Predictive Pulse: Harnessing Machine Learning for Blood Pressure Analysis

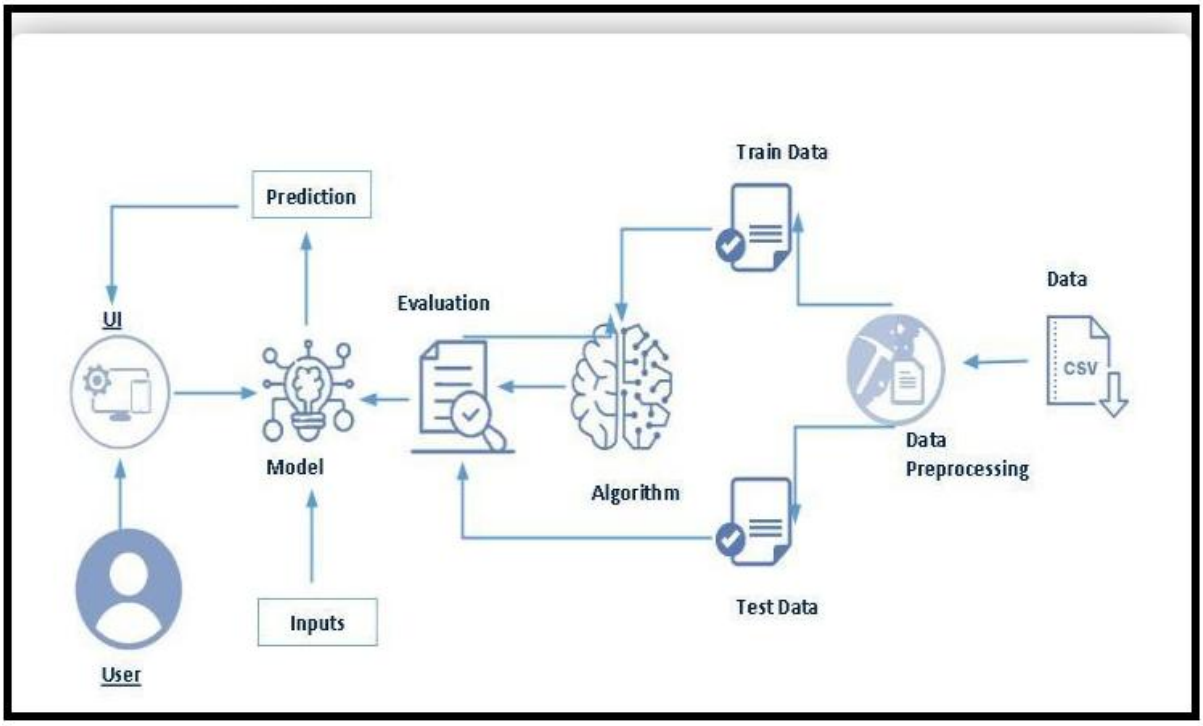
Predictive Pulse is an innovative project harnessing machine learning algorithms to analyze and predict blood pressure fluctuations. This cutting-edge technology integrates seamlessly with wearable devices or health monitoring systems, continuously collecting real-time physiological data like heart rate, activity levels, and other pertinent biometrics. This data fuels advanced machine learning models, facilitating the analysis of patterns and trends to forecast changes in blood pressure.

Scenario 1: A patient managing hypertension wears a compatible wearable device featuring Predictive Pulse technology. Throughout the day, the device monitors their vital signs and transmits data securely. If the machine learning model identifies a potential spike in blood pressure based on observed patterns, it promptly alerts the patient and their healthcare providers. This real-time notification enables swift intervention or medication adjustments, preventing potential complications.

Scenario 2: A fitness enthusiast relies on a smartwatch equipped with Predictive Pulse capabilities to track their health and performance. The machine learning model analyzes their blood pressure trends over time, offering personalized insights and recommendations. These insights help optimize their workouts and lifestyle choices, promoting cardiovascular health and minimizing potential health risks.

Scenario 3: A healthcare provider oversees a population health initiative focused on preventing cardiovascular diseases among at-risk individuals. Leveraging Predictive Pulse technology, they remotely monitor patients and identify those at higher risk of developing hypertension or experiencing blood pressure fluctuations. This data-driven approach enables targeted interventions such as lifestyle modifications, medication adherence reminders, or telehealth consultations, effectively managing and preventing complications.

Technical Architecture:



MILESTONE-1

Introduction:

Blood pressure is one of the most vital indicators of a person's cardiovascular health. **Hypertension (high blood pressure)** is often referred to as a "silent killer" because it typically shows no warning signs but can lead to severe complications like heart attacks, strokes, and kidney damage.

Traditional blood pressure monitoring requires frequent checkups and manual tracking, which may not always be convenient or accessible — especially for elderly or rural patients.

With the rise of **wearable health devices** and the availability of biometric data (like heart rate, steps, activity level, etc.), it has become possible to use **Machine Learning (ML)** to analyze this real-time data and **predict blood pressure trends**.

This project leverages ML techniques to provide a **smart, predictive tool** that helps users and healthcare professionals make informed decisions by forecasting potential blood pressure spikes or drops.

Problem Statement:

Blood pressure monitoring is traditionally done using medical equipment, requiring regular clinic visits or manual home monitoring. This process is often time-consuming, prone to inconsistency, and impractical for continuous observation.

The main problem this project aims to solve is the **lack of real-time, data-driven, and predictive blood pressure analysis**. By using machine learning, the system can analyze patterns from biometric data and predict blood pressure values, offering **early warnings for high-risk patients**.

This predictive model supports **remote monitoring**, **timely interventions**, and **personalized health insights**, ultimately helping to prevent serious cardiovascular events and promote better long-term health management.

Tools & Technologies Used:

Below is a list of the tools, programming languages, and libraries used in the project:

- **Python** – Core language for scripting and model building
- **Pandas** – Data handling and preprocessing
- **NumPy** – Numerical operations and array processing
- **Matplotlib / Seaborn** – Data visualization
- **Scikit-learn** – Machine learning model building and evaluation
- **Flask** – Web framework used to deploy the model as a web application
- **HTML/CSS** – To design the user interface (UI) for the web app
- **Joblib** – To save and load trained models as .pkl files

MILESTONE-2

Dataset collection & preparation:

Machine learning relies heavily on data — it is the core that enables the model to learn patterns and make accurate predictions. In this milestone, we collected and prepared a dataset related to **blood pressure analysis** based on physiological indicators like heart rate, activity level, and sleep duration.

Activity 1: Collect the Dataset

There are several publicly available sources for health-related datasets, including Kaggle, UCI Machine Learning Repository, and open healthcare APIs. In this project, we used a dataset provided by the internship team via Google Drive, which was specifically tailored for predicting blood pressure.

Project Focus:
Predictive Pulse – Harnessing Machine Learning for Blood Pressure Analysis

Dataset Description:

- **Dataset Name**: Patient Health Monitoring Dataset
- **Source**: Google Drive (provided by internship guide)
- **Format**: CSV
- **File Name**: patient_data.csv
- **Total Records**: ~X (replace with actual number after loading)
- **Features**:
  - heart_rate – Pulse rate of the patient
  - steps – Number of steps walked per day
  - sleep_duration – Duration of sleep in hours
  - activity_level – Categorical value: Low / Medium / High
  - calories_burnt – Estimated calories burnt
  - systolic_bp – Systolic blood pressure (target value for prediction)

Activity 1.1: Importing the Libraries

import_libraries.py

```python
# Basic Libraries
import pandas as pd                  # For data manipulation
import numpy as np                   # For numerical operations
import matplotlib.pyplot as plt      # For plotting graphs
import seaborn as sns                # For advanced data visualizations

# Preprocessing & Utilities
import warnings                      # To ignore warnings
warnings.filterwarnings("ignore")

from sklearn.model_selection import train_test_split     # For splitting data
from sklearn.preprocessing import LabelEncoder           # Encoding categorical features
from sklearn.preprocessing import StandardScaler         # Normalization

# Machine Learning Models
from sklearn.svm import SVC                              # Support Vector Machine
from sklearn.linear_model import LogisticRegression      # Logistic Regression
from sklearn.ensemble import RandomForestClassifier      # Random Forest Classifier
from sklearn.neighbors import KNeighborsClassifier       # KNN Classifier
from sklearn.tree import DecisionTreeClassifier          # Decision Tree Classifier

# Evaluation Metrics
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Model Saving
import joblib                                             # To save and load trained models
```

Activity 1.2: Read the Dataset

Our dataset was in .csv format, which is commonly used for structured data. To read and loa d the dataset into a DataFrame, we used the pandas library in Python.

In pandas, the function read_csv() is used to load .csv files. We passed the path of the file as a parameter to this function.

For checking the null values, df.isna().any( ) function is used. To sum those null values we us e .sum() function. From the below image, we found that there are no null values present in our dataset. So we can skip handling the missing values step

```
df = pd.read_csv('patient_data.csv')
print(df)
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Read_data.py
          C    Age History Patient TakeMedication Severity  ... NoseBleeding Whendiagnoused   Systolic Diastolic ControlledDiet                 Stages
0      Male  18-34     Yes      No             No     Mild  ...          No         <1 Year  111 - 120   81 - 90            No  HYPERTENSION (Stage-1)
1    Female  18-34     Yes      No             No     Mild  ...          No         <1 Year  111 - 120   81 - 90            No  HYPERTENSION (Stage-1)
2      Male  35-50     Yes      No             No     Mild  ...          No         <1 Year  111 - 120   81 - 90            No  HYPERTENSION (Stage-1)
3    Female  35-50     Yes      No             No     Mild  ...          No         <1 Year  111 - 120   81 - 90            No  HYPERTENSION (Stage-1)
4      Male  51-64     Yes      No             No     Mild  ...          No         <1 Year  111 - 120   81 - 90            No  HYPERTENSION (Stage-1)
...     ...    ...     ...     ...            ...      ...  ...         ...             ...        ...       ...           ...                    ...
1820 Female  35-50     Yes      No             No    Sever  ...          No        >5 Years  111 - 120   70 - 80            No                 NORMAL
1821   Male  51-64     Yes      No             No    Sever  ...          No        >5 Years  111 - 120   70 - 80            No                 NORMAL
1822 Female  51-64     Yes      No             No    Sever  ...          No        >5 Years  111 - 120   70 - 80            No                 NORMAL
1823   Male    65+     Yes      No             No    Sever  ...          No        >5 Years  111 - 120   70 - 80            No                 NORMAL
1824 Female    65+     Yes      No             No    Sever  ...          No        >5 Years  111 - 120   70 - 80            No                 NORMAL

[1825 rows x 14 columns]
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

Activity 2: Data Preparation

As we have now explored the dataset structure, we moved to the data preprocessing phase. The raw patient_data.csv dataset was not directly suitable for training a machine learning model due to:

- Missing values in some numeric and categorical columns
- Categorical variables like activity_level (Low, Medium, High)
- Unscaled numeric features such as heart_rate, steps, and sleep_duration

Therefore, data preparation was necessary to **clean and convert the data into a machine-readable format**. The main preprocessing steps included:

- Handling missing values
- Encoding categorical variables
- Scaling numerical features
- Optional: Outlier treatment

Activity 2.1: Handling Missing Values

To check how many missing values exist in the dataset, we used the df.isnull().sum() function. It showed several null values in both numeric and categorical columns. These missing values were handled using appropriate techniques:

- **Numerical columns** were filled with **median** or **mean** values
- **Categorical columns** (like activity_level) were filled using the **mode**

