Retrieval Augmented Generation

A Simple Introduction

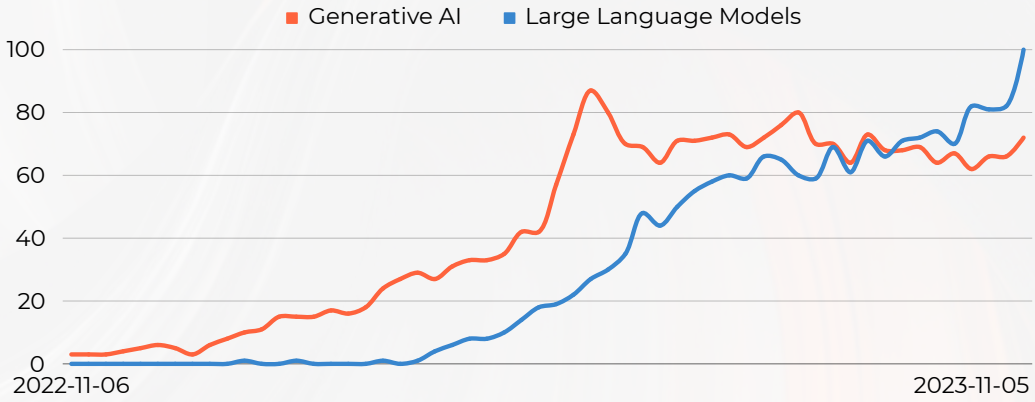
Abhinav Kimothi

Table of Contents

What is RAG?

Retrieval Augmented Generation

30th November, 2022 will be remembered as the watershed moment in artificial intelligence. OpenAI released ChatGPT and the world was mesmerised. Interest in previously obscure terms like Generative AI and Large Language Models (LLMs), was unstoppable over the following 12 months.



Google Trends - Interest Over Time (Nov'22 to Nov'23)

The Curse Of The LLMs

As usage exploded, so did the expectations. Many users started using ChatGPT as a source of information, like an alternative to Google. As a result, they also started encountering prominent weaknesses of the system. Concerns around copyright, privacy, security, ability to do mathematical calculations etc. aside, people realised that there are two major limitations of Large Language Models.

A Knowledge Cut-off date

Training an LLM is an expensive and time-consuming process. LLMs are trained on massive amount of data. The data that LLMs are trained on is therefore historical (or dated).
e.g. The latest GPT4 model by OpenAI has knowledge only till April 2023 and any event that happened post that date, the information is not available to the model.

Hallucinations

Often, it was observed that LLMs provided responses that were factually incorrect. Despite being factually incorrect, the LLM responses "sounded" extremely confident and legitimate. This characteristic of "lying with confidence" proved to be one of the biggest criticisms of ChatGPT and LLM techniques, in general.

> Users look at LLMs for knowledge and wisdom, yet LLMs are sophisticated predictors of what word comes next.

The Hunger For More

While the weaknesses of LLMs were being discussed, a parallel discourse around providing context to the models started. In essence, it meant creating a ChatGPT on proprietary data.

The Challenge

- Make LLMs respond with up-to-date information
- Make LLMs not respond with factually inaccurate information
- Make LLMs aware of proprietary information

  Providing LLMs with information not in their memory

Providing Context

While model re-training/fine-tuning/reinforcement learning are options that solve the aforementioned challenges, these approaches are time-consuming and costly. In majority of the use-case, these costs are prohibitive.

In May 2020, researchers in their paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" explored models which combine pre-trained parametric and non-parametric memory for language generation.

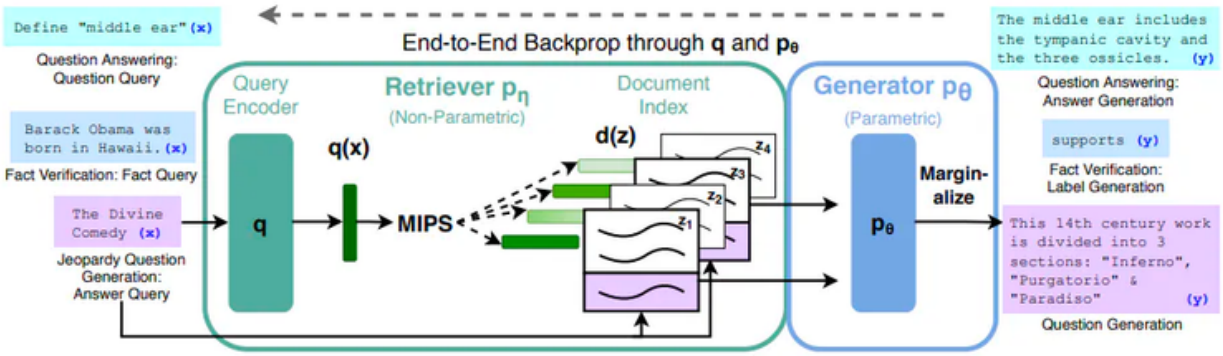
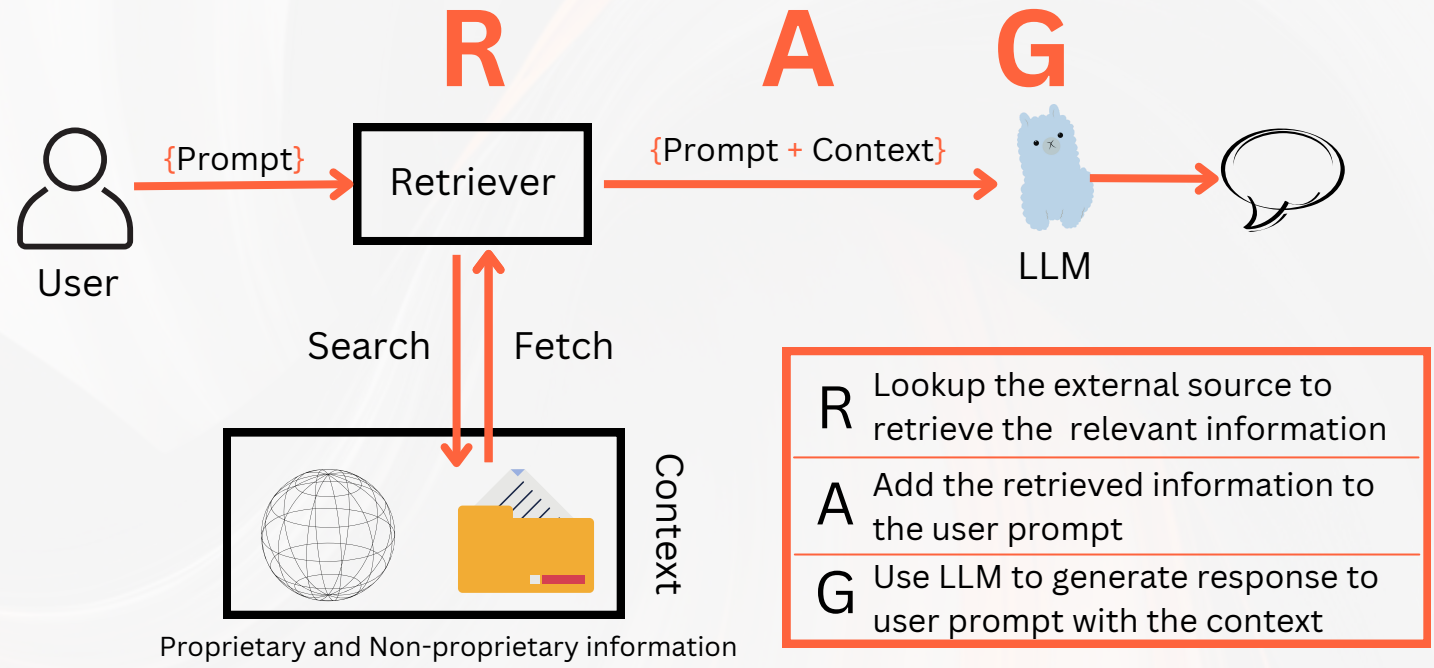


