Language Models

(LLMs)

GPT T5 BERT

Chat Models

Dialogue Taskhs

GPT-4

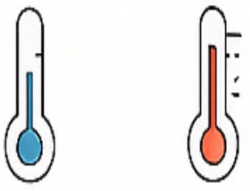
Embedding Models

Sentence Transformers

E5 MiniLM

Temperature

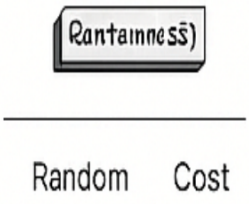
In randomness and creativity in resse



Low temperature

High temperature

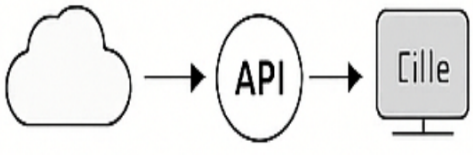
VS



Open vs. Closed Source

GPT-3 GPT-4	GPT-lteo GPT-J
Advantages	Advantages
Acces	Customization Costreflective
Disadvantages	Disadvantages
Hardware requirements	Hardware requirement

How to Use Models



Closed Models

Utilize

Open Source

GPT-3 GPT-J

gcased

Via API

Alftentages

Cost-effectivnne

Advarsages

Access-Cost

Disadvantages

Disadvartages

Hardware requl-rements

Embedding Models Usage

Closed Source

OpenAI Embeddings

Open Source

Ounn locally

Hugging Face

Texcepikcmodels Usage

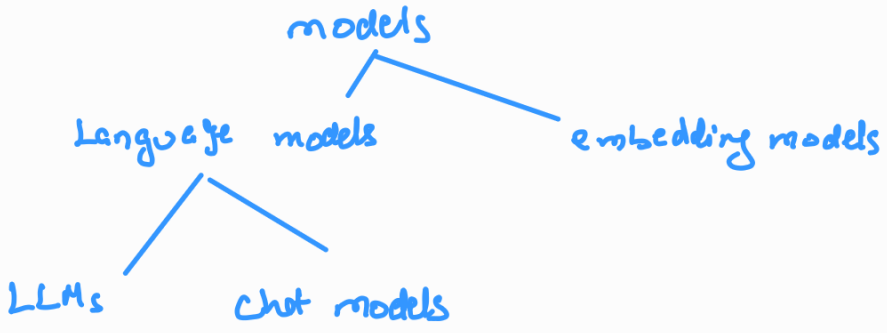
- Text similarity
- Other

Models:

→ The model component in langchain is a crucial part of the framework, designed to facilitate interactions with various Language models and embedding models

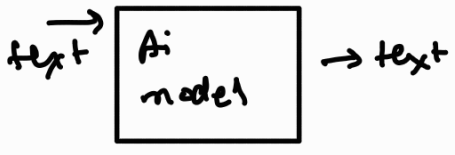
→ It abstracts the complexity of working directly with different LLMs, chat models and embedding models, providing a uniform interface to communicate with them.

→ This makes it easier to build applications that rely on Ai-generated text, text embeddings for similarity search and retrieval-augmented generation (RAG)



Language models:

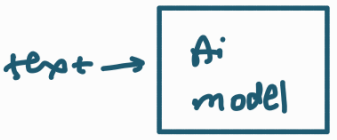
input → text
output → text



eg (Chatbot)

