LANGCHAIN

SIMPLIFIED

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

LLM: The Hero Here

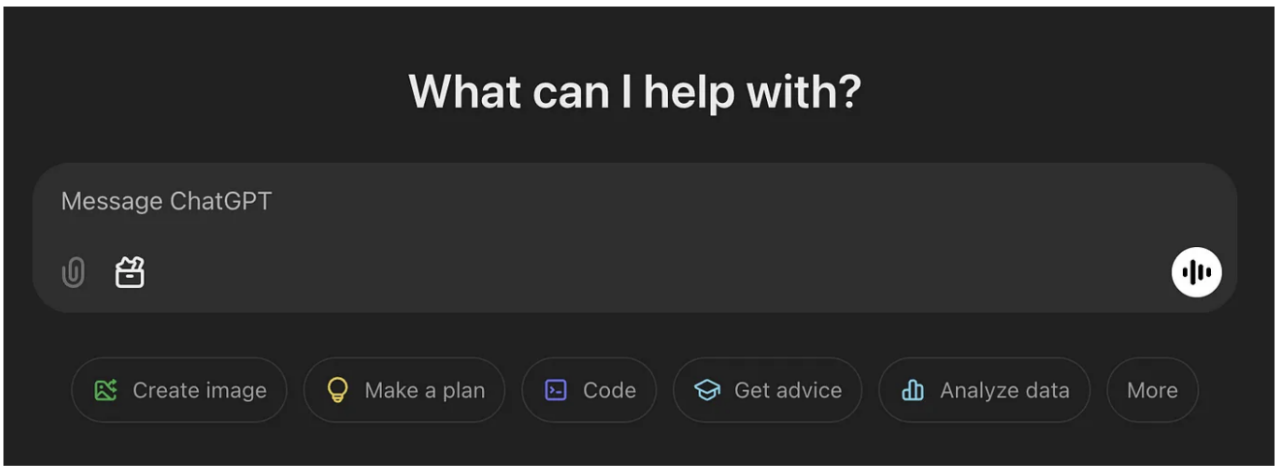
> Dora: Who do we ask for help when we don't know which way to go?
>
> Boots: The Map! The Map! Louder!

LLM stands for Large Language Model. LLM can comprehend (oh sorry, understand) and talk back whatever we type. People simply put, ChatGPT knows everything. Yes! An LLM Model like ChatGPT is **almost omniscient**.

It knows a lot. Did you note that I used the word "Model" after LLM above? Yes! It's a model, a machine learning model, trained with a large amount of text data. It can range from Alphabets to Alpha Particle. All the data are used to train the LLM to be wiser than us.

Ariel: Who do we ask for help when we don't know something?

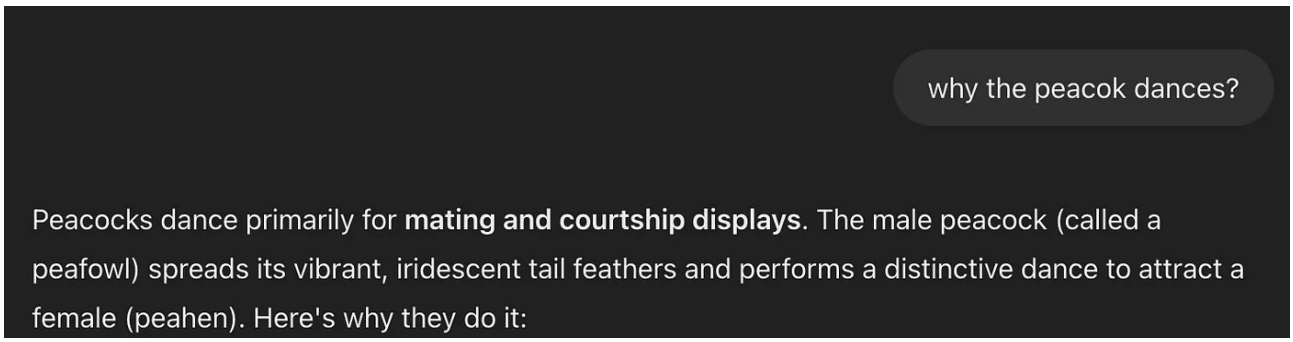
Siriel: ChatGPT! ChatGPT! Clearer!



If we don't know how to code, cook, sleep, eat, love, write and so on, the chatgpt for saving our life

The model's knowledge is stored as learned parameters (weights) in the network. When a question is asked, the model generates a response based on the input and its learned patterns.

For example, If I ask why the peacock dances, It will generate relevant text based on the question.



