**LangChain** is a framework designed to help developers build more powerful, flexible, and context-aware applications using language models by **abstracting and simplifying common tasks**.

It is focused on making it easier to develop, manage, and deploy applications that use LLMs by offering tools and abstractions for **common tasks**.

It's particularly useful for applications that require a combination of language model capabilities, external data interactions, and more complex workflows.

**Here are key features and aspects of LangChain:**

* **Chains**: LangChain provides abstractions for building chains of operations that can process inputs, pass them through various components, and generate outputs. A "chain" could involve multiple steps like **querying an LLM**, interacting with **external APIs**, or applying **custom logic**.
* **Prompts**: LangChain makes it easier to manage and format prompts that are sent to language models. It provides prompt templates, allowing you to structure and manage prompts dynamically, making them more reusable and adaptable.
* **Memory**: LangChain supports the concept of memory, allowing language models to **remember context across multiple interactions**. This is important for building conversational agents that need to track previous interactions over time.
* **Agents**: LangChain includes agents that allow LLMs to make decisions based on context. For example, an agent can decide whether it needs to call an external service, make a query, or simply generate a response based on the available input.
* **Integrations**: LangChain provides integrations with various data sources and APIs, such as databases, web scraping tools, and other external systems, making it easier for LLMs to fetch and utilize dynamic data in their responses.
* **Tooling and Utilities**: LangChain includes utility functions for common tasks like text summarization, question answering, document retrieval, and data parsing, allowing developers to use pre-built functionality rather than reimplementing these tasks.

**Langchain Exercises**

**Install Package**

pip install langchain

pip install langchain-openai

pip install langchain-community

1. **Exercise: Chat Model for Translation**

This exercise demonstrates how to use the ChatOpenAI model to translate a sentence from English to French.

from langchain\_openai import ChatOpenAI

from util import generateToken

from openai import OpenAI

generateToken()

# Initialize the OpenAI model

llm = ChatOpenAI(

    model="gpt-4o-2024-05-13",

    temperature=0,

    max\_tokens=None,

    timeout=None,

    max\_retries=2,

)

# Create a chain that combines the prompt and the model

messages = [

    (

        "system",

        "You are a helpful assistant that translates English to French. Translate the user sentence.",

    ),

    ("user", "I love programming."),

]

response = llm.invoke(messages)

print(response.content);

1. **Using Prompt Templates**

from langchain\_openai import ChatOpenAI

from util import generateToken

from langchain\_core.prompts import ChatPromptTemplate

generateToken()

# Initialize the OpenAI model

llm = ChatOpenAI(

    model="gpt-4o-2024-05-13",

    temperature=0,

    max\_tokens=None,

    timeout=None,

    max\_retries=2,

)

# Create a chain that combines the prompt and the model

prompt = ChatPromptTemplate.from\_messages(

    [

        (

            "system",

            "You are a helpful assistant that translates {input\_language} to {output\_language}.",

        ),

        ("user", "{user\_input}"),

    ]

)

# Invoke the chain with specific input parameters

chain = prompt | llm

parameter\_values = {

    "input\_language": "English",

    "output\_language": "German",

    "user\_input": "I love programming.",

}

response = chain.invoke(parameter\_values, prompt=prompt)

print(response.content)

parameter\_values ["output\_language"] ="Marathi"

response = chain.invoke(parameter\_values, prompt=prompt)

print(response.content)

1. **Exercise to Demo Chaining**

from langchain\_openai import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.schema import StrOutputParser

from langchain.schema.runnable import RunnablePassthrough

from langchain\_openai import ChatOpenAI

from util import generateToken

from openai import OpenAI

generateToken()

client = OpenAI()

topic = "A Lion and a Mouse"

llm = ChatOpenAI(model="gpt-4o-2024-05-13")

#Prompt1

title\_prompt = PromptTemplate.from\_template(

    "Write a great title for a story about {topic}"

)

#Chain1

title\_chain = title\_prompt | llm | StrOutputParser()

#Prompt2

story\_prompt = PromptTemplate.from\_template(

"""You are a writer. Given the title of story, it is your job to write a story for that title.

