BLOOD PRESSURE PREDICTION: PROJECT PLAN AND DEPLOYMENT

TABLE OF CONTENTS

1. INTRODUCTION

This document outlines a comprehensive plan for developing a predictive model to identify blood pressure stages. The increasing prevalence of hypertension and associated cardiovascular risks necessitates accurate and timely blood pressure monitoring. This project aims to address this need by creating a machine-learning-driven system capable of predicting blood pressure stages based on relevant health data.

The primary goal is to develop, train, and deploy a machine learning model that accurately predicts blood pressure stages. This plan encompasses the entire lifecycle of the project, from initial data collection and preprocessing to the deployment of a user-friendly Flask web application. The scope of this document includes detailed steps for data handling, exploratory data analysis, model selection, training, evaluation, and deployment.

Key technologies utilized in this project include Python, pandas for data manipulation, scikit-learn for machine learning algorithms, and Flask for creating the web application interface.

2. PROJECT OVERVIEW AND GOALS

This project addresses the critical need for accurate blood pressure monitoring and prediction. Its primary objective is to develop a robust machine learning model capable of accurately predicting blood pressure stages, categorized as Normal, Elevated, Hypertension Stage 1, and Hypertension Stage 2, based on a variety of health-related data inputs.

Beyond predictive accuracy, secondary goals include creating a user-friendly Flask web application, ensuring model interpretability to understand contributing factors, and establishing a scalable deployment pipeline for wider accessibility. The web application will enable users to input their health data and receive immediate blood pressure stage predictions.

The strategic importance of this project lies in its potential to facilitate proactive health management and early intervention. By identifying individuals at risk of developing hypertension, timely lifestyle modifications and medical interventions can be implemented, ultimately reducing the incidence of cardiovascular diseases. This project aligns with broader efforts to promote preventative healthcare and improve overall public health outcomes.

3.1 FUNCTIONAL REQUIREMENTS

The blood pressure prediction system must fulfill specific functional requirements to ensure accurate predictions and a user-friendly experience. These requirements dictate how users interact with the system and how the system processes user input to generate predictions.

- **User Input:** The system must allow users to input relevant health data, including but not limited to:
    - Age
    - Gender
    - Height (cm or inches)
    - Weight (kg or lbs)
    - Existing Medical Conditions (e.g., diabetes, kidney disease)
    - Family History of Hypertension
    - Lifestyle Factors:
        - Smoking Status (smoker/non-smoker)
        - Alcohol Consumption (frequency and quantity)
        - Physical Activity Level (sedentary, moderate, active)

- **Prediction Generation:** The core requirement is that the machine learning model must process the input data and generate a prediction of the user's blood pressure stage. The possible prediction outputs are:
    - Normal
    - Elevated
    - Hypertension Stage 1
    - Hypertension Stage 2
- **Result Display:** The web interface must clearly display the prediction result to the user. This includes:
    - Displaying the predicted blood pressure stage.
    - Providing a brief explanation of what the predicted stage means.
    - Offering general recommendations based on the prediction (e.g., consult a doctor, adopt a healthier lifestyle).
- **Error Handling:** The system must handle edge cases gracefully, including:
    - Missing Input Data: Prompt the user to provide the missing information.
    - Invalid Input Data: Validate the data entered by the user (e.g., ensuring age is a reasonable number) and display appropriate error messages if the data is invalid.

3.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements define the quality attributes of the blood pressure prediction system. These requirements ensure the system is performant, secure, usable, reliable, scalable, and maintainable.

- **Performance:** Prediction response time should be under 3 seconds. Web application load times must be efficient, with pages loading in under 2 seconds.
- **Security:** Data privacy is paramount; all health data must be encrypted both in transit and at rest. Input validation must be implemented to prevent malicious injection attacks.
- **Usability:** The user interface must be intuitive and user-friendly. Clear instructions and contextual help should be provided to guide users.
- **Reliability:** The system should maintain a minimum uptime of 99.9%. Graceful error handling is essential to prevent system crashes and data loss.
- **Scalability:** The system must be designed to accommodate an increasing number of users and growing data volumes. Horizontal scalability should be considered for future expansion.

- **Maintainability:** Code should be well-documented and adhere to Python best practices. Software versions for all dependencies should be tracked for compatibility and reproducibility. Ease of updates and modifications is crucial for long-term maintainability.

Comprehensive error handling is required across all system components. Adherence to Python best practices is mandatory for code quality, readability, and maintainability.

3.3 SYSTEM ARCHITECTURE OVERVIEW

The system architecture comprises three main layers: data acquisition and storage, the machine learning model component, and the Flask web application. The data acquisition layer handles user inputs and could potentially integrate with external health data sources. The machine learning model component houses the trained blood pressure prediction model, built using Python and scikit-learn, and provides prediction services.

The Flask web application serves as the user interface, built using HTML, CSS, and JavaScript, allowing users to input their health data and view the predicted blood pressure stage. The application interacts with the model component to generate predictions and presents the results in a user-friendly format.

The data flow begins with the user inputting health data via the Flask web application. This data is then sent to the machine learning model component for prediction. The predicted blood pressure stage is returned to the Flask application, which displays the result to the user. Technologies used include Flask for the front-end and a Python environment with relevant libraries for the ML backend.

4. DATA ACQUISITION AND UNDERSTANDING

The foundation of the blood pressure prediction model lies in the acquisition and thorough understanding of relevant health-related data. This process begins with identifying potential data sources. Publicly available datasets, such as those from the National Health and Nutrition Examination Survey (NHANES), can provide a rich source of health data. If real-world data is limited or unavailable, synthetic data generation techniques can be employed to create representative datasets, ensuring a diverse range of health profiles for training the model.

Alternatively, anonymized clinical datasets, obtained through collaborations with healthcare providers (subject to data access agreements and ethical considerations), can offer valuable insights.

Once the data is acquired, initial understanding is paramount. This involves a detailed review of the data schema to identify all available features and their corresponding data types (e.g., numerical, categorical). Understanding the meaning of each feature in the context of blood pressure prediction is crucial for effective feature engineering and model selection. Preliminary data quality checks are essential at this stage. These checks include identifying and addressing missing values, detecting and handling outliers, and resolving any inconsistencies within the dataset. Understanding the distribution of key variables will also inform subsequent preprocessing steps.

5. DATA PREPROCESSING

Data preprocessing is a crucial step in preparing the raw data for the machine learning model. The following steps will be implemented using pandas for data manipulation and scikit-learn for various preprocessing utilities:

5.1 HANDLING MISSING VALUES

Strategies for handling missing values will depend on the nature and extent of the missing data. Options include:

- **Imputation:** Replacing missing values with estimated values.
  - Mean/Median Imputation: Using the mean or median of the feature.
  - Mode Imputation: Using the most frequent value for categorical features.
  - K-Nearest Neighbors (KNN) Imputation: Using KNN to predict missing values based on similar data points.
  - Predictive Imputation: Training a separate model to predict missing values.
- **Removal:** Removing rows or columns with excessive missing values, but only when appropriate and with careful consideration of potential bias.

5.2 OUTLIER DETECTION AND TREATMENT

Outliers can significantly impact model performance. Methods for outlier detection and treatment include:

- **Z-score Method:** Identifying data points with a Z-score above a certain threshold (e.g., 3).
- **IQR Method:** Identifying data points outside the range of Q1 - 1.5 * IQR and Q3 + 1.5 * IQR.
- **Domain-Specific Thresholds:** Using expert knowledge to define acceptable ranges for certain features.

Strategies for handling outliers include capping (setting outlier values to a maximum or minimum threshold), transformation (e.g., log transformation), or removal (with caution).

5.3 FEATURE ENGINEERING

Feature engineering involves creating new, more informative features from existing ones. Examples include:

- Calculating Body Mass Index (BMI) from height and weight.
- Creating age categories (e.g., young adult, middle-aged, senior).
- Combining existing features to create interaction terms.

5.4 FEATURE SCALING

Feature scaling ensures all numerical features contribute equally to the model. Techniques include:

- **Normalization (MinMaxScaler):** Scaling features to a range between 0 and 1.
- **Standardization (StandardScaler):** Scaling features to have a mean of 0 and a standard deviation of 1.

5.5 ENCODING CATEGORICAL VARIABLES

Categorical variables must be encoded into numerical format. Methods include:

- **One-Hot Encoding:** Creating binary columns for each category of nominal variables.

- **Label Encoding:** Assigning a unique integer to each category of ordinal variables.

5.6 DATA SPLITTING

The preprocessed dataset will be divided into distinct sets:

- **Training Set:** Used to train the model.
- **Validation Set:** Used to tune model hyperparameters and assess performance during training.
- **Test Set:** Used to evaluate the final model performance on unseen data.

A typical split ratio is 70% for training, 15% for validation, and 15% for testing.

6. EXPLORATORY DATA ANALYSIS (EDA)

Exploratory Data Analysis (EDA) will be conducted to gain a deep understanding of the dataset's characteristics and the relationships between variables. This process is crucial for informing feature engineering, model selection, and validation strategies.

6.1 DESCRIPTIVE STATISTICS

Descriptive statistics will be computed for all relevant features to summarize their central tendencies and distributions. Key statistics will include:

- Mean, median, and mode to understand typical values.
- Standard deviation and variance to measure data spread.
- Minimum and maximum values to identify the range of each feature.
- Quantiles (e.g., 25th, 50th, 75th percentiles) to understand data distribution.

6.2 UNIVARIATE ANALYSIS

Univariate analysis will involve visualizing the distribution of individual features to identify patterns, skewness, and potential outliers. Common visualization techniques include:

- Histograms to display the frequency distribution of numerical features.
- Box plots to visualize the median, quartiles, and outliers of numerical features.

- Density plots to estimate the probability density function of numerical features.
- Bar charts to display the frequency of categorical features.

6.3 BIVARIATE ANALYSIS

Bivariate analysis will focus on exploring the relationships between pairs of variables, particularly between the features and the target variable (blood pressure stage). Techniques include:

- Scatter plots to visualize the relationship between two numerical features.
- Correlation matrices to quantify the linear relationships between all pairs of numerical features.
- Grouped bar charts to compare the distribution of categorical features across different blood pressure stages.
- Box plots comparing feature distributions across different blood pressure stages.

6.4 DATA VISUALIZATION

Python libraries such as Matplotlib and Seaborn will be extensively used to create compelling and informative visualizations. These visualizations will be crucial for:

- Identifying trends and patterns in the data.
- Detecting outliers and anomalies.
- Validating assumptions about the data.
- Informing feature engineering decisions.
- Communicating insights to stakeholders.

7. MODEL DEVELOPMENT AND TRAINING

This section details the comprehensive process for machine learning model development and training to predict blood pressure stages. Scikit-learn will be the primary library used for implementing these models and techniques.

7.1 MODEL SELECTION

Several machine learning algorithms are suitable for the multi-class classification of blood pressure stages. Potential models include:

- **Logistic Regression:** A linear model suitable for classification tasks, providing probabilities for each class. Its interpretability makes it useful for understanding feature importance.
- **Support Vector Machines (SVM):** Effective in high-dimensional spaces, SVM aims to find a hyperplane that best separates the different blood pressure stages.
- **Random Forest:** An ensemble method that combines multiple decision trees to improve prediction accuracy and robustness. Random Forest can handle non-linear relationships and provide feature importance rankings.
- **Gradient Boosting Classifiers:** Another ensemble method that builds trees sequentially, with each tree correcting errors made by previous trees, often resulting in high accuracy.
- **K-Nearest Neighbors (KNN):** A non-parametric algorithm that classifies data points based on the majority class among their k nearest neighbors.

Model selection will be justified based on data characteristics, interpretability requirements, and performance expectations. The choice will be empirically driven, comparing performance metrics on a validation set.

