Ministry of High Education,

Culture and Science City at Oct 6,

The High institute of Computer Science & Information Systems

المعهد العالي لعلوم الحاسب ونظم المعلومات

Graduation Project:

Breast Cancer Detection System

Project No: N42304 Academic year: 2022/2023

Prepared by:

ID	Name
20193357	محمد شوقي محمد رزق
20193706	ياسر نبيل محمد محمد
20192890	محمود سيد علي أحمد
20194076	مصطفى طارق مصطفى
20193298	صلاح الدين علاء محمد روحي
20190521	عبد الرحمن محمود محمد

Assistant:

Eng. Sama Ayman

Supervised by:

Dr. Ashraf Fahmy

Acknowledgement

We have made tremendous efforts in this project. However, it would not have been possible without the outstanding Culture and Science City, where our beloved computer science institute is hosted. We would like to express our sincere gratitude and appreciation to the professors and staff who have supported us.

First and foremost, we extend our heartfelt thanks to Dr. Ashraf Fahmy for his exceptional patience, guidance, and supervision throughout the project. His invaluable advice and provision of necessary information played a crucial role in completing our project. We are grateful to have had such a remarkable advisor and mentor for our graduation project.

Secondly, we would also like to express our deepest appreciation to our advisor, Eng. Sama Ayman, for her support, patience, motivation, and knowledge. Her guidance played a pivotal role throughout the project and the writing of this thesis. We are grateful for her contributions, as her dedication and expertise have been instrumental in shaping our work. Her steadfast presence has inspired us to overcome challenges.

Lastly, we would like to acknowledge the immense support of our families and friends. Even though they may not have fully understood the details of our work, their unwavering encouragement and support have meant the world to us. To them, we owe our deepest gratitude for standing by us, even in moments of uncertainty.

Abstract

Breast cancer is a significant health concern worldwide, and early detection plays a crucial role in improving patient outcomes. In this graduation project, we propose a deep learning-based approach for breast cancer detection using a convolutional neural network (CNN).

We trained our CNN model using a large dataset consisting of 277,524 histopathology images and 54,000 X-ray images, which were obtained from various sources. Our approach addresses the problem of breast cancer detection from two different imaging modalities, histopathology and X-ray, which provide complementary information.

The histopathology images provide microscopic-level details of breast tissue samples, while X-ray images provide macroscopic-level information about breast abnormalities. By combining the strengths of both modalities, we aim to improve the accuracy and reliability of breast cancer detection.

To link the two models, we developed an application that integrates both trained CNN models for histopathology and X-ray images. The application takes input images from both modalities and feeds them into their respective CNN models for feature extraction. The extracted features are then fused at a decision-level to make the final classification decision.

We evaluated the performance of our approach on a large and diverse dataset, and achieved promising results in terms of accuracy, sensitivity, and specificity. Our approach has the potential to improve the accuracy of breast cancer detection, which can aid in early diagnosis and timely intervention, ultimately leading to better patient outcomes.

Table of Contents

Table of Figures

1. INTRODUCTION

Main Idea of Breast Cancer Detection.

In 2020, there were 2.3 million women diagnosed with breast cancer and 685000 deaths globally. As of the end of 2020, there were 7.8 million women alive who were diagnosed with breast cancer in the past 5 years, making it the world's most prevalent cancer. There are more lost disability-adjusted life years (DALYs) by women to breast cancer globally than any other type of cancer. Breast cancer occurs in all countries in women at any age.[1]

- Develop a computer-based system that can automatically analyze images and identify potential signs of breast cancer with high accuracy.
- Enables the system to provide an efficient and reliable tool for early detection and diagnosis of breast cancer, potentially improving patient outcomes and facilitating timely intervention.

Importance of Breast Cancer Detection.

Early Detection: Cancer will have higher chances of successful treatment and improved survival rates. By leveraging CNNs, to analyze images, this project aims to enhance the ability to detect breast cancer at an early stage.[2]

Accuracy and Precision: Applying these models to breast cancer detection can enhance the accuracy and precision of diagnosing breast abnormalities.

Support for Radiologists: Deep learning models can serve as valuable decision support tools, aiding radiologists in their analysis and increasing their efficiency. By reducing the workload, this project can contribute to improved diagnostic accuracy, reduced interpretation time, and enhanced workflow for radiologists.[2]

Advantages of Breast Cancer Detection.

- Early Detection.
- Improved Accuracy.
- Decision Support for Radiologists.

Disadvantage of Breast Cancer Detection.

- Limited Generalization.[3]
- Difficult Implementation.[3]

Motive of Breast Cancer Detection.

- Enhancing accuracy of cancer detection.
- Enabling early detection.
- Assisting healthcare professionals.
- Improving patient outcomes.
- Expanding access to breast cancer detection.[4]
- Complementing existing diagnostic methods.[4]
- Reducing healthcare costs.[4]
- Empowering patients.

Objectives of the project.

- Collecting and preprocessing a large dataset of Mammography and Histopathology images for training and evaluation of two models.
- Designing and implementing a convolutional neural network (CNN) architecture for the breast cancer detection task.
- Training and evaluating the two CNN models on the Mammography and Histopathology datasets, using performance metrics such as accuracy, precision, recall, and F1 score.
- Investigating the robustness and generalizability of the two CNN models to different Mammography and Histopathology datasets and imaging modalities.

- Investigating the impact of hyperparameters on the performance of the CNN model, like learning rate, batch size, and epochs number.
- Developing an interactive mobile-based interface that allows healthcare professionals to upload images and receive a prediction of whether breast cancer is present or not.

Challenges.

- Data Availability and Quality.
- Generalization to Diverse Populations.
- Ethical and Legal Considerations.
- Limited Resources and Expertise.
- Model Overfitting.
- Continuous Model Updates.

Background & Tools

Theoretical Background:

CNNs have shown outstanding performance in various image recognition tasks, including object detection, face recognition, and medical image analysis.[5]

There are Several previous studies have attempted to build breast cancer detection models using CNNs. these studies faced several challenges. **One of the main challenges** is:

- **The limited availability of annotated datasets:** Building accurate breast cancer detection models requires large and diverse datasets with annotated images. and obtaining such datasets is challenging due to the privacy concerns of patients. As a result, many previous studies used relatively small and limited datasets, which may limit the generalizability and accuracy of the models.
- **The imbalanced of the positive and negative cases in the datasets:** Most of the mammograms are negative. Therefore, the datasets used

to train the models often have an imbalance of positive and negative cases. This imbalance can affect the model's performance.

- **The interpretability of the models:** It is challenging to understand how CNNs make their predictions. The lack of interpretability may limit the clinical adoption of the models, as doctors need to understand the reasoning behind the predictions to make informed decisions.
- **The high computational requirements of CNNs:** Training CNNs on large datasets requires significant computational resources, including GPUs and high-performance computing clusters.[7]

Methodology Steps:

Several studies have been conducted. For example, in a study by Wang et al. (2016), a CNN model was developed to classify mammography images into benign and malignant categories. The model achieved an accuracy of 89.47% on a dataset of 168 images. In another study by Akram et al. (2018), a CNN model was trained on a dataset of 10,500 mammography images to classify them as normal or abnormal. The model achieved an accuracy of 92.5%, which was significantly higher than other existing CAD systems.[8]

a) Data Collection:

We collected a large dataset of mammogram images of patients with and without breast cancer. The dataset was obtained from publicly available sources, Such as the Digital Database for Screening Mammography (DDSM) and the Breast Cancer Histopathological Image Classification (BreakHis) dataset.

b) Data Preprocessing:

The collected dataset was preprocessed to enhance the quality of images and remove any noise. The preprocessing steps included image resizing, normalization, and augmentation. The images were

resized to a standard size of 224x224 pixels, normalized to have zero mean and unit variance, and augmented using random rotations and flips.

c) Model Training:

Model was trained on preprocessed dataset using a deep learning framework such as TensorFlow and Keras. The training process involved feeding the model with batches of images and optimizing the model parameters to minimize the classification error.

d) Model Evaluation:

The model evaluated on a separate set of validation data to measure its accuracy and performance. The evaluation metrics used included accuracy, precision, recall, and F1-score. We also generated a confusion matrix to analyze the performance of the model in differentiating between malignant and benign cases.

e) Model Optimization:

Improve model performance by fine-tuning the model parameters and hyperparameters. We experimented with different learning rates, batch sizes, and optimization algorithms to find the optimal settings for the model.

f) Results Analysis:

Finally, we analyzed the results and compared them with the performance of previous studies. We also investigated the factors that affect the performance of the model, such as the size of the dataset, the choice of CNN architecture.

Tools, Programs, and Specifications:

tools and programs:

a) Python programming language:

Python is a widely-used programming language for machine learning and artificial intelligence applications. It provides an extensive set of libraries for data analysis, visualization, and machine learning, making it an ideal choice for developing the breast cancer detection model.

b) TensorFlow framework:

TensorFlow is a library for dataflow and differentiable programming across a range of tasks. It is widely used for building and training deep learning models, including CNNs, and provides prebuilt functions and libraries for developing and testing models.

c) Keras API:

Keras is an open-source neural network library written in Python. It is designed to be user-friendly, modular, and extensible. Keras provides a higher-level API that simplifies the development of deep learning models, including CNNs.

d) Scikit-learn library:

Scikit-learn is a machine learning library for Python that provides simple and efficient tools for data mining and data analysis. It includes various classification, regression, and clustering algorithms, making it useful for the analysis and classification of medical images.

e) Anaconda distribution:

Anaconda is a distribution of Python and R programming languages for scientific computing that aims to simplify package management and deployment. It includes popular data science and machine learning libraries, making it easy to install and manage the required tools and libraries for developing the breast cancer detection model.

f) Jupyter Notebook:

An open-source web application that allows users to create and share documents that contain live code, equations, visualizations, and

narrative text. It is an interactive environment that facilitates the development, testing, and documentation of machine learning models.

g) To build our mobile application:

- Flutter Framework and Dart Language.
- Android Studio
- Visual Studio Code
- Android Studio

Specifications:

Lenovo L340 Ideapad is a laptop computer that can be used for a variety of tasks, including machine learning. It is a reliable and efficient device for running machine learning models, including the breast cancer detection model we are working on.

➢ Full tech specs for Lenovo L340 IdeaPad:

- Operating System (OS): Windows 10 Pro
- Processor: 9th Gen Intel® Core™ i7-9750H, 2.6 GHz
- Number of Cores: 6 , Cache: 12 MB
- Graphics Card (GPU): NVIDIA® GeForce® GTX 1650 4GB
- Dedicated VRAM: 4 GB GDDR6
- Standard Memory Size: 16 GB
- Installed Memory (RAM): 16 GB DDR4 SDRAM
- Memory Type: DDR4 SDRAM
- Storage (1): 256GB PCIe NVMe SSD and 1TB WD HDD
- Display Size (Inches): 15.6″
- Resolution: FHD (1920 x 1080)

2. System Analysis and Design.

Analysis.

System Requirement.

- **Hardware:** A computer or server with sufficient processing power, memory, and storage space to handle the large amounts of data generated by medical imaging.
- **Software:** A breast cancer detection software with advanced algorithms that can accurately analyze medical images and identify potential cancerous regions.
- **Medical Imaging Equipment:** High-quality mammography, ultrasound, or MRI machines that can produce clear and accurate images of the breast tissue.
- **Network Connectivity:** The system should be connected to a secure network that allows authorized users to access and share medical images and patient data.
- **Security Measures:** The system should have robust security measures in place, such as encryption, access controls, and user authentication.
- **Compliance:** The system should comply with relevant regulations and standards, such as HIPAA and FDA regulations, to ensure patient safety and privacy.

User Requirement.

- **Accuracy:** The system should have a high level of accuracy in detecting breast cancer and differentiating between benign and malignant tumors.
- **Speed:** The system should be able to provide results quickly to avoid delays in diagnosis and treatment.

- **Non-invasiveness:** The system should be non-invasive, meaning that it should not require any invasive procedures, such as biopsies, to detect breast cancer.
- **Ease of use:** The system should be user-friendly and easy to use, even for nonexpert users.
- **Cost-effectiveness:** The system should be cost-effective, meaning that it should not require a significant investment of resources to operate.
- **Availability:** The system should be widely available and accessible to patients and healthcare providers around the world.
- **Reliability:** The system should be reliable and produce consistent results.
- **Integration with existing healthcare systems:** The system should be able to integrate with existing healthcare systems to ensure seamless communication and continuity of care.
- **Confidentiality:** The system should ensure confidentiality of patient data, as breast cancer is a sensitive medical condition.
- **Scalability:** The system should be scalable, meaning that it should be able to handle large volumes of data and patient information as the number of users grows.

Function Requirements.

- **Imaging Technology:** The system must be able to use imaging technology, such as mammography, ultrasound, or MRI, to detect breast cancer.
- **Image Processing:** The system must be able to process the images obtained from the imaging technology and identify any abnormalities that could be indicative of breast cancer.

- **Accuracy:** The system must have high accuracy in detecting breast cancer to minimize false positives and false negatives.
- **Sensitivity:** The system must be able to detect breast cancer at an early stage when treatment is more effective.
- **Specificity:** The system must be able to distinguish between cancerous and noncancerous lesions.
- **Speed:** The system must be able to process images quickly to provide timely results to healthcare providers and patients.
- **Integration:** The system should be able to integrate with electronic medical records (EMR) and other healthcare systems to streamline patient care.
- **User-friendly:** The system should be easy to use for healthcare providers and patients.
- **Cost-effective:** The system should be cost-effective and accessible to a wide range of patients and healthcare providers.
- **Regulatory Compliance:** The system must comply with all relevant regulatory requirements and standards for medical devices and diagnostic tests.

Non-Function Requirements.

- **Reliability:** The system must be reliable and consistently provide accurate results, with minimal downtime or system failures.
- **Security:** The system must be secure and protect patient data and medical records, complying with relevant data protection regulations.
- **Scalability:** The system should be able to handle a high volume of patients and data, and be able to scale up or down as needed.
- **Usability:** The system should be user-friendly and easy to use for healthcare providers and patients, with clear instructions.

- **Performance:** The system should perform well, with fast response times and minimal latency.
- **Interoperability:** The system should be able to integrate with other healthcare systems and devices, to enable seamless patient care.
- **Maintainability:** The system should be easy to maintain, with clear documentation and support available for healthcare providers and technical staff.
- **Accessibility:** The system should be accessible to patients from diverse backgrounds, including those with disabilities, and should provide multilingual support if needed.
- **Ethical and Legal Compliance:** The system must comply with ethical and legal requirements, such as patient confidentiality and informed consent.
- **Cost-effectiveness:** The system should be cost-effective and offer value for money, considering factors such as hardware and software costs, maintenance, training, and support.

Flowchart.

Figure 2.1 represents the Flowchart Diagram; in our case, it represents the steps or actions that must be followed to complete the process of detecting if Cancer exists in a given image.

a) **Application Access:** if the user has an account, he Logs in to the system or application, if not he must Sign Up first.
b) **Image preparation:** the user inputs the image and it must be valid then the image gets Preprocessing for the model.
c) **Model Inference and Prediction:** the model makes a Feature Extraction and classifies the image into Normal or Cancer then prints the prediction result.

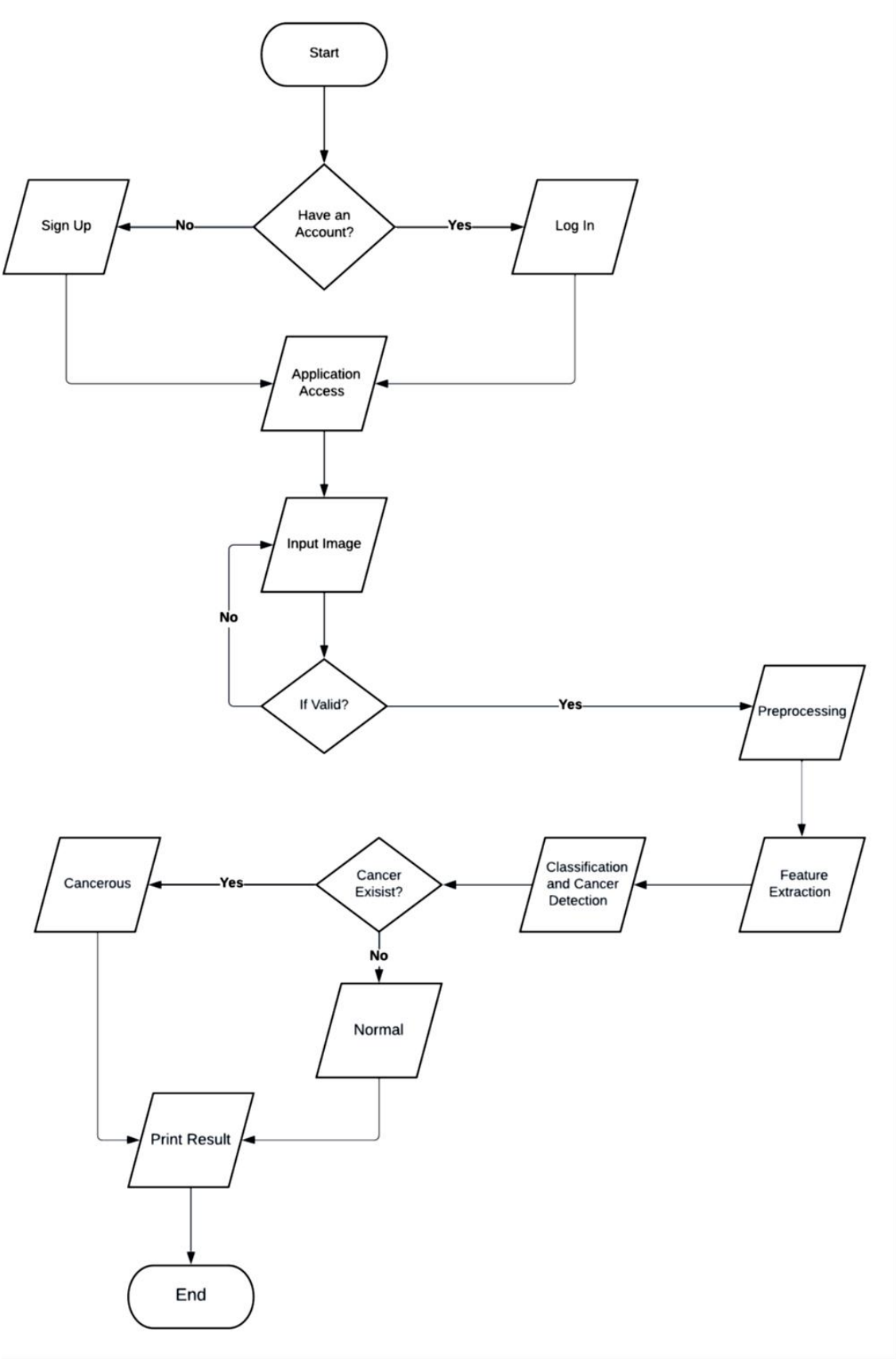

Figure 2.1 - Flowchart

Sequence Diagram.

