Module 1 Introduction - Basics of Retrieval Augmented Generation with Langchain and LlamaIndex

This module covers the basic concepts of Langchain and Llamaindex and prepares you to build a basic RAG application with both frameworks and help you decide which tool to select for your use case (spoiler: they both have their utility!). This is a recap of the Langchain concepts covered in the earlier **LangChain & Vector Databases in Production** course, together with a brief introduction to the Llamaindex framework. It also contains a clear summary of the different strengths and focus of each framework. This course will be focussed on advanced RAG topics so we recommend taking our earlier course and reading examples from LangChain and Llamaindex documentation to complement this module.

LangChain: Basic Concepts Recap

In this lesson, students will recap on the functionalities and components of LangChain, a framework designed to work with generative AI and LLM-based applications. This material was covered in depth with code and project examples in our earlier course **LangChain & Vector Databases in Production.** Students will be introduced to crucial preprocessing techniques like document loading and chunking, understanding the indexing of document segments and embedding models, as well as the structure and functionality of chains, memory modules, and vector stores. Additionally, students will gain insight into working with chat models, LLMs, embedding models, and constructing sequential chains to automate complex interactions.

LlamaIndex Introduction: Precision and Simplicity in Information Retrieval

In this lesson, students will learn about the LlamaIndex framework, designed to enhance the capabilities of Large Language Models by integrating them with Retrieval-Augmented Generation (RAG) systems. The framework allows LLM-based applications to fetch accurate and relevant information using vector stores, connectors, nodes, and index types for better-informed responses. The lesson covers vector stores and their importance in semantic search, the role of data connectors and LlamaHub in data ingestion, the creation of node objects from documents, and the indexing of data for quick retrieval. Students will also learn about the practical construction and usage of query engines, routers, and the distinction between saving indexes locally and on the cloud. Finally, it compares LlamaIndex with the LangChain frameworks and concludes by discussing the practical application and effectiveness of LlamaIndex in LLM applications.

Chat with Your Code: LlamaIndex and Activeloop Deep Lake for GitHub Repositories

In this project lesson, students will learn how to use LlamaIndex in conjunction with Activeloop Deep Lake to index GitHub repositories, enabling interaction with codebases through natural language queries. They will understand both tools' core functionalities, the synergy between data structuring and optimized storage, and the setup process for integrating these technologies. The lesson will guide students through installing necessary packages, setting up a Python virtual environment, loading and parsing GitHub repository data, building an index, and querying this index using a combination of LlamaIndex and Deep Lake. Additionally, the lesson covers the customization and flexibility of LlamaIndex's API for tailored data retrieval and response synthesis, and it concludes by comparing LlamaIndex with LangChain for building chatbots with external data

LangChain: Basic Concepts Recap

In this lesson, we'll review the essential features of LangChain. We will examine the architecture comprising various components, such as data loading, processing, and segmentation, to provide optimal information to language models. Additionally, we will highlight the significance of indexing and retrieval.

The material in this lesson was covered in depth with code and project examples in our earlier course **LangChain & Vector Databases in Production.**

This overview is structured to clearly convey LangChain's features, providing foundational knowledge for advanced Generative AI and LLM-based projects.

Here's a Notebook with all the code we will go through in this lesson.

1. Preprocessing the Data

LangChain's approach to structuring documents is particularly favorable for developers and researchers. It provides tools that help structure documents for convenient use with LLMs. Document loaders simplify the process of loading data into documents, and text splitters break down lengthy pieces of text into smaller chunks for better processing. Finally, the indexing process involves creating a structured database of information that the language model can query to enhance its understanding and responses.

1-1. Document Loaders

Document Loaders are responsible for loading documents into structured data. They handle various types of documents, including PDFs, and convert them into a data type that can be processed by the other LangChain functions. It enables loading data from multiple sources into `Document` objects. LangChain provides over 100 document loaders and integrations with other major providers in the space, like AirByte and Unstructured, and from all sources, such as private S3 buckets and public websites.

