Preface

I have been working in the field of artificial intelligence since 1982 and without a doubt Large Language Models (LLMs) like GPT-3 and infrastructure projects like LangChain are the largest technology breakthroughs that I have lived through. This book will use the LangChain and GPT Index (LlamaIndex) projects along with the OpenAI GPT-3 and ChatGPT APIs to solve a series of interesting problems.

Harrison Chase started the LangChain project in October 2022 and as I write this book in February 2023 the GitHub repository for LangChain https://github.com/hwchase17/langchain has 171 contributors. 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 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.

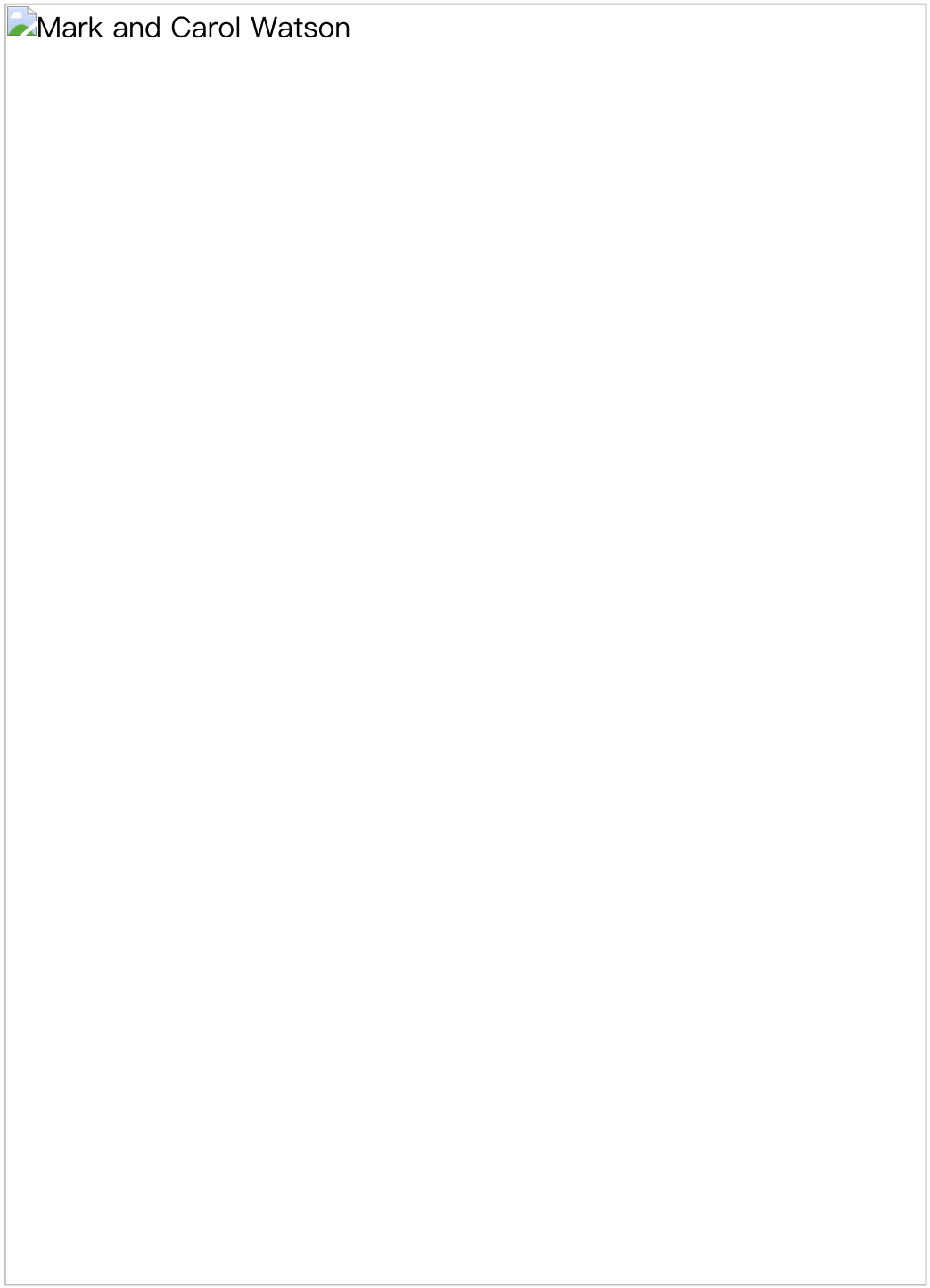
If you would like to support my work please consider purchasing my books on Leanpub and star my git repositories that you find useful on GitHub. You can also interact with me on social media on Mastodon and Twitter. I am also available as a consultant: https://markwatson.com.

