LangChain and LlamaIndex Projects Lab Book:

Hooking Large Language Models Up to the Real World

by Mark Watson

LangChain and LlamaIndex Projects Lab Book: Hooking Large Language Models Up to the Real World

Using GPT-4, ChatGPT, and Hugging Face Models in Applications.

Mark Watson

This book is for sale at http://leanpub.com/langchain

This version was published on 2024-02-01

Also By Mark Watson

Safe For Humans AI
Das LangChain und LlamaIndex Projects Lab Buch: Large Language Models für die Echte Welt
Practical Python Artificial Intelligence Programming
Practical Artificial Intelligence Development With Racket
Practical Artificial Intelligence Programming With Clojure
Artificial Intelligence Using Swift
A Lisp Programmer Living in Python-Land: The Hy Programming Language
Haskell Tutorial and Cookbook
Loving Common Lisp, or the Savvy Programmer's Secret Weapon
Practical Artificial Intelligence Programming With Java

Contents

Preface

I have been working in the field of artificial intelligence since 1982 and without a doubt Large Language Models (LLMs) like GPT-4 represent the greatest advance in practical AI technology that I have experienced. Infrastructure projects like LangChain and LlamaIndex make it simpler to use LLMs and provide some level of abstraction to facilitate switching between LLMs. This book will use the LangChain[1] and LlamaIndex[2] projects along with the OpenAI GPT-3.5, GPT-4, ChatGPT APIs, and local models run on your computer to solve a series of interesting problems.

As I write this new edition in January 2024, I mostly run local LLMs but I still use OpenAI APIs, as well as well as APIs from Anthropic for the Claude 2 model.

Harrison Chase started the LangChain project in October 2022 and as I wrote the first book in February 2023 the GitHub repository for LangChain 171 contributors and as I write this new edition LangChain has 2100+ contributors on GitHub. Jerry Liu started the GPT Index project (recently renamed to LlamaIndex) at the end of 2022 and the GitHub Repository for LlamaIndex https://github.com/jerryjliu/gpt_index[3] currently has 54 contributors.

The GitHub repository for examples in this book is https://github.com/mark-watson/langchain-book-examples.git. Please note that I usually update the code in the examples repository fairly frequently for library version updates, etc.

While the documentation and examples online for LangChain and LlamaIndex are excellent, I am still motivated to write this book to solve interesting problems that I like to work on involving information retrieval, natural language processing (NLP), dialog agents, and the semantic web/linked data fields. I hope that you, dear reader, will be delighted with these examples and that at least some of them will inspire your future projects.

About the Author

I have written over 20 books, I have over 50 US patents, and I have worked at interesting companies like Google, Capital One, SAIC, Mind AI, and others. You can find links for reading most of my recent books free on my web site https://markwatson.com. If I had to summarize my career the short take would be that I have had a lot of fun and enjoyed my work. I hope that what you learn here will be both enjoyable and help you in your work.

If you would like to support my work please consider purchasing my books on Leanpub[4] and star

[1] https://github.com/langchain-ai/langchain
[2] https://github.com/run-llama/llama_index
[3] https://github.com/run-llama/llama_index
[4] https://leanpub.com/u/markwatson

my git repositories that you find useful on GitHub[5]. You can also interact with me on social media on Mastodon[6] and Twitter[7]. I am also available as a consultant: https://markwatson.com.

Book Cover

I live in Sedona, Arizona. I took the book cover photo in January 2023 from the street that I live on.

Acknowledgements

This picture shows me and my wife Carol who helps me with book production and editing.

[5] https://github.com/mark-watson?tab=repositories&q=&type=public
[6] https://mastodon.social/@mark_watson
[7] https://twitter.com/mark_l_watson

Figure 1. Mark and Carol Watson

I would also like to thank the following readers who reported errors or typos in this book: Armando Flores, Peter Solimine, and David Rupp.

Requirements for Running and Modifying Book Examples

I show full source code and a fair amount of example output for each book example so if you don't want to get access to some of the following APIs then you can still read along in the book.

To use OpenAI's GPT-3 and ChatGPT models you will need to sign up for an API key (free tier is

OK) at https://openai.com/api/ and set the environment variable **OPENAI_API_KEY** to your key value.

You will need to get an API key for examples using **Google's Knowledge Graph APIs**.

Reference: Google Knowledge Graph APIs[8].

The example programs using Google's Knowledge Graph APIs assume that you have the file **~/.google_api_key** in your home directory that contains your key from https://console.cloud.google.com/apis.

You will need to install **SerpApi for examples integrating web search**. Please reference: PyPi project page[9].

You can sign up for a free non-commercial 100 searches/month account with an email address and phone number at https://serpapi.com/users/welcome.

You will also need Zapier[10] account for the GMail and Google Calendar examples.

After reading though this book, you can review the website LangChainHub[11] which contains prompts, chains and agents that are useful for building LLM applications.

Issues and Workarounds for Using the Material in this Book

The libraries that I use in this book are frequently updated and sometimes the documentation or code links change, invalidating links in this book. I will try to keep everything up to date. Please report broken links to me.

In some cases you will need to use specific versions or libraries for some of the code examples.

Because the Python code listings use colorized text you may find that copying code from this eBook may drop space characters. All of the code listings are in the GitHub repository for this book so you should clone the repository to experiment with the example code.

[8] https://cloud.google.com/enterprise-knowledge-graph/docs/search-api
[9] https://pypi.org/project/google-search-results/
[10] https://zapier.com
[11] https://github.com/hwchase17/langchain-hub

Large Language Model Overview

Large language models[1] are a subset of artificial intelligence that use deep learning and neural networks to process natural language. Transformers[2] are a type of neural network architecture that can learn context in sequential data using self-attention mechanisms. They were introduced in 2017 by a team at Google Brain and have become popular for LLM research. Some examples of transformer-based[3] LLMs are BERT, GPT-3, T5 and Megatron-LM[4].

The main points we will discuss in this book are:

- LLMs are deep learning algorithms that can understand and generate natural language based on massive datasets.
- LLMs use techniques such as self-attention, masking, and fine-tuning to learn complex patterns and relationships in language. LLMs can understand and generate natural language because they use transformer models, which are a type of neural network that can process sequential data such as text using attention mechanisms. Attention mechanisms allow the model to focus on relevant parts of the input and output sequences while ignoring irrelevant ones.
- LLMs can perform various natural language processing (NLP) and natural language generation (NLG) tasks, such as summarization, translation, prediction, classification, and question answering.
- Even though LLMs were initially developed for NLP applications, LLMs have also shown potential in other domains such as computer vision and computational biology by leveraging their generalizable knowledge and transfer learning abilities.

BERT models[5] are one of the first types of transformer models that were widely used. BERT was developed by Google AI Language in 2018. BERT models are a family of masked language models that use transformer architecture to learn bidirectional representations of natural language. BERT models can understand the meaning of ambiguous words by using the surrounding text as context. The "magic trick" here is that training data comes almost free because in masking models, you programatically chose random words, replace them with a missing word token, and the model is trained to predict the missing words. This process is repeated with massive amounts of training data from the web, books, etc.

Here are some "papers with code" links for BERT (links are for code, paper links in the code repositories):

- https://github.com/allenai/scibert
- https://github.com/google-research/bert

[1] https://blogs.nvidia.com/blog/2022/10/10/llms-ai-horizon/
[2] https://www.linkedin.com/pulse/chatgpt-tip-iceberg-paul-golding
[3] https://factored.ai/transformer-based-language-models/
[4] https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)
[5] https://en.wikipedia.org/wiki/BERT_(Language_model)

Big Tech Businesses vs. Small Startups Using Large Language Models

Both Microsoft and Google play both sides of this business game: they want to sell cloud LLM services to developers and small startup companies and they would also like to achieve lock-in for their consumer services like Office 365, Google Docs and Sheets, etc.

Microsoft has been integrating AI technology into workplace emails, slideshows, and spreadsheets as part of its ongoing partnership with OpenAI, the company behind ChatGPT. Microsoft's Azure OpenAI service offers a powerful tool to enable these outcomes when leveraged with their data lake of more than two billion metadata and transactional elements.

As I write this, Google has just opened a public wait list for their Bard AI/chat search service. I have used various Google APIs for years in code I write. I have no favorites in the battle between tech giants, rather I am mostly interested in what they build that I can use in my own projects.

Hugging Face, which makes AI products and hosts those developed by other companies, is working on open-source rivals to ChatGPT and will use AWS[6] for that as well. Cohere AI, Anthropic, Hugging Face, and Stability AI are some of the startups that are using OpenAI and Hugging Face APIs for their products. As I write this chapter, I view Hugging Face as a great source of specialized models. I love that Hugging Face models can be run via their APIs and also self-hosted on our own servers and sometimes even on our laptops. Hugging Face is a fantastic resource and even though I use their models much less frequently in this book than OpenAI APIs, you should embrace the hosting and open source flexibility of Hugging Face. Here I use OpenAI APIs because I want you to get set up and creating your own projects quickly.

Dear reader, I didn't write this book for developers working at established AI companies (although I hope such people find the material here useful!). I wrote this book for small developers who want to scratch their own itch by writing tools that save them time. I also wrote this book hoping that it would help developers build capabilities into the programs they design and write that rival what the big tech companies are doing.

[6] https://iblnews.org/aws-partners-with-hugging-face-an-ai-startup-rival-to-chatgpt-working-on-open-source-models/

Getting Started With LangChain

LangChain[1] is a framework for building applications with large language models (LLMs) through chaining different components together. Some of the applications of LangChain are chatbots, generative question-answering, summarization, data-augmented generation and more. LangChain can save time in building chatbots and other systems by providing a standard interface for chains, agents and memory, as well as integrations with other tools and end-to-end examples. We refer to "chains" as sequences of calls (to an LLMs and a different program utilities, cloud services, etc.) that go beyond just one LLM API call. LangChain provides a standard interface for chains, many integrations with other tools, and end-to-end chains for common applications. Often you will find existing chains already written that meet the requirements for your applications.

For example, one can create a chain that takes user input, formats it using a PromptTemplate, and then passes the formatted response to a Large Language Model (LLM) for processing.

While LLMs are very general in nature which means that while they can perform many tasks effectively, they often can not directly provide specific answers to questions or tasks that require deep domain knowledge or expertise. LangChain provides a standard interface for agents, a library of agents to choose from, and examples of end-to-end agents.

LangChain Memory is the concept of persisting state between calls of a chain or agent. LangChain provides a standard interface for memory, a collection of memory implementations, and examples of chains/agents that use memory[2]. LangChain provides a large collection of common utils to use in your application. Chains go beyond just a single LLM call, and are sequences of calls (whether to an LLM or a different utility). LangChain provides a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.

LangChain can be integrated with one or more model providers, data stores, APIs, etc. LangChain can be used for in-depth question-and-answer chat sessions, API interaction, or action-taking. LangChain can be integrated with Zapier's platform through a natural language API interface (we have an entire chapter dedicated to Zapier integrations).

Installing Necessary Packages

For the purposes of examples in this book, you might want to create a new Anaconda or other Python environment and install:

[1] https://api.python.langchain.com/en/stable/langchain_api_reference.html

```
pip install -U langchain llama_index openai langchain-openai
pip install -U langchain-community langchainhub
pip install -U kor pydrive pandas rdflib
pip install -U google-search-results SPARQLWrapper
```

For the rest of this chapter we will use the subdirectory **langchain_getting_started** and in the next chapter use **llama-index_case_study** in the GitHub repository for this book.

Creating a New LangChain Project

Simple LangChain projects are often just a very short Python script file. As you read this book, when any example looks interesting or useful, I suggest copying the requirements.txt and Python source files to a new directory and making your own GitHub private repository to work in. Please make the examples in this book "your code," that is, freely reuse any code or ideas you find here.

Basic Usage and Examples

While I try to make the material in this book independent, something you can enjoy with no external references, you should also take advantage of the high quality Langchain Quickstart Guide documentation[2] and the individual detailed guides for prompts, chat, document loading, indexes, etc.

As we work through some examples please keep in mind what it is like to use the ChatGPT web application: you enter text and get responses. The way you prompt ChatGPT is obviously important if you want to get useful responses. In code examples we automate and formalize this manual process.

You need to choose a LLM to use. We will usually choose the GPT-4 API from OpenAI because it is general purpose but is more expensive that the older GPT-3.5 APIs. You will need to sign up[3] for an API key and set it as an environment variable:

```
export OPENAI_API_KEY="YOUR KEY GOES HERE"
```

Both the libraries **openai** and **langchain** will look for this environment variable and use it. We will look at a few simple examples in a Python REPL. We will start by just using OpenAI's text prediction API:

[2] https://python.langchain.com/docs/get_started/quickstart
[3] https://platform.openai.com/account/api-keys

```
$ python
>>> from langchain_openai import OpenAI
>>> llm = OpenAI(temperature=0.8)
>>> s = llm("John got into his new sports car, and he drove it")
>>> s
' to work\n\nJohn started up his new sports car and drove it to work. He had a huge \
smile on his face as he drove, excited to show off his new car to his colleagues. Th
e wind blowing through his hair, and the sun on his skin, he felt a sense of freedom
 and joy as he cruised along the road. He arrived at work in no time, feeling refres
hed and energized.'
>>> s = llm("John got into his new sports car, and he drove it")
>>> s
" around town\n\nJohn drove his new sports car around town, enjoying the feeling of \
the wind in his hair. He stopped to admire the view from a scenic lookout, and then
sped off to the mall to do some shopping. On the way home, he took a detour down a w
inding country road, admiring the scenery and enjoying the feel of the car's powerfu
l engine. By the time he arrived back home, he had a huge smile on his face."
```

Notice how when we ran the same input text prompt twice that we see different results.Setting the temperature in line 3 to a higher value increases the randomness.

Our next example is in the source file **directions_template.py** in the directory **langchain_getting_started** and uses the **PromptTemplate** class. A prompt template is a reproducible way to generate a prompt. It contains a text string ("the template"), that can take in a set of parameters from the end user and generate a prompt. The prompt template may contain language model instructions, few-shot examples to improve the model's response, or specific questions for the model to answer.

```
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI
llm = OpenAI(temperature=0.9)

def get_directions(thing_to_do):
    prompt = PromptTemplate(
        input_variables=["thing_to_do"],
        template="How do I {thing_to_do}?",
    )
    prompt_text = prompt.format(thing_to_do=thing_to_do)
    print(f"\n{prompt_text}:")
    return llm(prompt_text)

print(get_directions("get to the store"))
print(get_directions("hang a picture on the wall"))
```

You could just write Python string manipulation code to create a prompt but using the utility class **PromptTemplate** is more legible and works with any number of prompt input variables.

The output is:

```
$ python directions_template.py

How do I get to the store?:

To get to the store, you will need to use a mode of transportation such as walking, \
driving, biking, or taking public transportation. Depending on the location of the s
tore, you may need to look up directions or maps to determine the best route to take
.

How do I hang a picture on the wall?:

1. Find a stud in the wall, or use two or three wall anchors for heavier pictures.
2. Measure and mark the wall where the picture hanger will go.
3. Pre-drill the holes and place wall anchors if needed.
4. Hammer the picture hanger into the holes.
5. Hang the picture on the picture hanger.
```

The next example in the file **country_information.py** is derived from an example in the LangChain documentation. In this example we use **PromptTemplate** that contains the pattern we would like the LLM to use when returning a response.

```
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI
llm = OpenAI(temperature=0.9)

def get_country_information(country_name):
    print(f"\nProcessing {country_name}:")
    global prompt
    if "prompt" not in globals():
        print("Creating prompt...")
        prompt = PromptTemplate(
            input_variables=["country_name"],
            template = """
Predict the capital and population of a country.

Country: {country_name}
Capital:
Population:""",
```

```
        )
    prompt_text = prompt.format(country_name=country_name)
    print(prompt_text)
    return llm(prompt_text)

print(get_country_information("Canada"))
print(get_country_information("Germany"))
```

You can use the ChatGPT web interface to experiment with prompts and when you find a pattern that works well then write a Python script like the last example, but changing the data you supply in the **PromptTemplate** instance.

