Langchain

Open source developer framework for building LLM applications

Python and Typescript packages

Focused on composition and modularity

Key value adds:

Modular components (& implementation of those components)

Use cases – Common ways to combine those components together

Modular components include:-

Prompts:-

Prompt templates

Output parsers:- 5+ Implementations

Retry/fixing logic

Example Selectors :- 5+ Implemetations

Models:-

LLMs : 20+ Implementations

Chat Models

Text Embedding Models: 10+ Integrations

Indexes :-

Document loaders:- 50+ Implementations

Text Splitters : 10+ Implementations

Vector Stores: 10+ Integrations

Retrievers: 10+ Implementation\Integrations

Chains

Can be used as building blocks for other chains

More application specific chain: 20+ different types

Agents

Agent types: 5+ types

Algorithm for getting LLM to use tools

Agent toolkits: 10K Implementation

Agents armed with specific tools for specific applications

Memory types

ConversationBufferMemory  
This memory allows for storing of messages and then extracts the messages in a variable

ConversationBufferWindowMemory:

This memory keeps a list of the interactions of the conversations over time. It only uses the last K interactions

ConversationTokenBufferMemory

This memory keeps a buffer of recent interactions in memory, and uses token length rather than number of interactions to determine when to flush interactions.

ConversationSummaryMemory

This memory creates a summary of conversation over time.

Additional Memory types

Vector data memory

Store text (from conversation or elsewhere) in a vector DB and retrieve the most relevant blocks of text.

Entity memories

Using on LLM, it remembers details about specific entities

You can also use multiple memories at one time.

Eg. Conversation memory+Entity memory to recall individuals

You can also store the conversation in a conversational DB (such as key- value store or SQL)

Sequential Chain:-

Seqential chain is another type of chains.

The idea is to combine multiple chains where the output of the one chain is the input of the next chain

There is two type of sequential chains:

1. SimpleSequentialChain: Single input/output
2. <https://images.app.goo.gl/KxKrvvwktsc1BBuD6>
3. SequentialChain: multiple inputs/outputs
4. <https://images.app.goo.gl/gcjmBD4gYQg33P359>

Router Chain

LLM can only inspect a few thousand words at a time

Embeddings

Embeddings vector captures content/meaning

Text with similar content will have similar vectors

Eg.

1 My dog Rover likes to chase squirrels.

2 Fluffy, my cat refuses to eat from can

3. The chevy Bolt accelerates to 60 mph in 6.7 seconds

The 1, 2 are about pets and 3 is about car

Vector Database

To store vectors that we created in previous step.

The way that we create Vector DB is to populate it with chunks of text coming from incoming document

For big Document:- break it into small chunks of text.

Thus we create Indexes of vector store.

Whenever query comes in we compare the vector store of that query with that of Vector DB and extract the most relevant one

Stuff Method

Docs

Final Asnser

LLMs

Prompts

Stuffing is the simplest method. You simply stuff all data into the prompt as context to pass to the language model.

Pros: It makes a single call to the LLM. The LLM has access to all data.

Cons. LLMs have a context length and for large documents or many documents this will not work as it will result in a prompt larger than the context length.

The first is "Map\_reduce".

Chunks LLM LLM Final Answer

This basically takes all the chunks, passes them along with the question to a language model, gets back a response, and then uses another language model call to summarize all of the individual responses into a final answer. This is really powerful because it can operate over any number of documents. And it's also really powerful because you can do the individual questions in parallel. But it does take a lot more calls. And it does treat all the documents as independent, which may not always be the most desired thing.

"Refine",

Chunks LLM LLM Final Answer

which is another method, is again used to loop over many documents. But it actually does it iteratively. It builds upon the answer from the previous document. So this is really good for combining information and building up an answer over time. It will generally lead to longer answers. And it's also not as fast because now the calls aren't independent. They depend on the result of previous calls. This means that it often takes a good while longer and takes just as many calls as "Map\_reduce", basically.