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder + Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.

So, What is RAG?

In 2023, RAG has become one of the most used technique in the domain of Large Language Models.



What is RAG?

User enters a prompt/query

↓

Retriever searches and fetches information relevant to the prompt (e.g. from the internet or internet data warehouse)

↓

Retrieved relevant information is augmented to the prompt as context

↓

LLM is asked to generate response to the prompt in the context (augmented information)

↓

User receives the response

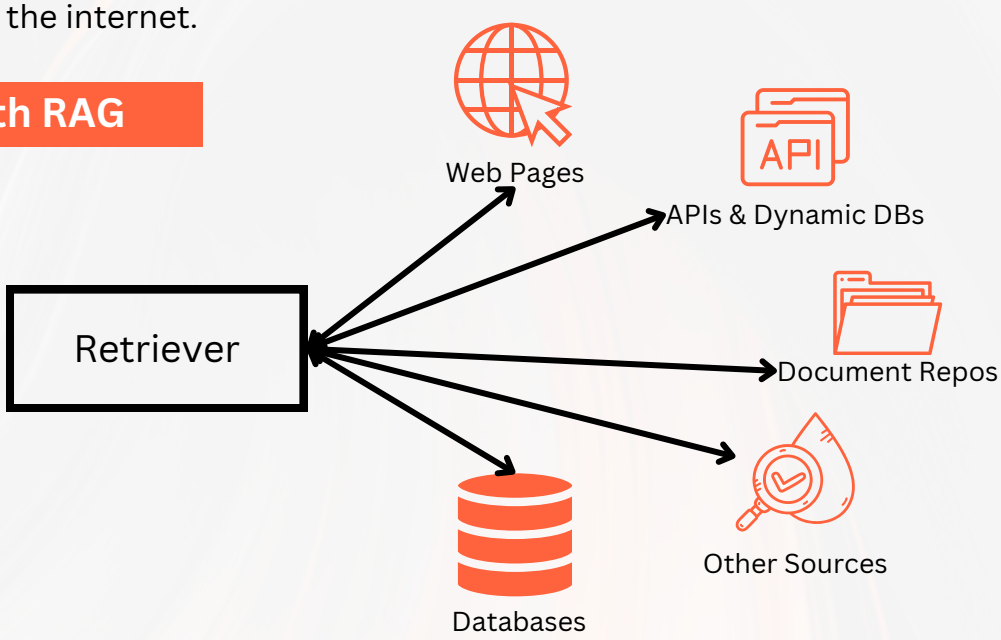
A Naive RAG workflow

How does RAG help?

Unlimited Knowledge

The Retriever of an RAG system can have access to external sources of information. Therefore, the LLM is not limited to its internal knowledge. The external sources can be proprietary documents and data or even the internet.

Without RAG | **With RAG**



An LLM has knowledge only of the data it has been trained on
Also called **Parametric Memory** (information stored in the model parameters)

Retriever searches and fetches information that the LLM has not necessarily been trained on. This adds to the LLM memory and is passed as context in the prompts. Also called **Non-Parametric Memory** (information available outside the model parameters)

- Expandable to all sources
- Easier to update/maintain
- Much cheaper than retraining/fine-tuning

The effort lies in creation of the knowledge base

Confidence in Responses

With the context (extra information that is retrieved) made available to the LLM, the confidence in LLM responses is increased.

Context Awareness

Added information assists LLMs in generating responses that are accurate and contextually appropriate

Source Citation

Access to sources of information improves the transparency of the LLM responses

Reduced Hallucinations

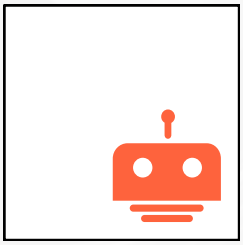
RAG enabled LLM systems are observed to be less prone to hallucinations than the ones without RAG

RAG Use Cases

The development of RAG technique is rooted in use cases that were limited by the inherent weaknesses of the LLMs. As of today some commercial applications of RAG are in -

Document Question Answering Systems

By providing access to proprietary enterprise document to an LLM, the responses are limited to what is provided within them. A retriever can search for the most relevant documents and provide the information to the LLM. Check out this blog for an example



Conversational agents

LLMs can be customised to product/service manuals, domain knowledge, guidelines, etc. using RAG. The agent can also route users to more specialised agents depending on their query. SearchUnify has an LLM+RAG powered conversational agent for their users.

Real-time Event Commentary

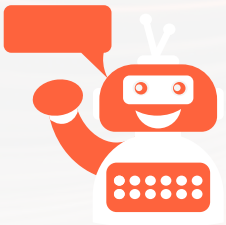
Imagine an event like a sports or a new event. A retriever can connect to real-time updates/data via APIs and pass this information to the LLM to create a virtual commentator. These can further be augmented with Text To Speech models.IBM leveraged the technology for commentary during the 2023 US Open

Content Generation

The widest use of LLMs has probably been in content generation. Using RAG, the generation can be personalised to readers, incorporate real-time trends and be contextually appropriate. Yarnit is an AI based content marketing platform that uses RAG for multiple tasks.

Personalised Recommendation

Recommendation engines have been a game changes in the digital economy. LLMs are capable of powering the next evolution in content recommendations. Check out Aman's blog on the utility of LLMs in recommendation systems.

