

## **Medical Center app**

**By**

• Carol Moussa Eucladius	(2100588)
• Aml Mahrous Habeb	(2100485)
• Nermen Awny Abdelmaseh	(2101359)
• Wafaa Ashraf Mohamed	(2101675)
• Sondos Abdelmohsen Abdelnafea	(2101647)
• Essam Mokhtar Nagy	(2100636)
• Mario Melad Sagher	(2101021)

**Supervised by:**

**Assoc. Prof / Mohamed M. Ashour**

**Assistant:**

**Eng. Nourhan Khashaba**

[Assuit]-2025

# Abstract

This project aims to develop a comprehensive mobile application called "Medical Center" that helps patients easily access various healthcare services through an intelligent and user-friendly interface. The application integrates multiple features including appointment booking with specialized doctors, home nursing services, health education through medical videos, and disease prediction based on symptoms using machine learning.

By leveraging modern technologies such as **Flutter**, **.NET**, and **Python-based ML models**, the application contributes to improving the accessibility and quality of healthcare, especially for users who face difficulties navigating multiple medical systems. The system is designed to be inclusive, scalable, and responsive to real patient needs, offering a new standard in mobile healthcare delivery.

## **Acknowledgments**

**We would like to express our heartfelt gratitude to our supervisor Assoc. Prof / Mohamed M. Ashour and assistant T.A. Nourhan Khashaba for their continuous support, valuable insights, and guidance throughout the development of our graduation project. Their expertise and encouragement have been instrumental in the successful implementation of the Medical Center application. We also thank our university, Egyptian E-Learning University (EELU), for providing us with the tools and knowledge necessary to bring this idea to life**

# Contents

Chapter 1: Introduction .....	7
1.1 Introduction.....	7
1.2 Background and Motivation for the Project.....	7
1.3 Importance of the Problem Being Addressed .....	8
1.4 Problem Statement.....	9
1.5 Objectives .....	10
1.5.1 Main Objective.....	10
1.5.2 Specific Objectives .....	10
1.6 Brief Overview of the Proposed Solution.....	11
Chapter 2 : Literature Review / Related Work.....	12
2.1 Existing Research and Technologies.....	12
2.2 Gaps in Current Solutions.....	13
2.3 Relevance of the Proposed Solution .....	14
2.4 Comparison with Existing Healthcare Applications.....	15
<b>Summary</b> .....	19
Chapter 3: Proposed System .....	21
<b>3.1 Approach Used to Solve the Problem</b> .....	21
3.2 System Architecture .....	22
3.3 System Diagram.....	24
3.3.1 :ERD Digram .....	24
3.3.2 UML Use Case Diagram.....	25

3.3.3 :UML class diagram:.....	26
3.3.4 :UML Sequence Diagram For Admin Login: .....	27
3.3.5 Sequence Diagram Of Meical Center Management: .....	28
3.3.6 UML Sequence Diagram: .....	29
3.3.7 Data flow diagram: .....	30
3.4Summary .....	31
3.5Algorithms or Frameworks Used.....	33
3.5.1Algorithm Comparison and Selection.....	33
3.5.2Frameworks Used .....	36
Chapter 4: Implementation .....	38
4.1 Technologies, Tools, and Programming Languages Used .....	38
4.2 Key Components/Modules of the System.....	39
4.2.1 User Interface Design .....	41
Splash and welcome Screen.....	41
Login & Regstration .....	42
Home Screen.....	44
Clinics specialties.....	45
Doctor Appointment Booking.....	46
Nurse Appointment .....	47
Symptom Checker.....	48
Health awareness .....	49
4.2.2 Machine Learning Model Implementation .....	50
Explanation of the code .....	53
4.2.3 Machine Learning Model Integration .....	75
4.2.4Backend Integration:.....	79

4.3 Challenges Faced and How They Were Resolved .....	83
Chapter 5 : Testing & Evaluation.....	85
5.1 Testing Strategies .....	85
5.2 Performance Metrics .....	86
Chapter 6: Results & Discussion .....	87
<b>6.1 Introduction</b> .....	87
<b>6.2 Summary of Findings</b> .....	87
<b>6.3 Interpretation of Results (Did the Project Meet its Objectives?)</b> .....	87
<b>6.4 Limitations of the Proposed Solution</b> .....	88
Chapter 7 : Conclusion & Future Work .....	90
7.1 Summary of Contributions.....	90
7.2 Future Work .....	91
Chapter 8 References: .....	93

# **Chapter 1: Introduction**

## **1.1 Introduction**

In today's fast-paced world, accessing reliable and timely healthcare services has become more essential than ever. With the increasing dependency on digital solutions, mobile health applications have emerged as an effective tool to bridge the gap between patients and healthcare providers. This project aims to build a smart medical center mobile application that not only facilitates appointment booking and health education but also utilizes machine learning techniques to provide symptom-based disease prediction, empowering users to make informed decisions about their health.

---

## **1.2 Background and Motivation for the Project**

The healthcare sector, particularly in developing countries, often faces challenges such as limited access to specialists, long waiting times, and a lack of early disease detection tools. Many people delay seeking medical attention due to uncertainty about which specialist to consult, which may lead to worsening health conditions.

The motivation behind this project stems from the need to provide a centralized, accessible, and intelligent system that can assist users in identifying potential health issues based on their symptoms, direct them to

the right medical specialty, and simplify the process of booking medical services. Moreover, integrating educational videos and awareness resources can help promote preventive care and health literacy.

With the advancement of mobile app development and machine learning technologies, it is now feasible to create a user-friendly, efficient, and intelligent platform to meet these needs.

---

### **1.3 Importance of the Problem Being Addressed**

Early diagnosis and timely medical consultation are critical to improving patient outcomes and reducing healthcare costs. However, many individuals lack the medical knowledge to interpret their symptoms correctly, leading to delays in seeking care or visiting the wrong specialist. This often results in the progression of otherwise manageable conditions.

By providing symptom-based disease predictions and specialist recommendations, the proposed system addresses a significant gap in the current healthcare experience. Additionally, offering features such as video-based awareness and home nursing services expands the system's value, making it a comprehensive solution to support public health and ease the burden on healthcare institutions.

---



## 1.4 Problem Statement

Many patients face uncertainty when it comes to identifying their health condition and choosing the appropriate medical specialist. This often leads to:

- Delayed diagnosis and treatment
- Increased pressure on general physicians
- Inefficient use of medical resources

This project addresses the problem of **inaccurate self-assessment and inefficient access to medical services** by offering an intelligent system that:

- Predicts potential diseases based on user-input symptoms using machine learning algorithms
- Recommends the appropriate medical specialty
- Facilitates direct appointment booking and provides additional services such as home care and health awareness videos

This problem deserves attention because timely access to the right medical service can significantly improve patient outcomes and reduce system-wide strain on healthcare facilities.

## **1.5 Objectives**

### **1.5.1 Main Objective**

To develop an intelligent mobile application that supports users in making informed health decisions by predicting possible diseases from symptoms, recommending the right medical specialist, and enabling seamless access to healthcare services.

### **1.5.2 Specific Objectives**

- Implement a symptom-based disease classification model using machine learning .
- Design a user-friendly interface using Flutter for ease of interaction.
- integrate .NET for real-time database management and user authentication, utilizing a SQL database for secure and scalable data storage.
- Enable users to book appointments with doctors based on specialization and availability.
- Provide access to home nursing services and educational health videos.

- Use Python and Google Colab to preprocess medical datasets and train models.
  - Ensure the system is scalable and adaptable for future enhancements.
- 

## **1.6 Brief Overview of the Proposed Solution**

The proposed system is a smart medical center application developed using Flutter and, .Net integrating machine learning models to provide intelligent healthcare services. Users can enter their symptoms, and the system predicts possible diseases and suggests the appropriate medical specialty. They can then view doctors' profiles, check available time slots, and book appointments. Additional features include access to home nursing services and health awareness videos, making it a complete digital healthcare assistant.

## **Chapter 2 : Literature Review / Related Work**

### **2.1 Existing Research and Technologies**

Recent advancements in mobile health (mHealth) technologies have significantly contributed to improving healthcare accessibility and patient engagement. A variety of mobile applications have emerged to support functions such as online appointment booking, basic symptom checkers, and virtual consultations. However, most existing systems tend to focus on isolated features, often lacking integration between diagnostic support and real-time medical services.

From a machine learning perspective, Support Vector Machines (SVM) and Naïve Bayes classifiers have been widely applied in disease prediction due to their accuracy in multiclass classification and efficiency with textual or categorical symptom data. Numerous studies validate their performance on real-world medical datasets, yet these models are commonly confined to academic or experimental settings, rarely integrated into fully functional applications.

Technologically, Flutter has become a leading choice for developing cross-platform mobile applications due to its performance and native-like UI capabilities.

### **Backend Services:**

The system utilizes a **.NET** backend for user authentication, data management, and secure operations, integrated with a **SQL database** for real-time data storage and retrieval, ensuring secure and scalable backend operations.

### **Machine Learning:**

**Python** and **Scikit-learn** are widely adopted for building and training machine learning models, with platforms like **Google Colab** offering scalable cloud-based training environments. **Medical datasets** from Kaggle are commonly used for benchmarking, providing standardized data for model evaluation.

### **UI/UX Design:**

Tools like **Figma** are essential for designing intuitive UI/UX prototypes and gathering user feedback before development.

### **Backend Logic and API Operations:**

**.NET** is used to manage backend logic and API operations, ensuring robustness and maintainability across different service components.

---

## **2.2 Gaps in Current Solutions**

Despite the growing ecosystem of healthcare apps, several critical limitations persist in existing solutions:

## **Limited Intelligence in Symptom Analysis**

Most applications use static rule-based approaches that lack learning capabilities, resulting in generic or inaccurate health predictions.

## **Fragmented Functionality**

Applications often focus on a single service—such as booking or symptom checking—without offering a cohesive, all-in-one platform.

## **Underutilization of AI in Production Environments**

While machine learning is explored academically, few projects deploy it within real-time, user-facing applications with backend integration.

## **User Experience Challenges**

A significant number of healthcare apps have poor design, making them difficult to navigate, particularly for older users or those with limited technical literacy.

---

## **2.3 Relevance of the Proposed Solution**

The proposed system addresses these limitations by offering a unified platform that combines:

- AI-based disease prediction using SVM trained on real medical datasets.

- Cross-platform mobile access using Flutter for both Android and iOS.
- **Real-time data management and authentication** using **.NET** for secure and scalable backend operations, integrated with a **SQL database** for efficient data storage and retrieval.
- **Backend logic and API handling** via **.NET** for efficient integration between services and seamless data communication. UI/UX designed in Figma to ensure accessibility and responsiveness.
- Utilization of validated datasets from Kaggle to ensure accuracy and relevance in disease prediction.

By integrating these tools into a single application, the project transforms isolated healthcare features into a connected, intelligent ecosystem tailored to user needs.

## 2.4 Comparison with Existing Healthcare Applications

Over the past few years, several healthcare applications have emerged to support patients in managing appointments, obtaining medical advice, or accessing health information. However, most of these apps tend to focus on one or two isolated features, often lacking the holistic approach that modern users need. Below is a breakdown of the most commonly seen systems and how the Medical Center project builds upon or improves them:

## 1. Doctor Booking Systems

**Examples:** Zocdoc, Vezeeta

- **Key Features:**

- Searching for doctors by specialty or location
- Viewing available time slots
- Booking consultations

- **Limitations:**

- Limited to booking only; no symptom support or education
- Often exclude rural or smaller clinics
- No integration with other medical services

- **How Medical Center improves:**

In addition to appointment booking, our system also provides intelligent recommendations for doctors based on symptom analysis, and includes additional modules like home care and educational videos.

---

## 2. Symptom Checker Applications

**Examples:** Ada Health, WebMD

- **Key Features:**

- Users input symptoms and receive potential diagnoses

- **Limitations:**

- Often work in isolation with no connection to real doctors
- Do not guide users to the next actionable step (e.g., booking)
- Limited personalization



- **How Medical Center improves:**

Our app uses trained ML model (SVM) to analyze symptoms and directly suggests a relevant specialist. It also allows the user to book an appointment instantly after receiving a prediction.

---

### **3. Health Awareness Platforms**

**Examples:** YouTube channels, Ministry of Health portals

- **Limitations:**

- Unorganized or not tailored to user needs
- Often text-heavy or lacking interaction
- May require constant internet connection

- **How Medical Center improves:**

The app offers categorized, curated medical videos, with topics such as chronic disease management, hygiene, nutrition, maternal care, etc., and allows offline viewing.

---

### **4. Home Care Service Apps**

**Examples:** Healthcare apps offering home visits

- **Limitations:**

- Require phone calls or web forms
- Limited availability
- No integration with patient history

- **How Medical Center improves:**

Users can request home nurse visits through the app in just a few

clicks, track visit history, and connect it with their overall medical profile.

---

## 5. AI-based Medical Tools

**Examples:** Experimental AI tools (CNN-based models)

- **Limitations:**

- Heavy computation
- Lack of transparency in predictions
- Not user-friendly

- **How Medical Center improves:**

The app uses lightweight ML model (SVM) that is:

- Fast and efficient on mobile
  - Transparent and explainable
  - Easy to update with new medical data
- 

## 6. User Experience & Accessibility Systems

**Limitations:**

- Small fonts, dense layouts
  - No support for low-connectivity usage
  - **How Medical Center improves:**
    - Clean, simple UI designed in Figma with large icons and readable text
    - Some features available offline
-

## 7. Testing and Quality Assurance Systems

### Limitations:

- Bugs in appointment logic
- Inaccurate predictions
- Security loopholes
- **What Medical Center did differently:**
  - Applied unit and integration testing
  - Evaluated ML models using precision, recall, F1-score
  - Tested usability with real users from different age groups

Used .NET for Secure Logins

---

## 8. Deployment and Maintenance Infrastructure

### Limitations:

- Poor maintenance and lack of version control
  - **Medical Center's approach:**
    - Hosted on a SQL-based backend for data storage and management
    - Integrated with version control (GitHub) for tracking changes and collaborating efficiently
    - Easy to push updates and bug fixes
    - Modular structure for future expansion (e.g., telemedicine features)
- 

## Summary

While many systems already exist to address individual healthcare needs, the Medical Center application stands out by merging

multiple essential features — intelligent diagnosis, scheduling, home care, and education — into a single accessible app. This integration, supported by lightweight ML and scalable architecture, allows users to experience a smarter, simpler, and more inclusive approach to digital healthcare.