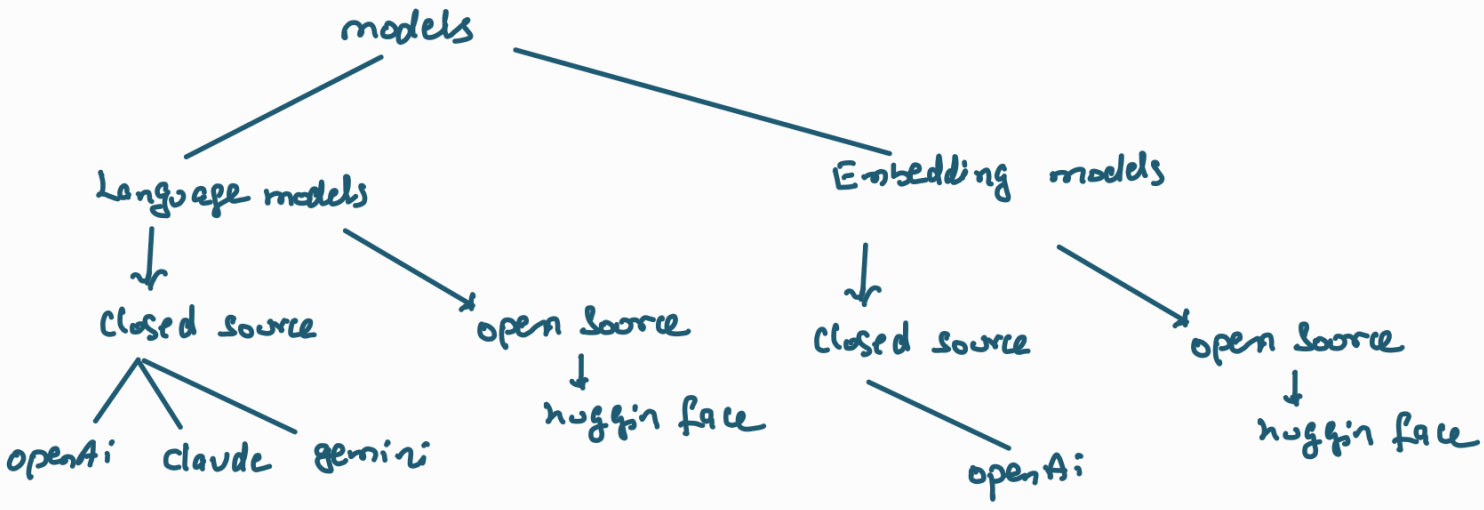
Embedding models:

input → text
output → embeddings / vectors



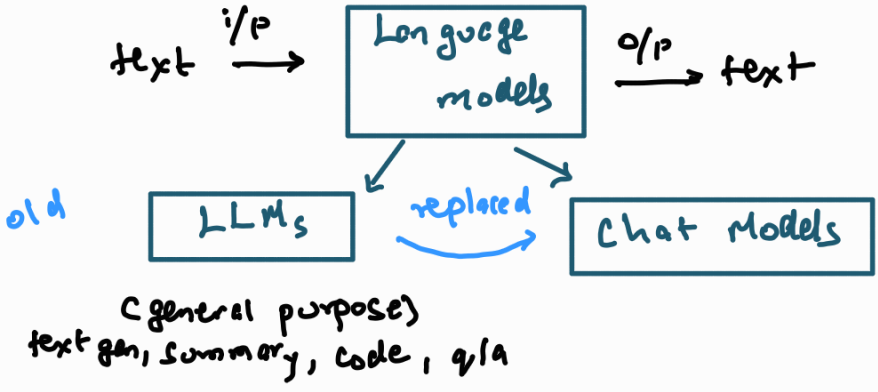
eg → semantic search (rag apps)

Plan of action



Language models :

Language models are Ai systems designed to process, generate and understand natural language text.



LLMs :

→ General purpose models that are used for raw-text generation.

→ takes a string (or plain text as input) and returns a string (or plain text as output)

→ These are traditionally older models and are not used much now.

Chat Models :

→ Language models that are specialized for conversational tasks.

→ They take a sequence of messages as inputs and return chat message as output (as opposed to using plain text)

feature	LLMs (Base Models)	Chat Models (Instruction tuned)
Purpose	free form text generation	optimised for multi-turn conversation
Training data	General text Corpora (books, articles)	fine tuned on chat datasets (dialogues, user-assistant conversation)
Memory & context	no built in memory	supports structured conversation history
Role awareness	no understanding of user and assistant	understands "system", "user" and "assistant" roles
Example models	GPT-3, llama-2-7B, mistral 7B	GPT-4, GPT 3.5, llama-2-chat mistral-instruct
use cases	Text generation, Summarization Creative writing, code generation	Conversational AI, Chatbots, virtual assistants, customer support, AI tutors

Set up:

i) create a folder

ii) open in VS code

iii) create a new venv

python -m new venv

iv) activate

venv\Scripts\activate

v) create requirements.txt

vi) pip install -r requirements.txt

vii) verify langchain Installation.

