Introduction to Tools

In this module, we will have a series of lessons that explore the power of tools in LangChain and demonstrate how they can be combined to tackle different challenges.
In many situations, employing chains and memory proves to be adequate for implementing fairly complex applications. While enhancing the model's responses by incorporating context from buffers and databases is possible, a wide range of APIs is publicly available to provide further assistance. The Tools components can help the models to interact with the outside world. We will introduce a number of integrations like Google Search and content scraper. Furthermore, learn how to create custom tools to use any resources.

Here are the lessons of this module along with a brief description of them:

- **LangChain's Tool Utilization: Examining Diverse Applications through Illustrative Scenarios:**
  In the first lesson, we delve into the **concept of tools** as modular and reusable components designed to accomplish specific tasks. By seamlessly integrating these tools into the system, users gain access to a diverse range of functionalities and information sources. We uncover the unique capabilities of LangChain's tools, setting the stage for exploring their practical applications.

- **Supercharge Your Blog Posts Automatically with LangChain and Google Search:**
  Moving on, the second lesson focuses on the application of artificial intelligence in the field of copyrighting. Language models have revolutionized writing assistance by identifying errors, changing tones, summarizing content, and extending articles. However, there are instances where specialized knowledge is required to provide expert-level suggestions for expanding specific parts of an article. The lesson guides readers through the process of building an application that seamlessly **expands text sections.** The application suggests better content based on the most relevant search results by leveraging an LLM (ChatGPT) to generate search queries and utilizing the Google Search API to capture relevant information.

- **Recreating the Bing Chatbot:**
  While large language models possess impressive capabilities, they also have limitations, as explored in our third lesson. The **hallucination problem**, where models confidently provide incorrect answers, can occur due to the cutoff date of their training process. To overcome this challenge, the lesson proposes leveraging the model's reasoning capability and using the top-matched results from a search engine as context for user queries. LangChain's **integration with the Google Search API and the Newspaper library enables the extraction of relevant stories from search results**, which are then utilized in the prompt for finding the correct answer.

- **Integrating Multiple Tools for Web-Based Question-Answering:**
  The fourth lesson shifts to the practical example of **combining Google Search with the Python-REPL tool.** This combination showcases the power of multiple tools working together to streamline information retrieval projects. The lesson walks readers through the process of finding answers to queries by searching the web and saving the retrieved answers to a text file. By harnessing the potential of these tools, developers can create powerful and efficient solutions for their projects.

- **Building a Custom Document Retrieval Tool with Deep Lake and Langchain: A Step-by-Step Workflow:**
  Finally, the fifth lesson presents a walkthrough of constructing an **efficient document retrieval system designed to extract valuable insights from service FAQs**. The retrieval system aims to provide users with quick and relevant information by promptly fetching pertinent documents that explain a company's operations. The system saves users' time and effort by sifting through multiple sources and FAQs while providing concise and precise answers.

In conclusion, this module highlights the importance of utilizing various tools in LangChain to tackle challenges in information retrieval, copyrighting, and document analysis. By combining the capabilities of different tools, developers, and researchers can create comprehensive and efficient solutions for a wide range of applications. Whether it's expanding text sections, finding answers from search results, or retrieving information from FAQs, the integration of tools empowers users to leverage the full potential of AI-driven systems.

LangChain's Tool Utilization: Examining Diverse Applications through Illustrative Scenarios

Introduction

Tools are modular, reusable components meticulously designed to accomplish specific tasks or provide answers to distinct types of questions. By integrating these tools seamlessly into the system, users can effortlessly tap into a diverse range of functionalities and information sources to tackle challenges and generate meaningful responses. In this lesson, we will explore the various Tools in LangChain, uncovering their unique capabilities.

A few notable examples of tools in LangChain, without getting into technical details, are:

- **Google Search**: This tool uses the Google Search API to fetch relevant information from the web, which can be used to answer queries related to current events, facts, or any topic where a quick search can provide accurate results.
- **Requests**: This tool employs the popular Python library "requests" to interact with web services, access APIs, or obtain data from different online sources. It can be particularly useful for gathering structured data or specific information from a web service.
- **Python REPL**: The Python REPL (Read-Eval-Print Loop) tool allows users to execute Python code on-the-fly to perform calculations, manipulate data, or test algorithms. It serves as an interactive programming environment within the LangChain system.
- **Wikipedia**: The Wikipedia tool leverages the Wikipedia API to search and retrieve relevant articles, summaries, or specific information from the vast repository of knowledge on the Wikipedia platform.
- **Wolfram Alpha**: With this tool, users can tap into the powerful computational knowledge engine of Wolfram Alpha to answer complex questions, perform advanced calculations, or generate visual representations of data.

LangChain Agents and Toolkits

In LangChain, an Agent is a bot that acts using natural language instructions and can use tools to answer its queries. Based on user input, it is also used to determine which actions to take and in what order. An action can either be using a tool (such as a search engine or a calculator) and processing its output or returning a response to the user. Agents are powerful when used correctly, as they can dynamically call chains based on user input.
An agent has access to a suite of tools and can decide which of these tools to call, depending on the user input. Tools are functions that perform specific duties. To create an agent in LangChain, you can use the `initialize_agent` function along with the `load_tools` function to prepare the tools the agent can use.
For example, you can create a simple agent using the following code. It can use the SerpApi service to fetch Google search results or the Python requests wrapper when required.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.agents import load_tools

from langchain.agents import initialize_agent

from langchain.agents import AgentType


tools = load_tools(['serpapi', 'requests_all'], llm=llm,
serpapi_api_key=SERPAPI_API_KEY)

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

agent.run("tell me what is midjourney?")
```

The agent will then use the tool to look for an answer to the query. We'll see some output examples later in this lesson.

Google-Search

LLMs inherently possess knowledge only up to the point at which they were trained, leaving them unaware of any information beyond that timeframe. Integrating search engines as tools within the LangChain system presents a significant advantage. The LangChain library provides a Google Search API wrapper that you can use within your project. You can use this wrapper as a **standalone utility** or as a tool **within an agent**.
First, make sure you have an API Key and Custom Search Engine ID for the Google Search API. If you don't already have a Custom Search Engine ID, the following tutorial is a helpful guide for generating one.

How to get Search Engine ID

How to get Search Engine ID - Elfsight Help Center

Also, getting Google Search API is straightforward. If you have Google Cloud Platform access, you'd just go to the credentials page and click Create credentials > API key.

Setting up API keys - API Console Help

These instructions apply for non Google Cloud Platform (GCP) APIs. If you're building a GCP application, see using API keys for GCP. If your client application does not support.google.com

Set them as environment variables like the following.
Then, you can use `GoogleSearchAPIWrapper` to receive `k` top search results given a query.

```
import os

os.environ["OPENAI_API_KEY"] = "<YOUR-OPENAI-API-KEY>"

os.environ["GOOGLE_API_KEY"] = "<YOUR-GOOGLE-API-KEY>"

os.environ["GOOGLE_CSE_ID"] = "<YOUR-GOOGLE-CSE-ID>"


# As a standalone utility:

from langchain. utilities import GoogleSearchAPIWrapper


search = GoogleSearchAPIWrapper()

search.results("What is the capital of Spain?", 3)Copy
```

The sample code.

```
 [{'title': 'Madrid | History, Population, Climate, & Facts | Britannica',

  'link': 'https://www.britannica.com/place/Madrid',

  'snippet': "May 23, 2023 ... Madrid, city, capital of Spain and of Madrid provincia
(province). Spain's arts and financial center, the city proper, and province form
a\xa0..."},

 {'title': 'Madrid - Eurocities',

  'link': 'https://eurocities.eu/cities/madrid/',

  'snippet': 'As the Spanish capital, Madrid is home to embassies and international
organizations, major companies and financial institutions. It ranks first in Spain
for the\xa0...'},

 {'title': 'Madrid - Wikipedia',

  'link': 'https://en.wikipedia.org/wiki/Madrid',

  'snippet': 'Madrid is the capital and most populous city of Spain. The city has
almost 3.6 million inhabitants and a metropolitan area population of approximately
6.7\xa0...'}]
```

The output.

In the LangChain library, using the available tools requires some necessary steps to be taken. First, you need to initialize an agent, which is the central manager for effectively using these tools. Then, we need to define the language model that we want the agent to use.

```
from langchain.llms import OpenAI

llm = OpenAI(model_name="text-davinci-003", temperature=0)
```

Now, we can initialize an agent and load the `google-search` tool for it to use. The agent will load the search results and provide them to the `llm` to answer our question. The `ZERO_SHOT_REACT_DESCRIPTION` type gives the freedom to choose any of the defined tools to provide context for model based on their description.(can use different agent types, read more)

```
from langchain.agents import initialize_agent, load_tools, AgentType

tools = load_tools(["google-search"])

agent = initialize_agent(

    tools,

    llm,

    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

    verbose=True   )


print( agent("What is the national drink in Spain?") )
```

```
> Entering new AgentExecutor chain...

