VIT®
BHOPAL

MEDICAL DISEASE PREDICTOR PROGRAM

Project Documentation & Report

Submitted by: BhaweshKumar Gautam

Reg No: 25BSA10052

Language: Python

Introduction

The **Medical Disease Predictor** is a Python-based command-line interface (CLI) application designed to assist users in identifying potential illnesses based on their current symptoms. By selecting from a curated list of common symptoms, the program analyzes the input against a predefined database of diseases, calculates the probability of a match, and provides a diagnosis along with severity levels, descriptions, and medical advice. This project aims to demonstrate the practical application of logic, data structures, and user input handling in Python.

Problem Statement

In day-to-day life, individuals often experience symptoms but lack immediate knowledge of what condition they might indicate. While professional medical advice is irreplaceable, a quick, logic-based tool can help categorize symptoms and suggest whether a condition is mild (like a cold) or requires more attention. The problem this project solves is the "uncertainty of self-diagnosis" by mapping user inputs to specific disease signatures using a scoring algorithm.

Functional Requirements

The system fulfills the following functional requirements:

- **Symptom Display:** The system must list all 20 available symptoms with unique IDs.
- **User Input:** The system must allow the user to input multiple symptom IDs one by one.
- **Input Validation:** The system must handle invalid inputs (non-integers) and out-of-range numbers (outside 1-20).
- **Matching Algorithm:** The system must compare user symptoms against disease profiles and calculate a percentage match score.
- **Sorting:** Results must be displayed in descending order of relevance (highest match first).
- **Result Display:** The system must output the Disease Name, Match %, Severity, Description, and Advice.
- **Session Management:** The user must be able to restart the program or exit after a diagnosis.

Non-functional Requirements

- **Usability:** The interface must be text-based and easy to read with clear separation lines (=, -).
- **Performance:** Diagnosis calculations should be instantaneous (O(N) complexity).
- **Robustness:** The program should not crash if the user types letters instead of numbers.

- **Portability:** The script must run on any machine with a standard Python 3.x interpreter.

System Architecture

The system follows a **Monolithic Procedural Architecture**. It does not rely on external databases or APIs. All data is stored in-memory using Python lists and dictionaries.

Flow:

Input Module -> Processing Unit (Matching & Sorting) -> Output Module

Design Diagrams

A. Use Case Diagram

- **Actor:** User
- **Use Cases:** View Symptoms, Enter IDs, View Diagnosis, Restart Program.

B. Workflow Diagram

1. **Start**
2. Display Symptoms List.
3. **Loop:** Ask for Symptom ID.
    a. If 0: Break Loop.
    b. If valid: Add to list.
    c. If invalid: Show error.
4. **Process:** Iterate through Diseases -> Count Matches -> Calculate Score.
5. **Sort:** Arrange results by Score (Bubble Sort).
6. **Output:** Print Diagnosis details.
7. **End:** Ask to Restart or Exit.

C. Sequence Diagram

User -> System: Request Start

System -> User: Show Symptoms List

User -> System: Input ID 1, ID 2... ID 0

System -> System: Validate Input

System -> System: Calculate Match %

System -> System: Sort Results

System -> User: Display Top Results

User -> System: Exit Command

D. ER Diagram (Data Structure Schema)

Since no external database is used, this represents the internal data structure:

- **Symptom Entity:** Attributes {ID, Name}
- **Disease Entity:** Attributes {Name, ID_List, Severity, Description, Tips}
- **Relationship:** A Disease **contains** multiple Symptom IDs.

Design Decisions & Rationale

- **Data Structure (List of Dictionaries):** Chosen for both Symptoms and Diseases. This allows structured data storage that is easy to iterate over and readable within the code.
- **Algorithm (Weighted Matching):** A percentage-based match (matches / total_symptoms * 100) was chosen instead of a simple binary "yes/no" to handle cases where a user might have only 3 out of 5 symptoms of the Flu.
- **Manual Sorting:** A Bubble Sort algorithm was implemented manually (for i in range(n)...) rather than using Python's built-in .sort() to demonstrate understanding of algorithmic logic.
- **Input Loop:** A while True loop with a try-except block was chosen to ensure the program is robust against invalid user inputs (e.g., typing "abc").