7.2 TRAINING METHODOLOGY

The training process will employ K-fold cross-validation to ensure the model's robustness and generalization ability. The dataset will be divided into K subsets (folds). The model will be trained on K-1 folds and validated on the remaining fold, repeating this process K times, with each fold serving as the validation set once. The average performance across all folds will be used to evaluate the model's effectiveness. This method provides a more reliable estimate of model performance compared to a single train-test split.

7.3 HYPERPARAMETER TUNING

Hyperparameter tuning is crucial for optimizing model performance. Techniques like GridSearchCV or RandomizedSearchCV will be used. GridSearchCV systematically searches through a pre-defined hyperparameter grid, while RandomizedSearchCV randomly samples hyperparameter

combinations. These techniques will identify the optimal set of hyperparameters that maximize the model's performance on the validation set.

7.4 HANDLING IMBALANCE

If the blood pressure stage classes are imbalanced, strategies to address this will be implemented. These may include:

- **SMOTE (Synthetic Minority Over-sampling Technique):** Generates synthetic samples for the minority classes to balance the class distribution.
- **Oversampling:** Duplicates samples from the minority classes.
- **Undersampling:** Randomly removes samples from the majority classes.

The choice of technique will depend on the specific dataset and the model's sensitivity to class imbalance. Evaluation metrics such as F1-score and area under the precision-recall curve (AUC-PR) will be used to assess performance on imbalanced datasets.

8. MODEL EVALUATION AND SELECTION

Rigorous evaluation is crucial for selecting the best-performing machine learning model. For predicting blood pressure stages (a multi-class classification problem), several key metrics will be employed.

Accuracy, representing the overall proportion of correct predictions, will provide an initial performance overview. However, given potential class imbalances, a more nuanced evaluation is necessary.

Precision, Recall, and F1-score will be calculated for each blood pressure stage. Precision measures the proportion of correctly predicted instances for a given class out of all instances predicted as that class. Recall measures the proportion of correctly predicted instances for a given class out of all actual instances of that class. The F1-score, the harmonic mean of precision and recall, balances both metrics.

A **Confusion Matrix** will visually represent the model's performance on the test set. It will display true positives, true negatives, false positives, and false negatives for each blood pressure stage, offering detailed insights into classification errors.

ROC AUC (Receiver Operating Characteristic Area Under the Curve) will be considered, particularly using one-vs-rest or micro/macro averaging approaches to accommodate the multi-class nature of the problem. It assesses the model's ability to discriminate between classes.

The final model selection will be based on a balanced consideration of these metrics. Prioritizing recall for higher blood pressure stages may be necessary to minimize missed diagnoses, aligning with the project's goal of proactive health management.

9. MODEL DEPLOYMENT STRATEGY

The trained machine learning model will be deployed to make it accessible via the Flask web application. The model will be persisted using Python's `pickle` or `joblib` module, saving the trained model's state to a file. This file will then be loaded within the Flask application's environment during initialization.

A dedicated API endpoint will be created within the Flask framework, which will receive user input via POST requests. The input data will be preprocessed in a manner consistent with the original training pipeline, ensuring data compatibility. This preprocessed data will then be fed to the loaded model for real-time blood pressure stage prediction.

The predicted blood pressure stage will be returned as a structured JSON response to the user interface, enabling easy interpretation and display. Considerations for the deployed model's performance and scalability will involve monitoring response times and resource utilization. Future integration with containerization technologies like Docker could streamline environment management and ensure portability across different deployment environments.

10. WEB APPLICATION DESIGN (FLASK)

The Flask web application will serve as the primary interface for users to interact with the blood pressure prediction model. The application's design focuses on simplicity, intuitiveness, and providing clear, actionable information. The application will consist of three main pages, each designed with a specific purpose in mind, leveraging HTML, CSS, and Jinja2 templating for dynamic content rendering.

10.1 INDEX PAGE (INPUT FORM)

The index page will be the landing page, featuring a clean and user-friendly form. Users will input their health-related data through clearly labeled input fields. These fields will include age, gender, height, weight, existing medical conditions, and lifestyle factors such as smoking status and physical activity level. Both client-side and server-side validation will be implemented to ensure data accuracy and completeness. Immediate client-side validation will provide instant feedback to the user, while server-side validation will ensure data integrity before processing. A prominent submission button will trigger the prediction process, sending the collected data to the model.

10.2 RESULT DETAILS PAGE

This page will display the machine learning model's prediction of the user's blood pressure stage. The prediction will be presented clearly, along with a brief explanation of what the predicted stage means in terms of health implications. Supplementary information, such as a confidence score or probabilities for each possible stage, may also be included to provide a more nuanced understanding of the prediction. The focus will be on delivering actionable information, encouraging users to consult healthcare professionals or adopt healthier lifestyles based on the prediction.

10.3 PROJECT CREATOR DETAILS PAGE

This page will provide essential information about the project's creators, their affiliations, and the overall purpose and vision behind the application. It will also include important disclaimers, emphasizing that the tool provides predictions based on machine learning and is not a substitute for professional medical advice. The disclaimer will strongly advise users to consult with qualified healthcare providers for accurate diagnoses and treatment plans. Contact information or links to the project repository may also be included to facilitate feedback and collaboration.

11. IMPLEMENTATION DETAILS AND BEST PRACTICES

This section highlights crucial implementation considerations and adherence to best practices to ensure a high-quality, maintainable, and reliable system.

Data Preprocessing

Importing The Libraries

```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix, ConfusionMatrixDisplay

# For models
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
```

Read The Dataset

```
# Load dataset
file_path = 'patient_data.csv'
df = pd.read_csv(file_path)

# Display first few rows to understand the data
print("First 5 rows of dataset:")
print(df.head())

# Check dataset info
print("\nDataset Info:")
print(df.info())

# Check for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
First 5 rows of dataset:
        C    Age History Patient TakeMedication Severity
BreathShortness  \
0    Male  18-34     Yes      No             No     Mild
No
1  Female  18-34     Yes      No             No     Mild
No
2    Male  35-50     Yes      No             No     Mild
No
3  Female  35-50     Yes      No             No     Mild
```

```
No
4    Male  51-64     Yes      No             No     Mild
No

  VisualChanges NoseBleeding Whendiagnoused   Systolic Diastolic  \
0            No          No         <1 Year  111 - 120   81 - 90
1            No          No         <1 Year  111 - 120   81 - 90
2            No          No         <1 Year  111 - 120   81 - 90
3            No          No         <1 Year  111 - 120   81 - 90
4            No          No         <1 Year  111 - 120   81 - 90

  ControlledDiet                   Stages
0             No  HYPERTENSION (Stage-1)
1             No  HYPERTENSION (Stage-1)
2             No  HYPERTENSION (Stage-1)
3             No  HYPERTENSION (Stage-1)
4             No  HYPERTENSION (Stage-1)

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1825 entries, 0 to 1824
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   C                1825 non-null   object
 1   Age              1825 non-null   object
 2   History          1825 non-null   object
 3   Patient          1825 non-null   object
 4   TakeMedication   1825 non-null   object
 5   Severity         1825 non-null   object
 6   BreathShortness  1825 non-null   object
 7   VisualChanges    1825 non-null   object
 8   NoseBleeding     1825 non-null   object
 9   Whendiagnoused   1825 non-null   object
 10  Systolic         1825 non-null   object
 11  Diastolic        1825 non-null   object
 12  ControlledDiet   1825 non-null   object
 13  Stages           1825 non-null   object
dtypes: object(14)
memory usage: 199.7+ KB
None

Missing Values:
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
```

Handling Missing Values

```
# Fill missing numerical values with the mean
df.fillna(df.mean(numeric_only=True), inplace=True)

# For categorical columns, fill missing values with the mode
for col in df.select_dtypes(include='object').columns:
    df[col].fillna(df[col].mode()[0], inplace=True)

# Verify that missing values are handled
print("\nMissing Values After Handling:")
print(df.isnull().sum())
```

```
Missing Values After Handling:
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

/tmp/ipython-input-8-4103537396.py:6: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
```

```
original object.


  df[col].fillna(df[col].mode()[0], inplace=True)
```

Handling Categorical Values

```
from sklearn.preprocessing import LabelEncoder

# Encode categorical variables
label_encoders = {}
for col in df.columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    label_encoders[col] = le

# Display the encoded dataset
print("\nEncoded Dataset:")
print(df.head())


Encoded Dataset:
   C  Age  History  Patient  TakeMedication  Severity  BreathShortness
\
0  1    0        1        0               0         0                0

1  0    0        1        0               0         0                0

2  1    1        1        0               0         0                0

3  0    1        1        0               0         0                0

4  1    2        1        0               0         0                0


   VisualChanges  NoseBleeding  Whendiagnoused  Systolic  Diastolic  \
0              0             1               1         1          3
1              0             1               1         1          3
2              0             1               1         1          3
3              0             1               1         1          3
4              0             1               1         1          3

   ControlledDiet  Stages
0               0       0
1               0       0
2               0       0
3               0       0
4               0       0
```

Label

```
from collections import Counter
from sklearn.model_selection import train_test_split

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

# Remove rare classes (e.g., classes with fewer than 2 samples)
min_samples_per_class = 2
class_counts = Counter(y)
valid_indices = [i for i, label in enumerate(y) if class_counts[label]
>= min_samples_per_class]

# Filter the data using iloc and then reset the index
X_filtered = X.iloc[valid_indices].reset_index(drop=True)
y_filtered = y.iloc[valid_indices].reset_index(drop=True)

# Perform stratified train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X_filtered, y_filtered, test_size=0.2, random_state=42,
stratify=y_filtered
)

# Print class distribution after filtering
print("\nClass Distribution After Filtering:")
print(Counter(y_filtered))
```

```
Class Distribution After Filtering:
Counter({'HYPERTENSION (Stage-1)': 648, 'HYPERTENSION (Stage-2)': 599,
'NORMAL': 336, 'HYPERTENSIVE CRISIS': 240})
```

Exploratory Data Analysis (EDA)

Descriptive Statistical Analysis

```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

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

# Display first few rows
print("First 5 rows of dataset:")
print(df.head())
```

```python
# Check dataset info
print("\nDataset Info:")
print(df.info())

# Summary statistics
print("\nSummary Statistics:")
print(df.describe())
```

```
First 5 rows of dataset:
        C    Age History Patient TakeMedication Severity
BreathShortness  \
0    Male  18-34     Yes      No             No     Mild
No
1  Female  18-34     Yes      No             No     Mild
No
2    Male  35-50     Yes      No             No     Mild
No
3  Female  35-50     Yes      No             No     Mild
No
4    Male  51-64     Yes      No             No     Mild
No

  VisualChanges NoseBleeding Whendiagnoused   Systolic Diastolic  \
0            No          No         <1 Year  111 - 120   81 - 90
1            No          No         <1 Year  111 - 120   81 - 90
2            No          No         <1 Year  111 - 120   81 - 90
3            No          No         <1 Year  111 - 120   81 - 90
4            No          No         <1 Year  111 - 120   81 - 90

  ControlledDiet                   Stages
0             No  HYPERTENSION (Stage-1)
1             No  HYPERTENSION (Stage-1)
2             No  HYPERTENSION (Stage-1)
3             No  HYPERTENSION (Stage-1)
4             No  HYPERTENSION (Stage-1)

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1825 entries, 0 to 1824
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   C                1825 non-null   object
 1   Age              1825 non-null   object
 2   History          1825 non-null   object
 3   Patient          1825 non-null   object
 4   TakeMedication   1825 non-null   object
 5   Severity         1825 non-null   object
 6   BreathShortness  1825 non-null   object
 7   VisualChanges    1825 non-null   object
```