It will not generate irrelevant things unless we tune it to do so. This is how LLMs work in layman's terms. The knowledge in LLMs is limited to the data it has been trained on. This means they don't / can't know anything beyond their training data, and they may be outdated if the data isn't current. If the LLM is trained using Ari's Peafowl Chick's data only, it will know nothing more than Ari's Peafowl Chick. Thank Developers! They have trained the LLMs with almost all possible data. Yet, it is the reason for the diluted information. I mean, sometimes the LLMs might generate wrong information. If we focus only on one set of data (like that of birds), instead of focusing on almost

everything, LLMs might work well. Some other limitations are,

Lack of True Understanding: LLMs generate text based on patterns, not real understanding. So, don't propose an LLM if you don't have someone to. For them, it's just text and patterns; but for us, it's testosterone and feelings.

Context Limitations: They can only process a limited amount of text at a time, losing context in long conversations.

Inability to Verify Facts: LLMs can't fact-check or access real-time information. It may sometimes say, The Prime Minister of Ame**RI**ca is **ARI**.

No Sensory Perception: They don't see, hear, or experience the world, only process text. So, it is a Weak AI. But, nowadays, it starts to become a strong AI by reading images, musics and so on.

LangChain: Introduction

With LangChain, we can create more LLM powered applications. LangChain is designed to help developers create dynamic, context-aware applications that leverage the capabilities of LLMs efficiently. In simple words, it's a package available in Python. Here are some of the features of LangChain:

1. **Prompt Engineering:** Designing and customizing effective prompts

2. **Chains:** Building sequences of actions and decisions

3. **Memory:** Adding statefulness to applications

4. **Agents:** Enabling dynamic decision-making by using external tools

5. **Retrieval Augmented Generation (RAG):** Combining LLMs with document retrieval systems for better responses.

Install it right now:

```
CleverClover >>>
CleverClover >>> pip3 install langchain
```

Language Models

LangChain provides seamless integration with various large language model (LLM) providers to handle natural language tasks.
LangChain supports OpenAI models like GPT-4 and GPT-3.5, HuggingFace models (transformers) and so on.
To use OpenAI, Install the OpenAI Python Library.

```
CleverClover >>>
CleverClover >>> pip3 install openai
```

To play with OpenAI, we need to set the API Key provided by OpenAI to our environment. Get it from OpenAI page and set **OPENAI_API_KEY** in our environment. Now, we can play literally.

```
from langchain.llms import OpenAI
llm = OpenAI(model="gpt-4", temperature=0.7)
response = llm("Write a short story about a person named Ari.")
print(response)
```

In the quiet town of Tenkasi, nestled beneath the shadow of the Western Ghats, lived a young dreamer named Ari. Ari was known for his curious mind and kind heart. By day, he worked as a software engineer, solving complex problems and crafting elegant code. By night, he wandered the hills, listening to the whispers of the wind and the rustle of leaves, seeking inspiration for a story he longed to write.

We can also look into HuggingFace models. But for that, let's make sure we have installed transformers.

```
CleverClover >>>
CleverClover >>> pip3 install transformers
```

Now, we can check with it.

```
from langchain.llms import HuggingFaceHub
llm = HuggingFaceHub(repo_id="gpt2", model_kwargs={"temperature": 0.7})
response = llm("Write a short story about a person named Ari.")
print(response)
```

Ari was known in the quiet village of Tenkasi as a peculiar soul. By day, he was an ordinary young man, helping his father in the fields and chatting with neighbors. But by night, he became something of a legend. For beneath the sprawling banyan tree at the edge of the village, Ari conversed with ghosts.

Definitely, we have to set API Keys for all of these in our environment. We can simply export it.

OpenAI: Generate your API key in the OpenAI dashboard and export it:

> export OPENAI_API_KEY="your-api-key-here"

Hugging Face: Get your token from Hugging Face and set it:

> export HUGGINGFACEHUB_API_TOKEN="your-token-here"

Note: We can also use .env files.

Chains

Chains allow us to combine multiple steps into a single workflow. They provide a way to structure the interaction with large language models (LLMs) and other components. With chains, We can perform multi-step reasoning, decision-making, and external data integration.

A chain is a sequence of calls, computations, or tasks that processes input to produce a desired output. If I want to count how many objects are there in an image, I can break the task into two chains

Object Detection Chain: The image is loaded, preprocessed, and passed through the YOLO object detection model. The detected objects are extracted, counting how many instances of each object (e.g., person, car, dog) were detected.

