HEALTHAI: INTELLIGENT HEALTHCARE ASSISTANT USING IBM GRANET

MENTOR: K. RATNA KUMARI MADAM

TEAM ID: LTVIP2025TMID32712

TEAM LEADER: *ADAPA DURGA BHAVANI*

Roll number: SBAP0032712 durgabhavani2469.adapa@gmail.com

TEAM MEMBERS:

1)REKHA DEVI

Roll number: SBAP0032720 devi5530750@gmail.com

2)BANDI PHANI SIVA SAI DEEPIKA RAMYANJALI

Roll number: SBAP0032724 phanianjali066@gmail.com

3)NIDADAVOLU CHILOCHANA LAKSHMI BHAVANI

Roll number: SBAP0032728 bhavaninidadhavolunidadhavolu@gmail.com

4)MADIMI VANDANA

Roll number: SBAP0032732 vandanamadimi@gmail.com

5)CHAVVAKULA SIRISHA

Roll number: SBAP0032749 sirishasiri4237@gmail.com

Project Title: Health AI – Generative AI with IBM Cloud

Track: Generative AI with IBM Cloud

Project Description:

Health AI is a Generative AI-powered web application that provides:

Symptom Identification: Users input symptoms, and the AI predicts possible diseases.

Home Remedies Suggestion: Users input a disease name, and the AI returns natural home remedies.

The app integrates the IBM Granite model (granite-3-3-2b-instruct) from Hugging Face and is deployed using FastAPI, HTML & CSS on Google Colab, optionally using Gradio for fast prototyping.

IBM Cloud: Model deployment & hosting

Hugging Face: Granite-3-3-2b-instruct model

FastAPI: Web backend framework

HTML/CSS: Frontend

Gradio or Streamlit: For UI (optional)

Google Colab: Development environment

Python: Programming Language

Functionalities:

1. Symptoms Identifier

Input: User enters symptoms (text)

Process: Granite model processes symptoms and suggests probable diseases.

Output: List of possible diseases

2. Home Remedies Suggestion

Input: User enters disease name

Process: AI model generates home remedies based on the disease

Output: Display natural, AI-generated home remedies

Architecture Overview:

yaml

Copy

Edit

User (Web UI)

|

| (HTML form / Gradio)

v

FastAPI (Python backend)

|

| (Prompt Construction)

v

IBM Granite Model (via Hugging Face API or local model)

|

v

AI-generated Response
|
v
Display in Web UI
Example Prompt to Model:
For Symptoms Identifier:
Given the following symptoms: "fever, cough, fatigue", predict the most likely diseases.
For Home Remedies:
Provide home remedies for the disease: "Common Cold".
Sample Code (FastAPI + Gradio)
Here’s a basic starter template for the FastAPI backend with Gradio:

```
from fastapi import FastAPI
import gradio as gr
from transformers import pipeline

# Load the model
model = pipeline("text-generation", model="ibm/granite-3-3b-instruct")

app = FastAPI()

def predict_disease(symptoms):
prompt = f"Given the symptoms: {symptoms}, predict the disease."
response = model(prompt, max_length=100)[0]['generated_text']
return response.strip()

def get_remedy(disease):
prompt = f"Suggest natural home remedies for: {disease}"
response = model(prompt, max_length=100)[0]['generated_text']
return response.strip()

demo = gr.Interface(
fn=predict_disease,
inputs="text",
outputs="text",
title="Symptom Identifier"
)

demo2 = gr.Interface(
fn=get_remedy,
inputs="text",
outputs="text",
title="Home Remedy Generator"
```

)

app = gr.mount_gradio_app(app, demo, path="/symptoms")
app = gr.mount_gradio_app(app, demo2, path="/remedies")
🚀 Deployment Suggestions:
Locally or Google Colab: Use Gradio with public sharing

IBM Cloud App Deployment: Use IBM Code Engine or Container Registry for FastAPI app

Optional: Use Docker for containerization

Folder Structure
cpp
Copy
Edit
HealthAI/
├── app/
│ ├── main.py
│ ├── model.py
│ └── prompts.py
├── static/
│ └── style.css
├── templates/
│ └── index.html
├── requirements.txt
└── README.md
Use Cases:
Educational AI projects

Early-stage health support tools

AI-based chatbot integration for medical help

Project Description:

HealthAI harnesses IBM Watson Machine Learning and Generative AI to provide intelligent healthcare assistance, offering users accurate medical insights. The platform includes a Patient Chat for answering health-related questions, Disease Prediction that evaluates user-reported symptoms to deliver potential condition details, Treatment Plans that provide personalized medical recommendations, and Health Analytics to visualize and monitor patient health metrics.

Utilizing IBM's Granite-13b-instruct-v2 model, HealthAI processes user inputs to deliver personalized and data-driven medical guidance, improving accessibility to healthcare information. Built with Streamlit and powered by IBM Watson, the platform ensures a seamless and user-friendly experience. With secure API key management and responsible data handling, HealthAI empowers users to make informed health decisions with confidence.

Scenarios:

Scenario 1: A user inputs their symptoms into the Disease Prediction system, describing issues like persistent headache, fatigue, and mild fever. The system analyzes the symptoms along with the patient's profile and health data to provide potential condition predictions, including likelihood assessments and recommended next steps.

Scenario 2: A user needs personalized treatment recommendations for a diagnosed condition. By entering their condition in the Treatment Plans generator, the AI processes the information along with patient data to create a comprehensive, evidence-based treatment plan that includes medications, lifestyle modifications, and follow-up testing.

Scenario 3: A user wants insights about their health trends. Using the Health Analytics dashboard, they can visualize their vital signs over time (heart rate, blood pressure, blood glucose, etc.) and receive AI-generated insights about potential health concerns and improvement recommendations.