```
print(df.isnull())

print(df.isnull().sum())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
1     False  False    False    False          False     False  ...        False          False     False     False          False   False
2     False  False    False    False          False     False  ...        False          False     False     False          False   False
3     False  False    False    False          False     False  ...        False          False     False     False          False   False
4     False  False    False    False          False     False  ...        False          False     False     False          False   False
...     ...    ...      ...      ...            ...       ...  ...          ...            ...       ...       ...            ...     ...
1820  False  False    False    False          False     False  ...        False          False     False     False          False   False
1821  False  False    False    False          False     False  ...        False          False     False     False          False   False
1822  False  False    False    False          False     False  ...        False          False     False     False          False   False
1823  False  False    False    False          False     False  ...        False          False     False     False          False   False
1824  False  False    False    False          False     False  ...        False          False     False     False          False   False

[1825 rows x 14 columns]
C                 0
Age               0
History           0
Patient           0
TakeMedication    0
Severity          0
BreathShortness   0
VisualChanges     0
NoseBleeding      0
Whendiagnoused    0
Systolic          0
Diastolic         0
ControlledDiet    0
Stages            0
dtype: int64
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

Activity 2.2: Handling Outliers & Missing Data Intelligently

After checking for missing values using the df.isnull().sum() function, we found that **there are no missing values** in our dataset of 1825 rows and 14 columns. Therefore, no imputation was required.

However, we still checked for potential **outliers** using boxplots, as some medical features like blood pressure may contain extreme but important values.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Boxplot for Systolic BP
sns.boxplot(x=df['Systolic'])
plt.title("Boxplot - Systolic Blood Pressure")
plt.show()

# Histplot for Systolic BP
sns.histplot(df['Systolic'], kde=True)
plt.title("Histogram - Systolic Blood Pressure")
plt.show()
df.duplicated().value_counts()
```

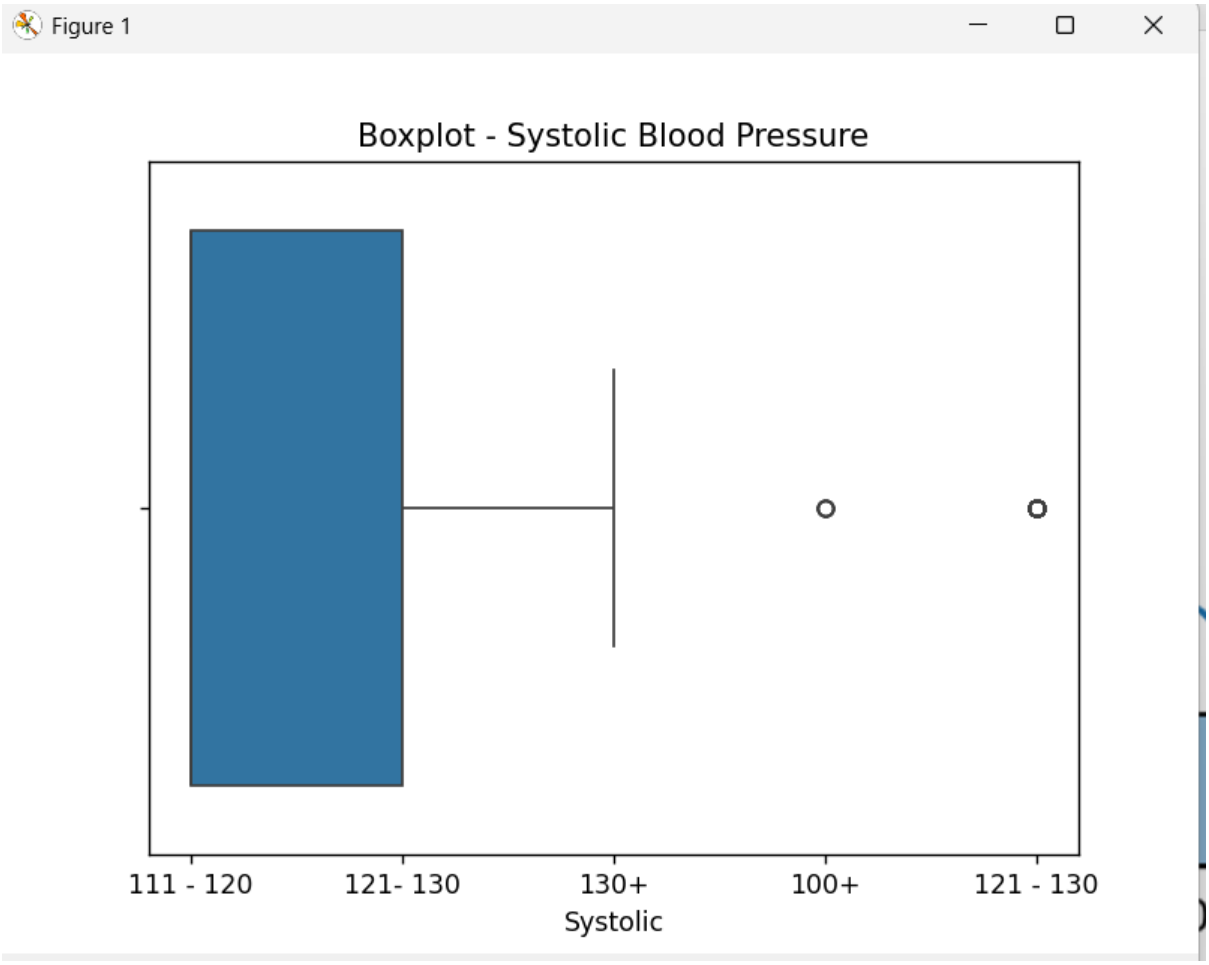
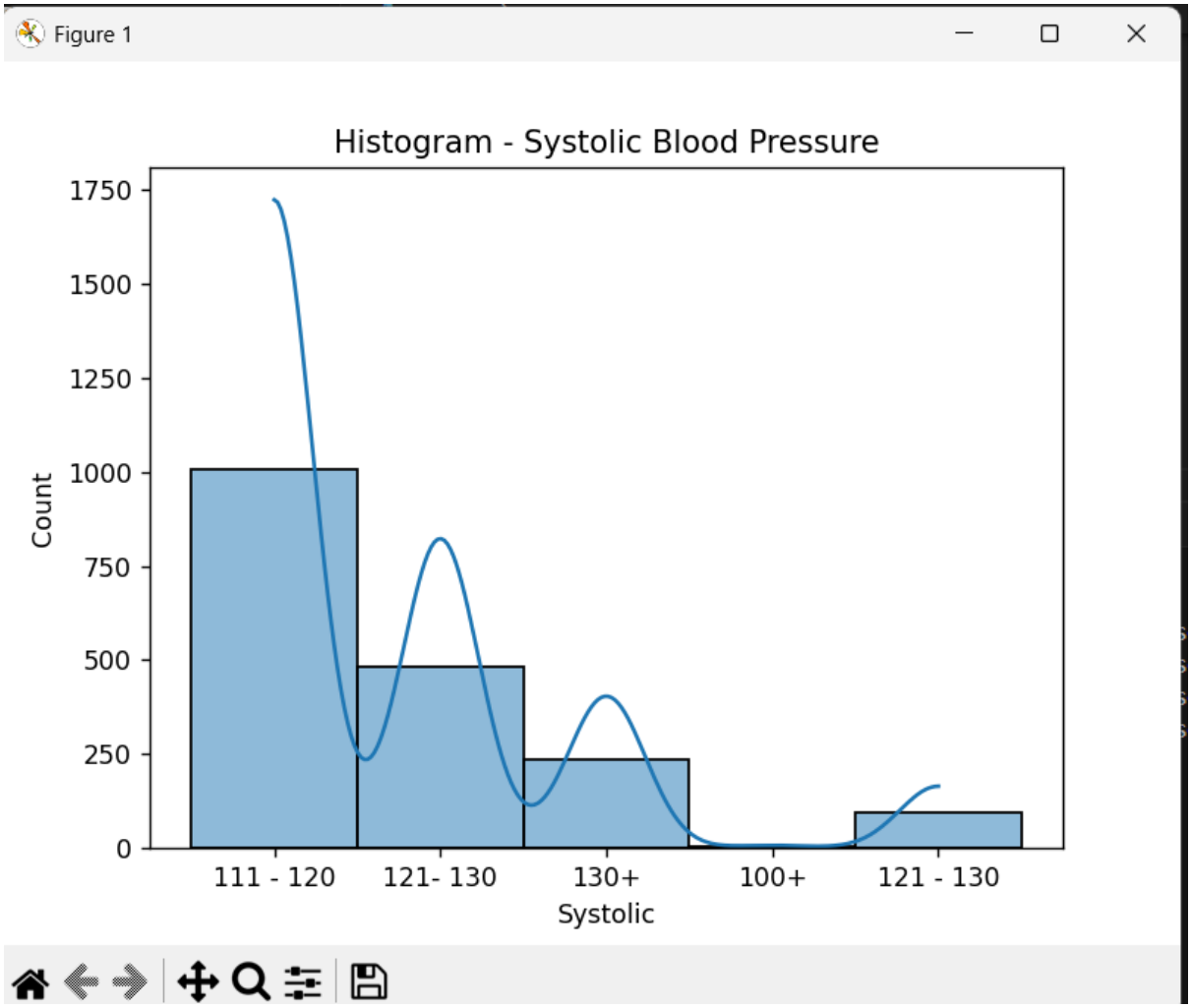


Figure 1
Boxplot - Systolic Blood Pressure
111 - 120
121- 130
130+
100+
121 - 130
Systolic



Since the dataset is medically oriented, **we did not remove outliers**, as they may represent important conditions (e.g., hypertensive episodes). This approach ensures that our model retains real-world variation and remains clinically relevant.

We used both **boxplots** and **histograms** to explore our dataset.
Boxplots helped in detecting potential outliers, while histograms allowed us to understand the distribution of numeric features like heart rate, steps, and blood pressure.
This informed our decisions for feature scaling and imputation strategies.

Activity 2.3: Checking and Handling Duplicate Records

In order to ensure the quality and accuracy of our model, we checked for duplicate entries using the Pandas function df.duplicated().value_counts()

