Building AI Applications with LangChain and GPT

You've probably talked to ChatGPT using the web interface, or used the API with the `openai` python package and wondered "what if I could teach it about my own data?". Today we're going to build such an application using LangChain, a framework for developing applications powered by language models.

In today's project, we'll build a chatbot powered by GPT-3.5 that can answer questions about LangChain, as it will have knowledge of the LangChain documentation. We'll cover:

- Getting setup with an OpenAI developer account and integration with Workspace;
- Install the LangChain package
- Preparing the data
- Embed the data using OpenAI's Embed API, and get a cost estimate for this operation
- Storing the data in a vector database
- How to query the vector database
- Putting together a basic chat application to "talk to the LangChain docs"

Before you begin

Unzip the required data by running the following cell

```
!test -f contents.zip && unzip contents.zip && rm contents.zip
```

The output is too long to be displayed in full. It has been truncated to the first 1000 lines.

```
Archive:  contents.zip
   creating: chroma-data-langchain-docs/
   creating: chroma-data-langchain-docs/index/
  inflating: chroma-data-langchain-docs/index/index_119c5fd7-e49b-4a0c-b566-
c6c01c75ca1b.bin
  inflating: chroma-data-langchain-docs/index/index_metadata_119c5fd7-e49b-4a0c-
b566-c6c01c75ca1b.pkl
  inflating: chroma-data-langchain-docs/index/id_to_uuid_119c5fd7-e49b-4a0c-b566-
c6c01c75ca1b.pkl
  inflating: chroma-data-langchain-docs/index/uuid_to_id_119c5fd7-e49b-4a0c-b566-
c6c01c75ca1b.pkl
  inflating: chroma-data-langchain-docs/chroma-collections.parquet
  inflating: chroma-data-langchain-docs/chroma-embeddings.parquet
   creating: my-embeddings/
   creating: my-embeddings/index/
  inflating: my-embeddings/index/index_3b116138-d698-42e6-8010-9361505ed6c9.bin
  inflating: my-embeddings/index/index_metadata_3b116138-d698-42e6-8010-
9361505ed6c9.pkl
  inflating: my-embeddings/index/uuid_to_id_3b116138-d698-42e6-8010-
9361505ed6c9.pkl
  inflating: my-embeddings/index/id_to_uuid_3b116138-d698-42e6-8010-
9361505ed6c9.pkl
  inflating: my-embeddings/chroma-collections.parquet
  inflating: my-embeddings/chroma-embeddings.parquet
   creating: rtdocs/
   creating: rtdocs/python.langchain.com/
   creating: rtdocs/python.langchain.com/en/
   creating: rtdocs/python.langchain.com/en/latest/
   creating: rtdocs/python.langchain.com/en/latest/getting_started/
```

Task 0: Setup

For the purpose of this training, we'll need to install a few packages:

- `langchain`: The LangChain framework
- `chromadb`: The package we'll use for the vector database
- `tiktoken`: A tokenizer we'll use to count GPT-3 tokens

```
# install openai (version 0.27.1)
!pip install openai==0.27.1
# install langchain (version 0.0.191)
!pip install langchain==0.0.191
# install chromadb
!pip install chromadb==0.3.26
# install tiktoken
!pip install tiktoken==0.4.0
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: langchain==0.0.191 in
/home/repl/.local/lib/python3.8/site-packages (0.0.191)
Requirement already satisfied: PyYAML>=5.4.1 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (6.0)
Requirement already satisfied: SQLAlchemy<3,>=1.4 in
/usr/local/lib/python3.8/dist-packages (from langchain==0.0.191) (1.4.40)
Requirement already satisfied: aiohttp<4.0.0,>=3.8.3 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (3.8.4)
Requirement already satisfied: async-timeout<5.0.0,>=4.0.0 in
/usr/local/lib/python3.8/dist-packages (from langchain==0.0.191) (4.0.2)
Requirement already satisfied: dataclasses-json<0.6.0,>=0.5.7 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (0.5.8)
Requirement already satisfied: numexpr<3.0.0,>=2.8.4 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (2.8.4)
Requirement already satisfied: numpy<2,>=1 in /usr/local/lib/python3.8/dist-
packages (from langchain==0.0.191) (1.23.2)
Requirement already satisfied: openapi-schema-pydantic<2.0,>=1.2 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (1.2.4)
Requirement already satisfied: pydantic<2,>=1 in /usr/local/lib/python3.8/dist-
packages (from langchain==0.0.191) (1.10.2)
Requirement already satisfied: requests<3,>=2 in /usr/local/lib/python3.8/dist-
packages (from langchain==0.0.191) (2.28.1)
Requirement already satisfied: tenacity<9.0.0,>=8.1.0 in
/home/repl/.local/lib/python3.8/site-packages (from langchain==0.0.191) (8.2.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.8/dist-
packages (from aiohttp<4.0.0,>=3.8.3->langchain==0.0.191) (21.4.0)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in
/usr/local/lib/python3.8/dist-packages (from aiohttp<4.0.0,>=3.8.3-
```