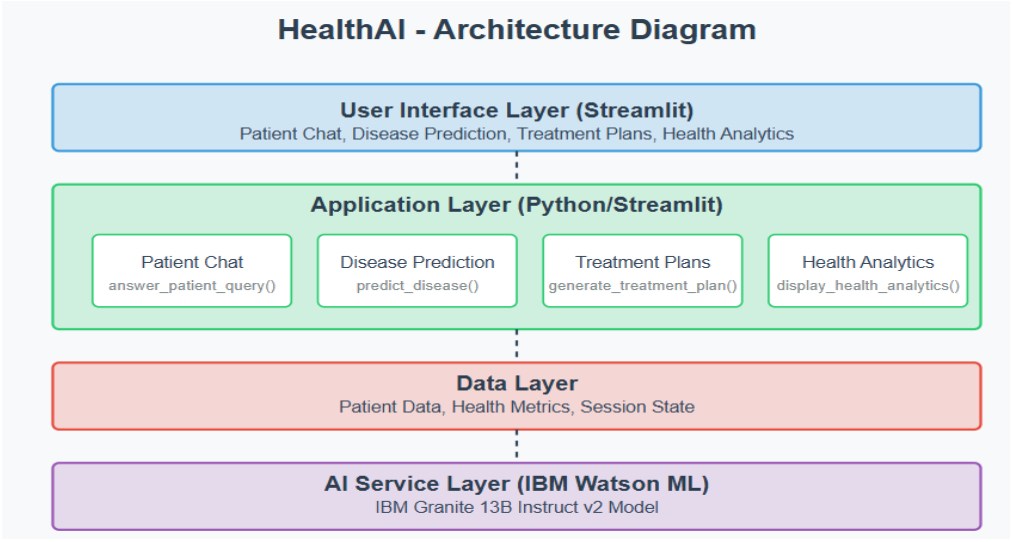
Scenario 4: A user has a health-related question. Through the Patient Chat interface, they can ask any medical query and receive a clear, empathetic response that includes relevant medical facts, acknowledges limitations, and suggests when to seek professional medical advice.

```
from fastapi import FastAPI, Request, Form
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from transformers import pipeline
import uvicorn

# Load Granite model pipeline (assume available via Hugging Face)
model = pipeline("text-generation", model="IBM/granite-3-3b-instruct")

app = FastAPI()
```

```
# Static files and templates
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
async def homepage(request: Request):
return templates.TemplateResponse("index.html", {"request": request})

@app.post("/predict")
async def predict(request: Request, user_input: str = Form(...), input_type: str =
Form(...)):
if input_type == "symptoms":
prompt = f"Given the symptoms: {user_input}, predict the disease."
else:
prompt = f"Suggest natural home remedies for: {user_input}"

result = model(prompt, max_length=100)[0]['generated_text']
return templates.TemplateResponse("index.html", {
"request": request,
"user_input": user_input,
"input_type": input_type,
"output": result
})

# To run the app: uvicorn main:app --reload

# requirements.txt should include:
# fastapi
# uvicorn
# transformers
# jinja2
from fastapi import FastAPI, Request, Form
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
from transformers import pipeline
import uvicorn

# Load Granite model pipeline (assume available via Hugging Face)
model = pipeline("text-generation", model="ibm/granite-3-3b-instruct")

app = FastAPI()

# Static files and templates
```

```
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
async def homepage(request: Request):
return templates.TemplateResponse("index.html", {"request": request})

@app.post("/predict")
async def predict(request: Request, user_input: str = Form(...), input_type: str =
Form(...)):
if input_type == "symptoms":
prompt = f"Given the symptoms: {user_input}, predict the disease."
else:
prompt = f"Suggest natural home remedies for: {user_input}"

result = model(prompt, max_length=100)[0]['generated_text']
return templates.TemplateResponse("index.html", {
"request": request,
"user_input": user_input,
"input_type": input_type,
"output": result
})

# To run the app: uvicorn main:app –reload
# requirements.txt should include:
# fastapi
# uvicorn
# transformers
# jinja2
```

Technical Architecture:



Pre-requisites:

1. **Streamlit Framework Knowledge:** Streamlit Documentation
2. **IBM Watson Machine Learning:** IBM Watson ML Documentation
3. **Python Programming Proficiency:** Python Documentation
4. **Data Visualization Libraries:** Plotly Documentation
5. **Version Control with Git:** Git Documentation

Development Environment Setup: Flask Installation Guide

project workflow

Activity 1: Model Selection and Architecture

- Activity 1.1: Research and select the appropriate AI model from IBM Watson for medical assistance (IBM Granite 13B Instruct v2).
- Activity 1.2: Define the architecture of the application, detailing interactions between the frontend, backend, and AI integration.
- Activity 1.3: Set up the development environment, installing necessary libraries and dependencies for Streamlit and IBM Watson ML.

Activity 2: Core Functionalities Development

- Activity 2.1: Develop the core functionalities: Patient Chat, Disease Prediction, Treatment Plan Generation, and Health Analytics.
- Activity 2.2: Implement patient data utilities to manage and visualize health metrics.

Activity 3: App.py Development

- Activity 3.1: Write the main application logic in app.py, establishing functions for each feature and integrating AI responses.
- Activity 3.2: Create prompting strategies for the IBM Granite model to generate high-quality medical content.

Activity 4: Frontend Development

- Activity 4.1: Design and develop the user interface using Streamlit components, ensuring a responsive and intuitive layout.
- Activity 4.2: Create dynamic visualizations with Plotly to display health metrics and trends.

