LLM-Powered Semantic Data Catalog and Discovery System: Project Report

Date: May 20, 2025

1. Abstract

This project implements an LLM-Powered Semantic Data Catalog and Discovery system. The primary goal is to bridge the gap between raw technical database metadata and human understanding by leveraging Large Language Models (LLMs). The system extracts technical metadata (schema, data types, relationships) from a MySQL database, enriches this metadata with LLM-generated semantic descriptions and tags, and pre-computes embeddings for these descriptions to enable efficient natural language search. It features a vector search backend using FAISS, LLM-based re-ranking for improved search relevance, and an LLM-powered module to infer potential new relationships within the database schema. A Streamlit-based user interface provides a user-friendly way to search the catalog and explore discovered data assets and relationships.

2. Introduction

2.1. Motivation

In modern data-driven organizations, understanding and locating relevant data within complex databases is a significant challenge. Traditional metadata often lacks semantic context, making it difficult for data analysts, scientists, and business users to discover and interpret data effectively. This project aims to address this by creating a "smart" data catalog that understands the meaning behind the data.

2.2. Problem Statement

Technical metadata, while accurate, is often insufficient for users unfamiliar with the database's intricacies. There's a need for a system that can:

- Automatically generate human-understandable descriptions for database schemas.
- Allow users to search for data using natural language queries.
- Help uncover hidden or undocumented relationships between data entities.

2.3. Objectives

- To develop a system for extracting technical metadata from a MySQL database.
- To enrich the extracted metadata with semantic descriptions and tags using an LLM.
- To generate and store embeddings for semantic descriptions to facilitate vector search.
- To implement a natural language search interface with LLM-based re-ranking.
- To utilize an LLM to infer potential new relationships within the database schema.
- To provide a user-friendly interface for interacting with the catalog.

3. System Architecture

The system follows a modular architecture, with distinct Python scripts responsible for different stages of the data processing and interaction pipeline:

1. **Database Setup (`database_setup.py`):** Initializes the MySQL database (`semantic_catalog_db`) with the necessary schema for storing raw metadata, enriched metadata, embeddings, and inferred relationships. It also populates sample business tables (`Customers`, `Products`, `Orders`, `Order_Items`) for demonstration.
2. **Metadata Extraction (`metadata_extractor.py`):** Connects to the source MySQL database, inspects its schema (tables, columns, data types, primary keys, foreign keys), and fetches sample data. This technical metadata is saved to `extracted_metadata.json`.
3. **LLM Enrichment (`llm_enrichment.py`):** Loads the `extracted_metadata.json`. For each table and column, it prompts an LLM (via an OpenAI-compatible API like LM Studio) to generate a semantic description and relevant tags. This enriched information is then stored in the `enriched_metadata` table in the `semantic_catalog_db`.
4. **Embedding Pre-computation (`precompute_embeddings.py`):** Fetches the semantic descriptions from the `enriched_metadata` table. It uses a Sentence Transformer model (e.g., `all-MiniLM-L6-v2`) to generate vector embeddings for these descriptions and stores them back into the `enriched_metadata` table.
5. **Relationship Inference (`relationship_inferer.py`):** Takes the schema information from `extracted_metadata.json`, formats it for an LLM, and prompts the LLM to infer potential relationships between tables/columns that might not be explicitly defined by foreign keys. These inferred relationships are stored in the `inferred_relationships` table.
6. **Search API (`search_api.py`):** A Flask-based API that:
    - Loads enriched metadata and pre-computed embeddings from the database on startup.
    - Builds a FAISS index for efficient similarity search.
    - Provides a `/search` endpoint: takes a user query, generates its embedding, searches the FAISS index, optionally re-ranks results with an LLM, and returns relevant metadata.
    - Provides an `/inferred-relationships` endpoint to retrieve all inferred relationships.
7. **Search UI (`search_ui.py`):** A Streamlit web application that provides a user interface for:
    - Entering natural language search queries.
    - Displaying search results (tables, columns with their descriptions and tags).
    - Displaying inferred relationships from the database.

Data Flow: Raw DB Schema -> `metadata_extractor.py` -> `extracted_metadata.json` -> `llm_enrichment.py` (with LLM) -> `enriched_metadata` table -> `precompute_embeddings.py` -> Embeddings in `enriched_metadata` table. `extracted_metadata.json` -> `relationship_inferer.py` (with LLM) -> `inferred_relationships` table. User Query -> `search_ui.py` -> `search_api.py` (FAISS search + LLM reranking) -> Results to UI.

4. Modules and Functionalities (Features)

- **Metadata Extraction:** Extracts schema (tables, columns, types, keys) and sample data from a MySQL database.
- **LLM-Powered Enrichment:** Uses a Large Language Model (LLM) to generate human-readable semantic descriptions and relevant tags for tables and columns.
- **Embedding Generation:** Pre-computes embeddings for semantic descriptions using Sentence Transformers.
- **Vector Search:** Utilizes FAISS for efficient similarity search based on embeddings.
- **Natural Language Search:** Provides a Streamlit-based UI for users to search the catalog using natural language queries.