## **Chapter 3: Proposed System**

### **3.1 Approach Used to Solve the Problem**

The proposed system aims to provide a comprehensive mobile healthcare application that allows users to predict diseases based on their symptoms, book appointments with doctors, and access healthcare services through an intuitive interface. The approach focuses on integrating machine learning models for disease prediction, real-time data management, and user-friendly design, all supported by scalable cloud services. The core objective is to provide a holistic healthcare solution by addressing the following:

#### **1. Data Collection & Integration**

The system uses real-world medical datasets (such as those from Kaggle) to train machine learning models. This ensures that the disease prediction algorithm is both accurate and representative of real medical conditions. By employing datasets related to symptoms, patient history, and diagnoses, the system generates intelligent predictions for the user.

#### **2. Symptom-Based Disease Prediction**

The SVM (Support Vector Machine) and Naïve Bayes algorithms are employed for predicting the likelihood of diseases based on the input symptoms. These models are trained on the data and then tested with new user-provided symptom input, making it a dynamic system capable of learning from past data.

#### **3. Real-Time Data & Backend Management**

The SQL database is used for managing the user data in real-time, including storing symptom inputs, booking appointments, and storing medical histories. The backend is further supported by .NET technologies for API management and logic handling.

#### **4. Cross-Platform Mobile Application**

The mobile application is built using Flutter to ensure a seamless user experience across Android and iOS. Flutter's rich widget library ensures

that the app is not only functional but also aesthetically pleasing and responsive.

## 5. User Interaction & UI/UX Design

Figma is utilized for the design of interactive prototypes and wireframes, ensuring the application is user-friendly, intuitive, and easy to navigate. User feedback gathered through design prototypes is incorporated into the final product.

---

## 3.2 System Architecture

The system architecture is structured to provide clear separation between the frontend (mobile app), backend (server-side services), and the machine learning layer. The system follows a layered architecture, with distinct roles for each component to ensure scalability and modularity.

### System Components:

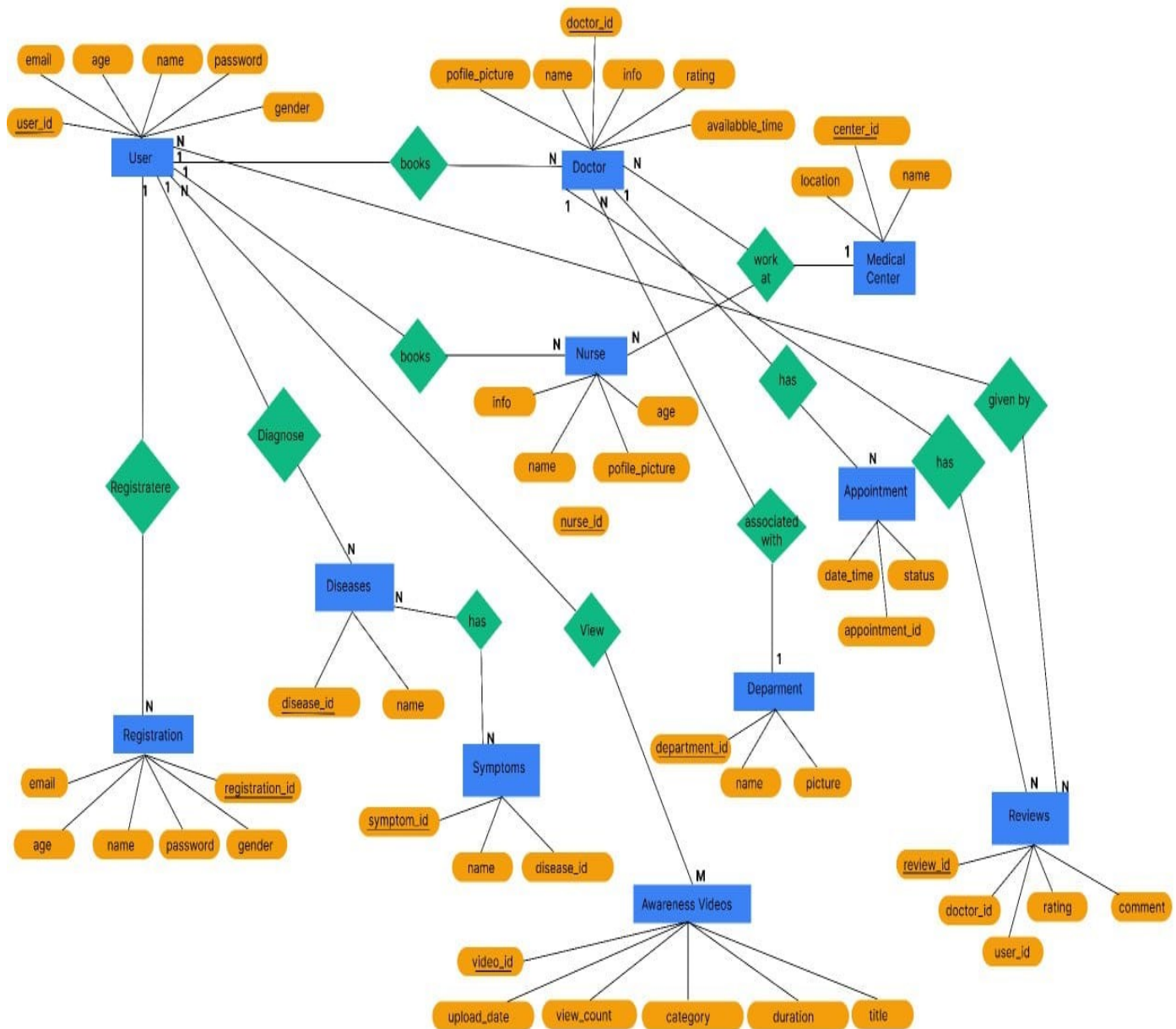
#### 1. Mobile Application (Flutter)

- **User Interface:** Provides users with an interactive environment to input symptoms, browse doctors, and book appointments.
- **User Input Handling:** Collects symptom data and processes it for disease prediction.
- **Display Predictions:** Displays disease predictions and suggests further actions based on the user's symptoms.
- **Backend (.NET & SQL Database)**
  - **Authentication & Security:** .NET handles user authentication, ensuring secure login/logout functionality using SQL-based user management.
  - **Data Storage:** The SQL database is used to store and retrieve user data, such as symptoms, disease predictions, and doctor appointment records, ensuring secure and reliable data management.
-

- **API & Logic:** The backend is implemented using .NET, which interacts with database and the machine learning model to process user data and provide relevant responses.
- 2. **Machine Learning Layer (Python & Scikit-learn)**
  - **SVM Model:** Trained on Kaggle's medical datasets, these models are used to predict diseases based on the symptoms entered by the user.
  - **Data Preprocessing & Model Training:** Google Colab is used to preprocess the data and train the machine learning models, ensuring scalability and ease of access.