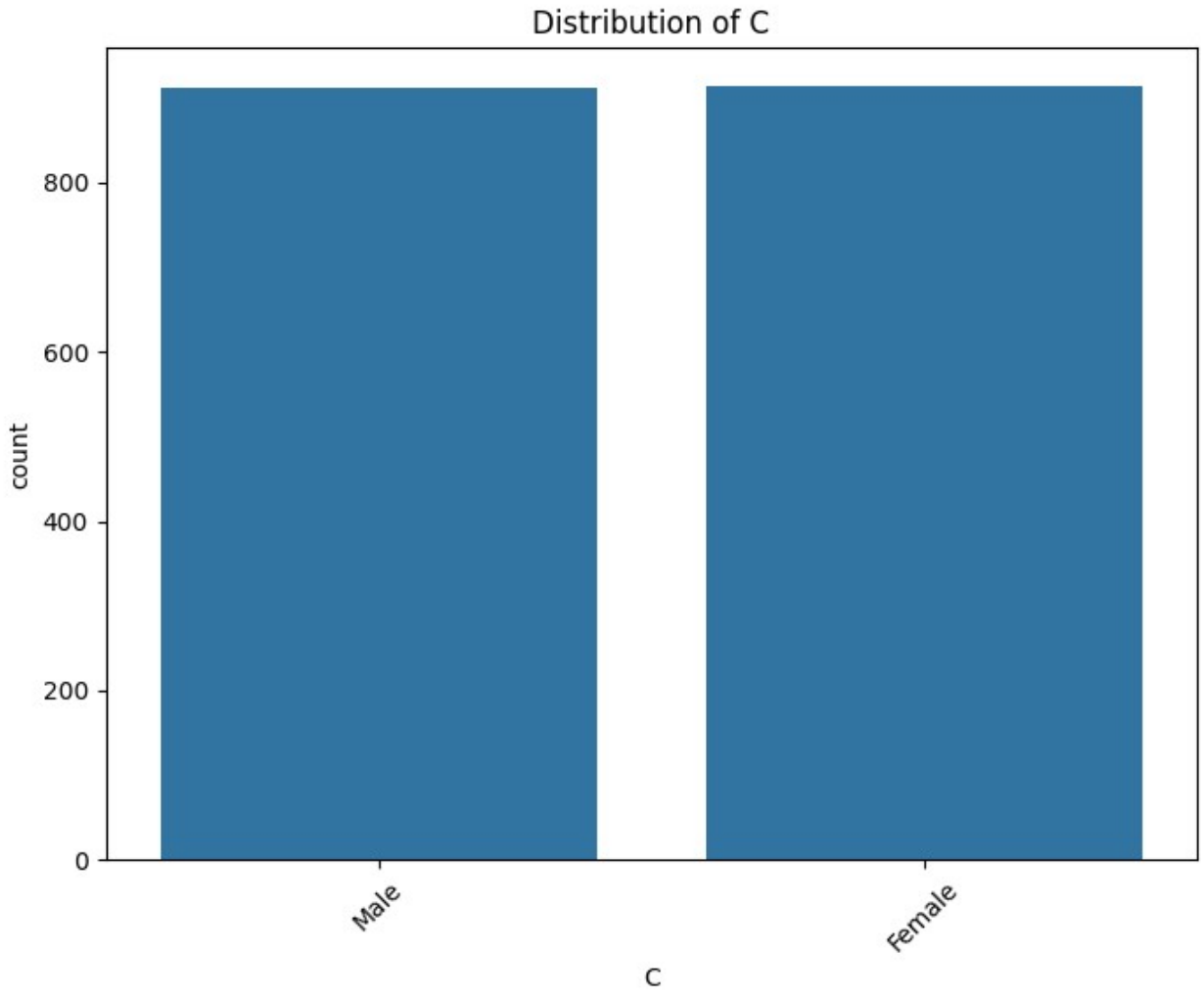
```
 8   NoseBleeding     1825 non-null   object
 9   Whendiagnoused   1825 non-null   object
 10  Systolic         1825 non-null   object
 11  Diastolic        1825 non-null   object
 12  ControlledDiet   1825 non-null   object
 13  Stages           1825 non-null   object
dtypes: object(14)
memory usage: 199.7+ KB
None

Summary Statistics:
              C    Age History Patient TakeMedication  Severity  \
count     1825   1825    1825    1825           1825      1825
unique       2      4       2       2              3         3
top     Female  51-64     Yes      No             No  Moderate
freq       913    475    1657     984            744       697

       BreathShortness VisualChanges NoseBleeding Whendiagnoused
Systolic  \
count             1825          1825         1825           1825
1825
unique               2             2            3              3
5
top                 No            No           No         <1 Year  111
- 120
freq              976           940          984            625
1008

       Diastolic ControlledDiet                  Stages
count       1825           1825                    1825
unique         5              2                       6
top      81 - 90             No  HYPERTENSION (Stage-1)
freq         708            984                     648
```

