# GenAI – Basic to Advance

Index

1. Basic Concepts
   1. Completion application by LLM model for Q&A Application
   2. Conversation Application by Chat model of LLM
   3. Similarity Search on CSV application by Text Embedding’s
   4. Prompt Template – Optimize the user Query with Dynamic query (extension of Q&A)
2. Develop Q&A application(completion) by model LLM (without RAG)
3. Develop Conversation application by Chat Model
4. Develop Semantic search based application by Text Embedding – Without RAG Based by CSV
5. Prompt Template – Optimize the user Query with Dynamic query (extension of Q&A)

Use case is same as point 2.

6. Example Selector

7. Output Parsers

8. Memory

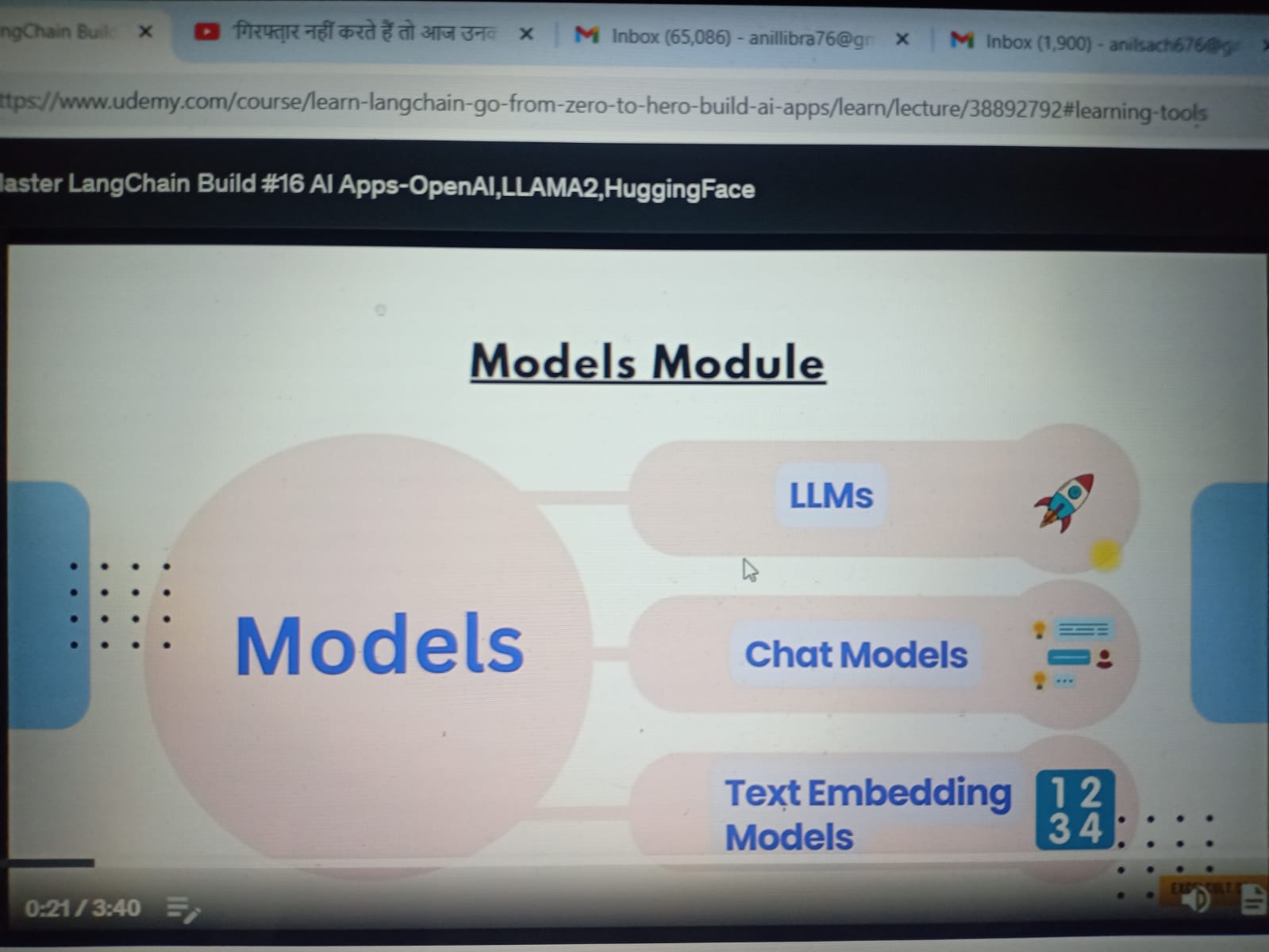
9. Data Loader

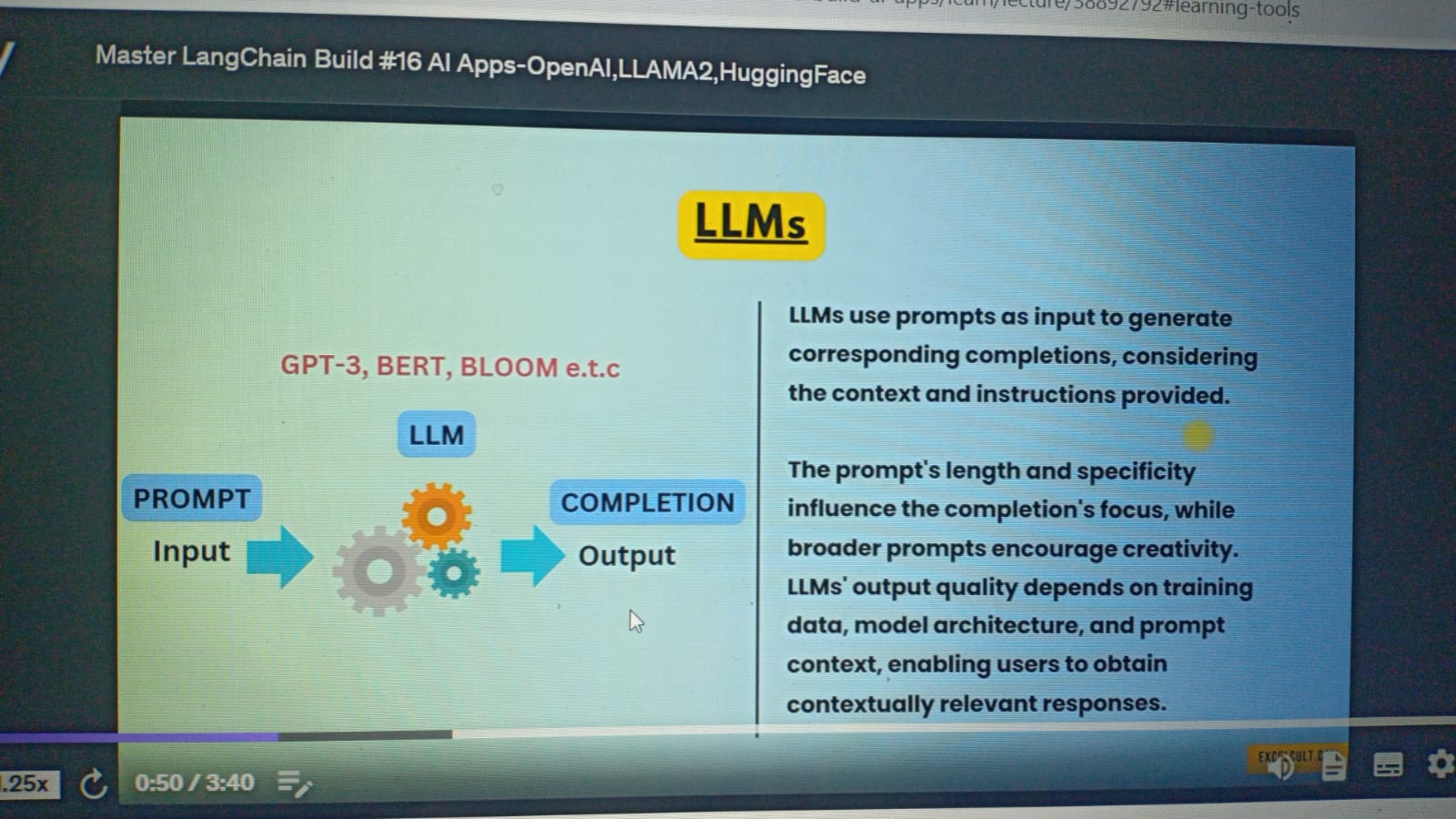
10 PDF-Q&A-Pinecone

11. Chains

12. Agents

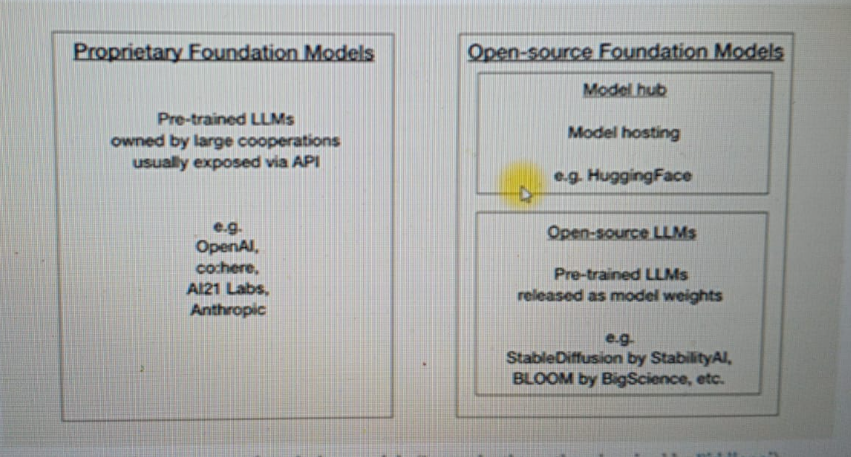
# Basic Concepts





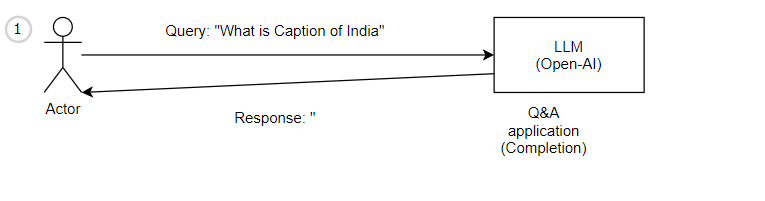
Open-Source: Gemini-Pro by Google

Paid: Open-AI



## **Q&A application by LLM - OpenAI without Custom DS (without RAG)**

Requirement: Provide your query to LLM (OpenAI) and get the answer



### Project\_Name: 1 Build Simple Q&A <https://github.com/anilSach676/GenAI/tree/main/1_llm_Q%26A/1%20Build%20Simple%20Q%26A%20Application>

### Question: “What is capital of India”

Response: India

Drawback:

* Its outdated data
* Not integrated with Custom Data (no RAG)

Fundamental:

from langchain\_openai import OpenAI

llm = OpenAI(model\_name="gpt-3.5-turbo-instruct",temperature=0)

answer=llm.invoke(question)

**Code**: (Frontend and Backend in same file)

#Hello! It seems like you want to import the Streamlit library in Python. Streamlit is a powerful open-source framework used for building web applications with interactive data visualizations and machine learning models. To import Streamlit, you'll need to ensure that you have it installed in your Python environment.

