LangChain vs LlamaIndex

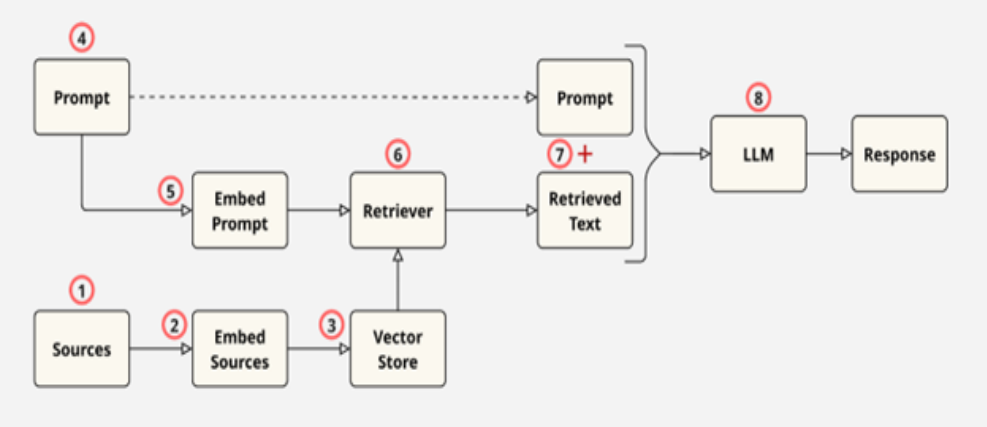
Estimated Reading Time: 10 minutes

In this reading, you'll explore the differences between LangChain and LlamaIndex, two frameworks for LLM-powered context augmentation. Both LangChain and LlamaIndex are commonly used to develop Retrieval-Augmented Generation (RAG) applications or chatbots with which you interact repeatedly. Additionally, they can be utilized to understand or summarize documents and extract data from various sources.

You'll specifically focus on the similarities and differences between LangChain and LlamaIndex in the context of RAG. This is because many other typical use cases involve components necessary for building RAG applications. For example, a chatbot often connects to a RAG system, enabling the LLM to extract additional contextual information from an external knowledge base. Therefore, by comparing LangChain and LlamaIndex in the context of RAG, you'll gain a comprehensive understanding of the differences between these frameworks for related workflows.

A Very Brief Review of RAG

To assist you in recalling the specific stages within RAG that you will reference when comparing LangChain and LlamaIndex, please refer to the following image, which illustrates a typical RAG system:



1. Load and chunk the source documents.
2. Embed the chunks or documents, converting text data into numerical vectors using an embedding model.
3. Store the vectors, typically in a vector database.
4. Accept the user's original prompt or query into the system.
5. Embed the user's prompt using the same model used for the source documents.
6. Use a retriever to compare the prompt embedding with the source document embeddings in the vector store and retrieve the most similar chunks or documents.
7. Augment the user's prompt with the information retrieved by the retriever.
8. Pass the augmented prompt to the LLM to produce a context-aware response.

Let's now compare how LangChain and LlamaIndex handle each of these steps.

1. Loading and Chunking Source Documents

Document Loading

In this section, you will explore the various methods of loading entire source documents in LangChain and LlamaIndex.

- LangChain
  - LangChain provides many different document loaders, including:
    - `TextLoader` to load plaintext files
    - `CSVLoader` to load tabular .csv files
    - `JSONLoader` to load JSON data
    - `WebBaseLoader` to load webpages using BeautifulSoup4 under the hood
    - `DoclingLoader`, which integrates with Docling to parse PDF, DOCX, PPTX, HTML and other formats
    - `UnstructuredLoader`, which can handle many types of files by using the open source `unstructured` library from unstructured.io
  - Entire directories can be loaded using LangChain's `DirectoryLoader`. Under the hood, `DirectoryLoader` uses `UnstructuredLoader` by default, though this can be changed to a different loader using a parameter.
  - Many other connectors and integrations are available to SQL databases, cloud object storage services such as AWS's S3, and specific application files such as Figma files.

- LlamaIndex
  - LlamaIndex provides a very powerful loader in its core library called `SimpleDirectoryReader`. `SimpleDirectoryReader` natively handles a large number of formats, including markdown, PDF, Word documents, PowerPoint decks, and more. Moreover, `SimpleDirectoryReader` can be used to load individual files or entire directories, including recursively traversing all subdirectories, as well as loading files with specific extensions defined in a list.

    - For additional connections that are not built into LlamaIndex's core library, LlamaIndex provides a registry of data connectors and various other integrations at LlamaHub. Some of the many data connectors available at LlamaHub include:
        - `DatabaseReader`, which can run queries against SQL databases
        - `JSONReader`, which can load JSON data, and
        - `RssReader`, which can load RSS feeds.