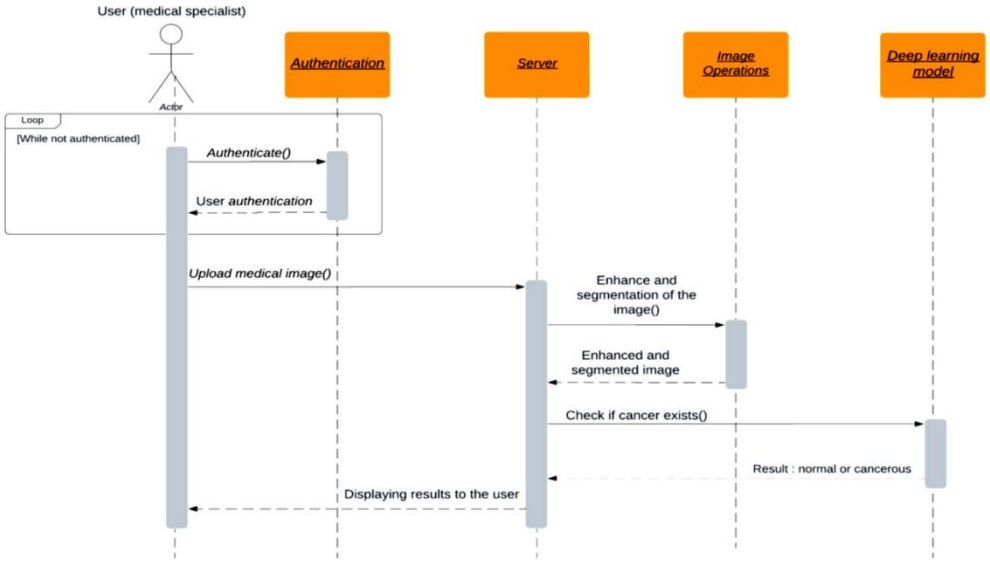

Figure 2.2 - Sequence Diagram

Figure 2.2 represents the Sequence Diagram; in our case it represents a scenario of a Breast Cancer Detection System and it has four major objects in order:-

- **Authentication:** The user needs to authenticate first before using the system or application.
- **Server:** The user uploads the medical image into the server that contains the machine learning model.
- **Image Operations:** The image then gets enhanced and segmented (Preprocessing and Feature Extraction), Preprocessing like; resizing and normalization, Feature Extraction happens inside a Deep learning-based model Convolutional Neural Network (CNN).
- **Deep learning model:** the Model classifies the image into Normal or Cancer and the result goes back to the user.

Activity Diagram.

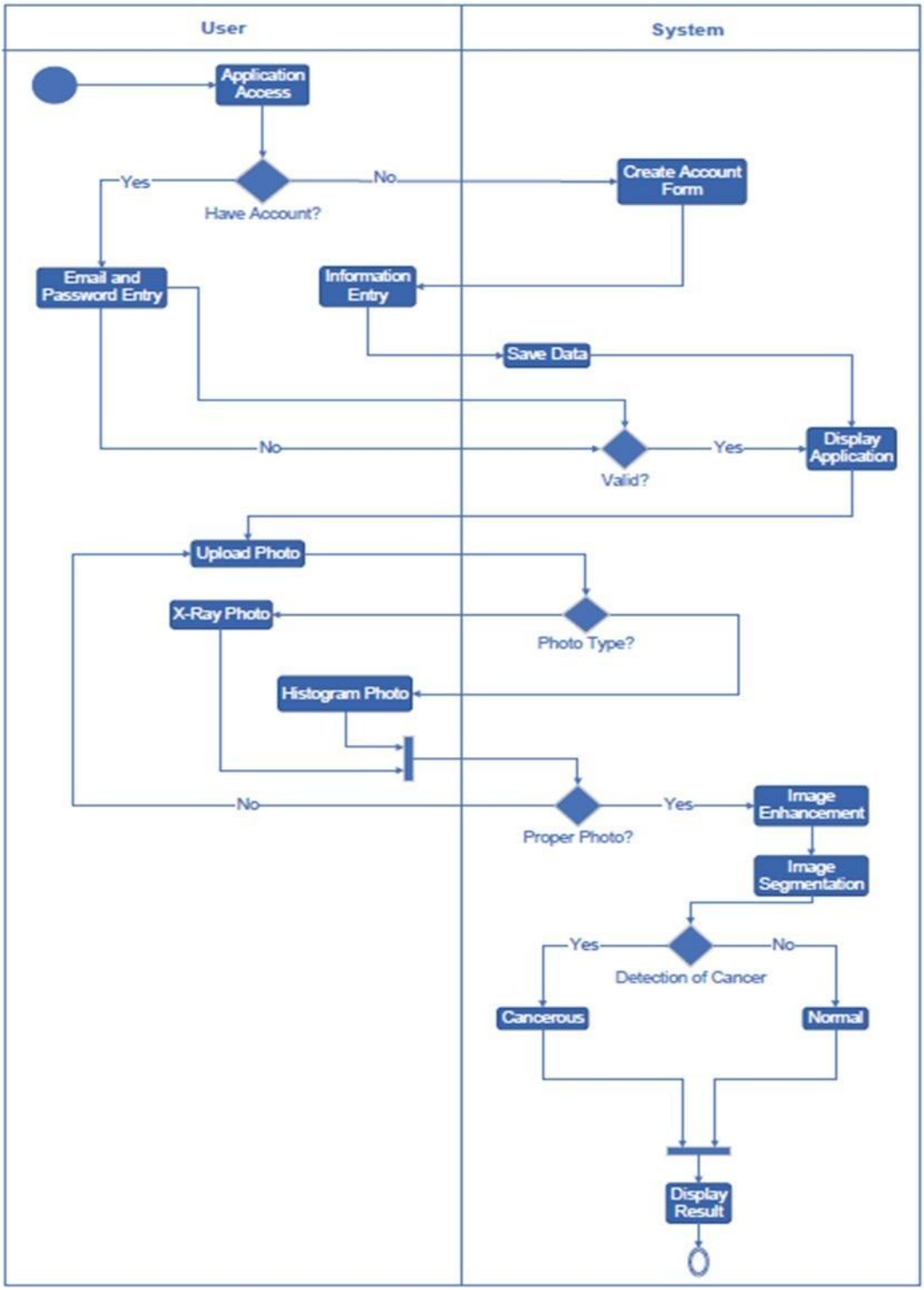

Figure 2.3 - Activity Diagram

Figure 2.3 represents the Activity Diagram**;** in our system the activity diagram represents the activities happen between the system and the user, and it goes through three phases:

1. First phase which called **Application Access:** once the user opens the application will be asked to log in, if he/she doesn't have an account he/she has the choice to create a new one and the system will provide him/her with a form to enter his/her information. After completing this phase correctly, the application contents will be displayed to the user.
2. Second phase which called **Uploading Photo:** the system asks the user to pick a proper photo and gives him/her several ways to do this such as:
   a) Choosing a photo from the gallery of the phone.
   b) Open the camera to take a photo with.
   c) Enter the link of a photo from the internet.
   d) Reading a QR code that has a link of a photo.

   After that the system asks the user to select the type of the photo weather it's an X-Ray or Histogram photo.
3. Third and Final phase which called **Cancer Detection:** The system checks if the photo is valid or not. If valid the system deploys deep learning and image processing and stages on the photo and displays the result to the user either it's normal or cancerous. If the photo is not valid the system will go back to the second phase.

Design.

UI / UX Design for the mobile application:

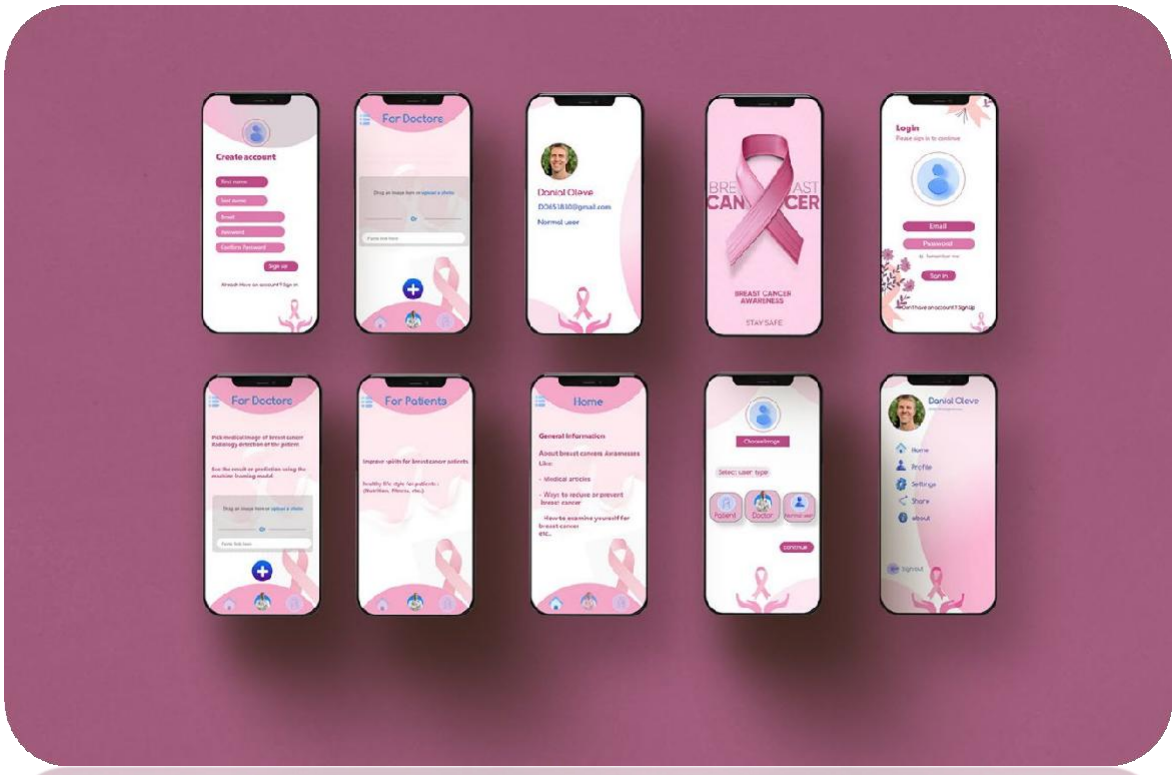

Figure 2.4-UI / UX Design

Figure 2.5-Splash Screen

Figure 2.7-Login Page

Figure 2.7-Sign Up Page

Figure 2.9-Home page

Figure 2.9-Patient Page

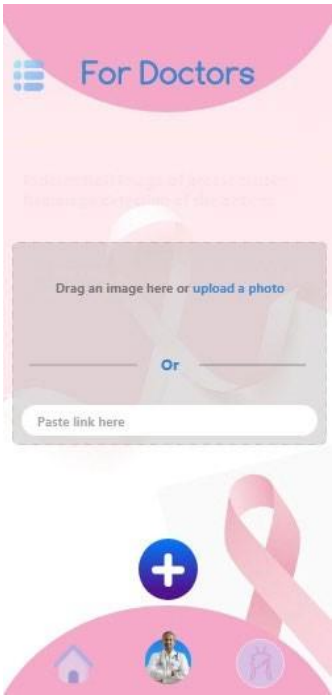

Figure 2.11-Doctors Page

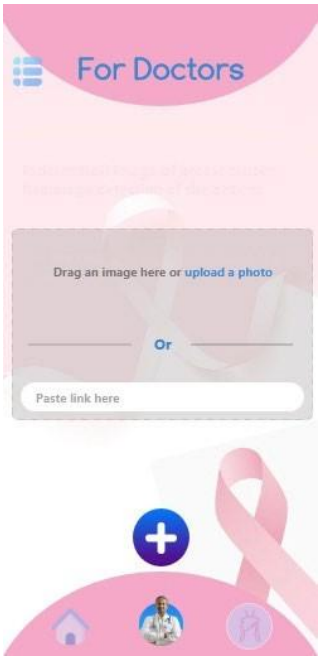

Figure 2.11-Load Image Page

Figure 2.13-Profile Page

Figure 2.13-User Type Page

3. Implementation.

X-Ray Model.

```
# TensorFlow libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential

# core libraries
import os
import glob
import numpy as np
import pandas as pd
import itertools

# basic libraries
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline
```

Figure 3.1 – Import Libraries

provided code imports various libraries and modules required for machine learning tasks using TensorFlow. It includes TensorFlow itself, Keras, scikit-learn, NumPy, Pandas, itertools, and Matplotlib.

TensorFlow-related imports include classes and modules such as **ImageDataGenerator**, **layers**, and **Sequential**, which are used for data preprocessing, building neural network models, and creating image data generators.

Core libraries like **os** and **glob** are imported to handle file and directory operations. **numpy** and **pandas** are imported for numerical computing and data manipulation tasks. **itertools** is used for efficient iteration and combination operations.

From **scikit-learn**, the code imports the **train_test_split** function for splitting data into training and testing sets, **and classification_report** and **confusion_matrix** for model evaluation.

matplotlib.pyplot is imported as **plt** for creating various types of plots, and **%matplotlib** inline is a Jupyter Notebook magic command that enables inline plotting.

```
# Load the data

# Get current working directory
current_dir = os.getcwd()

# Append data/mnist.npz to the previous path to get the full path
data_path = "/kaggle/input/rsna-breast-cancer-detection/train_images"

data_image_512_path = "/kaggle/input/rsna-breast-cancer-512-pngs"
```

```
breast_img = glob.glob("/kaggle/input/rsna-breast-cancer-512-pngs/*.png", recursive = True)
```

```
for imgname in breast_img[:3]:
    print(imgname)
```

```
/kaggle/input/rsna-breast-cancer-512-pngs/10289_1390886438.png
/kaggle/input/rsna-breast-cancer-512-pngs/21915_1598001440.png
/kaggle/input/rsna-breast-cancer-512-pngs/5123_1805049792.png
```

```
train_csv = pd.read_csv("/kaggle/input/rsna-breast-cancer-detection/train.csv")
test_csv = pd.read_csv("/kaggle/input/rsna-breast-cancer-detection/test.csv")
```

Figure 3.2 - Load Data

The **os.getcwd()** function is used to get the current working directory, and then the specific paths for the training images and the 512x512 PNG images are assigned to the variables **data_path** and **data_image_512_path** respectively.

By defining these paths, the code ensures that the subsequent data loading and processing steps can access the correct directories where the dataset files are located.

The **glob.glob()** function is used to retrieve a list of file paths that match a specified pattern. The pattern indicates a directory path followed by **/*.png**, indicating that it is looking for all files with the extension **.png** in the specified directory and its subdirectories. The **recursive=True** argument enables recursive searching; it will search for files in all subdirectories as well.

for loop iterates over the first three elements of the **breast_img** list and prints each element. This loop is used to quickly inspect and verify the file paths of the images in the **breast_img** list.

pd.read_csv() function from the Pandas library is used to read CSV (Comma-Separated Values) files and store the data in DataFrames.

```
train_csv
```

	site_id	patient_id	image_id	laterality	view	age	cancer	biopsy	invasive	BIRADS	implant	density	machine_
0	2	10006	462822612	L	CC	61.0	0	0	0	NaN	0	NaN	
1	2	10006	1459541791	L	MLO	61.0	0	0	0	NaN	0	NaN	
2	2	10006	1864590858	R	MLO	61.0	0	0	0	NaN	0	NaN	
3	2	10006	1874946579	R	CC	61.0	0	0	0	NaN	0	NaN	
4	2	10011	220375232	L	CC	55.0	0	0	0	0.0	0	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	...	
54701	1	9973	1729524723	R	MLO	43.0	0	0	0	1.0	0	C	
54702	1	9989	63473691	L	MLO	60.0	0	0	0	NaN	0	C	2
54703	1	9989	1078943060	L	CC	60.0	0	0	0	NaN	0	C	2
54704	1	9989	398038886	R	MLO	60.0	0	0	0	0.0	0	C	2
54705	1	9989	439796429	R	CC	60.0	0	0	0	0.0	0	C	2

54706 rows × 14 columns

```
test_csv
```

	site_id	patient_id	image_id	laterality	view	age	implant	machine_id	prediction_id
0	2	10008	736471439	L	MLO	81	0	21	10008_L
1	2	10008	1591370361	L	CC	81	0	21	10008_L
2	2	10008	68070693	R	MLO	81	0	21	10008_R
3	2	10008	361203119	R	CC	81	0	21	10008_R

```
train_csv.columns
```

```
Index(['site_id', 'patient_id', 'image_id', 'laterality', 'view', 'age',
       'cancer', 'biopsy', 'invasive', 'BIRADS', 'implant', 'density',
       'machine_id', 'difficult_negative_case'],
      dtype='object')
```

Figure 3.3 - Train & Test Files

train_csv and **test_csv** both are used to display all the contents of train and test CSV files. **train_csv.columns** used to display the labels of columns exist in train file.

```
train_csv.cancer.value_counts()
```

```
patients_count = len(set(train_csv.patient_id))
print(f"patients count: {patients_count}")
```

Figure 3.4 - Number of Images

train_csv.cancer.value_counts(): Calculates the number of cancer and non-cancer images in the **train_csv**, which are **53548** for **non-cancer** and **1158** for **cancer**.

patients_count = len(set(train_csv.patient_id)): Calculates the number of patients in the data**,** which are **11913** patients.

```
random_images_0 = []
random_images_1 = []

# save 8 random images path from subset_0 in arr random_images_0
for index, row in sampled_rows_subset_0.iterrows():
    image_path = row["path"] # data_image_512_path + "/" + str(row["patient_id"]) + "_" + str(row["image
    random_images_0.append(image_path)

# save 8 random images path from subset_1 in arr random_images_1
for index, row in sampled_rows_subset_1.iterrows():
    image_path = row["path"]
    random_images_1.append(image_path)
```

Figure 3.5 - Select Random Images

Two empty lists **random_images_0** and **random_images_1** are initialized to store the paths of randomly selected images from two subsets of data. The code iterates over each row in **sampled_rows_subset_0** and **sampled_rows_subset_1** DataFrames, extracts the image path from the "path" column of each row, and appends it to the corresponding list.

```
from tensorflow.keras.preprocessing.image import load_img, img_to_array

def plot_8_images_0_and_1_in_grid(images_path_0, images_path_1):
    plt.figure(figsize = (20, 20))

    s = 0
    for image_path in images_path_0:

            img = load_img(image_path, target_size=(512, 512))
            img = img_to_array(img)

            plt.subplot(4, 4, 2*s+1)
            plt.axis('off')
            plt.title('no cancer')
            plt.imshow(img.astype('uint8'))
            s += 1

    s = 1
    for image_path in images_path_1:

            img = load_img(image_path, target_size=(512, 512))
            img = img_to_array(img)
            plt.subplot(4, 4, 2*s)
            plt.axis('off')
            plt.title('cancer')
            plt.imshow(img.astype('uint8'))
            s += 1
```

Figure 3.6 - Grid of 8 Images

plot_8_images_0_and_1_in_grid plots 8 images in a grid format from two sets of image paths, **images_path_0** and **images_path_1**.

Load the image using **load_img(image_path, target_size=(512, 512))** and resize it to a target size of 512x512 pixels.

Convert the image to a NumPy array using **img_to_array(img)**.