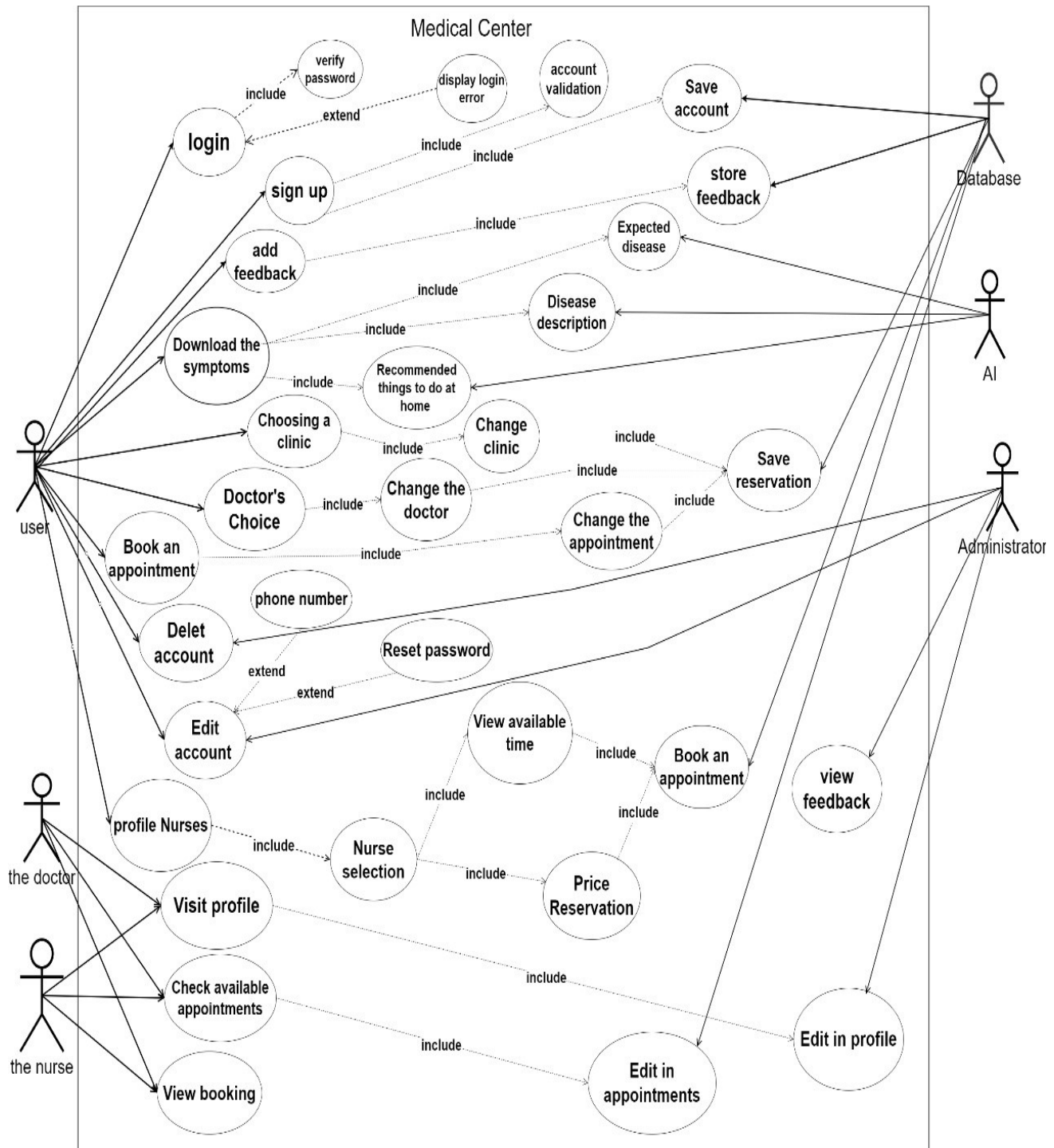
## 3.3 System Diagram

### 3.3.1 :ERD Digram

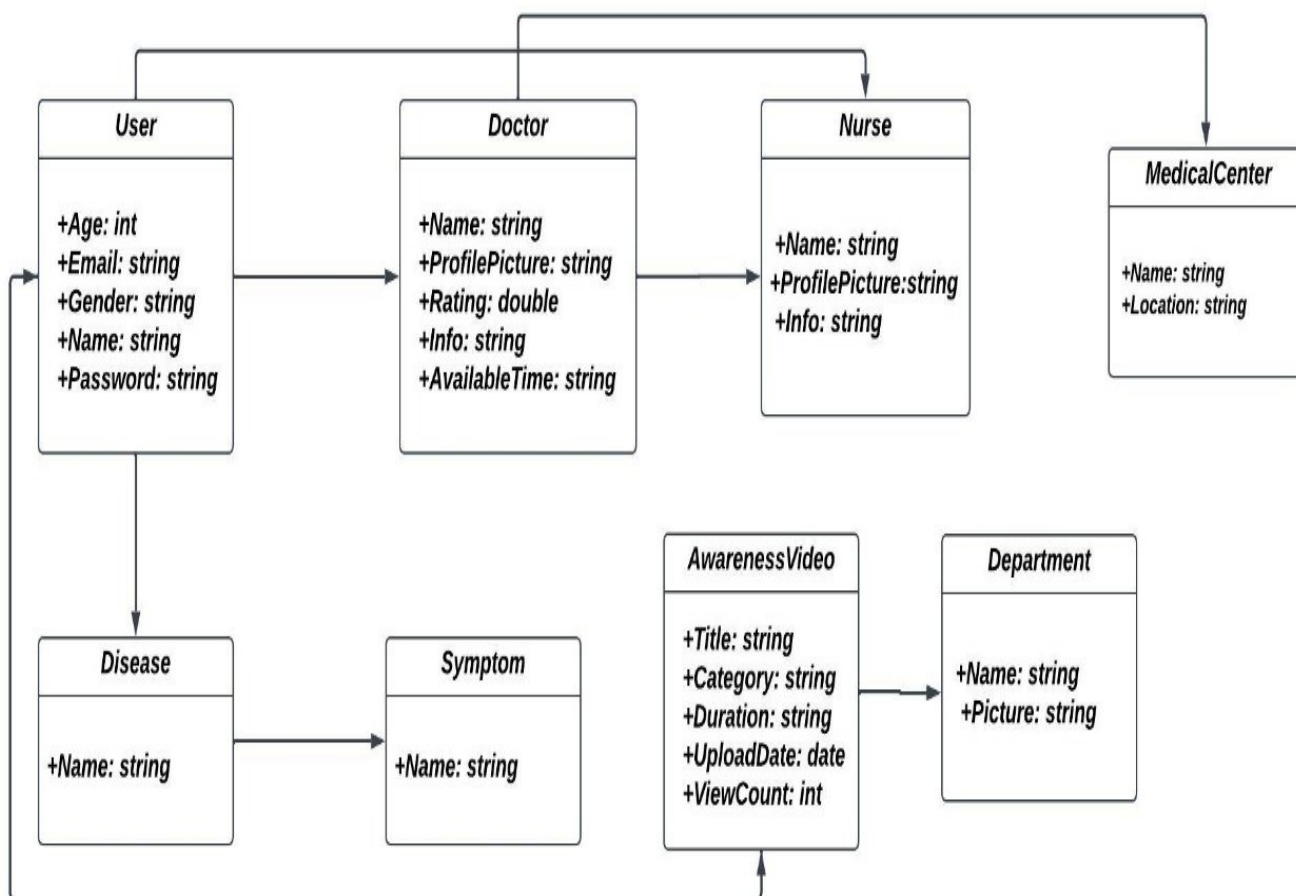




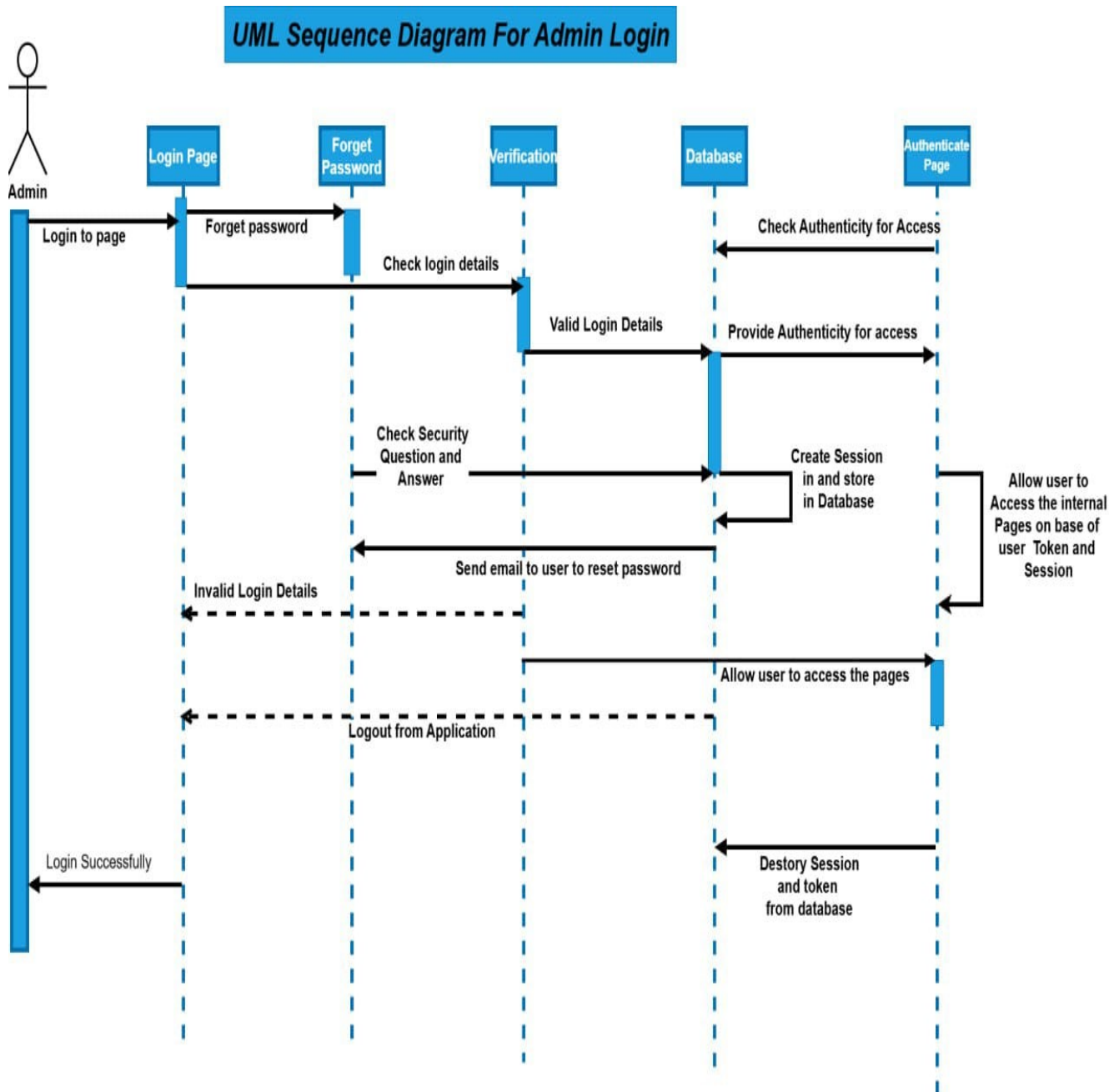
### 3.3.2 UML Use Case Diagram



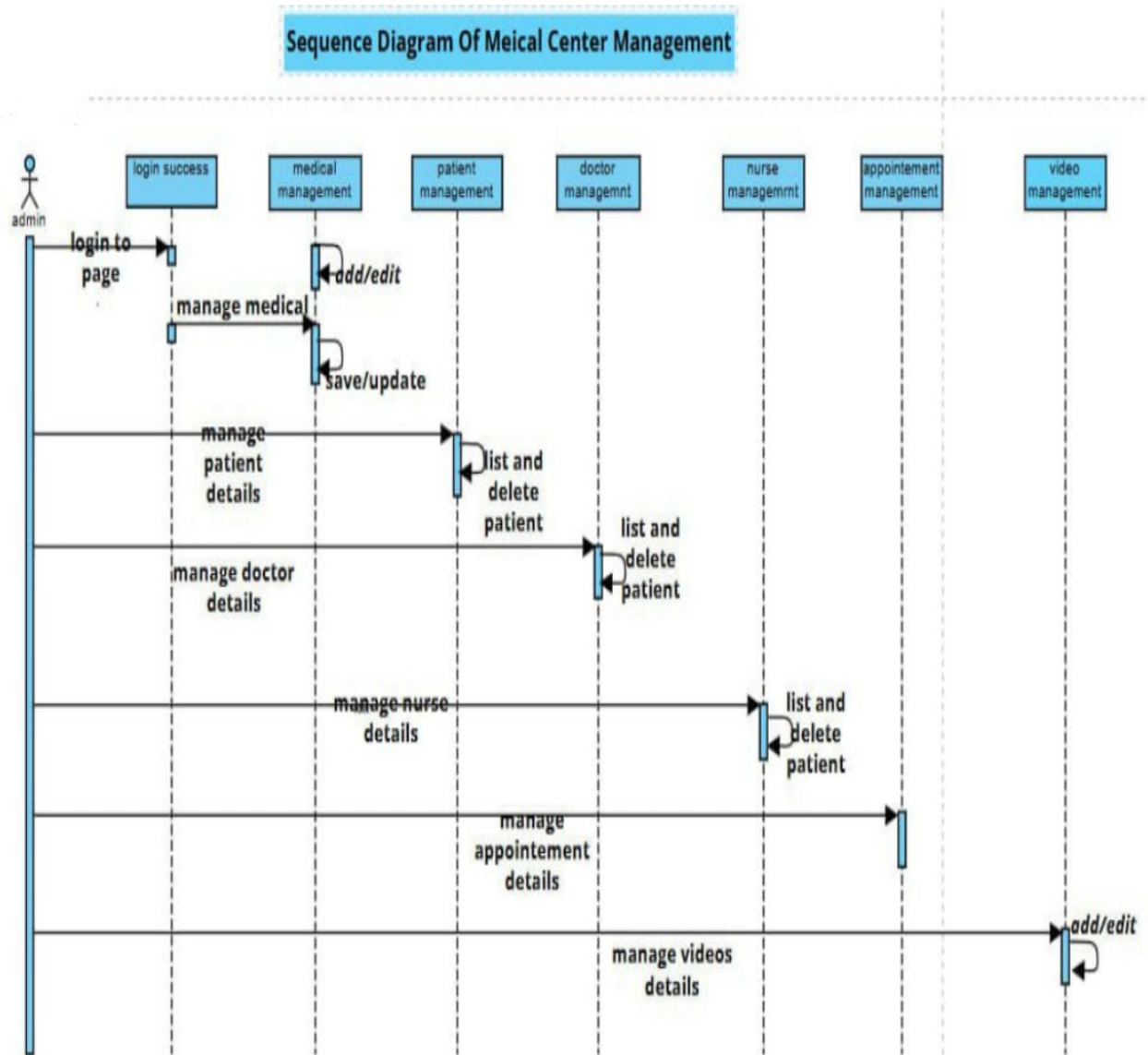
### 3.3.3 :UML class diagram:



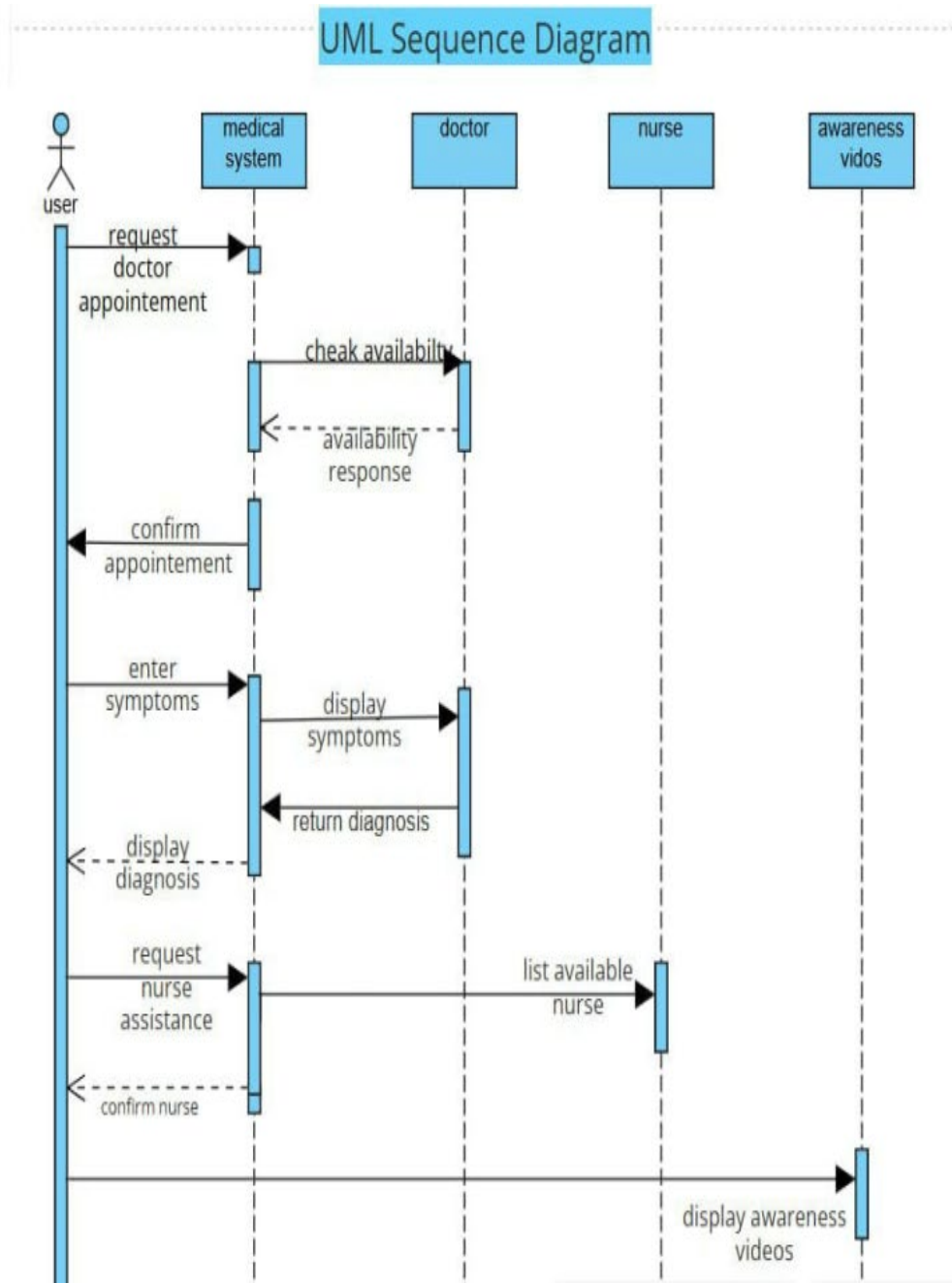
### 3.3.4 :UML Sequence Diagram For Admin Login:



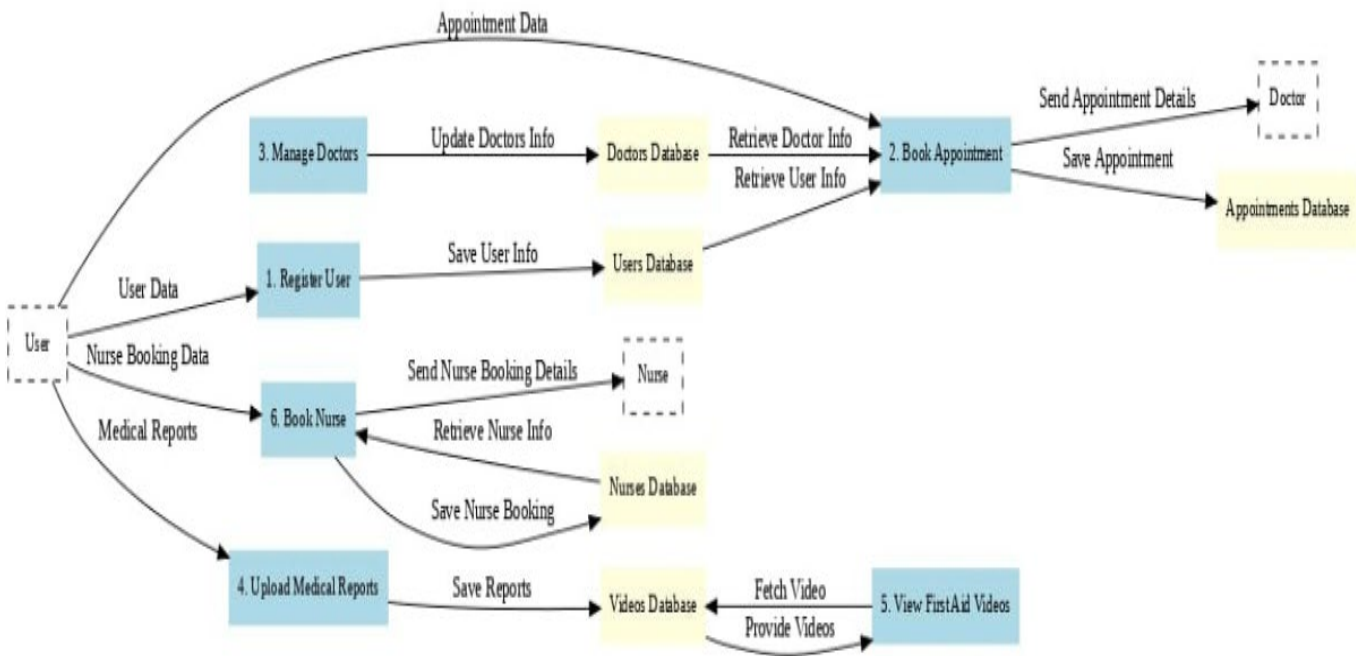
### 3.3.5 Sequence Diagram Of Meical Center Management:



### 3.3.6 UML Sequence Diagram:



### 3.3.7 Data flow diagram:



### **3.4Summary**

This summary provides an overview of the various diagrams and designs required for both the high-level system overview and detailed object-oriented design of the application. Each diagram serves a specific purpose and together they ensure a well-rounded understanding and development of the system.

#### **Summary of Design Requirements for the Application**

##### **Data Flow Diagram (DFD)**

A Data Flow Diagram (DFD) illustrates how data flows through the system. It breaks down the system's processes, showing data sources, data storage, and how data moves between these components.

##### **Entity Relationship Diagram (ERD)**

An Entity Relationship Diagram (ERD) models the system's data entities, their attributes, and the relationships between them. This helps in understanding the database structure.

##### **UML Use Case Diagram**

A UML Use Case Diagram depicts the functional requirements of the system by illustrating the interactions between users (actors) and the system (use cases). It shows what the system does from the user's perspective.

## **UML Sequence Diagram**

A UML Sequence Diagram illustrates how objects interact with each other over time. It shows the sequence of messages exchanged between objects to carry out a specific functionality.

## **UML Class Diagram**

A UML Class Diagram provides a detailed view of the system's structure. It shows the system's classes, their attributes, methods, and the relationships between the classes.



### **3.5 Algorithms or Frameworks Used**

#### **3.5.1 Algorithm Comparison and Selection**

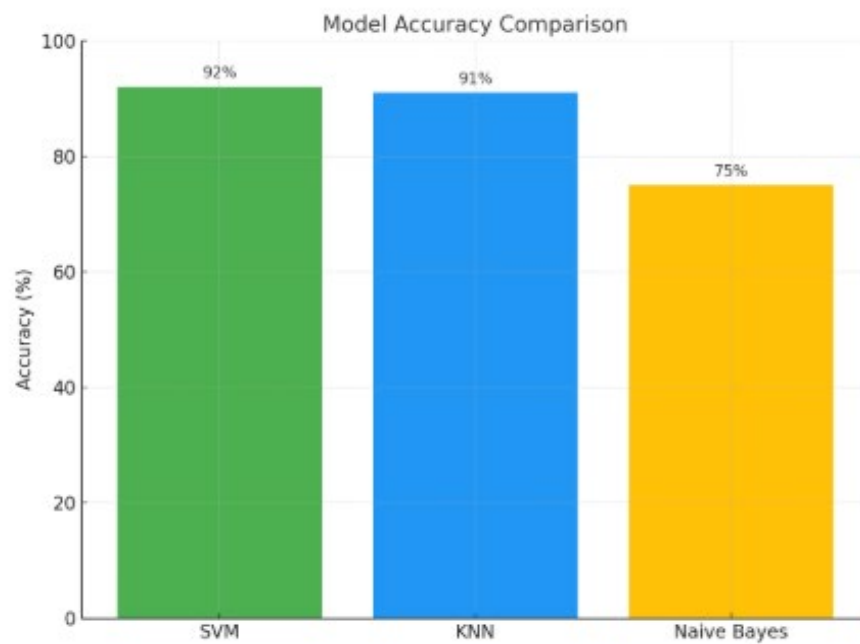
To determine the most suitable algorithm for disease prediction based on user-inputted symptoms, three different machine learning algorithms were implemented and evaluated: K-Nearest Neighbors (KNN), Naïve Bayes, and Support Vector Machine (SVM).

Each model was trained using medical datasets obtained from Kaggle and evaluated using the accuracy metric. The performance of each algorithm is summarized below:

KNN achieved an accuracy of approximately **91%**.

Naïve Bayes performed slightly better, with an accuracy of around **75%**.

SVM outperformed both algorithms with an accuracy of approximately **92%**, and thus was selected as the final model used in the system.



The accuracy of each model was calculated using the following formula:-

$$\frac{TP + TN}{TP + TN + FP + FN} = \text{Accuracy}$$

Where:

TP (True Positives): Number of correctly predicted positive cases.

TN (True Negatives): Number of correctly predicted negative cases.

FP (False Positives): Number of negative cases incorrectly predicted as positive.

FN (False Negatives): Number of positive cases incorrectly predicted as negative.

SVM was chosen due to its superior accuracy and its ability to handle high-dimensional feature spaces, which makes it a suitable choice for symptom-based disease classification.

---

### 3.5.2 Frameworks Used

- **Flutter Framework:**

Flutter is an open-source framework developed by Google for building cross-platform mobile applications. It allows developers to create apps that run on both **iOS** and **Android** using a single codebase, which reduces development time and ensures consistent performance across platforms.

Key features of Flutter:

- **High performance:** Flutter compiles to **native code**, resulting in better performance compared to frameworks that rely on interpreting code.
- **User Interface (UI):** Flutter provides a rich set of **widgets** for building beautiful and responsive UIs with ease. It supports both **Material Design** (Android-style UI) and **Cupertino** (iOS-style UI).
- **Faster development:** With **Hot Reload**, developers can see changes in real time, speeding up the development process without rebuilding the app from scratch.
- **Cross-platform compatibility:** Flutter allows developers to access platform-specific APIs and functionalities through **Flutter plugins**, enabling seamless integration with both Android and iOS features.
- **SQL Integration:** Flutter seamlessly integrates with a SQL database, making it ideal for building apps with features like user authentication, data storage, and secure access to medical records and appointments..

- **.NET Framework:**

The **.NET Framework** is used for building the backend logic and API services for applications. It provides a comprehensive platform for developing web applications, APIs, and services, and it's particularly effective in enterprise environments.

Key features of .NET:

- **Backend logic and APIs:** .NET enables the development of robust backend services and APIs to manage the app's business logic and interact with the database and external systems.
- **Integration with database:** .NET interacts with the SQL database to retrieve and store data, ensuring seamless communication between the mobile app and backend services. This allows for secure data handling, consistent information flow, and reliable system performance..
- **Cross-platform with .NET Core:** While the traditional .NET Framework is mostly used for Windows-based applications, **.NET Core** (the cross-platform version) allows backend services to run on **Windows, macOS, and Linux**.
- **Scalability and performance:** .NET is known for its high performance, scalability, and support for modern development practices, making it a reliable choice for building enterprise-level backend systems.

## **Chapter 4: Implementation**

### **4.1 Technologies, Tools, and Programming Languages Used**

In this project, a variety of technologies, tools, and programming languages were utilized to ensure the development of an efficient, scalable, and functional medical application. The following is an overview of the key technologies and tools used:

- **Flutter:** Flutter was chosen to design the user interface (UI) due to its ability to build cross-platform applications efficiently. With Flutter, we were able to create a responsive and engaging interface that functions seamlessly on both Android and iOS devices, offering a consistent user experience across platforms.
- **NET:** The backend services were developed using .NET, which handled the server-side logic of the application. This included managing complex operations such as retrieving doctor information, scheduling appointments, and ensuring seamless interaction between the frontend and the database.
- **SQL:** Structured Query Language (SQL) was used for data management within the application. SQL was employed to design and query relational databases, ensuring efficient storage and retrieval of critical data such as doctor details, appointment schedules, and patient profiles.
- **Python:** Python played a crucial role in integrating machine learning algorithms into the application. It was used to develop disease prediction models based on input symptoms, utilizing libraries like Scikit-learn and Pandas for data manipulation and model training.

- **Google Colab:** Google Colab was leveraged for experimenting and testing machine learning models before integrating them into the main system. The ease of use, along with free access to powerful GPUs, made it an ideal platform for training and evaluating predictive models.
  - **Machine Learning Algorithms:** SVM (Support Vector Machine) and Naive Bayes classifiers were employed to build the disease prediction models. These algorithms helped classify diseases based on the symptoms provided by users, making the application capable of offering initial diagnostic suggestions.
- 

## 4.2 Key Components/Modules of the System

The system was designed to consist of several key components or modules that work cohesively to provide the intended functionalities. The main components include:

- **User Interface (UI):** The UI was built using Flutter to provide an interactive and user-friendly experience. Users can easily navigate through the system to search for doctors, book appointments, or
- 
- interact with the symptom selection feature. The interface is designed to be intuitive, ensuring a smooth user experience.
- **Doctor Search and Appointment Booking:** This module enables users to search for doctors by specialty and view their available

working hours. Users can then book appointments directly through the application, making the process quick and efficient.

- **Symptom Selection and Disease Prediction:** This feature allows users to choose from a list of symptoms provided in the application. Based on the selected symptoms, the system uses machine learning models to analyze the data and generate an initial prediction of the possible disease. The system then suggests appropriate actions or directs the user to the relevant doctor based on the predicted disease.
  - **Data and User Management:**  
The system manages user data—including registration, authentication, and storage of patient records—using a structured SQL database and .NET backend logic. It securely stores information related to users, appointments, and medical histories, while ensuring efficient data access and consistency.
  - **Machine Learning Model:** SVM algorithm were integrated into the backend to predict possible diseases based on the symptoms entered by users. These models were trained and tested using historical medical data to ensure accuracy and reliability.
-



## 4.2.1 User Interface Design

### Splash and welcome Screen

On this page, the logo for the Medical Center application is displayed

Then welcome screen displayed

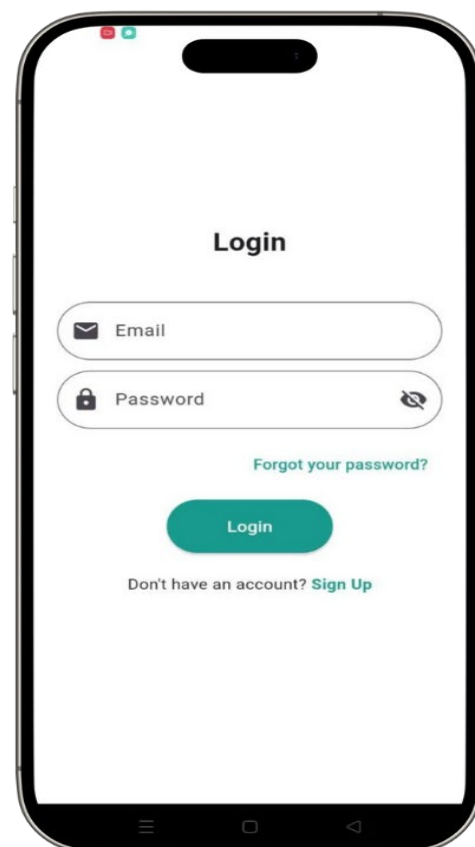
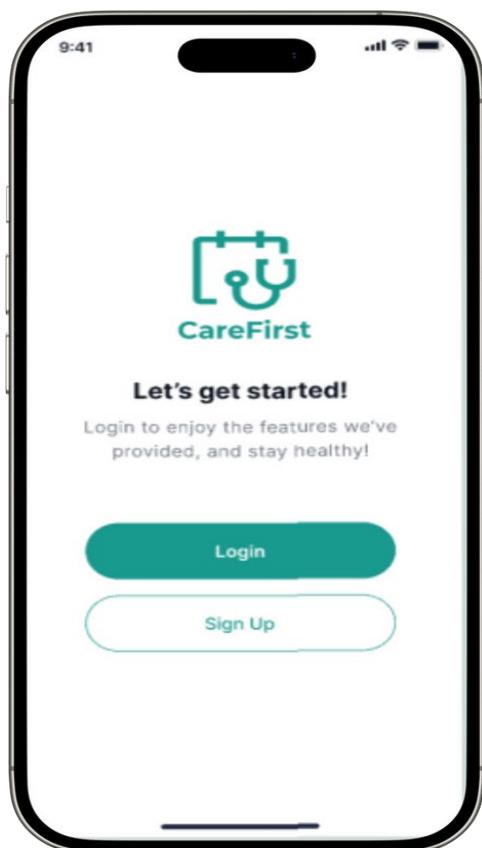


## Login & Registration

The Login Screen allows registered users to sign into the Medical Center app using their email or phone number and password. It provides a secure gateway to access personal services like booking history and health recommendations.