Read from Files/Directories

Handling various input formats and transforming them into the Document format is easy. For instance, you can load the CSV data using the CSVLoader. Each row in the CSV file will be transformed into a separate `Document`.

```
from langchain.document_loaders import CSVLoader

# Load data from a CSV file using CSVLoader

loader = CSVLoader("./data/data.csv")

documents = loader.load()

# Access the content and metadata of each document

for document in documents:

                content = document.page_content

                metadata = document.metadata
```

The sample code.

Some of the popular loaders include the `TextLoader` for text files, the `DirectoryLoader` for loading all the files in a directory, the `UnstructuredMarkdownLoader` for markdown files, and the `PyPDFLoader` for loading PDF files.

Public Source Loaders

Loaders for popular public sources allow the data to be transformed into `Document` objects. For example, the `WikipediaLoader` retrieves the content of the specified Wikipedia page and loads it into a `Document`.

```
from langchain.document_loaders import WikipediaLoader

# Load content from Wikipedia using WikipediaLoader

loader = WikipediaLoader("Machine_learning")

document = loader.load()
```

Another popular loader is `UnstructuredURLLoader`, which allows reading from public web pages.

Proprietary Data loaders

These loaders are designed to handle proprietary sources that may require additional authentication or setup. For example, a loader could be created to load custom data from an internal database or an API with proprietary access.

Popular loaders of this category are `GoogleDriveLoader` for loading documents from Google Drive and `MongodbLoader` for loading documents from a MongoDB database.

1-2. Document transformers (chunking methods)

A crucial part of retrieval is fetching only the relevant details of documents. This involves several transformation steps to prepare the documents for retrieval. One of the primary steps here is splitting (or chunking) a large document into smaller segments. LangChain provides several transformation algorithms and optimized logic for specific document types.

Chunking is essential when the source documents are too large for the maximum input size imposed by models and involves partitioning the content into chunks that the LLM can process and then be used for indexing and queries. For instance, the GPT-4 model initially had context window of 8,000 tokens and the ada-002 embedding model is still limited to 8,000 tokens. Considering an average text document contains roughly 500 tokens per page, a 8000 token context window can handle a maximum of 16 pages. To address a query using a 100-page document, we must, therefore, divide the document into smaller chunks.

LangChain offers several key chunking transformation strategies.

- **Fixed-size chunks** that define a fixed size that's sufficient for semantically meaningful paragraphs (for example, 300 words) and allows for some overlap (for example, an additional 30 words). Overlapping ensures continuity and context preservation between adjacent chunks of data, improving the coherence and accuracy of the created chunks. For example, you may use the `CharacterTextSplitter` splitter to split every N character or token if configured with a tokenizer.
- **Variable-sized chunks** partition the data based on content characteristics, such as end-of-sentence punctuation marks, end-of-line markers, or using features in the NLP libraries. It ensures the preservation of coherent and contextually intact content in all chunks. An example is the `RecursiveCharacterTextSplitter` splitter.
- **Customized chunking** when dealing with large documents, you might use variable-sized chunks but also append the document title to chunks from the middle of the document to prevent context loss. This can be done, for example, with the `MarkdownHeaderTextSplitter`.

Chunking offers a mixed bag of strengths and weaknesses in processing extensive documents with LLMs.

A key advantage is its ability to manage documents that exceed the context window of an LLM. This capability enables the model to handle and analyze significantly larger texts than it could in a single pass, expanding its applicability and utility in processing lengthy documents.

However, this approach comes with a notable drawback. In dividing a document into chunks, there's a risk of losing vital context related to the overall document. While individually coherent, each chunk might only partially capture the nuances and interconnected elements present in the full text. This can lead to a fragmented or incomplete understanding of the document, as important details and subtleties might be overlooked or misinterpreted when the text is not viewed cohesively.

