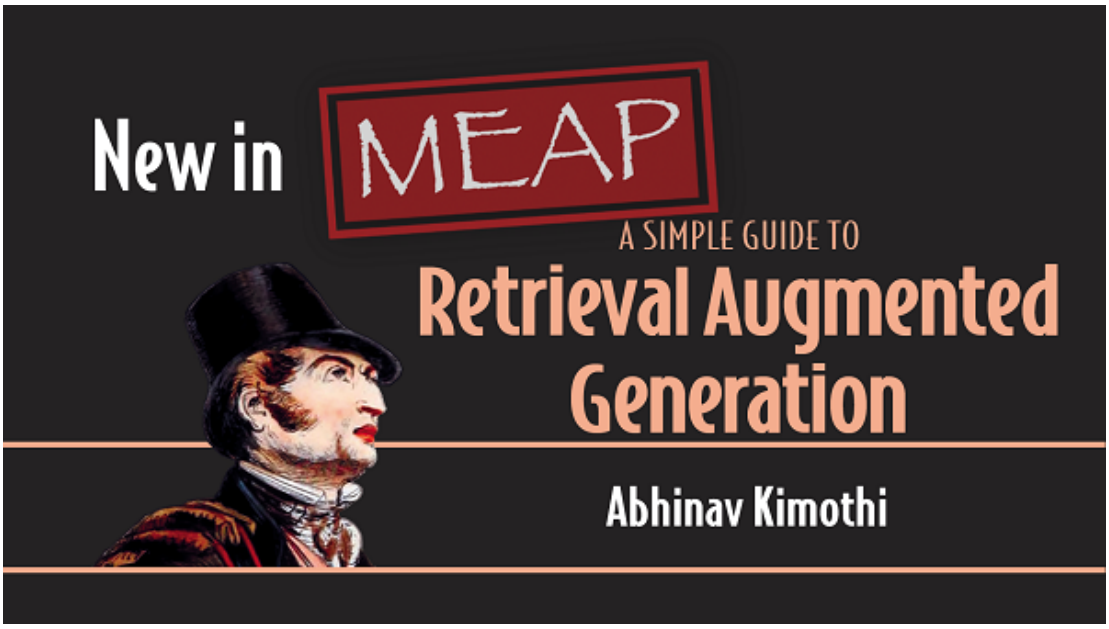
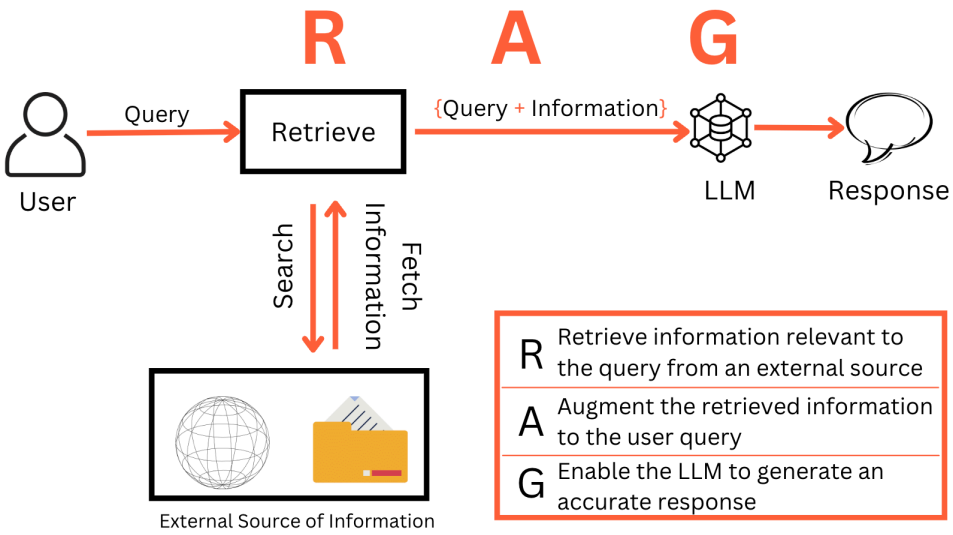


Introduction to Retrieval Augmented Generation (RAG)

01 March 2025

Abhinav Kimothi

Retrieval Augmented Generation, or RAG, has emerged to be one of the most popular techniques in the applied generative AI world. Large Language Models, or LLMs, is a generative AI technology that has recently gained tremendous popularity. However, despite their unprecedented ability to generate text, their responses are **not always correct**. Upon more careful observation, you may notice that LLM responses are plagued with **sub-optimal information** and **inherent memory limitations**. RAG addresses these limitations of LLMs by providing them with information **external** to these models. Thereby, resulting in LLM responses that are more reliable and trustworthy.



> The technique of **enhancing the parametric memory** of an LLM by creating access to an **explicit non-parametric memory**, from which a **retriever** can fetch relevant information, augment that information to the prompt, pass the prompt to an LLM to enable the LLM to generate a response that is **contextual, reliable, and factually accurate** is called **Retrieval Augmented Generation**

For an in-depth understanding of RAG, read A Simple Guide to Retrieval Augemnted Generation

---

About this notebook

This is a supplementary notebook for the session **Introduction to Retrieval Augmented Generation** by *Abhinav Kimothi* in week 7 of the **Artificial Intelligence: Generative AI, Cloud and MLOps (online)** course by *Department for Continuing Education at the University of Oxford*.

Contents

This notebook contains code in python and leverages the LangChain framework to build and evaluate the different components of a RAG pipeline.

- Indexing Pipeline
  - Data Loading
  - Chunking (or Data Splitting)
  - Embeddings (or Data Transformation)
  - Storage (Vector Databases)
- Generation Pipeline
  - Search & Retrieval
  - Prompt Augmentation
  - LLM Generation
- RAG Evaluation using RAGAs Framework
  - Synthetic Dataset Generation
  - Calculation of Evaluation Metrics

Structure

Each section of this notebook first **demonstrates** the components using an example and is followed by an **exercise for you to solve**.