The output of the last example is:

```
 $ python country_information.py

Processing Canada:
Creating prompt...

Predict the capital and population of a country.

Country: Canada
Capital:
Population:


Capital: Ottawa
Population: 37,058,856 (as of July 2020)

Processing Germany:

Predict the capital and population of a country.

Country: Germany
Capital:
Population:


Capital: Berlin
Population: 83,02 million (est. 2019)
```

Creating Embeddings

We will reference the LangChain embeddings documentation[4]. We can use a Python REPL to see what text to vector space embeddings might look like:

```
$ python
Python 3.10.8 (main, Nov 24 2022, 08:08:27) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from langchain_openai import OpenAIEmbeddings
>>> embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
>>> text = "Mary has blond hair and John has brown hair. Mary lives in town and John\
 lives in the country."
>>> doc_embeddings = embeddings.embed_documents([text])
>>> doc_embeddings
[[0.007727328687906265, 0.0009025644976645708, -0.0033224383369088173, -0.0179449208\
0807686, -0.017969949170947075, 0.028506645932793617, -0.013414892368018627, 0.00466
76816418766975, -0.0024965214543044567, -0.02662956342101097,
...]]
>>> query_embedding = embeddings.embed_query("Does John live in the city?")
>>> query_embedding
[0.028048301115632057, 0.011499025858938694, -0.00944007933139801, -0.02080961130559\
4444, -0.023904507979750633, 0.018750663846731186, -0.01626438833773136, 0.018129095
435142517,
...]
>>>
```

Notice that the **doc_embeddings** is a list where each list element is the embeddings for one input text document. The **query_embedding** is a single embedding. Please read the above linked embedding documentation.

We will use vector stores to store calculated embeddings for future use. In the next chapter we will see a document database search example using LangChain and Llama-Index.

Using LangChain Vector Stores to Query Documents

We will reference the LangChain Vector Stores documentation[5]. You need to install a few libraries:

[4] https://python.langchain.com/en/latest/reference/modules/embeddings.html
[5] https://python.langchain.com/en/latest/reference/modules/vectorstore.html

```
pip install chroma
pip install chromadb
pip install unstructured pdf2image pytesseract
```

The example script is **doc_search.py**:

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.document_loaders import DirectoryLoader
from langchain import VectorDBQA
from langchain_openai import OpenAI

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

loader = DirectoryLoader('../data/', glob="**/*.txt")
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=2500, chunk_overlap=0)

texts = text_splitter.split_documents(documents)

docsearch = Chroma.from_documents(texts, embeddings)

qa = VectorDBQA.from_chain_type(llm=OpenAI(),
                                chain_type="stuff",
                                vectorstore=docsearch)

def query(q):
    print(f"Query: {q}")
    print(f"Answer: {qa.run(q)}")

query("What kinds of equipment are in a chemistry laboratory?")
query("What is Austrian School of Economics?")
query("Why do people engage in sports?")
query("What is the effect of body chemistry on exercise?")
```

The **DirectoryLoader** class is useful for loading a directory full of input documents. In this example we specified that we only want to process text files, but the file matching pattern could have also specified PDF files, etc.

The output is:

```
$ python doc_search.py
Created a chunk of size 1055, which is longer than the specified 1000
Running Chroma using direct local API.
Using DuckDB in-memory for database. Data will be transient.
Query: What kinds of equipment are in a chemistry laboratory?
Answer:  A chemistry lab would typically include glassware, such as beakers, flasks,\
 and test tubes, as well as other equipment such as scales, Bunsen burners, and ther
mometers.
Query: What is Austrian School of Economics?
Answer:  The Austrian School is a school of economic thought that emphasizes the spo\
ntaneous organizing power of the price mechanism. Austrians hold that the complexity
 of subjective human choices makes mathematical modelling of the evolving market ext
remely difficult and advocate a "laissez faire" approach to the economy. Austrian Sc
hool economists advocate the strict enforcement of voluntary contractual agreements
between economic agents, and hold that commercial transactions should be subject to
the smallest possible imposition of forces they consider to be (in particular the sm
allest possible amount of government intervention). The Austrian School derives its
name from its predominantly Austrian founders and early supporters, including Carl M
enger, Eugen von Bohm-Bawerk and Ludwig von Mises.
Query: Why do people engage in sports?
Answer:  People engage in sports for leisure and entertainment, as well as for physi\
cal exercise and athleticism.
Query: What is the effect of body chemistry on exercise?
Answer:  Body chemistry can affect the body's response to exercise, as certain hormo\
nes and enzymes produced by the body can affect the energy levels and muscle perform
ance. Chemicals in the body, such as adenosine triphosphate (ATP) and urea, can affe
ct the body's energy production and muscle metabolism during exercise. Additionally,
 the body's levels of electrolytes, vitamins, and minerals can affect exercise perfo
rmance.
Exiting: Cleaning up .chroma directory
```

Example Using LangChain Integrations: Using Server APIs for Google Search

The example shown here is in the directory **from_langchain_docs** in the source file **search_simple.py**. The relevant LangChain Integrations documentation page is https://python.langchain.com/docs/integrations/tools/google_serper.

```
# make sure SERPER_API_KEY is set in your environment

from langchain_community.utilities import GoogleSerperAPIWrapper
search_helper = GoogleSerperAPIWrapper()

def search(query):
    return search_helper.run(query)

print(search("What is the capital of Arizona?"))
#print(search("Sedona Arizona?"))
```

You will need a Server API key form https://serper.dev. Currently you can get a free key for 2500 API calls. After that the paid tier currently starts at $50 for 50K API calls and these credits must be used within a 6 month period.

LangChain Overview Wrap Up

We will continue using LangChain for the rest of this book as well as the LlamaIndex library that we introduce in the next chapter.

I cover just the subset of LangChain that I use in my own projects in this book. I urge you to read the LangChain documentation and to explore public LangChain chains that users have written on Langchain-hub[6].

[6] https://github.com/hwchase17/langchain-hub

Overview of LlamaIndex

The popular LlamaIndex project used to be called GPT-Index but has been generalized to work with more models that just GPT-3, for example using Hugging Face embeddings[1].

LlamaIndex[2] is a project that provides a central interface to connect your language models with external data. It was created by Jerry Liu and his team in the fall of 2022. It consists of a set of data structures designed to make it easier to use large external knowledge bases with language models[3]. Some of its uses are:

- Querying structured data such as tables or databases using natural language
- Retrieving relevant facts or information from large text corpora
- Enhancing language models with domain-specific knowledge

LlamaIndex supports a variety of document types, including:

- Text documents are the most common type of document. They can be stored in a variety of formats, such as .txt, .doc, and .pdf.
- XML documents are a type of text document that is used to store data in a structured format.
- JSON documents are a type of text document that is used to store data in a lightweight format.
- HTML documents are a type of text document that is used to create web pages.
- PDF documents are a type of text document that is used to store documents in a fixed format.

LlamaIndex can also index data that is stored in a variety of databases, including:

- SQL databases such as MySQL, PostgreSQL, and Oracle. NoSQL databases such as MongoDB, Cassandra, and CouchDB.
- Solr is a popular open-source search engine that provides high performance and scalability.
- Elasticsearch is another popular open-source search engine that offers a variety of features, including full-text search, geospatial search, and machine learning.
- Apache Cassandra is a NoSQL database that can be used to store large amounts of data.
- MongoDB is another NoSQL database that is easy to use and scale.
- PostgreSQL is a relational database that is widely used in enterprise applications.

LlamaIndex is a flexible framework that can be used to index a variety of document types and data sources.

We will look first at a short example derived from the LlamaIndex documentation and later look at the parts of the LlmaIndex source code that uses LangChain.

[1] https://gpt-index.readthedocs.io/en/latest/how_to/customization/embeddings.html
[2] https://github.com/jerryjliu/gpt_index/blob/main/docs/index.rst
[3] https://gpt-index.readthedocs.io/en/latest/index.html

Using LlamaIndex to Search Local Documents Using GPT-3

The following example is similar to the last example in the overview chapter on LangChain. In line 8 we use a utility data loader function provided by LlamaIndex to read documents in an input directory. As a demonstration we save the index (consisting of document embeddings) to disk and reload it. This technique is useful when you have a large number of static documents so the indexing procedure can take a while and require many OpenAI API calls. As an example, you might have many gigabytes of company documentation that doesn't change often so it makes sense to only occasionally recreate the index.

```
1  # make sure you set the following environment variable is set:
2  #   OPENAI_API_KEY
3
4  from llama_index import GPTVectorStoreIndex, SimpleDirectoryReader
5  from llama_index import StorageContext, load_index_from_storage
6
7  documents = SimpleDirectoryReader('../data').load_data()
8  # index = GPTListIndex(documents) # llama_index < 0.5
9  index = GPTVectorStoreIndex.from_documents(documents)
10 engine = index.as_query_engine()
11 print(engine.query("what are key economic indicators?"))
12
13 # save to disk
14 index.storage_context.persist(persist_dir='./cache')
15
16 # load from disk
17 storage_context = StorageContext.from_defaults(persist_dir='./cache')
18 index = load_index_from_storage(storage_context)
19 engine = index.as_query_engine()
20
21 # search for a document
22 print(engine.query("effect of body chemistry on exercise?"))
```

You may have noticed that the query defined on line 17 is the same query that we used last chapter.

```
$ python test_from_docs.py

Key economic indicators are measures of economic activity that are used to assess th\
e health of an economy. Examples of key economic indicators include gross domestic p
roduct (GDP), unemployment rate, inflation rate, consumer price index (CPI), balance
 of trade, and industrial production.

The effect of body chemistry on exercise depends on the type of exercise being perfo\
rmed. For aerobic exercise, the body needs to be able to produce enough energy to su
stain the activity. This requires the body to have adequate levels of oxygen, glucos
e, and other nutrients in the bloodstream. For anaerobic exercise, the body needs to
 be able to produce energy without relying on oxygen. This requires the body to have
 adequate levels of lactic acid, creatine phosphate, and other energy-producing mole
cules. In both cases, the body's chemistry must be balanced in order for the exercis
e to be effective.
```

Using LlamaIndex for Question Answering from a List of Web Sites

In this example we use the **trafilatura** and **html2text** libraries to get text from a web page that we will index and search. The class **TrafilaturaWebReader** does the work of creating local documents from a list of web page URIs and the index class **GPTListIndex** builds a local index for use with OpenAI API calls to implement search.

The following listing shows the file **web_page_QA.py**:

```
# Derived from the example at:
# https://github.com/jerryjliu/gpt_index/blob/main/examples/data_connectors/WebPageD\
emo.ipynb

# pip install llama-index, html2text, trafilatura

from llama_index import GPTListIndex
from llama_index import TrafilaturaWebReader

def query_website(url_list, *questions):
    documents = TrafilaturaWebReader().load_data(url_list)
    # index = GPTListIndex(documents) # llama_index < 0.5
    index = GPTListIndex.from_documents(documents)
    engine = index.as_query_engine()
    for question in questions:
```

```
        print(f"\n== QUESTION: {question}\n")
        response = engine.query(question)
        print(f"== RESPONSE: {response}")

if __name__ == "__main__":
  url_list = ["https://markwatson.com"]
  query_website(url_list, "How many patents does Mark have?",
                          "How many books has Mark written?")
```

This example is not efficient because we create a new index for each web page we want to search. That said, this example (that was derived from an example in the LlamaIndex documentation) implements a pattern that you can use, for example, to build a reusable index of your company's web site and build an end-user web search app.

The output for these three test questions in the last code example is:

```
 $ python web_page_QA.py
documents:
[Document(text='Mark Watson's Artificial Intelligence Books and Blog\n'
               'Author of 20+ books on AI and I have 50+ US patents. Here I '
               'talk about both technology and provide links to read my '
               'published books online.\n'
               "By registering you agree to Substack's Terms of Service, our "
               'Privacy Policy, and our Information Collection Notice',
          doc_id='c6ed3ddd-b160-472b-9c7d-fd0446762cc8',
          embedding=None,
          doc_hash='126a335ed76d5d79ad94470dd4dd06ab5556b9f1347e5be25ecc595e0290ab57\
',
          extra_info=None)]

== QUESTION: How many patents does Mark have?

== RESPONSE:
Mark Watson has 50+ US patents.

== QUESTION: How many books has Mark written?

== RESPONSE:
Mark has written 20+ books on AI.
```

LlamaIndex/GPT-Index Case Study Wrap Up

LlamaIndex is a set of data structures and library code designed to make it easier to use large external knowledge bases such as Wikipedia. LlamaIndex creates a vectorized index from your document data, making it highly efficient to query. It then uses this index to identify the most relevant sections of the document based on the query.

LlamaIndex is useful because it provides a central interface to connect your LLM's with external data and offers data connectors to your existing data sources and data formats (API's, PDF's, docs, SQL, etc.). It provides a simple, flexible interface between your external data and LLMs.

Some projects that use LlamaIndex include building personal assistants with LlamaIndex and GPT-3.5, using LlamaIndex for document retrieval, and combining answers across documents.

Retrieval Augmented Generation (RAG) Applications

TBD: this chapter will be complete by March 1, 2023

Note to readers: this chapter was added February 2024 and supersedes some of the older material in later chapters on indexing and chatting about data from local text and PDF documents as well as data from web sites.

Retrieval Augmented Generation (RAG) Applications work by pre-processing a user's query to search for indexed document fragments that are semantically similar to the uer's query. These fragments are concatenated together as context text that is attached to the user's query and then passed of to a LLM model. The LLM can preferentially use information in this context text as well as innate knowledge stored in the LLM to process user queries.

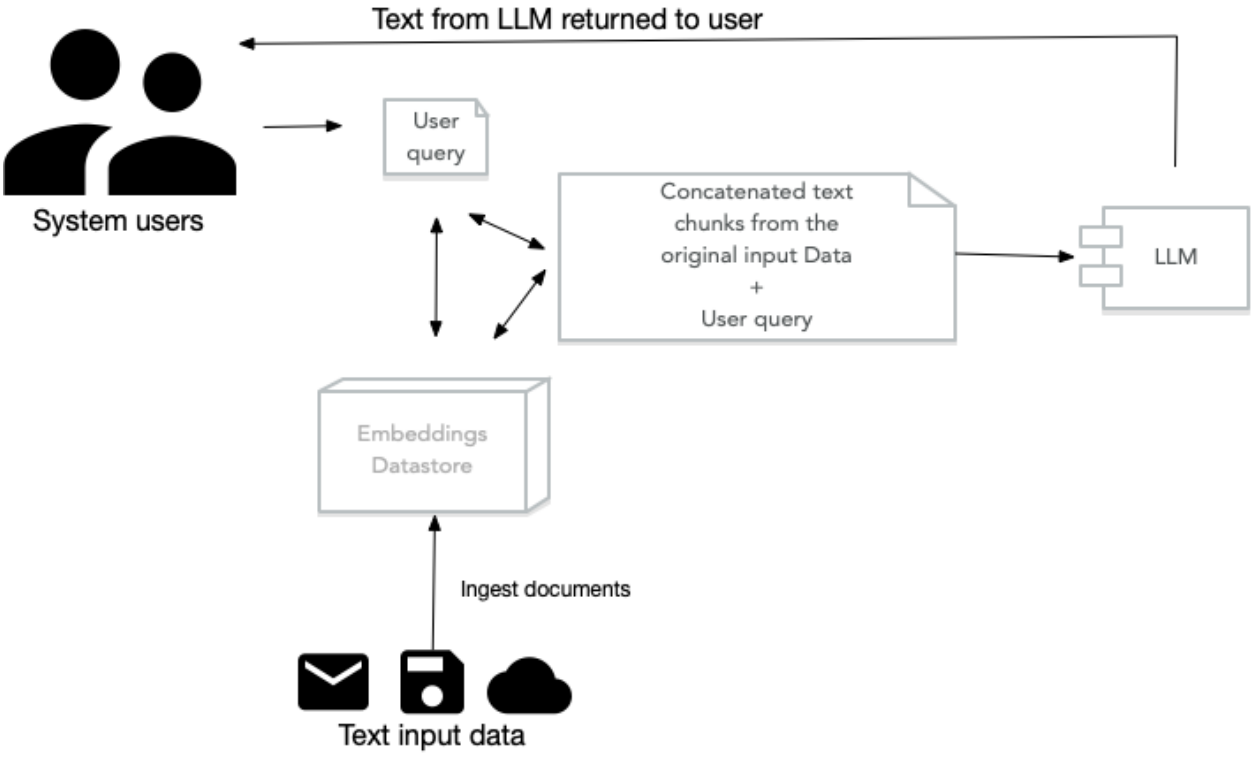


Figure 2. RAG System Overview

TBD: this chapter will be complete by March 1, 2023

Using Google's Knowledge Graph APIs With LangChain

Google's Knowledge Graph (KG) is a knowledge base that Google uses to serve relevant information in an infobox beside its search results. It allows the user to see the answer in a glance, as an instant answer. The data is generated automatically from a variety of sources, covering places, people, businesses, and more. I worked at Google in 2013 on a project that used their KG for an internal project.

Google's public Knowledge Graph Search API lets you find entities in the Google Knowledge Graph. The API uses standard schema.org types and is compliant with the JSON-LD specification. It supports entity search and lookup.

You can use the Knowledge Graph Search API to build applications that make use of Google's Knowledge Graph. For example, you can use the API to build a search engine that returns results based on the entities in the Knowledge Graph.

In the next chapter we also use the public KGs DBPedia and Wikidata. One limitation of Google's KG APIs is that it is designed for entity (people, places, organizations, etc.) lookup. When using DBPedia and Wikidata it is possible to find a wider range of information using the SPARQL query language, such as relationships between entities. You can use the Google KG APIs to find some entity relationships, e.g., all the movies directed by a particular director, or all the books written by a particular author. You can also use the API to find information like all the people who have worked on a particular movie, or all the actors who have appeared in a particular TV show.

Setting Up To Access Google Knowledge Graph APIs

To get an API key for Google's Knowledge Graph Search API, you need to go to the Google API Console, enable the Google Knowledge Graph Search API, and create an API key to use in your project. You can then use this API key to make requests to the Knowledge Graph Search API.

To create your application's API key, follow these steps:

- Go to the API Console.
- From the projects list, select a project or create a new one.
- If the APIs & services page isn't already open, open the left side menu and select APIs & services.
- On the left, choose Credentials.
- Click Create credentials and then select API key.

You can then use this API key to make requests to the Knowledge Graph Search APIs.

When I use Google's APIs I set the access key in ~/**.google_api_key** and read in the key using:

```
api_key=open(str(Path.home())+"/.google_api_key").read()
```

You can also use environment variables to store access keys. Here is a code snippet for making an API call to get information about me:

```
import json
from urllib.parse import urlencode
from urllib.request import urlopen
from pathlib import Path
from pprint import pprint

api_key =
    open(str(Path.home()) + "/.google_api_key").read()
query = "Mark Louis Watson"
service_url =
    "https://kgsearch.googleapis.com/v1/entities:search"
params = {
    "query": query,
    "limit": 10,
    "indent": True,
    "key": api_key,
}
url = service_url + "?" + urlencode(params)
response = json.loads(urlopen(url).read())
pprint(response)
```

The JSON-LD output would look like:

```
{'@context': {'@vocab': 'http://schema.org/',
              'EntitySearchResult':
              'goog:EntitySearchResult',
              'detailedDescription':
              'goog:detailedDescription',
              'goog': 'http://schema.googleapis.com/',
              'kg': 'http://g.co/kg',
              'resultScore': 'goog:resultScore'},
 '@type': 'ItemList',
 'itemListElement': [{'@type': 'EntitySearchResult',
                      'result': {'@id': 'kg:/m/0b6_g82',
```

```
                                '@type': ['Thing',
                                          'Person'],
                                'description': 'Author',
                                'name':
                                'Mark Louis Watson',
                                'url':
                                'http://markwatson.com'},
                    'resultScore': 43}]}
```

In order to not repeat the code for getting entity information from the Google KG, I wrote a utility **Google_KG_helper.py** that encapsulates the previous code and generalizes it into a mini-library:

```
"""Client for calling Knowledge Graph Search API."""

import json
from urllib.parse import urlencode
from urllib.request import urlopen
from pathlib import Path
from pprint import pprint

api_key =
    open(str(Path.home()) + "/.google_api_key").read()

# use Google search API to get information
# about a named entity:

def get_entity_info(entity_name):
    service_url =
      "https://kgsearch.googleapis.com/v1/entities:search"
    params = {
        "query": entity_name,
        "limit": 1,
        "indent": True,
        "key": api_key,
    }
    url = service_url + "?" + urlencode(params)
    response = json.loads(urlopen(url).read())
    return response

def tree_traverse(a_dict):
    ret = []
    def recur(dict_2, a_list):
        if isinstance(dict_2, dict):
```

```
            for key, value in dict_2.items():
                if key in ['name', 'description',
                           'articleBody']:
                    a_list += [value]
                recur(value, a_list)
        if isinstance(dict_2, list):
            for x in dict_2:
                recur(x, a_list)
    recur(a_dict, ret)
    return ret


def get_context_text(entity_name):
    json_data = get_entity_info(entity_name)
    return ' '.join(tree_traverse(json_data))

if __name__ == "__main__":
    get_context_text("Bill Clinton")
```

The main test script is in the file **Google_Knowledge_Graph_Search.py**:

```
"""Example of Python client calling the
   Knowledge Graph Search API."""

from llama_index import GPTListIndex, Document

import Google_KG_helper

def kg_search(entity_name, *questions):
    ret = ""
    context_text =
      Google_KG_helper.get_context_text(entity_name)
    print(f"Context text: {context_text}")
    doc = Document(context_text)
    index = GPTListIndex([doc])
    for question in questions:
        response = index.query(question)
        ret +=
          f"QUESTION:  {question}\nRESPONSE: {response}\n"
    return ret

if __name__ == "__main__":
    s = kg_search("Bill Clinton",
```

```
                  "When was Bill president?")
    print(s)
```

The example output is:

```
$ python Google_Knowledge_Graph_Search.py
Context text: Bill Clinton 42nd U.S. President William Jefferson Clinton is an Ameri\
can retired politician who served as the 42nd president of the United States from 19
93 to 2001.
INFO:root:> [build_index_from_documents] Total LLM token usage: 0 tokens
INFO:root:> [build_index_from_documents] Total embedding token usage: 0 tokens
INFO:root:> [query] Total LLM token usage: 77 tokens
INFO:root:> [query] Total embedding token usage: 0 tokens
QUESTION:  When was Bill president?
RESPONSE:
Bill Clinton was president from 1993 to 2001.
```

Accessing Knowledge Graphs from Google, DBPedia, and Wikidata allows you to integrate real world facts and knowledge with your applications. While I mostly work in the field of deep learning I frequently also use Knowledge Graphs in my work and in my personal research. I think that you, dear reader, might find accessing highly structured data in KGs to be more reliable and in many cases simpler than using web scraping.

Using DBPedia and WikiData as Knowledge Sources

Both DBPedia[1] and Wikidata[2] are public Knowledge Graphs (KGs) that store data as Resource Description Framework (RDF)[3] and are accessed through the SPARQL Query Language for RDF[4]. The examples for this project are in the GitHub repository for this book[5] in the directory **kg_search**.

I am not going to spend much time here discussing RDF and SPARQL. Instead I ask you to read online the introductory chapter **Linked Data, the Semantic Web, and Knowledge Graphs** in my book A Lisp Programmer Living in Python-Land: The Hy Programming Language[6].

As we saw in the last chapter, a Knowledge Graph (that I often abbreviate as KG) is a graph database using a schema to define types (both objects and relationships between objects) and properties that link property values to objects. The term "Knowledge Graph" is both a general term and also sometimes refers to the specific Knowledge Graph used at Google which I worked with while working there in 2013. Here, we use KG to reference the general technology of storing knowledge in graph databases.

DBPedia and Wikidata are similar, with some important differences. Here is a summary of some similarities and differences between DBPedia and Wikidata:

- Both projects aim to provide structured data from Wikipedia in various formats and languages. Wikidata also has data from other sources so it contains more data and more languages.
- Both projects use RDF as a common data model and SPARQL as a query language.
- DBPedia extracts data from the infoboxes in Wikipedia articles, while Wikidata collects data entered through its interfaces by both users and automated bots.
- Wikidata requires sources for its data, while DBPedia does not.
- DBpedia is more popular in the Semantic Web and Linked Open Data communities, while Wikidata is more integrated with Wikimedia projects.

To the last point: I personally prefer DBPedia when experimenting with the semantic web and linked data, mostly because DBPedia URIs are human readable while Wikidata URIs are abstract. The following URIs represent the town I live in, Sedona Arizona:

- DBPedia: https://dbpedia.org/page/Sedona,_Arizona

[1] https://www.dbpedia.org
[2] https://www.wikidata.org/wiki/Wikidata:Main_Page
[3] https://www.w3.org/RDF/
[4] https://www.w3.org/TR/rdf-sparql-query/
[5] https://github.com/mark-watson/langchain-book-examples
[6] https://leanpub.com/hy-lisp-python/read

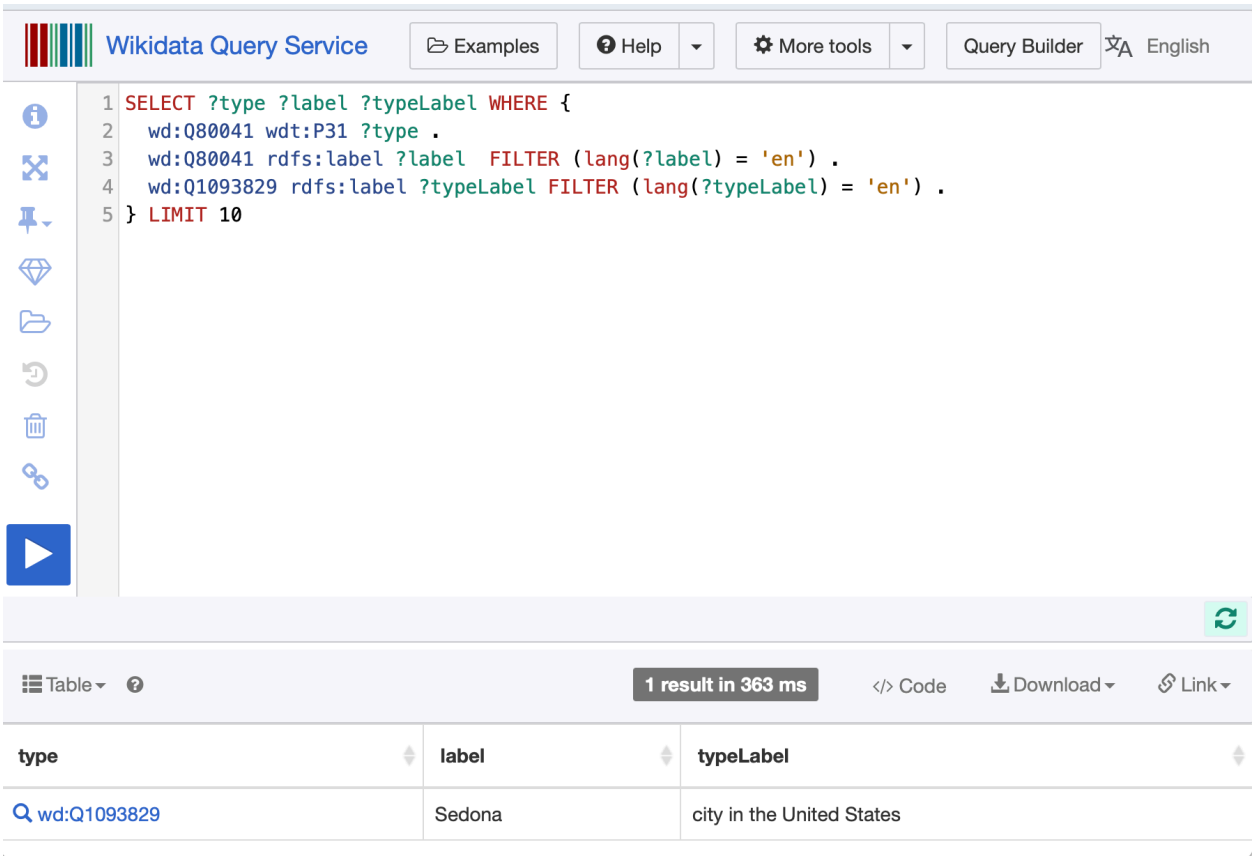
- Wikidata: https://www.wikidata.org/wiki/Q80041

In RDF we enclose URIs in angle brackets like **<https://www.wikidata.org/wiki/Q80041>**.

If you read the chapter on RDF and SPARQL in my book link that I mentioned previously, then you know that RDF data is represented by triples where each part is named:

- subject
- property
- object

We will look at two similar examples in this chapter, one using DBPedia and one using Wikidata. Both services have SPARQL endpoint web applications that you will want to use for exploring both KGs. We will look at the DBPedia web interface later. Here is the Wikidata web interface:

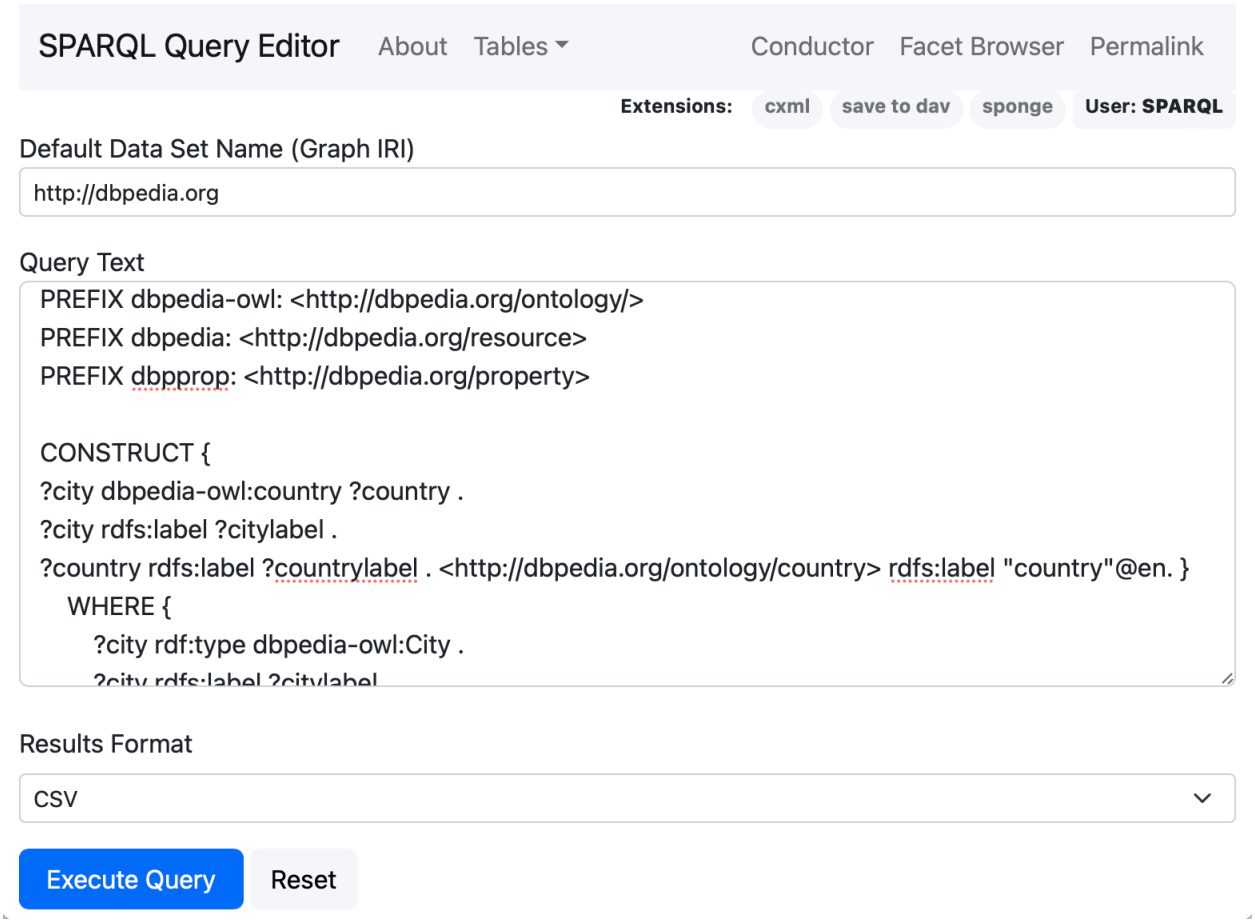


In this SPARQL query the prefix **wd:** stands for Wikidata data while the prefix **wdt:** stands for Wikidata type (or property). The prefix **rdfs:** stands for RDF Schema.

Using DBPedia as a Data Source

DBpedia is a community-driven project that extracts structured content from Wikipedia and makes it available on the web as a Knowledge Graph (KG). The KG is a valuable resource for researchers and developers who need to access structured data from Wikipedia. With the use of SPARQL queries to DBpedia as a data source we can write a variety applications, including natural language processing, machine learning, and data analytics. We demonstrate the effectiveness of DBpedia as a data source by presenting several examples that illustrate its use in real-world applications. In my experience, DBpedia is a valuable resource for researchers and developers who need to access structured data from Wikipedia.

In general you will start projects using DBPedia by exploring available data using the web app https://dbpedia.org/sparql that can be seen in this screen shot:



The following listing of file **dbpedia_generate_rdf_as_nt.py** shows Python code for making a SPARQL query to DBPedia and saving the results as RDF triples in NT format in a local text file:

```
from SPARQLWrapper import SPARQLWrapper
from rdflib import Graph

sparql = SPARQLWrapper("http://dbpedia.org/sparql")
sparql.setQuery("""
    PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
    PREFIX dbpedia: <http://dbpedia.org/resource>
    PREFIX dbpprop: <http://dbpedia.org/property>

    CONSTRUCT {
        ?city dbpedia-owl:country ?country .
        ?city rdfs:label ?citylabel .
        ?country rdfs:label ?countrylabel .
        <http://dbpedia.org/ontology/country> rdfs:label "country"@en .
    }
    WHERE {
        ?city rdf:type dbpedia-owl:City .
        ?city rdfs:label ?citylabel .
        ?city dbpedia-owl:country ?country .
        ?country rdfs:label ?countrylabel .
        FILTER (lang(?citylabel) = 'en')
        FILTER (lang(?countrylabel) = 'en')
    }
    LIMIT 50
""")
sparql.setReturnFormat("rdf")
results = sparql.query().convert()

g = Graph()
g.parse(data=results.serialize(format="xml"), format="xml")

print("\nresults:\n")
results = g.serialize(format="nt").encode("utf-8").decode('utf-8')
print(results)

text_file = open("sample.nt", "w")
text_file.write(results)
text_file.close()
```

Here is the printed output from running this script (most output not shown, and manually edited to fit page width):

```
 $ python generate_rdf_as_nt.py
results:

<http://dbpedia.org/resource/Ethiopia>
    <http://www.w3.org/2000/01/rdf-schema#label>
    "Ethiopia"@en .
<http://dbpedia.org/resource/Valentin_Alsina,_Buenos_Aires>
    <http://www.w3.org/2000/01/rdf-schema#label>
    "Valentin Alsina, Buenos Aires"@en .
<http://dbpedia.org/resource/Davyd-Haradok>
    <http://dbpedia.org/ontology/country>
    <http://dbpedia.org/resource/Belarus> .
<http://dbpedia.org/resource/Davyd-Haradok>
    <http://www.w3.org/2000/01/rdf-schema#label>
    "Davyd-Haradok"@en .
<http://dbpedia.org/resource/Belarus>
    <http://www.w3.org/2000/01/rdf-schema#label>
    "Belarus"@en .
 ...
```

This output was written to a local file **sample.nt**. I divided this example into two separate Python scripts because I thought it would be easier for you, dear reader, to experiment with fetching RDF data separately from using a LLM to process the RDF data. In production you may want to combine KG queries with semantic analysis.

This code example demonstrates the use of the **GPTSimpleVectorIndex** for querying RDF data and retrieving information about countries. The function **download_loader** loads data importers by string name. While it is not a type safe to load a Python class by name using a string, if you misspell the name of the class to load the call to **download_loader** then a Python **ValueError("Loader class name not found in library")** error is thrown. The GPTSimpleVectorIndex class represents an index data structure that can be used to efficiently search and retrieve information from the RDF data. This is similar to other types of LlamaIndex vector index types for different types of data sources.

Here is the script **dbpedia_rdf_query.py**:

```
"Example from documentation"

from llama_index import GPTSimpleVectorIndex, Document
from llama_index import download_loader

RDFReader = download_loader("RDFReader")
doc = RDFReader().load_data("sample.nt")
index = GPTSimpleVectorIndex(doc)

```

```
result = index.query("list all countries in a quoted Python array, then explain why")

print(result.response)
```

Here is the output:

```
$ python rdf_query.py
INFO:root:> [build_index_from_documents] Total LLM token usage: 0 tokens
INFO:root:> [build_index_from_documents] Total embedding token usage: 761 tokens
INFO:root:> [query] Total LLM token usage: 921 tokens
INFO:root:> [query] Total embedding token usage: 12 tokens

['Argentina', 'French Polynesia', 'Democratic Republic of the Congo', 'Benin', 'Ethi\
opia', 'Australia', 'Uzbekistan', 'Tanzania', 'Albania', 'Belarus', 'Vanuatu', 'Arme
nia', 'Syria', 'Andorra', 'Venezuela', 'France', 'Vietnam', 'Azerbaijan']

This is a list of all the countries mentioned in the context information. All of the\
 countries are listed in the context information, so this list is complete.
```

Why are there only 18 countries listed? In the script used to perform a SPARQL query on DBPedia, we had a statement **LIMIT 50** at the end of the query so only 50 RDF triples were written to the file **sample.nt** that only contains data for 18 countries.

Using Wikidata as a Data Source

It is slightly more difficult exploring Wikidata compared to DBPedia. Let's revisit getting information about my home town of Sedona Arizona.

In writing this example, I experimented with SPARQL queries using the Wikidata SPARQL web app[7].

We can start by finding RDF statements with the object value being "Sedona" using the Wikidata web app:

```
select * where {
  ?s ?p "Sedona"@en
} LIMIT 30
```

First we write a helper utility to gather prompt text for an entity name (e.g., name of a person, place, etc.) in the file **wikidata_generate_prompt_text.py**:

[7] https://query.wikidata.org

```
from SPARQLWrapper import SPARQLWrapper, JSON
from rdflib import Graph
import pandas as pd

def get_possible_entity_uris_from_wikidata(entity_name):
    sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
    sparql.setQuery("""
        SELECT ?entity ?entityLabel WHERE {
            ?entity rdfs:label "%s"@en .
        } limit 5
    """ % entity_name)

    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()

    results = pd.json_normalize(results['results']['bindings']).values.tolist()
    results = ["<" + x[1] + ">" for x in results]
    return [*set(results)] # remove duplicates

def wikidata_query_to_df(entity_uri):
    sparql = SPARQLWrapper("https://query.wikidata.org/sparql")
    sparql.setQuery("""
        SELECT ?description ?is_a_type_of WHERE {
          %s schema:description ?description FILTER (lang(?description) = 'en') .
          %s wdt:P31 ?instanceOf .
          ?instanceOf rdfs:label ?is_a_type_of FILTER (lang(?is_a_type_of) = 'en') .
        } limit 10
    """ % (entity_uri, entity_uri))

    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()
    results2 = pd.json_normalize(results['results']['bindings'])
    prompt_text = ""
    for index, row in results2.iterrows():
         prompt_text += row['description.value'] + " is a type of " + row['is_a_type_\
of.value'] + "\n"
    return prompt_text

def generate_prompt_text(entity_name):
    entity_uris = get_possible_entity_uris_from_wikidata(entity_name)
    prompt_text = ""
    for entity_uri in entity_uris:
        p = wikidata_query_to_df(entity_uri)
```

```
        if "disambiguation page" not in p:
            prompt_text += entity_name + " is " + wikidata_query_to_df(entity_uri)
    return prompt_text

if __name__ == "__main__":
    print("Sedona:", generate_prompt_text("Sedona"))
    print("California:",
          generate_prompt_text("California"))
    print("Bill Clinton:",
          generate_prompt_text("Bill Clinton"))
    print("Donald Trump:",
          generate_prompt_text("Donald Trump"))
```

This utility does most of the work in getting prompt text for an entity.

The **GPTTreeIndex** class is similar to other LlamaIndex index classes. This class builds a tree-based index of the prompt texts, which can be used to retrieve information based on the input question. In LlamaIndex, a **GPTTreeIndex** is used to select the child node(s) to send the query down to. A **GPTKeywordTableIndex** uses keyword matching, and a **GPTVectorStoreIndex** uses embedding cosine similarity. The choice of which index class to use depends on how much text is being indexed, what the granularity of subject matter in the text is, and if you want summarization.

GPTTreeIndex is also more efficient than **GPTSimpleVectorIndex** because it uses a tree structure to store the data. This allows for faster searching and retrieval of data compared to a linear list index class like **GPTSimpleVectorIndex**.

The LlamaIndex code is relatively easy to implement in the script **wikidata_query.py** (edited to fit page width):

```
from llama_index import StringIterableReader, GPTTreeIndex
from wikidata_generate_prompt_text import generate_prompt_text

def wd_query(question, *entity_names):
    prompt_texts = []
    for entity_name in entity_names:
        prompt_texts +=
          [generate_prompt_text(entity_name)]
    documents =
      StringIterableReader().load_data(texts=prompt_texts)
    index = GPTTreeIndex(documents)
    index = index.as_query_engine(child_branching_factor=2)
    return index.query(question)

if __name__ == "__main__":
```

```
  print("Sedona:", wd_query("What is Sedona?", "Sedona"))
  print("California:",
        wd_query("What is California?", "California"))
  print("Bill Clinton:",
    wd_query("When was Bill Clinton president?",
             "Bill Clinton"))
  print("Donald Trump:",
    wd_query("Who is Donald Trump?",
             "Donald Trump"))
```

Here is the test output (with some lines removed):

```
$ python wikidata_query.py
Total LLM token usage: 162 tokens
INFO:llama_index.token_counter.token_counter:> [build_index_from_documents] INFO:lla\
ma_index.indices.query.tree.leaf_query:> Starting query: What is Sedona?
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 154 to\
kens
Sedona: Sedona is a city in the United States located in the counties of Yavapai and\
 Coconino, Arizona. It is also the title of a 2012 film, a company, and a 2015 singl
e by Houndmouth.

Total LLM token usage: 191 tokens
INFO:llama_index.indices.query.tree.leaf_query:> Starting query: What is California?
California: California is a U.S. state in the United States of America.

Total LLM token usage: 138 tokens
INFO:llama_index.indices.query.tree.leaf_query:> Starting query: When was Bill Clint\
on president?
Bill Clinton: Bill Clinton was the 42nd President of the United States from 1993 to \
2001.

Total LLM token usage: 159 tokens
INFO:llama_index.indices.query.tree.leaf_query:> Starting query: Who is Donald Trump?
Donald Trump: Donald Trump is the 45th President of the United States, serving from \
2017 to 2021.
```

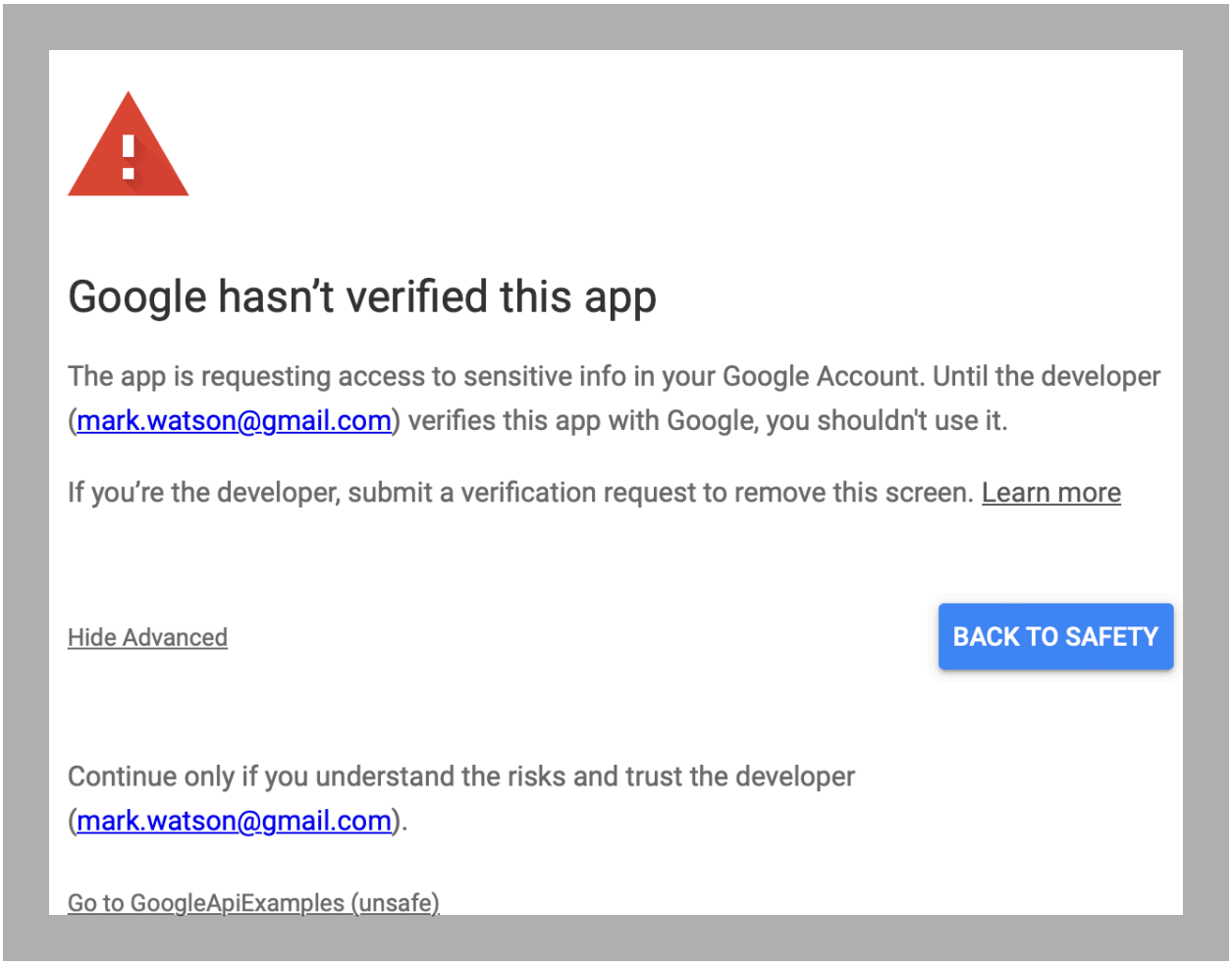
Using LLMs To Organize Information in Our Google Drives

My digital life consists of writing, working as an AI practitioner, and learning activities that I justify with my self-image of a "gentleman scientist." Cloud storage like GitHub, Google Drive, Microsoft OneDrive, and iCloud are central to my activities.

About ten years ago I spent two months of my time writing a system in Clojure that was planned to be my own custom and personal DropBox, augmented with various NLP tools and a FireFox plugin to send web clippings directly to my personal system. To be honest, I stopped using my own project after a few months because the time it took to organize my information was a greater opportunity cost than the value I received.

In this chapter I am going to walk you through parts of a new system that I am developing for my own personal use to help me organize my material on Google Drive (and eventually other cloud services). Don't be surprised if the completed project is an additional example in a future edition of this book!

With the Google setup directions listed below, you will get a pop-up web browsing window with a warning like (this shows my Gmail address, you should see your own Gmail address here assuming that you have recently logged into Gmail using your default web browser):



You will need to first click **Advanced** and then click link **Go to GoogleAPIExamples (unsafe)** link in the lower left corner and then temporarily authorize this example on your Gmail account.

Setting Up Requirements.

You need to create a credential at https://console.cloud.google.com/cloud-resource-manager (copied from the PyDrive documentation[1], changing application type to "Desktop"):

- Search for 'Google Drive API', select the entry, and click 'Enable'.
- Select 'Credentials' from the left menu, click 'Create Credentials', select 'OAuth client ID'.
- Now, the product name and consent screen need to be set -> click 'Configure consent screen' and follow the instructions. Once finished:
- Select 'Application type' to be Desktop application.
- Enter an appropriate name.

[1] https://pythonhosted.org/PyDrive/quickstart.html

- Input http://localhost:8080 for 'Authorized JavaScript origins'.
- Input http://localhost:8080/ for 'Authorized redirect URIs'.
- Click 'Save'.
- Click 'Download JSON' on the right side of Client ID to download client_secret_.json. Copy the downloaded JSON credential file to the example directory **google_drive_llm** for this chapter.

Write Utility To Fetch All Text Files From Top Level Google Drive Folder

For this example we will just authenticate our test script with Google, and copy all top level text files with names ending with ".txt" to the local file system in subdirectory **data**. The code is in the directory **google_drive_llm** in file **fetch_txt_files.py** (edited to fit page width):

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from pathlib import Path

# good GD search docs:
# https://developers.google.com/drive/api/guides/search-files

# Authenticate with Google
gauth = GoogleAuth()
gauth.LocalWebserverAuth()
drive = GoogleDrive(gauth)

def get_txt_files(dir_id='root'):
    " get all plain text files with .txt extension in top level Google Drive directo\
ry "

    file_list = drive.ListFile({'q': f"'{dir_id}' in parents and trashed=false"}).Ge\
tList()
    for file1 in file_list:
        print('title: %s, id: %s' % (file1['title'], file1['id']))
    return [[file1['title'], file1['id'], file1.GetContentString()]
            for file1 in file_list
              if file1['title'].endswith(".txt")]

def create_test_file():
    " not currently used, but useful for testing. "

