5
Embedding LLMs within Your Applications

This chapter kickstarts the hands-on portions of this book, focusing on how we can **leverage large language models (LLMs)** to build powerful AI applications. In fact, LLMs have introduced a whole new paradigm in software development, paving the way for new families of applications that have the peculiarity of making the communication between the user and the machine smooth and conversational. Plus, those models enhanced existing applications, such as chatbots and recommendation systems, with their unique reasoning capabilities.

Developing LLM-powered applications is becoming a key factor for enterprises to keep themselves competitive in the market, and this leads to the spreading of new libraries and frameworks that make it easier to embed LLMs within applications. Some examples are Semantic Kernel, Haystack, LlamaIndex, and LangChain. In this chapter, we are going to cover LangChain and use its modules to build hands-on examples. By the end of this chapter, you will have the technical foundations to start developing your LLM-powered applications using LangChain and open-source Hugging Face models.

In this chapter, we will cover the following topics:

- A brief note about LangChain
- Getting started with LangChain
- Working with LLMs via the Hugging Face Hub

Technical requirements

To complete the hands-on sections of this chapter, the following prerequisites are needed:

- A Hugging Face account and user access token.
- An OpenAI account and user access token.
- Python 3.7.1 or later version.
- Python packages: Make sure to have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `google-search-results`, `faiss`, and `tiktoken`. Those can be easily installed via `pip install` in your terminal.

You can find all the code and examples used in this chapter in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications

A brief note about LangChain

Just as generative AI has evolved so rapidly over the last year, so has LangChain. In the months between the writing of this book and its publication, the AI orchestrator has gone through massive changes. The most remarkable traces back to January 2024, when the first stable version of LangChain was released, introducing a new organization of packages and libraries.

It consists of the following:

- A core backbone where all the abstractions and runtime logic are stored
- A layer of third-party integrations and components

- A set of pre-built architectures and templates to leverage
- A serving layer to consume chains as APIs
- An observability layer to monitor your applications in the development, testing, and production stages

You can look at the architecture in greater detail at https://python.langchain.com/docs/get_started/introduction.

There are three packages you can install to start using LangChain:

- `langchain-core`: This contains the base abstractions and runtime for the whole LangChain ecosystem.
- `langchain-experimental`: This holds experimental LangChain code, intended for research and experimental uses.
- `langchain-community`: This contains all third-party integrations.

On top of that, there are three additional packages that we're not going to cover in this book, yet can be leveraged to monitor and maintain your LangChain applications:

- `langserve`: LangServe is a tool that lets you deploy **LangChain runnables and chains** as a REST API, making it easier to integrate LangChain applications into production environments.
- `langsmith`: Think of LangSmith as an **innovative testing framework** for evaluating language models and AI applications. It helps visualize inputs and outputs at each step in the chain, aiding understanding and intuition during development.
- `langchain-cli`: The **official command-line interface** for LangChain, it facilitates interactions with LangChain projects, including template usage and quickstarts.

Last but not least, LangChain introduced the **LangChain Expression Language** (**LCEL**) to enhance the efficiency and flexibility of text processing tasks.

Key features of LCEL include:

- **Streaming asynchronous support**: This allows for the efficient handling of data streams.
- **Batch support**: This enables processing data in batches.
- **Parallel execution**: This enhances performance by executing tasks concurrently.
- **Retries and fallbacks**: This ensures robustness by handling failures gracefully.
- **Dynamically routing logic**: This allows logic flow based on input and output.
- **Message history**: This keeps track of interactions for context-aware processing.

We are not going to cover LCEL in this book; however, all the code samples can be converted into LCEL if you want to speed up your development and leverage its native integration with the end-to-end LangChain development stack.

Important note

Before we start working with LangChain, it is important to note that all packages are versioned slightly differently, yet all releases are cut with high frequency by a maintainer with a clearer communication strategy for breaking changes.

In the upcoming chapters, you will see some packages that have been moved, for example, to the `experimental` package, meaning that they are more prone to ex-

perimental uses. Similarly, some third-party integrations have been moved to the `community` package.

Starting from the next section, we are going to cover the backbone concepts – such as memory, VectorDB, and agents – that remain solid in the LangChain framework and, more generally, in the landscape of LLM development.

Getting started with LangChain

As introduced in *Chapter 2*, LangChain is a lightweight framework meant to make it easier to integrate and orchestrate LLMs and their components within applications. It is mainly Python based, yet it recently extended its support to JavaScript and TypeScript.

In addition to LLM integration (which we will cover in an upcoming dedicated section), we saw that LangChain offers the following main components:

- Models and prompt templates
- Data connections
- Memory
- Chains
- Agents

These components are illustrated in the following diagram:

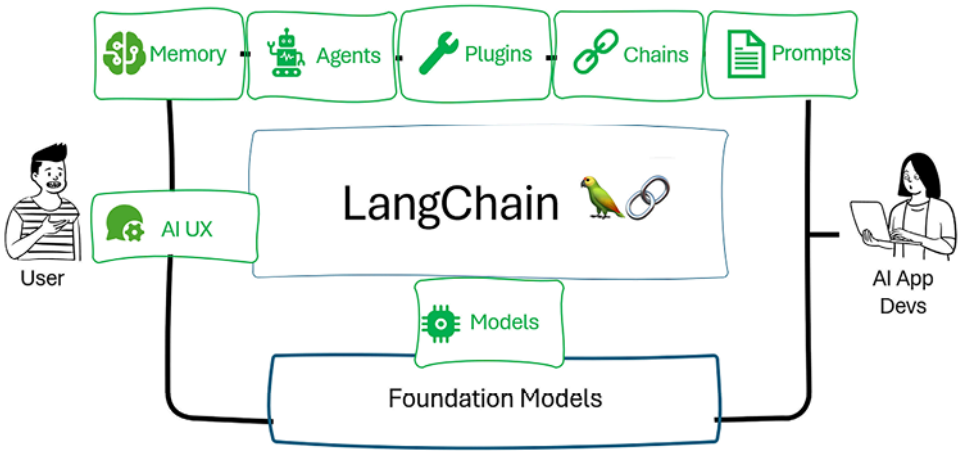


Figure 5.1: LangChain's components

The next sections will take a deep dive into each of these components.

