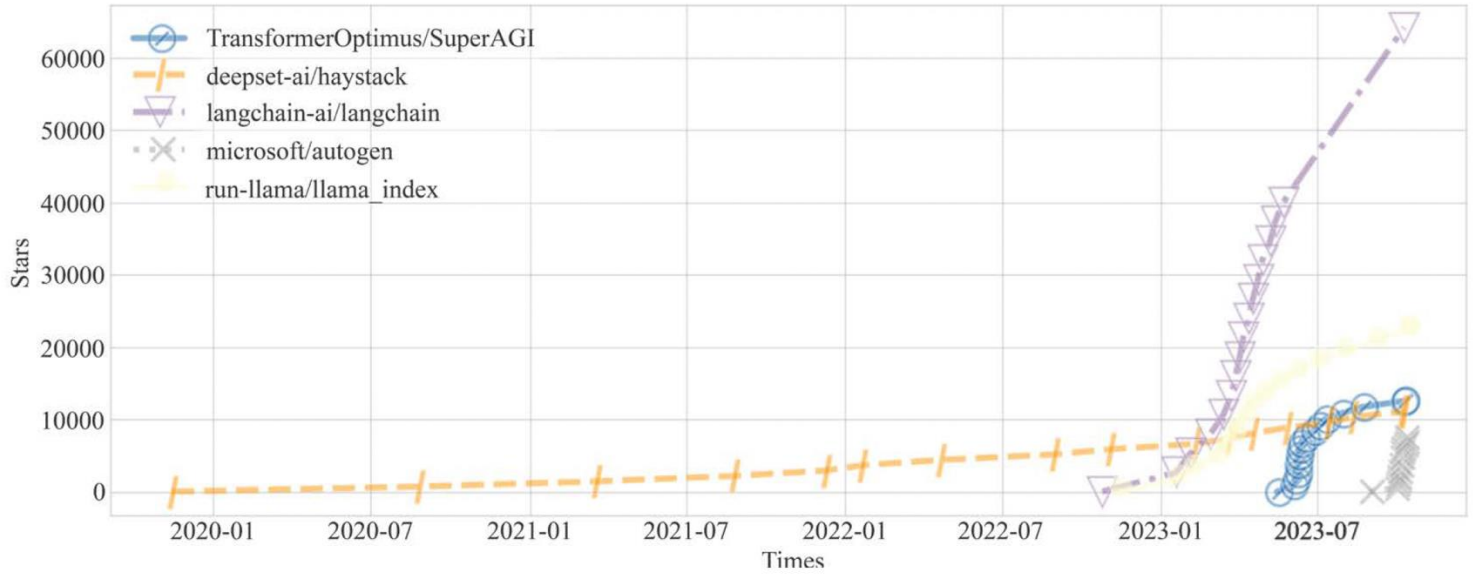


Figure 2.11: Comparison of popularity between different frameworks in Python

amazon?google??chinesiche frameworks?

Kap 3: Langchain Getting Started

Dependencies Setup

Model Integration

Kleine Applikation bauen

Langchain Getting Started

Einrichtung;

z.B.