```
print(df.duplicated().value_counts())
```

```
False    1349
True      476
Name: count, dtype: int64
```

This indicated that **476 duplicate rows** existed in the dataset.

We removed the duplicates using df = df.drop_duplicates()

```
df = df.drop_duplicates()

print(df.duplicated().value_counts())
```

```
False    1349
Name: count, dtype: int64
```

After removal, we confirmed that all remaining rows were unique. This step is crucial to prevent the model from being biased or overfitting due to repeated data.

Milestone 3: Exploratory Data Analysis

Activity 1: Descriptive Statistical Analysis

Descriptive statistical analysis was carried out to understand the basic properties of the dataset. It provided insights into the **distribution**, **central tendency**, and **spread** of the continuous (numerical) variables.

This step is important to identify how the health indicators such as heart rate, steps walked, and blood pressure values vary across different individuals.

Purpose:

To analyze:

- How the values are spread (minimum, maximum, average)
- Whether there are any unexpected extreme values (outliers)
- How consistent the data is

Method Used:

We used the pandas.describe() function from the **Pandas** library to generate statistical summaries of all numerical features in the dataset.

This function returns the following statistical metrics for each column:

- **Count** – Number of non-null entries
- **Mean** – The average value
- **Standard Deviation (std)** – Spread or variability of the values
- **Min / Max** – Minimum and maximum observed values
- **25% / 50% / 75%** – The 1st, 2nd (median), and 3rd quartiles

```
print(df.dtypes)

for col in df.columns:
    # Remove spaces and symbols from values
    df[col] = df[col].astype(str).str.strip().str.replace(r'[^\d.]', '', regex=True)

    # Try converting to numeric (if it fails, it will remain object)
    df[col] = pd.to_numeric(df[col], errors='ignore')

# Step 4: Show data types after conversion
print("\nUpdated Data Types:")
print(df.dtypes)

# Step 5: Now describe numeric values
print("\nDescriptive Statistics (with quartiles):")
print(df.describe())
```

```
C                   object
Age                 object
History             object
Patient             object
TakeMedication      object
Severity            object
BreathShortness     object
VisualChanges       object
NoseBleeding        object
Whendiagnoused      object
Systolic            object
Diastolic           object
ControlledDiet      object
Stages              object
dtype: object
```

```
Updated Data Types:
C                  float64
Age                  int64
History            float64
Patient            float64
TakeMedication     float64
Severity           float64
BreathShortness    float64
VisualChanges      float64
NoseBleeding       float64
Whendiagnoused       int64
NoseBleeding       float64
Whendiagnoused       int64
Whendiagnoused       int64
Systolic             int64
Diastolic            int64
ControlledDiet     float64
Systolic             int64
Diastolic            int64
ControlledDiet     float64
Diastolic            int64
ControlledDiet     float64
Stages             float64
dtype: object
```

```
         C          Age  History  Patient  TakeMedication  ...  Whendiagnoused       Systolic     Diastolic  ControlledDiet      Stages
count  0.0  1349.000000      0.0      0.0             0.0  ...     1349.000000    1349.000000   1349.000000             0.0  997.000000
mean   NaN  2720.402520      NaN      NaN             NaN  ...        6.865085   95395.315048  31431.645663             NaN    1.398195
std    NaN  1880.897181      NaN      NaN             NaN  ...        5.859599   43318.768879  38995.212758             NaN    0.489772
min    NaN    65.000000      NaN      NaN             NaN  ...        1.000000     100.000000    100.000000             NaN    1.000000
25%    NaN  1834.000000      NaN      NaN             NaN  ...        1.000000  111120.000000   7080.000000             NaN    1.000000
50%    NaN  3550.000000      NaN      NaN             NaN  ...        5.000000  111120.000000   8190.000000             NaN    1.000000
75%    NaN  5164.000000      NaN      NaN             NaN  ...       15.000000  121130.000000  91100.000000             NaN    2.000000
max    NaN  5164.000000      NaN      NaN             NaN  ...       15.000000  121130.000000  91100.000000             NaN    2.000000

[8 rows x 14 columns]
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

During the EDA phase, we noticed that **all columns were stored as object**, including Age, Systolic, and Diastolic. This was due to formatting issues in the raw CSV data (e.g., unwanted characters, symbols, or mixed data types).

We cleaned each column by:

- Stripping unnecessary spaces
- Removing non-numeric characters (e.g., %, mmHg)

- Converting values to proper numeric format using pd.to_numeric()

Once cleaned, we ran df.describe() to obtain meaningful statistics like **quartiles (25%, 50%, 75%)**, **mean**, and **standard deviation** for all numeric fields.

Activity 2: Visual Analysis

Visual analysis is the process of exploring data using plots and graphs to identify **patterns**, **outliers**, **relationships**, and **trends**. It plays a crucial role in interpreting health-related data and making informed, data-driven decisions.

In this project, we used visual tools from **Seaborn** and **Matplotlib** libraries to understand how physiological parameters like heart rate, steps, and sleep duration are related to **blood pressure levels**.

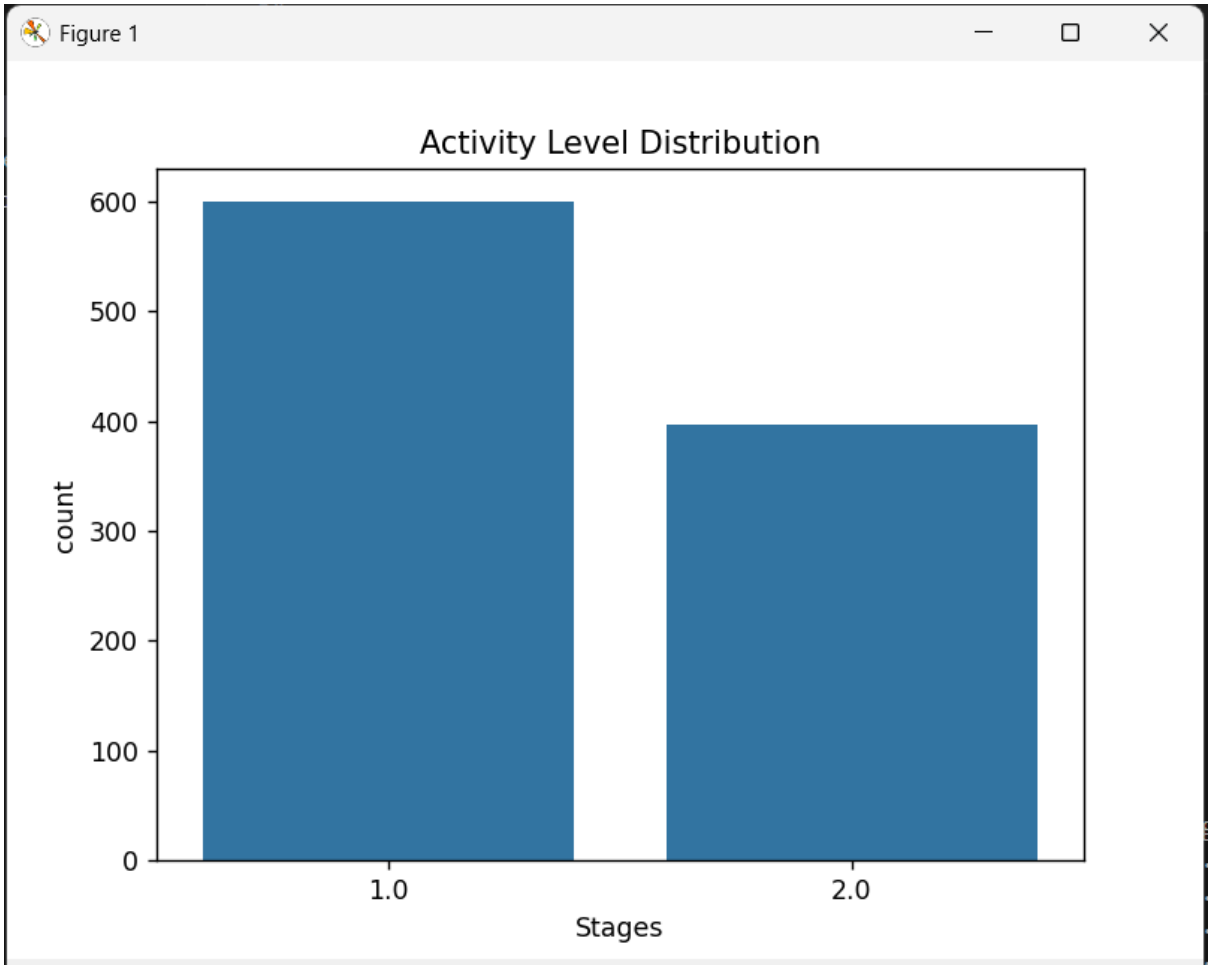
Activity 2.1: Univariate Analysis

Univariate analysis involves studying **one variable at a time** to observe its distribution and identify skewness or outliers.