1-3. Indexing

Indexing is a process that involves storing and organizing data from various sources into a vector store, which is essential for efficient storing and retrieving. The process typically consists of storing the chunk along with an embedding representation of it, which captures the meaning of the text and makes it easy to retrieve chunks by semantic similarity. Embeddings are usually generated by embedding models, such as the `OpenAIEmbeddings` models.

2. Models

2-1. LLMs

LangChain provides an `LLM` class for interfacing with various language model providers, such as OpenAI, Cohere, and Hugging Face. Here's an example of some integrations. The rest of the list can be found in the conclusion section. Every LLM listed has a direct link to the document page with more detailed information.

LLM	DESCRIPTION
OpenAI	An AI research organization, very popular for the ChatGPT product
Hugging Face Hub	A platform that hosts thousands of pre-trained models and datasets
Cohere	A platform that provides natural language understanding APIs powered by large-scale neural networks
Llama-cpp	A library that makes it easy to load (small) language models locally on your PC
Azure OpenAI	A cloud service that provides access to OpenAI's LLMs

Before seeing an example of interacting with a "chat model", you must install the required packages to set up the environment.

```
pip install langchain==0.0.346 openai==1.3.7 tiktoken==0.5.2 cohere==4.37
```

The next step is setting the OpenAI API key in your environment to use the endpoints. Please remember to replace the `<YOUR_OPENAI_API_KEY>` placeholder with your key. Alternatively, you can load environment variables from a `.env` file using the `dotenv` library.

```
import os

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

Now, you can run the following code to interact with the OpenAI GPT-3.5 Turbo chat model.

```
from langchain.chat_models import ChatOpenAI

from langchain.schema import HumanMessage, SystemMessage

chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
```

```
messages = [
    SystemMessage(
        content="You are a helpful assistant."
    ),
    HumanMessage(
        content="What is the capital of France?"
    ),
]
chat(messages)
```

The sample code.

```
AIMessage(content='The capital of France is Paris.')
```

The output.

Chat models are a variation of language models that use a message-based input and output system. Chat models operate using LLMs but have a different interface that uses “messages” instead of raw text input/output.

A Chat model in LangChain has three types of messages:

- `SystemMessage` sets the behavior and objectives of the chat model. You would give specific instructions here like, “Act like a Marketing Manager.” or “Return only a JSON response and no explanation text.”
- The `HumanMessage` type is where you would input the user’s prompts to be sent to the model.
- Lastly, the `AIMessage` represents the responses from the models, which can be used to pass the history of previous messages to the model.

2-2. Embedding Models

Text embedding models are a standardized interface for various embedding model providers like OpenAI, Cohere, and HuggingFace. These models transform text into vector representations, enabling operations like semantic search through text similarity in vector space.
In LangChain, the `embed_documents` method is used to embed multiple texts, providing a list of vector representations.

```
from langchain.embeddings import OpenAIEmbeddings
# Initialize the model
embeddings_model = OpenAIEmbeddings()
# Embed a list of texts
embeddings = embeddings_model.embed_documents(
    ["Hi there!", "Oh, hello!", "What's your name?", "My friends call me World",
"Hello World!"])
```

```
print("Number of documents embedded:", len(embeddings))
print("Dimension of each embedding:", len(embeddings[0]))
```

The sample code.