Title: {title}"""

)

#Chain2

story\_chain = story\_prompt | llm | StrOutputParser()

#Joining Chain1 and Chain2

combined\_chain = {"title": title\_chain} | RunnablePassthrough.assign(story=story\_chain)

#Invoking the chain

result = combined\_chain.invoke({"topic": topic})

print(result['title'])

print("=" \* 100)

print(result['story'])

1. **Exercise: Embedding Documents**

This exercise shows how to embed documents using the OpenAIEmbeddings model.

from langchain\_openai import OpenAIEmbeddings

from langchain.evaluation import load\_evaluator, EmbeddingDistance

embeddings\_model = OpenAIEmbeddings(model="text-embedding-3-large", dimensions=5)

items =   [

        "Hi there!",

        "Oh, hello!",

        "What's your name?",

        "My friends call me World",

        "Hello World!",

    ]

# Getting Embeddings for all items

embeddings = embeddings\_model.**embed\_documents**(items)

for i, embedding in enumerate(embeddings):

    print(f"Embedding {i+1}: {embedding}")

# Calculate cosine similarities using langchain

evaluator = load\_evaluator("embedding\_distance", embeddings=embeddings\_model, distance\_metric=EmbeddingDistance.EUCLIDEAN)

print(evaluator.evaluate\_strings(prediction="I shall go", reference="I shall go"))

**Alternative for distance:**

**pip install** rapidfuzz

Add the following to python code:

evaluator = load\_evaluator("string\_distance")

print(evaluator.evaluate\_strings(prediction="I shall go", reference="I shall go"))

**Reading PDF and Chunking**

**pip install pypdf**

from langchain\_community.document\_loaders import PyPDFLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

import os

from util import generateToken

from openai import OpenAI

generateToken()

client = OpenAI()

def main():

    # 1. Ask for a file path in console

    file\_path = input("Please enter the path to the PDF file: ")

    if not file\_path.endswith(".pdf"):

        print("Only PDF files are allowed")

        return

    if not os.path.exists(file\_path):

        print("File does not exist")

        return

    try:

        loader = PyPDFLoader(file\_path)

        text\_content = loader.load()

        print("PDF Content Read Successfully")

        # 3. Split the PDF into multiple chunks

        text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=3000, chunk\_overlap=100)

        chunks = text\_splitter.split\_documents(text\_content)

        # 4. Loop and display the content in console

        for i, chunk in enumerate(chunks):

            print(f"Chunk {i + 1}:")

            print(chunk.page\_content)

            print("-" \* 80)

    except Exception as e:

        print(f"An error occurred: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Similariy Search with OpenSearch**

from util import generateToken

import os

from dotenv import load\_dotenv

from opensearchpy import OpenSearch

from langchain\_community.vectorstores import OpenSearchVectorSearch

from langchain\_openai import OpenAIEmbeddings

from langchain\_text\_splitters import CharacterTextSplitter

from langchain\_community.document\_loaders import PyPDFLoader

generateToken()

host = os.environ.get('OPENSEARCH\_HOST')

port = os.environ.get('OPENSEARCH\_PORT')

username = os.environ.get('OPENSEARCH\_USERNAME')

password = os.environ.get('OPENSEARCH\_PASSWORD')

# OpenSearch configuration

OPENSEARCH\_CONFIG = {

    "hosts": [{"host": host, "port": port}],

    "http\_auth": (username, password),

    "http\_compress": True,

    "use\_ssl": True,

    "verify\_certs": False,

    "ssl\_assert\_hostname": False,

    "ssl\_show\_warn": False

}

INDEX\_NAME = "files"

# Main function to generate embeddings and insert documents

def main():

    embeddings\_model = OpenAIEmbeddings(model="text-embedding-3-large")

    # 1. Ask for a file path in console

    file\_path = input("Please enter the path to the PDF file: ")

    loader = PyPDFLoader(file\_path)

    text\_content = loader.load()

    print("PDF Content Read Successfully")

    # 3. Split the PDF into multiple chunks

    text\_splitter = CharacterTextSplitter(chunk\_size=3000, chunk\_overlap=100)

    chunks = text\_splitter.split\_documents(text\_content)

    docsearch = **OpenSearchVectorSearch.from\_documents**(

        chunks, embeddings\_model,

        # is\_appx\_search=False,

        bulk\_size=1000,

        opensearch\_url= "https://localhost:9200",

        verify\_certs=False,

        http\_auth=(username, password)

    )

    query = "Challenges of GenAI"

    #performing similarity search

    results = docsearch.similarity\_search(query, k=1)

    print(results[0].page\_content)

    #print(results[1].page\_content)

if \_\_name\_\_ == "\_\_main\_\_":

    main()