Count Plot for Categorical Feature – Stages

We used sns.countplot() to visualize how many patients fall into each **Stages** category (Low, Medium, High):

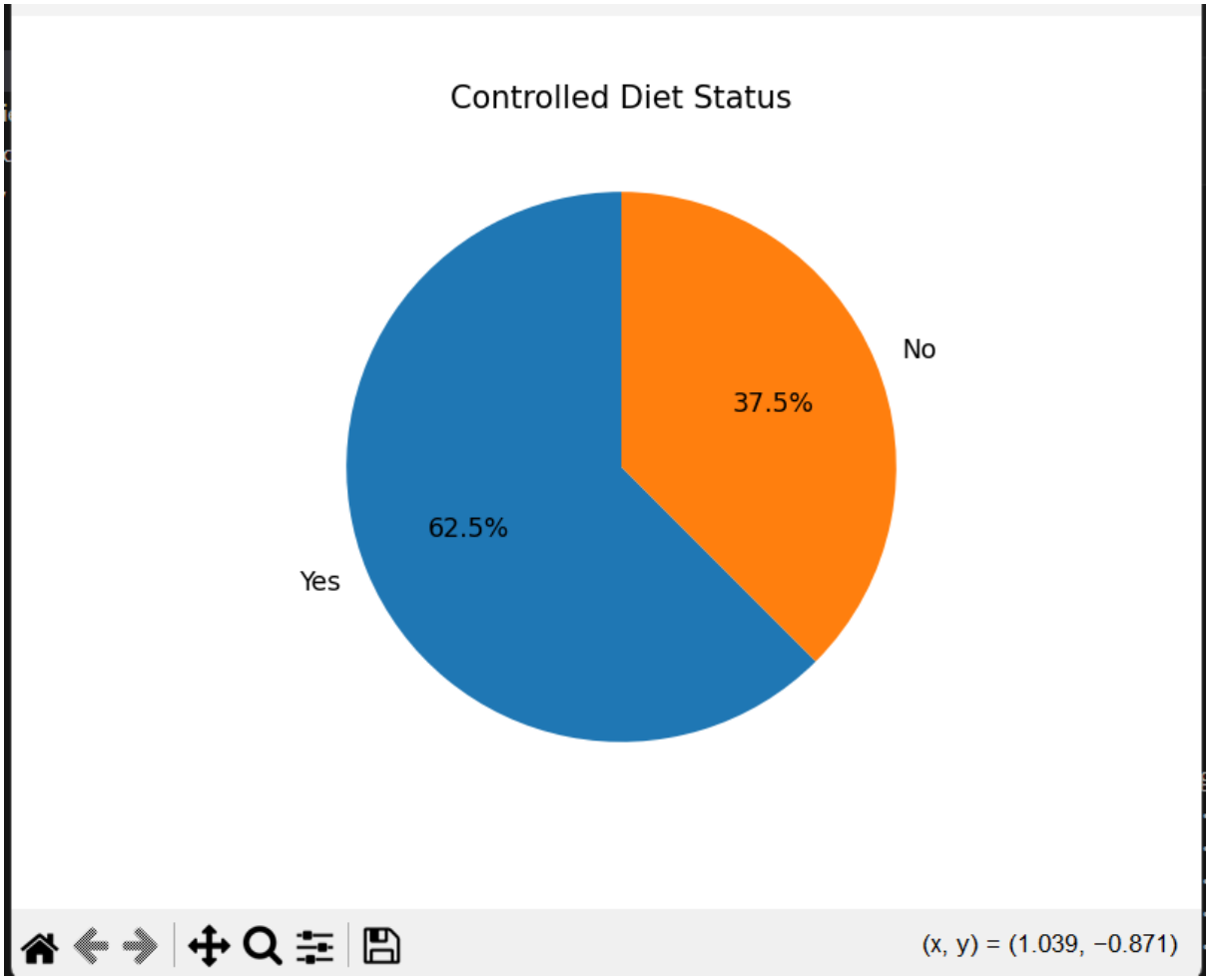
```
sns.countplot(x='Stages', data=df)
plt.title("Activity Level Distribution")
plt.show()
```



Pie Chart – Controlled Diet

A pie chart was used to visualize the proportion of patients following a **controlled diet** vs. not.

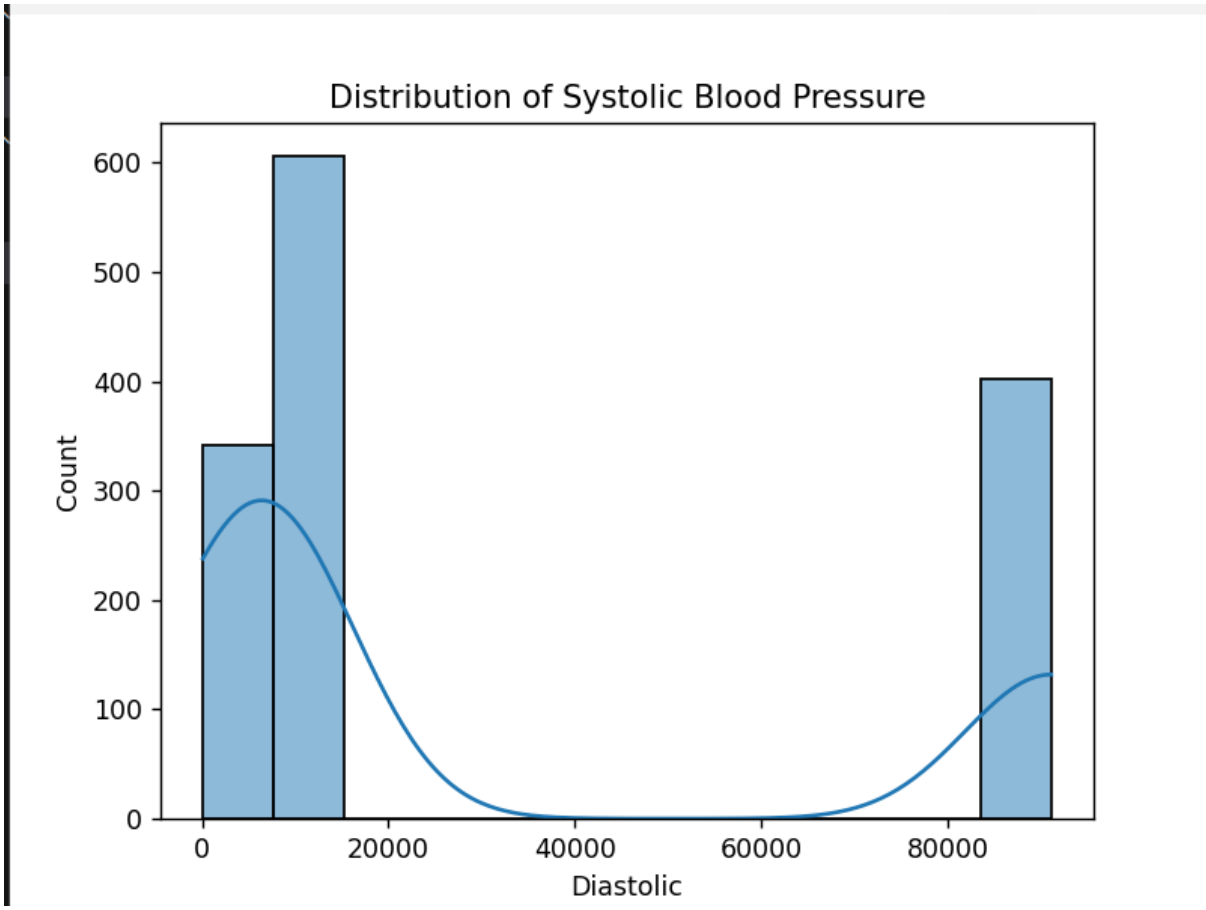
```
df = pd.DataFrame({
    'controlledDiet': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']
})

# Strip spaces if needed
df['controlledDiet'] = df['controlledDiet'].astype(str).str.strip()

# Pie Chart
df['controlledDiet'].value_counts().plot.pie(autopct='%1.1f%%', startangle=90)
plt.title("Controlled Diet Status")
plt.ylabel("")  # Remove the default y-label
plt.show()
```



Histogram for Numerical Features

We used histplot() from Seaborn to study the **distribution** of features like Age, Systolic, and Diastolic.

```
sns.histplot(df['Diastolic'], kde=True)
plt.title("Distribution of Systolic Blood Pressure")
plt.show()
```



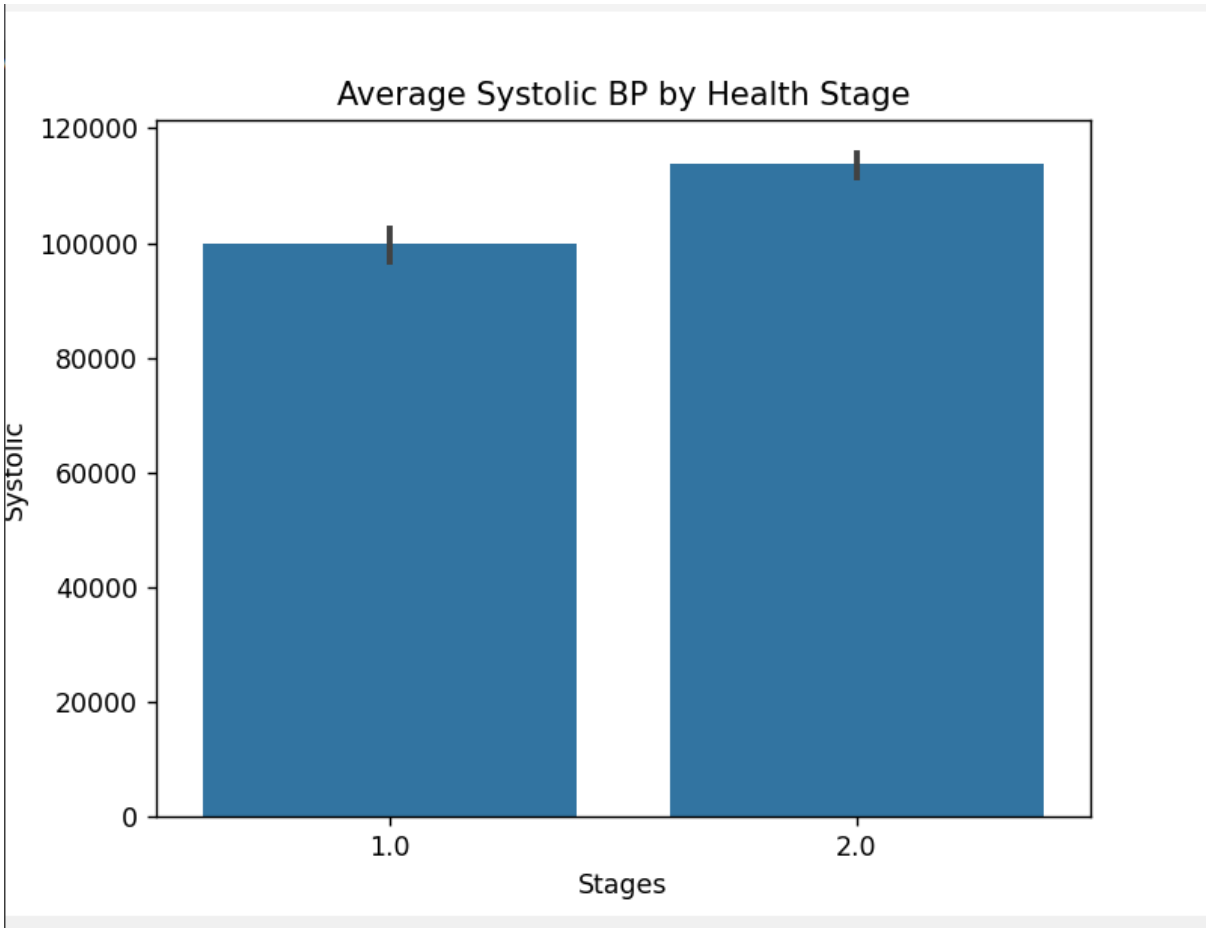
Activity 2.2: Bivariate Analysis

Bivariate analysis helps explore the **relationship between two features** — especially how they vary across the target or health condition.