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.


Mark and Carol Watson

Mark and Carol Watson

I would also like to thank the following readers who reported errors or typos in this book: Armando Flores.

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.

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.

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 account for the GMail and Google Calendar examples.

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

Large Language Model Overview

Large language models are a subset of artificial intelligence that use deep learning and neural networks to process natural language. Transformers 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 LLMs are BERT, GPT-3, T5 and Megatron-LM.

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 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

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 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.

Getting Started With LangChain

Harrison Chase started the LangChain project in October 2022 and as I write this book in February 2023 the GitHub repository for LangChain https://github.com/hwchase17/langchain has 171 contributors.

LangChain 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 pip install langchain llama_index openai
  2 pip install kor pydrive pandas rdflib
  3 pip install 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 [documentation](Langchain Quickstart Guide) 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 repsonses. 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-3.5 API from OpenAI because it is general purpose and much less expensive than OpenAI's previous model APIs. You will need to sign up for an API key and set it as an environment variable:

```
1 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:

```
 1 $ python
  2 >>> from langchain.llms import OpenAI
  3 >>> llm = OpenAI(temperature=0.8)
  4 >>> s = llm("John got into his new sports car, and he dro\
  5 ve it")
  6 >>> s
  7 ' to work\n\nJohn started up his new sports car and drove\
  8  it to work. He had a huge smile on his face as he drove,
  9  excited to show off his new car to his colleagues. The w
 10 ind blowing through his hair, and the sun on his skin, he
 11  felt a sense of freedom and joy as he cruised along the
 12 road. He arrived at work in no time, feeling refreshed an
 13 d energized.'
 14 >>> s = llm("John got into his new sports car, and he dro\
 15 ve it")
 16 >>> s
 17 " around town\n\nJohn drove his new sports car around tow\
 18 n, enjoying the feeling of the wind in his hair. He stopp
 19 ed to admire the view from a scenic lookout, and then spe
 20 d off to the mall to do some shopping. On the way home, h
 21 e took a detour down a winding country road, admiring the
 22  scenery and enjoying the feel of the car's powerful engi
```

```
ne. 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.mSetting the temperature in line 3 to a higher value increases the randomness.

Our next example is in the source file **directions_template.py** 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.llms 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 utiltiy 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 trans\
portation such as walking, driving, biking, or taking pub
lic transportation. Depending on the location of the stor
e, you may need to look up directions or maps to determin
e the best route to take.
```

```
10
11 How do I hang a picture on the wall?:
12
13 1. Find a stud in the wall, or use two or three wall anch\
14 ors for heavier pictures.
15 2. Measure and mark the wall where the picture hanger wil\
16 l go.
17 3. Pre-drill the holes and place wall anchors if needed.
18 4. Hammer the picture hanger into the holes.
19 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.

```
1 from langchain.prompts import PromptTemplate
 2 from langchain.llms import OpenAI
 3 llm = OpenAI(temperature=0.9)
 4
 5 def get_country_information(country_name):
 6     print(f"\nProcessing {country_name}:")
 7     global prompt
 8     if "prompt" not in globals():
 9         print("Creating prompt...")
10         prompt = PromptTemplate(
11             input_variables=["country_name"],
12             template = """
13 Predict the capital and population of a country.
14
15 Country: {country_name}
16 Capital:
17 Population:""",
18         )
19     prompt_text = prompt.format(country_name=country_name)
20     print(prompt_text)
21     return llm(prompt_text)
22
23 print(get_country_information("Canada"))
24 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. 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.embeddings import OpenAIEmbeddings
>>> embeddings = OpenAIEmbeddings()
>>> 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, -0.0033224\
```

```
383369088173, -0.01794492080807686, -0.017969949170947075
, 0.028506645932793617, -0.013414892368018627, 0.00466768
16418766975, -0.0024965214543044567, -0.02662956342101097
,
...]]
>>> query_embedding = embeddings.embed_query("Does John l\
ive in the city?")
>>> query_embedding
[0.028048301115632057, 0.011499025858938694, -0.009440079\
33139801, -0.020809611305594444, -0.023904507979750633, 0
.018750663846731186, -0.01626438833773136, 0.018129095435
142517,
...]
>>>
```

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. You need to install a few libraries:

```
pip install chroma
pip install chromadb
pip install unstructured
```