Requirements.txt

```
#langchain
    langchain

    langchain-core

#open Ai
    langchain-openai

    openai

#Anthropic
    langchain-anthropic

#google
    langchain-google-genai

    langchain-generative ai

#huggingface
    langchain-hugging face

    transformers

    huggingface-hub

#ML utils

    numpy

    scikit-learn

# env management
    python-dotenv
```

test

```
import langchain

print (langchain.__version__)
```

LLMs

Chat Models

Embedding Models.

1) LLMs

↳ create account

↳ buy 5 $ credits

↳ create an API key.

Demo [do not use this ↓] cuz old, use chat models instead.

```
from langchain-openai import OpenAI
from dotenv import load-dotenv

load_dotenv()

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

result = llm.invoke("what is the capital of India")
```

OpenAI → inherits from BaseOpenAI

⇒ the capital of india is New Delhi

Conclusion

→ input is string

→ output is string

ChatModel OpenAI → Models

```
from langchain-openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()

model = ChatOpenAI(model="gpt-4", temperature=0)

result = model.invoke("what is the capital of India")
print(result)
print(result.content)
```

ChatOpenAI → inherits from BaseChatModel

print(result) → returns an object of object

result.content → contains the output

→ the capital of india is New Delhi

input: text

output: object {object}

max_completion_tokens = 10 (words)

used to limit the number of token to be received.

#temperature (0 - 2) → defines how random the output will be compared to previous output

→ temperature is a parameter that controls the randomness of a language model's output.

→ it affects how creative or deterministic the responses are

- lower values (0.0 - 0.3) → more deterministic & predictable
- higher values (0.7 - 1.5) → more random, creative, diverse

use cases:

	recommended temp
factual answers (math, code)	0.0 - 0.3
Balanced (Q/A, explanations)	0.5 - 0.7
Creative writing, Story telling	0.9 - 1.2
Maximum randomness (wild ideas, brainstorming)	1.5+

Anthropic - Claude Api use.

```
from langchain_anthropic import ChatAnthropic
from dotenv import load_dotenv

load_dotenv()

model = ChatOpenAI(model='claude-3-5-sonnet-20241022')

result = model.invoke("what is the capital of India")
print(result)  ──────────→ (returns an object of object)
print(result.content)  ──→ contains the output
```

→ the capital of india is NewDelhi

Google Gemini: Api

```
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv

load_dotenv()

model = ChatGoogleGenerativeAI(model="gemini-2.5-pro", temperature=1)

result = model.invoke("what is the capital of India")
print(result)  ──────────→ (returns an object of object)
print(result.content)  ──→ contains the output
```

→ the capital of india is NewDelhi

Open source Models

→ OS models language models are freely available AI models that can be downloaded, modified and deployed without any restrictions from a central provider.

→ Unlike closed source models such as OpenAi GPT4 or google gemini, open source models allow full control and customization

feature	open-source model	closed source model
cost	free	paid Api usage
control	modify, fine-tune deploy anywhere	locked to provider's infrastructure
Data privacy	run locally (no data sent to external servers)	sends queries to servers
Customization	can fine tune on specific datasets	very limited to no access to fine tuning of dataset
Deployment	can be deployed on-premise servers or cloud	must use vendor API

Some famous Open Source Models

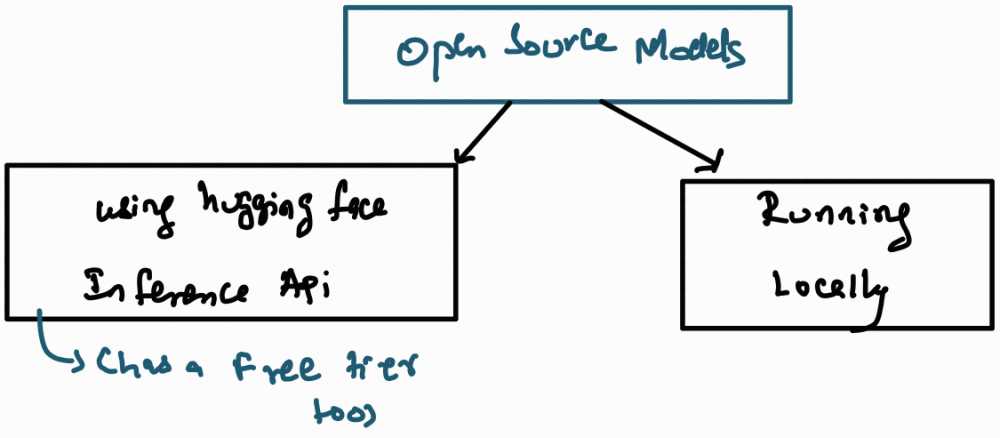
1) Llama
2) bloom
3) mistral

Where to find Open Source AI models?

i) huggingface → largest repository of open-source LLMs

ii) Open router → consists of both open source and closed source LLMs

how to use?



