LLM

An LLM (Large Language Model) is like a super-smart language "expert" that understands and generates human-like text. It's a computer program trained on massive amounts of text from books, websites, and articles. This "expert" can understand your questions, write essays or stories, translate languages, summarise information, and even converse with you. It's a tool that helps computers understand and generate human language, making it easier for us to interact with technology in more natural and conversational ways.

In computing, LLM stands for "Large Language Model." It refers to a sophisticated artificial intelligence (AI) system designed to understand, process, and generate human language. These models are built using deep learning techniques and are trained on vast amounts of text data to learn language patterns, structures, and nuances.

LLMs employ neural networks with many layers (hence the term "deep learning") to process and understand text at a very complex level. They can perform various language-related tasks such as text generation, translation, summarisation, sentiment analysis, question answering, and more.

Examples of well-known LLMs include GPT (Generative Pre-trained Transformer) models developed by OpenAI, which have different versions such as GPT-2, GPT-3, etc. These models have revolutionised natural language processing tasks by demonstrating exceptional abilities in understanding and generating human-like text.

Creating an application in Python that generates SQL queries based on prompts using LangChain involves several steps:

1. Install the necessary libraries: Start by installing the required packages, including LangChain, OpenAI API, and any database libraries you'll be using.
2. Set up credentials: Obtain your OpenAI API key and configure it in your Python environment. Similarly, set up the necessary connection parameters for your database.
3. Define the SQL query chain: Import the necessary classes from LangChain and create an instance of SQLDatabaseChain. This chain will handle the interaction between the LLM and the database.
4. Provide database metadata: Pass the metadata of your database, such as table names, column names, and data types, to the SQLDatabaseChain instance. This will allow the LLM to understand the structure of your data.
5. Handle user prompts: Implement a function or method to accept natural language prompts from the user. This function should prepare the prompt text for the LLM.
6. Generate SQL queries: Use the SQLDatabaseChain instance to generate SQL queries based on the user's prompts. The chain will leverage the LLM to interpret the prompts and translate them into syntactically correct SQL statements.
7. Execute the SQL queries: Connect to the database using the appropriate database driver and execute the generated SQL queries.
8. Process query results: Parse the query results and present them to the user in a meaningful format, such as tables, charts, or visualizations.
9. Refine the query chain: Iterate the process, refining the prompt structure and the chain's parameters to improve the accuracy and relevance of the generated SQL queries.
10. Install the required libraries:

Bash

pip install langchain psycopg2-binary

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Create a Python script:

Python

import langchain as lc

import psycopg2

# Configure LangChain

llm = lc.LLMClient()

llm.api\_key = "YOUR\_LANGCHAIN\_API\_KEY"

# Connect to PostgreSQL database

conn = psycopg2.connect(

host="localhost",

port=5432,

database="your\_database",

user="your\_username",

password="your\_password",

)

# Define SQL query chain

db\_chain = lc.SQLDatabaseChain(

llm=llm,

database=conn,

verbose=True,

)

# Prompt the user for a question

prompt = input("Enter your question: ")

# Generate SQL query from prompt

sql\_query, \_ = db\_chain.run(prompt)

# Execute SQL query and print results

cursor = conn.cursor()

cursor.execute(sql\_query)

results = cursor.fetchall()

print("Results:")

for row in results:

print(row)

# Close database connection

conn.close()

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Run the Python script:

Bash

python my\_application.py

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

This script will prompt the user for a question about the database, generate an SQL query from the prompt, execute the query, and print the results. You can customize the prompt to improve the accuracy of the generated queries. You can also add additional features to the application, such as saving the generated queries for later use or providing a graphical user interface.

To create an application in Python that generates SQL queries based on prompts using LangChain, follow these steps:

1. Install the required libraries:
   * langchain: Install the LangChain library using pip: pip install langchain
   * openai: Install the OpenAI API library using pip: pip install openai
   * psycopg2: Install the psycopg2 library for interacting with PostgreSQL databases: pip install psycopg2-binary
2. Set up the database connection:
   * Create a connection string for your PostgreSQL database. For example, if your database is hosted locally, the connection string might look like this: postgresql://username:password@localhost:5432/database\_name
3. Create a SQLDatabaseChain instance:
   * Import the necessary modules:

Python

import langchain as lc

import psycopg2

from openai import OpenAI

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

* Create an OpenAI client:

Python

openai\_client = OpenAI('YOUR\_API\_KEY')

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

* Connect to the PostgreSQL database:

Python

connection = psycopg2.connect(your\_connection\_string)

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

* Create the SQLDatabaseChain instance:

Python

db\_chain = lc.SQLDatabaseChain(llm=openai\_client, database=connection, verbose=True)

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Define a prompt template:
   * The prompt template should provide instructions for the language model to generate a SQL query based on the user's input. You can include details about the database schema, specific tables to use, and examples of relevant queries.