Task 1: Load data

To be able to embed and store data, we need to provide LangChain with Documents. This is easy to achieve in LangChain thanks to **Document Loaders**. In our case, we're targeting a "Read the docs" documentation, for which there is a loader `ReadTheDocsLoader`. In the folder `rtdocs`, you'll find all the HTML files from the LangChain documentation (**https://python.langchain.com/en/latest/index.html**).

▶ How did we obtain the data

Our first task is to load these HTML files as documents that we can use with langchain: we're going to use the `ReadTheDocsLoader`. It will read the directory containing all HTML files and transform them into `Document` objects. `ReadTheDocsLoader` will read each HTML file, remove HTML tags to only keep the text and return it as a `Document`. At the end of this task, we'll have a variable `raw_documents` containing a list of `Document`: one `Document` per HTML file.

Note that in this step we won't actually load the documents into a database, we're simply loading the documents in a list.

Instructions

1. import `ReadTheDocsLoader` from `langchain.document_loaders`
2. Create the loader, pointing to the `rtdocs/python.langchain.com/en/latest` directory and enabling the HTML parser feature with `features='html.parser'`
3. Load the data in `raw_documents` by calling `loader.load()`

```python
# Import ReadTheDocsLoader
from langchain.document_loaders import ReadTheDocsLoader

# Create a loader for the `rtdocs/python.langchain.com/en/latest` folder
loader = ReadTheDocsLoader("rtdocs/python.langchain.com/en/latest",
features="html.parser")

# Load the data
raw_documents = loader.load()
```

Task 2: Slice the documents into smaller chunks

In the previous step, we turned each HTML file into a Document. These files may be very long, and are potentially too large to embed fully. It's also a good practice to avoid embedding large documents:

- long documents often contain several concepts. Retrieval will be easier if each concept is indexed separately;
- retrieved documents will be injected in a prompt, so keeping them short will keep the prompt small(ish)

LangChain has a collection of tools to do this: **Text Splitters**. In our case, we'll be using the most straightfoward one and simplest to use: the **Recursive Character Text Splitter**. The recursive text splitter will recursively reduce the input by splitting it by paragraph, then sentences, then words as needed until the chunk is small enough.

Instructions

1. Import the `RecursiveCharacterTextSplitter` from `langchain.text_splitter`
2. Create a text splitter configured with `chunk_size=1000` and `chunk_overlap=200`
   These values are arbitrary and you'll need to try different ones to see which best serve your use case
3. split the `raw_documents` and store them as `documents`, using the `.split_documents()` method

```python
# Import RecursiveCharacterTextSplitter
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Create the text splitter
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200
)

# Split the documents
documents = splitter.split_documents(raw_documents)
```

```
documents[0]
```

```
Document(page_content='.md\n.pdf\nConcepts\n Contents \nChain of Thought\nAction
Plan Generation\nReAct\nSelf-ask\nPrompt Chaining\nMemetic Proxy\nSelf
Consistency\nInception\nMemPrompt\nConcepts#\nThese are concepts and terminology
commonly used when developing LLM applications.\nIt contains reference to external
papers or sources where the concept was first introduced,\nas well as to places in
LangChain where the concept is used.\nChain of Thought#\nChain of Thought (CoT) is
a prompting technique used to encourage the model to generate a series of
intermediate reasoning steps.\nA less formal way to induce this behavior is to
include “Let’s think step-by-step” in the prompt.\nChain-of-Thought Paper\nStep-by-
Step Paper\nAction Plan Generation#\nAction Plan Generation is a prompting
technique that uses a language model to generate actions to take.\nThe results of
these actions can then be fed back into the language model to generate a subsequent
action.\nWebGPT Paper\nSayCan Paper\nReAct#', metadata={'source':
'rtdocs/python.langchain.com/en/latest/getting_started/concepts.html'})
```