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 OpenAI, VectorDBQA

embeddings = OpenAIEmbeddings()

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

```
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=2500, ch\
unk_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 laborat\
ory?")
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. It 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 sp\
ecified 1000
Running Chroma using direct local API.
Using DuckDB in-memory for database. Data will be transie\
nt.
Query: What kinds of equipment are in a chemistry laborat\
ory?
Answer:  A chemistry lab would typically include glasswar\
e, such as beakers, flasks, and test tubes, as well as ot
her equipment such as scales, Bunsen burners, and thermom
eters.
Query: What is Austrian School of Economics?
Answer:  The Austrian School is a school of economic thou\
ght that emphasizes the spontaneous organizing power of t
he price mechanism. Austrians hold that the complexity of
 subjective human choices makes mathematical modelling of
 the evolving market extremely difficult and advocate a "
```

```
laissez faire" approach to the economy. Austrian School e
conomists advocate the strict enforcement of voluntary co
ntractual agreements between economic agents, and hold th
at commercial transactions should be subject to the small
est possible imposition of forces they consider to be (in
 particular the smallest possible amount of government in
tervention). The Austrian School derives its name from it
s predominantly Austrian founders and early supporters, i
ncluding Carl Menger, Eugen von Bohm-Bawerk and Ludwig vo
n Mises.
Query: Why do people engage in sports?
Answer:  People engage in sports for leisure and entertai\
nment, as well as for physical 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 hormones and enzymes produced by th
e body can affect the energy levels and muscle performanc
e. Chemicals in the body, such as adenosine triphosphate
(ATP) and urea, can affect the body's energy production a
nd muscle metabolism during exercise. Additionally, the b
ody's levels of electrolytes, vitamins, and minerals can
affect exercise performance.
Exiting: Cleaning up .chroma directory
```

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.

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.

LlamaIndex 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. 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.

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 # Derived from a documentation example at:
2 # https://github.com/jerryjliu/gpt_index
3
4 # make sure you set the following environment variable
5 # is set: OPENAI_API_KEY
6
7 from llama_index import GPTSimpleVectorIndex,
8                         SimpleDirectoryReader
9 documents = SimpleDirectoryReader('../data').load_data()
10 index = GPTSimpleVectorIndex(documents)
11
12 # save to disk
13 index.save_to_disk('index.json')
14 # load from disk
15 index = GPTSimpleVectorIndex.load_from_disk('index.json')
16
17 # search for a document
18 r = index.query("effect of body chemistry on exercise?")
19 print(r)
```

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

```
1 INFO:root:> [build_index_from_documents] Total LLM token \
2 usage: 0 tokens
3 INFO:root:> [build_index_from_documents] Total embedding \
4 token usage: 2272 tokens
5 INFO:root:> [query] Total LLM token usage: 1018 tokens
6 INFO:root:> [query] Total embedding token usage: 7 tokens
7
8 The effect of body chemistry on exercise can vary dependi\
9 ng on the individual. Factors such as hydration levels, e
10 lectrolyte balance, and the availability of energy-storin
11 g molecules such as adenosine triphosphate (ATP) can all
12 affect the bodys ability to perform exercise. Additionall
13 y, hormones such as adrenaline and cortisol can influence
14  the bodys response to exercise, as can the presence of c
15 ertain nutrients and minerals.
```

Note that in general, many organic chemicals could be listed in response to the query **effect of body chemistry on exercise?** but that just **adenosine triphosphate (ATP)** was mentioned in the test document in the file **data/chemistry.txt**.

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/exampl\
es/data_connectors/WebPageDemo.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)
    for question in questions:
        print(f"\n== QUESTION: {question}\n")
        response = index.query(question)
        print(f"== RESPONSE: {response}")

if __name__ == "__main__":
  url_list = ["https://markwatson.com"]
  query_website(url_list, "What programming languages doe\
s Mark use?",
                          "How many books has Mark writte\
n?",
                          "What musical instruments does \
Mark play?")
```

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:

```
INFO:root:> [build_index_from_documents] Total LLM token \
usage: 0 tokens
INFO:root:> [build_index_from_documents] Total embedding \
token usage: 0 tokens

== QUESTION: What programming languages does Mark use?

INFO:root:> [query] Total LLM token usage: 2210 tokens
INFO:root:> [query] Total embedding token usage: 0 tokens
== RESPONSE:
Mark uses a variety of programming languages, including H\
askell, Ruby, Clojure, JavaScript, Java, Common Lisp, Pyt
hon, and Smalltalk.

== QUESTION: How many books has Mark written?

INFO:root:> [query] Total LLM token usage: 2189 tokens
INFO:root:> [query] Total embedding token usage: 0 tokens
== RESPONSE:
Mark has written 20+ books.

== QUESTION: What musical instruments does Mark play?

INFO:root:> [query] Total LLM token usage: 2197 tokens
INFO:root:> [query] Total embedding token usage: 0 tokens
== RESPONSE:
Mark plays guitar, didgeridoo, and American Indian flute.
```

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.