### Key Features:

- Email/Phone and Password fields
- Secure login implemented using .NET server-side validation and SQL-based user authentication



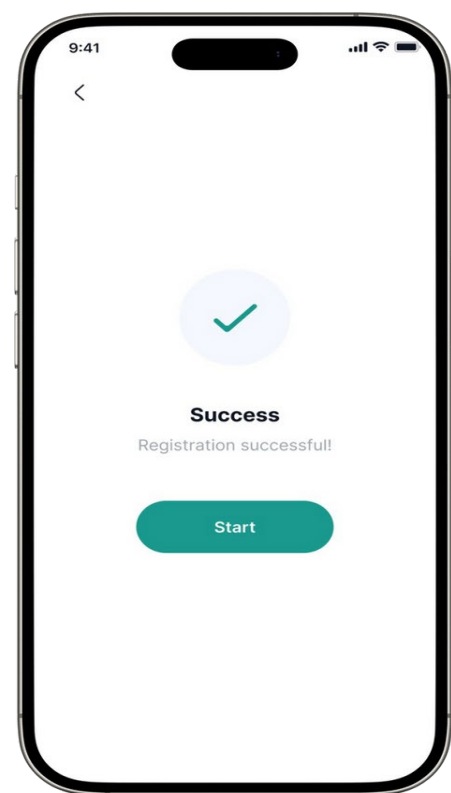
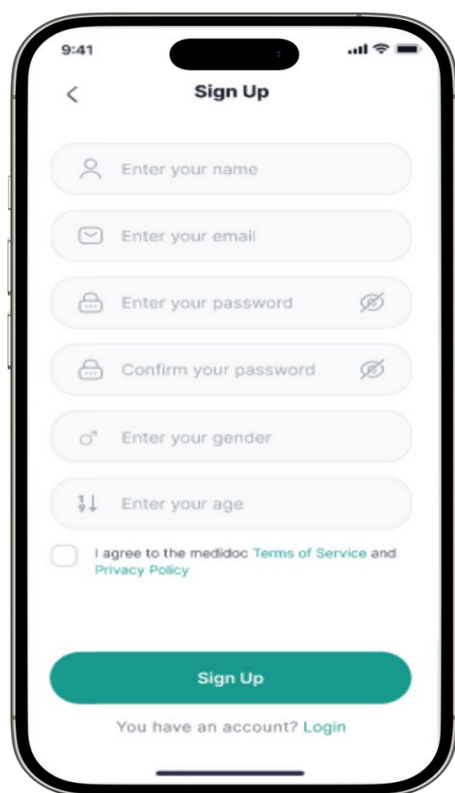
## Registration Screen

### Description:

The Registration Screen enables new users to create an account by entering their personal information. This allows them to fully access all features of the Medical Center application.

### Key Features:

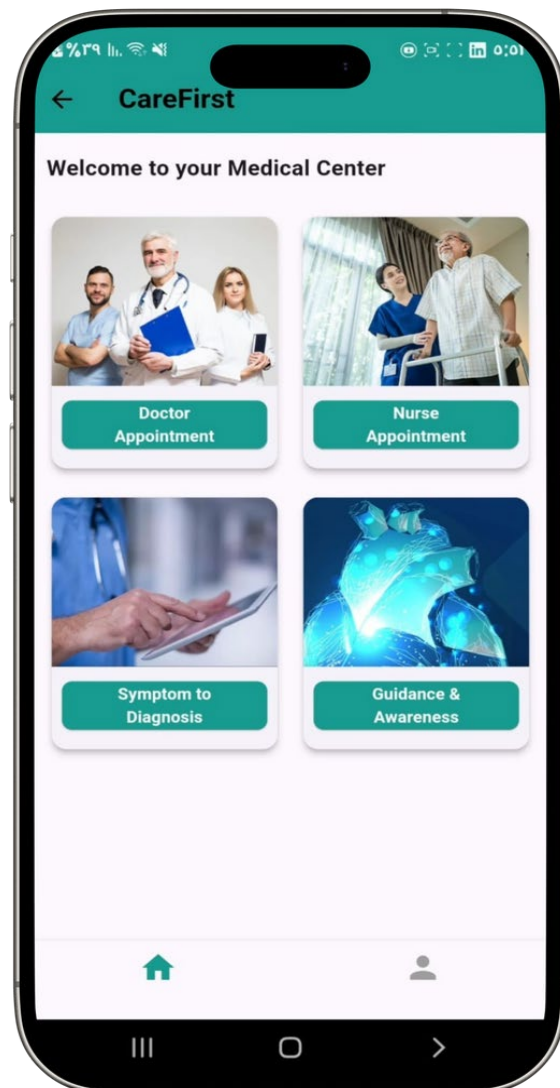
- Name, Email, Phone, and Password fields
- Input validation for user security
- Direct access to Login Screen after successful registration
- "Forgot Password" option



## Home Screen

The Home Page serves as the central hub of the application, offering users quick and intuitive access to the four main features. Each section is visually distinct, ensuring clarity and ease of navigation. Below is the organized structure of the Home Page:

- 1- Doctor Appointment
- 2- Nurse Appointment
- 3- Symptom to Diagnosis
- 4- Guidance and awareness



## Clinics specialties

We have 5 specialties available for booking:

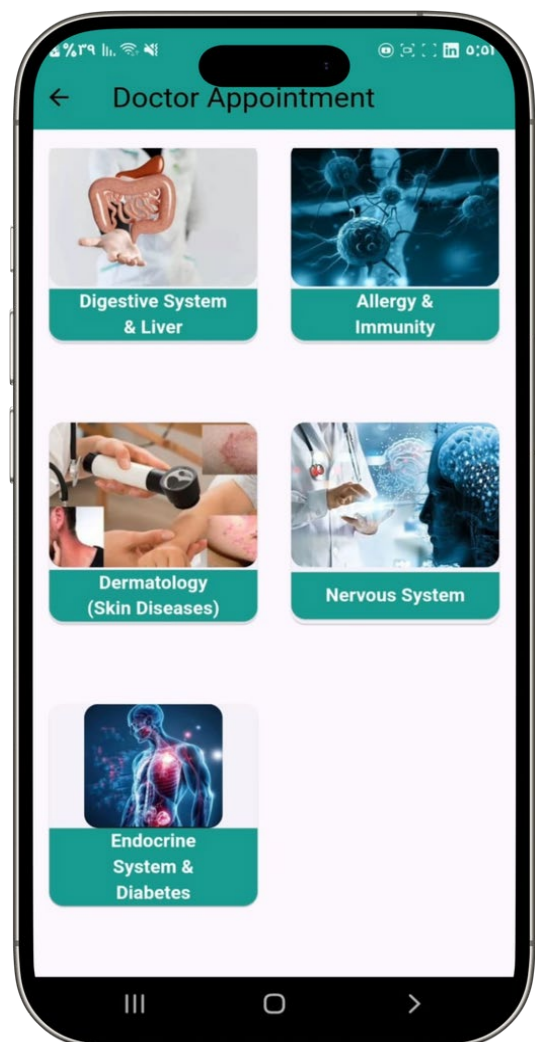
1-Digestive System & Liver

2- Allergy & Immunity

3- Dermatology (Skin Diseases)

4- Nervous System

5- Endocrine System & Diabetes



## Doctor Appointment Booking

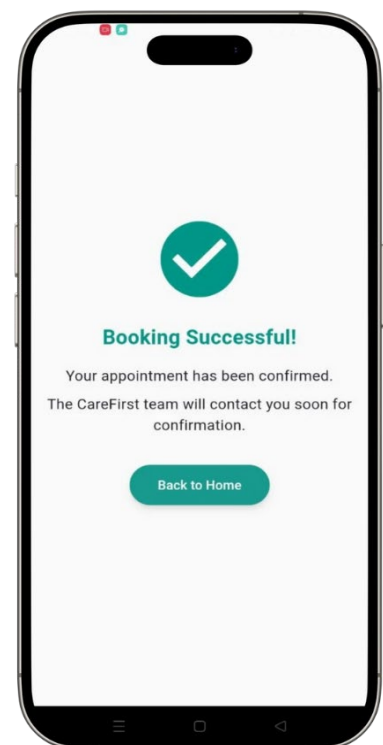
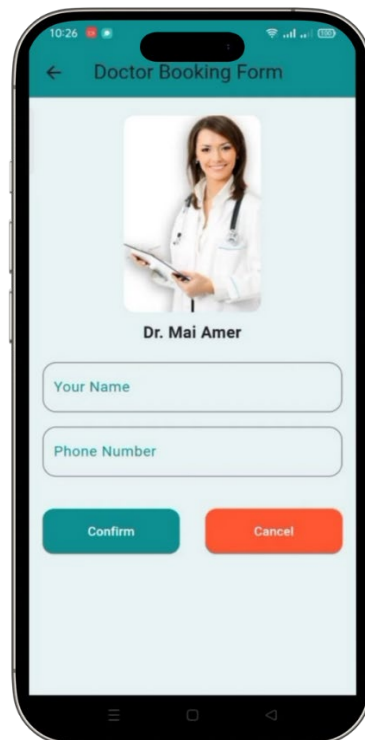
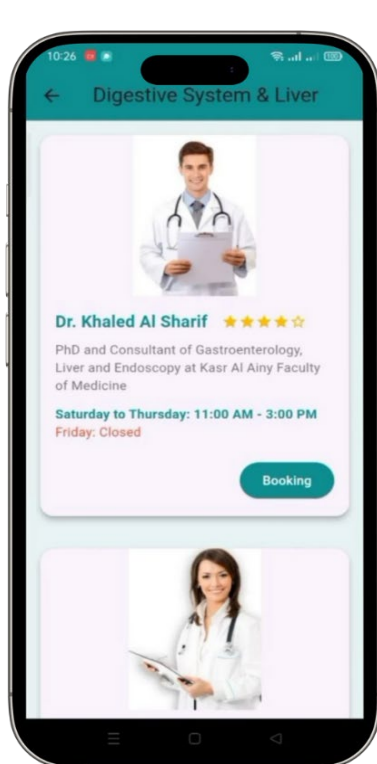
### Easy Appointment Booking & Doctor Selection

The Doctor Booking Screen allows users to easily find and schedule appointments with \_healthcare\_professionals.

Users can browse doctors by specialty, view their available time slots, and confirm their appointments through a simple and intuitive interface.

### Key Features:

- Search for doctors by medical specialty.
- View doctor profiles and available times.
- Book an appointment with a few simple steps.
- Receive booking confirmation and reminders.

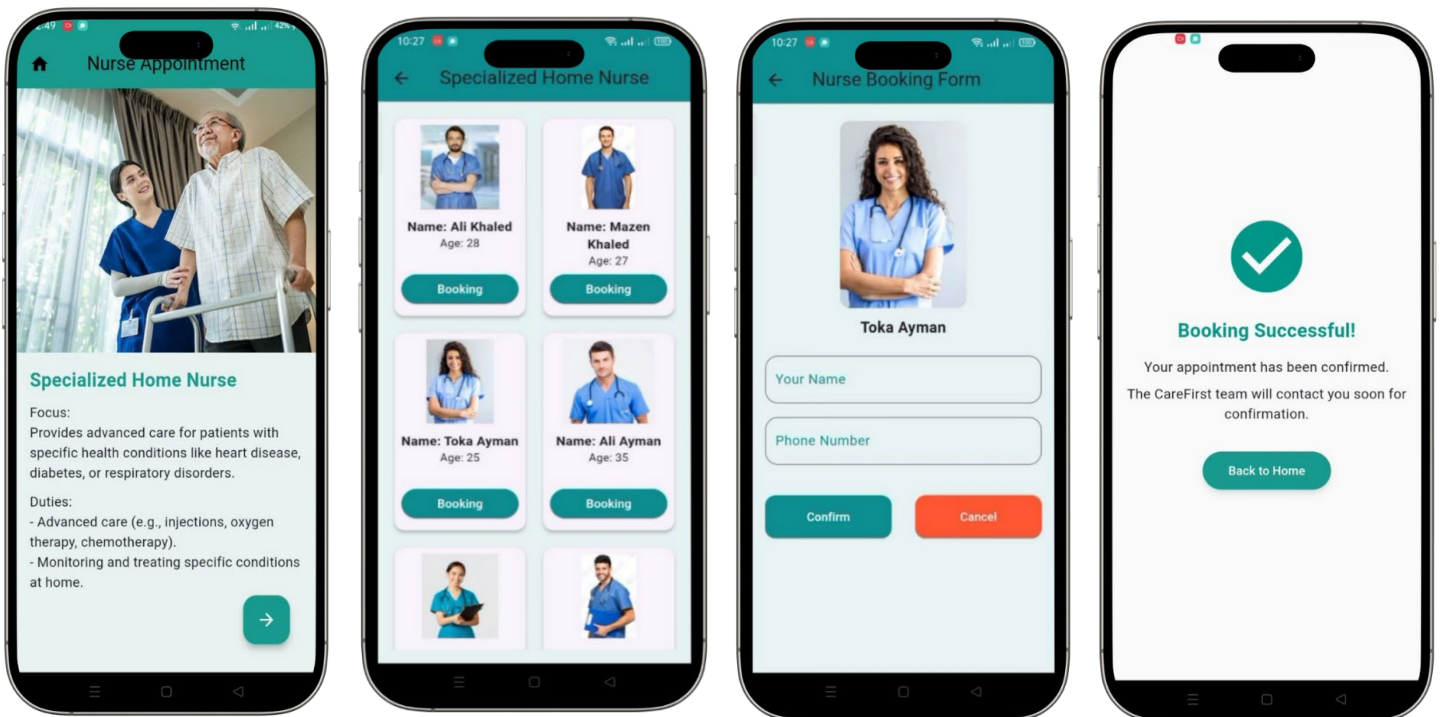


## Nurse Appointment

Users can select the type of care needed, choose preferred dates and times, and submit their request directly through the app.