#Once you have Streamlit installed, you can import it into your Python script using the import statement,

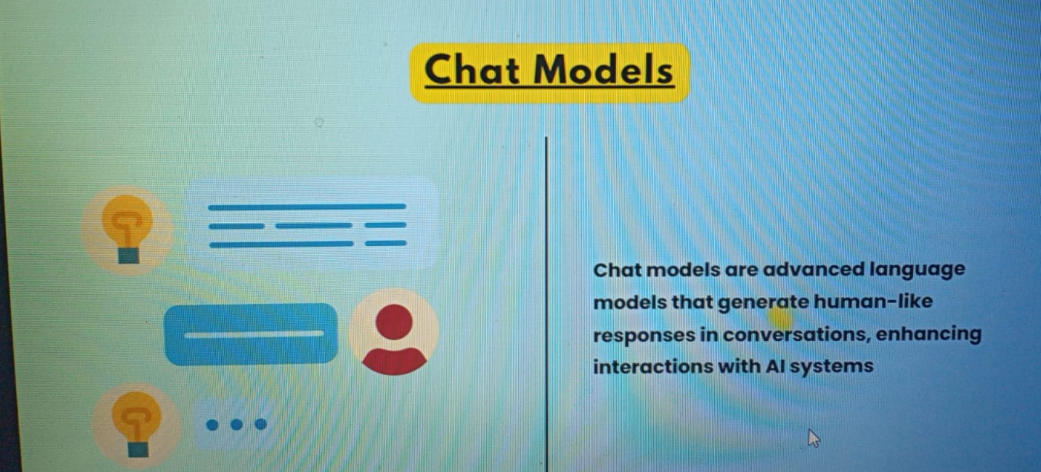
import streamlit as st

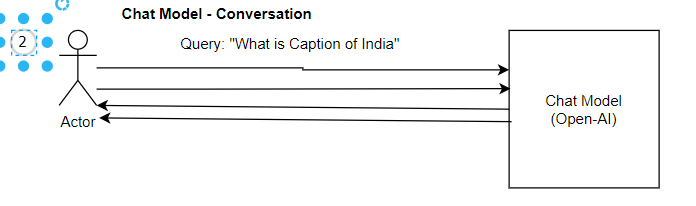
#As Langchain team has been working aggresively on improving the tool, we can see a lot of changes happening every weeek,

#As a part of it, the below import has been depreciated

#from langchain.llms import OpenAI

## Develop Conversation application by Chat Model of LLM (OpenAI)





### Project\_Name: “2 Build Simple Conversational https://github.com/anilSach676/GenAI/tree/main/1\_llm\_Q%26A/2%20Build%20Simple%20Conversational%20ApplicationQuestion: “What is capital of India”

Drawback:

* Its outdated data
* Not integrated with Custom Data (no RAG)

Fundamental:

from langchain\_openai import ChatOpenAI

from langchain.schema import ( AIMessage, HumanMessage, SystemMessage)

chat = ChatOpenAI(temperature=.7, model='gpt-3.5-turbo')

chat.invoke(

    [

        SystemMessage(content="You are a sarcastic AI assistant"),

        HumanMessage(content="Please answer in 30 words: How can I learn driving a car")

    ]

)

ourConversation=chat.invoke(

    [

    SystemMessage(content="You are a 3 years old girl who answers very cutely and in a funny way"),

    HumanMessage(content="How can I learn driving a car"),

    AIMessage(content="I can't drive yet! But I have a driver, my dad..."),

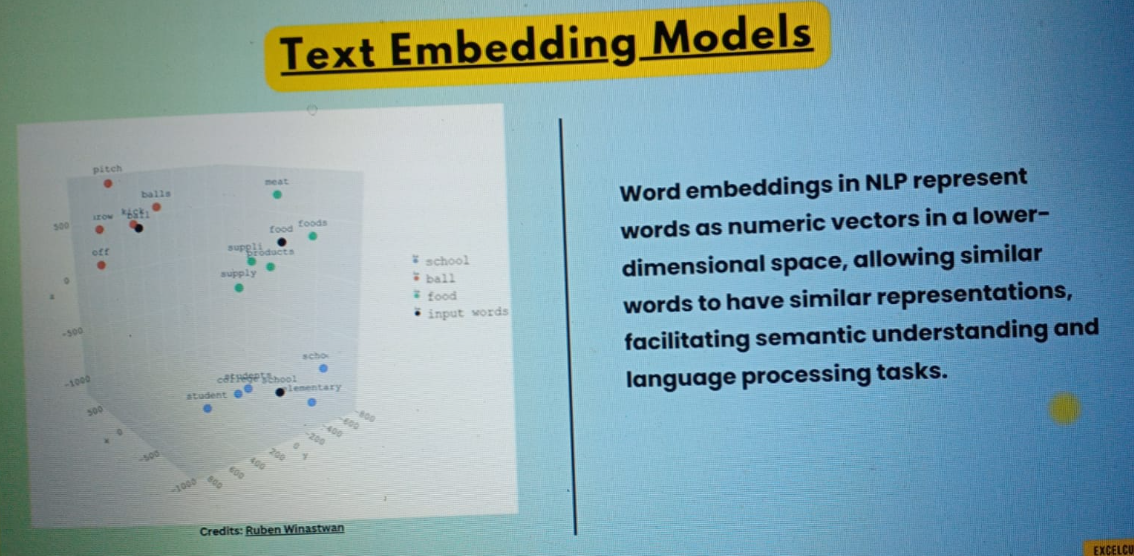
    HumanMessage(content="Can you teach me driving?")