Demonstration

This notebook demonstrates **RAG using a webpage on the internet**. We know that LLMs inherently **do not have access to the internet** and have a **knowledge cut-off date** that prevents them from having access to latest information. RAG with websearch overcomes this limitation.

In this notebook, we take the example of the **Wikipedia Article on 2023 Men's ODI Cricket World Cup**

Exercise

LLMs also do not have access to any data that is not in their training set. Therefore, **proprietary data is not available** to LLMs. This where RAG helps in searching through proprietary data files.

In this notebook, you'll be asked to build a RAG system on a PDF file which is neither available on the internet, nor is a part of any LLMs training dataset.

I hope you have as much fun going through this notebook, as I had while creating it. Let's get started!!

> Important Note: This notebook requires OpenAI credits. The notebook uses **Text-embedding-3-small** embeddings model and **GPT-4o-mini** LLM.

> **Running the entire notebook once including the exercises will cost about USD $0.11 (11 cents)**

Installing Dependencies

All the necessary libraries for running this notebook along with their versions can be found in **requirements.txt** file in the root directory of this repository

You should go to the root directory and run the following command to install the libraries

```
pip install -r requirements.txt
```

This is the recommended method of installing the dependencies

Alternatively, you can run the command from this notebook too. The relative path may vary so ensure that you are in the root directory of this repository

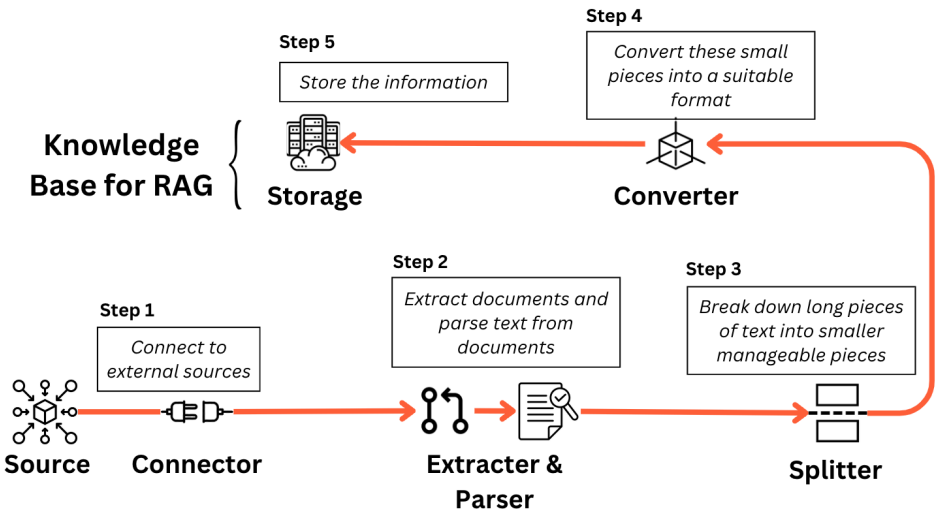
In [ ]:
```
%pip install -r ./requirements.txt --quiet
```

Indexing Pipeline

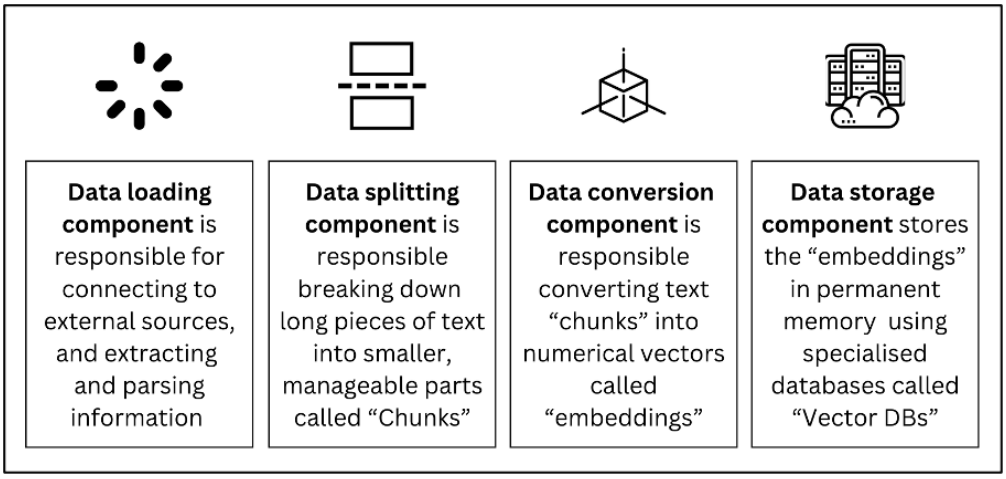
A core RAG system contains two pipelines -

- Indexing Pipeline that creates the external knowledge base
- Generation Pipeline that facilitates real-time interaction with the knowledge base

The indexing pipeline can be understood in five steps



There are consequently **four components** that facilitate these five steps -



Let's take a look at these four components

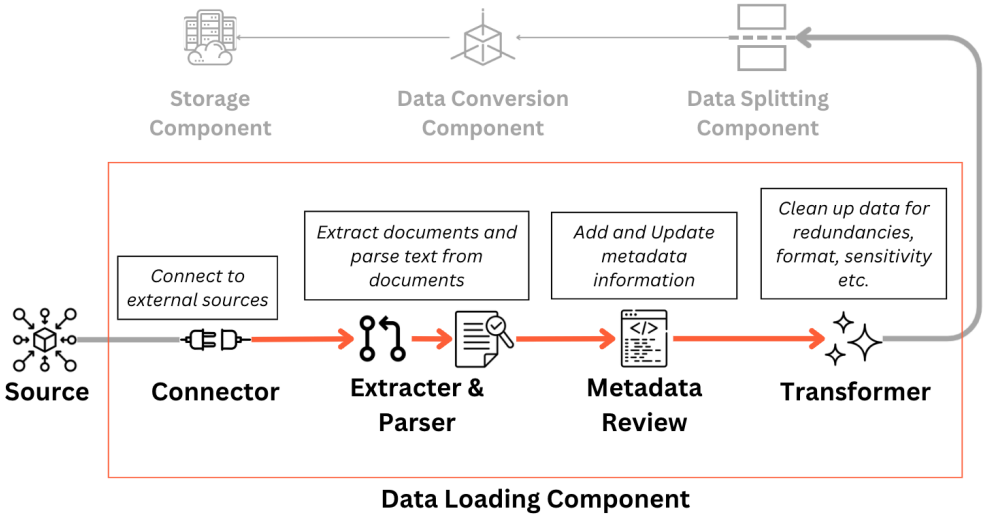
Data Loading

What is Data Loading?

