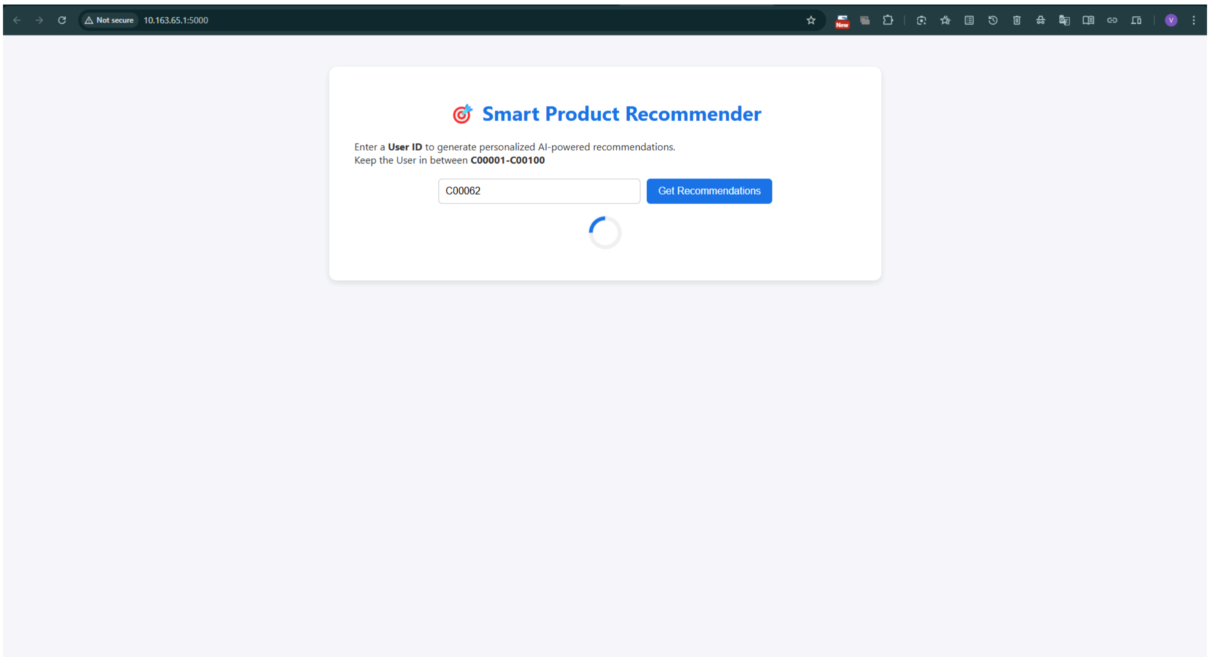
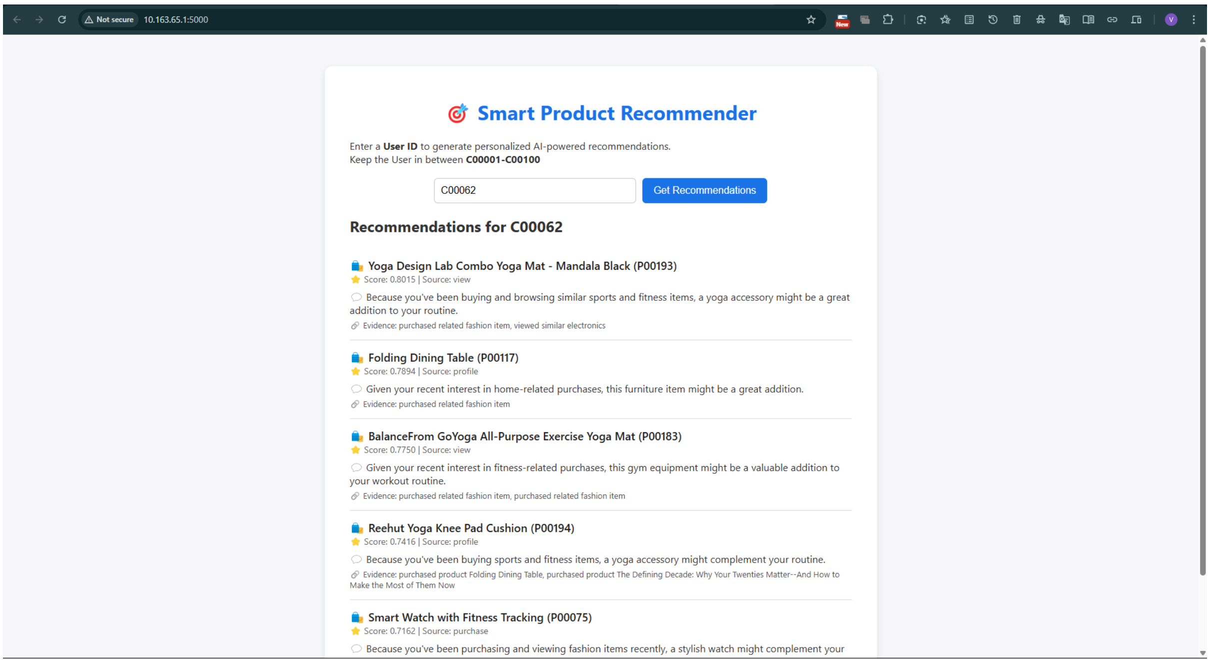
E-commerce Product Recommender