Disadvantages

High hardware requirements : running a large model requires GPUs

Set up complexity : requires installation of dependencies like pytorch, cuda, transformers

Lack of RLHF : most open source models don't have fine tuning with (reinforcement learning from human feed back)
human feedback, making them weaker in instruction-following

Limited Multimodal abilities : open model's do not support images, videos audios like GPT-4 V

Using Open Source Models using

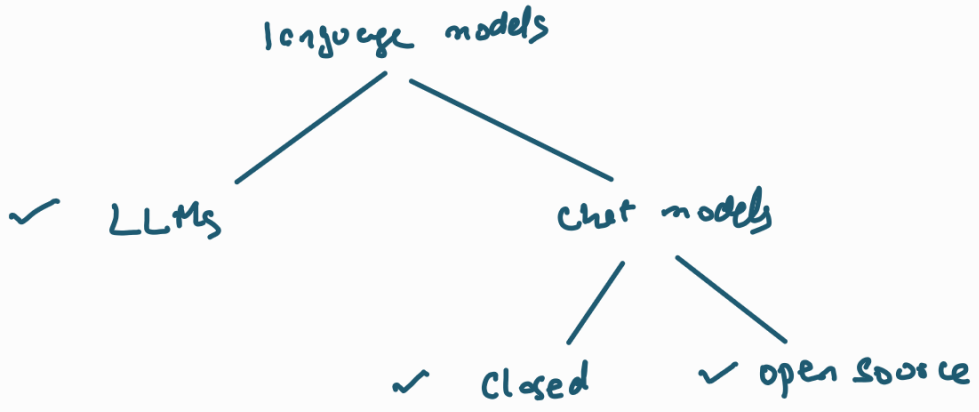
Hugging face Inference Api

```
from langchain_huggingface import ChatHuggingface, HuggingFaceEndPoint

from dotenv import load_dotenv

load_dotenv()


llm = HuggingfaceEnd (
        repo_id = " TinyLlama /TinyLlama -1.1B - chat v 1.0",
        task = "text- generation"
        )

model = ChatHuggingface(llm = llm)

result = model.invoke(" what is the capital of India" )

print( result.content)
```

Using local LLM

```
from langchain_huggingface import ChatHuggingface, HuggingFacePipeline

from dotenv import load_dotenv

load_dotenv()


llm = HuggingfaceEnd (
        repo_id = " TinyLlama /TinyLlama -1.1B - chat v 1.0",
        task = "text- generation", pipeline - kwargs= (temperature = 1,
        max_new_token = 100 )
```

```
model = ChatHuggingFace(llm=llm)

result = model.invoke("what is the capital of India")

print(result.content)
```



Embedding Models :

→ used to convert words (texts) to vectors, in a way

→ these vectors are usually continuous values and capture semantic relationship between the data.

→ the goal is to represent the data in a way that preserves its important characteristics while simplifying it into a more manageable form.

common types of embeddings..

i) word Embeddings (eg word2vec, glove

ii) Sentence or document Embeddings : eg (sentence-BERT)

iii) Image Embeddings : eg (ResNet, VGG)

iv) Multimodal Embeddings : eg (CLIP)

Note: The larger the dimension, the more semantic understanding (contextual meaning)

Sample code:

```
from langchain_openai import OpenAIEmbeddings
from dotenv import load_dotenv
load_dotenv()

embeddings = OpenAIEmbeddings(model="text-embedding-3-large",
                              dimensions=4)

result = embeddings.embed_query("Delhi is the capital of India")
                                ↳ can pass a single sentence.

print(str(result))
```

-> [-0.294366, 0.501512, -0.0136084, 0.834177]

ii) Embedding_docs

```
from langchain_openai import OpenAIEmbeddings
from dotenv import load_dotenv
load_dotenv()

embeddings = OpenAIEmbeddings(model="text-embedding-3-large",
                              dimensions=4)
docs = ["delhi is the capital of India",
        "kolkata is the capital of WestBengal",
        "Chennai is the capital of Tamil Nadu"]

result = embeddings.embed_documents(docs)
```

```
print(str(result))
```

```
>>  [
    [-0.299366 , 0.501512 , -0.0136 0584, 0.2134177],
    [-0.299322 , 0.501514 , -0.0136 0585, 0.2134178],
    [-0.299347 , 0.501516 , -0.0136 0586, 0.2134179]
    ]
```

Embedding using Open Source Model.