pip als python management. requirements.txt für dependency persistenz

Venv als virtuelle Umgebung

Jupyter zum Impementieren von Beispielen

Docker für depolyment

Alternativen: conda…

Langchain Getting Started

Liste der unterstützen Integrationen *https://integrations.langchain.com/llms.*

Chatmodels
LLMs
Embedding Models
extrem umfassend

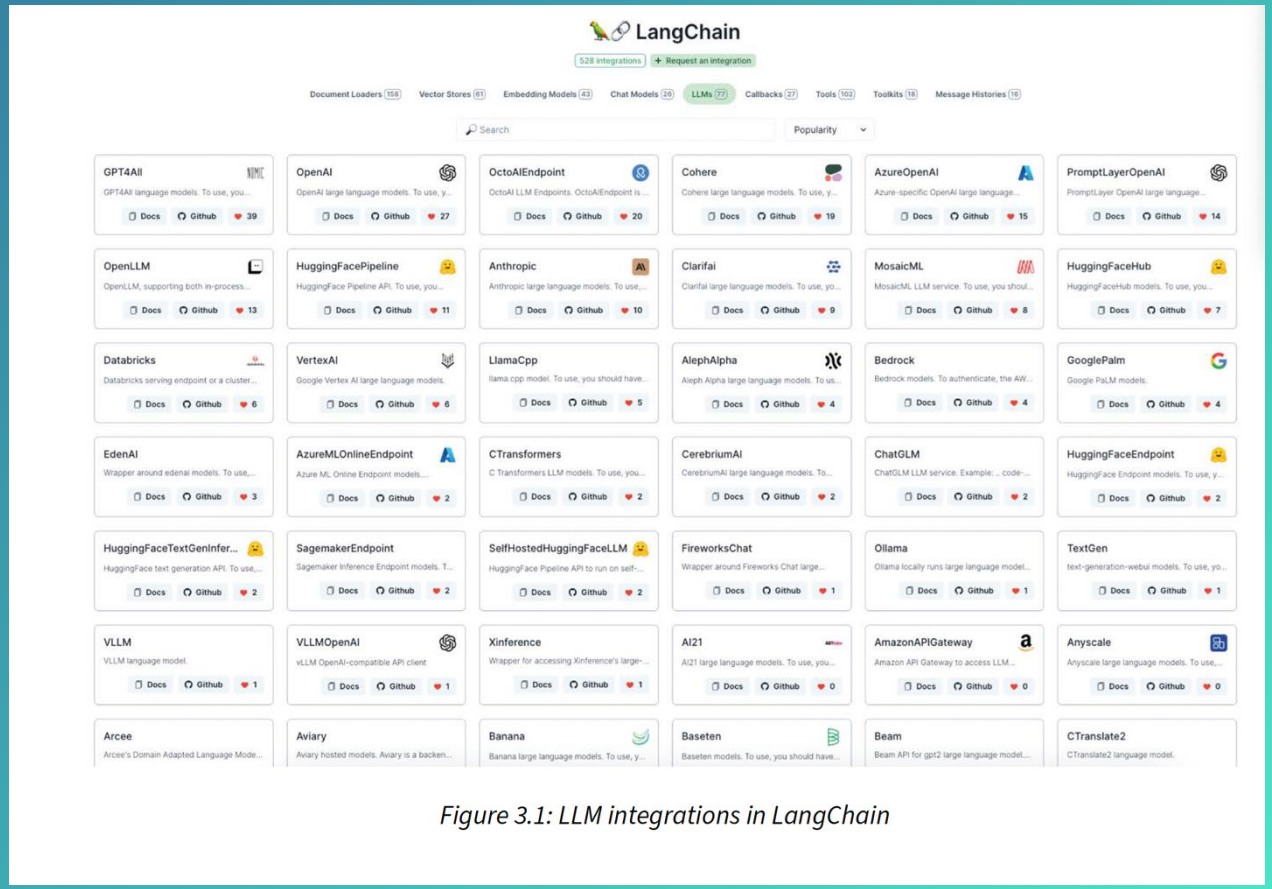


Figure 3.1: LLM integrations in LangChain

Langchain Getting Started

brauchen einen API Key um auf ein Modell zurgreifen zu können. Zb. OpenAI

Oder in die .bash mit EXPORT

```
import os
os.environ["OPENAI_API_KEY"] = "<your token>"
```

oder per programm.

Dann integration:

```
import os

OPENAI_API_KEY = "... "
# I'm omitting all other keys

def set_environment():
    variable_dict = globals().items()
    for key, value in variable_dict:
        if "API" in key or "ID" in key:
            os.environ[key] = value
```

from config import set_environment

set_environment()

FAKE LLM um ohne token zu testen

Langchain Getting Started – Agent Example

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import OpenAI
tools = load_tools(["python_repl"])
llm = OpenAI(temperature=0., model="text-davinci-003")
agent = initialize_agent(
tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=True
)
agent.run("whats 4 + 4")
```

Langchain Getting Started – Selbst ausprobieren mit HuggingFace Key

Hugging Face. Hub für open source gehostete Modelle, Embeddings, Datensets, Evaluationen, Simulationen, Gradio Interface.