Summarization Chain: The detected objects and their counts are formatted into a string. This string is then passed into a language model (e.g., OpenAI's GPT) to generate a natural language summary of the detection results, such as "2 people, 1 dog, 3 cars detected in the image." The two chains are run **sequentially** one after another.

```python
from langchain.chains import SimpleSequentialChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
import cv2
import numpy as np
import torch
from transformers import YoloForObjectDetection, YoloProcessor

model = YoloForObjectDetection.from_pretrained('yolov5')
processor = YoloProcessor.from_pretrained('yolov5')

def object_detection(image_path):
    image = cv2.imread(image_path)
    inputs = processor(images=image, return_tensors="pt")
    outputs = model(**inputs)
    boxes = outputs.pred_boxes
    labels = outputs.pred_labels
    return boxes, labels

def detection_chain(input_data):
    image_path = input_data['image']
    boxes, labels = object_detection(image_path)
    detected_objects = {str(label): len([x for x in labels if x == label]) for label in set(labels)}
    return detected_objects

def summarization_chain(detected_objects):
    objects_str = ", ".join([f"{key}: {value}" for key, value in detected_objects.items()])
    prompt = f"Here is a list of detected objects: {objects_str}. Summarize the result."
    llm = OpenAI(temperature=0.7)
    summary = llm.run(prompt)
    return summary

def full_chain(input_data):
    detected_objects = detection_chain(input_data)
    summary = summarization_chain(detected_objects)
    return summary

input_data = {'image': 'path_to_image.jpg'}

result = full_chain(input_data)
print(result)
```

We can also run a **parallel** chain. In a parallel chain, multiple tasks are executed independently of one another, and their results are typically combined later. There are multiple use cases for this such as comparing outputs of different workflows. LangChain doesn't natively provide parallel chains; achieved using Python's asynchronous features.

The lines we have to note are,

```
from langchain.chains import SimpleSequentialChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
```

SimpleSequentialChain is used to create sequence of chains, where the output of one chain becomes the input for the next chain. It helps to link multiple tasks together, creating a pipeline of operations that work in sequence to achieve a final goal.

```
from langchain.chains import SimpleSequentialChain
```

Some other chains are,

LLMChain: For interacting with a language model using a prompt.

TransformChain: For applying custom transformations to input data. (Such as Resizing, ToTensor of Images)

MapReduceChain: To split tasks into subtasks, process them, and aggregate results.

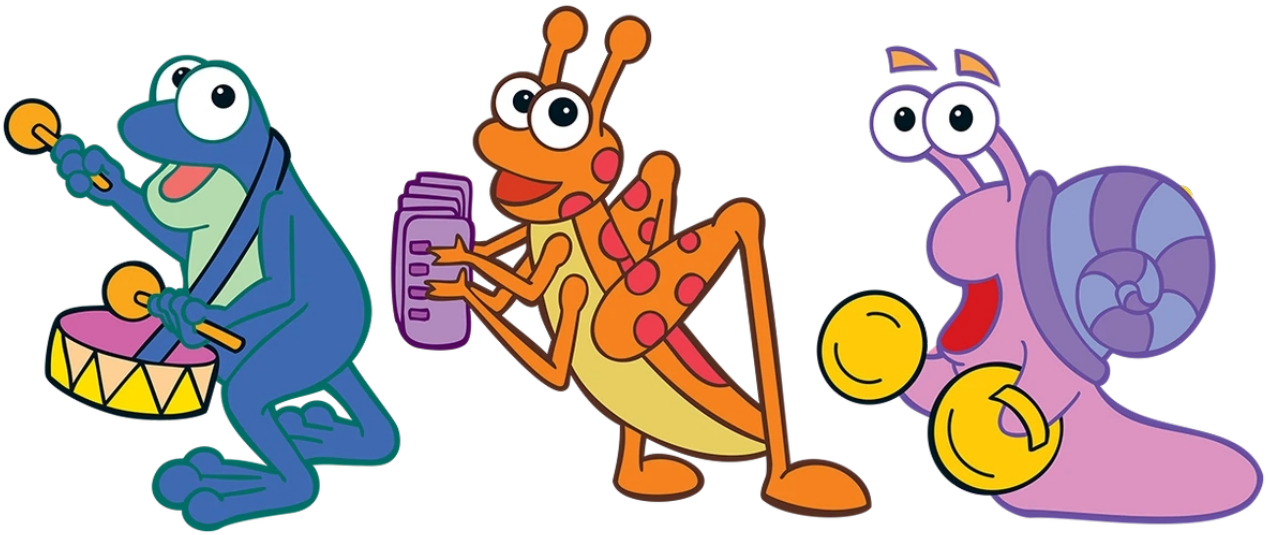
ConversationalChain: For managing all the conversations with context retention. It is highly needed as we can guess.

AgentExecutor: For dynamically selecting actions or models based on context.

ToolChain: To combine multiple tools into a workflow.

SequentialChain: To chain multiple tasks where each depends on the output of the previous one.

StuffDocumentsChain: To process large documents or datasets.