 I should research this online.

Action: Google Search

Action Input: National drink in Spain

Observation: Sangría is Spain's national drink, so, of course, it belongs on this
list! Spain is famous for its wine, which is the base of this drink. Although there
is nothing officially considered the national alcoholic drink of Spain, many people
would say is the Spanish sherry wine (also known as vino de ... Apr 20, 2021 .......

Thought: I now know the final answer.

Final Answer: The national drink of Spain is Sherry wine.


> Finished chain.

{'input': 'What is the national drink in Spain?',

 'output': 'The national drink of Spain is Sherry wine.'}
```

Requests

The internet is an extensive source of information that Large Language Models cannot access directly. LangChain offers a convenient wrapper built **around the Python Requests module** to facilitate seamless interaction between LLMs and this wealth of information. This wrapper accepts **a URL as an input and efficiently retrieves data from the specified URL**, allowing LLMs to obtain and process web-based content effortlessly.

In this example, we'll set up **fake RESTful** backend using **mockapi.io** To do it, follow these steps:

1. Go to mockapi.io and sign up for a free account.
2. After signing up, log in to your account.
3. Click on "New Project" (the "+" icon) and give your project a name. You don't need to fill in any optional fields.
4. Once the project is created, click on it to view your unique API endpoint.
5. Click on "New Resource" to create a new resource for your API. For example, if you want to create an endpoint for users, you can name the resource "users."
6. Define the schema for your resource. For instance, if you want each user to have an `id`, `name`, and `email`, you can use the following schema:

```
{
    "id": "integer",
    "name": "string",
    "email": "string"
}
```

Click on the "Create" button to create the resource with the defined schema.
This fake backend will have an endpoint to retrieve information about fake users stored in the backend. A dictionary will represent each user. For instance:

```
{
    "id": "1",
    "name": "John Doe",
    "email": "john.doe@example.com"
}
```

Let's use the LangChain tools to interact with our fake RESTful backend. First, import the necessary libraries and initialize the agent with the desired tools. Then, ask the agent to do an HTTP call at "https://644696c1ee791e1e2903b0bb.mockapi.io/user": this is the address of our specific mockapi instance, where you should find 30 users. If you want to try your mockapi instance, then replace the address with "https://<YOUR-MOCKAPI-SERVER-ID>.mockapi.io/user".

```python
from langchain.agents import AgentType

tools = load_tools(["requests_all"], llm=llm)

agent = initialize_agent(

    tools,

    llm,

    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

    verbose=True )


response = agent.run("Get the list of users at
https://644696c1ee791e1e2903b0bb.mockapi.io/user and tell me the total number of
users")
```

To get a request:

```
> Entering new AgentExecutor chain...

I need to get the content from this URL

Action: requests_get

Action Input: https://644696c1ee791e1e2903b0bb.mockapi.io/user

Observation:
```

```
[{"createdAt": "2023-04-24T07:55:47.634Z", "name": "Mr. Kelly Balistreri", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/1244.jpg", "id":
"1"},{"createdAt": "2023-04-24T03:54:44.108Z", "name": "Bradley Cronin", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/615.jpg", "id":
"2"},{"createdAt": "2023-04-24T14:32:29.991Z" , "name": "Jennifer Block Sr.",
"avatar": "https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/105.jpg", "id":
"3"},


[…]


{"createdAt": "2023-04-24T06:10:38.771Z", "name": "Paula Kshlerin", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/1145.jpg", "id":
"28"},{"createdAt": "2023-04-24T03:15:33.343Z", "name": "Roberto Blanda", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/575.jpg", "id":
"29"},{"createdAt": "2023-04-23T18:20:58.632Z", "name":  "Mr. Lisa Erdman", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/1172.jpg", "id":
"30"}]
```

```
Thought:


I now know the list of users

Final Answer: The list of users from the URL
https://644696c1ee791e1e2903b0bb.mockapi.io/user is: [{"createdAt": "2023-04-
24T07:55:47.634Z", "name": "Mr. Kelly Balistreri", "avatar": "https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/1244.jpg", "id":
"1"},{"createdAt": "2023-04-24T03:54:44.108Z", "name": "Bradley Cronin", "avatar":
"https://cloudflare-
ipfs.com/ipfs/Qmd3W5DuhgHirLHGVixi6V76LhCkZUz6pnFt5AJBiyvHye/avatar/615.jpg", "id":
"2"},{"createdAt": "2023-04-24T14:32:29.991Z", "name":"


[…]


Thought:

I now know the final answer

Final Answer: There are 30 users.

> Finished chain.
```

As seen in this example, the **agent, initialized with the Request tool** and the OpenAI language model, processes the given prompt. It identifies the need to fetch data from the provided URL using a GET request, which is facilitated by the tool of the `request`. Upon retrieving the user data, the agent analyzes the number of users and returns the result, completing the task.

Python-REPL

Another tool feature in LangChain is the Python REPL tool, which allows you to **execute Python code generated by the language model.** This can be **useful for complex calculations** where the language model generates code to calculate the answer since LLMs are not good at solving algorithmic and math problems.
Here’s an example of Python-REPL tool usage:

```python
tools = load_tools(["python_repl"], llm=llm)

agent = initialize_agent(

    tools,

    llm,

    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

    verbose=True

  )


print( agent.run("Create a list of random strings containing 4 letters, list should
contain 30 examples, and sort the list alphabetically") )
```

The sample code.

Copy

```
> Entering new AgentExecutor chain...

 I need to generate a list of random strings, sort them, and then print the result

Action: Python REPL

Action Input:

import random

my_list = []

for i in range(30):

    my_list.append(''.join(random.choices(string.ascii_lowercase, k=4)))

my_list.sort()

print(my_list)


Observation: NameError("name 'string' is not defined")

Thought: I need to import the string module

Action: Python REPL

Action Input:


import random

import string

my_list = []

for i in range(30):

    my_list.append(''.join(random.choices(string.ascii_lowercase, k=4)))

my_list.sort()

print(my_list)

Observation: ['aojl', 'biyx', 'bkjq', 'bttr', 'cuef', 'culv', 'czzv', 'djwy', 'eflj',
'ekpr', 'enhg', 'epdq', 'epel', 'hxkp', 'jbrk', 'lbaw', 'mdho', 'nrmc', 'nuqk',
'nybt', 'ptdx', 'smkx', 'sosm', 'srjl', 'swnl', 'uuub', 'vgpw', 'ycli', 'zfln',
'zhsz']


Thought: I now know the final answer

Final Answer: ['aojl', 'biyx', 'bkjq', 'bttr', 'cuef', 'culv', 'czzv', 'djwy',
'eflj', 'ekpr', 'enhg', 'epdq', 'epel', 'hxkp', 'jbrk', 'lbaw', 'mdho', 'nrmc',
'nuqk', 'nybt', 'ptdx', 'smkx', 'sosm', 'srjl', 'swnl', 'uuub', 'vgpw', 'ycli',
'zfln', 'zhsz']
```

```
> Finished chain.
['aojl', 'biyx', 'bkjq', 'bttr', 'cuef', 'culv', 'czzv', 'djwy', 'eflj', 'ekpr',
'enhg', 'epdq', 'epel', 'hxkp', 'jbrk', 'lbaw', 'mdho', 'nrmc', 'nuqk', 'nybt',
'ptdx', 'smkx', 'sosm', 'srjl', 'swnl', 'uuub', 'vgpw', 'ycli', 'zfln', 'zhsz']
```

The output.

Wikipedia

The Wikipedia API tool in LangChain is a powerful tool that allows language models to interact with the Wikipedia API to fetch information and use it to answer questions. Be aware that you need to install the Wikipedia python package using the `pip install Wikipedia` command. The codes are tested using the `1.4.0` version. However, use the latest version of the libraries.

```
agent = initialize_agent(

    tools,

    llm,

    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

    verbose=True

)


tools = load_tools(["wikipedia"])

print( agent.run("What is Nostradamus know for") )
```

The sample code.

Copy

```
> Entering new AgentExecutor chain...

 He is a famous prophet

Action: Python REPL

Action Input: print("Nostradamus is known for his prophecies")

Observation: Nostradamus is known for his prophecies


Thought: I now know the final answer

Final Answer: Nostradamus is known for his prophecies


> Finished chain.