The first step towards building a knowledge base (or non-parametric memory) of a RAG-enabled system is to source data from its original location. This data may be in the form of word documents, pdf files, csv, HTML etc. Further, the data may be stored in file, block or object stores, in data lakes, data warehouses or even in third party sources that can be accessed via the open internet. This process of sourcing data from its original location is called **Data Loading**.

Data Loading includes the following four steps:

- **Connection** to the source of the data
- **Extraction and Parsing of text** from the source format
- Reviewing and updating **metadata** information
- Cleaning or **transforming** the data



Connecting & Parsing an external URL

Let us load the url of our example i.e. the Wikipedia Page of the 2023 Cricket World Cup

In [ ]:
```
#This is the url of the wikipedia page on the 2023 Cricket World Cup
url="https://en.wikipedia.org/wiki/2023_Cricket_World_Cup"
```

LangChain provides a wide array of **document loaders** (over 100) that help in loading data from a large number of data sources like *Webpages, PDF files, Cloud Storage Systems, Social Platforms, Messaging Services, Tools and more*.

Here we use one of them, **AsyncHtmlLoader** that loads the HTML data from web URLs.

In [ ]:
```
#Import library
from langchain_community.document_loaders import AsyncHtmlLoader

#Instantiate the AsyncHtmlLoader object
loader = AsyncHtmlLoader (url)

#Loading the extracted information
html_data = loader.load()
```

To verify the extracted text and the metadata, let us print a few tokens

In [ ]:
```
import textwrap

print(textwrap.fill(f"First 1000 characters of extracted content -\n\n{ht
```

Metadata Review

In [ ]:
```
print(f"Metadata information - \n\n{html_data[0].metadata}")
```

We can see that some **content has been extracted**. Also, some **metadata** information is present.

Document Transformation

The content is in **HTML format** which does not convey a lot of factual information.

LangChain also provides a bunch of document transformers for converting formats.

We will now transform this data into a readable format using **Html2TextTransformer** class.

In [ ]:
```
from langchain_community.document_transformers import Html2TextTransforme

#Instantiate the Html2TextTransformer function
html2text = Html2TextTransformer()


#Call transform_documents
html_data_transformed = html2text.transform_documents(html_data)
```

Let us review the extracted content, now transformed by the Html2TextTransformer

In [ ]:
```
print(f"First 100 characters of extracted content -\n\n{html_data_transfo
```

Now, we see that we have text in a **readable english** language!

Optional: BeautifulSoupTransformer

But you may notics that there's a lot of information like Menu Options, Header and footer information that may not be very useful.

Another options is the **BeautifulSoupTransformer** in LangChain that allows you to extract specific tags from HTML pages. Let us extract information contained in 'p' tags.

In [ ]:
```
from langchain_community.document_transformers import BeautifulSoupTransf

soup_transformer = BeautifulSoupTransformer()

html_data_p_tags = soup_transformer.transform_documents(html_data, tags_t
```

In [ ]:
```
print(textwrap.fill(
f"First 100 characters of extracted content -\n\n{html_data_p_tags[0].pag
```

We have seen how to load text from an external source.

Now it's time for you to try data loading!

Exercise: PDF Document (Employee Leave Policy)

The file that you're going to read is the **Employee Leave Policy** of a fictitious organisation named AKAIWorks LLP. The file is present in a **PDF format** and the folder location is **'./Assets/Data/'**

In [ ]:
```
filepath='./Assets/Data/EmployeeLeavePolicy.pdf'
```

Exercise:

Your task is to load the pdf file using the PyPDFLoader and print the first 1000 characters of the text that has been read

In [ ]:
```
from langchain_community.document_loaders import PyPDFLoader

#START YOUR CODE HERE


#END YOUR CODE HERE
```

▶ Click for Hint

▶ Click for solution

Congratulations

With this, you have successfully completed the data loading step of the indexing pipeline. We move now to the next step of **Chunking**

But before that, check out the document loaders and transformers available in LangChain

Document Loaders -
[https://python.langchain.com/docs/integrations/document_loaders/]

Document Transformers -
[https://python.langchain.com/docs/integrations/document_transformers/]

---

2. Data Splitting or Chunking

Breaking down long pieces of text into manageable sizes is called **Data Splitting** or **Chunking**. This is done for various reasons like Context Window Limitations, Search Complexity, Lost in the middle kind of issues.

Understanding Chunking: What is it ?

In cognitive psychology, chunking is defined as process by which individual pieces of information are bound together into a meaningful whole. (https://psycnet.apa.org/record/2003-09163-002) and a chunk is a familiar collection of elementary units. The idea is that chunking is an essential technique through which human beings perceive the world and commit to memory. The simplest example is how we remember long sequences of digits like phone numbers, credit card numbers, dates or even OTPs. We don't remember the entire sequences but in our minds, we break them down into chunks.

The role of chunking in RAG and the underlying idea is somewhat similar to what it is in real life. Once you've extracted and parsed text from the source, instead of committing it all to memory as a single element, you break it down into smaller chunks.

> Breaking down long pieces of text into manageable sizes is called Chunking

Understanding Chunking: Why is it necessary ?

There are two main benefits of chunking —

- It leads to better retrieval of information. If a chunk represents a single idea (or fact) it can be retrieved with more confidence that if there are multiple ideas (or facts) within the same chunk.

- It leads to better generation. The retrieved chunk has information that is focussed on the user query and does not have any other text that may confuse the LLM. Therefore, the generation is more accurate and coherent.

Apart from these two benefits there are two limitations of LLMs that chunking addresses.