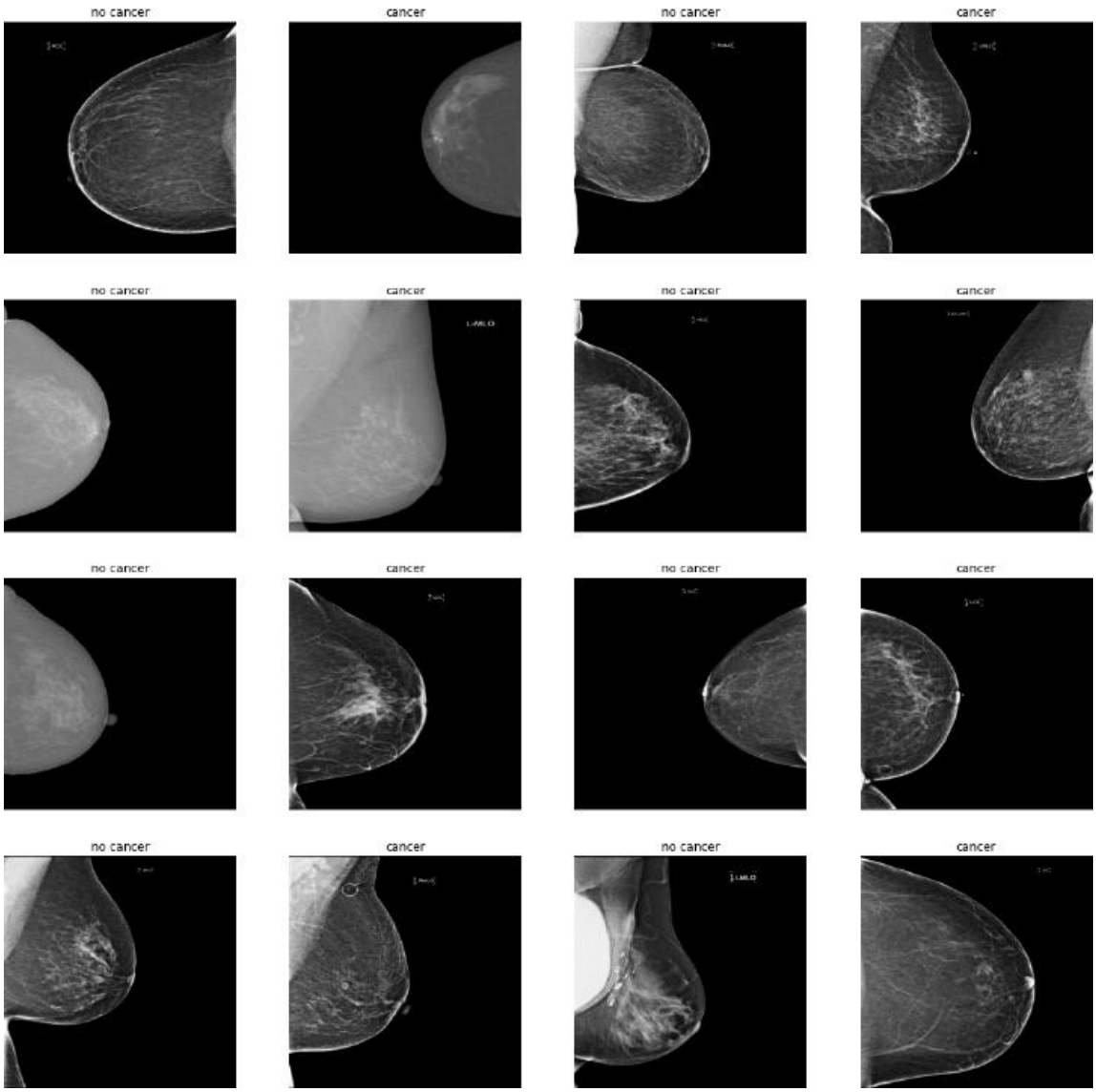

Figure 3.7 – Grid Output

A grid of subplots and displays 8 images, 4 from each subset, with labels indicating the presence or absence of cancer.

The images are loaded, resized, and displayed using Matplotlib. This function can be used to visualize a selection of images from the two subsets for analysis or evaluation purposes.

```
#### making oversampling to balance 0's and 1'

train_subset_1 = train_subset_1.sample(n=53548, replace=True)

# Concatenate the subsets
train_subset_main = pd.concat([train_subset_0, train_subset_1])
```

Figure 3.8 - Oversampling

The code performs oversampling to balance the number of samples between class 0 and class 1 in the dataset.

The **sample()** function is used on **train_subset_1** with the **n** parameter set to **53548**. This function randomly samples 53,548 samples from **train_subset_1** with replacement, meaning that some samples may be selected multiple times. The purpose is to increase the number of samples from class 1 to match the number of samples in class 0.

The resulting oversampled **train_subset_1** is concatenated with **train_subset_0**, which contains samples from class 0, using the **pd.concat()** function. This concatenation combines the samples from both classes into a new DataFrame called **train_subset_main**.

```
# train_csv if we want to train all datatset without oversampling
# but we are using train_subset_main to train the model on dataset that is balanced
# between 0's an 1's for cancer

train_df, val_df = train_test_split(train_subset_main,
                                    test_size = 0.20,
                                    random_state = 2018)

print('train', train_df.shape[0], 'validation', val_df.shape[0])
print('train', train_df['cancer'].value_counts())
print('validation', val_df['cancer'].value_counts())
```

```
train 85676 validation 21420
train 0    42967
1    42709
Name: cancer, dtype: int64
validation 1    10839
0    10581
Name: cancer, dtype: int64
```

Figure 3.9 - Train & Test Split

train_df and **val_df** are created using the **train_test_split()** function from scikit-learn, which splits the **train_subset_main** DataFrame into training and validation sets.

The **test_size** parameter is set to **0.20**, indicating that **20%** of the data will be allocated to the validation set (**val_df**), and the remaining **80%** will be allocated to the training set (**train_df**). The **random_state** parameter is set to 2018 to ensure reproducibility of the split.

The number of samples in the training and validation sets: **train_df.shape[0]** and **val_df.shape[0]**, respectively.

The distribution of samples for each class (0 and 1) in the training and validation sets: **train_df['cancer'].value_counts()** and **val_df['cancer'].value_counts()**.

```
import shutil

# Define the destination directory for train with 2 classes 0 'normal' and 1 'cancer'.
train_dir = '/kaggle/working/train'

# Create the train directory (with sub dir 0 and 1) if it doesn't exist.
if not os.path.exists(train_dir):
    os.makedirs(train_dir)
    os.makedirs(train_dir + "/0")
    os.makedirs(train_dir + "/1")


# Copy the images to the train directory.
i = 0
for _, row in train_df.iterrows():
    destination_path = os.path.join(train_dir + "/" + str(row["cancer"]) + "/" + str(row["patient_id"])
    shutil.copy2(row["path"], destination_path)
    i += 1
```

Figure 3.10 – Organize Train Images

Import the **shutil** module for file operations.

train_dir is the destination directory where the images will be organized based on their classes.

The code creates **train_dir** if not exist, along with subdirectories for classes 0 and 1, using **os.makedirs()**.

The destination path for the image is constructed based on the class label (**row["cancer"]**) and the patient ID (**row["patient_id"]**).

The **shutil.copy2()** function is used to copy the image from its original path (**row["path"]**) to the destination path.

The counter **i** is incremented to keep track of the number of images copied.

```
train_datagen = ImageDataGenerator(rescale = 1./255.)
val_datagen = ImageDataGenerator(rescale = 1./255.,)
```

```
train_path = '/kaggle/working/train'
val_path = '/kaggle/working/val'

train_generator = train_datagen.flow_from_directory(
    train_path,
    target_size = (512, 512),
    batch_size = 10,
    classes=['0', '1'],
    shuffle=True,
    seed=42
)
validation_generator = val_datagen.flow_from_directory(
        val_path,
        target_size = (512, 512),
        batch_size = 10,
        classes=['0', '1'],
        shuffle=False,
        seed=42
)
```

Figure 3.11 - Data Generator

Both **train_datagen** and **val_datagen** are configured with the **rescale** parameter set to **1./255.** This parameter scales the pixel values of the images to the range of [0, 1], effectively normalizing the image data.

train_datagen.flow_from_directory() function creates the **train_generator**. The **train_path** is specified as the directory containing the training images.

target_size parameter is set to (512, 512), images should be resized to this size.

The **batch_size** is set to 10, meaning that each batch will contain 10 images.

The **classes** parameter is set to ['0', '1'], specifying the class labels for the data.

shuffle is set to **True**, which shuffles the data within each batch.

seed is set to 42, providing a seed value for reproducibility.

```
basic_model = Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu', input_shape=(512,512,3)),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  #layers.Dense(128, activation='relu'),
  layers.Dense(2, activation='softmax')
])

basic_model.summary()
```

Figure 3.12 - CNN Model

basic_model is created as a sequential model.

The first layer is a **Conv2D** layer with 16 filters, a kernel size of 3x3, 'same' padding, and 'relu' activation. It takes an input shape of (512, 512, 3), indicating an input image size of 512x512 pixels with 3 color channels.

The next layer is a **MaxPooling2D** layer, which performs max pooling with a default pool size of 2x2.

This pattern is repeated with two more pairs of **Conv2D** and **MaxPooling2D** layers, gradually increasing the number of filters to 32 and 64, respectively.

After the last **MaxPooling2D** layer, a **Flatten** layer is added to flatten the output of the previous layer into a 1D vector.

The model then has a fully connected **Dense** layer with 2 units (classes) and a softmax activation function, which outputs the probability distribution over the two classes.

- Total params: 547,874
- Trainable params: 547,874
- Non-trainable params: 0

```
basic_model.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Figure 3.13 – Model Optimizer

optimizer='adam': The Adam optimizer is chosen as the optimization algorithm. Adam is an adaptive learning rate optimization algorithm that is commonly used for training neural networks. It dynamically adjusts the learning rate based on the gradients of the model parameters.

loss='categorical_crossentropy': The categorical cross-entropy loss function is selected. This loss function is suitable for multi-class classification problems where the target variable is represented as a one-hot encoded vector. It measures the dissimilarity between the predicted probability distribution and the true distribution.

metrics=['accuracy']: The accuracy metric is specified to be computed during training and evaluation. Accuracy is a common evaluation metric for classification tasks, and it calculates the proportion of correctly predicted samples.

By calling **compile** with these settings, the **basic_model** is prepared for training. It will use the Adam optimizer to minimize the categorical cross-entropy loss function, and the accuracy metric will be calculated to assess the model's performance.

```
basic_model_history = basic_model.fit(
    train_generator,
    validation_data = validation_generator,
    steps_per_epoch = len(train_generator),
    epochs = 15
)
```

```
Epoch 14/15
8568/8568 [==============================] - 834s 97ms/step - loss: 0.0169 - accuracy: 0.9982 - val_los
s: 0.0825 - val_accuracy: 0.9955
Epoch 15/15
8568/8568 [==============================] - 836s 98ms/step - loss: 0.0150 - accuracy: 0.9983 - val_los
s: 0.0950 - val_accuracy: 0.9951
```

Figure 3.14 – Model Training

The code trains the **basic_model** using the **fit** method. It specifies the training configuration and data generators, and trains the model for a total of 15 epochs.

train_generator is used to provide batches of training data to the model. The **validation_generator** is used for validation, providing batches of validation data to evaluate the model's performance after each epoch.

The **steps_per_epoch** parameter is set to the length of the **train_generator**, which determines the number of steps (batches) per epoch. This ensures that the model sees all the training data within each epoch.

```
basic_model.evaluate(validation_generator)
```

```
2142/2142 [==============================] - 140s 66ms/step - loss: 0.0950 - accuracy: 0.9951
[0.09503517299890518, 0.9950980544090271]
```

Figure 3.15 - Validation

The **evaluate** method returns the evaluation results, which can be used to assess the model's performance on the validation data. These results include accuracy, and loss.

```
def plot_history(history, epochs):
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']

    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs_range = range(epochs)

    plt.figure(figsize=(8, 8))
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, acc, label='Training Accuracy')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')

    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, loss, label='Training Loss')
    plt.plot(epochs_range, val_loss, label='Validation Loss')
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.show()
```

Figure 3.16 - Visualization

plot_history that can be used to plot the training and validation accuracy, as well as the training and validation loss, over the course of training. This function is useful for visualizing the performance of a trained model and understanding how it evolves over epochs. The function creates a figure with two subplots.

Output:

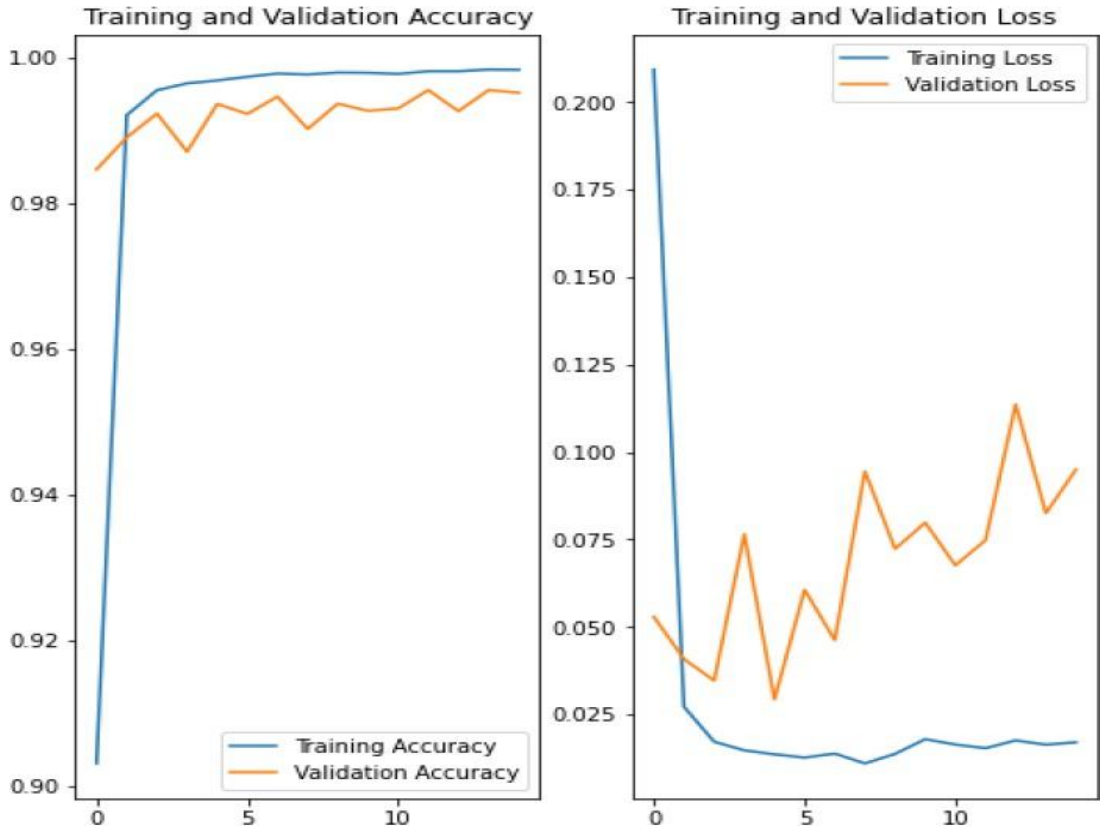

Figure 3.17 - Visualization Output

The first subplot displays the training accuracy (**acc**) and validation accuracy (**val_acc**) on the y-axis against the range of epochs (**epochs_range**) on the x-axis. It also adds a legend and a title to the plot.

The second subplot shows the training loss (**loss**) and validation loss (**val_loss**) on the y-axis against the range of epochs on the x-axis. Again, it includes a legend and a title for the plot.

Finally, the function calls **plt.show()** to display the figure with the plotted curves.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.figure(figsize = (8, 8))
    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)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

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

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

Figure 3.18 - Confusion Matrix

plot_confusion_matrix function is used to visualize a confusion matrix, which is a performance evaluation metric for classification models. It takes the confusion matrix as input and creates a plot to display the matrix. The function also provides options for normalizing the matrix and customizing the plot appearance.

Output:

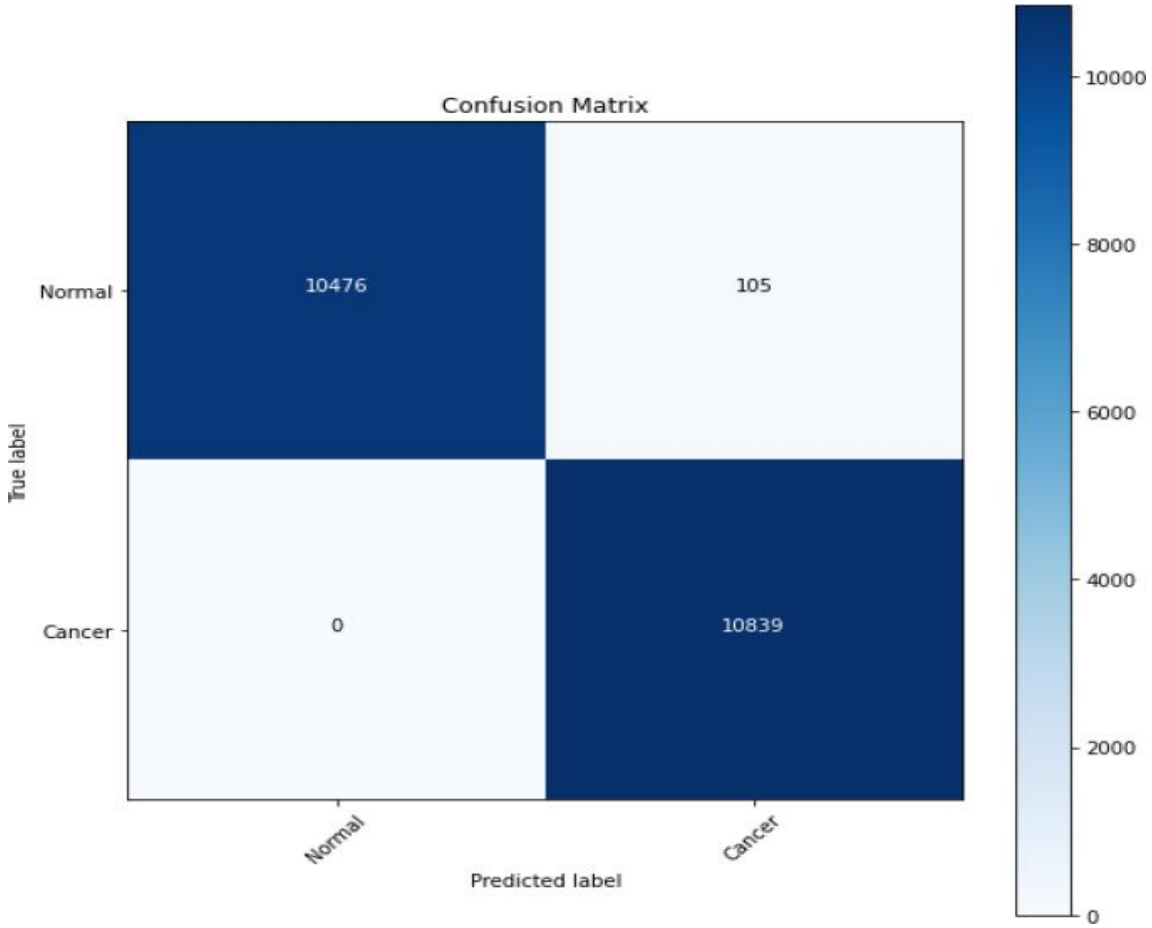

Figure 3.19 - Confusion Matrix Output

The function uses **plt.imshow()** to display the confusion matrix as an image with a colormap. It sets the title of the plot using **plt.title()** and adds a colorbar using **plt.colorbar()**. The x-axis and y-axis tick labels are set using **plt.xticks()** and **plt.yticks()** respectively.

If **normalize** is set to **True**, the confusion matrix is normalized by dividing each element by the sum of its corresponding row. This can be useful when comparing models with different sample sizes or class distributions.

The function then prints the confusion matrix, either with or without normalization, using **print()**. It also adds text annotations to the plot using **plt.text()**, indicating the counts in each cell of the matrix.

The function adjusts the layout of the plot using **plt.tight_layout()**, and sets the labels for the x-axis and y-axis using **plt.xlabel()** and **plt.ylabel()**.

