**TESTING**

Testing is crucial to ensure the system behaves as expected and meets the user requirements. The following testing strategies cover both **unit testing** (for functions) and **functional testing** (for the user interface and the system's overall behavior).

**1. Unit Testing**

The core logic for symptom matching and data handling can be tested using Python’s built-in `unittest` module.

**Unit Test File: `test\_health\_diagnosis.py`**

import unittest

from io import StringIO

import csv

from health\_diagnosis\_system import load\_health\_data, match\_symptoms

# Sample data for testing

TEST\_CSV\_DATA = """symptom,diagnosis,treatment,causes

fever,Infection,Rest; Hydration; Antipyretics,Viral infection; Bacterial infection

cough,Common Cold,Rest; Fluids; Cough suppressant,Viral infection; Allergies; Smoking

headache,Migraine,Pain relievers; Rest; Dark room,Stress; Hormonal changes; Food triggers

"""

class TestHealthDiagnosisSystem(unittest.TestCase):

def setUp(self):

"""Sets up mock CSV data for testing."""

self.csv\_data = StringIO(TEST\_CSV\_DATA)

self.health\_data = {}

reader = csv.DictReader(self.csv\_data)

for row in reader:

symptom = row['symptom'].lower().strip()

diagnosis = row['diagnosis'].strip()

treatment = row['treatment'].strip()

causes = row['causes'].strip()

self.health\_data[symptom] = {

'diagnosis': diagnosis,

'treatment': treatment,

'causes': causes

}

def test\_load\_health\_data(self):

"""Test that the data is loaded correctly from CSV."""

test\_data = load\_health\_data('health\_data.csv')

# Ensure data is not empty

self.assertIsNotNone(test\_data)

self.assertIn('fever', test\_data)

self.assertEqual(test\_data['fever']['diagnosis'], 'Infection')

def test\_match\_single\_symptom(self):

"""Test symptom matching with a single valid symptom."""

symptoms\_input = "fever"

result = match\_symptoms(self.health\_data, symptoms\_input)

self.assertEqual(result[0]['symptom'], 'fever')

self.assertEqual(result[0]['diagnosis'], 'Infection')

def test\_match\_multiple\_symptoms(self):

"""Test symptom matching with multiple valid symptoms."""

symptoms\_input = "fever, cough"

results = match\_symptoms(self.health\_data, symptoms\_input)

self.assertEqual(len(results), 2)

self.assertEqual(results[0]['diagnosis'], 'Infection')

self.assertEqual(results[1]['diagnosis'], 'Common Cold')

def test\_invalid\_symptom(self):

"""Test matching for an invalid or unknown symptom."""

symptoms\_input = "unknown\_symptom"

result = match\_symptoms(self.health\_data, symptoms\_input)

self.assertEqual(result[0]['diagnosis'], 'Unknown')

def test\_mixed\_valid\_invalid\_symptoms(self):

"""Test matching with a combination of valid and invalid symptoms."""

symptoms\_input = "fever, unknown\_symptom"

results = match\_symptoms(self.health\_data, symptoms\_input)

self.assertEqual(len(results), 2)

self.assertEqual(results[0]['diagnosis'], 'Infection')

self.assertEqual(results[1]['diagnosis'], 'Unknown')

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Explanation of Unit Tests:**

**1. `test\_load\_health\_data()`:**

- This tests that the CSV data is correctly loaded into the program, checking if the expected values for symptoms like "fever" are present.

**2. `test\_match\_single\_symptom()`:**

- Ensures that a single valid symptom (like "fever") is matched correctly, and the corresponding diagnosis is correct.

3. **`test\_match\_multiple\_symptoms()`:**

- Tests if multiple symptoms (e.g., "fever" and "cough") are correctly matched with the appropriate diagnoses.

**4. `test\_invalid\_symptom()`:**

- Checks if an invalid or unknown symptom is handled correctly, returning "Unknown" for the diagnosis.