- **Context Window of LLMs**: LLMs, due to the inherent nature of the technology, have a limit on the number of tokens (loosely, words) they can work with at a time. This includes both the number of tokens in the prompt (or the input) and the number of tokens in the completion (or the output). The limit on the total number of tokens that an LLM can process in one go is called the context window size. If we pass an input that is longer than the context window size, the LLM chooses to ignore all text beyond the size. It becomes very important to be careful with the amount to text that is being passed to the LLM.
- **Lost in the middle problem**: Even in those LLMs which have a long context window (Claude 3 by Anthropic has a context window of up to 200,00 tokens), an issue with accurately reading the information has been observed. It has been noticed that accuracy declines dramatically if the relevant information is somewhere in the middle of the prompt. This problem can be addressed by passing only the relevant information to the LLM instead of the entire document.

Fixed Size Chunking

A very common approach is to pre-determine the size of the chunk and the amount of overlap between the chunks. There are several chunking methods that follow a fixed size chunking approach.

- Character-Based Chunking: Chunks are created based on a fixed number of characters
- Token-Based Chunking: Chunks are created based on a fixed number of tokens.
- Sentence-Based Chunking: Chunks are defined by a fixed number of sentences
- Paragraph-Based Chunking: Chunks are created by dividing the text into a fixed number of paragraphs.

Let's try Character-Based Chunking.

In [ ]:
```
from langchain.text_splitter import RecursiveCharacterTextSplitter #Chara

text_splitter = RecursiveCharacterTextSplitter(
separators=["\n","."], #The character that should be used to split. More
chunk_size=1000, #Number of characters in each chunk
chunk_overlap=100, #Number of overlapping characters between chunks
)

text_chunks=text_splitter.create_documents([html_data_transformed[0].page
```

```
#Show the number of chunks created
print(f"The number of chunks created : {len(text_chunks)}")
```

Now, let's see the size distribution of the chunks that have been created

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np


data = [len(doc.page_content) for doc in text_chunks]

plt.boxplot(data)
plt.title('Box Plot of chunk lengths')
plt.xlabel('Chunk Lengths')
plt.ylabel('Values')

plt.show()

print(f"The median chunk length is : {round(np.median(data),2)}")
print(f"The average chunk length is : {round(np.mean(data),2)}")
print(f"The minimum chunk length is : {round(np.min(data),2)}")
print(f"The max chunk length is : {round(np.max(data),2)}")
print(f"The 75th percentile chunk length is : {round(np.percentile(data,
print(f"The 25th percentile chunk length is : {round(np.percentile(data,
```

Document-structured based Chunking

The aim of chunking is to keep meaningful data together. If we are dealing with data in form of HTML, Markdown, JSON or even computer code, it makes more sense to split the data based on the structure rather than a fixed size. Another approach for chunking is to take into consideration the format of the extracted and loaded data. A markdown file, for example is organised by headers, a code written in a programming language like python or java is organized by classes and functions and HTML, likewise, is organised in headers and sections. For such formats a specialised chunking approach can be employed.

Examples of structure-based splitting:

- Markdown: Split based on headers (e.g., #, ##, ###)
- HTML: Split using tags
- JSON: Split by object or array elements
- Code: Split by functions, classes, or logical blocks

Let's recollect out HTML document from the url.

In [ ]:
```
loader = AsyncHtmlLoader (url)

html_data = loader.load()
```

To split the HTML text based on tags (e.g., h1, section, table, etc.), LangChain provides **HTMLSectionSplitter**. It splits the text and adds metadata for each section. Let's take a look.

In [ ]:
```
from langchain_text_splitters import HTMLSectionSplitter

sections_to_split_on = [
    ("h1", "Header 1"),
     ("h2", "Header 2"),
     ("table","Table"),
     #("div", "Div"),
     #("img","Image"),
     ("p","P"),


]

splitter = HTMLSectionSplitter(sections_to_split_on)

split_content = splitter.split_text(html_data[0].page_content)
```

The above document object '**split_content**' will have chunks divided based on the provided HTML tags. Let's look at the top 10 documents.

In [ ]:
```
split_content[:10]
```

We can see the metatadata indicating the section tag of the chunk. So how many chunks were created?

In [ ]:
```
len(split_content)
```

Let's see how many chunks for each of the sections

In [ ]:
```
from collections import Counter

class_counter = Counter()

for doc in split_content:
    document_class = next(iter(doc.metadata.keys()))
    class_counter[document_class] += 1

print(class_counter)
```

Now, let us look at the lengths of these chunks

In [ ]:
```
data = [len(doc.page_content) for doc in split_content]

plt.boxplot(data)
plt.title('Box Plot of chunk lengths')
plt.xlabel('Chunk Lengths')
plt.ylabel('Values')

plt.show()

print(f"The median chunk lenght is : {round(np.median(data),2)}")
print(f"The average chunk lenght is : {round(np.mean(data),2)}")
print(f"The minimum chunk lenght is : {round(np.min(data),2)}")
print(f"The max chunk lenght is : {round(np.max(data),2)}")
print(f"The 75th percentile chunk length is : {round(np.percentile(data,
print(f"The 25th percentile chunk length is : {round(np.percentile(data,
```

Some of the chunk lengths are longer than 1000. Let's try to control that.

In [ ]:
```
text_splitter = RecursiveCharacterTextSplitter(
separators=["\n\n","\n","."], #The character that should be used to split
chunk_size=1000, #Number of characters in each chunk
chunk_overlap=100, #Number of overlapping characters between chunks
)

final_chunks=text_splitter.split_documents(split_content)