```
from langchain.llms import HuggingFaceHub
llm = HuggingFaceHub(
model_kwargs={"temperature": 0.5, "max_length": 64},
repo_id="google/flan-t5-xxl"
)
prompt = "In which country is Tokyo?"
completion = llm(prompt)
print(completion)
```

Langchain Getting Started – selbst ausprobieren

Replicate für Text2Image Integration.

```
from langchain.llms import Replicate
text2image = Replicate(
model="stability-ai/stable-diffusion:db21e45d3f7023abc2a46ee38a23973f6
dce16bb082a930b0c49861f96d1e5bf",
input={"image_dimensions": "512x512"},
)
image_url = text2image("a book cover for a book about creating generative
ai applications in Python")
```

Langchain Getting Started

Andere: Google Vertex Gemini Modelle, Antrophic Llama etc über AWS, Azure Cloud Open AI und LLama

Langchain Getting Started --- Lokaler Modell Setup S.86 selbst ausprobieren

```
from transformers import pipeline
import torch
generate_text = pipeline(
model="aisquared/dlite-v1-355m",
torch_dtype=torch.bfloat16,
trust_remote_code=True,
device_map="auto",
framework="pt"
)
generate_text("In this chapter, we'll discuss first steps with generative
AI in Python.")
```

Langchain Getting Started -- lokaler Modell setup

Das Modell in eine Chain einbringen:

```
from langchain import PromptTemplate, LLMChain
template = """Question: {question}
Answer: Let's think step by step."""
prompt = PromptTemplate(template=template, input_variables=["question"])
llm_chain = LLMChain(prompt=prompt, llm=generate_text)
```

llama.cpp selbst lesen bzw. eine der nächsten Sitzungen.

https://python.langchain.com/docs/integrations/llms/llamacpp/

Langchain Getting Started -- Customer Service Application

Beispiel Features: Incoming customer emails -- sentiment/Lange Texte verdaulich machen -- Intent Klassifikation um Issues der Kunden einzuordnen

Klassifikation mit leichtgewichtigen Modellen lösen. Huggingface benutzen.

Langchain Getting Started -- Customer Service Application

Email Sentiment: brauchen Huggingface Account

```
from transformers import pipeline


customer_email = """
I am writing to pour my heart out about the recent unfortunate experience
I had with one of your coffee machines that arrived broken. I anxiously
unwrapped the box containing my highly anticipated coffee machine.
However, what I discovered within broke not only my spirit but also any
semblance of confidence I had placed in your brand.
Its once elegant exterior was marred by the scars of travel, resembling a
war-torn soldier who had fought valiantly on the fields of some espresso
battlefield. This heartbreaking display of negligence shattered my dreams
of indulging in daily coffee perfection, leaving me emotionally distraught
and inconsolable
"""
```

Langchain Getting Started -- Customer Service Application: email sentiment

```
sentiment_model = pipeline(
task="sentiment-analysis",
model="cardiffnlp/twitter-roberta-base-sentiment"
)
print(sentiment_model(customer_email))
```

[{'label': 'LABEL_0', 'score': 0.5822020173072815}]. Kunde unzufrieden.

=> Ausprobieren mit OpenAI Modell.

Langchain Getting Started -- Customer Service Application -- summarization

```
from langchain import HuggingFaceHub
summarizer = HuggingFaceHub(
repo_id="facebook/bart-large-cnn",
model_kwargs={"temperature":0, "max_length":180}
)
def summarize(llm, text) -> str:
return llm(f"Summarize this: {text}!")
summarize(summarizer, customer_email)
```

Langchain Getting Started -- Customer Service Application: Intent classification

```
from langchain.llms import VertexAI
from langchain import PromptTemplate, LLMChain
```

Ausprobieren mit OpenAI:

Langchain Getting Started -- Customer Service Application: Intent classification