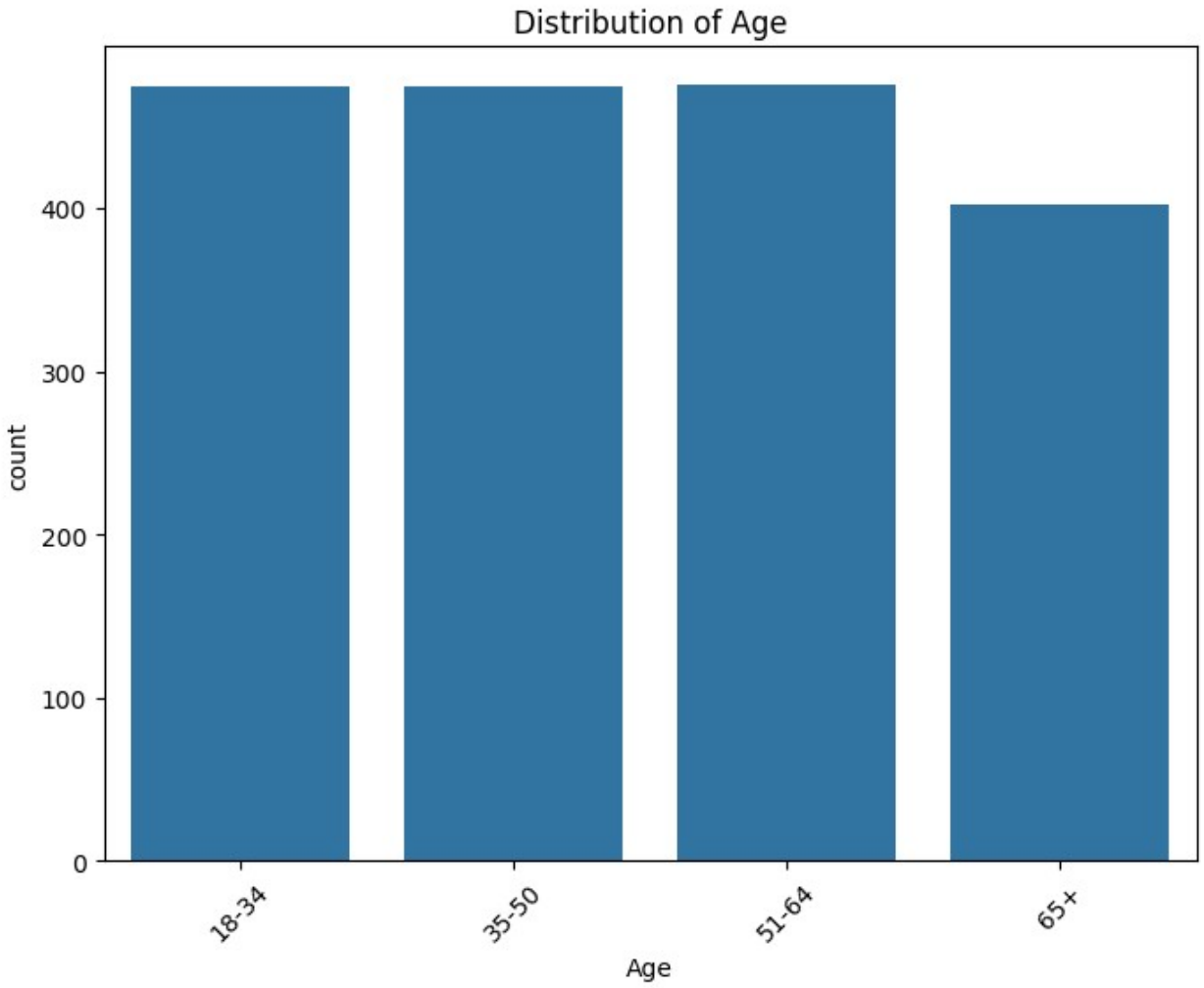
Visual Analysis

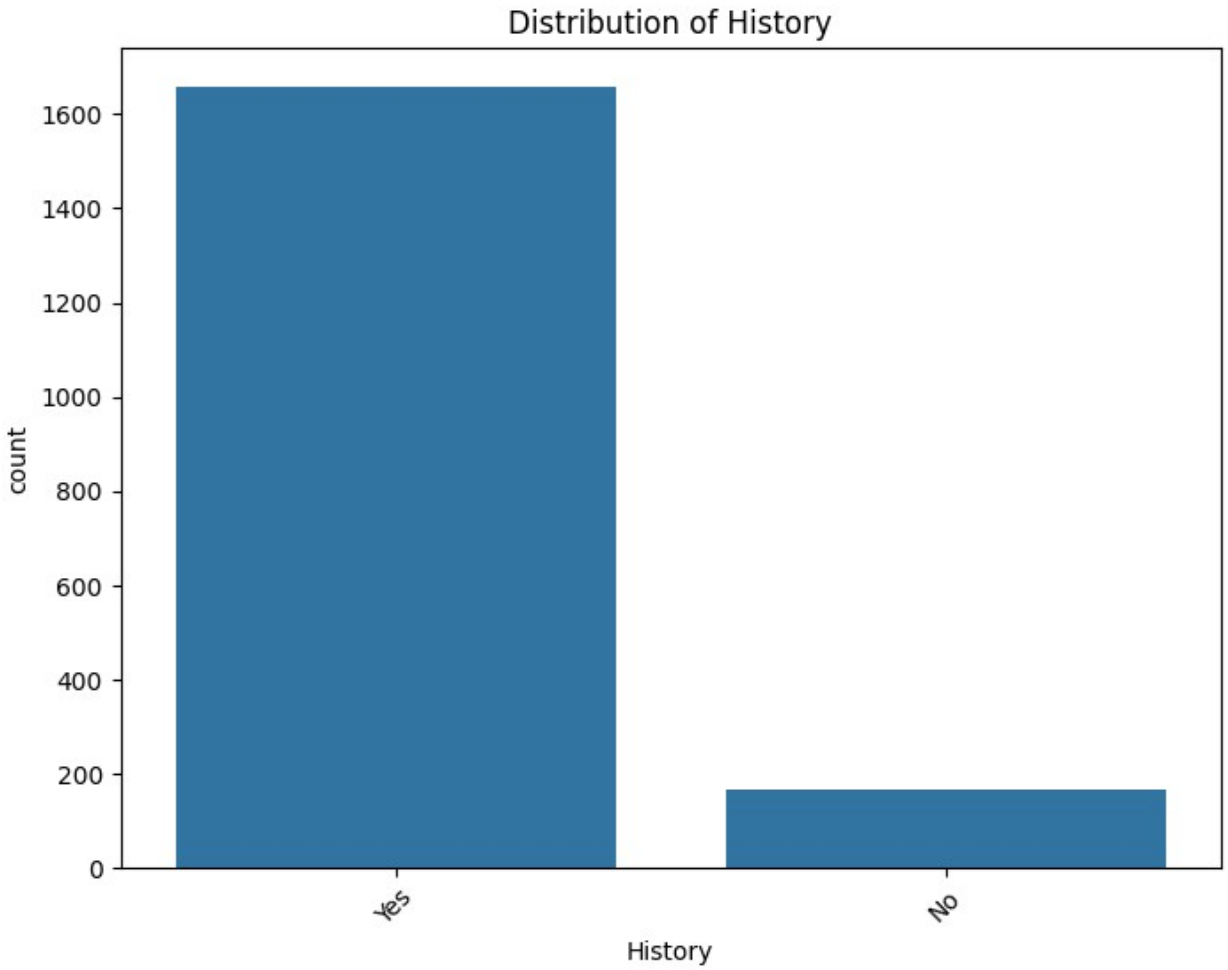
Univariate Analysis

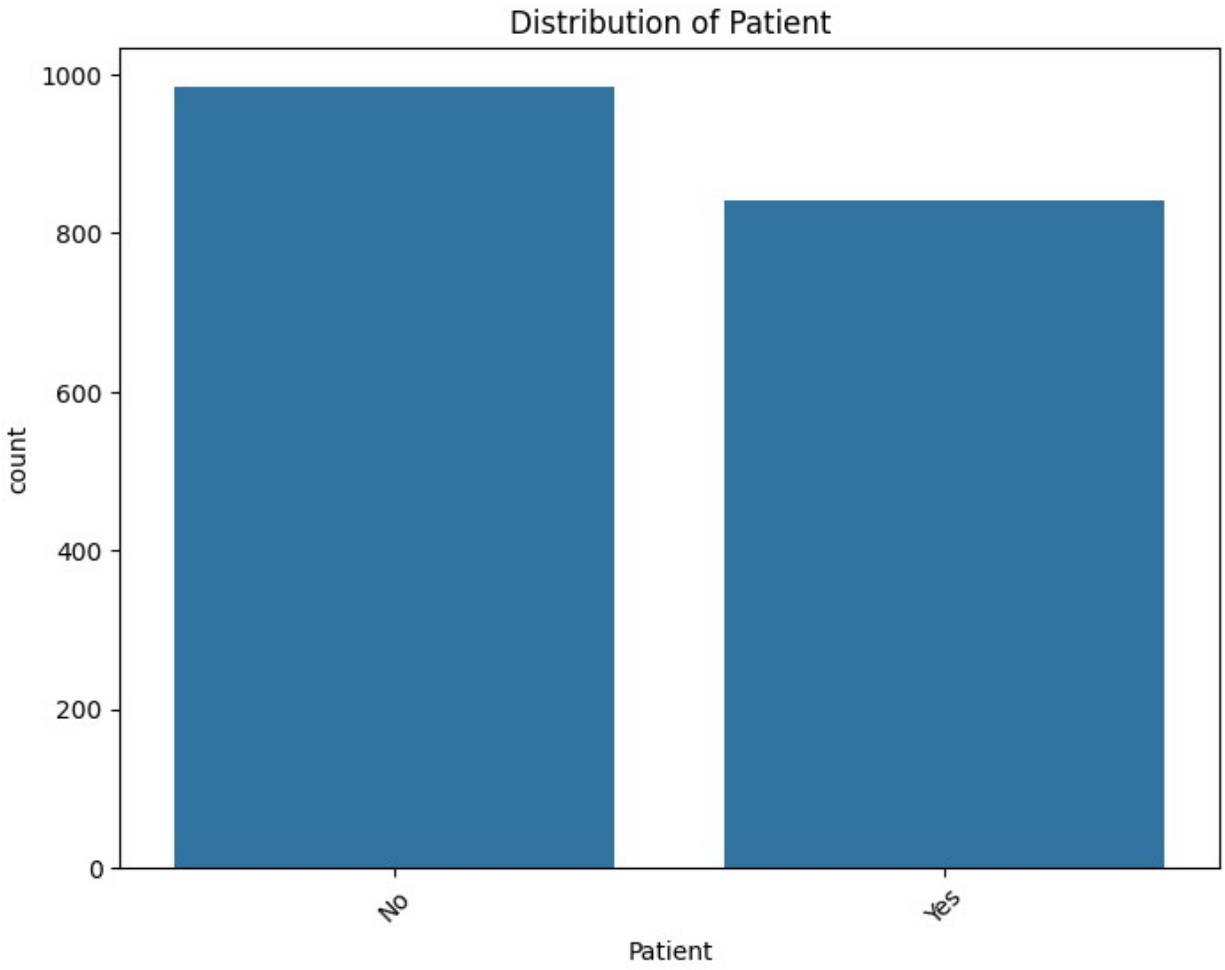
```
# Univariate analysis for categorical variables
categorical_cols = df.select_dtypes(include=['object']).columns
for col in categorical_cols:
    plt.figure(figsize=(8, 6))
    sns.countplot(x=col, data=df)
    plt.title(f'Distribution of {col}')
    plt.xticks(rotation=45)
    plt.show()

# Univariate analysis for numerical variables
numerical_cols = df.select_dtypes(include=['number']).columns
```

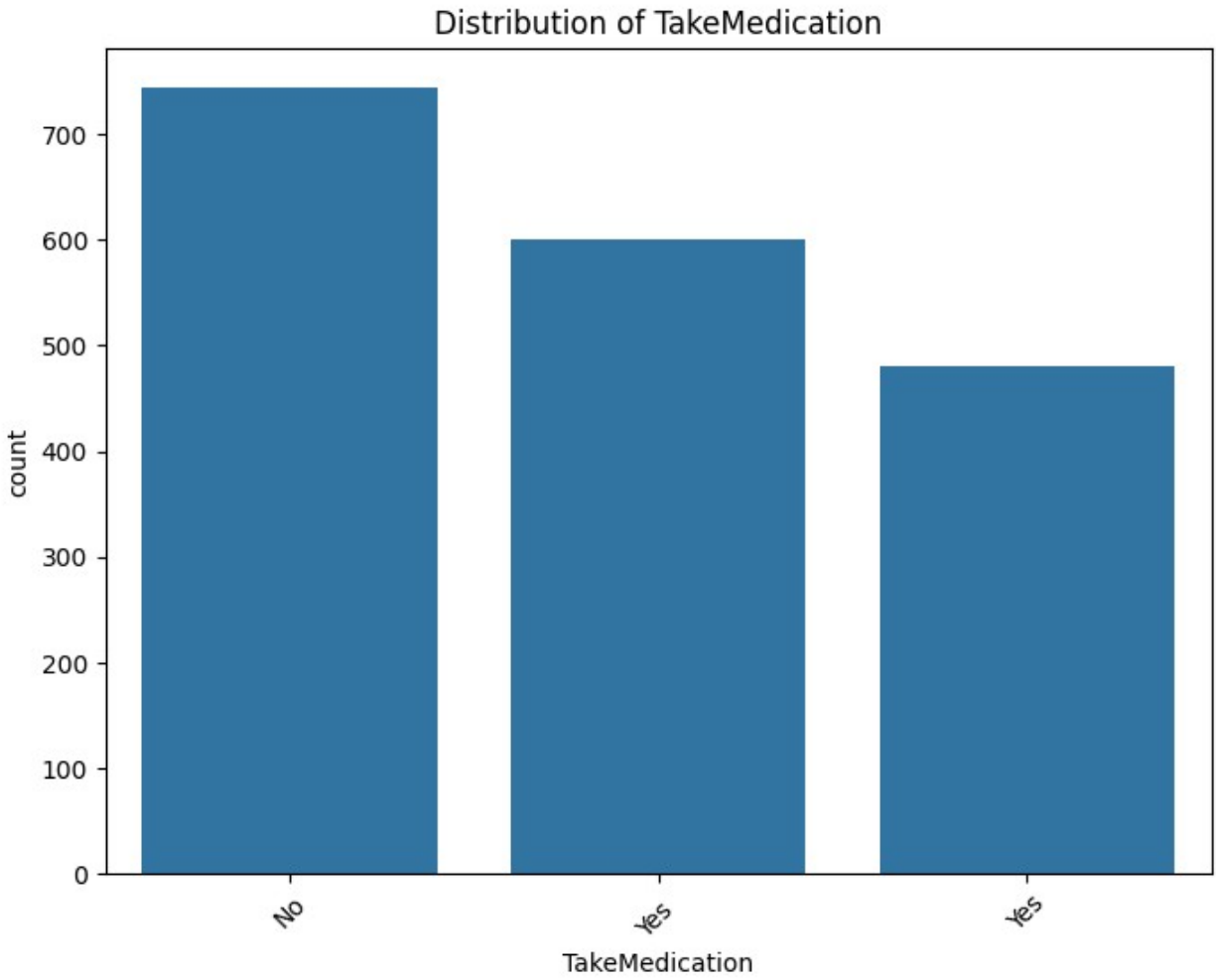
```
for col in numerical_cols:
    plt.figure(figsize=(8, 6))
    sns.histplot(data=df, x=col, kde=True)
    plt.title(f'Distribution of {col}')
    plt.show()
```