Activity 5: Deployment

- Activity 5.1: Prepare the application for deployment by configuring environment variables for API credentials.

- Activity 5.2: Deploy the application on a suitable hosting platform to make it accessible to users.

Milestone 1: Model Selection and Architecture

- In this milestone, we focus on selecting the appropriate AI model from IBM Watson for our medical assistance needs. This involves researching the capabilities and performance of various models, ensuring that the chosen model aligns well with our application's objectives of creating a Patient Chat system, Disease Prediction, Treatment Plan Generation, and Health Analytics.

Research and select the appropriate AI model

1. Understand the Project Requirements: Review the specific needs of the healthcare application.

2. Explore IBM Watson ML Documentation: Examine the various models available, including their functionalities and limitations.

3. Select the Optimal Model: Choose IBM's Granite 13B Instruct v2 model for its strong performance with healthcare-related content.

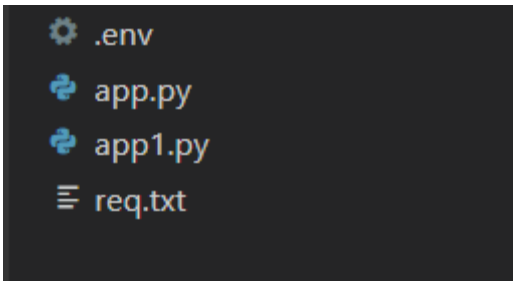
Activity 1.2: Define the architecture of the application

1. Draft an Architectural Diagram: Create a visual representation of the application architecture.
2. Detail Frontend Functionality: Outline how users will interact with the application through a Streamlit interface.
3. Outline Backend Responsibilities: Specify how the backend will process user input and communicate with the IBM Granite model.
4. Describe AI Integration Points: Define how the application will make API calls to IBM Watson ML.

Activity 1.3: Set up the development environment

1. Install Python and Pip: Ensure Python is installed along with pip for managing dependencies.
2. Create a Virtual Environment: Set up a virtual environment to isolate project dependencies.
3. Install Required Libraries:
   pip install streamlit pandas numpy plotly ibm-watson-machine-learning python-dotenv

4. Set Up Application Structure: Create the initial directory structure for the HealthAI application.



Milestone 2: Core Functionalities Development

This Milestone contains about Core Functionalities Development.

Develop core functionalities

Activity 2.1: Develop core functionalities

1. Patient Chat System:
    - Implement conversational interface for answering health questions
    - Create prompting system for the IBM Granite model to provide medical advice
    - Develop session-based chat history management
2. Disease Prediction System:
    - Create symptom input interface
    - Develop prediction function using patient data and reported symptoms
    - Structure output format to show potential conditions with likelihood and next steps
3. Treatment Plan Generator:
    - Build input interface for condition and patient details
    - Create prompting system for personalized treatment plans
    - Structure output to include medications, lifestyle changes, and follow-up care
4. Health Analytics Dashboard:
    - Implement patient data visualization with interactive charts
    - Create metrics summary with trend indicators
    - Develop AI-generated insights based on health trends

Activity 2.2: Implement data management utilities

1. Patient Data Generation:
    - Create sample data with realistic health metrics

- Implement date-based trend generation for visualization
- Structure data for efficient analysis and display

2. Patient Profile Management:
    - Develop interface for managing patient details
    - Create session state handling for persistent data
    - Implement profile update functionality
- and return JSON responses.

2. Integrate the Gemini AI Responses in Each Function
    - Implement function calls within each route to leverage Gemini's models based on user-provided details.
    - Ensure each response from Gemini API is processed in a way that enhances readability and displays clearly to the use.
    - Implement profile update functionality

3. App.py Development

4. In this activity App.py Development is being explained.

Write the main application logic

Activity 3.1: Write the main application logic

The app.py file is organized into several key sections:\

1. Imports and Setup:
    - Import necessary libraries (Streamlit, pandas, plotly, IBM Watson ML)
    - Load environment variables for API keys
    - Initialize IBM Granite model connection
2. Core Functions:
    - init_granite_model(): Set up connection to IBM Watson ML
    - predict_disease(): Analyze symptoms for potential diagnoses
    - generate_treatment_plan(): Create personalized treatment recommendations
    - answer_patient_query(): Process health questions with AI responses
3. UI Components:
    - Main application layout with sidebar navigation
    - Tab-based interface for different features
    - Custom CSS styling for enhanced user experience

4. Feature Implementation:

   - display_patient_chat(): Chatbot interface for health questions
   - display_disease_prediction(): Symptom analysis system
   - display_treatment_plans(): Treatment plan generator
   - display_health_analytics(): Interactive health dashboard

Activity 3.2: Create prompting strategies

Patient Query Prompting:

```
def answer_patient_query(query):
    """Use IBM Granite to answer patient health questions"""
    model = init_granite_model()

    # Create prompt for answering patient query
    query_prompt = f"""
    As a healthcare AI assistant, provide a helpful, accurate, and evidence-based response to the following patient question:

    PATIENT QUESTION: {query}

    Provide a clear, empathetic response that:
    - Directly addresses the question
    - Includes relevant medical facts
    - Acknowledges limitations (when appropriate)
    - Suggests when to seek professional medical advice
    - Avoids making definitive diagnoses
    - Uses accessible, non-technical language

    RESPONSE:
    """

    answer = model.generate_text(prompt=query_prompt)
    return answer
```