### Key Features:

- Request certified home nurses for medical assistance.
- Specify service details such as date, time, and location.
- Receive confirmation and visit schedules.
- Seamless and quick booking process.

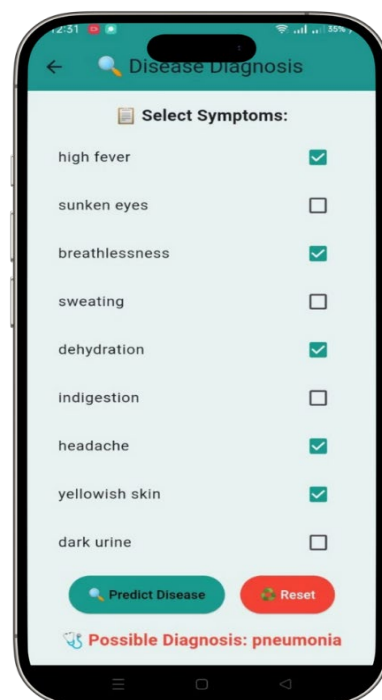
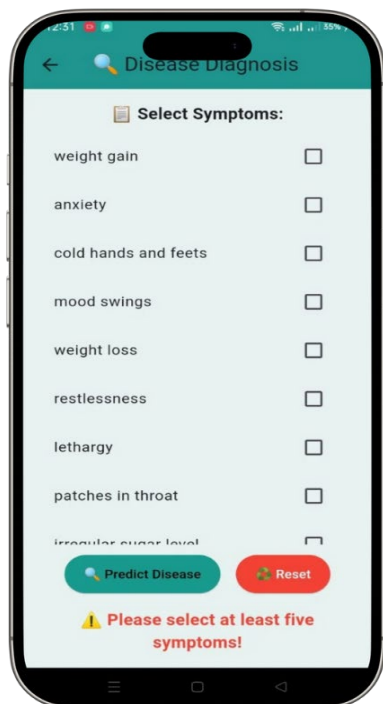


## Symptom Checker

The Symptom Checker Screen allows users to input their current health symptoms and receive preliminary predictions of possible diseases. The system utilizes trained machine learning models (SVM and Naïve Bayes) to provide quick and reliable suggestions based on the entered symptoms.

### Key Features:

- Select symptoms from a pre-defined list or search manually.
- Submit symptoms for analysis by the ML model.
- View a list of possible diseases based on the symptoms provided.
- Guidance to book an appointment with a relevant specialist.





## Health awareness

The Health Education Videos Screen provides users with access to a curated library of medical awareness videos. These videos cover various health topics such as disease prevention, healthy living, nutrition, and first aid practices.

### Key Features:

- Browse categorized medical videos.
- Watch videos online or download them for offline viewing.
- Learn about symptoms, treatments, and preventive measures.
- Enhance user awareness and promote healthy lifestyle habits.



## 4.2.2 Machine Learning Model Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline
from sklearn.model_selection import
train_test_split, KFold, cross_val_score, GridSearchCV, RandomizedSearchCV
from sklearn.metrics import f1_score, accuracy_score, confusion_matrix, classification_report
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression, Perceptron, RidgeClassifier, SGDClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

df = pd.read_csv('/content/dataset (2).csv')
df
df1 = pd.read_csv('/content/Symptom-severity.csv')
df1
df.describe()

df.info()
df.isna()
df.isna().sum()
df.isna().sum()
df.isnull().sum()

cols = df.columns
data = df[cols].values.flatten() #transform to 1D array

y=df['Symptom_1']
print('Symptom_1',y.shape)

df.head(10)
```

```
s = pd.Series(data)
s = s.str.strip() # delete spaces
s = s.values.reshape(df.shape)

df = pd.DataFrame(s, columns=df.columns)

df = df.fillna(0)
df['Disease'].value_counts()
df['Disease'].value_counts()
df1['Symptom'].unique()
vals = df.values
symptoms = df1['Symptom'].unique()

for i in range(len(symptoms)):
    vals[vals == symptoms[i]] = df1[df1['Symptom'] == symptoms[i]]['weight'].values[0]

d = pd.DataFrame(vals, columns=cols)

d = d.replace('dischromic _patches', 0)
d = d.replace('spotting_ urination', 0)
df = d.replace('foul_smell_of urine', 0)
df.shape
df = df.drop(df.index[1000:2000])
df.shape
X = df.iloc[:,1:].values
y = df['Disease'].values
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,
random_state=42)
print('X_train_data : ',X_train.shape)
print('y_train_data : ',y_train.shape)
print('X_test_data : ',X_test.shape)
print('y_test_data : ',y_test.shape)
```

```

# Support Vector Classifier(SVC)
from sklearn.svm import SVC
clf_svc = SVC(C=3)
#clf_svc = SVC(C=3)
clf_svc.fit(X_train,y_train)
y_pred_SVC = clf_svc.predict(X_test)
accuracy_SVC = accuracy_score(y_test,y_pred_SVC)
print(' Support Vector Classifier' , accuracy_SVC)
df.sample(5)
clf_svc.predict([[5,3 ,2,5,2]])
df.head(10)

import joblib
# تحميل النموذج
loaded_model = joblib.load('svm_disease_prediction.pkl')
# اختبار النموذج على بيانات جديدة
prediction = loaded_model.predict([[3, 7, 4, 2, 3]])
print("Prediction:", prediction)

import joblib
# حفظ النموذج
joblib.dump(clf_svc, '/content/svm_disease_prediction.pkl')
print("Model saved successfully! 🇪🇬")

from google.colab import files
files.download('/content/svm_disease_prediction.pkl')

```

## Explanation of the code

This script organizes files from a main folder into training, testing, and validation folders based on specified ratios. Here's a summary of what it does

1. It imports the necessary libraries 'os' for interacting with the file system.
2. It sets the paths for the main folder and the folders where the data will be organized for training, testing, and validation.
3. It defines the ratios for splitting the data into training, testing, and validation sets.

This script is useful for organizing data into separate sets for machine learning tasks such as training and evaluating models. It ensures that the data distribution follows the specified ratios, facilitating fair evaluation and validation of the model

### **import numpy as np**

- Imports the numpy library, which is essential for numerical operations in Python.
- Provides support for large, multi-dimensional arrays and matrices.
- Includes a collection of mathematical functions to operate on these arrays efficiently.

### **import pandas as pd**

- Imports the pandas library, which is widely used for data manipulation and analysis in Python.
- Provides data structures like Series (1D) and DataFrame (2D) for handling structured data.
- Commonly used for reading, writing, filtering, and transforming datasets.

### **import matplotlib.pyplot as plt**

- Imports the pyplot module from the matplotlib library, used for data visualization in Python.
- Provides functions for creating a wide variety of static, animated, and interactive plots.
- Commonly used to visualize datasets through charts like line plots, bar charts, histograms, and more.

## **import seaborn as sns**

- Imports the seaborn library, which is built on top of matplotlib for statistical data visualization.
- Provides a high-level interface for creating attractive and informative statistical graphics.
- Commonly used for creating complex visualizations like heatmaps, violin plots, and pair plots with less code.

## **sns.set()**

- Sets the default aesthetic style for the plots created using seaborn.
- It adjusts elements like background color, grid lines, and font style to make the plots more visually appealing.
- This function applies global settings for all seaborn plots in the current session.

## **%matplotlib inline**

- This is a magic command in Jupyter notebooks that ensures plots are displayed directly below the code cell that generates them.
- It allows the output of plotting commands to be rendered within the notebook, instead of in a separate window.
- It's commonly used for interactive environments like Jupyter to streamline the plotting workflow.

```
from sklearn.model_selection import train_test_split, KFold,  
cross_val_score, GridSearchCV, RandomizedSearchCV
```

- Imports several functions and classes from the `model_selection` module of scikit-learn.
- `train_test_split`: Used to split data into training and testing sets for machine learning models.
- `KFold`: Used for cross-validation by splitting data into K subsets for model evaluation.
- `cross_val_score`: Computes the cross-validation score for a given model.
- `GridSearchCV`: Used for hyperparameter tuning by exhaustively searching a specified parameter grid.
- `RandomizedSearchCV`: Performs hyperparameter tuning by sampling from a parameter grid randomly, often faster than `GridSearchCV`.

```
from sklearn.metrics import f1_score, accuracy_score, confusion_matrix,  
classification_report
```

- Imports several metrics from scikit-learn used to evaluate the performance of machine learning models.
- `f1_score`: Measures the balance between precision and recall, useful for imbalanced datasets.



- `accuracy_score`: Calculates the accuracy of the model, i.e., the proportion of correct predictions.
- `confusion_matrix`: Provides a matrix to evaluate the performance of a classification model by comparing predicted and actual values.
- `classification_report`: Generates a detailed report with precision, recall, f1-score, and support for each class.

### **`from sklearn.neighbors import KNeighborsClassifier`**

- Imports the `KNeighborsClassifier` from scikit-learn, which is used for classification tasks.
- This algorithm classifies data based on the majority class among the 'K' nearest neighbors in the feature space.
- It's a simple, yet powerful algorithm, often used for classification problems with a well-defined decision boundary.

### **`from sklearn.ensemble import RandomForestClassifier`**

- Imports the `RandomForestClassifier` from scikit-learn, used for classification tasks.
- A Random Forest is an ensemble method that constructs multiple decision trees and merges them to improve accuracy and prevent overfitting.
- It is highly effective for both classification and regression problems, handling complex datasets with ease.

```
from sklearn.linear_model import LogisticRegression, Perceptron,  
RidgeClassifier, SGDClassifier
```

- Imports several linear models from scikit-learn for classification tasks.
- **LogisticRegression**: A widely used algorithm for binary and multi-class classification tasks, based on a logistic function.
- **Perceptron**: A simple neural network model used for binary classification, based on a linear decision boundary.
- **RidgeClassifier**: A classifier that applies ridge regularization (L2) to linear classification, helping to prevent overfitting.
- **SGDClassifier**: A classifier that uses stochastic gradient descent for training, useful for large datasets and linear models.

```
from sklearn.svm import SVC
```

- Imports the Support Vector Classifier (SVC) from scikit-learn, used for classification tasks.
- **SVC** uses a hyperplane to separate different classes in the feature space, aiming to maximize the margin between classes.
- It's effective in high-dimensional spaces and is commonly used for both linear and non-linear classification problems.

```
from sklearn.naive_bayes import GaussianNB
```

- Generates descriptive • Imports the Gaussian Naive Bayes classifier from scikit-learn, used for classification tasks.
- This algorithm is based on Bayes' theorem and assumes that the features follow a Gaussian (normal) distribution.
- It's particularly effective for classification problems with continuous features and is known for its simplicity and efficiency.

### **from sklearn.tree import DecisionTreeClassifier**

- Imports the DecisionTreeClassifier from scikit-learn, used for classification tasks.
- A decision tree splits the data into subsets based on feature values, making decisions at each node to classify data.
- It's a versatile algorithm that can handle both numerical and categorical data, and is easily interpretable.

### **from sklearn.preprocessing import OneHotEncoder, LabelEncoder**

- Imports the OneHotEncoder and LabelEncoder from scikit-learn, used for encoding categorical features.
- OneHotEncoder: Converts categorical variables into a format that can be provided to machine learning algorithms by converting each category into a new binary column.

- LabelEncoder: Converts categorical labels into numeric values, assigning each category a unique integer label.

```
df = pd.read_csv('/content/dataset (2).csv')
```

- Reads the CSV file located at '/content/dataset (2).csv' into a DataFrame using pandas.
- This allows the data to be easily manipulated and analyzed in Python.
- The DataFrame is stored in the variable 'df'.

```
df
```

- Displays the DataFrame 'df' to view the contents of the dataset loaded from the CSV file.

```
df1 = pd.read_csv('/content/Symptom-severity.csv')
```

- Reads the CSV file located at '/content/Symptom-severity.csv' into another DataFrame using pandas.
- The second dataset, 'Symptom-severity.csv', is now stored in the variable 'df1'.

