Egyptian E-Learning University

Faculty of Computers & Information Technology

A Real-time AI-powered Healthcare Diagnostic Application for Patients

By

Youssef Ashraf Abdelaziz	2101727
Moaz Osama Ahmed	2101130
Mohamed Tharwat Abdelaziz	2101135
Kareem Yasser Khalaf	2100834
Youssef Abdelnaby Subra	2101713
Abdelrahman Youssef	2100287

Supervised by

Dr. Mohamed Zidan

Lecturer in Computer and Information Technology Egyptian E-Learning University

Assistant

Eng. Mohamed Moustafa Saad

Teacher Assistant in Computer and Information Technology Egyptian E-Learning University

Sohag - 2025

Abstract

Tamenny is an AI-powered mobile healthcare application designed to provide users with fast, preliminary medical diagnoses by analyzing medical scans and user-reported symptoms. The app addresses key healthcare challenges such as limited access to specialists, high diagnostic costs, and the urgent need for early detection of chronic diseases. Using advanced AI technologies, ***Tamenny*** can assess medical images related to lung cancer, heart disease, brain cancer, and knee osteoarthritis with high confidence and accuracy.

The application integrates a user-friendly chatbot that enables real-time symptom analysis through natural language interaction, offering users possible conditions and recommendations based on their input. Additionally, ***Tamenny*** features a built-in community platform where users can connect with specialized doctors for medical advice, second opinions, or guidance, creating a supportive and interactive healthcare environment.

The methodology involves training deep learning models on publicly available medical datasets, optimizing for accuracy and performance. The models are then integrated into the mobile app to ensure seamless image classification and chatbot responses. Initial evaluations show promising accuracy rates in disease classification tasks, indicating the potential of ***Tamenny*** as a reliable pre-diagnostic tool.

By combining image-based AI diagnostics, conversational interfaces, and a doctor-patient community, ***Tamenny*** aims to enhance healthcare accessibility, reduce patient anxiety, and encourage timely medical intervention, particularly for individuals in remote or underserved areas.

Acknowledgments

We would like to express our deepest and most sincere gratitude to **Dr. Mohamed Zidan** for his unwavering guidance, continuous support, and constant encouragement throughout every stage of our graduation project. His profound expertise, insightful feedback, and thoughtful suggestions have been instrumental in shaping the direction and quality of our work. His mentorship not only enriched our technical knowledge but also inspired us to strive for excellence and maintain a strong sense of commitment and responsibility.

We also extend our heartfelt thanks to **Eng. Mohamed Mostafa Saad** for his invaluable technical advice and hands-on assistance during the implementation phase. His prompt responses to our inquiries, constructive suggestions, and practical solutions helped us overcome numerous challenges and significantly enhance the performance and functionality of our application. His contribution was a vital pillar in our project's success.

Furthermore, we would like to express our sincere appreciation to all those who stood by us during this journey, our families, friends, and colleagues. Their unwavering support, endless patience, and constant motivation have been a source of strength for us. Their belief in our capabilities and in the value of this project kept us going through moments of difficulty and doubt.

To everyone who contributed, directly or indirectly, to the realization of this project thank you from the bottom of our hearts.

Contents

List Of Figures

List Of Tables

Chapter 1

Introduction

1.1 Introduction

In recent years, the demand for accessible, efficient, and cost-effective healthcare solutions has grown significantly. This surge is largely due to the increasing prevalence of chronic and life-threatening diseases such as lung cancer, heart disease, brain tumors, and knee osteoarthritis. These conditions, if not detected and treated early, can lead to severe health complications, decreased quality of life, and even death. However, early detection and timely intervention are often hindered by several barriers, including the limited availability of specialized healthcare professionals in remote or rural regions, the high costs of medical consultations and diagnostic procedures, and the long waiting times typically associated with traditional healthcare systems.

With the rapid advancement of Artificial Intelligence (AI), the healthcare sector is witnessing a revolutionary transformation. AI technologies have introduced new possibilities for analyzing medical data, interpreting medical images, and predicting health outcomes with high accuracy. Leveraging these capabilities, AI-powered applications can now assist in the early detection of serious medical conditions, enabling quicker decision-making and reducing the burden on healthcare systems.

Our project, ***Tamenny***, responds directly to these challenges by providing a smart, mobile-based healthcare solution that empowers users to receive preliminary medical evaluations without needing immediate clinical intervention. By using AI algorithms trained to recognize patterns in medical scans and patient-reported symptoms, ***Tamenny*** offers users accurate assessments related to lung cancer, heart disease, brain cancer, and knee osteoarthritis. Furthermore, the app features an intelligent chatbot that facilitates symptom checking through natural language interaction and provides users with possible explanations and recommended next steps. This is complemented by a community-based feature that connects users with qualified healthcare professionals for additional advice, creating a supportive and informative environment.

By integrating AI technology with user-centric design, ***Tamenny*** aims to make healthcare more accessible, especially for individuals who face logistical, financial, or geographic barriers to traditional care. It acts as a bridge between patients and the formal medical system, encouraging early action and informed health decisions.

1.2 background and motivation for the project

In today's rapidly evolving world, access to timely and effective healthcare remains a significant challenge for many individuals around the globe. Despite advancements in medicine and technology, a large portion of the population continues to face obstacles when it comes to obtaining early and accurate diagnoses. People often delay seeking medical attention due to several factors, such as financial burdens, long waiting times, lack of transportation, or the absence of specialized healthcare professionals in their regions. These delays are especially dangerous when it comes to chronic and life-threatening conditions such as lung cancer**,** brain tumors**,** knee osteoarthritis**,** and heart disease. These diseases require early detection and intervention to improve survival rates and patient outcomes, yet the traditional healthcare systems often fall short in offering accessible and timely diagnostic services.

This disparity between the need for early diagnosis and the availability of medical resources was a major driving force behind our project. We recognized the urgent necessity of developing a scalable and intelligent solution that could support users in conducting preliminary health assessments from the comfort of their homes. The growth of Artificial Intelligence (AI) in recent years opened up new avenues to close this gap. AI has demonstrated great potential in analyzing medical images, interpreting symptoms, and generating predictive insights with high accuracy. Leveraging these capabilities, we envisioned a mobile platform that empowers users with the tools they need to better understand their health and make informed decisions.

Our team was particularly motivated by the potential to reach underserved communities rural areas, low-income populations, and regions with limited access to specialists by placing diagnostic power in the hands of the users. The project not only seeks to reduce dependency on traditional systems but also to decentralize healthcare through smart, AI-powered assistance.

In addition, we recognized that technology alone isn't enough. Health concerns are deeply personal, and users often seek reassurance, emotional support, and expert guidance. To address this, we included a community feature within the app, allowing users to engage with medical professionals and fellow patients. The chatbot functionality also adds an intuitive and interactive element, enabling users to describe their symptoms in everyday language and receive a meaningful preliminary analysis. This combination of intelligent diagnosis and human connection increases the app's usefulness and impact.

Ultimately, our motivation stems from a desire to contribute to a more inclusive healthcare ecosystem one that bridges the gap between technology and humanity. With ***Tamenny***, we aim to enhance early detection, reduce the burden on healthcare systems, and support users in managing their health with confidence and ease.

1.3 Importance of the Problem Being Addressed

Diseases such as cancer including lung and brain cancer and degenerative joint conditions like knee osteoarthritis are often referred to as "silent threats" because they can progress gradually without showing obvious symptoms in the early stages. By the time noticeable discomfort or functional limitations appear, the disease may have already reached an advanced stage, making treatment more complex, expensive, and less likely to be successful. For instance, lung cancer detected in stage I has a significantly higher survival rate compared to cases discovered in stage III or IV. Similarly, knee osteoarthritis that is not managed early can lead to irreversible joint damage and disability, greatly impacting a person's quality of life.

The consequences of late diagnosis extend beyond individual patients. They place immense strain on national healthcare systems due to the need for more intensive care, long-term treatment, and higher medical expenditures. Early detection, on the other hand, not only improves survival rates and recovery outcomes but also reduces the cost of care and resource consumption.

Unfortunately, a large segment of the global population particularly those in rural, remote, or low-income communities faces significant barriers in accessing healthcare services. These include a lack of medical infrastructure, a shortage of specialized doctors, and limited access to diagnostic tools. As a result, individuals are often left undiagnosed until symptoms become severe or debilitating.

Bridging this gap through modern technology is both a practical and urgent necessity. By leveraging artificial intelligence, we can develop tools that analyze medical scans and symptoms with high accuracy, offering users preliminary insights into their condition without the need for immediate clinical intervention. An AI-powered application like ***Tamenny*** addresses these issues by bringing intelligent, scalable, and low-cost diagnostics directly to the user's mobile device. This empowers individuals to take charge of their health, make informed decisions early, and seek timely medical assistance when necessary.

Moreover, the integration of such technologies supports the broader vision of personalized and preventive healthcare. It promotes health equity by ensuring that early diagnosis is not a privilege limited to urban or wealthy populations but a right accessible to all. By tackling the root causes of delayed diagnosis imited access, awareness, and affordability, AI-based solutions like ***Tamenny*** offer a transformative pathway toward a more inclusive and responsive healthcare system.

1.4 Problem Statement

Definition:
A critical challenge facing global healthcare systems today is the lack of accessible, affordable, and timely diagnostic tools for serious health conditions such as lung cancer, brain cancer, heart disease, and knee osteoarthritis. These illnesses often progress silently and, without early intervention, can lead to irreversible damage or even death. In many parts of the world, particularly in remote or underserved communities, individuals face significant obstacles in obtaining proper medical evaluations due to a shortage of healthcare professionals, long waiting times, high costs, and travel constraints. As a result, a large portion of the population is left undiagnosed until the disease has reached an advanced stage when treatment becomes less effective, more invasive, and substantially more expensive.

Moreover, even in urban areas with access to hospitals and clinics, the overload on healthcare systems often leads to delayed appointments and overburdened specialists. People may ignore initial symptoms due to uncertainty or inconvenience, only seeking help when their condition has significantly deteriorated.

This systemic delay in diagnosis directly impacts treatment outcomes, patient survival rates, and healthcare costs.

Justification:

Recent advances in Artificial Intelligence, particularly in machine learning and deep learning, have opened the door to new possibilities in healthcare diagnostics. By training models to analyze medical scans and detect patterns associated with specific diseases, AI systems can act as first-line diagnostic tools, offering users rapid, reliable, and cost-effective evaluations from their mobile devices. These tools are not intended to replace doctors but to support and enhance their diagnostic capabilities, especially in the early detection phase.

Developing an AI-powered application that can provide preliminary assessments for conditions like lung cancer, heart disease, brain tumors, and joint degeneration would bridge the gap between the patient and the healthcare system. It empowers individuals to take control of their health, promoting awareness, early action, and ultimately better health outcomes. This is particularly important for populations with limited access to healthcare professionals, where early screening can mean the difference between life and death.

Solving this problem is no longer just a technological challenge it is a humanitarian necessity. By integrating AI technologies into user-friendly mobile platforms like ***Tamenny***, we aim to transform how healthcare is delivered and accessed, making early diagnostics available to everyone, regardless of their location or income level.

1.5 Objectives

The main objective of this project is to design and develop a mobile healthcare application that utilizes AI technologies to provide users with preliminary medical assessments by analyzing medical images and symptoms. In pursuit of this goal, the project encompasses several specific objectives. First, it aims to build robust AI models capable of accurately classifying lung cancer, brain tumors, and knee osteoarthritis using a variety of medical imaging techniques, such as CT scans, MRIs, and X-rays. These models are trained on validated medical datasets and optimized for mobile deployment to ensure efficient performance.

Second, the project seeks to integrate a smart chatbot within the application, which will interact with users in natural language, assess described symptoms, and provide real-time, context-aware health insights. This feature not only enhances user engagement but also serves as a valuable tool for individuals who may not have immediate access to a medical professional.

Another key objective is the development of a community feature that fosters direct communication between users and specialized medical professionals. This interaction is crucial for building trust, offering second opinions, and delivering emotional support, especially for users in remote or underserved areas.

Furthermore, the system is being designed with a strong emphasis on user-friendliness, accuracy, and scalability. This involves creating an intuitive interface, ensuring responsive design for various devices, and incorporating security protocols to protect user data. The final objective includes a thorough evaluation phase, where the AI models and the overall system will be tested using both established medical datasets and real-world scenarios. This step is vital for validating the reliability, usability, and clinical relevance of the application.

1.6 Brief Overview of the Proposed Solution

Tamenny is an AI-powered mobile application designed to assist users in receiving fast, initial medical evaluations. The app allows users to upload medical scans (e.g., CT, MRI, or X-rays) related to lung cancer, brain tumors, and knee osteoarthritis. These images are processed by trained AI models that provide predictions about potential health conditions. In addition, a built-in chatbot enables users to describe their symptoms in natural language and receive guided insights. A community section further allows users to interact with certified medical professionals for second opinions or guidance. Altogether, this system aims to reduce the diagnostic delay, support patient awareness, and improve overall healthcare accessibility.

Chapter 2

Related Work

Title	Year	Author	result
Heart Disease Diagnosis Using Deep Learning	**2023**	**Cheerla Ganesh, Lordson Gnana Durai A,Baavana Bandarupalli,S. Pavithra, R. Prabanjan**	Breakthrough heart disease predictor hits 90% accuracy (4.4% higher than current methods) by analyzing critical health markers. Enables earlier detection and better treatment decisions to save lives.
Enhancing Lung Cancer Classification: Leveraging Existing Convolutional Neural Networks within a 1D Framework	**2024**	**Nurul Najiha Jafery, Siti Noraini Sulaiman, Muhammad Khusairi Osman, Noor Khairiah A. Karim, Zainal Hisham Che Soh**	Advanced AI detects lung cancer early with 94.67% accuracy (VGG-16 model), significantly boosting survival rates through timely diagnosis.
Osteoporosis Prediction Using VGG16 and ResNet50	**2024**	**Ashadu Jaman Shawon, Ibrahim Ibne Mostafa Gazi, Humaira Rashid Hiya, and Ajoy Roy.**	A breakthrough AI model detects osteoporosis with 96% accuracy by integrating ResNet50 (98%) and VGG16 (78%) through advanced image analysis techniques.

Brain Tumor Detection Using Convolutional Neural Network	**2024**	**Vijay Mane, Atharva Balasaheb Chivate, Prajyot Ambekar, Atul Rajaram Chavan, and Ameya Pangavhane**	A novel CNN-based deep learning model detects brain tumors in MRI scans with 94.2% accuracy and 0.1314 loss. It demonstrates high precision, showcasing the effectiveness of CNNs in medical imaging analysis. Future enhancements may include tumor segmentation, multi-class classification, and integration of multi-modal data for improved diagnostics.

Table 2.1 : Related Work

Chapter 3

Proposed system

3.1 Approach Used to Solve the Problem

To effectively address the challenges of delayed medical diagnosis, limited health awareness, and restricted access to nearby healthcare services, ***Tamenny*** was conceptualized as a comprehensive digital health companion. The system combines multiple technologies and user-centric design principles to empower individuals to manage their health proactively and independently. The proposed approach integrates artificial intelligence, social interaction, geolocation, health education, and conversational interfaces within a single, intuitive mobile application. Each component of this hybrid solution contributes to a holistic healthcare experience, ensuring users receive timely information, support, and guidance.

In the area of early diagnosis, ***Tamenny*** leverages AI-based medical image analysis powered by deep learning. A custom-trained Convolutional Neural Network (CNN) was developed using publicly available medical datasets containing X-ray and MRI scans for specific conditions such as osteoarthritis and chest anomalies. Once a user uploads an image, the model performs pattern recognition to detect signs of potential abnormalities. The system then provides a preliminary diagnostic report along with a confidence score and personalized recommendations. This functionality is especially beneficial for users in remote or underserved areas, as it offers rapid insights without requiring immediate access to medical professionals.

To enhance user engagement and emotional support, the application includes a community interaction feature inspired by social networking platforms. Users can share their health experiences, post questions, and interact with others through comments and reactions. The community module operates through a moderated, real-time database system that supports asynchronous communication while maintaining user privacy and respectful discourse. This design encourages peer-to-peer support, promotes open health discussions, and improves long-term user retention by turning health tracking into a socially engaging activity.

Recognizing the importance of accessibility to professional care, ***Tamenny*** integrates geolocation services to help users find nearby healthcare providers. Using the device's GPS and mapping APIs such as Google Maps Platform, the app displays a list of nearby clinics, doctors, and hospitals. Each provider profile includes essential information such as specialization, location, distance, user ratings, and direct contact options. An interactive map interface with route navigation further assists users in physically reaching the selected healthcare facility, thus bridging the gap between virtual assistance and real-world medical access.

To improve public health awareness, ***Tamenny*** offers a dedicated section for educational content, including health tips, wellness articles, and updates from trusted medical sources. This section is managed through a content management system (CMS) that allows for dynamic content organization and regular updates. The information is categorized by topic and aligned with common health concerns derived from user behavior within the app. By exposing users to preventive knowledge and healthy practices, this module contributes to informed decision-making and the adoption of long-term health-positive habits.

Finally, to provide users with immediate, conversational support, ***Tamenny*** features an intelligent chatbot built using Natural Language Processing (NLP) techniques. The chatbot is capable of understanding user inputs through intent classification and keyword extraction, allowing it to respond to a variety of health-related inquiries. It assists users in navigating the app, answering symptom-related questions, suggesting relevant content, and providing guidance on whether to seek further medical consultation. The chatbot's dialogue system was fine-tuned on health-specific interactions, making it a valuable 24/7 assistant that enhances accessibility, especially for users seeking quick answers or those with limited technical experience.

3.2 System architecture

Tamenny's system is composed of three main layers: the Client Layer (mobile app), the Backend Layer (database and services), and the AI Engine (for scan analysis and recommendation controll) . The architecture follows a modular and scalable design, ensuring maintainability and performance across mobile platforms. The following diagrams illustrate the interaction between components, user flow, and data structure.

3.2.1 Entity Relationship Diagram Overview

The Entity Relationship (ER) diagram below visualizes the core data model of ***Tamenny***, representing how different components within the system interact. It highlights the main entities such as **User**, **Post**, **Comment**, **Scan**, and **Doctor**, and defines their relationships, enabling seamless management of user-generated content, medical data, and healthcare provider information.

The **User** entity is central to the system, representing individuals using the app. Users can create multiple posts, upload medical scans, and add comments. Posts support interactions through likes and comments, reflecting a social and supportive environment. Each scan is associated with its uploader, providing a personal diagnostic history. The **Doctor** entity remains independent and is used to populate nearby healthcare services based on geolocation data.

The diagram is designed to reflect how these entities are implemented in Firestore and optimized through collections and subcollections, ensuring efficient data access, scalability, and real-time synchronization.

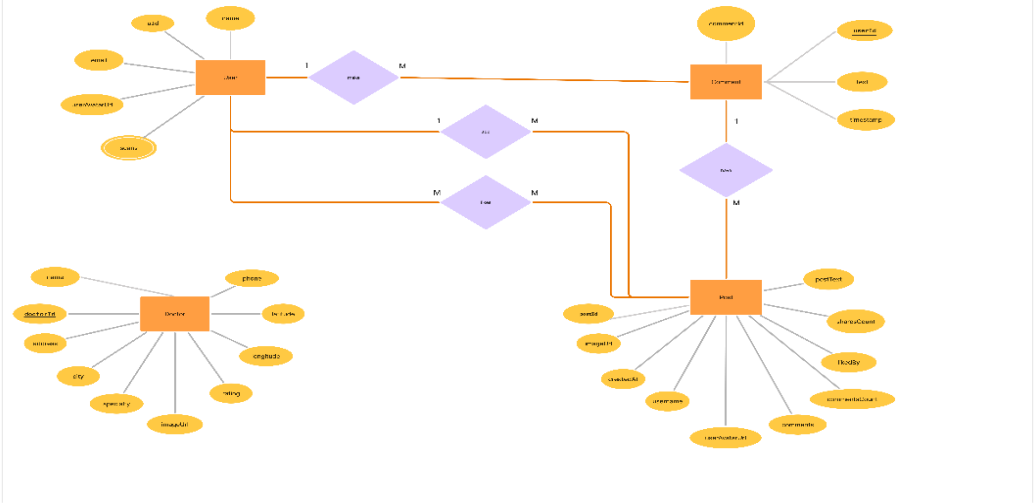


Figure 3.1 : ER Diagram detailing the app's core database structure

The following logical schema diagram illustrates the implementation-level view of the ***Tamenny*** app's database. It defines the main tables, attributes, and foreign key relationships among **Users**, **Posts**, **Comments**, and **Doctors**. This model reflects how the data is stored and managed in the backend using a NoSQL-like structure simulated with relational logic for clarity and maintainability. [12],[13]

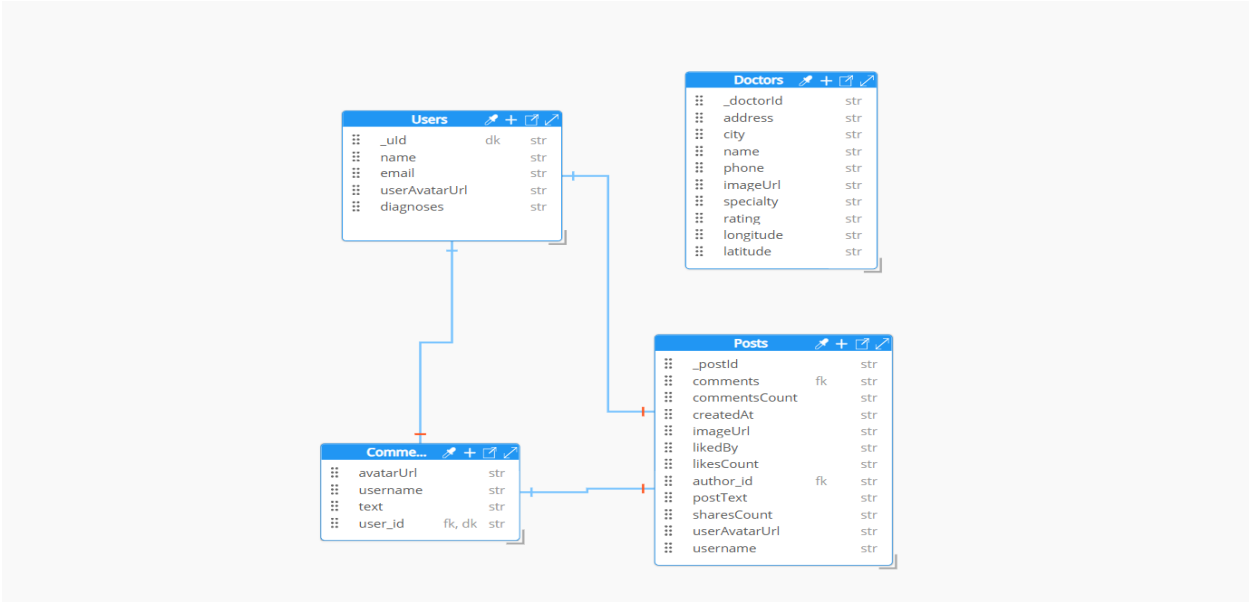


Figure 3.2 : for the logical schema structure

3.2.2 User Flow Architecture

- **Authentication Flow**

When the user opens the application, they are directed to the authentication interface, which acts as the entry point to the system. This interface includes three main options: **Login** for existing users, **Register** for new users who wish to create an account, and **Forgot Password** for users who need to reset their credentials. After completing any of these actions successfully, the user is navigated to the **Home Screen**, where they can access the application's main features

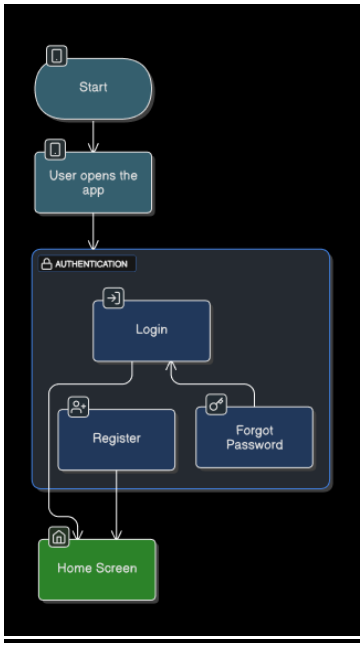


Figure 3.3 : Authentication flow

- **AI Diagnosis Flow**

After successfully logging into the application and reaching the home screen, the user can upload a medical scan through the **Upload Scan** feature. Once the scan is uploaded, it is automatically sent to the AI engine for processing.

The system then analyzes the scan and generates a **Diagnosis Result**, which is displayed to the user along with the relevant health indicators. Based on the AI's analysis, the user is either shown a confirmed diagnosis or possible conditions. Finally, the user is prompted to **Provide Feedback** regarding the accuracy or helpfulness of the diagnosis, which can be used to further improve the system's performance.

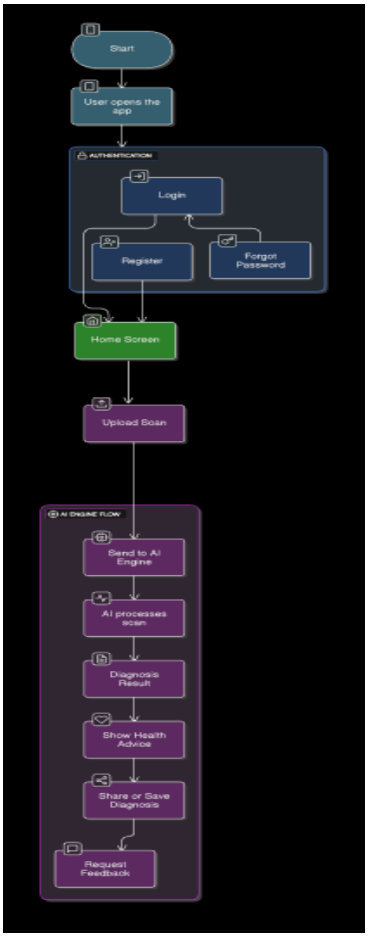


Figure 3.4 : AI Diagnosis Flow

- **Scan History Flow**

From the home screen, users can access the **Scan History** section to view previously submitted medical scans. Within this section, the system displays a list of past scans through the **Show Previous Scans** feature. Each scan in the list can be either **opened** for review or **deleted** if the user no longer wishes to keep it stored. This functionality enables users to manage their medical history efficiently and provides quick access to prior diagnostic results when needed.