Python

prompt\_template = """

Given an input question, first create a syntactically correct SQL query to run, then look at the results of the query and return the answer.

Only use the following tables: {table\_info}.

"""

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Process user input and generate SQL queries:
   * Create a function to handle user input and generate SQL queries:

Python

def generate\_sql\_from\_prompt(user\_input):

# Replace {table\_info} with a list of tables to use

table\_info = ['table\_name1', 'table\_name2']

prompt = prompt\_template.format(table\_info=table\_info)

query = db\_chain.run(prompt=prompt, user\_input=user\_input)

return query

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Create an interactive application:
   * Create a loop to continuously prompt the user for input and generate SQL queries:

Python

while True:

user\_input = input("Enter your question or query: ")

sql\_query = generate\_sql\_from\_prompt(user\_input)

print(f"Generated SQL query: {sql\_query}")

Use code with caution. [Learn more](https://bard.google.com/faq#coding)

content\_copy

1. Run the application:
   * Save the code as a Python script and run it in your preferred environment. You can then input your SQL queries or questions, and the application will generate the corresponding SQL statements.

To make the code more modular and maintainable, you can break it down into separate modules for each major component:

1. Configure Database Connections:

Python

def configure\_database\_connections(config\_path):

"""

Configures database connections based on the specified configuration file.

Args:

config\_path (str): Path to the configuration file containing database connection details.

Returns:

List of SQLDatabase objects representing the connected databases.

"""

# Load configuration file

config = ConfigurationParser.from\_file(config\_path)

# Define database connections

databases = []

for database\_config in config['databases']:

uri = database\_config['uri']

type = database\_config['type']

if type == 'mongodb':

database = MongoDBDatabase(uri)

elif type == 'mysql':

database = MySQLDatabase(uri)

elif type == 'oracle':

database = OracleDatabase(uri)

else:

raise ValueError("Unsupported database type: {}".format(type))

databases.append(database)

return databases

Use code with caution. Learn more

2. Define SQL Query Chain:

Python

def define\_sql\_query\_chain(query\_chain\_path):

"""

Load and defines the SQL query chain from the specified YAML file.

Args:

query\_chain\_path (str): Path to the YAML file containing the SQL query chain.

Returns:

SQLDatabaseChain object representing the defined query chain.

"""

# Load query chain YAML file

with open(query\_chain\_path, 'r') as f:

query\_chain = yaml.load(f, Loader=yaml.FullLoader)

# Initialize SQL database chain

sql\_query\_chain = SQLDatabaseChain(llm=OpenAI(), database=database, query\_chain=query\_chain)

return sql\_query\_chain

Use code with caution. Learn more

3. Provide Database Metadata:

Python

def provide\_database\_metadata(databases):

"""

Retrieves metadata for each connected database.

Args:

databases (List[SQLDatabase]): List of connected SQLDatabase objects.

Returns:

Dict: Dictionary containing database metadata.

"""

metadata = {}

for database in databases:

metadata[database.name] = database.get\_metadata()

return metadata

Use code with caution. Learn more

4. Generate SQL Queries:

Python

def generate\_sql\_queries(sql\_query\_chain, prompt):

"""

Generates SQL queries based on the provided prompt and query chain.

Args:

sql\_query\_chain (SQLDatabaseChain): SQL database chain object.

prompt (str): User-provided query prompt.

Returns:

List: List of generated SQL queries.

"""

# Generate SQL queries using the query chain

sql\_queries = sql\_query\_chain.run(prompt)

return sql\_queries

Use code with caution. Learn more

5. Execute Generated SQL Queries:

Python

def execute\_generated\_sql\_queries(databases, sql\_queries):

"""

Executes the generated SQL queries on the connected databases.

Args:

databases (List[SQLDatabase]): List of connected SQLDatabase objects.

sql\_queries (List[str]): List of generated SQL queries.

Returns:

Dict: Dictionary containing query results per database.

"""

query\_results = {}

for database in databases:

query\_results[database.name] = database.execute(sql\_queries)

return query\_results

Use code with caution. Learn more

6. Process Query Results:

Python

def process\_query\_results(query\_results, metadata):

"""

Processes the query results from each database.

Args:

query\_results (Dict): Dictionary containing query results per database.

metadata (Dict): Dictionary containing database metadata.

Returns:

List: List of processed query results.

"""

processed\_results = []

for database\_name, db\_results in query\_results.items():

# Process query results using database metadata

processed\_results.append(\_

Use code with caution. Learn more