Models and prompts

LangChain offers more than 50 integrations with third-party vendors and platforms, including **OpenAI**, Azure OpenAI, Databricks, and MosaicML, as well as the integration with the Hugging Face Hub and the world of open-source LLMs. In *Part 2* of this book, we will be trying various LLMs, both proprietary and open-source, and leveraging LangChain's integrations.

Just to provide an example, let's see how easy it is to consume the OpenAI GPT-3 model (you can retrieve your OpenAI API key at https://platform.openai.com/account/api-keys):

```
from langchain.llms import OpenAI
llm = OpenAI(openai_api_key="your-api-key")
print(llm('tell me a joke'))
```

Here is the corresponding output:

```
Q: What did one plate say to the other plate?
A: Dinner's on me!
```

Note

While running examples with LLMs, the output will vary at each run, due to the stochasticity of the models themselves. If you want to reduce the margin of variations in your output, you can make your model more "deterministic" by tuning the temperature hyperparameter. This parameter ranges from 0 (deterministic) to 1 (stochastic).

By default, the **OpenAI** module uses the `gpt-3.5-turbo-instruct` as a model. You can specify the model you want to use by passing the model's name as a parameter.

As said previously, we will dive deeper into LLMs in the next section; so, for now, let's focus on prompts. There are two main components related to LLM prompts and prompts design/engineering:

- **Prompt templates**: A prompt template is a component that defines how to generate a prompt for a language model. It can include variables, placeholders, prefixes, suffixes, and other elements that can be customized according to the data and the task.

For example, suppose you want to use a language model to generate a translation from one language to another. You can use a prompt template like this:

```
Sentence: {sentence}
Translation in {language}:
```

`{sentence}` is a variable that will be replaced by the actual text. `Translation in {language}:` is a prefix that indicates the task and the expected output format.

You can easily implement this template as follows:

```
from langchain import PromptTemplate
template = """Sentence: {sentence}
Translation in {language}:"""
prompt = PromptTemplate(template=template, input_variables=["sentence", "language"])
print(prompt.format(sentence = "the cat is on the table", language = "spanish"))
```

Here is the output:

```
Sentence: the cat is on the table
Translation in spanish:
```

Generally speaking, prompt templates tend to be agnostic with respect to the LLM you might decide to use, and it is adaptable to both completion and chat models.

Definition

A completion model is a type of LLM that takes a text input and generates a text output, which is called a completion. The completion model tries to continue the prompt in a coherent and relevant way, according to the task and the data it was trained on. For example, a completion model can generate summaries, translations, stories, code, lyrics, and more, depending on the prompt.

A chat model is a special kind of completion model that is designed to generate conversational responses. A chat model takes a list of messages as input, where each message has a role (either system, user, or assistant) and content. The chat model tries to generate a new message for the assistant role, based on the previous messages and the system instruction.

The main difference between completion and chat models is that completion models expect a single text input as a prompt, while chat models expect a list of messages as input.

- **Example selector**: An example selector is a component in LangChain that allows you to choose which examples to include in a prompt for a language model. A prompt is a text input that guides the language model to produce a desired output. Examples are pairs of inputs and outputs that demonstrate the task and the format of the output as follows:

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

The idea recalls the concept of few-shot learning we covered in *Chapter 1*.

LangChain offers the example selector class called `BaseExampleSelector` that you can import and modify as you wish. You can find the API reference at https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/.

Data connections

Data connections refer to the building blocks needed to retrieve the additional non-parametric knowledge we want to provide the model with.

The idea is to cover the typical flow of incorporating user-specific data into applications that are made of five main blocks, as illustrated in the following figure:

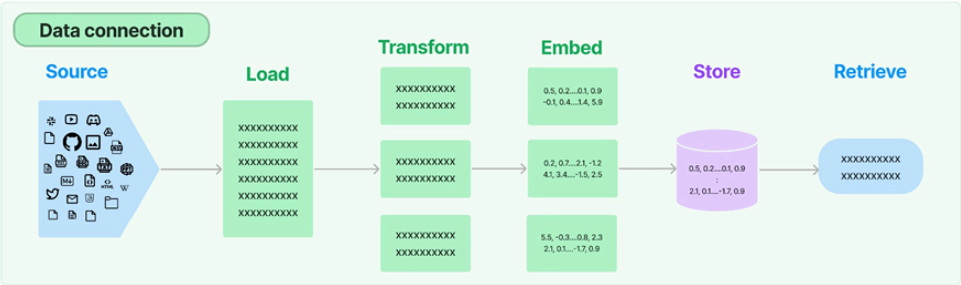


Figure 5.2: Incorporating user-specific knowledge into LLMs (source: https://python.langchain.com/docs/modules/data_connection/)

Those blocks are addressed with the following LangChain tools:

- **Document loaders**: They are in charge of loading documents from different sources such as CSV, file directory, HTML, JSON, Markdown, and PDF. Document loaders expose a `.load` method for loading data as documents from a configured source. The output is a `Document` object that contains a piece of text and associated metadata.

For example, let's consider a sample CSV file to be loaded (you can find the whole code in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications):

```
from langchain.document_loaders.csv_loader import CSVLoader
loader = CSVLoader(file_path='sample.csv')
data = loader.load()
print(data)
```

Here is the output:

```
[Document(page_content='Name: John\nAge: 25\nCity: New York', metadata={'source': 'sample.csv', '
```

- **Document transformers**: After importing your documents, it's common to modify them to better match your needs. A basic instance of this is breaking down a lengthy document into smaller chunks that fit your model's context window. Within LangChain, there are various pre-built document transformers available called **text splitters**. The idea of text splitters is to make it easier to split documents into chunks that are semantically related so that we do not lose context or relevant information.

With text splitters, you can decide how to split the text (for example, by character, heading, token, and so on) and how to measure the length of the chunk (for example, by number of characters).