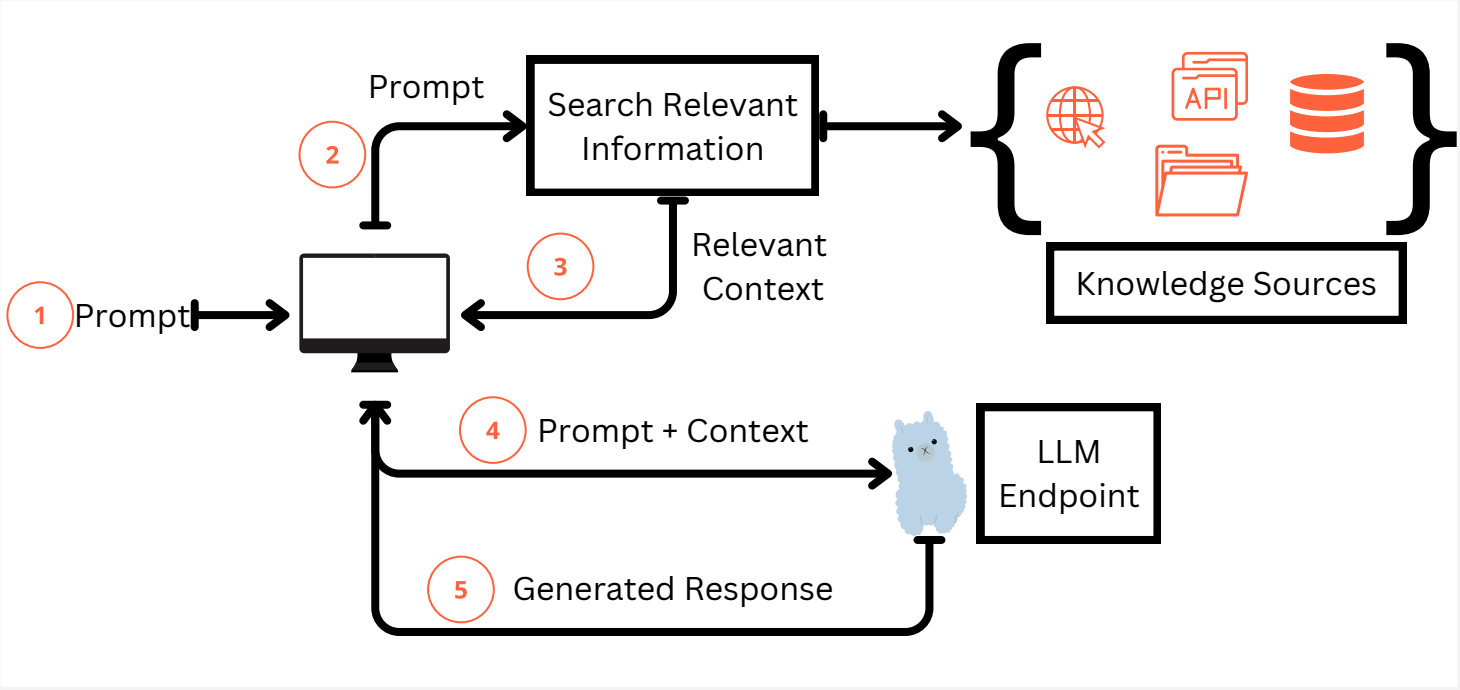


Virtual Assistants

Virtual personal assistants like Siri, Alexa and others are in plans to use LLMs to enhance the experience. Coupled with more context on user behaviour, these assistants can become highly personalised.

RAG Architecture

Let's revisit the five high level steps of an RAG enabled system



RAG System

1. User writes a prompt or a query that is passed to an orchestrator
2. Orchestrator sends a search query to the retriever
3. Retriever fetches the relevant information from the knowledge sources and sends back
4. Orchestrator augments the prompt with the context and sends to the LLM
5. LLM responds with the generated text which is displayed to the user via the orchestrator

Two pipelines become important in setting up the RAG system. The first one being setting up the knowledge sources for efficient search and retrieval and the second one being the five steps of the generation.

Indexing Pipeline

Data for the knowledge is ingested from the source and indexed. This involves steps like splitting, creation of embeddings and storage of data.

RAG Pipeline

This involves the actual RAG process which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model

Indexing Pipeline

The indexing pipeline sets up the knowledge source for the RAG system. It is generally considered an offline process. However, information can also be fetched in real time. It involves four primary steps.

Loading

This step involves extracting information from different knowledge sources a loading them into documents.

Splitting

This step involves splitting documents into smaller manageable chunks. Smaller chunks are easier to search and to use in LLM context windows.

Embedding

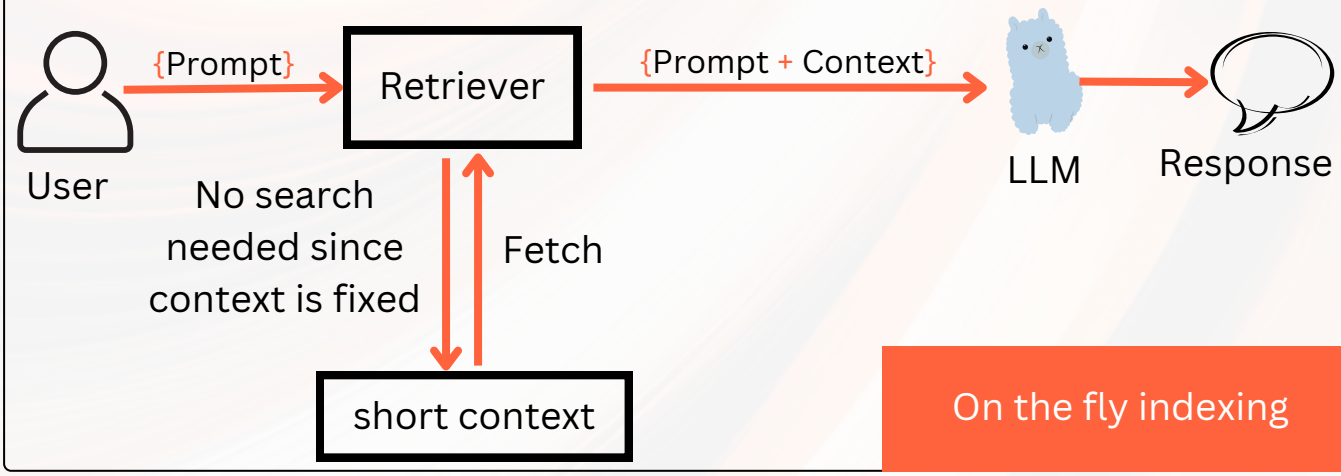
This step involves converting text documents into numerical vectors. ML models are mathematical models and therefore require numerical data.

