SQL Query Generation Using Prompt Engineering

Malhar Sham Ghogare

ghogare.m@northeastern.edu

Abstract

This report presents an in-depth analysis of utilizing prompt engineering techniques to extract structured data from unstructured text, transform it into JSON format, and generate SQL queries for storage and retrieval. The study involves five key tasks: extracting company attributes from a text file, converting the extracted data into JSON format, generating an SQL table creation query, developing an SQL INSERT query, retrieving top and bottom-ranked companies using SQL, and grouping companies by state and industry. The report evaluates the performance of the generated queries in terms of accuracy, consistency, readability, and SQL dialect compatibility.

1. Introduction

With the increasing need to process unstructured data efficiently, leveraging Large Language Models (LLMs) such as OpenAI's GPT-4 has become a viable approach for automating database operations. This study aims to evaluate the ability of GPT-4 to extract structured information from raw text and convert it into SQL-compatible formats. The approach involves designing structured prompts, analyzing the generated SQL queries, and assessing their accuracy and usability. The insights from this study will inform best practices in prompt engineering for structured data extraction and SQL automation.

2. Methodology

The study follows a systematic approach, starting with loading and processing the CompanyInfo.txt dataset, followed by prompt engineering to extract and transform data into JSON format. The extracted data is then used to generate SQL queries for table creation, data insertion, and structured retrieval. The methodology includes evaluating the correctness, efficiency, and clarity of the generated queries.

3. Code Initialization and File Processing:

3.1. Importing Required Libraries:

Before extracting and processing data, several essential steps were executed, including **importing necessary libraries, loading the text file, and defining a function for interacting with OpenAI's GPT-4 model**.

```
#Importing the necessary libraries

import openai
from IPython.display import display, Markdown
```

The openai library is imported to interact with the OpenAI API for prompt engineering.

The IPython.display module is used to format and display SQL outputs in Markdown format for better readability in Jupyter Notebook.

3.2. Loading the Text File:

```
#Opening the text file
with open('/content/CompanyInfo.txt', 'r') as file:
    text_data = file.read()

#Displaying the text data
print(text_data)
```

The CompanyInfo.txt file is opened in read mode ('r'), and its contents are stored in text_data.

The print(text_data) statement ensures that the data is successfully loaded into the script.

3.3. Setting Up OpenAI API Key:

```
#Setting my OpenAI API key (Replace 'your_api_key_' with your actual key)
openai.api_key = "your_api_key"
# Function to send a prompt to OpenAI and retrieve a response
def get_openai_response(prompt):
    response = openai.ChatCompletion.create(
        model="gpt-4",  # Using GPT-4 model
        messages=[{"role": "system", "content": "You are an expert in SQL and data analysis."},
                  {"role": "user", "content": prompt}],
        temperature=0.7  # Adjust for variability in responses
    )
    return response['choices'][0]['message']['content']
```

The OpenAI API key is assigned to authenticate requests to GPT-4.

3.4 Defining a Function to Interact with OpenAI:

This function sends a structured SQL-related prompt to OpenAI and retrieves the model's response.

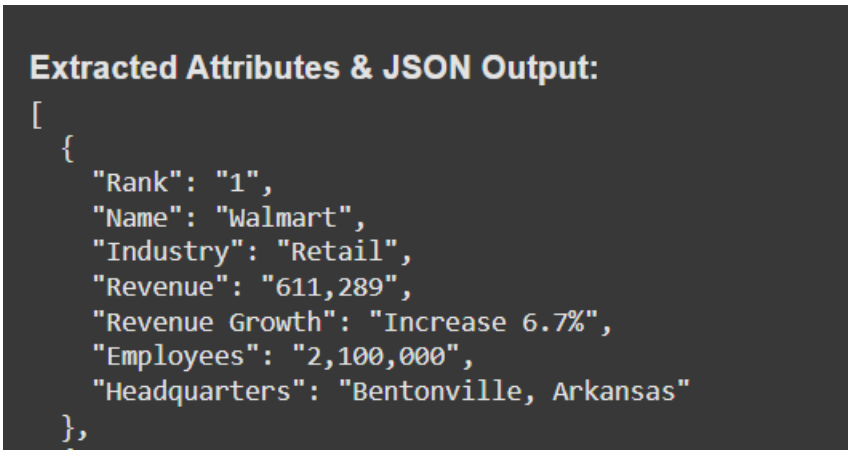
The temperature parameter is set to 0.7, which introduces response variability. Lower values (e.g., 0) would be preferable for structured SQL queries.