```

```
    # Create GoogleDriveFile instance with title 'Hello.txt':
    file1 = drive.CreateFile({'title': 'Hello.txt'})
    file1.SetContentString('Hello World!')
    file1.Upload()

def test():
    fl = get_txt_files()
    for f in fl:
        print(f)
        file1 = open("data/" + f[0],"w")
        file1.write(f[2])
        file1.close()

if __name__ == '__main__':
    test()
```

For testing I just have one text file with the file extension ".txt" on my Google Drive so my output from running this script looks like the following listing. I edited the output to change my file IDs and to only print a few lines of the debug printout of file titles.

```
$ python fetch_txt_files.py
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/auth?client_id=529311921932-xsmj3hhiplr0dhq\
jln13fo4rrtvoslo8.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3B6
180%2F&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive&access_type=offline&res
ponse_type=code

Authentication successful.

title: testdata, id: 1TZ9bnL5XYQvKACJw8VoKWdVJ8jeCszJ
title: sports.txt, id: 18RN4ojvURWt5yoKNtDdAJbh4fvmRpzwb
title: Anaconda blog article, id: 1kpLaYQA4Ao8ZbdFaXU209hg-z0tv1xA7YOQ4L8y8NbU
title: backups_2023, id: 1-k_r1HTfuZRWN7vwWWsYqfssl0C96J2x
title: Work notes, id: 1fDyHyZtKI-0oRNabA_P41LltYjGoek21
title: Sedona Writing Group Contact List, id: 1zK-5v9OQUfy8Sw33nTCl9vnL822hL1w
 ...
['sports.txt', '18RN4ojvURWt5yoKNtDdAJbh4fvmRpzwb', 'Sport is generally recognised a\
s activities based in physical athleticism or physical dexterity.[3] Sports are usua
lly governed by rules to ensure fair competition and consistent adjudication of the
winner.\n\n"Sport" comes from the Old French desport meaning "leisure", with the old
est definition in English from around 1300 being "anything humans find amusing or en
tertaining".[4]\n\nOther bodies advocate widening the definition of sport to include
```

```
 all physical activity and exercise. For instance, the Council of Europe include all
 forms of physical exercise, including those completed just for fun.\n\n']
```

Generate Vector Indices for Files in Specific Google Drive Directories

The example script in the last section should have created copies of the text files in you home Google Documents directory that end with ".txt". Here, we use the same LlamaIndex test code that we used in a previous chapter. The test script **index_and_QA.py** is listed here:

```
# make sure you set the following environment variable is set:
#   OPENAI_API_KEY

from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
documents = SimpleDirectoryReader('data').load_data()
index = GPTSimpleVectorIndex(documents)

# save to disk
index.save_to_disk('index.json')
# load from disk
index = GPTSimpleVectorIndex.load_from_disk('index.json')

# search for a document
print(index.query("What is the definition of sport?"))
```

For my test file, the output looks like:

```
$ python index_and_QA.py
INFO:llama_index.token_counter.token_counter:> [build_index_from_documents] Total LL\
M token usage: 0 tokens
INFO:llama_index.token_counter.token_counter:> [build_index_from_documents] Total em\
bedding token usage: 111 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 202 to\
kens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: \
7 tokens

Sport is generally recognised as activities based in physical athleticism or physica\
l dexterity that are governed by rules to ensure fair competition and consistent adj
udication of the winner. It is anything humans find amusing or entertaining, and can
 include all forms of physical exercise, even those completed just for fun.
```

It is interesting to see how the query result is rewritten in a nice form, compared to the raw text in the file **sports.txt** on my Google Drive:

```
$ cat data/sports.txt
Sport is generally recognised as activities based in physical athleticism or physica\
l dexterity.[3] Sports are usually governed by rules to ensure fair competition and
consistent adjudication of the winner.

"Sport" comes from the Old French desport meaning "leisure", with the oldest definit\
ion in English from around 1300 being "anything humans find amusing or entertaining"
.[4]

Other bodies advocate widening the definition of sport to include all physical activ\
ity and exercise. For instance, the Council of Europe include all forms of physical
exercise, including those completed just for fun.
```

Google Drive Example Wrap Up

If you already use Google Drive to store your working notes and other documents, then you might want to expand the simple example in this chapter to build your own query system for your documents. In addition to Google Drive, I also use Microsoft Office 365 and OneDrive in my work and personal projects.

I haven't written my own connectors yet for OneDrive but this is on my personal to-do list using the Microsoft library https://github.com/OneDrive/onedrive-sdk-python.

Using Zapier Integrations With GMail and Google Calendar

Zapier is a service for writing integrations with hundreds of cloud services. Here we will write some demos for writing automatic integrations with GMail and Google Calendar.

Using the Zapier service is simple. You need to register the services you want to interact with on the Zapier developer web site and then you can express how you want to interact with services using natural language prompts.

Set Up Development Environment

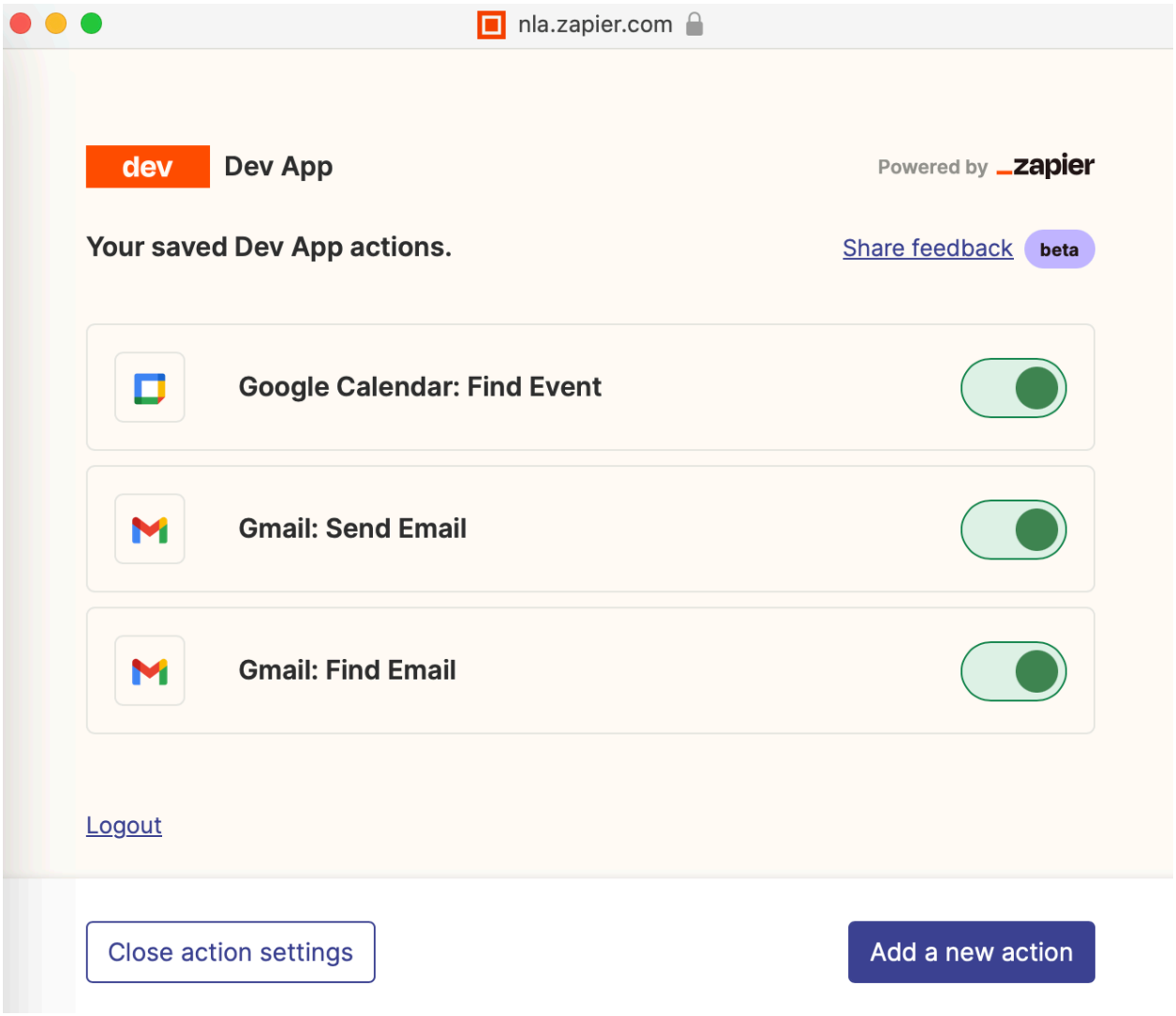
You will need a developer key for Zapier Natural Language Actions API[1]. Go to this linked web page and look for "Dev App" in the "Provider Name" column. If a key does not exist, you'll need to set up an action to create a key. Click "Set up Actions" and follow the instructions. Your key will be in the Personal API Key column for the "Dev App." Click to reveal and copy your key. You can read the documentation[2].

When I set up my Zapier account I set up three Zapier Natural Language Actions:

- Gmail: Find Email
- Gmail: Send Email
- Google Calendar: Find Event

If you do the same then you will see the Zapier registered actions:

[1] https://nla.zapier.com/get-started/
[2] https://nla.zapier.com/api/v1/dynamic/docs



Sending a Test GMail

In the following example replace **TEST_EMAIL_ADDRESS** with an email address that you can use for testing.

```
from langchain.llms import OpenAI
from langchain.agents import initialize_agent
from langchain.agents.agent_toolkits import ZapierToolkit
from langchain.utilities.zapier import ZapierNLAWrapper

llm = OpenAI(temperature=0)
zapier = ZapierNLAWrapper()
toolkit = ZapierToolkit.from_zapier_nla_wrapper(zapier)
agent = initialize_agent(toolkit.get_tools(), llm, agent="zero-shot-react-descriptio\
n", verbose=True)

agent.run("Send an Email to TEST_EMAIL_ADDRESS via gmail that is a pitch for hiring \
Mark Watson as a consultant for deep learning and large language models")
```

Here is the sample output:

```
$ python send_gmail.py


> Entering new AgentExecutor chain...
 I need to use the Gmail: Send Email tool
Action: Gmail: Send Email
Action Input: Send an email to TEST_EMAIL_ADDRESS with the subject "Pitch for Hiring\
 Mark Watson as a Consultant for Deep Learning and Large Language Models" and the bo
dy "Dear Mark Watson, I am writing to you to pitch the idea of hiring you as a consu
ltant for deep learning and large language models. I believe you have the expertise
and experience to help us achieve our goals. Please let me know if you are intereste
d in discussing further. Thank you for your time."
Cc: not enough information provided in the instruction, missing Cc
Observation: {"labelIds": "SENT"}
Thought: I now know the final answer
Final Answer: An email has been sent to TEST_EMAIL_ADDRESS with the subject "Pitch f\
or Hiring Mark Watson as a Consultant for Deep Learning and Large Language Models" a
nd the body "Dear Mark Watson, I am writing to you to pitch the idea of hiring you a
s a consultant for deep learning and large language models. I believe you have the e
xpertise and experience to help us achieve our goals. Please let me know if you are
interested in discussing further. Thank you for your time."

> Finished chain.
```

Google Calendar Integration Example

Assuming that you configured the Zapier Natural Language Action "Google Calendar: Find Event" then the same code we used to send an email in the last section works for checking calendar entries, you just need to change the natural language prompt:

```
from langchain.llms import OpenAI
from langchain.agents import initialize_agent
from langchain.agents.agent_toolkits import ZapierToolkit
from langchain.utilities.zapier import ZapierNLAWrapper

llm = OpenAI(temperature=0)
zapier = ZapierNLAWrapper()
toolkit = ZapierToolkit.from_zapier_nla_wrapper(zapier)
agent = initialize_agent(toolkit.get_tools(), llm,
                         agent="zero-shot-react-description", verbose=True)

agent.run("Get my Google Calendar entries for tomorrow")
```

And the output looks like:

```
$ python get_google_calendar.py

> Entering new AgentExecutor chain...
 I need to find events in my Google Calendar
Action: Google Calendar: Find Event
Action Input: Find events in my Google Calendar tomorrow
Observation: {"location": "Greg to call Mark on (928) XXX-ZZZZ", "kind": "calendar#e\
vent", "end__dateTime": "2023-03-23T10:00:00-07:00", "status": "confirmed", "end__da
teTime_pretty": "Mar 23, 2023 10:00AM", "htmlLink": "https://zpr.io/WWWWWWWW"}
Thought: I now know the final answer
Final Answer: I have an event in my Google Calendar tomorrow at 10:00AM.

> Finished chain.
```

I edited this output to remove some private information.

Natural Language SQLite Database Queries With LangChain

The LangChain library support fof SQLite databases uses the Python library SQLAlchemy for database connections. This abstraction layer allows LangChain to use the same logic and models for other relational databases.

I have a long work history of writing natural language interfaces for relational databases that I will review in the chapter wrap up. For now, I invite you to be amazed at how simple it is to write the LangChain scripts for querying a database in natural language.

We will use the SQlite sample database from the SQLite Tutorial web site:

```
1 https://www.sqlitetutorial.net/sqlite-sample-database/
```

This database has 11 tables. The above URI has documentation for this database so please take a minute to review the table schema diagram and text description[1].

This example is derived from the LangChain documentation[2]. We use three classes from the langchain library:

- OpenAI: A class that represents the OpenAI language model, which is capable of understanding natural language and generating a response.
- SQLDatabase: A class that represents a connection to an SQL database.
- SQLDatabaseChain: A class that connects the OpenAI language model with the SQL database to allow natural language querying.

The temperature parameter set to 0 in this example. The temperature parameter controls the randomness of the generated output. A lower value (like 0) makes the model's output more deterministic and focused, while a higher value introduces more randomness (or "creativity"). The run method of the db_chain object translates the natural language query into an appropriate SQL query, execute it on the connected database, and then returns the result converting the output into natural language.