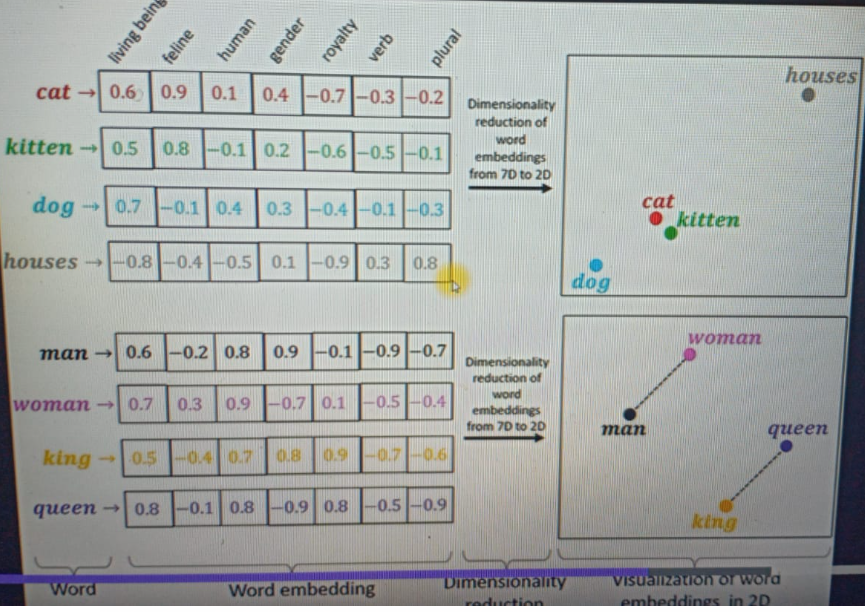
    ]

)

print(ourConversation.content)

## Develop Semantic search based application by Text Embedding – Without RAG Based by CSV



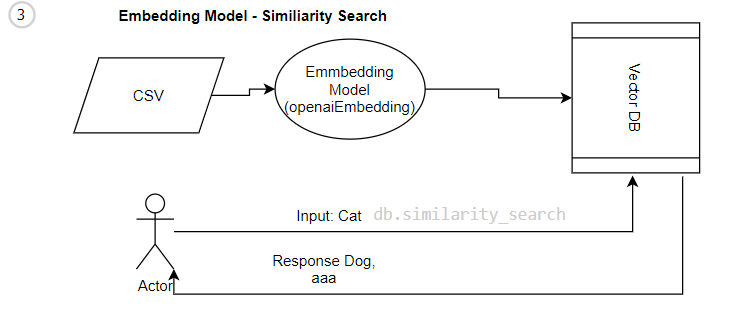


UseCAse:

Find Similar words in CSV as per user input

Sol:

1. Create CSV file and create it embeddings.
2. Store embedding in Vector db
3. Find Similarity search on vector db by user input
4. This is not RAG



from langchain\_openai import OpenAIEmbeddings

from langchain\_community.vectorstores import FAISS

embeddings = OpenAIEmbeddings()

from langchain.document\_loaders.csv\_loader import CSVLoader

loader = CSVLoader(file\_path='myData.csv', csv\_args={

    'delimiter': ',',

    'quotechar': '"',

    'fieldnames': ['Words']

})

data = loader.load()

db = FAISS.from\_documents(data, embeddings)

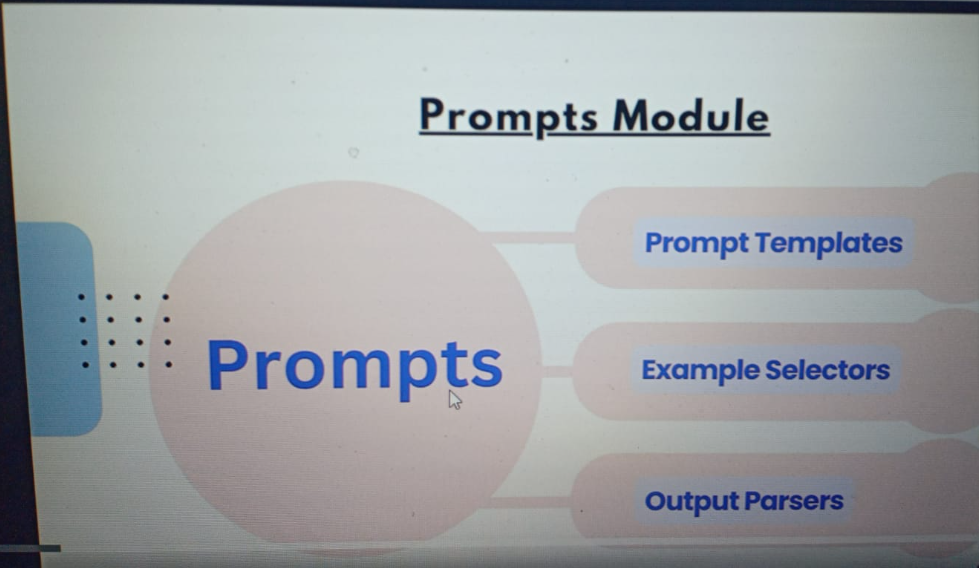
docs = db.similarity\_search(user\_input)

docs[0]

docs[2]

## This is not using RAG as Doc[0] and doc[1] is not sending to LLM for Augmented

## Prompt Template – Optimize the user Query with Dynamic query (extension of Q&A)



Prompt Template are forming a dynamic user query.

Ex: Without Prompt Template

Query = “Tell me the capital of India”

Llm(query)

With Prompt Template

Query = “Tell me the capital of {country} ”

Llm(query)

from langchain\_openai import OpenAI

from langchain import PromptTemplate

llm = OpenAI(model\_name="gpt-3.5-turbo-instruct")

template = """

{our\_text}

Can you create a post for tweet in {wordsCount} words for the above?

