LangChain Models

I. Types of Models in LangChain

LangChain's model component supports two primary types of AI models:

1. **Language Models:**

- **Function:** "Language Models are models where you give some text as input, such as 'What is the capital of India,' and this model understands and processes that text, returning text in response."
- **Application:** Primarily used for building applications like chatbots, as they involve text-based input and output.

1. **Embedding Models:**

- **Function:** "Embedding Models take a text as input... and this time, instead of returning text, they return a series of numbers." These numbers are called **embeddings** and are "nothing but vectors or a set of numbers, which kind of represent this text or capture the contextual meaning of this text."
- **Application:** Crucial for **semantic search** and building **RAG (Retrieval Augmented Generation) based applications**. An example cited is chatting with documents, which requires semantic search powered by embedding models.

II. Deep Dive into Language Models: LLMs vs. Chat Models

Language Models are two types:

1. **LLMs (Large Language Models):**

- **Purpose:** "General-purpose models which means you can use them for any kind of NLP application. You can use them for text generation, text summarization, code generation, and question answering."
- **Input/Output:** They take a "string in plain text as input and return a string in plain text as output."
- **Current Status in LangChain:** LLMs are "kind of old now, and in LangChain, you will slowly notice that their support is ending." For new projects, it is "not recommended" to work with LLMs.

2. **Chat Models:**

- **Purpose:** "Language models that are specialized for conversational tasks. They take a sequence of messages as inputs and return chat messages as output." They are "great for building a chatbot."
- **Key Differences from LLMs (as per table comparison):Purpose:** LLMs are for "Free-form text generation" (Q&A, summarization, translation, code generation), while Chat Models are for "Multi-turn conversations between the user and the AI."
- **Training:** LLMs are trained on "general type of training data" (books, articles, Wikipedia text), whereas Chat Models undergo "fine-tuning on chat datasets" (dialogues, conversational logs) after initial general training.
- **Memory:** LLMs have "no memory concept," meaning they don't remember previous interactions. Chat Models "support conversation history."
- **Role Awareness:** Chat Models allow assigning roles to the AI (e.g., "You are a very knowledgeable doctor..."). LLMs do not offer this feature. Chat models understand the roles of "self, user, and AI."
- **Typical Use Cases:** LLMs are suited for text generation, summarization, translation, code generation. Chat Models are preferred for "conversational AI chatbots, virtual assistants, customer support chatbots, or AI tutors."

- **Current Status in LangChain:** "Today, if you talk about it, LangChain recommends that you should work more with Chat Models, and the support for LLMs is slowly ending."

III. Practical Implementation with Language Models (Coding Demo)

A. Setup and Environment:

- Create a dedicated folder (e.g., LangChain Models).
- Open in VS Code and set up a **virtual environment** (python -m venv venv).
- Activate the virtual environment (venv/Scripts/activate).
- Install required libraries from a requirements.txt file (e.g., langchain-openai, python-dotenv, langchain-anthropic, langchain-google-genai, langchain-huggingface, scikit-learn, numpy, sentence-transformers).
- Organize code into LLMs, Chat Models, and Embedding Models folders.

B. Closed-Source Language Models:

1. **OpenAI GPT Models (LLM Interface):**

- Requires an **OpenAI API key**. This is a **paid service**; free credits are no longer offered. A minimum of $5 top-up is suggested for practice.
- API keys should be stored securely in a .env file (e.g., OPENAI_API_KEY="your_key").
- **Code Structure:**from langchain_openai import OpenAI
- from dotenv import load_dotenv
- load_dotenv()
- llm = OpenAI(model_name="gpt-3.5-turbo-instruct") # Older LLM model
- result = llm.invoke("What is the capital of India")
- print(result)
- **Note:** The invoke method is a "very important function in LangChain" and is present across core components (models, chains, prompts). LLMs take string input and return string output.

2. **OpenAI Chat Models (Chat Model Interface):**

- Uses ChatOpenAI instead of OpenAI.
- **Code Structure:**from langchain_openai import ChatOpenAI
- from dotenv import load_dotenv
- load_dotenv()
- model = ChatOpenAI(model_name="gpt-4") # Newer Chat Model
- result = model.invoke("What is the capital of India")
- print(result.content) # Output is a structured object, extract 'content'
- **Output Difference:** Chat Models return a more structured output containing metadata (completion tokens, prompt tokens, total tokens) in addition to the actual answer, which is found under the content key.