Nostradamus is known for his prophecies
```

The output.

Wolfram-Alpha

In LangChain, you can integrate Wolfram Alpha by using the WolframAlphaAPIWrapper utility. First, you need to set up a Wolfram Alpha developer account and get your APP ID.

Wolfram|Alpha APIs: Computational Knowledge Integration

Easily add top-of-the-line computational knowledge into your applications with Wolfram|Alpha APIs. Options from free to pre-built and custom solutions.

products.wolframalpha.com

Then, install the Wolfram Alpha Python library with `pip install Wolframalpha`. After that, you can set the Wolfram Alpha APP ID as an environment variable in your code:

```
import os
os.environ["WOLFRAM_ALPHA_APPID"] = "your_app_id"
from langchain. utilities.wolfram_alpha import WolframAlphaAPIWrapper
wolfram = WolframAlphaAPIWrapper()
result = wolfram.run("What is 2x+5 = -3x + 7?")
print(result)  # Output: 'x = 2/5'
```

Utilizing the WolframAlphaAPIWrapper from the LangChain library, this code solves the algebraic equation. Let’s observe the WolframAlpha as a tool:

```
tools = load_tools(["wolfram-alpha"])
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True   )

print( agent.run("How many days until the next Solar eclipse") )
```

```
> Entering new AgentExecutor chain... I need to find out when the next Solar eclipse
is
Action: Wolfram Alpha
Action Input: When is the next Solar eclipse
Observation: Assumption: next | solar eclipse | after today
```

```
Answer: Saturday, October 14, 2023 (6 months from now)

Thought: I now know the final answer

Final Answer: 6 months from now, Saturday, October 14, 2023.*> Finished chain.

'6 months from now, Saturday, October 14, 2023.'
```

A Showcase of Resourceful Collaboration

Agents have the capability to employ a variety of tools in order to provide comprehensive and accurate responses to complex inquiries. These agents are better equipped to deliver well-rounded solutions by integrating resources such as Wikipedia and Wolfram-Alpha.
Here’s an example of it:

```
tools = load_tools(["wolfram-alpha", "wikipedia"], llm=llm)

agent = initialize_agent(

                tools,

                llm,

                agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

                verbose=True     )


agent.run("Who is Olivia Wilde's boyfriend? What is his current age raised to the 0.23 power?")
```

```
> Entering new AgentExecutor chain...


I should look up Olivia Wilde's boyfriend on Wikipedia and then use Wolfram Alpha to calculate his age raised to 0.23 power.

Action: Wikipedia

Action Input: Olivia Wilde's boyfriend


Observation:

Page: Olivia Cooke

Summary: Olivia Kate Cooke (born 27 December 1993) is an English actress. In television, she has starred as Emma Decody in the thriller Bates Motel (2013–2017), Becky Sharp in the period drama Vanity Fair (2018), and Alicent Hightower in the fantasy drama House of the Dragon (2022–present).

Thought: I now know Olivia Wilde's boyfriend is Harry Styles.

Action: Wolfram Alpha

Action Input: Harry Styles' age raised to the 0.23 power
```

```
Observation: Wolfram Alpha wasn't able to answer it

Thought: I should try a different approach

Action: Wolfram Alpha

Action Input: Olivia Wilde's age raised to the 0.23 power

Observation: Assumption: (age | of Olivia Wilde (actor) | today )^0.23

Answer: 9.0278 days^(23/100)

Thought: I now know the final answer

Final Answer: Harry Styles' age raised to 0.23 power is 9.0278 days^(23/100).*>
Finished chain.


"Harry Styles' age raised to the 0.23 power is 9.0278 days^(23/100)."
```

Conclusion

LangChain agents successfully bring together various tools, like Wikipedia and Wolfram-Alpha, to create a seamless knowledge integration experience. By combining the strengths of these resources, the agents are able to provide clear, accurate, and detailed answers to more complex questions.
Defining custom tools involves creating new classes, functions, or modules that serve specific purposes within your language processing pipeline. These custom tools can enhance or modify existing language processing capabilities provided by the LangChain library or create entirely new functionalities tailored to your specific needs. More on Defining Custom tools will be covered in lessons to come. Happy Learning!
In the next lesson, we'll build a bot able to refine paragraphs of articles by looking for additional context information online and adding it to that paragraph.

You can find the code of this lesson in this online Notebook.

Supercharge Your Blog Posts Automatically with LangChain and Google Search

Introduction

These days, artificial intelligence is changing the copyrighting field by serving as a writing assistant. These language models can find spelling or grammatical errors, change tones, summarize, or even extend the content. However, there are times when the model may not have the specialized knowledge in a particular field to provide expert-level suggestions for extending parts of an article.
In this lesson, we will take you step by step through the process of building an application that can **effortlessly expand text sections.** The process begins by asking an LLM (ChatGPT) to **generate a few search queries** based on the text at hand. These queries are then will be used **to search the Internet using Google Search API** that, captures relevant information on the

subject. Lastly, the m**ost relevant results will be presented as context to the model to suggest better content.**

We've got three variables here that hold an article's title and content (`text_all`). (From Artificial Intelligence News) Also, the `text_to_change` variable specifies which part of the text we want to expand upon. These constants are mentioned as a reference and will remain unchanged throughout the lesson.

```
title = "OpenAI CEO: AI regulation ‘is essential’"

text_all = """ Altman highlighted the potential benefits of AI technologies like
ChatGPT and Dall-E 2 to help address significant challenges such as climate change
and cancer, but he also stressed the need to mitigate the risks associated with
increasingly powerful AI models. Altman proposed that governments consider
implementing licensing and testing requirements for AI models that surpass a certain
threshold of capabilities. He highlighted OpenAI’s commitment to safety and extensive
testing before releasing any new systems, emphasising the company’s belief that
ensuring the safety of AI is crucial. Senators Josh Hawley and Richard Blumenthal
expressed their recognition of the transformative nature of AI and the need to
understand its implications for elections, jobs, and security. Blumenthal played an
audio introduction using an AI voice cloning software trained on his speeches,
demonstrating the potential of the technology. Blumenthal raised concerns about
various risks associated with AI, including deepfakes, weaponised disinformation,
discrimination, harassment, and impersonation fraud. He also emphasised the potential
displacement of workers in the face of a new industrial revolution driven by AI."""

text_to_change = """ Senators Josh Hawley and Richard Blumenthal expressed their
recognition of the transformative nature of AI and the need to understand its
implications for elections, jobs, and security. Blumenthal played an audio
introduction using an AI voice cloning software trained on his speeches,
demonstrating the potential of the technology."""
```

The following diagram explains the workflow of this project.

First we **generate candidate search queries** from the selected paragraph that we want to expand. The queries are then used **to extract relevant documents using a search engine** (e.g. Bing or Google Search), which are the split into small chunks. We then compute embeddings of these chunks and **save chunks and embeddings** in a Deep Lake dataset. Last, the most similar chunks to the paragraph that we want to expand are retrieved from Deep Lake, and used in a prompt to **expand the paragraph with further knowledge**.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.
Refer to the course introduction if you are looking for the specific versions we used to write the codes in this lesson.
Additionally, install the *newspaper3k* package with version `0.2.8`.

```
!pip install -q newspaper3k==0.2.8 python-dotenv
```

Generate Search Queries

The code below uses OpenAI’s ChatGPT model to **process an article and suggest three relevant search phrases.** We define a prompt that asks the model to suggest Google search queries that could be used to with **finding more information about the subject**. The `LLMChain` ties the `ChatOpenAI` model and `ChatPromptTemplate` together to create the chain to communicate with the model. Lastly, it splits the response by newline and removes the first characters to extract the data. The mentioned format works because we asked the API to generate each query in a new line that starts with `-`. (It is possible to achieve the same effect by using the `OutputParser` class) Prior to running the code provided below, make sure to store your OpenAI key in the `OPENAI_API_KEY` environment variable.

```
from langchain.chat_models import ChatOpenAI

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

from langchain.prompts.chat import (ChatPromptTemplate,HumanMessagePromptTemplate)

template = """ You are an exceptional copywriter and content creator.

You're reading an article with the following title:

----------------

{title}

----------------

You've just read the following piece of text from that article.

----------------

{text_all}

----------------

Inside that text, there's the following TEXT TO CONSIDER that you want to enrich with new details.

----------------

{text_to_change}

----------------

What are some simple and high-level Google queries that you'd do to search for more info to add to that paragraph?

Write 3 queries as a bullet point list, prepending each line with -."""

human_message_prompt = HumanMessagePromptTemplate(

    prompt=PromptTemplate(

        template=template,

        input_variables=["text_to_change", "text_all", "title"], ))

chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])
```

```
# Before executing the following code, make sure to have
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
chat = ChatOpenAI(model_name="text-davinci-003", temperature=0.9)
chain = LLMChain(llm=chat, prompt=chat_prompt_template)

response = chain.run({
    "text_to_change": text_to_change,
    "text_all": text_all,
    "title": title })
queries = [line[2:] for line in response.split("\n")]
print(queries)
```

The sample code.