"""

prompt = PromptTemplate(

    input\_variables=["wordsCount","our\_text"],

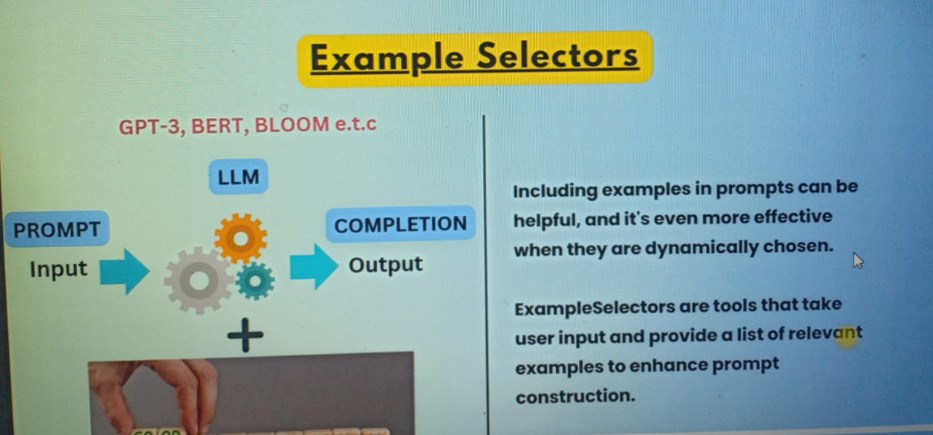
    template=template,

)

final\_prompt = prompt.format(wordsCount='3',our\_text="I love trips, and I have been to 6 countries. I plan to visit few more soon.")

print (llm.invoke(final\_prompt))

## Example Selector



**Few Shot Templates**: is a way to teach computers to make predications using only a small amount of information instead of needing lot of examples, computers can learn from just a few examples.

They can find the patterns in the examples and use those patterns to understand and recognize new things. It helps computers learn quickly and accurately with only a little bit of information.

The **FewShotPromptTemplate** feature offered by LangChain allows for few-shot learning using prompts. In the context of large language models (LLMs), the primary sources of knowledge are parametric knowledge (learned during model training) and source knowledge (provided within model input at inference time).

The FewShotPromptTemplate enables the inclusion of a few examples within prompts, which the model can read and use to apply to user input, enhancing the model's ability to handle specific tasks or scenarios.

from langchain.prompts import PromptTemplate

from langchain import FewShotPromptTemplate

Let's create a list of examples, that can be passed to the model later for our task

examples = [

    {

        "query": "What is a mobile?",

        "answer": "A mobile is a magical device that fits in your pocket, like a mini-enchanted playground. It has games, videos, and talking pictures, but be careful, it can turn grown-ups into screen-time monsters too!"

    }, {

        "query": "What are your dreams?",

        "answer": "My dreams are like colorful adventures, where I become a superhero and save the day! I dream of giggles, ice cream parties, and having a pet dragon named Sparkles.."

    }

]

Let's create a example template

example\_template = """

Question: {query}

Response: {answer}

"""

Let's create a prompt example from above created example template

example\_prompt = PromptTemplate(

    input\_variables=["query", "answer"],

    template=example\_template

)

The previous original prompt can be divided into a prefix and suffix. <br>The prefix consists of the instructions or context given to the model, while the suffix includes the user input and output indicator.

prefix = """You are a 5 year old girl, who is very funny,mischievous and sweet:

Here are some examples:

"""

suffix = """

Question: {userInput}

Response: """

 Let's create a few shot prompt template, by using the above details

few\_shot\_prompt\_template = FewShotPromptTemplate(

    examples=examples,

    example\_prompt=example\_prompt,

    prefix=prefix,

    suffix=suffix,

    input\_variables=["userInput"],

    example\_separator="\n\n"

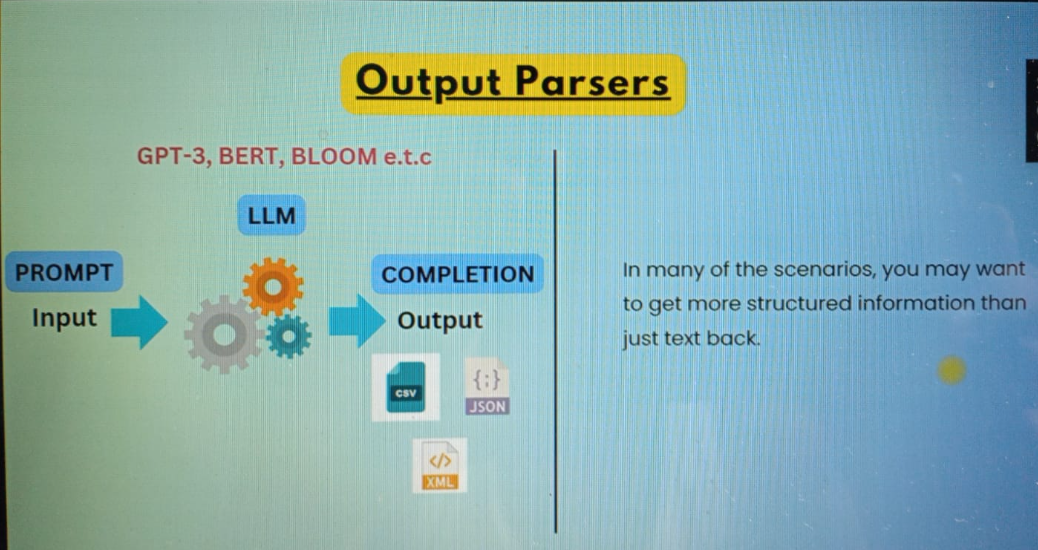
)

query = "What is a house?"