#Show the number of chunks created
print(f"The number of chunks created : {len(final_chunks)}")
```

In [ ]:
```
data = [len(doc.page_content) for doc in final_chunks]

plt.boxplot(data)
plt.title('Box Plot of chunk lengths')  # Title
plt.xlabel('Chunk Lengths')  # Label for x-axis
plt.ylabel('Values')  # Label for y-axis

plt.show()

print(f"The median chunk lenght is : {round(np.median(data),2)}")
print(f"The average chunk lenght is : {round(np.mean(data),2)}")
print(f"The minimum chunk lenght is : {round(np.min(data),2)}")
print(f"The max chunk lenght is : {round(np.max(data),2)}")
print(f"The 75th percentile chunk length is : {round(np.percentile(data,
print(f"The 25th percentile chunk length is : {round(np.percentile(data,
```

There, we have our final chunks!

Exercise: PDF Document (Employee Leave Policy)

In the previous exercise you read the **Employee Leave Policy** of AKAIWorks LLP. Now, you should use the recursive character splitter to create chunks of this document.

In [ ]:
```
pdfloader=PyPDFLoader(file_path=filepath,mode="single") #instantiate the

pdf_data=pdfloader.load() #load the data

print(len(pdf_data[0].page_content))
```

Your task is to now chunk this document into manageable sizes using **RecursiveCharacterTextSplitter** and store in document object `pdf_doc_chunks` and print the number of chunks created.

In [ ]:
```
#START YOUR CODE HERE



#END YOUR CODE HERE
```

▶ Click for Hint

▶ Click for Solution

Let's check out the distribution of chunk sizes.

Run the cell below.

Remember the document object should be called `pdf_doc_chunks`

In [ ]:
```
data = [len(doc.page_content) for doc in pdf_doc_chunks]

plt.boxplot(data)
plt.title('Box Plot of chunk lengths')  # Title
plt.xlabel('Chunk Lengths')  # Label for x-axis
plt.ylabel('Values')  # Label for y-axis

plt.show()

print(f"The median chunk lenght is : {round(np.median(data),2)}")
print(f"The average chunk lenght is : {round(np.mean(data),2)}")
print(f"The minimum chunk lenght is : {round(np.min(data),2)}")
print(f"The max chunk lenght is : {round(np.max(data),2)}")
print(f"The 75th percentile chunk length is : {round(np.percentile(data,
print(f"The 25th percentile chunk length is : {round(np.percentile(data,
```

Congratulations

With this, you have successfully completed the chunking of the data. We move now to the next step of creating **Embeddings**

But before that, check out the splitters available in LangChain

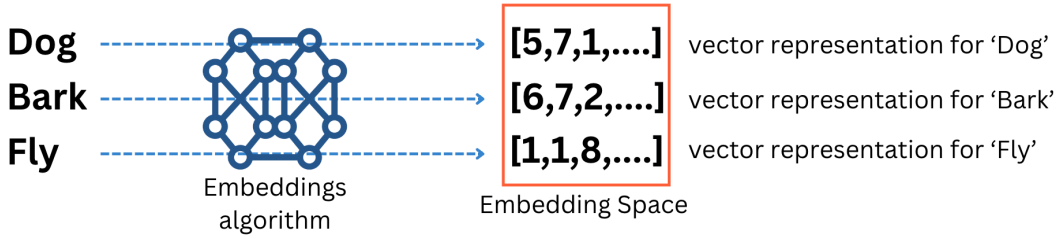
Text Splitters - [https://python.langchain.com/docs/concepts/text_splitters/]

---

3. Data Transformation or Embeddings

Computers, at the very core, do mathematical calculations. Mathematical calculations are done on numbers. Therefore, for a computer to process any kind of non-numeric data like text or image, it must be first converted into a numerical form.

Embeddings is a design pattern that is extremely helpful in the fields of data science, machine learning and artificial intelligence. Embeddings are vector representations of data. As a general definition, embeddings are data that has been transformed into n-dimensional matrices. A word embedding is a vector representation of words.



Open Source Embeddings from HuggingFace

Let's begin with an opensource embeddings from HuggingFace!

In [ ]:
```
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-

hf_embeddings = embeddings.embed_documents([chunk.page_content for chunk
```

In [ ]:
```
print(f"The lenght of the embeddings vector is {len(hf_embeddings[0])}")
print(f"The embeddings object is an array of {len(hf_embeddings)} X {len(
```

OpenAI Embeddings

OpenAI, the company behind ChatGPT and GPT series of Large Language Models also provide three Embeddings Models.

1. text-embedding-ada-002 was released in December 2022. It has a dimension of 1536 meaning that it converts text into a vector of 1536 dimensions.
2. text-embedding-3-small is the latest small embedding model of 1536 dimensions released in January 2024. The flexibility it provides over ada-002 model is that users can adjust the size of the dimensions according to their needs.
3. text-embedding-3-large is a large embedding model of 3072 dimensions released together with the text-embedding-3-small model. It is the best performing model released by OpenAI yet.

OpenAI models are proprietary and can be accessed using the OpenAI API and are priced based on the number of input tokens for which embeddings are desired.

Note: You will need an **OpenAI API Key** which can be obtained from OpenAI

To initialize the **OpenAI client**, we need to pass the api key. There are many ways of doing it.

[Option 1] Creating a .env file for storing the API key and using it # Recommended

Install the **dotenv** library

The dotenv library is a popular tool used in various programming languages, including Python and Node.js, to manage environment variables in development and deployment environments. It allows developers to load environment variables from a .env file into their application's environment.

- Create a file named .env in the root directory of their project.
- Inside the .env file, then define environment variables in the format VARIABLE_NAME=value.

e.g.

OPENAI_API_KEY=YOUR API KEY

In [ ]:
```
from dotenv import load_dotenv
import os

if load_dotenv():
    print("Success: .env file found with some environment variables")
else:
    print("Caution: No environment variables found. Please create .env fi
```

[Option 2] Alternatively, you can set the API key in code.

However, this is not recommended since it can leave your key exposed for potential misuse. Uncomment the cell below to use this method.

In [ ]:
```
#os.environ["OPENAI_API_KEY"] = "sk-proj-xxxxxxxxxxxx"
```

We can also test if the key is valid or not

In [ ]:
```
api_key=os.environ["OPENAI_API_KEY"]

from openai import OpenAI

client = OpenAI()


if api_key:
    try:
        client.models.list()
        print("OPENAI_API_KEY is set and is valid")
    except openai.APIError as e:
        print(f"OpenAI API returned an API Error: {e}")
        pass
    except openai.APIConnectionError as e:
        print(f"Failed to connect to OpenAI API: {e}")
        pass
    except openai.RateLimitError as e:
        print(f"OpenAI API request exceeded rate limit: {e}")
        pass

else:
    print("Please set you OpenAI API key as an environment variable OPENA
```

Now we will use the **OpenAIEmbeddings** library from langchain

In [ ]:
```
# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings


os.environ["TOKENIZERS_PARALLELISM"]="false"