```
 ['AI technology implications for elections', 'AI technology implications for jobs',
'AI technology implications for security']
```

The output.

The queries you receive from the model might differ from the results above. It is because we set the model's `temperature` argument to `0.9` which makes it highly creative, so it generates more diverse results.

Get Search Results

We must set up the API Key and a custom search engine to be able to use Google search API. To get the key, head to the Google Cloud console and generate the key by pressing the CREATE CREDENTIALS buttons from the top and choosing API KEY. Then, head to the Programmable Search Engine dashboard and remember to select the "Search the entire web" option. The Search engine ID will be visible in the details. You might also need to enable the "Custom Search API" service under the Enable APIs and services. (You will receive the instruction from API if required) We can now configure the environment variables `GOOGLE_CSE_ID` and `GOOGLE_API_KEY`, allowing the Google wrapper to connect with the API.

The next step is to use the **generated queries from the previous section** to get a number of sources from Google searches. The LangChain library provides the `GoogleSearchAPIWrapper` utility that takes care of receiving search results and makes a function to run it `top_n_results`. Then, the `Tool` class will create a wrapper around the said function to make it compatible with agents and help them to interact with the outside world. We only ask for the top 5 results and concatenate the results for each query in the `all_results` variable.

```
from langchain.tools import Tool
from langchain.utilities import GoogleSearchAPIWrapper
```

```
# Remember to set the "GOOGLE_CSE_ID" and "GOOGLE_API_KEY" environment variable.
search = GoogleSearchAPIWrapper()
TOP_N_RESULTS = 5

def top_n_results(query):
    return search.results(query, TOP_N_RESULTS)

tool = Tool(
    name = "Google Search",
    description="Search Google for recent results.",
    func=top_n_results)
all_results = []

for query in queries:
    results = tool.run(query)
    all_results += results
```

The `all_results` variable holds 15 web addresses. (3 queries from ChatGPT x 5 top Google search results) However, it is not optimal flow to use all the contents as a context in our application. There are technical, financial, and contextual considerations to keep in mind.

Firstly, the input length of the LLMs is restricted to a range of 2K to 4K tokens, which varies based on the model we choose. Although we can overcome this limitation by opting for a different chain type, it is more efficient and tends to yield superior outcomes when we adhere to the model's window size.
Secondly, it's important to note that increasing the number of words we provide to the API results in a higher cost. While dividing a prompt into multiple chains is possible, we should be cautious as the cost of these models is determined by the token count.
And lastly, the content that the stored search results will provide is going to be close in context. So, it is a good idea to use the most relevant results.

Find the Most Relevant Results

As mentioned before, Google Search will return the URL for each source. However, we need the content of these pages. The `newspaper` package can extract the contents of a web link using the `.parse()` method. The following code will loop through the results and attempt to extract the content.

```
import newspaper
pages_content = []
```

```
for result in all_results:
        try:
                article = newspaper.Article(result["link"])
                article.download()
                article.parse()

                if len(article.text) > 0:
                        pages_content.append({ "url": result["link"], "text":
article.text })
        except:
                continue
print("Number of pages: ", len(pages_content))
```

The sample code.

Copy

```
Number of pages: 14
```

The output.

The output above shows that 14 pages were processed while we expected 15. There are specific scenarios in which the `newspaper` library may encounter difficulties extracting information. These include search results that lead to a PDF file or websites that restrict access to web scraping. Now, it is crucial to split the saved contents into smaller chunks to ensure the articles do not exceed the model's input length. The code below splits the text by either newline or spaces, depending on the situation. It makes sure that each chunk has 3000 characters with 100 overlaps between the chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document

text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=100)

docs = []
for d in pages_content:
    chunks = text_splitter.split_text(d["text"])
    for chunk in chunks:
        new_doc = Document(page_content=chunk, metadata={ "source": d["url"] })
        docs.append(new_doc)

print("Number of chunks: ", len(docs))
```

The sample code.

```
Number of chunks: 46
```

The output.

As you can see, 46 chunks of data are in the `docs` variable. It is time to find the most relevant chunks to pass them as context to the large language model. The `OpenAIEmbeddings` class will use OpenAI to convert the texts into vector space that holds semantics. We proceeded to embed both document chunks and the desired sentence from the main article that was chosen for expansion. The selected sentence was chosen at the beginning of this lesson and represented by the `text_to_change` variable.

```
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

docs_embeddings = embeddings.embed_documents([doc.page_content for doc in docs])

query_embedding = embeddings.embed_query(text_to_change)
```

Finding the distance between the high-dimensionality embedding vectors is possible using the **cosine similarity metric**. It determines how close two points are within the vector space. Since the embeddings contain contextual information, their proximity indicates a shared meaning. So, the document with a higher similarity score can be used as the source.
We used the `cosine_similarity` function from the `sklearn` library. It calculates the distance between each chunk and the chosen sentence to return the index of the best three results.

```
import numpy as np

from sklearn.metrics.pairwise import cosine_similarity


def get_top_k_indices(list_of_doc_vectors, query_vector, top_k):

    # convert the lists of vectors to numpy arrays

    list_of_doc_vectors = np.array(list_of_doc_vectors)

    query_vector = np.array(query_vector)


    # compute cosine similarities

    similarities = cosine_similarity(query_vector.reshape(1, -1),
list_of_doc_vectors).flatten()


    # sort the vectors based on cosine similarity
    sorted_indices = np.argsort(similarities)[::-1]

    # retrieve the top K indices from the sorted list
    top_k_indices = sorted_indices[:top_k]

    return top_k_indices

top_k = 3
best_indexes = get_top_k_indices(docs_embeddings, query_embedding, top_k)
best_k_documents = [doc for i, doc in enumerate(docs) if i in best_indexes]
```

Extend the Sentence

We can now define the prompt using the additional information from Google search. There are six input variables in the template:

- `title` that holds the main article's title;
- `text_all` to present the whole article we are working on;
- `text_to_change` is the selected part of the article that requires expansion;
- `doc_1`, `doc_2`, `doc_3` to include the close Google search results as context.

The remaining part of the code should be familiar, as it follows the same structure used for generating Google queries. It defines a `HumanMessage` template to be compatible with the ChatGPT API, which is defined with a high-temperature value to encourage creativity. The `LLMChain` class will create a chain that combines the model and prompt to finish up the process by using `.run()` method

```
template = """You are an exceptional copywriter and content creator.

You're reading an article with the following title:

----------------

{title}

----------------

You've just read the following piece of text from that article.

----------------

{text_all}

----------------

Inside that text, there's the following TEXT TO CONSIDER that you want to enrich with new details.

----------------

{text_to_change}

----------------
```

```
Searching around the web, you've found this ADDITIONAL INFORMATION from distinct
articles.

----------------

{doc_1}

----------------

{doc_2}

----------------

{doc_3}

----------------

Modify the previous TEXT TO CONSIDER by enriching it with information from the
previous ADDITIONAL INFORMATION."""

human_message_prompt = HumanMessagePromptTemplate(

    prompt=PromptTemplate(

        template=template,

        input_variables=["text_to_change", "text_all", "title", "doc_1", "doc_2",
"doc_3"], ))

chat_prompt_template = ChatPromptTemplate.from_messages([human_message_prompt])


chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.9)

chain = LLMChain(llm=chat, prompt=chat_prompt_template)


response = chain.run({

    "text_to_change": text_to_change,

    "text_all": text_all,

    "title": title,

    "doc_1": best_k_documents[0].page_content,

    "doc_2": best_k_documents[1].page_content,

    "doc_3": best_k_documents[2].page_content })


print("Text to Change: ", text_to_change)

print("Expanded Variation:", response)
```

The sample code.

```
Text to Change:
```

```
Senators Josh Hawley and Richard Blumenthal expressed their recognition of the
transformative nature of AI and the need to understand its implications

for elections, jobs, and security. Blumenthal played an audio introduction using an
AI voice cloning software trained on his speeches, demonstrating

the potential of the technology.


Expanded Variation:

During a Senate Judiciary Subcommittee on Privacy, Technology, and the Law hearing
titled "Oversight of AI: Rules for Artificial Intelligence,"

Senators Josh Hawley and Richard Blumenthal expressed their recognition of the
transformative nature of AI and its implications for elections, jobs,

and security. Blumenthal even demonstrated the potential of AI voice cloning software
trained on his speeches, highlighting the need for AI

regulations. Recent advances in generative AI tools can create hyper-realistic
images, videos, and audio in seconds, making it easy to spread fake and

digitally created content that could potentially mislead voters and undermine
elections. Legislation has been introduced that would require candidates

to label campaign advertisements created with AI and add a watermark indicating the
fact for synthetic images. Blumenthal raised concerns about

various risks associated with AI, including deepfakes, weaponized disinformation,
discrimination, harassment, and impersonation fraud. The Senate

Judiciary Subcommittee on Privacy, Technology, and the Law has jurisdiction over
legal issues pertaining to technology and social media platforms,

including online privacy and civil rights, as well as the impact of new or emerging
technologies.
```

The output.