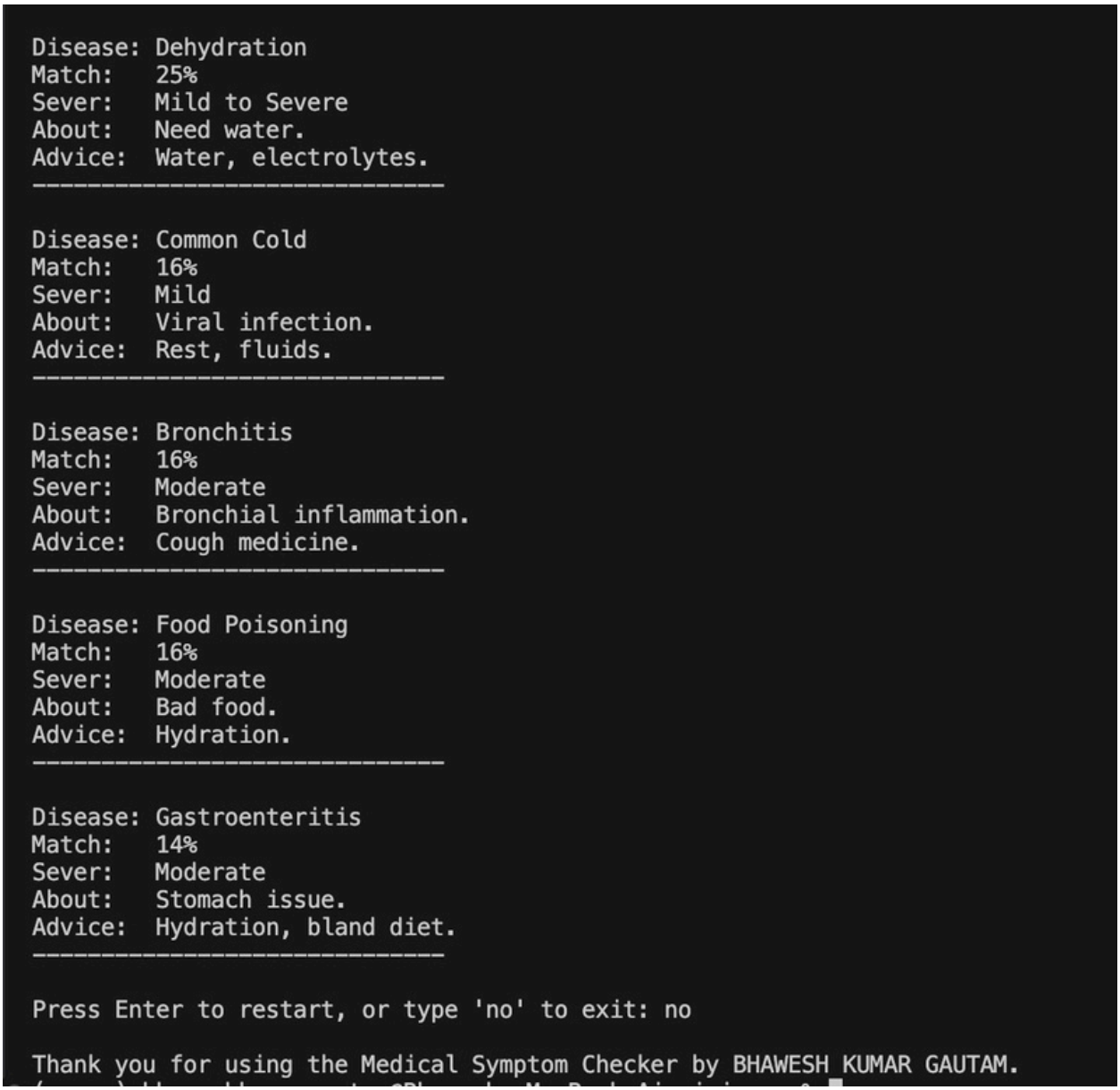
Implementation Details

- **Language:** Python 3.x
- **Core Logic:**
    - **Input:** Uses input() loop.

- **Storage:** symptoms_list stores mappings. diseases stores complex objects.
- **Calculation:** Score = (Matches Found / Total Symptoms in Disease) * 100

- **Libraries Used:** Standard library only (no pandas or numpy required), ensuring lightweight execution.

Screenshots / Results



Testing Approach

The system was tested using **Black Box Testing** methods:

1. **Positive Testing:** Entered IDs 1, 2, 3 (Flu symptoms) -> Verified that "Influenza" appeared with high probability.
2. **Negative Testing:** Entered -5, 100, and abc -> Verified that the system caught the error and prompted for input again without crashing.
3. **Boundary Testing:** Entered 0 immediately -> Verified system handles empty input gracefully.

4. **Logic Testing:** Verified that diseases with higher match percentages appeared at the top of the list.
5.

Challenges Faced

1. **Handling Non-Integer Inputs:** Initially, the program crashed if a user typed text. This was solved by implementing a try-except ValueError block.
2. **Sorting Logic:** Implementing the sorting logic manually to ensure the disease with the highest probability shows up first required careful index management in the nested loops.
3. **Symptom Overlap:** Many diseases share symptoms (e.g., Fever). The challenge was creating a dataset distinct enough to give meaningful results.
4.

Learnings & Key Takeaways

- **Data Structures:** Gained deep understanding of how to structure complex data using nested dictionaries and lists in Python.
- **Algorithms:** Learned how to implement a basic scoring algorithm and a sorting algorithm (Bubble Sort) from scratch.
- **User Experience:** Understood the importance of handling user errors gracefully to prevent program crashes.
- **Logic Building:** Improved capability to translate real-world medical logic into conditional code statements.

Future Enhancements

1. **GUI Implementation:** Upgrade the text-based interface to a Graphical User Interface using Tkinter or PyQt.
2. **Database Integration:** Move data from the code to an SQL database (SQLite) to allow adding new diseases without changing the source code.
3. **API Integration:** Connect to a live medical API to get real-time data on disease outbreaks.
4. **Machine Learning:** Implement a Decision Tree Classifier (using Scikit-Learn) for more accurate predictions based on vast datasets.

References

1. **Python Documentation:** https://docs.python.org/3/
2. **Medical Data Reference:** Mayo Clinic (for symptom-disease mapping).
3. **Algorithm Reference:** "Introduction to Algorithms" for sorting logic.