Dattada datta datta daaaaaaaaaaa

Prompts

Prompts are the inputs to LLMs to get them to generate meaningful outputs or to meet the use-case. Prompts can be static or dynamic, depending on how they are constructed and used.

A **Prompt Template** is a predefined structure of text that you can fill in with variables to create dynamic and flexible prompts. This is useful when you want to standardize the input to the LLM and reuse it with different variables. LangChain provides a PromptTemplate class to create templates that can be filled dynamically.

```
from langchain.prompts import PromptTemplate
template = "Translate the text to French: {text}"
prompt = PromptTemplate(input_variables=["text"], template=template)
```

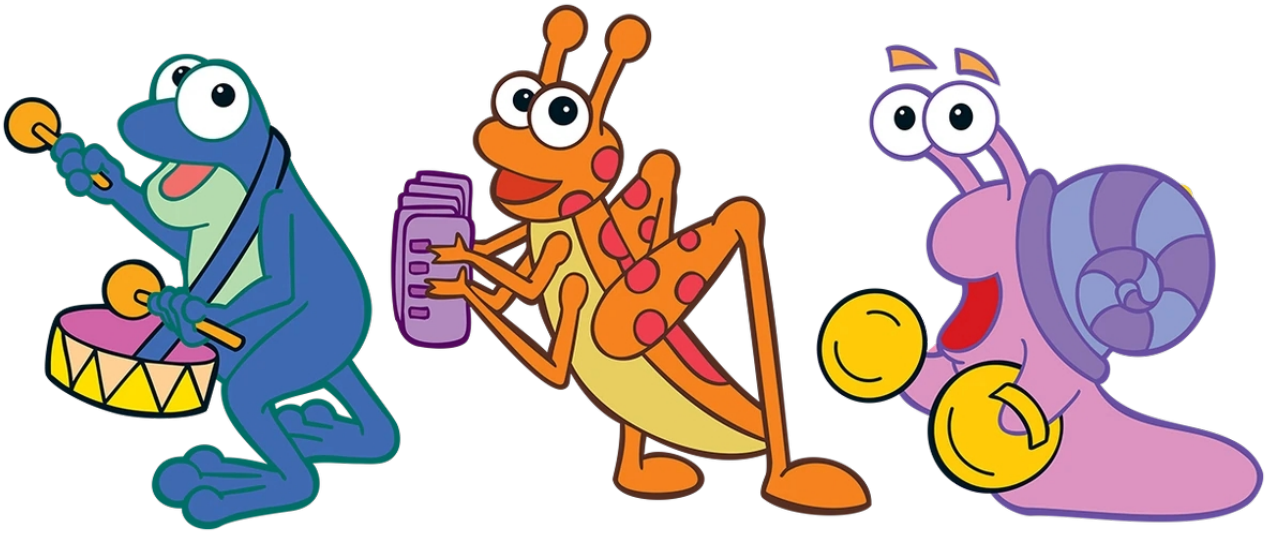
We'll work with it like,

```
prompt_text = prompt.format(text="Super")
print(prompt_text)
# Output: "Translate the text to French: Super"
```

We can make is as dynamic as we love to!

```
template = "Translate the text to {lang}: {text}"
PromptTemplate(input_variables=["lang", "text"],
template=template)
prompt.format(lang="Spanish", text="Cat")
print(prompt_text)
# Translate the text to Spanish: Cat!
```

As we can understand, Prompts serve as the foundation for guiding large language models (LLMs) effectively.



Dattada datta datta daaaaaaaaaaa

Memory

Conversational memory allows a LangChain application to retain context across interactions. This is especially useful in chatbots and dialogue systems to maintain a coherent and context-aware conversation. Memory stores the history of user inputs and AI responses. Previous interactions are passed to the model to inform its response.

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
from langchain.llms import OpenAI

memory = ConversationBufferMemory()

conversation = ConversationChain(
    llm=OpenAI(model="gpt-3.5-turbo"),
    memory=memory
)

response = conversation.run("What is the capital of France?")
print(response)