Storing

This step involves storing the embeddings vectors. Vectors are typically stored in Vector Databases which are best suited for searching.

Offline Indexing pipelines are typically used when a knowledge base with large amount of data is being built for repeated usage e.g. a number of enterprise documents, manuals etc.

In cases where only a fixed small amount of one time data is required e.g. a 300 word blog, there is no need for storing the data. The blog text can either be directly passed in the LLM context window or a temporary vector index can be created.



Loading Data

As we've been discussing, the utility of RAG is to access data for all sorts of sources. These sources can be -

- Websites & HTML pages
- Documents like word, pdf etc.
- Code in python, java etc.
- Data in json, csv etc.
- APIs
- File Directories
- Databases
- And many more

The first step is to extract the information present in these source locations.

This is a good time to introduce two popular frameworks that are being used to develop LLM powered applications.

LangChain

Use cases: Good for applications that need enhanced AI capabilities, like language understanding tasks and more sophisticated text generation

Features: Stands out for its versatility and adaptability in building robust applications with LLMs

Agents: Makes creating agents using large language models simple through their agents API

LlamaIndex

Use cases: Good for tasks that require text search and retrieval, like information retrieval or content discovery

Features: Excels in data indexing and language model enhancement

Connectors: Provides connectors to access data from databases, external APIs, or other datasets

Both frameworks are rapidly evolving and adding new capabilities every week. It's not an either/or situation and you can use both together (or neither).

Example : Loading a YouTube Video Transcript using LangChain Loaders

Let's begin by sourcing the transcript from this video - "DALL·E 2 Explained" by OpenAI (https://www.youtube.com/watch?v=qTgPSKKjfVg)

Below is the code using YoutubeLoader from langchain.document_loaders

```
from langchain.document_loaders import YoutubeLoader


loader = YoutubeLoader.from_youtube_url(
    "https://www.youtube.com/watch?v=qTgPSKKjfVg",
    add_video_info=True
)


loader.load()
```

LangChain Document Loader : YoutubeLoader

Loader object

[Document(page_content="Have you ever seen a polar bear playing bass? Or a robot painted like a Picasso? Didn't think so. DALL-E 2 is ....
....
....
.....umans\nand clever systems can work together to make new things – amplifying our creative potential.", metadata={'source': 'qTgPSKKjfVg', 'title': 'DALL·E 2 Explained', 'description': 'Unknown', 'view_count': 853564, 'thumbnail_url': 'https://i.ytimg.com/vi/qTgPSKKjfVg/hq720.jpg', 'publish_date': '2022-04-06 00:00:00', 'length': 167, 'author': 'OpenAI'})]

The Document object contains the page_content which is the transcript extracted from the youtube video as well as the metadata description

Example : Loading a Webpage Text using LlamaIndex Reader

This is a blog published on Medium -
What is a fine-tuned LLM?
(https://medium.com/mlearning-ai/what-is-a-fine-tuned-llm-67bf0b5df081)

Below is the code using SimpleWebPageReader from llama_hub

```
from llama_hub.web.simple_web.base import SimpleWebPageReader


loader = SimpleWebPageReader()


docs = loader.load_data(
    urls=["https://medium.com/mlearning-ai/\
    what-is-a-fine-tuned-llm-67bf0b5df081"]
    )
```

LlamaIndex LlamaHub Web Page Reader

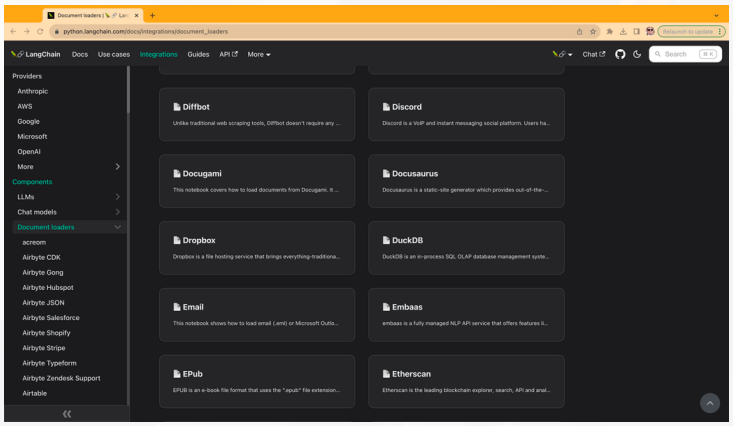
Loader object

[Document(id_='17761da4-6a3a-4ce5-8590-c65ee446788f', embedding=None, metadata={}, excluded_embed_metadata_keys=[], excluded_llm_metadata_keys=[], relationships={}, hash='6471b3ffe4d3abb1aba2ca99d1d0448e2c3cbd157ddca256fab9fa363e09ed85', text='<!doctype html><html lang="en"><head><title data-rh="true">What is a fine-tuned LLM?. Fine-tuning large language models... | by Abhinav Kimothi |
...
</body></html>', start_char_idx=None, end_char_idx=None, text_template='{metadata_str}\n\n{content}', metadata_template='{key}: {value}', metadata_seperator='\n')]

The LlamaIndex Document object contains more attributes than a LangChain Document. Apart from text and metadata, it also has id, templates and other customizations available

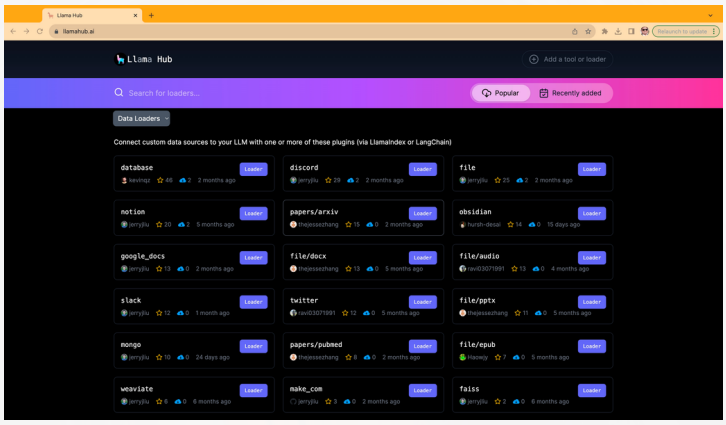
Both LangChain and LlamaIndex offer loader integrations with more than a hundred data sources and the list keeps on growing

LangChain Document Loaders



LangChain provides integrations with a variety of sources

LlamaHub Data Loaders



LlamaIndex provides data loaders via LlamaHub

These Document Loaders are particularly helpful in quickly making connections and accessing information. For specific sources, custom loaders can also be developed.