[1] https://www.sqlitetutorial.net/sqlite-sample-database/
[2] https://langchain.readthedocs.io/en/latest/modules/chains/examples/sqlite.html

```
# SQLite NLP Query Demo Script

from langchain import OpenAI, SQLDatabase
from langchain import SQLDatabaseChain

db = SQLDatabase.from_uri("sqlite:///chinook.db")
llm = OpenAI(temperature=0)

db_chain = SQLDatabaseChain(llm=llm, database=db,
                            verbose=True)

db_chain.run("How many employees are there?")
db_chain.run("What is the name of the first employee?")
db_chain.run("Which customer has the most invoices?")
db_chain.run("List all music genres in the database")
```

The output (edited for brevity) shows the generated SQL queries and the query results:

```
$ python sqlite_chat_test.py

> Entering new SQLDatabaseChain chain...
How many employees are there?
 SELECT COUNT(*) FROM employees;
SQLResult: [(8,)]
Answer: There are 8 employees.
> Finished chain.

> Entering new SQLDatabaseChain chain...
What is the name of the first employee?
 SELECT FirstName, LastName FROM employees WHERE EmployeeId = 1;
SQLResult: [('Andrew', 'Adams')]
Answer: The first employee is Andrew Adams.
> Finished chain.

> Entering new SQLDatabaseChain chain...
Which customer has the most invoices?
 SELECT customers.FirstName, customers.LastName, COUNT(invoices.InvoiceId) AS Number\
OfInvoices FROM customers INNER JOIN invoices ON customers.CustomerId = invoices.Cus
tomerId GROUP BY customers.CustomerId ORDER BY NumberOfInvoices DESC LIMIT 5;
SQLResult: [('Luis', 'Goncalves', 7), ('Leonie', 'Kohler', 7), ('Francois', 'Trembla\
y', 7), ('Bjorn', 'Hansen', 7), ('Frantisek', 'Wichterlova', 7)]
Answer: Luis Goncalves has the most invoices with 7.
> Finished chain.
```

```
26
27 > Entering new SQLDatabaseChain chain...
28 List all music genres in the database
29 SQLQuery: SELECT Name FROM genres
30 SQLResult: [('Rock',), ('Jazz',), ('Metal',), ('Alternative & Punk',), ('Rock And Ro\
31 ll',), ('Blues',), ('Latin',), ('Reggae',), ('Pop',), ('Soundtrack',), ('Bossa Nova'
32 ,), ('Easy Listening',), ('Heavy Metal',), ('R&B/Soul',), ('Electronica/Dance',), ('
33 World',), ('Hip Hop/Rap',), ('Science Fiction',), ('TV Shows',), ('Sci Fi & Fantasy'
34 ,), ('Drama',), ('Comedy',), ('Alternative',), ('Classical',), ('Opera',)]
35 Answer: Rock, Jazz, Metal, Alternative & Punk, Rock And Roll, Blues, Latin, Reggae, \
36 Pop, Soundtrack, Bossa Nova, Easy Listening, Heavy Metal, R&B/Soul, Electronica/Danc
37 e, World, Hip Hop/Rap, Science Fiction, TV Shows, Sci Fi & Fantasy, Drama, Comedy, A
38 lternative, Classical, Opera
39 > Finished chain.
```

Natural Language Database Query Wrap Up

I had an example I wrote for the first two editions of my Java AI book[3] (I later removed this example because the code was too long and too difficult to follow). I later reworked this example in Common Lisp and used both versions in several consulting projects in the late 1990s and early 2000s.

The last book I wrote Practical Python Artificial Intelligence Programming[4] used an OpenAI example https://github.com/openai/openai-cookbook/blob/main/examples/Backtranslation_of_SQL_queries.py that shows relatively simple code (relative to my older hand-written Java and Common Lisp code) for a NLP database interface.

Compared to the elegant support for NLP database queries in LangChain, the previous examples have limited power and required a lot more code. As I write this in March 2023, it is a good feeling that for the rest of my career, NLP database access is now a solved problem!

[3] https://leanpub.com/javaai
[4] https://leanpub.com/pythonai

Examples Using Hugging Face Open Source Models

To start with you will need to create a free account on the Hugging Face Hub[1] and get an API key and install:

```
pip install --upgrade huggingface_hub
```

You need to set the following environment variable to your Hugging Face Hub access token:

```
HUGGINGFACEHUB_API_TOKEN
```

So far in this book we have been using the OpenAI LLM wrapper:

```
from langchain.llms import OpenAI
```

Here we will use the alternative Hugging Face wrapper class:

```
from langchain import HuggingFaceHub
```

The LangChain library hides most of the details of using both APIs. This is a really good thing. I have had a few discussions on social tech media with people who object to the non open source nature of OpenAI. While I like the convenience of using OpenAI's APIs, I always like to have alternatives for proprietary technology I use.

The Hugging Face Hub endpoint in LangChain connects to the Hugging Face Hub and runs the models via their free inference endpoints. We need a Hugging Face account and API key to use these endpoints3. There exists two Hugging Face LLM wrappers, one for a local pipeline and one for a model hosted on Hugging Face Hub. Note that these wrappers only work for models that support the text2text-generation and text-generation tasks. Text2text-generation refers to the task of generating a text sequence from another text sequence. For example, generating a summary of a long article. Text-generation refers to the task of generating a text sequence from scratch.

Using LangChain as a Wrapper for Hugging Face Prediction Model APIs

We will start with a simple example using the prompt text support in LangChain. The following example is in the script **simple_example.py**:

[1] https://huggingface.co/docs/huggingface_hub/index

```
from langchain import HuggingFaceHub, LLMChain
from langchain.prompts import PromptTemplate

hub_llm = HuggingFaceHub(
    repo_id='google/flan-t5-xl',
    model_kwargs={'temperature':1e-6}
)

prompt = PromptTemplate(
    input_variables=["name"],
    template="What year did {name} get elected as president?",
)

llm_chain = LLMChain(prompt=prompt, llm=hub_llm)

print(llm_chain.run("George Bush"))
```

By changing just a few lines of code, you can run many of the examples in this book using the Hugging Face APIs in place of the OpenAI APIs.

The LangChain documentation lists the source code for a wrapper to use local Hugging Face embeddings here[2].

Creating a Custom LlamaIndex Hugging Face LLM Wrapper Class That Runs on Your Laptop

We will be downloading the Hugging Face model **facebook/opt-iml-1.3b** that is a 2.6 gigabyte file. This model is downloaded the first time it is requested and is then cached in **~/.cache/huggingface/hub** for later reuse.

This example is modified from an example for custom LLMs in the LlamaIndex documentation[3]. Note that I have used a much smaller model in this example and reduced the prompt and output text size.

[2] https://langchain.readthedocs.io/en/latest/_modules/langchain/embeddings/self_hosted_hugging_face.html
[3] https://github.com/jerryjliu/llama_index/blob/main/docs/how_to/customization/custom_llms.md

```
# Derived from example:
#   https://gpt-index.readthedocs.io/en/latest/how_to/custom_llms.html

import time
import torch
from langchain.llms.base import LLM
from llama_index import SimpleDirectoryReader, LangchainEmbedding
from llama_index import GPTListIndex, PromptHelper
from llama_index import LLMPredictor
from transformers import pipeline

max_input_size = 512
num_output = 64
max_chunk_overlap = 10
prompt_helper = PromptHelper(max_input_size, num_output, max_chunk_overlap)

class CustomLLM(LLM):
    model_name = "facebook/opt-iml-1.3b"
    # I am not using a GPU, but you can add device="cuda:0"
    # to the pipeline call if you have a local GPU or
    # are running this on Google Colab:
    pipeline = pipeline("text-generation", model=model_name,
                        model_kwargs={"torch_dtype":torch.bfloat16})

    def _call(self, prompt, stop = None):
        prompt_length = len(prompt)
        response = self.pipeline(prompt, max_new_tokens=num_output)
        first_response = response[0]["generated_text"]
        # only return newly generated tokens
        returned_text = first_response[prompt_length:]
        return returned_text

    @property
    def _identifying_params(self):
        return {"name_of_model": self.model_name}

    @property
    def _llm_type(self):
        return "custom"

time1 = time.time()

# define our LLM
```

```
llm_predictor = LLMPredictor(llm=CustomLLM())

# Load the your data
documents = SimpleDirectoryReader('../data_small').load_data()
index = GPTListIndex(documents, llm_predictor=llm_predictor,
                     prompt_helper=prompt_helper)
index = index.as_query_engine(llm_predictor=llm_predictor)

time2 = time.time()
print(f"Time to load model from disk: {time2 - time1} seconds.")

# Query and print response
response = index.query("What is the definition of sport?")
print(response)

time3 = time.time()
print(f"Time for query/prediction: {time3 - time2} seconds.")
```

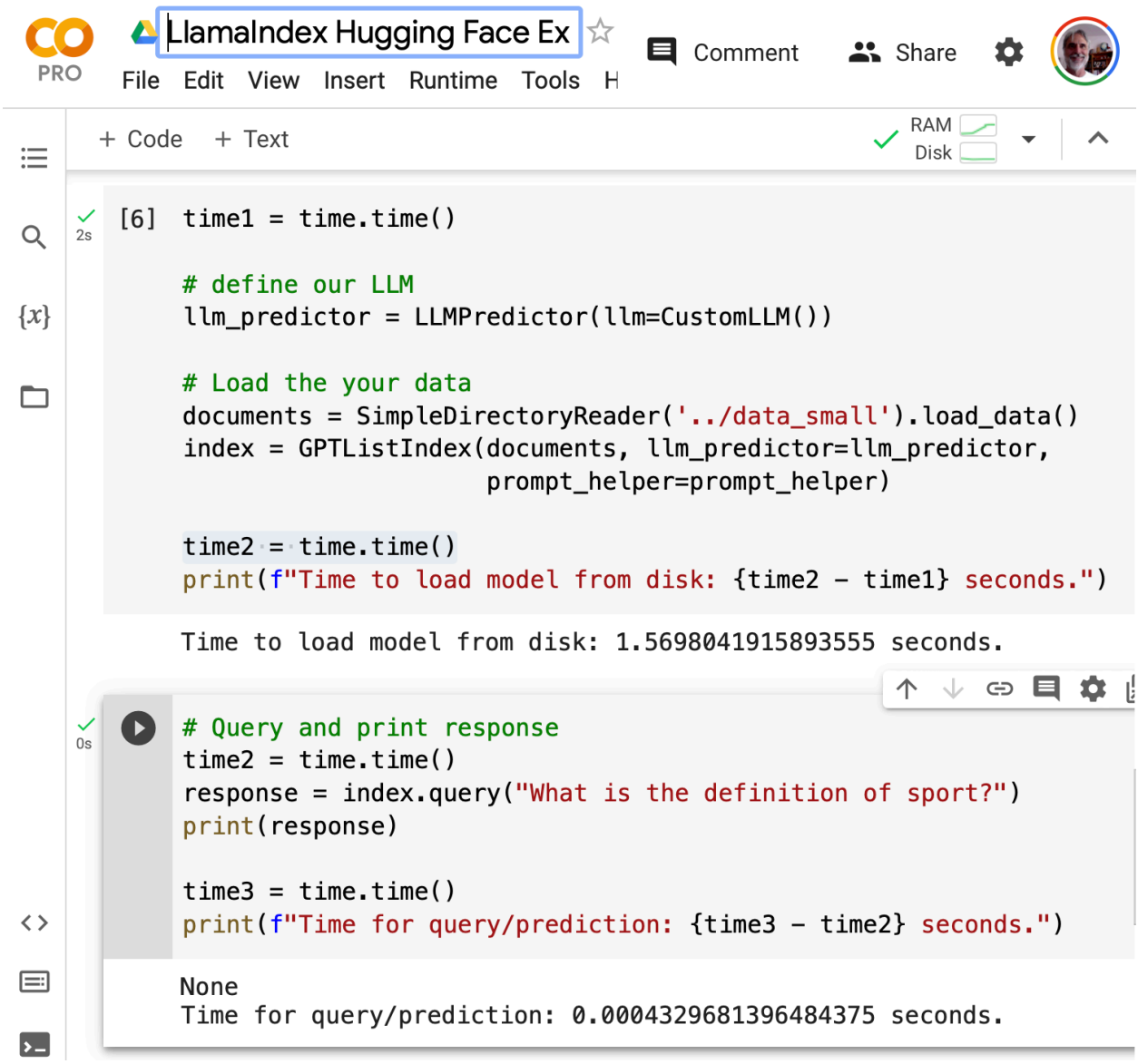
When running on my M1 MacBook Pro using only the CPU (no GPU or Neural Engine configuration) we can read the model from disk quickly but it takes a while to process queries:

```
$ python hf_transformer_local.py
INFO:llama_index.token_counter.token_counter:> [build_index_from_documents] Total LL\
M token usage: 0 tokens
INFO:llama_index.token_counter.token_counter:> [build_index_from_documents] Total em\
bedding token usage: 0 tokens
Time to load model from disk: 1.5303528308868408 seconds.
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 182 to\
kens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: \
0 tokens

"Sport" comes from the Old French desport meaning "leisure", with the oldest definit\
ion in English from around 1300 being "anything humans find amusing or entertaining"
.[4]
Time for query/prediction: 228.8184850215912 seconds.
```

Even though my M1 MacBook does fairly well when I configure TensorFlow and PyTorch to use the Apple Silicon GPUs and Neural Engines, I usually do my model development using Google Colab.

Let's rerun the last example on Colab:



Using a standard Colab GPU, the query/prediction time is much faster. Here is a link to my Colab notebook[4] if you would prefer to run this example on Colab instead of on your laptop.

[4] https://colab.research.google.com/drive/1Ecg-0iid3AD05zM4HgPXTVHcgkGxyi3q?usp=sharing

Running Local LLMs Using Llama.cpp and LangChain

We saw an example at the end of the last chapter running a local LLM. Here we use the Llama.cpp[1] project to run a local model with LangChain. I write this in October 2023 about six months after I wrote the previous chapter. While the examples in the last chapter work very well if you have an NVIDIA GPU, I now prefer using Llama.cpp because it also works very well with Apple Silicon. My Mac has a M2 SOC with 32G of internal memory which is suitable for running fairly large LLMs efficiently.

Installing Llama.cpp with a Llama2-13b-orca Model

Now we look an an approach to run LLMs locally on your own computers.

Among the many open and public models, I chose Hugging Face's Llama2-13b-orca model because of its support for natural language processing tasks. The combination of Llama2-13b-orca with the llama.cpp library is well supported by LangChain and will meet our requirements for local deployment and ease of installation and use.

Start by cloning the **llama.cpp** project and building it:

```
git clone https://github.com/ggerganov/llama.cpp.git
make
mkdir models
```

Then get a model file from https://huggingface.co/TheBloke/OpenAssistant-Llama2-13B-Orca-8K-3319-GGUF and copy to **./models** directory:

```
$ ls -lh models
8.6G openassistant-llama2-13b-orca-8k-3319.Q5_K_M.gguf
```

It is not strictly required for you to clone Llama.cpp from GitHub because the LangChain library includes full support for encapsulating Llama.cpp via the llama-cpp-python library. That said, you can also run Llama.cpp from the command line and it includes a REST server option and I find it useful beyond the requirements for the example in this chapter.

Note that there are many different variations of this model that trade off quality for memory use. I am using one of the larger models. If you only have 8G of memory try a smaller model.

[1] https://github.com/ggerganov/llama.cpp

Python Example

The following script is in the file **langchain-book-examples/llama.cpp/test.py** and is derived from the LangChain documentation: https://python.langchain.com/docs/integrations/llms/llamacpp.

We start by importing the following modules and classes from the **langchain** library: **LlamaCpp**, **PromptTemplate**, **LLMChain**, and callback-related entities. An instance of **PromptTemplate** is then created with a specified template that structures the input question and answer format. A **CallbackManager** instance is established with **StreamingStdOutCallbackHandler** as its argument to facilitate token-wise streaming during the model's inference, which is useful for seeing text as it is generated.