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 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 Je\
fferson Clinton is an American retired politician who ser
ved as the 42nd president of the United States from 1993
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 and Wikidata are public Knowledge Graphs (KGs) that store data as Resource Description Framework (RDF) and are accessed through the SPARQL Query Language for RDF. The examples for this project are in the GitHub repository for this book 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.

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:

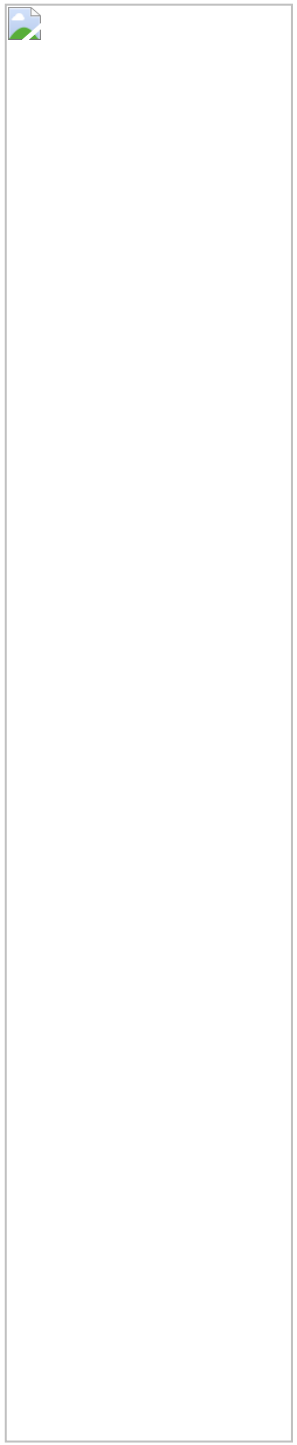
- DBPedia: https://dbpedia.org/page/Sedona,_Arizona
- Wikidata: https://www.wikidata.org/wiki/Q80041

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:

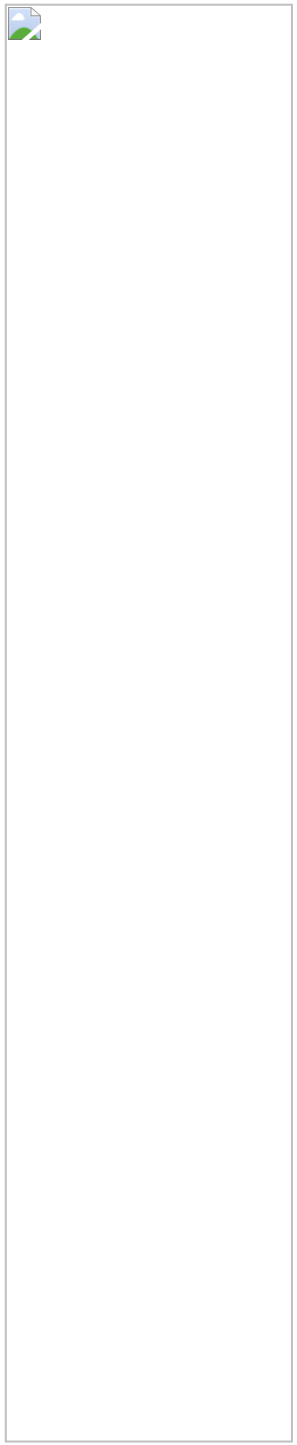


In this SPARQL query the prefix **wd:** stands for Wikidata data while the prefix **wdt:** stands for Wikidata type (or property). The prefix **rdfs:** stands for RDF Schema.

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_Aire\
s>
    <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 Pyth\
on 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', 'Ethiopia', 'Australia', 'Uzbekista
n', 'Tanzania', 'Albania', 'Belarus', 'Vanuatu', 'Armenia
', 'Syria', 'Andorra', 'Venezuela', 'France', 'Vietnam',
'Azerbaijan']

This is a list of all the countries mentioned in the cont\
ext information. All of the countries are listed in the c
ontext 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.

