ChestVision

Ecosystem for medical image analysis and disease prediction on thoracic x-rays

Bachelor Thesis

Yannick Künzli

June 2024

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
Software Engineering Group

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Software
Engineering Group

Acknowledgements

I want to thank Prof. Dr. Jacques Pasquier, my supervisor, and all the people that helped me and read my thesis during its development.

Abstract

A specialised Flask web application, named "ChestVision", was developed and implemented using Python. The application is designed to facilitate the analysis of thoracic x-rays and predict potential diseases. Its primary function allows users to store and manage images, patient data, and their corresponding diseases.
The application leverages the National Institutes of Health (NIH) Chest X-ray Dataset and a deep learning model that implements a pretrained MobileNet layer to predict between 13 diseases, providing a comprehensive tool for medical image analysis. Secondary functions, such as accessing patient records, displaying the state of disease prediction were added during development.

Keywords: Flask, Python, CXR8, Deep Learning, Medical Image Analysis, MobileNet

Table of Contents

List of Figures

Listings

1 Introduction

1.1. Motivation and Goals

The advancements of technology have revolutionized various sectors, including healthcare. With the increasing prevalence of diseases and the need for efficient diagnostic tools, the role of technology, particularly machine learning and artificial intelligence, has become more significant than ever. This project, ChestVision [7], aims at building a simple application that can help medical practitioners make their diagnosis.

The primary goal of this project is to develop a Flask web application that serves as a comprehensive platform for managing patient data, specifically X-ray images and their associated diseases. The application is designed to not only store and manage data but also to analyze it using a deep learning Convolutional Neural Network (CNN) model. The CNN model is capable of analyzing X-ray images and predicting the presence of thoracic and lung diseases, thereby assisting in the diagnostic process.

The motivation behind this project stems from the critical need for efficient and accurate diagnostic tools in the healthcare sector. Diseases such as atelectasis, cardiomegaly, and pneumonia affect a significant portion of the population. Early detection and diagnosis can profoundly impact patient outcomes, and having a reliable tool to aid in this process is invaluable.

Additionally, the project aims to demonstrate the practical application of various technologies, including Python, Flask, SQLAlchemy, TensorFlow, Keras, Open Computer Vision Library (OpenCV), Bootstrap, Cascading Style Sheets (CSS)/JavaScript (JS)/HyperText Markup Language (HTML), and Jinja, in developing a functional and user-friendly web application.

1.2. Organization

Chapter 1: Introduction

This chapter presents the motivation and goals of this work, provides an overview of the structure of each subsequent chapter, and outlines the formatting conventions used throughout the document.

Chapter 2: NIH Chest X-ray Dataset description and context

This chapter provides an overview of the NIH Chest X-ray Dataset [9], essential for developing ChestVision. It includes a description of the dataset, statistical insights, and detailed descriptions of the diseases the CNN model can detect, along with their diagnostic methods using chest X-ray imaging.

Chapter 3: ChestVision - User demonstration

This chapter offers a user-centric demonstration of the ChestVision application, detailing different scenarios and functionalities from the perspective of an end-user. It describes the user experience for each view of the website, illustrating how the application can be used in a clinical setting.

Chapter 4: ChestVision - Technologies and Architecture

In this chapter, the global architecture of the ChestVision application is explained using a deployment diagram. It also discusses all the technologies involved in creating this web application, providing insights into how these technologies integrate to form a cohesive system.

Chapter 5: Model implementation

This chapter focuses on the implementation of the CNN model. It describes how the data and images from the dataset are processed, the training of the model, and provides a detailed description of the model architecture. It also highlights the advantages of certain image processing methods and presents the model's results.

Chapter 6: Application implementation

This chapter covers the implementation of the ChestVision application. It explains the Application Programming Interface (API), the relational database, and the authentication system. It also includes the frontend development, detailing the creation of the 'Home', 'Predictions', and 'Patients' pages.

Chapter 7: Conclusion

The final chapter summarizes the functionalities of the ChestVision application, presents the model's results, and discusses improvements. It highlights the strengths and limitations observed and suggests future directions for enhancing the application.

Appendix

Contains extracts of artefacts or service messages, abbreviations and references used throughout this work.

1.3. Notations and Conventions

- Formatting conventions:

  - Abbreviations and acronyms as follows Convolutional Neural Network (CNN) for the first usage and CNN for any further usage;
  - `http://127.0.0.1:5000` is used for web addresses;
  - Code is formatted as follows:

```python
import json

def greet():
   print('Hello World !')

def main():
   greet()

if __name__ == '__main__':
   main()
```

- The work is divided into seven chapters that are formatted in sections and subsections. Every section or subsection is organized into paragraphs, signalling logical breaks.
- Figure s, Table s and Listings s are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j*.
- As far as gender is concerned, I systematically select the masculine form due to simplicity. Both genders are meant equally.

2 NIH Chest X-ray Dataset description and context

2.1. Dataset description

In the field of medical imaging and computer-aided detection and diagnosis (CAD) systems, the quality and comprehensiveness of datasets are incredibly important. Good datasets serve as the foundation for developing robust and accurate models that can assist healthcare professionals in making timely and precise diagnoses. They enable the training and validation of machine learning algorithms, ensuring these models can generalize well to real-world clinical scenarios. In medical imaging, where the accurate identification of pathologies is critical, high-quality datasets facilitate the development of CAD systems that enhance diagnostic accuracy, improve patient outcomes, and ultimately contribute to more efficient healthcare delivery. This chapter dives into the specifics of the dataset used in our project, highlighting its characteristics and significance in advancing thoracic disease diagnosis through machine learning.

2.1.1. Pathologies

The **National Institutes of Health (NIH) Chest X-ray Dataset**[9] is a comprehensive collection of chest X-ray images that are labeled with 14 common thorax disease categories. These include: atelectasis, cardiomegaly, effusion, infiltration, mass, nodule, pneumonia, pneumothorax, consolidation, edema, emphysema, fibrosis, pleural thickening and hernia.

2.1.2. Goal

The dataset was created to facilitate the development of CAD systems that can assist in the interpretation of chest X-ray images, which is one of the most common and cost-effective medical imaging examinations but can be challenging to interpret accurately.

The NIH Chest X-ray Dataset is an enhanced version of the dataset used in previous work, with six additional disease categories and more images. It contains 112,120 frontal-view X-ray images of 30,805 unique patients, with each image potentially having multiple disease labels. The labels were mined from the associated radiological reports using Natural Language Processing (NLP).

2.1.3. Data

The dataset includes:

1. 112,120 frontal-view chest X-ray Portable Network Graphics (PNG) images in 1024*1024 resolution.
2. Meta data for all images, including Image Index, Finding Labels, Follow-up #, Patient ID, Patient Age, Patient Gender, View Position, Original Image Size, and Original Image Pixel Spacing.
3. Bounding boxes for approximately 1000 images.
4. Two data split files (train_val_list.txt and test_list.txt) that divide the images into training/validation and testing sets on the patient level.

2.1.4. Limitations

The dataset has some limitations, including the potential for erroneous labels due to the NLP extraction process (though the accuracy is estimated to be over 90%), a limited number of disease region bounding boxes, and the fact that the original radiology reports are not publicly shared.

2.1.5. Personal decisions

In the context of this study, I opted not to use the provided data split files. Instead, I divided the images during the model's training phase using the train_test_split module from the sklearn.model_selection library.

The model was not trained on the 'Hernia' disease category. This decision was made due to the relatively small number of hernia images in the entire dataset, which was less than a thousand. By excluding this category, it allowed for a more balanced training across the other diseases, which in turn improved the overall accuracy of the model.

2.1.6. Distribution of co-occurrence diseases

The diagram below (Figure 2.1) shows the proportion of images with multi-labels in each of the 14 pathology classes and the labels' co-occurrence statistics.

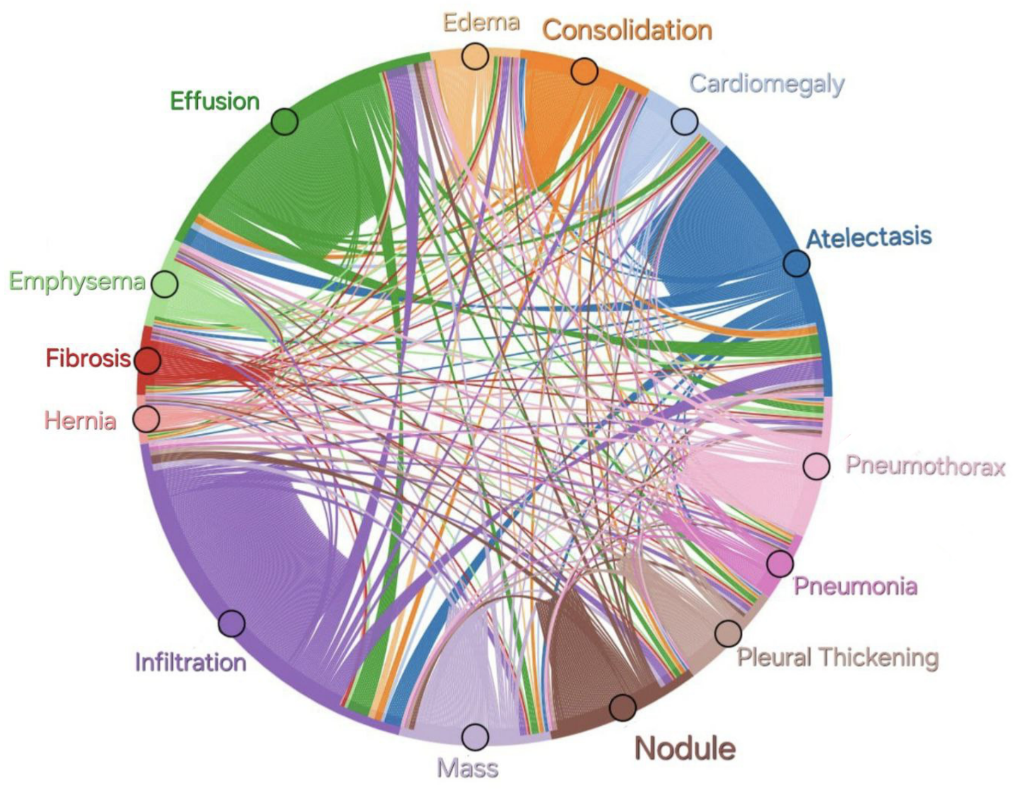


Fig. 2.1.: Distributions of 14 disease categories with co-occurrence statistics

2.2. Statistical data analysis in python

Statistical analysis plays a crucial role in understanding the underlying patterns and distributions within a dataset. In the context of medical imaging and disease diagnosis, it helps us gain insights into the prevalence of various conditions, the demographic characteristics of the patient population, and potential biases in the data. By performing a thorough statistical analysis, we can ensure that our machine learning models are trained on a representative sample, leading to more reliable and generalizable predictions. This section aims to provide an overview of the statistical characteristics of the dataset, including gender distribution and the distribution of disease findings, to highlight the dataset's comprehensiveness and potential areas of interest.

2.2.1. Gender distribution

The analysis of the gender distribution within the dataset reveals a balanced representation of male and female patients. This balance is crucial for developing a model that does not exhibit gender bias and can accurately predict diseases across both sexes. A balanced gender distribution ensures that the model is equally effective in diagnosing conditions in both male and female patients, contributing to fair and unbiased medical assessments.

In this small code snippet, we simply print the number of patient per gender (men and women). We can see that the dataset is fairly well divided with 53.98% of men and 46.02% of women.

```python
print('Number of men : ', all_xray_df[all_xray_df['Patient Gender'] == 'M']['Patient ID'].nunique())
print('Number of women : ', all_xray_df[all_xray_df['Patient Gender'] == 'F']['Patient ID'].nunique())
```

List. 2.1: Gender Distribution in the dataset

```
Number of men : 16630
Number of women : 14175
```

2.2.2. Disease distribution

In the following section, we perform a statistical data analysis using Python. The goal is to visualize the distribution of findings from the diagnoses tied to the X-rays.

The distribution of disease findings in the dataset provides valuable insights into the prevalence of various thoracic conditions. The statistical analysis shows that some conditions are more prevalent than others, which can influence the model's performance. For instance, cardiomegaly has a higher representation in the dataset, which may lead to higher accuracy for this condition. On the other hand, diseases like pneumonia, which have a lower prevalence, present a greater challenge for the model to accurately predict. Understanding these distributions helps in assessing the model's strengths and identifying areas where additional data might be needed to improve diagnostic accuracy.

We start by importing two essential Python libraries: Numerical Python (NumPy)[34], for numerical operations, and matplotlib.pyplot[31], for creating plots. Next, we count the number of images associated with each pathology and store these counts in label_counts.

We then create a bar plot to visualize these counts. The x-axis represents the different pathologies, and the y-axis represents the number of images associated with each pathology.

```python
import numpy as np
import matplotlib.pyplot as plt

label_counts = all_xray_df['Finding Labels'].value_counts()[:15]
fig, ax1 = plt.subplots(1,1,figsize = (12, 8))
ax1.bar(np.arange(len(label_counts))+0.5, label_counts)
ax1.set_xticks(np.arange(len(label_counts))+0.5)
_ = ax1.set_xticklabels(label_counts.index, rotation = 90)
```

List. 2.2: Findings' Distribution plot

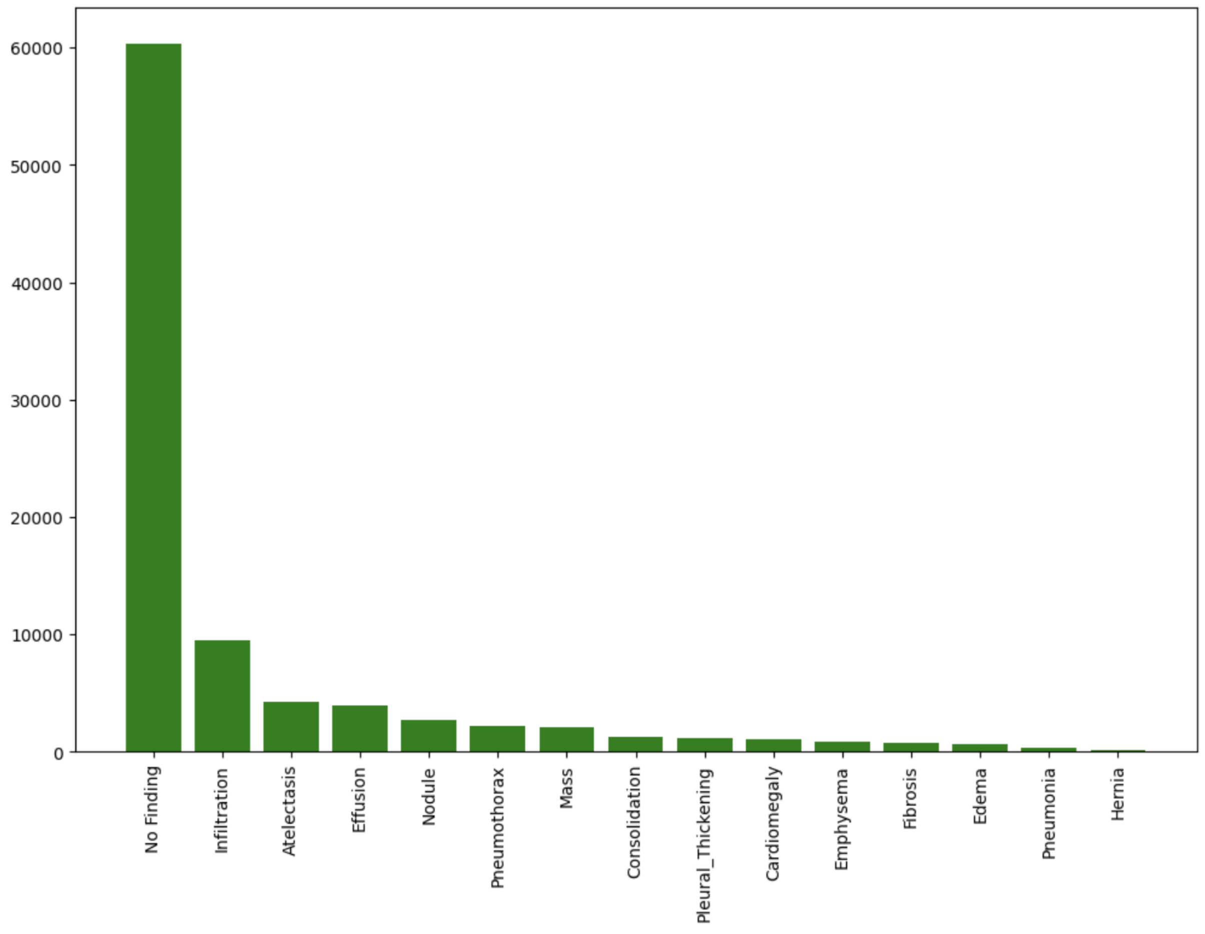


Fig. 2.2.: Distribution of findings

2.2.3. Image sample

Finally, the code below displays 16 images from the dataset along with their diagnosis.

```python
t_x, t_y = next(train_gen)
fig, m_axs = plt.subplots(4, 4, figsize = (16, 16))
for (c_x, c_y, c_ax) in zip(t_x, t_y, m_axs.flatten()):
    c_ax.imshow(c_x[:,:,0], cmap = 'bone', vmin = -1.5, vmax = 1.5)
    c_ax.set_title(', '.join([n_class for n_class, n_score in zip(all_labels, c_y)
                              if n_score>0.5]))
    c_ax.axis('off')

fig.savefig('sample.png')
```

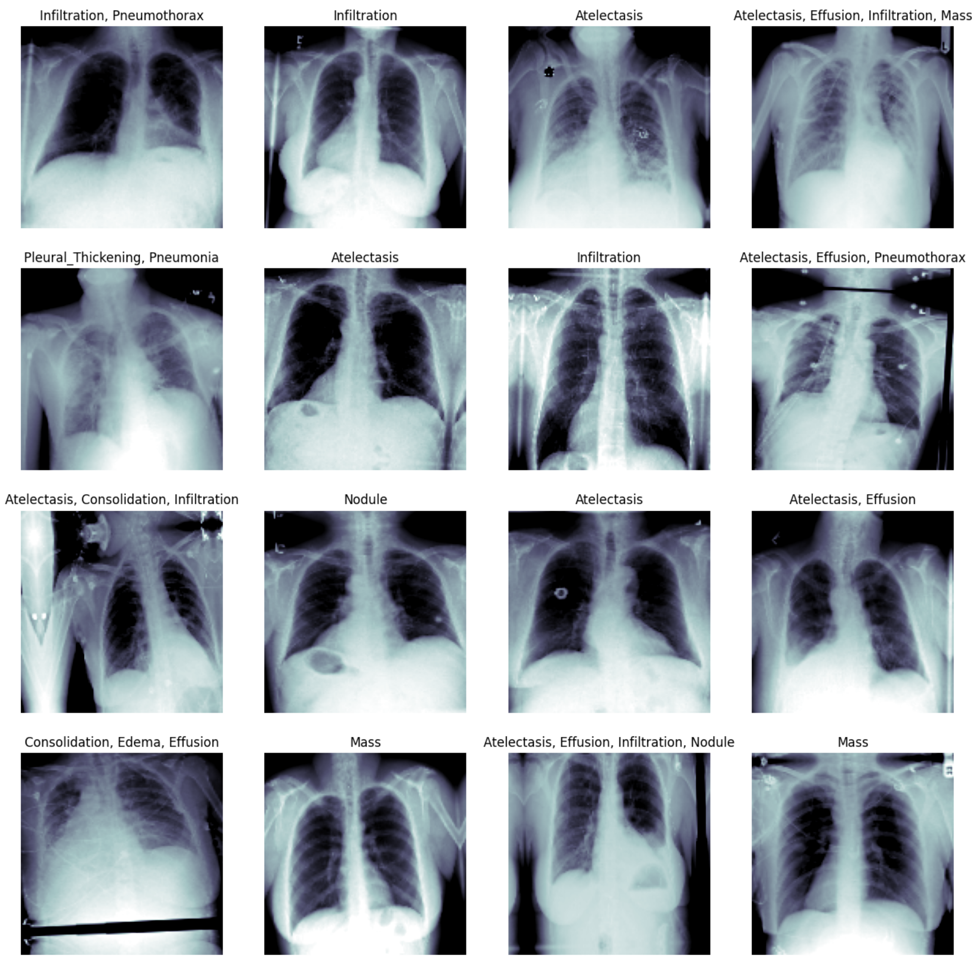

Infiltration, Pneumothorax
Infiltration
Atelectasis
Atelectasis, Effusion, Infiltration, Mass
Pleural_Thickening, Pneumonia
Atelectasis
Infiltration
Atelectasis, Effusion, Pneumothorax
Atelectasis, Consolidation, Infiltration
Nodule
Atelectasis
Atelectasis, Effusion
Consolidation, Edema, Effusion
Mass
Atelectasis, Effusion, Infiltration, Nodule
Mass

Fig. 2.3.: Sample of the dataset

2.3. Diseases descriptions and diagnosis

As I am not a doctor or a healthcare professional, all of the definitions and images below were found directly on the websites associated with the images.

2.3.1. Atelectasis

Description : Atelectasis[4] is a condition where lungs collapse partially or completely. It may be caused by a blocked airway or pressure from outside (also could be blocked my muccus). Symptoms include difficulty in breathing, rapid shallow breathing, chest pain, and coughing.

Diagnosis on x-ray imaging : On the image on the left, we can see a normal chest x-ray, with the heart correctly positioned one third of its length on the right side of the spine and two thirds on the left side. On the image on the right, we see a clear deviation to the left side caused by the collapse of the left lung.

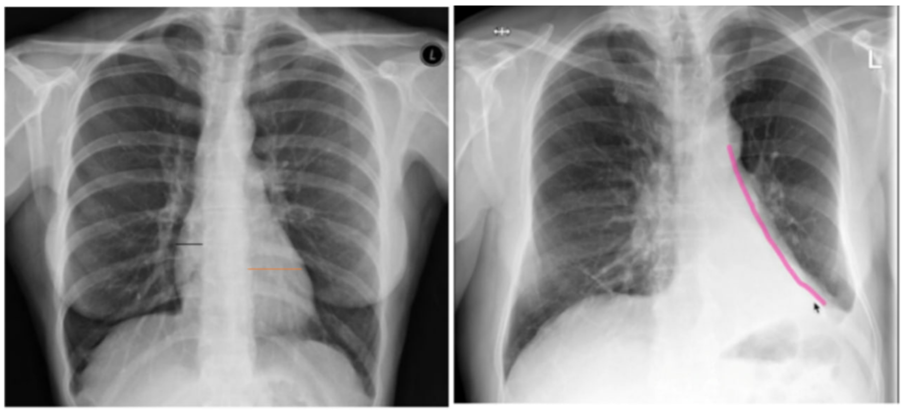


Fig. 2.4.: Example of Atelectasis on chest x-ray[4]

2.3.2. Cardiomegaly

Description : Cardiomegaly[6] is a medical condition in which the heart becomes enlarged. It is usually the result of underlying conditions that make the heart work harder, such as obesity, heart valve disease, high blood pressure.