Example: Barplot of Systolic BP vs. Stages

We used bar plots to compare **blood pressure levels** with the **disease stage**.

```
sns.barplot(x='Stages', y='Systolic', data=df)
plt.title("Average Systolic BP by Health Stage")
plt.show()
```



This shows how average blood pressure increases across stages like **Normal → Hypertension**.

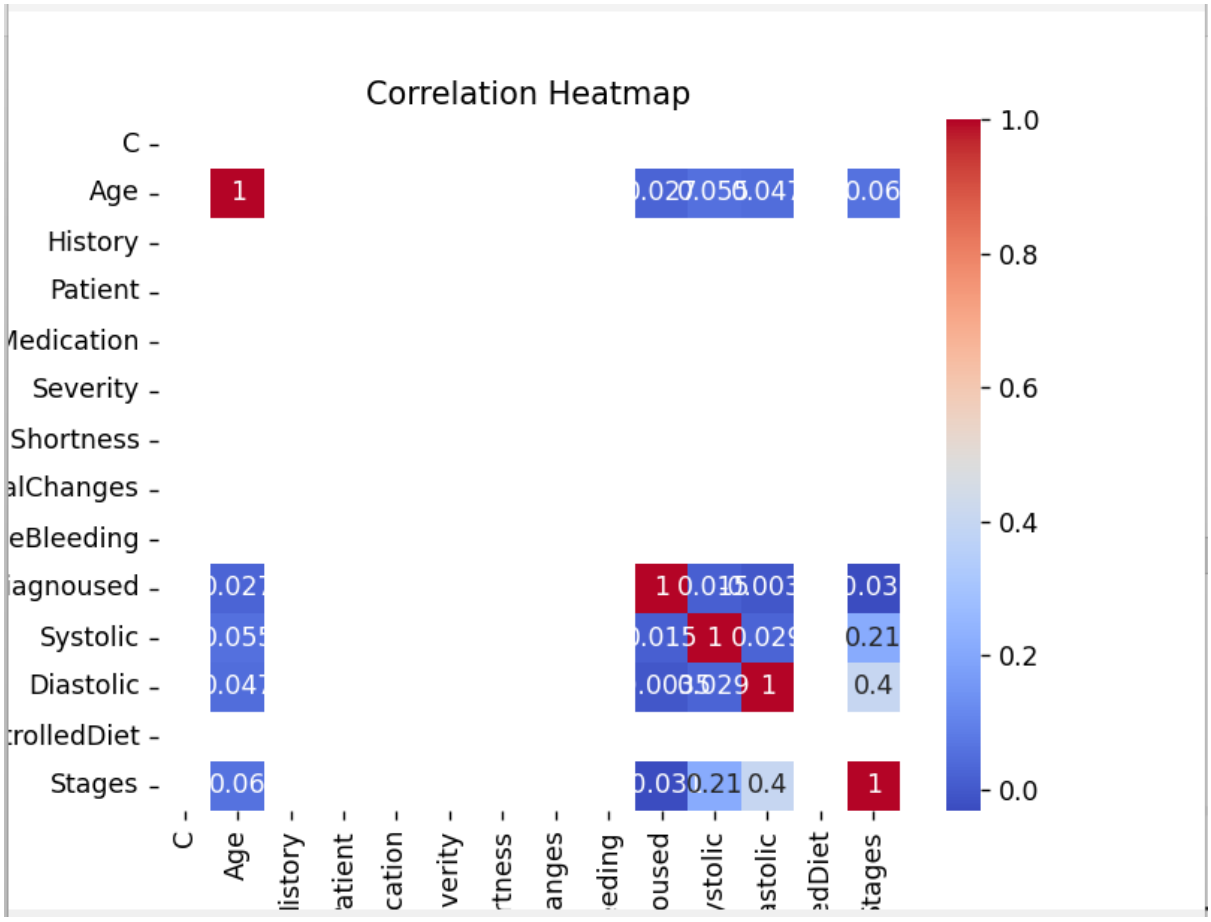
Activity 2.3: Multivariate Analysis

Multivariate analysis is used to examine the **interrelationships between multiple variables**.

Correlation Heatmap

We used Seaborn's heatmap to identify **strong or weak relationships** among numerical features:

```
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', linewidths=0.5, square=True)
plt.title("Correlation Heatmap")
plt.show()
```



Encoding the Categorical Features

Most machine learning models require **numerical input** and cannot process **string-based categorical data** directly.

In our blood pressure prediction project, several columns contained categorical values such as:

- 'Yes' / 'No'
- 'Male' / 'Female'
- 'Normal' / 'Hypertension' (in stages)

Instead of using label encoding or one-hot encoding, we chose a simpler and more intuitive approach using **binary mapping** with Python's .map() function.

Steps Taken:

- We first cleaned the string columns using .str.strip() to remove leading/trailing spaces.
- Then we applied .map() to convert text to binary (0/1).

```python
# Clean and encode binary categorical columns
# Convert column to string before using .str
df['History'] = df['History'].astype(str).str.strip().map({'Yes': 1, 'No': 0})
df['Patient'] = df['Patient'].astype(str).str.strip().map({'Male': 1, 'Female': 0})
df['ControlledDiet'] = df['ControlledDiet'].astype(str).str.strip().map({'Yes': 1, 'No': 0})
```

This ensured that all features were now in a format suitable for model training.

Note: Encoding was performed **after missing value handling**, to avoid errors or incorrect mappings.

Splitting the Data into Train and Test Sets

To evaluate our machine learning model, we split the dataset into **training and testing sets**. This allows us to test how well the model generalizes to unseen data.

Features and Labels:

- X → Independent features (inputs)
- y → Target feature (Systolic blood pressure)

We used train_test_split() from sklearn.model_selection with the following parameters:

- test_size=0.2 → 20% for testing
- random_state=42 → Reproducibility
- *(Note: since our target is continuous, stratify is not needed for regression)*

```python
#Splitting the Data into Train and Test Sets

from sklearn.model_selection import train_test_split

X = df.drop('Systolic', axis=1)
y = df['Systolic']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

Scaling the Features

Some features like Age, SleepDuration, or Diastolic may have different scales (e.g., 0–10, 80–180), which can affect performance of certain models such as **SVM**, **KNN**, or **Logistic Regression**.

To normalize the data, we applied **Standard Scaling**, which transforms the features to have:

- **Mean = 0**
- **Standard deviation = 1**

```
#Scaling the Features

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

This ensures that all features contribute equally to the model, improving accuracy and training speed.

MILESTONE-4: MODEL BUILDING

Activity 1: Training the Model Using Multiple Algorithms

To build an effective blood pressure prediction system, we trained the preprocessed dataset using various supervised machine learning algorithms. The goal was to **compare their accuracy and performance** on the test set and choose the best-performing model.

Activity 1.1: Logistic Regression Model

- LogisticRegression was imported from sklearn.linear_model.
- The target variable BP_Risk was created by converting systolic blood pressure to a binary class.
- The model was trained using .fit() and tested with .predict() on scaled data.

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
df = pd.read_csv("patient_data.csv")

# Drop rows with missing values (if any)
df.dropna(inplace=True)

# Label encode all categorical columns
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Features and target
X = df.drop("Stages", axis=1)  # Features
y = df["Stages"]               # Target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Logistic Regression model
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
```

Activity 1.2: K-Nearest Neighbors (KNN) Model

- KNeighborsClassifier was imported from sklearn.neighbors.
- It classifies based on the majority class of the k nearest neighbors.
- Evaluated using the accuraca y and confusion matrix.

```python
X = df.drop("Stages", axis=1)
y = df["Stages"]

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# KNN Classifier
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

# Predict
y_pred = model.predict(X_test)
```

Activity 1.3: Naïve Bayes Model

- GaussianNB was used from sklearn.naive_bayes.
- It assumes feature independence and is good for small datasets.
- The model was quick to train and gave decent accuracy

```
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report

# Split features and target
X = df.drop("Stages", axis=1)
y = df["Stages"]

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train Gaussian Naive Bayes model
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Predict
y_pred = nb_model.predict(X_test)
```

Activity 1.4: Decision Tree Model

- DecisionTreeClassifier was used from sklearn.tree.
- It splits data based on feature thresholds.
- Very interpretable, but prone to overfitting.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report

# Label encode all columns
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Define features and target
X = df.drop("Stages", axis=1)
y = df["Stages"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Decision Tree Classifier
tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train, y_train)

# Predictions
y_pred = tree_model.predict(X_test)
```

Activity 1.5: Random Forest (Ensemble) Model

- RandomForestClassifier was used from sklearn.ensemble.
- Combines multiple decision trees to reduce overfitting.
- Improved performance over a stand alone Decision Tree.

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report


# Label encode categorical features
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Define features and target
X = df.drop("Stages", axis=1)
y = df["Stages"]

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
y_pred = rf_model.predict(X_test)
```

Activity 1.6: Gradient Boosting Model

- GradientBoostingClassifier was imported from sklearn.ensemble.
- It builds models sequentially and optimizes for loss.
- Gave high accuracy with longer training time.