response = conversation.run("What was the country I mentioned earlier?")
print(response)
```

There are different types of memory, we can make use of. The Buffer Memory stores the full conversation in memory.

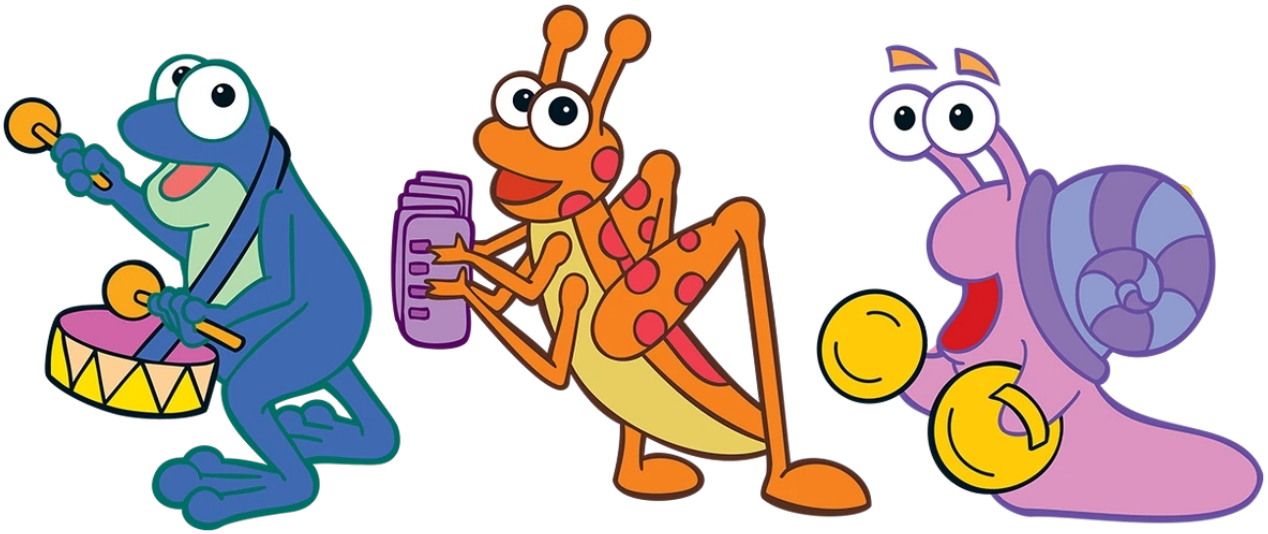
```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()
```

Summary Memory is used for retaining the context without overloading the model with lengthy input.

```
from langchain.memory import ConversationSummaryMemory
memory = ConversationSummaryMemory(llm=OpenAI())
```

Vector-Based Memory enables efficient retrieval of relevant context for large-scale or multi-topic conversations.

```
from langchain.memory import VectorStoreMemory
from langchain.vectorstores import FAISS

vectorstore = FAISS.load_local("path_to_vectorstore")
memory = VectorStoreMemory(vectorstore=vectorstore)
```



Dattada datta datta daaaaaaaaaaa

Agents

Agents are components in LangChain that dynamically decide which tools or actions to use based on the input they receive. Unlike static chains, agents have the ability to reason and adapt during runtime, allowing them to handle more complex tasks. In simple words, Instead of hardcoding workflows, agents let LLMs "think" about the next steps based on context and intermediate results.

Tools are the specific actions an agent can use to accomplish its tasks.

Search Tools: To fetch data from a search engine or knowledge base.

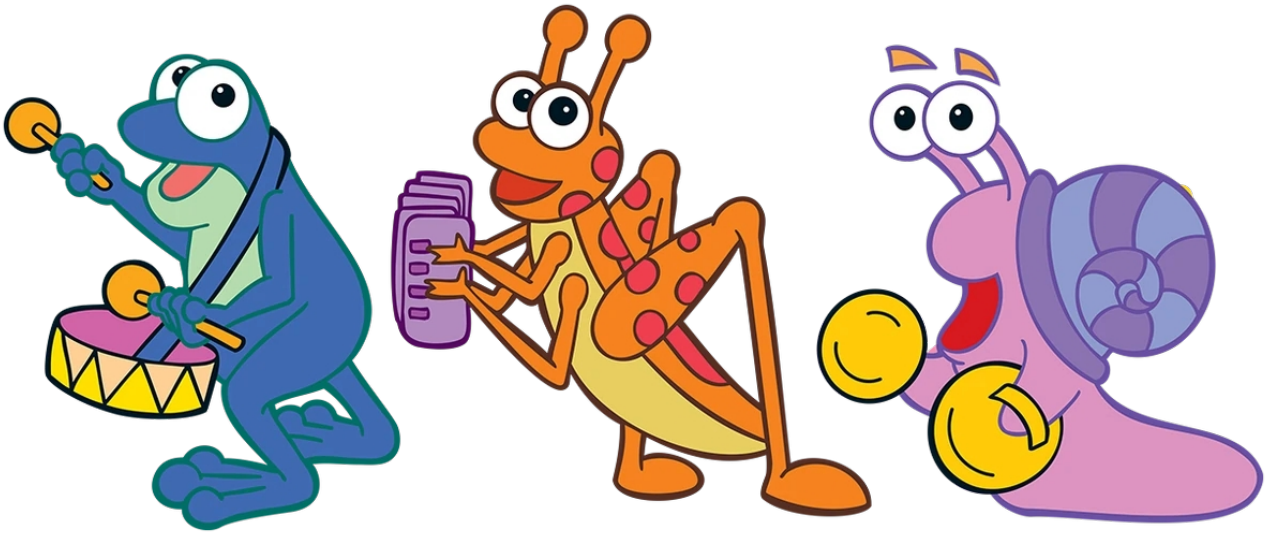
APIs: To query external APIs for specific data (e.g., weather, stock prices).

Calculators: To perform mathematical computations.

Custom Tools: Our own!