It is worthwhile exploring documentation for both

LlamaIndex: **https://docs.llamaindex.ai/en/stable/**

LangChain: **https://python.langchain.com/docs/get_started/introduction**

- *Loading documents from a list of sources may turn out to be a complicated process. Make sure to plan for all the sources and loaders in advance.*

- *More often than naught, transformations/clean-ups to the loaded data will be required like removing duplicate content, html parsing, etc. LangChain also provides a variety of document transformers*

Document Splitting

Once the data is loaded, the next step in the indexing pipeline is splitting the documents into manageable chunks. The question arises around the need of this step. Why is splitting of documents necessary. There are two reasons for that -

Ease of Search

Large chunks of data are harder to search over. Splitting data into smaller chunks therefore helps in better indexation.

Context Window Size

LLMs allow only a finite number of tokens in prompts and completions. The context therefore cannot be larger than what the context window permits.

Chunking Strategies

While splitting documents into chunks might sound a simple concept, there are certain best practices that researchers have discovered. There are a few considerations that may influence the overall chunking strategy.

1 Nature of Content

Consider whether you are working with lengthy documents, such as articles or books, or shorter content like tweets or instant messages. The chosen model for your goal and, consequently, the appropriate chunking strategy depend on your response.

2 Embedding Model being Used

We will discuss embeddings in detail in the next section but the choice of embedding model also dictates the chunking strategy. Some models perform better with chunks of specific length

3 Expected Length and Complexity of User Queries

Determine whether the content will be short and specific or long and complex. This factor will influence the approach to chunking the content, ensuring a closer correlation between the embedded query and the embedded chunks

4 Application Specific Requirements

The application use case, such as semantic search, question answering, summarization, or other purposes will also determine how text should be chunked. If the results need to be input into another language model with a token limit, it is crucial to factor this into your decision-making process.

Chunking Methods

Depending on the aforementioned considerations, a number of text splitters are available. At a broad level, text splitters operate in the following manner:

- Divide the text into compact, semantically meaningful units, often sentences.
- Merge these smaller units into larger chunks until a specific size is achieved, measured by a length function.
- Upon reaching the predetermined size, treat that chunk as an independent segment of text. Thereafter, start creating a new text chunk with some degree of overlap to maintain contextual continuity between chunks.

Two areas to focus on, therefore are -

How the text is split?

How the chunk size is measured?

A very common approach is where we pre-determine the size of the text chunks.

Additionally, we can specify the overlap between chunks (Remember, overlap is preferred to maintain contextual continuity between chunks).

This approach is simple and cheap and is, therefore, widely used. Let's look at some examples -

Split by Character

In this approach, the text is split based on a character and the chunk size is measured by the number of characters.

Example text : **alice_in_wonderland.txt** (the book in .txt format)

using **LangChain's CharacterTextSplitter**

```
with open('../Data/alice_in_wonderland.txt') as f:
    AliceInWonderland = f.read()

from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 2000,
    chunk_overlap  = 100,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([AliceInWonderland])
print(texts[0])
print(texts[1])
```

texts[0]

"TITLE: Alice's Adventures in Wonderland\nAUTHOR: Lewis Carroll\n\n\n CHAPTER I \n(Down the Rabbit-Hole)\n\n Alice was beginning to get very tired of sitting by her sister\non the bank, and of having nothing to do: once or twice she had\npeeped into the book her sister was reading, but it had no\npictures or conversations in it, `and what is the use of a book,'\nthought Alice `without pictures or conversation?'\n\n So she was considering in her own mind (as well as she could,\nfor the hot day made her feel very sleepy and stupid), whether\nthe pleasure of making a daisy-chain would be worth the trouble\nof getting up and picking the daisies, when suddenly a White\nRabbit with pink eyes ran close by her.\n\n There was nothing so VERY remarkable in that; nor did Alice\nthink it so VERY much out of the way to hear the Rabbit say to\nitself, `Oh dear! Oh dear! I shall be late!' (when she thought\nit over afterwards, it occurred to her that she ought to have\nwondered at this, but at the time it all seemed quite natural);\nbut when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOAT-\nPOCKET, and looked at it, and then hurried on, Alice started to\nher feet, for it flashed across her mind that she had never\nbefore seen a rabbit with either a waistcoat-pocket, or a watch to\ntake out of it, and burning with curiosity, she ran across the\nfield after it, and fortunately was just in time to see it pop\ndown a large rabbit-hole under the hedge.\n\n In another moment down went Alice after it, never once\nconsidering how in the world she was to get out again.\n\n The rabbit-hole went straight on like a tunnel for some way,\nand then dipped suddenly down, so suddenly that Alice had not a\nmoment to think about stopping herself before she found herself\nfalling down a very deep well."

Chunk 1

Overlap

texts[1]

"In another moment down went Alice after it, never once\nconsidering how in the world she was to get out again.\n\n The rabbit-hole went straight on like a tunnel for some way,\nand then dipped suddenly down, so suddenly that Alice had not a\nmoment to think about stopping herself before she found herself\nfalling down a very deep well.\n\n Either the well was very deep, or she fell very slowly, for she\nhad plenty of time as she went down to look about her and to\nwonder what was going to happen next. First, she tried to look\ndown and make out what she was coming to, but it was too dark to\nsee anything; then she looked at the sides of the well, and\nnoticed that they were filled with cupboards and book-shelves;\nhere and there she saw maps and pictures hung upon pegs. She\ntook down a jar from one of the shelves as she passed; it was\nlabelled `ORANGE MARMALADE', but to her great disappointment it\nwas empty: she did not like to drop the jar for fear of killing\nsomebody, so managed to put it into one of the cupboards as she\nfell past it."