```
def print_accuracy(confusion_matrix):
    correct_predictions = 0
    for i in range(len(confusion_matrix)):
        correct_predictions += confusion_matrix[i][i]

    print(f"accuracy = {correct_predictions/confusion_matrix.sum()*100}%")
```

Figure 3.20 - Accuracy

The **print_accuracy** function is used to calculate and print the accuracy based on a given confusion matrix. It takes the confusion matrix as input and performs the following steps:

Initializes a variable **correct_predictions** to keep track of the number of correct predictions.

Iterates over the rows and columns of the confusion matrix.

Adds the value in the diagonal of the matrix (corresponding to correct predictions) to the **correct_predictions** variable.

Calculates the accuracy by dividing the sum of correct predictions by the total number of predictions and multiplying by 100 to get the percentage.

Prints the accuracy value.

This function is useful for quickly evaluating the accuracy of a classification model based on the confusion matrix. The accuracy metric provides an overall measure of how well the model is performing in terms of correctly predicting the target classes.

Histopathology Model.

```
#Pandas is a popular library for data manipulation and analysis.
import pandas as pd
#NumPy is a popular library for numerical computing with Python.
import numpy as np
#Is a computer vision library that provides various tools for image and
video processing.
import cv2
#Provides tools for opening, manipulating, and saving many different images
file formats.
import PIL
#Pyplot provides tools for creating visualizations, such as graphs and
charts.
import matplotlib.pyplot as plt
#Plotly is a library for creating interactive visualizations.
import plotly.express as px
#Seaborn provides a high-level interface for creating statistical graphics.
import seaborn as sns
#Provides a function for finding all the pathnames matching a specified
pattern according to the rules used by the Unix shell.
import glob
#Provides functions for generating random numbers.
import random
#Provides a way of using operating system dependent functionality like
reading or writing to the file system.
import os
#Provides a way of listing all files and directories in a directory.
from os import listdir
#This sets the seed value for the random number generator to 100. This
ensures that the same sequence of random numbers is generated every time
the program is run.
random.seed(100)
#This sets the seed value for the NumPy random number generator to 100.
This ensures that the same sequence of random numbers is generated every
time the program is run.
np.random.seed(100)
```

Importing libraries:

`**pandas**` is imported and assigned the alias `pd`. Pandas is a popular library for data manipulation and analysis.

`**numpy**` is imported and assigned the alias `np`. NumPy is a library for numerical computing with Python.

`**cv2**` is imported. It is a computer vision library that provides various tools for image and video processing.

`**PIL**` is imported. It provides tools for opening, manipulating, and saving many different images file formats.

`**matplotlib.pyplot**` is imported and assigned the alias `plt`. It provides tools for creating visualizations, such as graphs and charts.

`**plotly.express**` is imported and assigned the alias `px`. Plotly is a library for creating interactive visualizations.

`**seaborn**` is imported and assigned the alias `sns`. It provides a high-level interface for creating statistical graphics.

`**glob**` is imported. It provides a function for finding all the pathnames matching a specified pattern according to the rules used by the Unix shell.

`**random**` is imported. It provides functions for generating random numbers.

`**os**` is imported. It provides a way of using operating system dependent functionality like reading or writing to the file system.

`**listdir**` is imported from `os`. It provides a way of listing all files and directories in a directory.

Setting seed values:

- `random.seed(100)` sets the seed value for the random number generator from the `random` module to 100. This ensures that the same sequence of random numbers is generated every time the program is run using the `random` module.

- `np.random.seed(100)` sets the seed value for the NumPy random number generator to 100. This ensures that the same sequence of random numbers is generated every time the program is run using NumPy's random functions.

```
#"glob.glob": This is a function from the glob module that returns a list
of pathnames that match a specified pattern.
#"recursive=True": This argument specifies that glob.glob should search for
files in all subdirectories of the specified directory, not just the top-
level directory.
breast_img = glob.glob('C:/breast histopathology
images/IDC_regular_ps50_idx5/**/*.png', recursive = True)
```

`**glob.glob**`: This is a function from the `glob` module that returns a list of pathnames that match a specified pattern. It takes a pattern as an argument and searches for files or directories that match that pattern.

- `**'C:/breast histopathology images/IDC_regular_ps50_idx5/**/*.png'**`: This is the pattern passed to `glob.glob`. In this case, it's looking for files with a `.png` extension (PNG image files) in the directory `'C:/breast histopathology images/IDC_regular_ps50_idx5/'` and all its subdirectories.

`**recursive=True**`: This argument is passed to `glob.glob` to specify that the search should be performed recursively in all subdirectories. By setting `recursive=True`, `glob.glob` will search for files in both the top-level directory and its subdirectories.

`**breast_img**`: This variable is assigned the result of the `glob.glob` function, which is a list of pathnames that match the specified pattern. Each pathname is a string representing the path to a PNG image file that was found in the directory and its subdirectories.

Overall, the code is searching for PNG image files in the directory `'C:/breast histopathology images/IDC_regular_ps50_idx5/'` and its subdirectories, and storing the list of matching file pathnames in the `breast_img` variable for further processing or analysis.

```
#This is a Python code that prints the first three elements in the
"breast_img" list.
for imgname in breast_img[:3]:
      print(imgname)
```

This code uses a loop to iterate over the first three elements of the `**breast_img**` list. For each element, it prints the value of `imgname`.

- `for imgname in breast_img[:3]:`: This line sets up a loop that iterates over the first three elements of the `breast_img` list. The loop variable `imgname` is assigned each element's value in sequence.

`**print(imgname)**`: This line prints the value of `imgname` during each iteration of the loop. It will display the pathname of each image file that matched the pattern and was stored in the `breast_img` list.

By executing this code, you will see the pathnames of the first three image files in the `breast_img` list printed to the console.

```
#This is a Python code that sorts the images in the "breast_img" list into
two separate lists based on their file names.
#These are the final lists containing the pathnames of all non-cancerous
and cancerous images in the "breast_img" list, respectively.
non_can_img = []
can_img = []

for img in breast_img:
#Conditional statement that check the fifth character from the end of the
file name to determine whether an image is cancerous or non-cancerous.
    if img[-5] == '0' :
#These statements append the current image file pathname to the appropriate
list.
        non_can_img.append(img)

    elif img[-5] == '1' :
        can_img.append(img)
```

`non_can_img = []` and `can_img = []`: These lines initialize two empty lists, **`non_can_img` and `can_img`**, which will be used to store the pathnames of non-cancerous and cancerous images, respectively.

`for img in breast_img`:`: This line sets up a loop that iterates over each element, `img`, in the `breast_img` list. Each element represents the pathname of an image file.

`if img[-5] == '0':` and `elif img[-5] == '1':`: These lines check the fifth character from the end of the image file's name to determine whether it is cancerous or non-cancerous. If the fifth character from the end is '0', it indicates a non-cancerous image. If it is '1', it indicates a cancerous image.

`non_can_img.append(img)` and **`can_img.append(img)`:** These statements append the current image file pathname, `img`, to the appropriate list (`non_can_img` for non-cancerous images or `can_img` for cancerous images) based on the conditional check.

By executing this code, the image files in the `breast_img` list will be categorized into `non_can_img` and `can_img` lists based on their filenames, with non-cancerous images stored in `non_can_img` and cancerous images stored in `can_img`.

```
#This is a Python code that calculates and prints the total number of non
cancerous and cancerous images in the "breast_img" list.

#These are two separate statements that use the "len()" function to
determine the number of elements in the "non_can_img" and "can_img" lists,
respectively.
non_can_num = len(non_can_img) # No cancer
can_num = len(can_img)   # Cancer

total_img_num = non_can_num + can_num

print('Number of Images of no cancer: {}' .format(non_can_num))    # images
of Non cancer
print('Number of Images of cancer : {}' .format(can_num))    # images of
cancer
print('Total Number of Images : {}' .format(total_img_num))
```

`non_can_num = len(non_can_img)`: This line calculates the number of elements (image pathnames) in the `non_can_img` list, which represents the non-cancerous images. The `len()` function returns the length of a list, which corresponds to the number of elements in that list. The result is assigned to the variable `non_can_num`.

`can_num = len(can_img)`: This line calculates the number of elements (image pathnames) in the `can_img` list, which represents the cancerous

images. The `len()` function is used to determine the length of the `can_img` list, and the result is assigned to the variable `can_num`.

`**total_img_num = non_can_num + can_num**`: This line calculates the total number of images by summing the counts of non-cancerous and cancerous images.

`**print('Number of Images of no cancer: {}'.format(non_can_num))**`: This line prints the number of non-cancerous images. It uses string formatting with the `{}` placeholder to insert the value of `non_can_num` into the printed string.

`**print('Number of Images of cancer: {}'.format(can_num))**`: This line prints the number of cancerous images. It uses string formatting to insert the value of `can_num` into the printed string

`**print('Total Number of Images: {}'.format(total_img_num))**`: This line prints the total number of images (both non-cancerous and cancerous). It uses string formatting to insert the value of `total_img_num` into the printed string.

	state of cancer	Numbers of Patients
0	0	198738
1	1	78786

Figure 3.21 - Data Stats

Output:

Number of Images of no cancer: 198738

Number of Images of cancer : 78786

Total Number of Images : 277524

```
#"px.bar()": This is a function from the plotly.express library that
creates a bar chart.
#"data_frame=data_insight_1": This argument specifies the Pandas DataFrame
that contains the data to be plotted.
#"x='state of cancer'": This argument specifies the column name in the
DataFrame to be plotted on the x-axis of the bar chart.
#"y='Numbers of Patients'": This argument specifies the column name in the
DataFrame to be plotted on the y-axis of the bar chart.
bar = px.bar(data_frame=data_insight_1, x = 'state of cancer', y='Numbers
of Patients', color='state of cancer')
#This statement modifies the layout of the bar chart by adding a title and
centering it.
bar.update_layout(title_text='Number of Patients with cancer (1) and
patients with no cancer (0)', title_x=0.5)
#This statement displays the bar chart in the output.
bar.show()
```

bar = px.bar(data_frame=data_insight_1, x='state of cancer', y='Numbers of Patients', color='state of cancer')

This line creates a bar chart using the `px.bar()` function. The `data_frame` argument specifies the DataFrame `data_insight_1` that contains the data to be plotted. The `x` argument specifies the column name 'state of cancer' to be plotted on the x-axis of the bar chart. The `y` argument specifies the column name 'Numbers of Patients' to be plotted on the y-axis of the bar chart. The

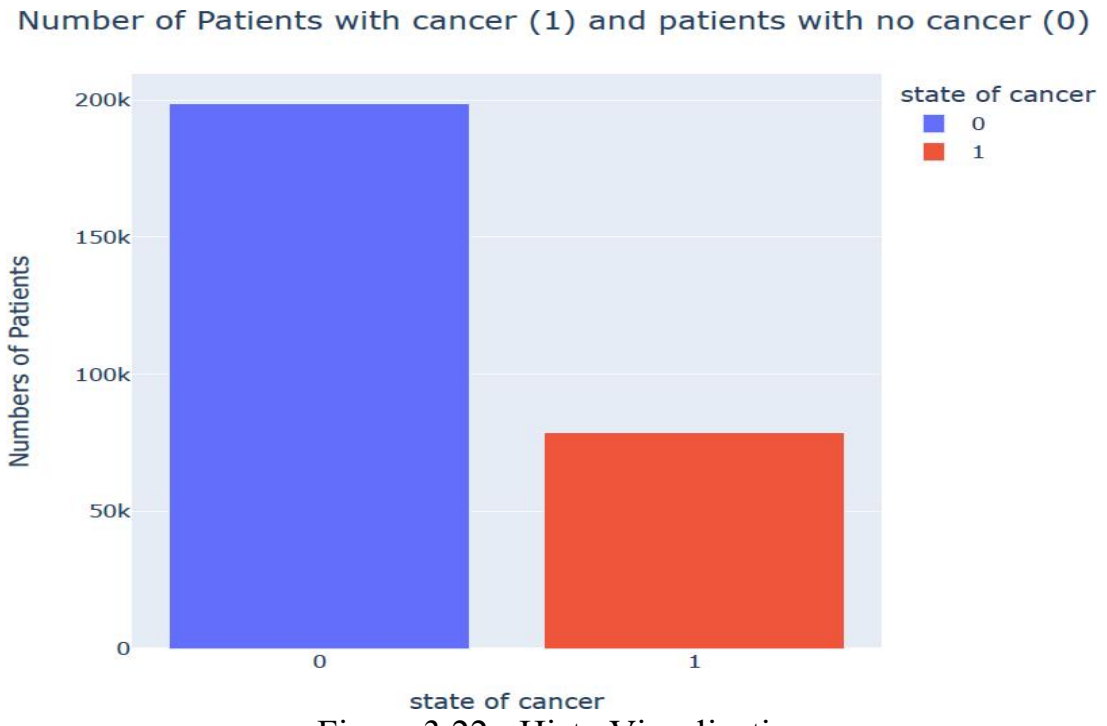

Figure 3.22 - Histo Visualization

`color` argument specifies that the bars should be colored based on the 'state of cancer' column values.

bar.update_layout(title_text='Number of Patients with cancer (1) and patients with no cancer (0)', title_x=0.5)

This line modifies the layout of the bar chart by adding a title. The `title_text` argument specifies the title of the chart as 'Number of Patients with cancer (1) and patients with no cancer (0)'. The `title_x` argument centers the title horizontally by setting it to 0.5.

```
import keras.utils as image
#This statement creates a new matplotlib figure object with a size of 15x15
inches.
plt.figure(figsize = (15, 15))
#This statement generates 18 random integers between 0 and the length of
the list.
some_non = np.random.randint(0, len(non_can_img), 18)
some_can = np.random.randint(0, len(can_img), 18)

s = 0
#"for" loop iterates over the randomly selected non-cancer images, loads
each image.
for num in some_non:

        img = image.load_img((non_can_img[num]), target_size=(100, 100))
#Converts it to a NumPy array.
        img = image.img_to_array(img)
        plt.subplot(6, 6, 2*s+1)
        plt.axis('off')
        plt.title('no cancer')
        plt.imshow(img.astype('uint8'))
#The "s" variable is used to calculate the index of the current
subplot.
        s += 1

s = 1
for num in some_can:
        img = image.load_img((can_img[num]), target_size=(100, 100))
        img = image.img_to_array(img)
        plt.subplot(6, 6, 2*s)
        plt.axis('off')
        plt.title('cancer')
        plt.imshow(img.astype('uint8'))
        s += 1
```

This is a Python code that uses the **matplotlib.pyplot** library to plot a set of randomly selected images from the breast cancer dataset.

import keras.utils as image: This line imports the **keras.utils** module and assigns it the name **image**.

plt.figure(figsize = (15, 15)): This statement creates a new matplotlib figure object with a size of 15x15 inches.

some_non = np.random.randint(0, len(non_can_img), 18): This statement generates 18 random integers between 0 and the length of the list **non_can_img** (which contains the file paths for the non-cancer images).

some_can = np.random.randint(0, len(can_img), 18): This statement generates 18 random integers between 0 and the length of the list **can_img** (which contains the file paths for the cancer images).

s = 0: This initializes the variable **s** to zero.

The next **for** loop iterates over the randomly selected non-cancer images, loads each image using the **keras.utils.image.load_img()** function, converts it to a NumPy array using **keras.utils.image.img_to_array()**, and then plots it using **matplotlib.pyplot.imshow()**. The **s** variable is used to calculate the index of the current subplot.

s = 1: This initializes the variable **s** to one.

The next **for** loop iterates over the randomly selected cancer images, loads each image using the **keras.utils.image.load_img()** function, converts it to a NumPy array using **keras.utils.image.img_to_array()**, and then plots it using **matplotlib.pyplot.imshow()**. The **s** variable is used to calculate the index of the current subplot.

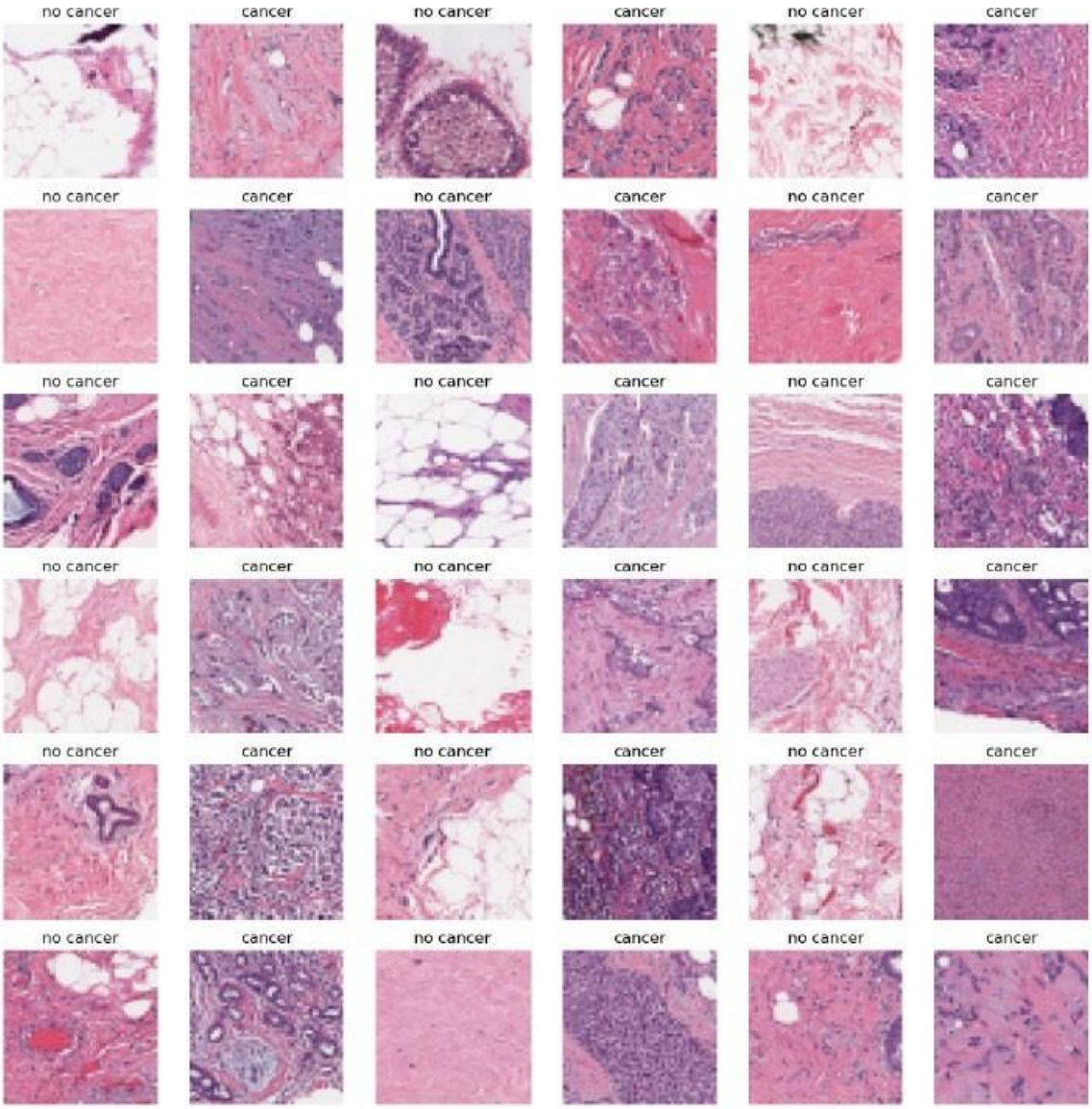

Figure 3.23 - Histo Grid

When executed, this code will display a 6x6 grid of randomly selected breast cancer images, with each row containing one non-cancer image and one cancer image.

