README Documentation for Text Query to JSON Response Conversion

Overview

Welcome to the project! This solution is designed to take user text queries and convert them into structured JSON responses by interacting with a dataset. Built using Python, the system combines data extraction, SQL query generation, and result processing to provide clear and accurate answers to user questions.

Features

- **Data Extraction and Storage**:
  a. The program reads data from a CSV file and allows you to select specific columns.
  b. It preprocesses the column names into a more user-friendly snake_case format.
  c. The processed data can be saved both to a new CSV file and a SQLite database for easy querying.
- **SQL Query Generation**:
  a. We leverage an external tool called Groq to create SQL queries based on what the user inputs.
  b. The system intelligently handles synonyms and common typos, ensuring that even vague or incorrect queries are interpreted accurately.
  c. It maintains a focus on necessary columns to guarantee relevant results.
- **Query Execution and Result Processing**:
  a. Once the SQL queries are generated, the program executes them and formats the results into a clear JSON structure.
  b. It's equipped to handle situations where no data is found or errors occur during execution, providing helpful fallback responses.
- **Edge Case Handling**:
  a. The system checks for empty or irrelevant queries and responds appropriately.
  b. If any issues arise during SQL execution, it catches exceptions and avoids crashing, ensuring a smooth user experience.

Implementation Details

Key Components

1. **DataExtractor**:
    a. This class is in charge of loading the dataset, preprocessing the data, and saving it in the desired formats.
2. **Key Methods**:
    a. `load_data()`: Loads the CSV data and validates the selected columns.
    b. `convert_to_snake_case()`: Changes column names to snake_case for consistency.
    c. `preprocess_data()`: Prepares the data for use by renaming columns and adding unique IDs.
    d. `save_to_csv()` and `save_to_sql()`: Save the data to specified CSV and SQLite formats.
3. **QueryExecutor**:
    a. Responsible for running SQL queries and formatting the results.
4. **Key Methods**:
    a. `process_query_result()`: Formats the raw SQL results into a user-friendly JSON format.
    b. `make_df()`: Converts the results into a Pandas DataFrame for easier manipulation.
    c. `df_to_custom_json()`: Turns the DataFrame into a structured JSON list.
    d. `execute_query()`: Runs the SQL query and handles the results.
5. **SQLGenerator**:
    a. This class generates SQL queries based on user input.
6. **Key Methods**:
    a. `build_sql_query()`: Constructs a detailed prompt for generating SQL.
    b. `generate_content()`: Interacts with Groq to produce the final SQL query.

Text to SQL Approach

The method we use for converting text queries into SQL is both innovative and practical. By utilizing natural language processing, we can take user questions—no matter how they're phrased—and translate them into precise SQL commands.

Benefits:

- **User-Friendly**: Users don't need to know SQL to interact with the database. They can simply ask questions in plain language.
- **Flexibility**: The system accommodates a wide range of queries, including those with synonyms and common errors, making it robust against variations in user input.
- **Efficiency**: The direct conversion to SQL means that users receive quick, accurate responses without needing to navigate complex interfaces.

Model Selection and API Hosting

The SQL generation model is hosted on Groq, which allows us to take advantage of powerful language processing capabilities. Groq provides an API that we can easily call to create SQL queries.

Why Groq?

- **Advanced Processing**: Groq is designed for handling sophisticated tasks like natural language understanding, making it an excellent choice for this project.
- **Scalability**: By using Groq's cloud-based model, we can scale our solution to handle larger datasets and more complex queries without worrying about local resource limitations.
- **Ease of Integration**: The API is straightforward to use, allowing us to focus on building features rather than managing infrastructure.

Usage Guide

Installing Dependencies

Before running the application or script, make sure to install all the required dependencies. You can typically do this by running:

```
pip install -r requirements.txt
```

Also, place your larger dataset CSV file in the project directory with the name take_home_dataset.csv. Make sure you delete all the previous files present in the data folder while running the script on the larger dataset.

Starting the API Server

To launch the application and test the API, use the following command in your terminal:

```
uvicorn app:app --reload
```

Then, test the end point http://127.0.0.1:8000/query using Postman. Here is the payload tht should be attached in the body.

```
{"queries":["largest order no","cosmetics and clothes","apparel products","cosmetics and personal care products", "order ID greater than 4000."]}
```

Response:

I have provided a response.json file in the project folder which you can go through to see the format of the results.

Running the Script Directly

If you prefer to see immediate results from your queries without setting up an API, you can run the script directly:

```
python text_to_sql.py
```

What This Does: This command executes the text_to_sql.py script. It will run the main function, which processes a predefined list of user queries. The results will be printed to the terminal in JSON format, allowing you to quickly verify the functionality of your code. The response will be a list of jsons against each user query, all combined in a single final array and printed on the terminal.