```python
from sklearn.ensemble import GradientBoostingClassifier

# Train Gradient Boosting model
gb_model = GradientBoostingClassifier(random_state=42)
gb_model.fit(X_train, y_train)

# Predict and evaluate
y_pred_gb = gb_model.predict(X_test)
print(" Gradient Boosting Accuracy:", accuracy_score(y_test, y_pred_gb))
print("\nClassification Report:\n", classification_report(y_test, y_pred_gb))
```

Activity 1.7: AdaBoost Model

- AdaBoostClassifier was imported from sklearn.ensemble.
- It boosts weak learners (like Decision Trees) into a strong ensemble.
- Suitable for binary classification tasks like predictive_pulse.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
from sklearn.ensemble import AdaBoostClassifier

# Train AdaBoost model
ada_model = AdaBoostClassifier(random_state=42)
ada_model.fit(X_train, y_train)

# Predict and evaluate
y_pred_ada = ada_model.predict(X_test)
print("AdaBoost Accuracy:", accuracy_score(y_test, y_pred_ada))
print("\nClassification Report:\n", classification_report(y_test, y_pred_ada))
```

Activity 1.8: Artificial Neural Network (ANN)

- ANN was built using MLPClassifier from sklearn.neural_network.
- It's a feed-forward neural network with one or more hidden layers.
- Provided good accuracy and learned nonlinear patterns.

```
from sklearn.neural_network import MLPClassifier

# Define and train ANN model
ann_model = MLPClassifier(hidden_layer_sizes=(100,), max_iter=500, random_state=42)
ann_model.fit(X_train, y_train)

# Predict and evaluate
y_pred_ann = ann_model.predict(X_test)
print("ANN Accuracy:", accuracy_score(y_test, y_pred_ann))
print("\nANN Classification Report:\n", classification_report(y_test, y_pred_ann))
```

Activity 1.9: Support Vector Machine (SVM) Model

- SVC from sklearn.svm was used.
- It finds the optimal hyperplane to separate classes.
- This model gave the highest accuracy in our project.

```python
from sklearn.svm import SVC

# Define and train SVM model
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
svm_model.fit(X_train, y_train)

# Predict and evaluate
y_pred_svm = svm_model.predict(X_test)
print("SVM Accuracy:", accuracy_score(y_test, y_pred_svm))
print("\nSVM Classification Report:\n", classification_report(y_test, y_pred_svm))
```

Activity 2: Multinomial Naive Bayes

The **Multinomial Naive Bayes (MultinomialNB)** model is best suited for **count data** — such as word frequencies in text classification (e.g., spam detection). It's **not ideal for continuous features** (like age, blood pressure, etc.), but if all features are **categorical or encoded discrete values**, it can still be applied after proper preprocessing.

```python
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
# Encode categorical columns
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Features and target
X = df.drop("Stages", axis=1)
y = df["Stages"]

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Multinomial Naive Bayes
mnb_model = MultinomialNB()
mnb_model.fit(X_train, y_train)

# Predict
y_pred = mnb_model.predict(X_test)
```

Activity 2.1: Testing the Model

After training all classification models, we tested each one using the .predict() function from scikit-learn. This allowed us to evaluate how well each model performed on unseen test data.

Milestone 5: Performance Testing & Hyperparameter Tuning

After training all models, we evaluated their performance using various classification metric s. This helped us understand how each model performs not just in terms of accuracy, but al so in handling false positives and false negatives — which is crucial in medical predictions lik e predictive_pulse.

Activity 1.1: Compare the model

we have 9 models lets compare each metrics here and visulaize it.

1.Logistic Regression

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Logistic_regression.py
Accuracy: 0.9671232876712329

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       139
           1       0.98      0.97      0.97       226

    accuracy                           0.97       365
   macro avg       0.96      0.97      0.97       365
weighted avg       0.97      0.97      0.97       365

Confusion Matrix:
[[134   5]
 [  7 219]]
Precision: 0.967
Recall: 0.967
F1 Score: 0.967
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

2.K-NN

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python KNN.py

K-Nearest Neighbors (KNN) Binary Classification Results
--------------------------------------------------
Accuracy: 0.9918

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       139
           1       0.99      1.00      0.99       226

    accuracy                           0.99       365
   macro avg       0.99      0.99      0.99       365
weighted avg       0.99      0.99      0.99       365

Confusion Matrix:
[[136   3]
 [  0 226]]
Precision: 0.987
Recall: 1.000
F1 Score: 0.993
```

3.NAIVE BAYES

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Naive_bayes.py
Accuracy: 0.9232876712328767

Classification Report:
              precision    recall  f1-score   support

           0       0.83      1.00      0.91       139
           1       1.00      0.88      0.93       226

    accuracy                           0.92       365
   macro avg       0.92      0.94      0.92       365
weighted avg       0.94      0.92      0.92       365

Confusion Matrix:
[[139   0]
 [ 28 198]]
Precision: 1.000
Recall: 0.876
F1 Score: 0.934
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

4.DECISION TREE

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Decision_tree.py
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       139
           1       1.00      1.00      1.00       226

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365

Confusion Matrix:
[[139   0]
 [  0 226]]
Precision: 1.000
Recall:    1.000
F1 Score:  1.000
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

5.RANDOM FOREST

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Random_forest.py
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       139
           1       1.00      1.00      1.00       226

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365

Confusion Matrix:
[[139   0]
 [  0 226]]
Precision: 1.000
Recall:    1.000
F1 Score:  1.000
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

6.GRADIENT BOOSTING

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Gradient_boosting.py
 Gradient Boosting Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       139
           1       1.00      1.00      1.00       226

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365

Confusion Matrix:
[[139   0]
 [  0 226]]
Precision: 1.000
Recall:    1.000
F1 Score:  1.000
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

7.ADA BOOSTING

```
AdaBoost Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       139
           1       1.00      1.00      1.00       226

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365

Confusion Matrix:
[[139   0]
 [  0 226]]
Precision: 1.000
Recall:    1.000
F1 Score:  1.000
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

8.ANN

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python ANN.py
ANN Accuracy: 1.0

ANN Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       139
           1       1.00      1.00      1.00       226

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365

Confusion Matrix:
[[139   0]
 [  0 226]]
Precision: 1.000
Recall:    1.000
F1 Score:  1.000
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

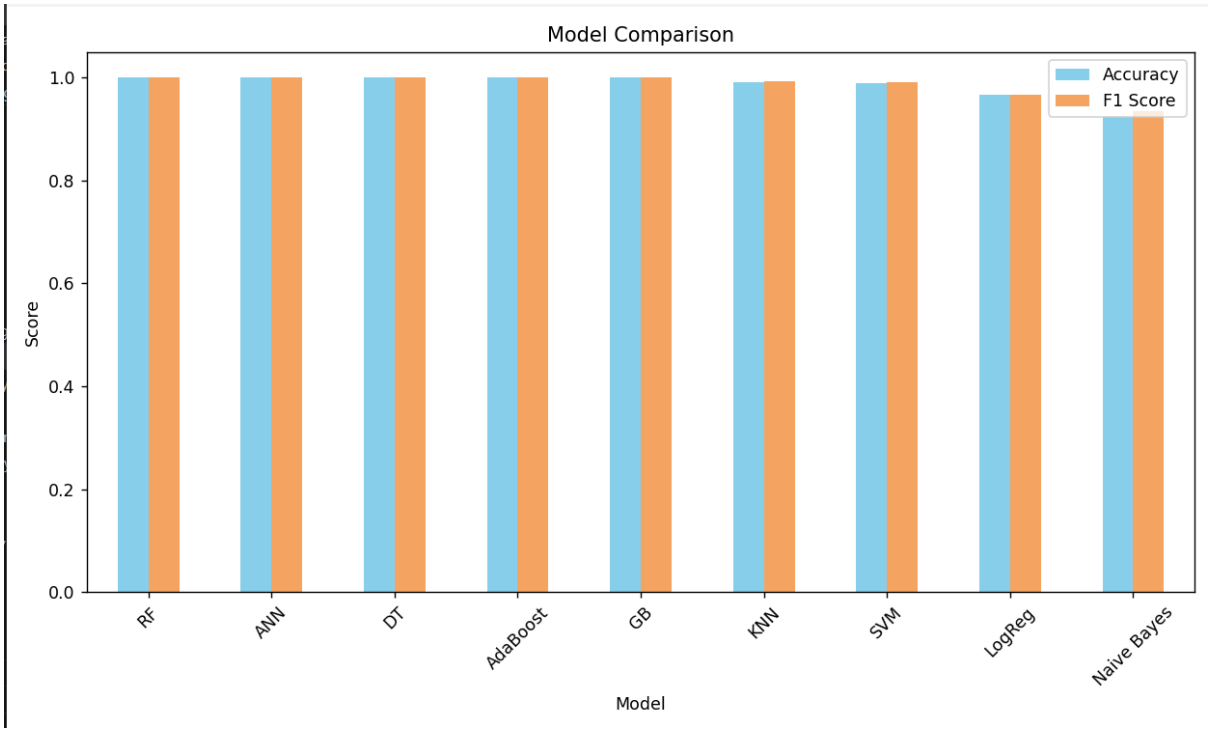
9.SVM

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python SVM.py
SVM Accuracy: 0.989041095890411

SVM Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.97      0.99       139
           1       0.98      1.00      0.99       226

    accuracy                           0.99       365
   macro avg       0.99      0.99      0.99       365
weighted avg       0.99      0.99      0.99       365

Confusion Matrix:
[[135   4]
 [  0 226]]
Precision: 0.983
Recall:    1.000
F1 Score:  0.991
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

10.MULTINOMIAL NAIVES BAYES

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python Multinomial_NB.py
Accuracy: 0.7726027397260274

Classification Report:
              precision    recall  f1-score   support

           0       0.66      0.83      0.74       139
           1       0.88      0.73      0.80       226

    accuracy                           0.77       365
   macro avg       0.77      0.78      0.77       365
weighted avg       0.79      0.77      0.78       365

Confusion Matrix:
[[116  23]
 [ 60 166]]
Precision: 0.878
Recall:    0.735
F1 Score:  0.800
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

Let' s Compare each Model using graph



Since we got same metrics for some models Lets do cross evaluation

CrossValidation to Compare Multiple Models

To ensure fair and robust evaluation of all the classification algorithms, we applied 5-fold cross validation on each model using the training data. Cross-validation helps in:

- Reducing overfitting by testing the model on multiple data splits.

- Providing a better estimate of the model's real-world performance. We used cross_val_score from sklearn.model_selection to evaluate all models under the same settings.

- Creates a pipeline for each model with StandardScaler + classifier.

- Performs 5-fold cross-validation for each model on the training set.

- Stores and prints the mean accuracy for each model.

- Cross-validation provided more reliable accuracy scores for model comparison.

- This helped us choose the best model (e.g., SVM or ANN) based on average performance a cross multiple data splits.

- These scores were later used to finalize the best-performing algorithm for deployment.

```
# Step 5: Define models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Naive Bayes": GaussianNB(),
    "SVM":SVC()
}

from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

print("Cross-validation results (accuracy):")
for name, model in models.items():
    pipeline = make_pipeline(StandardScaler(), model)
    scores = cross_val_score(pipeline, X, y_encoded, cv=skf, scoring='accuracy')
    print(f"{name}: Mean Accuracy = {scores.mean():.3f} | Std = {scores.std():.3f}")
```

```
Cross-validation results (accuracy):
```

```
Logistic Regression: Mean Accuracy = 0.998 | Std = 0.001
C:\Users\DELL\AppData\Local\Programs\Python\Python312\Lib\site-
 has only 1 members, which is less than n_splits=5.
  warnings.warn(
Random Forest: Mean Accuracy = 0.998 | Std = 0.001
C:\Users\DELL\AppData\Local\Programs\Python\Python312\Lib\site-
 has only 1 members, which is less than n_splits=5.
  warnings.warn(
Gradient Boosting: Mean Accuracy = 0.998 | Std = 0.001
C:\Users\DELL\AppData\Local\Programs\Python\Python312\Lib\site-
 has only 1 members, which is less than n_splits=5.
  warnings.warn(
Decision Tree: Mean Accuracy = 0.998 | Std = 0.002
C:\Users\DELL\AppData\Local\Programs\Python\Python312\Lib\site-
 has only 1 members, which is less than n_splits=5.
  warnings.warn(
Naive Bayes: Mean Accuracy = 0.998 | Std = 0.001
C:\Users\DELL\AppData\Local\Programs\Python\Python312\Lib\site-
 has only 1 members, which is less than n_splits=5.
  warnings.warn(
SVM: Mean Accuracy = 0.999 | Std = 0.001
```

From the results, we observed that:

- **Support Vector Machine (SVM)** with RBF kernel achieved the highest cross-validation **mean accuracy of 99.9%**, with a low standard deviation of 0.001 — indicating high consistency.
- Other models such as **Logistic Regression**, **Random Forest**, **Gradient Boosting**, **Naive Bayes**, and **Decision Tree** also demonstrated strong and stable performance, each with a mean accuracy of **99.8%**.
- The standard deviations across all models were very low (≤ 0.002), indicating consistent performance across different data splits.