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**:

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

def get_possible_eitity_uris_from_wikidata(entity_name):
    sparql = SPARQLWrapper("https://query.wikidata.org/spa\
rql")
    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']['bindin\
gs']).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/spa\
rql")
    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']['bindi\
ngs'])
```

```
    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_eitity_uris_from_wikidata(e\
ntity_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
```

```
 4
 5 def wd_query(question, *entity_names):
 6     prompt_texts = []
 7     for entity_name in entity_names:
 8         prompt_texts +=
 9           [generate_prompt_text(entity_name)]
10     documents =
11       StringIterableReader().load_data(texts=prompt_texts)
12     index = GPTTreeIndex(documents)
13     return index.query(question)
14
15 if __name__ == "__main__":
16   print("Sedona:", wd_query("What is Sedona?", "Sedona"))
17   print("California:",
18         wd_query("What is California?", "California"))
19   print("Bill Clinton:",
20     wd_query("When was Bill Clinton president?",
21              "Bill Clinton"))
22   print("Donald Trump:",
23     wd_query("Who is Donald Trump?",
24              "Donald Trump"))
```

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

```
 1 $ python wikidata_query.py
 2 Total LLM token usage: 162 tokens
 3 INFO:llama_index.token_counter.token_counter:> [build_ind\
 4 ex_from_documents] INFO:llama_index.indices.query.tree.le
 5 af_query:> Starting query: What is Sedona?
 6 INFO:llama_index.token_counter.token_counter:> [query] To\
 7 tal LLM token usage: 154 tokens
 8 Sedona: Sedona is a city in the United States located in \
 9 the counties of Yavapai and Coconino, Arizona. It is also
10  the title of a 2012 film, a company, and a 2015 single b
11 y Houndmouth.
12
13 Total LLM token usage: 191 tokens
14 INFO:llama_index.indices.query.tree.leaf_query:> Starting\
15  query: What is California?
16 California: California is a U.S. state in the United Stat\
17 es of America.
18
19 Total LLM token usage: 138 tokens
20 INFO:llama_index.indices.query.tree.leaf_query:> Starting\
21  query: When was Bill Clinton president?
22 Bill Clinton: Bill Clinton was the 42nd President of the \
```

```
23 United States from 1993 to 2001.
24
25 Total LLM token usage: 159 tokens
26 INFO:llama_index.indices.query.tree.leaf_query:> Starting\
27  query: Who is Donald Trump?
28 Donald Trump: Donald Trump is the 45th President of the U\
29 nited States, serving from 2017 to 2021.
```

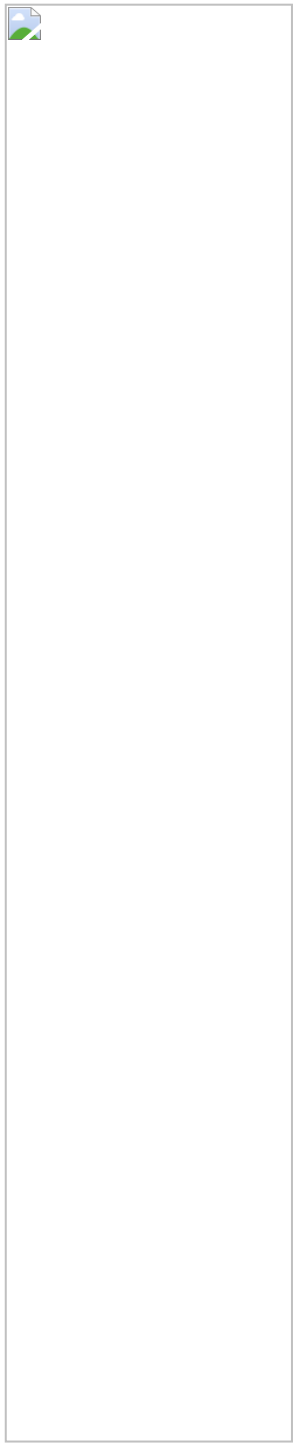
Using LLMs To Organize Information in Our Google Drives

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, 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.
- 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):