# Instantiate the embeddings object
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
```

```
# Create embeddings for all chunks
openai_embeddings = embeddings.embed_documents([chunk.page_content for ch
```

In [ ]:
```
print(f"The lenght of the embeddings vector is {len(openai_embeddings[0])
print(f"The embeddings object is an array of {len(openai_embeddings)} X {
```

In [ ]:
```
openai_embeddings
```

Exercise: PDF Document (Employee Leave Policy)

Now, the chunks that you created in the previous section you will need to convert them into embeddings.

Use **text-embedding-3-small** embeddings using **OpenAIEmbeddings** from langchain. Store the embeddings in `pdf_doc_embeddings`

In [ ]:
```
#START YOUR CODE HERE



#END YOUR CODE HERE
```

▶ Click for Hint

▶ Click for Solution

In [ ]:
```
print(f"The lenght of the embeddings vector is {len(pdf_doc_embeddings[0]
print(f"The embeddings object is an array of {len(pdf_doc_embeddings)} X
```

Congratulations

With this, you have successfully completed the creation of embeddings. We move now to the next step of storing the embeddings in a **Vector Store**

Read more about Embedding Models Here

---

4. Vector Storage

The data has been loaded, split, and converted into embeddings. For us to use this information repeatedly, we need to store it in memory so that it can be accessed on demand. Vector Databases are built to handle high dimensional vectors. These databases specialize in indexing and storing vector embeddings for fast semantic search and retrieval.

In [ ]:
```
import faiss
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_community.vectorstores import FAISS

index = faiss.IndexFlatIP(len(openai_embeddings[0]))
```

```
vector_store = FAISS(
    embedding_function=embeddings,
    index=index,
    docstore=InMemoryDocstore(),
    index_to_docstore_id={},
)

vector_store.add_documents(documents=final_chunks)
```

We can also save the vector store in persistent memory!

In [ ]:
```
vector_store.save_local(folder_path="./Memory",index_name="CWC_index")
```

Exercise: PDF Document (Employee Leave Policy)

Create and store a **FAISS** index. Use **IndexFlatIP**

Use **text-embedding-3-small** embeddings using **OpenAIEmbeddings** from langchain.

In [ ]:
```
storage_file_path="./Memory"
storage_index_name="PDF_index"
```

In [ ]:
```
#START YOUR CODE HERE


#END YOUR CODE HERE
```

▶ Click for Hint

▶ Click for Solution

Congratulations

With this, you have successfully completed the creation the **Vector Store**.

The four steps of loading, chunking, embedding and storing complete the **indexing pipeline**. Indexing pipeline is an **offline process**. The Vector Index needs to be created once and then updated at a periodic frequency.

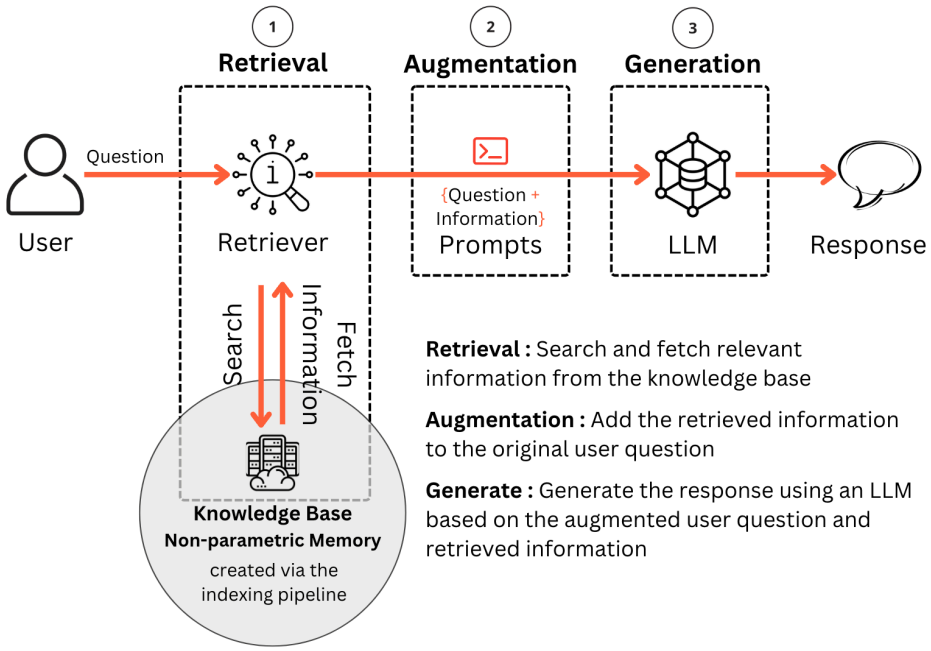
Now, we will move on to the **generation pipeline** and we will use this created index or knowledge base to handle real-time generations.

Generation Pipeline

The generation pipeline consists of three steps -

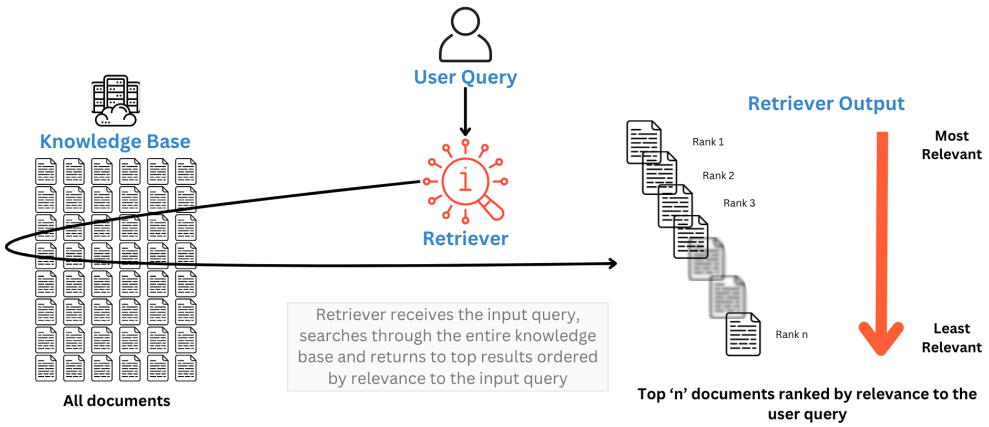
1. Retrieval
2. Augmentation

3. Generation



1. Retrieval

We will now retrieve a relevant passage from the knowledge base that is pertinent to our query - **"Who won the World Cup?"**



In [ ]:
```
# Load the FAISS vector store with safe deserialization
vector_store = FAISS.load_local(folder_path="./Memory",index_name="CWC_in