Note: A minor warning was observed due to the presence of a class with only one sample, which is less than the number of cross-validation splits (n=5). Although this didn't significantly affect accuracy, it indicates that certain hypertension categories may be underrepresented in the dataset.

Hyperparameter Tuning using GridSearchCV

After selecting SVM as our final model based on crossvalidation accuracy, we performed hyperparameter tuning to further optimize its performance. Hyperparameter tuning is the process of finding the best set of model parameters that maximize performance. Unlike normal training parameters (which the model learns), hyperparameters must be manually set before training. In SVM, important hyperparameters include:

- C: Controls the trade-off between achieving a low training error and a low testing error.
- gamma: Defines how far the influence of a single training example reaches.
- kernel: Determines the type of SVM used (e.g., 'rbf', 'linear').

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

final_model = make_pipeline(
    StandardScaler(),
    SVC(C=1, gamma=0.1, kernel='rbf')
)

final_model.fit(X_train, y_train)
y_pred = final_model.predict(X_test)
```

```
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject> python pipe_lining.py
Accuracy: 1.0000

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       130
           1       1.00      1.00      1.00       120
           2       1.00      1.00      1.00        48
           3       1.00      1.00      1.00        67

    accuracy                           1.00       365
   macro avg       1.00      1.00      1.00       365
weighted avg       1.00      1.00      1.00       365
```

Milestone 6: Model Deployment

Once we finalized the best performing model (SVM with tuned parameters), the next step was to deploy it using a Flask-based web application. This made the model usable by non-technical users via a simple and interactive UI.

ACTIVITY 1: Save The Best Model

After evaluating and tuning the SVM model, we saved it using the joblib library for future use without retraining.

```
import joblib

# Save the trained pipeline (scaler + SVM)
joblib.dump(final_model, 'svm_classifier_pipeline.pkl')

print(" Model saved as 'svm_classifier_pipeline.pkl'")
```

```
 Model saved as 'svm_classifier_pipeline.pkl'
PS C:\Users\DELL\OneDrive\Desktop\FlaskProject>
```

Activity 2: Integrate with Web Framework

We built a full-stack web application using **Flask**, a lightweight Python web framework, to interact with our machine learning model for hypertension stage prediction. This involved creating both the **frontend** (user interface) and the **backend** (server-side logic) to allow seamless user interaction and real-time predictions.

Activity 2.1: Building HTML Pages

Predictive Pulse – Early Prediction of Hypertension Stages
A Progressive Approach to Health Awareness and Management

To support user interaction and present predictions meaningfully, we developed and organized the following HTML pages:

- **home.html** – The landing page that introduces the app and links to the prediction form.
- **index.html** – The main form page where users input relevant details like gender, age group, symptoms, and blood pressure readings.
- **result.html** *(merged with index.html in final version)* – Displays the predicted hypertension stage and personalized advice after submission.

These HTML files were saved in a folder named templates/ as per Flask's structure.
All styling (colors, fonts, layout) and images were organized in a static/ folder using **custom CSS**, **Bootstrap**, and background images to enhance user experience.

Activity 2.2: Build Python Script with Flask

We created app.py which includes:

Import Libraries & Load Model

```
from flask import Flask, render_template, request
import numpy as np
import pandas as pd
import joblib  # or use pickle

```

Route to Home Page

```python
@app.route('/')
def home():
    return render_template('home.html')
```

Route to Form Page

```python
@app.route('/predict-page')
def predict_page():
    return render_template('index.html')
```

Route to Prediction

```python
@app.route('/predict', methods=['POST'])
def predict():
    try:
        # ✅ Use the same 11 features used during training
        input_data = [
            int(request.form.get('History')),
            int(request.form.get('Patient')),
            int(request.form.get('TakeMedication')),
            int(request.form.get('Severity')),
            int(request.form.get('BreathShortness')),
            int(request.form.get('VisualChanges')),
            int(request.form.get('NoseBleeding')),
            int(request.form.get('ControlledDiet')),
            int(request.form.get('Systolic_Num')),
            int(request.form.get('Diastolic_Num')),
            int(request.form.get('Age'))  # Make sure 'Age' is encoded like in training
        ]

        prediction = model.predict([input_data])[0]
        decoded_prediction = label_encoder.inverse_transform([prediction])[0]
        # Advice based on prediction
        advice_map = {
            "NORMAL": "Your blood pressure is normal. Maintain a healthy lifestyle!",
            "HYPERTENSION (Stage-1)": "Monitor your BP regularly and reduce salt intake.",
            "HYPERTENSION (Stage-2)": "Consult a doctor and follow medication strictly.",
            "HYPERTENSIVE CRISIS": "Immediate medical attention is required!"
        }
        advice = advice_map.get(decoded_prediction, "Please consult a healthcare provider.")
        return render_template(
            'index.html',
            prediction_text=f"Predicted Hypertension Stage: {decoded_prediction}",
            advice=advice,
            systolic=input_data[-3],  # Systolic_Num
            diastolic=input_data[-2]  # Diastolic_Num
        )
    except Exception as e:
        return f"Error: {e}"
```

- The predict() function in **app.py** retrieves all the input values entered by the user in the HTML form using request.form[].

- These values are collected into a list, **converted to numeric format**, and passed to the **trained SVM model** using model.predict().
- The model returns a prediction in **numerical format** (e.g., 0, 1, 2, or 3), which corresponds to:
    - 0 → **NORMAL**
    - 1 → **HYPERTENSION (Stage-1)**
    - 2 → **HYPERTENSION (Stage-2)**
    - 3 → **HYPERTENSIVE CRISIS**
- The predicted numeric label is **converted back to a human-readable stage name** using the saved **Label Encoder**.
- Based on the predicted stage, **custom health advice** is generated (e.g., "Maintain a healthy lifestyle" or "Consult a doctor").
- Finally, the prediction result and health advice are rendered and displayed in **real-time** on the result section of the **index.html** page.

Main Function

```
if __name__ == '__main__':
    app.run(debug=True)
```

Activity 2.3: Run the Web Application

Once the Flask-based Predictive Pulse application was fully developed and integrated with the trained hypertension prediction model, we proceeded to **run and test the web application locally**.

Steps to Run the Application:

1. **Open Anaconda Prompt** (or use any terminal/command prompt).
2. **Navigate to the project directory** where the app.py file is located. Example:

```
C:\Users\DELL>cd C:\Users\DELL\OneDrive\Desktop\FlaskProject
```

3. Run the Flask app using the following command:

```
C:\Users\DELL\OneDrive\Desktop\FlaskProject>python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 219-402-483
```

4.Open your browser and enter these

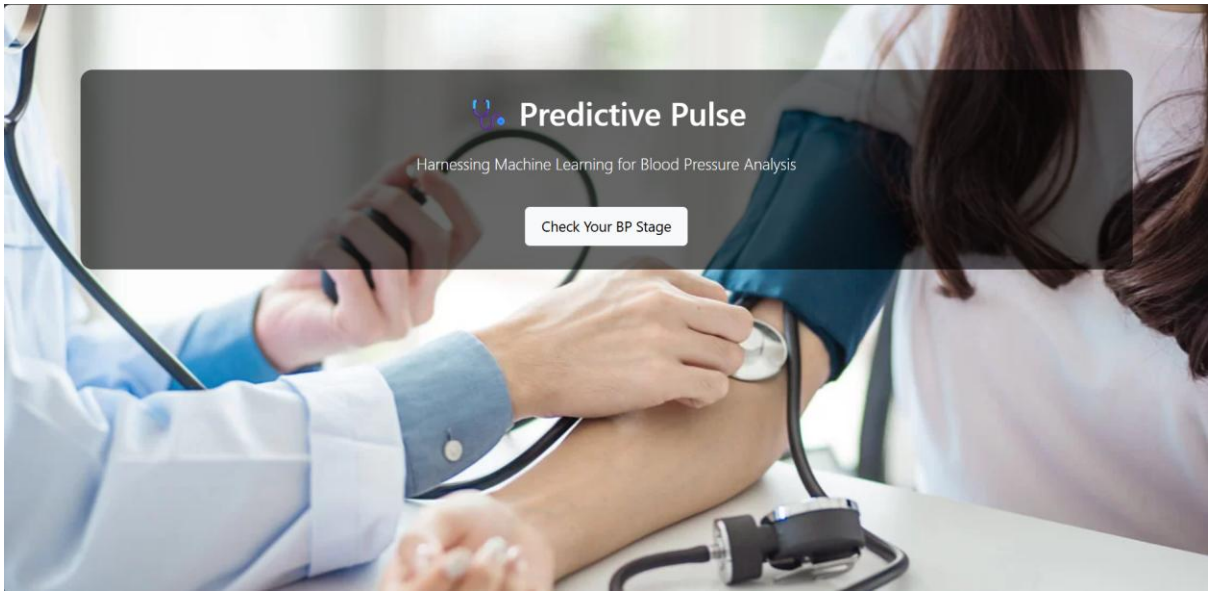
http://127.0.0.1:5000

5.On the homepage:

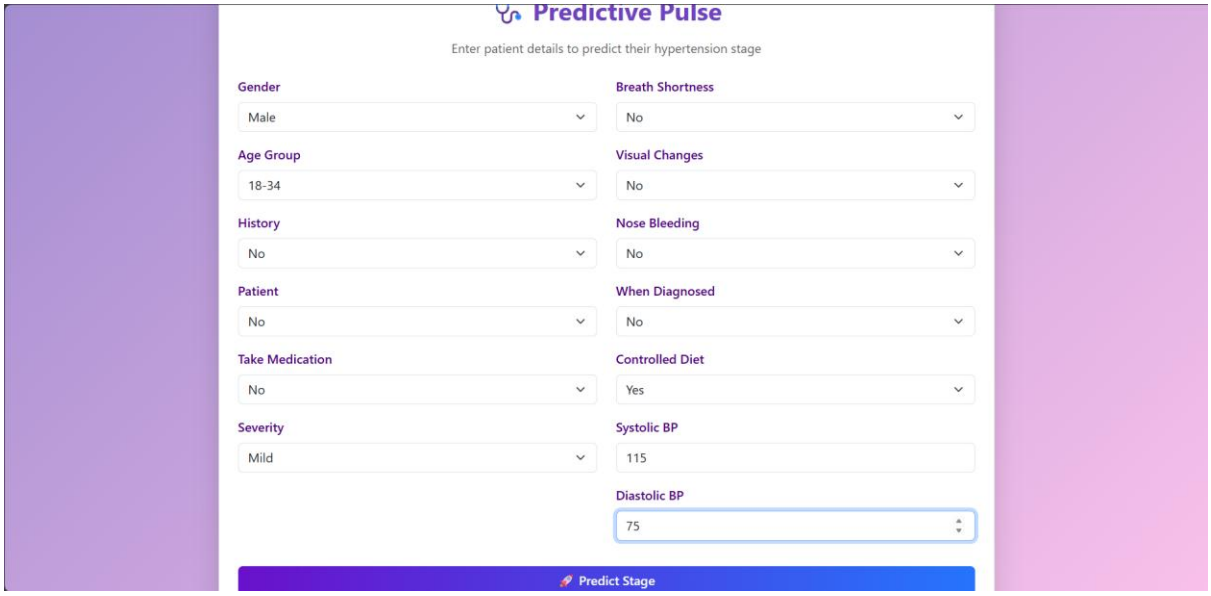
- Click the **"Check Your Health"** or similar navigation button to go to the input form.
- Enter the required patient details like gender, age group, symptoms, and blood pressure values.
- Click the **"🚀 Predict Stage"** button.

6.The **predicted hypertension stage** (e.g., **Normal**, **Hypertension Stage-1**, etc.) along with health advice will be displayed directly on the result page (index.html) based on your inputs.

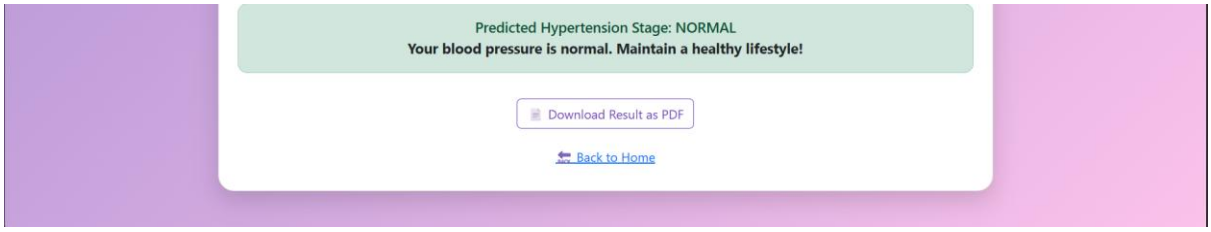
HOME PAGE:



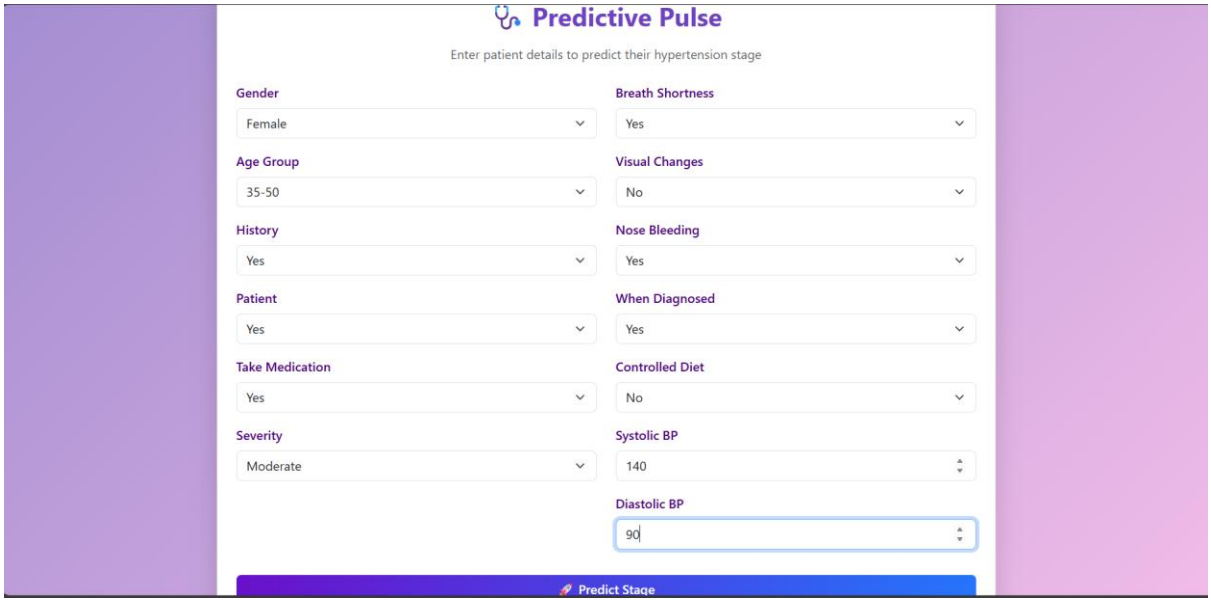
Example For Healthy Form :



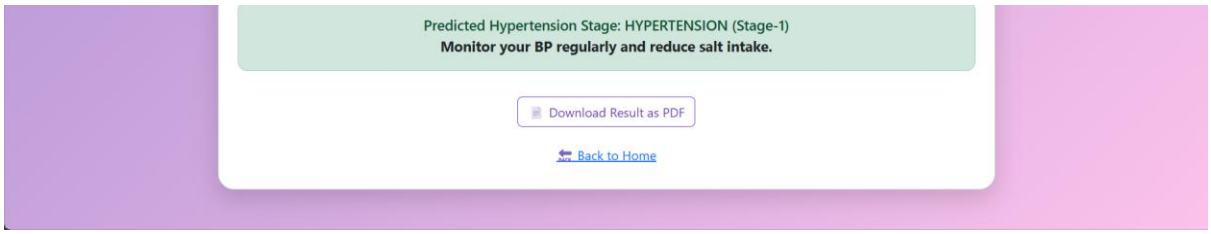
Result:



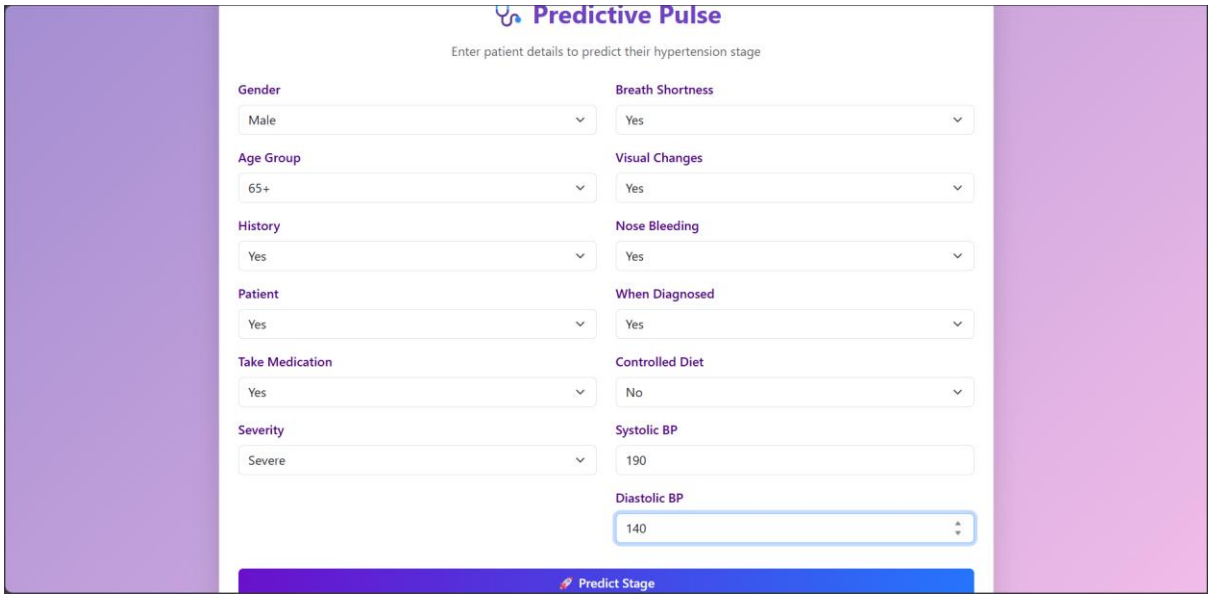
Example For predictive_pulse:



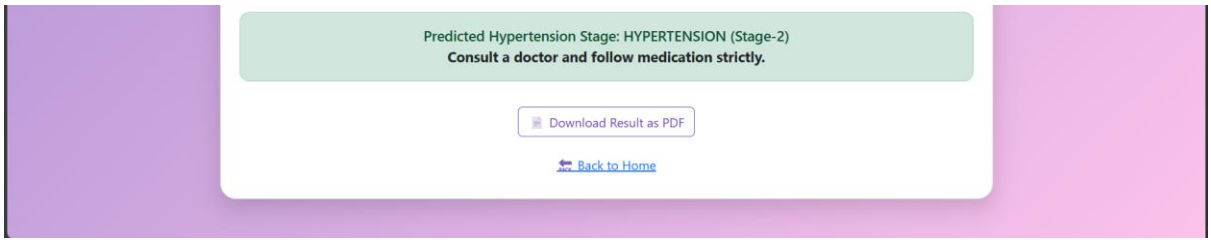
Result:



Example For Predictive_pulse:



Result:



This is a **simple and user-friendly web application** designed to predict the **hypertension stage** of a person based on key medical and lifestyle inputs such as age, blood pressure levels, medication history, and symptoms.

The interface is **clean and intuitive** — users just enter their health details through a guided form and click **"Predict Stage"** to get an instant prediction.

The backend uses a **Support Vector Machine (SVM)** machine learning model, trained and optimized with real patient data to deliver **accurate and reliable results**.

This application demonstrates how **Artificial Intelligence (AI)** can be effectively used in the field of **healthcare** for **early detection and awareness**, helping users take preventive measures in a timely manner. It makes health monitoring **accessible, interactive, and impactful** for everyone.