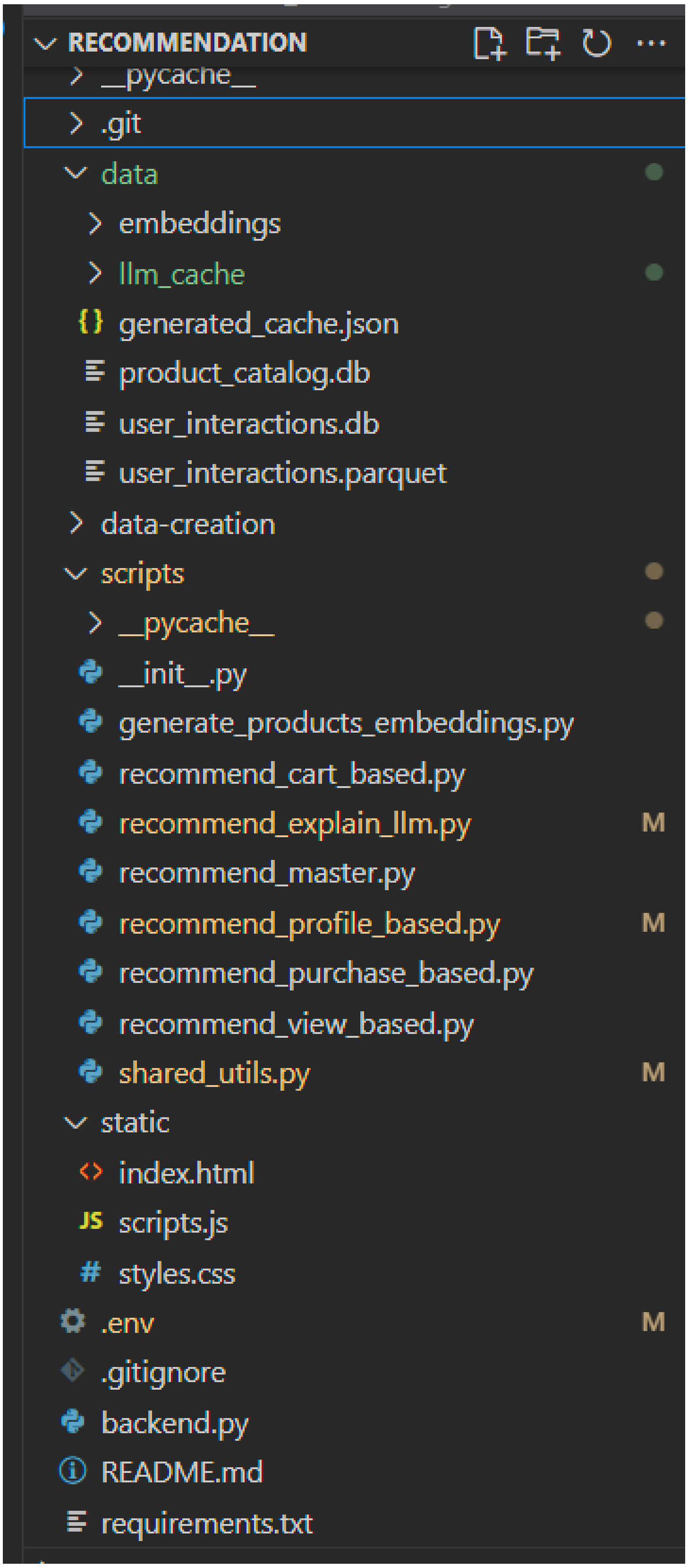
Repository Link: https://github.com/pathakx/Unthinkable

Samples:





Directories Structure:



Working of Code:

Recommendation

- **data/**
    - **_pycache__/**
    - **embeddings/**
    - **llm_cache/**
    - **generated_cache.json**
    - **product_catalog.db**
    - **user_interactions.db**
    - **user_interactions.parquet**
- **data-creation/**
    - **create_product_catalog_db.py**

      This Python script, **create_product_catalog_db.py**, is a **database utility module** for managing an e-commerce product catalog using **SQLite**.