print(few\_shot\_prompt\_template.format(userInput=query))

print(llm.invoke(few\_shot\_prompt\_template.format(userInput=query)))

## Output Parsers



from langchain.output\_parsers import StructuredOutputParser, ResponseSchema

response\_schemas = [

    ResponseSchema(name="currency", description="answer to the user's question"),

    ResponseSchema(name="abbrevation", description="Whats the abbrebation of that currency")

]

output\_parser = StructuredOutputParser.from\_response\_schemas(response\_schemas)

format\_instructions = output\_parser.get\_format\_instructions()

prompt = PromptTemplate(

    template="answer the users question as best as possible.\n{format\_instructions}\n{query}",

    input\_variables=["query"],

    partial\_variables={"format\_instructions": format\_instructions}

)

prompt = prompt.format(query="what's the currency of America?")

output = llm.invoke(prompt)

```json

{

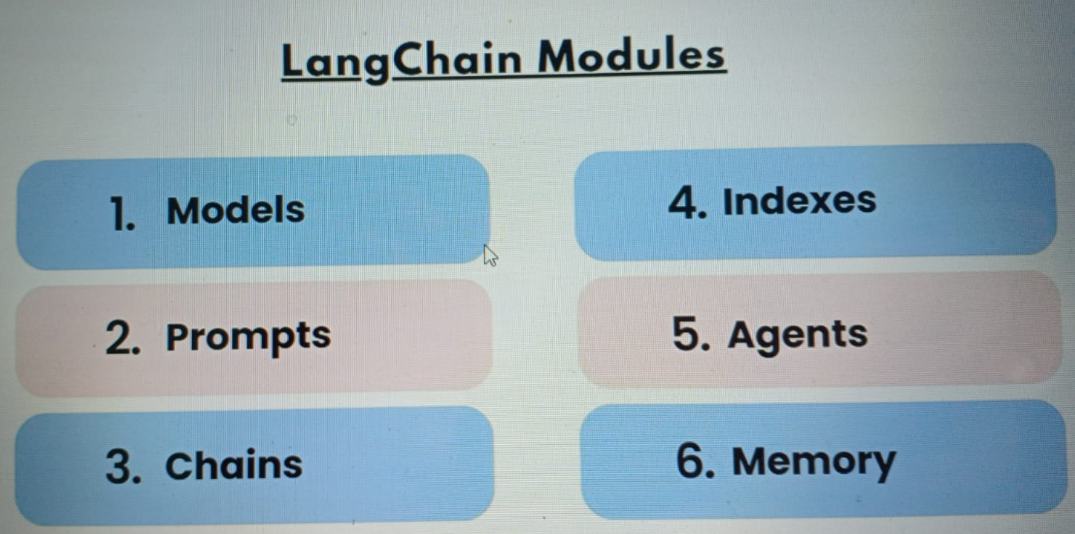
"currency": "United States dollar",

"abbreviation": "USD"

}

```

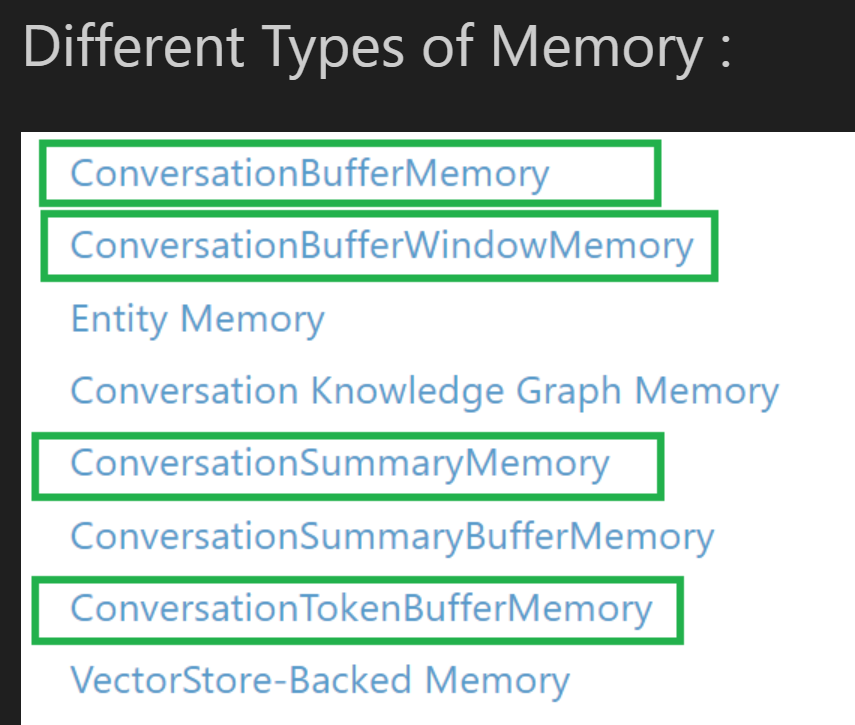
Memory



**Conversational memory**: is what enables chatbots to respond to our queries in a conversational manner. It allows for coherent conversations by considering past interactions, rather than treating each query as independent. Without conversational memory, chatbots would lack the ability to remember and build upon previous interactions.

**By default, chatbot agents are stateless**, meaning they process each incoming query as a separate input, without any knowledge of past interactions. They only focus on the current input and don't retain any information from previous interactions.