- **LLM Re-ranking:** Employs an LLM to re-rank initial search results for improved relevance.
- **Relationship Inference:** Uses an LLM to analyze the schema and infer potential new relationships between tables.
- **API & UI:** Flask API serves search results and inferred relationships; Streamlit UI provides user interaction.

5. Technologies Used

- **Python 3.x**
- **MySQL:** For storing technical and enriched metadata.
- **LangChain:** Framework for developing applications powered by LLMs.
- **Sentence Transformers (`sentence-transformers`):** For generating embeddings.
- **FAISS (`faiss-cpu`):** For efficient similarity search on vectors.
- **Flask:** Web framework for the search API.
- **Streamlit:** For building the interactive search UI.
- **LM Studio (or other OpenAI-compatible LLM server):** For serving the LLM (e.g., `gemma-3-4b-it-qat`).
- **`mysql-connector-python`:** For MySQL database connectivity.
- **`numpy`:** For numerical operations, often a dependency for ML libraries.

6. Implementation Details

- **`database_setup.py`**: Initializes the MySQL database schema and populates it with sample data. Creates tables for `enriched_metadata` (including columns for `object_type`, `object_name`, `semantic_description`, `tags`, `embedding_vector`) and `inferred_relationships` (columns for `source_table`, `source_column`, `target_table`, `target_column`, `relationship_type`, `description`).
- **`metadata_extractor.py`**: Connects to the MySQL database, inspects its schema using SQL queries (`SHOW TABLES`, `DESCRIBE table_name`, `SHOW CREATE TABLE table_name`), fetches sample data (`SELECT * FROM table_name LIMIT 5`), and saves this information into `extracted_metadata.json`. Handles JSON serialization for complex data types like `bytes` and `datetime`.
- **`llm_enrichment.py`**: Loads `extracted_metadata.json`. For each table and column, it constructs prompts for an LLM to generate semantic descriptions and tags. It uses the LangChain library to interact with the LLM. Skips processing for metadata tables (`enriched_metadata`, `inferred_relationships`) and `embedding_vector` columns to avoid errors and unnecessary processing. Stores results in the `enriched_metadata` table.
- **`precompute_embeddings.py`**: Fetches semantic descriptions from `enriched_metadata`. Uses a Sentence Transformer model (e.g., `all-MiniLM-L6-v2`) to convert these text descriptions into dense vector embeddings. These embeddings are then stored as BLOBs in the `embedding_vector` column of the `enriched_metadata` table.
- **`relationship_inferer.py`**: Formats the schema information from `extracted_metadata.json` into a textual representation suitable for LLM input. Prompts the LLM to identify and describe potential relationships not explicitly defined by foreign keys. Parses the LLM's response and stores these inferred relationships in the `inferred_relationships` table.
- **`search_api.py`**:
    - On startup, loads all enriched metadata and embeddings. Builds a FAISS index from the embeddings for fast k-NN search.

    - `/search` endpoint: Takes a query, generates its embedding, performs a FAISS search, then uses an LLM to re-rank the top N results based on semantic relevance to the query. Deduplicates results based on logical entity (`object_type`, `object_name`, `parent_table_name`).
    - `/inferred-relationships` endpoint: Retrieves and returns all relationships from the `inferred_relationships` table, filtering out any involving the internal metadata tables.
- `search_ui.py`: A Streamlit application.
    - Provides a search bar for user queries and a button to view inferred relationships.
    - Calls the `/search` and `/inferred-relationships` endpoints of the `search_api.py`.
    - Displays search results in a structured, user-friendly format, including semantic descriptions and tags.
    - Presents inferred relationships clearly.
    - Includes custom CSS for improved visual appeal, a sidebar for branding, and a two-column layout for results.

7. Setup and Execution Steps

1. **Prerequisites:**
    - Python 3.8+
    - MySQL Server installed and running.
    - LM Studio (or a similar tool) running with an OpenAI-compatible API endpoint (e.g., serving `gemma-3-4b-it-qat` at `http://127.0.0.1:1234/v1`).
2. **Project Files:** Ensure all project scripts are in the project directory.
3. **Python Virtual Environment (Recommended):**

```
python -m venv venv
.\venv\Scripts\activate
```

4. **Install Python Dependencies:** Create `requirements.txt`:

```
mysql-connector-python
langchain
langchain-openai
sentence-transformers
faiss-cpu
flask
streamlit
numpy
```

Install:

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

5. **Configure Database Connection:** Update `DB_CONFIG` in `database_setup.py`, `metadata_extractor.py`, `llm_enrichment.py`, `precompute_embeddings.py`, `relationship_inferer.py`, `search_api.py`. Example:

```
DB_CONFIG = {
    'host': 'localhost',
    'user': 'your_user',
    'password': 'your_password',
    'database': 'semantic_catalog_db'
}
```

6. **Configure LLM Connection:** Update `LLM_BASE_URL` and `LLM_MODEL_NAME` in `llm_enrichment.py`, `relationship_inferer.py`, `search_api.py`.