```
template = """Given this text, decide what is the issue the customer is
concerned about. Valid categories are these:
* product issues
* delivery problems
* missing or late orders
* wrong product
* cancellation request
* refund or exchange
* bad support experience
* no clear reason to be upset
Text: {email}
Category:
"""
```

Langchain Getting Started -- Customer Service Application: Intent classification

```
prompt = PromptTemplate(template=template, input_variables=["email"])
llm = VertexAI()
llm_chain = LLMChain(prompt=prompt, llm=llm, verbose=True)
print(llm_chain.run(customer_email))
```

Langchain Getting Started Kap 4 – Building an Assistant

Schlagwörter: Intelligenz, Produktivität, Vertrauen

Im Kapitel:

- **Weniger Halluzination durch Faktencheck**
- **Zusammenfassen von Information**
- **Informationsextraktion aus Dokumenten**
- **Q/A mit Tools**
- **Reasoning Strategien**

Langchain Getting Started – Building an Assistant: Faktencheck

Verifizieren von Claims in einem Output durch Evidenz von externen Quellen

-> Claim Detection

-> Evidenzretrieval

-> Verdict Prediction

https://github.com/Cartus/Automated-Fact-Checking-Resources by Zhijiang Guo):

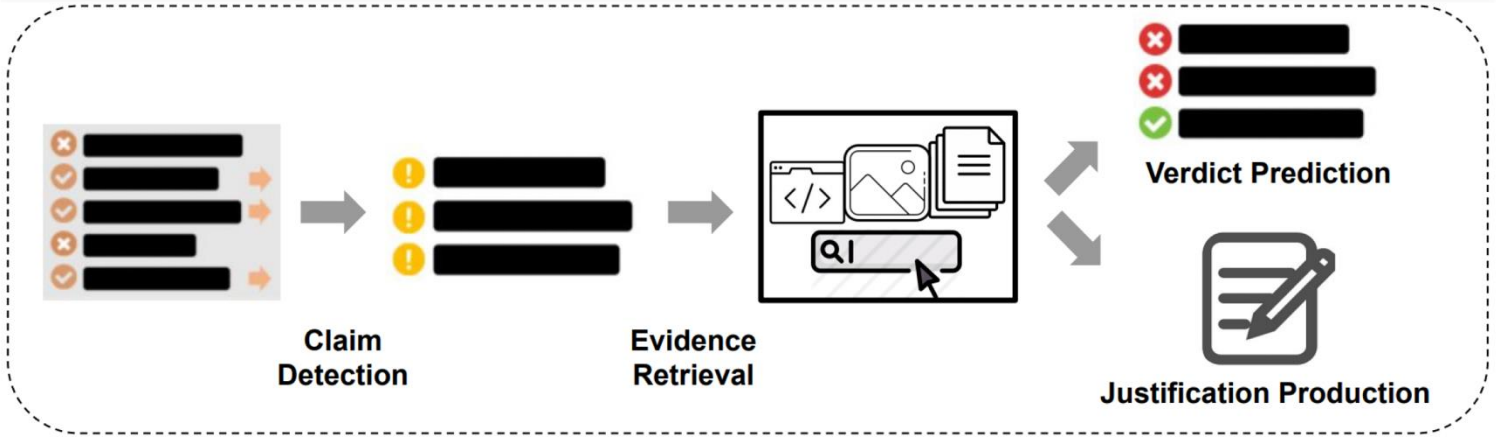


Figure 4.1: Automatic fact-checking pipeline in three stages

Langchain Getting Started – Building an Assistant: Faktencheck

Quellen: LLM selbst – Reflektion, Wikipedia, Sachbücher, Wissensbasen

Create a Statement with high temperature with alternatives. Then Check.

Prompt: Here is a statement {statement}\n

Make a bullet point list of the assumptions you made when producing the above statement.\n

Feedback in the model:

Here is a bullet point list of assertions: {assertions}. For each assertion determine whether it‘s true or false. If it is false explain why\n\n.

Langchain Getting Started – Building an Assistant: Faktencheck

Final judgement:

In the light of above facts how would you answer the question {question}

Langchain Getting Started – Building an Assistant: Faktencheck

LLM CheckerChain. Plug and Play.