Diagnosis on x-ray imaging : When compared to the normal image that was defined in the chapter about Atelectasis (Figure 2.4), Figure 2.5 clearly shows the enlarged heart.

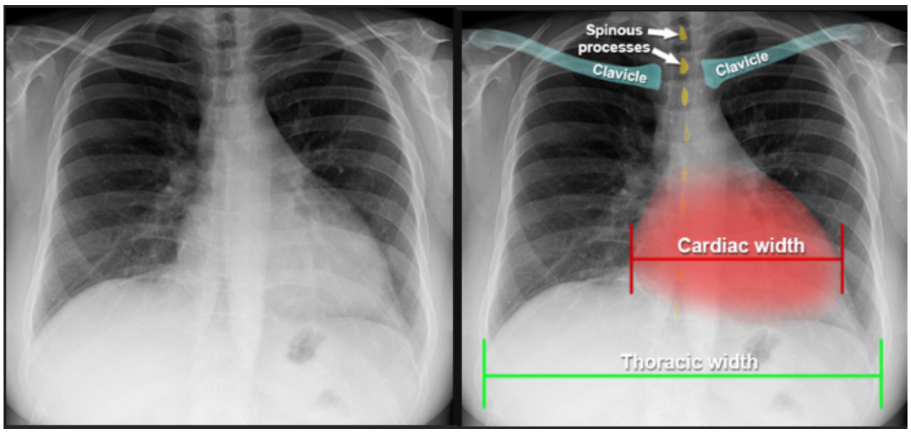


Fig. 2.5.: Example of Cardiomegaly on chest x-ray [6]

2.3.3. Consolidation

Description : In the context of lung disease, consolidation [8] refers to the process where the lung tissue becomes filled with liquid instead of air, often due to an infection or inflammation.

Diagnosis on x-ray imaging : on this image, the consolidation is located on the lower right zone. It is described as a patchy area.

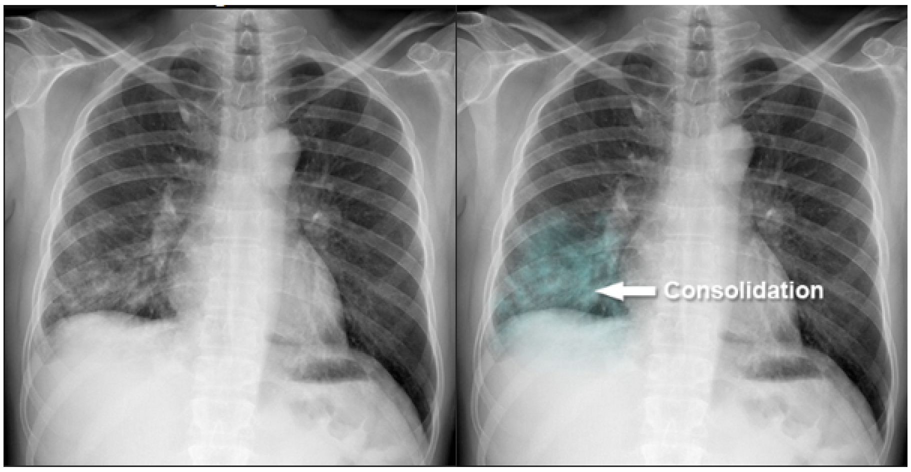


Fig. 2.6.: Example of Consolidation on chest x-ray [8]

2.3.4. Edema

Description : Edema [10] is swelling caused by too much fluid trapped in the body's tissues. It can affect any part of the body but it's more likely to show up in the legs and feet.

Diagnosis on x-ray imaging : lungs are supposed to be a lot darker, as air does not reflect light, however here, as the lungs are filled with fluids, the image is very white, because fluids do reflect the x-ray light.

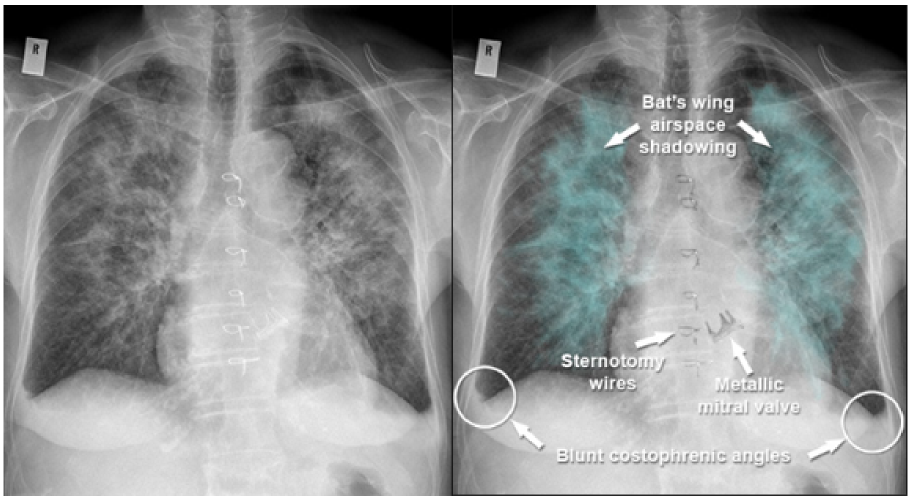


Fig. 2.7.: Example of Edema on chest x-ray [10]

2.3.5. Pleural Effusion

Description : In medical terminology, an effusion [37] refers to the accumulation of fluid in an anatomic space, usually without loculation. Specific examples include subdural, mastoid, pericardial and pleural effusions. Often seen on the base of the lung.

Diagnosis on x-ray imaging : Here the lung is clearly filled with fluid on more than half of its volume.

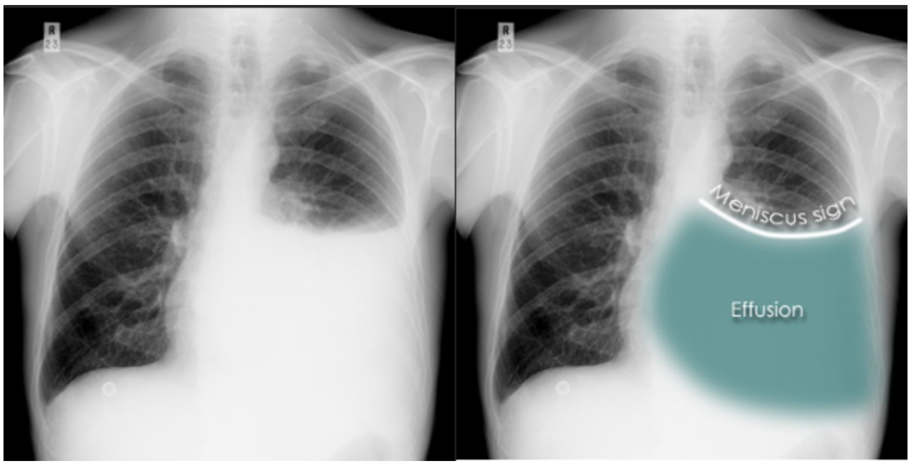


Fig. 2.8.: Example of Effusion on chest x-ray [37]

2.3.6. Emphysema

Description : Emphysema [11] is a lung condition that causes shortness of breath. In people with emphysema, the air sacs in the lungs (alveoli) are damaged, leading to difficulty in breathing. Often in the top of lungs.

Diagnosis on x-ray imaging : Here emphysema manifests as areas of low density (black) with thinning of the pulmonary vessels. The vessels are a lot darker than the normal pulmonary vessels.

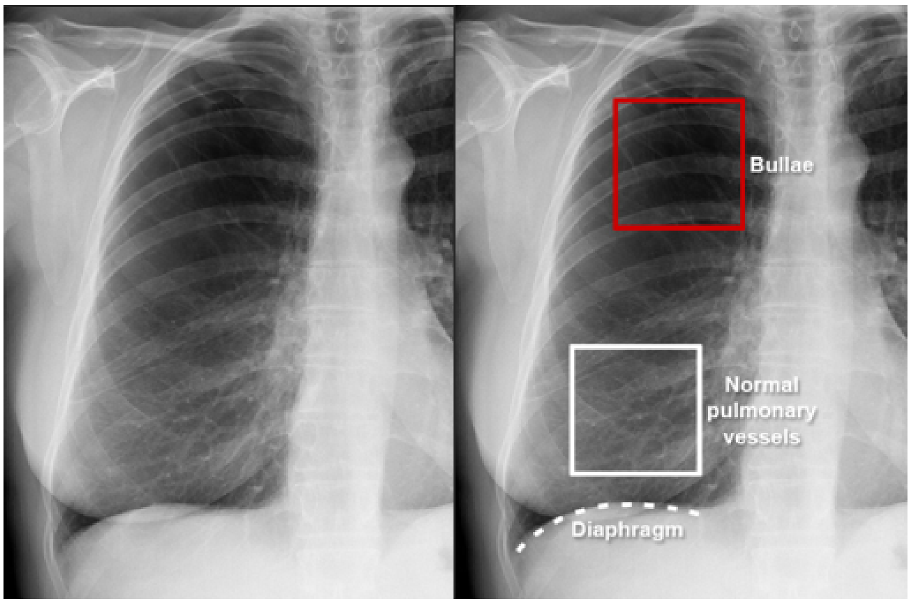


Fig. 2.9.: Example of Emphysema on chest x-ray [11]

2.3.7. Fibrosis

Description : Fibrosis [13], also known as fibrotic scarring, is a pathological wound healing in which connective tissue replaces normal parenchymal tissue leading to considerable tissue remodelling and the formation of permanent scar tissue.

Diagnosis on x-ray imaging : pulmonary fibrosis causes reticular shadowing of the lung peripheries, it may cause the contours of the heart to be less distinct or "shaggy".

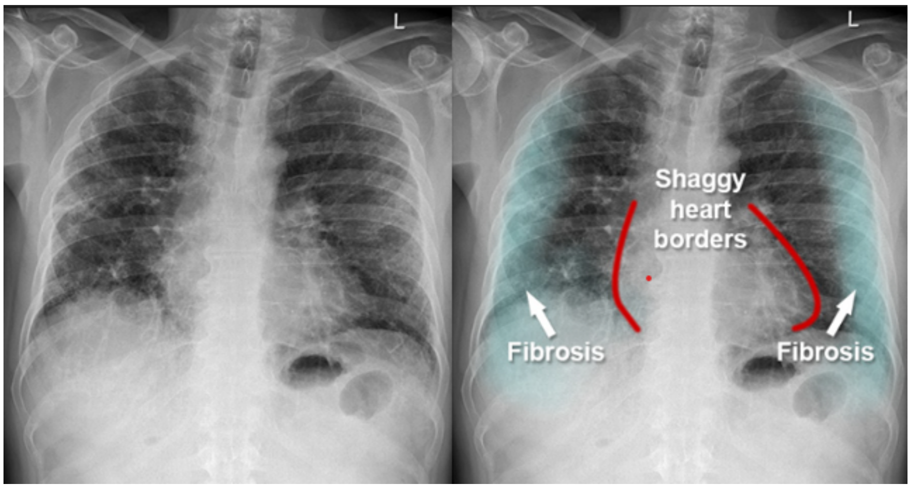


Fig. 2.10.: Example of Fibrosis on chest x-ray [13]

2.3.8. Infiltration

Description : In the context of lung diseases, infiltration [24] usually refers to the accumulation of cells or fluids in the lung tissues that are not normally present. This can be due to an infection, inflammation, or other disease processes.

Diagnosis on x-ray imaging : all of the white spots showing on the image on the right are caused when a substance denser than air (e.g., pus, oedema, blood, proteins, or cells) lingers within the lung parenchyma.

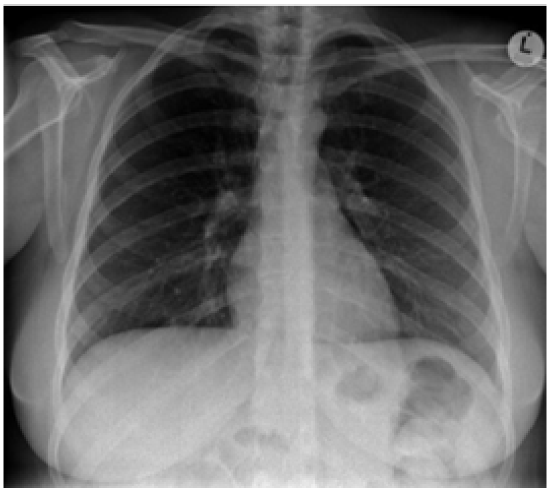


Fig. 2.11.: Example of healthy lungs on chest x-ray [21]

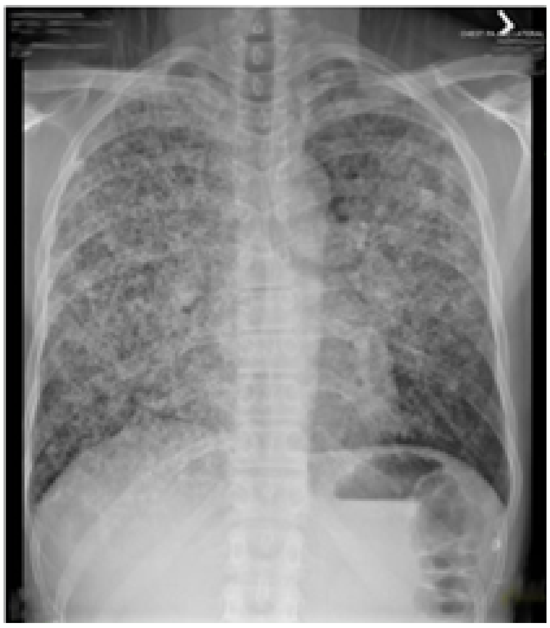


Fig. 2.12.: Example of Infiltration on chest x-ray [24]

2.3.9. Mass

Description : In the context of medicine, a mass [30] refers to a lump or growth that can occur in various parts of the body, including the lungs. It can be benign (non-cancerous) or malignant (cancerous), and its cause can range from infection to cancer.

Diagnosis on x-ray imaging : a large round area of increased density shows the presence of a mass in the region of the left hilum.

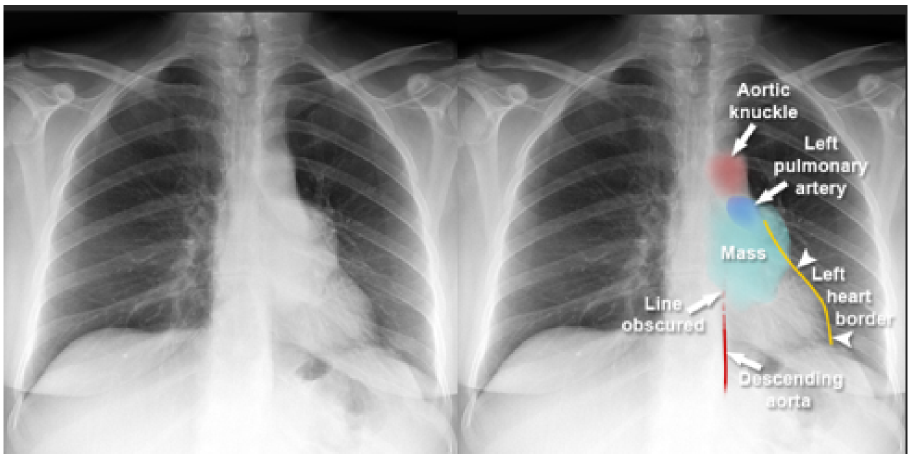


Fig. 2.13.: Example of Mass on chest x-ray [30]

2.3.10. Nodule

Description : A nodule [33] is a growth or lump that develops on or within the body. For example, it can develop beneath the skin, in the lungs, or on glands such as the thyroid. When a health condition presents with nodules, it is considered nodularity. Most of the time, nodules are tumoral or ganglions.

Diagnosis on x-ray imaging : the nodule on this image is clear and is located in the left upper lobe.

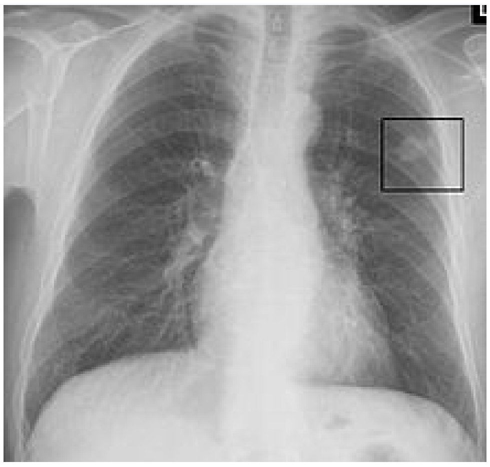


Fig. 2.14.: Example of Nodule on chest x-ray [33]

2.3.11. Pleural Thickening

Description : This [38] is a condition where the pleura (the thin membranes covering the lungs) become thickened. This is usually caused by inflammation of the pleura and subsequent scarring. It can be caused by several conditions including infection, asbestos exposure, and pleural effusion. The cause is often tumoral, it can also be caused by a pneumothorax drainage. Often seen at the top of the lungs.

Diagnosis on x-ray imaging : Pleural thickening is displayed as a shadowing on the right and as a loss of right lung volume.

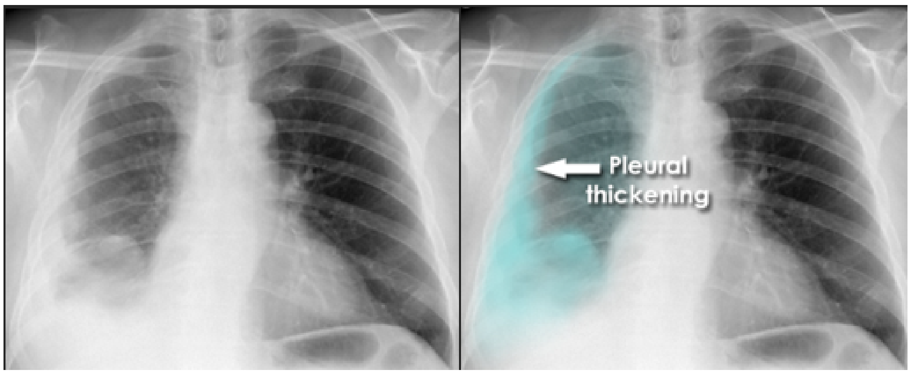


Fig. 2.15.: Example of Pleural Thickening on chest x-ray [38]

2.3.12. Pneumonia

Description : Pneumonia [39] is an infection that inflames the air sacs in one or both lungs. The air sacs may fill with fluid or pus, causing cough with phlegm or pus, fever, chills, and difficulty breathing.

Diagnosis on x-ray imaging : This chest X-ray shows an area of lung inflammation indicating the presence of pneumonia.

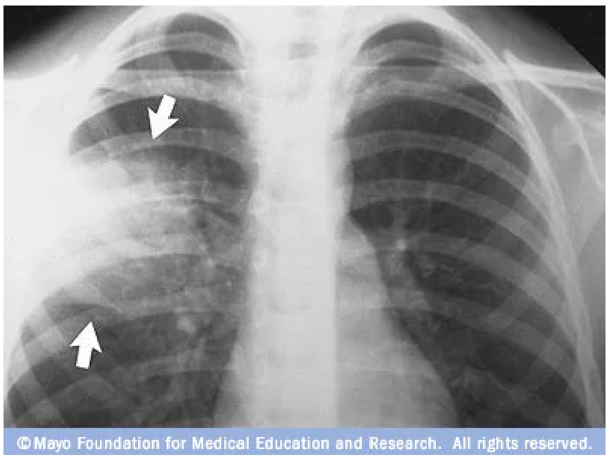

©Mayo Foundation for Medical Education and Research. All rights reserved.

Fig. 2.16.: Example of Pneumonia on chest x-ray [39]

2.3.13. Pneumothorax

Description : Also known as a collapsed lung, pneumothorax [40] occurs when air leaks into the space between your lung and chest wall. This air pushes on the outside of your lung and makes it collapse.

Diagnosis on x-ray imaging : Pneumothorax is displayed with a visible pleural edge (blue line).

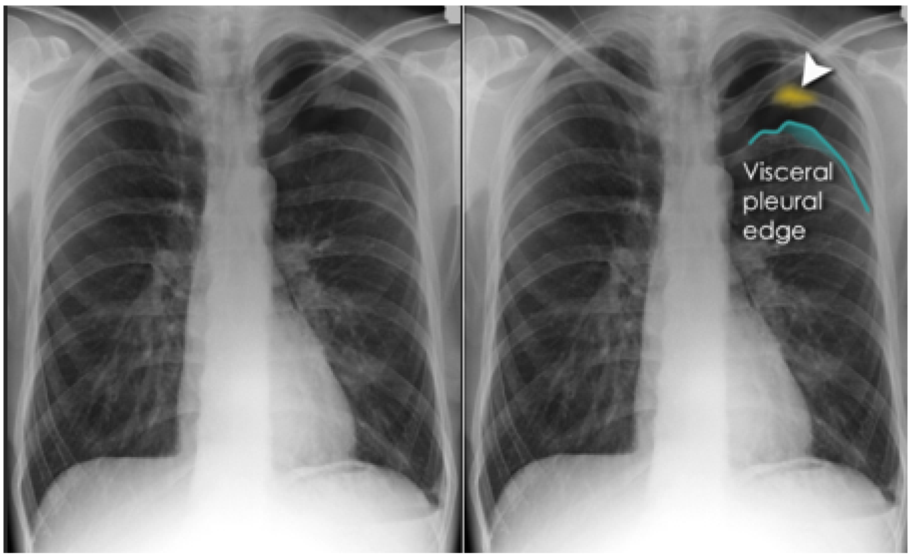


Fig. 2.17.: Example of Pneumothorax on chest x-ray [40]

2.3.14. Hernia

Description : A hernia [22] is a condition that occurs when an organ or fatty tissue squeezes through a weak spot in a surrounding muscle or connective tissue called fascia1. The most common types of hernia are inguinal (inner groin), incisional (resulting from an incision), femoral (outer groin), umbilical (belly button), and hiatal (upper stomach)

Diagnosis on x-ray imaging : Hiatus hernias may be very large, as is the case in this image. Seeing a gas/fluid level helps to make the diagnosis. The stomach can actually get as high as the heart because of hernias.