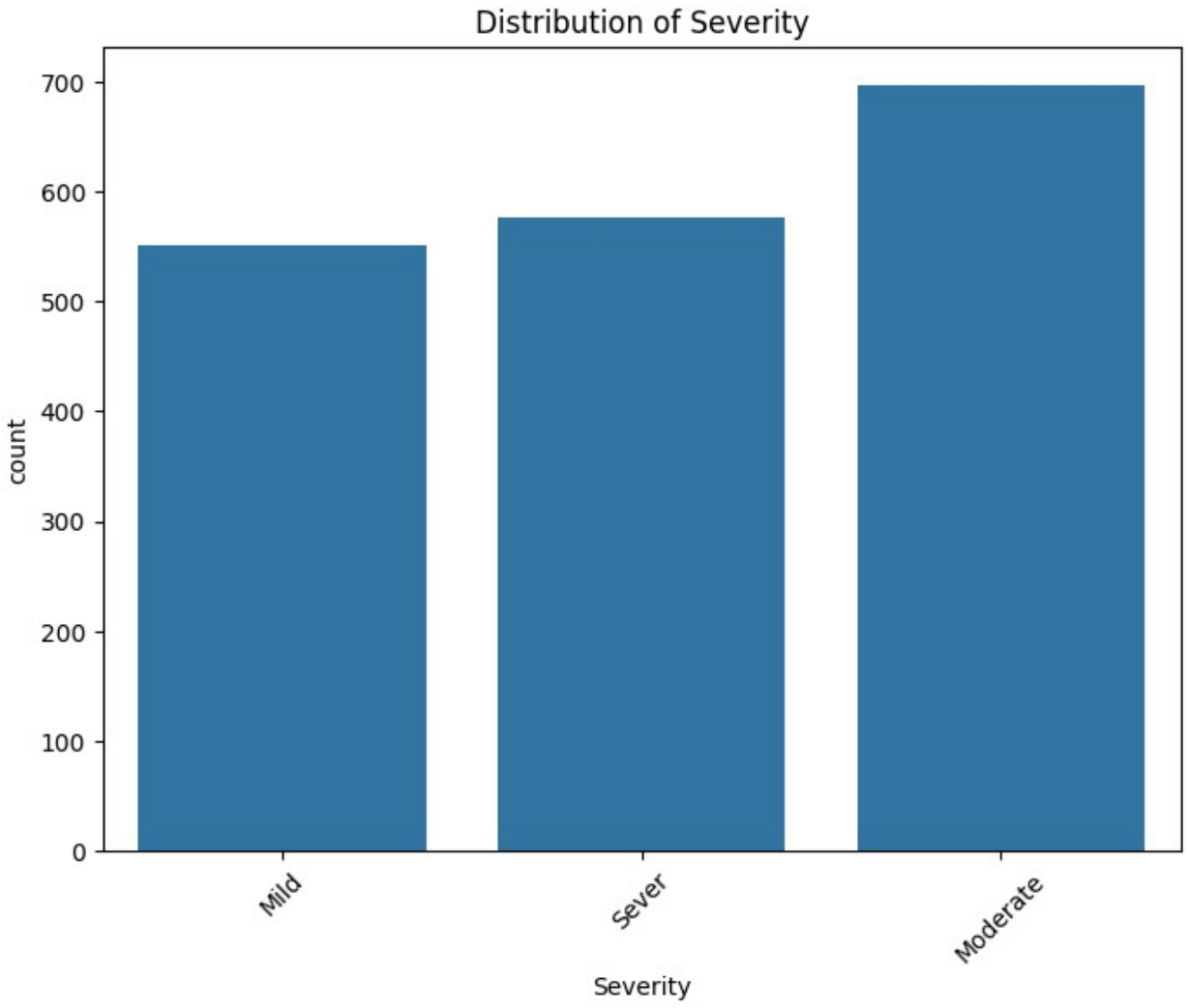


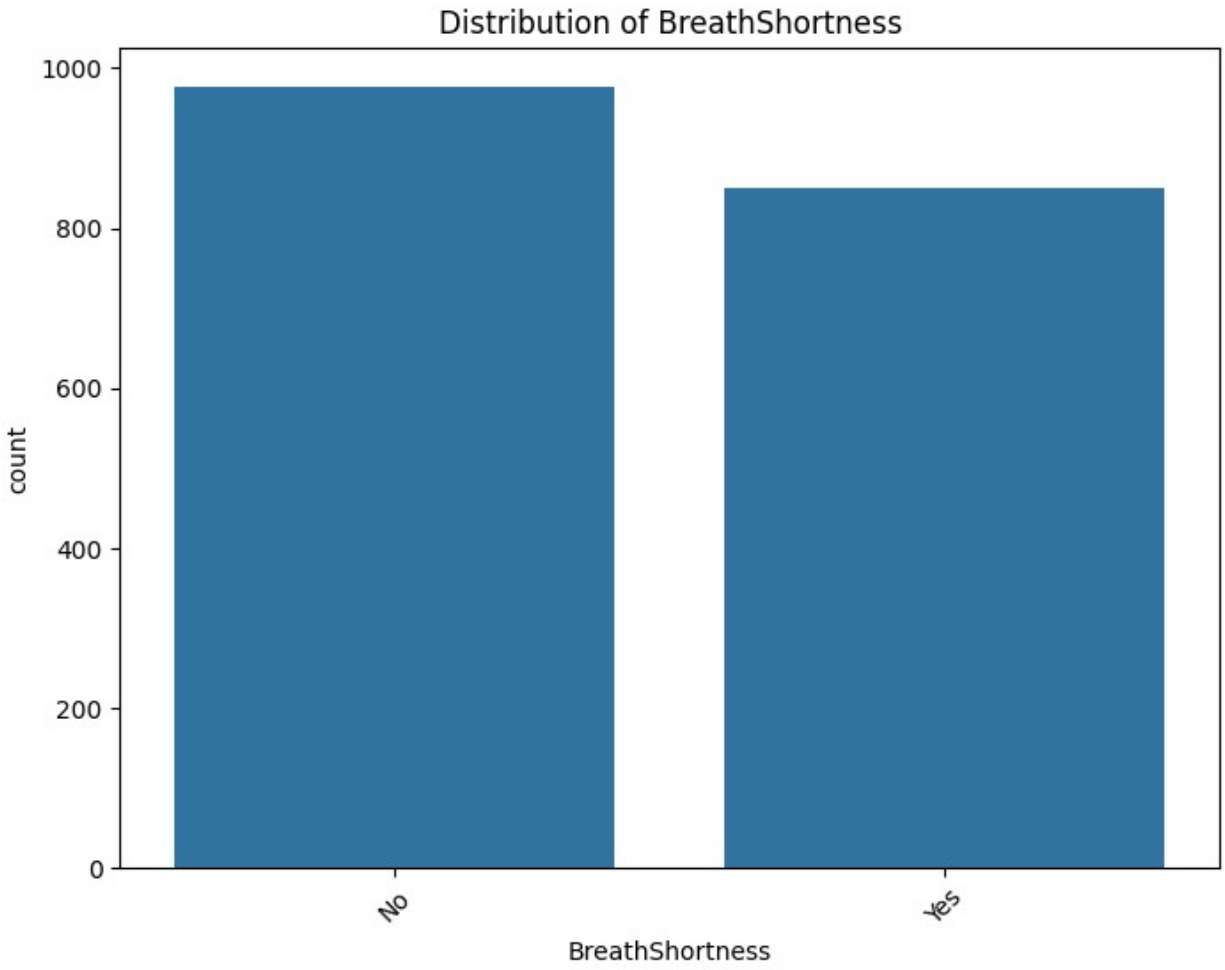

Distribution of Age
count
0
100
200
300
400
18-34
35-50
51-64
65+
Age

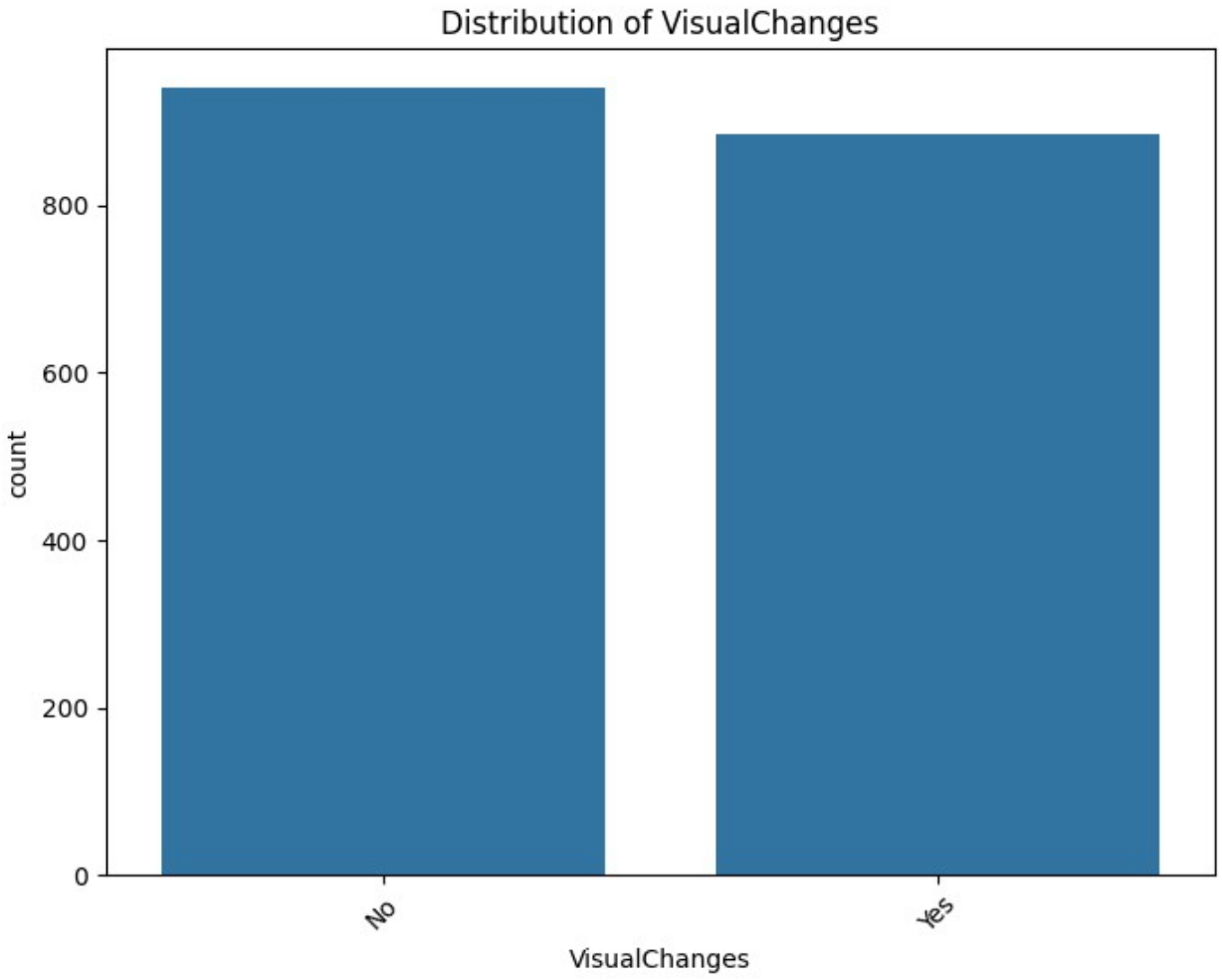

Distribution of History
1600
1400
1200
1000
800
600
400
200
0
count
Yes
No
History