For example, let's split a document using the `RecursiveCharacterTextSplitter` module, which operates at a character level. For this purpose, we will be using a `.txt` file about mountains (you can find the whole code in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications):

```
with open('mountain.txt') as f:
    mountain = f.read()
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 100, #number of characters for each chunk
    chunk_overlap  = 20,#number of characters overlapping between a preceding and following chunk
    length_function = len #function used to measure the number of characters
)
texts = text_splitter.create_documents([mountain])
print(texts[0])
print(texts[1])
print(texts[2])
```

Here, `chunk_size` refers to the number of characters in each chunk while `chunk_overlap` represents the number of characters overlapping between successive chunks. Here is the

output:

```
page_content="Amidst the serene landscape, towering mountains stand as majestic guardians of natu
page_content='The crisp mountain air carries whispers of tranquility, while the rustling leaves c
```

- **Text embedding models**: In *Chapter 1*, in the *Under the hood of an LLM* section, we introduced the concept of embedding as a way to represent words, subwords, or characters in a continuous vector space.

Embeddings are the key step in incorporating non-parametric knowledge into LLMs. In fact, once properly stored in a VectorDB (which will be covered in the next section), they become the non-parametric knowledge against which we can measure the distance of a user's query.

To get started with embedding, you will need an embedding model.

Then, LangChain offers the `Embedding` class with two main modules, which address the embedding of, respectively, the non-parametric knowledge (multiple input text) and the user query (single input text).

For example, let's consider the embeddings using the **OpenAI** embedding model `text-embedding-ada-002` (for more details about OpenAI embedding models, you can refer to the official documentation at https://platform.openai.com/docs/guides/embeddings/what-are-embeddings):

```
from langchain.embeddings import OpenAIEmbeddings
from dotenv import load_dotenv
load_dotenv()
os.environ["OPENAI_API_KEY"]
embeddings_model = OpenAIEmbeddings(model ='text-embedding-ada-002' )
embeddings = embeddings_model.embed_documents(
    [
        "Good morning!",
        "Oh, hello!",
        "I want to report an accident",
        "Sorry to hear that. May I ask your name?",
        "Sure, Mario Rossi."
    ]
)
print("Embed documents:")
print(f"Number of vector: {len(embeddings)}; Dimension of each vector: {len(embeddings[0])}")
embedded_query = embeddings_model.embed_query("What was the name mentioned in the conversation?")
print("Embed query:")
print(f"Dimension of the vector: {len(embedded_query)}")
print(f"Sample of the first 5 elements of the vector: {embedded_query[:5]}")
```

Here is the output:

```
Embed documents:
Number of vector: 5; Dimension of each vector: 1536
Embed query:
Dimension of the vector: 1536
Sample of the first 5 elements of the vector: [0.00538721214979887, -0.0005941778072156012, 0.038
```

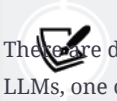
Once we have both documents and the query embedded, the next step will be to compute the similarity between the two elements and retrieve the most suitable information from the docu-

ment embedding. We will see the details of this when talking about vector stores.

- **Vector stores**: A vector store (or VectorDB) is a type of database that can store and search over unstructured data, such as text, images, audio, or video, by using embeddings. By using embeddings, vector stores can perform a fast and accurate similarity search, which means finding the most relevant data for a given query.

Definition

Similarity is a measure of how close or related two vectors are in a vector space. In the context of LLMs, vectors are numerical representations of sentences, words, or documents that capture their semantic meaning, and the distance between those vectors should be representative of their semantic similarity.



There are different ways to measure similarity between vectors, and while working with LLMs, one of the most popular measures in use is cosine similarity.

This is the cosine of the angle between two vectors in a multidimensional space. It is computed as the dot product of the vectors divided by the product of their lengths. Cosine similarity is insensitive to scale and location, and it ranges from -1 to 1, where 1 means identical, 0 means orthogonal, and -1 means opposite.

The following is an illustration of the typical flow while using a vector store.

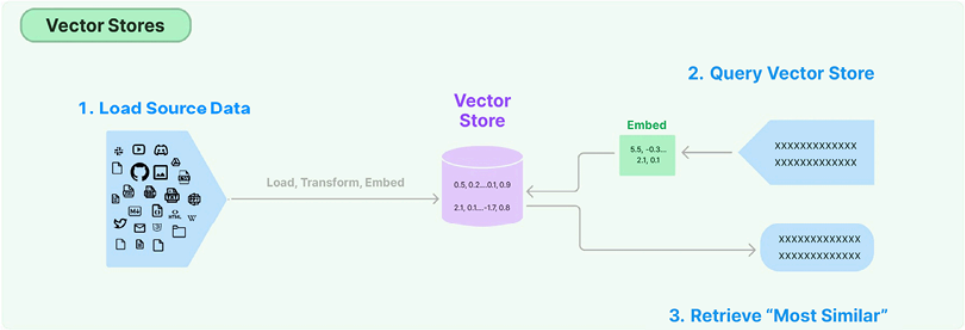


Figure 5.3: Sample architecture of a vector store (source: https://python.langchain.com/docs/modules/data_connection/vectorstores/)

LangChain offers more than 40 integrations with third-party vector stores. Some examples are **Facebook AI Similarity Search (FAISS)**, Elasticsearch, MongoDB Atlas, and Azure Search. For an exhaustive list and descriptions of all the integrations, you can check the official documentation at https://python.langchain.com/docs/integrations/vectorstores/.

As an example, let's leverage the FAISS vector store, which has been developed by Meta AI research for efficient similarity search and clustering of dense vectors. We are going to leverage the same `dialogue.txt` file saved in the previous section:

```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from dotenv import load_dotenv
load_dotenv()
os.environ["OPENAI_API_KEY"]
```

```
# Load the document, split it into chunks, embed each chunk and load it into the vector store.
raw_documents = TextLoader('dialogue.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=50, chunk_overlap=0, separator = "\n",)
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEmbeddings())
```

Now that we've embedded and saved the non-parametric knowledge, let's also embed a user's query so that it can be used to search the most similar text chunk using cosine similarity as a measure:

```
query = "What is the reason for calling?"
docs = db.similarity_search(query)
print(docs[0].page_content)
```

The following is the output:

```
I want to report an accident
```

As you can see, the output is the piece of text that is more likely to contain the answer to the question. In an end-to-end scenario, it will be used as context to the LLM to generate a conversational response.