Conclusion

In this lesson, we gained insights into leveraging Google search results to enrich the prompt to the model by incorporating additional information. The demonstration showcased the utilization of embedding vectors to identify content that shares a similar meaning or context—also the process of adding relevant information to a prompt to achieve better output. Incorporating external information, such as Google search, is a potent tool for enhancing models by offering supplementary context in situations lacking sufficient data.

In the upcoming lesson, we will employ the same concept to create a chatbot capable of providing accurate answers by utilizing Google search results.

You can find the code of this lesson in this online Notebook.

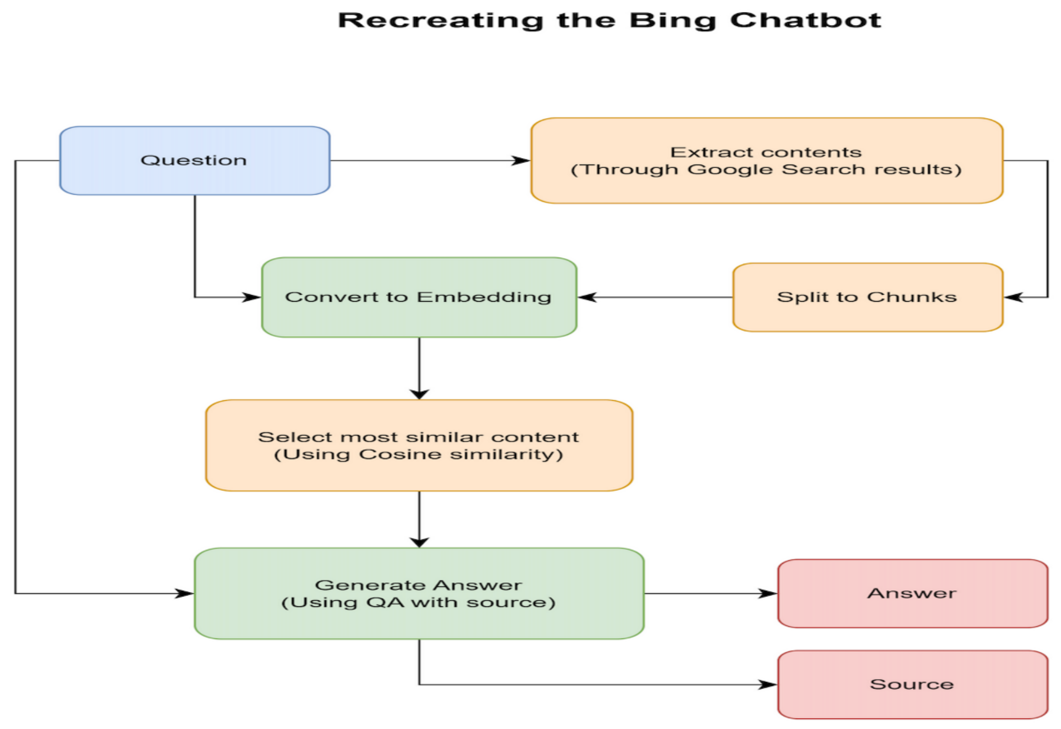
Recreating the Bing Chatbot

Introduction

While the Large Language Models (LLMs) possess impressive capabilities, they have certain limitations that can present challenges when deploying them in a production environment. The **hallucination problem** makes them answer certain questions wrongly with high confidence. This issue can be attributed to various factors, one of which is that their training process has a cut-off date. So, these models do not have access to events preceding that date.

A workaround approach is to present the required information to the model and leverage its reasoning capability to find/extract the answer. Furthermore, it is possible to present the top-matched results a search engine returns as the context for a user's query.

This lesson will explore the idea of **finding the best articles from the Internet** as the context for a chatbot to find the correct answer. We will use LangChain's integration with Google Search API and the Newspaper library to extract the stories from search results. This is followed by choosing and using the most relevant options in the prompt.
Notice that the same pipeline could be done with the **Bing API**, but we'll use the **Google Search API in this project** because it is used in other lessons of this course, thus avoiding creating several keys for the same functionality. Please refer to the following tutorial (or Bing Web Search API for direct access) on obtaining the Bing Subscription Key and using the LangChain Bing search wrapper.
What we are going to do is explained in the following diagram.


Recreating the Bing Chatbot
Question
Extract contents (Through Google Search results)
Convert to Embedding
Split to Chunks
Select most similar content (Using Cosine similarity)
Generate Answer (Using QA with source)
Answer
Source

The user query is used to extract relevant articles using a search engine (e.g. Bing or Google Search), which are then split into chunks. We then compute the embeddings of each chunk, rank them by cosine similarity with respect to the embedding of the query, and put the most relevant chunks into a prompt to generate the final answer, while also keeping track of the sources.

Ask Trending Questions

Let's start this lesson by seeing an example. The following piece must be familiar by now. It uses the OpenAI `GPT-3.5-turbo` model to create an assistant to answer questions. We will ask the model to name the latest Fast & Furious movie, released recently. Therefore, the model couldn't have seen the answer during the training.
Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain import LLMChain, PromptTemplate

from langchain.llms import OpenAI

llm = OpenAI(temperature=0)

template = """You are an assistant that answers the following question correctly and honestly: {question}\n\n"""

prompt_template = PromptTemplate(input_variables=["question"], template=template)

question_chain = LLMChain(llm=llm, prompt=prompt_template)

question_chain.run("what is the latest fast and furious movie?")
```

The sample code.

```
The latest Fast and Furious movie is Fast & Furious 9, which is set to be released in May 2021.
```

The output.

The response shows that the model references the previous movie title as the answer. This is because the new movie (10th sequel) has yet to be released in its fictional universe!
Let's fix the problem.

Google API

Before we start, let's set up the API Key and a custom search engine. If you don't have the keys from the previous lesson, head to the Google Cloud console and generate the key by pressing the CREATE CREDENTIALS buttons from the top and choosing API KEY. Then, head to the Programmable Search Engine dashboard and remember to select the "Search the entire web" option. The Search engine ID will be visible in the details. You might also need to enable the "Custom Search API" service under the Enable APIs and services. (You will receive the instruction from API if required) Now we can set the environment variables for both Google and OpenAI APIs.

```
import os


os.environ["GOOGLE_CSE_ID"] = "<Custom_Search_Engine_ID>"

os.environ["GOOGLE_API_KEY"] = "<Google_API_Key>"

os.environ["OPENAI_API_KEY"] = "<OpenAI_Key>"
```

Get Search Results

This section uses LangChain's `GoogleSearchAPIWrapper` class to receive search results. It works in combination with the `Tool` class that presents the utilities for agents to help them interact with the outside world. In this case, creating a tool out of any function, like `top_n_results` is possible. The API will return the page's title, URL, and a short description.

```
from langchain.tools import Tool

from langchain.utilities import GoogleSearchAPIWrapper


search = GoogleSearchAPIWrapper()

TOP_N_RESULTS = 10


def top_n_results(query):

    return search.results(query, TOP_N_RESULTS)


tool = Tool(

    name = "Google Search",

    description="Search Google for recent results.",

    func=top_n_results

)


query = "What is the latest fast and furious movie?"


results = tool.run(query)


for result in results:

    print(result["title"])

    print(result["link"])

    print(result["snippet"])

    print("-"*50)
```

The sample code.

```
Fast & Furious movies in order | chronological and release order ...

https://www.radiotimes.com/movies/fast-and-furious-order/

Mar 22, 2023 ... Fast & Furious Presents: Hobbs & Shaw (2019); F9 (2021); Fast and Furious 10 (2023).
Tokyo Drift also marks the first appearance of Han Lue, a ...