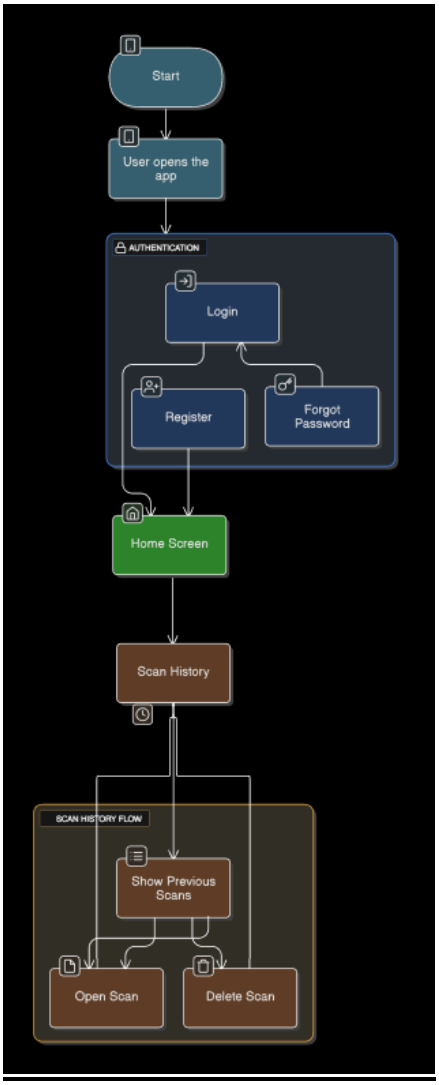


Figure 3.5 : Scan History Flow

- **Community Flow**

From the Home Screen, users can access the **Community Posts** section to engage with shared content from other users. Within this section, the system allows users to **view posts**, providing a scrollable feed of published content based on the user's preferences to ensure the complete benefit.

Once inside a post, users can **like**, **comment on**, or **share** the post with others. This functionality fosters user interaction, encourages community engagement, and creates a space for social support and knowledge exchange within the application.

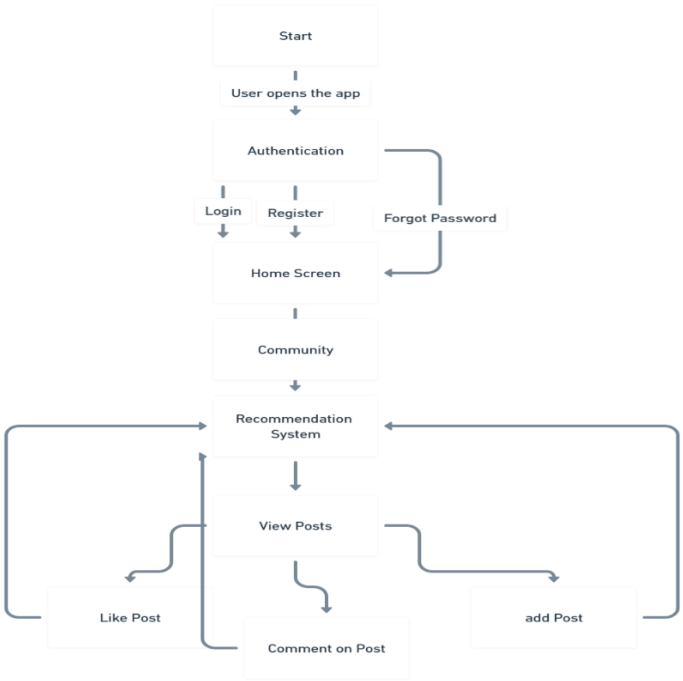


Figure 3.6 : Community Flow

- **Nearby Doctors Flow**

From the Home Screen, users can navigate to the Nearby Doctors section to locate medical professionals based on their current geographic location. The system automatically detects the user's position and displays a list of doctors available in the surrounding area.

Upon selecting a doctor from the list, the user is directed to a detailed profile page containing the doctor's name, specialty, contact information, and rating. Additionally, users have the option to open the doctor's location on an interactive map for navigation purposes, or initiate a direct phone call for appointment booking or inquiries.
This flow streamlines the process of accessing nearby healthcare providers, making medical assistance more reachable and immediate for users.

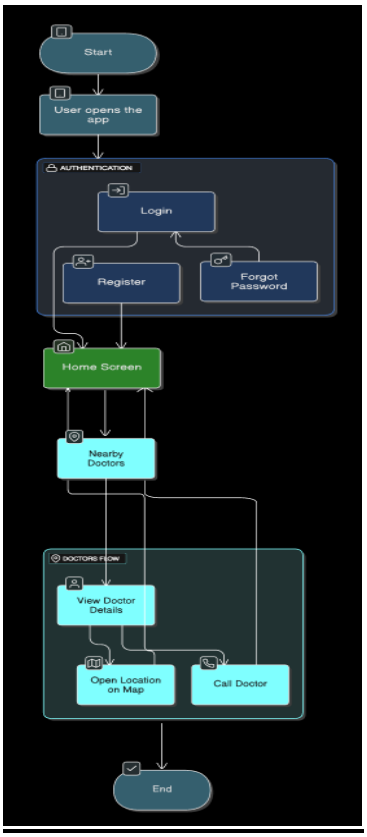


Figure 3.7 : Nearby Doctors Flow

- **Map flow**

From the Home Screen, users can access the Map section to explore the locations of nearby doctors. Upon entering this section, the system presents an Interactive Map View that displays the user's current location along with the surrounding area.

Within the map, users have the option to activate the "Show Nearby Doctors" feature, which reveals markers indicating the locations of available doctors in the vicinity. This functionality aims to streamline access to medical services, offering users a visual and intuitive way to discover healthcare options near them.

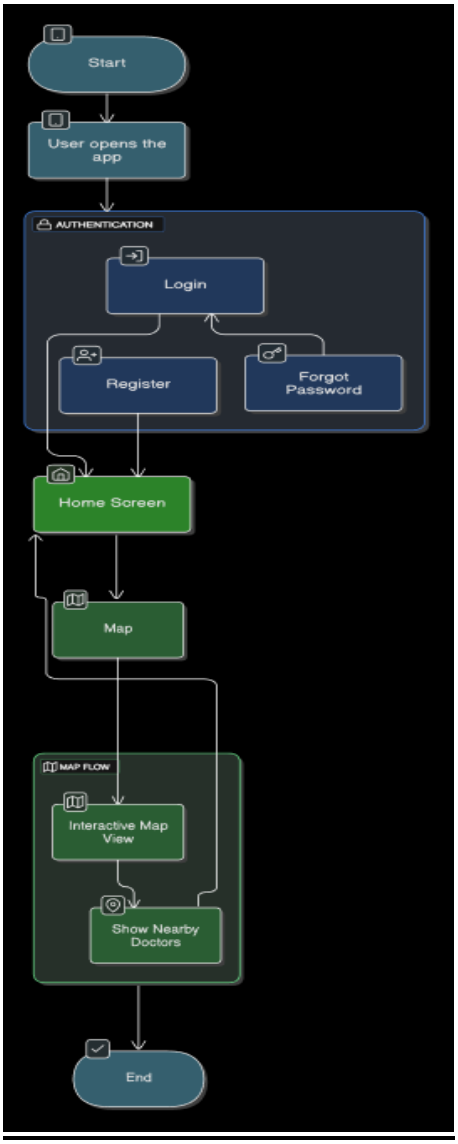


Figure 3.8 : Map Flow

- **New flow**

From the Home Screen, users can navigate to the News section to stay updated with the latest health-related updates and announcements w general and specific based on the user preferences with the recommendation System . Within this section, the system presents a list of the most recent news articles. Users can browse through the list and open individual news items to read full articles. This feature is designed to keep users informed, enhance their awareness of current health trends, and deliver important updates directly within the app environment.

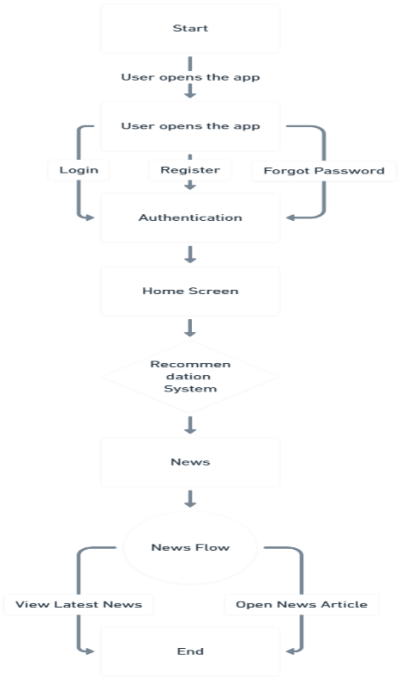


Figure 3.9 : News Flow

- **Health Tips Screen**

From the Home Screen, users can navigate to the Health Tips section to access educational content focused on wellness and healthy living. Within this section, users are presented with categorized health tips, allowing them to explore advice and information based on specific topics. By selecting a topic, users can view relevant tips tailored to their interests or health needs. This feature is designed to promote preventive care, increase user awareness, and encourage healthier lifestyle choices through accessible and practical guidance.

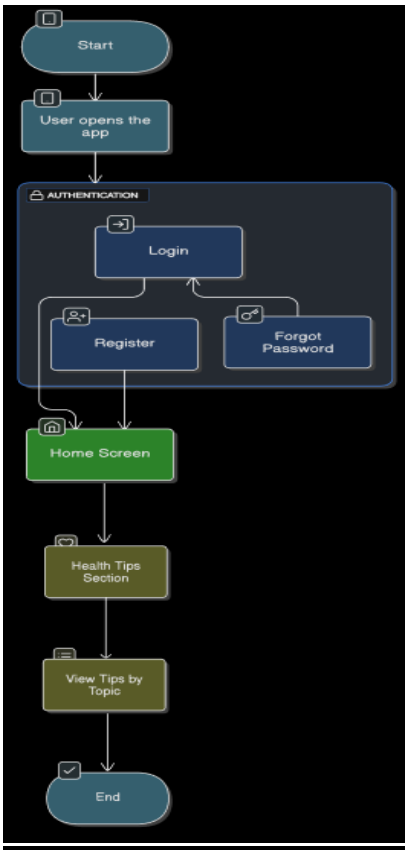


Figure 3.10 : Health Tips Flow

- **Profile Flow**

From the Home Screen, users can access the Profile section to manage their personal settings and preferences. Within this section, users have the option to view or edit their profile information, log out of the application, and review their scan history. Additionally, the Profile area includes access to app settings such as enabling or disabling notifications, switching between dark and light modes, and changing the application language. This flow is designed to provide users with full control over their account and application experience, ensuring a personalized and user-friendly interface.

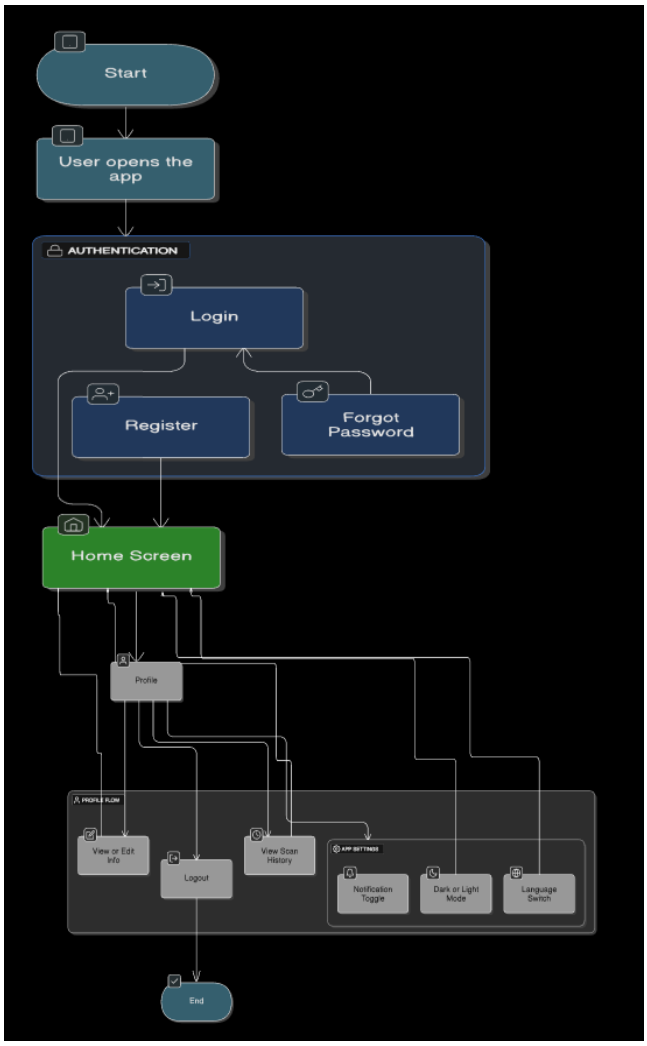


Figure 3.11 : Profile Flow

3.3 Algorithms or frameworks used.

3.3.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized type of deep learning architecture particularly suited for image-based tasks such as classification, detection, and segmentation. Their structure is optimized to automatically extract spatial hierarchies and patterns from image data, which makes them an ideal tool for medical imaging applications. [1],[2],[3]

Benefits Of CNN Model:

In this project, CNNs were implemented to classify brain tumor images with high accuracy and robustness. The power of CNNs in **automatic feature learning** removed the need for manual extraction of image features, allowing the model to learn directly from raw input images. This contributed to a significant improvement in diagnostic performance. Moreover, the **scalability** of CNNs enabled us to adapt the model architecture according to the complexity of the dataset, while their **robustness** ensured reliable predictions across different types of brain tumors. The **high accuracy** and generalization ability of CNNs made them an essential component in our medical diagnosis system. [1],[2],[3]

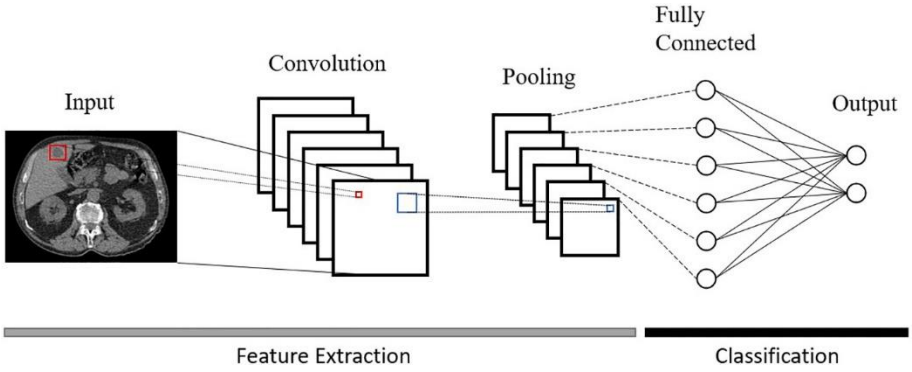


Figure 3.12 : Architecture of CNN

CNN Architecture:

The architecture of a CNN model typically consists of several layers, each designed to process and extract information from input images in a hierarchical fashion. The **Input Layer** receives the medical image, which is usually in the form of a 3D array representing the image height, width, and color channels. For example, an RGB brain scan of size 128×128 pixels would be represented as (128, 128, 3). The first core layer applied is the **Convolutional Layer**, which performs the operation of scanning the image using multiple filters (kernels). These filters detect low-level features such as edges, textures, or color variations by sliding across the image and computing dot products between the filter and the local regions of the input.

After the convolution operation, a non-linear activation function is applied most commonly the **ReLU (Rectified Linear Unit)** which introduces non-linearity into the model by converting negative values to zero. This is crucial for enabling the model to learn complex and non-linear patterns in medical images. To reduce the spatial dimensions of the feature maps and computational cost, a **Pooling Layer** is then applied, most often **Max Pooling**. This selects the maximum value from small sub-regions of the image, effectively downsampling the image and making the model more efficient and less prone to overfitting.

In addition, **Batch Normalization** is used to normalize the inputs of layers, helping stabilize and speed up the training process. **Dropout Layers** are also introduced as a form of regularization, randomly disabling a fraction of neurons during training to prevent the model from becoming too reliant on specific pathways. Following these operations, the 2D feature maps are passed through a **Flattening Layer**, which transforms them into a 1D vector, allowing them to be input into the fully connected layers.

The **Fully Connected (Dense) Layers** perform the final classification by combining all learned features. Each neuron in these layers is connected to every neuron in the previous layer. Finally, the **Output Layer** uses a **Softmax activation function**, which provides a probability distribution over the output classes, such as Normal, Benign, or Malignant. This structured and layered approach allows CNNs to learn and refine medical image features at multiple levels of abstraction, resulting in highly accurate and reliable classification outcomes. [1],[2],[3]

3.3.2 U-Net

Benefits of U-Net and Its Application in the Model:

U-Net provides several key advantages that make it an excellent choice for semantic segmentation, especially in the medical field. One of the major benefits is its high accuracy on small datasets, which is particularly important in medical imaging, where annotated data is often limited. In my model, this characteristic allowed for effective training despite the small number of labeled samples, ensuring that the segmentation quality remained high. Another important feature is the ability of U-Net to preserve context and fine-grained details thanks to the use of skip connections, which transfer spatial information directly from the **encoder layers** to the corresponding decoder layers. This was critical in my model, as it enabled precise delineation of anatomical structures like organs or tumors. U-Net is also known for its efficient training process, as it does not rely on pretraining and can be trained from scratch with relatively few resources. This made it practical for use in my segmentation pipeline. Finally, the architecture is highly versatile and adaptable to different tasks. In my implementation, I used U-Net for segmenting organ boundaries in CT images, which greatly improved the accuracy of the subsequent diagnosis and analysis tasks. [4],[5]

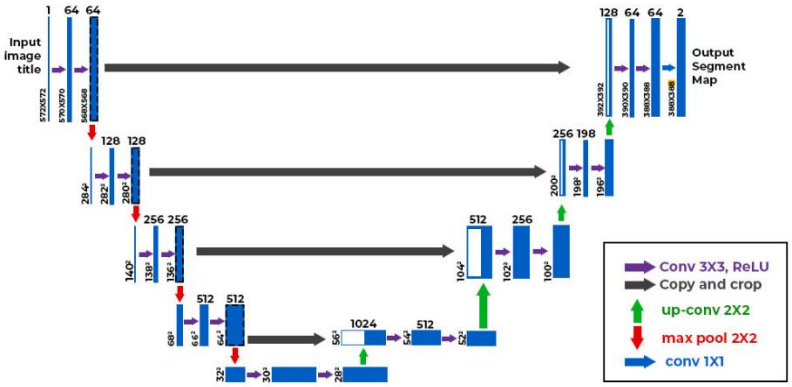


Figure 3.13 : U-Net Architecture

U-Net Architecture:

The architecture of U-Net follows a U-shaped structure composed of a Contracting Path (Encoder), a Bottleneck, an Expanding Path (Decoder), and a final Output Layer. The **Contracting Path (Encoder)**is responsible for capturing the context of the image by applying repeated blocks of operations. Each block typically consists of two **Convolutional Layers** with ReLU Activation, optionally followed by **Batch Normalization** to stabilize training, and then a **Max Pooling Layer** to downsample the feature maps. As we go deeper into the encoder, the spatial resolution decreases, while the number of channels increases, allowing the model to learn more abstract representations of the image content. At the bottom of the architecture lies the Bottleneck, which acts as the transition between encoding and decoding. It captures the most compressed and abstract features of the image and usually includes two **Convolutional Layers** with ReLU Activation. This part serves as a high-level summarization of the input, preserving the essential semantic information needed for reconstruction. The **Expanding Path (Decoder)** is then responsible for reconstructing the segmentation mask from the compressed representation. Each step in the decoder begins with an Up-sampling Layer, which increases the resolution of the feature maps often using **Transposed Convolution** (also called Up-convolution). The upsampled features are then concatenated with the corresponding feature maps from the encoder via Skip Connections, which restore spatial details that might have been lost during the downsampling process. After concatenation, two more **Convolutional Layers** with **ReLU Activation** are applied to refine the features. Finally, the **Output Layer** applies a **1×1 Convolutional Layer**t o map the feature maps to the desired number of output classes. Depending on the segmentation task, a **Softmax Activation** is used for multi-class output, or a **Sigmoid Activation** for binary segmentation. This structure enables U-Net to generate pixel-wise classification maps with high precision**.** [4],[5]

3.3.3 ResNet-50

Benefits of ResNet-50 in Medical Imaging:

ResNet-50 offers several advantages that made it highly effective for the classification of knee images in this project. The most significant benefit lies in its ability to utilize **pretrained weights** from large-scale datasets like ImageNet. This enabled **transfer learning**, allowing the model to be adapted for medical image classification tasks even with limited knee X-ray data. Furthermore, its **deep architecture** with 50 layers allowed the network to learn complex hierarchical features essential for distinguishing subtle differences in joint structure and identifying early signs of osteoarthritis. Another advantage is its **residual connections**, which helped mitigate the vanishing gradient problem and made it possible to train a deeper model without degradation in accuracy. This improved the model's **robustness and generalization** capabilities. The model also benefited from **data augmentation techniques** during training, such as random flips and rotations, which further reduced the risk of overfitting. These characteristics made ResNet-50 an ideal backbone for reliable and accurate classification of knee conditions.[6]

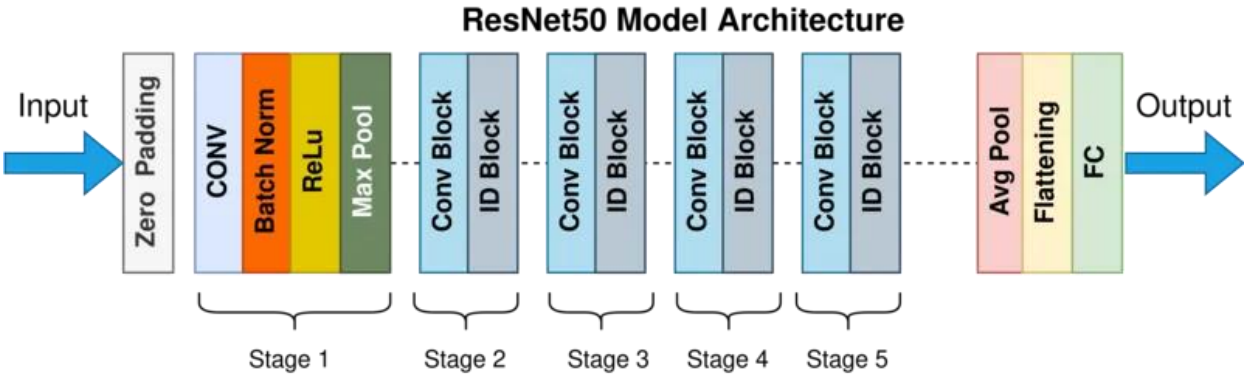


Figure 3.14 : Resnet-50 Architecture

ResNet-50 Architecture:

The architecture of **ResNet-50** is built around the concept of **residual learning**, where shortcut (or skip) connections allow gradients to pass through multiple layers without vanishing. The network begins with an **initial convolutional layer**, where a **7×7 convolution** with 64 filters and stride 2 is followed by **batch normalization** and a **ReLU** activation function. This is immediately followed by a **3×3 max pooling layer** with stride 2, which reduces the spatial resolution of the input.

The core of ResNet-50 is composed of **residual blocks**, each of which contains **three convolutional layers**: a 1×1 convolution to reduce dimensions, a 3×3 convolution for processing, and another 1×1 convolution to restore dimensions. These layers are accompanied by **batch normalization** and **ReLU** activation. Each residual block also includes a **shortcut connection**, which adds the input directly to the output of the block enabling the model to learn residual mappings. The blocks are organized into four stages:

- **Stage 1** includes 3 residual blocks and increases feature channels from 64 to 256.
- **Stage 2** has 4 blocks, expanding channels from 128 to 512.
- **Stage 3** includes 6 blocks, going from 256 to 1024 channels.
- **Stage 4** has 3 blocks and increases features from 512 to 2048 channels.

As the stages progress, **spatial resolution decreases** due to stride-2 convolutions, while the **depth of features increases**.

Following the final stage, the network uses a **global average pooling layer**, which reduces the spatial dimension to a single **2048-dimensional vector** a compact representation of the learned features. Finally, a **fully connected (dense) layer** maps this vector to the number of target classes, and the output passes through a **softmax activation** function to generate probability scores for multi-class classification tasks.

This architecture combines depth, stability, and powerful feature extraction, making ResNet-50 well-suited for medical image classification and analysis. [6]

3.3.4 Flan-T5

Flan-T5 is an advanced transformer-based model that builds upon the Text-to-Text Transfer Transformer (T5) by incorporating instruction tuning, thereby improving its ability to follow natural-language prompts and generalize across diverse tasks. In the ***Tamenny*** application, Flan-T5 powers the bilingual (Arabic/English) symptom-analysis chatbot. By framing every interaction as a text-to-text problem, the model can flexibly handle symptom descriptions, follow-up questions, and clarification requests, producing fluent responses that guide users toward appropriate next steps. This versatility reduces the need for handcrafted dialogue rules and allows rapid adaptation to new medical domains with only minimal additional fine-tuning.[7]

Benefits of Flan-T5 in This Model:

Harnessing Flan-T5 confers several advantages for ***Tamenny's*** conversational layer. First, its **instruction-tuned** training regimen delivers strong zero-shot and few-shot performance, enabling the chatbot to interpret varied symptom narratives even those not explicitly seen during fine-tuning. Second, the model's **text-to-text framework** simplifies pipeline design: all inputs and outputs remain plain text, making integration with the mobile front-end straightforward. Third, Flan-T5's pretrained knowledge allows effective operation on relatively small, domain-specific datasets, which is crucial because large, labeled medical-dialogue corpora are scarce. [7]

Finally, its availability in multiple size variants lets developers strike an optimal balance between response latency on mobile devices and linguistic richness, ensuring that users receive timely, contextually appropriate guidance.

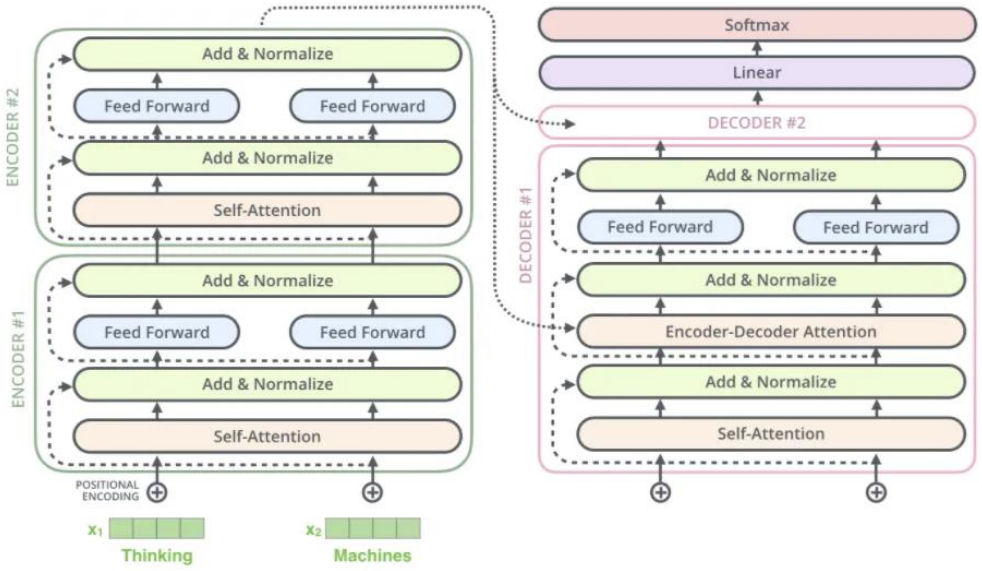


Figure 3.15 : Flan T5 Architecture

Flan-T5 Architecture:

Flan-T5 adopts the canonical transformer **encoder–decoder** structure tailored to text-generation tasks. The **Encoder** begins by converting tokenized input text into dense vector embeddings augmented with positional encodings that preserve word order. Within each encoder block, **Self-Attention** layers allow every token to attend to all others, capturing long-range dependencies, while position-wise **Feed-Forward Networks** refine token representations; both components are wrapped in **Residual Connections** and **Layer Normalization** to stabilize learning. The encoder outputs a contextualized sequence that feeds into the **Bottleneck**, serving as a compressed semantic representation.