```
df1 = pd.read_csv('/content/Symptom-severity.csv')
```

- Reads the CSV file located at '/content/Symptom-severity.csv' into another DataFrame using pandas.

- The second dataset, 'Symptom-severity.csv', is now stored in the variable 'df1'.

### **df1**

- Displays the DataFrame 'df1' to view the contents of the second dataset loaded from the CSV file.

### **y = df['Symptom\_1']**

- Extracts the 'Symptom\_1' column from the DataFrame 'df' and stores it in the variable 'y'.
- This represents the target variable for a machine learning model, containing the values of 'Symptom\_1'.

### **print('Symptom\_1', y.shape)**

- Prints the shape of the 'y' variable, showing the number of entries in the 'Symptom\_1' column.
- The shape is typically displayed as a tuple (number\_of\_rows, number\_of\_columns), which helps in understanding the structure of the data.

### **df.head(10)**

- Displays the first 10 rows of the DataFrame 'df'.
- This is useful for quickly inspecting the top entries of the dataset to understand its structure and contents.

### **df.describe()**

statistics of the DataFrame 'df', including measures like mean, standard deviation, minimum, and maximum values.

- This function helps to understand the distribution and spread of numerical data in the dataset.

### **df.info()**

- Provides a summary of the DataFrame 'df', including the number of non-null entries, data types of each column, and memory usage.
- This is helpful to quickly check for missing values and to understand the structure of the dataset.

### **df.isna()**

- Checks for missing values in the DataFrame 'df'.
- Returns a DataFrame of the same shape as 'df' with Boolean values: True for missing values and False for non-missing values.

### **df.isna().sum()**

- Calculates the total number of missing (NaN) values in each column of the DataFrame 'df'.
- It returns a Series with the column names as the index and the count of missing values as the values.

### **df.isna().sum()**

- Calculates the total number of missing (NaN) values in each column of the DataFrame 'df'.
- It returns a Series with the column names as the index and the count of missing values as the values.

### **df.isnull().sum()**

- Calculates the total number of missing (NaN) values in each column of the DataFrame 'df'.
- It returns a Series with the column names as the index and the count of missing values as the values.
- The function 'isnull()' is equivalent to 'isna()', both used to check for missing values.

### **cols = df.columns**

- Extracts the column names from the DataFrame 'df' and stores them in the variable 'cols'.
- This allows you to access all column names for further operations.

### **data = df[cols].values.flatten() #transform to 1D array**

- Converts the entire DataFrame 'df' into a 1D array by flattening the values.

- The 'flatten()' function reshapes the DataFrame from a 2D array to a 1D array, making it easier for some types of data processing.

```
s = pd.Series(data)
```

- Converts the 1D array 'data' into a pandas Series, which allows you to perform Series-specific operations.
- This step makes the data more flexible for further manipulation or processing.

```
s = s.str.strip() # delete spaces
```

- Strips any leading or trailing spaces from each string element in the Series 's'.
- This operation is useful for cleaning string data, especially when dealing with raw input from datasets.

```
s = s.values.reshape(df.shape)
```

- Reshapes the Series 's' back into the original shape of the DataFrame 'df'.
- The 'reshape' function ensures the data aligns with the original DataFrame structure, making it suitable for further analysis or insertion back into the DataFrame.

```
df = pd.DataFrame(s, columns=df.columns)
```

- Converts the Series 's' back into a DataFrame with the same column names as the original DataFrame 'df'.



- This step ensures that the reshaped data is organized into the correct tabular structure with appropriate column labels.

```
df = df.fillna(0)
```

- Fills any missing (NaN) values in the DataFrame 'df' with 0.
- This method is commonly used to handle missing data by replacing NaN values with a specific value, such as 0 in this case.

```
df['Disease'].value_counts()
```

- Returns the count of unique values in the 'Disease' column of the DataFrame 'df'.
- This is useful for understanding the distribution of different diseases in the dataset, showing how many instances exist for each disease.

```
df1['Symptom'].unique()
```

- Returns an array of unique values from the 'Symptom' column of the DataFrame 'df1'.
- This is useful for identifying all distinct symptoms present in the dataset without any duplicates.

```
vals = df.values
```

- Extracts the underlying values of the DataFrame 'df' into a NumPy array 'vals'.

- This allows for direct manipulation of the data as an array.

```
symptoms = df1['Symptom'].unique()
```

- Retrieves the unique symptoms from the 'Symptom' column of the DataFrame 'df1'.
- This array contains all distinct symptoms present in the dataset, useful for iterating over them and making replacements.

```
for i in range(len(symptoms)):
```

```
    vals[vals == symptoms[i]] = df1[df1['Symptom'] ==  
symptoms[i]]['weight'].values[0]
```

- Loops through each unique symptom in the 'symptoms' array.
- Replaces the occurrences of each symptom in the 'vals' array with its corresponding weight from the 'df1' DataFrame.

```
d = pd.DataFrame(vals, columns=cols)
```

- Converts the modified 'vals' array back into a DataFrame 'd', with the original column names from 'df'.
- This step restores the data back into a structured format after manipulation.

```
d = d.replace('dischromic_patches', 0)
```

- Replaces any occurrences of the symptom 'dischromic \_patches' in the DataFrame 'd' with 0.
- This step is used to handle specific symptoms that might need to be excluded or represented as 0 in the dataset.

```
d = d.replace('spotting_ urination', 0)
```

- Replaces any occurrences of the symptom 'spotting\_ urination' in the DataFrame 'd' with 0.
- Similar to the previous replacement, it standardizes this symptom value to 0.

```
df = df.replace('foul_smell_of urine', 0)
```

- Replaces any occurrences of the symptom 'foul\_smell\_of urine' in the DataFrame 'd' with 0.
- This ensures that this particular symptom is treated as 0 in the dataset.

```
df.shape
```

- Returns the shape of the DataFrame 'df', represented as a tuple (number\_of\_rows, number\_of\_columns).
- This is useful for understanding the dimensions of the dataset, including how many samples (rows) and features (columns) it contains.

```
df = df.drop(df.index[1000:2000])
```

- Drops the rows with indices from 1000 to 1999 in the DataFrame 'df'.

- This operation removes a subset of the data, which can be useful for reducing the dataset size or removing specific rows.

### **df.shape**

- Returns the shape of the DataFrame 'df' after the rows have been dropped.
- This shows the updated number of rows and columns in the DataFrame, helping to verify that the correct rows were removed.

### **X = df.iloc[:, 1:].values**

- Selects all rows and columns from index 1 to the end (excluding the first column) from the DataFrame 'df'.
- This operation creates the feature matrix 'X', which includes all the predictor variables for machine learning.
- The '.values' converts the DataFrame into a NumPy array.

### **y = df['Disease'].values**

- Extracts the 'Disease' column from the DataFrame 'df' as the target variable 'y'.
- The '.values' converts the column into a NumPy array, making it compatible with machine learning models.

### **from sklearn.model\_selection import train\_test\_split**

- Imports the 'train\_test\_split' function from scikit-learn, which is used to split datasets into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,  
random_state=42)
```

- Splits the feature matrix 'X' and target variable 'y' into training and testing sets.
- 30% of the data is allocated to the testing set (test\_size=0.30), and the remaining 70% is used for training.
- The 'random\_state=42' ensures that the split is reproducible, producing the same split every time the code is run.

```
print('X_train_data : ', X_train.shape)
```

- Prints the shape (number of rows and columns) of the 'X\_train' feature matrix, showing the dimensions of the training data.

```
print('y_train_data : ', y_train.shape)
```

- Prints the shape (number of entries) of the 'y\_train' target variable, showing the number of training labels.

```
print('X_test_data : ', X_test.shape)
```

- Prints the shape (number of rows and columns) of the 'X\_test' feature matrix, showing the dimensions of the testing data.

```
print('y_test_data : ', y_test.shape)
```

- Prints the shape (number of entries) of the 'y\_test' target variable, showing the number of testing labels.

### **# Support Vector Classifier (SVC)**

#### **from sklearn.svm import SVC**

- Imports the Support Vector Classifier (SVC) from scikit-learn, a machine learning model used for classification tasks.
- SVC is used to find a hyperplane that best separates the data into classes.

#### **clf\_svc = SVC(C=3)**

- Creates an instance of the SVC model with a regularization parameter 'C' set to 3.
- The 'C' parameter controls the trade-off between achieving a low error on the training data and maintaining a simple decision boundary.

#### **# clf\_svc = SVC(C=3)**

- This line is commented out, but it's identical to the line above. It would create another instance of the SVC model.

#### **clf\_svc.fit(X\_train, y\_train)**

- Trains the SVC model using the training data ('X\_train' as features and 'y\_train' as target labels).
- The model learns the relationship between features and target labels to make predictions.

```
y_pred_SVC = clf_svc.predict(X_test)
```

- Makes predictions on the test set ('X\_test') using the trained SVC model.
- The predictions are stored in 'y\_pred\_SVC', representing the predicted target labels.

```
accuracy_SVC = accuracy_score(y_test, y_pred_SVC)
```

- Calculates the accuracy of the SVC model by comparing the predicted labels ('y\_pred\_SVC') with the true labels ('y\_test').
- The 'accuracy\_score' function returns the percentage of correct predictions.

```
print('Support Vector Classifier', accuracy_SVC)
```

- Prints the accuracy of the SVC model.
- This shows how well the model performs on the test set.

```
df.sample(5)
```

- Randomly selects 5 rows from the DataFrame 'df'.
- This method is useful for quickly inspecting a random sample of data from the dataset, which can help in identifying any patterns or inconsistencies.

```
clf_svc.predict([[5, 3, 2, 5, 2]])
```

- Uses the trained Support Vector Classifier (SVC) model to make a prediction for a new data point with the feature values [5, 3, 2, 5, 2].
- The input data must be formatted as a 2D array (even if it's a single sample), hence the double square brackets.
- The model predicts the target label for this new data point based on the patterns it learned during training.

```
df.head(10)
```

- Displays the first 10 rows of the DataFrame 'df'.
- This is useful for quickly inspecting the top entries of the dataset to understand its structure and contents.

```
import joblib
```

- Imports the 'joblib' library, which is used for serializing (saving) and deserializing (loading) Python objects, such as machine learning models.
- Joblib is commonly used for saving models after training and loading them for future use.

```
# Loading the model
```

```
loaded_model = joblib.load('svm_disease_prediction.pkl')
```



- Loads the saved SVM model from the file 'svm\_disease\_prediction.pkl'.
- This allows you to use a previously trained model without having to retrain it each time.

```
print("Prediction:", prediction)
```

- Prints the prediction result for the new data point, showing the model's output.
- The prediction corresponds to the target label that the model estimates based on the input features.

```
import joblib
```

- Imports the 'joblib' library, which is used for serializing (saving) and deserializing (loading) Python objects like machine learning models.
- Joblib is commonly used for saving trained models to disk so they can be reused later.

```
# Saving the model
```

```
joblib.dump(clf_svc, '/content/svm_disease_prediction.pkl')
```

- Saves the trained Support Vector Classifier (SVC) model (stored in 'clf\_svc') to the file '/content/svm\_disease\_prediction.pkl'.
- This allows you to persist the model for future use without needing to retrain it each time.

```
print("Model saved successfully! 🎉")
```

- Prints a success message to indicate that the model has been saved successfully.

```
from google.colab import files
```









- Imports the 'files' module from Google Colab, which provides functions for interacting with files in the Colab environment.
- This module allows you to download files from Colab to your local machine.

```
files.download('/content/svm_disease_prediction.pkl')
```

- Downloads the saved model file ('svm\_disease\_prediction.pkl') from the Colab environment to your local machine.
- This makes the model accessible for use outside of the Colab environment, such as on your personal system.

## 4.2.3 Machine Learning Model Integration

This project comprises multiple interconnected files and folders, collaboratively designed to build an intelligent disease prediction system utilizing machine learning algorithms. Below is a detailed explanation of each component and its role within the overall architecture.

 _pycache_	4/10/2025 6:50 PM	File folder	
 venv	4/10/2025 6:50 PM	File folder	
 dataset (2)	3/13/2025 2:49 PM	Microsoft Excel C...	368 KB
 dataset_cleaned	4/16/2025 8:01 PM	Microsoft Excel C...	354 KB
 server	4/16/2025 8:04 PM	Python File	8 KB
 svm_disease_prediction.pkl	3/14/2025 7:16 PM	PKL File	496 KB
 svm_disease_prediction_new.pkl	4/16/2025 8:04 PM	PKL File	840 KB
 Symptom-severity	3/13/2025 2:49 PM	Microsoft Excel C...	3 KB

### 1. \_\_pycache\_\_

- **Description:** A system-generated folder by Python.
- **Function:** Stores compiled .pyc files to improve the runtime performance of Python scripts.
- **Note:** Not intended for manual editing. Usually ignored in version control or final submission.

### 2. venv

- **Description:** The virtual environment for the project.
- **Function:** Contains all the dependencies and Python libraries (such as Flask, pandas, scikit-learn) used in the project in an isolated environment.

- **Benefit:** Prevents conflicts between different project libraries and ensures consistent behavior across systems.

### 3. dataset (2).xlsx

- **Description:** The original dataset file (raw data).
- **Content:** Includes multiple medical records where each row represents symptoms associated with a specific disease.
- **Usage:** Used as the initial input for data analysis and model training.

### 4. dataset\_cleaned.xlsx

- **Description:** A cleaned and preprocessed version of the original dataset.
- **Function:** Missing values and invalid entries were removed, and columns were formatted properly.
- **Usage:** This is the version used for training machine learning models, improving their accuracy and consistency.

### 5. Symptom-severity.xlsx

- **Description:** Contains severity scores for each symptom.
- **Content:** A table mapping symptoms to numerical severity levels (e.g., 1 = mild, 5 = severe).
- **Usage:** Used to weight symptoms during prediction, helping the model assess the criticality of each input case.

### 6. svm\_disease\_prediction.pkl

- **Description:** A trained Support Vector Machine (SVM) model file.

- **Function:** Predicts the most likely disease based on user-input symptoms.
- **Notes:** The model was saved using pickle and can be loaded for real-time inference through the backend server.

#### 7. `svm_disease_prediction_new.pkl`

- **Description:** An improved version of the original SVM model.
- **Improvements:** Trained on the cleaned dataset and possibly optimized parameters, resulting in better performance.
- **Usage:** Recommended for use in the final deployment.