```
 1 from pydrive.auth import GoogleAuth
  2 from pydrive.drive import GoogleDrive
  3 from pathlib import Path
  4
  5 # good GD search docs:
  6 # https://developers.google.com/drive/api/guides/search-f\
  7 iles
  8
  9 # Authenticate with Google
 10 gauth = GoogleAuth()
 11 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 directory "

    file_list = drive.ListFile({'q': f"'{dir_id}' in pare\
nts and trashed=false"}).GetList()
    for file1 in file_list:
        print('title: %s, id: %s' % (file1['title'], file\
1['id']))
    return [[file1['title'], file1['id'], file1.GetConten\
tString()]
            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.t\
xt':
    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=5\
29311921932-xsmj3hhiplr0dhqjln13fo4rrtvoslo8.apps.googleu
sercontent.com&redirect_uri=http%3A%2F%2Flocalhost%3B6180
```

```
%2F&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive
&access_type=offline&response_type=code

Authentication successful.

title: testdata, id: 1TZ9bnL5XYQvKACJw8VoKWdVJ8jeCszJ
title: sports.txt, id: 18RN4ojvURWt5yoKNtDdAJbh4fvmRpzwb
title: Anaconda blog article, id: 1kpLaYQA4Ao8ZbdFaXU209h\
g-z0tv1xA7YOQ4L8y8NbU
title: backups_2023, id: 1-k_r1HTfuZRWN7vwWWsYqfssl0C96J2x
title: Work notes, id: 1fDyHyZtKI-0oRNabA_P41LltYjGoek21
title: Sedona Writing Group Contact List, id: 1zK-5v9OQUf\
y8Sw33nTCl9vnL822hL1w
 ...
['sports.txt', '18RN4ojvURWt5yoKNtDdAJbh4fvmRpzwb', 'Spor\
t is generally recognised as activities based in physical
 athleticism or physical dexterity.[3] Sports are usually
 governed by rules to ensure fair competition and consist
ent adjudication of the winner.\n\n"Sport" comes from the
 Old French desport meaning "leisure", with the oldest de
finition in English from around 1300 being "anything huma
ns find amusing or entertaining".[4]\n\nOther bodies advo
cate widening the definition of sport to include all phys
ical 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, SimpleDirec\
toryReader
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_ind\
ex_from_documents] Total LLM token usage: 0 tokens
INFO:llama_index.token_counter.token_counter:> [build_ind\
ex_from_documents] Total embedding token usage: 111 token
s
INFO:llama_index.token_counter.token_counter:> [query] To\
tal LLM token usage: 202 tokens
INFO:llama_index.token_counter.token_counter:> [query] To\
tal embedding token usage: 7 tokens

Sport is generally recognised as activities based in phys\
ical athleticism or physical dexterity that are governed
by rules to ensure fair competition and consistent adjudi
cation of the winner. It is anything humans find amusing
or entertaining, and can include all forms of physical ex
ercise, 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 phys\
ical athleticism or physical dexterity.[3] Sports are usu
ally governed by rules to ensure fair competition and con
sistent adjudication of the winner.

"Sport" comes from the Old French desport meaning "leisur\
e", with the oldest definition in English from around 130
0 being "anything humans find amusing or entertaining".[4
]

Other 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 exe
rcise, 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.

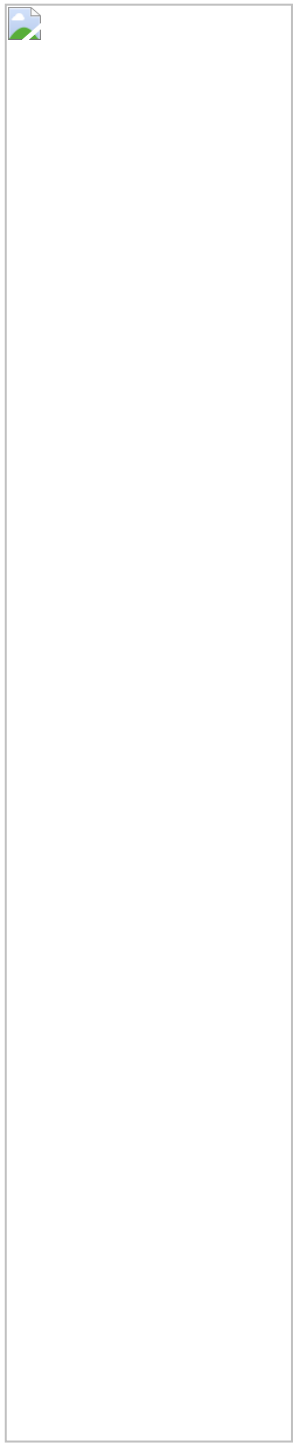
Set Up Development Environment

You will need a developer key for Zapier Natural Language Actions API. 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.