The **Decoder** generates output autoregressively: at each time step, **Masked Self-Attention** ensures that prediction of the next token considers only previously generated tokens, whereas **Cross-Attention** layers fuse the decoder's partial output with the encoder's context. Each decoder block mirrors the encoder's structure with its own **Feed-Forward Network**, residual pathways, and normalization layers, facilitating deep information flow. A final **Linear Projection Layer** maps decoder states to vocabulary logits, and a **Softmax** operation produces probability distributions from which output tokens are selected typically via greedy decoding or beam search during inference. This architecture empowers Flan-T5 to deliver coherent, context-aware responses that enhance the user experience within the ***Tamenny*** health-assistant ecosystem.[7]

3.3.5 XLM-RoBERTa

Benefits of XLM-RoBERTa base in text classifaction

XLM-RoBERTa base offers several key advantages for text classification tasks, particularly in multilingual and complex language settings. As a transformer-based model trained on 100 languages, it is highly effective in handling multilingual data, allowing for accurate classification across different languages without needing separate models. Its deep contextual understanding, developed through extensive pretraining on massive multilingual corpora, enables it to grasp subtle semantic and syntactic nuances, which significantly boosts classification accuracy. Compared to earlier models like mBERT, XLM-RoBERTa achieves state-of-the-art performance on various multilingual NLP benchmarks and excels in both zero-shot and cross-lingual transfer learning scenarios, making it especially valuable for low-resource languages. The base version strikes a practical balance between performance and efficiency, with around 125 million parameters, making it suitable for fine-tuning on standard GPU setups. Additionally, its ability to process long and complex text inputs makes it ideal for real-world applications, such as analyzing medical notes, social media posts, or legal documents. Integration into modern NLP pipelines is straightforward thanks to robust support from the Hugging Face Transformers library, ensuring easy deployment and community-backed development.[8]

XLM-RoBERTa architecture :

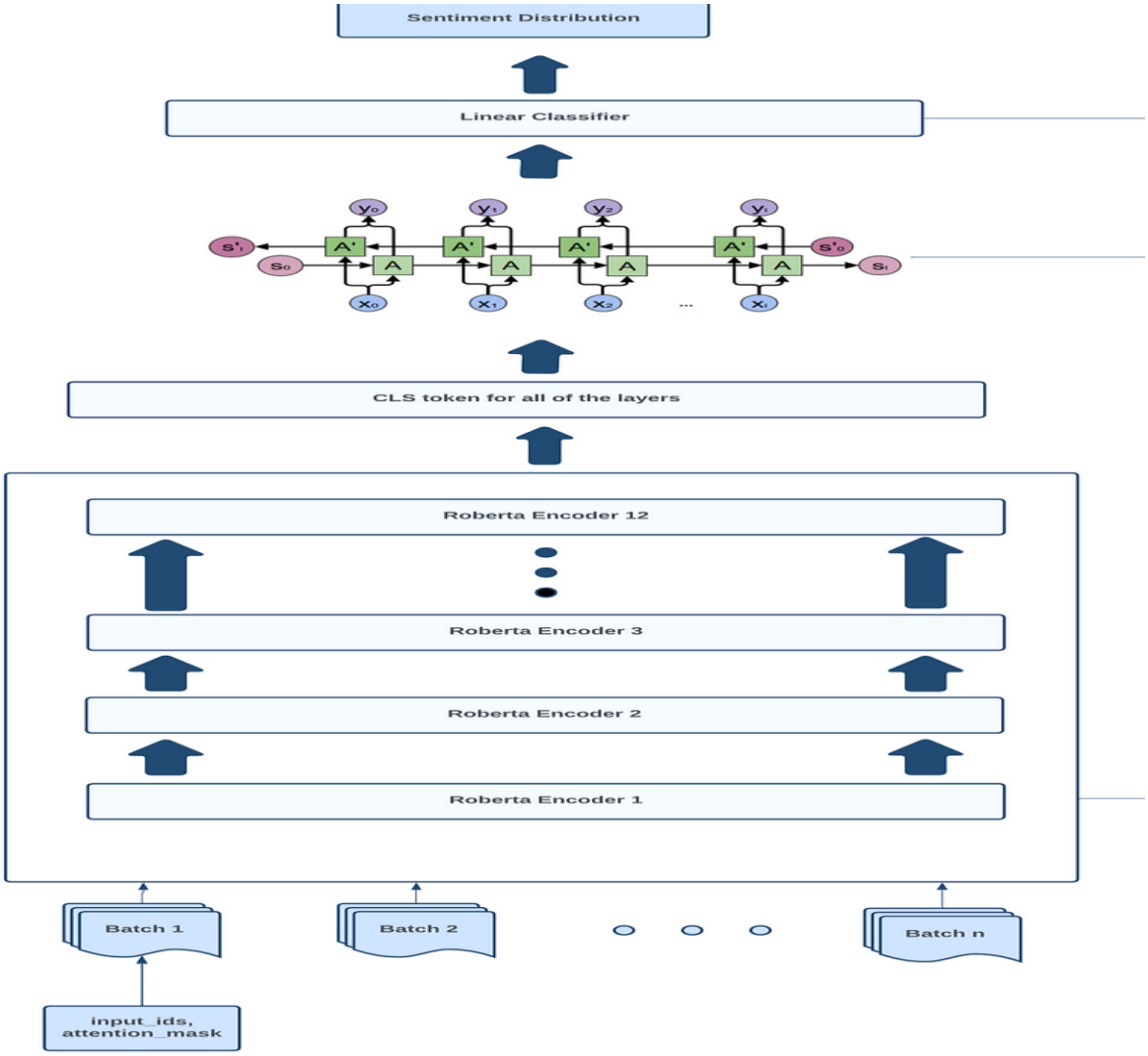


Figure 3.16 : XLM-RoBERTa

XLM-RoBERTa base is built on the Transformer encoder architecture, originally introduced by Vaswani et al., and follows the same design as RoBERTa, with adaptations for multilingual data. It consists of 12 Transformer layers (also called encoder blocks), each containing self-attention mechanisms and feed-forward neural networks, with a hidden size of 768 and 12 attention heads. The model processes input text using a subword tokenizer based on SentencePiece, allowing it to handle diverse languages and scripts without relying on language-specific rules.

Unlike BERT, XLM-RoBERTa is trained without the Next Sentence Prediction (NSP) objective, using only masked language modeling (MLM) over a much larger and more diverse multilingual corpus. This allows the model to learn richer contextual representations across languages. The architecture's attention mechanism enables the model to capture dependencies between words regardless of their position in the sentence, making it highly effective for understanding and classifying text in varied linguistic contexts.[8]

3.3.6 NN model in multivariate regression

Benefits of using NN model in multivariate regression:

Neural networks offer significant advantages in multivariate regression tasks due to their ability to model complex, nonlinear relationships between multiple inputs and outputs. Unlike traditional linear regression models, neural networks can learn intricate patterns and interactions in the data without requiring manual feature engineering. This makes them particularly useful in scenarios where the underlying data distribution is non-linear or where multiple output variables are interdependent. Their flexibility allows them to scale to high-dimensional datasets and adapt to a wide range of problem domains, from financial forecasting to medical diagnostics. Furthermore, by sharing hidden layers across multiple outputs, neural networks can capture common underlying features, leading to more accurate and generalized predictions compared to training separate models for each target variable. With advances in training techniques and hardware acceleration, neural networks have become practical and powerful tools for multivariate regression tasks in real-world applications. [9]

NN architecture:

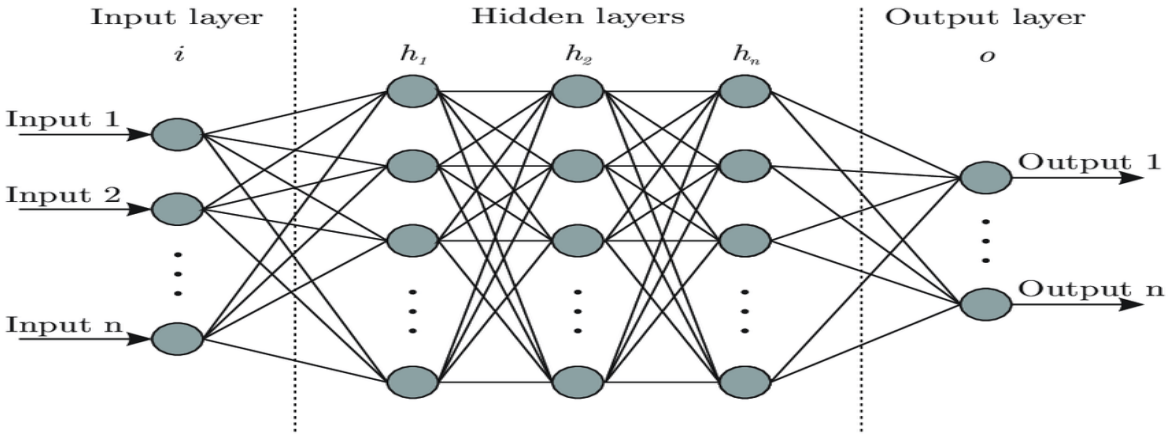


Figure 3.17 NN Architecture

The neural network architecture is composed of an input layer, multiple hidden layers, and an output layer tailored to the specific task. Each hidden layer consists of fully connected (dense) units followed by ReLU (Rectified Linear Unit) activation functions to introduce non-linearity. To improve generalization and prevent overfitting, dropout layers are included after selected hidden layers. Batch normalization is applied to stabilize and accelerate training. The output layer uses a softmax activation function for multi-class classification tasks, producing a probability distribution over the target classes. The model is trained using the cross-entropy loss function and optimized with the Adam optimizer for efficient convergence. [9]

3.3.7 Frameworks and Libraries Used

The development of the medical imaging models relied on a range of powerful frameworks and libraries to streamline deep learning, image processing, data manipulation, and visualization. The core deep learning framework was **TensorFlow**, combined with the high-level **Keras** API to simplify the construction and training of neural networks like CNN and U-Net. For efficient image handling and augmentation, **OpenCV**, **PIL**, and **Albumentations** were employed, allowing for advanced preprocessing and transformation of medical images.

NumPy played a crucial role in performing numerical operations and managing multi-dimensional arrays throughout the pipeline.

For evaluation and machine learning utilities, **Scikit-learn** was used to compute classification metrics, generate confusion matrices, and split datasets. To visualize data distributions and training performance, **Matplotlib** and **Seaborn** provided clear, customizable plotting tools. Additionally, GPU acceleration during training was enabled through **CUDA** via **TensorFlow**, ensuring faster computation.

Chapter 4

Implementation

4.1 Technologies, Tools, and Programming Languages Used:

The system was developed using a combination of advanced technologies and essential tools across both the mobile application and AI model implementation layers. For mobile development, the **Dart** programming language was used in conjunction with **Flutter**, enabling cross-platform deployment for both Android and iOS. State management was handled using **Provider**, with **BLoC** applied in more complex scenarios. Navigation within the app utilized **Flutter Navigator** and **MaterialPageRoute**, while the user interface design followed **Material Design 3** principles and employed **custom widgets** and **responsive layouts** for optimal display across devices. Local data storage was managed through **Hive**, and API interactions were facilitated by **Dio**, which also handled error responses. Users could upload images using **Image Picker** and **File**, while smooth user experience was ensured through **Lottie** animations and **AnimatedContainer**. Development and testing were supported by tools such as **Flutter DevTools** and widget testing features.

For AI model development and backend integration, **Python** served as the core programming language. In natural language processing tasks, the **FLAN-T5 Base** model was integrated using the **Transformers** library. Development and experimentation were conducted within **Jupyter Notebook** environments. For medical image processing tasks such as segmentation and classification, models like **U-Net** and **CNN** were trained using **TensorFlow** and **Keras**. Image handling and preprocessing were achieved through tools like **OpenCV**, **PIL**, and **Albumentations**, while **NumPy** provided the computational backbone for numerical operations. Data visualization during training and evaluation phases was handled using **Matplotlib** and **Seaborn**. Model evaluation was performed using **Scikit-learn**, which supported metrics such as confusion matrices and classification reports. To enhance training performance, techniques like **EarlyStopping** and **ModelCheckpoint** were applied, and hardware acceleration was leveraged through **CUDA**. Additionally, the **OS** library in Python assisted in managing file paths and directories throughout the project lifecycle.

4.2 Key Components/Modules of the System:

The onboarding module introduces new users to the app's concept in a simple and friendly way through a few welcome screens. It includes a get started button and remembers completion so it only appears once.

The scan upload module allows users to upload medical scans directly from their camera or gallery. During the upload, a circular progress indicator is shown, along with motivational tips or health-related messages to improve the experience.

The result page module displays the AI-generated diagnosis in a clear and easy-to-read format. Users can save the report locally or share it with the community for feedback and support.

The community module provides a space for users to share their health experiences, ask questions, and interact with others. Users can comment, like posts, and follow discussions, creating a supportive and engaged environment.

The map module shows nearby doctors and clinics based on the user's location. An interactive map displays clickable markers with the doctor's name, specialty, distance, and contact options, using OpenStreetMap to support navigation.

The search module helps users quickly find what they're looking for, whether it's doctors, posts, scan history, or FAQs. It includes intelligent suggestions, filters, and sorting to improve usability.

The notification module delivers push alerts for important updates like scan result availability, replies in the community, or scheduled health tips, helping users stay informed and engaged.

The settings module supports both Arabic and English with full RTL support for Arabic users. It also includes preferences like enabling dark mode to personalize the experience.[16]

4.2.1 User Interface Screens and Descriptions

This section provides detailed descriptions of the user interface (UI) screens for the ***Tamenny*** application, a smart healthcare platform leveraging artificial intelligence for preliminary medical diagnoses and facilitating connections with nearby doctors. Each screen is designed to enhance user experience, ensuring accessibility and functionality for diverse user groups. The descriptions below outline the purpose, key features, and contributions to the overall user experience. [10]

Onboarding & Authentication:

- **Splash Screen**

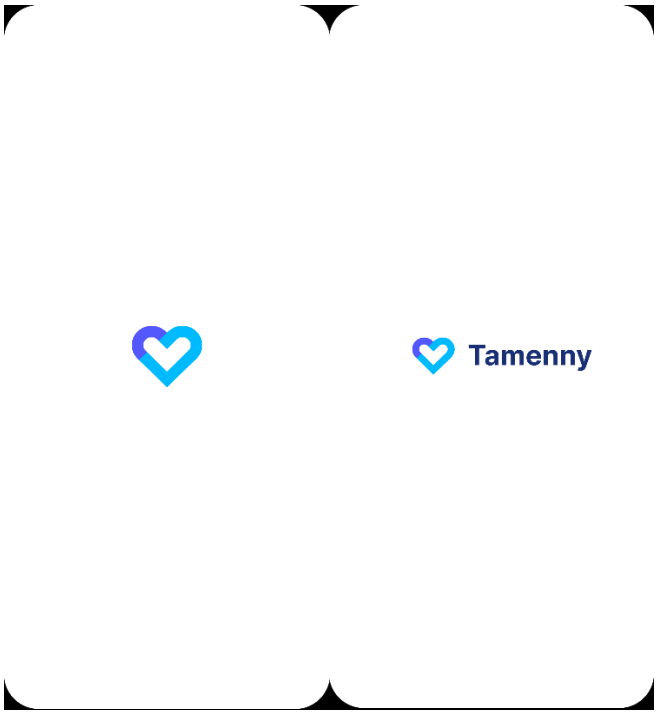


Figure 4.18 : Splash Screen

When the ***Tamenny*** application is launched, the first interface that appears is the Splash Screen. This screen serves both functional and aesthetic purposes. Functionally, it provides a short loading period for the application to initialize essential resources in the background. Visually, it offers users an immediate, polished impression that sets the tone for the overall experience.

The design of the Splash Screen is intentionally minimalistic, incorporating the ***Tamenny*** logo and the application name against a clean background.

In some versions, light animations are used to enhance visual appeal and create a sense of smoothness and professionalism. This subtle motion engages users while maintaining a calm and focused atmosphere.

Typically, the Splash Screen remains visible for 2 to 3 seconds before smoothly transitioning to the next step in the user journey whether that's the Welcome Screen or the Onboarding flow. Despite its brief appearance, the Splash Screen plays an important role in shaping user perception. A well-designed splash experience communicates brand identity, assures quality, and helps establish user trust from the very first interaction.

- **Welcome Screen**

Figure 4.19 : Welcome Screen

The Welcome Screen serves as the user's official introduction to the ***Tamenny*** application. Positioned immediately after the Splash Screen, it plays a vital role in establishing a strong first impression by combining visual appeal with clear navigation options. At the center of this screen is the ***Tamenny*** logo accompanied by a short, meaningful tagline such as "Your Health, Our Priority" which reinforces the brand's mission and identity.

The design reflects the app's healthcare focus through calming color schemes and clean visuals, helping to build trust and ease from the very beginning. Depending on the user's status, the screen offers intuitive pathways: first-time users are invited to begin the onboarding process through a prominent "Get Started" button, while returning users are given the option to jump directly to the login or registration screens.

This screen significantly contributes to the overall user experience by ensuring that navigation is clear and welcoming, regardless of the user's level of technical familiarity. Its simplicity and visual clarity make it easy for users to understand where to go next, setting the tone for a seamless and accessible interaction with the application.

- **Onboarding Screens**

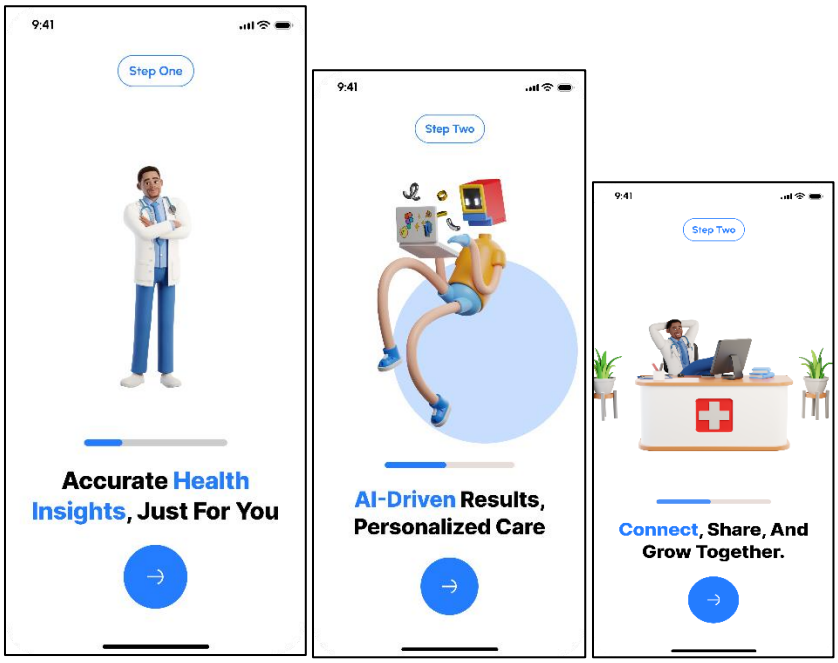


Figure 4.20 : Onboarding Screens

The Onboarding Screens are designed to provide new users with a smooth and informative introduction to the core functionalities of the ***Tamenny*** application. Upon first use, users are guided through a series of visually engaging slides that emphasize the app's main features ranging from symptom input and AI-powered diagnosis, to finding nearby doctors through an interactive map, and participating in the in-app community.

Each slide presents a specific feature with clear, concise language supported by relevant graphics or icons that enhance understanding and retention. Navigation is intuitive, allowing users to swipe between screens at their own pace. Additionally, "Next" and "Previous" controls are available for structured progression, while a "Skip" option lets users bypass the onboarding flow if they prefer to explore the app independently.

These screens play an essential role in enhancing the user experience, particularly for individuals who may be unfamiliar with digital health platforms. By offering a quick, focused tour of what the app can do, the onboarding process helps users feel confident, informed, and ready to engage with ***Tamenny*** from the very beginning.

- **Login & Signup Screens**

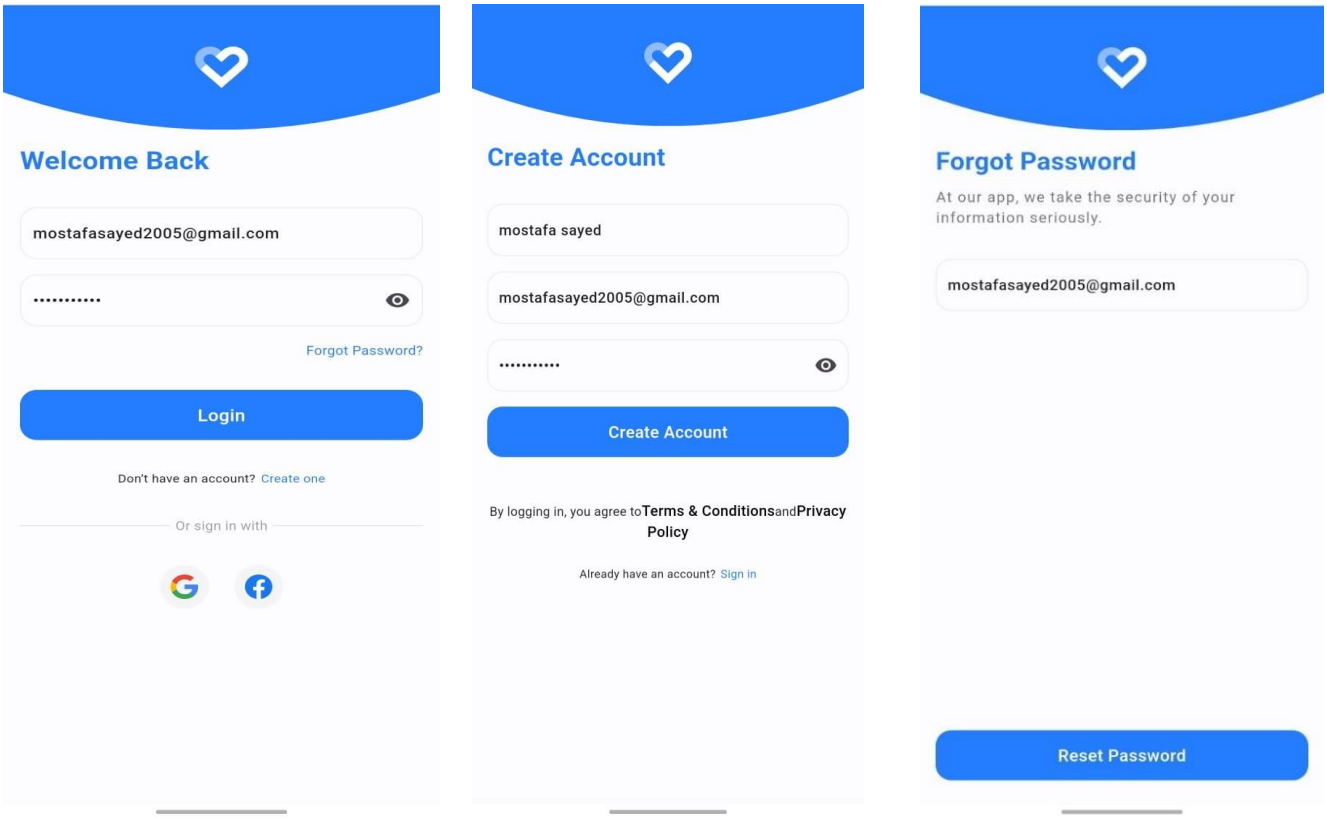


Figure 4.21 : Login and Signup

The Login and Signup Screens form the core of the user authentication flow within the ***Tamenny*** application. These screens are designed to be intuitive and accessible, allowing both new and returning users to securely access their accounts with minimal friction. Existing users are presented with the Login Screen, where they can enter their email and password to sign in. In case of forgotten credentials, a "Forgot Password" link is provided to facilitate account recovery through a secure reset process.

For new users, the Signup Screen enables the creation of a personal account by collecting essential information such as name, email, and password, along with optional demographic fields like age or gender. To further enhance convenience, both screens may offer alternative sign-in methods through third-party platforms such as Google or Facebook, simplifying the process for users who prefer social authentication.

Additionally, the **Forgot Password** flow is designed to be simple yet secure. When users select the “Forgot Password” option, they are guided through a multi-step recovery process that typically involves entering their registered email address, receiving a verification code or reset link, and then creating a new password. This ensures that users can regain access to their accounts swiftly without compromising security. The process is optimized to minimize frustration, especially for users who may not be technically inclined.

By offering a clean, well-structured interface and robust account recovery options, the authentication flow significantly contributes to the overall user experience. It ensures that users regardless of their technical background—can easily and confidently access the app, making the onboarding process smoother and reducing potential entry barriers.

4.2.2 Account Management and Personal Data

- **User Profile Screen**

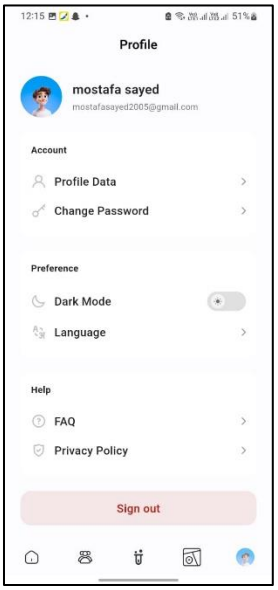


Figure 4.22 : User Profile

The User Profile Screen serves as the central hub where users can view and manage their personal and medical information, as well as customize their overall experience within the ***Tamenny*** application. Designed with clarity and accessibility in mind, this screen begins with a header displaying the user's name and email address, along with a placeholder avatar that reflects the user's profile picture. While the image itself is currently a simple blue cartoon icon, users can later update their photo through the profile editing interface.

Beneath the header, the screen is structured into three main categories: **Account**, **Preference**, and **Help**. Each of these categories leads to dedicated sub-screens where users can manage detailed aspects of their experience in a focused and organized manner.