7. **Execution Order:**

    1. `python database_setup.py`
    2. `python metadata_extractor.py`
    3. `python llm_enrichment.py` (Ensure LLM server is running)
    4. `python precompute_embeddings.py`
    5. `python relationship_inferer.py` (Ensure LLM server is running)
    6. `python search_api.py` (Keep running)
    7. In a new terminal: `streamlit run search_ui.py`

8. Results and Discussion

This section would typically showcase the system in action.

- **Figure 1: Screenshot of the Search UI home page.** *(Placeholder: Imagine a screenshot of the Streamlit UI with the search bar and branding)*
- **Figure 2: Screenshot of search results for a query like "customer order details".** *(Placeholder: Imagine a screenshot showing relevant tables and columns, their semantic descriptions, and tags)*
- **Figure 3: Screenshot of the Inferred Relationships display.** *(Placeholder: Imagine a screenshot showing a list of relationships inferred by the LLM)*

Discussion: The system successfully extracts, enriches, and indexes database metadata. The natural language search, powered by FAISS and LLM re-ranking, allows users to find relevant data assets effectively. The LLM-based relationship inference provides valuable insights into potential data connections that are not explicitly defined in the schema. The UI offers an intuitive way to interact with these features.

9. Challenges Faced and Solutions

- **MySQL "Specified key was too long" Error:** Encountered in `database_setup.py` for unique constraints on `VARCHAR` columns in `inferred_relationships`.
    - **Solution:** Reduced the length of `VARCHAR` columns involved in the unique key.
- **`TypeError` for `bytes` Serialization:** Occurred in `metadata_extractor.py` when trying to JSON serialize byte data (e.g., from BLOB/BINARY columns sample data).

    - **Solution:** Updated the `custom_json_serializer` to convert `bytes` to a string representation (e.g., base64 or a placeholder).
- **LLM Context Length Errors/Performance:** Initial attempts to enrich all metadata or infer relationships with very large schema prompts led to errors or slow responses from the LLM.
    - **Solution:**
        - In `llm_enrichment.py`: Implemented logic to skip LLM processing for metadata tables (`enriched_metadata`, `inferred_relationships`) and `embedding_vector` columns.
        - Refined prompting strategies to be more concise.
- **`KeyError` in `relationship_inferer.py`:** Due to incorrect parsing of the LLM's response or assumptions about the structure of `extracted_metadata.json`.
    - **Solution:** Added checks for key existence and adjusted parsing logic to correctly access nested data in the schema representation.
- **Search Result Deduplication:** Initial search results sometimes contained duplicates or near-duplicates.
    - **Solution:** Implemented deduplication logic in `search_api.py` after LLM re-ranking, first by ID, then refined to deduplicate based on a composite key representing the logical entity (e.g., `object_type`, `object_name`, `parent_table_name`).
- **Broken UI Logo Image:** Multiple attempts were needed to find a working and appropriate image URL for the Streamlit UI.
    - **Solution:** Iteratively tested different image URLs until a reliable one was found.

10. Conclusion

The LLM-Powered Semantic Data Catalog and Discovery system successfully demonstrates the potential of combining traditional metadata management techniques with the power of Large Language Models. It provides an effective solution for enhancing data discoverability and understanding within complex database environments. The project achieved its objectives of extracting, enriching, indexing, and providing a search interface for semantic metadata, along with inferring new data relationships.

11. Future Scope

- **Incremental Updates:** Implement mechanisms for incremental metadata extraction and enrichment, avoiding full re-processing.
- **User Feedback Loop:** Allow users to validate/correct LLM-generated content (descriptions, tags, relationships) and feed this back into the system.
- **Knowledge Graph Integration:** Represent and visualize metadata and relationships as a knowledge graph (e.g., using Neo4j, RDF).
- **Advanced Data Profiling:** Include more detailed data profiling statistics (e.g., nullability, uniqueness, distributions) in the catalog.
- **Support for More Data Sources:** Extend to other databases (PostgreSQL, SQL Server) and data formats (CSV, Parquet).
- **Enhanced UI/UX:** Add features like advanced filtering, sorting, data lineage visualization, and personalized views.
- **Security and Access Control:** Implement robust security if deployed in a production or multi-user environment.
- **Evaluation Framework:** Develop a framework to quantitatively evaluate the quality of semantic descriptions, search relevance, and inferred relationships.

12. References (Illustrative)

- LangChain Documentation: https://python.langchain.com/
- Sentence Transformers Library: https://www.sbert.net/
- FAISS Library: https://faiss.ai/
- Streamlit Documentation: https://docs.streamlit.io/
- Flask Documentation: https://flask.palletsprojects.com/
- MySQL Documentation: https://dev.mysql.com/doc/ *(You can add specific papers or articles if you referred to any)*