```
from matplotlib.image import imread
import cv2

some_non_img = random.sample(non_can_img, len(non_can_img))
some_can_img = random.sample(can_img, len(can_img))

non_img_arr = []
can_img_arr = []

for img in some_non_img:

    n_img = cv2.imread(img, cv2.IMREAD_COLOR)
#The "cv2.resize" function is then used to resize each image to a smaller
size of 50 x 50 pixels with linear interpolation.
    n_img_size = cv2.resize(n_img, (50, 50), interpolation =
cv2.INTER_LINEAR)
#Resized images are then appended to the "non_img_arr" and "can_img_arr"
lists, along with a label of 0 for non-cancerous images and 1 for cancerous
images, respectively.
    non_img_arr.append([n_img_size, 0])

for img in some_can_img:

    c_img = cv2.imread(img, cv2.IMREAD_COLOR)
    c_img_size = cv2.resize(c_img, (50, 50), interpolation =
cv2.INTER_LINEAR)
    can_img_arr.append([c_img_size, 1])
```

In this code, **matplotlib.image.imread** and **cv2.imread** are used to read and load image files from the file paths specified in **some_non_img** and **some_can_img**, which contain a random sampling of non-cancerous and cancerous images, respectively.

The **cv2.resize** function is then used to resize each image to a smaller size of 50 x 50 pixels with linear interpolation. The resized images are then appended to the **non_img_arr** and **can_img_arr** lists, along with a label of **0** for non-cancerous images and **1** for cancerous images, respectively.

This process results in two lists of images and their corresponding labels, which can be used as training data for a machine learning model to classify breast cancer images.

```
X = []
y = []
# Combine the feature arrays of non-cancerous images (non_img_arr) and
cancerous images (can_img_arr)
# Into a single array called breast_img_arr. Shuffle the combined array
randomly.
breast_img_arr = np.concatenate((non_img_arr, can_img_arr))
random.shuffle(breast_img_arr)

# Iterate over each feature-label pair in the breast_img_arr.
# Append the feature to the list X and the label to the list y.
for feature, label in breast_img_arr:
    X.append(feature)
    y.append(label)

# Convert the X and y lists into numpy arrays.
X = np.array(X)
y = np.array(y)

# Print the shape of the X array.
print('X shape: {}'.format(X.shape))
```

In this code, the **non_img_arr** and **can_img_arr** lists created in the previous code block are concatenated into a single **breast_img_arr** list using the **numpy.concatenate()** function. This list contains all of the resized breast cancer images along with their corresponding labels.

The **random.shuffle()** function is then used to shuffle the order of the images in the **breast_img_arr** list.

Next, two empty lists **X** and **y** are created to store the feature data (the resized images) and label data (the cancer/non-cancer label) respectively.

A loop iterates over every image and corresponding label in **breast_img_arr** and appends the image data to the **X** list and the label to the **y** list.

Finally, the **X** and **y** lists are converted to NumPy arrays using the **numpy.array()** function and their shapes are printed to the console to confirm that the data has been properly processed.

output:

X shape : (277524, 50, 50, 3)

```
from sklearn.model_selection import train_test_split
from keras.utils.np_utils import to_categorical

# Split the data into training and testing sets using train_test_split
function.
# The test set will be 20% of the total data, and the random state is set
to 42 for reproducibility.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Set the rate for categorical conversion to 0.5.
rate = 0.5

# Calculate the number of samples to be used for training based on the
rate.
num = int(X.shape[0] * rate)

# Convert the categorical labels (y_train and y_test) into one-hot encoded
vectors using to_categorical.
y_train = to_categorical(y_train, 2)
y_test = to_categorical(y_test, 2)

# Print the shapes of the training and testing data.
print('X_train shape: {}'.format(X_train.shape))
print('X_test shape: {}'.format(X_test.shape))
print('y_train shape: {}'.format(y_train.shape))
print('y_test shape: {}'.format(y_test.shape))
```

This code is splitting the dataset into training and testing sets. It imports the **train_test_split** function from **sklearn.model_selection** and **to_categorical** function from **keras.utils.np_utils**.

The **train_test_split** function splits the dataset into two parts, one for training the model and the other for testing it. It takes **X** and **y** as input, where **X** is the array of input data (the images), and **y** is the array of labels (0 for no cancer and 1 for cancer). It also takes **test_size** which specifies the size of the test set as a proportion of the total dataset, and **random_state** which sets the seed used by the random number generator to ensure reproducibility.

The **to_categorical** function is used to convert the labels from integers to a binary class matrix for training. This is necessary for the model to be able to learn the difference between the two classes. The **num_classes** argument is set to 2 since there are two classes in the dataset (0 and 1).

The **rate** and **num** variables are used to reduce the size of the training data for faster training. **rate** specifies the proportion of the dataset to use, and **num** is the number of samples to use.

Finally, the code prints the shapes of the training and testing data arrays **X_train**, **X_test**, **y_train**, and **y_test**.

output:

X_train shape : (222019, 50, 50, 3)

X_test shape : (55505, 50, 50, 3)

y_train shape : (222019, 2)

y_test shape : (55505, 2)

```
import tensorflow as tf
#The line tf.random.set_seed(100) sets the random seed for TensorFlow
tf.random.set_seed(100)
import tensorflow as tf tf.random.set_seed(100)
```

This code sets the random seed for the TensorFlow library to a fixed value of 100. Setting a random seed ensures that the results of the model will be reproducible. By setting the seed, the same sequence of random numbers will be generated every time the code is run.

```
model = tf.keras.Sequential([
    # Convolutional layer with 32 filters, filter size of 3x3, same
padding, ReLU activation, and input shape of (50, 50, 3) for RGB images.
    tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='relu',
input_shape=(50, 50, 3)),

    # Max pooling layer with a pool size of 2x2 and default stride of 2.
    tf.keras.layers.MaxPooling2D(strides=2),

    # Convolutional layer with 64 filters, filter size of 3x3, same
padding, and ReLU activation.
    tf.keras.layers.Conv2D(64, (3, 3), padding='same', activation='relu'),

    # Max pooling layer with a pool size of 3x3 and stride of 2.
    tf.keras.layers.MaxPooling2D((3, 3), strides=2),

    # Convolutional layer with 128 filters, filter size of 3x3, same
padding, and ReLU activation.
    tf.keras.layers.Conv2D(128, (3, 3), padding='same', activation='relu'),

    # Max pooling layer with a pool size of 3x3 and stride of 2.
    tf.keras.layers.MaxPooling2D((3, 3), strides=2),

    # Convolutional layer with 128 filters, filter size of 3x3, same
padding, and ReLU activation.
    tf.keras.layers.Conv2D(128, (3, 3), padding='same', activation='relu'),

    # Max pooling layer with a pool size of 3x3 and stride of 2.
    tf.keras.layers.MaxPooling2D((3, 3), strides=2),

    # Flatten layer to convert the 2D feature maps into a 1D vector.
    tf.keras.layers.Flatten(),

    # Fully connected (dense) layer with 128 units and ReLU activation.
    tf.keras.layers.Dense(128, activation='relu'),

    # Output layer with 2 units and softmax activation for binary
classification.
    tf.keras.layers.Dense(2, activation='softmax')
])
```

This is a convolutional neural network with four convolutional layers and two fully connected layers. The first convolutional layer has 32 filters of size (3,3), followed by a max pooling layer with stride 2. The second convolutional layer has 64 filters of size (3,3), followed by a max pooling layer with stride 2. The third convolutional layer has 128 filters of size (3,3), followed by a max

pooling layer with stride 2. The fourth convolutional layer also has 128 filters of size (3,3), followed by a max pooling layer with stride 2. After the convolutional layers, there is a flatten layer to convert the output to a one-dimensional vector, followed by a fully connected layer with 128 neurons and ReLU activation. The final layer is a softmax layer with two neurons, representing the two classes of the problem (cancer and non-cancer).

tf.keras.layers.Conv2D is a 2D convolution layer in Keras, used for processing spatial data. It applies a set of filters to the input image and produces a set of feature maps. The parameters of the layer include the number of filters, filter/kernel size, stride, padding, and activation function.

MaxPooling2D is a type of pooling layer in convolutional neural networks (CNNs) that reduces the spatial size (width and height) of the input while keeping the number of channels (depth) constant. It works by dividing the input image into a set of non-overlapping rectangles and taking the maximum value of each rectangle, which is then used as the output for that region.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 50, 50, 32)        896

 max_pooling2d (MaxPooling2D  (None, 25, 25, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 25, 25, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 5, 5, 128)        0
 2D)

 conv2d_3 (Conv2D)           (None, 5, 5, 128)         147584

 max_pooling2d_3 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 128)               65664

 dense_1 (Dense)             (None, 2)                 258

=================================================================
Total params: 306,754
Trainable params: 306,754
```

Figure 3.24 - Histo Summery

The model has 4 convolutional layers with 32, 64, 128, and 128 filters respectively. Each convolutional layer is followed by a max pooling layer. After the convolutional layers, there is a flatten layer to convert the 2D output from the last convolutional layer to a 1D feature vector. Finally, there are two fully connected layers, one with 128 units and another with 2 units, corresponding to the two possible classes in the dataset (cancer and no cancer). The model has a total of 307,754 trainable parameters.

```
# The optimizer is responsible for updating the weights of the model during
training in order to minimize the loss function.
The Adam optimizer is used with a learning rate of 0.0001.
#loss='binary_crossentropy': This line specifies the loss function to be
used during training. Binary cross-entropy is commonly used for binary
classification tasks, where the model predicts one of two classes.
#metrics=['accuracy']: This line specifies the metrics to be computed
during training and evaluation.
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),loss
='binary_crossentropy',metrics=['accuracy'])
```

now the model is compiled and ready to be trained using the **compile()** method. The chosen optimizer is Adam with a learning rate of 0.0001, and the loss function is binary cross-entropy since this is a binary classification problem. The accuracy will also be monitored during training.

```
# Train the model on the training data (X_train) and corresponding labels
(y_train).
# The validation data will be used to evaluate the model's performance
after each epoch.
# The training will be performed for 25 epochs.
# The batch size determines the number of samples processed before the
model's weights are updated.
history = model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=25, batch_size=75)
```

The code is to train the CNN model for 25 epochs with a batch size of 75, using the training set **X_train** and **y_train**, while evaluating the performance on the validation set **X_test** and **y_test**. The model is compiled with the Adam optimizer, a learning rate of 0.0001, and binary cross-entropy as the loss function. The training history is stored in the **history** variable.

```
# Evaluate the trained model on the test data (X_test) and corresponding
labels (y_test).
model.evaluate(X_test, y_test)
```

The **model.evaluate()** function returns the loss value and metrics values for the model evaluated on a given dataset. Here's an example of how you can use it:

output:

1735/1735 [==============================] - 34s 19ms/step - loss: 0.2114 - accuracy: 0.9638

[0.21135878562927246, 0.9638410806655884]

```
# Import the necessary library for computing the confusion matrix.
from sklearn.metrics import confusion_matrix

# Predict the labels for the test data using the trained model.
Y_pred = model.predict(X_test)

# Convert the predicted probabilities to class labels by selecting the
class with the highest probability.
Y_pred_classes = np.argmax(Y_pred, axis=1)

# Convert the true labels to class labels.
Y_true = np.argmax(y_test, axis=1)

# Compute the confusion matrix using the true labels and predicted classes.
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)

# Create a figure and axis for plotting the heatmap.
f, ax = plt.subplots(figsize=(8, 8))

# Plot the confusion matrix as a heatmap, with annotations and specific
visual properties.
sns.heatmap(confusion_mtx, annot=True, linewidths=0.01, cmap="BuPu",
linecolor="gray", fmt='.1f', ax=ax)

# Set the label for the x-axis.
plt.xlabel("Predicted Label")

# Set the label for the y-axis.
plt.ylabel("True Label")

# Set the title of the plot.
plt.title("Confusion Matrix")
# Display the plot.
plt.show()
```

Y_pred: The predicted labels for the test set obtained using the trained model.

Y_pred_classes: The class labels for the predicted labels obtained using **np.argmax** function.

Y_true: The true class labels for the test set obtained in the preprocessing step.

confusion_mtx: The confusion matrix obtained using **confusion_matrix** function from **sklearn.metrics**.

f,ax = plt.subplots(figsize=(8, 8)): Creating a subplot for displaying the heatmap of the confusion matrix.

sns.heatmap: Plotting the heatmap of the confusion matrix using seabornlibrary.

plt.xlabel, **plt.ylabel**, **plt.title**: Adding labels and title to the plot.

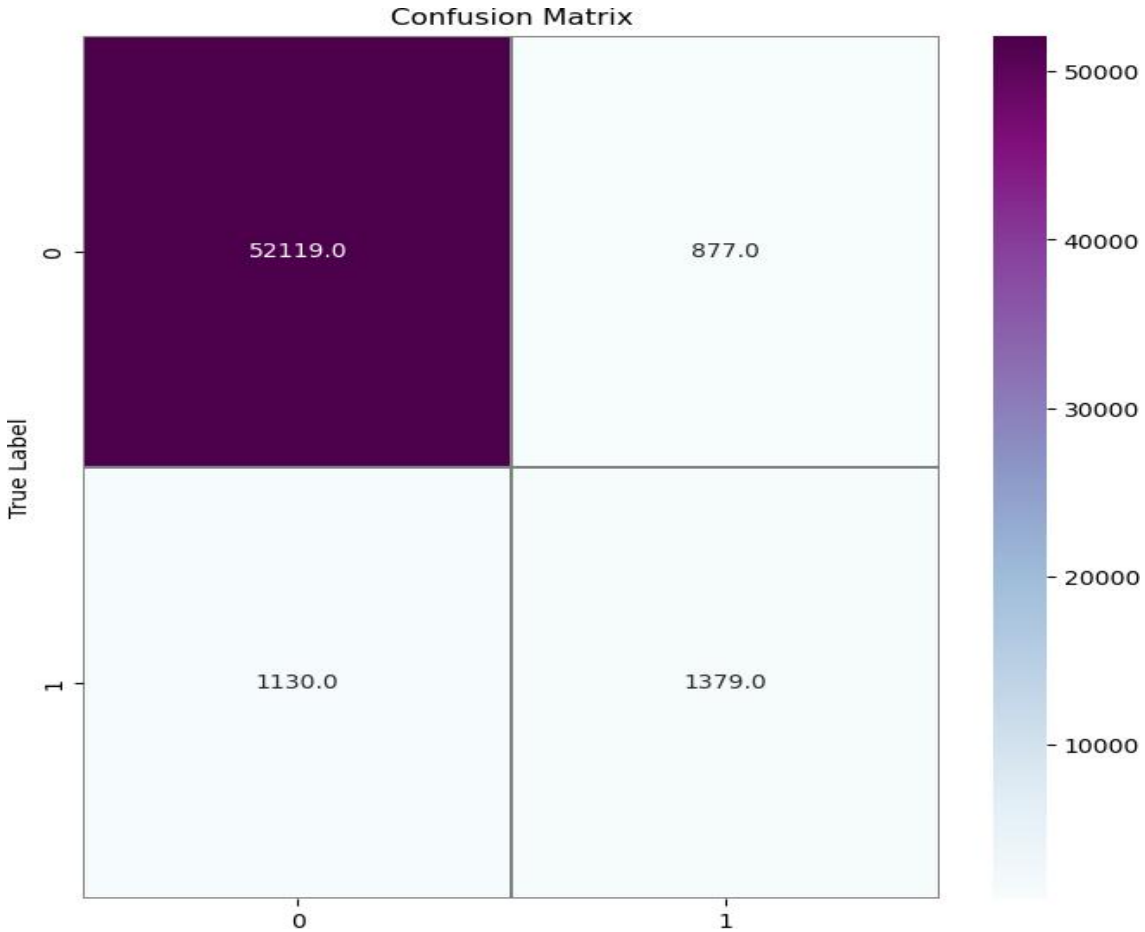

Figure 3.25 - Histo Confusion Matrix

confusion matrix based on the predictions made by the trained model on the test dataset. The confusion matrix shows the number of **true positive (TP), true negative (TN), false positive (FP),** and **false negative (FN)** predictions.

that the **confusion_matrix** function returns a 2x2 array for binary classification, where the rows represent the true class labels and the columns represent the predicted class labels.

```
# Plot the training accuracy values over the epochs.
plt.plot(history.history['accuracy'])

# Plot the validation accuracy values over the epochs.
plt.plot(history.history['val_accuracy'])

# Set the title of the plot.
plt.title('Model Accuracy')

# Set the label for the y-axis.
plt.ylabel('accuracy')

# Set the label for the x-axis.
plt.xlabel('epoch')

# Create a legend for the plot, with labels 'train' and 'test', located in
the upper left corner.
plt.legend(['train', 'test'], loc='upper left')

# Display the plot.
plt.show()
```

plt.plot(history.history['accuracy']): This line plots the training accuracy values. It assumes that you have a history object that contains the training history of your model, and you are accessing the 'accuracy' values from it.

plt.plot(history.history['val_accuracy']): This line plots the validation accuracy values. It assumes that the history object contains the validation accuracy values under the key 'val_accuracy'.

plt.title('Model Accuracy'): Sets the title of the plot as 'Model Accuracy'.

plt.ylabel('accuracy'): Sets the label for the y-axis as 'accuracy'.

plt.xlabel('epoch'): Sets the label for the x-axis as 'epoch'.

plt.legend(['train', 'test'], loc='upper left'): Creates a legend for the plot with

labels 'train' and 'test'. The legend is located in the upper left corner of the plot.

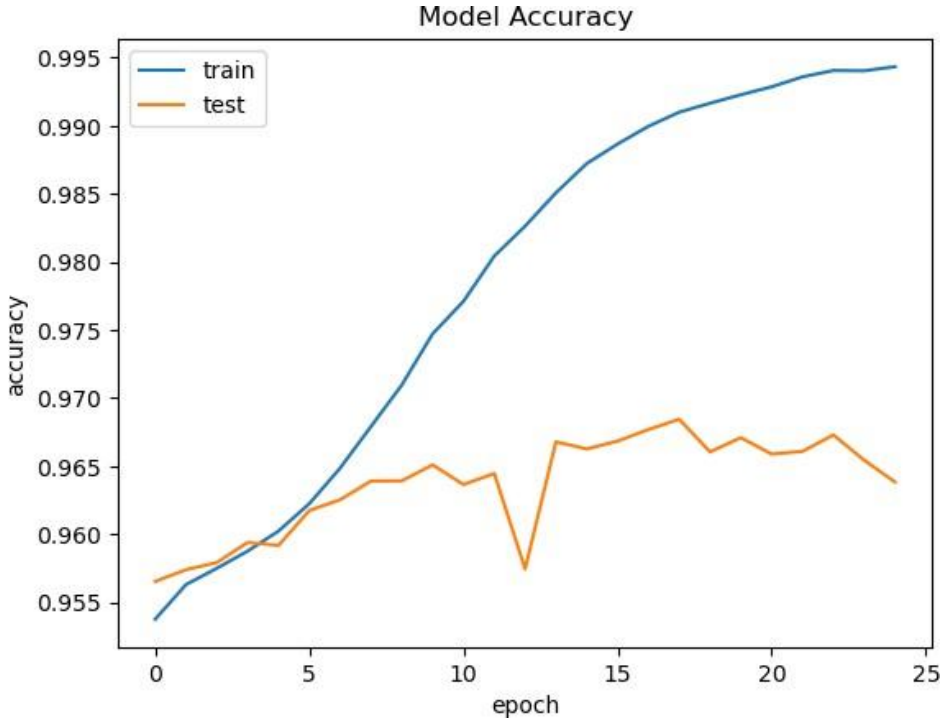

Figure 3.26 - Histo Accuracy

plt.show(): Displays the plot on the screen.

plots the training accuracy values and validation accuracy values over the epochs.

```
# Plot the training loss values over the epochs.
plt.plot(history.history['loss'])

# Plot the validation loss values over the epochs.
plt.plot(history.history['val_loss'])

# Set the title of the plot.
plt.title('Model Loss')

# Set the label for the y-axis.
plt.ylabel('loss')

# Set the label for the x-axis.
plt.xlabel('epoch')

# Create a legend for the plot, with labels 'train' and 'test', located in
the upper left corner.
plt.legend(['train', 'test'], loc='upper left')
# Display the plot.
plt.show()
```

plt.show(): Displays the plot on the screen.