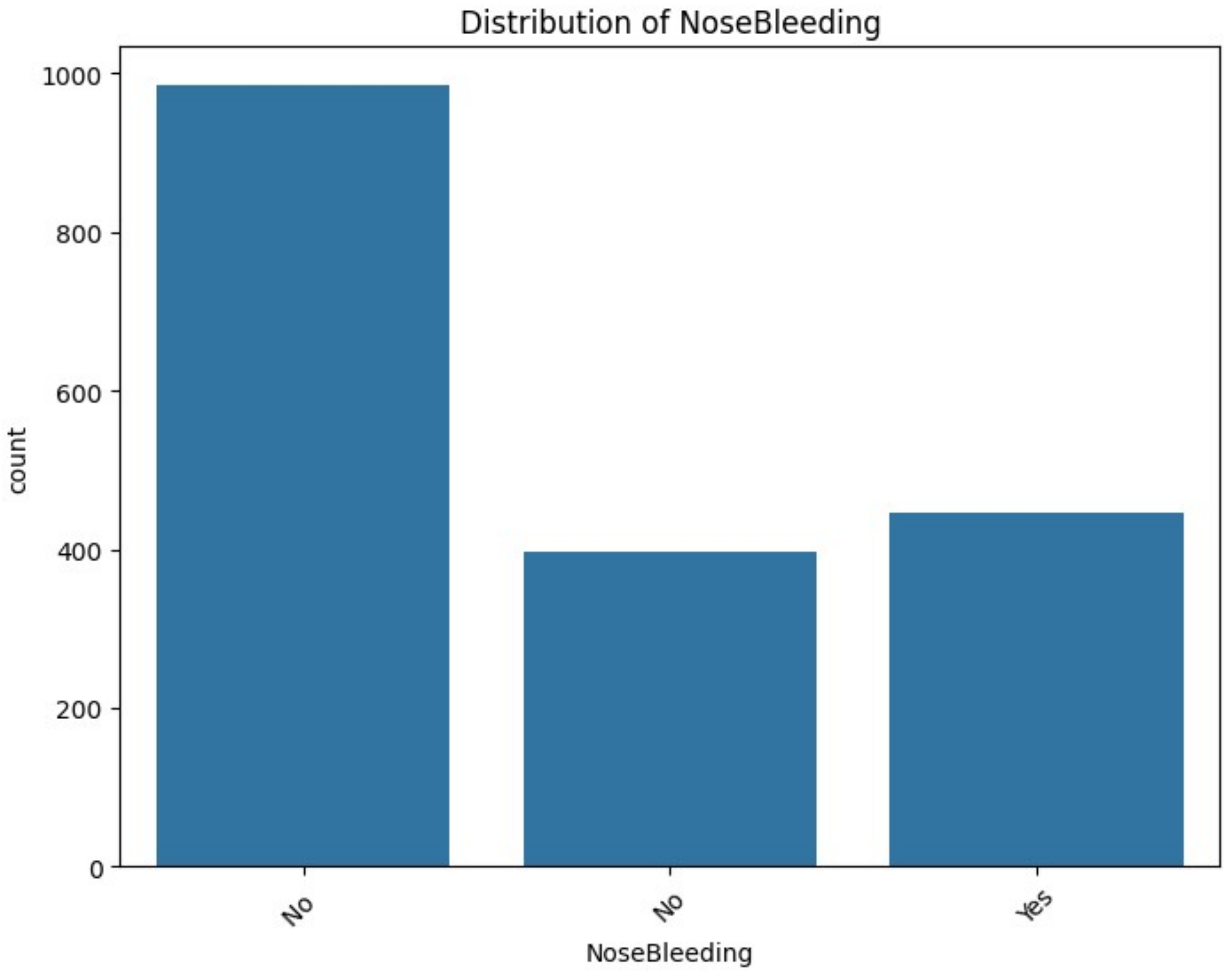

Distribution of Patient
1000
800
600
400
200
0
count
No
Yes
Patient

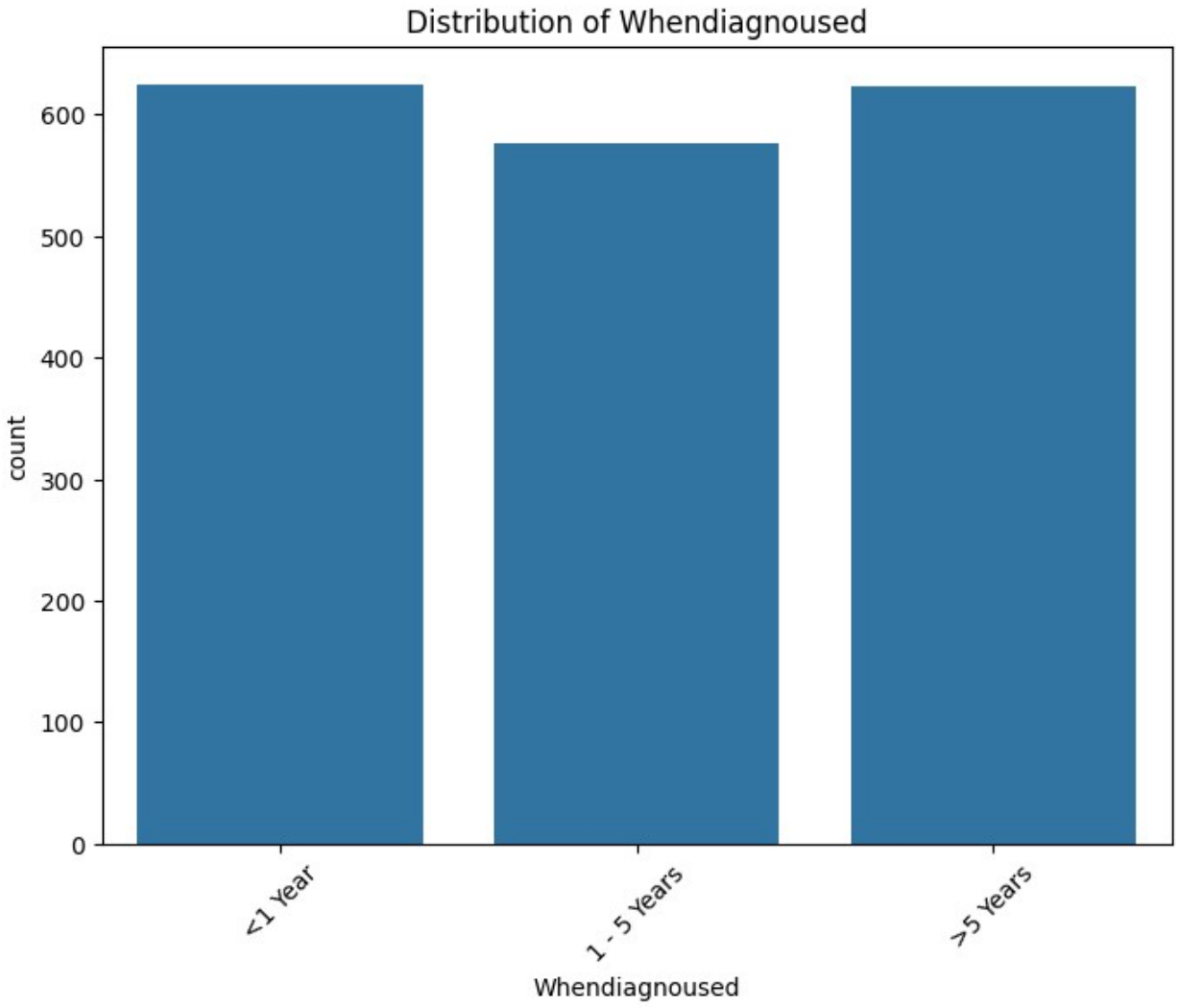

Distribution of TakeMedication
700
600
500
400
300
200
100
0
count
No
Yes
Yes
TakeMedication

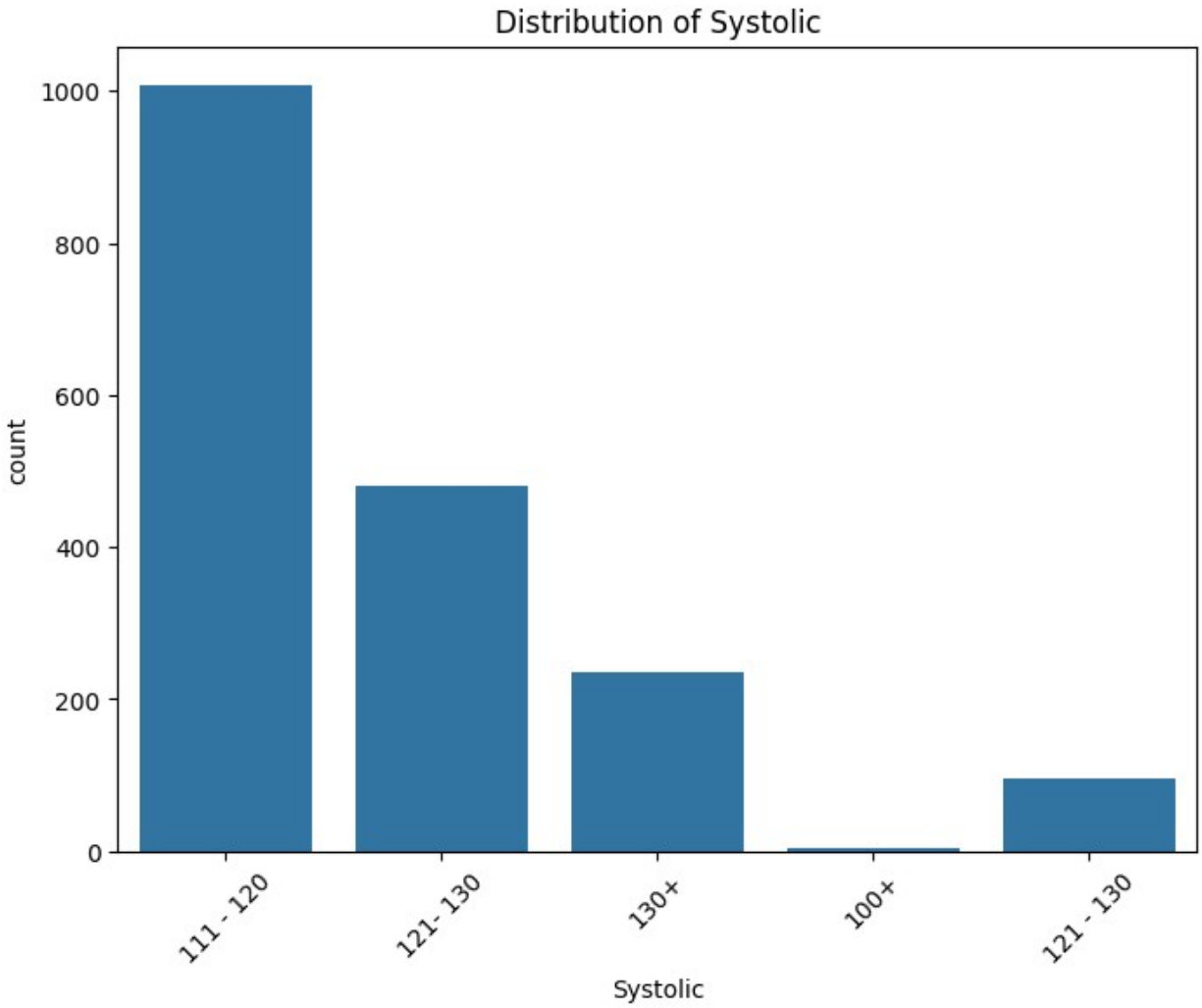

Distribution of Severity
700
600
500
400
300
200
100
0
count
Mild
Sever
Moderate
Severity

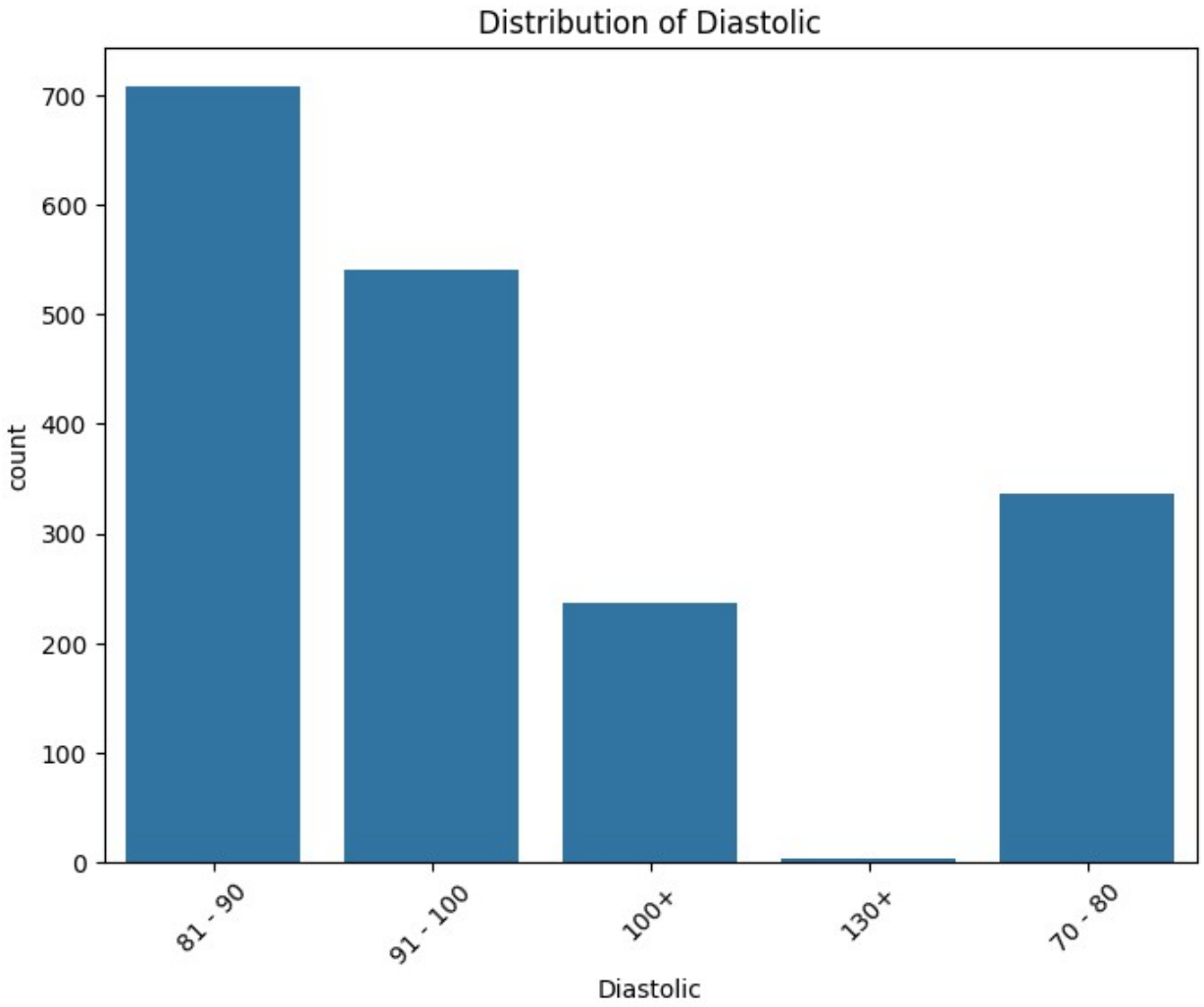

Distribution of BreathShortness
1000
800
600
400
200
0
count
No
Yes
BreathShortness