# Define a query
query = "Who won the world cup?"

# Perform similarity search
retrieved_docs = vector_store.similarity_search(query, k=2)  # Get top 2

# Display results
for i, doc in enumerate(retrieved_docs):
    print(textwrap.fill(f"\nRetrieved Chunk {i+1}:\n{doc.page_content}",w
    print("\n\n")
```

This is the most basic implementation of a retriever in the generation pipeline of a RAG-enabled system. This method of retrieval is enabled by embeddings. We used the text-embedding-3-small from OpenAI. FAISS calculated the similarity score based on these embeddings.

---

2. Augmentation

The information fetched by the retriever should also be sent to the LLM in form of a natural language prompt. This process of combining the user query and the retrieved information is called augmentation.

In [ ]:
```
retrieved_context=retrieved_docs[0].page_content + retrieved_docs[1].page

# Creating the prompt
augmented_prompt=f"""

Given the context below answer the question.

Question: {query}

Context : {retrieved_context}

Remember to answer only based on the context provided and not from any ot

If the question cannot be answered based on the provided context, say I d

"""

print(textwrap.fill(augmented_prompt,width=150))
```

---

3. Generation

Generation is the final step of this pipeline. While LLMs may be used in any of the previous steps in the pipeline, the generation step is completely reliant on the LLM. The most popular LLMs are the ones being developed by OpenAI, Anthropic, Meta, Google, Microsoft and Mistral amongst other developers.

We have built a simple retriever using FAISS and OpenAI embeddings and, we created a simple augmented prompt. Now we will use OpenAI's model, GPT-4o-mini, to generate the response.

In [ ]:
```
from langchain_openai import ChatOpenAI


# Set up LLM and embeddings
llm = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0,
    max_tokens=None,
```

```
    timeout=None,
    max_retries=2
)

messages=[("human",augmented_prompt)]

ai_msg = llm.invoke(messages)
```

In [ ]:
```
ai_msg.content
```

And there you have it. The response is rooted in the HTML document and based on the chunks retrieved from the vector database.

Exercise: PDF Document

Your exercise is to get an answer to the question - **How many paternity leaves can I avail?**

The FAISS Index **PDF_index** has already been created by you in the previous exercise. Now use **similarity_search** and **ChatOpenAI** library to get your answer.

Begin with the loading the index and retrieve the chunks. Retrieve top 2 chunks.

In [ ]:
```
#START YOUR CODE HERE


#END YOUR CODE HERE

# Display results
for i, doc in enumerate(retrieved_docs):
    print(textwrap.fill(f"\nRetrieved Chunk {i+1}:\n{doc.page_content}",w
    print("\n\n")
```

▶ Click for Solution

Now craft the augmented prompt!

In [ ]:
```
#START YOUR CODE HERE


#END YOUR CODE HERE

print(textwrap.fill(augmented_prompt,width=150))
```

▶ Click for Solution

Finally, make the call to the LLM. Use OpenAI's **gpt-4o-mini** model

In [ ]:
```
# START YOUR CODE HERE


# END YOUR CODE HERE

print(ai_msg.content)
```

▶ Click for Solution

Congratulations

With this, you have completed the construction of the core RAG pipeline!!! In the cell below you'll find all the above generation pipeline code in a single function.

In [ ]:
```
import re

# Function to clean text
def clean_text(text):
    # Replace non-breaking space with regular space
    text = text.replace('\xa0', ' ')

    # Remove any HTML tags (if any)
    text = re.sub(r'<[^>]+>', '', text)  # Removes HTML tags

    # Remove references in brackets (e.g., [7], [39])
    text = re.sub(r'\[.*?\]', '', text)  # Removes references inside squa

    # Remove extra spaces and newlines
    text = ' '.join(text.split())  # This will remove extra spaces and ne

    return text

def rag_function(query, db_path, index_name):
    embeddings=OpenAIEmbeddings(model="text-embedding-3-small")

    db=FAISS.load_local(folder_path=db_path, index_name=index_name, embed

    retrieved_docs = db.similarity_search(query, k=2)

    retrieved_context=[clean_text(retrieved_docs[0].page_content + retrie


    augmented_prompt=f"""

    Given the context below answer the question.

    Question: {query}

    Context : {retrieved_context}

    Remember to answer only based on the context provided and not from an

    If the question cannot be answered based on the provided context, say

    """

    llm = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0,
    max_tokens=None,
    timeout=None,
    max_retries=2
    )

    messages=[("human",augmented_prompt)]
```

```
    ai_msg = llm.invoke(messages)