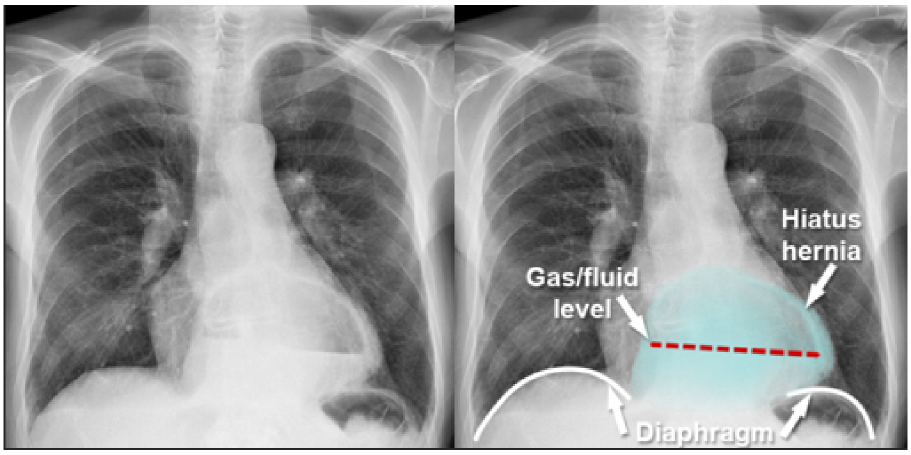


Fig. 2.18.: Example of Hernia on chest x-ray [22]

3 ChestVision - User demonstration

3.1. Description of the application and context

ChestVision is a web application that is designed to assist medical practitioners in analyzing x-ray medical imaging. In countries like Switzerland, quick access to healthcare professionals for health assessments is common. However, in many other regions, the scarcity of doctors severely limits access to healthcare. The need for fast and accurate diagnoses is even more critical in these under-resourced areas.

The development and deployment of applications like ChestVision in these regions could enable more people to receive diagnoses, thereby alleviating the workload of overburdened doctors. With this technology, a technician in medical radiology could capture the images, and the application would provide an initial diagnosis. A healthcare professional could then confirm or adjust the diagnosis as needed. This process would allow more people to receive diagnoses while freeing up valuable time for doctors.

Moreover, this technology has the potential to outperform medical doctors in terms of accuracy, as demonstrated by Google's Health[20] team. Their model, used in under-resourced countries, has helped prevent vision loss caused by diabetes.

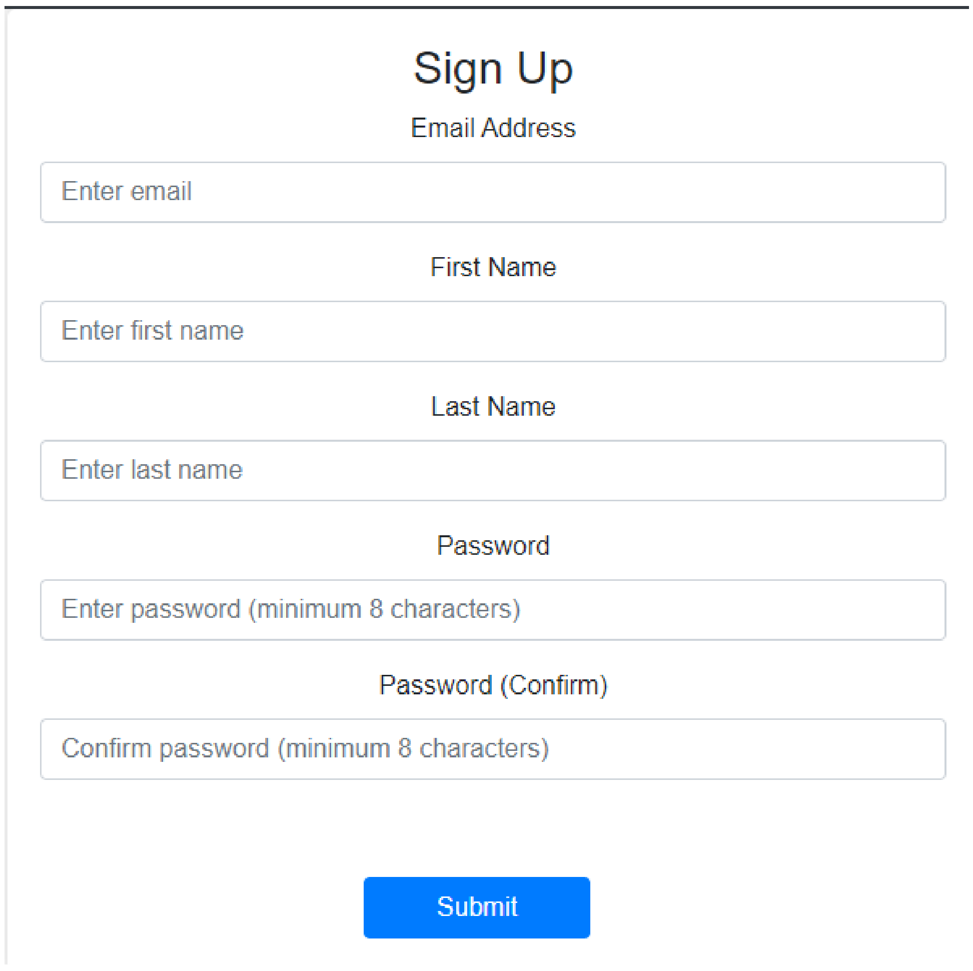
3.2. Description of the pages

3.3. Scenarios

In this chapter, we dive into the various functionalities of the ChestVision application. This exploration is facilitated through the use of annotated screenshots from the actual application along with textual descriptions. The explanations are designed to be accessible and not not necessitate any prior knowledge in computer science. This chapter serves as a user guide, detailing each scenario within the application and how to navigate them.

3.3.1. Sign-up & Login pages

Figure 3.1 shows a very straightforward sign-up page, you can sign-up using any email and password. For logging in, users simply enter the credentials created during sign-up. In the real-world environment, the sign-up page would not exist and only doctors would have access to this application and would be able to access the patients' information.


Sign Up

Email Address

Enter email

First Name

Enter first name

Last Name

Enter last name

Password

Enter password (minimum 8 characters)

Password (Confirm)

Confirm password (minimum 8 characters)

Submit

Fig. 3.1.: Sign Up Page

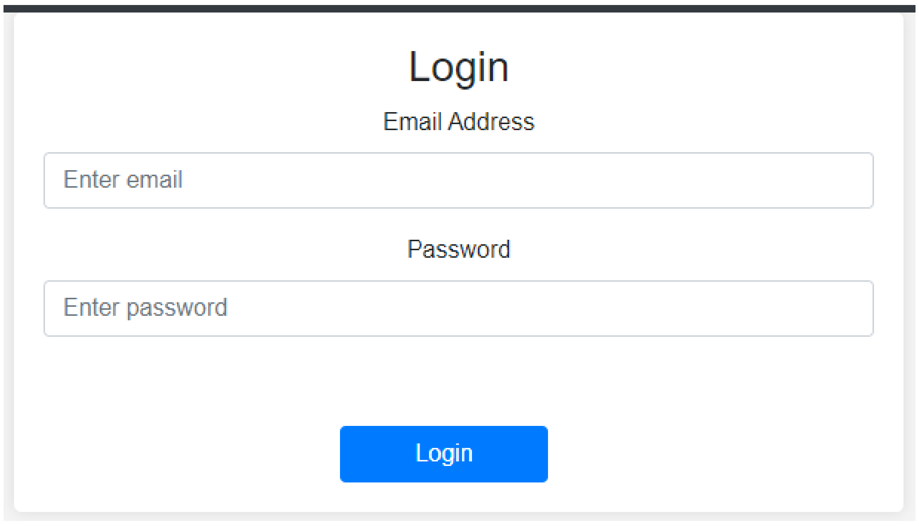


Fig. 3.2.: Login Page

3.3.2. Homepage

The homepage is composed of three parts. The first one, is the image below (Figure 3.3). It takes as input one or multiple thoracic x-ray images, and predicts which diseases are contained within the image. If the model does not estimate that the likelihood of any disease is above 30%, it will display "No Findings". This threshold was chosen to balance the trade-off between precision and recall in the model's predictions.

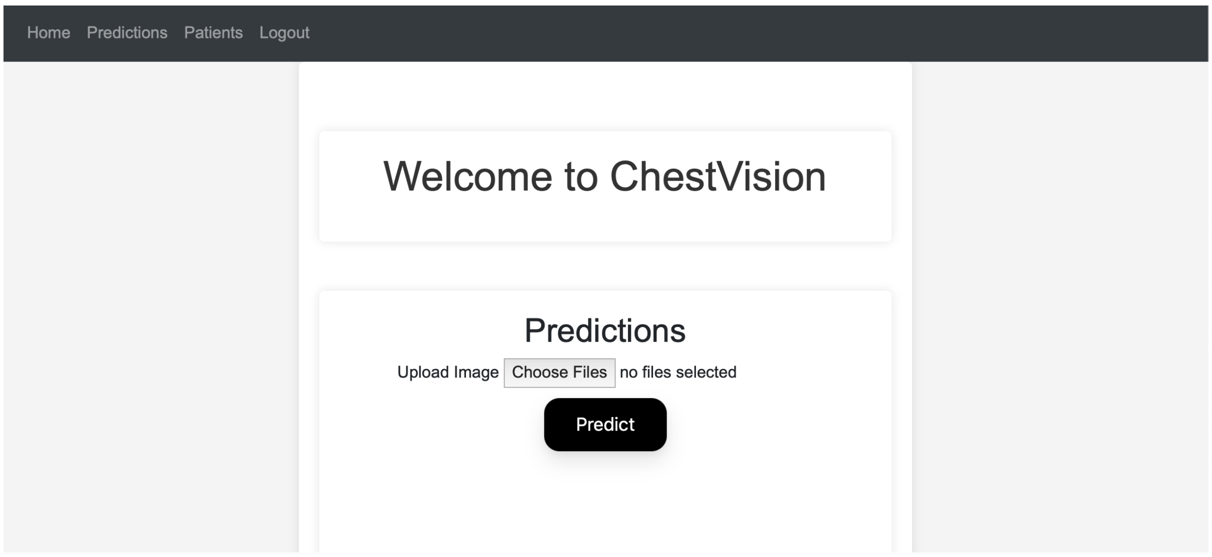
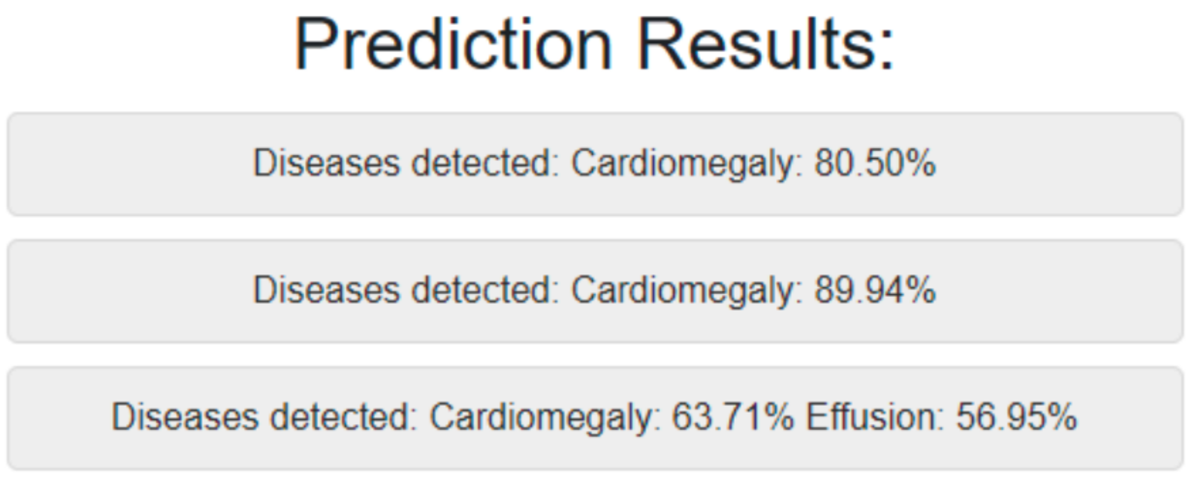


Fig. 3.3.: Homepage

The second part of this page is the results, as you can see with the example given in this image. This is an example with 3 images as input, it gives the disease(s) detected with their respective percentage of likelihood.


Prediction Results:

Diseases detected: Cardiomegaly: 80.50%

Diseases detected: Cardiomegaly: 89.94%

Diseases detected: Cardiomegaly: 63.71% Effusion: 56.95%

Fig. 3.4.: Homepage 2

Finally, the third part of the homepage is this image. Below the textual prediction of the model, you can find the actual images that were given as input along with their respective predictions. This was made using the matplotlib library in python and you can actually find these images in the folder myApp/static/assets/uploaded_images/predictionsi.png with "i" being the number of the image that was generated using the ith input image that was given to the model. All of these images are stored in that folder and are being displayed in the frontend code. Those images will be overwritten during the next prediction of the model, it always starts with "predictions1.png" so not all images may be overwritten depending on the number of images given to the model as input during the next prediction.
The main difference between this homepage and the "predictions" page that is showed in the next chapter, is that the homepage does not interact with the database and it is therefore not possible to add, or modify a patient from the homepage.

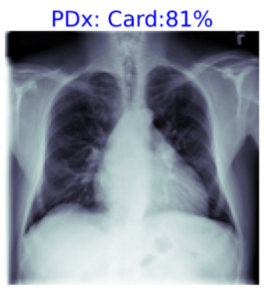


Fig. 3.5.: First result

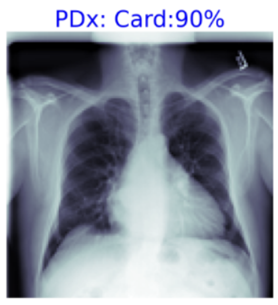


Fig. 3.6.: Second Result

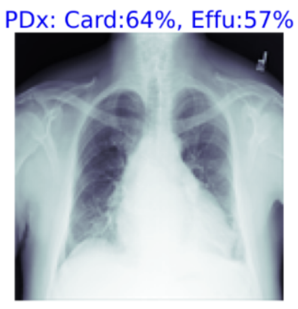
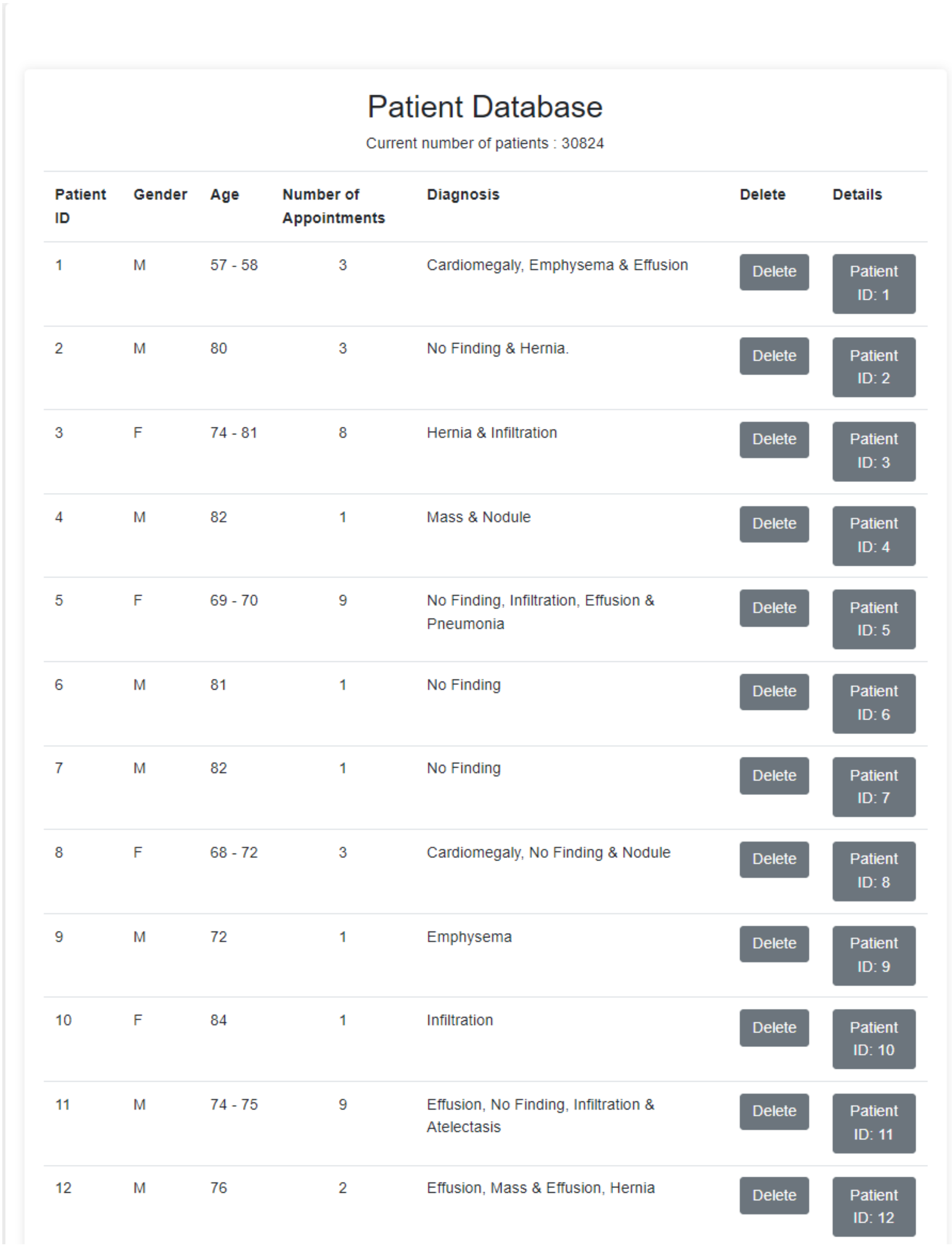


Fig. 3.7.: Third Result

3.3.3. Patients page

On the patients page, which is the database page, you can firstly see the number of patients that are currently stored in the database. Secondly, there is a table that displays 25 patients per page. You can go to the next or the previous page at the bottom of the table. There is also the possibility to delete the patient from the database using the delete button next to the patient's detail button.


Patient Database

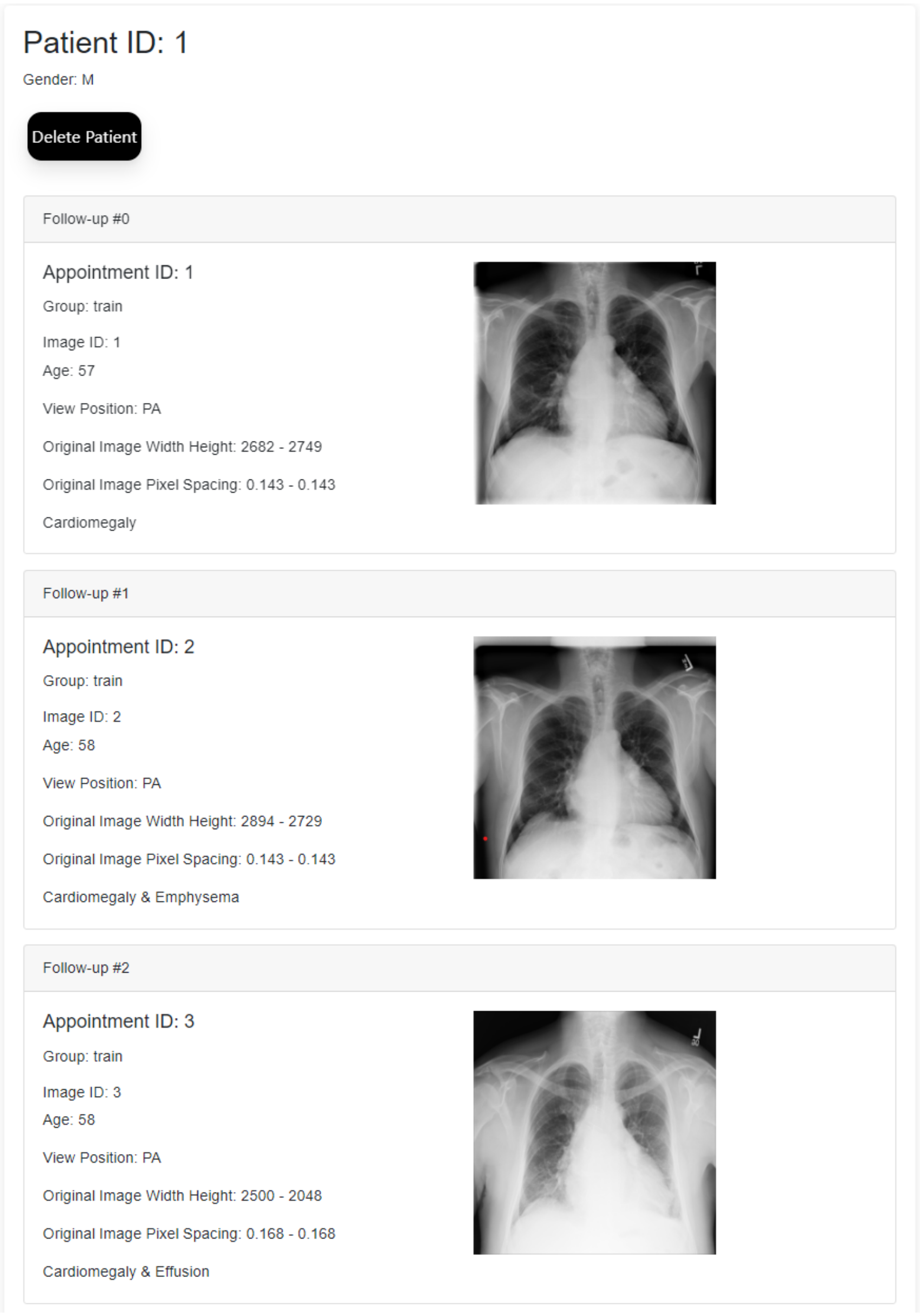
Current number of patients : 30824

Patient ID	Gender	Age	Number of Appointments	Diagnosis	Delete	Details
1	M	57 - 58	3	Cardiomegaly, Emphysema & Effusion	Delete	Patient ID: 1
2	M	80	3	No Finding & Hernia.	Delete	Patient ID: 2
3	F	74 - 81	8	Hernia & Infiltration	Delete	Patient ID: 3
4	M	82	1	Mass & Nodule	Delete	Patient ID: 4
5	F	69 - 70	9	No Finding, Infiltration, Effusion & Pneumonia	Delete	Patient ID: 5
6	M	81	1	No Finding	Delete	Patient ID: 6
7	M	82	1	No Finding	Delete	Patient ID: 7
8	F	68 - 72	3	Cardiomegaly, No Finding & Nodule	Delete	Patient ID: 8
9	M	72	1	Emphysema	Delete	Patient ID: 9
10	F	84	1	Infiltration	Delete	Patient ID: 10
11	M	74 - 75	9	Effusion, No Finding, Infiltration & Atelectasis	Delete	Patient ID: 11
12	M	76	2	Effusion, Mass & Effusion, Hernia	Delete	Patient ID: 12

Fig. 3.8.: Patients Page

3.3.4. Patient's detail page

On the patient's detail page, you also have the possibility to delete the patient from the database. All of the patient's information for each followup is displayed on this page including all of the image information too. The information given in the followups cannot be modified. It is however possible to add a followup for a patient, as will be explained in the "Adding a diagnosis to an existing patient" sub chapter in the following.