Chunk 2

Let's find out how many chunks were created

```
print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 93

Length of the First Chunk is => 1777 characters

Length of the Last Chunk is => 816 characters

Recursive Split by Character

A subtle variation to splitting by character is Recursive Split. The only difference is that instead of a single character used for splitting, this technique uses a list of characters and tries to split hierarchically till the chunk sizes are small enough. This technique is generally recommended for generic text.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **LangChain's RecursiveCharacterTextSplitter**

This is a generic text that is not formatted. Let's compare the two strategies.

with **CharacterTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 2000,
    chunk_overlap  = 400,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([IntroToLLM])

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 1
Length of the First Chunk is => 64383 characters
Length of the Last Chunk is => 64383 characters

Text splitter fails to convert the text into chunks since there are no '\n\n' character present in the raw transcript

with **RecursiveCharacterTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 2000,
    chunk_overlap  = 400,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([IntroToLLM])

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 40
Length of the First Chunk is => 1998 characters
Length of the Last Chunk is => 1967 characters

Recursive text splitter performs well in dealing with generic text

Split by Tokens

For those well versed with Large Language Models, tokens is not a new concept. All LLMs have a token limit in their respective context windows which we cannot exceed. It is therefore a good idea to count the tokens while creating chunks. All LLMs also have their tokenizers.

Tiktoken Tokenizer

OpenAI

Tiktoken tokenizer has been created by OpenAI for their family of models. Using this strategy, the split still happens based on the character. However, the length of the chunk is determined by the number of tokens.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **LangChain's TokenTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import TokenTextSplitter
import tiktoken

text_splitter = TokenTextSplitter(
    chunk_size = 1000,
    chunk_overlap  = 20,
    length_function = len,

)

texts = text_splitter.create_documents([IntroToLLM])

encoding = tiktoken.get_encoding("cl100k_base")

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Total Number of Tokens in the document => {len(encoding.encode(IntroToLLM))} tokens")

print(f"Length of the First Chunk is => {len(encoding.encode(texts[0].page_content))} tokens")

print(f"Length of the Last Chunk is => {len(encoding.encode(texts[0].page_content))} tokens")
```

Total Number of Chunks Created => 14
Total Number of Tokens in the document => 12865 tokens
Length of the First Chunk is => 1014 tokens
Length of the Last Chunk is => 1014 tokens

Tokenizers are helpful in creating chunks that sit well in the context window of an LLM

Hugging Face Tokenizer

Hugging Face

Hugging Face has become the go-to platform for anyone building apps using LLMs or even other models. All models available via Hugging Face are also accompanied by their tokenizers.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **Transformers** and **LangChain's RecursiveCharacterTextSplitter**

Example tokenizer : GPT2TokenizerFast

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from transformers import GPT2TokenizerFast
from langchain.text_splitter import RecursiveCharacterTextSplitter


tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")

text_splitter = RecursiveCharacterTextSplitter.from_huggingface_tokenizer(
    tokenizer, chunk_size=100, chunk_overlap=0
)

texts = text_splitter.split_text(IntroToLLM)

print(texts[0])
print(texts[1])
```

texts[0]
"hi everyone so recently I gave a 30-minute talk on large language models just kind of like an intro talk um unfortunately that talk was not recorded but a lot of people came to me after the talk and they told me that uh they really liked the talk so I would just I thought I would just re-record it and basically put it up on YouTube so here we go the busy person's intro to large language models director Scott okay so let's begin first of all what is a large language model

Chunk 1

No Overlap as specified

texts[1]
really well a large language model is just two files right um there be two files in this hypothetical directory so for example work with the specific example of the Llama 270b model this is a large language model released by meta Ai and this is basically the Llama series of language models the second iteration of it and this is the 70 billion parameter model of uh of this series so there's multiple models uh belonging to the Lama 2 Series uh 7 billion um 13 billion 34 billion and 70 billion is the the

Chunk 2

Do take a look at Hugging Face documents on Tokenizers

Hugging Face

https://huggingface.co/docs/transformers/tokenizer_summary

Other Tokenizer

Other libraries like Spacy, NLTK and SentenceTransformers also provide splitters

```python
from langchain.text_splitter import NLTKTextSplitter

text_splitter = NLTKTextSplitter(chunk_size=1000)

texts = text_splitter.split_text(IntroToLLM)
print(texts[0])
```

```python
from langchain.text_splitter import SpacyTextSplitter

text_splitter = SpacyTextSplitter(chunk_size=1000)

texts = text_splitter.split_text(IntroToLLM)
print(texts[0])
```

Specialized Chunking

Chunking often aims to keep text with common context together. With this in mind, we might want to specifically honour the structure of the document itself for example HTML, Markdown, Latex or even code.

Example : https://medium.com/p/29a7e8610843

Example HTML : “Context is Key: The Significance of RAG in Language Models”
(A blog on Medium - https://medium.com/p/29a7e8610843)

using **LangChain’s HTMLHeaderTextSplitter** & **RecursiveCharacterTextSplitter**

```python
from langchain.text_splitter import HTMLHeaderTextSplitter

from langchain.text_splitter import RecursiveCharacterTextSplitter

url = "https://medium.com/p/29a7e8610843"

headers_to_split_on = [
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("h4", "Header 4"),
]

html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)

# for local file use html_splitter.split_text_from_file(<path_to_file>)
html_header_splits = html_splitter.split_text_from_url(url)

chunk_size = 5000
chunk_overlap = 300
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size, chunk_overlap=chunk_overlap
)