    response=ai_msg.content

    return retrieved_context, response
```

In [ ]:
```
rag_function(query="who won on 5th of november?", db_path="./Memory", ind
```

Is the RAG system that we have created generating the responses on the expected lines? Is the LLM still hallucinating? Before trying to improve the performance of the system we need to be able to measure and benchmark it.

Evaluation

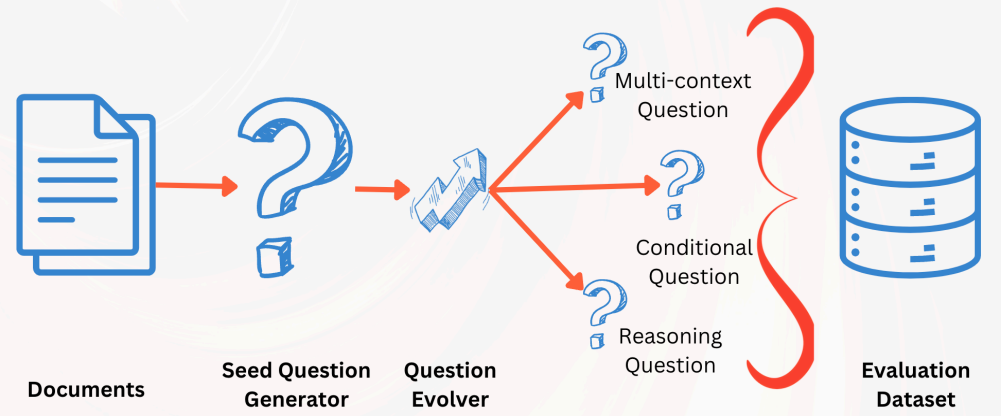
Ragas is a framework that helps you evaluate your Retrieval Augmented Generation (RAG) pipelines. It has been developed by the good folks at exploding gradients.

We will look at this evaluation in 2 parts.

1. Creation of synthetic test data for evaluation.
2. Calculation of evaluation metrics.

2.1 Creation of Synthetic Data

Synthetic Data Generation uses LLMs to generate diverse questions and answers from the documents in the knowledge base. LLMs can be prompted to create questions like simple questions, multi-context questions, conditional questions, reasoning questions etc. using the documents from the knowledge base as context.



In [ ]:
```
from ragas.llms import LangchainLLMWrapper
from ragas.embeddings import LangchainEmbeddingsWrapper
```

In [ ]:
```
generator_llm = LangchainLLMWrapper(ChatOpenAI(model="gpt-4o-mini"))
generator_embeddings = LangchainEmbeddingsWrapper(OpenAIEmbeddings(model=
```

In [ ]:
```
from ragas.testset import TestsetGenerator

generator = TestsetGenerator(llm=generator_llm, embedding_model=generator
dataset = generator.generate_with_langchain_docs(html_data_transformed, t
```

In [ ]:
```
dataset.to_pandas()
```

In [ ]:
```
sample_queries = dataset.to_pandas()['user_input'].to_list()
```

In [ ]:
```
expected_responses=dataset.to_pandas()['reference'].to_list()
```

In [ ]:
```
sample_queries
```

In [ ]:
```
dataset_to_eval=[]

for query, reference in zip(sample_queries,expected_responses):
    rag_call_response=rag_function(query=query, db_path="./Memory", index
    relevant_docs=rag_call_response[0]
    response=rag_call_response[1]
    dataset_to_eval.append(
        {
            "user_input":query,
            "retrieved_contexts":relevant_docs,
            "response":response,
            "reference":reference
        }
    )
```

In [ ]:
```
dataset_to_eval
```

In [ ]:
```
from ragas import EvaluationDataset
evaluation_dataset = EvaluationDataset.from_list(dataset_to_eval)
```

In [ ]:
```
from ragas import evaluate

evaluator_llm = LangchainLLMWrapper(ChatOpenAI(model="gpt-4o-mini"))

from ragas.metrics import LLMContextRecall, Faithfulness, FactualCorrectn

result = evaluate(dataset=evaluation_dataset,metrics=[LLMContextRecall(),
result
```

Exercise: PDF Document

In this final exercise, you need to generate a synthetic dataset using Ragas and then evaluate the responses of your RAG pipeline for the synthetically generated queries.

In [ ]:
```
### Complete the Code Below

generator_llm = ""
generator_embeddings = ""
generator = ""
dataset = ""
```

▶ Click for Solution

In [ ]:
```
sample_queries_pdf = dataset.to_pandas()['user_input'].to_list()
expected_responses_pdf=dataset.to_pandas()['reference'].to_list()
```

Now, create the dataset to eval

In [ ]:
```
dataset_to_eval_pdf=[]
# START YOUR CODE


# END YOUR CODE

evaluation_dataset_pdf = EvaluationDataset.from_list(dataset_to_eval_pdf)
```

▶ Click for Solution

Now, we calculate a few evaluation metrics - **ContextRecall, Faithfulness, FactualCorrectness, AnswerCorrectness & ResponseRelevancy**

In [ ]:
```
result = evaluate(dataset=evaluation_dataset_pdf,metrics=[LLMContextRecal
```

In [ ]:
```
result
```

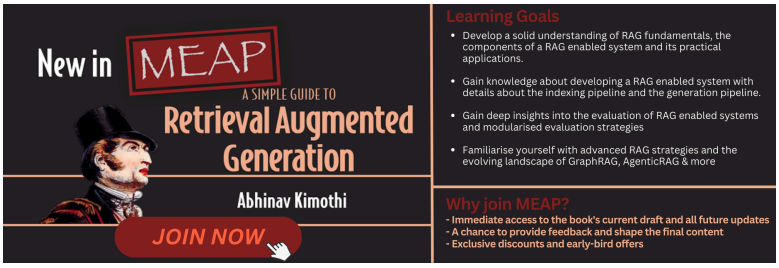
Congratulations!

For completing this introduction to RAG. I hope you had fun. For any queries, please get in touch!

Hi! I'm Abhinav! I am an entrepreneur and Vice President of Artificial Intelligence at Yarnit. I have spent over 15 years consulting and leadership roles in data science, machine learning and AI. My current focus is in the applied Generative AI domain focussing on solving enterprise needs through contextual intelligence. I'm passionate about AI advancements constantly exploring emerging technologies to push the boundaries and create positive impacts in the world. Let's build the future, together!

If you haven't already, get your copy of A Simple Guide to Retrieval Augmented Generation here

USE CODE `OUkimothi` for a 45% discount!



If you'd like to chat, I'd be very happy to connect

GITHUB LINKEDIN MEDIUM INSTAGRAM EMAIL

FOLLOW LINKTREE GUMROAD