```
from langchain.chains import LLMCheckerChain
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.7)
text = "What type of mammal lays the biggest eggs?"
checker_chain = LLMCheckerChain.from_llm(llm, verbose=True)
checker_chain.run(text)
```

Langchain Building an Assistant: Summarizing information

Basis Prompt:

```
from langchain import OpenAI    => geht es wirklich??? Ausprobieren
prompt = """
                summarize this text in one sentence: {text}
            """
llm = OpenAI()
sumary = llm(prompt.format(text = text)
```

Langchain Building an Assistant: Summarizing information

Decorator Syntax als „pythonic“ Interface → langchain decorators Bibliothek.
Translates prompt document into executable code.

```
From langchain_decorators import llm_prompt
@llm_prompt
        def sumarize(text:_ str, length = „short“) -> str
            “““summarize this text in length {length}: {text}
             “““ return
        summary = summarize(text = „let me tell you a boring story from when I was young...")
```

Langchain Building an Assistant: Prompt templates for summarization

Text in vordefinierte Prompts eingeben. Langchain EXPRESSION Language

```
from langchain import PromptTemplate, OpenAI
from langchain.schema import StrOutPutParser
llm = OpenAI()
promt = PromptTemplate.from_template( „sumarize this text: {text}“)
runable = prompt | llm | StrOutputParser()
summary = runable.invoke(“text“: text)
```

LCEL deklarativ Chains erstellen: async, batch, stream fallbeack, parallel,trace

Langchain Building an Assistant: Summarizing information

COD: Chain of density. Inkrementell die Informationsdichte erhöhen. Beispiel mit GPT4. Summaries anfordern mit Längenkontrolle.

template = """Article: { text }

You will generate increasingly concise, entity-dense summaries of the
above article.

Repeat the following 2 steps 5 times.

Step 1. Identify 1-3 informative entities (";" delimited) from the article
which are missing from the previously generated summary.

Step 2. Write a new, denser summary of identical length which covers every
entity and detail from the previous summary plus the missing entities.

A missing entity is:

- relevant to the main story,
- specific yet concise (5 words or fewer),
- novel (not in the previous summary),
- faithful (present in the article),
- anywhere (can be located anywhere in the article).

Langchain Building an Assistant: Summarizing information

Guidelines:
- The first summary should be long (4-5 sentences, ~80 words) yet highly non-specific, containing little information beyond the entities marked as missing. Use overly verbose language and fillers (e.g., "this article discusses") to reach ~80 words.
- Make every word count: rewrite the previous summary to improve flow and make space for additional entities.
- Make space with fusion, compression, and removal of uninformative phrases like "the article discusses".
- The summaries should become highly dense and concise yet self-contained, i.e., easily understood without the article.
- Missing entities can appear anywhere in the new summary.
- Never drop entities from the previous summary. If space cannot be made, add fewer new entities.
Remember, use the exact same number of words for each summary.
Answer in JSON. The JSON should be a list (length 5) of dictionaries whose keys are "Missing_Entities" and "Denser_Summary".
"""

Langchain Building an Assistant: Summarizing information PDF Document

```
from langchain.chains.summarize import load_summarize_chain
from langchain import OpenAI
from langchain.document_loaders import PyPDFLoader
pdf_file_path = "<pdf_file_path>"
pdf_loader = PyPDFLoader(pdf_file_path)
docs = pdf_loader.load_and_split()
llm = OpenAI()
chain = load_summarize_chain(llm, chain_type="map_reduce")
chain.run(docs)
```

Langchain Building an Assistant: Summarizing information

Token usage: Callback Modul e.g. OpenAI – genauer im Buch bzw. Tutorials anschauen.
Verbrauch kann auf token genau durch die API gemessen werden.
Darauch lassen sich Kosten projezieren

Langchain Building an Assistant: Informationsextraktion von Dokumenten

OpenAI API wandelt eine natürlichsprachliche Anfrage in API Calls um. Datenbank Queries zum Extrahieren von strukturierten Daten aus Text.

Schema z.B. in Json Schema beschreiben oder in Pydantic.