# Split
splits = text_splitter.split_documents(html_header_splits)
len(splits)
```

LangChain All LangChain Splitters

Things to Keep in Mind

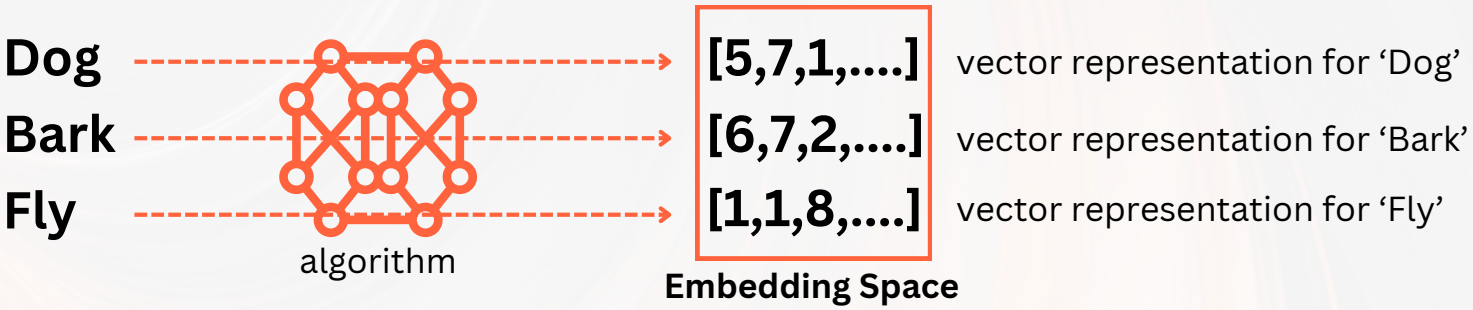
- Ensure data quality by preprocessing it before determining the optimal chunk size. Examples include removing HTML tags or eliminating specific elements that contribute noise, particularly when data is sourced from the web.

- Consider factors such as content nature (e.g., short messages or lengthy documents), embedding model characteristics, and capabilities like token limits in choosing chunk sizes. Aim for a balance between preserving context and maintaining accuracy.

- Test different chunk sizes. Create embeddings for the chosen chunk sizes and store them in your index or indices. Run a series of queries to evaluate quality and compare the performance of different chunk sizes.

Embeddings

All Machine Learning/AI models work with numerical data. Before the performance of any operation all text/image/audio/video data has to be transformed into a numerical representation. **Embeddings are vector representations of data that capture meaningful relationships between entities**. As a general definition, embeddings are data that has been transformed into n-dimensional matrices for use in deep learning computations. A word embedding is a vector representation of words.



The process of embedding **transforms** data (like text) into vectors, **compresses** the input information resulting in an **embedding space** *specific to the training data*

While we keep our discussion around embeddings limited to RAG application and how to create embeddings for our data, a great resource to find more about embeddings is this book by Vicky Boykis [What are embeddings]

The good news for anyone building RAG Applications is that embeddings once created can also generalize to other tasks and domains through **transfer learning** — the ability to switch contexts — which is one of the reasons embeddings have exploded in popularity across machine learning applications

Popular Embedding Models

word2vec Google's Word2Vec is one of the most popular pre-trained word embeddings. The official paper - https://arxiv.org/pdf/1301.3781.pdf

GLOVE The 'Global Vectors' model is so termed because it captures statistics directly at a global level. The official paper - https://nlp.stanford.edu/pubs/glove.pdf

fastText Facebook's AI research, fastText builds embeddings composed of characters instead of words. The official paper - https://arxiv.org/pdf/1607.04606.pdf

Elmo Embeddings from Language Models, are learnt from the internal state of a bidirectional LSTM. The official paper - https://arxiv.org/pdf/1802.05365.pdf

BERT Bidirectional Encoder Representations from Transformers is a transformer bases approach. The official paper - https://arxiv.org/pdf/1810.04805.pdf

ada v2 by OpenAI

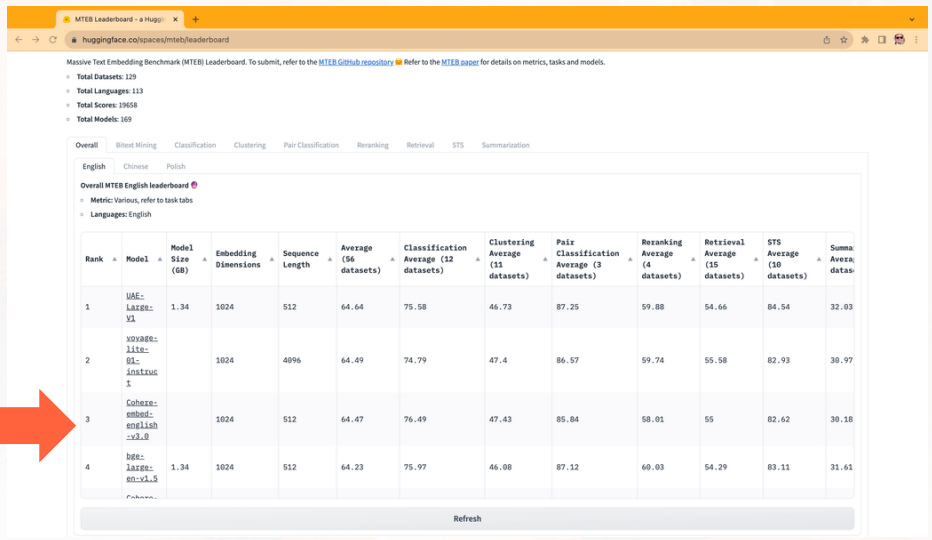
used by GPT series of models

textembedding-gecko

by Google's Vertex.ai

Other Open Source Embeddings

Checkout MTEB leaderboard at Hugging Face



How to Choose Embeddings?

Ever since the release of ChatGPT and the advent of the aptly described LLM Wars, there has also been a mad rush in developing embeddings models. There are many evolving standards of evaluating LLMs and embeddings alike.

When building RAG powered LLM apps, there is no right answer to "Which embeddings model to use?". However, you may notice particular embeddings working better for specific use cases (like summarization, text generations, classification etc.)

	Models	Use Cases
Text similarity: Captures semantic similarity between pieces of text.	text-similarity-{ada, babbage, curie, davinci}-001	Clustering, regression, anomaly detection, visualization
Text search: Semantic information retrieval over documents.	text-search-{ada, babbage, curie, davinci}-{query, doc}-001	Search, context relevance, information retrieval
Code search: Find relevant code with a query in natural language.	code-search-{ada, babbage}-{code, text}-001	Code search and relevance

OpenAI used to recommend different embeddings models for different use cases. However, now they recommend **ada v2** for all tasks.

Average (56 datasets)	Classification Average (12 datasets)	Clustering Average (11 datasets)	Pair Classification Average (3 datasets)	Reranking Average (4 datasets)	Retrieval Average (15 datasets)	STS Average (10 datasets)	Summarization Average (1 dataset)

MTEB Leaderboard at Hugging Face evaluates almost all available embedding models across seven use cases - *Classification*, *Clustering*, *Pair Classification*, *Reranking*, *Retrieval*, *Semantic Textual Similarity (STS)* and *Summarization*.

Another important consideration is **cost**. With OpenAI models you can incur significant costs if you are working with a lot of documents. The cost of open source models will depend on the implementation.

Creating Embeddings

Once you've chosen your embedding model, there are several ways of creating the embeddings. Sometimes, our friends, LlamaIndex and LangChain come in pretty handy to convert documents (*split into chunks*) into vector embeddings. Other times you can use the service from a provider directly or get the embeddings from HuggingFace

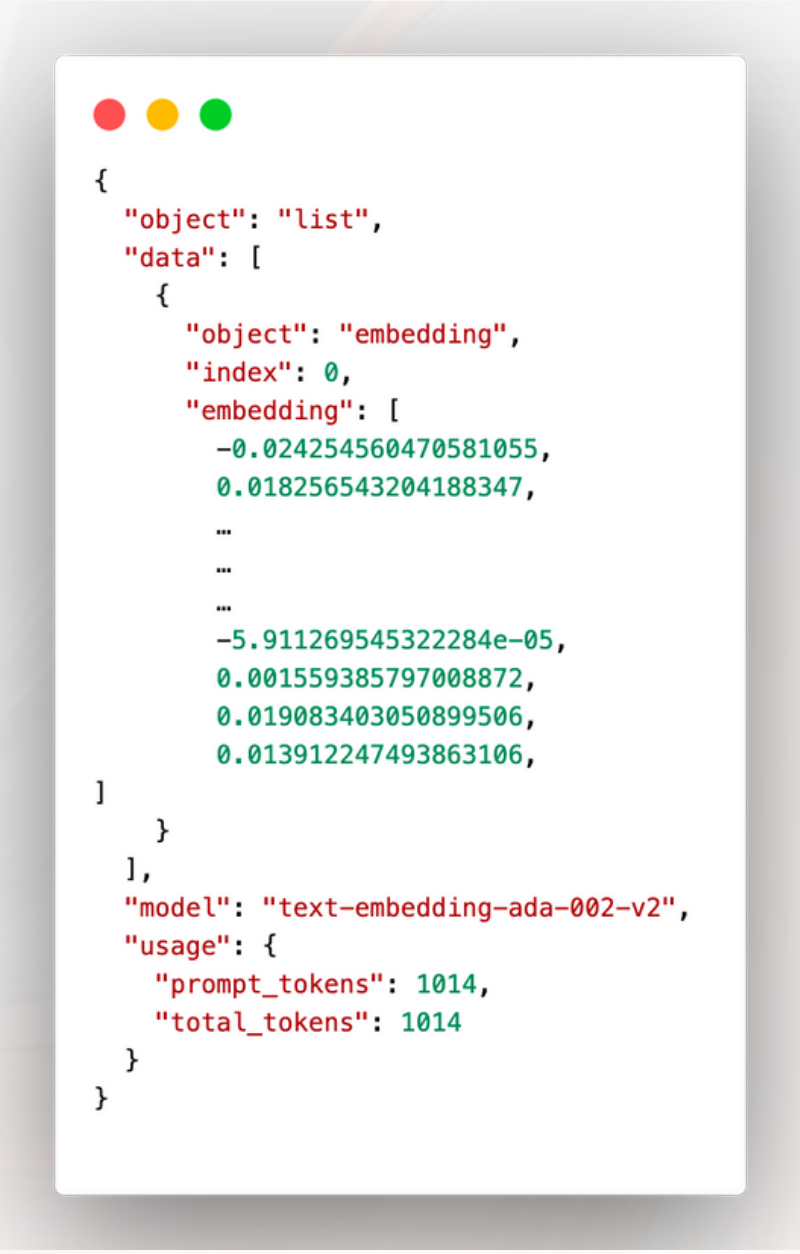
Example : OpenAI text-embedding-ada-002

using Embedding.create() function from openai library

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import TokenTextSplitter

text_splitter = TokenTextSplitter(
    chunk_size = 1000,
    chunk_overlap  = 20,
    length_function = len,

)

texts = text_splitter.create_documents([IntroToLLM])

import openai
openai.api_key='sk-#####eGeY##########RZj'

response = openai.Embedding.create(
    input=texts[0].page_content,
    model="text-embedding-ada-002"
)

print(response)
```

