Building an LLM-Powered Course Query System

Cuong Dang

2024-12-16

1. Project Overview and Objectives

This project aims to deploy a LangChain-powered application that uses the provided CSV file "FTCM_Course List_Spring2025.csv" as a database for a course query system. This project will involve data cleaning, database design, and developing a conversational application using LangChain.

2. Code Explanations

2.1 SQL Generator

```
prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    ("human", human_prompt),
])

# Build query generation chain
sql_generator = (
    RunnablePassthrough.assign(schema=get_schema)
    | prompt
    | llm.bind(stop=["\nSQLResult:"])
    | StrOutputParser()
)
```

- Takes in the custom system_prompt and human_prompt that is run in the SQL generation chain.

- Create a SQL query that answer the user's natural language question and return it in as a string.

2.2 Natural Language Response & Complete Chain

```
nl_prompt = ChatPromptTemplate.from_messages([
    ("system", nl_system_prompt),
    ("human", response_prompt),
])

# Complete chain with natural language output
complete_chain = (
    RunnablePassthrough.assign(query=sql_generator)
    | RunnablePassthrough.assign(
        schema=get_schema,
        response=lambda x: db.run(x['query']),
    )
    | nl_prompt
    | llm
)
```

- nl_prompt takes in nl_system_prompt and response_prompt, returns a natural human language response when SQL query response from database and the original user's question is the input.
- complete_chain wrapped the sql generation and the natural language response together. Before the nl_prompt, SQL query response is created by running the generating query with db.run(query).

2.3 Input Validator

```
class InputValidator:
    """Input validation and sanitization for database queries"""

    # Restricted words that might indicate harmful queries
    RESTRICTED_WORDS = {
        'delete', 'drop', 'truncate', 'update', 'insert',
        'alter', 'create', 'replace', 'modify', 'grant'
    }
```

```
    # Allowed table names from our database
    ALLOWED_TABLES = ['spring_2025_courses']  # in lowercase

    def __init__(self):
        self.error_messages = []

    def validate_question(self, question: str) -> bool:

        self.error_messages = []

        # Check if question is empty or too long
        if not question or not question.strip():
            self.error_messages.append("Question cannot be empty")
            return False

        if len(question) > 500:
            self.error_messages.append("Question is too long (max 500 characters)")
            return False

        # Check for basic SQL injection attempts
        question_lower = question.lower()
        if any(word in question_lower for word in self.RESTRICTED_WORDS):
            self.error_messages.append("Question contains restricted keywords")
            return False

        # Check for excessive special characters
        if re.search(r'[;{}\\]', question):
            self.error_messages.append("Question contains invalid characters")
            return False

        return True

    def get_error_messages(self) -> List[str]:
        """Get all error messages from validation"""
        return self.error_messages
```

- Checks for invalid questions from the users.
- Checks for restricted words that can alter the database as well as any possible requests that can create an error in the chain.

2.4 Query Validator

```python
class QueryValidator:
    """Validate generated SQL queries before execution"""

    def __init__(self, db_connection):
        self.db = db_connection
        self.error_messages = []

    def get_error_messages(self) -> List[str]:
        """Get all error messages from validation"""
        return self.error_messages

    def validate_sql(self, sql: str) -> bool:
        """Validate generated SQL query"""
        sql_lower = sql.lower()

        # Check if query is read-only (SELECT only)
        if not sql_lower.strip().startswith('select'):
            self.error_messages.append("Only SELECT queries are allowed")
            return False

        # Check for multiple statements
        if ';' in sql[:-1]:  # Allow semicolon at the end
            self.error_messages.append("Multiple SQL statements are not allowed")
            return False

        # Validate table names
        table = self._extract_table_names(sql_lower)
        if table not in InputValidator.ALLOWED_TABLES:
            self.error_messages.append("Query contains invalid table names")
            return False

        return True

    def _extract_table_names(self, sql: str) -> str:
        """Extract table names from SQL query"""
        from_matches = re.findall(r'from\s+([a-zA-Z_][a-zA-Z0-9_]*)', sql.lower())
        return from_matches[0]
```

- Check if the query generated from the sql_generator is valid or not.
- Check for invalid table names and queries that can alter the database.

2.5 Safe Query Executor

```
class SafeQueryExecutor:
    """Safe execution of natural language queries"""

    def __init__(self, db_connection, llm_chain):
        self.input_validator = InputValidator()
        self.query_validator = QueryValidator(db_connection)
        self.db = db_connection
        self.chain = llm_chain

    def execute_safe_query(self, question: str) -> Dict:
        """
        Safely execute a natural language query

        Args:
            question (str): User's natural language question

        Returns:
            dict: Query result and status
        """
        # Validate input
        if not self.input_validator.validate_question(question):
            return {
                'success': False,
                'error': self.input_validator.get_error_messages(),
                'query': None,
                'result': None
            }

        try:
            # Generate SQL query
            sql_query = sql_generator.invoke({"question": question})

            # Validate generated SQL
            if not self.query_validator.validate_sql(sql_query):
                return {
                    'success': False,
                    'error': self.query_validator.get_error_messages(),
                    'query': sql_query,
                    'result': None
                }
```

```
            # Execute query
            result = complete_chain.invoke({"question": question})

            return {
                'success': True,
                'error': None,
                'query': sql_query,
                'result': result
            }

        except Exception as e:
            return {
                'success': False,
                'error': [str(e)],
                'query': None,
                'result': None
            }
```

- Class that wrap the InputValidator and QueryValidator classes to validate the user's question as well as the llm response.

2.6 Gradio Interface

```
import gradio as gr
from functions_qwen import safe_executor, example_queries

def execute_query(query):
    result = safe_executor.execute_safe_query(query)

    if result['success'] == True:
        return result['result']
    else:
        return result['error']

def update_textbox(prompt):
    return gr.update(value=prompt)

with gr.Blocks() as demo:
    response_box = gr.Textbox(label="Response", interactive=False)
    msg = gr.Textbox(label="Type your message or select a prompt")
```

```
    with gr.Row():
        prompt_dropdown = gr.Dropdown(choices=[""] + example_queries, label="Select a premade
        submit = gr.Button("Submit")
    clear = gr.ClearButton([msg, response_box])

    prompt_dropdown.change(update_textbox, inputs=[prompt_dropdown], outputs=[msg])
    submit.click(execute_query, inputs=[msg], outputs=[response_box])
    msg.submit(execute_query, inputs=[msg], outputs=[response_box])
    clear.click(lambda: None, None, [msg, response_box], queue=False)

if __name__ == "__main__":
    demo.launch()
```

- Separate file that import functions from functions_qwen.py
- Create a simply gradio interface with a dropdown of sample prompts, textbox for user's questions, and a response textbox above to show the result

3. Challenges Faced and Solutions Implemented

- **Challenge**: Multi-round Interaction
  - **Solution**: I couldn't find a solution for this. I tried using LangGraph memory (https://python.langchain.com/docs/versions/migrating_memory/conversation_buffer_memory/) but it still didn't work. Whenever I tried using the memory chain, the model would generate its own question and pass it into the human prompt and create a loop of answering itself. Not sure how to prevent this
- **Challenge**: Deploying on HuggingFace
  - **Solution**: The code works locally, but I face difficulties with deploying on huggingface because the llm model for the code takes too much space in GPU (limit is 16g) and the app would stop building. I tried using smaller models such as "Qwen/Qwen2.5-14B-Instruct", but it still wouldn't work. Then I found out that I can add my api key into the app and use the same model that I used locally.