Disease Prediction Prompting:

```
prediction_prompt = f"""
As a medical AI assistant, predict potential health conditions based on the following patient data:

Current Symptoms: {symptoms}
Age: {age}
Gender: {gender}
Medical History: {medical_history}

Recent Health Metrics:
- Average Heart Rate: {avg_heart_rate} bpm
- Average Blood Pressure: {avg_bp_systolic}/{avg_bp_diastolic} mmHg
- Average Blood Glucose: {avg_glucose} mg/dL
- Recently Reported Symptoms: {recent_symptoms}

Format your response as:
1. Potential condition name
2. Likelihood (High/Medium/Low)
3. Brief explanation
4. Recommended next steps

Provide the top 3 most likely conditions based on the data provided.
"""

prediction = model.generate_text(prompt=prediction_prompt)
return prediction
```

Treatment Plan Prompting:

```
treatment_prompt = f"""
As a medical AI assistant, generate a personalized treatment plan for the following scenario:

Patient Profile:
- Condition: {condition}
- Age: {age}
- Gender: {gender}
- Medical History: {medical_history}

Create a comprehensive, evidence-based treatment plan that includes:
1. Recommended medications (include dosage guidelines if appropriate)
2. Lifestyle modifications
3. Follow-up testing and monitoring
4. Dietary recommendations
5. Physical activity guidelines
6. Mental health considerations

Format this as a clear, structured treatment plan that follows current medical guidelines while being personalized to this patient's specific needs.
"""

treatment_plan = model.generate_text(prompt=treatment_prompt)
return treatment_plan
```

Design and develop the user interface

Activity 4.1: Design and develop the user interface

1. Main Application Layout:
    - Configure page title, icon, and layout preferences
    - Implement a sidebar for patient profiles and feature selection
    - Create custom CSS for enhanced visual appearance
2. Feature-Specific Interfaces:
    - Patient Chat: Chat-style interface with message history
    - Disease Prediction: Symptom input form and prediction display
    - Treatment Plans: Condition input and treatment plan output
    - Health Analytics: Interactive charts and metrics summary

Activity 4.2: Create dynamic visualizations

1. Health Metric Charts:
    - Heart rate trend line chart
    - Blood pressure dual-line chart
    - Blood glucose trend line chart with reference line
    - Symptom frequency pie chart
2. Metrics Summary:
    - Key health indicators with trend deltas
    - Color-coded metrics to indicate normal/abnormal ranges
    - Interactive tooltip information

Design and develop the user interface

Activity 4.1: Design and develop the user interface

1. Main Application Layout:

- Configure page title, icon, and layout preferences
- Implement a sidebar for patient profiles and feature selection
- Create custom CSS for enhanced visual appearance

2. Feature-Specific Interfaces:
    - Patient Chat: Chat-style interface with message history
    - Disease Prediction: Symptom input form and prediction display
    - Treatment Plans: Condition input and treatment plan output
    - Health Analytics: Interactive charts and metrics summary

Activity 4.2: Create dynamic visualizations

1. Health Metric Charts:
    - Heart rate trend line chart
    - Blood pressure dual-line chart
    - Blood glucose trend line chart with reference line
    - Symptom frequency pie chart
2. Metrics Summary:
    - Key health indicators with trend deltas
    - Color-coded metrics to indicate normal/abnormal ranges
    - Interactive tooltip information

3. Deployment

4. In this milestone the Deployment topic is being explained.

Prepare for deployment

Activity 5.1: Prepare for deployment

1. Environment Variable Configuration:

Create a .env file for IBM Watson API credentials:
WATSONX_API_KEY=your_api_key_here

- WATSONX_PROJECT_ID=your_project_id_here
- Implement secure loading of credential

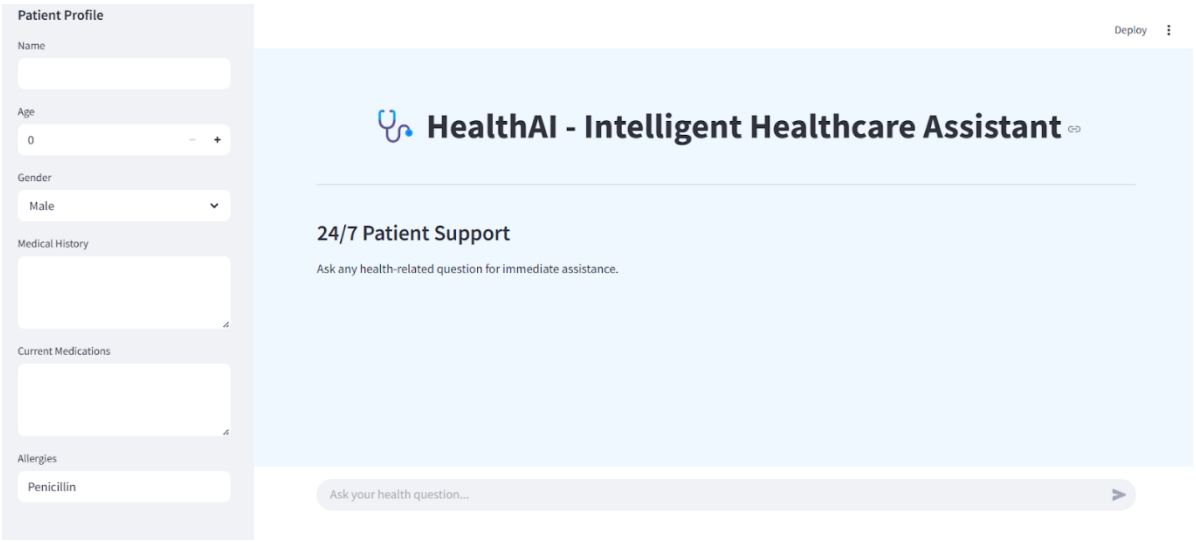
2. Dependency Management:
    - Create requirements.txt file with all necessary packages
    - Document installation process for deployment