(Sentence Transformer)

```
from langchain_huggingface import HuggingFaceEmbeddings

embedding = HuggingFaceEmbeddings(model="sentence-transformers/
                                  all-MiniLM-L6-v2")

text = "Delhi is the capital of India"

docs = ["delhi is the capital of India",
        "kolkata is the capital of WestBengal",
        "chennai is the capital of Tamil Nadu"]

vector = embedding.embed_query(text)
vector1 = embeddings.embed_documents(docs)
print(vector)
print(vector1)
```

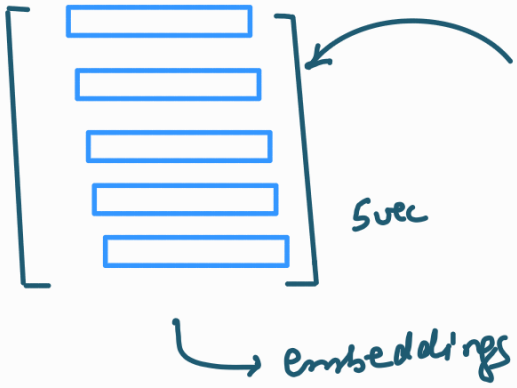
```
>>
```

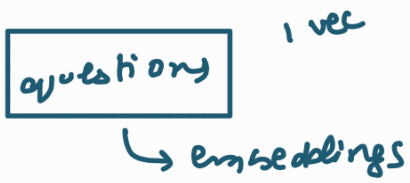
O/P
384 dimension vectors.

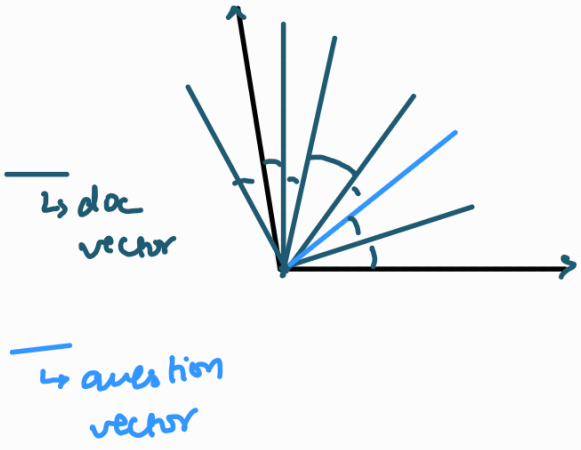
Document Similarity Application.

5 documents

dimensions for eg: 10







cosine similarity. [a part of scikit learn metric]

[the one with highest similarity score is the answer]

[the same concept is used in RAG applications]

→ lets say we have 4 documents and 1 question related to the document, and we need to find which which document it is.

→ how do we do that??

1) generate Embeddings for documents and store it
2) generate Embeddings for the query and store it
3) find the cosine_similarity between the documents and query.

4) The vector with highest cosine similarity

5) map it with the document to find the document.

Code

```
from sklearn.metrics.pairwise import cosine_similarity
from langchain_openai import OpenAIEmbeddings
from dotenv import load_dotenv

load_dotenv()

embedding = OpenAIEmbeddings(model="text-embedding-3-large",
                             dimensions=300)

documents: [
        "Sachin is the God of Cricket",
        "Virat is an Aggressive batsmen",
        "Jasprit is an Amazing bowler",
        "Rohit has lots of double centuries"]

query = "tell me about virat"

documentEmbeddings = embedding.embed_documents(documents)

queryEmbeddings = embedding.embed_query(query)

scores = cosine_similarity([queryEmbeddings], documentEmbeddings)[0]

index, score = sorted(list(enumerate(scores)), key=lambda x: x[1])[-1]

print(document[index])

print(f"Similarity score is {score}")
```

>> Virat is an Aggressive batsman
Similarity score is 0.62885