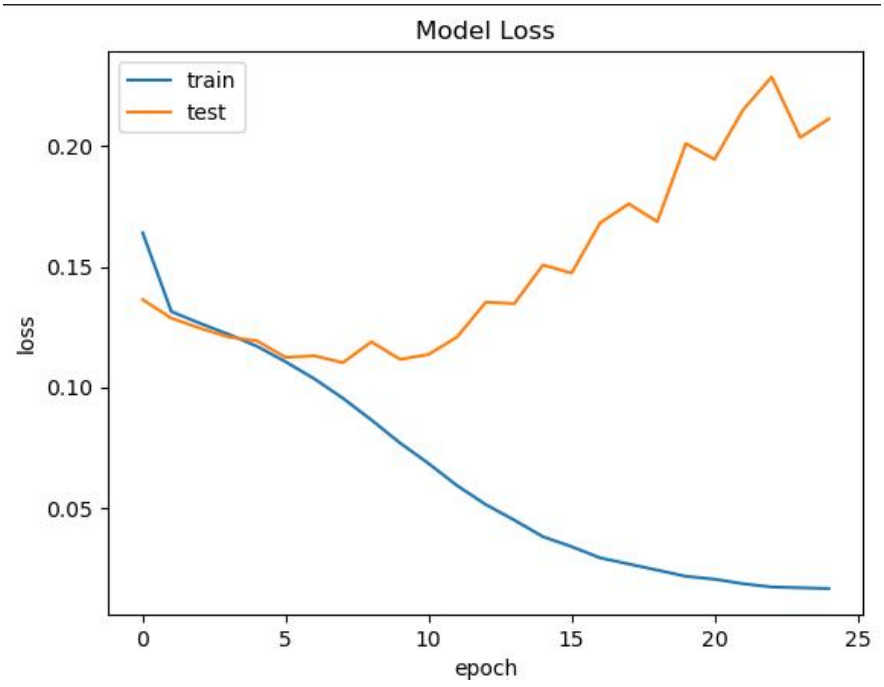

Figure 3.27 - Histo Loss

plots the training loss values and validation loss values over the epochs.

```
def img_plot(arr, index=0):

    # Set the title of the plot as 'Test Image'.
    plt.title('Test Image')

    # Display the image at the specified index from the array.
    plt.imshow(arr[index])

# Set the value of the index variable to 804.
index = 804
# Call the img_plot function, passing X_test as the array and index as the
index parameter.
img_plot(X_test, index)
```

This code defines a function named **img_plot** that takes an array (arr) as a parameter and an optional index parameter with a default value of 0. The purpose of this function is to plot an image from the array.

plt.title('Test Image') sets the title of the plot as 'Test Image'.

plt.imshow(arr[index]) displays the image at the specified index from the array (arr). The **imshow()** function is used to visualize the image.

img_plot(X_test, index) calls the img_plot function, passing X_test as the array and index as the index parameter.

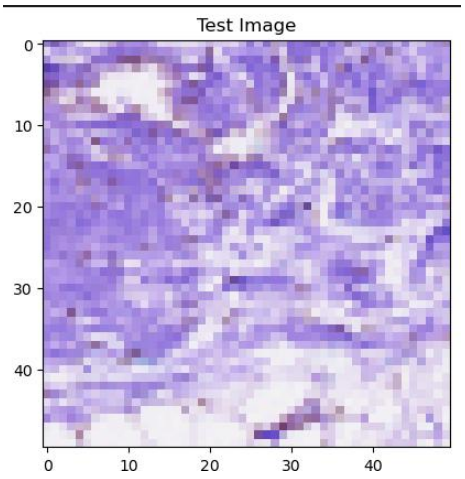

Figure 3.28 - Histo Test Image

the code defines a function img_plot that can plot images from an array. It then sets the value of the index variable to 804 and calls the img_plot function, passing in the X_test array and the index value. This results in the display of the image at index 804 from the X_test array with the title 'Test Image'.

```
# Create a new variable named "input" and assign it a slice of the X_test
array.
# The slice includes only the element at the specified index and is of
shape (1, width, height, channels).
input = X_test[index:index+1]
pred = model.predict(input)[0].argmax()
label = y_test[index].argmax()
```

input = X_test[index:index+1]: This line creates a new variable named "input" and assigns it a slice of the X_test array. The slice includes only the element at the specified index and has a shape of (1, width, height, channels). This is done to prepare the input data for the model prediction.

model.predict() function takes the "input" array as input and returns an array of predicted probabilities for each class. Since the input array has a shape of (1, width, height, channels), we use [0] to select the predicted probabilities for the first (and only) image in the input. The argmax() function is then applied to determine the index of the class with the highest probability.

label = y_test[index].argmax(): This line extracts the label for the specified index from the y_test array. The y_test array typically contains the true labels for the corresponding images in the X_test array. The argmax() function is used to find the index of the class with a value of 1 (indicating the true label). This line retrieves the true label for the specified index.

```
print('Predicted Value using  cnn model',pred)

print("True Value",label)
```

Output:

Predicted Value using cnn model 0

True Value 0

Application.

Frontend.

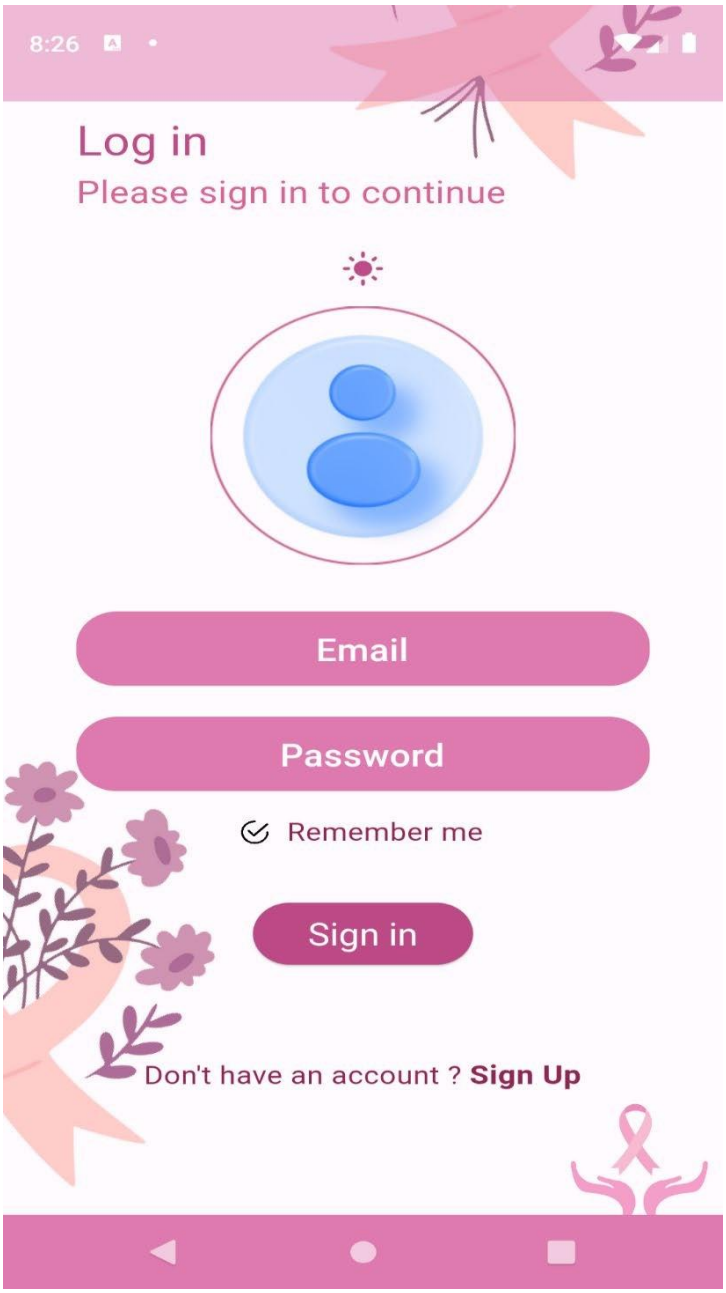

Figure 3.29 - Login Screen

It requires an email and password to login to the app.

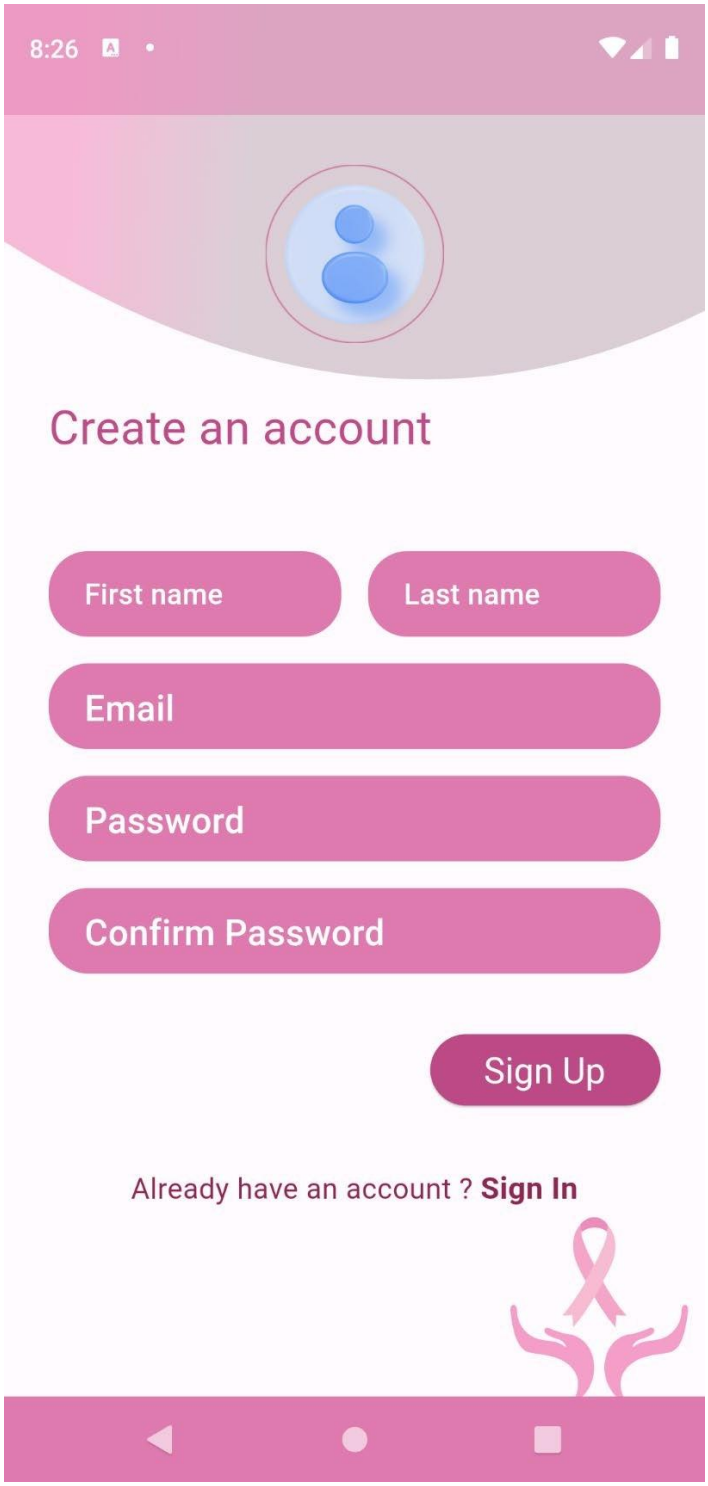

Figure 3.30 - First sign-up Screen

If the user does not have an account, he creates an account and it required First name, Last name, Email, and Password.

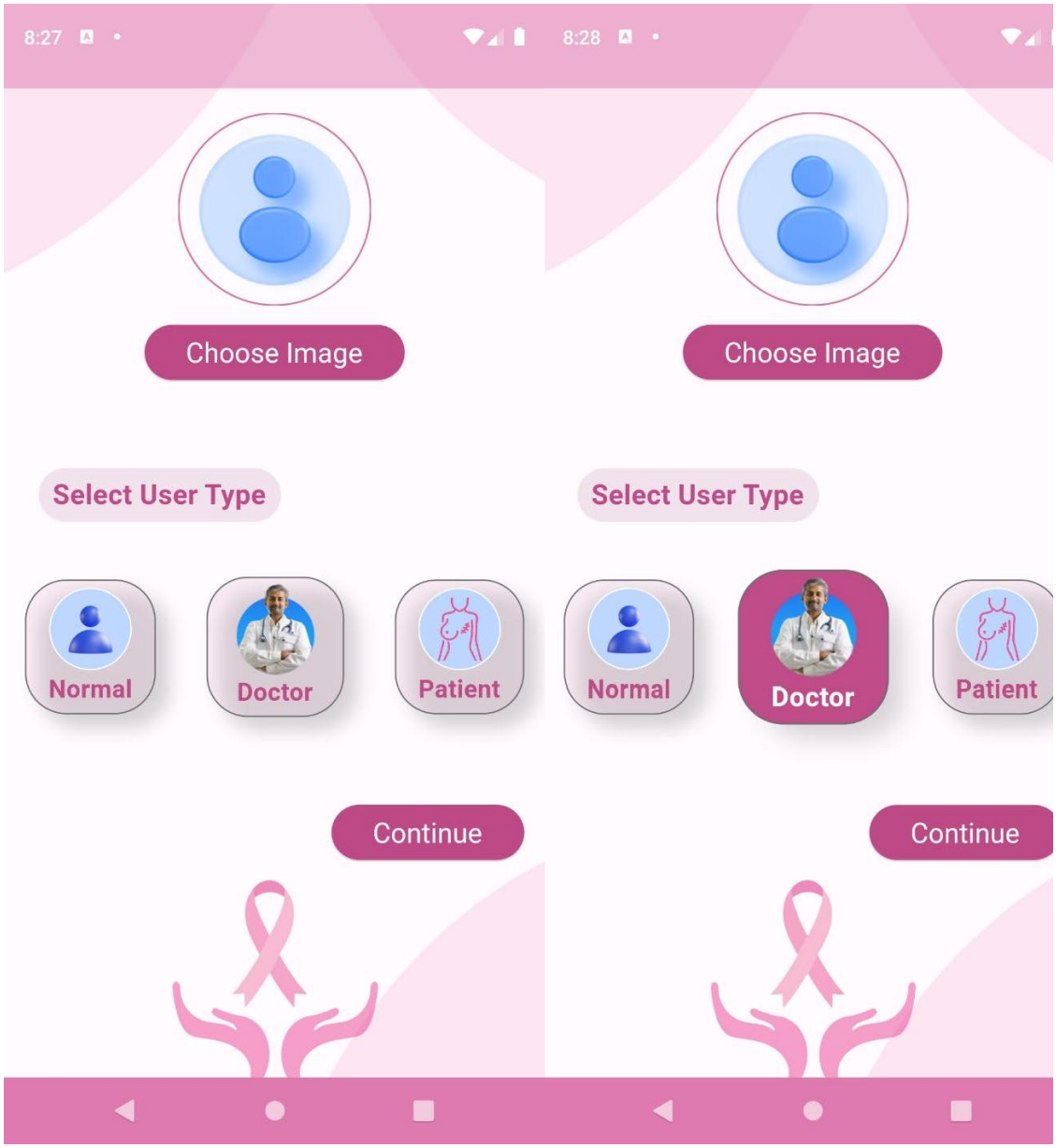

Figure 3.31 - Second sign-up Screen

Continuing to create an account on a second screen (optional), the user can choose an image of his, and select what kind of user he is if the user didn’t select the user type will be a "Normal user" as a default.

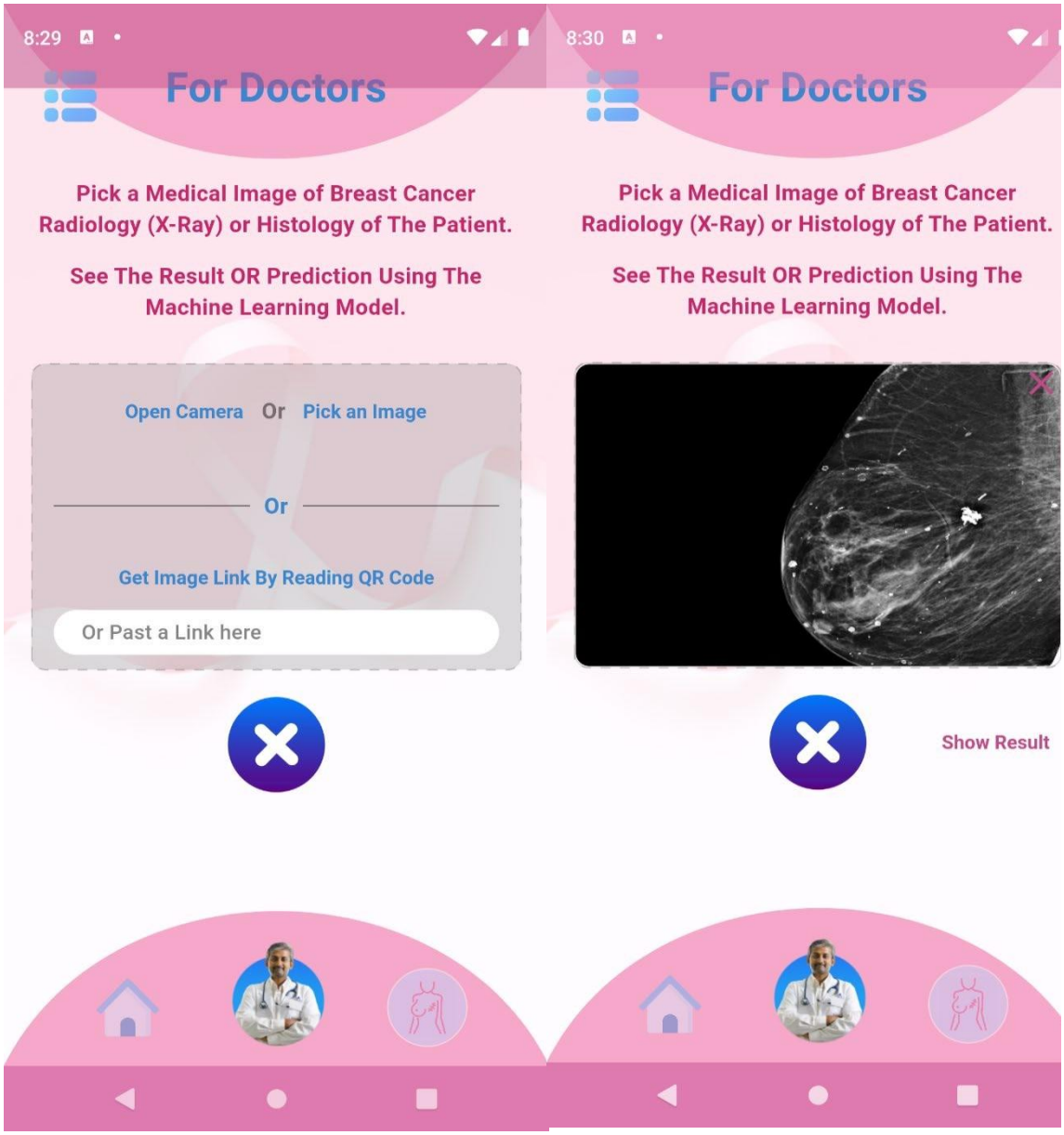

Figure 3.32 - Doctor Screen

The user (Doctor or radiologist) can pick an image and there are four ways to do that:

- **Open the Camera** and take a picture of the medical image if he has it physically.
- **Pick an image from the device** if he has it in the file system.
- Get an image link by **reading the QR code** of the medical image if the image exists online.
- Alternatively, he can simply **paste the link** in the text field.