However, in applications like chatbots, it is crucial to remember past interactions. Conversational memory facilitates this by allowing the agent to recall and utilize information from previous conversations.



from langchain\_openai import OpenAI

from langchain.chains import ConversationChain

from langchain.chains.conversation.memory import (ConversationBufferMemory,

                                                  ConversationSummaryMemory,

                                                  ConversationBufferWindowMemory

                                                  )

**ConversationBufferMemory**

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    memory=ConversationBufferMemory()

)

conversation("My name is sharath!")

Prompt after formatting:

***The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.***

***Current conversation:***

***Human: Good morning AI!***

***AI: Good morning, human! It's a pleasure to interact with you today. How are you feeling?***

***Human: My name is sharath!***

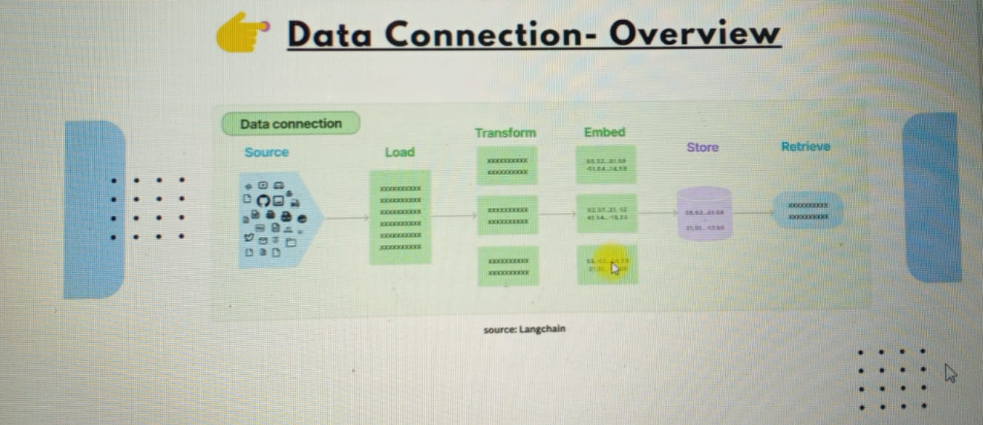
***AI:***

**> Finished chain.**

## Data Loader:

Document loaders

Load documents from many different sources



from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import CharacterTextSplitter

from langchain.embeddings import OpenAIEmbeddings

from langchain.vectorstores import Chroma

from langchain.chains import RetrievalQA

from langchain.embeddings.sentence\_transformer import SentenceTransformerEmbeddings

loader = TextLoader('Sample.txt')

documents = loader.load()

len(documents)

**## Document transformers**

**#### Split document and drop redundant documents**

text\_splitter = CharacterTextSplitter (chunk\_size=200,

chunk\_overlap=0)

texts= text\_splitter.split\_documents(documents)

**## Text embedding models**

**#### Take unstructured text and turn it into a list of floating point numbers**

embeddings=OpenAIEmbeddings()

**## Vector stores**

**#### Store and search over embedded data**

**#### Load Embeddings of Text into Chroma**

db = Chroma.from\_documents(texts, embeddings)

**## Retrievers**

**#### Query your data**

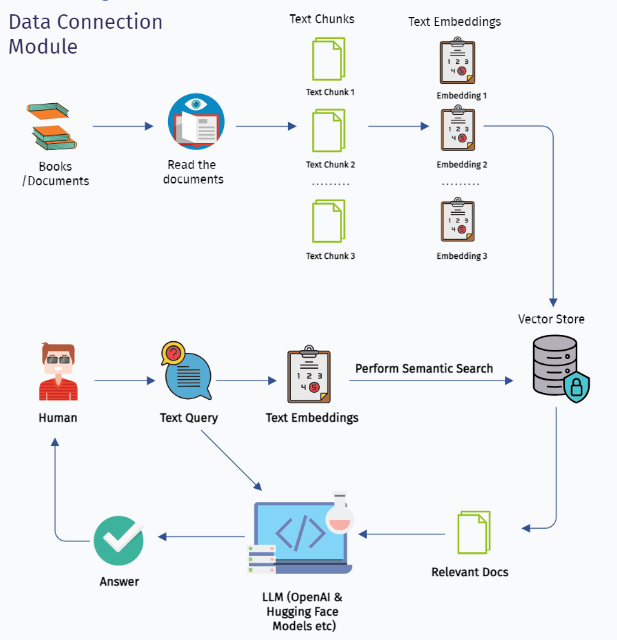
A vector store retriever is a retrieval system that utilizes a vector store to fetch documents. It acts as a simplified interface to the Vector Store class, adapting it to adhere to the Retriever interface. By leveraging the search functionalities provided by the vector store, such as similarity search and Maximal Marginal Relevance (MMR), the vector store retriever conducts queries on the text data contained within the vector store.

retriever = db.as\_retriever(search\_kwargs={"k": 2})

docs = retriever.get\_relevant\_documents("What is the capital of india?")

docs = retriever.get\_relevant\_documents("What is the currency india?")