Patient ID: 1

Gender: M

Follow-up #0

Appointment ID: 1

Group: train

Image ID: 1

Age: 57

View Position: PA

Original Image Width Height: 2682 - 2749

Original Image Pixel Spacing: 0.143 - 0.143

Cardiomegaly

Follow-up #1

Appointment ID: 2

Group: train

Image ID: 2

Age: 58

View Position: PA

Original Image Width Height: 2894 - 2729

Original Image Pixel Spacing: 0.143 - 0.143

Cardiomegaly & Emphysema

Follow-up #2

Appointment ID: 3

Group: train

Image ID: 3

Age: 58

View Position: PA

Original Image Width Height: 2500 - 2048

Original Image Pixel Spacing: 0.168 - 0.168

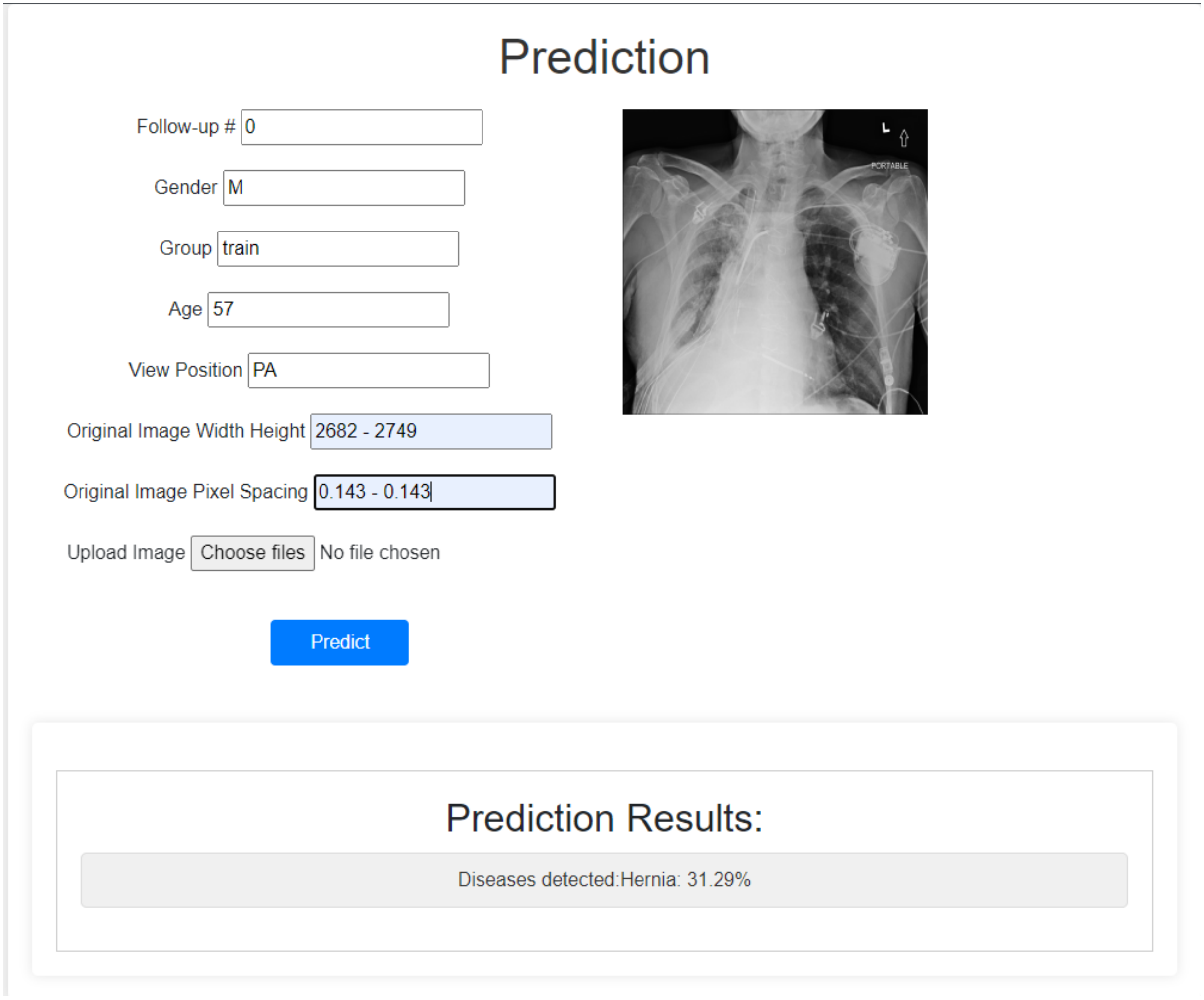
Cardiomegaly & Effusion

Fig. 3.9.: Patient's detail Page

3.3.5. Predictions page

Adding a new patient

On this page, you can type "0" in the Follow-up# field, to create a new patient. You can then input a single image, along with the patient and the image's information, all of this information is stored together in the appointment's table of the database. The rest of the page works the same as is showed in the homepage, except that here you only get the prediction result written down and the image is displayed on the top right.


Prediction

Follow-up # 0

Gender M

Group train

Age 57

View Position PA

Original Image Width Height 2682 - 2749

Original Image Pixel Spacing 0.143 - 0.143

Upload Image Choose files No file chosen

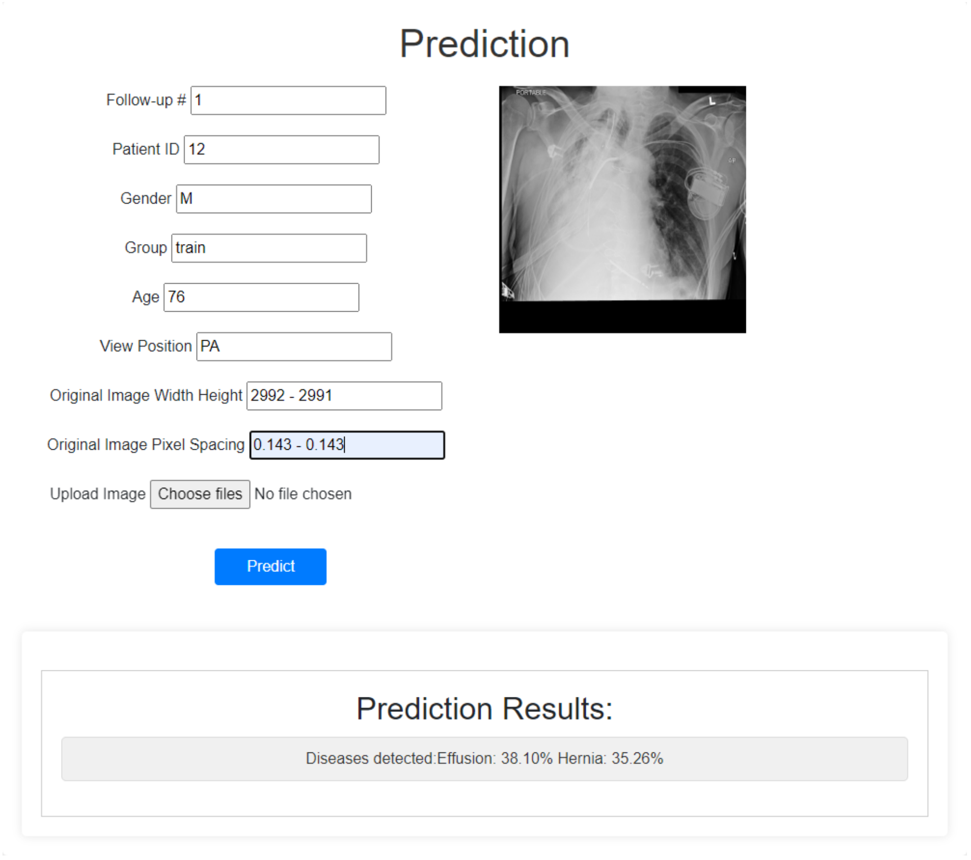
Prediction Results:

Diseases detected:Hernia: 31.29%

Fig. 3.10.: Adding a patient

Adding a diagnosis to an existing patient

On the same page, as you can see on the image below (Figure 3.11), you can also type a number bigger than 0 for the Follow-up#, this will add an additional input field for the Patient ID number. This way, you can choose a patient that already exists and add a follow-up to this patient along with the new image and its diagnostic.



Prediction

Follow-up #	1
Patient ID	12
Gender	M
Group	train
Age	76
View Position	PA
Original Image Width Height	2992 - 2991
Original Image Pixel Spacing	0.143 - 0.143
Upload Image	Choose files No file chosen

Predict

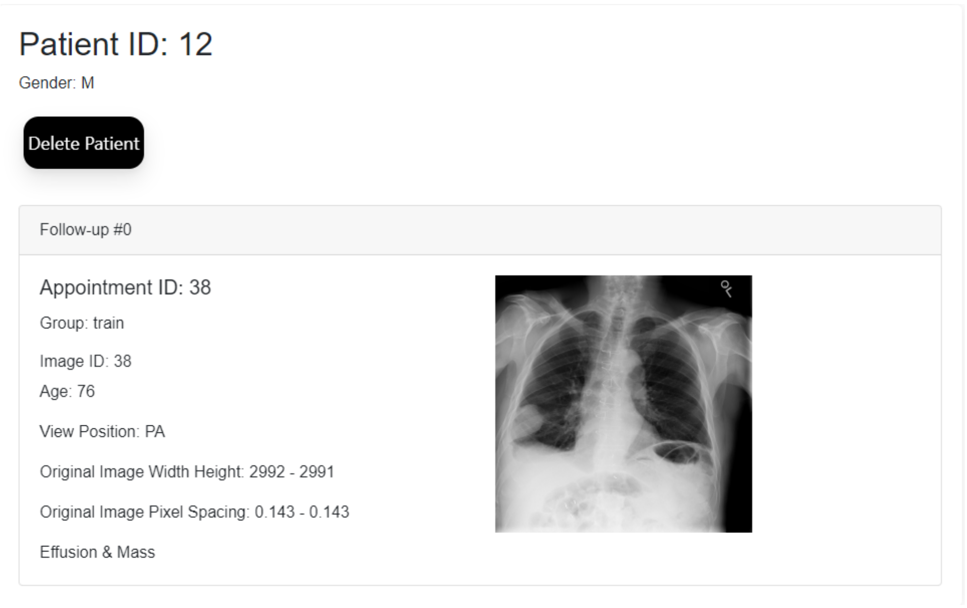
Prediction Results:

Diseases detected:Effusion: 38.10% Hernia: 35.26%

Fig. 3.11.: Adding a Diagnosis to a patient

As you can see from the two images that are displayed on the following page, this adds a follow-up with all of the given information and the image with its diagnostic to a specific patient, in this example, it was added to patient 12.

Before :



Patient ID: 12

Gender: M

Follow-up #0

Appointment ID: 38

Group: train

Image ID: 38

Age: 76

View Position: PA

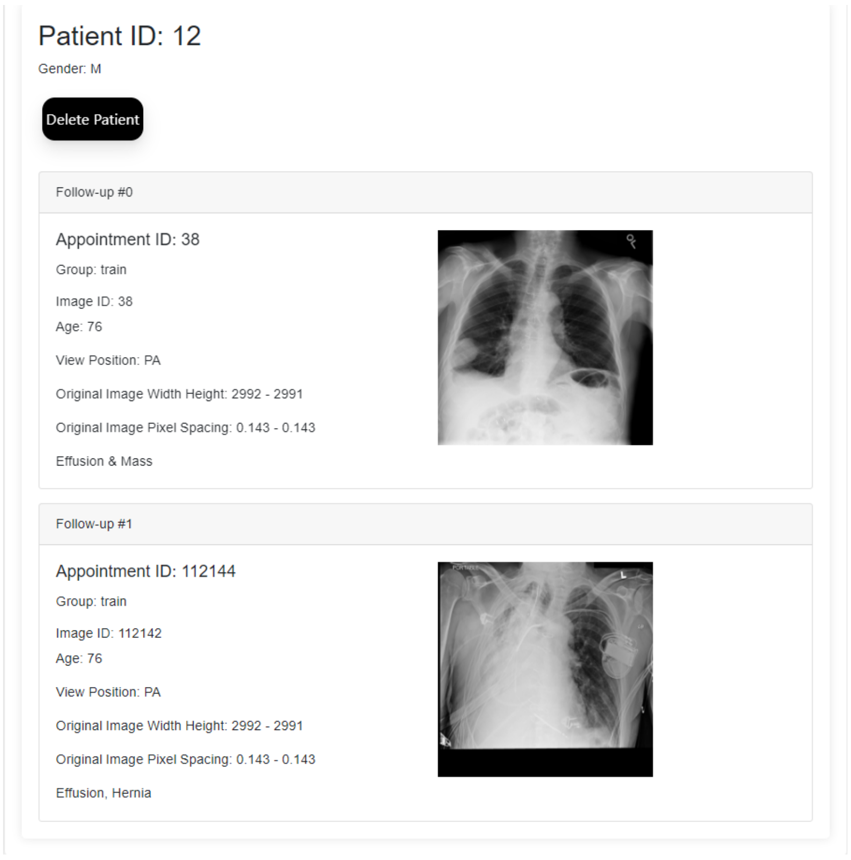
Original Image Width Height: 2992 - 2991

Original Image Pixel Spacing: 0.143 - 0.143

Effusion & Mass

Fig. 3.12.: Patient 12 before

After :



Patient ID: 12

Gender: M

Follow-up #0

Appointment ID: 38

Group: train

Image ID: 38

Age: 76

View Position: PA

Original Image Width Height: 2992 - 2991

Original Image Pixel Spacing: 0.143 - 0.143

Effusion & Mass

Follow-up #1

Appointment ID: 112144

Group: train

Image ID: 112142

Age: 76

View Position: PA

Original Image Width Height: 2992 - 2991

Original Image Pixel Spacing: 0.143 - 0.143

Effusion, Hernia

Fig. 3.13.: Patient 12 after

4
ChestVision - Technologies and architecture

4.1. Technologies

This chapter provides an overview of the key technologies employed in the development of the application. These technologies, ranging from web frameworks to machine learning libraries, form the backbone of the application.

4.1.1. Flask

Flask [14] is a lightweight and flexible web framework for Python, designed to get applications up and running quickly with minimal setup. In this application, Flask is used to handle the routing, request processing, and template rendering. It provides the structure for defining endpoints such as user authentication and patient management, facilitating the core operations of the application.

4.1.2. Flask-SQLAlchemy

Flask-SQLAlchemy [17] is an extension for Flask that integrates SQLAlchemy, a powerful Object-Relational Mapper (ORM) for database interactions. It simplifies database management by allowing the use of Python objects to interact with the database instead of writing raw Structured Query Language (SQL). In this application, Flask-SQLAlchemy is used to define and manage the Patient, Appointment, Image, Finding, and User models, making database operations straightforward and efficient.

4.1.3. Flask-Login

Flask-Login [15] is a Flask extension that handles user session management, including login, logout, and session persistence. It integrates with the User model to manage authentication, ensuring secure access to protected routes within the application. This extension is important for managing user authentication and maintaining session integrity.

4.1.4. Flask-Migrate

Flask-Migrate [16] is an extension that integrates Alembic with Flask-SQLAlchemy for managing database migrations. It helps in evolving the database schema over time without losing data. In the application, Flask-Migrate is used to handle changes to the database schema, such as adding new tables or modifying existing ones, ensuring the database structure remains up-to-date with application requirements.

4.1.5. Werkzeug Security

Werkzeug [48] is a comprehensive Web Server Gateway Interface (WSGI) web application library that includes utilities for secure password hashing and verification. In this application, Werkzeug Security is used to hash and verify user passwords, ensuring that credentials are stored securely and authentication processes are robust.

4.1.6. Keras and Tensorflow

Keras [28] is a high-level neural networks Application Programming Interface (API) that runs on top of TensorFlow [46], a flexible and comprehensive machine learning framework. These libraries are used in the application to load and run the machine learning model (modelx2.h5) for image prediction. They provide the tools necessary for preprocessing images and making predictions, leveraging deep learning techniques.

4.1.7. Additional Python libraries

- JavaScript Object Notation (JSON): Used for handling JSON data in API responses and requests, facilitating data exchange between the server and client.
- NumPy [34]: A fundamental package for numerical operations and array manipulations, essential for processing image data.
- Open Computer Vision Library (OpenCV) [35]: An open-source computer vision library used for image processing tasks, such as reading and transforming images before prediction.
- Matplotlib [31]: A plotting library used for creating visualizations, which can aid in analyzing and presenting data.
- Keras Preprocessing: Provides utilities for preparing image data before feeding it into the machine learning model, ensuring consistency in data input.
- Werkzeug Utils: Additional utility functions from the Werkzeug library used for various helper tasks throughout the application.
- Pandas [36]: A powerful data manipulation and analysis library, used for handling and analyzing structured data.
- Seaborn [43]: A statistical data visualization library based on Matplotlib, used for creating attractive and informative statistical graphics.
- Holoviews [23]: A high-level data visualization library that makes it easier to visualize large and complex datasets.
- Bokeh [5]: A visualization library that provides interactive plots and dashboards. Used in conjunction with Holoviews to display interactive visualizations in notebooks.
- Keras Callbacks (EarlyStopping): Used to monitor training and stop early when the performance metric has stopped improving, preventing overfitting.
- Glob [19]: A library for finding all the pathnames matching a specified pattern, used for handling file operations.
- Itertools [25] (chain): A library providing functions that create iterators for efficient looping, used for chaining multiple iterables together.
- Scikit-learn [42]: A machine learning library used for various tasks, including calculating Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) scores to evaluate the performance of classification models.

4.2. Architecture

4.2.1. Overview

The architecture of this application is designed to seamlessly integrate various components and deliver a robust and maintainable system. It leverages a combination of frontend and backend technologies to create an efficient workflow for handling user authentication, data processing, and machine learning predictions. This chapter provides a detailed overview of the application's architecture, including its structure, components, and the interactions between them.

4.2.2. Application structure

The application is organized into several key folders and files, each serving a distinct purpose. The main components include the Flask application, database, machine learning model, and static assets. Below is a breakdown of the folder structure:

- myApp
  - Documentation : contains documentation files, such as the data entry (the patients' information) Comma-Separated Values (CSV) file that can be found on the dataset's web page along with a JSON version of the same file that I made.
  - Instance : contains the database file that was generated with SQLite.
  - Models : contains "modelh2x.h5" which is the machine learning model that was trained using the code provided on this application's Github repository.
  - Website : contains the main flask application modules.
    - static : contains the static files like "styles.css" and all of the images.
    - templates : contains the HyperText Markup Language (HTML) templates and web pages.
    - __init__.py : initializes the Flask application.
    - auth.py : manages the authentication routes.
    - models.py : defines the dabase tables.
    - views.py : defines the main application routes and prediction logic with the image management for the model's input.
  - main.py : the entry point for running the application.
  - database_script.py : script to populate the database.

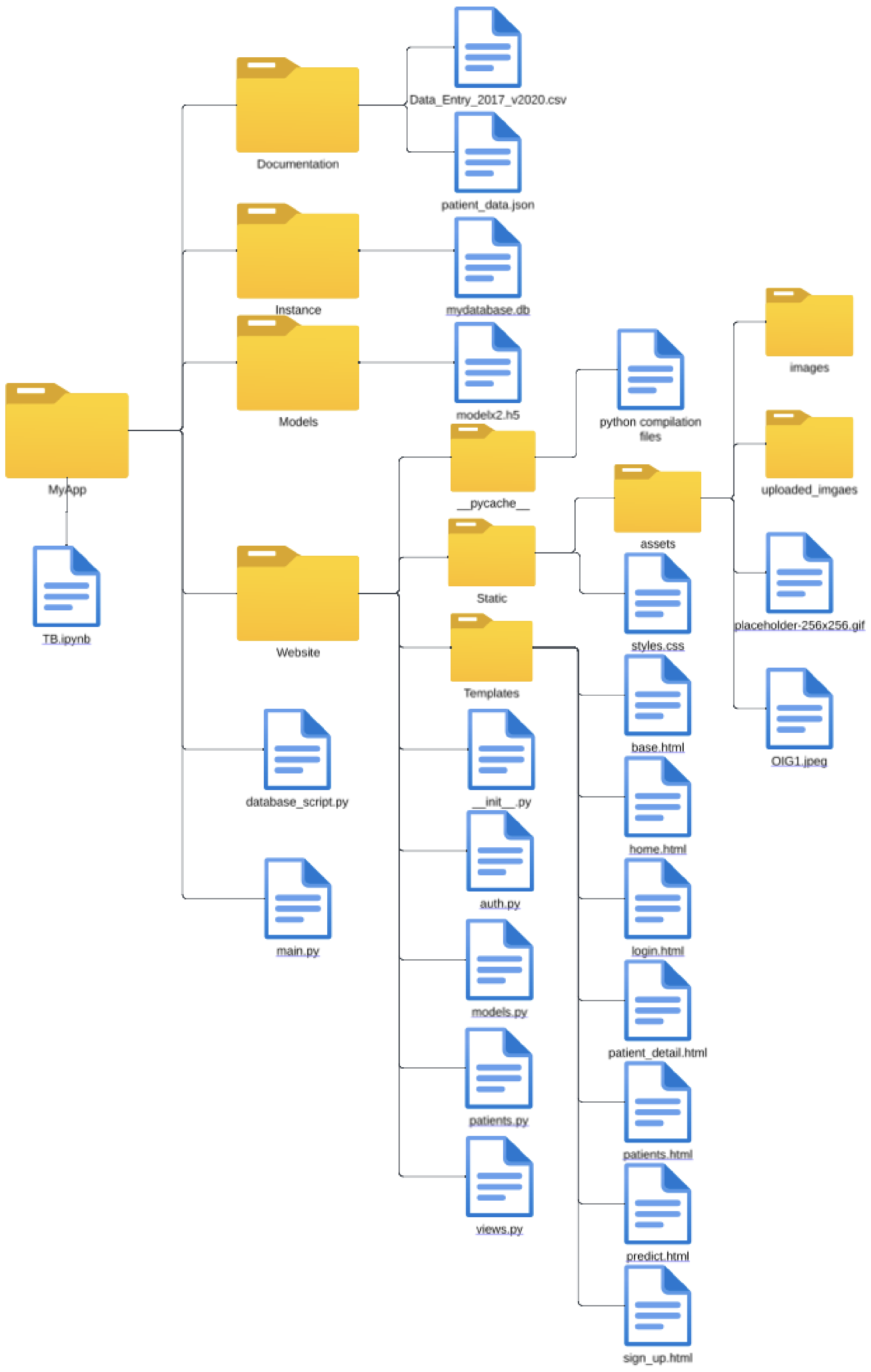


Fig. 4.1.: Folder structure diagram

4.2.3. Key components

- Frontend
  - HTML/Cascading Style Sheets (CSS)/JavaScript (JS) [26]: Used to create the user interface, with Jinja2 for template rendering.
  - Bootstrap: Utilized for responsive design and styling.
- Backend
  - Flask: Serves as the web framework, handling routing, request processing, and template rendering.
  - Flask_SQLAlchemy: An ORM for managing database interactions.
  - Flask_Login: Manages user authentication and session handling.
  - Flask_Migrate: Handles database migrations for schema changes.
- Database
  - SQLite: Used for storing application data, managed via Flask_SQLAlchemy.
- Model
  - TensorFlow/Keras: Libraries used to load and run the machine learning model for predictions.
  - OpenCV (CV2): Used for image preprocessing before feeding data to the ML model.
  - NumPy: Utilized for numerical operations on image data.
- Security
  - Werkzeug.Security: Provides utilities for hashing passwords and other security-related functions.