      It establishes a flexible database schema by creating a products table with standard fields (e.g., product_id, pricing, ratings) and a dedicated features column to store **category-specific attributes as a JSON string**. The script provides essential functions to **connect**, **create** the robust products table, **insert** product records (serializing the flexible features), and **query** the data.

    - **Generate_catalog_from_gemini.py**

      This Python script is an **automated product catalog generator** for e-commerce.

      It uses the **Gemini 2.0 Flash API** to generate synthetic, realistic product data (including names, brands, prices, and features) for a predefined list of categories. To maximize efficiency and minimize API calls, it utilizes a local **JSON cache** to store generated products. Finally, it parses the Gemini's structured JSON output and inserts all the product listings, along with generated unique IDs and category metadata, into a local **SQLite database**.

- `User_interaction.py`

  This Python script **simulates a user interaction pipeline** by generating and storing synthetic e-commerce data.

  First, it defines configuration constants for the number of customers and products, and initializes a **SQLite database**. The generate_interaction_data function creates a Pandas DataFrame of random user-product events (view, add_to_cart, purchase) with timestamps.

  The script then **initializes the SQLite table** and uses store_interactions to persist the DataFrame to the database.

  Finally, the main execution block demonstrates how to **load and verify** the stored interactions for a random user, logging the process throughout.

- `scripts/`
  - `_pycache__/`
  - `__init__.py`
  - `generate_products_embeddings.py`

    This script is designed to **generate dense vector embeddings for product catalog data** from a SQLite database.

    It reads product attributes in chunks, composes a single **rich descriptive text** for each product by aggregating fields like name, brand, category, price, and features.

    The script then uses a **SentenceTransformer model** (defaulting to *all-MiniLM-L6-v2*) to encode these texts into dense embeddings in batches.

    Finally, it saves the generated embeddings, along with essential product metadata, into a **Parquet file** and creates a **manifest JSON file** detailing the run configuration and embedding properties.

  - `Recommend_cart_based.py`

    This Python function, recommend_cart_based, implements a **content-based recommendation strategy** focusing on products recently added to a user's cart.

    It first filters the user's interactions to isolate "add_to_cart" events, prioritizing the top 3 most relevant cart items.

    The core logic **builds a FAISS index** from product embeddings to efficiently search for similar items.

For each of the top cart items, it queries the FAISS index to find the **most similar products**, excluding the source product and already recommended items, up to a specified ***k*** limit.

- **recommend_explain_llm.py**

  This script provides a service to **generate personalized explanations for product recommendations** using the Gemini LLM, with caching for efficiency.

  It first **loads a Gemini client** (or defaults to a mock mode) and implements a time-limited **file-based cache** for generated explanations.

  The main function **summarizes a user's recent activity** and gathers target product details from Pandas DataFrames.

  It constructs a **personalized prompt** in a zero-shot fashion, calls the Gemini API to request a concise, JSON-formatted explanation, and **replaces product IDs with names** via a SQLite lookup before returning and caching the result.

- **recommend_master.py**

  This script orchestrates a **hybrid product recommendation pipeline** by combining multiple content-based strategies.

  It loads user interactions and pre-computed product embeddings, then calls four distinct recommendation functions: **view-based, cart-based, purchase-based, and profile-based**.

  The results from all sources are merged, retaining the highest score for any overlapping product, and the final top 5 recommendations are selected.

  Finally, it enriches the results by generating a **personalized LLM explanation** and fetching the product's actual name via a SQLite lookup.

- Recommend_profile_based.py

  This function, recommend_profile_based, generates **personalized product recommendations** using a user's pre-computed profile vector.

It retrieves the **dense embedding (vector)** that represents the user's aggregated interests and preferences.

The script then constructs a **FAISS index** from all product embeddings and performs a similarity search using the user vector as the query.

Finally, it returns the **top k most similar products** to the user's profile, excluding any items explicitly marked as "seen" to ensure novelty.

- **`recommend_purchase_based.py`**

  This function, recommend_purchase_based, generates **cross-sell recommendations** by finding products similar to a user's past purchases.

  It filters the user's interactions to **identify their top 3 most relevant purchased items**, based on computed event probabilities.

  The script then uses a **FAISS index** created from product embeddings to efficiently search for items most similar to the embedding of each key purchased product.

  The output is a list of **top *k* unique, similar candidates** derived from the user's purchase history, indicating "purchase" as the source event.