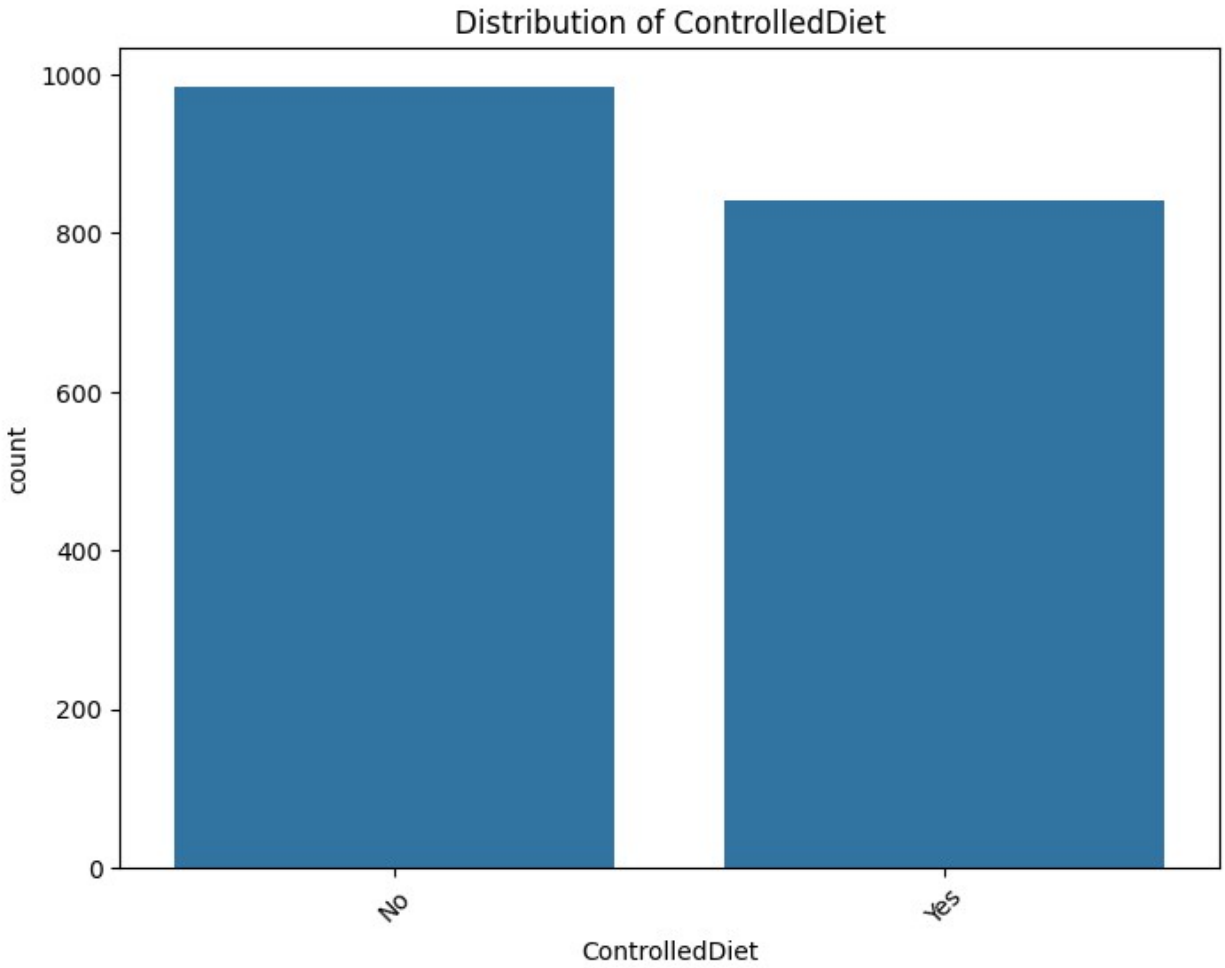

Distribution of VisualChanges
count
800
600
400
200
0
No
Yes
VisualChanges

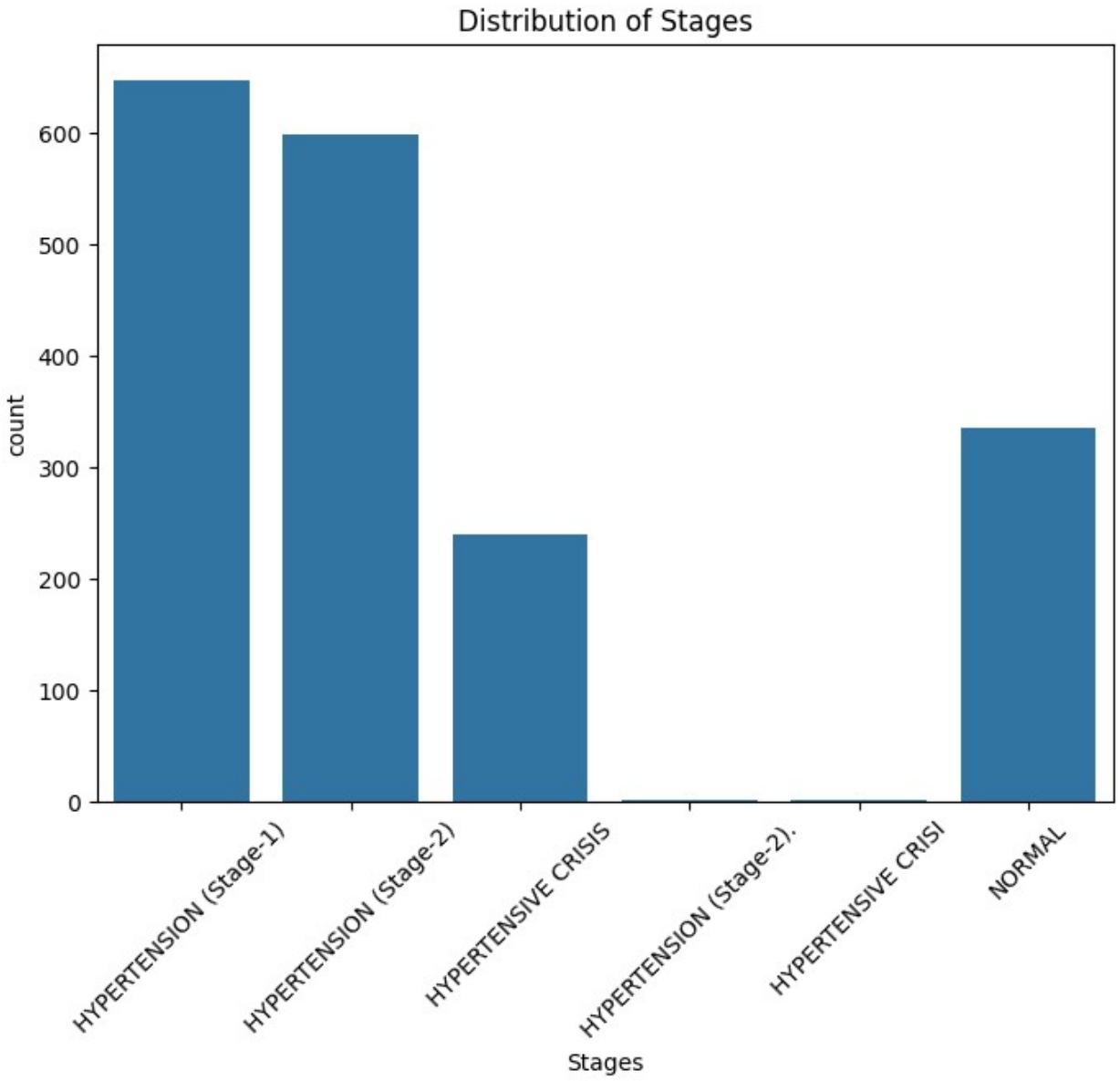

Distribution of NoseBleeding
1000
800
600
400
200
0
count
No
No
Yes
NoseBleeding


Distribution of Whendiagnoused
count
600
500
400
300
200
100
0
<1 Year
1 - 5 Years
>5 Years
Whendiagnoused


Distribution of Systolic
1000
800
600
400
200
0
count
111 - 120
121- 130
130+
100+
121 - 130
Systolic


Distribution of Diastolic
count
700
600
500
400
300
200
100
0
81 - 90
91 - 100
100+
130+
70 - 80
Diastolic

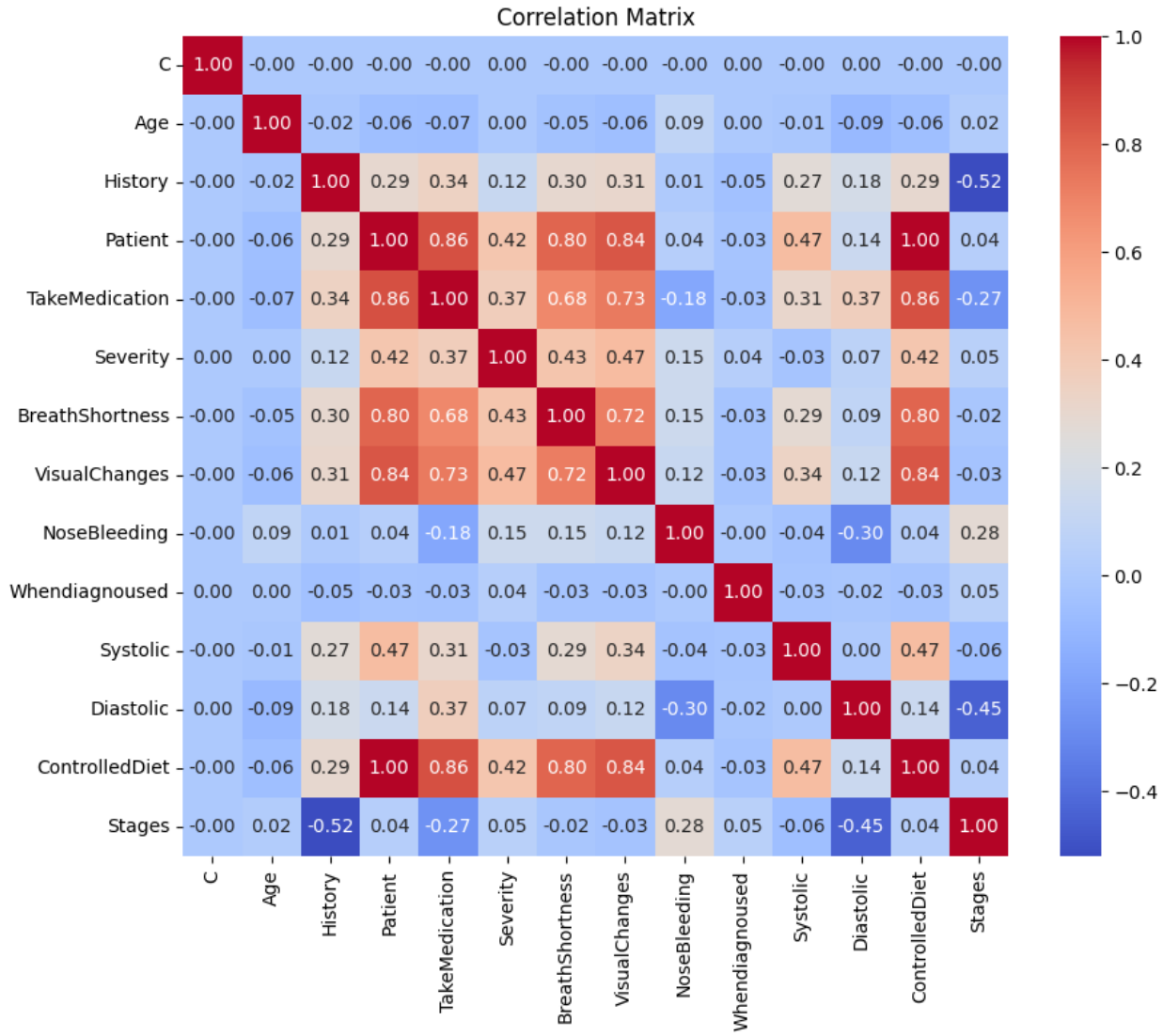

Distribution of ControlledDiet
1000
800
600
400
200
0
count
No
Yes
ControlledDiet