4.2.4. Routing and Functionality

The application routes are defined in "views.py" and "auth.py", handling various user interactions and API calls.

- "views.py" routes:
  - "/": Redirects to the home page.
  - "/delete-patient": Deletes a patient record.
  - "/predict": Handles the prediction used on the "Predictions" page. It preprocesses the image, then makes a prediction using the model and then saves the image, and the prediction along with the patient and the image's information in the database.
  - "/predictions": Handles the predictions of the "Home" page of the web application. It also preprocesses the images and makes predictions, then it generates a graph with the image and the result but it does not interact with the database.
  - "/patients": Displays a list of patients on the "Patients" page.

- "/patients/<int:patient_id>": Shows details for a specific patient when you click on "Patient ID: <patient_number>" in the "Details" column of the "Patients" page.
- "/get_patient/<int:patient_id>": Fetches patient data in the database to display it in the patient_detail page.

- "auth.py" routes:
  - "/login": Logs the user in the application.
  - "/logout": Logs the user out.
  - "/sign-up": Signs a new user up.

4.2.5. Database design

The database schema consists of several tables, each with a specific purpose and relationships to other tables.

Serialization methods

Serialization methods in each table convert SQLAlchemy objects to JSON-serializable dictionaries. This ensures that data can be easily transferred between the server and client.

Tables

- Patient: the table name is "patients", it has the columns "id" and "gender", it contains a one-to-many relationship with the "appointments" table and a serialization method.
- Appointment: the table name is "appointments", it contains the columns "id", "patient_id", "follow_up_number" and "group", it has a one-to-one relationship with the "images" table and has a serialization method.
- Image: the table name is "images", it contains the columns "id", "appointment_id", "image_index", "patient_age", "view_position", "original_image_width_height" and "original_image_pixel_spacing". It has a one-to-one relationship with the "findings" table and has a serialization method as well.
- Finding: the table name is "findings", it contains the columns "id", "image_id", "finding_label". It also has a serialization method.
- User: the table's name is "users" and contains the columns "id", "email", "password", "first_name", "last_name". It also uses "UserMixin" from Flask_Login for user session management.

Database diagram

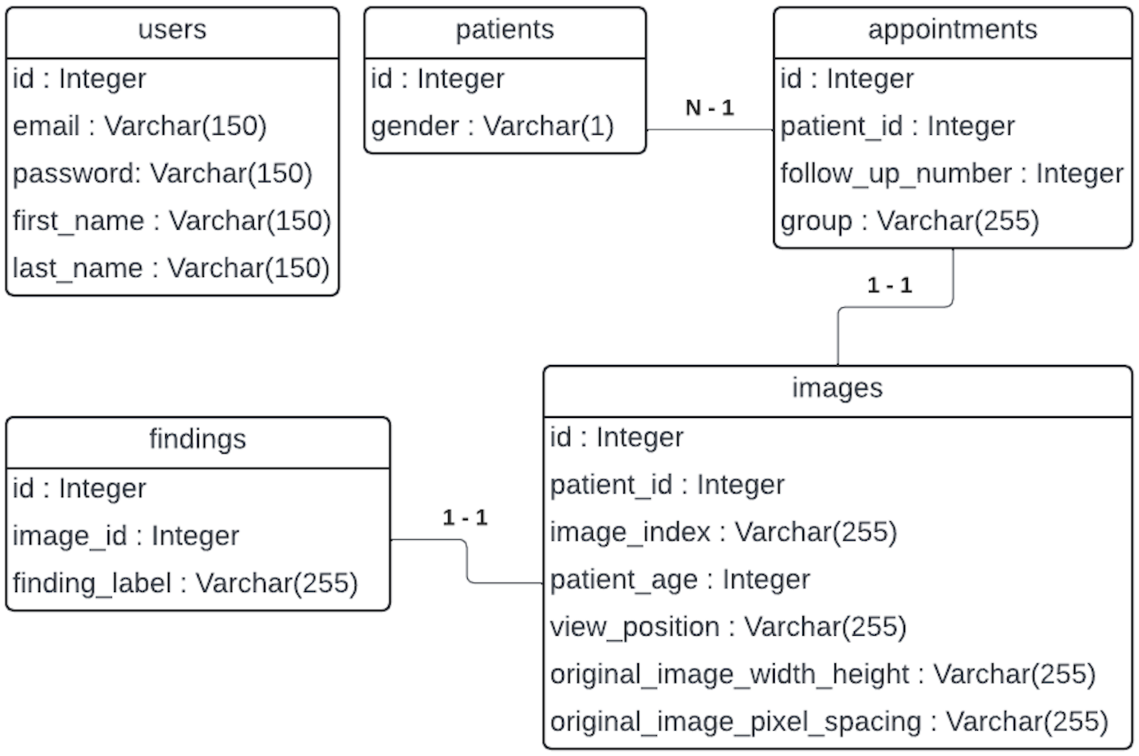


Fig. 4.2.: Database Diagram

As you can see, this database has a one-to-many relationship between patients and appointments, meaning a patient can have multiple appointments, but one appointment can only have one patient. It also has two one-to-one relationships: between the tables appointments and images, and images and findings. This structure implies that for each appointment, there is one image taken, and each image has one corresponding finding (one finding can be multiple diseases, one disease, or "No Finding"; findings are stored in a one-dimensional array).

This architecture makes it easier to manage and query the database. Keeping these tables separate allows us to maintain clear and distinct records for each entity, ensuring that the data is organized and easily retrievable. This design also allows for future improvements, such as enabling multiple images per appointment. This way, the structure can be easily adapted.

- Patient: Contains patient demographic information and has a one-to-many relationship with the appointments table.
- Appointment: Records details about each appointment, linking to a single patient and image.
- Image: Stores image data and metadata, linked to a specific appointment.
- Finding: Contains the results of the image analysis, associated with a specific image.
- User: Manages user credentials and authentication details.

4.2.6. Security measures

The application employs a few security measures to protect user data and ensure secure operations.

- Password Hashing: Passwords are hashed using Werkzeug.Security to ensure they are stored securely.
- User Authentication: Flask_Login manages user sessions, ensuring that only authenticated users can access certain routes and functionalities.

4.2.7. Local deployment

For local deployment, all components reside on the same machine, simplifying development and testing but it was not deployed for this work. The entire app can be cloned from the ChestVision [7] Github repository.

Local deployment diagram

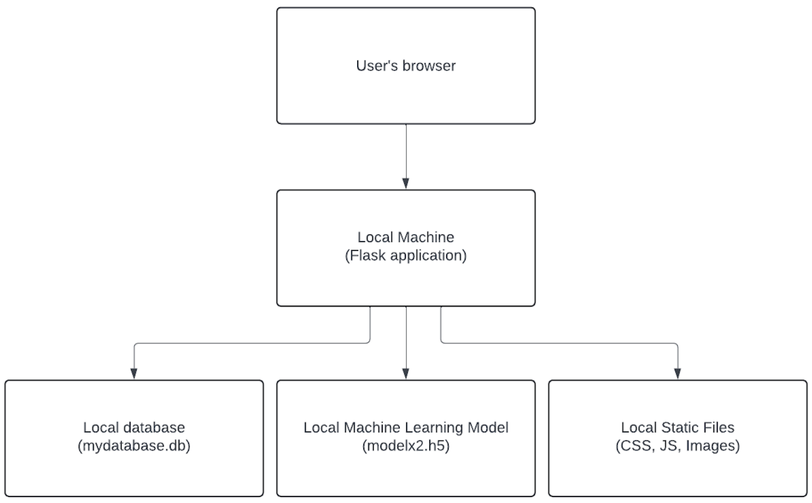


Fig. 4.3.: Deployment Diagram

- User's Browser: Interacts with the application via HyperText Transfert Protocol (HTTP)/HyperText Transfert Protocol Secure (HTTPS) on the local machine.
- Local Machine: Hosts the Flask application, database, and ML model.
- Local Database: SQLite database managed by Flask_SQLAlchemy.
- Local ML Model: TensorFlow/Keras model loaded and used for predictions.
- Local Static Files: CSS, JS, and images served locally.

5 Machine Learning Model implementation

5.1. Data preparation

This chapter covers the first part of implementing the machine learning model, the data preparation. Many parts of the code were inspired by various authors. [45] [29] [18] [9]

5.1.1. Data downloading and resizing

The data consists of 112'120 images of 1024x1024 pixels. First, I downloaded all of the images using the code found on the Dataset's website.

```
def downloadImagesZips():
    # URLs for the zip files
    links = [
        'https://nihcc.box.com/shared/static/vfk49d74nhbxq3nqjg0900w5nvkorp5c.gz',
        'https://nihcc.box.com/shared/static/i28rlmbvmfjbl8p2n3ril0pptcmcu9d1.gz',
        'https://nihcc.box.com/shared/static/f1t00wrtdk94satdfb9olcolqx20z2jp.gz',
        'https://nihcc.box.com/shared/static/0aowwzs5lhjrceb3qp67ahp0rd1l1etg.gz',
        'https://nihcc.box.com/shared/static/v5e3goj22zr6h8tzualxfsqlqaygfbsn.gz',
        'https://nihcc.box.com/shared/static/asi7ikud9jwnkrnkj99jnpfkjdes7l6l.gz',
        'https://nihcc.box.com/shared/static/jn1b4mw4n6lnh74ovmcjb8y48h8xj07n.gz',
        'https://nihcc.box.com/shared/static/tvpxmn7qyrgl0w8wfh9kqfjskv6nmm1j.gz',
        'https://nihcc.box.com/shared/static/upyy3ml7qdumlgk2rfcvlb9k6gvqq2pj.gz',
        'https://nihcc.box.com/shared/static/l6nilvfa9cg3s28tqv1qc1olm3gnz54p.gz',
        'https://nihcc.box.com/shared/static/hhq8fkdgvcari67vfhs7ppg2w6ni4jze.gz',
        'https://nihcc.box.com/shared/static/ioqwiy20ihqwyr8pf4c24eazhh281pbu.gz'
    ]
    for idx, link in enumerate(links):
        fn = 'images/Zipped/images_%02d.tar.gz' % (idx+1)
        print('downloading '+fn+'...')
        urllib.request.urlretrieve(link, fn) # download the zip file
    print("Download complete. Please check the checksums")
```

List. 5.1: Zip Files Downloading

After which, I reduced the size of each image to 256x256 pixels so that the image folder was easier to store for the application. The sized down version was still 9.67GigaBytes (GB), down from more than 40GB. These images need to be downloaded in order to firstly, train the model and to be able to display these images in the application, on each patient's page.

```
import cv2
def resizeFiles():
    dim = (256,256)
    for file in os.listdir('images/images'):
        if file.endswith('.png'):
            img = cv2.imread(f'images/images/{file}')
            #cv2.INTER_AREA is a resampling method notably used for shrinking images
            resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
            cv2.imwrite(f'myApp/website/static/assets/images/{file}', resized)
    print('Done')
```

List. 5.2: Image resizer method

5.1.2. Data analysis

To verify that the data was correctly downloaded, here (listing 5.3) is a code snippet that counts the number of images in the CSV file and counts the numbers of images found in my folders. The print method of this code returns "Scans found 112120, Total Headers 112120" which confirms that everything has gone according to plan so far.

```
import pandas as pd
import glob
import os
all_xray_df = pd.read_csv('./Documentation/Data_Entry_2017_v2020.csv')
all_image_paths = {os.path.basename(x): x for x in
                   glob.glob(os.path.join('.', 'images', 'images', '*.png'))}
print('Scans found:', len(all_image_paths), ', Total Headers', all_xray_df.shape[0])
all_xray_df['path'] = all_xray_df['Image Index'].map(all_image_paths.get)
all_xray_df.sample(3)
```

List. 5.3: Download Verification Code

Then, we need to do a bit of data analysis to understand the dataset and be able to process it to feed it to the model for training. First, let's see the number of diagnosis using a graph.

```python
import numpy as np
import matplotlib.pyplot as plt
label_counts = all_xray_df['Finding Labels'].value_counts()[:15]
fig, ax1 = plt.subplots(1,1,figsize = (12, 8))
ax1.bar(np.arange(len(label_counts))+0.5, label_counts)
ax1.set_xticks(np.arange(len(label_counts))+0.5)
_ = ax1.set_xticklabels(label_counts.index, rotation = 90)
```

List. 5.4: Distribution of findings

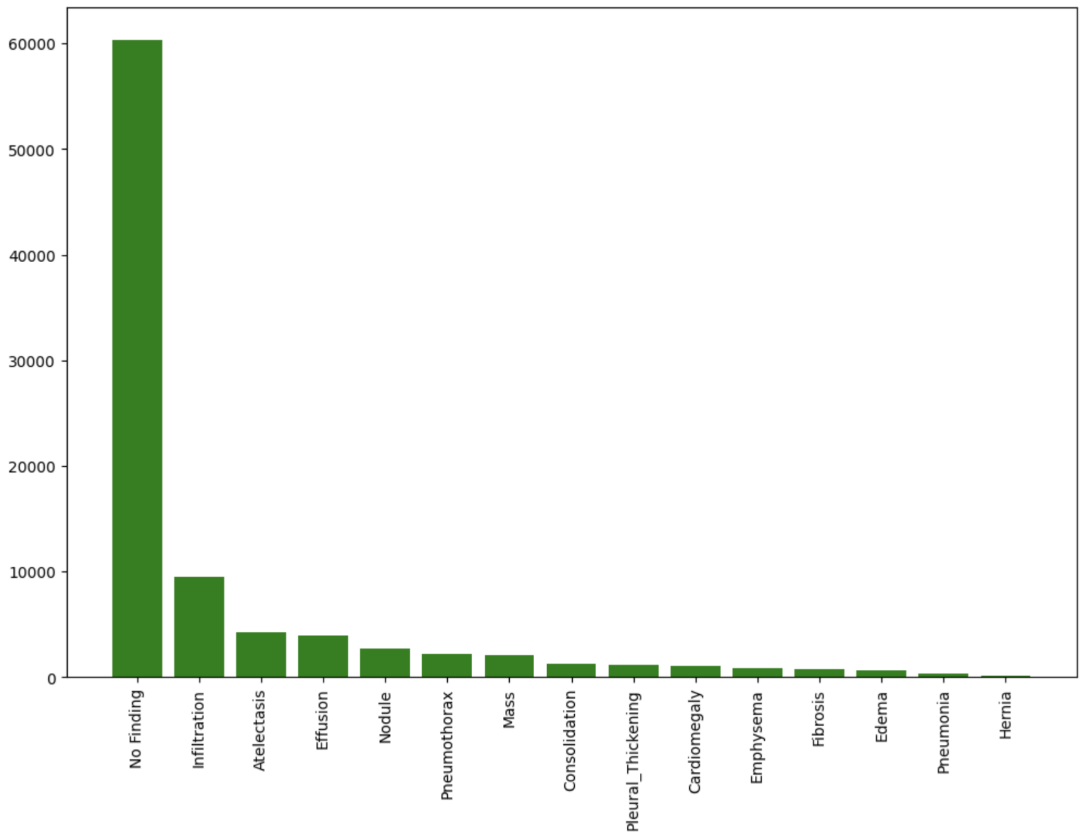


Fig. 5.1.: Findings' distribution

As you can see, the most striking feature of the chart is the overwhelming number of instances labeled as "No Finding". This shows that there is a substantial imbalance in the dataset. Such an imbalance can affect the performance of our machine learning model as it may get biased towards predicting the majority class.

Let's remove the No Findings images and draw this graph again.

```python
label_counts = all_xray_df[~all_xray_df['Finding Labels'].str.contains('|', regex=
    False)]['Finding Labels'].value_counts()[:15]
label_counts = label_counts[~label_counts.index.str.contains('No Finding')]
fig, ax1 = plt.subplots(1,1,figsize = (12, 8))
ax1.bar(np.arange(len(label_counts))+0.5, label_counts, color = 'green')
ax1.set_xticks(np.arange(len(label_counts))+0.5)
```

```
_ = ax1.set_xticklabels(label_counts.index, rotation = 90)
```

List. 5.5: Distribution of findings second graph

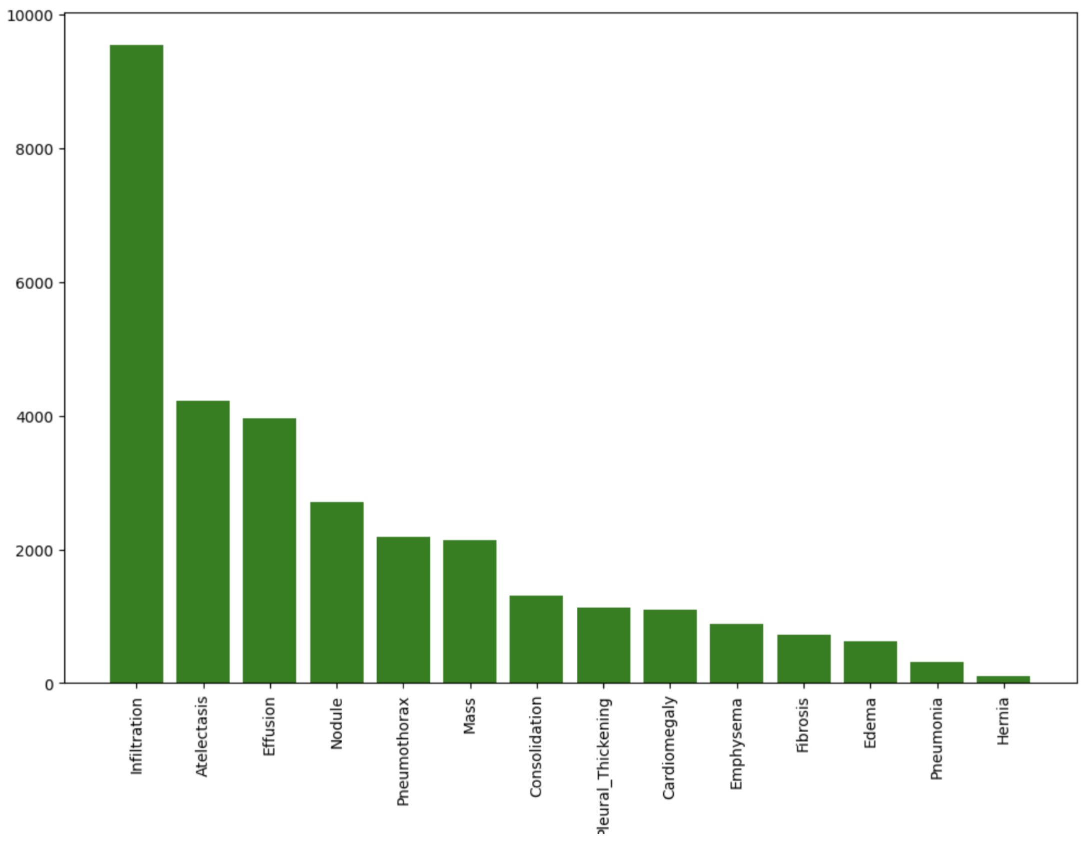


Fig. 5.2.: Findings' distribution without "No Finding"

Here we see that the dataset is still pretty imbalanced, especially the finding "Hernia" has a very low number of occurrences. We need to balance the dataset and implement a one thousand case minimum, which will remove "Hernia" since its count is lower than a thousand. We also create binary columns for each label, where the presence of the label is indicated by "1" and its absence by "0". Then, we calculate weights based on the number of findings for each disease with the addition of a small constant to ensure non-zero weights, we also normalize those weights so that they add up to one. These weights are used to resample the dataset to a sample of 40'000 images. Finally, we calculate the frequency of each label in the resampled collection and convert it to a percentage and draw a new plot.

```
from itertools import chain
all_xray_df['Finding Labels'] = all_xray_df['Finding Labels'].map(lambda x: x.replace(
    'No Finding', ''))
all_labels = np.unique(list(chain(*all_xray_df['Finding Labels'].map(lambda x: x.split
    ('|')).tolist())))
all_labels = [x for x in all_labels if len(x)>0]
print('All Labels ({}): {}'.format(len(all_labels), all_labels))
for c_label in all_labels:
    if len(c_label)>1: # leave out empty labels
        all_xray_df[c_label] = all_xray_df['Finding Labels'].map(lambda finding: 1.0 if
            c_label in finding else 0)
```

```
all_xray_df.sample(3)
all_labels

# keep at least 1000 cases
MIN_CASES = 1000
all_labels = [c_label for c_label in all_labels if all_xray_df[c_label].sum()>
    MIN_CASES]
print('Clean Labels ({})'.format(len(all_labels)),
      [(c_label,int(all_xray_df[c_label].sum())) for c_label in all_labels])

# since the dataset is very unbiased, we can resample it to be a more reasonable
    collection
# weight is 0.1 + number of findings
sample_weights = all_xray_df['Finding Labels'].map(lambda x: len(x.split('|')) if len(
    x)>0 else 0).values + 4e-2
sample_weights /= sample_weights.sum()
all_xray_df = all_xray_df.sample(40000, weights=sample_weights)
label_counts = all_xray_df['Finding Labels'].value_counts()[:15]

label_counts = 100*np.mean(all_xray_df[all_labels].values,0)
fig, ax1 = plt.subplots(1,1,figsize = (12, 8))
ax1.bar(np.arange(len(label_counts))+0.5, label_counts)
ax1.set_xticks(np.arange(len(label_counts))+0.5)
ax1.set_xticklabels(all_labels, rotation = 90)
ax1.set_title('Adjusted Frequency of Diseases in Patient Group')
_ = ax1.set_ylabel('Frequency (%)')
```