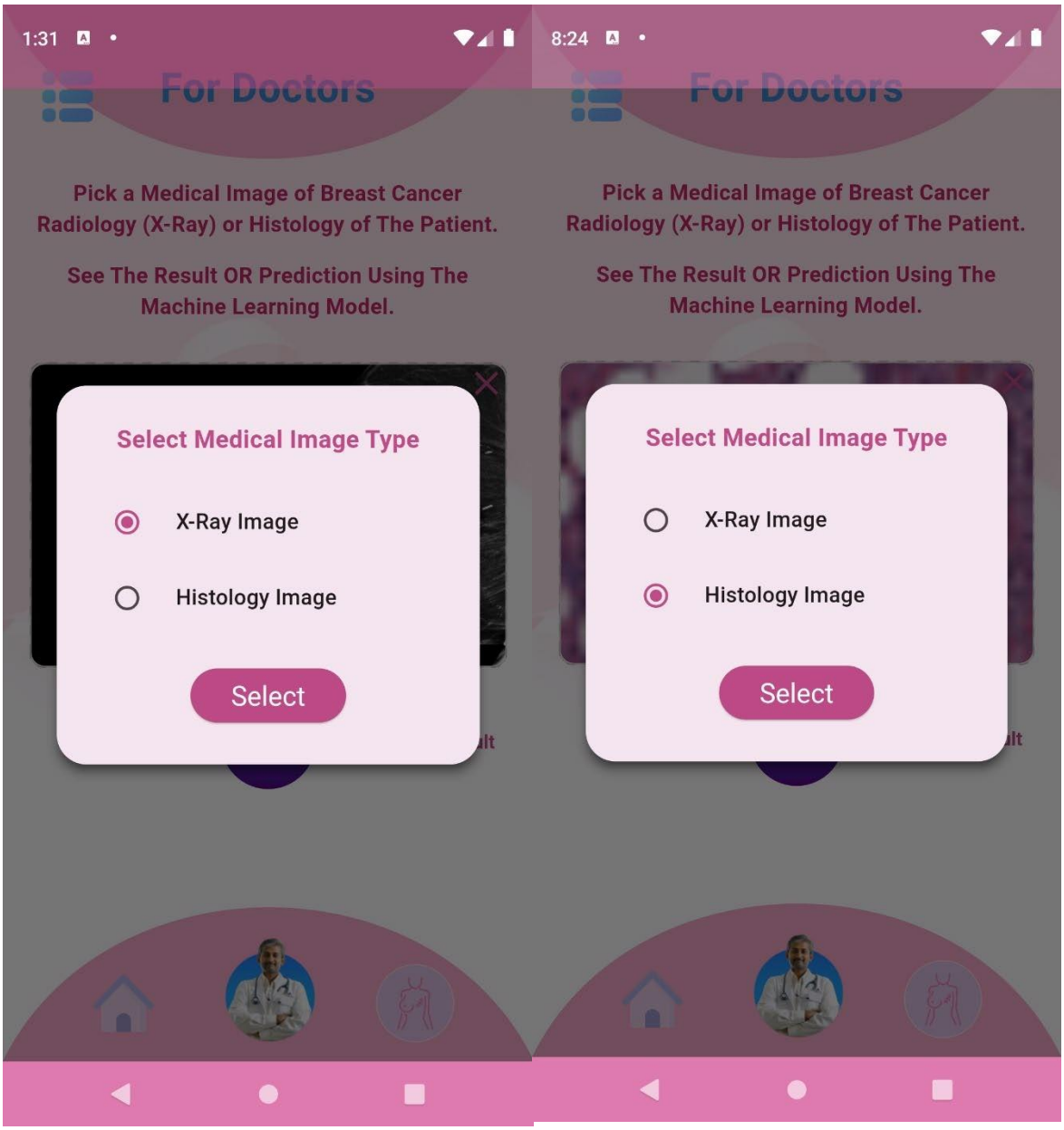

Figure 3.32 - Model Type Screen

After the user picks an image (x-ray or histology) and clicks the **show result** button, he must select the medical image type to get the right **model** for the image.

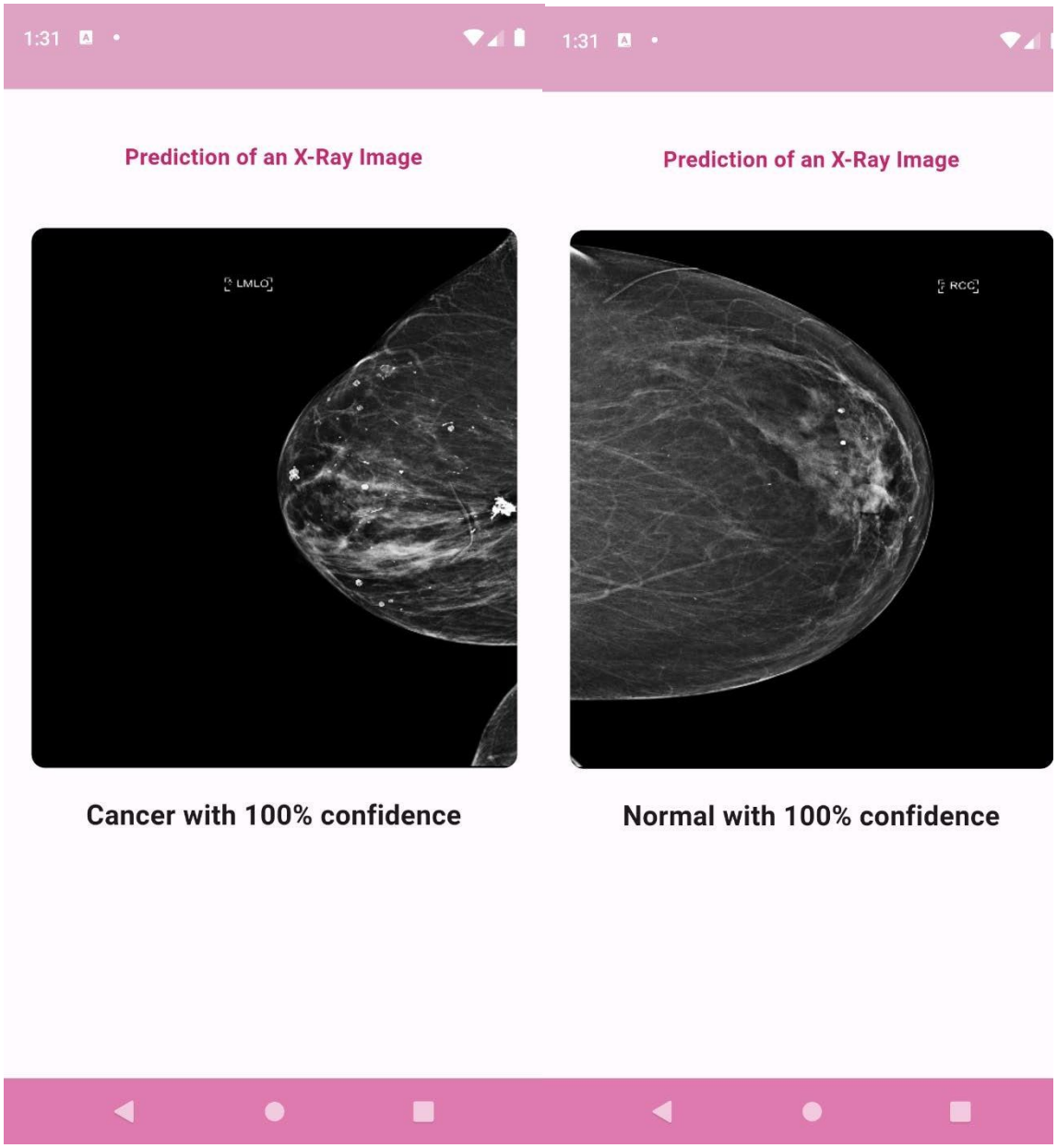

Figure 3.33 - Prediction Screen

The user (Doctor or radiologist) can see the model prediction on the chosen image.

In the case above, the medical image is **X-Ray** and the right **model** is specified.

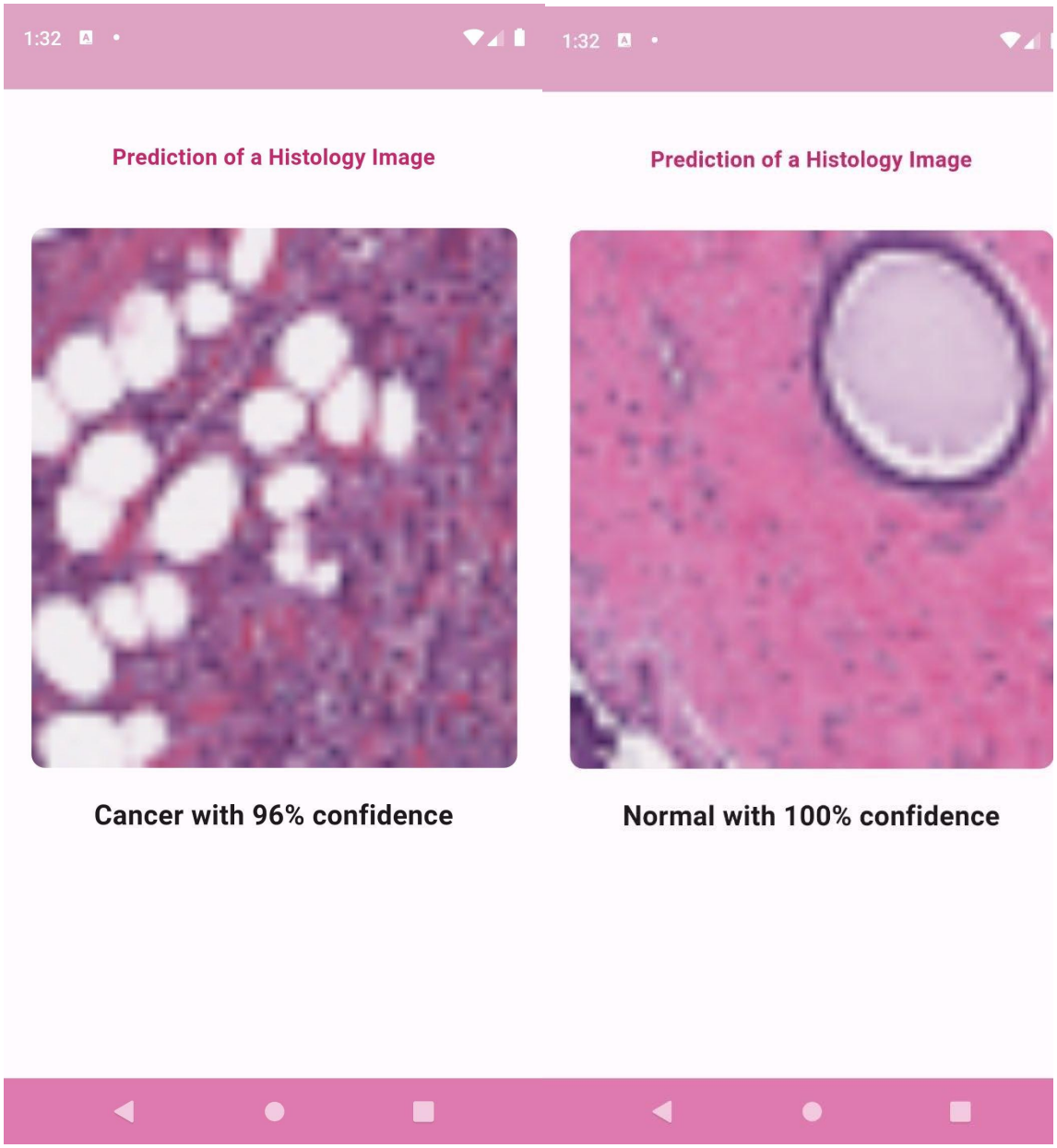

Figure 3.34 - Prediction Result Screen

In the case above, the medical image is **Histology** and the right **model** is specified.

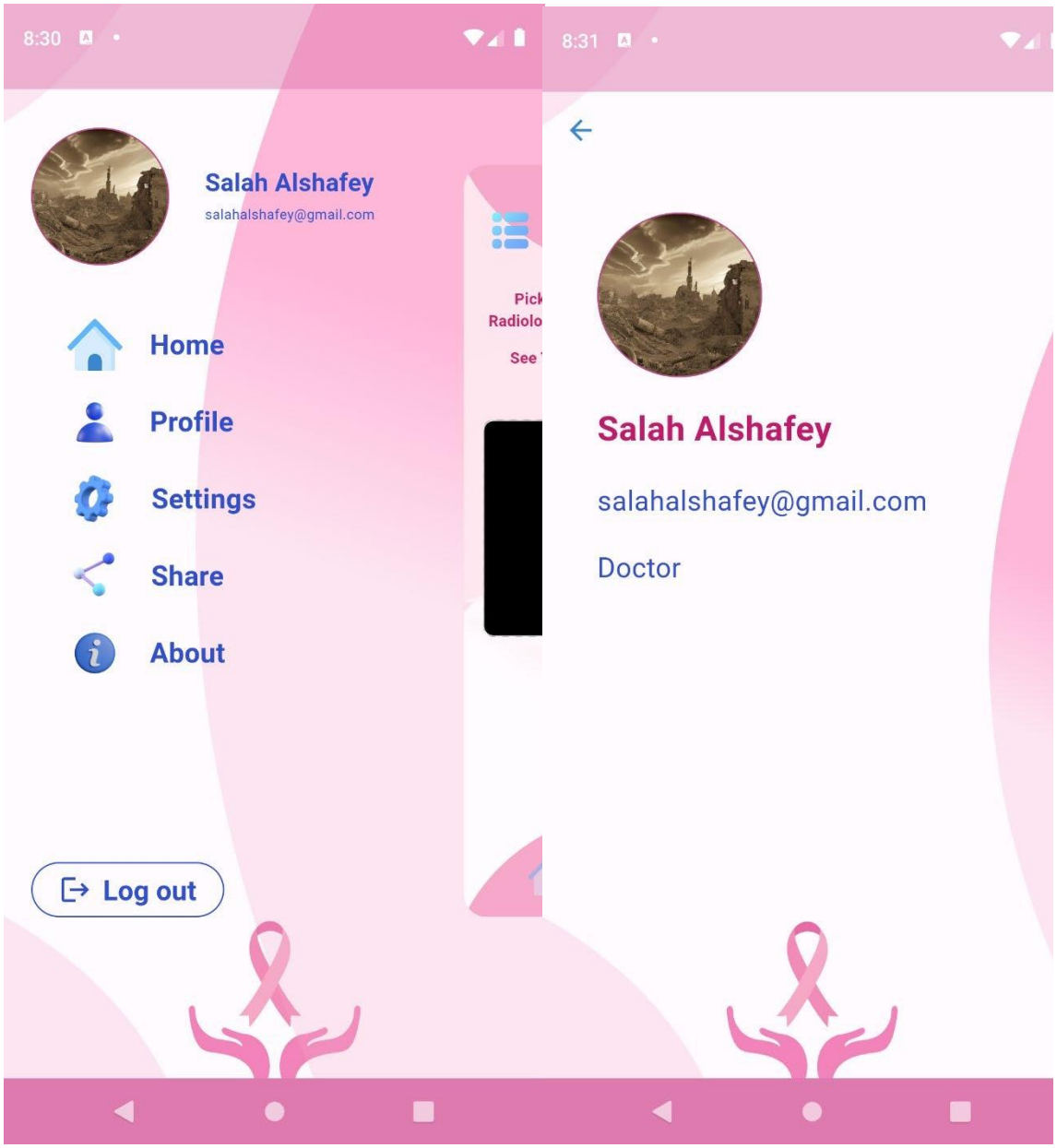

Figure 3.35 - Menu and Profile Screen

Containing these major things:

- The user profile basic information.
- General information about the app.
- Capability of sharing the app.

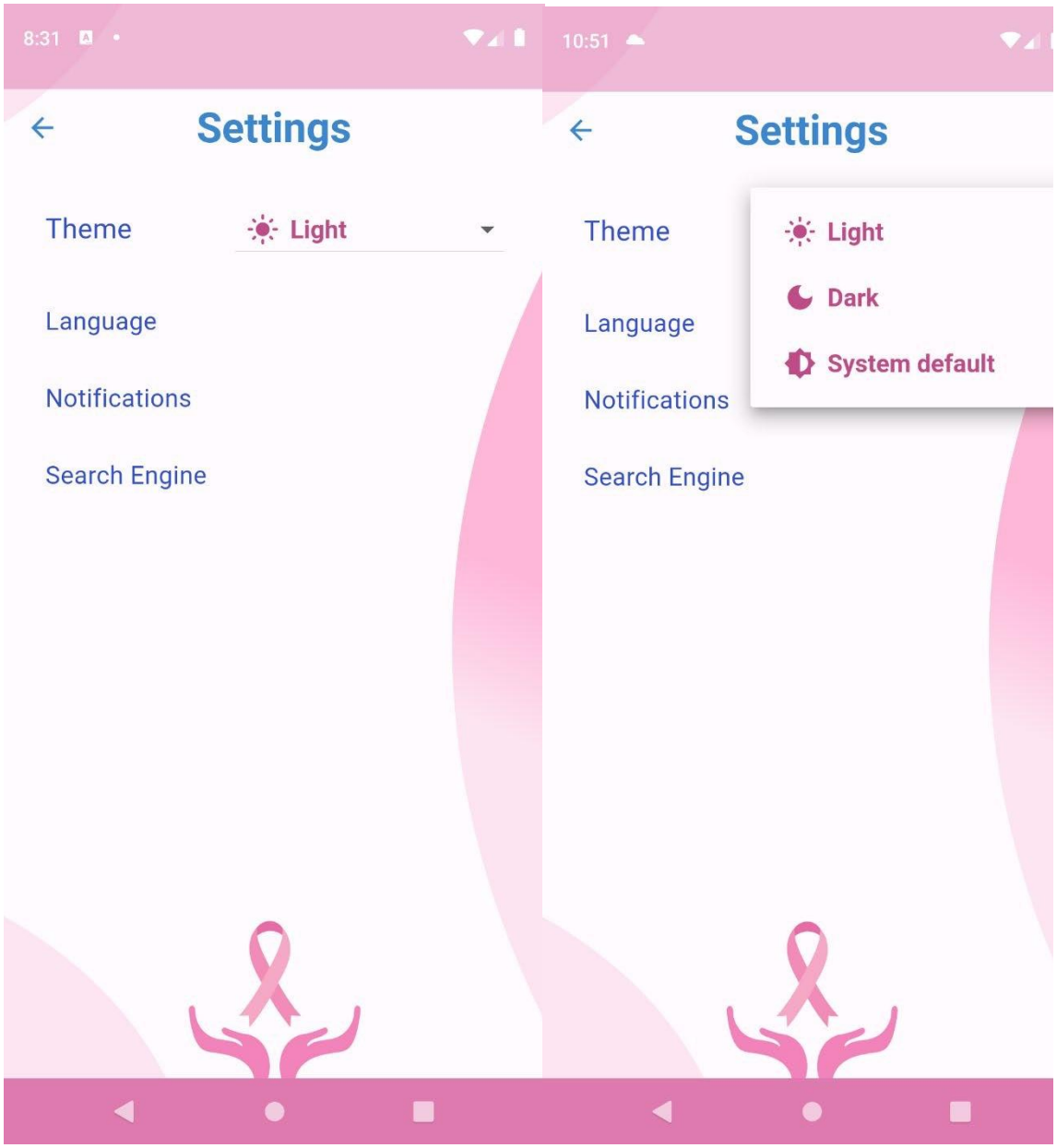

Figure 3.36 - Settings Screen

General settings of the app like the Theme, the user can change the theme to dark, light, or system default.