3. **Configurable Parameters (for Chat Models):**

- **Temperature:** Controls the **randomness/creativity** of the model's output (values 0 to 2).
- Lower values (0-0.3): More deterministic, predictable (for factual answers, code generation).

- Higher values (0.9-1.5+): More random, creative, diverse (for story writing, poems, brainstorming).
- **Max Completion Tokens:** Restricts the **length of the output** in tokens (roughly words). This is crucial for managing costs with paid APIs, as pricing is often per token.

4. **Anthropic Claude Model:**

- Uses ChatAnthropic.
- Requires an **Anthropic API key** (also a paid service).
- Similar code structure to ChatOpenAI, demonstrating LangChain's **consistent interface**.

5. **Google Gemini Model:**

- Uses ChatGoogleGenerativeAI.
- Requires a **Google Gemini API key**.
- Similar consistent code structure.

C. Open-Source Language Models:

- **Definition:** "Freely available AI models that can be downloaded, modified, fine-tuned, and deployed without restrictions from a central provider."
- **Advantages over Closed-Source Models:Cost:** Free to use locally (no API costs).
- **Control:** Full control over fine-tuning, modification, and deployment.
- **Data Privacy:** Processing happens locally, preventing sensitive data from being sent to external servers.
- **Customization:** Can be fine-tuned on custom datasets.
- **Deployment:** Can be deployed on private servers or cloud.
- **Famous Open-Source Models:** LLaMA, Mistral, Falcon, Bloom.
- **Where to Find: Hugging Face** is "the largest repository of open-source LLMs," offering thousands of models.

1. **Ways to Use:Hugging Face Inference API:** Use the model hosted on Hugging Face's servers via an API. Requires a Hugging Face Access Token (free for basic use, paid for higher limits).

- Uses ChatHuggingFace and HuggingFaceEndpoint.
- Example: Using "TinyLlama/TinyLlama-1.1B-Chat-v1.0".

2. **Local Download and Execution:** Download the model to your machine and run it locally.

- Uses ChatHuggingFace and HuggingFacePipeline.
- **Disadvantages:** Requires "solid hardware" (expensive GPUs), can be complex to set up, may have "less refinement" in responses compared to closed-source models (due to less RLHF), and generally offers "limited multimodal abilities" (mostly text-based at present).
- **Demonstration:** Running TinyLlama locally, noting significant processing time and hardware demands.

IV. Embedding Models: Deep Dive and Application

A. Open-Source Embedding Models (OpenAI):

- Uses OpenAIEmbeddings.
- Requires an OpenAI API key.

- **Configuration:** Specify the model (e.g., "text-embedding-3-large") and the desired dimensions of the output vector (e.g., 32). Larger dimensions capture more contextual meaning but are more expensive.
- **Methods:**embed_query("single sentence"): Generates an embedding for a single text.
- embed_documents(["doc1", "doc2", "doc3"]): Generates embeddings for multiple documents as a list of vectors.
- **Cost:** "Cost actually is very less, 1 million tokens process karne ka cost is roughly 1 dollar. So cost bahut kam hai because obviously model bahut kam output deta hai numbers mein output deta hai." OpenAI embeddings are generally recommended for their accuracy.

B. Open-Source Embedding Models (Hugging Face):

- Uses HuggingFaceEmbeddings.
- Example: Using "sentence-transformers/all-MiniLM-L6-v2" (a small, 90MB model).
- Can be used via Inference API or locally downloaded.
- **Local Execution:** Model files (including tokenizer) are downloaded to the machine on first run.

V. Document Similarity Application (Semantic Search)

- **Scenario:** Given a set of documents (e.g., short bios of cricketers) and a user query (e.g., "Tell me about Virat Kohli"), find the most relevant document.

1. **Process:**Generate **embeddings (vectors)** for all available documents.
2. Generate an embedding for the user's query.
3. Calculate **cosine similarity** between the query embedding and each document embedding. Cosine similarity measures the angle between vectors; a higher score indicates greater similarity.
4. Sort the similarity scores to find the document with the highest similarity.
5. Retrieve and display the most similar document.

- **Libraries Used:** OpenAIEmbeddings, load_dotenv, cosine_similarity from sklearn.metrics.pairwise, numpy.
- **Concept Connection:** This process forms the basis of **RAG-based applications**, where document embeddings are typically stored in a **vector database** for efficient retrieval. This avoids re-generating embeddings for static documents and optimizes the retrieval process for new queries.