List. 5.6: Resampling data

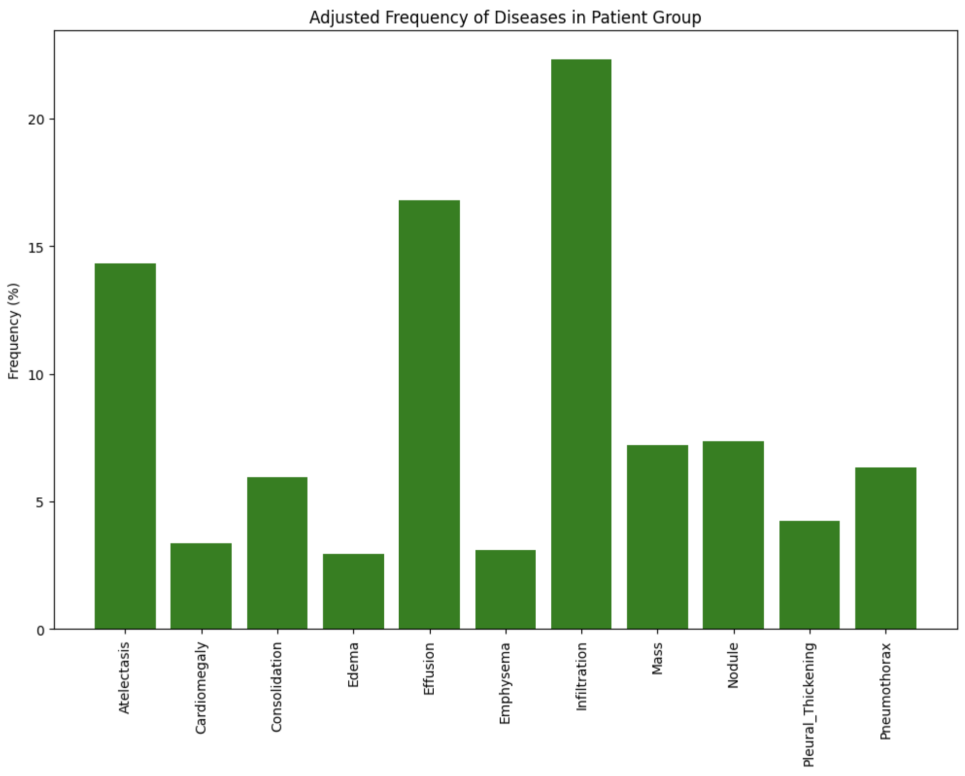


Fig. 5.3.: Resampled data

5.1.3. Data splitting

The following code snippet splits the dataframe into training and validation sets. The test size is set to 25%, meaning 75% will be used for training. The "random_state=2018" ensures the split will be reproducible. Finally, the "stratify" ensures that the split maintains the same proportion of classes in both training and validation sets. It uses the first 4 characters of the "Finding Labels" string for stratification. The purpose of splitting is to create balanced training and validation datasets, which preserve the distribution of findings across the split.

```
from sklearn.model_selection import train_test_split
train_df, valid_df = train_test_split(all_xray_df,
                                   test_size = 0.25,
                                   random_state = 2018,
                                   stratify = all_xray_df['Finding Labels'].map(lambda x:
                                        x[:4]))
print('train', train_df.shape[0], 'validation', valid_df.shape[0])
```

List. 5.7: Data splitting

5.1.4. Data augmentation

First, in order to use Keras' ImageDataGenerator, we need our labels to be in the following format (in a list) (example): [Atelectasis, Consolidation, Infiltration], however we currently have this: Atelectasis|Consolidation|Infiltration. The two lines of code in listing 5.8, will modify both the training and validation datasets accordingly.

```
valid_df['newLabel'] = valid_df.apply(lambda x: x['Finding Labels'].split('|'), axis
    =1)
train_df['newLabel'] = train_df.apply(lambda x: x['Finding Labels'].split('|'), axis
    =1)
```

List. 5.8: Formatting

The ImageDataGenerator in Keras is a tool for real-time image augmentation during model training. Its primary purpose is to enhance the robustness and generalization of deep learning models. It provides variations in the training data at each epoch, which helps prevent overfitting. During training, it dynamically generates augmented versions of input images, introducing variations such as rotation, shifts, flips, brightness changes and so on. If we look at the code now, the "samplewise_center=True" simply centers each sample by subtracting its mean from each pixel. This helps to remove the mean bias from the images, making the training process more stable and efficient. The "samplewise_std_normalization=True" normalizes each sample by dividing each pixel by its standard deviation. The goal of this normalization is to help the model to converge faster and improve performance. The next two lines of the code, simply randomly split horizontally some images but it does not flip them vertically. The height and width shifts randomly move the inputs vertically by 5% or horizontally by 10%. The rotation randomly rotates inputs by up to 5 degrees. The shear range, applies shearing transformations (image deformation, unlike rotations, which preserve the lengths and angles, shearing changes the shape of an object) randomly. Next, the reflect fill mode determines how points outside the boundaries of an image are filled during data augmentation, in this case, the image is reflected at the boundary. Finally, the zoom range randomly zooms inside the pictures by 15%.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
IMG_SIZE = (128, 128)
core_idg = ImageDataGenerator(samplewise_center=True,
                              samplewise_std_normalization=True,
                              horizontal_flip = True,
                              vertical_flip = False,
                              height_shift_range= 0.05,
                              width_shift_range=0.1,
                              rotation_range=5,
                              shear_range = 0.1,
                              fill_mode = 'reflect',
                              zoom_range=0.15)
```

List. 5.9: Data augmentation instance

The following code sets up the data generators on the test, training and validation data. It also loads and preprocesses the images while applying the data augmentation to the training and validation data to enhance the training process.

```
train_gen = core_idg.flow_from_dataframe(dataframe=train_df,
                             directory=None,
                             x_col = 'path',
                            y_col = 'newLabel',
                             class_mode = 'categorical',
                            classes = all_labels,
                            target_size = IMG_SIZE,
                             color_mode = 'grayscale',
                            batch_size = 32)
valid_gen = core_idg.flow_from_dataframe(dataframe=valid_df,
                             directory=None,
                             x_col = 'path',
                            y_col = 'newLabel',
                             class_mode = 'categorical',
                            classes = all_labels,
                            target_size = IMG_SIZE,
                             color_mode = 'grayscale',
                            batch_size = 256) # we can use much larger batches for
                                evaluation
test_X, test_Y = next(core_idg.flow_from_dataframe(dataframe=valid_df,
                             directory=None,
                             x_col = 'path',
                            y_col = 'newLabel',
                             class_mode = 'categorical',
                            classes = all_labels,
                            target_size = IMG_SIZE,
                             color_mode = 'grayscale',
                            batch_size = 1024))
```

List. 5.10: Data Preprocessing

5.2. Model description

In this section, we will discuss the model that was created and that is used in the application.

```python
from keras.applications.mobilenet import MobileNet
from keras.layers import GlobalAveragePooling2D, Dense, Dropout, Flatten
from keras.models import Sequential
from keras import optimizers, callbacks, regularizers
base_mobilenet_model = MobileNet(input_shape = t_x.shape[1:],
                              include_top = False, weights = None)
multi_disease_model = Sequential()
multi_disease_model.add(base_mobilenet_model)
multi_disease_model.add(GlobalAveragePooling2D())
multi_disease_model.add(Dropout(0.5))
multi_disease_model.add(Dense(512))
multi_disease_model.add(Dropout(0.5))
multi_disease_model.add(Dense(len(all_labels), activation = 'sigmoid'))
multi_disease_model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                         metrics = ['binary_accuracy', 'mae'])
multi_disease_model.summary()
```

List. 5.11: Deep Learning Model

MobileNet [32] is an efficient neural network architecture designed for mobile and embedded vision applications. It uses depthwise separable convolutions to reduce the number of parameters and computational cost compared to standard convolutions. This approach enables MobileNet to maintain performance while being lightweight and fast. In this model, the base MobileNet processes input images to generate meaningful feature maps, which can then be used for further tasks such as multi-label classification.

In the code, we first create a Sequential model. The Sequential model in Keras is a linear stack of layers. It allows us to create a model by simply adding layers to it one by one. In our case, it is used to build a neural network by adding the pre-trained MobileNet as its base followed by a few additional layers. After adding the MobileNet (the MobileNet does not include its top layers, which would have been fully connected layers used for classification (because "include_top = False" in our code), we add a Global Average Pooling Layer, this layer replaces the fully connected layers found in classification networks. It reduces each feature map to a single value by taking the average of all values in that feature map, which reduces the number of parameters and helps prevent overfitting. Again, in order to help prevent overfitting, we add a Dropout layer, which is a regularization technique. This layer randomly sets 50% of the input units to 0 at each update during training, which helps the model generalize better. Then, we add a Dense layer, that consists of 512 neurons and it is a fully connected layer that learns complex features from the pooled feature maps generated by MobileNet. After that, we add another Dropout Layer followed by an Output layer, this is the final layer. It has a number of neurons equal to the number of labels ("len(all_labels)"). Each neuron corresponds to a different disease and uses a sigmoid activation function to output a probability between 0 and 1 for each disease.

The entire model is then compiled with the Adam optimizer, using binary cross-entropy as the loss function, which is suitable for multi-label classification problems where each label is independent. The metrics used are binary accuracy and Mean Absolute Error (MAE).

5.2.1. Callbacks for Model Training

The ModelCheckpoint is used to save the model weights to a file only when there is an improvement in the validation loss. The EarlyStopping stops training when the validation loss has stopped improving, here it will stop if the loss has not improved in 5 epochs. The Callback list combines the checkpoint and early stopping callbacks into a list that can be passed to the "fit" function later.

```
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, EarlyStopping,
    ReduceLROnPlateau
weight_path="{}_best_.weights.h5".format('xray_class')
checkpoint = ModelCheckpoint(weight_path, monitor='val_loss', verbose=1,
                            save_best_only=True, mode='min', save_weights_only = True)
early = EarlyStopping(monitor="val_loss",
                      mode="min",
                      patience=5)
callbacks_list = [checkpoint, early]
```

List. 5.12: Callbacks

5.2.2. Model training, prediction and evaluation

First we will do a very short training followed by a much longer one to see how the results improve.

Using the ".fit" method, we can train the model very quickly on a single epoch with 100 steps using the training data generator we created earlier. This small training already shows a training accuracy of 83.89% and a validation accuracy of 86.73%.

Then, we use the ".predict" method to generate predictions on the test dataset. We will analyze the result of those 32 predictions using and ROC curve graph. The AUC is provided in the legend. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. It is used to evaluate the diagnostic ability of a binary classifier. An AUC of 1.0 indicates perfect classification, while an AUC of 0.5 suggest no discriminative power (equivalent to random guessing). Our initial results here are all very close to 0.5, with 0.44 for Fibrosis which is the worst result (the model performed worse than random guessing) and the best result it 0.53 for Edema, which is slightly better than random guessing.

```
multi_disease_model.fit(train_gen,
                        steps_per_epoch=100,
                        validation_data = (test_X, test_Y),
                        epochs = 1,
                        callbacks = callbacks_list)

pred_Y = multi_disease_model.predict(test_X, batch_size = 32, verbose = True)

from sklearn.metrics import roc_curve, auc
fig, c_ax = plt.subplots(1,1, figsize = (9, 9))
for (idx, c_label) in enumerate(all_labels):
    fpr, tpr, thresholds = roc_curve(test_Y[:,idx].astype(int), pred_Y[:,idx])
    c_ax.plot(fpr, tpr, label = '%s (AUC:%0.2f)' % (c_label, auc(fpr, tpr)))
c_ax.legend()
c_ax.set_xlabel('False Positive Rate')
c_ax.set_ylabel('True Positive Rate')
```

```
fig.savefig('barely_trained_net.png')
```

List. 5.13: ".fit" training ".predict" prediction and plots for evaluation

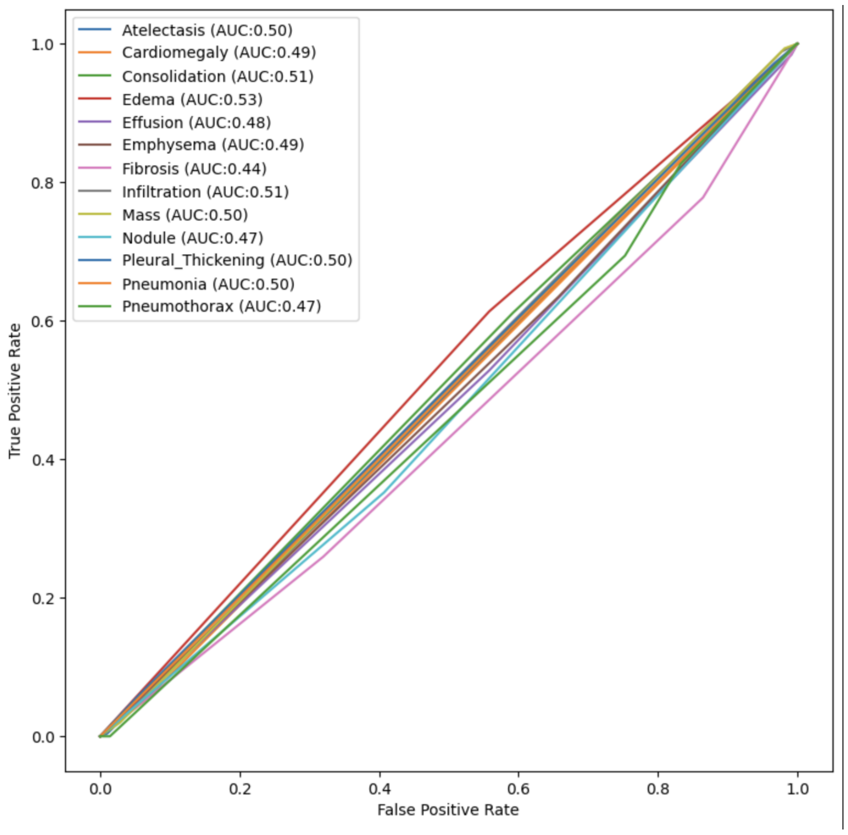


Fig. 5.4.: Initial ROC Curve

The updated ROC curve (see Figure 5.5) illustrates the performance of our model after extended training. The AUC values indicate significant improvements in the model's ability to distinguish between different diseases.

- Cardiomegaly achieves the highest AUC value of 0.90, suggesting that the model correctly classifies this condition 90% of the time.
- Pneumonia has the lowest AUC value of 0.61, which, although not optimal, still provides some predictive power.
- The average AUC across all conditions is approximately 0.75, demonstrating a generally good performance by the model.

While there are substantial variations in the results, the overall trend shows that the model is capable of effectively aiding in the diagnosis of multiple thoracic and lung diseases. Those results are in accord with the prediction made in chapter 2.2.2. "Disease distribution", we stated that the condition "cardiomegaly" has a higher representation in the dataset, while "pneumonia" has very low prevalence in it. Those results indeed show that the model classifies correctly cardiomegaly the best and his worst performance is pneumonia.

```
from sklearn.metrics import roc_curve, auc
fig, c_ax = plt.subplots(1,1, figsize = (9, 9))
```

```
for (idx, c_label) in enumerate(all_labels):
    fpr, tpr, thresholds = roc_curve(test_Y[:,idx].astype(int), pred_Y[:,idx])
    c_ax.plot(fpr, tpr, label = '%s (AUC:%0.2f)' % (c_label, auc(fpr, tpr)))
c_ax.legend()
c_ax.set_xlabel('False Positive Rate')
c_ax.set_ylabel('True Positive Rate')
fig.savefig('trained_net.png')
```

List. 5.14: Evaluation with longer training

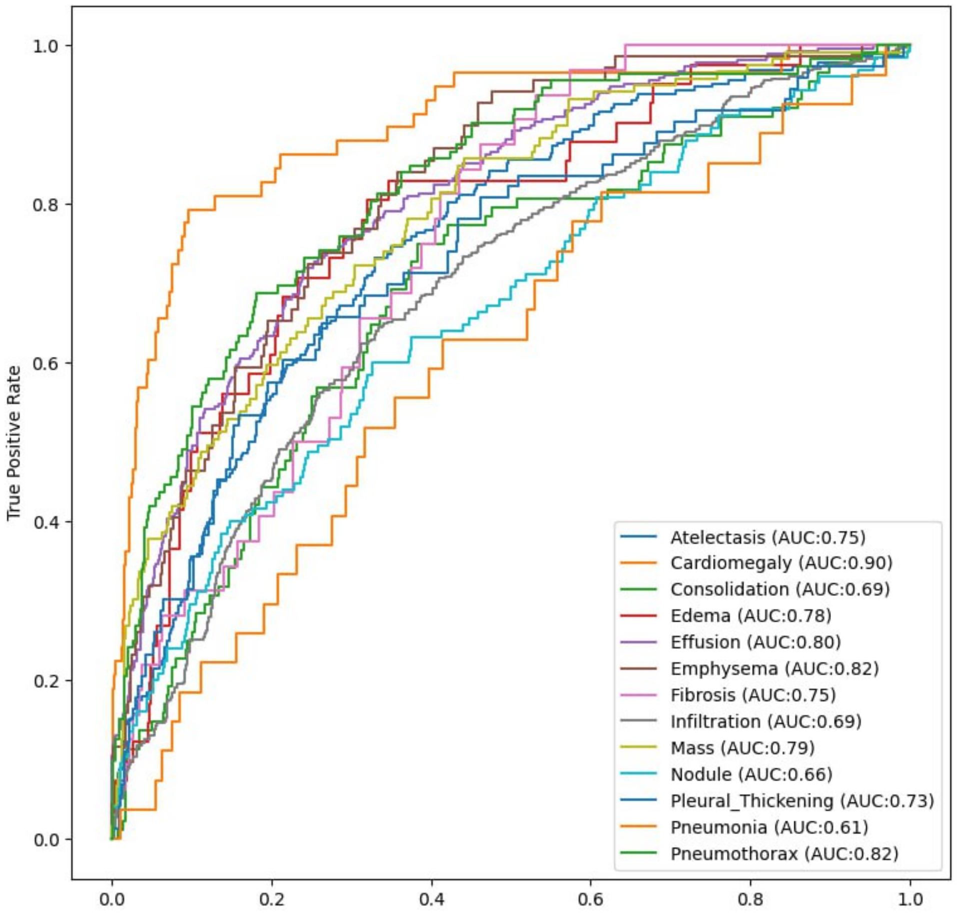


Fig. 5.5.: Final ROC Curve

5.2.3. Optimizer selection

In this subchapter, we explore the impact of various optimization algorithms on the performance of our multi-disease classification model. The selection of an appropriate optimizer can significantly influence the convergence speed and final performance of the neural network. Here, we evaluate five different optimizers: Stochastic Gradient Descent (SGD) [44], SGD with momentum, Adagrad [2], Adadelta [1], and Adam [3].

Optimizers Overview

- SGD:

  - Basic form of gradient descent where updates are performed using individual samples.
  - Can be slow to converge and sensitive to the learning rate.
- SGD with Momentum:
  - Enhances SGD by adding a fraction of the previous update to the current update, helping to accelerate convergence in the right direction.
- Adagrad:
  - Adaptively adjusts the learning rate for each parameter, providing larger updates for infrequent parameters and smaller updates for frequent ones.
  - Particularly useful for sparse data.
- Adadelta:
  - An extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
  - Uses a moving window of gradient updates to improve performance.
- Adam (Adaptive Moment Estimation):
  - Combines the benefits of Adagrad and RMSProp (Root Mean Squared Propagation). It computes adaptive learning rates for each parameter and has been shown to work well in practice for a wide range of problems.

Experimental Setup

We train the multi-disease classification model using each optimizer for up to 50 epochs. Early stopping is employed to halt training if no improvement in validation loss is observed for 5 consecutive epochs. The model performance is evaluated using the validation loss.

```
import keras

sgd = keras.optimizers.SGD(learning_rate=0.1)
sgd_momentum = keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
adagrad = keras.optimizers.Adagrad()
adadelta = keras.optimizers.Adadelta()
adam = keras.optimizers.Adam(learning_rate = 0.0005)
adam2 = keras.optimizers.Adam(learning_rate = 0.005)
adam3 = keras.optimizers.Adam(learning_rate = 0.05)
adam4 = keras.optimizers.Adam()
optimizers_list = [('sgd',sgd ),
                 ('sgd_momentum',sgd_momentum ),
                 ('adagrad',adagrad),
                 ('adadelta',adadelta),
                 ('adam, lr=0.0005', adam),
                   ('adam, lr=0.005', adam2),
                   ('adam, lr=0.05', adam3),
                   ('adam, default', adam4)]

early = EarlyStopping(monitor="val_loss",
                    mode="min",
                    patience=5)
callbacks_list = [early]


```