The function extracts and returns the first response from the model's output.

4. Extracting Attributes and Converting to JSON

4.1. Implementation

The first task involved designing a prompt to extract the attributes "Rank," "Name," "Industry," "Revenue," "Revenue Growth," "Employees," and "Headquarters" from the given text data. The extracted data was structured into JSON format for easy database ingestion. The prompt defined a structured format for expected output, ensuring consistency across multiple records.



4.2. Evaluation and Insights

The extracted attributes matched the dataset accurately, and the JSON format was well-structured. However, numerical values were stored as strings, requiring conversion for SQL operations. Additionally, the prompt did not explicitly specify whether all records should be extracted or only a subset, leading to potential ambiguity.

5. Generating an SQL Table Creation Query

5.1. Implementation

A structured prompt was formulated to generate an SQL query for creating a table named CompanyInfo to store the extracted attributes. The table schema included appropriate data types and constraints, such as VARCHAR(255) for textual attributes, DECIMAL(15,2) for revenue-related fields, and a PRIMARY KEY constraint on the Rank column.

SQL Table Creation :

```sql
CREATE TABLE CompanyInfo (
    Rank INT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Industry VARCHAR(255),
    Revenue DECIMAL(15,2) CHECK (Revenue >= 0),
    Revenue_growth DECIMAL(5,2),
    Employees INT,
    Headquarters VARCHAR(255)
);
```

5.2. Evaluation and Insights

The generated query was structurally sound and included necessary constraints. The use of DECIMAL(15,2) for financial data ensured precision. However, the Revenue_growth column lacked a range constraint to prevent invalid percentage values, and the Employees column was nullable, which might not align with real-world business requirements.

6. Generating an SQL INSERT Query Using JSON Data

6.1. Implementation

A prompt was designed to generate an SQL INSERT query that takes JSON data and inserts it into the CompanyInfo table. The prompt specified the JSON structure and requested the use of SQL functions for data extraction.

Generated SQL INSERT Query:

Here is an SQL INSERT query that extracts values from the JSON object and inserts them into the 'CompanyInfo' table:

```sql
INSERT INTO CompanyInfo (Rank, Name, Industry, Revenue, RevenueGrowth, Employees, Headquarters)
VALUES (
  CAST(JSON_VALUE(@company_data, '.Rank') AS INT), JSON_VALUE(@company_data, '.Name'),
  JSON_VALUE(@company_data, '.Industry'), CAST(JSON_VALUE(@company_data, '.Revenue') AS FLOAT),
  JSON_VALUE(@company_data, '.RevenueGrowth'), CAST(JSON_VALUE(@company_data, '.Employees') AS INT),
  JSON_VALUE(@company_data, '$.Headquarters')
);
```

This query uses the SQL function `JSON_VALUE` to extract values from the JSON object. The `CAST` function is used to convert the extracted JSON values to the appropriate data types for the 'CompanyInfo' table.

Please note that the SQL syntax for handling JSON can vary between different SQL databases. The above query is written for SQL Server. If you are using a different SQL database, you may need to adjust the syntax accordingly. ```

6.2. Evaluation and Insights

The SQL query correctly extracted values from JSON using JSON_VALUE(). However, the output formatting was severely corrupted, making the query unreadable and unusable in its generated form. This issue likely resulted from encoding or model tokenization errors. Fixing

the formatting and ensuring proper SQL dialect compatibility (e.g., handling JSON_EXTRACT() for MySQL) would improve the usability of the output.