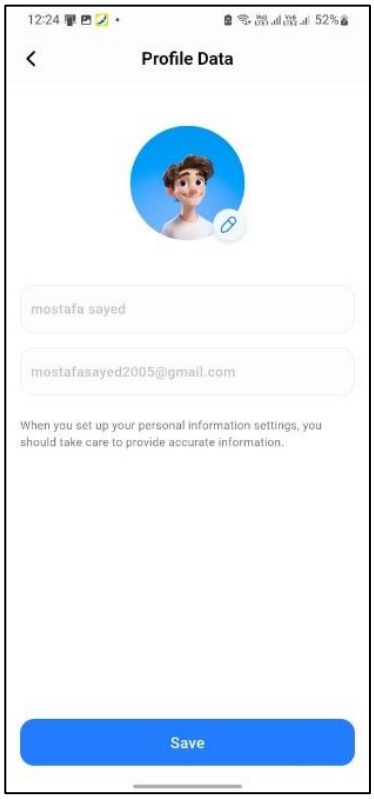


Figure 4.23 : Profile Data

The **Account** section allows users to access their profile data, where they can update key personal such as name, avatar, email This section also includes an option to securely update the email via a separate screen dedicated to password management.

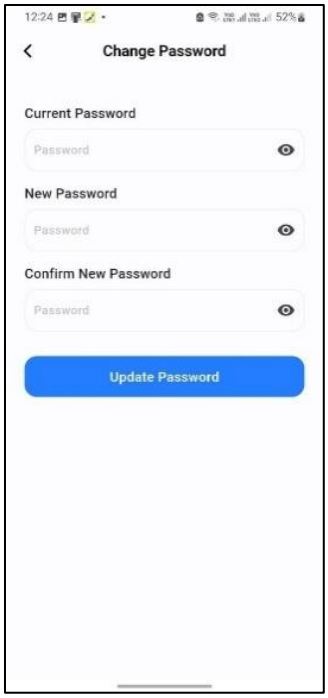


Figure 4.24 : Change Password

This screen allows users to securely update their password from within the **Account** section. It includes three fields: **Current Password**, **New Password**, and **Confirm New Password**, with visibility toggles for ease of use. Clear error messages and a confirmation prompt ensure a smooth, secure experience.

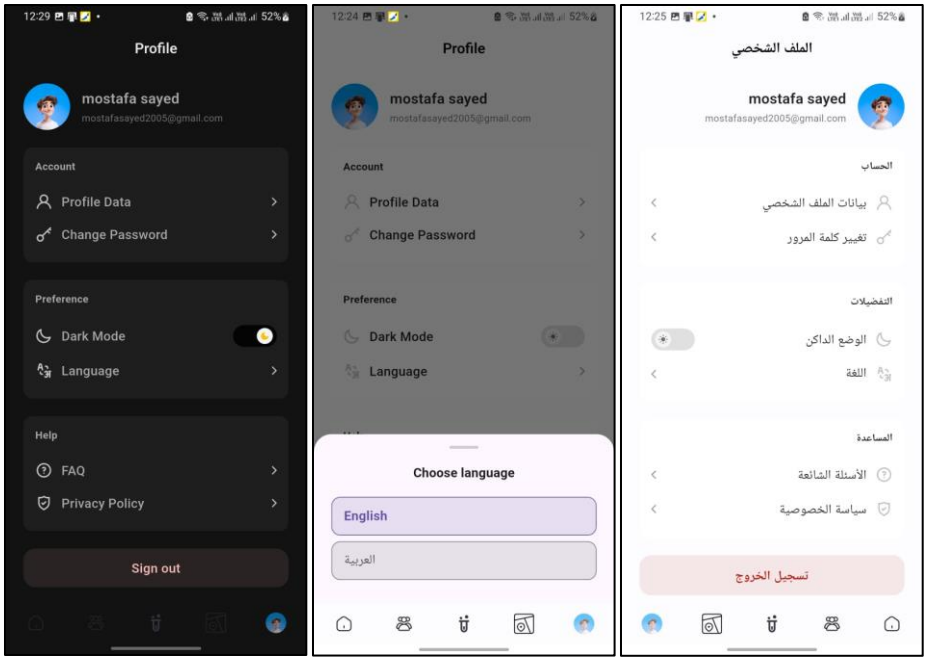


Figure 4.25 : Preference

The **Preference** section is designed for quick personalization. It provides users with direct control over the application's appearance and behavior, such as toggling Dark Mode or switching between languages. These adjustments are reflected instantly, contributing to a comfortable and inclusive user experience for both LTR and RTL language users.

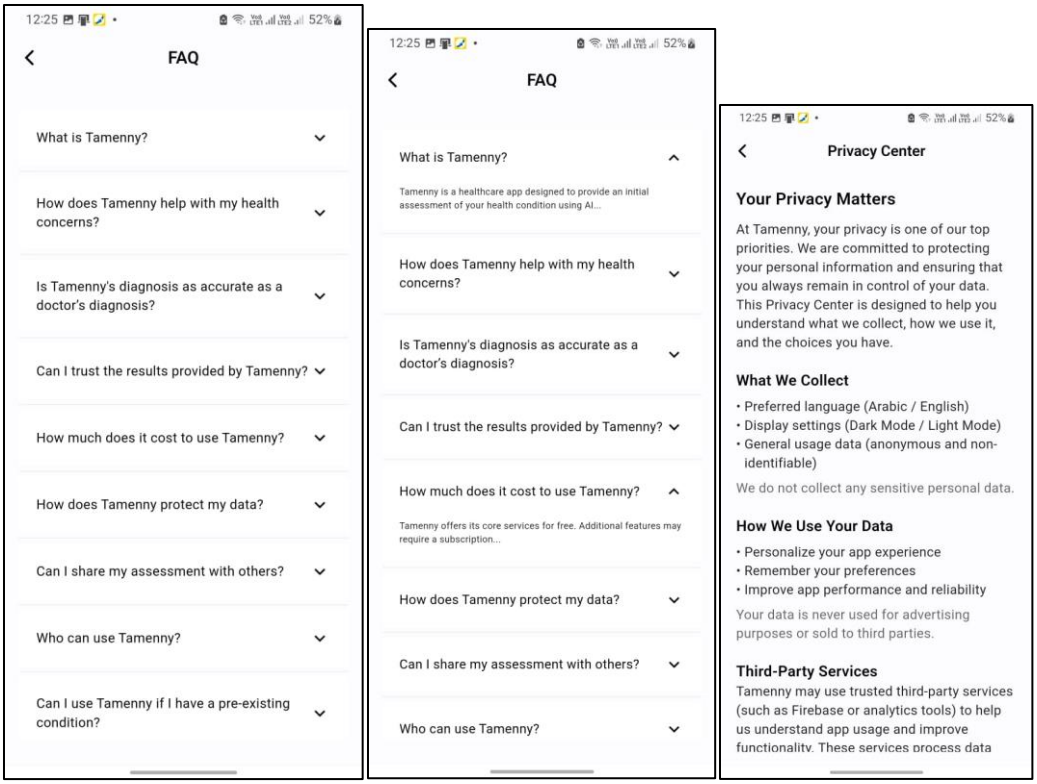


Figure 4.26 : FAQ

The **Help** section connects users with essential support resources. Through this section, users can navigate to frequently asked questions for quick guidance, or view the app's privacy policy to better understand how their data is handled.

Figure 4.27 : Sign out Screen

A clearly marked **“Sign Out”** button is located at the bottom of the screen in a bold pink box, encouraging secure session termination. Additionally, the Profile tab in the bottom navigation bar remains highlighted to indicate the user’s current location. While the main Profile screen serves as a launchpad, all detailed interactions and edits take place within their respective sub-screens maintaining clarity and minimizing interface clutter.

By centralizing access to personal data, app settings, and support materials, the User Profile Screen empowers users with full control over their experience. This approach promotes user trust, encourages engagement, and ensures that the AI-driven services are tailored accurately to each individual’s context and needs.

4.2.3 Medical Data Collection and Analysis

Scan Upload Screen:

The Scan Upload Screen is designed to enable users to seamlessly upload medical images or documents, such as X-rays, MRIs, or lab reports, for AI-driven analysis. This feature allows users to either select files directly from their device or capture images using their camera, providing flexibility in how they submit their medical data. Once uploaded, the screen displays previews of the images, accompanied by editing options to refine the submissions as needed. Additionally, a caption field is included, allowing users to add notes or context to support the AI's analysis. By offering a straightforward and intuitive upload process, this screen enhances the user experience, ensuring that supplementary data is easily provided to improve the accuracy of the AI-driven diagnosis.

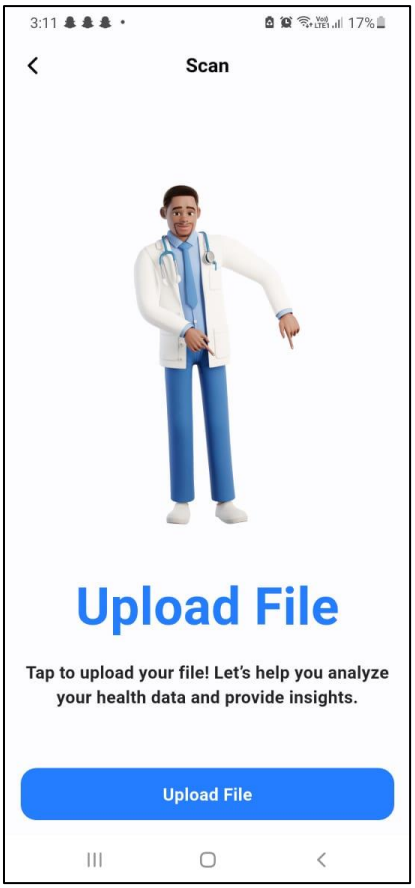


Figure 4.28 : Upload screen

Diagnosis Results Screen

The Diagnosis and Recommendations Screen is designed to deliver a clear and user-friendly report of the AI's preliminary analysis, presenting users with a detailed yet accessible overview of potential medical conditions, confidence scores, and actionable next steps, such as consulting a specialist or monitoring symptoms. Visual aids, including charts and icons, are incorporated to enhance comprehension and make complex medical information more digestible. The screen also provides options for users to save the report or share it, such as through PDF export, ensuring flexibility and convenience. By offering a transparent and intuitive presentation of critical health insights, this screen significantly contributes to the user experience, empowering individuals to make informed decisions about their health with greater confidence.

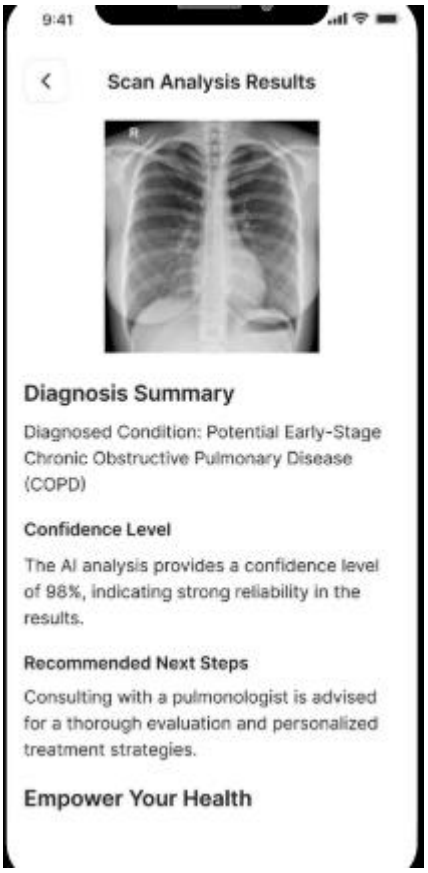


Figure 4.29 :Results Screen

4.2.4 Doctor Services and Medical Assistance

- **Map Screen**

Figure 4.30 : Map Screen

The Map Screen enables users to quickly locate nearby doctors and clinics through an interactive map interface. Utilizing geolocation, the map centers on the user's current position and displays surrounding healthcare providers with clear markers. Each marker reveals key details such as doctor name and specialty through concise tooltips. Users can freely zoom and pan the map to explore different areas, ensuring flexibility in browsing. By offering an intuitive, real-time view of available medical support, this screen plays a vital role in bridging the gap between AI-driven diagnosis and real-world medical consultation.

- **Doctor Detail Screen**

Figure 4.31 : Doctor Screen

The Doctor Details Screen offers users comprehensive information about a selected healthcare provider, supporting informed decision-making and seamless interaction. It presents key details such as the doctor's name, photo, specialty, qualifications, patient ratings, and contact information in a clean, structured layout. Users can review available appointment slots and proceed to book directly through the interface or initiate contact via a call option. By combining transparency with convenience, this screen enhances user confidence and simplifies the path to receiving professional medical care.

- **Search Screen**

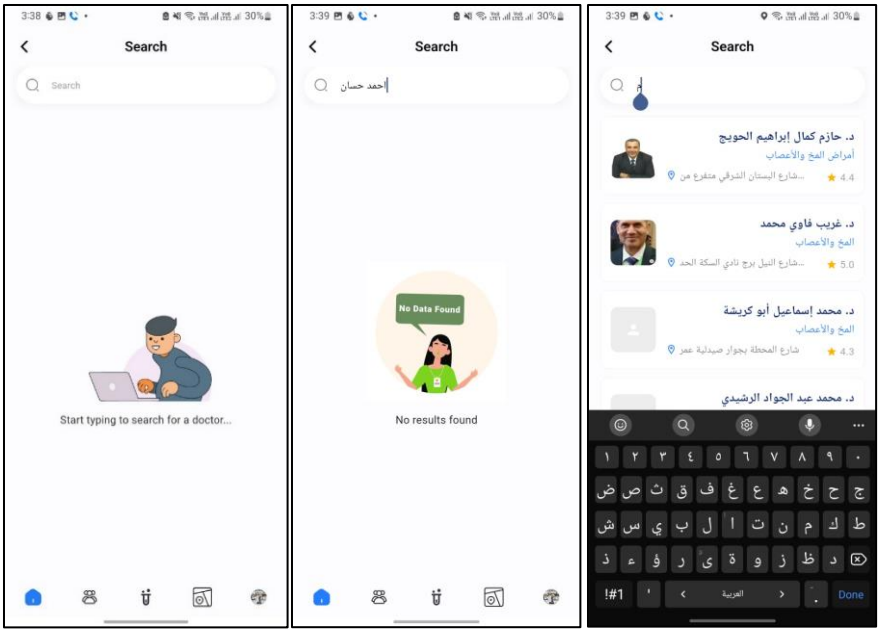


Figure 4.32 : Search

The Search Screen provides users with a powerful and intuitive tool to explore both medical professionals and health conditions based on tailored criteria. Through a single search bar, users can input keywords related to doctors such as names, specialties, or clinic locations as well as terms related to diagnoses, including symptoms or condition names. To refine results and improve accuracy, the screen includes adaptive filters: users can sort doctors by location, distance, availability, and rating, while diagnosis searches can be narrowed by condition type, severity, or related symptoms. Results are displayed in either a list or grid format, offering flexibility in how information is consumed. Each result is interactive and leads directly to the relevant Doctor or Diagnosis Details Screen. By enabling precise and goal-oriented navigation, this screen plays a critical role in improving accessibility, saving time, and helping users make informed health decisions.

4.2.5 Health Community Screen

The Health Community Screen acts as an interactive space within the ***Tamenny*** application, where users can share personal experiences, ask health-related questions, and provide support to others in a moderated, structured environment. This section of the app builds a sense of belonging, encouraging both emotional and informational engagement.

The screen is organized into a live feed of user posts, sorted by health-related categories such as chronic conditions, wellness routines, and general advice. Users can interact by liking, commenting, or contributing their own posts. A search bar supports topic discovery, and a "New Post" button is readily accessible for content contribution.

Below is a breakdown of each interface involved in the community experience:

Main Community Feed Screen:

This is the entry point of the health community section. It showcases a scrollable feed of posts submitted by users across different categories. Each post displays basic information such as the user's name, the content, number of likes, and comments.

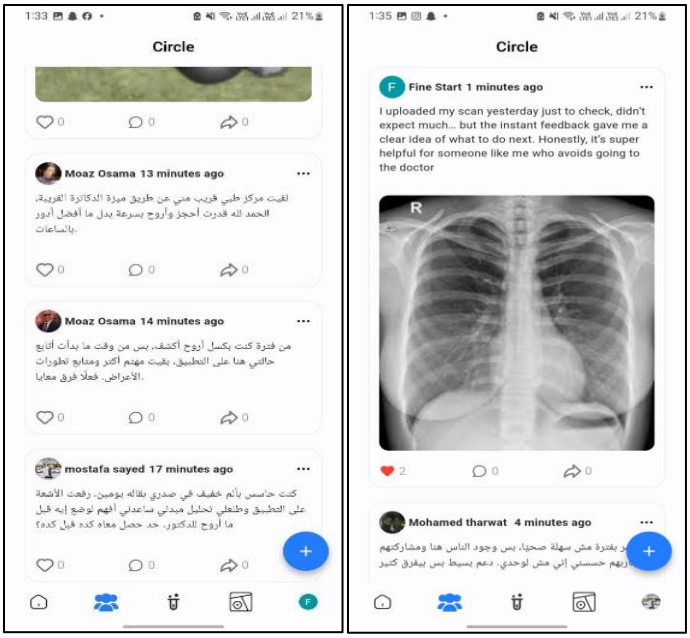


Figure 4.33 : Community

- **Add Post Screen**

This screen appears when the user taps on "New Post." It allows users to compose and submit a new community post. Fields typically include a text input area and an optional image upload.

Figure 4.34 : Add Post

- **Post Details & Comments Screen**

Once a post is selected from the main feed, the user is taken to a detailed view showing the full content of the post, user interactions, and all associated comments. This screen provides a clearer context for in-depth discussions.

Figure 4.35 : Post Details and Comments

- **Add Comment Screen**

This interface allows users to add their own responses to a post. It may appear as a modal or an embedded input area within the post detail screen, depending on the app's design.

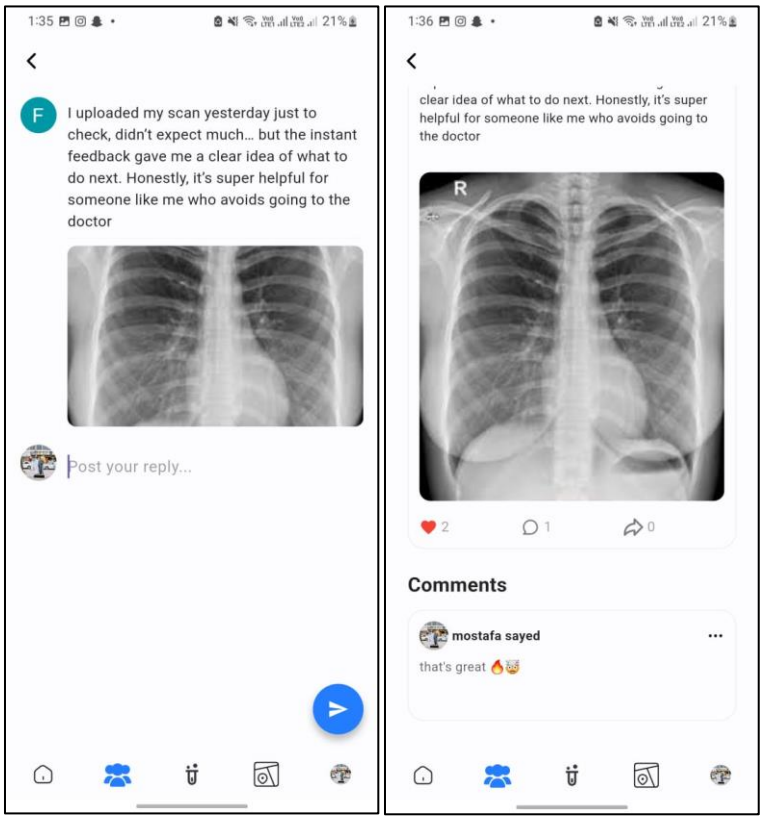


Figure 4.36 : Add Comment

By offering these layered interactions, the Health Community feature not only boosts user engagement but also supports mental and emotional well-being by fostering meaningful connections between users who share similar health journeys.

4.2.6 Tamenny Chatbot Screen

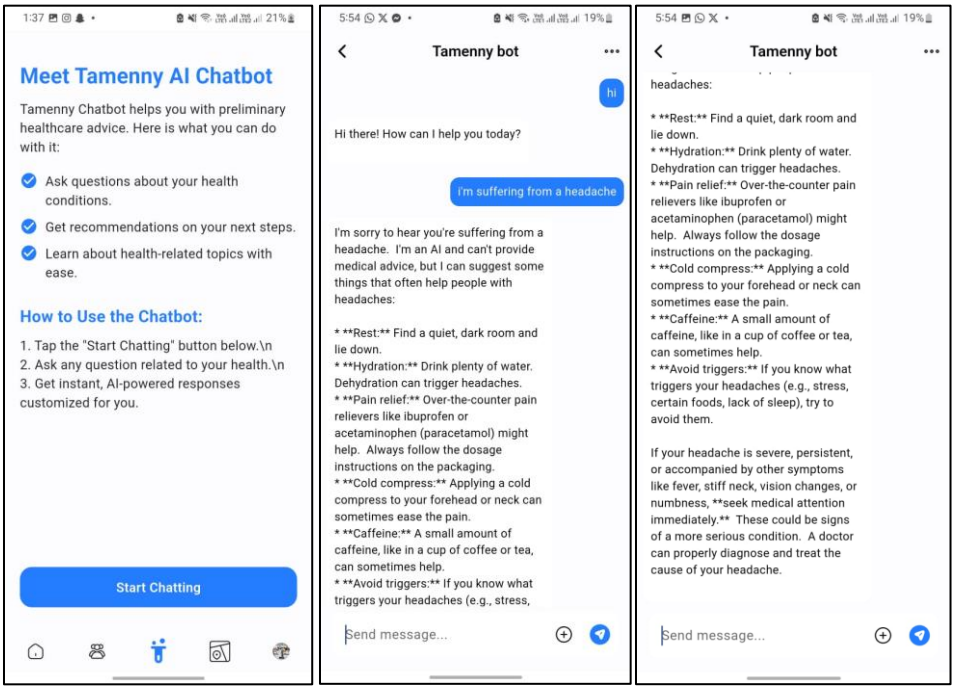


Figure 4.37 : Chatbot

The ***Tamenny*** Chatbot Screen introduces users to a smart, conversational assistant designed to provide quick and personalized health support. Through a friendly chat interface, users can ask health-related questions, describe symptoms, get preliminary guidance, or be directed to relevant app sections such as booking a doctor, viewing nearby clinics, or checking symptoms.

The screen features a clean layout with a chat bubble interface, a text input field at the bottom, and optional quick-reply buttons for faster interaction. The chatbot greets the user and responds in a conversational tone, offering accurate and helpful suggestions based on their input. The chatbot avatar adds a human-like touch to the interaction, making the experience feel more personal.

This screen serves as a central hub for users who prefer natural conversation over traditional navigation.

It simplifies complex workflows by offering direct paths through a single interaction point, especially useful for first-time or less tech-savvy users.The chatbot not only improves accessibility but also encourages proactive health management through timely and guided support

- **Notifications Screen**

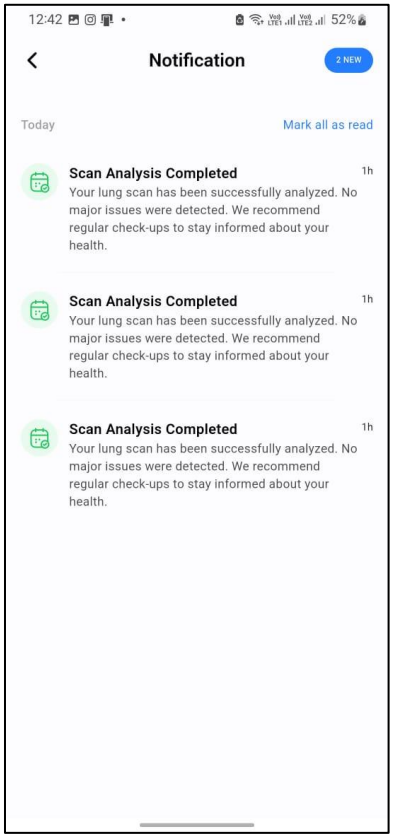


Figure 4.38 : Notifications

The Notifications Screen keeps users informed about important updates and interactions within the app. It presents a chronological list of alerts, including appointment reminders, doctor messages, community replies, and system announcements. Each notification includes a timestamp and, when relevant, action buttons that allow users to confirm appointments or respond directly. Users can also mark individual notifications as read to manage their inbox more effectively. By centralizing all communication in one accessible interface, this screen supports timely engagement and ensures that users never miss essential updates related to their health journey.

4.2.7 AI Models

Lung cancer:

In this project, we aimed to build a Convolutional Neural Network (CNN)-based model to detect lung cancer from CT scan images. We started by collecting and preprocessing the image data. The preprocessing involved resizing all CT scan images to a consistent dimension of 128x128x3 and normalizing their pixel values to improve model performance. We also encoded the labels to numerical values for classification. After that, we designed a custom CNN model that includes residual blocks to handle deeper networks, dropout layers to prevent overfitting, batch normalization layers to stabilize learning, and dense layers for final output. We trained the model using the Adam optimizer, and applied early stopping and model checkpointing to ensure optimal training without overfitting. Finally, we evaluated the model's performance using accuracy, loss curves, and a confusion matrix, and visualized the results using Matplotlib and Seaborn. All development and experimentation were conducted in Jupyter Notebook using Python and relevant libraries like TensorFlow, Keras, scikit-learn, NumPy, and PIL. [1],[2]

Model Parameter

Parameter	Value
Programming Language	Python
Image Size	128x128x3
Libraries Used	TensorFlow, Keras, NumPy, PIL, scikit-learn, Matplotlib, Seaborn
Development Environment	Jupyter Notebook
Model Architecture	Custom CNN with residual blocks
Normalization	Pixel values scaled to [0, 1]
Label Encoding	One-hot encoding
Optimizer	Adam
Regularization	Dropout layers
Stability Measures	Batch Normalization
Training Strategy	Early stopping, model checkpointing

Table 4.2 : Lung Parameters

Brain Cancer:

To implement the brain cancer classification system, a comprehensive pipeline was developed using Python with libraries such as TensorFlow/Keras, PIL, OpenCV, NumPy, Scikit-learn, Matplotlib, and Seaborn, executed within a Jupyter Notebook environment. The process commenced with the image preprocessing module, where MRI images were resized to 128x128 pixels, converted to NumPy arrays, normalized to mitigate noise, and their labels encoded into one-hot vectors using LabelEncoder and to_categorical for multiclass classification across four categories: Glioma, Meningioma, Pituitary, and No tumor. Subsequently, a convolutional neural network (CNN) classification model was designed with an input shape of (128, 128, 3), incorporating layers such as Conv2D, BatchNormalization, ReLU, MaxPooling2D, Dropout, and Dense, augmented by a residual block to enhance feature learning and a Softmax layer for final classification.