When comparing document loading capabilities between LangChain and LlamaIndex, LlamaIndex stands out with its powerful `SimpleDirectoryLoader`, which effortlessly handles various file types, offering a superior out-of-the-box experience. On the other hand, LangChain shines in flexibility: its `DirectoryLoader` can be configured to use any of LangChain's numerous loaders, highlighting a key design difference. LlamaIndex generally provides effective native solutions for most tasks, only resorting to external libraries when necessary. In contrast, LangChain relies heavily on integrations and a modular design, with much of its core functionality dependent on external packages and libraries.

Document chunking

LangChain and LlamaIndex offer a variety of chunking strategies:

- LangChain
    - For typical use cases, LangChain offers length-based text splitters. These come in two types:
        - A character-based text splitter, called `CharacterTextSplitter`, divides text using a specific character sequence, such as \n\n for line breaks. Additionally, the chunk sizes are limited by a settable maximum character length.
        - Token-based text splitters limit the maximum chunk length by a settable number of tokens. One example is `TokenTextSplitter`, which encodes text into tokens, splits the tokens into chunks based on length, and then decodes the chunks back into text. Additionally, `CharacterTextSplitter` can also be used for token-based text splitting. In this context, `CharacterTextSplitter` splits text based on a specific character sequence but limits the chunk size by the maximum number of tokens instead of character length.
    - An extension of some of the above concepts is provided by the `RecursiveCharacterTextSplitter`. `RecursiveCharacterTextSplitter` splits on characters from a specific list. For instance, if the list is `["\n\n", "."]`, the text is first split on two newline characters. Then, if any of the resulting chunks are too long, those chunks are further split on the next character in the list, in this case, a period. Hence, the recursive nature of the text splitting.
    - Document-structured text splitters split documents based on some of the inherent elements in the files. For instance, the `MarkdownHeaderTextSplitter` splits on markdown headings, such as `#`, `##` etc. LangChain provides document splitters for markdown files, code, HTML, and a recursive splitter for JSON.
    - There is also a semantic meaning-based splitter called `SemanticChunker` that splits text if the similarity between two sentences is below a certain threshold. This can be useful for finding natural breaks between concepts in text.

- LlamaIndex
    - First off, it's worth noting that LlamaIndex refers to a text chunk as a `node`.
    - For basic use cases, LlamaIndex provides a `SentenceSplitter`, which works similarly to LangChain's `RecursiveCharacterTextSplitter` except that it is token-based in the sense that the chunk size is defined by the number of tokens.
    - LlamaIndex provides several file-based "node parsers" that are similar to LangChain's document-structured text splitters. There are splitters for HTML, JSON, and markdown files. There is also a code splitter, though in contrast to LangChain, the code splitter is text-based, not file-based.
    - `SemanticSplitterNodeParser` provides similar functionality to LangChain's `SemanticChunker` and splits text if the similarity between two sentences falls below a threshold.
    - Finally, LlamaIndex provides a `LangChainNodeParser` which wraps around any LangChain text splitter. This allows you to use any splitter from LangChain in LlamaIndex!

Once again, for basic usage LlamaIndex typically provides a slightly better experience with its `SentenceSplitter` being set up by default with a more comprehensive splitting list than LangChain's `RecursiveCharacterTextSplitter`. However, both frameworks provide comprehensive coverage of splitting strategies, with LlamaIndex even providing a wrapper for LangChain's splitters if you prefer those while using the other aspects of LlamaIndex.

2. Embed the chunks or documents

Both LangChain and LlamaIndex offer a large number of embedding model integrations, including models from HuggingFace and OpenAI. Moreover, LlamaIndex provides a wrapper around LangChain embedding models, allowing you to use any embedding model that is compatible with LangChain within LlamaIndex.
The main difference between embedding designs in LangChain and LlamaIndex is that in LlamaIndex, vector embeddings are usually generated and stored in a vector store within a single command. In contrast, LangChain typically generates the embeddings first and then stores them in the vector store in a separate step.

3. Store Vectors in a Vector Store

This section explores the differences between LangChain and LlamaIndex with respect to storing the vector embeddings.

- LangChain
    - LangChain does not have a single, all-encompassing vector store class, and typically relies on integrations with external libraries to integrate a vector database. The only commonly used vector store defined within the LangChain core library is the `InMemoryVectorStore` which stores vectors in memory. Integrations for external libraries and full-fledged vector databases include:
        - `Chroma` for integration with Chroma DB
        - `FAISS` for integration with the Facebook AI Similarity Search (FAISS) library and vector database
        - `Milvus` for integration with the Milvus vector database
        - `PGVector` for integration with `pgvector`-extended PostgreSQL

- LlamaIndex
    - LlamaIndex provides a powerful `VectorStoreIndex` class that stores vectors in memory by default but can be integrated with full-fledged vector databases such as Chroma DB or Faiss. Thus, LlamaIndex's `VectorStoreIndex` class ensures ease of use because the vector store backend can be swapped out without changing any of the downstream code.
    - Moreover, as opposed to the typical workflow in LangChain, in LlamaIndex the vectors are embedded and stored in the vector store with just one command: `index = VectorStoreIndex(nodes)`, where `nodes` contains the document chunks created by one of LlamaIndex's chunking methods.