## PDF-Q&A-Pinecone



Steps:

***Install & Import Dependencies***

***Load Documents***

***Transformer Documents***

***Generate Text Embeddings***

***Vector store - PINECONE***

***Retrieve Answers***

***Structure the Output***

from langchain.document\_loaders import PyPDFDirectoryLoader

import pinecone

from langchain.vectorstores import Pinecone #this below has been replaced by the below import

***2, Load Documents***

def load\_docs(directory):

  loader = PyPDFDirectoryLoader(directory)

  documents = loader.load()

  return documents

***Transformer Documents***

#This function will split the documents into chunks

def split\_docs(documents, chunk\_size=1000, chunk\_overlap=20):

  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=chunk\_size, chunk\_overlap=chunk\_overlap)

  docs = text\_splitter.split\_documents(documents)

  return docs

***Generate Text Embeddings***

embeddings = SentenceTransformerEmbeddings(model\_name="all-MiniLM-L6-v2")

***Vector store - PINECONE***

from pinecone import Pinecone as PineconeClient #Importing the Pinecone class from the pinecone package

from langchain\_community.vectorstores import Pinecone

# Set your Pinecone API key

# Recent changes by langchain team, expects ""PINECONE\_API\_KEY" environment variable for Pinecone usage! So we are creating it here

# we are setting the environment variable "PINECONE\_API\_KEY" to the value and in the next step retrieving it :)

os.environ["PINECONE\_API\_KEY"] = "f4321sa787-edyte-4092-adn1-er56gv8b056"

PINECONE\_API\_KEY=os.getenv("‘PINECONE\_API\_KEY’")

# Initialize the Pinecone client

PineconeClient(api\_key=PINECONE\_API\_KEY, environment="gcp-starter")

index\_name="mcqcreator"

# Pass your chunks and embeddings

index = Pinecone.from\_documents(docs, embeddings, index\_name=index\_name)

***Retrieve Answers***

#This function will help us in fetching the top relevent documents from our vector store - Pinecone

def get\_similiar\_docs(query, k=2):

    similar\_docs = index.similarity\_search(query, k=k)

    return similar\_docs

from langchain.chains.question\_answering import load\_qa\_chain

llm = OpenAI()

chain = load\_qa\_chain(llm, chain\_type="stuff")

#This function will help us get the answer to the question that we raise

def get\_answer(query):

  relevant\_docs = get\_similiar\_docs(query)

  print(relevant\_docs)

  response = chain.run(input\_documents=relevant\_docs, question=query)

  return response

our\_query = "How is India's economy?"

answer = get\_answer(our\_query)

print(answer)

## Chains



<https://docs.kanaries.net/articles/langchain-chains-what-is-langchain>

Chains refer to sequences of calls - whether to an LLM, a tool, or a data preprocessing step

* Chains are the vital core of LangChain. These logical connections between one or more LLMs are the backbone of LangChain's functionality. Chains can range from simple to complex, contingent on the necessities and the LLMs involved. Let's delve deeper into both types:
* **Basic Chains**: A basic chain is the simplest form of a chain that can be crafted. It involves a single LLM receiving an input prompt and using that prompt for text generation.

**Ex: SimpleSequentialChain**

*from langchain.prompts import PromptTemplate*

*from langchain.llms import HuggingFace*

*from langchain.chains import LLMChain*

*prompt = PromptTemplate(*

*input\_variables=["city"],*

*template="Describe a perfect day in {city}?",*

*)*

*llm = HuggingFace(*

*model\_name="gpt-neo-2.7B",*

*temperature=0.9)*

*llmchain = LLMChain(llm=llm, prompt=prompt)*

*llmchain.run("Paris")*

**Advanced Chains**

Advanced chains, also known as utility chains, are made up of multiple LLMs to address a particular task. A suitable example is the SummarizeAndTranslateChain, which is aimed at tasks like summarization and translation.

Ex:

* **load\_summarize\_chain**: For document summarization
* **LLMRequestsChain:** Preparing the question & inputs to the http request like google.com

For instance, LangChain features a specific utility chain named TopicModellingChain, which reads articles and generates a list of relevant topics.

**Agents**:

Agents in LangChain present an innovative way to **dynamically call LLMs based on user input**. They not only have access to an LLM but also a suite of tools (like Google Search, Python REPL, math calculator, weather APIs, etc.) that can interact with the outside world.

*from langchain.agents import initialize\_agent, AgentType, load\_tools*

*from langchain.llms import OpenAI*

*llm = OpenAI(temperature=0)*

***tools = load\_tools(["pal-math"], llm=llm)***

*agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)*

*response* ***= agent.run****("If my age is half of my dad's age and he is going to be 60 next year, what is my current age?")*

*print(response) # Outputs: "My current age is 29.5 years old."*

In this case, the agent leverages the pal-math tool and an OpenAI LLM to solve a math problem embedded in a natural language prompt. It demonstrates a practical case where the agent brings additional value by understanding the prompt, choosing the correct tool to solve the task, and eventually returning a meaningful response. When we wants output of one chain is input of other chain.

Simple Chain:

It allows to make code more customized for extending the model.

llm = OpenAI()

prompt = PromptTemplate(

    input\_variables=["place"],

    template="Best places to visit in {place}?",

)

chain = LLMChain(llm=llm, prompt=prompt)

print(chain.invoke("India"))

SimpleSequentialChain: Sequential Chains involves making a series of consecutive calls to the language model.

This approach proves especially valuable when there is a need to utilize the output generated from one call as the input for another call.

from langchain.chains import SimpleSequentialChain

from langchain.llms import HuggingFaceEndpoint

template = """You have to suggest 5 best places to visit in {place}?

YOUR RESPONSE:

"""

prompt\_template = PromptTemplate(

    input\_variables=["place"],

    template=template)

HF\_llm = HuggingFaceEndpoint(repo\_id="mistralai/Mistral-7B-Instruct-v0.2")

place\_chain = LLMChain(llm=llm, prompt=prompt\_template)

template = """Given a list a places, please estimate the expenses to visit all of them in local currency and also the days needed

{expenses}

YOUR RESPONSE:

"""

prompt\_template = PromptTemplate(

    input\_variables=["expenses"],

    template=template)

llm = OpenAI()

expenses\_chain = LLMChain(llm=llm, prompt=prompt\_template)

final\_chain = SimpleSequentialChain(chains=[place\_chain, expenses\_chain], verbose=True)

review = final\_chain.invoke("India")