- **Retrievers**: A retriever is a component in LangChain that can return documents relevant to an unstructured query, such as a natural language question or a keyword. A retriever does not need to store the documents itself, but only to retrieve them from a source. A retriever can use different methods to find relevant documents, such as keyword matching, semantic search, or ranking algorithms.

The difference between a retriever and a vector store is that a retriever is more general and flexible than a vector store. A retriever can use any method to find relevant documents, while a vector store relies on embeddings and similarity metrics. A retriever can also use different sources of documents, such as web pages, databases, or files, while a vector store needs to store the data itself.

However, a vector store can also be used as the backbone of a retriever if the data is embedded and indexed by a vector store. In that case, the retriever can use the vector store to perform a similarity search over the embedded data and return the most relevant documents. This is one of the main types of retrievers in LangChain, and it is called a vector store retriever.

For example, let's consider the FAISS vector store we previously initialized and "mount" a retriever on top of that:

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
retriever = db.as_retriever()
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff", retriever=retriever)
query = "What was the reason of the call?"
qa.run(query)
```

Here is the output:

```
' The reason for the call was to report an accident.'
```

Overall, data connection modules offer a plethora of integrations and pre-built templates that make it easier to manage the flow of your LLM-powered application. We will see some concrete applications of these building blocks in the upcoming chapters, but in the next section, we are going to take a deep dive into another one of LangChain's main components.

Memory

In the context of LLM-powered applications, memory allows the application to keep references to user interactions, both in the short and long term. For example, let's consider the well-known ChatGPT. While interacting with the application, you have the possibility to ask follow-up questions referencing previous interactions without explicitly telling the model.

Plus, all conversations are saved into threads, so that, if you want to follow up on a previous conversation, you can re-open the thread without providing ChatGPT with all the contexts. This is made possible thanks to ChatGPT's ability to store users' interactions into a memory variable and use this memory as context while addressing follow-up questions.

LangChain offers several modules for designing your memory system within your applications, enabling it with both reading and writing skills.

The first step to do with your memory system is to actually store your human interactions somewhere. To do so, you can leverage numerous built-in memory integrations with third-party providers, including Redis, Cassandra, and Postgres.

Then, when it comes to defining how to query your memory system, there are various memory types you can leverage:

- **Conversation buffer memory**: This is the "plain vanilla" memory type available in LangChain. It allows you to store your chat messages and extract them in a variable.
- **Conversation buffer window memory**: It is identical to the previous one, with the only difference being allowing a sliding window over only *K* interactions so that you can manage longer chat history over time.
- **Entity memory**: Entity memory is a feature of LangChain that allows the language model to remember given facts about specific entities in a conversation. An entity is a person, place, thing, or concept that can be identified and distinguished from others. For example, in the sentence "Deven and Sam are working on a hackathon in Italy," Deven and Sam are entities (person), as well as hackathon (thing) and Italy (place).

Entity memory works by extracting information on entities from the input text using an LLM. It then builds up its knowledge about that entity over time by storing the extracted facts in a memory store. The memory store can be accessed and updated by the language model whenever it needs to recall or learn new information about an entity.

- **Conversation knowledge graph memory**: This type of memory uses a knowledge graph to recreate memory.

Definition

A knowledge graph is a way of representing and organizing knowledge in a graph structure, where nodes are entities and edges are relationships between them. A knowledge graph can store and integrate data from various sources, and encode the semantics and context of the data. A knowledge graph can also support various tasks, such as search, question answering, reasoning, and generation.

Another example of a knowledge graph is DBpedia, which is a community project that extracts structured data from Wikipedia and makes it available on the web. DBpedia covers

```
topics such as geography, music, sports, and films, and provides links to other datasets like
GeoNames and WordNet.
```

You can use this type of memory to save the input and output of each conversation turn as knowledge triplets (such as subject, predicate, and object) and then use them to generate relevant and consistent responses based on the current context. You can also query the knowledge graph to get the current entities or the history of the conversation.

- **Conversation summary memory**: When it comes to longer conversations to be stored, this type of memory can be very useful, since it creates a summary of the conversation over time (leveraging an LLM).
- **Conversation summary buffer memory**: This type of memory combines the ideas behind buffer memory and conversation summary memory. It keeps a buffer of recent interactions in memory, but rather than just completely flushing old interactions (as occurs for the conversation buffer memory) it compiles them into a summary and uses both.
- **Conversation token buffer memory**: It is similar to the previous one, with the difference that, to determine when to start summarizing the interactions, this type of memory uses token lengths rather than the number of interactions (as occurs in summary buffer memory).
- **Vector store-backed memory**: This type of memory leverages the concepts of embeddings and vector stores previously covered. It is different from all the previous memories since it stores interactions as vectors, and then retrieves the top *K* most similar texts every time it is queried, using a retriever.

LangChain provides specific modules for each of those memory types. Let's consider an example with the conversation summary memory, where we will also need an LLM to generate the summary of the interactions:

```
from langchain.memory import ConversationSummaryMemory, ChatMessageHistory
from langchain.llms import OpenAI
memory = ConversationSummaryMemory(llm=OpenAI(temperature=0))
memory.save_context({"input": "hi, I'm looking for some ideas to write an essay in AI"}, {"output'
memory.load_memory_variables({})
```

Here is the output:

```
{'history': '\nThe human asked for ideas to write an essay in AI and the AI suggested writing on 1
```

As you can see, the memory summarized the conversation, leveraging the **OpenAI** LLM we initialized.

There is no recipe to define which memory to use within your applications; however, there are some scenarios that might be particularly suitable for specific memories. For example, a knowledge graph memory is useful for applications that need to access information from a large and diverse corpus of data and generate responses based on semantic relationships, while a conversation summary buffer memory could be suitable for creating conversational agents that can maintain a coherent and consistent context over multiple turns, while also being able to compress and summarize the previous dialogue history.

Chains

Chains are predetermined sequences of actions and calls to LLMs that make it easier to build complex applications that require combining LLMs with each other or with other components.

LangChain offers four main types of chain to get started with:

- **LLMChain**: This is the most common type of chain. It consists of a prompt template, an LLM, and an optional **output parser**.

Definition

An output parser is a component that helps structure language model responses. It is a class that implements two main methods: `get_format_instructions` and `parse`. The `get_format_instructions` method returns a string containing instructions for how the output of a language model should be formatted. The `parse` method takes in a string (assumed to be the response from a language model) and parses it into some structure, such as a dictionary, a list, or a custom object.

This chain takes multiple input variables, uses `PromptTemplate` to format them into a prompt, passes it to the model, and then uses `OutputParser` (if provided) to parse the output of the LLM into a final format.

For example, let's retrieve the prompt template we built in the previous section:

```
from langchain import PromptTemplate
template = """Sentence: {sentence}
Translation in {language}:"""
prompt = PromptTemplate(template=template, input_variables=["sentence", "language"])
```

Now, let's put it into an LLMChain:

```
from langchain import OpenAI, LLMChain
llm = OpenAI(temperature=0)
llm_chain = LLMChain(prompt=prompt, llm=llm)
llm_chain.predict(sentence="the cat is on the table", language="spanish")
```

Here is the output:

```
' El gato está en la mesa.'
```

- **RouterChain**: This is a type of chain that allows you to route the input variables to different chains based on some conditions. You can specify the conditions as functions or expressions that return a Boolean value. You can also specify the default chain to use if none of the conditions are met.

For example, you can use this chain to create a chatbot that can handle different types of requests, such as planning an itinerary or booking a restaurant reservation. To achieve this goal, you might want to differentiate two different prompts, depending on the type of query the user will make:

```
itinerary_template = """You are a vacation itinerary assistant. \
You help customers finding the best destinations and itinerary. \
You help customer screating an optimized itinerary based on their preferences.
Here is a question:
{input}"""
restaurant_template = """You are a restaurant booking assistant. \
You check with customers number of guests and food preferences. \
```

```
You pay attention whether there are special conditions to take into account.
Here is a question:
{input}"""
```

Thanks to RouterChain, we can build a chain that is able to activate a different prompt depending on the user's query. I won't post the whole code here (you can find the notebook on the book's GitHub at https://github.com/PacktPublishing/Building-LLM-Powered-Applications), but you can see a sample output of how the chain reacts to two different user's queries:

```
print(chain.run("I'm planning a trip from Milan to Venice by car. What can I visit in between?"))
```

Here is the output:

```
> Entering new MultiPromptChain chain...
itinerary: {'input': "I'm planning a trip from Milan to Venice by car. What attractions can I visi
> Finished chain.
Answer:
There are many attractions that you can visit while traveling from Milan to Venice by car. Some of
```

Here it is with a second query:

```
print(chain.run("I want to book a table for tonight"))
```

Here is the output:

```
> Entering new MultiPromptChain chain...
restaurant: {'input': 'I want to book a table for tonight'}
> Finished chain.
. How many people are in your party?
Hi there! How many people are in your party for tonight's reservation?
```

- **SequentialChain**: This is a type of chain that allows you to execute multiple chains in a sequence. You can specify the order of the chains and how they pass their outputs to the next chain. The simplest module of a sequential chain, takes by default the output of one chain as the input of the next chain. However, you can also use a more complex module to have more flexibility to set input and output among chains.

As an example, let's consider an AI system that is meant to first generate a joke on a given topic, and then translate it in to another language. To do so, we will first create two chains:

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
llm = OpenAI(temperature=.7)
template = """You are a comedian. Generate a joke on the following {topic}
Joke:"""
prompt_template = PromptTemplate(input_variables=["topic"], template=template)
joke_chain = LLMChain(llm=llm, prompt=prompt_template)
template = """You are translator. Given a text input, translate it to {language}
Translation:"""
.prompt_template = PromptTemplate(input_variables=["language"], template=template)
translator_chain = LLMChain(llm=llm, prompt=prompt_template)
```

Now, let's combine them using the `SimpleSequentialChain` module:

```
# This is the overall chain where we run these two chains in sequence.
from langchain.chains import SimpleSequentialChain
overall_chain = SimpleSequentialChain(chains=[joke_chain, translator_chain], verbose=True)
translated_joke = overall_chain.run("Cats and Dogs")
```

Here is the output:

```
> Entering new SimpleSequentialChain chain...
Why did the cat cross the road? To prove to the dog that it could be done!
 ¿Por qué cruzó el gato la carretera? ¡Para demostrarle al perro que se podía hacer!
> Finished chain.
```

- **TransformationChain**: This is a type of chain that allows you to transform the input variables or the output of another chain using some functions or expressions. You can specify the transformation as a function that takes the input or output as an argument and returns a new value, as well as specify the output format of the chain.

For example, let's say we want to summarize a text, but before that, we want to rename one of the protagonists of the story (a cat) as "Silvester the Cat." As a sample text, I asked Bing Chat to generate a story about cats and dogs (you can find the whole `.txt` file in the GitHub repository of this book):

```
from langchain.chains import TransformChain, LLMChain, SimpleSequentialChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
transform_chain = TransformChain(
    input_variables=["text"], output_variables=["output_text"], transform=rename_cat
)
template = """Summarize this text:
{output_text}
Summary:"""
prompt = PromptTemplate(input_variables=["output_text"], template=template)
llm_chain = LLMChain(llm=OpenAI(), prompt=prompt)
sequential_chain = SimpleSequentialChain(chains=[transform_chain, llm_chain])
sequential_chain.run(cats_and_dogs)
```

As you can see, we've combined a simple sequential chain with a transformation chain, where we set as a transformation function the `rename_cat` function (you can see the whole code in the GitHub repository).

The output is the following:

```
" Silvester the Cat and a dog lived together but did not get along. Silvester the Cat played a pra
```

Overall, LangChain chains are a powerful way to combine different language models and tasks into a single workflow. Chains are flexible, scalable, and easy to use, and they enable users to leverage the power of language models for various purposes and domains. Starting from the next chapter, we are going to see chains in action in concrete use cases, but before getting there, we need to cover the last component of LangChain: agents.