**5. `test\_mixed\_valid\_invalid\_symptoms()`:**

- Ensures that when both valid and invalid symptoms are entered, the valid ones return the correct diagnoses while the invalid ones are flagged as "Unknown".

**2. Functional Testing**

Functional testing ensures the GUI behaves as expected. These tests can be done manually or automated using libraries like `pyautogui` or `unittest` along with `tkinter`.

**Test Cases for the GUI:**

| Test Case ID | Description | Steps | Expected Result |

|--------------|-------------|-------|-----------------|

| TC1 | Submit valid single symptom | 1. Start the application. <br> 2. Enter "fever" in the input field. <br> 3. Click "Submit". | The output area shows the correct diagnosis (Infection), treatment (Rest, Hydration), and causes (Viral infection, Bacterial infection). |

| TC2 | Submit multiple valid symptoms | 1. Start the application. <br> 2. Enter "fever, cough" in the input field. <br> 3. Click "Submit". | The output area shows the correct diagnoses and treatments for both symptoms. |

| TC3 | Submit invalid symptom | 1. Start the application. <br> 2. Enter "xyz" in the input field. <br> 3. Click "Submit". | The system displays "Unknown" as the diagnosis with "No information available" for treatment and causes. |

| TC4 | Submit valid and invalid symptoms | 1. Start the application. <br> 2. Enter "fever, xyz" in the input field. <br> 3. Click "Submit". | "Fever" returns the correct diagnosis, treatment, and causes, while "xyz" shows "Unknown". |

| TC5 | Clear button functionality | 1. Start the application. <br> 2. Enter "fever" in the input field. <br> 3. Click "Submit". <br> 4. Click "Clear". | The input field and output area are cleared. |

| TC6 | Submit without input | 1. Start the application. <br> 2. Leave the input field empty. <br> 3. Click "Submit". | An error message should pop up saying "Please enter at least one symptom". |

**Manual Functional Testing Example:**

**1. Test Case: Submit valid single symptom.**

- Steps:

1. Open the application.

2. Enter "fever" in the input field.

3. Click the "Submit" button.

- Expected Outcome:

- The diagnosis should display "Infection".

- Treatment should show "Rest, Hydration, Antipyretics".

- Causes should show "Viral infection, Bacterial infection".

**2. Test Case: Submit multiple valid symptoms.**

- Steps:

1. Open the application.

2. Enter "fever, cough" in the input field.

3. Click the "Submit" button.

- Expected Outcome:

- "Fever" should return "Infection", and "Cough" should return "Common Cold".

- Corresponding treatments and causes should be displayed for both symptoms.

**3. Automated GUI Testing**

Automated GUI testing can be performed using Python libraries like `pyautogui`, `unittest`, and `tkinter`:

Example with `unittest` (Note: Testing GUIs can be tricky and might need frameworks specialized in UI testing):

import unittest

from tkinter import Tk

from health\_diagnosis\_system import load\_health\_data, match\_symptoms, submit\_symptoms

class TestHealthDiagnosisGUI(unittest.TestCase):

def setUp(self):

self.root = Tk()

self.health\_data = load\_health\_data('health\_data.csv')

def test\_gui\_submit(self):

# Simulate entering a symptom

input\_field.insert(0, "fever")

# Call the submit function

submit\_symptoms()

# Check the output field to see if the diagnosis for "fever" is correct

result\_text = output\_textbox.get("1.0", "end")

self.assertIn("Diagnosis: Infection", result\_text)

def tearDown(self):

self.root.destroy()

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**4. Edge Case Testing**

- Test empty input: Ensure that submitting an empty input field prompts an error message.

- Test non-string input: Ensure that numeric or special characters as input prompt appropriate handling (e.g., ignored or flagged as invalid).

- Test large input: Test for robustness by inputting a large number of symptoms, including valid and invalid ones, to see how the system handles it.

By combining **unit testing** for the backend logic and **functional testing** for the GUI and overall behavior, you can ensure that the system performs reliably across different inputs and scenarios. Automated tests can be run consistently to maintain the system as new features or data are added.