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

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

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



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-description", verbose=True)

agent.run("Send an Email to TEST_EMAIL_ADDRESS via gmail \
that is a pitch for hiring Mark Watson as a consultant fo
r 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 th\
e subject "Pitch for Hiring Mark Watson as a Consultant f
or Deep Learning and Large Language Models" and the body
"Dear Mark Watson, I am writing to you to pitch the idea
of hiring you as a consultant for deep learning and large
 language models. I believe you have the expertise and ex
perience to help us achieve our goals. Please let me know
 if you are interested in discussing further. Thank you f
or your time."
Cc: not enough information provided in the instruction, m\
issing Cc
Observation: {"labelIds": "SENT"}
Thought: I now know the final answer
Final Answer: An email has been sent to TEST_EMAIL_ADDRES\
S with the subject "Pitch for Hiring Mark Watson as a Con
sultant for Deep Learning and Large Language Models" and
the body "Dear Mark Watson, I am writing to you to pitch
```

```
the idea of hiring you as a consultant for deep learning
and large language models. I believe you have the experti
se and experience to help us achieve our goals. Please le
t me know if you are interested in discussing further. Th
ank 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-descripti\
on", 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#event", "end__dateTime": "2023-
03-23T10:00:00-07:00", "status": "confirmed", "end__dateT
ime_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 tomor\
```

```
14 row at 10:00AM.
15
16 > 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 documentatiuon for this database so please take a minute to review the table schema diagram and text description.

This example is derived from the LangChain documentation. 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.

```
# 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 Employee\
Id = 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(in\
voices.InvoiceId) AS NumberOfInvoices FROM customers INNE
R JOIN invoices ON customers.CustomerId = invoices.Custom
erId GROUP BY customers.CustomerId ORDER BY NumberOfInvoi
ces DESC LIMIT 5;
SQLResult: [('Luis', 'Goncalves', 7), ('Leonie', 'Kohler'\
, 7), ('Francois', 'Tremblay', 7), ('Bjorn', 'Hansen', 7)
, ('Frantisek', 'Wichterlova', 7)]
Answer: Luis Goncalves has the most invoices with 7.
```

```
> Finished chain.

> Entering new SQLDatabaseChain chain...
List all music genres in the database
SQLQuery: SELECT Name FROM genres
SQLResult: [('Rock',), ('Jazz',), ('Metal',), ('Alternati\
ve & Punk',), ('Rock And Roll',), ('Blues',), ('Latin',),
 ('Reggae',), ('Pop',), ('Soundtrack',), ('Bossa Nova',),
 ('Easy Listening',), ('Heavy Metal',), ('R&B/Soul',), ('
Electronica/Dance',), ('World',), ('Hip Hop/Rap',), ('Sci
ence Fiction',), ('TV Shows',), ('Sci Fi & Fantasy',), ('
Drama',), ('Comedy',), ('Alternative',), ('Classical',),
('Opera',)]
Answer: Rock, Jazz, Metal, Alternative & Punk, Rock And R\
oll, Blues, Latin, Reggae, Pop, Soundtrack, Bossa Nova, E
asy Listening, Heavy Metal, R&B/Soul, Electronica/Dance,
World, Hip Hop/Rap, Science Fiction, TV Shows, Sci Fi & F
antasy, Drama, Comedy, Alternative, Classical, Opera
> Finished chain.
```

Natural Language Database Query Wrap Up

I had an example I wrote for the first two editions of my Java AI book (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 used an OpenAI example https://github.com/openai/openai-cookbook/blob/main/examples/Backtranslation_of_SQL_queries.py that relatively simnple 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!

Examples Using Hugging Face Open Source Models

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

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

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

```
1 HUGGINGFACEHUB_API_TOKEN
```

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

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

Here we will use the alternative Hugging Face wrapper class:

```
1 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 from langchain import HuggingFaceHub, LLMChain
2 from langchain.prompts import PromptTemplate
3
4 hub_llm = HuggingFaceHub(
5     repo_id='google/flan-t5-xl',
6     model_kwargs={'temperature':1e-6}
```

```
)

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

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.

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. Note that I have used a much smaller model in this example and reduced the prompt and output text size.

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

import time
import torch
from langchain.llms.base import LLM
from llama_index import SimpleDirectoryReader, LangchainE\
mbedding
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_na\
me,
                        model_kwargs={"torch_dtype":torch\
.bfloat16})

    def _call(self, prompt, stop = None):
        prompt_length = len(prompt)
        response = self.pipeline(prompt, max_new_tokens=n\
um_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_d\
ata()
index = GPTListIndex(documents, llm_predictor=llm_predict\
or,
                     prompt_helper=prompt_helper)

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

```
onds.")

