AetherMart Milestone 4: Vectorization & AI Data Prep README

Overview

This document details the implementation of Task 3 (Vector Database Setup & Search) and Task 4 (Refined AI Data Preparation) for AetherMart Milestone 4. These tasks involved leveraging MariaDB 11.8's native vector capabilities and Google's Gemini API to build a multi-modal semantic search engine across Products, Reviews, and Customers.

Task 3: Vector Database Setup & Search (Technical Implementation)

This task focused on the technical steps required to enable vector storage and fast similarity search within MariaDB.

1. Technology Used

- **Database:** MariaDB 11.8.3
- **Key Features:**
  - VECTOR(768) data type: Chosen to match the output dimension of the embedding-001 model. Stores vectors as binary data.
  - VEC_FromText(vector_string): SQL function used to convert the JSON-like string representation of a vector (e.g., '[0.1, -0.2, ...]') received from Python/API into MariaDB's internal binary format for storage.
  - VEC_DISTANCE_COSINE(vec1, vec2): SQL function to calculate the cosine distance between two vectors. Lower distance means higher similarity (0 = identical). Used as the core of our similarity search.
  - VECTOR INDEX: A specialized index type using the HNSW (Hierarchical Navigable Small Worlds) algorithm for efficient Approximate Nearest Neighbor (ANN) search. **Crucially, this index type requires the target column to be defined as NOT NULL**.

2. Two-Step Implementation Workflow

Due to the NOT NULL constraint for VECTOR INDEX conflicting with adding a column to existing tables, a strict two-step process was implemented for each data type (Products, Reviews, Customers):

Step 1: Embed Data (Python Script - e.g., embed_reviews.py)

- **Objective:** Prepare the table schema and load all vector data.
- **Key Actions:**
  1. **Connect:** Establish DB connection (pymysql).

2. **Table Prep (prepare_*_table function):**
    - Remove Conflicts: Check for and DROP PARTITIONING if present (required for Reviews).
    - Clean Slate: DROP INDEX IF EXISTS ..., DROP COLUMN IF EXISTS ... to handle re-runs.
    - Add Column: ALTER TABLE ... ADD COLUMN <embedding_col> VECTOR(768) NULL; (Added as NULLable to allow inserting into existing tables).
3. **Fetch Data:** Select necessary columns for feature engineering (detailed in Task 4).
4. **Feature Engineering:** Construct the context-rich string to be embedded (detailed in Task 4).
5. **Generate & Load Embeddings (Loop):**
    - Call Google Gemini API (genai.embed_content(..., task_type="retrieval_document")) for each feature string.
    - Convert the returned list vector to string format (vector_to_string function).
    - UPDATE <Table> SET <embedding_col> = VEC_FromText(%s) WHERE id = %s; using the vector string.
    - Include time.sleep(1.1) to manage API rate limits.
    - Commit transactions periodically.

Step 2: Finalize Schema & Index (SQL Script - e.g., create_review_index.sql)

- **Objective:** Make the vector column non-nullable and create the fast search index. **Run *only after* Step 1 is complete.**
- **Key Actions:**
    1. **Modify Column:** ALTER TABLE <Table> MODIFY COLUMN <embedding_col> VECTOR(768) NOT NULL; (This only succeeds if Step 1 filled all rows).
    2. **Create Index:** CREATE VECTOR INDEX idx_<embedding_col> ON <Table> (<embedding_col>);

3. Search Implementation (semantic_search.py)

- **Objective:** Allow users to perform semantic searches.
- **Key Actions:**
    1. **Get Query:** Prompt user for text input.
    2. **Vectorize Query:** Call Google Gemini API (genai.embed_content(..., task_type="retrieval_query")) to get the search vector.
    3. **Execute Search Query (SQL):**
        - Use VEC_DISTANCE_COSINE comparing the stored <embedding_col> with the query vector (passed via VEC_FromText(%s)).
        - ORDER BY distance ASC LIMIT 5 to find the 5 most similar items.
    4. **Display Results:** Convert distance to similarity (1 - distance) * 100 for user understanding.