One of LlamaIndex's main advantages is that chunk metadata is automatically created and stored within the `VectorStoreIndex` object. In contrast, LangChain may require manual metadata setup. Additionally, due to LangChain's modular design, the methods for handling metadata can vary depending on the backend vector store. However, LangChain's vector store integrations are not wrapped by a single class, offering more flexibility and granular control by exposing unique features of each vector store type.

4. Accepting the user's original prompt

Accepting the user's original prompt is a step in the RAG process, but there are no specific implementations within LangChain or LlamaIndex with respect to how that prompt is received. Both LangChain and LlamaIndex operate under the assumption that a prompt will be passed to them from a different workflow or process.

5. Embed the user's original prompt

Both LangChain and LlamaIndex typically embed the user's original prompt by using a retriever created from the vector store object. This is done for two reasons:

- The embedding model used to embed the prompt should be the same as the model used to generate the chunk embeddings inside the vector store, so it makes sense to store the embedding model within the vector store object
- The prompt embedding is typically only used for retrieval tasks, where the information is retrieved from the vector store

Thus, embedding is typically combined with retrieval, and there are no specific implementation differences of note between LangChain and LlamaIndex.

6. Retrieve relevant chunks

For typical use cases where the top k chunks are retrieved based on similarity with the user prompt, LangChain and LlamaIndex offer similar functionality. However, both frameworks provide various advanced retrievers and retrieval patterns. For example, LangChain includes a parent document retriever that retrieves the parent document containing the relevant chunk, rather than just the chunk itself. Note that such advanced retrieval patterns in LangChain require two vector stores: one for the chunks and one for the parent documents.

Detailed discussions of retrieval patterns are beyond the scope of this document. If your project requires a specific retrieval pattern, it is recommended to explore the retrieval options provided by LangChain and LlamaIndex. Doing so can help you choose the framework that best fits your needs and increases the likelihood that your application will work as intended.

7. Augment the user's original prompt

Both LangChain and LlamaIndex use prompt templates for prompt augmentation. These templates contain placeholders that are filled with the user's original prompt and the retrieved information when that information becomes available. However, the methods of prompt augmentation differ between LangChain and LlamaIndex.

- LangChain
    - LangChain does not combine prompt augmentation with any of the preceeding or succeeding steps. This ensures that prompt template customization is easy.
- LlamaIndex
    - LlamaIndex combines the prompt augmentation step with the LLM response generation step when a "response synthesizer" is used. Alternatively, if a "query engine" is used, the process is further combined with the query embedding and retrieval steps. Thus, although the default prompt templates provided by LlamaIndex work well for most cases, customizing these templates is slightly more challenging because the prompt augmentation step is combined with the LLM response generation step.

8. Pass the augmented prompt to the LLM

Passing the augmented prompt to the LLM works differently for LangChain and LlamaIndex.

- LangChain
    - For LangChain, the augmented prompt is passed to the LLM manually. For instance, given `messages` generated by a prompt template and an `llm` object, LangChain generates a response from the LLM using `response = llm.invoke(messages)`.
- LlamaIndex
    - For LlamaIndex, the process is combined with the prompt augmentation step. There are several options here, including:
        - Using a "prompt synthesizer," which takes the user's original prompt and the retrieved nodes as inputs, performs prompt augmentation internally, and passes the augmented prompt to the LLM to generate a response as output.
        - Using a "query engine" which is an object that is derived from the vector store index and takes the user's original prompt as an input. The query engine then performs the query embedding, retrieval, prompt augmentation, and LLM response generation steps internally.

Conclusion

Despite the various strengths and weaknesses of LangChain and LlamaIndex, both are capable of handling most typical RAG workflows. LangChain excels in its numerous integrations, modular design, and the ease of accessing and customizing its components. However, it often requires manual setup, such as configuring metadata in the vector store. On the other hand, LlamaIndex is known for its simplicity and ease of development. Although customizations can be more challenging, LlamaIndex provides sensible defaults and internal solutions that work well for typical tasks. Notably, LlamaIndex's `SimpleDirectoryReader` document loader can handle many different file types out of the box, and the `VectorStoreIndex` class wraps external vector databases in a native class, making downstream code agnostic to the vector store backend.

Author

Wojciech "Victor" Fulmyk