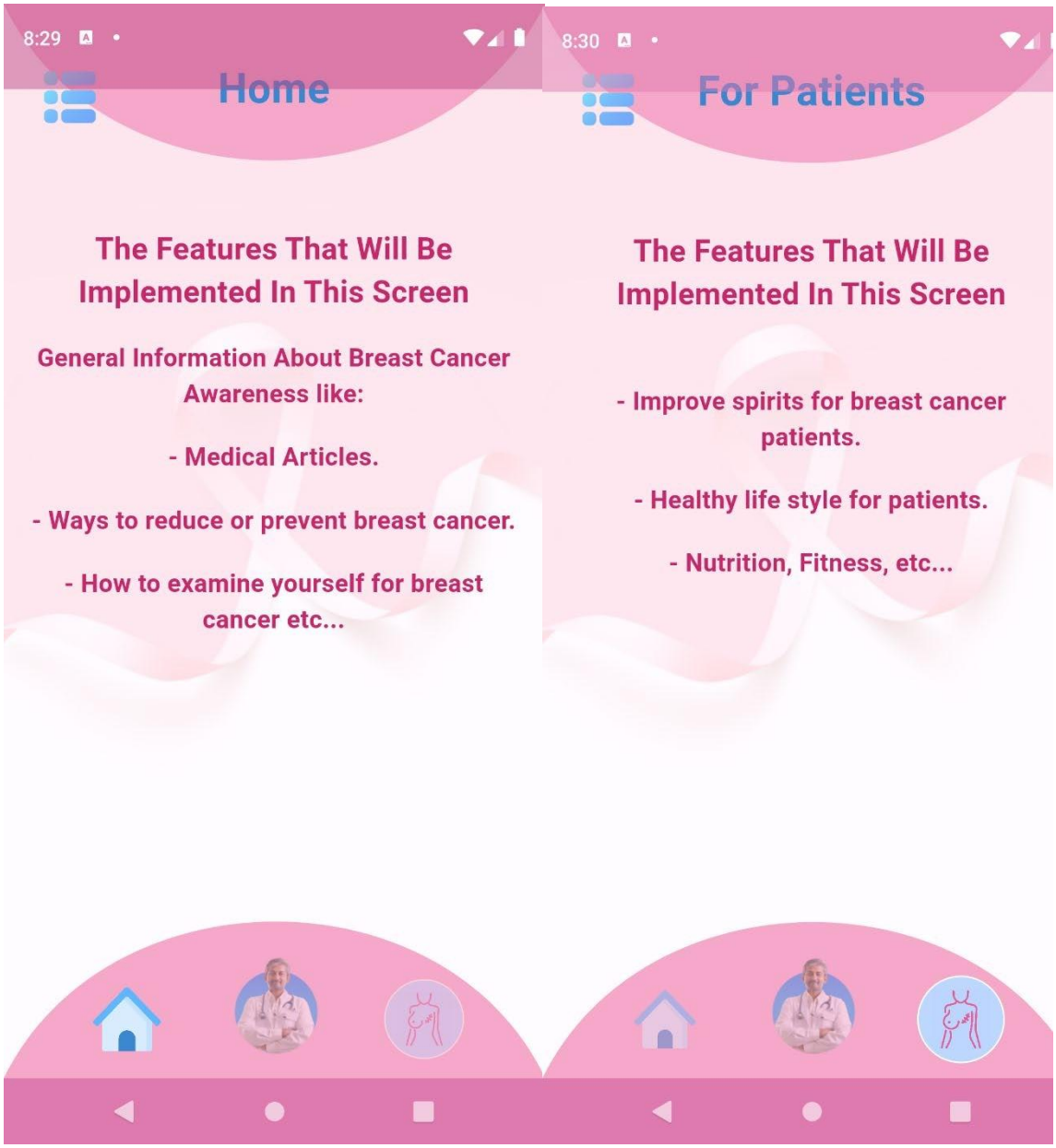

Figure 3.37 - Home and Patient Screens

These screens are “optional” and will be implemented in the future.

Backend.

```dart
class AccountFirebaseStorageImpl implements AccountRemoteStorage {
  @override
  Future<String> upload(String fileName, File file) async {
    try {
      final ref = FirebaseStorage.instance.ref().child('$folderName/$fileName');

      await ref.putFile(file);

      final downloadURL = await ref.getDownloadURL();
      return downloadURL;
    } catch (error) {
      throw ServerException();
    }
  }

  @override
  Future<void> delete(String fileName) async {
    try {
      await FirebaseStorage.instance
          .ref()
          .child('$folderName/$fileName')
          .delete();
    } catch (error) {
      throw ServerException();
    }
  }
}
```

Figure 3.38 - Account Storage

Figure 1.1 represents A code implements a remote storage module for handling file upload and deletion using Firebase Cloud Storage in Dart. The **AccountFirebaseStorageImpl** class implements the **AccountRemoteStorage** abstract class and provides the actual implementations for the abstract methods using Firebase Cloud Storage.

The **upload** method takes a **fileName** and a **File** object as parameters. It creates a reference to a location in Firebase Cloud Storage using the provided **fileName** and uploads the **File** using the **putFile()** method. It then retrieves the download URL of the uploaded file and returns it. If any error occurs during the upload process, a **ServerException** is thrown.

The **delete** method takes a **fileName** as a parameter. It creates a reference to the file in Firebase Cloud Storage using the provided **fileName** and deletes the file using the **delete()** method. If any error occurs during the deletion process, a **ServerException** is thrown.

```dart
class AccountFirebaseAuthenticationImpl implements AccountRemoteAuthentication {
  @override
  Future<String> signInWithEmailAndPassword(
      String email, String password) async {
    try {
      final credential = await FirebaseAuth.instance
          .signInWithEmailAndPassword(email: email, password: password);

      return credential.user!.uid;
    } on FirebaseAuthException catch (e) {
      if (e.code == 'user-not-found') {
        throw UserNotFoundException();
      } else if (e.code == 'wrong-password') {
        throw WrongPasswordException();
      } else if (e.code == 'invalid-email') {
        throw EmailNotValidException();
      } else {
        throw ServerException();
      }
    } catch (error) {
      throw ServerException();
    }
  }

  @override
  Future<String> signUpWithEmailAndPassword(
      String email, String password) async {
    try {
      final credential = await FirebaseAuth.instance
          .createUserWithEmailAndPassword(email: email, password: password);

      return credential.user!.uid;
    } on FirebaseAuthException catch (e) {
      if (e.code == 'weak-password') {
        throw WeakPasswordException();
      } else if (e.code == 'email-already-in-use') {
        throw EmailAlreadyInUseException();
      } else if (e.code == 'invalid-email') {
        throw EmailNotValidException();
      } else {
        throw ServerException();
      }
    } catch (error) {
      throw ServerException();
    }
  }
}
```

Figure 3.39 - Account Authentication

Figure 1.2 represents A code implements an authentication system using Firebase in Dart. The class **AccountFirebaseAuthenticationImpl** implements the **AccountRemoteAuthentication** interface and provides the actual implementations for the abstract methods. It utilizes the **FirebaseAuth** class from the **firebase_auth** package to perform the authentication operations.

The **signInWithEmailAndPassword** method attempts to sign in a user with the provided email and password. It uses the **signInWithEmailAndPassword** method provided by **FirebaseAuth** and returns the User ID (UID) of the authenticated user. It catches specific **FirebaseAuthException** errors and throws custom exceptions based on the error code. If any other error occurs, it throws a **ServerException**.

The **signUpWithEmailAndPassword** method attempts to create a new user account with the provided email and password. It uses the **createUserWithEmailAndPassword** method provided by **FirebaseAuth** and returns the UID of the newly created user.

```dart
class AccountFirestoreImpl implements AccountRemoteDataSource {
  AccountFirestoreImpl();

  @override
  Future<UserInformationModel> getUser(String userId) async {
    try {
      final document = await FirebaseFirestore.instance
          .collection('users')
          .doc(userId)
          .get();

      if (document.data() == null) {
        throw EmptyDataException();
      }

      return UserInformationModel.fromFirestore(document.data()!, document.id);
    } on EmptyDataException {
      rethrow;
    } catch (error) {
      throw ServerException();
    }
  }

  @override
  Future<void> addUser(UserInformationModel user) async {
    try {
      await FirebaseFirestore.instance
          .collection('users')
          .doc(user.id)
          .set(user.toFirestore());
    } catch (error) {
      throw ServerException();
    }
  }

  @override
  Future<void> storeUserImageAndType(
      String userId, String? imageUrl, String userType) async {
    try {
      await FirebaseFirestore.instance.collection('users').doc(userId).update(
        {
          'image_url': imageUrl,
          'user_type': userType,
        },
      );
    } catch (error) {
      throw ServerException();
    }
  }
}
```

Figure 3.40 - Account Data Store

Figure 1.3 represents a code implements a data source for account-related operations using Firestore in Dart. The class **AccountFirestoreImpl** implements the **AccountRemoteDataSource** abstract class and provides the actual implementations for the abstract methods using Firestore as the data source.

The **getUser** method retrieves a document from the Firestore collection 'users' based on the provided **used**. If the document's data is null, it throws **EmptyDataException**. Otherwise, it constructs a **UserInformationModel** object from the document's data and ID and returns it. Any other error during the retrieval process will result in a **ServerException** being thrown.

The **addUser** method adds a new document to the Firestore collection 'users' with the data from the provided **UserInformationModel** object. It uses the **set()** method to set the document with the provided data. If any error occurs during this process, a **ServerException** is thrown.

The **storeUserImageAndType** method updates an existing document in the Firestore collection 'users' with the provided **userId**. It sets the 'image_url' and 'user_type' fields with the provided values using the **update()** method. If any error occurs during this process, a **ServerException** is thrown.

```dart
Future<String> _getPrediction(
    BuildContext context, ForDoctorScreenState provider, bool isXray) async {
  String prediction = "";
  try {
    final model = await FirebaseModelDownloader.instance.getModel(
      isXray ? "xray_model_for_deployment" : "histo_model_for_deployment",
      FirebaseModelDownloadType.localModelUpdateInBackground,
      FirebaseModelDownloadConditions(
        iosAllowsCellularAccess: true,
        iosAllowsBackgroundDownloading: false,
        androidChargingRequired: false,
        androidWifiRequired: false,
        androidDeviceIdleRequired: false,
      ),
    );
    final labelspath = await saveLabelsToFile();

    Tflite.close();
    await Tflite.loadModel(
        model: model.file.path, // "assets/xray_model_for_deployment.tflite",
        labels: labelspath, //"assets/labels.txt",
        numThreads: 1, // defaults to 1
        isAsset:
            false, // defaults to true, set to false to load resources outside assets
        useGpuDelegate:
            false // defaults to false, set to true to use GPU delegate
        );

    final filepath = provider.fileImage != null
        ? provider.fileImage!.path
        : await _getNetworkImageToFile(provider.networkImage!);

    final recognitions = await Tflite.runModelOnImage(
        path: filepath, // required
        imageMean: 0.0, // defaults to 117.0
        imageStd: isXray ? 255.0 : 1.0, // defaults to 1.0
        numResults: 2, // defaults to 5
        threshold: 0.2, // defaults to 0.1
        asynch: true // defaults to true
        );

    prediction = recognitions![0]["label"] +
        " with " +
        ((recognitions[0]["confidence"] as double) * 100).toStringAsFixed(0) +
        "% confidence";
  } catch (error) {
    throw Error("Something went wrong!!!");
  }

  return prediction;
}
```

Figure 3.41 - Prediction

Figure 1.3 represents a code defines a private asynchronous method named **_getPrediction**. This method is responsible for performing a prediction using a machine learning model.

The method begins by initializing a **prediction** variable as an empty string. It then enters a try block where it downloads a machine learning model using the **FirebaseModelDownloader** class, based on the value of the **isXray** Boolean parameter. It also calls a function to save the labels file and assigns the file path to a **labelspath** variable.

The previously loaded TensorFlow Lite models are closed, and the desired model is loaded using the **Tflite.loadModel** method. It specifies the model file path, labels file path, number of threads, and asset loading and GPU delegate flags.

Next, the method determines the **filepath** based on whether a local file image is provided or a network image is available. It calls a function to get the network image file path if necessary.

The TensorFlow Lite model is then run on the image specified by the **filepath** using the **Tflite.runModelOnImage** method. The results are stored in the **recognitions** variable.

Finally, the method constructs the **prediction** string by accessing the label and confidence values from the first recognition result. The prediction string is returned as the result of the method.

If any errors occur during the process, an **Error** with the message "Something went wrong!!!" is thrown.

Conclusion

The development and integration of CNN models for histopathology and X-ray images in the breast cancer detection system represent a significant breakthrough in the field of medical diagnostics. We aimed to improve the accuracy and efficiency of breast cancer diagnosis.

Through extensive training on large datasets of breast imaging data, including x-ray and histopathology images, our CNN models demonstrated remarkable performance in accurately identifying and classifying breast cancer indicators. Considering the high accuracy rate of 99%, shows great assistance to healthcare professionals with early detection.

One of the key strengths of our project, the integration of these CNN models through a mobile app. Allows for immediate access to results, providing users with timely and reliable information about their breast health.

Moreover, it underscores the importance of multidimensional data analysis by incorporating multiple imaging modalities. By incorporating both x-rays and histopathology images, we harnessed the complementary nature of these modalities, capturing a comprehensive view of breast tissue abnormalities.

CNN models have the opportunity for real-world implementation. The integration of deep learning models and the development of user-friendly applications can revolutionize breast cancer diagnosis, enabling earlier detection, personalized treatment, and improved patient outcomes. As this system helps to enhance the progress in the field of breast cancer detection, With its high accuracy, efficient analysis, and multimodal approach. Our system has the potential to contribute to the early detection and effective management of breast cancer, ultimately making a positive impact on patients' lives and advancing the fight against this devastating disease.

References

1. Harris, J. R., Lippman, M. E., Morrow, M., & Osborne, C. K. (2019). Diseases of the Breast. Philadelphia: Wolters Kluwer Health.

2. Arevalo, J., González, F. A., Ramos-Pollán, R., Oliveira, J. L., & López, M. A. (2016). Representation learning for mammography mass lesion classification with convolutional neural networks. Computer Methods and Programs in Biomedicine, 127, 248-257.

3. Holzinger, A., Langs, G., Denk, H., Zatloukal, K., & Müller, H. (2019). Causability and explainability of artificial intelligence in medicine. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 9(4), e1312.

4. S. Lee et al., "Comparative study of deep learning methods for breast cancer recurrence prediction," Healthcare Informatics Research, vol. 24, no. 2, pp. 105-114, 2018.

5. M. Masud, A. E. Eldin Rashed, and M. S. Hossain, “Convolutional neural network-based models for diagnosis of breast cancer,” Neural Computing and Applications, vol. 5, 2020.

6. C Kaushal, A. Singla, “Analysis of breast cancer for histological dataset based on different feature extraction and classification algorithms” Adv Intelligent Sys Comp, 1165 (2021), pp. 821-833.

7. P. Kumar, S. Srivastava, R. K. Mishra, and Y. P. Sai, “End-to-end improved convolutional neural network model for breast cancer detection based on mammographic data,” The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, Article ID 154851292097326, 2018.

Appendix.

This project submitted to the High Institute of Computer Science & Information Systems, in partial fulfillment of the requirements for the degree of Bachelor of Computers and Information.

According to the instructions of High Institute of Computer Science & Information Systems, the project has uploaded to the GitHub:

https://github.com/salahalshafey/breast_cancer_awareness.git

https://github.com/salahalshafey/breast_cancer_detection.git

Arabic Summary of the project

سرطان الثدي هو أكثر أنواع السرطان شيوعاً مع أكثر من ٢,٢ مليون حالة سنوياً، لهذا السبب يعتبر سرطان الثدي مصدر قلق صحي كبير في جميع أنحاء العالم، ويلعب التشخيص والكشف المبكر لهذا المرض الخطير دو ًرا مهما في تحسين نتائج المرضى.

في هذا المشروع، استخدمنا تقنية تسمى التعلم العميق للكشف عن سرطان الثدي باستخدام الشبكة العصبية التلافيفية (CNN).

قمنا بتدريب نموذجي CNN باستخدام مجموعة بيانات كبيرة تتكون مما يقارب ٧٧٢ ألف صورة من صور الأنسجة المرضية (histopathology)، وما يقارب ٤٥ ألف صورة بالأشعة السينية (X-rays)، تم الحصول عليها من مصادر متعددة أهمها موقع كاجل (Kaggle).

اتبعنا نهجاً يعالج مشكلة الكشف عن سرطان الثدي من طريقتين مختلفتين للتصوير، صور الأنسجة المرضية والأشعة السينية. توفر صور الأنسجة المرضية تفاصيل مجهرية للثدي من عينات الأنسجة، بينما توفر صور الأشعة السينية معلومات على المستوى المرئي حول الثدي.

من خلال الجمع بين نقاط القوة في كلتا الطريقتين، نحن نهدف إلى تحسين دقة وموثوقية الكشف عن سرطان الثدي.

لربط النموذجين، قمنا بتطوير تطبيق (للهاتف المحمول) يدمج كل من نموذجي الـCNN المدربة على صور الأنسجة المرضية وصور الأشعة السينية. يأخذ التطبيق صور كمدخلات من كلا الطريقتين ويغذيها في نماذج الـCNN الخاصة بهم لاستخراج الميزات (feature extraction)، ثم يتم دمج الميزات المستخرجة في مستوى القرار (output layer) لاتخاذ قرار التصنيف النهائي (وجود سرطان أو عدم وجوده).

قمنا بتقييم أداء النموذجين بمجموعة كبيرة من البيانات أو الصور المتنوعة، وحققت نتائج واعدة من حيث الدقة والسرعة.

نهجنا لديه القدرة على تحسين دقة الكشف عن سرطان الثدي، والذي يمكن أن يساعد المتخصصين في التشخيص المبكر والتدخل واتخاذ القرار في الوقت المناسب، مما يؤدي في النهاية إلى نتائج أفضل للمرضى.