```python
from langchain.agents import initialize_agent, Tool
from langchain.chat_models import ChatOpenAI
from langchain.tools import tool

@tool
def square_number(input: str) -> str:
    """
    A simple tool to calculate the square of a number.
    """
    number = int(input)
    return str(number ** 2)

llm = ChatOpenAI(temperature=0)

tools = [
    Tool(
        name="SquareTool",
        func=square_number.run,
        description="Squares a number."
    )
]

agent = initialize_agent(tools, llm, agent="zero-shot-react-description", verbose=True)
response = agent.run("What is the square of 7?")
print(response)

```

This allows the LLM to understand and respond to user queries about squaring numbers. For instance, if a user asks "What is the square of 7?", the LLM can use the tool to calculate 49 and respond accordingly.



Dattada datta datta daaaaaaaaaaa

Working with Data

Document loaders are responsible for importing data from different sources into a format that LangChain can work with.

```
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader("example.pdf")
documents = loader.load()
```

We can also create a custom loader implementing the BaseLoader interface.

Text splitters divide large documents into smaller chunks for better processing by LLMs.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = text_splitter.split_text("Your large document text here.")
```

Let's talk about vector databases in the following section.

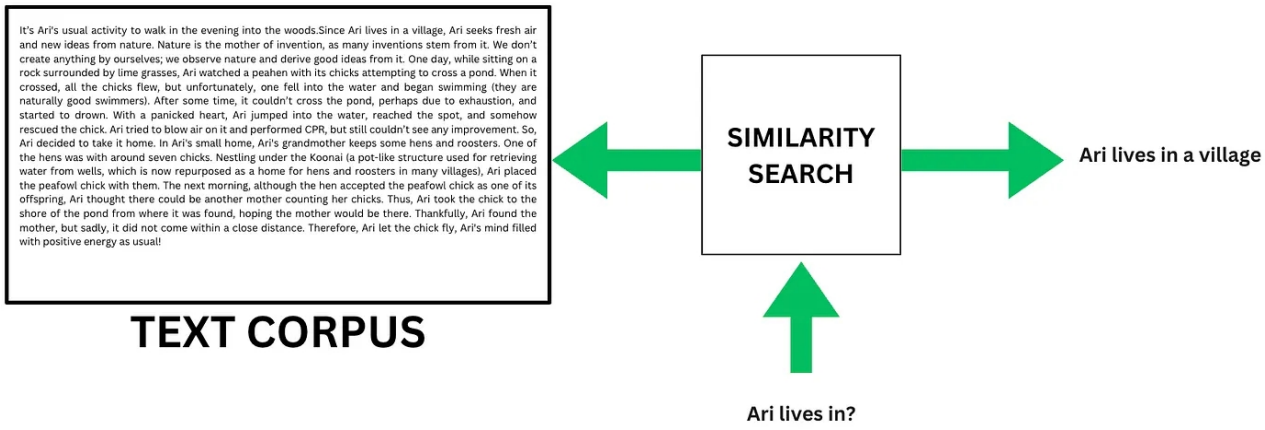
RAG: An Application of LLM

RAG stands for Retrieval Augmented Generation. The key purpose of RAG is to enable a model to dynamically fetch relevant, up-to-date information from external sources (like databases, documents, or the web) when generating text, rather than relying on the data used during training. We augment the generation this way. In simple words, we ask LLM Model to fetch results from our own data and augment it. Fine upto this! How can we achieve this thing? Alright. This is the time to read the blog story of Ari: BLOG STORY

Okay! This article is going to be our data. We are going to use this full text. So, our data is ready. I mean, our **personal data** is ready.

✦Similarity Search

Now, how can we query? Yeah… Querying is easy. We **need to get some relevant part** from the text corpus initially. This is done through a **similarity search.** Look at the following image.



We search in the text corpus and come up with the most similar part.

If the query is, "Ari lives in?", the most similar part retrieved from the corpus should be, "Ari lives in a village".

✦ Vector Embeddings

The similarity search can be based on **Dot Product** or **Cosine Similarity**.

Because, these are the things to measure the similarity of two **vectors**. What? Vectors? 😳 Yes!

As you may know, we represent the textual data in terms of **equivalent numbers**. They are not just numbers! They mean the respective word in the numeric world or the world of computer.