The training module compiled the model using the Adam optimizer with a learning rate of 0.0001, categorical crossentropy loss, a batch size of 16, and up to 80 epochs with a 25% validation split, utilizing EarlyStopping and ModelCheckpoint callbacks to save the best model based on validation accuracy. Finally, the evaluation module generated training and validation accuracy and loss curves, produced a confusion matrix and classification report using Scikit-learn, and saved the confusion matrix visualization as an image, ensuring a robust workflow for data processing, model training, and performance assessment. [3]

Model Parameters

Parameter	Value
Image Size	128×128 pixels
Input Shape	(128, 128, 3)
Normalization	Applied (to reduce noise)
Label Encoding	One-hot vectors
Optimizer	Adam (learning rate 0.0001)
Loss Function	Categorical Crossentropy
Batch Size	16
Epochs	Up to 80 (with EarlyStopping)
Validation Split	25%
Callbacks	EarlyStopping, ModelCheckpoint
Layers	Conv2D, BatchNormalization, ReLU, MaxPooling2D, Dropout, Dense
Final Layer	Softmax
Classes	Glioma, Meningioma, Pituitary, No tumor

Table 4.3 : Brain Parameters

Heart segmentation:

To implement the heart segmentation U-Net model, a systematic pipeline was followed using Python with libraries such as TensorFlow/Keras, OpenCV, NumPy, and Albumentations, leveraging CUDA for GPU acceleration. The process started with data preprocessing and augmentation, where a custom data generator class was developed to load grayscale medical images and their corresponding masks. Images were resized to 256×256 pixels, normalized to a [0, 1] range, and augmented with transformations like flipping, rotation, and noise using Albumentations to improve model generalization. Next, the U-Net architecture, a convolutional neural network (CNN), was designed with an encoder for feature extraction, a decoder for reconstructing segmentation maps, and skip connections to retain spatial details. The training pipeline was then established, compiling the model with binary cross-entropy loss and the Adam optimizer, training it using mini-batches from the data generator, and evaluating performance on a separate validation set. Finally, the trained model was saved in HDF5 format (.h5) for future inference or fine-tuning, ensuring an efficient and optimized workflow. [4],[5]

Model Parameters

Parameter	Value
Image Size	256×256 pixels
Normalization Range	[0, 1]
Loss Function	Binary Cross-Entropy
Optimizer	Adam
Batch Size	Configurable (e.g., 8, 16)
Augmentations	Flipping, Rotation, Noise
Model Format	HDF5 (.h5)
Hardware Acceleration	CUDA (GPU)

Table 4.4 : Heart Parameters

Knee:

To implement the knee classification system, a comprehensive pipeline was developed using Python and PyTorch, leveraging tools like Torchvision, PIL, Pandas, NumPy, TQDM, and Scikit-learn. The process began with dataset preparation, where the load_dataset_as_dataframe function scanned the folder structure to create a DataFrame mapping image paths to labels, and a custom KneeDataset class was implemented to load images and apply transformations compatible with PyTorch's DataLoader. Data augmentation and normalization were applied using transforms.Compose, resizing images to 224x224 pixels, applying random horizontal flips, rotations, and color jitter, and normalizing pixel values to align with ResNet-50's pretrained requirements. DataLoaders were created for training and testing, enabling mini-batch processing, shuffling, and parallel loading. The model architecture utilized a pretrained ResNet-50 from ImageNet, with its final fully connected layer replaced to match the number of knee dataset classes. The training loop executed over multiple epochs, setting the model to training mode, computing predictions, calculating loss with CrossEntropyLoss, and updating weights using the Adam optimizer, while tracking loss and accuracy with TQDM progress bars; the best model was saved as a .pth file based on the highest training accuracy. The evaluation module loaded the best model, set it to evaluation mode, disabled gradient computation, and computed test accuracy using Scikit-learn's accuracy_score, ensuring robust performance on unseen data. [6]

Model Parameters

Parameter	Value
Image Size	224×224 pixels
Input Normalization	ImageNet mean and std
Data Augmentations	Resize, Horizontal Flip, Rotation, Color Jitter
Model Architecture	ResNet-50 (pretrained on ImageNet)
Final Layer	Fully connected (custom for dataset classes)
Loss Function	CrossEntropyLoss
Optimizer	Adam
Batch Size	Configurable (e.g., 32)
Epochs	Configurable
Device	CUDA (if available) or CPU
Model Save Format	.pth
Evaluation Metric	Accuracy (via accuracy_score)

Table 4.5 : Knee Parameters

Chatbot “Flan T-5”:

To implement and utilize the FLAN-T5 base model for chatbot or general NLP applications, a structured pipeline was followed, leveraging its instruction-tuned encoder-decoder Transformer architecture. The process began with model setup, where the FLAN-T5 base model, comprising approximately 250 million parameters, was loaded using a deep learning framework like Hugging Face’s Transformers library in Python, ensuring compatibility with the required text-to-text input/output format. Data preparation involved formatting inputs as instruction-based prompts, such as “Translate this sentence: ...” or “Answer this question: ...”, to align with FLAN-T5’s training on diverse tasks like translation, summarization, question answering, and classification. The model configuration step included setting up the encoder and decoder, each with 12 layers, a hidden size of 768, a feed-forward network dimension of 2048, 8 attention heads, and relative positional embeddings, alongside preparing the tokenizer with a vocabulary size of 32,128.

For inference or fine-tuning, the model was either used directly for tasks by feeding instruction prompts or further fine-tuned on custom datasets using a text-to-text framework, optimizing with a suitable loss function like cross-entropy and an optimizer such as Adam. Finally, evaluation and deployment involved testing the model's performance on diverse tasks to ensure generalization to unseen instructions and integrating it into a chatbot system to handle varied user inputs effectively, leveraging its instruction-following capability.[7]

Model Parameters

Parameter	Value
Model Name	Flan-T5-base
Architecture	Encoder-Decoder Transformer
Total Parameters	~250 Million
Encoder Layers	12
Decoder Layers	12
Hidden Size	768
FFN Dimension	2048
Attention Heads	8
Vocabulary Size	32128
Positional Encoding	Relative positional embedding
Pretraining Objective	Span corruption (denoising autoencoder)
Training Style	Instruction tuning on multi-task mixture

Table 4.6 : ChatBot Parameters

Recommendation System "XLM-RoBERTa & Multiregressor NN":

XLM-RoBERTa:

To fine-tune and deploy the XLM-RoBERTa base model for multiclass medical text classification, a structured pipeline was implemented using the Hugging Face Transformers library. The process began with data preparation, where a medical posts dataset was loaded and labeled using LabelEncoder, then stratified into training, validation, and test splits. The model setup involved loading the xlm-roberta-base architecture a multilingual encoder-only Transformer model with approximately 125 million parameters, 12 self-attention layers, 768 hidden dimensions, and a feed-forward layer size of 3072. The tokenizer was initialized with a vocabulary size of 250,002 and a maximum sequence length of 128 tokens.

Model configuration was customized to support the classification task by adjusting the dropout rate to 0.3 and specifying the number of output labels according to the dataset. A partial fine-tuning strategy was employed by freezing all layers except the last two Transformer layers and the classification head, enabling efficient training while preserving pre-trained knowledge. The training utilized an AdamW optimizer, a cosine learning rate scheduler with a warmup ratio of 0.1, and gradient clipping (max norm = 1.0). Training was conducted over 15 epochs with early stopping (patience = 2), using a batch size of 8 for training and 16 for evaluation. Metrics such as accuracy and weighted F1-score were monitored to identify the best checkpoint.

Finally, the model and tokenizer were saved for later deployment, and performance evaluation was conducted on the test set to assess generalization across unseen samples in a multilingual medical domain. [8]

Parameter	Value
Model Name	xlm-roberta-base
Architecture	Transformer-based (Encoder only)
Total Parameters	~125 Million
Hidden Size	768
FFN Dimension	3072
Attention Heads	12
Number of Layers	12
Vocabulary Size	250002
Positional Encoding	Sinusoidal positional embeddings
Max Sequence Length	128
Dropout	0.3
Batch Size (Train / Eval)	8 / 16
Optimizer	AdamW
Learning Rate	5e-5
Epochs	15
Scheduler	Cosine Learning Rate Scheduler
Loss Function	Cross Entropy
Evaluation Metric	Weighted F1-score
Training Objective	Sequence classification
Fine-Tuning Strategy	Freeze all but last 2 encoder layers + classifier
Early Stopping	Enabled (patience = 2 epochs)

Table 4.7 XLM-RoBERTa Parameters

Multivariate NN regressor :

To develop a lightweight neural network model capable of predicting recommendation scores for five medical categories Heart, Brain, Lung, Knee, and Neutral a regression-based pipeline was constructed using TensorFlow and Keras. The process began with loading a structured dataset containing both ratio-based and availability-based features, followed by selecting ten relevant input features and five numerical target scores. Data preprocessing included a train-validation split with a 75–25 ratio and feature normalization using `StandardScaler` to standardize the input distributions. The scaler's mean and standard deviation values were exported to a JSON file (`scaler_values_for_flutter.json`) to ensure consistent preprocessing in external environments like mobile apps.

The model architecture consisted of a simple feedforward neural network with two hidden layers of 64 ReLU-activated neurons each and a final dense layer outputting five continuous values. The model was compiled with the Adam optimizer and trained using Mean Absolute Error (MAE) as the loss function over 300 epochs. Evaluation metrics included both overall and per-disease MAE and R^2 scores, allowing for granular performance analysis. After training, the model was saved in both `.h5` and `.tflite` formats, enabling deployment on both server-side and mobile platforms, thereby supporting real-time personalized recommendations across multiple medical domains. [9]

Parameter	Value
Model Type	Feedforward Neural Network (Fully Connected Layers)
Input Features	10
Output Targets	5 (score_Heart, score_Brain, score_Lung, score_Knee, score_Neutral)
Hidden Layers	2 Dense Layers
Hidden Layer Size	64 neurons each
Activation Function	ReLU
Output Layer	Dense Layer with 5 neurons (no activation)
Loss Function	Mean Absolute Error (MAE)
Optimizer	Adam
Evaluation Metrics	MAE, R^2 (R-squared)
Epochs	300
Batch Size	32
Scaler Used	StandardScaler (mean and std saved to JSON)
Model Exported As	.h5 (Keras) and .tflite (TensorFlow Lite)
Preprocessing Step	Normalization using StandardScaler

Table 4.8 Multivariate NN regressor

4.3 Challenges Faced and How They Were Resolved:

Challenges: Handling diverse instruction-based inputs with consistent prompt engineering, managing the large computational cost of a 250-million-parameter FLAN-T5 model in resource-constrained environments, preventing overfitting and catastrophic forgetting during fine-tuning on custom datasets, ensuring generalization to unseen tasks despite variable instruction formats, resolving tokenization and vocabulary alignment issues for domain-specific terms, coping with wide variations in CT-scan sizes and quality, mitigating overfitting caused by limited lung-cancer data, training deep CNNs without vanishing gradients, balancing highly skewed class distributions, cleaning noisy MRI images that reduced brain-tumor accuracy, curbing overfitting and bias across imbalanced tumor types, capturing fine spatial features in MRI scans, matching image–mask pairs for U-Net segmentation, avoiding GPU-memory crashes with large datasets, applying synchronized augmentations to image-mask pairs, skipping corrupt or missing files during loading, organizing folder-based datasets and assigning numeric labels for ResNet-50, preprocessing images to the fixed 224 × 224 input size and normalization the model expects, controlling overfitting with limited orthopedic images, guaranteeing CPU/GPU compatibility during deployment, tracking accuracy improvements and saving the best checkpoints, providing visual feedback during long training cycles, validating generalization on unseen test data, maintaining consistent application state across many Flutter screens, implementing full multilingual (LTR + RTL) support, building a robust Dark-Mode theme with correct contrast and dynamic widget styling, easing user anxiety during AI diagnosis wait times with engaging feedback, and integrating real-time maps and doctor listings while preserving performance and location privacy.

Solutions: To address the challenge of handling diverse instruction-based inputs in the FLAN-T5 model, standardized instruction templates were designed to maintain consistency with its training paradigm. The high computational cost associated with running a 250M parameter model was mitigated through GPU acceleration using CUDA and by optimizing batch sizes during training and inference. Overfitting during fine-tuning on custom datasets was addressed with regularization techniques, learning rate scheduling, and limiting the number of fine-tuning epochs to preserve the model's pretrained capabilities.

To enhance generalization on unseen tasks, validation on diverse instruction formats and few-shot examples within prompts were used. Tokenization and vocabulary alignment issues were solved by utilizing the pretrained tokenizer and augmenting the vocabulary for domain-specific terms. In the lung cancer detection system, image size and quality inconsistencies were resolved by resizing and normalizing all CT scans to a standard shape (128x128x3). To combat dataset limitations and overfitting, dropout layers, early stopping, and model checkpointing were implemented. The deep CNN training challenges were addressed by using residual connections to stabilize learning and enhance feature extraction. Class imbalance issues were handled using stratified sampling for balanced train-test splitting. For brain cancer classification, noise in MRI images was reduced through preprocessing and normalization techniques. Overfitting was prevented by using Dropout and EarlyStopping, while class imbalance was mitigated by applying data augmentation to underrepresented tumor classes. Spatial feature extraction challenges were tackled by adding residual blocks and multiple convolutional layers. In the U-Net segmentation model, mismatches between images and masks were resolved through filename sorting and count verification. GPU memory limitations were managed by enabling dynamic memory allocation in TensorFlow. Synchronized augmentations were handled using the Albumentations library to apply consistent transformations.

Training instability was addressed via data augmentation and validation monitoring, and corrupt files were bypassed using defensive programming techniques. In the knee osteoarthritis classification using ResNet-50, folder-based dataset labeling was automated using a dataset loading function, while image preprocessing challenges were solved using resizing and normalization techniques compatible with ResNet-50. Overfitting from limited data was reduced using augmentations like flips and color jitter. CPU/GPU compatibility was ensured using torch.device, and model tracking was enabled through accuracy monitoring and conditional saving. Progress feedback was added using TQDM for real-time updates, and generalization was validated using test data accuracy scoring. In the Tamenny mobile application, inconsistent app state across modules was resolved by adopting the Bloc (Cubit) state management pattern. Multilingual support (LTR and RTL) was implemented using flutter_localizations and manual layout adjustments. Dark Mode design challenges were handled by building a dynamic theming system using ThemeData that adjusted widget styles based on the selected mode.

To improve user experience during AI diagnosis, Lottie animations and rotating health tips were added to reduce anxiety during loading. Real-time doctor map integration was optimized by implementing GPS permission handling, map caching, marker clustering, and advanced filtering to ensure performance and privacy across all devices.

Chapter 5

Testing & Evaluation

5.1 Testing Strategies

To ensure the reliability and correctness of the ***Tamenny*** system, various levels of testing were conducted. Unit testing was used extensively during the development of machine learning models and the mobile application. For instance, individual Python functions responsible for preprocessing, model inference, and data transformation were tested in isolation. Likewise, components of the Flutter app such as widgets, form validation, and routing logic were covered using widget tests.

Integration testing ensured the smooth interaction between system modules, including image upload, diagnosis processing via AI models, and displaying output results. This type of testing was critical in validating the end-to-end behavior of combined components, particularly in cloud-based scenarios where models are hosted remotely. User testing involved gathering feedback from real users through trial deployments. Testers interacted with the app's chatbot, diagnosis interface, and map-based doctor finder. Their feedback was instrumental in refining UI behavior, improving text clarity, and ensuring a smooth user experience for both Arabic and English speakers.

5.2 Performance Metrics

The performance of ***Tamenny's*** diagnostic models was assessed using standard metrics including accuracy, precision, recall, F1-score, and confusion matrices. For image classification tasks, such as brain and lung cancer detection, accuracy curves and confusion matrices were plotted to visualize training stability and classification performance.

Below are selected visual outputs from the experiments:

Figure 5.39 : recommendation System

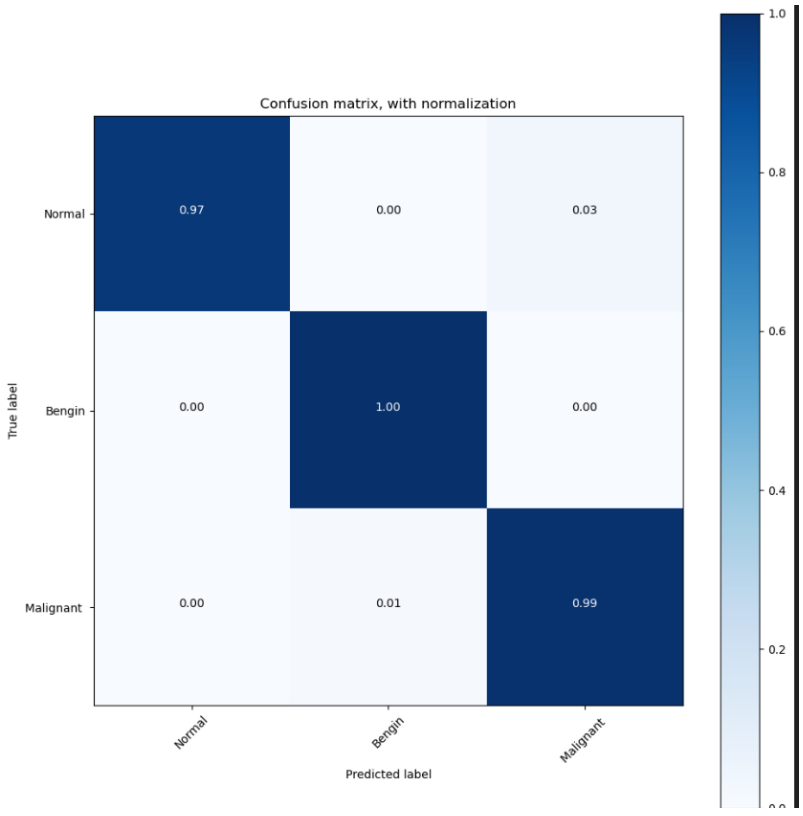


Figure 5.40 : Plot for lung cancer

	precision	recall	f1-score	support
0	1.00	0.97	0.98	29
1	0.99	1.00	1.00	145
2	0.99	0.99	0.99	101
accuracy			0.99	275
macro avg	0.99	0.99	0.99	275
weighted avg	0.99	0.99	0.99	275

```
[[ 28   0   1]
 [  0 145   0]
 [  0   1 100]]
```

Figure 5.41 : Accuracy for Lung model

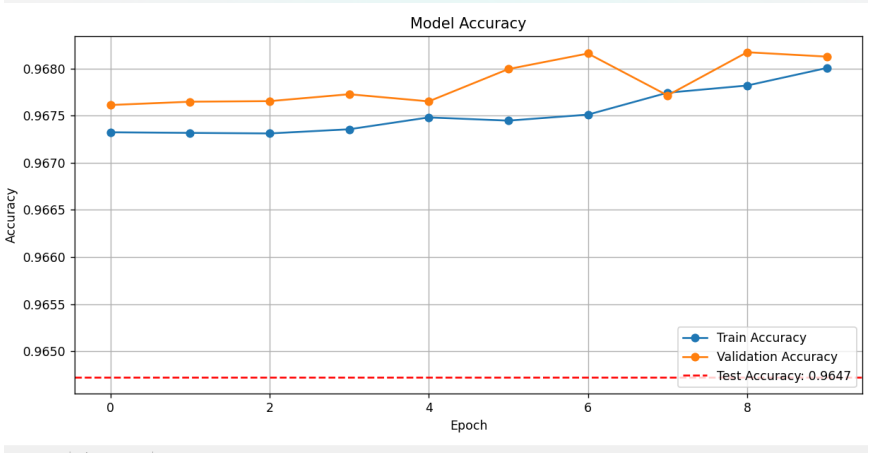


Figure 5.42 : Model Accuracy

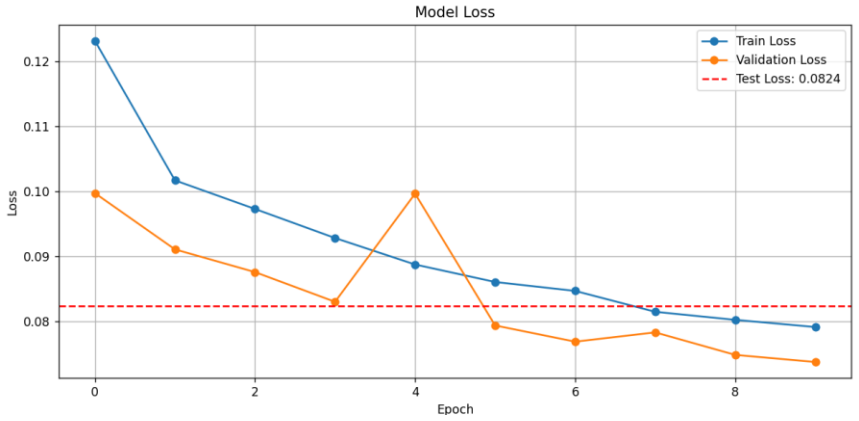


Figure 5.43 : Model Loss

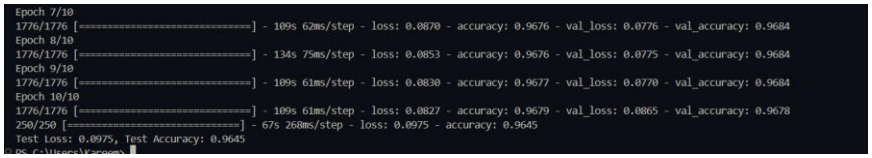


Figure 5.44 : Heart

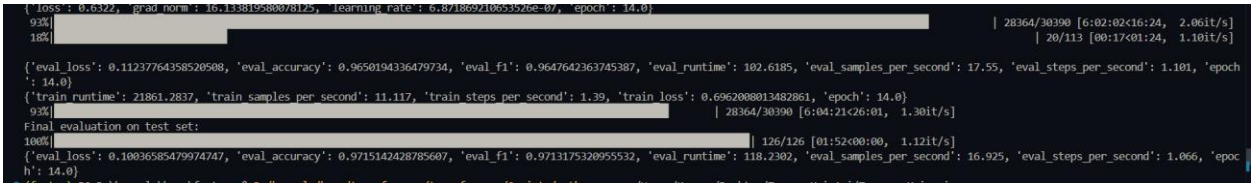


Figure 5.45 : Recommendation System Classifier

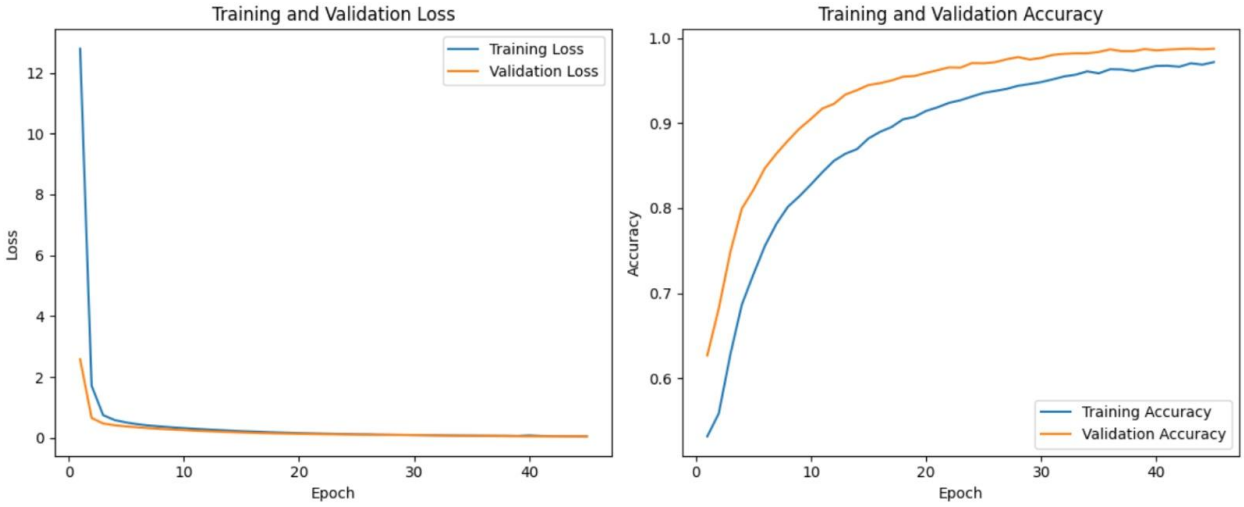


Figure 5.46 : Chatbot Plots

The above figures show typical training accuracy and loss curves, along with confusion matrices for the brain cancer CNN, lung cancer model, heart segmentation using U-Net, and system-level response tracking for the chatbot. Models consistently reached over 90% accuracy, with confusion matrices confirming strong classification behavior across target classes.

5.3 Comparison with Existing Solutions

Tamenny's integrated AI models outperform many baseline and traditional healthcare solutions in speed, accessibility, and preliminary diagnostic quality. The CNN-based cancer classifiers achieved higher classification accuracy than simpler image-processing approaches, and their use of dropout, batch normalization, and residual connections significantly reduced overfitting.

For segmentation, the U-Net architecture showed stronger boundary detection accuracy compared to traditional morphological or edge-based methods. The recommendation chatbot, built on the instruction-tuned FLAN-T5 model, was more adaptable to natural, multilingual queries than older rule-based or menu-driven systems. Additionally, ***Tamenny's*** combination of image diagnosis, symptom analysis, and doctor navigation in a single mobile app sets it apart from many fragmented health applications.

Chapter 6

Results & Discussion

6.1 Introduction

In this chapter, we present a comprehensive discussion of the outcomes achieved through the development and deployment of ***Tamenny***, our real-time, AI-powered healthcare diagnostic application. The primary aim is to evaluate not only the technical robustness of the implemented system but also its broader implications in real-world healthcare settings. As healthcare accessibility remains a pressing global concern particularly in underserved and rural areas this project seeks to contribute meaningfully to early detection, health awareness, and user empowerment. By integrating advanced deep learning models with a user-centric mobile interface, *Tamenny* strives to bridge the widening gap between timely medical advice and limited access to specialized healthcare services.

This discussion focuses on four main pillars that define the value of the solution: First, we analyze the app's ability to deliver accurate, AI-driven preliminary diagnoses for critical conditions such as brain tumors, lung cancer, heart disease, and knee osteoarthritis using medical imaging. Second, we examine the effectiveness of the interactive community platform, which was designed to enhance emotional support and encourage knowledge exchange among users. Third, we evaluate the impact of the natural language chatbot, which enables real-time symptom interpretation, guidance, and navigation within the app, thereby simulating an always-available virtual health assistant. Finally, we reflect on the application's contribution to improving healthcare accessibility by offering fast, affordable, and mobile-based diagnostic support, particularly for users who face geographical or economic barriers to traditional care.