Task 3: count tokens and get a cost estimate of embedding

We're now ready to embed our documents. Before we do so, we'd like to get an idea of how big it is and how much it will cost to embed. To do so, we'll use the `tiktoken` library (no relation to TikTok, there is no dancing involved). tiktoken allows to encode and decode strings of text into tokens. In our case, we're mostly interested in how many tokens our documents translate to.

> 💡 To better understand what a token is to GPT, head to **OpenAI's Tokenizer page** where you can see how a text translates to tokens.

Prices for different models can be found on their **pricing page**.

Instructions

1. Import `tiktoken`
2. Create a tokenizer for the `text-embedding-ada-002` model using the `.encoding_for_model()` method
3. Count tokens in each document using the `.encode()` method
4. Calculate the sum of all tokens
5. Calculate a cost estimate. The `text-embedding-ada-002` model costs `$0.0004` for 1000 tokens

```
# Import tiktoken
import tiktoken

# Create an encoder
encoder = tiktoken.encoding_for_model("text-embedding-ada-002")

# Count tokens in each document
doc_tokens = [len(encoder.encode(doc.page_content)) for doc in documents]

# Calculate the sum of all token counts
total_tokens = sum(doc_tokens)

# Calculate a cost estimate
cost = (total_tokens/1000) * 0.0004
print(f"Total tokens: {total_tokens} - cost: ${cost:.2f}")
```

```
Total tokens: 1530817 - cost: $0.61
```

Task 4: embed the documents and store embeddings in the vector database

We're now ready to embed our documents. Since embedding costs money, we'll want to save the embeddings into a database. LangChain can take care of all that using a **Vector Store**.

There are plenty of vector stores to choose from (see the **full list**). Today we'll use **Chroma**, but you could be using any other as they have the same interface in LangChain. Once again you'll need to try many of them to see which best fits your use case: some vector stores have specific features (like multimodality or multilingual), so be sure to check them out.

Chroma is simple to use and can be persisted to disk. If you do not whish to embed the full set of documents yourself, feel free to skip this step and use the provided folder `chroma-data-langchain-docs`: we've already embedded all documents and persisted it in this folder.

Instructions

1. Import `Chroma` from `langchain.vectorstores`
2. Import `OpenAIEmbeddings` from `langchain.embeddings.openai`
3. Create the embedding function
4. Create a database from our documents, using `Chroma.from_documents()`. Pass the documents, embedding function and `persist_directory`.
   Warning: executing this will embed thousands of documents and will cost about $0.6
5. Persist the data to disk by calling `.persist()` on the database

```python
# Import chroma
from langchain.vectorstores import Chroma

# Import OpenAIEmbeddings
from langchain.embeddings.openai import OpenAIEmbeddings

# Create the mebedding function
embedding_function = OpenAIEmbeddings()

# Create a database from the documents and embedding function
db = Chroma.from_documents(documents=documents, embedding=embedding_function,
persist_directory="my-embeddings")

# Persist the data to disk
db.persist()
```

Alternative: use the provided embeddings

We have already executed the step above to embed all documents and stored the result in the `chroma-data-langchain-docs` folder. Instead of embedding all the documents yourself, you can use these embeddings at no cost.

The result of this step is the same as the step above, but will not call the OpenAI API and cost nothing.