Activity 5.2: Deploy the application

1. Local Deployment Testing:
    - Run the application using streamlit run app.py
    - Test all features for functionality
    - Verify responsive design and performance
2. Cloud Deployment Options:
    - Deploy on Streamlit Cloud for public access
    - Configure environment variables in the deployment platform
    - Set up monitoring and error logging

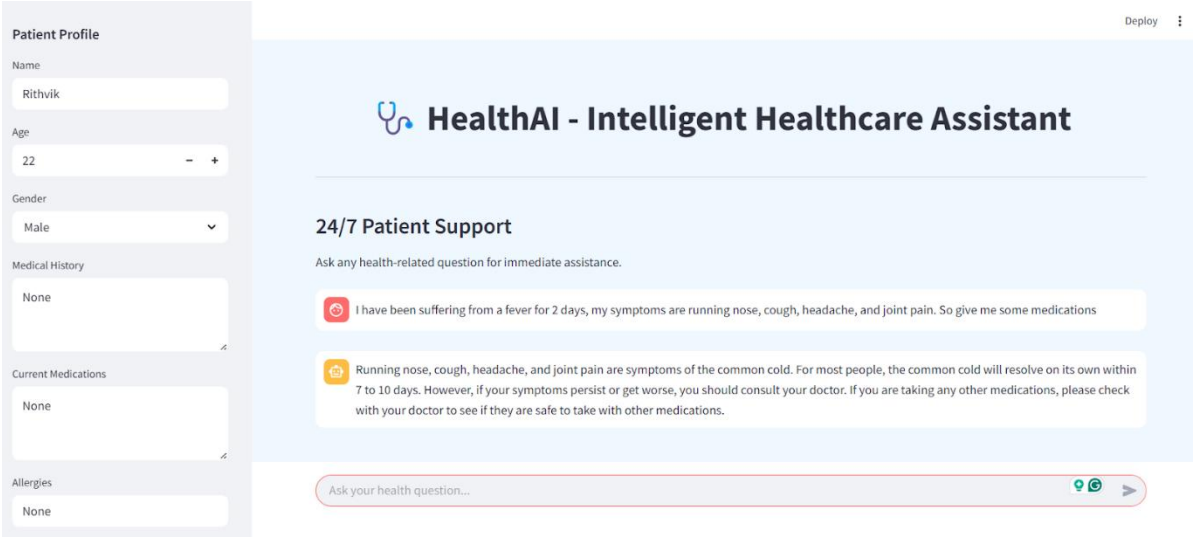
Exploring Website's Web Pages:

Patient Chat Page

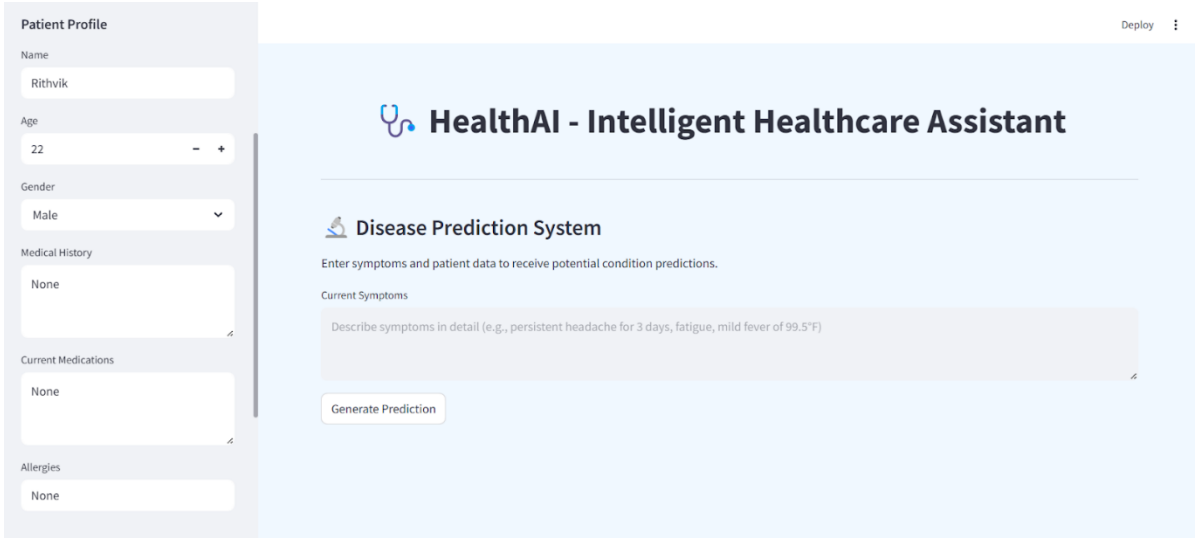


Description: Here user have access to responsive healthcare communication platform enabling seamless dialogue about wellness concerns, with chronological message tracking for context retention. The system delivers intelligent, algorithmically-generated wellness guidance while providing verifiable medical insights supported by authoritative healthcare sources, creating a comprehensive virtual consultation experience.