```
plt.figure(figsize=(20,5))
for optimizer in optimizers_list:
    print(optimizer)
    multi_disease_model.compile(optimizer = optimizer[1], loss = 'binary_crossentropy'
        ,
                              metrics = ['binary_accuracy', 'mae'])
    history = multi_disease_model.fit(train_gen,
                                     validation_data = (test_X, test_Y),
                                     epochs = 50,
                                     callbacks = callbacks_list)
    plt.plot(history.history['val_loss'])
plt.legend([x[0] for x in optimizers_list], loc='upper right')
plt.title('model accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
plt.savefig('optimizer_selection.png', bbox_inches='tight')
```

List. 5.15: Optimizer selection

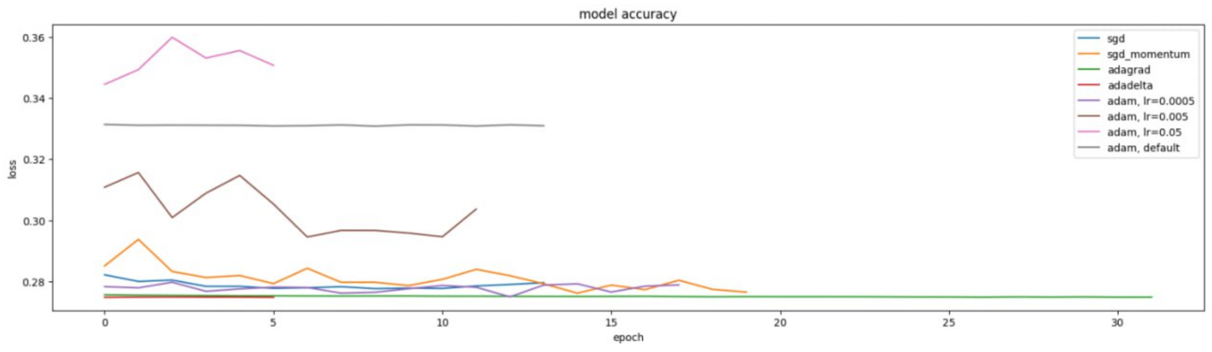


Fig. 5.6.: Optimizer selection graph

As you can see on the graph, the training always stops before reaching 50 epochs and in some cases it stops very early, this is due to the 5 epochs early stopping callback we added to prevent overfitting.

Results and Analysis

The resulting plot compares the validation loss across different optimizers over the training epochs. The key observations include:

- Adam Optimizer:
  - Adam often converges quickly and achieves the lowest validation loss, indicating its effectiveness for this problem.
- SGD with Momentum:
  - Adding momentum to SGD significantly improves convergence speed and stability compared to standard SGD.
- Adagrad and Adadelta:
  - Both optimizers show adaptive learning rate benefits, with Adadelta generally maintaining better stability in later epochs.

- Standard SGD:
  - Standard SGD without momentum shows the slowest convergence and higher validation loss, demonstrating its inefficiency for this problem.

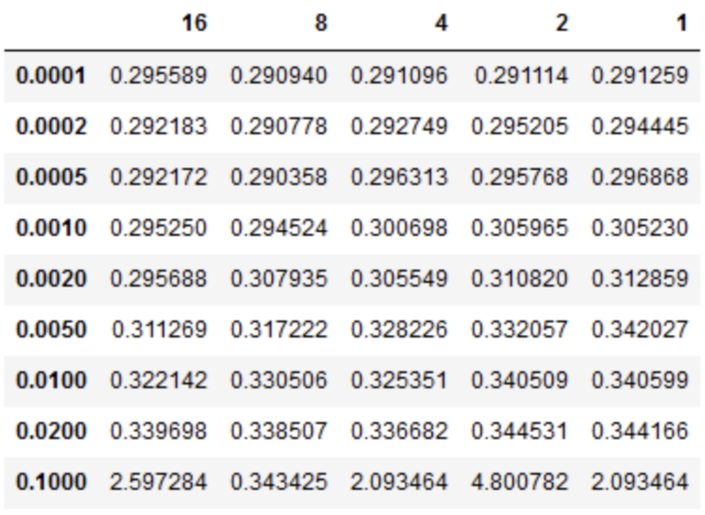
From our experiments, Adam proves to be the most effective optimizer for our multi-disease classification task, achieving the lowest validation loss and faster convergence. These results highlight the importance of optimizer selection in training deep learning models effectively.

5.2.4. Parameter tuning

Based on the findings made by the authors of this **Github page** [18], we can see a few interesting points. These are also improvements that can be made to the model used in our application.

Batch Size and Learning Rate

The table compares batch size accumulation steps (32 × n) with learning rates. Their model achieves better loss when using learning rates around 0.0005, combined with a gradient accumulation step size of 8 or a batch size of 256. Interestingly, similar performance is observed for both smaller and larger batch sizes. Therefore, we can confidently conclude that batch sizes of 1024 or 2048 would not substantially improve performance. Moving forward, we recommend using the ADAM optimizer with a batch size of 256 and gradient accumulation.



	16	8	4	2	1
0.0001	0.295589	0.290940	0.291096	0.291114	0.291259
0.0002	0.292183	0.290778	0.292749	0.295205	0.294445
0.0005	0.292172	0.290358	0.296313	0.295768	0.296868
0.0010	0.295250	0.294524	0.300698	0.305965	0.305230
0.0020	0.295688	0.307935	0.305549	0.310820	0.312859
0.0050	0.311269	0.317222	0.328226	0.332057	0.342027
0.0100	0.322142	0.330506	0.325351	0.340509	0.340599
0.0200	0.339698	0.338507	0.336682	0.344531	0.344166
0.1000	2.597284	0.343425	2.093464	4.800782	2.093464

Fig. 5.7.: Learning Rate Table

Image size and color

As is showed with the following graph, it seems that an image size of 256 yields great results.

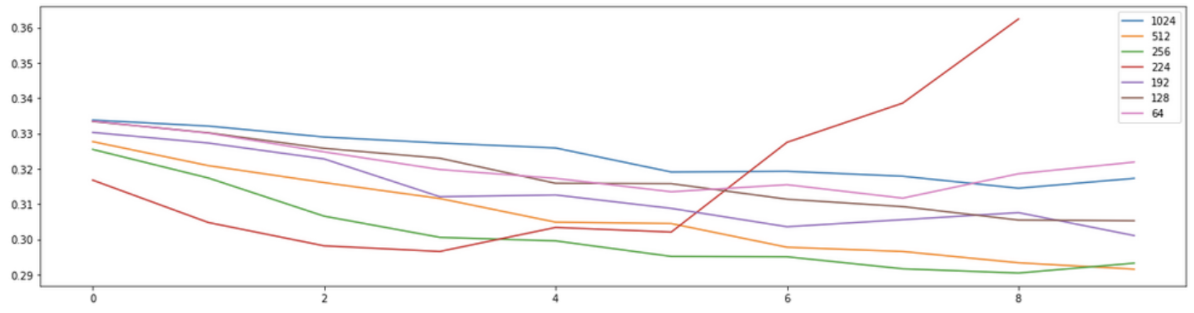


Fig. 5.8.: Image Size Graph

In our application, we opted for grayscale images. Unlike RGB images, which use three color channels, grayscale images use only one channel. This choice allows us to conserve memory and run larger batch sizes during training, thus resulting in shorter training times.

5.3. Model performance

The multi-disease classification model was trained using the Adam optimizer with a learning of 0.0005, which demonstrated superior performance compared to other optimizers such as SGD, SGD with momentum, Adagrad, Adadelta, and Adams with learning rates of 0.005, 0.05 and the default value. The result of Adam with a value of 0.0005 for the learning rate is very close to the results of Adagrad and Adadelta, however, based on the findings of the Github [18] page mentioned earlier, during longer training Adam gets better, which is why I chose to train my model using the Adam optimizer. After 14 epochs of training, the model achieved a training loss of 0.2606, with a binary accuracy of 89.65% and a MAE of 0.1549. On the validation set, the model reported a validation loss of 0.2788, a binary accuracy of 89.11%, and an MAE of 0.1586. These metrics indicate that the model has learned to generalize well to unseen data, maintaining a balance between fitting the training data and preserving accuracy on the validation set. The high binary accuracy and relatively low MAE across both training and validation datasets underscore the model's robustness and effectiveness in diagnosing multiple diseases from X-ray images. This performance aligns with the expectations for a well-trained Convolutional Neural Network (CNN), particularly when using a powerful optimizer like Adam, which adaptively adjusts the learning rate and handles sparse gradients effectively.

6 Application Implementation

6.1. Backend

The backend of the ChestVision [7] application forms the core of its functionality, managing data, processing requests, and ensuring secure and efficient interactions between the user and the server. This section delves into the implementation of the REpresentational State Transfer (REST) API, detailing the endpoints and their roles in handling various operations such as data retrieval and manipulation. It also covers the authentication mechanisms put in place to safeguard patient information, along with a comprehensive overview of the database structure and the scripts used for populating it.

6.1.1. API

The RESTful API in the ChestVision application serves as the communication bridge between the frontend and backend. It comprises various endpoints designed to handle different operations, such as retrieving patient data, submitting new diagnostic results, and updating patient records. Each endpoint is crafted to respond to specific HTTP requests (GET, POST, PUT, DELETE) and is integrated with the frontend to provide real-time data and functionalities. This section details the purpose and implementation of each endpoint, including code examples and explanations of how error handling and data validation are managed to ensure robust and reliable interactions.

- GET / : simply renders the "home.html" page if the user is logged in.

- DELETE /delete-patient : Deletes the selected patient by ID, if the patient exists.
- POST /predict : takes a single image as input, returns the diagnostic, fetches the patient information if adding a follow-up to an existing patient, or creates a new patient if "Follow-up#" is set to 0.
- POST /predictions : takes one or multiple images as input, simply returns the findings without interacting with the database.
- GET /patients : returns a list of 25 patients per page on the "Patients" pages.
- GET /patients/<int:patient_id> : renders the "patient_detail.html" page.
- GET /get_patient/<int:patient_id> : returns the specific patient's information.

Some code snippets

The "/" route is really simple, if the request is GET it renders the home.html page.

```python
@views.route('/', methods=['GET', 'POST'])
@login_required
def home():
    if request.method == 'POST':
        pass
    return render_template("home.html", user=current_user)
```

List. 6.1: "/" route

The "/predict" route is the most complicated route. It first loads the model "modelx2.h5", then gets the image uploaded by the user in the HTML page. After that, it processes the image, reducing its size, converting it to grayscale and converts it to a NumPy array. Then, it feeds it to the model and generates a prediction for the image and displays the prediction if the probability found by the model of a specific disease is higher than 30%. After which, it saves the image in the images folder and checks if the given "Follow-up#" given on the page by the user is higher than 0. If it is not, it will create a new patient, if it is, it will add a new follow-up to an existing patient. All of this is then saved in the database and finally, the page predict.html is rendered with the given image and its diagnostic.

```python
@views.route('/predict', methods=['GET', 'POST'])
@login_required
def predict():
    if request.method == 'POST':
        model = tf.keras.models.load_model('./models/modelx2.h5')
        uploaded_file = request.files['image']
        finding_label = ''

        if uploaded_file and uploaded_file.filename != '':
            image = PilImage.open(io.BytesIO(uploaded_file.read()))
            gray = image.convert('L')
            img = gray.resize(IMG_SIZE, PIL.Image.Resampling.LANCZOS)
            numpydata = np.array(img)
            input_image = preprocess_image(numpydata)

            prediction = model.predict(input_image)
            disease_probabilities = prediction[0]

            result_text = 'Diseases detected:'
            for disease, probability in zip(all_labels, disease_probabilities):
                if probability >= 0.3: # Only include diseases with probability >= 30%
                    finding_label += disease + ', '
                    result_text += f'{disease}: {probability*100:.2f}% '

            finding_label = finding_label[:-2] # Remove the last comma and space

            image_path = f"./website/static/assets/images/{uploaded_file.filename}" #
                 Use the uploaded file's filename

            if os.path.exists(image_path):
                print("Image already exists")
            else:
                print("Adding image to folder")
                image.save(image_path)
                print("Image saved")


            # Get details from form
            gender = request.form.get('gender')
            follow_up_number = int(request.form.get('follow_up_number'))
            group = request.form.get('group')
            age = request.form.get('age')
            view_position = request.form.get('view_position')
            original_image_width_height = request.form.get('original_image_width_height
                 ')
            original_image_pixel_spacing = request.form.get('
                 original_image_pixel_spacing')

            # Check if follow_up_number is greater than 0
            if follow_up_number > 0:
                # If it is, get the existing patient
                patient_id = request.form.get('patient_id')
                patient = Patient.query.filter_by(gender=gender).first()
                if patient:
                    patient = Patient.query.get(patient_id)
                    appointment = Appointment(patient_id=patient.id,
                                              follow_up_number=follow_up_number,
                                              group=group)
                else:
                    # If patient does not exist, create a new patient
                    patient = Patient(gender=gender)
                    db.session.add(patient)
                    db.session.commit()
            else:
                # If follow_up_number is not greater than 0, create a new patient
                patient = Patient(gender=gender)
                db.session.add(patient)
                db.session.commit()

            # Create new appointment
            appointment = Appointment(patient_id=patient.id,
                                      follow_up_number=follow_up_number,
                                      group=group)
            db.session.add(appointment)
            db.session.commit()

            # Create new image
            image = Image(appointment_id=appointment.id,
                          image_index=uploaded_file.filename,
                          patient_age=age,
                          view_position=view_position,
                          original_image_width_height=original_image_width_height,
                          original_image_pixel_spacing=original_image_pixel_spacing)
            db.session.add(image)
            db.session.commit()

            # Create new finding
            finding = Finding(image_id=image.id,
                              finding_label=finding_label)
            db.session.add(finding)
            db.session.commit()

            myfile = uploaded_file.filename

            return render_template('predict.html', prediction_texts=[result_text], user
                =current_user, myfile=myfile)
          else:
            return render_template('predict.html', prediction_texts=['No files selected
                '], user=current_user)
        return render_template('predict.html', user=current_user)
```

List. 6.2: "/predict" route

Finally, the "/get_patient/<int:patient_id>" route is used to query the database and it uses the serialize method defined directly in the Patient's table to find all of the patient's information. It then returns it all as a JSON object.

```
@views.route('/get_patient/<int:patient_id>', methods=['GET'])
def get_patient(patient_id):
    patient = Patient.query.get(patient_id)
    if patient:
        return jsonify(patient.serialize())
    else:
        return jsonify({})
```

List. 6.3: "/get_patient/<int:patient_id>" route

6.1.2. Authentication

In a real-world scenario, patient authentication is crucial for applications such as ChestVision. Ensuring the privacy of patients' data is a critical part of these applications. In our case, anyone can sign up and log in, but if this application were to be deployed in a hospital, then only doctors should be able to connect to the application, and each patient's data should only be accessible by their own doctors.

Routes

- POST /login : logs the user in.
- POST /sign-up : signs the user up.
- GET /logout : logs the user out.

For both the login and logout routes, we utilize the "Flask-Login" library to manage user sessions.

- Login: The "login_user" function is used to log in a user and start a session. The "remember=True" parameter allows users to stay logged in across browser sessions.
- Logout: The "logout_user" function logs the user out, terminating the session.

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        user = User.query.filter_by(email=email).first()
        if user:
            if check_password_hash(user.password, password):
                flash('Logged in successfully!', category='success')
                login_user(user, remember=True)
                return redirect(url_for('views.home'))
            else:
                flash('Incorrect password, try again.', category='error')
        else:
            flash('Email does not exist.', category='error')

    return render_template("login.html", user=current_user)
```

List. 6.4: "/login" route

```
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('auth.login'))
```

List. 6.5: "/logout" route

Our application employs several security measures to protect user data:

- Password Hashing: Passwords are hashed using the pbkdf2:sha256 algorithm before being stored in the database. This ensures that even if the database is compromised, the actual passwords remain secure.
- Secure Transmission: In a deployed environment, HTTPS should be used to encrypt data transmitted between the client and server, preventing eavesdropping and man-in-the-middle attacks.

It also uses form validation and perfoms serveral checks during user registration.

- Email Validation: Ensures that the email is at least 4 characters long.
- Name Validation: Ensures that the first and last names are at least 2 characters long.
- Password Validation: Ensures that passwords are at least 7 characters long and that the password confirmation matches.

```
@auth.route('/sign-up', methods=['GET', 'POST'])
def sign_up():
    if request.method == 'POST':
        email = request.form.get('email')
        first_name = request.form.get('firstName')
        last_name = request.form.get('lastName')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')

        user = User.query.filter_by(email=email).first()
        if user:
            flash('Email already exists.', category='error')
        elif len(email) < 4:
            flash('Email must be greater than 3 characters.', category='error')
        elif len(first_name) < 2:
            flash('First name must be greater than 1 character.', category='error')
        elif len(last_name) < 2:
            flash('Last name must be greater than 1 character.', category='error')
        elif password1 != password2:
            flash('Passwords don\'t match.', category='error')
        elif len(password1) < 7:
            flash('Password must be at least 7 characters.', category='error')
        else:
            new_user = User(email=email,
                            first_name=first_name,
                            last_name=last_name,
                            password=generate_password_hash(password1,
                                                            method='pbkdf2:sha256'))
            db.session.add(new_user)
            db.session.commit()
            login_user(new_user, remember=True)
            flash('Account created!', category='success')
            return redirect(url_for('views.home'))

    return render_template("sign_up.html", user=current_user)
```

List. 6.6: "/sign-up" route

6.1.3. Database

Description

The database for ChestVision is designed to store and manage essential data related to patients, their appointments, diagnostic images, and findings. The database schema includes the following main tables:

- Users: Stores information about the users (doctors), including their email, password, first name, and last name.
- Patients: Contains patient details such as gender and a list of appointments.
- Appointments: Records appointments for patients, including the follow-up number and group information.
- Images: Stores diagnostic images taken during appointments, along with metadata such as the image index, patient age, and view position.
- Findings: Contains diagnostic findings associated with each image.

The relationships between these tables ensure that data integrity is maintained and allow for efficient data retrieval. For example, each patient can have multiple appointments, and each appointment can include multiple findings but only one image per appointment.

Database population script

To facilitate development and testing, a population script is provided to seed the database with sample data from the JSON file "patient_data.json" found in the documentation folder of the application. This script reads patient data, including appointments, images, and findings, and populates the database accordingly. The provided script in listing 6.7 performs the following tasks:

- Initialization: Initializes the application context and reads patient data from a JSON file (patient_data.json).
- Adding Patients: Iterates through each patient in the JSON data and creates a new Patient record in the database.
- Adding Appointments: For each patient, iterates through their appointments, extracting follow-up information and creating new Appointment records. Each appointment is associated with the patient and includes a group attribute.
- Adding Images: For each appointment, extracts image information and creates new Image records. These records include metadata such as image index, patient age, view position, original image dimensions, and pixel spacing.
- Adding Findings: For each image, extracts diagnostic findings and creates new Finding records, associating them with the respective image.
- Committing Data: Commits the changes to the database for each patient, ensuring data integrity and proper association between records.

```python
from website import db, create_app
from website.models import Patient, Appointment, Image, Finding
import json

app = create_app()
x = 0
with app.app_context():
    with open('./Documentation/patient_data.json') as f:
        data = json.load(f)
    # Assuming data is your JSON data
    for patient in data["Patients"]:
        new_patient = Patient(id=patient["PatientID"],
                              gender=patient["PatientGender"])
        db.session.add(new_patient)
        db.session.commit()

        #print(patient["PatientID"])
        #print(patient["PatientGender"])

        for appointment in patient["Appointments"]:
            for follow_up, image in appointment.items():
                # Include the 'group' attribute when creating a new Appointment
                new_appointment = Appointment(patient_id=new_patient.id,
                                              follow_up_number=int(follow_up.split('#')
                                                  [-1]),
                                              group=image[0]["Group"]) # Assuming all images in an appointment have the same group
                db.session.add(new_appointment)
                db.session.commit()

                #print("Appointment id")
                #print(new_appointment.id)
                #print(follow_up)
                #print(image[0]["Group"])

                new_image = Image(appointment_id=new_appointment.id,
                                  image_index=image[0]["ImageIndex"],
                                  patient_age=image[0]["PatientAge"],
                                  view_position=image[0]["ViewPosition"],
                                  original_image_width_height=image[0]["OriginalImageWidthHeight"],
                                  original_image_pixel_spacing=image[0]["OriginalImagePixelSpacing[x-y]"])
                db.session.add(new_image)
                db.session.commit()

                #print("Image id")
                #print(new_image.id)
                #print(image[0]["ImageIndex"])
                #print(image[0]["PatientAge"])
                #print(image[0]["ViewPosition"])
                #print(image[0]["OriginalImageWidthHeight"])
                #print(image[0]["OriginalImagePixelSpacing[x-y]"])

                for finding_label in image[0]["FindingLabels"]:
                    new_finding = Finding(image_id=new_image.id,
                                          finding_label=finding_label)
                    db.session.add(new_finding)
                    db.session.commit()

                    #print(finding_label)

        # Commit the changes for each patient
        print(f"Patient {x} has been added to the database.")
        x += 1
        db.session.commit()

print('Done!')
```

List. 6.7: Database population script

This script automates the process of populating the database with structured patient data, facilitating the development and testing of the ChestVision application.

6.2. Frontend

The frontend of the ChestVision application is designed to provide a seamless and user-friendly experience for medical practitioners. This section explores the development of the user interface, focusing on the integration of Jinja templates and Bootstrap to create a responsive and interactive platform. It describes the various views available to users, such as patient management and diagnostic result pages, and explains how these components interact with the backend to display real-time data and predictions.

6.2.1. Integration with Jinja and Bootstrap

Jinja Templates: Jinja, a templating engine for Python, is used to dynamically generate HTML pages by injecting data from the backend. This allows for the creation of reusable templates and components, making the development process more efficient. For example, the base template might include common elements like the navigation bar and footer, which can be extended by other templates.

Bootstrap: Bootstrap is a popular front-end framework that ensures the application is responsive and visually appealing. It provides a range of pre-designed components, such as forms, buttons, and navigation bars, which can be easily customized to fit the application's design requirements. Bootstrap's grid system is used to create a responsive layout that adapts to different screen sizes, ensuring that the application is accessible on both desktop and mobile devices.

6.2.2. HTML Pages

The frontend views include:

- Login and Sign-Up Pages: These forms allow users to authenticate themselves and access the system.
- Home Page: This page allows the user to input multiple images to the model and get their associated disease predictions, without any access to the database.
- Predictions Page: This page allow the user to add a follow-up or create a new patient. He can do so by inputting a single image and get a prediction of its associated disease, all of the patient's and image's information will be saved to the database along with the prediction and the image will be saved in the images folder.
- Patients Page: The patient's page consists of a table that displays each patient's information with the possibility to click on "details" to access the information of each of the follow-ups for each patient. This page only displays 25 patients per page.
- Patient's details Page: This is the details page, it contains all of the follow-ups of each patients, it displays their x-ray images and their diagnostic along with all of the image's information.

base.html

This page is used as the base template for jinja. It imports bootstrap and creates the navbar. The navbar uses the "navbar navbar-expand-lg navbar-dark bg-dark" class from bootstrap and it creates clickable buttons that redirect to each of the views mentioned above.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
     <button
```

```
      class="navbar-toggler"
      type="button"
      data-toggle="collapse"
      data-target="#navbar"
    >
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbar">
      <div class="navbar-nav">
        {% if user.is_authenticated %}
        <a class="nav-item nav-link" id="home" href="/">Home</a>
        <a class="nav-item nav-link" id="predict" href="/predict">Predictions</a>
        <a class="nav-item nav-link" id="patients" href="/patients">Patients</a>
        <a class="nav-item nav-link" id="logout" href="/logout">Logout</a>
        {% else %}
        <a class="nav-item nav-link" id="login" href="/login">Login</a>
        <a class="nav-item nav-link" id="signUp" href="/sign-up">Sign Up</a>
        {% endif %}
      </div>
    </div>
  </nav>
```

List. 6.8: navbar

As mentioned earlier, this page is the base template of every other HTML page, meaning that the navbar will be displayed on every page, even though it is only defined in this page. To make this happen, we include this container div that includes a block of content. Then, on every other page, we need to include this Jinja code at the beginning: "% extends "base.html" %"

```
<div class="container">{% block content %} {% endblock %}</div>
```

List. 6.9: Jinja content block

Login.html and sign_up.html

Both the login and sign-up pages are simple forms, here (listing 6.10) is the code of login page.

```
{% extends "base.html" %}
{% block title %}Login{% endblock %}
{% block content%}
<form method="POST">
  <h3 align="center">Login</h3>
  <div class="form-group">
    <label for="email">Email Address</label>
    <input
      type="email"
      class="form-control"
      id="email"
      name="email"
      placeholder="Enter email"
    />
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input
      type="password"
      class="form-control"
      id="password"
      name="password"
      placeholder="Enter password"
    />
  </div>
  <br />
  <button type="submit" class="btn btn-primary">Login</button>
</form>
{% endblock %}
```

List. 6.10: Login page

home.html

The homepage, consists of a form that takes images as inputs, these are then given to the model in the backend which, in turn, returns the findings found in each image. The predictions are then displayed along with the image.