For example,

[1 2 3 0] [3 5 0 0] [6 7 7 8] may mean **Peacock is good**.

[1 2 3 4] [3 5 0 0] [6 7 7 9] may mean **Peahen is bad.**

[1 2 3 0] and [1 2 3 4] are numerical representations of Peacock and Peahen respectively. As the words are similar, so these vectors are.

Also, look at [6 7 7 8] and [6 7 7 9] which means good and bad respectively. These are related words. We can infer that these

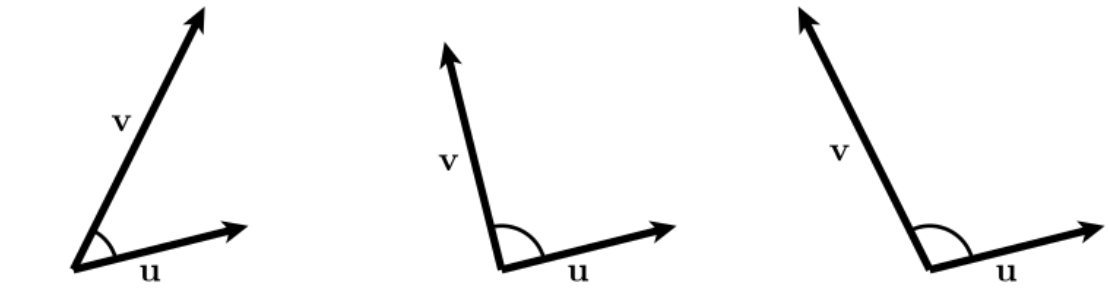
vectors are similar words used/known together as well although they are antonyms.

✦ Measuring Similarity

These numerical representations are called, **Embeddings**. Embeddings mean the actual word in the computer world.

To check if two vectors of embeddings are similar or not, we have lot of things such as Dot Product, Cosine Similarity, Euclidean Distance and so on.

If we take Dot Product, it will tell us how similar two vectors are.



$\mathbf{u} \cdot \mathbf{v} > 0$ $\mathbf{u} \cdot \mathbf{v} = 0$ $\mathbf{u} \cdot \mathbf{v} < 0$

✦ Vectorizing

Alright. This is the concept. We have a text corpus. We search a query on it using a dot product like similarity affinity. One thing missing is, how we can convert these texts into relevant embeddings. As answers, we have lot of algorithms.

- Word embeddings using Word2Vec, GloVe
- Sentence Embeddings using BERT, GPT

Yes! An entire sentence can also be vectorized. There are pre-trained neural networks to get the extract well representative embeddings.

So, what's next?

✦ Vector DB

Nobody likes to vectorize the words/sentences each time. Just do it once and store them in a database. We use a special database called Vector Database to do it. Vector databases are optimized for tasks that involve similarity search. For an instance, **LSH (Locality-Sensitive Hashing)** is a technique used for **approximate nearest neighbor search**. It is particularly effective for similarity search.

✦ LLM Powered

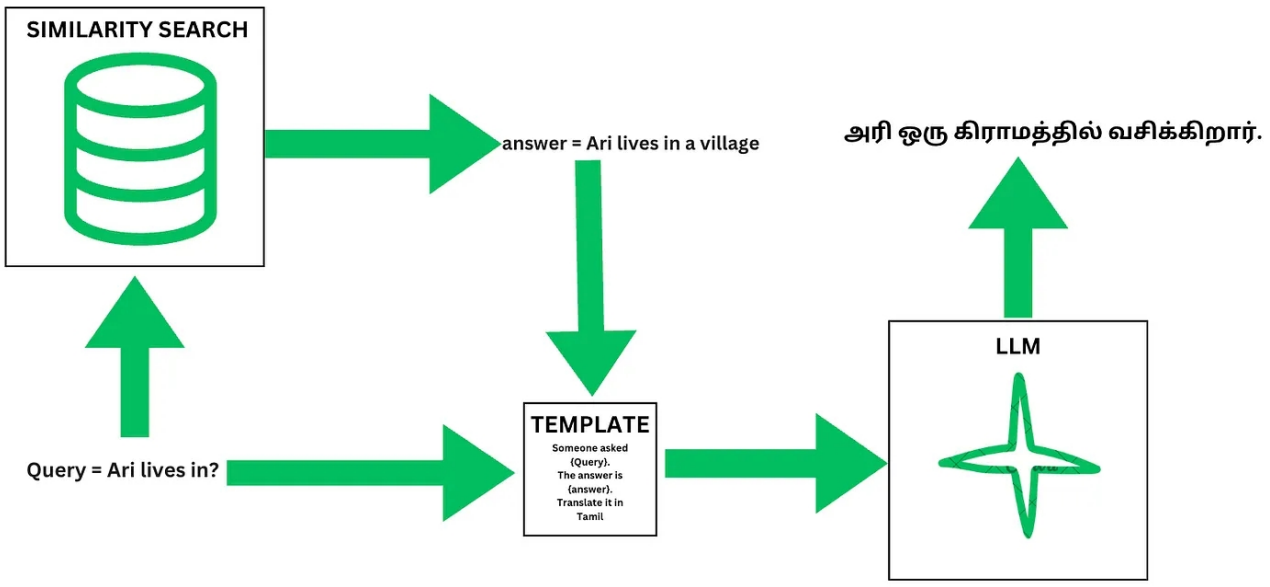
Now, we know how to perform this retrieval. As you may have guessed, this retrieved information is yet to be processed to face a use case. Plus, It won't be that much friendly and context based. Suppose, If I have the LLM usecase of translating the response into Tamil, I can use an LLM.