Task 4: Refined AI Data Preparation (Intelligence &

Feature Engineering)

This task focused on *improving the quality of the input* provided to the AI embedding model (Gemini) during Step 1 of Task 3. The goal was to enhance the AI's contextual understanding and thus the relevance of search results.

1. Key Insight

Simply vectorizing raw, isolated text fields leads to poor semantic understanding ("Garbage In, Garbage Out"). **Feature Engineering** – creating structured, context-rich strings *before* embedding – is crucial.

2. Techniques Applied

Refinement 1: Products (embed_products.py)

- **Problem:** Vectorizing only product_description might confuse similar descriptions in different contexts (e.g., "mouse" toy vs. "mouse" electronic).
- **Engineered Feature:** Combined multiple relevant fields.

```
text_to_embed = (
  f"Product: {product['product_name']}. "
  f"Description: {product['product_description']} "
  f"Category: {product['category_name']}. "
  f"Price: ${product['price']}"
)
```

- **Benefit:** Provided the AI with category and price context, improving its ability to differentiate products and understand user queries relating to these aspects.

Refinement 2: Reviews (embed_reviews.py)

- **Problem 1:** Initial review data was junk (faker.sentence()) and ratings were inconsistent (NULL, invalid).
- **Solution 1 (Data Cleaning):** Created and ran fix_review_data.sql to populate review_text with meaningful sentences and ensure rating was a valid integer (1-5).
- **Problem 2:** Vectorizing only the cleaned review_text doesn't explicitly link the text's sentiment to the numerical rating.
- **Solution 2 (Feature Engineering + Hybrid Search Prep):**
  - Mapped numerical ratings to sentiment words (1->"Poor", ..., 5->"Excellent").
  - Constructed a sentiment-aware string:

```
sentiment_map = {1: "Poor", 2: "Fair", 3: "Good", 4: "Very Good", 5: "Excellent"}
sentiment_word = sentiment_map.get(review['rating'], "Unknown")
text_to_embed = (
  f"Review Text: {review['review_text']}. "
  f"Sentiment: {sentiment_word} (Rating: {review['rating']}/5)"
```

)

- **Benefit:** This explicit linking allowed the implementation of **Hybrid Search** in semantic_search.py. The script could now check the user's query for sentiment words ("good", "bad") and add a corresponding SQL WHERE rating IN (...) clause alongside the VEC_DISTANCE_COSINE search, providing highly relevant, sentiment-filtered results.

Refinement 3: Customers (embed_customers.py) - Exemplary Level

- **Problem:** Vectorizing simple demographics (name, city) is useless for finding "lookalike" customers based on behavior.
- **Solution (Advanced Feature Engineering):** Created a "Customer Purchase Profile" by querying their order history.
  - Used a complex SQL JOIN across 5 tables (Customers, Orders, Order_Items, Products, Categories) within the Python script.
  - Employed GROUP_CONCAT(DISTINCT category_name SEPARATOR ', ') to generate a summary list of purchased categories.
  - Constructed the profile string:

```
text_to_embed = (
  f"Customer {customer['first_name']} {customer['last_name']} "
  f"from {customer['city']}, {customer['state']}. "
  f"This customer primarily buys: {customer['purchase_categories'] or 'Unknown'}."
)
```

- **Benefit:** Vectorizing this behavioral profile enabled true "lookalike" customer searches based on purchasing patterns, directly addressing the business requirement and achieving the "Exemplary" rubric criteria for Task 4. Added "Evidence Queries" in semantic_search.py to display recent purchases and average rating, validating the lookalike match.

Conclusion

By combining the technical vector database setup (Task 3) with intelligent feature engineering (Task 4), we successfully built a powerful, multi-modal semantic search engine for AetherMart, capable of understanding user intent across products, reviews, and customer behavior. The two-step embedding/indexing process was crucial for navigating MariaDB constraints.