- `Recommend_view_based.py`

  This function, recommend_view_based, generates **content-based recommendations** by leveraging a user's recent product viewing history.

  It first filters the user's interactions to **identify the top 3 most relevant "view" events**, weighting them by recency/frequency.

  For each of these top viewed products, it uses a **FAISS index** built from product embeddings to find the most similar products.

  The function returns a list of **top k unique, similar items**, ensuring the user is not recommended the exact product they just viewed.

- **`Shared_utils.py`**

  This module contains **utility functions for a content-based recommendation system**, managing data loading and core vector operations.

  It handles data persistence by ensuring user interactions from an **SQLite database are exported to a Parquet file** for fast access.

Core functions include **loading product embeddings** from a Parquet file and building a **FAISS IndexFlatIP** for efficient similarity search using dot product (cosine similarity after L2 normalization).

The script's main logic focuses on **computing a dynamic user profile vector** by aggregating the embeddings of interacted products, weighting them by event type and applying a **time-based decay** to prioritize recent activity.

- **static/**
  - **Index.html**

    This is the HTML code for the **front-end interface** of a product recommendation web application.

    It defines the basic structure, including a title, a brief instruction, and an input field where a user can enter a User ID (e.g., C00023). A **"Get Recommendations" button** is provided to trigger the recommendation process.

    The page includes three dynamic elements: a **loader message**, a **results container**, and a **spinner**, all initially hidden, which are used to display the status and output. The page links to external CSS (styles.css) and JavaScript (scripts.js) files to handle styling and user interaction.

  - **scripts.js**

    This JavaScript code implements the **client-side logic** for fetching and displaying product recommendations on a web page.

    It attaches an event listener to the "Get Recommendations" button, triggering the getRecommendations asynchronous function. This function first **validates the user input** and shows a loading spinner.

    It then makes an **asynchronous POST request** to the /api/recommend endpoint with the entered user ID. Upon receiving a response, it hides the spinner and handles any potential API errors.

Finally, the code **iterates through the recommendation data** (including product name, score, source, and LLM-generated explanation) and dynamically updates the results container with structured HTML for display.

- **styles.css**

  This CSS code provides the **styling for the recommendation application's front-end**.

  It centers the content within a limited **white container (.wrapper)** with a shadow effect on a light blue background. Styles are defined for the main heading, the centered **input and button group (.input-group)**, and form elements for a clean look. The code also styles the **recommendation result blocks**, distinguishing the title, metadata, explanation, and evidence. Finally, it includes the CSS and keyframes for a **blue loading spinner** animation.

- **.env**

  The .env file contains the **secret key** used to authenticate your application with Google's Gemini API, which looks like this:

  GEMINI_API_KEY="YOUR API KEY"

- **.gitignore**

  It contains .env to ignore the .env file while committing to GitHub.

- **backend.py**

  This script, backend.py, sets up a **Flask web server** to handle API requests for personalized product recommendations.

  It configures the Flask application, enables CORS, and sets up basic logging. The root route (/) serves the static index.html frontend file.

  The core functionality lies in the /api/recommend POST route, which **takes a user_id**, calls the recommend_for_user master function imported from the local scripts, and returns the results as a JSON response. It includes error handling and logs the recommendation process.

- **README.md**

  This is a comprehensive GitHub README.md for the **Unthinkable** project, a Python-based platform for innovative solutions. It clearly outlines the **project structure** and provides step-by-step instructions for **installation**, including cloning the repository, setting up a virtual environment, and installing dependencies. The guide details how to **configure environment variables** and **run the main backend** script (backend.py). Finally, it includes sections for **future plans**, **contributing**, **licensing**, and **author information**.

- **Requirements.txt**

  This **requirements.txt file** lists all the Python dependencies needed to run the hybrid recommendation system.

  The core packages support:

  1. **Web Framework:** flask and flask-cors for the backend API server.
  2. **LLM/AI:** google-generativeai (for Gemini API) and jsonschema (to validate its output).
  3. **Data Processing:** pandas, numpy, and database tools like SQLAlchemy and sqlite-utils.
  4. **Vector Search:** sentence-transformers, torch, pyarrow, and faiss-cpu for generating product embeddings and performing fast similarity search.

How to test it?

1. Clone the repository

> git clone https://github.com/pathakx/Unthinkable.git

> cd Unthinkable

2. Set up a virtual environment (recommended)

> conda create -n <nanme> pyhton=3.10

3. Install dependencies

> pip install -r requirements.txt

4. Configure environment variables

> GEMINI_API_KEY=your_api_key_here

5. Run the backend:

> python backend.py

Open local host link and enter the Customer ID in between C00001 to C00100