Beyond measuring performance metrics, this chapter also serves to explore the human dimension of the solution how users interact with, trust, and benefit from the system. It raises critical questions about digital health readiness, ethical considerations, and the role of AI in shaping future healthcare delivery models. Ultimately, this discussion helps determine whether ***Tamenny*** fulfills its intended purpose and identifies areas that require further enhancement to transition from a prototype into a clinically recognized, deployable healthcare companion.

6.2 Summary of findings.

Through extensive testing, performance evaluation, and early user feedback, the **Tamenny** application has proven to be a promising AI-powered healthcare solution. One of its most significant achievements lies in the high accuracy of its diagnostic models. The **lung cancer detection model**, built using a Convolutional Neural Network (CNN), achieved an accuracy of approximately **99%**, demonstrating strong performance in distinguishing between healthy and affected lung scans. Similarly, the **brain tumor classification model**, which handles four classes (glioma, meningioma, pituitary tumor, and no tumor), reached an impressive **98% accuracy**, showcasing its reliability in complex multi-class medical scenarios. The **knee osteoarthritis model**, based on ResNet-50 and enhanced with segmentation techniques, surpassed **96% accuracy**, making it particularly effective in identifying joint degeneration. For **heart disease detection**, the system achieved **nearly 96% precision**, establishing it as a powerful tool for early cardiovascular risk detection.

Beyond diagnostics, ***Tamenny*** also integrates an intelligent **AI-driven chatbot** powered by the Flan-T5 architecture. This chatbot enables users to describe their symptoms in natural language and receive intelligent feedback and medical recommendations. Supporting both **Arabic and English**, it significantly enhances usability for a culturally and linguistically diverse user base. As a virtual assistant, it empowers users with quick answers, condition insights, and navigation help within the app, especially for those without immediate access to healthcare professionals.

A key feature that strengthens user engagement is the **medical community system**, where users can share their health experiences, ask medical questions, and interact with others facing similar issues. This feature transforms ***Tamenny*** from a diagnostic tool into a **social support platform** that fosters emotional connection, health education, and empowerment through shared knowledge.

Early interaction data suggests that this community-driven approach increases user retention and encourages proactive health behavior.

To complement this, ***Tamenny*** **introduces a smart recommendation system** that further personalizes the user experience. Posts created by users are automatically classified into five health-related categories (Heart, Brain, Lung, Knee, or Neutral) using a pre-trained **XLM-R model**. Based on each user's behavior including scan uploads, post interactions (likes/comments), and authored posts the system builds an **interest profile** that reflects their health concerns. These interest ratios, combined with the availability of content in each category, are passed into a **multi-output regression model**, which determines the optimal number of posts to display per category. This dynamic content delivery ensures that users see relevant and engaging information aligned with their health focus and available content.

The recommendation system also handles edge cases efficiently. In cold-start scenarios, where no user history exists, the app displays **20 posts per category** by default. When content is exhausted in one category, it smartly redistributes from others based on user interest proportions. Additionally, the same profiling is used to **recommend medical news** and **reorder category icons** in the user interface, prioritizing the most relevant health areas.

From a technical perspective, ***Tamenny*** offers **real-time processing capabilities**. Users can upload medical scans and receive instant predictions through integration with cloud-based AI services. This ensures minimal delays, high availability, and fast response crucial features for urgent health situations.

Finally, the app's **user interface**, developed with **Flutter**, has received praise for its consistency and accessibility across platforms. Whether on Android or iOS, users enjoy a seamless and intuitive experience. Usability testing confirmed that individuals of different ages and technical skill levels could navigate the app with ease, highlighting its **inclusive and user-friendly design**.

Taken together, these findings confirm that ***Tamenny*** **successfully delivers on its promise**: providing a reliable, responsive, and intelligent healthcare assistant that merges AI power with human-centered design, enabling diagnosis, dialogue, community engagement, and personalized recommendations in one integrated platform.

6.3 Interpretation of results

The outcomes of the ***Tamenny*** project clearly demonstrate that it has successfully met all its core objectives both technically and in terms of practical healthcare impact.

The **first and most critical objective**, which focused on developing accurate AI-powered diagnostic models, was achieved through the design and training of deep learning algorithms specialized in classifying medical images related to **brain tumors**, **lung cancer**, **heart disease**, and **knee osteoarthritis**. Each model was tailored to a specific condition and consistently delivered **high accuracy and reliability** during testing. Their performance indicates a strong potential to serve as **pre-screening tools**, facilitating **early detection** and prompting users to seek professional medical care at the appropriate time.

The **second objective** was the implementation of a **symptom-based chatbot**, which was realized using the **Flan-T5 architecture**. This AI assistant is capable of processing **natural language input in both English and Arabic**, offering real-time interpretations of symptoms and delivering intelligent, user-friendly guidance. It assists users by suggesting possible conditions, clarifying diagnostic outcomes, and navigating the app particularly aiding individuals with limited medical or digital literacy.

The **third major goal** was to establish a platform for **community interaction**, which has also been fully implemented. The application now enables users to share medical experiences, comment on posts, and engage in peer-to-peer communication. This feature addresses the **emotional and social dimensions of healthcare**, providing a space for shared understanding, support, and empowerment among users facing similar health challenges.

The **fourth objective** focused on delivering a **mobile-friendly and platform-independent** application. This was achieved using **Flutter**, ensuring seamless functionality across **Android and iOS** devices. The user interface was carefully designed for **simplicity and accessibility**, and was positively received during usability testing by users from diverse age groups and backgrounds.

The **fifth and final objective** involved a structured **evaluation of the system's performance**, covering diagnostic accuracy, interface usability, and processing efficiency. Testing confirmed that ***Tamenny*** **performs reliably under various conditions** and meets the real-time demands of medical assistance scenarios.

Additionally, the introduction of an intelligent **recommendation system** enhanced the app's ability to deliver personalized health content. By leveraging user behavior such as post interactions, scan uploads, and authored content the system builds an **interest profile** for each user. This is used to tailor the feed dynamically using a **multi-output regression model**, ensuring that users receive the most relevant posts and medical news based on their interests. Posts are also automatically classified into five categories (Heart, Brain, Lung, Knee, or Neutral) using a **pre-trained XLM-R model**, ensuring contextual accuracy and relevance. The system also manages edge cases efficiently, supporting cold-start scenarios and intelligent fallback when content is limited.

In summary, these accomplishments validate that ***Tamenny*** **is not merely a conceptual prototype**, but a **fully functional, scalable, and impactful healthcare application**. It exemplifies the power of integrating **AI technologies** with **human-centered design** to deliver innovative solutions that improve healthcare **accessibility**, **early detection**, and **user engagement** on a practical level.

6.4 Limitations of the proposed solution.

While the ***Tamenny*** application has demonstrated considerable success in meeting its objectives and delivering valuable healthcare functionalities, it is important to acknowledge several limitations that define the boundaries of its current scope and highlight areas requiring further development. One of the foremost challenges lies in the nature of the datasets used to train the diagnostic models. Although these datasets were publicly available and of high quality, they may not fully capture the complex diversity of real-world medical data, including rare diseases, edge cases, and variations across different ethnic, age, or socio-economic groups.

This limitation can affect the generalizability and fairness of the AI predictions when applied in a broader population.

Additionally, the system has not yet received formal clinical certification from recognized health authorities such as the FDA or national health ministries.

As a result, its current use must be regarded as informational rather than diagnostic, meaning it cannot legally substitute or replicate the decision-making process of licensed medical professionals.

Without regulatory approval, the application cannot yet be integrated into formal healthcare systems or relied upon for critical clinical decisions.Another significant constraint is the app's reliance on stable internet connectivity.

Because the AI models operate through cloud-based inference engines, users must maintain an active internet connection to receive diagnostic results. This may limit the app's utility in remote or rural areas where network infrastructure is weak or inconsistent ironically, the very communities that could benefit most from accessible AI-driven healthcare. Furthermore, while the integrated AI chatbot provides a valuable and interactive layer of support, it is inherently limited by the scope and quality of the data on which it was trained. As a result, it may struggle to respond appropriately to rare symptoms, ambiguous language, or overlapping conditions, and users must be clearly informed that it is not a replacement for professional medical advice. Lastly, while ***Tamenny*** employs secure technologies such as encryption and authentication to safeguard user data, it has not yet achieved compliance with global privacy regulations like HIPAA (Health Insurance Portability and Accountability Act) or GDPR (General Data Protection Regulation). This represents a crucial gap, particularly if the app is to be deployed in jurisdictions with strict healthcare data governance laws.

These limitations, while not undermining the app's core value, underscore the distinction between a functioning prototype and a clinically validated medical product. They point the way toward future enhancements, including more inclusive data collection, formal clinical trials, offline capabilities, expanded chatbot intelligence, and legal compliance measures. Addressing these issues will be essential for ***Tamenny's*** transition from a promising innovation into a trusted and scalable solution within the global healthcare landscape.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

Our app represents a significant step forward in the integration of artificial intelligence and healthcare, offering users a proactive way to manage their well-being. By delivering fast and accurate preliminary assessments for brain glioma, heart disease, lung cancer, and knee osteoarthritis, the app empowers individuals to take informed steps toward professional medical consultation. The AI-driven diagnostics, trained and validated against clinical standards, serve as a reliable first point of analysis helping to bridge the gap between symptom onset and specialist intervention.

Beyond technology, the app fosters a sense of community and shared experience. The integrated support space encourages users to connect, share, and support one another, which is invaluable for emotional well-being during challenging health journeys. This peer-to-peer interaction enriches the overall healthcare experience by providing a compassionate layer that complements clinical care.

Designed with inclusivity and ease of use at its core, the app ensures that healthcare insights are accessible to a diverse audience. Its intuitive interface, multilingual support, and strong data security framework create a trusted environment where users can confidently explore their health status. Moreover, the inclusion of an AI chatbot offers real-time guidance and educational resources, transforming the app into a 24/7 health companion.

Ultimately, our app is more than just a diagnostic tool it is a comprehensive platform that empowers users through knowledge, connection, and actionable insights. Whether managing an existing condition, seeking early detection, or looking for community support, users can rely on our app to enhance their healthcare journey. Together, we are shaping a smarter, more connected, and more compassionate future for personal health management.

7.2 Future Work

As we continue to develop the ***Tamenny*** healthcare platform, our future work will prioritize both expanding accessibility and enhancing user experience. One of the first steps is to publish the app on the Google Play Store, making it easily accessible to a broader audience of users worldwide. We are also committed to continuous updates that will add new disease detection models, improve AI accuracy through the integration of larger and more diverse datasets including rare diseases, edge cases and enhance the community experience with moderation tools and multilingual support. In addition, we will refine the app interface based on user feedback to ensure smoother navigation and interaction, helping ***Tamenny*** remain a trusted, user-friendly health companion.

At the same time, we recognize the importance of addressing key limitations to ensure clinical value and broader system integration. We plan to pursue formal clinical validation through trials and seek regulatory certifications such as FDA approval, enabling ***Tamenny's*** safe adoption within formal healthcare systems. To overcome current technical constraints, we aim to develop offline functionality using local AI models to serve remote areas with poor connectivity. We will also advance the AI chatbot's capabilities by expanding its knowledge base, allowing it to better handle rare symptoms, ambiguous language, and complex medical cases. Furthermore, we will implement robust data protection in full compliance with global privacy standards like HIPAA and GDPR, and we will work toward seamless integration with Electronic Health Records (EHR), positioning ***Tamenny*** as a scalable and clinically trusted tool in modern healthcare ecosystems.

References

[1] Nurul Najiha Jafery,Siti Noraini Sulaiman,Muhammad Khusairi Osman,Noor Khairiah A. Karim,Zainal Hisham Che Soh, 2024 , "Enhancing Lung Cancer Classification: Leveraging Existing Convolutional Neural Networks within a 1D Framework", "Institute of Electrical and Electronics Engineers(IEEE)".

[2] Lakshmanaprabu S.K., Mohanty S.N., Shankar K., Arunkumar N., Ramirez G. Optimal deep learning model for classification of lung cancer on CT images. Future Gener. Comput. Syst. 2019;92:374–382. [Google Scholar]

[3] Arabahmadi M., Farahbakhsh R., Rezazadeh J. Deep Learning for Smart healthcare—A Survey on Brain Tumor Detection from Medical Imaging. Sensors. 2022;

[4] Shah D., Patel S., Bharti S.K. Heart Disease Prediction using Machine Learning Techniques. SN Comput. Sci. 2020

[5] Otoom A.F., Abdallah E.E., Kilani Y., Kefaye A., Ashour M. Effective diagnosis and monitoring of heart disease. Int. J. Softw. Eng. Its April 2015

[6] Karim MdR, Jiao J, Döhmen T, Cochez M, Beyan O, Rebholz-Schuhmann D, Decker S. DeepKneeExplainer: Explainable Knee Osteoarthritis Diagnosis From Radiographs and Magnetic Resonance Imaging. IEEE Access 2021

[7] Bora A, Cuayáhuitl H. Systematic Analysis of Retrieval-Augmented Generation-Based LLMs for Medical Chatbot Applications. *Machine Learning and Knowledge Extraction*. 2024; Volume 6, Issue 4: Pages 923–944.

[8]K. K. Jayanth, G. B. Mohan, and R. P. Kumar, *Indian Language Analysis with XLM-RoBERTa: Enhancing Parts of Speech Tagging for Effective Natural Language Preprocessing*, Nov. 22, 2023.

[9] Cruz-Conesa, A., et al. (2022). Multivariate regression via artificial neural networks for compost quality prediction. Computers and Electronics in Agriculture, *194*, 106726. Zhang, Y., et al. (2019). Hierarchical deep learning for multivariate time series. Proceedings of the AAAI Conference, *33*, 7495-7502.

[10] Google Developers. Flutter SDK: Version 3.5.4. Available online: https://flutter.dev/ . Used as the main framework for building cross-platform mobile applications with a single codebase.

[11] Dart Team. Dart Programming Language. Available online: https://dart.dev/ . The programming language used in conjunction with Flutter for writing application logic and business components.

[12] Google Firebase. Firebase for Flutter. Available online: https://firebase.flutter.dev/ . A suite of cloud-based tools integrated in the app for authentication, real-time database access, and backend services.

[13] Supabase Inc. Supabase for Flutter. Available online: https://supabase.com/docs/guides/with-flutter . An open-source backend-as-a-service used as an alternative to Firebase for authentication and real-time PostgreSQL database functionalities. Integrated using the supabase_flutter package.

[14] A. Mahimkar, "IQ-OTH/NCCD - Lung Cancer Dataset," Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/adityamahimkar/iqothnccd-lung-cancer-dataset

[15] obulisainaren, "Multi Cancer Dataset", Kaggle,2022 [Online].

Available: https://www.kaggle.com/datasets/obulisainaren/multi-cancer

[16] Msoud Nickparvar. (2021). Brain Tumor MRI Dataset [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DSV/2645886

[17] Ahmed Hamada , "Br35H :: Brain Tumor Detection 2020" , Kaggle,2020[Online].

Available:https://www.kaggle.com/datasets/ahmedhamada0/brain-tumor-detection/data

[18] Pradeep2665 , "Brain MRI",Kaggle,2022,[Online].

Available : https://www.kaggle.com/datasets/pradeep2665/brain-mri

[19] Nikhil Tomar, "CT Heart Dataset, Kaggle 2021" [Online], Available : https://www.kaggle.com/datasets/nikhilroxtomar/ct-heart-segmentation

[20] M. Gobara, "Osteoporosis Database," Kaggle, 2022. [Online]. Available: https://www.kaggle.com/datasets/mohamedgobara/osteoporosis-database

[21] Niyar R Barman and 2 collaborators- Symptom2Disease - Kaggle -2023. [online]

https://www.kaggle.com/datasets/niyarrbarman/symptom2disease/data

[22] Nirali vaghan -Chatbot dataset - kaggle - 2023 - [Online]. available:

https://www.kaggle.com/datasets/niraliivaghani/chatbot-dataset

Appendices

- # Lung Cancer

1. Importing Libraries

```
import os
import numpy as np
from PIL import Image
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dropout,Flatten, Dense, BatchNormalization, ReLU, Add
from tensorflow.keras.models import Model
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from sklearn.preprocessing import LabelEncoder
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

2. Loading and Preprocessing the Dataset

```
dataset_path = 'The IQ-OTHNCCD lung cancer dataset/'

input_shape = (128, 128, 3)
num_classes = 3


images = []
labels = []
class_names = ['Normal','Malignant', 'Benign']
class_mapping = {class_name: index for index, class_name in enumerate(class_names)}

for class_dir in os.listdir(dataset_path):
    class_dir_path = os.path.join(dataset_path, class_dir)
    if os.path.isdir(class_dir_path):
        print(f"Reading images from class directory: {class_dir}")
        for image_file in os.listdir(class_dir_path):
            image_path = os.path.join(class_dir_path, image_file)
            try:
                image = Image.open(image_path)
                image = image.resize(input_shape[:2])
                image = np.array(image)
                if image.shape == input_shape:
                    images.append(image)
                    labels.append(class_dir)
                else:
                    print(f"Image {image_path} is skipped due to shape mismatch.")
            except Exception as e:
                print(f"Error reading image {image_path}: {e}")
```

3. Data Normalization and Encoding

```
images = np.array(images, dtype=np.float32) / 255.0
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(labels)
labels = to_categorical(integer_encoded, num_classes=num_classes)
```

4. Dataset Splitting

```
shuffled_indices = np.arange(images.shape[0])
np.random.shuffle(shuffled_indices)
images = images[shuffled_indices]
labels = labels[shuffled_indices]


train_images, test_images, train_labels, test_labels = train_test_split(images, labels, test_size=0.25, random_state=42)
```

5. Building the CNN Model

```
inputs = Input(shape=input_shape)

x = Conv2D(64, (3,3), padding='same')(inputs)
x = BatchNormalization()(x)
x = ReLU()(x)
x = MaxPooling2D((2, 2))(x)

shortcut = x
x = Conv2D(64, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = ReLU()(x)
x = Conv2D(64, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = Add()([x, shortcut])
x = ReLU()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = ReLU()(x)
x = Dropout(0.5)(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = ReLU()(x)
x = Dropout(0.5)(x)
x = MaxPooling2D((2, 2))(x)
```

```
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(num_classes, activation='softmax')(x)


model = Model(inputs, x)
```

6. Compiling and Training the Model

```
opt = Adam(learning_rate=0.0001)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

callbacks = [EarlyStopping(    monitor="val_accuracy",
    patience=20,
    verbose=0,
    mode="max",
    baseline=None,
    restore_best_weights=True,),
            ModelCheckpoint(filepath='model.h5.keras', monitor='val_accuracy', save_best_only=True)]

history = model.fit(train_images, train_labels,callbacks=callbacks, batch_size=16, epochs=80, validation_data=(test_images, test_labels))
model.save('model.h5')
```

7. Evaluation and Visualization

```
loss, accuracy = model.evaluate(test_images, test_labels)
print(f'Test Loss: {loss}, Test Accuracy: {accuracy}')
```

```
plt.figure(figsize=(12, 5))


plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')


plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')

plt.show()
```

8. Predicting and Evaluating

```
predictions = model.predict(test_images)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(test_labels, axis=1)
cm = confusion_matrix(true_classes, predicted_classes)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```

9. Plot Confusion Matrix Function

```
plt.figure(figsize=(8, 6))
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, roc_auc_score,auc, accuracy_score
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

10. Plot Normalized Confusion Matrix , classification Report and seaborn Heatmap

```
cnf_matrix = confusion_matrix(true_classes, predicted_classes)
np.set_printoptions(precision=2)

fig=plt.figure(figsize=(10, 10))
plot_confusion_matrix(cnf_matrix, classes=['Normal', 'Bengin', 'Malignant '],normalize=True,
                      title='Confusion matrix, with normalization')
plt.show()
fig.savefig('confusionmodel.jpg', bbox_inches='tight', dpi=150)
print(classification_report(true_classes, predicted_classes))
print(confusion_matrix(true_classes, predicted_classes))

cm = confusion_matrix(true_classes, predicted_classes)

plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d')
plt.title('Confusion Matrix')
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()
```

• Brain Cancer

1. Importing Libraries

```
import numpy as np
import pandas as pd
import tensorflow as tf

from keras.models import Sequential, load_model
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense, BatchNormalization
from keras.callbacks import EarlyStopping
from tensorflow.keras.utils import plot_model
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, zero_one_loss, classification_report
from sklearn.model_selection import train_test_split
from imutils import paths
import matplotlib.pyplot as plt
import seaborn as sb
import splitfolders
import cv2
import random
```

2. Dataset Splitting, Loading and Preparing the Images

```
inputFolder = 'C:\\Users\\pc\\brain'
outputFolder = 'C:\\Users\\pc\\image_dataset_(Brain)'
```

```
splitfolders.ratio(inputFolder, outputFolder, seed = 44, ratio = (0.8, 0.0, 0.2))
```

```
imgPaths = list(paths.list_images(outputFolder))
print(imgPaths[0].split('\\'))

random.shuffle(imgPaths)

classes = []
for imgPath in imgPaths:
  className = imgPath.split('\\')[-2]

  if className not in classes:
    classes.append(className)

print(classes)
```

3. Image Preprocessing and Labeling

```
X_train = []
y_train = []

X_test = []
y_test = []

IMGSIZE = 128
img_names = []

for imgPath in imgPaths:
  trainOrTest = imgPath.split('\\')[-3]
  className = imgPath.split('\\')[-2]

  img = cv2.imread(imgPath)
  img = cv2.resize(img, (IMGSIZE,IMGSIZE))

  if trainOrTest == 'train':
    X_train.append(img)
    y_train.append(classes.index(className))
    img_names.append(imgPath.split('\\')[-1])

  elif trainOrTest == 'test':
    X_test.append(img)
    y_test.append(classes.index(className))

print(len(X_train), len(y_train))
```

```
['C:', 'Users', 'pc', 'image_dataset_(Brain)', 'image_dataset_(Brain)', 'test', 'brain_glio
ma', 'brain_glioma_0001.jpg']
['brain_menin', 'brain_glioma', 'brain_pituitary', 'no_tumor']
16000 16000
```

4. Data Overview

```
df = pd.DataFrame({
    'File Name' : img_names,
    'Category' : y_train
})

print(classes.index('brain_menin'))
print(classes.index('brain_glioma'))
print(classes.index('brain_pituitary'))
print(classes.index('no_tumor'))

print(classes)
df.head()
```

```
0
1
2
3
['brain_menin', 'brain_glioma', 'brain_pituitary', 'no_tumor']
```

[5]:

	File Name	Category
0	brain_menin_1693.jpg	0
1	brain_tumor_3625.jpg	2
2	Tr-no_0841.jpg	3
3	brain_tumor_4804.jpg	2
4	no332.jpg	3

5. Building the CNN Model

```
model = Sequential()

model.add(Conv2D(filters = 32, kernel_size = (3, 3), activation = 'relu', input_shape = (IMGSIZE, IMGSIZE, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu'))
model.add(MaxPooling2D((2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 128, kernel_size = (3, 3), activation = 'relu'))
model.add(MaxPooling2D((2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 256, kernel_size = (3, 3), activation = 'relu'))
model.add(MaxPooling2D((2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation = 'relu'))
model.add(Dropout(0.25))

model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.25))

model.add(Dense(64, activation = 'relu'))
model.add(Dense(4, activation = 'softmax'))
```

6. Compiling and Training , Saving and Loading the Model

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
early_stopping = EarlyStopping(monitor = 'val_loss', patience = 5, restore_best_weights = True)
results = model.fit(X_train, y_train, validation_data = (X_test, y_test), epochs = 25, batch_size = 32, callbacks = [early_stopping])


y_pred = model.predict(X_test)

model.summary()
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 126, 32)	896
batch_normalization (BatchNormalization)	(None, 126, 126, 32)	128
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
dropout (Dropout)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
dropout_1 (Dropout)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
dropout_2 (Dropout)	(None, 14, 14, 128)	0
conv2d_3 (Conv2D)	(None, 12, 12, 256)	295,168
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 256)	0
dropout_3 (Dropout)	(None, 6, 6, 256)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 256)	2,359,552
dropout_4 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
dropout_5 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8,256
dense_3 (Dense)	(None, 4)	260

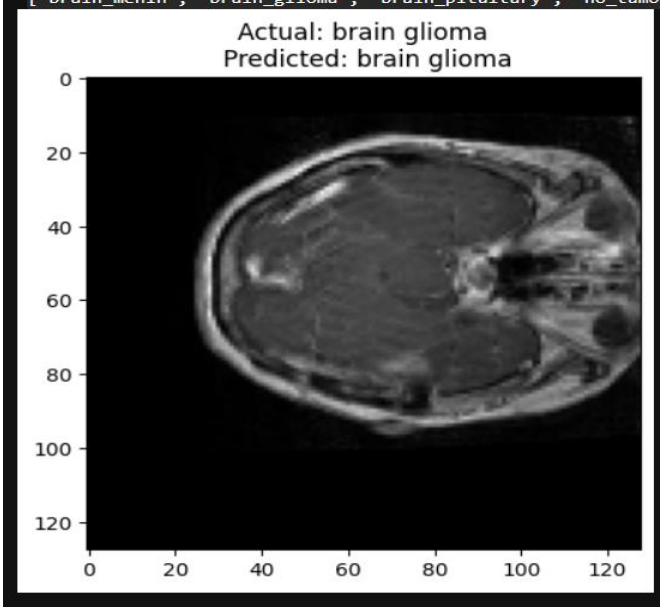
```
classes = ['brain_menin', 'brain_glioma', 'brain_pituitary', 'no_tumor']
IMGSIZE = 128
```

```
model.save('C:\\Users\\pc\\model_image_(Brain).h5')
loaded_model = load_model('C:\\Users\\pc\\model_image_(Brain).h5')
```

7. Single Image Prediction

```
test_img_path = 'C:\\Users\\pc\\image_dataset_(Brain)\\image_dataset_(Brain)\\test\\brain_glioma\\brain_glioma_0032.jpg'
test_img = cv2.imread(test_img_path)
test_img = cv2.resize(test_img, (IMGSIZE,IMGSIZE))

plt.imshow(test_img)

test_img = test_img[tf.newaxis, ...]
test_predict = loaded_model(test_img)


str1 = test_img_path.split('\\')[-1].split('_')[0]
str2 = test_img_path.split('\\')[-1].split('_')[1]
str3 = classes[np.argmax(test_predict)].split('_')[0] + ' ' + classes[np.argmax(test_predict)].split('_')[1]

plt.title('Actual: ' + str1 + ' ' + str2 + '\n Predicted: ' + str3)

print(test_predict)
print(classes)
plt.show()
```

```
tf.Tensor([[4.4729648e-05 9.9995267e-01 9.5756968e-07 1.6504063e-06]], shape=(1, 4), dtype=float32)
['brain_menin', 'brain_glioma', 'brain_pituitary', 'no_tumor']
```



- # Heart Segmentation

1 . Environment Initialization and GPU Configuration