Instructions

1. Import `Chroma` from `langchain.vectorstores`
2. Import `OpenAIEmbeddings` from `langchain.embeddings.openai`
3. Create the embedding function
4. Load the database from the `chroma-data-langchain-docs` directory, and provide the embedding function

```python
# Import chroma
from langchain.vectorstores import Chroma

# Import OpenAIEmbeddings
from langchain.embeddings.openai import OpenAIEmbeddings

# Create the embedding function
embedding = OpenAIEmbeddings()

# Load the database from existing embeddings
db = Chroma(persist_directory="chroma-data-langchain-docs",
embedding_function=embedding)
```

Step 5: query the vector database

Now that we have a vector database, we can query it. A vector database stores embeddings (vectors) and allow to search through them using K-Nearest Neighbors algorithm (or a variation of it). When we query it the following will happen:

1. Embed the text query to obtain a vector. It is crucial that this embedding is made using the same embedding technique that was used to embed the documents;
2. Calculate the distance (or similarity) between the query vector and all other vectors;
3. Sort results by similarity;
4. Return the most similar documents.

To do this with LangChain, we can use the `.similarity_search_with_score()` method of the database.

Instructions

1. Call the `similarity_search_with_score` on `db` with the search query as parameter. Store the results in `results`;
2. Print the results; `results` is a list of tuples like this: `[(doc, score), (doc, score), ...]`

```python
# Call the `similarity_search_with_score` method on `db`
results = db.similarity_search_with_score("how do i load data from wikipedia?")

# Print the results
for (doc, score) in results:
    print('score', score)
    print(doc.page_content)
    print('-----------------')
```

```
score 0.2888728082180023
.ipynb
.pdf
Wikipedia
 Contents
Installation
Examples
Wikipedia#
Wikipedia is a multilingual free online encyclopedia written and maintained by a
community of volunteers, known as Wikipedians, through open collaboration and
using a wiki-based editing system called MediaWiki. Wikipedia is the largest and
most-read reference work in history.
This notebook shows how to load wiki pages from wikipedia.org into the Document
format that we use downstream.
Installation#
First, you need to install wikipedia python package.
#!pip install wikipedia
Examples#
WikipediaLoader has these arguments:
query: free text which used to find documents in Wikipedia
optional lang: default="en". Use it to search in a specific language part of
Wikipedia
optional load_max_docs: default=100. Use it to limit number of downloaded
documents. It takes time to download all 100 documents, so use a small number for
experiments. There is a hard limit of 300 for now.
-----------------
score 0.3242723345756531
.ipynb
.pdf
```

Step 6: Create a QA chain

Let's put it all together into a chat-like application. We want the user to ask a question, then search for relevant documents. We'll then create a prompt that includes the documents and the question so GPT can answer it (if possible).

First, we'll query the database in a similar manner to previous step. We'll use `.similarity_search()`:

```
question = "show an example of adding memory to a chain"
context_docs = db.similarity_search(question)
```

Copy

Next, we will create a prompt that contains the question and the relevant documents:

> You can think of a PromptTemplate as an fstring in python: values in curly brances are used as placeholder and will be replaced by values we pass when running the chain.

```
prompt = PromptTemplate(
    template="""Use the following pieces of context to answer the question at
the end. If you don't know the answer, just say that you don't know, don't try to
make up an answer.

<context>
{context}
</context>

Question: {question}
Helpful Answer:""",
    input_variables=["context", "question"]
)
```

Copy

To call the LLM with this prompt, we need to create an `LLMChain` and pass it an LLM and the prompt:

```
llm = ChatOpenAI(temperature=0)
qa_chain = LLMChain(llm=llm, prompt=prompt)
```

Copy

We can now call our chain like so:

```
qa_chain({"context": "<the context>", "question": "<the question>"})
```

Copy

This will return a dict with a `text` key containing the LLM response.