--------------------------------------------------
```

FAST X | Official Trailer 2 - YouTube

https://www.youtube.com/watch?v=aOb15GVFZxU

Apr 19, 2023 ... Fast X, the tenth film in the Fast & Furious Saga, launches the final ... witnessed it all and has spent the last 12 years masterminding a ...

Fast & Furious 10: Release date, cast, plot and latest news on Fast X

https://www.radiotimes.com/movies/fast-and-furious-10-release-date/

Apr 17, 2023 ... Fast X is out in cinemas on 19th May 2023 – find out how to rewatch all the Fast & Furious movies in order, and read our Fast & Furious 9 review ...

Fast & Furious - Wikipedia

https://en.wikipedia.org/wiki/Fast_%26_Furious

The main films are known as The Fast Saga. Universal expanded the series to include the spin-off film Fast & Furious Presents: Hobbs & Shaw (2019), ...

How many 'Fast & Furious' movies are there? Here's the list in order.

https://www.usatoday.com/story/entertainment/movies/2022/07/29/fast-and-furious-movies-order-of-release/10062943002/

Jul 29, 2022 ... There are currently nine films in the main "Fast and Furious" franchise, with the 10th, "Fast X," set to release on May 19, 2023. There are ...

How to Watch Fast and Furious Movies in Chronological Order - IGN

https://www.ign.com/articles/fast-and-furious-movies-in-order

Apr 6, 2023 ... Looking to go on a Fast and Furious binge before the next movie comes out? ... This is the last Fast film with Paul Walker's Brian O'Conner, ...

'Fast and Furious 10': Everything We Know So Far

https://www.usmagazine.com/entertainment/pictures/fast-and-furious-10-everything-we-know-so-far/

7 days ago ... Fast X will be the second-to-last film in the blockbuster franchise, and Dominic Toretto's next adventure is set to be one of the biggest so far ...

Latest 'Fast & Furious' Movie Leads Weekend Box Office - WSJ

https://www.wsj.com/articles/latest-fast-furious-movie-leads-weekend-box-office-11624815451

Jun 27, 2021 ... “F9,” however, offers the clearest test yet on the post-pandemic habits of moviegoers. The movie is the most-anticipated title released since ...

Fast & Furious Movies In Order: How to Watch Fast Saga ...

https://editorial.rottentomatoes.com/guide/fast-furious-movies-in-order/

After that, hop to franchise best Furious 7. Follow it up with The Fate of the Furious and spin-off Hobbs & Shaw and then the latest: F9 and Fast X. See below ...

The looming release of the latest "Fast and Furious" movie heightens ...

```
https://www.cbsnews.com/losangeles/news/looming-release-of-latest-fast-and-furious-movie-heightens-
concerns-over-street-racing-takeovers/

17 hours ago ... With the latest installment of the "Fast and Furious" franchise set to hit theaters this
weekend, local law enforcement is banding together ...

--------------------------------------------------
```

The output.

Now, we use the `results` variable's `link` key to download and parse the contents. The newspaper library takes care of everything. However, it might be unable to capture some contents in certain situations, like anti-bot mechanisms or having a file as a result.

```
import newspaper

pages_content = []

for result in results:

        try:

                article = newspaper.Article(result["link"])

                article.download()

                article.parse()

                if len(article.text) > 0:

                        pages_content.append({ "url": result["link"], "text":
article.text })

        except:

                continue
```

Process the Search Results

We now have the top 10 results from the Google search. (Honestly, who looks at Google's second page?) However, it is not efficient to pass all the contents to the model because of the following reasons:

- The model's context length is limited.
- It will significantly increase the cost if we process all the search results.
- In almost all cases, they share similar pieces of information.

So, let's find the **most relevant results,**

Incorporating the LLMs embedding generation capability will enable us to find contextually similar content. It means converting the text to a high-dimensionality tensor that captures meaning. The **cosine similarity function** can find the **closest article with respect to the user's question**.

It starts by splitting the texts using the `RecursiveCharacterTextSplitter` class to ensure the content lengths are inside the model's input length. The `Document` class will create a data structure from each chunk that enables saving metadata like `URL` as the source. The model can later use this data to know the content's location.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.docstore.document import Document


text_splitter = RecursiveCharacterTextSplitter(chunk_size=4000, chunk_overlap=100)
```

```
docs = []

for d in pages_content:

        chunks = text_splitter.split_text(d["text"])

        for chunk in chunks:

            new_doc = Document(page_content=chunk, metadata={ "source": d["url"] })

            docs.append(new_doc)
```

The subsequent step involves utilizing the OpenAI API's `OpenAIEmbeddings` class, specifically the `.embed_documents()` method for search results and the `.embed_query()` method for the user's question, to generate embeddings.

```
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

docs_embeddings = embeddings.embed_documents([doc.page_content for doc in docs])

query_embedding = embeddings.embed_query(query)
```

Lastly, the `get_top_k_indices` function accepts the content and query embedding vectors and returns the index of top K candidates with the highest cosine similarities to the user's request. Later, we use the indexes to retrieve the best-fit documents.

```
import numpy as np

from sklearn.metrics.pairwise import cosine_similarity

def get_top_k_indices(list_of_doc_vectors, query_vector, top_k):

  # convert the lists of vectors to numpy arrays

  list_of_doc_vectors = np.array(list_of_doc_vectors)

  query_vector = np.array(query_vector)

  # compute cosine similarities

  similarities = cosine_similarity(query_vector.reshape(1, -1),
list_of_doc_vectors).flatten()

  # sort the vectors based on cosine similarity

  sorted_indices = np.argsort(similarities)[::-1]

  # retrieve the top K indices from the sorted list

  top_k_indices = sorted_indices[:top_k]


  return top_k_indices

top_k = 2

best_indexes = get_top_k_indices(docs_embeddings, query_embedding, top_k)

best_k_documents = [doc for i, doc in enumerate(docs) if i in best_indexes]
```

Chain with Source

Finally, we used the selected articles in our prompt (using the `stuff` method) to assist the model in finding the correct answer. LangChain provides the `load_qa_with_sources_chain()` chain, which is designed to accept a list of `input_documents` as a source of information and a `question` argument which is the user's question. The final part involves preprocessing the model's response to extract its answer and the sources it utilized.

```
from langchain.chains.qa_with_sources import load_qa_with_sources_chain

from langchain.llms import OpenAI

chain = load_qa_with_sources_chain(OpenAI(temperature=0), chain_type="stuff")

response = chain({"input_documents": best_k_documents, "question": query},
return_only_outputs=True)


response_text, response_sources = response["output_text"].split("SOURCES:")

response_text = response_text.strip()

response_sources = response_sources.strip()

print(f"Answer: {response_text}")

print(f"Sources: {response_sources}")
```

The sample code.

```
Answer: The latest Fast and Furious movie is Fast X, scheduled for release on May 19,
2023.

Sources: https://www.radiotimes.com/movies/fast-and-furious-10-release-date/,
https://en.wikipedia.org/wiki/Fast_%26_Furious
```

The output.

The use of search results helped the model find the correct answer, even though it never saw it before during the training stage. The question and answering chain with source also provides information regarding the sources utilized by the model to derive the answer.

Conclusion

In this lesson, we learned how to utilize external knowledge from a search engine to make a robust application. The context can be presented from various sources such as PDFs, text documents, CSV files, or even the Internet! We used Google search results as the source of information, and it enabled the model to respond to the question it previously couldn't answer correctly.
In the next lesson, we'll see how to build a bot that leverages multiple tools to answer questions.

You can find the code of this lesson in this online Notebook.

Integrating Multiple Tools for Web-Based Question-Answering

Introduction

As developers and information enthusiasts, we often find ourselves needing to utilize various tools and libraries to fetch and process data. By leveraging multiple tools simultaneously, we can create powerful, efficient, and comprehensive solutions for the systems we build with LangChain. This lesson will demonstrate a practical example of combining the power of Google Search with the versatile Python-REPL tool for an effective result. You will learn how to harness the potential of multiple tools working together to streamline your own information retrieval projects.

Let's be more specific about what exactly we want to accomplish:

1. Find the answer to a query by searching the web: The agent should use its tools and language model to identify the most relevant sources for it.
2. Save the answer to a file: After retrieving the answer, the agent is expected to save it to a text file.

Setting Up Libraries

First, we set the necessary API keys as environment variables.

```
import os


os.environ["OPENAI_API_KEY"] = "<YOUR-OPENAI-API-KEY>"

os.environ["GOOGLE_API_KEY"] = "<YOUR-GOOGLE-SEARCH-API-KEY>"

os.environ["GOOGLE_CSE_ID"] = "<YOUR-CUSTOM-SEARCH-ENGINE-ID>"Copy
```

You can sign up for these keys by following these instructions:

Setting up API keys - API Console Help

These instructions apply for non Google Cloud Platform (GCP) APIs. If you're building a GCP application, see using API keys for GCP. If your client application does not support.google.com

programmablesearchengine.google.com

Next thing, we want to import the libraries we aim to use for our project.

Remember to install the required packages with the following command:

`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.llms import OpenAI

from langchain.agents import Tool, initialize_agent, AgentType


from langchain.utilities import GoogleSearchAPIWrapper, PythonREPLCopy
```

We're going to declare some wrappers. The `GoogleSearchAPIWrapper` wrapper allows us to easily create a tool for using the Google Search APIs, whereas the `PythonREPL` wrapper allows the model to execute generated Python code.