```
import os
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import Sequence
from tensorflow.keras import layers, models
import albumentations as A
from albumentations.core.composition import OneOf

# Enable GPU memory growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)
```

2. Dataset Paths Setup

```
images_train = r"D:\BIG DATA\Models\CT-Heart-segmentation-using-U-NET\CT-Heart\new_data\train\image"
masks_train = r"D:\BIG DATA\Models\CT-Heart-segmentation-using-U-NET\CT-Heart\new_data\train\mask"
images_val = r"D:\BIG DATA\Models\CT-Heart-segmentation-using-U-NET\CT-Heart\new_data\valid\image"
masks_val = r"D:\BIG DATA\Models\CT-Heart-segmentation-using-U-NET\CT-Heart\new_data\valid\mask"
```

✓ **Recommended directory structure:**

```
new_data/
    train/
        image/  # CT scan images
        mask/   # Corresponding segmentation masks
    valid/
        image/  # Validation CT scans
        mask/   # Validation masks
```

3.DataGenerator Implementation

```
class DataGenerator(Sequence):
    def __init__(self, image_dir, mask_dir, batch_size=4, img_size=(256, 256), augment=False):
        self.image_filenames = sorted(os.listdir(image_dir))
        self.mask_filenames = sorted(os.listdir(mask_dir))
        self.image_paths = [os.path.join(image_dir, fname) for fname in self.image_filenames]
        self.mask_paths = [os.path.join(mask_dir, fname) for fname in self.mask_filenames]
        self.batch_size = batch_size
        self.img_size = img_size
        self.augment = augment

        if len(self.image_paths) != len(self.mask_paths):
            raise ValueError("Image and mask count mismatch.")
```

4.Data Augmentation Setup

```
        self.augmentations = A.Compose([
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.2),
            A.RandomRotate90(p=0.5),
            A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=15, p=0.5),
            OneOf([
                A.GaussNoise(p=0.5),
                A.Blur(blur_limit=3, p=0.5),
                A.MotionBlur(p=0.5),
            ], p=0.3),
        ])
```

5. Data Loading and Preprocessing

```
    def __getitem__(self, index):
        batch_images = self.image_paths[index * self.batch_size:(index + 1) * self.batch_size]
        batch_masks = self.mask_paths[index * self.batch_size:(index + 1) * self.batch_size]

        images, masks = [], []

        for img_path, mask_path in zip(batch_images, batch_masks):
            img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
            mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
            if img is None or mask is None:
                continue
        batch_images = self.image_paths[index * self.batch_size:(index + 1) * self.batch_size]
        batch_masks = self.mask_paths[index * self.batch_size:(index + 1) * self.batch_size]

        images, masks = [], []

        for img_path, mask_path in zip(batch_images, batch_masks):
            img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
            mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
            if img is None or mask is None:
                continue

            img = cv2.resize(img, self.img_size)
            mask = cv2.resize(mask, self.img_size)

            if self.augment:
                augmented = self.augmentations(image=img, mask=mask)
                img = augmented['image']
                mask = augmented['mask']

            images.append(np.expand_dims(img, axis=-1) / 255.0
            masks.append(np.expand_dims(mask, axis=-1) / 255.0)

        return np.array(images, dtype=np.float32), np.array(masks, dtype=np.float32)
```

6.U-NET Model Definition

```
def unet_model(input_size=(256, 256, 1)):
    inputs = tf.keras.Input(input_size)

    # Downsampling
    c1 = layers.Conv2D(64, 3, activation='relu', padding='same')(inputs)
    c1 = layers.Conv2D(64, 3, activation='relu', padding='same')(c1)
    p1 = layers.MaxPooling2D((2, 2))(c1)

    # ... (additional downsampling blocks)
    c1 = layers.Conv2D(64, 3, activation='relu', padding='same')(c1)
    p1 = layers.MaxPooling2D((2, 2))(c1)

    # ... (additional downsampling blocks)

    # Bottleneck
    c5 = layers.Conv2D(1024, 3, activation='relu', padding='same')(p4)
    c5 = layers.Conv2D(1024, 3, activation='relu', padding='same')(c5)

    # Upsampling
    u6 = layers.UpSampling2D((2, 2))(c5)
    u6 = layers.Concatenate()([u6, c4])
    c6 = layers.Conv2D(512, 3, activation='relu', padding='same')(u6)
    c6 = layers.Conv2D(512, 3, activation='relu', padding='same')(c6)

    # ... (additional upsampling blocks)

    outputs = layers.Conv2D(1, 1, activation='sigmoid')(c9)
    model = models.Model(inputs=[inputs], outputs=[outputs])
    return model
```

7. Model Training

```
# Initialize data generators
train_gen = DataGenerator(images_train, masks_train, batch_size=4, augment=True)
val_gen = DataGenerator(images_val, masks_val, batch_size=4, augment=False)

# Compile model
model = unet_model()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train model
history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=10,
    verbose=1
)
model.save("D:unet_heart_segmentation_model.h5")
```

```
2026/2026 [==============================] - 1044s 515ms/step - loss: 0.1225 - accuracy: 0.9670 - val_loss: 0.1234 - val_accuracy: 0.9651
Epoch 6/10
2026/2026 [==============================] - 1037s 512ms/step - loss: 0.1199 - accuracy: 0.9670 - val_loss: 0.1250 - val_accuracy: 0.9651
Epoch 7/10
2026/2026 [==============================] - 948s 467ms/step - loss: 0.1190 - accuracy: 0.9670 - val_loss: 0.1218 - val_accuracy: 0.9653
Epoch 8/10
2026/2026 [==============================] - 1095s 540ms/step - loss: 0.1181 - accuracy: 0.9670 - val_loss: 0.1225 - val_accuracy: 0.9653
Epoch 9/10
2026/2026 [==============================] - 958s 473ms/step - loss: 0.1180 - accuracy: 0.9670 - val_loss: 0.1211 - val_accuracy: 0.9655
Epoch 10/10
2026/2026 [==============================] - 972s 480ms/step - loss: 0.1230 - accuracy: 0.9670 - val_loss: 0.1058 - val_accuracy: 0.9651
```

- # Knee Osteoarthritis

1. Import Libraries

```
import os
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, models
from PIL import Image
from sklearn.metrics import accuracy_score
from tqdm import tqdm
```

2. KneeDataset Class Definition

```
class KneeDataset(Dataset):
    def __init__(self, df, transform=None):
        self.df = df.reset_index(drop=True)
        self.transform = transform


    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        img_path = self.df.loc[idx, 'image_path']
        label = self.df.loc[idx, 'label']
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, label
```

3. Load Dataset Function

```python
def load_dataset_as_dataframe(subdir):
    data_dir = f'D:\\kneeeee\\OS Collected Data\\{subdir}'
    image_paths = []
    labels = []
    classes = [d for d in os.listdir(data_dir) if os.path.isdir(os.path.join(data_dir, d))]
    classes.sort()
    class_to_idx = {class_name: idx for idx, class_name in enumerate(classes)}

    for class_name in classes:
        class_dir = os.path.join(data_dir, class_name)
        label = class_to_idx[class_name]
        for img_name in os.listdir(class_dir):
            img_path = os.path.join(class_dir, img_name)
            if os.path.isfile(img_path):
                image_paths.append(img_path)
                labels.append(label)

    data = pd.DataFrame({
        'image_path': image_paths,
        'label': labels
    })
    return data
```

4. Data Transformations and Loading

```python
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=10),
    transforms.ColorJitter(contrast=0.2, brightness=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

train_dataset = KneeDataset(train_df, transform=train_transform)
test_dataset = KneeDataset(test_df, transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=0)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=0)
```

5. Model Setup

```
model = models.resnet50(weights='IMAGENET1K_V1')
num_ftrs = model.fc.in_features
num_classes = len(set(train_df['label']))
model.fc = nn.Linear(num_ftrs, num_classes)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

6. Training Setup

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

best_accuracy = 0.0
best_model_path = "D:\\best_knee_model_resnet5.pth"

num_epochs = 100
```

7. Training Loop

```python
pbar = tqdm(train_loader, desc=f"Epoch [{epoch+1}/{num_epochs}]")
for images, labels in pbar:
    images = images.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()

    outputs = model(images)
    loss = criterion(outputs, labels)

    loss.backward()
    optimizer.step()

    running_loss += loss.item() * images.size(0)

    _, preds = torch.max(outputs, 1)
    total_correct += torch.sum(preds == labels.data)
    total_samples += labels.size(0)

    pbar.set_postfix({
        'loss': f'{loss.item():.4f}',
        'acc': f'{(total_correct/total_samples*100):.2f}%'
    })

epoch_loss = running_loss / len(train_dataset)
epoch_acc = total_correct.double() / len(train_dataset)
print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}')

if epoch_acc > best_accuracy:
    best_accuracy = epoch_acc
    torch.save(model.state_dict(), best_model_path)
    print(f"Best model saved with training accuracy: {best_accuracy:.4f}")
```

8. Final Evaluation

```
model.load_state_dict(torch.load(best_model_path))
model.eval()

all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        _, preds = torch.max(outputs, 1)

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

accuracy = accuracy_score(all_labels, all_preds)
print(f'Final Accuracy on Test Set: {accuracy:.4f}')
```

- # Chatbot

1.Import Libraries

```
import json
import numpy as np
import tensorflow as tf
from transformers import AutoTokenizer, TFAutoModelForSeq2SeqLM, DataCollatorForSeq2Seq
from datasets import load_dataset
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
```

2.Load Dataset

```
# Load Dataset
dataset = load_dataset("json", data_files="Chatdata.json", split="train")
```

✓ 0.5s

3. Convert to DataFrame for Analysis

```
# Convert to DataFrame
df = pd.DataFrame(dataset)

# Basic info
print(" pairs of entries:", len(df))
print(" Columns:", df.columns.tolist())
```

4.Load Model and Tokenizer

```
# 2. Load Tokenizer and Model
model_name = "google/flan-t5-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForSeq2SeqLM.from_pretrained(model_name)
```

5.Freezing Layers (Partial Fine-Tuning)

```
for layer in model.encoder.block[:5]:
    layer.trainable = False

for layer in model.decoder.block[:7]:
    layer.trainable = False
```

6. Preprocessing and Tokenization

```
#  Define max lengths for inputs and outputs
MAX_INPUT_LENGTH = 128
MAX_TARGET_LENGTH = 64
```

```
#  Preprocessing Function
def preprocess(example):
    inputs = tokenizer(example['input'], max_length=MAX_INPUT_LENGTH, truncation=True, padding='max_length')
    targets = tokenizer(example['output'], max_length=MAX_TARGET_LENGTH, truncation=True, padding='max_length')
    inputs["labels"] = targets["input_ids"]
    return inputs
```

7. Tokenize the Entire Dataset and Split

```
# Tokenize Dataset
tokenized_dataset = dataset.map(preprocess)

# Split dataset for evaluation
split_dataset = tokenized_dataset.train_test_split(test_size=0.2)
tokenized = {
    "train": split_dataset["train"],
    "test": split_dataset["test"]
}
```

8. Data Collator

```
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model, return_tensors="tf")
```

```python
# Convert to tf.data.Dataset
tf_dataset = tokenized["train"].to_tf_dataset(
    columns=["input_ids", "attention_mask", "labels"],
    shuffle=True,
    batch_size=16,
    collate_fn=data_collator
)

# Convert test split to tf.data.Dataset for validation
tf_validation_dataset = tokenized["test"].to_tf_dataset(
    columns=["input_ids", "attention_mask", "labels"],
    shuffle=False,
    batch_size=16,
    collate_fn=data_collator
)
```

9. Important Words and Token IDs

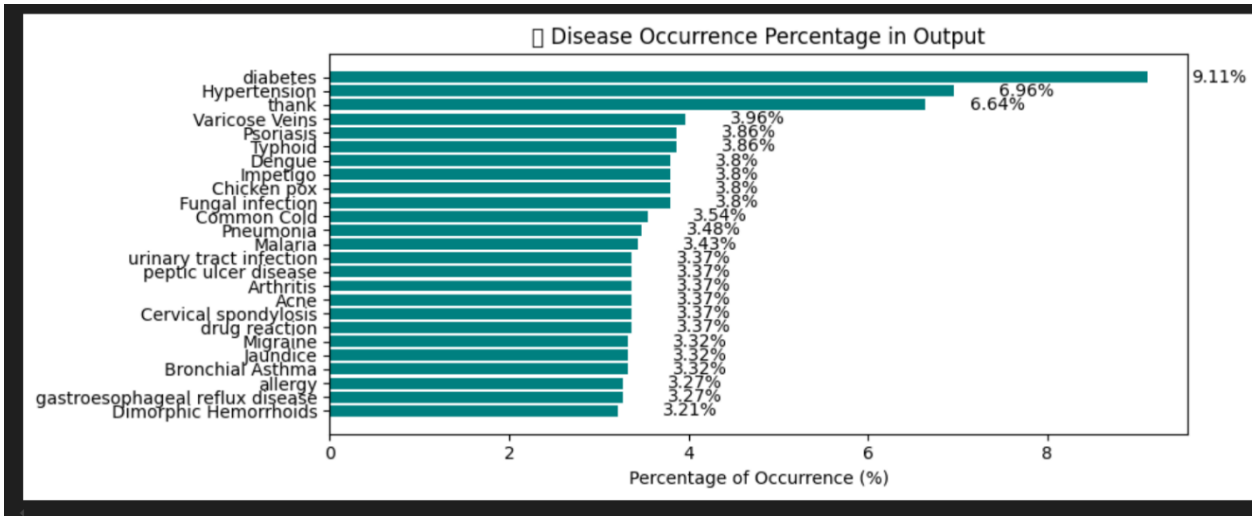
```python
important_words =['Psoriasis', 'Varicose Veins', 'Typhoid', 'Chicken pox', 'Impetigo', 'Dengue',
 'Fungal infection', 'Common Cold', 'Pneumonia', 'Dimorphic Hemorrhoids',
 'Arthritis', 'Acne', 'Bronchial Asthma', 'Hypertension',"thank", 'Migraine',
 'Cervical spondylosis', 'Jaundice', 'Malaria', 'urinary tract infection',
 'allergy', 'gastroesophageal reflux disease', 'drug reaction',
 'peptic ulcer disease', 'diabetes']

important_ids = tokenizer(important_words, add_special_tokens=False).input_ids
important_ids = [item for sublist in important_ids for item in sublist]  # flatten

# Cast important_ids to tf.int64 to match the likely dtype of y_true (labels)
important_ids = tf.constant(important_ids, dtype=tf.int64)
```

[6]

10. Disease Frequency Visualization

```
disease_counts = {disease: 0 for disease in important_words}

for disease in important_words:
    disease_counts[disease] = df['output'].str.lower().str.contains(disease.lower()).sum()

# Convert to DataFrame for plotting
count_df = pd.DataFrame(list(disease_counts.items()), columns=['Disease', 'Count'])

# Filter out zero-count entries (not in dataset)
count_df = count_df[count_df['Count'] > 0]

# Calculate percentages
total = count_df['Count'].sum()
count_df['Percentage'] = (count_df['Count'] / total * 100).round(2)

# Sort by frequency
count_df = count_df.sort_values(by='Count', ascending=False)
```

```
# Plot
plt.figure(figsize=(11, 5))
plt.barh(count_df['Disease'], count_df['Percentage'], color='teal')
plt.xlabel("Percentage of Occurrence (%)")
plt.title("📊 Disease Occurrence Percentage in Output")
plt.gca().invert_yaxis()
for index, value in enumerate(count_df['Percentage']):
    plt.text(value + 0.5, index, f"{value}%", va='center')
plt.tight_layout()
plt.show()
```



11. Custom Weighted Loss Function

```python
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')

def custom_weighted_loss(y_true, y_pred):
    loss = loss_object(y_true, y_pred)

    important_mask = tf.cast(
        tf.reduce_any(tf.equal(tf.expand_dims(y_true, -1), important_ids), axis=-1),
        tf.float32
    )
    boosted_loss = loss * (1.0 + 2.5 * important_mask)
    return tf.reduce_mean(boosted_loss)
```

12.Custom Metric: ImportantTokenAccuracy

```python
class ImportantTokenAccuracy(tf.keras.metrics.Metric):
    def __init__(self, name="important_accuracy", **kwargs):
        super().__init__(name=name, **kwargs)
        self.total = self.add_weight(name="total", initializer="zeros")
        self.correct = self.add_weight(name="correct", initializer="zeros")

    # UserWarning fix: Renamed reset_states to reset_state
    def reset_state(self):
        self.total.assign(0)
        self.correct.assign(0)

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_pred_ids = tf.argmax(y_pred, axis=-1)

        important_mask = tf.reduce_any(
            tf.equal(tf.expand_dims(tf.cast(y_true, tf.int64), -1), important_ids),
            axis=-1
        )

        important_tokens = tf.boolean_mask(y_true, important_mask)
        important_preds = tf.boolean_mask(y_pred_ids, important_mask)

        matches = tf.cast(tf.equal(important_tokens, important_preds), tf.float32)

        # Ensure shapes are compatible for sum and size if boolean_mask results in empty tensors
        if tf.size(matches) > 0:
            self.correct.assign_add(tf.reduce_sum(matches))
            self.total.assign_add(tf.cast(tf.size(matches), tf.float32))

    def result(self):
        return tf.math.divide_no_nan(self.correct, self.total)
```

13. Model Compilation and Training

```python
# Compile the model
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)

model.compile(
    optimizer=optimizer,
    loss=custom_weighted_loss,
    metrics=[ImportantTokenAccuracy()]
)
```

```python
# Train
print("Starting training...")
model.fit(
    tf_dataset,
    validation_data=tf_validation_dataset,
    epochs=50
)
print("Training finished.")
```

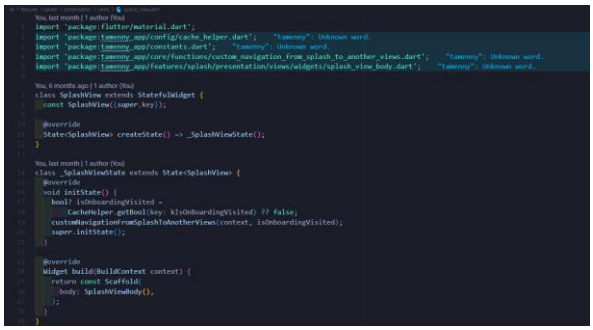
```
148/148 [==============================] - 153s 571ms/step - loss: 13.0730 - important_accuracy: 0.5213 - val_loss: 3.9458 - val_important_accuracy: 0.6173
Epoch 2/50
148/148 [==============================] - 74s 499ms/step - loss: 2.7327 - important_accuracy: 0.5854 - val_loss: 1.0038 - val_important_accuracy: 0.7337
Epoch 3/50
148/148 [==============================] - 74s 499ms/step - loss: 1.1972 - important_accuracy: 0.6708 - val_loss: 0.7192 - val_important_accuracy: 0.7818
Epoch 4/50
148/148 [==============================] - 74s 500ms/step - loss: 0.8940 - important_accuracy: 0.7456 - val_loss: 0.6051 - val_important_accuracy: 0.8219
Epoch 5/50
148/148 [==============================] - 74s 499ms/step - loss: 0.7510 - important_accuracy: 0.7864 - val_loss: 0.5299 - val_important_accuracy: 0.8454
Epoch 6/50
148/148 [==============================] - 74s 499ms/step - loss: 0.6499 - important_accuracy: 0.8086 - val_loss: 0.4645 - val_important_accuracy: 0.8605
Epoch 7/50
148/148 [==============================] - 74s 499ms/step - loss: 0.5792 - important_accuracy: 0.8350 - val_loss: 0.4104 - val_important_accuracy: 0.8841
Epoch 8/50
148/148 [==============================] - 74s 499ms/step - loss: 0.5151 - important_accuracy: 0.8510 - val_loss: 0.3615 - val_important_accuracy: 0.9020
Epoch 9/50
148/148 [==============================] - 74s 499ms/step - loss: 0.4812 - important_accuracy: 0.8583 - val_loss: 0.3198 - val_important_accuracy: 0.9133
Epoch 10/50
148/148 [==============================] - 74s 499ms/step - loss: 0.4167 - important_accuracy: 0.8762 - val_loss: 0.2835 - val_important_accuracy: 0.9246
Epoch 11/50
148/148 [==============================] - 74s 499ms/step - loss: 0.3782 - important_accuracy: 0.8876 - val_loss: 0.2504 - val_important_accuracy: 0.9387
Epoch 12/50
148/148 [==============================] - 74s 499ms/step - loss: 0.3397 - important_accuracy: 0.8952 - val_loss: 0.2197 - val_important_accuracy: 0.9430
Epoch 13/50
148/148 [==============================] - 74s 499ms/step - loss: 0.3128 - important_accuracy: 0.9050 - val_loss: 0.1954 - val_important_accuracy: 0.9458
...
148/148 [==============================] - 74s 499ms/step - loss: 0.0337 - important_accuracy: 0.9916 - val_loss: 0.0276 - val_important_accuracy: 0.9915
Epoch 50/50
148/148 [==============================] - 74s 499ms/step - loss: 0.0360 - important_accuracy: 0.9900 - val_loss: 0.0266 - val_important_accuracy: 0.9901
Training finished.
```

14.Save The Model

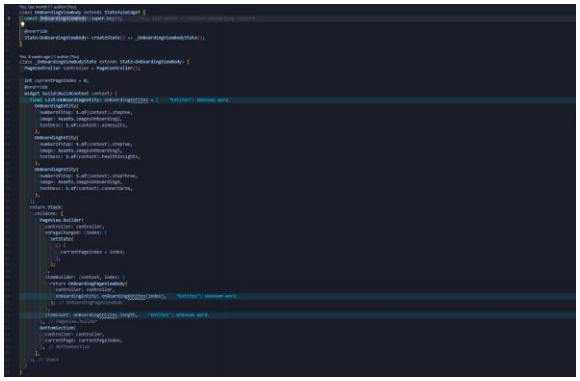
```python
# Save the fine-tuned model
model.save_pretrained("finetuned-flan-t5-base")
tokenizer.save_pretrained("finetuned-flan-t5-base")
```

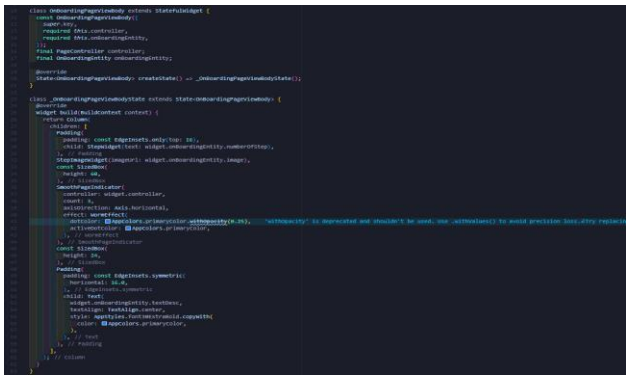
- **Flutter**

1 .Splash Feature



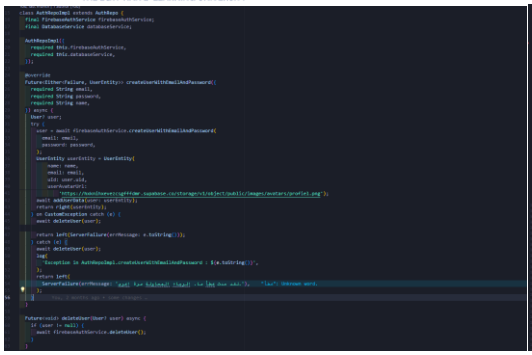
2 . Onboarding Feature





3 .Authentication Feature

```dart
abstract class AuthRepo {
  Future<Either<Failure, UserEntity>> createUserWithEmailAndPassword({
    required String email,
    required String password,
    required String name,
  });
  Future<Either<Failure, UserEntity>> signInWithEmailAndPassword({
    required String email,
    required String password,
  });
  Future<Either<Failure, UserEntity>> signInWithGoogle();
  Future<Either<Failure, UserEntity>> signInWithFacebook();

  Future<Either<Failure, void>> forgotPassword({required String email});

  Future addUserData({required UserEntity user});
  Future saveUserData({required UserEntity user});
  Future<UserEntity> getUserData({required String uid});
}
```



```dart
  @override
  Future<Either<Failure, UserEntity>> signInWithFacebook() async {
    User? user;
    try {
      user = await firebaseAuthService.signInWithFacebook();
      UserEntity userEntity = UserModel.fromFirebaseUser(user).toEntity();
      var isUserExist = await databaseService.checkIfDataExists(
        path: BackendEndPoint.isUserExists,
        documentId: user.uid,
      );
      if (!isUserExist) {
        await addUserData(user: userEntity);
      } else {
        userEntity = await getUserData(uid: user.uid);
      }
      await saveUserData(user: userEntity);
      return right(userEntity);
    } catch (e) {
      await deleteUser(user);
      log('Exception in AuthRepoImpl.signInWithFacebook: ${e.toString()}');
      return left(
        ServerFailure(errMessage: 'Something happened'),
      );
    }
  }

  @override
  Future addUserData({required UserEntity user}) async {
    await databaseService.addData(
      path: BackendEndPoint.addUserData,
      data: UserModel.fromEntity(user).toJson(),
      documentId: user.uId,
    );
  }

  @override
  Future<UserEntity> getUserData({required String uid}) async {
    return UserModel.fromJson(
      await databaseService.getData(
        path: BackendEndPoint.getUserData,
        documentId: uid,
      ),
    ).toEntity();
  }

  @override
  Future saveUserData({required UserEntity user}) async {
    final userBox = Hive.box<UserModel>('user');
    await userBox.put('currentUser', UserModel.fromEntity(user));
  }

  @override
  Future<Either<Failure, void>> forgotPassword({required String email}) async {
    try {
      await firebaseAuthService.forgotPassword(email: email);
      return right(null);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
}
```

```dart
part 'signup_state.dart';

You, 2 months ago | 1 author (You)
class SignupCubit extends Cubit<SignupState> {
  SignupCubit(this._authRepo) : super(SignupInitial());

  final AuthRepo _authRepo;

  createUserWithEmailAndPassword({
    required String name,
    required String email,
    required String password,
  }) async {
    emit(SignupLoading());
    var result = await _authRepo.createUserWithEmailAndPassword(
      email: email,
      password: password,
      name: name,
    );
    result.fold(
      (failure) => emit(SignupFailure(errMessage: failure.errMessage)),
      (userEntity) => emit(SignupSuccess(userEntity: userEntity)),
    );
  }
}
```