Patient Chat output:

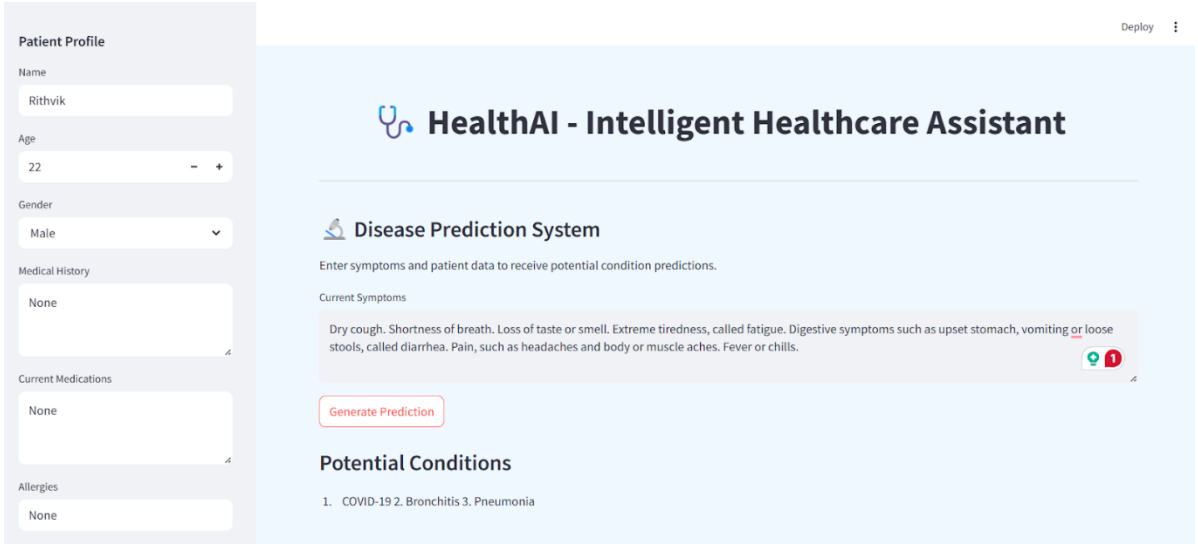


Disease Prediction Page:

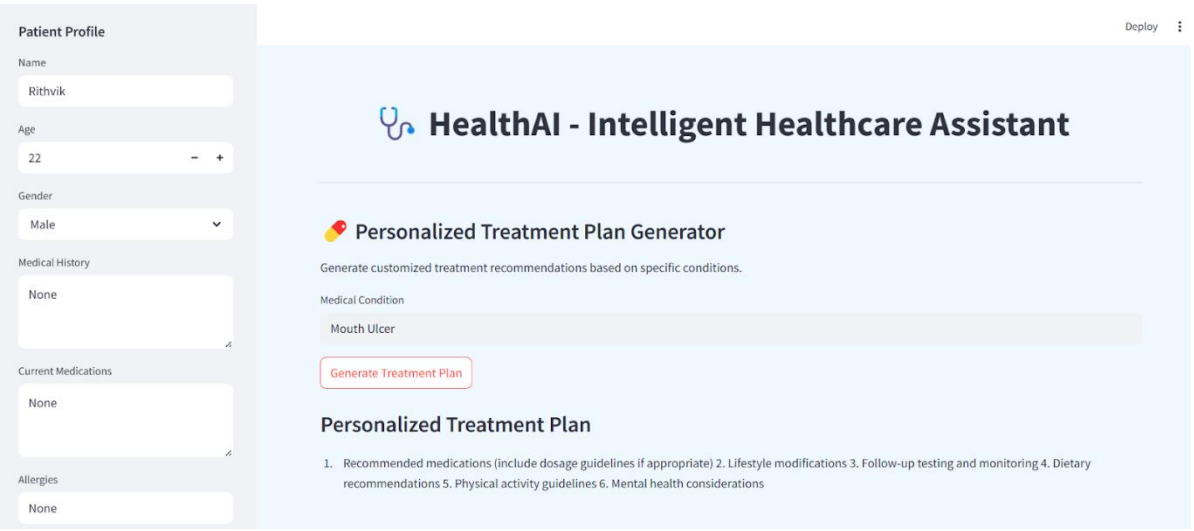


Description: The webpage presents sophisticated diagnostic assessment tool featuring an intuitive symptom documentation section where users can detail their health concerns in depth. The interface includes a prominent analysis initiation control that, when activated, processes entered data to generate a comprehensive analysis of possible medical conditions. Results are presented with statistical probability indicators and detailed explanatory content for each potential diagnosis. The system incorporates essential medical advisories emphasizing the informational nature of the analysis and recommending professional healthcare consultation.

Disease Prediction Output:

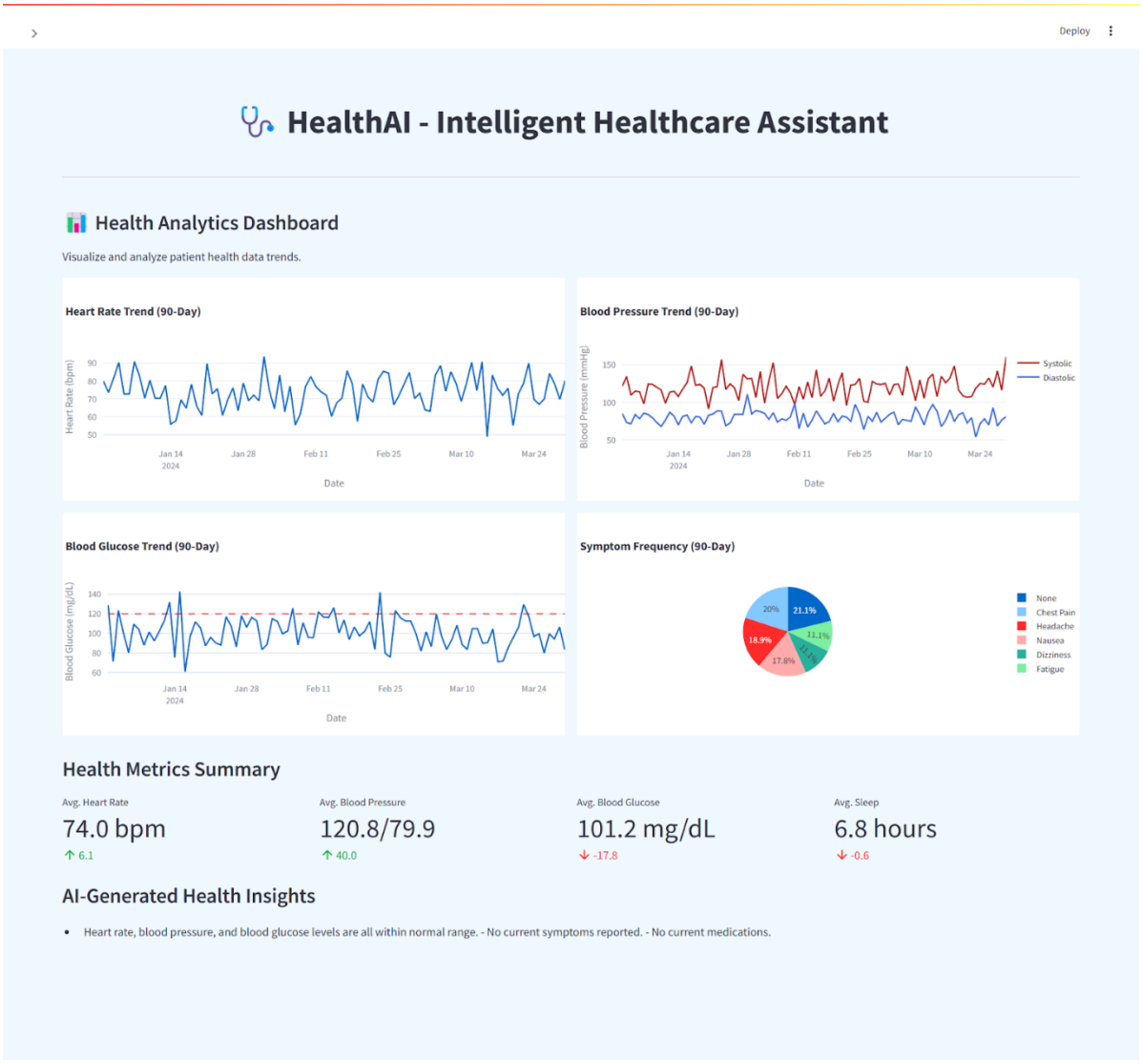


Treatment Plans Page:



Description: This page contains medical condition entry area where users can specify their health concerns, complemented by a prominent action button for plan creation. Upon activation, the system produces comprehensive, organized therapeutic recommendations tailored to the specified condition. The interface includes essential medical advisories clarifying the informational nature of the provided guidance.

Health Analytics Dashboard:



Description: The interface features dynamic visualizations including heart rate trends, blood pressure patterns, and blood glucose fluctuations over time. A color-coded pie chart illustrates symptom occurrence frequency, while the metrics summary section provides key health indicators with directional trend markers. The dashboard is enhanced with AI-powered insights that analyze collected data to offer personalized health recommendations and observations.

Conclusion

The HealthAI project effectively demonstrates the potential of AI in revolutionizing healthcare assistance. By integrating IBM's Granite language model, the platform enables users to receive personalized health insights through Patient Chat, Disease Prediction, Treatment Plan Generation, and Health Analytics, making healthcare information more accessible.

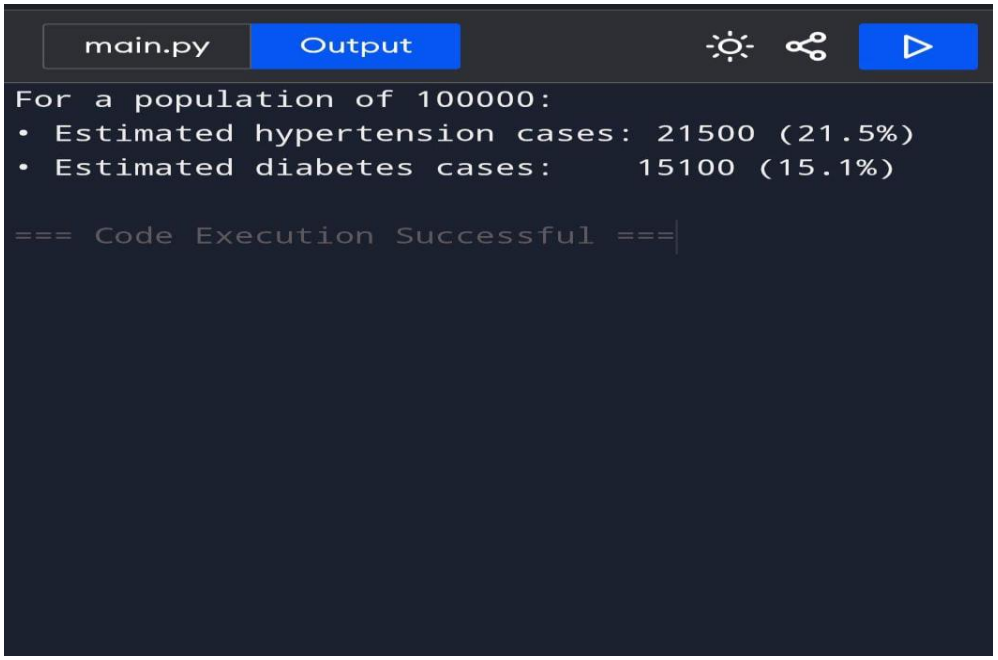
Utilizing IBM Watson Machine Learning, the application ensures accurate health question answering, detailed disease prediction, personalized treatment recommendations, and insightful health trend analysis. The structured development process—spanning model selection, core feature implementation, backend and frontend development, and deployment—led to the creation of an interactive, user-friendly platform.