```
search = GoogleSearchAPIWrapper()

python_repl = PythonREPL()
```

The next code block creates an instance of the OpenAI language model with a temperature parameter set to 0. This parameter influences the randomness of the model's output, and by setting it to 0, the generated responses will be more deterministic and focused.

```
llm = OpenAI(model="text-davinci-003", temperature=0)Copy
```

Here we have our toolkit set assembled of:

1. The `google-search` tool is a convenient way to perform Google searches when an agent needs information about current events. The tool makes use of Google's API to provide relevant search results.
2. The `python_repl` tool: This tool wraps a Python shell, allowing the execution of Python commands directly.

```
toolkit = [

    Tool(

        name="google-search",

        func=search.run,

        description="useful for when you need to search Google to answer questions
about current events"

    ),

    Tool(

        name="python_repl",

        description="A Python shell. Use this to execute Python commands. Input
should be a valid Python command. Useful for saving strings to files.",

        func=python_repl.run

    ) ]
```

These tools are then added to the `toolkit` list, which is used to initialize an agent with the specified tools. The agent can then perform various tasks using the tools in its toolkit. The agent can be easily extended by adding more tools to the toolkit, allowing it to handle a wide range of tasks and situations. Let's instantiate the agent.

```
agent = initialize_agent(

        toolkit,

        llm,

        agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,

        verbose=True )
```

The parameter `agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION` specifies the agent's strategy, which means that the agent will attempt to perform tasks without any prior examples, relying solely on its understanding of the problem description and the available tools (and their descriptions). Now let’s run the experiment! We should be able to ask the Agent directly by giving him instructions on what we want:

```
agent.run("Find the birth date of Napoleon Bonaparte and save it to a file 'final.txt'.")
```

You should see an output like the following.

```
> Entering new AgentExecutor chain...

 I need to find the date of Napoleon's birth and save it to a file.

Action: google-search

Action Input: "Napoleon Bonaparte birth date"

Observation: Napoleon Bonaparte later known by his regnal name Napoleon I, was a Corsican-born French
military commander and political leader who rose to prominence ... Nov 9, 2009 ... Napoleon Bonaparte was
born on August 15, 1769, in Ajaccio, on the Mediterranean island of Corsica. He was the second of eight
surviving ... Napoleone Buonaparte was born in Ajaccio, Corsica, on 15 August 1769. He was the second of
eight children born to Carlo Buonaparte, a lawyer descended from ... May 1, 2023 ... Napoleon I, French in
full Napoléon Bonaparte, original Italian Napoleone Buonaparte, byname the Corsican or the Little
Corporal, ... Napoleon Bonaparte was born on August 15, 1769. He was most notably known as a French
military and political leader, who became prominent during the French ... Furthermore, Charles Bonaparte's
“journal” notes the birth on 15 August 1769. It is true that Napoleon very occasionally used the papers of
his brother ... Jun 23, 2002 ... Napoleon Bonaparte was born at Ajaccio, Corsica, ... importance of
Napoleon the individual argued about his origins and his date of birth. In 1764, Charles-Marie Bonaparte
wed the young Letizia Ramolino and settled with her in the Bonaparte family residence. Napoleon was born
there on August 15 ... Napoleon Bonaparte was born the 15th of August, 1769 on Corsica, just three months
after the island had been defeated by the French. Napoleon was born on the 15th of August, 1769, in French
occupied Corsica. His father was Carlo Maria di Buonaparte, and his mother, Maria Letizia Ramolino.

Thought: I have the date of Napoleon's birth.

Action: python_repl

Action Input:


with open('answer.txt', 'w') as f:

    f.write('Napoleon Bonaparte was born on August 15, 1769')


Observation:

Thought: I have saved the answer to the file.

Final Answer: Napoleon Bonaparte was born on August 15, 1769.

> Finished chain.
```

As you can see from the printed output, the agent first used the `google-search` tool with the query `"Napoleon Bonaparte birth date"`. Upon seeing its result, the agent then wrote the following Python program to save the answer to the `answer.txt` local file:

```
with open('answer.txt', 'w') as f:

    f.write('Napoleon Bonaparte was born on August 15, 1769')
```

You should now have a local file `answer.txt` containing a text similar to `Napoleon Bonaparte, born on August 15, 1769`.
Let’s also find the death date of Napoleon and append it to the `answer.txt` file.

```
query = "Find when Napoleon Bonaparte died and append this information " \
    "to the content of the 'answer.txt' file in a new line."
agent.run(query)
```

You should see something similar to the following printed output.

```
> Entering new AgentExecutor chain...
 I need to find the date of Napoleon's death and then write it to a file.
Action: google-search
Action Input: "When did Napoleon Bonaparte die?"
Observation: Napoleon Bonaparte later known by his regnal name Napoleon I, was a Corsican-born French
military commander and political leader who rose to prominence ... Aug 15, 2022 ... Napoleon was only 51
when he died on the island of St. Helena, where he was out of power and exiled from his beloved France. By
May 5, 1821, ... Nov 9, 2009 ... In October 1815, Napoleon was exiled to the remote, British-held island
of Saint Helena, in the South Atlantic Ocean. He died there on May 5, ... Apr 25, 2014 ... Napoleon
Bonaparte died at 5.49pm on 5 May 1821, at Longwood on the island of Saint Helena. An autopsy was carried
out on 6 May; ... Jan 21, 2014 ... Was Napoleon poisoned? ... weeks before his demise at age 51, “I die
before my time, murdered by the English oligarchy and its assassin. Jan 17, 2007 ... 17, 2007&#151; --
Napoleon Bonaparte died in exile in 1821. But his story never does. His personal physician reported on his
death ... May 22, 2023 ... He was the third son of Napoleon I's brother Louis Bonaparte, who was king of
Holland from 1806 to 1810, ... How did Napoleon III die? Jan 20, 2003 ... Napoleon was not poisoned, they
said. He died of stomach cancer. At a news conference in Paris, Jacques di Costanzo, ... May 1, 2023 ...
Napoleon I, French in full Napoléon Bonaparte, original Italian ... It was during Napoleon's year in Paris
that his father died of a stomach ... In 1785, when Napoleon was not yet sixteen, his father died of
stomach cancer ... Napoleon Bonaparte has continued to inspire passion and interest Read more.
Thought: I now know the date of Napoleon's death.
Action: python_repl
Action Input:


with open('answer.txt', 'a') as f:
    f.write('\nNapoleon Bonaparte died on May 5, 1821.')


Observation:
Thought: I now know the final answer.
Final Answer: Napoleon Bonaparte died on May 5, 1821.


> Finished chain.
```

Your final `answer.txt` should look like the following:

```
Napoleon Bonaparte was born on August 15, 1769
Napoleon Bonaparte died on May 5, 1821.
```

This process demonstrates the agent's ability to integrate multiple tools for a single task seamlessly.

Conclusion

In conclusion, we've illustrated how a LangChain agent can effectively employ multiple tools and techniques to accomplish a task, such as doing question-answering on the web and storing the answers in a file. This example highlights the potential of LangChain agents to provide valuable assistance across diverse scenarios.

In the upcoming lesson, we'll see how to create an ad-hoc tool for retrieving relevant documents from a Deep Lake vector store.

Building a Custom Document Retrieval Tool with Deep Lake and LangChain: A Step-by-Step Workflow

Introduction

This lesson is a walkthrough on constructing an efficient document retrieval system designed to extract valuable insights from the FAQs of a service. The goal of this system is to swiftly provide users with relevant information by promptly fetching pertinent documents that explain a company's operations.

Sifting through multiple sources or FAQs can be a tiresome task for users. Our retrieval system steps in here, providing concise, precise, and quick answers to these questions, thereby saving users time and effort.

Workflow

1. **Setting up Deep Lake**: Deep Lake is a type of vector store database designed for storing and querying high-dimensional vectors efficiently. In our case, we're using Deep Lake to store document embeddings and their corresponding text.
2. **Storing documents in Deep Lake**: Once Deep Lake is set up, we'll create embeddings for our documents. In this workflow, we're using OpenAI's model for creating these embeddings. Each document's text is fed into the model, and the output is a high-dimensional vector representing the text's semantic content. The embeddings and their corresponding documents are then stored in Deep Lake. This will set up our vector database, which is ready to be queried.
3. **Creating the retrieval tool**: Now, we use Langchain to create a custom tool that will interact with Deep Lake. This tool is essentially a function that takes a query as input and returns the most similar documents from Deep Lake as output. To find the most similar documents, the tool first computes the embedding of the query using the same model we used for the documents. Then, it queries Deep Lake with this query embedding, and Deep Lake returns the documents whose embeddings are most similar to the query embedding.
4. **Using the tool with an agent**: Finally, we use this custom tool with an agent from Langchain. When the agent receives a question, it uses the tool to retrieve relevant documents from Deep Lake, and then it uses its language model to generate a response based on these documents.

Let's start! First, we load the OpenAI API key as an environment variable:

```
import os

os.environ["OPENAI_API_KEY"] = "<INSERT_YOUR_OPENAI_API_KEY>"

os.environ["ACTIVELOOP_TOKEN"] = "<INSERT_YOUR_ACTIVELOOP_TOKEN>"Copy
```

Setting up Deep Lake

Next, we'll set up a Deep Lake vector database and add some documents to it. The hub path for a Deep Lake dataset is in the format `hub://<org_id>/<dataset_name>`. Remember to install the required packages with the following command: `pip install langchain==0.0.208 deeplake openai tiktoken`.