Instructions

1. Import the necessary pieces:

```python
from langchain.prompts import PromptTemplate
from langchain.chains.llm import LLMChain
from langchain.chat_models import ChatOpenAI
```

Copy

2. Store the question as `question`, query the database and store the result as `context_docs`
3. Create a prompt with 2 variables: `context` and `question`
4. Create a chain with an LLM and the prompt
5. Call the chain and print the result. The LLM output is in the `text` key

```python
# Import
from langchain.prompts import PromptTemplate
from langchain.chains.llm import LLMChain
from langchain.chat_models import ChatOpenAI

# Set the question variable
question = "show an example of adding memory to a chain"

# Query the database as store the results as `context_docs`
context_docs = db.similarity_search(question)

# Create a prompt with 2 variables: `context` and `question`
prompt = PromptTemplate(
    template=""""Use the following pieces of context to answer the question at the
end. If you don't know the answer, just say that you don't know, don't try to make
up an answer.

<context>
{context}
</context>

Question: {question}
Helpful Answer, formatted in markdown:""",
    input_variables=["context", "question"]
)

# Create an LLM with ChatOpenAI
llm = ChatOpenAI(temperature=0)

# Create the chain
qa_chain = LLMChain(llm=llm, prompt=prompt)

# Call the chain
result = qa_chain({
    "question": question,
    "context": "\n".join([doc.page_content for doc in context_docs])
})

# Print the result
print(result["text"])
```

~~~
```
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

conversation = ConversationChain(llm=chat, memory=ConversationBufferMemory())
conversation.run("Answer briefly. What are the first 3 colors of a rainbow?")
# -> The first three colors of a rainbow are red, orange, and yellow.
conversation.run("And the next 4?")
# -> The next four colors of a rainbow are green, blue, indigo, and violet.
'The next four colors of a rainbow are green, blue, indigo, and violet.'
```
~~~

```
This code shows an example of adding memory to a ConversationChain object using the
ConversationBufferMemory class. The memory allows the chain to persist data across
multiple calls, making it a stateful object. The example demonstrates how the chain
can remember the previous question and provide a response based on that memory.
```

```python
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

conversation = ConversationChain(llm=llm, memory=ConversationBufferMemory())
conversation.run("Answer briefly. What are the first 3 colors of a rainbow?")
# -> The first three colors of a rainbow are red, orange, and yellow.
conversation.run("And the next 4?")
```

```
'The next four colors of a rainbow are green, blue, indigo, and violet.'
```

To go further

Our little chat app is working, but can be improved. Consider the following improvements:

- **Clean up the documents.**
  Each document currently starts with useless text, and ends with a copyright notice. These texts do not provide value in our case and should be removed before embedding. Try to make an additional step after loading the `raw_documents`. In this step, iterate over all documents and find a way to remove unnecessary text.
- **Add streaming to the LLM.**
  Instead of waiting for the full response, you can get a better experience by streaming the LLM response (much like the ChatGPT web interface). Look at the `playground.ipynb` notebook for an example, or check the **docs**.
- **Return sources to the user.**
  Getting a response is nice, but linking to the relevant docs is even better. There are a few techniques to return the source documents to the user. The simplest is to print the metadata of all the `context_docs` returned by the semantic search. In this example, you could use `doc.metadata["source"]` to create a link to the langchain docs. A more advanced technique can be found **here**.
- **Add memory support to the chain.**
  You can pass a `memory` argument to the `LLMChain`. Since we have multiple input variables, make sure to specify which one the memory should use: `ConversationBufferMemory(input_key="question")`. You will also need to rerite the question before searching for documents, as the full context will not always be contained in the user input. You can find a working example in `going-further/adding-memory.ipynb`

Useful links

- Learn more about embeddings: https://txt.cohere.com/sentence-word-embeddings/
- Learn more about vector databases: https://www.pinecone.io/learn/vector-database/

LangChain caveats

While LangChain is a great tool to discover concepts and techniques, it falls short to help you deliver a production-ready application:

- Documentation is lacking. The documentation is not very complete and mostly made of examples, it lacks explanations and proper descriptions. To fully discover how a given module is working, you will need to peek in the code (**here on GitHub**)
- Too many abstractions. LangChain brings a lot of classes, some would argue too many. A `PromptTemplate` for example is pretty much the equivalent of a simple python f-string.
- Very opinionated. As soon as you want to customize a chain (or agent), you will find that it's not always the smost simple thing. Many end-up re-implementing LangChain classes or functions to suit their needs better.

LangChain is a great starting point and will let you write compelling demos very quickly. Consider these as prototypes, and not something that can be brought to production.