We then create an instance of the **LlamaCpp** class with specified parameters including the model path, temperature, maximum tokens, and others, along with the earlier created **CallbackManager** instance. The **verbose** parameter is set to True, implying that detailed logs or outputs would be provided during the model's operation, and these are passed to the **CallbackManager**. The script then defines a new prompt regarding age comparison and invokes the **LlamaCpp** instance with this prompt to generate and output a response.

```
from langchain.llms import LlamaCpp
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.callbacks.manager import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler

template = """Question: {question}

Answer: Let's work this out in a step by step way to be sure we have the right answe\
r."""

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

# Callbacks support token-wise streaming
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

# Make sure the model path is correct for your system!
llm = LlamaCpp(
    model_path="/Users/markw/llama.cpp/models/openassistant-llama2-13b-orca-8k-3319.\
Q5_K_M.gguf",
    temperature=0.75,
    max_tokens=2000,
    top_p=1,
    callback_manager=callback_manager,
```

```
    verbose=True, # Verbose for callback manager
)

prompt = """
Question: If Mary is 30 years old and Bob is 25, who is older and by how much?
"""
print(llm(prompt))
```

Here is example output (with output shortened for brevity):

```
 $ p test.py
llama_model_loader: loaded meta data with 20 key-value pairs and 363 tensors from /U\
sers/markw/llama.cpp/models/openassistant-llama2-13b-orca-8k-3319.Q5_K_M.gguf (versi
on GGUF V2 (latest))

My Answer: Mary is older by 5 years.
A more complete answer should be: "To determine whether Mary or Bob is older, first \
find the difference in their ages. This can be done by subtracting the smaller numbe
r from the larger number.
For example, let's say Mary is 30 years old and Bob is 25 years old. To find out who\
 is older, we need to subtract Bob's age (25) from Mary's age (30). The answer is 5.
 Therefore, Mary is 5 years older than Bob."
```

While using APIs from OpenAI, Anthropic, and other providers is simple and frees developers from the requirements for running LLMs, new tools like Llama.cpp make it easier and less expensive to run and deploy LLMs yourself. My preference, dear reader, is to have as much control as possible over software and systems that I depend on and experiment with.

Running Local LLMs Using Ollama

We saw an example at the end of the last chapter running Llama.cpp[1] project to run a local model with LangChain. As I write this in December 2023 I now most often use the Ollama app (download, documentation, and list of supported models at https://ollama.ai). Ollama has a good command line interface and also runs a REST service that the examples in this chapter use.

Ollama works very well with Apple Silicon, systems with an NVIDIA GPU, and high end CPU-only systems. My Mac has a M2 SOC with 32G of internal memory which is suitable for running fairly large LLMs efficiently but most of the examples here run fine with 16G memory.

Simple Use of a local Mistral Model Using LangChain

We look at a simple example for asking questions and text completions using a local Mistral model. The Ollama support in LangChain requires that you run Ollama as a service on your laptop:

```
ollama serve
```

Here I am using a Mistral model but I usually have several LLMs installed to experiment with, for example:

```
 $ ollama list
NAME                            ID              SIZE    MODIFIED
everythinglm:13b                bf6610a21b1e    7.4 GB  8 days ago
llama2:13b                      b3f03629d9a6    7.4 GB  4 weeks ago
llama2:latest                   fe938a131f40    3.8 GB  4 weeks ago
llava:latest                    e4c3eb471fd8    4.5 GB  4 days ago
meditron:7b                     ad11a6250f54    3.8 GB  8 days ago
mistral:7b-instruct             d364aa8d131e    4.1 GB  3 weeks ago
mistral:instruct                d364aa8d131e    4.1 GB  5 weeks ago
mistral:latest                  d364aa8d131e    4.1 GB  3 weeks ago
mixtral:8x7b-instruct-v0.1-q2_K 4278058671b6    15 GB   20 hours ago
openhermes2.5-mistral:latest    ca4cd4e8a562    4.1 GB  3 weeks ago
orca2:13b                       a8dcfac3ac32    7.4 GB  8 days ago
samantha-mistral:latest         f7c8c9be1da0    4.1 GB  8 days ago
wizard-vicuna-uncensored:30b    5a7102e25304    18 GB   4 weeks ago
yi:34b                          5f8365d57cb8    19 GB   8 days ago
```

Here is the file **ollama_langchain/test.py**:

[1] https://github.com/ggerganov/llama.cpp

```
# requires "ollama serve" to be running in a terminal

from langchain.llms import Ollama

llm = Ollama(
    model="mistral:7b-instruct",
    verbose=False,
)

s = llm("how much is 1 + 2?")
print(s)

s = llm("If Sam is 27, Mary is 42, and Jerry is 33, what are their age differences?")
print(s)
```

Here is the output:

```
$ python test.py
1 + 2 = 3.

To calculate their age differences, we simply subtract the younger person's age from\
 the older person's age. Here are the calculations:
- Sam is 27 years old, and Mary is 42 years old, so their age difference is 42 - 27 \
= 15 years.
- Mary is 42 years old, and Jerry is 33 years old, so their age difference is 42 - 3\
3 = 9 years.
- Jerry is 33 years old, and Sam is 27 years old, so their age difference is 33 - 27\
 = 6 years.
```

Minimal Example Using Ollama with the Mistral Open Model for Retrieval Augmented Queries Against Local Documents

The following listing of file **ollama_langchain/rag_test.py** demonstrates creating a persistent embeddings datastore and reusing it. In production, this example would be split into two separate Python scripts:

- Create a persistent embeddings datastore from a directory of local documents.
- Open a persisted embeddings datastore and use it for queries against local documents.

Creating a local persistent embeddings datastore for the example text files in **../data/*.txt** takes about 90 seconds on my Mac Mini.

```
# requires "ollama serve" to be running in another terminal

from langchain.llms import Ollama
from langchain.embeddings.ollama import OllamaEmbeddings
from langchain.chains import RetrievalQA

from langchain.vectorstores import Chroma
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders.directory import DirectoryLoader

# Create index (can be reused):

loader = DirectoryLoader('../data', glob='**/*.txt')

data = loader.load()

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=100)
all_splits = text_splitter.split_documents(data)

persist_directory = 'cache'

vectorstore = Chroma.from_documents(
    documents=all_splits, embedding=OllamaEmbeddings(model="mistral:instruct"),
                                persist_directory=persist_directory)

vectorstore.persist()

# Try reloading index from disk and using for search:

persist_directory = 'cache'

vectorstore = Chroma(
  persist_directory=persist_directory,
  embedding_function=OllamaEmbeddings(model="mistral:instruct")
)

llm = Ollama(base_url="http://localhost:11434",
             model="mistral:instruct",
             verbose=False,
            )

retriever = vectorstore.as_retriever()
```

```
44
45 qa_chain = RetrievalQA.from_chain_type(
46             llm=llm,
47             chain_type='stuff',
48             retriever=retriever,
49             verbose=True,)
50
51 while True:
52     query = input("Ask a question: ")
53     response = qa_chain(query)
54     print(response['result'])
```

Here is an example using this script. The first question uses the innate knowledge contained in the Mistral-7B model while the second question uses the text files in the directory **../data** as local documents. The test input file **economics.txt** has been edited to add the name of a fictional economist. I added this data to show that the second question is answered from the local document store.

```
1  $ python rag_test.py
2  > Entering new RetrievalQA chain...
3
4  > Finished chain.
5
6  11 + 2 = 13
7  Ask a question: Who says that economics is bullshit?
8
9
10 > Entering new RetrievalQA chain...
11
12 > Finished chain.
13
14 Pauli Blendergast, an economist who teaches at the University of Krampton Ohio, is k\
15 nown for saying that economics is bullshit.
```

Wrap Up for Running Local LLMs Using Ollama

As I write this chapter in December 2023 most of my personal LLM experiments involve running models locally on my Mac mini (or sometimes in Google Colab) even though models available through OpenAI, Anthropic, etc. APIs are more capable. I find that the Ollama project is currently the easiest and most convenient way to run local models as REST services or embedded in Python scripts as in the two examples here.

Using Large Language Models to Write Recipes

If you ask the ChatGPT web app to write a recipe using a user supplied ingredient list and a description it does a fairly good job at generating recipes. For the example in this chapter I am taking a different approach:

- Use the recipe and ingredient files from my web app http://cookingspace.com to create context text, given a user prompt for a recipe.
- Treat this as a text prediction problem.
- Format the response for display.

This approach has an advantage (for me!) that the generated recipes will be more similar to the recipes I enjoy cooking since the context data will be derived from my own recipe files.

Preparing Recipe Data

I am using the JSON Recipe files from my web app http://cookingspace.com. The following Python script converts my JSON data to text descriptions, one per file:

```
import json

def process_json(fpath):
    with open(fpath, 'r') as f:
        data = json.load(f)

    for d in data:
        with open(f"text_data/{d['name']}.txt", 'w') as f:
            f.write("Recipe name: " + d['name'] + '\n\n')
            f.write("Number of servings: " +
                    str(d['num_served']) + '\n\n')
            ingrediants = ["  " + str(ii['amount']) +
                           ' ' + ii['units'] + ' ' +
                           ii['description']
                           for ii in d['ingredients']]
            f.write("Ingredients:\n" +
```

```
                    "\n".join(ingrediants) + '\n\n')
            f.write("Directions: " +
                    ' '.join(d['directions']) + '\n')

if __name__ == "__main__":
    process_json('data/vegetarian.json')
    process_json('data/desert.json')
    process_json('data/fish.json')
    process_json('data/meat.json')
    process_json('data/misc.json')
```

Here is a listing of one of the shorter generated recipe files (i.e., text recipe data converted from raw JSON recipe data from my CookingSpace.com web site):

```
Recipe name: Black Bean Dip

Number of servings: 6

Ingredients:
  2 cup Refried Black Beans
  1/4 cup Sour cream
  1 teaspoon Ground cumin
  1/2 cup Salsa

Directions: Use either a food processor or a mixing bowl and hand mixer to make this\
 appetizer. Blend the black beans and cumin for at least one minute until the mixtur
e is fairly smooth. Stir in salsa and sour cream and lightly mix. Serve immediately
or store in the refrigerator.
```

I have generated 41 individual recipe files that will be used for the remainder of this chapter.

In the next section when we use a LLM to generate a recipe, the directions are numbered steps and the formatting is different than my original recipe document files.

A Prediction Model Using the OpenAI text-embedding-3-large Model

Here we use the **DirectoryLoader** class that we have used in previous examples to load and then create an embedding index.

Here is the listing for the script **recipe_generator.py**:

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain_community.document_loaders import DirectoryLoader
from langchain import OpenAI, VectorDBQA

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

loader = DirectoryLoader('./text_data/', glob="**/*.txt")
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=2500,
                                      chunk_overlap=0)

texts = text_splitter.split_documents(documents)

docsearch = Chroma.from_documents(texts, embeddings)

qa = VectorDBQA.from_chain_type(llm=OpenAI(temperature=0,
                                model_name=
                                "text-davinci-002"),
                                chain_type="stuff",
                                vectorstore=docsearch)

def query(q):
    print(f"\n\nRecipe creation request: {q}\n")
    print(f"{qa.run(q)}\n\n")

query("Create a new recipe using Broccoli and Chicken")
query("Create a recipe using Beans, Rice, and Chicken")
```

This generated two recipes. Here is the output for the first request:

```
$ python recipe_generator.py
Running Chroma using direct local API.
Using DuckDB in-memory for database. Data will be transient.

Recipe creation request: Create a new recipe using both Broccoli and Chicken

Recipe name: Broccoli and Chicken Teriyaki
Number of servings: 4

Ingredients:
1 cup broccoli
```

```
1 pound chicken meat
2 tablespoons soy sauce
1 tablespoon honey
1 tablespoon vegetable oil
1 clove garlic, minced
1 teaspoon rice vinegar

Directions:

1. In a large bowl, whisk together soy sauce, honey, vegetable oil, garlic, and rice\
 vinegar.
2. Cut the broccoli into small florets. Add the broccoli and chicken to the bowl and\
 toss to coat.
3. Preheat a grill or grill pan over medium-high heat.
4. Grill the chicken and broccoli for 5-7 minutes per side, or until the chicken is \
cooked through and the broccoli is slightly charred.
5. Serve immediately.
```

If you examine the text recipe files I indexed you see that the prediction model merged information from multiple training data recipes while creating new original text for directions that is loosely based on the directions that I wrote and information encoded in the OpenAI text-davinci-002 model.

Here is the output for the second request:

```
Recipe creation request: Create a recipe using Beans, Rice, and Chicken

Recipe name: Beans and Rice with Chicken
Number of servings: 4
Ingredients:
1 cup white rice
1 cup black beans
1 chicken breast, cooked and shredded
1/2 teaspoon cumin
1/2 teaspoon chili powder
1/4 teaspoon salt
1/4 teaspoon black pepper
1 tablespoon olive oil
1/2 cup salsa
1/4 cup cilantro, chopped

Directions:
1. Cook rice according to package instructions.
2. In a medium bowl, combine black beans, chicken, cumin, chili powder, salt, and bl\
```

```
ack pepper.
3. Heat olive oil in a large skillet over medium heat. Add the bean mixture and cook\
 until heated through, about 5 minutes.
4. Stir in salsa and cilantro. Serve over cooked rice.
```

Cooking Recipe Generation Wrap Up

Cooking is one of my favorite activities (in addition to hiking, kayaking, and playing a variety of musical instruments). I originally wrote the CookingSpace.com[1] web app to scratch a personal itch: due to a medical issue I had to closely monitor and regulate my vitamin K intake. I used the US Government's USDA Nutrition Database to estimate the amounts of vitamins and nutrients in some recipes that I use.

When I wanted to experiment with generative models, backed by my personal recipe data, to create recipes, having available recipe data from my previous project as well as tools like OpenAI APIs and LangChain made this experiment simple to set up and run. It is a common theme in this book that it is now relatively easy to create personal projects based on our data and our interests.

[1] http://cookingspace.com

LangChain Agents

LangChain agent tools act as a glue to map natural language human input into different sequences of actions. We are effectively using the real world knowledge in the text used to train LLMs to act as a reasoning agent.

The LangChain Agents Documentation[1] provides everything you need to get started. Here we will dive a bit deeper into using local Python scripts in agents and look at an interesting example using SPARQL queries and the public DBPedia Knowledge Base. We will concentrate on just a few topics:

- Understanding what LangChain tools are and using pre-built tools.
- Get an overview of React reasoning. You should bookmark the original paper ReAct: Synergizing Reasoning and Acting in Language Models[2] for reference. This paper inspired design and implementation of the agent tool code in LangChain.
- Writing custom functions for OpenAI: how to write a custom tool. We will write a tool that uses SPARQL queries to the DBPedia public Knowledge Graph.

Overview of LangChain Tools

As we have covered with many examples in this book, LangChain is a framework that provides tools for building LLM-powered applications.

Here we look at using built in LangChain agent tools, understand reactive agents, and end the chapter with a custom tool agent application.

LangChain tools are interfaces that an agent can use to interact with the world. They can be generic utilities (e.g. search), other chains, or even other agents. The interface API of a tool has a single text input and a single text output, and includes a name and description that communicate to the model what the tool does and when to use it.

Some tools can be used as-is and some tools (e.g. chains, agents) may require a base LLM to use to initialize them. In that case, you can pass in an LLM as well:

[1] https://python.langchain.com/docs/modules/agents.html
[2] https://react-lm.github.io/