```
# We'll use an embedding model to compute the embeddings of our documents

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import DeepLake


# instantiate embedding model

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")


# create Deep Lake dataset

# We'll store the documents and their embeddings in the deep lake vector db

# TODO: use your organization id here. (by default, org id is your username)

my_activeloop_org_id = "<YOUR-ACTIVELOOP-ORG-ID>"

my_activeloop_dataset_name = "langchain_course_custom_tool"

dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)
```

You should now be able to visualize your dataset on the Activeloop website.

Storing documents in Deep Lake

We can then add some FAQs related to PayPal as our knowledge base.

```
# add faqs to the dataset

faqs = [

    "What is PayPal?\nPayPal is a digital wallet that follows you wherever you go. Pay any way you want. Link your credit cards to your PayPal Digital wallet, and when you want to pay, simply log in with your username and password and pick which one you want to use.",

    "Why should I use PayPal?\nIt's Fast! We will help you pay in just a few clicks. Enter your email address and password, and you're pretty much done! It's Simple! There's no need to run around searching for your wallet. Better yet, you don't need to type in your financial details again and again when making a purchase online. We make it simple for you to pay with just your email address and password.",

    "Is it secure?\nPayPal is the safer way to pay because we keep your financial information private. It isn't shared with anyone else when you shop, so you don't
have to worry about paying businesses and people you don't know. On top of that, we've got your back. If your eligible purchase doesn't arrive or doesn't match its description, we will refund you the full purchase price plus shipping costs with PayPal's Buyer Protection program.",

    "Where can I use PayPal?\nThere are millions of places you can use PayPal worldwide. In addition to online stores, there are many charities that use PayPal to raise money. Find a list of charities you can donate to here. Additionally, you can send funds internationally to anyone almost anywhere in the world with PayPal. All you need is their email address. Sending payments abroad has never been easier.",

    "Do I need a balance in my account to use it?\nYou do not need to have any balance in your account to use PayPal. Similar to a physical wallet, when you are making a purchase, you can choose to pay for your items with any of the credit cards that are attached to your account. There is no need to pre-fund your account."

]

db.add_texts(faqs)


# Get the retriever object from the deep lake db object

retriever = db.as_retriever()
```

Creating the retrieval tool

Now, we'll construct the custom tool function that will retrieve the relevant documents from the Deep Lake database:

```
from langchain.agents import tool


# We define some variables that will be used inside our custom tool

# We're creating a custom tool that looks for relevant documents in our deep lake db

CUSTOM_TOOL_N_DOCS = 3 # number of retrieved docs from deep lake to consider

CUSTOM_TOOL_DOCS_SEPARATOR ="\n\n" # how to join together the retrieved docs to form a single string


# We use the tool decorator to wrap a function that will become our custom tool

# Note that the tool has a single string as input and returns a single string as output

# The name of the function will be the name of our custom tool

# The docstring of the function will be the description of our custom tool

# The description is used by the agent to decide whether to use the tool for a specific query

@tool

def retrieve_n_docs_tool(query: str) -> str:

    """ Searches for relevant documents that may contain the answer to the query."""
    docs = retriever.get_relevant_documents(query)[:CUSTOM_TOOL_N_DOCS]

    texts = [doc.page_content for doc in docs]

    texts_merged = CUSTOM_TOOL_DOCS_SEPARATOR.join(texts)

    return texts_mergedCopy
```

Our function, `retrieve_n_docs_tool`, is designed with a specific purpose in mind - to search for and retrieve relevant documents based on a given query. It accepts a single string input, which is the user's question or query, and it's designed to return a single string output.

To find the relevant documents, our function makes use of the retriever object's `get_relevant_documents` method. Given the query, this method searches for and returns a list of the most relevant documents. But we only need some of the documents it finds. We only want the top few. That's where the `[: CUSTOM_TOOL_N_DOCS]` slice comes in. It allows us to select the top `CUSTOM_TOOL_N_DOCS` number of documents from the list, where CUSTOM_TOOL_N_DOCS is a predefined constant that tells us how many documents to consider. In this case, that's 3 documents, as we specified.

Now that we have our top documents, we want to extract the text from each of them. We achieve this using a list comprehension that iterates over each document in our list, `docs`, and extracts the `page_content` or text from each document. The result is a list of the top 3 relevant document texts.

Next, we want to join these individual texts from a list into a single string using `.join(texts)` method. Finally, our function returns `texts_merged`, a single string that comprises the joined texts from the relevant documents. The `@tool` decorator wraps the function, turning it into a custom tool.

Using the tool with an agent

We can now initialize the agent that uses our custom tool.

```
# Load a LLM to create an agent using our custom tool

from langchain.llms import OpenAI

# Classes for initializing the agent that will use the custom tool

from langchain.agents import initialize_agent, AgentType


# Let's create an agent that uses our custom tool

# We set verbose=True to check if the agent is using the tool for generating the
final answer

llm = OpenAI(model="text-davinci-003", temperature=0)

agent = initialize_agent([retrieve_n_docs_tool], llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
```

The `initialize_agent` function takes in three parameters: a list of the custom tools, the language learning model, and the type of agent. We're using the OpenAI LLM and specifying the agent type as `ZERO_SHOT_REACT_DESCRIPTION`. With `verbose=True` we can check if the agent is using the tool when generating the final answer.

Once the agent has been set up, it can be queried:

```
response = agent.run("Are my info kept private when I shop with Paypal?")

print(response)
```

You should see something like the following printed output.

```
> Entering new AgentExecutor chain...

 I need to find out what Paypal does to protect my information

Action: retrieve_n_docs_tool

Action Input: "Paypal privacy policy"

Observation: Is it secure?

PayPal is the safer way to pay because we keep your financial information private. It isn't
shared with anyone else when you shop, so you don't have to worry about paying businesses and
people you don't know. On top of that, we've got your back. If your eligible purchase doesn't
arrive or doesn't match its description, we will refund you the full purchase price plus
shipping costs with PayPal's Buyer Protection program.


Why should I use PayPal?

It's Fast! We will help you pay in just a few clicks. Enter your email address and password,
and you're pretty much done! It's Simple! There's no need to run around searching for your
wallet. Better yet, you don't need to type in your financial details again and again when
making a purchase online. We make it simple for you to pay with just your email address and
password.


What is PayPal?

PayPal is a digital wallet that follows you wherever you go. Pay any way you want. Link your
credit cards to your PayPal Digital wallet, and when you want to pay, simply log in with your
username and password and pick which one you want to use.

Thought: I now understand how Paypal keeps my information secure

Final Answer: Yes, your information is kept private when you shop with Paypal. PayPal is a
digital wallet that follows you wherever you go and keeps your financial information private.
It is not shared with anyone else when you shop, and PayPal also offers Buyer Protection to
refund you the full purchase price plus shipping costs if your eligible purchase doesn't
arrive or doesn't match its description.


> Finished chain.
```

… along with this printed `response`.

```
Yes, your information is kept private when you shop with Paypal. PayPal is a digital wallet
that follows you wherever you go and keeps your financial information private. It is not
shared with anyone else when you shop, and PayPal also offers Buyer Protection to refund you
the full purchase price plus shipping costs if your eligible purchase doesn't arrive or
doesn't match its description.
```

By reading the agent printed output, we see that the agent decided to use the `retrieve_n_docs_tool` tool to retrieve relevant documents to the `Paypal privacy policy` query. The final answer is then generated using the information contained in the retrieved documents.

Conclusion

The experiment showcases the power of AI in information retrieval and comprehension, explicitly using a custom tool to provide accurate and contextual responses to user queries.
This experiment solves the problem of efficient and relevant information retrieval. Instead of manually reading through a large number of documents or frequently asked questions to find the information they need, the user can simply ask a question and get a relevant response. This can greatly enhance user experience, especially for customer support services or any platform that relies on providing accurate information swiftly.
Congratulations on finishing this module on tools! Now you can test your new knowledge with the module quizzes. The next (and last!) module will be about LLM agents.