You'll need an OpenAI apikey to create these embeddings
You can get one here - https://platform.openai.com/api-keys

Example Response



```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "index": 0,
      "embedding": [
        -0.024254560470581055,
        0.018256543204188347,
        …
        …
        …
        -5.911269545322284e-05,
        0.001559385797008872,
        0.019083403050899506,
        0.013912247493863106,
]
    }
  ],
  "model": "text-embedding-ada-002-v2",
  "usage": {
    "prompt_tokens": 1014,
    "total_tokens": 1014
  }
}
```

response.data[0].embedding will give the created embeddings that can be stored for retrieval

Cost

Model	Usage
ada v2	$0.0001 / 1K tokens

In this example, 1014 tokens will cost about $.0001. Recall that for this youtube transcript we got 14 chunks. So creating the embeddings for the entire transcript will cost about 0.14 cents. This may seem low, but when you scale up to thousands of documents being updated frequently, the cost can become a concern.

Example : msmarco-bert-base-dot-v5

using HuggingFaceEmbeddings from langchain.embeddings

```
from langchain.embeddings import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(
    model_name='sentence-transformers/msmarco-bert-base-dot-v5'
    )
text = texts[0].page_content

query_result = embeddings.embed_query(text)
```

Example : embed-english-light-v3.0

using CohereEmbeddings from langchain.embeddings

```
from langchain.embeddings import CohereEmbeddings

embeddings = CohereEmbeddings(
    model="embed-english-light-v3.0")

text = texts[0].page_content

query_result = embeddings.embed_query(text)
```

LangChain

All the available embeddings classes on LangChain

Storing

We are at the last step of creating the indexing pipeline. We have loaded and split the data, and created the embeddings. Now, for us to be able to use the information repeatedly, we need to store it so that it can be accessed on demand. For this we use a special kind of database called the **Vector Database**.

What is a Vector Database?

For those familiar with databases, **indexing** is a data structure technique that allows users to quickly retrieve data from a database. Vector databases specialise in indexing and storing embeddings for **fast retrieval** and **similarity search**.

A strip down variant of a Vector Database is a Vector Index like FAISS (Facebook AI Similarity Search). It is this vector indexing that improves the search and retrieval of vector embeddings. Vector Databases augment the indexing with typical database features like data management, metadata storage, scalability, integrations, security etc.

In short, Vector Databases provide -

- Scalable Embedding Storage.
- Precise Similarity Search.
- Faster Search Algorithm.

Popular Vector Databases

Facebook AI Similarity search is a vector index released with a library in 2017

Pinecone is one of the most popular managed Vector DB for large scale

Weaviate is an open source vector database that stores both objects and vectors

Chromadb is also an open source vector database.

With the growth in demand for vector storage, it can be anticipated that all major database players will add the vector indexing capabilities to their offerings.

How to choose a Vector Database?

All vector databases offer the same basic capabilities. Your choice should be influenced by the nuance of your use case matching with the value proposition of the database.

A few things to consider -

- Balance search accuracy and query speed based on application needs. Prioritize accuracy for precision applications or speed for real-time systems.
- Weigh increased flexibility vs potential performance impacts. More customization can add overhead and slow systems down.
- Evaluate data durability and integrity requirements vs the need for fast query performance. Additional persistence safeguards can reduce speed.
- Assess tradeoffs between local storage speed and access vs cloud storage benefits like security, redundancy and scalability.
- Determine if tight integration control via direct libraries is required or if ease-of-use abstractions like APIs better suit your use case.
- Compare advanced algorithm optimizations, query features, and indexing vs how much complexity your use case necessitates vs needs for simplicity.
- Cost considerations - while you many incur regular cost in a fully managed solution, a self hosted one might prove costlier if not managed well

User Friendly for PoCs

Chroma

Higher Performance

FAISS

Weaviate

Customization

Pinecone

There are many more Vector DBs. For a comprehensive understanding of the pros and cons of each, this blog is highly recommended