```
Number of documents embedded: 5
Dimension of each embedding: 1536
```

The output.

The above code snippet illustrates how the embedding model can effectively convert phrases such as "Hi there!" into a 1536-dimensional vector. The remarkable aspect of these embeddings lies in their consistent output dimensionality, regardless of the input's length, while capturing the meaning of the sequences. These attributes enable us to measure sentence similarity using similarity metrics, such as cosine similarity.

3. The Role of Vector Stores

The increasing use of embeddings has created a demand for databases that can effectively store and search these embeddings. Vector stores tailored for managing vector data, such as Deep Lake by Activeloop, have become essential. They are a fundamental framework for storing and retrieving embeddings produced by LLMs.

Embeddings are high-dimensional vectors that capture the semantics of textual data. They are generated using LLMs and are crucial for tasks like text similarity, clustering, and retrieval. Traditional databases are not optimized for high-dimensional data. Vector stores, on the other hand, are built to handle such data, offering faster and more accurate retrieval.

The advantages of using vector stores in LangChain:

- **Speed**: Vector stores are optimized for quick data retrieval, which is essential for real-time applications.
- **Scalability**: As your application grows, so does your data. Vector stores can handle this growth efficiently.
- **Precision**: Vector stores ensure you get the most relevant results using specialized algorithms for nearest neighbor search.

3-1. Retrievers

Once the data is in the database, you need to retrieve it. Retrievers in LangChain are interfaces that return documents in response to a query. The most straightforward approach is to use basic similarity metrics such *as cosine similarity*, which compares the angle between vector embeddings of the question and potential answers, ensuring that the responses generated are semantically aligned with the query. This method effectively narrows down the most contextually relevant information from a vast dataset, improving the precision and relevance of the responses.
However, there are more advanced retrieval approaches to increase precision.

LangChain offers a variety of retrieval methods; some examples include:

- Parent Document Retriever: It creates multiple embeddings, allowing you to look up smaller chunks but return larger contexts. It becomes easier to discover related content when dealing with smaller chunks through cosine similarity, and then the parent document (e.g., the whole document containing that chunk) will be used to offer additional context to the LLM for generating the final answer.
- Self-Query Retriever: User questions often reference something that isn't just semantic but contains logic that can be represented as metadata filters. Self-query allows us to generate several filters based on the user's input prompt and apply them to the document's metadata. It improves performance by getting the

most out of users' prompts and using the document and its metadata to retrieve the most relevant content. For example, a query like “What were Microsoft's revenues in 2021?” would generate the filter “datetime = 2021,” which would then be used to filter the documents (assuming that we know their publish dates).

4. Chains

Chains consist of powerful, reusable components that can be linked together to perform complex tasks. Integrating prompt templates with LLMs using chains allows a powerful synergy. Taking the output of one LLM and using it as input for the next makes it feasible to connect multiple prompts sequentially. Additionally, it allows us to integrate LLMs with other components, such as long-term memory and output guarding. Chains can enhance the overall quality and depth of interactions.

The following two chain classes can be helpful in different situations: `LLMChain` and `SequentialChain`.

4-1. LLMChain

LLMChain is the simplest form of chain in LangChain that transforms user inputs using a `PromptTemplate`, a fundamental and widely used tool for interacting with LLMs. The following code receives the user’s input and a parser class to create a `PromptTemplate` object that can then interact with a model. The prompt object defines the parameters of our request to the model and determines the expected format of the output. Then, we can use the `LLMChain` class to tie the prompt and model to make predictions. Note that the `StrOutputParser` class will ensure that we receive a string containing the responses from the LLM.

```
from langchain.chains import LLMChain

from langchain.chat_models import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.schema import StrOutputParser

template = """List all the colors in a rainbow"""

prompt = PromptTemplate(template=template, input_variables=[], output_parser=
StrOutputParser())

chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

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

llm_chain.predict()
```

The sample code.

```
The colors in a rainbow are:

1. Red

2. Orange

3. Yellow

4. Green

5. Blue

6. Indigo

7. Violet
```

The output.

It is also possible to use the newly introduced LangChain Expression Language (LCEL) scheme to make the code more readable. The following code will generate the same output as the previous, but it is easier to interpret the data flow from the user prompt and parse the output.

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.schema import StrOutputParser

prompt = PromptTemplate.from_template(
    "List all the colors in a {item}."
)
runnable = prompt | ChatOpenAI(temperature=0) | StrOutputParser()
runnable.invoke({"item": "rainbow"})
```

The sample code.