"Map\_rerank"

400

Select highest score

910

330

Chunks LLM LLM Final Answer

is a pretty interesting and a bit more experimental one where you do a single call to the language model for each document. And you also ask it to return a score.  And then you select the highest score. This relies on the language model to know what the score should be. So you often have to tell it, "Hey, it should be a high score if it's relevant to the document and really refine the instructions there". Similar to "Map\_reduce", all the calls are independent. So you can batch them and it's relatively fast. But again, you're making a bunch of language model calls. So it will be a bit more expensive. The most common of these methods is the "stuff method", which we used in the notebook to combine it all into one document. The second most common is the "Map\_reduce" method, which takes these chunks and sends them to the language model. These methods here, stuff, map\_reduce, refine, and rerank can also be used for lots of other chains besides just question answering. For example, a really common use case of the "Map\_reduce" chain is for summarization, where you have a really long document and you want to recursively summarize pieces of information in it. **That's it for question answering over documents.**As you may have noticed, there's a lot going on in the different chains that we have here. And so in the next section, we'll cover ways to better understand what exactly is going on inside all of these chains.

Chat with your data

RAG (Retrieval Augmented Generation)

A diagram of a document loading process

Description automatically generated

Loaders

* Loaders deal with the specifics of accessing and converting data
  + Accessing
    - Websites
    - Data Bases
    - Youtube
    - arXiv
    - ..
* Data Types
  + PDF
  + HTML
  + JSON
  + Word , PowerPoint..

Returns list of ‘Document’ objects:

[

Document{page\_content=’MachineLearning-Lecture01 \nInstructor (Andrew Ng):Okay…..’,

Metadata = {‘source’:’docs/cs229..lectures/ML-Lecture01.pdf’,’page’: 0})

….

Document{page\_content=’[End of Audio] \nDuration: 69 minutes ’,

Metadata = {‘source’:’docs/cs229..lectures/ML-Lecture01.pdf’,’page’: 21})

]

A diagram of a document splitting

Description automatically generated

Splitting documents into smaller chunks

Retaining meaningful relationships!

Eg.

…..

On this model. The Toyota Camry has a head-snapping 80 HP and eight-speed automatic transmission that will

….

Chunk 1: on this model. The Toyota Cmry has a head-snapping

Chunk2: 80 HP and an eight-speed automatic transmission that will

Question : what are the specifications on the Camry

Won’t be able to answer this

Example Splitter

Langchain.text\_splitter.CharacterTextSpliter(

Separator:str = “\n\n”

Chunk\_size = 4000,

Chunk\_overlap=200,

Length\_function=<builtin function len>,

)

Methods:

Create\_documnets() – Create documents from a list of texts

Split\_documents() – Split documents from a list of documents

Types of splitters

Langchain.text\_splitter

CharacterTextSplitter() – Implementation of splitting text that looks at characters

MarkdownHeaderTextSplitter() – Implementation of splitting markdown files based on specified headers

TokenTextSplitter() - Implementation of splitting text that looks at tokens.

SenetenceTransformersTokenTextSpitter() - Implementation of splitting text that looks at tokens

RecursiveCharacterTextSplitter()- Implementation of splitting text that looks at characters. Recursively tries to split by different characters to find one that works

Language() – for CPP, Python, Ruby, Markdown etc

NLTKTextSplitter()- Implementation of splitting text that looks at sentences using NLTK

SpacyTextSplitter() - Implementation of splitting text that looks at sentences using Spacy

Retrieval

A diagram of a flowchart

Description automatically generated

* Accessing/indexing the data in the vector store
  + Basic semantic similarity
  + Maximum marginal relevance
  + Including Metadata
* LLM aided retrieval

Maximum marginal relevance (MMR)

You may not always want to choose the most similar responses

Sometime the relevant data is also important

**MMR algorithm**

* Query the Vector Store
* Choose the fetch\_k most similar responses
* Within those responses choose the k most diverse responses

**LLM Aided Retrieval**

* There are several situations where the Query applied to the DB is more than just the Question asked.
* One is SelfQuery, where we use an LLM to convert the user question into a query

Eg.

Question:- What are some movies about aliens made in 1980?

Query Parser

Query Parser

Aliens

Eq(“year”,1980)

Filter Search term

Using LLM we can create filter and search term

**Compression**

* Increase the number of results you can put in the context by shrinking the response to only the relevant information

**Question Answering**

* Multiple relevant documents have been retrieved from the vector store
* Potentially compress the relevant splis to fit into LLM context
* Send the information along with our question to an LLM to select and format an answer

**RetrievalQA chain**

* Question is applied to the Vector Store as a query
* Vector store provides k relevant documents
* Docs and original question are sent to an LLM

More details on :-

<https://analyzingalpha.com/langchain-python-tutorial#Chains>