Built with Streamlit, HealthAI facilitates seamless visualization of health data and AI-generated insights, ensuring an efficient and responsive experience. This project highlights how targeted AI models and a well-structured framework can enhance healthcare accessibility. With future scalability in mind, HealthAI has the potential to expand its capabilities, incorporating more advanced diagnostics and broader medical applications.

```
# Sample data based on published studies

total_population = 100_000  # example total population size


# Prevalence rates for Andhra Pradesh

hypertension_prevalence = 0.215  # 21.5% – rural Andhra Pradesh
:contentReference[oaicite:1]{index=1}

diabetes_prevalence = 0.151     # 15.1% – urban Andhra Pradesh
:contentReference[oaicite:2]{index=2}


# Compute estimated counts
```

```
hypertension_count = total_population * hypertension_prevalence

diabetes_count = total_population * diabetes_prevalence


# Compute percentages

hypertension_pct = hypertension_prevalence * 100

diabetes_pct = diabetes_prevalence * 100


print(f"For a population of {total_population}:")

print(f"• Estimated hypertension cases: {hypertension_count:.0f}
({hypertension_pct:.1f}%)")

print(f"• Estimated diabetes cases:    {diabetes_count:.0f} ({diabetes_pct:.1f}%)")
```

output:-




```
# blood_pressure_dashboard.py

import pandas as pd

import matplotlib.pyplot as plt

from io import BytesIO

import base64


def generate_blood_pressure_chart():

# Sample blood pressure data
```

```
data = {
'District': ['Visakhapatnam', 'Vijayawada', 'Guntur', 'Nellore', 'Kurnool',
'Tirupati', 'Anantapur', 'Kadapa', 'Rajahmundry', 'Kakinada'],
'Low BP': [75, 70, 78, 65, 72, 76, 68, 70, 79, 66],
'High BP': [130, 135, 120, 138, 125, 132, 115, 140, 128, 127]
}

df = pd.DataFrame(data)

# Create the plot
plt.figure(figsize=(12, 6))
plt.bar(df['District'], df['Low BP'], width=0.4, align='center', color='#4b8bbe', label='Low BP')
plt.bar(df['District'], df['High BP'], width=0.4, align='edge', color='#e74c3c', label='High BP')

plt.title('Blood Pressure Comparison Across Districts', pad=20)
plt.xlabel('Districts', labelpad=10)
plt.ylabel('Blood Pressure (mmHg)', labelpad=10)
plt.xticks(rotation=45, ha='right')
plt.legend()
plt.tight_layout()

# Save to buffer and convert to base64
buffer = BytesIO()
plt.savefig(buffer, format='png', dpi=100)
plt.close()
buffer.seek(0)
return base64.b64encode(buffer.read()).decode('utf-8')

# Generate HTML with embedded chart
chart_image = generate_blood_pressure_chart()
```

```
html_template = f"""
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Blood Pressure Dashboard</title>
<script src="https://cdn.tailwindcss.com"></script>
<style>
.chart-container {{
background: white;
border-radius: 12px;
box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
padding: 24px;
margin: 20px auto;
max-width: 1000px;
}}
.header {{
background: linear-gradient(135deg, #3498db, #2c3e50);
color: white;
padding: 20px 0;
margin-bottom: 30px;
border-radius: 0 0 12px 12px;
}}
</style>
</head>
<body class="bg-gray-50">
<div class="header text-center">
<h1 class="text-3xl font-bold">Andhra Pradesh Blood Pressure Analysis</h1>
<p class="mt-2">District-wise comparison of blood pressure metrics</p>
```

```
</div>

<div class="chart-container">
<h2 class="text-xl font-semibold mb-4">Systolic vs Diastolic Pressure</h2>
<img src="{chart_image}" alt="Blood Pressure Comparison Chart" class="w-full">

<div class="mt-6">
<h3 class="text-lg font-medium mb-2">Key Observations</h3>
<ul class="list-disc pl-5 space-y-1">
<li>The highest systolic pressure was recorded in Kadapa (140 mmHg)</li>
<li>The lowest diastolic pressure was in Nellore (65 mmHg)</li>
<li>Average difference between systolic and diastolic pressure is approximately 55 mmHg</li>
</ul>
</div>
</div>

<div class="text-center mt-8 text-gray-500 text-sm">
<p>Data collected from statewide health surveys (2024)</p>
</div>
</body>
</html>
"""

# Save the HTML file
with open('blood_pressure_dashboard.html', 'w') as f:
f.write(html_template)

print("Dashboard generated successfully: blood_pressure_dashboard.html")
```

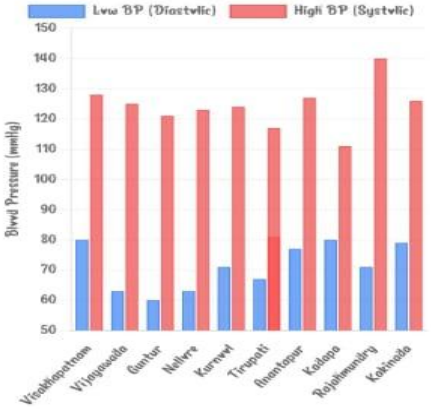
Andhra Pradesh Blvvd Pressure Analysis

Cvmparative visualizativn vf systvlic and diastvlic blvvd pressure acrvss districts

District-wise Blvvd Pressure Cvmparisvn

Lvw BP (Diastvlic)

High BP (Systvlic)



Highest Systvlic BP

140 mmHg

Rajahmundry

Lvwest Diastvlic BP

60 mmHg

Guntur

Average Pulse Pressure

53 mmHg

Difference between systvlic and diastvlic