```
from typing import Optional
from pydantic import BaseModel
class Experience(BaseModel):
        start_date: Optional[str]
        end_date: Optional[str]
        description: Optional[str]
class Study(Experience):
        degree: Optional[str]
        university: Optional[str]
        country: Optional[str]
        grade: Optional[str]
```

Langchain Getting Started – Building an Assistant: Informationsextraktion

Das Schema wird dann in die Funktion eingebracht.

```
from langchain.chains import create_extraction_chain_pydantic
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import PyPDFLoader
```

Langchain Getting Started – Building an Assistant: Informationsextraktion

```
pdf_file_path = "<pdf_file_path>"
pdf_loader = PyPDFLoader(pdf_file_path)
docs = pdf_loader.load_and_split()
# please note that function calling is not enabled for all models!
llm = ChatOpenAI(model_name="gpt-3.5-turbo-0613")
chain = create_extraction_chain_pydantic(pydantic_schema=Resume, llm=llm)
chain.run(docs)
```

Langchain Getting Started – Building an Assistant: Informationsextraktion

OpenAI injiziert diese Funktionsaufrufe in die Systemnachricht in einer bestimmten Syntax, für die ihre Modelle optimiert wurden.
LangChain hat die Funktionalität, Funktionsaufrufe als Eingabeaufforderungen zu injizieren. Das bedeutet, dass wir Modelle von anderen Anbietern als OpenAI für Funktionsaufrufe innerhalb von LLM-Anwendungen verwenden können.
Instruktiontuning und Funktionsaufrufe ermöglichen es Modellen, aufrufbaren Code zu erzeugen. Dies führt zu Tool-Integrationen, bei denen LLM-Agenten diese Funktionsaufrufe ausführen können, um LLMs mit Live-Daten, Diensten und Laufzeitumgebungen zu verbinden.

Langchain Getting Started – Building an Assistant: QA mit Tools

LLMs können sich selbst nicht mit der externen Welt verbinden (noch nicht?)
Langchain konn über Tools zb wetter, restaurantreservierung, rezepte… einbringen:
data-aware, agentic.

Langchain Getting Started – Building an Assistant: QA mit Tools Beispiel

```
from langchain.agents import (
AgentExecutor, AgentType, initialize_agent, load_tools
)
from langchain.chat_models import ChatOpenAI
def load_agent() -> AgentExecutor:
llm = ChatOpenAI(temperature=0, streaming=True)
tools = load_tools(
tool_names=["ddg-search", "arxiv", "wikipedia"],
llm=llm
)
```

Langchain Getting Started – Building an Assistant: Q/A

```
return initialize_agent(
tools=tools, llm=llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=True
)
```

Langchain Getting Started – Building an Assistant: Interface Streamlit

```
import streamlit as st
from langchain.callbacks import StreamlitCallbackHandler
chain = load_agent()
st_callback = StreamlitCallbackHandler(st.container())
if prompt := st.chat_input():
        st.chat_message("user").write(prompt)
        with st.chat_message("assistant"):
                st_callback = StreamlitCallbackHandler(st.container())
                response = chain.run(prompt, callbacks=[st_callback])
                st.write(response)
```

PYTHONPATH=. streamlit run question_answering/app.py

Langchain Getting Started Kap2,3,4 – Zusammenfassung

- Stochastic Parrots vs zielgerichtete Conversational AI
- Key Komponenten von LangChain
- Wie funktioniert es erste Beispiele
- Vergleich mit anderen Frameworks
- Dependencies Setup
- Model Integration
- Kleine Applikation bauen
- Weniger Halluzination durch Faktencheck
- Zusammenfassen von Information
- Informationsextraktion aus Dokumenten
- Q/A mit Tools
- Reasoning Strategien