Med Safe

Project Introduction & Description

This project helps patients identify potential drug–drug interactions, enhancing safety without requiring medical knowledge. Interactions are classified as major, moderate, or minor, making them easy to understand. Using a preloaded CSV of 30,465 interaction records, the system checks user-input drug pairs and displays any known interactions. A simple Streamlit interface ensures clarity and accessibility, reflecting real-world gaps in interaction data tools and Technologies.

1. Configuration and Repository Management

- **.gitignore**
  Defines files and directories that should not be tracked by Git, such as cache files and IDE-specific settings. This keeps the repository clean and focused on source code.
- **README.md**
  Provides a high-level overview of the project, including its purpose, setup instructions, and guidance on running the application and tests.
- **.coverage**
  An automatically generated file created during test execution. It stores code coverage data used to assess how much of the codebase is exercised by tests.

2. Development Environment Artifacts

- **.idea/**
  Contains IDE-specific configuration files (for example, PyCharm). These files are not required to run the program and do not affect application logic.
- **pycache/**
  Stores Python bytecode generated during execution. It improves performance but has no relevance to the project's logic or structure.

3. Data Layer

- **drug_interaction.csv**
  The primary data source for the application. It contains structured records of drug–drug interactions and is loaded at runtime. Keeping the data separate from the logic allows updates or replacements without changing the codebase.

4. Core Application Logic

This layer contains the main project-specific functionality.

- **loader.py**
  Handles reading and parsing the CSV file. It converts raw data into structured Python objects that can be safely and consistently processed by other modules. This file acts as the boundary between external data and internal logic.
- **checker.py**
  Implements the core interaction detection logic. It evaluates combinations of drugs and determines whether known interactions exist. This module contains the most critical decision-making logic in the project.
- **search.py**
  Provides search and indexing functionality over the loaded interaction data. It isolates query-related logic and keeps it separate from both the data loading and the user interface.

5. User Interface Layer

- **streamlit.py**
  Implements a simple web-based interface using Streamlit. It connects user input to the underlying logic modules and displays results. No business logic is implemented here; the file strictly coordinates input, processing, and output.

6. Testing Layer

Each core module has an associated test file to ensure correctness and reliability.

- **test_loader.py**
  Confirms that CSV data is loaded and parsed correctly.
- **test_checker.py**
  Validates the interaction detection logic using controlled inputs and expected outputs.
- **test_search.py**
  Ensures that search and filtering functions behave as intended.
- **test_interaction.py**
  Covers edge cases and integration scenarios related to interaction detection.

All tests are isolated, reproducible, and focused on project-specific behavior.

Core Logic Overview

Data Loading – load_interactions (loader.py)

The CSV file containing interaction data is loaded into memory. Each row is converted into a dictionary with standardized keys and normalized drug names (drug_a, drug_b). The final result is a list of dictionaries representing all known interactions.

Interaction Checking – check_interactions (checker.py)

This function identifies interactions among a given list of drugs.

Input

- A list of drug names.
- An interaction index built from the dataset.

Process

- Normalize drug names by trimming whitespace and converting them to lowercase.
- Generate all unique drug pairs using combinations.
- Look up each pair in the interaction index using a frozenset to handle unordered pairs.

Output

- A list of interaction records found for the provided drugs.

Key Characteristics

- Case-insensitive matching.
- Supports any number of drugs.
- Correctly handles unordered drug pairs.

Interaction Index – build_interaction_index (search.py)

- This function converts the list of interaction records into a dictionary optimized for fast lookup.

Process

- Create an empty dictionary.
- For each interaction, form a frozenset of the two drug names.
- Use the frozenset as a key and store the interaction data as the value.

Result

- A dictionary mapping unordered drug pairs to their interaction details.
- This approach allows constant-time lookups instead of scanning the full dataset for each query

Design Decisions and Rationale

- **Dictionary-based indexing**
  Chosen for performance. Lookups are O(1) per drug pair, which is significantly more efficient than iterating over large datasets.
- **Use of frozenset**
  Drug interactions are unordered. A frozenset ensures that "Drug A + Drug B" is treated the same as "Drug B + Drug A."
- **Normalization of input**
  Lowercasing and trimming input avoids false mismatches caused by formatting differences.
- **Clear separation of responsibilities**
  Each function has a single, well-defined role. This improves readability, maintainability, and testability.
- **Scalability**
  Performance depends on the number of drug pairs generated, not on dataset size or external services.

  The Role of LLM in the project

  During the early planning phase, we considered several ways to implement the drug–drug interaction checker. Large Language Models suggested options like using external APIs, prebuilt libraries, or local dictionaries to store interaction data. Each approach has its pros, but after careful evaluation, we decided that a CSV-based dictionary was the best fit for this project.

  Here's why:

  Performance – Many APIs and libraries rely on large datasets or require network requests, which can slow things down. A local dictionary built from a CSV file lets us check any drug pair instantly, without scanning the whole dataset. This keeps the system fast and responsive for users.

  Relevance and Accuracy – External APIs and libraries often include a wide range of interactions, some of which aren't relevant for our goals. By using a curated CSV, we include only verified, necessary drug pairs. This reduces noise and makes the results more reliable for patients.

  Simplicity and Maintainability – APIs come with network errors, rate limits, authentication, and possible changes in endpoints. Libraries can be bulky or include extra features we don't need. A CSV-based dictionary is lightweight, easy to update, and requires no changes to the code or external dependencies.

  Predictability and Stability – External sources can go offline or change unexpectedly, potentially breaking the program. With a local dictionary, behavior is consistent, and we have full control over the data, making testing and verification straightforward.