```
{% extends "base.html" %}
{% block title %}Home{% endblock %}
{% block content%}
<div class="container mt-5">
    <h1 align="center">Welcome to my disease prediction app</h1>
</div>
<div class="container mt-5" align="center" padding-bottom="50px">
  <h2 align="center">Predictions</h2>
<form action="/predictions" method="POST" enctype="multipart/form-data">
  <div class="form-group">
    <label for="image">Upload Image</label>
    <input type="file" id="image" name="image" accept="image/*" multiple required>
    <button type="submit" class="button-84">Predict</button>
  </div>
</form>
  {% if prediction_texts %}
  <div class="result">
      <h2>Prediction Results:</h2>
      <ul>
          {% for prediction_text in prediction_texts %}
          <li>{{ prediction_text }}</li>
          {% endfor %}
      </ul>
  </div>
</div>
{% endif %}
{% for i in range(1, num_images+1) %}
<img src="{{ url_for('static', filename='assets/uploaded_images/predictions' + i|
    string + '.png') }}" alt="Predictions" class="img-fluid">
{% endfor %}
</div>
```

List. 6.11: home page

predict.html

The predictions page contains a big HTML form that needs to be filled with all of the information necessary to fill the 3 database tables associated to a follow-up. Every field is required. It then displays the result along with the image. Here, you can only input a single image. It also displays a placeholder image whenever no image has been inputted.

```html
<div class="row" padding-top="50px">
    <div class="col-md-6">
        <form method="POST" action="/predict" enctype="multipart/form-data">
            <div class="form-group">
                <label for="follow_up_number">Follow-up #</label>
                <input type="number" id="follow_up_number" name="follow_up_number"
                    required onchange="checkFollowUpNumber()">
            </div>
            <div class="form-group" id="patient_id_group" style="display: none;">
                <label for="patient_id">Patient ID</label>
                <input type="number" id="patient_id" name="patient_id">
            </div>
            <div class="form-group">
                <label for="gender">Gender</label>
                <input type="text" id="gender" name="gender" required>
            </div>
            <div class="form-group">
                <label for="group">Group</label>
                <input type="text" id="group" name="group" required>
            </div>
            <div class="form-group">
                <label for="age">Age</label>
                <input type="number" id="age" name="age" required>
            </div>
            <div class="form-group">
                <label for="view_position">View Position</label>
                <input type="text" id="view_position" name="view_position" required>
            </div>
            <div class="form-group">
                <label for="original_image_width_height">Original Image Width Height</
                    label>
                <input type="text" id="original_image_width_height" name="
                    original_image_width_height" required>
            </div>
            <div class="form-group">
                <label for="original_image_pixel_spacing">Original Image Pixel Spacing</
                    label>
                <input type="text" id="original_image_pixel_spacing" name="
                    original_image_pixel_spacing" required>
            </div>
            <div class="form-group">
                <label for="image">Upload Image</label>
                <input type="file" id="image" name="image" accept="image/*" multiple
                    required>
            </div>
            <button type="submit" class="btn btn-primary">Predict</button>
        </form>
    </div>
    <div class="col-md-6">
        {% if myfile %}
```

```
            <img src="{{ url_for('static', filename='assets/images/' + myfile) }}" alt=
                "Image" class="img-fluid">
        {% else %}
            <img src="{{ url_for('static', filename='assets/placeholder-256x256.gif')
                }}" alt="Placeholder Image" class="img-fluid">
        {% endif %}
    </div>
</div>
<div class="container mt-5">
    {% if prediction_texts %}
    <div class="result">
        <h2>Prediction Results:</h2>
        <ul>
            {% for prediction_text in prediction_texts %}
            <li class="{% if 'No Finding' in prediction_text %}no-finding{% else %}{{
                prediction_text.lower().replace(' ', '-') }}{% endif %}">{{
                prediction_text }}</li>
            {% endfor %}
        </ul>
    </div>
    {% endif %}
</div>
```

List. 6.12: Database form

This page also contains a small script with the "checkFollowUpNumber()" function that checks if the Follow-Up number is bigger than 0, if it is, it adds a new field in the form for the Patient ID number, this ID number is then used to add a follow-up to an existing patient.

```
<script>
    function checkFollowUpNumber() {
        var followUpNumber = document.getElementById('follow_up_number').value;
        if (followUpNumber > 0) {
            document.getElementById('patient_id_group').style.display = 'block';
        } else {
            document.getElementById('patient_id_group').style.display = 'none';
        }
    }
</script>
```

List. 6.13: FollowUpNumber script

patients.html

This page contains a big table that displays 25 patients per page and it contains a page navigation system to find the next 25 patients. It also contains a small script with the "deletePatient" function that deletes a patient using its ID.

```
<table class="table">
    <tr>
        <th scope="col">Patient ID</th>
        <th scope="col">Gender</th>
        <th scope="col" class="age">Age</th>
        <th scope="col" text-align="center">Number of Appointments</th>
        <th scope="col">Diagnosis</th>
        <th scope="col">Delete</th>
        <th scope="col">Details</th>
    </tr>
    {% for patient in patients.items %}
        <tr>
            <td>{{ patient.id }}</td>
            <td>{{ patient.gender }}</td>
            <td>
                {% set ns = namespace(min_age=100, max_age=0) %}
                {% for appointment in patient.appointments %}
                    {% for image in appointment.images if image.patient_age %}
                        {% if image.patient_age < ns.min_age %}
                            {% set ns.min_age = image.patient_age %}
                        {% endif %}
                        {% if image.patient_age > ns.max_age %}
                            {% set ns.max_age = image.patient_age %}
                        {% endif %}
                    {% endfor %}
                {% endfor %}
                {% if ns.min_age == ns.max_age %}
                    {{ ns.min_age }}
                {% else %}
                    {{ ns.min_age }} - {{ ns.max_age }}
                {% endif %}
            </td>
            <td class="appointmentsNum">{{ patient.appointments.count() }}</td>
            <td>
                {% set ns = namespace(diagnosis_set=[]) %}
                {% for appointment in patient.appointments %}
                    {% for image in appointment.images %}
                        {% for finding in image.findings if finding.finding_label %}
                            {% set label = finding.finding_label %}
                            {% if label == "Pleural_Thickening" %}
                                {% set label = "Pleural Thickening" %}
                            {% endif %}
                            {% if label not in ns.diagnosis_set %}
                                {% set ns.diagnosis_set = ns.diagnosis_set + [label] %}
                            {% endif %}
                        {% endfor %}
                    {% endfor %}
                {% endfor %}
                {% if ns.diagnosis_set|length > 1 %}
                    {{ ns.diagnosis_set[:-1]|join(', ') }} & {{ ns.diagnosis_set[-1] }}
                {% else %}
                    {{ ns.diagnosis_set[0] }}
                {% endif %}
            </td>
            <td>
                <button class="btn btn-secondary" type="button" style="display: block;
                    width: 100%; height: 100%;" onClick="deletePatient({{ patient.id }})
                    ">
                    Delete
                </button>
            </td>
            <td><a href="{{ url_for('views.patient_detail', patient_id=patient.id) }}">
                <button class="btn btn-secondary" type="button">
                    Patient ID: {{ patient.id }}
                </button></a></td>
        </tr>
    {% endfor %}
</table>
```

List. 6.14: Patients' table

Listing 6.15 shows the code of the page navigation system.

```
<nav aria-label="Page navigation example">
    <ul class="pagination justify-content-end">
      {% if patients.has_prev %}
          <li class="page-item">
              <a class="page-link" href="{{ url_for('views.patients', page=patients.
                  prev_num) }}">Previous</a>
          </li>
      {% else %}
          <li class="page-item disabled">
              <a class="page-link" href="#" tabindex="-1">Previous</a>
          </li>
      {% endif %}
      <li class="page-item"><a class="page-link" href="#">{{ patients.page }}</a></li>
      {% if patients.has_next %}
          <li class="page-item">
              <a class="page-link" href="{{ url_for('views.patients', page=patients.
                  next_num) }}">Next</a>
          </li>
      {% else %}
          <li class="page-item disabled">
              <a class="page-link" href="#">Next</a>
          </li>
      {% endif %}
    </ul>
</nav>
```

List. 6.15: Navigation system

Finally, the deletePatient function:

```
<script>
    function deletePatient(patientId) {
        fetch("/delete-patient", {
          method: "POST",
          body: JSON.stringify({ patientId: patientId }),
        }).then((_res) => {
          window.location.href = "/patients";
        });
      }
</script>
```

List. 6.16: deletePatient function

patients_detail.html

This page first displays the patient id along with the patient's gender, it also implements a delete patient button. Then, it displays the rest of the information using a for loop that display the information found in the appointments, images and findings tables of the database and also displays the image of each appointment. Finally, it also contains the same "deletePatient" function that was showed above.

```
{% for appointment in patient.appointments %}
    <div class="card mt-3">
        <div class="card-header">
            Follow-up #{{ appointment.follow_up_number }}
        </div>
        <div class="card-body">
            <div class="row">
                <div class="col-md-6">
                    <h5 class="card-title">Appointment ID: {{ appointment.id }}</h5>
                    <p class="card-text">Group: {{ appointment.group }}</p>
                    {% set myns = namespace(diag=[]) %}
                    {% for image in appointment.images %}
                        <h6>Image ID: {{ image.id }}</h6>
                        <p>Age: {{ image.patient_age }}</p>
                        <p>View Position: {{ image.view_position }}</p>
                        <p>Original Image Width Height: {{ image.
                            original_image_width_height }}</p>
                        <p>Original Image Pixel Spacing: {{ image.
                            original_image_pixel_spacing }}</p>
                        {% for finding in image.findings if finding.finding_label %}
                            {% set myns.diag = myns.diag + [finding.finding_label] %}
                        {% endfor %}
                        {% if myns.diag|length > 1 %}
                            {{ myns.diag[:-1]|join(', ') }} & {{ myns.diag[-1] }}
                        {% else %}
                            {{ myns.diag[0] }}
                        {% endif %}
                    {% endfor %}
                </div>
                <div class="col-md-6">
                    {% for image in appointment.images %}
                        <img src="{{ url_for('static', filename='assets/images/' + image
                            .image_index) }}" alt="Image for appointment {{ appointment.
                            id }}">
                    {% endfor %}
                </div>
            </div>
        </div>
    </div>
{% endfor %}
```

List. 6.17: Information loops

7 Conclusion

7.1. Review

When I started developing ChestVision [7], the first thing I did was research articles, videos, and examples of models used to predict diseases from medical images. I found plenty of examples of working models, but very few were integrated into actual applications. Personally, I had some experience with frontend development but not much with backend development. I also lacked experience in training models and integrating them, making every part of this work a great learning experience for me.

Initially, I found a complete working example of a model that could only detect pneumonia in thoracic x-ray images. This allowed me to become familiar with the code and explore this field while experimenting. After that, I managed to save that model and integrate it into a simple HTML page, which served as the proof of concept I needed to propose the type of application I wanted to develop to my teacher.

Later on, I attempted for a long time to train a deep learning model capable of detecting all 13 diseases in the dataset. However, as difficulties arose, I took a break from that and shifted my focus to developing the application itself. Throughout development, I used my simple pneumonia model to test it.

Discovering backend development in Python was particularly fascinating as it opened up a new realm of possibilities and challenges. Initially, I began by creating a basic HTML page to test my backend. It was during this phase that I stumbled upon Flask and its associated components like Flask-Login, which proved to be incredibly useful and straightforward to implement. These modules, especially Flask-Login with its integration of Werkzeug security, facilitated the development of critical application components. For instance, setting up an authentication system with session management became remarkably easier.

Following that, I dove into working with databases. Here, I found SQLAlchemy to be a standout module, significantly simplifying the database integration process. However, I first started with adapting the dataset, initially provided in CSV format. To enhance usability, I transformed this dataset into JSON format using a custom script. This conversion proved advantageous later on during frontend-backend integration, as JSON is widely utilized in JavaScript, particularly for API development. The JSON format streamlined the creation of database tables and serialization methods, facilitating seamless data transfer.

Subsequently, I proceeded to develop the API. Starting with authentication routes in my auth.py file, I built upon existing examples to acclimate myself to creating and testing routes. Once these foundational routes were established, I implemented a basic route to test my initial pneumonia detection model. With these components validated, I expanded the frontend functionality, including patient tables and detailed pages for each of the 30,805 patients in the dataset. This expansion necessitated creating corresponding routes to handle new pages and functionalities. Concurrently, I refined the application's layout, particularly improving image display on the homepage and enhancing prediction label visibility.

Satisfied with the frontend's interaction with the backend, I returned to training the model. Starting afresh, I began by analyzing the CSV dataset through graphs to understand it better. Addressing the challenge posed by the dataset's size, I researched techniques for handling large image datasets, given that attempting to train the model with the entire set of 112,120 images at full resolution caused significant constraints, for example I ran into the following issue: I was missing about 40GB of Random Access Memory (RAM). I implemented batches of images that I could feed one by one to the model during training, but this still took a really long time and yielded suboptimal results, which made me explore other options.

My breakthrough came when I discovered the Sequential model and the pretrained MobileNet, coupled with keras' image preprocessing capabilities. Subsequent dataset resampling significantly improved training efficiency, albeit it remained a time-intensive process. Nevertheless, these efforts paid off with models achieving approximately 80% accuracy. Further optimizations using different learning rates and optimizers pushed the model's accuracy to around 89%, marking a substantial improvement.

With a robust working model in hand, I integrated it into the application by creating a new "Predictions" view. This view interacted with the database to provide predictions based on the model's outputs. Recognizing the limitation of having only one image per appointment in the dataset, I enhanced the homepage to allow users to input multiple images for immediate predictions without database interaction, facilitating easy testing of the model.

7.2. Results Review

Overall, I am pleased with the outcomes of my model and the overall outlook of the application. This project has been a significant learning experience for me. As mentioned in Chapter 5, the model achieved an accuracy of 89.11%, which I consider a noteworthy result. I believe the application is also user-friendly and effective.

7.2.1. Future Improvements

Looking ahead, there are several areas where I aim to enhance the application:

- Improved Security Features: Enhance security measures such as validating email addresses and implementing stronger password checks. This would bolster the overall security posture of the application.
- Deployment: Currently, the application is only deployed locally. I plan to deploy it on a production server to make it accessible beyond my local environment. This will involve configuring the deployment environment and ensuring smooth operation in a real-world setting.
- Expansion of Disease Detection: Expand the scope of disease detection by incorporating additional diseases beyond pneumonia. For instance, integrating models trained on Magnetic Resonance Imaging (MRI) images for detecting diseases in other organs like the brain could broaden the application's utility in medical diagnostics.

These future developments aim to further refine the application's functionality, security, and usability, thereby improving its effectiveness in medical imaging diagnostics.

7.3. Future research

There are many areas of research that are really interesting. Here are a few that could be used to make this project better.

- Multi-Modal Integration: Explore the integration of multiple imaging modalities (e.g., combining X-ray with MRI or Computed Tomography (CT) scans) for comprehensive disease detection. Research on how different modalities can complement each other to improve diagnostic accuracy.
- Enhanced Data Preprocessing Techniques: Develop and refine data preprocessing pipelines to handle diverse medical imaging datasets more effectively. Explore techniques such as data augmentation, noise reduction, and normalization specific to medical images.
- Clinical Validation and User Studies: Conduct clinical validation studies to evaluate the performance of your application in real-world medical settings. Gather feedback from healthcare professionals to assess usability, accuracy, and clinical relevance.
- Integration with Healthcare Systems: Investigate integration strategies to seamlessly integrate your application with existing Healthcare Information Systems (HIS). Explore interoperability standards such as Health Level Seven (HL7) Fast Healthcare Interoperability Resources (FHIR) for data exchange.

Each of these areas represents potential directions for future research and development that build upon what I have built in this work. By exploring these topics, we can contribute to advancing the field of medical image analysis and enhancing the practical application of AI in healthcare.

A
Common Acronyms

CNN	Convolutional Neural Network
AI	Artificial Intelligence
NIH	National Institutes of Health
SQL	Structured Query Language
CSS	Cascading Style Sheets
JS	JavaScript
HTML	HyperText Markup Language
OpenCV	Open Computer Vision Library
CAD	computer-aided detection and diagnosis
NLP	Natural Language Processing
ML	Machine Learning
DL	Deep Learning
PNG	Portable Network Graphics
ORM	Object-Relational Mapper
WSGI	Web Server Gateway Interface
API	Application Programming Interface
JSON	JavaScript Object Notation
NumPy	Numerical Python
HTTP	HyperText Transfert Protocol
HTTPS	HyperText Transfert Protocol Secure
ROC	Receiver Operating Characteristic
AUC	Area Under the Curve
TPR	True Positive Rate
FPR	False Positive Rate
SGD	Stochastic Gradient Descent
MAE	Mean Absolute Error
REST	REpresentational State Transfer
CSV	Comma-Separated Values
RAM	Random Access Memory
GB	GigaBytes
MRI	Magnetic Resonance Imaging
CT	Computed Tomography
HIS	Healthcare Information Systems
HL7	Health Level Seven
FHIR	Fast Healthcare Interoperability Resources

B License of the Documentation

Copyright (c) 2024 Yannick Künzli.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from [12].

C
Repository of the Project

A Github repository for this application was created. The repository of the app is available here: **https://github.com/YannickKunz/ChestVision**
The code for training the model is located in the jupyter notebook. All of the code for the application is located in the folder myApp. After cloning the repository, you also need to download the images on the official website, you can use the code in the first cells of the notebook to download them and unzip them. The structure of the project needs to be myApp/website/static/assets/images, and this is the folder where the images need to be located.

D. Referenced Web Resources

[1] Adadelta. https://keras.io/api/optimizers/adadelta/ (accessed June 20, 2024). 47

[2] Adagrad. https://keras.io/api/optimizers/adagrad/ (accessed June 20, 2024). 47

[3] Adam. https://keras.io/api/optimizers/adam/ (accessed June 20, 2024). 47

[4] Atelectasis diagnosis. https://www.maimonidesem.org/blog/cxr-consolidation-or-atelectasis (accessed May 05, 2024). vi, 11

[5] Bokeh. https://bokeh.org (accessed June 20, 2024). 30

[6] Cardiomegaly diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/cardiac_disease/cardiomegaly (accessed May 05, 2024). vi, 11, 12

[7] ChestVision. https://www.github.com/YannickKunz/ChestVision (accessed June 20, 2024). 2, 36, 52, 68

[8] Consolidation diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/pulmonary-disease/lung_zones (accessed May 05, 2024). vi, 12

[9] NIH Chest X-ray dataset. https://nihcc.app.box.com/v/ChestXray-NIHCC (accessed May 10, 2024). 3, 6, 37

[10] Edema diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/cardiac_disease/pulmonary_oedema (accessed May 05, 2024). vi, 12, 13

[11] Emphysema diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/pulmonary-disease/chest_xray_copd (accessed May 05, 2024). vi, 13, 14

[12] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (accessed July 28, 2005).

[13] Fibrosis diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/pulmonary-disease/pulmonary_fibrosis (accessed May 05, 2024). vi, 14

[14] Flask Documentation. https://flask.palletsprojects.com/en/3.0.x/ (accessed June 18, 2024). 29

[15] Flask-Login Documentation. https://flask-login.readthedocs.io/en/latest/ (accessed June 18, 2024). 29

[16] Flask-Migrate Documentation. https://flask-migrate.readthedocs.io/en/latest/ (accessed June 18, 2024). 29

[17] Flask-SQLAlchemy Documentation. https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/ (accessed June 18, 2024). 29

[18] Github NIH Chest X Rays Classification. https://github.com/paloukari/NIH-Chest-X-rays-Classification/tree/master/src (accessed May 10, 2024). 37, 50, 51

[19] Glob. https://docs.python.org/3/library/glob.html (accessed June 20, 2024). 30

[20] ARDA: Using Artificial Intelligence in Opthalmology - Google Health. https://health.google/caregivers/arda/ (accessed May 10, 2024). 19

[21] Healthy lungs. https://www.radiologymasterclass.co.uk/tutorials/chest/chest_home_anatomy/chest_anatomy_start (accessed May 05, 2024). vi, 15

[22] Hernia diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/mediastinum_hilum/hiatus_hernia (accessed May 05, 2024). vi, 18

[23] Holoviews. https://holoviews.org (accessed June 20, 2024). 30

[24] Infiltration diagnosis. https://bestpractice.bmj.com/topics/en-gb/1094 (accessed May 05, 2024). vi, 14, 15

[25] Itertools. https://docs.python.org/3/library/itertools.html (accessed June 20, 2024). 30

[26] JavaScript. https://developer.mozilla.org/fr/docs/Web/JavaScript (accessed June 20, 2024). 33

[27] NIH Chest X-ray dataset on Kaggle. https://www.kaggle.com/datasets/nih-chest-xrays/data (accessed May 10, 2024).

[28] Keras Website. https://keras.io (accessed June 18, 2024). 29

[29] Lung Diseases Data Analysis by Stephane Bernadac. https://www.kaggle.com/code/sbernadac/lung-deseases-data-analysis (accessed May 10, 2024). 37

[30] Mass diagnosis. https://www.radiologymasterclass.co.uk/gallery/chest/mediastinum_hilum/retrocardiac_mass (accessed May 05, 2024). vi, 15

[31] Matplotlib. https://matplotlib.org (accessed June 20, 2024). 8, 30

[32] MobileNet. https://keras.io/api/applications/mobilenet/ (accessed June 20, 2024). 44

[33] Nodule diagnosis. https://www.healthline.com/health/benign-lung-nodules (accessed May 05, 2024). vi, 16

[34] NumPy. https://numpy.org (accessed June 19, 2024). 8, 30

[35] OpenCV. https://opencv.org (accessed June 19, 2024). 30

[36] Pandas. https://pandas.pydata.org (accessed June 19, 2024). 30

[37] Pleural effusion diagnosis. https://www.radiologymasterclass.co.uk/tutorials/chest/chest_pathology/chest_pathology_page4 (accessed May 05, 2024). vi, 13

[38] Pleural thickening diagnosis. https://www.radiologymasterclass.co.uk/tutorials/chest/chest_pathology/chest_pathology_page4 (accessed May 05, 2024). vi, 16

[39] Pneumonia diagnosis. https://www.mayoclinic.org/diseases-conditions/pneumonia/multimedia/chest-x-ray-showing-pneumonia/img-20005827 (accessed May 05, 2024). vi, 17

[40] Pneumothorax diagnosis. https://www.radiologymasterclass.co.uk/tutorials/chest/chest_pathology/chest_pathology_page4 (accessed May 05, 2024). vi, 17

[41] Python. https://docs.python.org/3/ (accessed June 20, 2024).

[42] Scikit-learn. https://scikit-learn.org/stable/ (accessed June 20, 2024). 30

[43] Seaborn. https://seaborn.pydata.org (accessed June 19, 2024). 30

[44] SGD. https://keras.io/api/optimizers/sgd/ (accessed June 20, 2024). 47

[45] Train Simple XRay CNN on Kaggle by K. Scott Mader. https://www.kaggle.com/code/kmader/train-simple-xray-cnn (accessed May 10, 2024). 37

[46] Tensorflow. https://www.tensorflow.org (accessed June 19, 2024). 29

[47] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald Summers. Chestx-ray14: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. 09 2017.

[48] Werkzeug Security Documentation. https://pypi.org/project/Werkzeug/ (accessed June 18, 2024). 29