Agents

Agents are entities that drive decision-making within LLM-powered applications. They have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal: in fact, while in a chain, the sequence of actions is hardcoded, in agents, the LLM is used as the reasoning engine with the goal of planning and executing the right actions in the right order.

A core concept while talking about agents is that of tools. In fact, an agent might be good at planning all the right actions to fulfill a user's query, but what if it cannot actually execute them, since it is missing information or executive power? For example, imagine I want to build an agent that is capable of answering my questions by searching the web. By itself, the agent has no access to the web, so I need to provide it with this tool. I will do so by using SerpApi (the Google Search API) integration provided by LangChain (you can retrieve your API key at https://serpapi.com/dashboard).

Let's see it in Python:

```
from langchain import SerpAPIWrapper
from langchain.agents import AgentType, initialize_agent
from langchain.llms import OpenAI
from langchain.tools import BaseTool, StructuredTool, Tool, tool
import os
from dotenv import load_dotenv
load_dotenv()
os.environ["SERPAPI_API_KEY"]
search = SerpAPIWrapper()
tools = [Tool.from_function(
        func=search.run,
        name="Search",
        description="useful for when you need to answer questions about current events"
    )]
agent = initialize_agent(tools, llm = OpenAI(), agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbo
agent.run("When was Avatar 2 released?")
```

The following is the output:

```
> Entering new AgentExecutor chain...
 I need to find out when Avatar 2 was released.
Action: Search
Action Input: "Avatar 2 release date"
Observation: December 16, 2022
Thought: I now know the final answer.
Final Answer: Avatar 2 was released on December 16, 2022.
> Finished chain.
'Avatar 2 was released on December 16, 2022.'
```

Note that, while initializing my agent, I set the agent type as `ZERO_SHOT_REACT_DESCRIPTION`. This is one of the configurations we can pick and, specifically, it configures the agent to decide which tool to pick based solely on the tool's description with a ReAct approach:

Definition

The ReAct approach is a way of using LLMs to solve various language reasoning and decision-making tasks. It was introduced in the paper *ReAct: Synergizing*

Reasoning and Acting in Language Models by Shunyu Yao et al., back in October 2022.

The ReAct approach prompts LLMs to generate both verbal reasoning traces and text actions in an interleaved manner, allowing for greater synergy between the two. Reasoning traces help the model to plan, track, and update its actions, as well as handle exceptions. Actions allow the model to interact with external sources, such as knowledge bases or environments, to gather additional information.

On top of this configuration, LangChain also offers the following types of agents:

- **Structured input ReAct**: This is an agent type that uses the ReAct framework to generate natural language responses based on structured input data. The agent can handle different types of input data, such as tables, lists, or key-value pairs. The agent uses a language model and a prompt to generate responses that are informative, concise, and coherent.
- **OpenAI Functions**: This is an agent type that uses the OpenAI Functions API to access various language models and tools from OpenAI. The agent can use different functions, such as GPT-3, Codex, DALL-E, CLIP, or ImageGPT. The agent uses a language model and a prompt to generate requests to the OpenAI Functions API and parse the responses.
- **Conversational**: This is an agent type that uses a language model to engage in natural language conversations with the user. The agent can handle different types of conversational tasks, such as chit-chat, question answering, or task completion. The agent uses a language model and a prompt to generate responses that are relevant, fluent, and engaging.
- **Self ask with search**: This is an agent type that uses a language model to generate questions for itself and then search for answers on the web. The agent can use this technique to learn new information or test its own knowledge.
- **ReAct document store**: This is an agent type that uses the ReAct framework to generate natural language responses based on documents stored in a database. The agent can handle different types of documents, such as news articles, blog posts, or research papers.
- **Plan-and-execute agents**: This is an experimental agent type that uses a language model to choose a sequence of actions to take based on the user's input and a goal. The agent can use different tools or models to execute the actions it chooses. The agent uses a language model and a prompt to generate plans and actions and then uses `AgentExecutor` to run them.

LangChain agents are pivotal whenever you want to let your LLMs interact with the external world. Plus, it is interesting to see how agents leverage LLMs not only to retrieve and generate responses, but also as reasoning engines to plan an optimized sequence of actions.

Together with all the LangChain components covered in this section, agents can be the core of LLM-powered applications, as we will see in the next chapters. In the next section, we are going to shift toward the world of open-source LLMs, introducing the Hugging Face Hub and its native integration with LangChain.

Working with LLMs via the Hugging Face Hub

Now that we are familiar with LangChain components, it is time to start using our LLMs. If you want to use open-source LLMs, leveraging the Hugging Face Hub integration is extremely versatile. In fact, with just one access token you can leverage all the open-source LLMs available in Hugging Face's repositories.

As it is a non-production scenario, I will be using the free Inference API; however, if you are meant to build production-ready applications, you can easily scale to the Inference Endpoint,

which grants you a dedicated and fully managed infrastructure to host and consume your LLMs.

So, let's see how to start integrating LangChain with the Hugging Face Hub.

Create a Hugging Face user access token

To access the free Inference API, you will need a user access token, the credential that allows you to run the service. The following are the steps to activate the user access token:

1. **Create a Hugging Face account**: You can create a Hugging Face account for free at https://huggingface.co/join.
2. **Retrieve your user access token**: Once you have your account, go to the upper-right corner of your profile and go to **Settings** | **Access Tokens**. From that tab, you will be able to copy your secret token and use it to access Hugging Face models.