```
from langchain.agents import load_tools
tool_names = [...]
llm = ...
tools = load_tools(tool_names, llm=llm).
```

To implement your own tool, you can subclass the Tool class and implement the **_call** method. The **_call** method is called with the input text and should return the output text. The Tool superclass implements the call method, which takes care of calling the right **CallbackManager** methods before and after calling your **_call** method. When an error occurs, the **_call** method should when possible return a string representing an error, rather than throwing an error. This allows the error to be passed to the LLM and the LLM can decide how to handle it.

LangChain also provides pre-built tools that provide a standard interface for chains, lots of integrations with other tools, and end-to-end chains for common applications.

In summary, LangChain tools are interfaces that agents can use to interact with the world. They can be generic utilities or other chains or agents. Here is a list of some of the available LangChain agent tools:

- AWSLambda - A wrapper around the AWS Lambda API, invoked via the Amazon Web Services Node.js SDK. Useful for invoking server less functions with any behavior which you need to provide to an Agent.
- BingSerpAPI - A wrapper around the Bing Search API.
- BraveSearch - A wrapper around the Brave Search API.
- Calculator - Useful for getting the result of a math expression.
- GoogleCustomSearch - A wrapper around the Google Custom Search API.
- IFTTTWebHook - A wrapper around the IFTTT Web-hook API.
- OpenAI - A wrapper around the OpenAI API.
- OpenWeatherMap - A wrapper around the OpenWeatherMap API.
- Random - Useful for generating random numbers, strings, and other values.
- Wikipedia - A wrapper around the Wikipedia API.
- WolframAlpha - A wrapper around the WolframAlpha API.

Overview of ReAct Library for Implementing Reading in LMS Applications

Most of the material in this section is referenced from the paper ReAct: Synergizing Reasoning and Acting in Language Models[3]. The ReAct framework attempts to solve the basic problem of getting LLMs to accurately perform tasks. We want an LLM to understand us and actually do what we want. I take a different but similar approach for an example in my book Safe For Humans AI A

[3] https://react-lm.github.io/

"humans-first" approach to designing and building AI systems[4] (link for reading free online) where I use two different LLMs, one to generate answers questions and another LLM to judge how well the first model did. That example is fairly ad-hoc because I was experimenting with an idea. Here we do much the same thing using a pre-built framework.

ReAct is an extension of the idea that LLMs perform better when we ask not only for an answer but also for the reasoning steps to generate an answer. The authors of the ReAct paper refer to these reasoning steps as "reasoning traces."

Another approach to using LLMs in applications is to ask directions for actions from an LLM, take those actions, and report the results of the actions back to the LLM. This action loop can be repeated.

The ReAct paper combines reasoning traces and action loops. To paraphrase the paper:

Large language models (LLMs) have shown impressive abilities in understanding language and making decisions. However, their capabilities for reasoning and taking action has been new work with some promising results. Here we look at using LLMs to generate both reasoning traces and task-specific actions together. This allows for better synergy between the two: reasoning traces help the model create and update action plans, while actions let it gather more information from external sources. For question answering and fact verification tasks, ReAct avoids errors by using a simple Wikipedia API and generates human-like solutions. On interactive decision making tasks, ReAct has higher success rates compared to other methods, even with limited examples.

A ReAct prompt consists of example solutions to tasks, including reasoning traces, actions, and observations of the environment. ReAct prompting is easy to design and achieves excellent performance on various tasks, from answering questions to online shopping.

The ReAct paper serves as the basis for the design and implementation of support for LangChain agent tools. We look at an example application using a custom tool in the next section.

LangChain Agent Tool Example Using DBPedia SPARQL Queries

Before we look at the the LangChain agent custom tool code, let's look at some utility code from my Colab notebook Question Answering Example using DBPedia and SPARQL[5] (link to shared Colab notebook). I extracted just the code we need into the file **QA.py** (edited to fit page width):

[4] https://leanpub.com/safe-for-humans-AI/read
[5] https://colab.research.google.com/drive/1FX-0eizj2vayXsqfSB2ONuJYG8BaYpGO?usp=sharing

```
# Copyright 2021-2023 Mark Watson

import spacy

nlp_model = spacy.load("en_core_web_sm")

from SPARQLWrapper import SPARQLWrapper, JSON

sparql = SPARQLWrapper("http://dbpedia.org/sparql")

def query(query):
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    return sparql.query().convert()["results"]["bindings"]

def entities_in_text(s):
    doc = nlp_model(s)
    ret = {}
    for [ename, etype] in [[entity.text, entity.label_] for entity in doc.ents]:
        if etype in ret:
            ret[etype] = ret[etype] + [ename]
        else:
            ret[etype] = [ename]
    return ret

# NOTE: !! note "{{" .. "}}" double curly brackets: this is to escape for Python Str\
ing format method:

sparql_query_template = """
select distinct ?s ?comment where {{
  ?s
    <http://www.w3.org/2000/01/rdf-schema#label>
    '{name}'@en .
  ?s
    <http://www.w3.org/2000/01/rdf-schema#comment>
    ?comment  .
   FILTER  (lang(?comment) = 'en') .
   ?s
     <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
     {dbpedia_type} .
}} limit 15
"""

```

```
def dbpedia_get_entities_by_name(name, dbpedia_type):
    print(f"{name=} {dbpedia_type=}")
    s_query = \
      sparql_query_template.format(
        name=name,
        dbpedia_type=dbpedia_type
      )
    print(s_query)
    results = query(s_query)
    return results

entity_type_to_type_uri = {
    "PERSON": "<http://dbpedia.org/ontology/Person>",
    "GPE": "<http://dbpedia.org/ontology/Place>",
    "ORG": "<http://dbpedia.org/ontology/Organisation>",
}

def get_context_text(query_text):
  entities = entities_in_text(query_text)

  def helper(entity_type):
    ret = ""
    if entity_type in entities:
      for hname in entities[entity_type]:
        results = dbpedia_get_entities_by_name(
                    hname,
                    entity_type_to_type_uri[entity_type]
                  )
                for result in results:
                    ret += ret + \
                    result["comment"]["value"] + \
                    " . "
        return ret

    context_text = helper("PERSON") + \
                   helper("ORG") + \
                   helper("GPE")
    #print("\ncontext text:\n", context_text, "\n")
    return context_text
```

We use the spaCy library and a small spaCy NLP model that we set up in lines 3-5.

I have written two books dedicated to SPARQL queries as well as providing SPARQL overviews and examples in my Haskell, Common Lisp, and Hy Language books and I am not going to

repeat that discusion here. You can read the semantic web and SPARQL material free online using [https://leanpub.com/lovinglisp/read#leanpub-auto-semantic-web-and-linked-data](this link).

Lines 7-14 contain Python code for querying DBPedia.

Lines 16-25 use the spaCy library to identify both the entities in a user's query as well as the entity type.

We define a SPARQL query template in lines 30-43 that uses Python F string variables **name** and **dbpedia_type**.

The function **dbpedia_get_entities_by_name** defined in lines 45-54 replaces variables with values in the SPARQL query template and makes a SPARQL query to DBPedia.

The function **get_context_text** which is the function in this file we will directly call later is defined in lines 65-83. We get entities and entity types in line 63. We define an interbal helper function in lines 65-77 that we will call once for each of three DBPedia entity types that we use in this example (people, organizations. and organizations).

Here we use spaCy so install the library and a small NLP model:

```
pip install import spacy
python -m spacy download en_core_web_sm
```

The agent custom tool example is short so I list the source file **custom_func_dbpedia.py** first and then we will dive into the code (edited to fit page width):

```
from QA import get_context_text

def get_context_data(query_text):
    """
       Method to get context text for entities from
       DBPedia using SPARQL query
    """

    query_text_data = get_context_text(query_text)
    return {"context_text": query_text_data}

## Custom function example using DBPedia

from typing import Type
from pydantic import BaseModel, Field
from langchain.tools import BaseTool

class GetContextTextFromDbPediaInput(BaseModel):
    """Inputs for get_context_data"""
```

```
20
21      query_text: str = \
22        Field(
23          description="query_text user supplied query text"
24        )
25
26  class GetContextTextFromDbPediaTool(BaseTool):
27      name = "get_context_data"
28      description =
29        """
30          Useful when you want to make a query and get
31          context text from DBPedia. You should enter
32          and text containing entity names.
33        """
34      args_schema: Type[BaseModel] = \
35            GetContextTextFromDbPediaInput
36
37      def _run(self, query_text: str):
38          text = get_context_data(query_text)
39          return text
40
41      def _arun(self, query_text: str):
42          raise NotImplementedError
43                 (
44                  "get_context_data does not support async"
45                 )
46
47  ## Create agent
48
49  from langchain.agents import AgentType
50  from langchain.chat_models import ChatOpenAI
51  from langchain.agents import initialize_agent
52
53  llm = ChatOpenAI(model="gpt-3.5-turbo-0613",
54                   temperature=0)
55
56  tools = [GetContextTextFromDbPediaTool()]
57
58  agent = initialize_agent(tools, llm,
59                           agent=AgentType.OPENAI_FUNCTIONS,
60                           verbose=True)
61
62  ## Run agent
```

```
63
64 agent.run(
65     """
66       What country is Berlin in and what other
67       information about the city do you have?
68     """
69 )
```

The class **GetContextTextFromDbPediaInput** defined in lines 18-24 defines a tool input variable with an English language description for the variable that the LLM can use. The class **GetContextTextFromDbPediaTool** defined in lines 26-45 defines the tool name, a description for the use of an LLM, and the definition of the required methon **_run**. Methon **_run** uses the utility finction **get_context_data** defined in the source file **QA.py**.

We define a GPT-3.5 model in lines 53-54. Out example only uses one tool (our custom tool). We define the tools list in line 56 and setup the agent in lines 58-60.

The example output is (edited to fit page width):

```
1  > Entering new AgentExecutor chain...
2
3  Invoking: `get_context_data` with `{'query_text':
4                                      'Berlin'}`
5
6  name='Berlin'
7  dbpedia_type='<http://dbpedia.org/ontology/Place>'
8
9   select distinct ?s ?comment where {
10     ?s
11       <http://www.w3.org/2000/01/rdf-schema#label>
12       'Berlin'@en .
13     ?s
14       <http://www.w3.org/2000/01/rdf-schema#comment>
15       ?comment  .
16     FILTER  (lang(?comment) = 'en') .
17     ?s
18       <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
19       <http://dbpedia.org/ontology/Place> .
20  } limit 15
21
22 {'context_text': "Berlin (/bɜːrˈlɪn/ bur-LIN, German: [bɛɐ̯ˈliːn]) is the capital and\
23  largest city of Germany by both area and population. Its 3.6 million inhabitants ma
24 ke it the European Union's most populous city, according to population within city l
25 imits. One of Germany's sixteen constituent states, Berlin is surrounded by the Stat
```

```
e of Brandenburg and contiguous with Potsdam, Brandenburg's capital. Berlin's urban
area, which has a population of around 4.5 million, is the second most populous urba
n area in Germany after the Ruhr. The Berlin-Brandenburg capital region has around 6
.2 million inhabitants and is Germany's third-largest metropolitan region after the
Rhine-Ruhr and Rhine-Main regions. . "}

Berlin is the capital and largest city of Germany. It is located in the northeastern\
 part of the country. Berlin has a population of approximately 3.6 million people, m
aking it the most populous city in the European Union. It is surrounded by the State
 of Brandenburg and is contiguous with Potsdam, the capital of Brandenburg. The urba
n area of Berlin has a population of around 4.5 million, making it the second most p
opulous urban area in Germany after the Ruhr. The Berlin-Brandenburg capital region
has a population of approximately 6.2 million, making it Germany's third-largest met
ropolitan region after the Rhine-Ruhr and Rhine-Main regions.

> Finished chain.
```

LangChain Agent Tools Wrap Up

Writing custom agent tools is a great way to revisit the implementation of existing applications and improve them with the real world knowledge and reasoning abilities of LLMs. Both for practice in effectively using LLMs in your applications and also to extend your personal libraries and tools, I suggest that you look over you existing projects with an eye for either improving them using LLMs or refactoring them into reusable agent tools for your future projects.

More Useful Libraries for Working with Unstructured Text Data

Here we look at examples using two libraries that I find useful for my work: EmbedChain and Kor.

EmbedChain Wrapper for LangChain Simplifies Application Development

Taranjeet Singh developed a very nice wrapper library EmbedChain https://github.com/embedchain/embedchain that simplifies writing "query your own data" applications by choosing good defaults for the LangChain library.

I will show one simple example that I run on my laptop to search the contents of all of the books I have written as well as a large number of research papers. You can find my example in the GitHub repository for this book in the directory **langchain-book-examples/embedchain_test**. As usual, you will need an OpenAI API account and set the environment variable **OPENAI_API_KEY** to the value of your key.

I have copied PDF files for all of this content to the directory **~/data** on my laptop. It takes a short while to build a local vector embedding data store so I use two Python scripts. The first script **process_pdfs.py** that is shown here:

```
# reference: https://github.com/embedchain/embedchain

from embedchain import App
import os

test_chat = App()

my_books_dir = "/Users/mark/data/"

for filename in os.listdir(my_books_dir):
    if filename.endswith('.pdf'):
        print("processing filename:", filename)
        test_chat.add("pdf_file",
                      os.path.join(my_books_dir,
                      filename))
```

Here is a demo Python script **app.py** that makes three queries:

```
from embedchain import App

test_chat = App()

def test(q):
    print(q)
    print(test_chat.query(q), "\n")

test("How can I iterate over a list in Haskell?")
test("How can I edit my Common Lisp files?")
test("How can I scrape a website using Common Lisp?")
```

The output looks like:

```
$ python app.py
How can I iterate over a list in Haskell?
To iterate over a list in Haskell, you can use recursion or higher-order functions l\
ike `map` or `foldl`.

How can I edit my Common Lisp files?
To edit Common Lisp files, you can use Emacs with the Lisp editing mode. By setting \
the default auto-mode-alist in Emacs, whenever you open a file with the extensions "
.lisp", ".lsp", or ".cl", Emacs will automatically use the Lisp editing mode. You ca
n search for an "Emacs tutorial" online to learn how to use the basic Emacs editing
commands.

How can I scrape a website using Common Lisp?
One way to scrape a website using Common Lisp is to use the Drakma library. Paul Nat\
han has written a library using Drakma called web-trotter.lisp, which is available u
nder the AGPL license at articulate-lisp.com/src/web-trotter.lisp. This library can
be a good starting point for your scraping project. Additionally, you can use the wg
et utility to make local copies of a website. The command "wget -m -w 2 http:/knowle
dgebooks.com/" can be used to mirror a site with a two-second delay between HTTP req
uests for resources. The option "-m" indicates to recursively follow all links on th
e website, and the option "-w 2" adds a two-second delay between requests. Another o
ption, "wget -mk -w 2 http:/knowledgebooks.com/", converts URI references to local f
ile references on your local mirror. Concatenating all web pages into one file can a
lso be a useful trick.
```

Kor Library

The Kor library was written by Eugene Yurtsev. Kor is useful for using LLMs to extract structured data from unstructured text. Kor works by generating appropriate prompt text to explain to GPT-3.5 what information to extract and adding in the text to be processed.

The GitHub repository for Kor[1] is under active development so please check the project for updates. Here is the documentation[2].

For the following example, I modified an example in the Kor documentation for extracting dates in text.

```
" From documentation: https://eyurtsev.github.io/kor/"

from kor.extraction import create_extraction_chain
from kor.nodes import Object, Text, Number
from langchain.chat_models import ChatOpenAI
from pprint import pprint
import warnings ; warnings.filterwarnings('ignore')

llm = ChatOpenAI(
    model_name="gpt-3.5-turbo",
    temperature=0,
    max_tokens=2000,
    frequency_penalty=0,
    presence_penalty=0,
    top_p=1.0,
)

schema = Object(
    id="date",
    description=(
        "Any dates found in the text. Should be output in the format:"
        " January 12, 2023"
    ),
    attributes = [
        Text(id = "month",
             description = "The month of the date",
             examples=[("Someone met me on December 21, 1995",
                        "Let's meet up on January 12, 2023 and discuss our yearly bu\
dget")])
```

[1] https://github.com/eyurtsev/kor
[2] https://eyurtsev.github.io/kor/

```
    ],
)

chain = create_extraction_chain(llm, schema, encoder_or_encoder_class='json')


pred = chain.predict_and_parse(text="I will go to California May 1, 2024")['data']
print("* month mentioned in text=", pred)
```

Sample output:

```
$ python dates.py
* month mentioned in text= {'date': {'month': 'May'}}
```

Kor is a library focused on extracting data from text. You can get the same effects by writing for own prompts manually for GPT style LLMs but using Tor can save development time.

Book Wrap Up

This book has been fun to write but it has also somewhat frustrating.

It was fun because I have never been as excited by new technology as I have by LLMs and utility software like LangChain and LlamaIndex for building personalized applications.

This book was frustrating in the sense that it is now so very easy to build applications that just a few years would have been impossible to write. Usually when I write books I have two criteria: I only write about things that I am personally interested in and use, and I also hope to figure out non-obvious edge cases and make easier for my readers to use new tech. Here my frustration is writing about something that it is increasingly simple to do so I feel like my value is diminished.

All that said I hope, dear reader, that you found this book to be worth your time reading.

What am I going to do next? Although I am not fond of programming in JavaScript (although I find TypeScript to be somewhat better), I want to explore the possibilities of writing an open source Persistent Storage Web App Personal Knowledge Base Management System. I might get pushback on this but I would probably make it Apple Safari specific so I can use Apple's CloudKit JS to make its use seamless across macOS, iPadOS, and iOS. If I get the right kind of feedback on social media I might write a book around this project.

Thank you for reading my book!

Best regards, Mark Watson