Bivariate Analysis

```
from sklearn.preprocessing import LabelEncoder

# Encode categorical columns
label_encoders = {}
for col in df.columns:
    if df[col].dtype == 'object':  # Check if column is categorical
        le = LabelEncoder()
        df[col] = le.fit_transform(df[col])
        label_encoders[col] = le

# Now compute correlation matrix
correlation_matrix = df.corr()
```

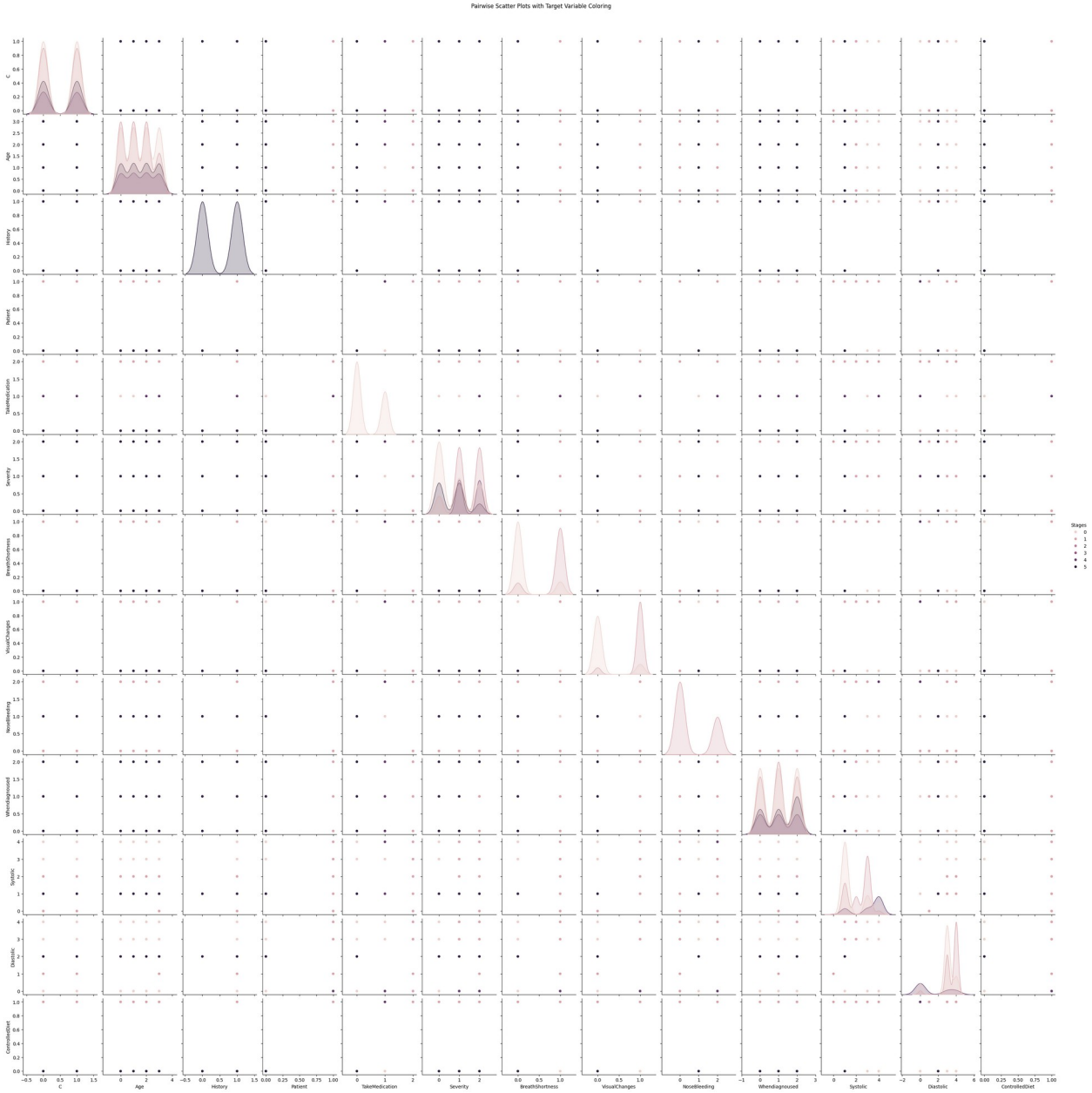
```
# Plot heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```



Multivariate analysis

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

# Select only numeric columns for pairwise scatter plots
numeric_df = df.select_dtypes(include=['number'])
```

```
# Pairplot to visualize pairwise relationships
sns.pairplot(numeric_df, hue='Stages', diag_kind='kde')  # Use 'hue' if you have a target variable
plt.suptitle('Pairwise Scatter Plots with Target Variable Coloring', y=1.02)
plt.show()
```



Model Building

```
# Import required libraries
import pandas as pd
import numpy as np
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestRegressor
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

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

# Encode categorical variables
label_encoders = {}
for col in df.columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    label_encoders[col] = le

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

# Remove rare classes (if any)
min_samples_per_class = 2
class_counts = Counter(y)
valid_indices = [i for i, label in enumerate(y) if class_counts[label]
>= min_samples_per_class]

X_filtered = X.iloc[valid_indices].reset_index(drop=True)
y_filtered = y.iloc[valid_indices].reset_index(drop=True)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_filtered,
y_filtered, test_size=0.2, random_state=42, stratify=y_filtered)

# Define models
models = {
    "Logistic Regression": LogisticRegression(random_state=42),
    "Random Forest Regressor": RandomForestRegressor(random_state=42),
    "Decision Tree": DecisionTreeClassifier(random_state=42),
    "Gaussian Naïve Bayes": GaussianNB(),
    "Multinomial Naïve Bayes": MultinomialNB()
}

# Train and evaluate each model
results = {}

for name, model in models.items():
    print(f"\nTraining {name}...")
```

```
    # Handle MultinomialNB: Ensure all features are non-negative
    if name == "Multinomial Naïve Bayes":
        # MultinomialNB requires non-negative features
        X_train_non_negative = X_train.clip(lower=0)
        X_test_non_negative = X_test.clip(lower=0)

        model.fit(X_train_non_negative, y_train)
        y_pred = model.predict(X_test_non_negative)
    else:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    results[name] = {"model": model, "accuracy": acc}

    print(f"{name} Accuracy: {acc:.4f}")
    print(classification_report(y_test, y_pred))
    print("-" * 60)

# Select best model
best_model_name = max(results, key=lambda k: results[k]['accuracy'])
best_model = results[best_model_name]['model']
print(f"\n Best Model: {best_model_name} with Accuracy:
{results[best_model_name]['accuracy']:.4f}")

# Save best model
import joblib
joblib.dump(best_model, 'best_model.pkl')
print("\nModel saved as 'best_model.pkl'")


Training Logistic Regression...
Logistic Regression Accuracy: 0.9753
              precision    recall  f1-score   support

           0       0.96      0.97      0.97       130
           1       1.00      1.00      1.00       120
           4       1.00      1.00      1.00        48
           5       0.94      0.93      0.93        67

    accuracy                           0.98       365
   macro avg       0.98      0.97      0.97       365
weighted avg       0.98      0.98      0.98       365

------------------------------------------------------------

Training Random Forest Regressor...
Random Forest Regressor Accuracy: 1.0000
              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00       130
           1       1.00      1.00      1.00       120
           4       1.00      1.00      1.00        48
           5       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

------------------------------------------------------------

Training Decision Tree...
Decision Tree Accuracy: 1.0000
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       130
           1       1.00      1.00      1.00       120
           4       1.00      1.00      1.00        48
           5       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

------------------------------------------------------------

Training Gaussian Naïve Bayes...
Gaussian Naïve Bayes Accuracy: 1.0000
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       130
           1       1.00      1.00      1.00       120
           4       1.00      1.00      1.00        48
           5       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

------------------------------------------------------------

Training Multinomial Naïve Bayes...
Multinomial Naïve Bayes Accuracy: 0.8219
              precision    recall  f1-score   support

           0       0.81      0.94      0.87       130
           1       0.84      0.94      0.89       120
           4       0.79      0.56      0.66        48
           5       0.84      0.57      0.68        67
```

```
    accuracy                           0.82       365
   macro avg       0.82      0.75      0.77       365
weighted avg       0.82      0.82      0.81       365

------------------------------------------------------------

 Best Model: Random Forest Regressor with Accuracy: 1.0000

Model saved as 'best_model.pkl'
```

11.1 PYTHON BEST PRACTICES

Strict adherence to PEP 8 style guidelines is mandatory for code readability and consistency. Comprehensive code documentation using docstrings for modules, classes, and functions is essential. Modular code design will enhance maintainability and reusability. Efficient algorithm selection will be prioritized for performance.

11.2 COMPREHENSIVE ERROR HANDLING

Robust error handling strategies will be implemented throughout the system. This includes utilizing `try-except` blocks for API calls in the Flask web application, handling file I/O errors and data type mismatches in the data processing pipeline, and implementing custom error pages for an improved user experience. Detailed logging will be incorporated for effective debugging.

11.3 SOFTWARE VERSION DOCUMENTATION

Documenting all major library versions (e.g., `pandas`, `scikit-learn`, `Flask`, `NumPy`, `Matplotlib`, `Seaborn`) in a `requirements.txt` file is critical. This ensures reproducibility of the environment and compatibility across development and deployment stages. Use `pip freeze > requirements.txt` to capture the current environment.

11.4 TESTING STRATEGY

Unit tests will be developed for individual functions and modules using the `unittest` or `pytest` frameworks. Integration tests will verify the seamless interaction between different components of the system. Testing will be continuously integrated into the development process to ensure code quality and system reliability.

12. FUTURE ENHANCEMENTS AND SCALABILITY

The blood pressure predictive modeling project has significant potential for future enhancements and scalability. Integrating diverse data sources, such as wearable device data and electronic health records, could improve prediction accuracy. Implementing a CI/CD pipeline would automate model retraining and deployment, ensuring continuous improvement.

Exploring advanced deep learning models may capture complex relationships in the data. Adding user accounts and personalized health dashboards could provide tailored insights. Expanding predictive capabilities to other health metrics offers a broader health assessment tool.

Deployment on cloud platforms like AWS or Azure would increase availability and scalability. The current architecture, with its modular design and API-driven approach, facilitates these extensions. The Flask application can be containerized using Docker and orchestrated with Kubernetes for scalable deployment.

13. CONCLUSION

In conclusion, this document outlines a comprehensive and systematic plan for developing a predictive model for blood pressure stages. This project holds significant potential for contributing to proactive health management, offering a valuable tool for early identification of individuals at risk.

The outlined methodology covers every critical phase, from meticulous data preprocessing and exploratory analysis to advanced model training and deployment through a user-friendly Flask web application. Adherence to best practices in software development and machine learning ensures a reliable, maintainable, and effective solution.

By integrating robust error handling, detailed documentation, and a commitment to continuous improvement, this project aims to deliver a high-quality predictive model that empowers individuals to take control of their health.