#### 8. `server.py`

- **Description:** The main backend script that runs the Flask server.
  - **Function:**
    - Launches a local server to receive user inputs (symptoms).
    - Loads the trained .pkl model.
    - Processes the input symptoms and returns a disease prediction.
  - **Contents:**
    - `@app.route('/')`: A simple test route returning a welcome message.
    - `@app.route('/predict', methods=['POST'])`: Accepts symptom input and returns the prediction.
  - **Dependencies:** Requires Flask, pickle (or joblib), and pandas.
-

## Relationships Between Components

1. The raw medical data is stored in the dataset (2).xlsx file. This serves as the initial input.
2. This dataset is cleaned and formatted, resulting in dataset\_cleaned.xlsx, which contains high-quality, structured data suitable for machine learning.
3. The cleaned data is used to train a machine learning model (SVM). The trained model is saved in svm\_disease\_prediction.pkl, and later refined into svm\_disease\_prediction\_new.pkl for better accuracy.
4. The file Symptom-severity.xlsx provides numerical severity levels for symptoms, adding another dimension of importance when analyzing patient input.
5. The trained .pkl model is loaded into server.py, the main backend script that operates a Flask server.
6. Users provide a list of symptoms, either through a web interface or API tools like Postman. These symptoms are passed to the backend (server.py), which processes the input using the trained model and returns the predicted disease.

---

## In Summary

The flow of the system starts with medical data and ends with a smart diagnosis system that responds to user input using machine learning.

## 4.2.4 Backend Integration:

```
Future<void> signIn() async {
  try {
    print('Attempting to sign in with email: ${email.text}');
    emit(AuthLoading());
    final res = await api.post(
      Endpoints.signIn,
      data: {
        'email': email.text.trim(),
        'password': password.text,
      },
    );
    user = SigninModel.fromJson(res);
    await CacheHelper.saveData(key: ApiKey.token, value: user!.token);
    print('Sign in successful, token saved');
    await getUserData();
    emit(AuthSuccess());
  } catch (e) {
    print('Sign in failed: $e');
    if (e is DioException && e.response != null) {
      String errorMessage = 'Unknown error';
      if (e.response?.data is Map) {
        final data = e.response?.data as Map;
        errorMessage = data['message']?.toString() ??
          data['error']?.toString() ??
          data.toString();
      } else if (e.response?.data is String) {
        errorMessage = e.response!.data.toString();
      } else {

```

The signIn method sends the user's email and password to the API, handles the response by saving the token if successful, and updates the app state accordingly. It also manages errors by emitting error states with messages

```
Future<void> signUp() async {
  try {
    print('Attempting to sign up with email: ${emailup.text}');
    emit(SignUpLoading());
    final response = await api.post(
      Endpoints.signUp,
      data: {
        ApiKey.name: name.text.trim(),
        ApiKey.email: emailup.text.trim(),
        ApiKey.password: passwordup.text,
        ApiKey.phoneNumber: phone.text.trim(),
        ApiKey.confirmPassword: confirmPassword.text,
        ApiKey.gender: int.parse(gender.text),
        ApiKey.age: int.parse(age.text),
      },
    );
    print('Sign up successful. Response: $response');
    emit(SignUpSuccess());
  } catch (e) {
    print('Sign up failed: $e');
    emit(SignUpError(message: e.toString()));
  }
}
```

The signUp method sends user registration data (name, email, password, phone, gender, age) to the API. It emits loading, success, or error states based on the API response.



```
Future<void> resetPassword() async {  
  try {  
    print('Resetting password');  
    emit(ResetPasswordLoading());  
    final verifiedOtp = CacheHelper.getStringData(key: 'verifiedOtp');  
    if (verifiedOtp == null) {  
      throw Exception('No verified OTP found in cache');  
    }  
    await api.post(  
      Endpoints.resetPassword,  
      data: {  
        'Token': verifiedOtp,  
        'Password': newPassword.text,  
        'ConfirmPassword': confirmNewPassword.text,  
      },  
    );  
    print('Password reset successfully');  
    await CacheHelper.removeData(key: 'otpInput');  
    await CacheHelper.removeData(key: 'verifiedOtp');  
    emit(ResetPasswordSuccess());  
  } catch (e) {  
    print('Failed to reset password: $e');  
    emit(ResetPasswordError(message: e.toString()));  
  }  
}
```

The resetPassword method sends the new password along with a verified OTP token to the backend to update the user's password. It handles loading, success, and error states during the process

```
Future<void> updateProfile() async {  
  try {  
    print('Updating user profile');  
    emit(UpdateProfileLoading());  
    final response = await api.post(  
      Endpoints.updateProfile,  
      data: {  
        ApiKey.name: name.text.trim(),  
        ApiKey.phoneNumber: phone.text.trim(),  
        ApiKey.gender: int.parse(gender.text),  
        ApiKey.age: int.parse(age.text),  
      },  
    );  
    print('Profile updated successfully. Response: $response');  
    emit(UpdateProfileSuccess(user: UserModel.fromJson(response)));  
  } catch (e) {  
    print('Failed to update profile: $e');  
    emit(UpdateProfileError(message: e.toString()));  
  }  
}
```

The updateProfile method sends updated user information (name, phone, gender, age) to the backend to update the user's profile. It manages loading, success, and error states during the update process.

## 4.3 Challenges Faced and How They Were Resolved

Throughout the implementation process, several challenges were encountered. Below are some of the major challenges and the strategies used to resolve them:

- **Challenge: Integrating Machine Learning Models into the System**

Initially, it was challenging to integrate machine learning model(SVM) into the mobile application. The complexity of training models and ensuring they performed well with real-time data was a hurdle.

**Solution:** The models were developed and tested using Python and Google Colab, where we used real medical datasets to train and refine them. Once the models were trained, they were integrated into the backend of the application using Python APIs, ensuring that the disease prediction functionality worked seamlessly.

- **Challenge: Ensuring a Smooth User Experience**

With multiple technologies involved in the development (Flutter for the frontend, SQL database for data storage, and .NET for server-side logic), it was crucial to ensure smooth communication between all components.

**Solution:** Careful attention was given to optimizing API calls and ensuring that data was synchronized properly across the frontend and backend. We used efficient data retrieval strategies and

implemented asynchronous operations to avoid delays and ensure real-time updates.

- **Challenge: Handling Large Volumes of Data**

As the application grew, managing large amounts of data—such as user profiles, doctor information, and medical records—became a challenge. The database queries were slow, leading to performance issues.

**Solution:** SQL optimization techniques, such as indexing, query optimization, and efficient database schema design, were implemented to ensure faster data retrieval.

---

This chapter provides an overview of the technologies, key components, and challenges faced during the implementation of the medical application. The solutions outlined above were essential for overcoming the difficulties and ensuring that the system was functional, secure, and user-friendly.

## **Chapter 5 : Testing & Evaluation**

### **5.1 Testing Strategies**

To ensure the reliability and correctness of the system, several testing strategies were adopted throughout the development cycle:

- **Unit Testing:**

Unit testing was applied to individual components such as the authentication module, appointment scheduling functions, and disease prediction logic. This ensured that each function performed as expected in isolation.

- **Integration Testing:**

Integration tests were conducted to verify that the interaction between different system components—such as the mobile app, .NET backend APIs, the SQL database, and the machine learning model—worked seamlessly. Special attention was given to data flow consistency between the frontend and backend.

- **User Testing:**

User testing sessions were held with a group of real users (including students and individuals interested in health applications). Participants were asked to perform typical tasks, such as registering, selecting symptoms, viewing predictions, and booking appointments. Feedback was collected to identify usability issues and improve user experience.

---

## 5.2 Performance Metrics

The performance of the system was evaluated using a set of quantitative and qualitative metrics:

- **Accuracy:**

The machine learning models (SVM, Naïve Bayes, and KNN) were tested using a labeled dataset of symptoms and diseases. Accuracy scores were calculated, with the SVM model achieving the highest accuracy, followed by KNN and Naïve Bayes.

- **Prediction Speed:**

The system responds with a predicted disease in less than two seconds after symptom selection, ensuring a smooth and real-time experience for users.

- **Scalability:**

Thanks to the use of a structured SQL database, the system is scalable and capable of handling multiple concurrent users without performance degradation.

- **User Satisfaction:**

Informal surveys conducted during user testing showed high satisfaction with the application's simplicity and responsiveness, though some suggestions were made for future enhancements like multilingual support and additional symptoms.

## **Chapter 6: Results & Discussion**

### **6.1 Introduction**

This chapter presents the key outcomes of the project and discusses how the developed system performed in relation to its initial goals. It provides a summary of findings, interprets the effectiveness of the solution, and outlines the limitations observed during implementation and testing.

---

### **6.2 Summary of Findings**

The development and testing of the Medical Center Application yielded several important results:

- The machine learning models (SVM, Naïve Bayes, KNN) successfully predicted diseases based on user-selected symptoms with promising accuracy levels.
  - The mobile application offered a user-friendly interface with smooth navigation and clear options for symptom selection, viewing predictions, and booking appointments.
  - Integration with .net ensured real-time data synchronization and efficient handling of appointment-related operations.
  - The app performed reliably under moderate load conditions, with minimal delays during prediction and booking processes.
- 

### **6.3 Interpretation of Results (Did the Project Meet its Objectives?)**

The project successfully met its primary objectives, which were:

- **Developing a mobile-based medical platform:**  
✓ Achieved. A Flutter-based mobile application was created, providing users with access to essential health services from symptom input to appointment booking.
- **Integrating a machine learning-based disease prediction system:**  
✓ Achieved. Three ML models were implemented and tested. The SVM model delivered the highest accuracy (92%) and was used as the default prediction model in the final version.
- **Enabling real-time interaction and booking functionality:**  
✓ Achieved. .net integration allowed seamless and immediate updates to appointments, ensuring synchronization between users and doctors.
- **Improving user experience and usability:**  
✓ Mostly achieved. User testing indicated high levels of satisfaction with system responsiveness and interface design, though a few users suggested expanding the symptom list and including more detailed predictions in future versions.

Overall, the system accomplished its core objectives and demonstrated the potential of using AI-enhanced mobile applications in the medical domain.

---

## 6.4 Limitations of the Proposed Solution

Despite its success, the system has several limitations:

- **Limited symptom and disease dataset:**  
The prediction system was trained on a limited dataset, which may not cover rare or complex diseases. This reduces the system's ability to handle edge cases accurately.



- **No multilingual support:**

The application currently supports only English, which may limit its accessibility for users with limited English proficiency.

- **No real-time doctor availability integration:**

While users can book appointments, the system does not yet reflect doctors' real-time availability or working hours dynamically.

- **No personalization or historical tracking:**

The system does not currently save user history or offer personalized recommendations based on previous symptoms or bookings.

- **Not a substitute for professional diagnosis:**

The disease prediction is a preliminary suggestion and should not be relied upon for critical medical decisions. The system is designed to assist, not replace, human healthcare professionals.

## **Chapter 7 : Conclusion & Future Work**

### **7.1 Summary of Contributions**

This project involved developing a mobile application designed to integrate multiple healthcare services into a unified and user-friendly platform.

We addressed a critical need for an accessible and reliable tool to simplify doctor appointment bookings, facilitate home care service scheduling, provide trusted health education resources, and support early disease prediction based on user-reported symptoms.

The application leveraged advanced mobile development frameworks such as Flutter for cross-platform compatibility, and .net for backend services including authentication and database management.

In addition, the project incorporated machine learning algorithms, specifically Support Vector Machines (SVM), to enable accurate and fast disease prediction based on symptoms.

The project was divided into carefully structured phases, covering requirement analysis, UI/UX design, machine learning model selection, mobile development, integration, testing, and optimization,

ensuring the delivery of a highly functional and efficient final product.

## 7.2 Future Work

### 1. Integrating Medical Image Analysis

Future versions of the system can include image classification models (e.g., for skin rashes or X-rays) using **MediaPipe** or **TensorFlow Lite**, enhancing diagnostic accuracy through visual data.

### 2. Smart Doctor Recommendation System

Implementing a recommendation engine based on user symptoms, doctor ratings, specialization, and geographic location to suggest the most suitable healthcare provider.

### 3. Natural Language Processing for Symptom Input

Allowing users to describe their symptoms in plain language and leveraging **NLP** to extract relevant medical data, improving usability for non-technical users.

### 4. Improved Accessibility Features

Incorporating accessibility tools such as **voice command navigation**, **screen readers**, and **haptic feedback** to make the app more inclusive for users with disabilities.

### 5. Integration with Electronic Medical Records (EMR)

Future iterations can include linking with EMR systems to use patients' medical histories for more accurate and personalized predictions.

### 6. Continuous Health Monitoring via IoT Devices

Expanding the system to connect with sensors (e.g., DHT22,

blood pressure monitors) for real-time health tracking and automatic symptom analysis.

#### **7. Emergency Alert System**

Implementing an alert mechanism to notify emergency contacts or healthcare providers when high-risk symptoms are detected.

#### **8. Model Training with Real Clinical Data**

Collaborating with hospitals or healthcare institutions to access real-world datasets, improving model performance while ensuring data privacy.

#### **9. Clinical Validation and Pilot Studies**

Conducting clinical trials or pilot studies with medical professionals to evaluate the model's practical effectiveness and refine predictions in real scenarios.

---

## Chapter 8 References:

1. Figma, "Figma: The collaborative interface design tool," [Online]. Available: <https://www.figma.com>.
2. Flutter Team. "Flutter Documentation." [Online]. Available: <https://flutter.dev/docs>
3. Microsoft. ".NET Documentation." [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/>
4. Scikit-learn Developers. "Support Vector Machines." [Online]. Available: <https://scikit-learn.org/stable/modules/svm.html>
5. Kaggle. "Datasets for Disease Prediction." [Online]. Available: <https://www.kaggle.com/>
6. Analytics Vidhya. "Introduction to Disease Prediction using ML." [Online]. Available: <https://www.analyticsvidhya.com/>
7. Brownlee, J. (2020). "**Machine Learning Algorithms From Scratch.**" Machine Learning Mastery. [Online]. Available: <https://machinelearningmastery.com>
8. Google Colab Team. "Welcome to Colaboratory." [Online]. Available: <https://colab.research.google.com>

9. W3Schools. “SQL Tutorial.” [Online]. Available:  
<https://www.w3schools.com/sql>
-