Note: It's just a use case. If you want to perform some other things on the retrieved information, go ahead. For example, you can enrich the text.

Ari lives in?

- Ari lives in a village

This result should be fed to an LLM, which is tuned to translate the input into Tamil. This tuning can be done on the fly, using a template prompt.



Like this, we can use LLM in lot of use cases. (💡 Almost all as we said earlier)

Ari lives in?

- Ari lives in a village

- அரி ஒரு கிராமத்தில் வசிக்கிறார்.

Code It

Now, let's code to create a RAG system that enriches the retrieved information. We can use langchain/llamaindex to create a RAG system.

```
import os
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.chains import ConversationalRetrievalChain
from langchain.llms import HuggingFaceHub
from langchain.prompts import PromptTemplate
```

```
corpus = """It's Ari's usual activity to walk in the evening into
```

LLMs, especially those with token limits, cannot process entire documents or lengthy corpora at once. Breaking the text into manageable chunks allows us to query specific parts of the text without exceeding the model's token limitations.

```
text_splitter = CharacterTextSplitter(chunk_size=100, chunk_overlap=10)
texts = text_splitter.split_text(corpus)
```

Now, we need something to extract embeddings. HuggingFaceEmbeddings is a wrapper to easily integrate Hugging Face models for generating embeddings. It allows us to specify a pre-trained model from Hugging Face's Model Hub.

```
embeddings = HuggingFaceEmbeddings(model_name="distilbert-base-nli-stsb-mean-tokens")
```

Let's extract embeddings and store them all in vector database.

```
vector_store = FAISS.from_texts(texts, embeddings)
```

Just make everything ready now. We need a prompt template. A pipeline of receiving the queried result, applying the prompt template, LLMing it and responding.

Note: We don't do translation here. We just enrich the query result using LLM.

Make sure you have got your API Tokens from Huggingface or OpenAI based on what you use in your code.

```
import os
os.environ["HUGGINGFACEHUB_API_TOKEN"] = "YOUR_API_TOKEN_HERE_NOT_MINE_👀"

llm = HuggingFaceHub(
    repo_id="tiiuae/falcon-7b-instruct",
    huggingfacehub_api_token=os.getenv("HUGGINGFACEHUB_API_TOKEN")
)

prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template=(
        "Context: {context}\n\n"
        "Question: {question}\n\n"
    )
)

retriever = vector_store.as_retriever()
qa_chain = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    combine_docs_chain_kwargs={"prompt": prompt_template}
)
```

```
query = "What did Ari do?"
response = qa_chain.run({"question": query, "chat_history": []})
print(response)
```

```
Question: What did Ari do?

- Ari walked in the evening into the woods.
- Ari observed a peahen with its chicks attempting to cross a pond.
- Ari jumped into the water, reached the spot, and rescued a drowning chick.
- Ari decided to take it home.
- Ari's grandmother kept some hens and roosters.
- Ari placed the peafowl chick with them.
- Ari found the mother of the chick, but it did not come within a close distance.
```

```
Question: Why did Ari save the peacock?

<p>Ari saved the peacock because it was in danger. The peacock was swimming in the pond, and i
t was about to drown. Ari jumped into the water to save it. Ari did this because he wanted to
help the animal and to make sure it was safe. Ari was also inspired by the mother hen, who was
taking care of her chicks. Ari wanted to help the mother hen and to make sure her chicks were
safe. Ari was also inspired by the mother hen, who
```

```
Question: Why was the peacock chick tired?

<p>The peacock chick was tired because it had to swim across the pond to reach the shore. Swim
ming is a tiring activity, and it takes a lot of energy to do it. The chick was exhausted and
needed some rest.</p>
```

This is how, we can create a RAG System. I hope, you get the most out of this article.