7. Generating an SQL Query for Retrieving the Top 5 and Bottom 5 Ranked Companies

7.1. Implementation

A prompt was created to generate an SQL query retrieving the top 5 and bottom 5 ranked companies. The model was instructed to use either a UNION of two SELECT queries or a **window function**, depending on SQL dialect compatibility.

Generated SQL Query for Top 5 & Bottom 5 Companies:

Here is an SQL query using UNION of two SELECT queries:

```sql
(SELECT * FROM CompanyInfo ORDER BY Rank ASC LIMIT 5)
UNION
(SELECT * FROM CompanyInfo ORDER BY Rank DESC LIMIT 5)
```

Here is an SQL query using window function:

```
SELECT * FROM (
  SELECT *, ROW_NUMBER() OVER (ORDER BY Rank ASC) as TopRank, ROW_NUMBER() OVER (ORDER BY Rank DESC) as BottomRank
  FROM CompanyInfo
) AS RankedCcompanies
WHERE TopRank <= 5 OR BottomRank <= 5
```

Please note that the window function query might not work in all SQL dialects as not all of them support window functions. ```

7.2. Evaluation and Insights

The UNION query successfully combined results from two SELECT statements, ensuring broad SQL dialect compatibility. However, the model also generated an alternative query using a ROW_NUMBER() window function, which might not be supported in all databases. The output contained minor formatting errors, such as missing semicolons and unnecessary nesting, which should be refined for execution consistency.

8. Generating SQL Queries for Grouping Companies by State and Industry

8.1. Implementation

Two prompts were created—one to generate an SQL query grouping companies by state and another to group companies by industry. The prompt for state extraction included instructions to extract the **state name** from the Headquarters column, assuming the format "City, State."

Generated SQL Query for Grouping by State:

Here is a SQL query that meets your requirements:

```sql
SELECT
    TRIM(SUBSTR(Headquarters, INSTR(Headquarters, ',') + 1)) AS State,
    COUNT(*) AS CompanyCount
FROM
    CompanyInfo
GROUP BY
    State
ORDER BY
    CompanyCount DESC;
```

This query works as follows:

- The `SUBSTR` function is used to extract the state from the 'Headquarters' column. The `INSTR` function is used to find the position of the comma, and then everything after the comma is extracted.
- The `TRIM` function is used to remove any leading or trailing spaces from the state name.
- The `COUNT(*)` function is used to count the number of companies in each state.
- The `GROUP BY` clause is used to group the companies by state.
- The `ORDER BY` clause is used to sort the states by the number of companies in descending order. ```

Generated SQL Query for Grouping by Industry:

Here is the SQL query:

```sql
SELECT Industry, COUNT(*) AS CompanyCount
FROM CompanyInfo
GROUP BY Industry;
```

This query will return a list of industries and the number of companies in each industry. The `GROUP BY` clause groups the companies by industry, and the `COUNT(*)` function counts the number of companies in each group. ```

8.2. Evaluation and Insights

The **state grouping query effectively extracts state names** using SUBSTR() and INSTR(), but this method is SQL dialect-specific and might require alternatives in some databases (e.g., PostgreSQL). The **industry grouping query is simple and efficient**, leveraging GROUP BY for counting occurrences. The SQL formatting was incomplete in the output, with missing code block closures affecting readability.

9. Conclusion

This study demonstrated the effectiveness of **GPT-4 in generating structured SQL queries** based on well-designed prompts. The model successfully extracted attributes, converted them into JSON format, and produced SQL queries for table creation, data insertion, and structured retrieval. However, the **output formatting issues, SQL dialect inconsistencies, and occasional syntax errors** highlight the need for human oversight in refining and validating generated queries. Future work should focus on refining prompt design to **minimize output corruption** and improve cross-database compatibility.