Copy

```
The colors in a rainbow are:

1. Red
2. Orange
3. Yellow
4. Green
5. Blue
6. Indigo
7. Violet
```

The output.

4-2. Sequential

After invoking a language model using `LLMChain`, we can now experiment with making a series of subsequent calls to an LLM. This approach is especially beneficial for using the output of one call as the input for the next, streamlining the process, and enabling complex interactions across various applications. Here's an example of how to use the `SequentialChain`.

```
from langchain.prompts import PromptTemplate
post_prompt = PromptTemplate.from_template(
    """You are a business owner. Given the theme of a post, write a social media post
to share on my socials.
```

```
Theme: {theme}

Content: This is social media post based on the theme above:""")

review_prompt = PromptTemplate.from_template(

    """You are an expert social media manager. Given the presented social media post,
it is your job to write a review for the post.


Social Media Post:

{post}

Review from a Social Media Expert:""")


from langchain.chat_models import ChatOpenAI

from langchain.schema import StrOutputParser

llm = ChatOpenAI(temperature=0.0)

chain = (

    {"post": post_prompt | llm | StrOutputParser()}

    | review_prompt

    | llm

    | StrOutputParser())

chain.invoke({"theme": "Having a black friday sale with 50% off on everything."})
```

The sample code.

```
This social media post is highly effective in promoting the Black Friday sale. The use of
emojis and exclamation marks adds excitement and grabs the attention of the audience. The post
clearly states the offer - a 50% off on everything in-store and online, which is a great deal.
It also highlights the variety of products available, from trendy fashion pieces to must-have
accessories, appealing to a wide range of customers.


The post encourages urgency by mentioning that the sale is for a limited time only and advises
customers to arrive early to catch the best deals. The call to action is clear, directing
customers to visit the website or head to the store to explore the products. The post also
encourages customers to spread the word and tag their shopping buddies, which can help
increase the reach and engagement of the post.


Overall, this social media post effectively communicates the details of the Black Friday sale,
creates excitement, and encourages customers to take advantage of the unbeatable offer. It is
well-written, visually appealing, and likely to generate interest and engagement from the
audience.
```

The output.

The example above uses the mentioned LCEL language to create two distinct chains. The first chain will generate a social media post based on a theme, and the other one will act as a social media expert to review the generated post. It is evident that the output of the `post_prompt` will be marked as the `post` variable and passed to the `review_prompt` template. You can see a variation of this code in the lesson notebook, where the model returns both social media posts and the review. Also, refer to the documentation for the use of the `SequenceChain` class.

5. Memory

Memory is the backbone for maintaining context in ongoing dialogues, ensuring that the LLM can provide coherent and contextually relevant responses. Memory in LangChain is essential for context preservation and enhancing user experience. Traditional conversational models often struggle with maintaining context. LangChain's Memory module addresses this by storing both input and output messages in a structured manner.

The system can offer more personalized and relevant responses by remembering and referring to past interactions, significantly improving the user experience. This flexibility makes it suitable for conversational applications. Memory module methods save the current conversational context, including both user input and system output.

Conclusion

In this lesson, we explored LangChain's essential components. LangChain stands out with its versatile document loaders, which are capable of importing varied data types. We discussed its document transformation techniques, especially chunking, which breaks down long texts into manageable segments. We also learned about vector stores and their benefits, including faster data retrieval, scalability to handle data growth, and precision in retrieving the most relevant results.

Additionally, we covered LangChain's use of retrievers and chains. Retrievers provide various methods to return documents in response to queries. Chains facilitate the creation of complex workflows by linking multiple components and integrating LLMs with external data and long-term memories like chat history. This results in more effective responses and coherent conversations.

Overall, it is important to have a good grasp of LangChain's capabilities to pursue advanced projects in the field.

>> Notebook.