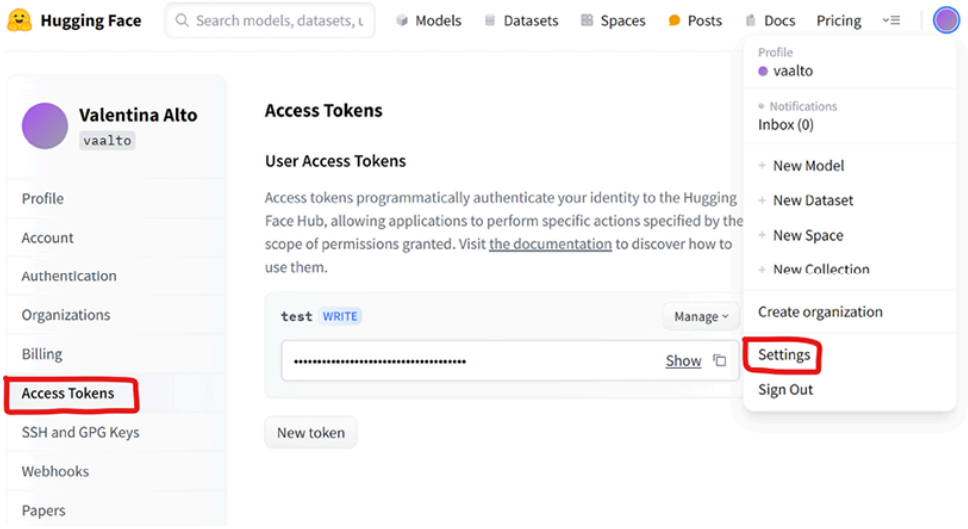


Figure 5.4: Retrieving access tokens from the Hugging Face account (source: https://huggingface.co/settings/tokens)

3. **Set permissions**: Access tokens enable users, applications, and notebooks to perform specific actions based on their assigned roles. There are two available roles:
    1. **Read:** This allows tokens to provide read access to repositories you have permission to read. This includes public and private repositories owned by you or your organization. This role is suitable for tasks like downloading private models or inference.
    2. **Write:** In addition to read access, tokens with this role grant write access to repositories where you have writing privileges. This token is useful for activities like training models or updating model cards.

In our series of use cases, we will keep a write permission on our token.

4. **Managing your user access token**: Within your profile, you can create and manage multiple access tokens, so that you can also differentiate permissions. To create a new token, you can click on the **New token** button:

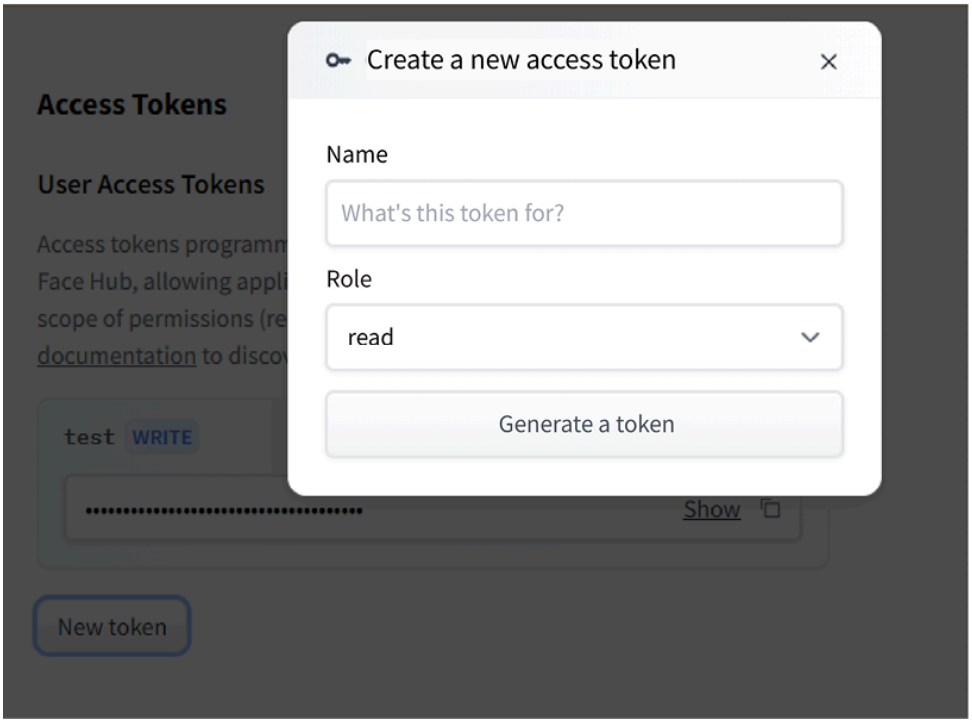


Figure 5.5: Creating a new token

5. Finally, at any time, you can delete or refresh your token under the **Manage** button:

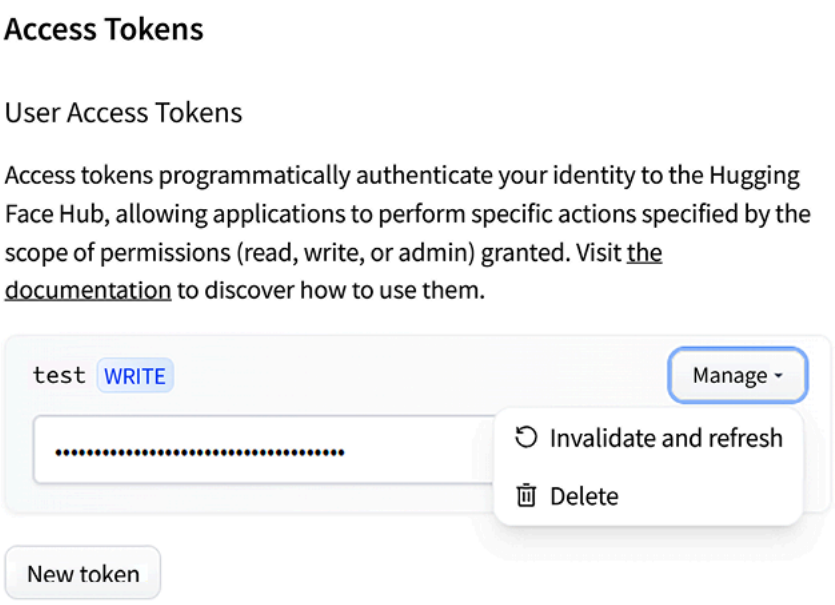


Figure 5.6: Managing tokens

It is important not to leak your token, and a good practice is to periodically regenerate it.

Storing your secrets in an .env file

With our user access token generated in the previous section, we have the first secret to be managed.