# 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} second\
s.")
```

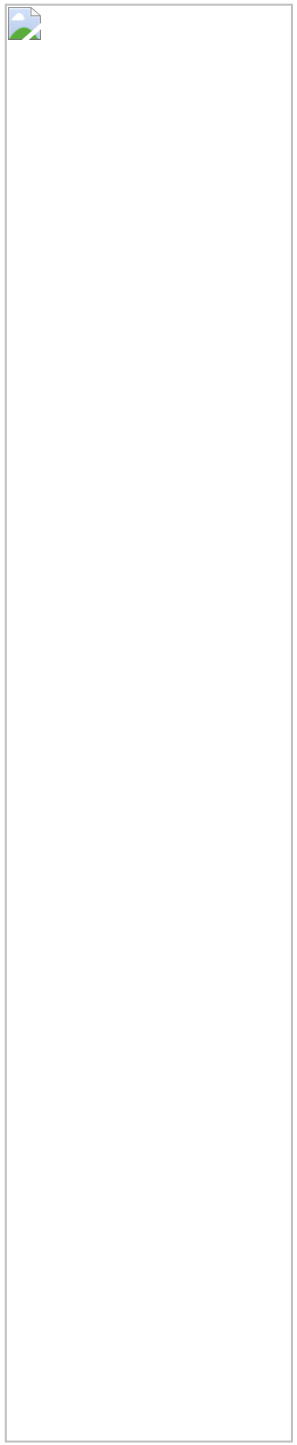
When running on my M1 MacBook Pro using only the CPU (no GPU or Neural Engine configuration) we can read the model from disk quickly but it takes a while to process queries:

```
$ python hf_transformer_local.py
INFO:llama_index.token_counter.token_counter:> [build_ind\
ex_from_documents] Total LLM token usage: 0 tokens
INFO:llama_index.token_counter.token_counter:> [build_ind\
ex_from_documents] Total embedding token usage: 0 tokens
Time to load model from disk: 1.5303528308868408 seconds.
INFO:llama_index.token_counter.token_counter:> [query] To\
tal LLM token usage: 182 tokens
INFO:llama_index.token_counter.token_counter:> [query] To\
tal embedding token usage: 0 tokens

"Sport" comes from the Old French desport meaning "leisur\
e", with the oldest definition in English from around 130
0 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 a standard Colab GPU, the query/prediction time is much faster. Here is a link to my Colab notebook if you would prefer to run this example on Colab instead of on your laptop.

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 be
ans and cumin for at least one minute until the mixture i
s 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-davinci-002 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.document_loaders import DirectoryLoader
from langchain import OpenAI, VectorDBQA

embeddings = OpenAIEmbeddings()

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 transie\
nt.

Recipe creation request: Create a new recipe using both B\
roccoli and Chicken

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

Ingredients:
1 cup broccoli
1 pound chicken meat
```

```
15 2 tablespoons soy sauce
16 1 tablespoon honey
17 1 tablespoon vegetable oil
18 1 clove garlic, minced
19 1 teaspoon rice vinegar
20
21 Directions:
22
23 1. In a large bowl, whisk together soy sauce, honey, vege\
24 table oil, garlic, and rice vinegar.
25 2. Cut the broccoli into small florets. Add the broccoli \
26 and chicken to the bowl and toss to coat.
27 3. Preheat a grill or grill pan over medium-high heat.
28 4. Grill the chicken and broccoli for 5-7 minutes per sid\
29 e, or until the chicken is cooked through and the broccol
30 i is slightly charred.
31 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:

```
1 Recipe creation request: Create a recipe using Beans, Ric\
 2 e, and Chicken
 3
 4 Recipe name: Beans and Rice with Chicken
 5 Number of servings: 4
 6 Ingredients:
 7 1 cup white rice
 8 1 cup black beans
 9 1 chicken breast, cooked and shredded
10 1/2 teaspoon cumin
11 1/2 teaspoon chili powder
12 1/4 teaspoon salt
13 1/4 teaspoon black pepper
14 1 tablespoon olive oil
15 1/2 cup salsa
16 1/4 cup cilantro, chopped
17
18 Directions:
19 1. Cook rice according to package instructions.
20 2. In a medium bowl, combine black beans, chicken, cumin,\
```

```
 chili powder, salt, and black pepper.
3. Heat olive oil in a large skillet over medium heat. Ad\
d 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 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.

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