```dart
class SigninCubit extends Cubit<SignInState> {
  SigninCubit(this._authRepo) : super(SignInInitial());
  final AuthRepo _authRepo;

  signInWithEmailAndPassword({
    required String email,
    required String password,
  }) async {
    emit(SignInLoading());
    var result = await _authRepo.signInWithEmailAndPassword(
      email: email,
      password: password,
    );
    result.fold(
      (failure) {
        emit(SignInFailure(errMessage: failure.errMessage));
      },
      (userEntity) {
        emit(SignInSuccess(userEntity: userEntity));
      },
    );
  }

  signInWithGoogle() async {
    emit(SignInLoading());
    var result = await _authRepo.signInWithGoogle();
    result.fold(
      (failure) {
        emit(SignInFailure(errMessage: failure.errMessage));
      },
      (userEntity) {      You, 2 months ago • some changes …
        emit(SignInSuccess(userEntity: userEntity));
      },
    );
  }

  signInWithFacebook() async {
    emit(SignInLoading());
    var result = await _authRepo.signInWithFacebook();
    result.fold(
      (failure) {
        emit(SignInFailure(errMessage: failure.errMessage));
      },
      (userEntity) {
        emit(SignInSuccess(userEntity: userEntity));
      },
    );
  }
}
```

```dart
class ForgotPasswordCubit extends Cubit<ForgotPasswordState> {
  ForgotPasswordCubit(this.authRepo) : super(ForgotPasswordInitial());

  final AuthRepo authRepo;

  forgotPassword({required String email}) async {
    emit(ForgotPasswordLoading());
    var result = await authRepo.forgotPassword(email: email);
    result.fold((f) => emit(ForgotPasswordFailure(errMessage: f.errMessage)),
        (d) => emit(ForgotPasswordSuccess()));
  }
}
```

4 .Home Feature

```dart
You, last month | 1 author (You)
abstract class MedicalNewsRepo {
  Future<Either<Failure, List<ArticleEntity>>> getLatestMedicalNews();
}
```

```dart
abstract class LatestScansRepo {
  Future<Either<Failure, List<DiagnosisResultEntity>>> fetchLatestScans(
      {required String userId});
}
```

```dart
class MedicalNewsRepoImpl extends MedicalNewsRepo {
  final MedicalNewsApiService _medicalNewsApiService;

  MedicalNewsRepoImpl(this._medicalNewsApiService);

  @override
  Future<Either<Failure, List<ArticleEntity>>> getLatestMedicalNews() async {
    try {
      var data = await _medicalNewsApiService.get(
          endPoint:
              'latest?apikey=${AppKeys.kMedicalNewsApiKey}&category=health&language=en');    "apikey": Unknown word.
      List<ArticleEntity> articles = [];
      for (var article in data['results']) {
        articles.add(ArticleModel.fromJson(article).toEntity());
      }
      return right(articles);
    } catch (e) {
      log('MedicalNewsRepoImpl => {getLatestMedicalNews} => ${e.toString()}');
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
}
```

```dart
class LatestScansRepoImpl implements LatestScansRepo {
  final DatabaseService databaseService;

  LatestScansRepoImpl(this.databaseService);

  @override
  Future<Either<Failure, List<DiagnosisResultEntity>>> fetchLatestScans(
      {required String userId}) async {
    try {
      Map<String, dynamic> user = await databaseService.getData(
          path: BackendEndPoint.getUserData, documentId: userId);
      List<DiagnosisResultEntity>? diagnoses =
          (user['diagnoses'] as List<dynamic>?)?.map((diagnosis) {
        return DiagnosisResultModel.fromJson(diagnosis as Map<String, dynamic>)
            .toEntity();
      }).toList();
      return right(diagnoses ?? []);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
}
```

```dart
class LatestScansCubit extends Cubit<LatestScansState> {
  LatestScansCubit(this.latestScansRepo) : super(LatestScansInitial());

  final LatestScansRepo latestScansRepo;

  fetchLatestScans() async {
    emit(LatestScansLoading());
    var result = await latestScansRepo.fetchLatestScans(      You, 3 weeks ago • add latest scans …
        userId: getIt<UserCubit>().currentUser!.uId);

    result.fold((f) => emit(LatestScansFailure(errMessage: f.errMessage)),
        (diagnosis) => emit(LatestScansSuccess(diagnosis: diagnosis)));
  }
}
```

```dart
class MedicalNewsCubit extends Cubit<MedicalNewsState> {
  MedicalNewsCubit(this.medicalNewsRepo) : super(MedicalNewsInitial());

  final MedicalNewsRepo medicalNewsRepo;
  List<ArticleEntity> articlesList = [];

  getMedicalNews() async {
    emit(MedicalNewsLoading());
    var result = await medicalNewsRepo.getLatestMedicalNews();
    result.fold((f) => emit(MedicalNewsFailure(errMessage: f.errMessage)),
        (articles) {
      articlesList = articles;
      emit(MedicalNewsSuccess(articles: articles));
    });
  }
}
```

5. Map Feature

```dart
abstract class NearbyDoctorsRepo {
  Future<Either<Failure, List<DoctorEntity>>> fetchNearbyDoctors();
}
```

```dart
You, 4 weeks ago | 1 author (You)
class NearbyDoctorsRepoImpl extends NearbyDoctorsRepo {
  final DatabaseService databaseService;

  NearbyDoctorsRepoImpl(this.databaseService);

  @override
  Future<Either<Failure, List<DoctorEntity>>> fetchNearbyDoctors() async {
    try {
      var result =
          await databaseService.getData(path: BackendEndPoint.getDoctors);
      List<DoctorEntity> doctors = (result as List<Map<String, dynamic>>)
          .map((doctor) => DoctorModel.fromJson(doctor).toEntity())
          .toList();
      return right(doctors);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
}
```

```dart
class NearbyDoctorsCubit extends Cubit<NearbyDoctorsState> {
  NearbyDoctorsCubit(this.nearbyDoctorsRepo) : super(NearbyDoctorsInitial());
  final NearbyDoctorsRepo nearbyDoctorsRepo;
  List<DoctorEntity> doctorsList = [];

  fetchNearbyDoctors() async {
    emit(NearbyDoctorsLoading());
    var result = await nearbyDoctorsRepo.fetchNearbyDoctors();
    result.fold((f) => emit(NearbyDoctorsFailure(errMessage: f.errMessage)),
        (doctors) {
      emit(NearbyDoctorsSuccess(doctors: doctors));
      doctorsList = doctors;
    });
  }
}
```

6. Profile Feature

```dart
abstract class ChangePasswordRepo {
  Future<Either<Failure, void>> changePassword(
      {required String currentPassword, required String newPassword});
}
```

```dart
class ChangePasswordRepoImpl implements ChangePasswordRepo {
  final FirebaseAuthService firebaseAuthService;

  ChangePasswordRepoImpl(this.firebaseAuthService);
  @override
  Future<Either<Failure, void>> changePassword(
      {required String currentPassword, required String newPassword}) async {
    try {
      bool verifiedPassword =
          await firebaseAuthService.checkPassword(currentPassword);
      if (verifiedPassword) {
        await FirebaseAuth.instance.currentUser!.updatePassword(newPassword);
      } else {
        throw Exception('verified password is $verifiedPassword and cant go');
      }

      return right(null);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
}
```

```dart
class EditProfileCubit extends Cubit<EditProfileState> {
  EditProfileCubit(this.storageService, this.databaseService)
      : super(EditProfileInitial());

  final StorageService storageService;
  final DatabaseService databaseService;

  final ImagePicker _picker = ImagePicker();    The value of the field '_picker' isn't used.Try removing the field, or using it.
  final userCubit = getIt<UserCubit>();


  Future<void> updateProfile({
    required String userId,
    XFile? newAvatar,
    String? newUsername,
    String? newEmail,
  }) async {
    await _performUpdate(userId, newAvatar, newUsername, newEmail);
  }

  Future<void> _performUpdate(      You, 3 weeks ago • user can update full data …
    String userId,
    XFile? newAvatar,
    String? newUsername,
    String? newEmail,
  ) async {
    emit(EditProfileLoading());

    try {
      String? imageUrl;

      if (newAvatar != null) {
        String fileName =
            '${DateTime.now().millisecondsSinceEpoch}_${newAvatar.path.split('/').last}';
        String path = 'avatars/$fileName';

        imageUrl = await storageService.uploadFile(file: newAvatar, path: path);
        await databaseService.updateUserAvatar(
            userId: userId, imageUrl: imageUrl);
      }

      if (newUsername != null && newUsername.isNotEmpty) {
        await databaseService.updateData(
          path: BackendEndPoint.updateUserData,
          documentId: userId,
          data: {'name': newUsername},
        );
      }
```

```dart
      if (newEmail != null && newEmail.isNotEmpty) {
        User? user = FirebaseAuth.instance.currentUser;
        if (user != null) {
          await user.verifyBeforeUpdateEmail(newEmail);
          await user.reload();
        } else {
          throw Exception("User not logged in");
        }
      }

      final userBox = Hive.box<UserModel>('user');
      final currentUser = userBox.get('currentUser');

      if (currentUser != null) {
        var updatedUser = currentUser;
        if (imageUrl != null) {
          updatedUser = updatedUser.copyWith(userAvatarUrl: imageUrl);
        }
        if (newUsername != null && newUsername.isNotEmpty) {
          updatedUser = updatedUser.copyWith(name: newUsername);
        }
        if (newEmail != null && newEmail.isNotEmpty) {
          updatedUser = updatedUser.copyWith(email: newEmail);
        }
        await userBox.put('currentUser', updatedUser);
        userCubit.saveUser(updatedUser);
      }

      emit(EditProfileSuccess(imageUrl ?? ''));
    } catch (e) {
      if (e is FirebaseAuthException && e.code == 'requires-recent-login') {
        emit(EditProfileError('Please re-authenticate to update your email.'));
      } else {
        emit(EditProfileError(e.toString()));
      }
    }
  }
}
```

```dart
class ChangePasswordCubit extends Cubit<ChangePasswordState> {
  ChangePasswordCubit(this.changePasswordRepo, this.firebaseAuthService)
      : super(ChangePasswordInitial());

  final ChangePasswordRepo changePasswordRepo;
  final FirebaseAuthService firebaseAuthService;

  changePassword(
      {required String enteredPassword, required String newPassword}) async {
    emit(ChangePasswordLoading());

    var result = await changePasswordRepo.changePassword(
        currentPassword: enteredPassword, newPassword: newPassword);
    result.fold(
        (f) => emit(ChangePasswordFailure(errMessage: f.errMessage)),
        (v) => emit(
              ChangePasswordSuccess(),
            ));
  }
}
```

7. Ai Diagnosis Feature

```dart
class DiagnosisRepoImpl extends DiagnosisRepo {
  final AIDiagnosisService aiDiagnosisService;
  final DatabaseService databaseService;
  final StorageService storageService;

  DiagnosisRepoImpl(
    this.aiDiagnosisService,
    this.databaseService,
    this.storageService,
  );

  @override
  Future<void> addDiagnosisToFireStore({
    required String userId,
    required Map<String, dynamic> diagnosis,
  }) async {
    try {
      await FirebaseFirestore.instance.collection('users').doc(userId).set({
        'diagnoses': FieldValue.arrayUnion([diagnosis]),
      }, SetOptions(merge: true)); // Merges with the existing document
    } catch (e) {
      print("Error adding diagnosis to Firestore: $e");    Don't invoke 'print' in production code.dTry using a logging framework.
      rethrow;
    }
  }

  @override
  Future<Either<Failure, DiagnosisResultEntity>> startDiagnosis({
    required XFile image,
  }) async {
    try {
      final userId = getIt<UserCubit>().currentUser!.uId;

      final Map<String, dynamic> result =
          await aiDiagnosisService.lungDiagnosis(imageFile: File(image.path));

      final String diagnosisResult = result['result'] ?? 'Unknown';
      String diagnosisSummary;

      if (diagnosisResult == 'Malignant') {
        diagnosisSummary =
            'The AI model has analyzed the lung scan and identified the findings as malignant. This suggests the presence of potentially cancerous tissue that may require immediate medical attention and further diag
      } else if (diagnosisResult == 'Benign') {
        diagnosisSummary =
            'The AI model has analyzed the lung scan and identified the findings as benign. This indicates that the detected tissue or nodule is non-cancerous and does not currently show signs of malignancy. However
      } else {
        diagnosisSummary = '';
      }

      final String scanImageUrl = await addDignosisImageToSupabase(    "Dignosis": Unknown word.
        userId: userId,
        image: image,
      );

      final diagnosis = DiagnosisResultEntity(
        status: diagnosisResult,
        scanImageUrl: scanImageUrl,
        diagnosisId: const Uuid().v4(),
        scannedAt: DateTime.now(),
        diagnosisSummary: diagnosisSummary,
      ); // DiagnosisResultEntity
```

```dart
      final String scanImageUrl = await addDignosisImageToSupabase(    "Dignosis": Unknown word.
        userId: userId,
        image: image,
      );

      final diagnosis = DiagnosisResultEntity(
        status: diagnosisResult,
        scanImageUrl: scanImageUrl,
        diagnosisId: const Uuid().v4(),
        scannedAt: DateTime.now(),
        diagnosisSummary: diagnosisSummary,
      ); // DiagnosisResultEntity

      await addDiagnosisToFireStore(
        userId: userId,
        diagnosis: DiagnosisResultModel.fromEntity(diagnosis).toJson(),
      );

      return Right(diagnosis);
    } catch (e) {
      return Left(ServerFailure(errMessage: e.toString()));
    }
  }

  @override
  Future<String> addDignosisImageToSupabase(    "Dignosis": Unknown word.
    required String userId,
    required XFile image,
  }) async {
    try {
      final imageUrl = await storageService.uploadFile(
        file: image,
        path: BackendEndPoint.addDiagnosisImage,
      );
      return imageUrl;
    } catch (e) {
      throw Exception('Failed to upload image to Supabase: $e');
    }
  }
}
```

8.Community Feature

```dart
abstract class CommunityRepo {
  Stream<Either<Failure, List<PostEntity>>> getPosts();
  Future<Either<Failure, void>> addPost({required PostEntity post});
  Future<Either<Failure, void>> addComment(
      {required CommentEntity comment, required PostEntity post});
  Future<Either<Failure, void>> addLike({
    required PostEntity post,
    required String userId,
  });
}

```

```dart
class CommunityRepoImpl implements CommunityRepo {
  final DatabaseService databaseService;

  CommunityRepoImpl(this.databaseService);

  @override
  Stream<Either<Failure, List<PostEntity>>> getPosts() async* {
    try {
      await for (var (result as List<Map<String, dynamic>>) in databaseService
          .streamData(
              path: BackendEndPoint.getPosts,
              query: {'orderBy': 'createdAt', 'descending': true})) {
        List<PostEntity> posts = (result as List<dynamic>).map((data) {
          return PostModel.fromJson(data).toEntity();
        }).toList();
        yield right(posts);
      }
    } catch (e) {
      yield left(ServerFailure(errMessage: e.toString()));
    }
  }

  @override
  Future<Either<Failure, void>> addPost({required PostEntity post}) async {
    try {
      await databaseService.addData(
          path: BackendEndPoint.addPost,
          documentId: post.postId,
          data: PostModel.fromEntity(post).toJson());
      return right(null);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }

  @override
  Future<Either<Failure, void>> addComment(
      {required CommentEntity comment, required PostEntity post}) async {
    try {
      await databaseService.updateData(
          path: BackendEndPoint.addComment,
          documentId: post.postId,
          data: {
            'comments': FieldValue.arrayUnion(
                [CommentModel.fromEntity(comment).toJson()])
          });
      return right(null);
    } catch (e) {
      return left(ServerFailure(errMessage: e.toString()));
    }
  }
```

```dart
@override
Future<Either<Failure, void>> addLike(
    {required PostEntity post, required String userId}) async {
  try {
    final postRef =
        FirebaseFirestore.instance.collection('posts').doc(post.postId);

    final alreadyLiked = post.likedBy.contains(userId);

    if (alreadyLiked) {
      await postRef.update({
        'likedBy': FieldValue.arrayRemove([userId]),
        'likesCount': FieldValue.increment(-1),
      });
    } else {
      await postRef.update({
        'likedBy': FieldValue.arrayUnion([userId]),
        'likesCount': FieldValue.increment(1),
      });
    }      You, 4 weeks ago • user can like post in community …

    return const Right(null);
  } catch (e) {
    return Left(ServerFailure(errMessage: eitherM.toString()));
  }
 }
}
```

- # Recommendation System:

✓ Xlm-r classifier

1.Imports

```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments, EarlyStoppingCallback, AutoConfig
from datasets import Dataset
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, f1_score
import pandas as pd
import numpy as np
import torch
import os
import joblib
import random
```

2.Setting Random Seeds for Reproducibility

```python
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(seed)
```

3.Data loading and preprocessing

```
df = pd.read_excel(r"C:/Users/Kareem/Desktop/postsclassifier/TOP_ELPOP.xlsx", names=["Source", "Post", "Tag"], skiprows=1)
df = df[df['Source'] != 'Tag'].dropna()

label_encoder = LabelEncoder()
df['label'] = label_encoder.fit_transform(df['Tag'])
joblib.dump(label_encoder, r"D:\label_encoder.joblib")

train_texts, test_texts, train_labels, test_labels = train_test_split(df['Post'], df['label'], test_size=0.1, stratify=df['label'], random_state=seed)
train_texts, val_texts, train_labels, val_labels = train_test_split(train_texts, train_labels, test_size=0.1, stratify=train_labels, random_state=seed)
```

4.Model initialization and dataset splitting

```
model_name = "xlm-roberta-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)

config = AutoConfig.from_pretrained(model_name)
config.num_labels = len(label_encoder.classes_)
config.hidden_dropout_prob = 0.3
config.attention_probs_dropout_prob = 0.3

model = AutoModelForSequenceClassification.from_pretrained(model_name, config=config)

for name, param in model.named_parameters():
    if not (("layer.10" in name) or ("layer.11" in name) or ("classifier" in name)):
        param.requires_grad = False

def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=128)

train_dataset = Dataset.from_dict({"text": train_texts.tolist(), "label": train_labels.tolist()}).map(tokenize_function, batched=True)
val_dataset = Dataset.from_dict({"text": val_texts.tolist(), "label": val_labels.tolist()}).map(tokenize_function, batched=True)
test_dataset = Dataset.from_dict({"text": test_texts.tolist(), "label": test_labels.tolist()}).map(tokenize_function, batched=True)

train_dataset = train_dataset.remove_columns(["text"])
val_dataset = val_dataset.remove_columns(["text"])
test_dataset = test_dataset.remove_columns(["text"])      You, 2 weeks ago • post classifier uploaded
```

5.Compute metrics function and Training Arguments adjustments

```python
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    return {
        "accuracy": accuracy_score(labels, preds),
        "f1": f1_score(labels, preds, average="weighted"),
    }

training_args = TrainingArguments(
    output_dir=r"D:\xlmr_final_modelhhh\xlmr_logs",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16,
    num_train_epochs=15,
    weight_decay=0.1,
    warmup_ratio=0.1,
    lr_scheduler_type="cosine",
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    logging_dir=r"D:\xlmr_final_modelhhh\xlmr_logs\logs",
    logging_strategy="steps",
    logging_steps=10,
    save_total_limit=2,      You, 2 weeks ago • post classifier uploaded
    fp16=torch.cuda.is_available(),
    max_grad_norm=1.0,
    report_to=[]
)
```

6.Training and Evaluating the Model

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=2)]
)


trainer.train()

model_path = r"D:\xlmr_final_model"
os.makedirs(model_path, exist_ok=True)
model.save_pretrained(model_path)
tokenizer.save_pretrained(model_path)

print("Final evaluation on test set:")
results = trainer.evaluate(test_dataset)
print(results)
```

✓ NN multi regressior:

1.Imports

```python
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, r2_score
import json
```

2.Data loading and preparation

```python
df = pd.read_csv("D:/final_user_based_dataset_scores_100_extended.csv")

feature_cols = [
    'ratio_Heart', 'ratio_Brain', 'ratio_Lung', 'ratio_Knee', 'ratio_Neutral',
    'available_Heart', 'available_Brain', 'available_Lung', 'available_Knee', 'available_Neutral'
]
target_cols = ['score_Heart', 'score_Brain', 'score_Lung', 'score_Knee', 'score_Neutral']

X = df[feature_cols].copy()
y = df[target_cols].copy()

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.25, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

scaler_data = {
    "mean": scaler.mean_.tolist(),
    "std": scaler.scale_.tolist()
}
with open("scaler_values_for_flutter.json", "w") as f:
    json.dump(scaler_data, f, indent=4)
```

3.Training and evaluation with saving:

```
model = models.Sequential([
    layers.Input(shape=(10,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(5)
])

model.compile(optimizer='adam', loss='mae', metrics=['mae'])

model.fit(X_train_scaled, y_train, epochs=300, batch_size=32,
          validation_data=(X_val_scaled, y_val), verbose=1)

y_pred = model.predict(X_val_scaled)

print("MAE:", mean_absolute_error(y_val, y_pred))
print("R²:", r2_score(y_val, y_pred))

for i, col in enumerate(target_cols):
    mae = mean_absolute_error(y_val.iloc[:, i], y_pred[:, i])
    r2 = r2_score(y_val.iloc[:, i], y_pred[:, i])
    print(f"- {col}: MAE={mae:.2f}, R²={r2:.4f}")

model.save("disease_recommendation_model.h5")

converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

with open("disease_recommendation_model5.tflite", "wb") as f:
    f.write(tflite_model)
```

✓ Tammeny Main api:

1.imports

```python
from fastapi import FastAPI, File, UploadFile, Form
from fastapi.responses import JSONResponse
import uvicorn
from PIL import Image      Kareem1099, 4 weeks ago • fresh staaaart
import numpy as np
import torch
from torchvision import models
from io import BytesIO
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import joblib
import shutil
import cv2
import os
```

2.post-classifier endpoint

```python
app = FastAPI()

# -------------------- Load XLM-R Text Classification Model --------------------
text_model_path = r"D:\xlmr_final_model"
text_tokenizer = AutoTokenizer.from_pretrained(text_model_path)
text_model = AutoModelForSequenceClassification.from_pretrained(text_model_path)
text_model.eval()
label_encoder = joblib.load(r"D:\label_encoder.joblib")

@app.post("/predict_Text")
async def predict_text(text: str = Form(...)):
    try:
        inputs = text_tokenizer(text, return_tensors="pt", truncation=True, padding=True, max_length=128)
        with torch.no_grad():
            outputs = text_model(**inputs)
            logits = outputs.logits
            predicted_class_id = torch.argmax(logits, dim=1).item()
            predicted_label = label_encoder.inverse_transform([predicted_class_id])[0]

        return predicted_label
    except Exception as e:      You, 15 hours ago • Uncommitted changes
        return JSONResponse(str(e))
```

3.lung end point:

```python
lung_model = load_model('D:\\Models\\Models\\Lung cancer detection\\model.h5.keras')
lung_class_names = ['Normal', 'Malignant', 'Benign']

def preprocess_lung_image(img):
    img = img.resize((128, 128))
    img = image.img_to_array(img)
    img = img / 255.0
    img = np.expand_dims(img, axis=0)
    return img

@app.post("/predict_Lung")
async def predict_lung(file: UploadFile = File(...)):
    try:
        contents = await file.read()
        img = Image.open(BytesIO(contents)).convert('RGB')
        processed_img = preprocess_lung_image(img)
        predictions = lung_model.predict(processed_img)
        predicted_index = np.argmax(predictions[0])
        predicted_class = lung_class_names[predicted_index]
        return {
            "result": predicted_class,
        }
    except Exception as e:
        return JSONResponse({"error": str(e)})
```

3.knee endpoint

```python
knee_model = models.resnet50(weights=None)
knee_model.fc = torch.nn.Linear(knee_model.fc.in_features, 2)
checkpoint = torch.load("D:\\best_knee_model_resnet5.pth", map_location=torch.device("cpu"))
knee_model.load_state_dict(checkpoint)
knee_model.eval()

def preprocess_knee_image(image: Image.Image):
    image = image.resize((224, 224))
    image_array = np.array(image) / 255.0
    if image_array.ndim == 2:
        image_array = np.stack([image_array] * 3, axis=-1)
    image_array = np.transpose(image_array, (2, 0, 1))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image_array = (image_array - mean[:, None, None]) / std[:, None, None]
    image_tensor = torch.tensor(image_array, dtype=torch.float32).unsqueeze(0)
    return image_tensor

@app.post("/predict_Knee")
async def predict_knee(file: UploadFile = File(...)):
    try:
        image_bytes = await file.read()
        image = Image.open(BytesIO(image_bytes)).convert("RGB")
        processed_image = preprocess_knee_image(image)
        with torch.no_grad():
            prediction = knee_model(processed_image)
        predicted_class = torch.argmax(prediction, dim=1).item()
        result = "Healthy knee" if predicted_class == 0 else "There is knee Osteoarthritis"
        return {
            "result": result,      Kareem1099, 4 weeks ago • fresh staaaart
        }
    except Exception as e:
        return JSONResponse({"error": str(e)})
```

4.brain endpoint

```python
brain_model = load_model("C:/Users/Kareem/Desktop/TammenyMainApi/models/model_image_(Brain).h5")
brain_classes = ['brain_menin', 'brain_glioma', 'brain_pituitary', 'no_tumor']

def preprocess_brain_image(file_path: str):
    img = cv2.imread(file_path)
    img = cv2.resize(img, (128, 128))
    img = img / 255.0
    img = np.expand_dims(img, axis=0)
    return img

@app.post("/predict_Brain")
async def predict_brain(file: UploadFile = File(...)):
    try:
        temp_path = "temp_brain.jpg"
        with open(temp_path, "wb") as buffer:
            shutil.copyfileobj(file.file, buffer)

        processed_img = preprocess_brain_image(temp_path)
        predictions = brain_model.predict(processed_img)
        predicted_index = np.argmax(predictions[0])
        predicted_class = brain_classes[predicted_index]

        os.remove(temp_path)

        return {
            "result": predicted_class,
        }       Kareem1099, 4 weeks ago • fresh staaaart
    except Exception as e:
        return JSONResponse({"error": str(e)})
```

5.heart endpoint

```python
heart_model = load_model("D:\\BIG DATA\\Models\\CT-Heart-segmentation-using-U-NET\\CT-Heart\\unet_finetuned.h5")

def preprocess_heart_image(image_bytes):      Kareem1099, 4 weeks ago • fresh staaaart
    image = Image.open(BytesIO(image_bytes)).convert("RGB")
    image = image.resize((256, 256))
    image = np.array(image) / 255.0
    image = np.expand_dims(image, axis=0)
    return image

@app.post("/predict_Heart Disease Analysis")
async def predict_heart(file: UploadFile = File(...)):
    try:
        image_bytes = await file.read()
        processed_image = preprocess_heart_image(image_bytes)
        prediction = heart_model.predict(processed_image)[0, :, :, 0]
        threshold = 0.5
        mask = (prediction > threshold).astype(np.uint8)
        has_heart_issue = bool(np.sum(mask) > 0)
        result = "Heart issue detected" if has_heart_issue else "Normal appearance"
        return {
            "result": result,
        }
    except Exception as e:
        return JSONResponse({"error": str(e)})
```

6.start the server

```python
# ---------------------- Run Server ----------------------
if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```