Definition

Secrets are data that needs to be protected from unauthorized access, such as passwords, tokens, keys, and credentials. Secrets are used to authenticate and authorize requests to API endpoints, as well as to encrypt and decrypt sensitive data.

Throughout this hands-on portion of the book, we will keep all our secrets within an `.env` file.

Storing Python secrets in an `.env` file is a common practice to enhance security and maintainability in projects. To do this, create a file named `.env` in your project directory and list your sensitive information as key-value pairs: in our scenario, we will have `HUGGINGFACEHUB_API_TOKEN="your_user_access_token"`. This file should be added to your project's `.gitignore` to prevent accidental exposure.

To access these secrets in your Python code, use the `python-dotenv` library to load the `.env` file's values as environment variables. You can easily install it in your terminal via `pip install python-dotenv`.

This approach keeps sensitive data separate from your code base and helps ensure that confidential information remains confidential throughout the development and deployment processes.

Here, you can see an example of how to retrieve your access token and set it as an environmental variable:

```
import os
from dotenv import load_dotenv
load_dotenv()
os.environ["HUGGINGFACEHUB_API_TOKEN"]
```

Note that, by default, `load_dotenv` will look for the `.env` file in the current working directory; however, you can also specify the path to your secrets file:

```
from dotenv import load_dotenv
from pathlib import Path
dotenv_path = Path('path/to/.env')
load_dotenv(dotenv_path=dotenv_path)
```

Now that we have all the ingredients to start coding, it is time to try out some open-source LLMs.

Start using open-source LLMs

The nice thing about the Hugging Face Hub integration is that you can navigate its portal and decide, within the model catalog, what to use. Models are also clustered per category (**Computer Vision**, **Natural Language Processing**, **Audio**, and so on) and, within each category, per capability (within **Natural Language Processing**, we have summarization, classification, Q&A, and so on), as shown in the following screenshot:

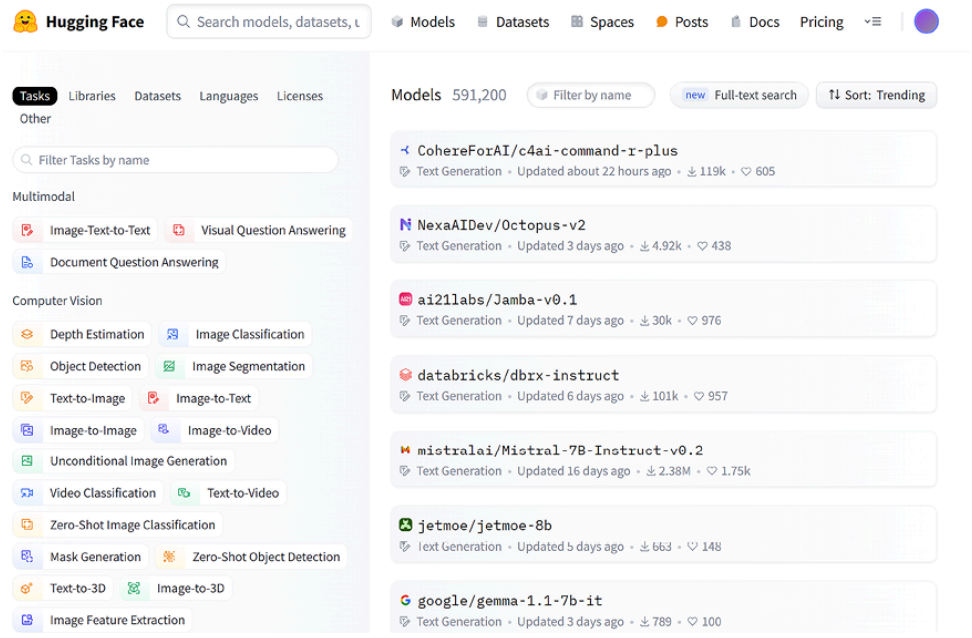


Figure 5.7: Home page of Hugging Face's model catalog

Since we are interested in LLMs, we will focus on the text generation category. For this first experiment, let's try Falcon LLM-7B:

```
from langchain import HuggingFaceHub
repo_id = "tiiuae/falcon-7b-instruct"
llm = HuggingFaceHub(
    repo_id=repo_id, model_kwargs={"temperature": 0.5, "max_length": 1000}
)
print(llm("what was the first disney movie?"))
```

Here is the corresponding output:

```
The first Disney movie was 'Snow White and the Seven Dwarfs'
```

As you can see, with just a few lines of code, we integrated an LLM from the Hugging Face Hub. With analogous code, you can test and consume all the LLMs available in the Hub.

Note that, throughout this book, we will be leveraging specific models for each application, both proprietary and open source. However, the idea is that you can use the model you prefer by simply initializing it as the main LLM and running the code as it is, simply changing the LangChain LLM integration. This is one of the main advantages of LLM-powered applications since you don't have to change the whole code to adapt to different LLMs.

Summary

In this chapter, we dove deeper into the fundamentals of LangChain, since it will be the AI orchestrator used in the upcoming chapters: we got familiar with LangChain components such as memory, agents, chains, and prompt templates. We also covered how to start integrating LangChain with the Hugging Face Hub and its model catalog, and how to use the available LLMs and start embedding them into your code.

From now on, we will look at a series of concrete end-to-end use cases, starting from a semantic Q&A search app, which we are going to develop in the next chapter.

References

- LangChain's integration with OpenAI – https://python.langchain.com/docs/integrations/llms/openai
- LangChain's prompt templates – https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/
- LangChain's vector stores – https://python.langchain.com/docs/integrations/vectorstores/
- FAISS index – https://faiss.ai/
- LangChain's chains – https://python.langchain.com/docs/modules/chains/
- ReAct approach – https://arxiv.org/abs/2210.03629
- LangChain's agents – https://python.langchain.com/docs/modules/agents/agent_types/
- Hugging Face documentation – https://huggingface.co/docs
- LangChain Expression Language (LCEL) – https://python.langchain.com/docs/expression_language/
- LangChain stable version – https://blog.langchain.dev/langchain-v0-1-0/

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

https://packt.link/llm