PNEUMONIA DETECTION CHALLENGE

Submitted by

Augustine Richard R
Sridharan Chandran
Rohit

Date: 5 Aug 23

—

Pneumonia Detection using Computer Vision

—

Mentor : Jyant Mahara

Table of Contents

PROBLEM STATEMENT

DOMAIN*:* Health Care

CONTEXT*:*

Computer vision can be used in health care for identifying diseases.

Objectives of the Project includes:

1. Detect Inflammation of the lungs.
2. Build an algorithm to detect a visual signal for pneumonia in medical images. Specifically, it needs to automatically locate lung opacities on chest radiographs.

DATA DESCRIPTION*:*

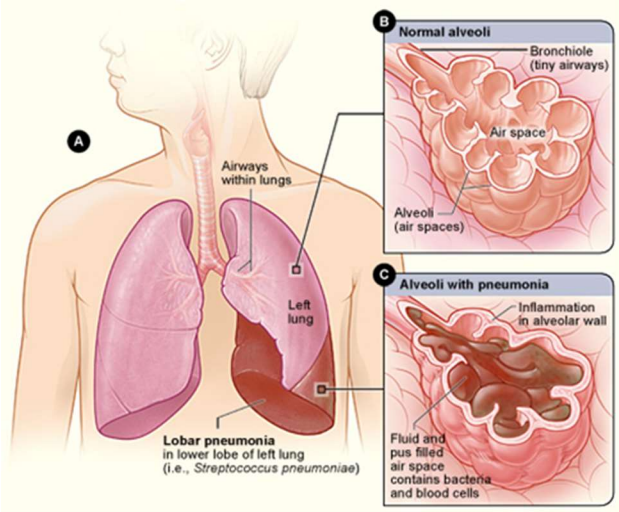
In the dataset, some of the features are labeled “Not Normal No Lung Opacity.” This extra third class indicates that while pneumonia was determined not to be present, there was nonetheless some type of abnormality on the image and oftentimes this finding may mimic the appearance of true pneumonia. Dicom original images: - Medical images are stored in a special format called DICOM files (*.dcm). They contain a combination of header metadata as well as underlying raw image arrays for pixel data. - Dataset has been attached along with this project.

Original link to the dataset: https://www.kaggle.com/c/rsna-pneumonia-detection-challenge/data.

Acknowledgements:
https://www.kaggle.com/c/rsna-pneumonia-detection-challenge/overview/acknowledgements.

What is Pneumonia?

Pneumonia, a lung infection impacting either one or both lungs, triggers the infiltration of air sacs, called alveoli, with fluid or pus. This ailment can be prompted by bacteria, viruses, or fungi. Symptoms span a spectrum from mild to severe and encompass coughing, with or without mucus, fever, chills, and breathing difficulties. The gravity of pneumonia hinges on factors such as age, overall health, and the root cause of the infection.



To pinpoint pneumonia, your medical practitioner will delve into your medical history, conduct a physical assessment, and prescribe diagnostic assessments like a chest X-ray. This data aids in identifying the specific type of pneumonia.

Remedies for pneumonia encompass antibiotics, antiviral, or antifungal medications. Pneumonia recovery might extend over several weeks. In case your symptoms deteriorate, seeking prompt medical attention is vital. Severe pneumonia could necessitate hospitalization, entailing intravenous antibiotic administration and oxygen therapy.

Certain forms of pneumonia can be averted through vaccines. Adhering to good hygiene practices and embracing a heart-healthy lifestyle can also mitigate your vulnerability to pneumonia.

Symptoms of Pneumonia

Pneumonia symptoms range from mild to severe, with greater risk for serious cases in young children, older adults, and individuals with health issues. Symptoms include chest pain, chills, cough, fever, shortness of breath, and more. Older adults and those with weakened immune systems might display atypical symptoms or lower temperature. Babies may experience vomiting, fever, cough, and breathing difficulties, including bluish skin, grunting, and rapid breathing.

Diagnostic tests for pneumonia

- Chest X-ray: Detects lung inflammation, commonly used for diagnosis.
- Blood tests: Like complete blood count (CBC) to assess immune response.
- Pulse oximetry: Measures blood oxygen levels using a pulse oximeter on the finger or ear.
- Hospitalized or severe cases may involve additional tests:
    - Blood gas test: Measures blood oxygen directly from an artery.
    - Sputum test: Analyzes sputum or mucus to identify the germ causing pneumonia.
    - Blood culture test: Identifies pneumonia germ and potential bloodstream infection.
    - Polymerase chain reaction (PCR) test: Quickly detects pneumonia-causing germ DNA.
    - Bronchoscopy: Examines airways, collects lung tissue and fluid samples for diagnosis.
    - Chest CT scan: Reveals lung extent and complications with higher detail than X-rays.
    - Pleural fluid culture: Involves thoracentesis, testing fluid between lungs and chest wall for bacteria.

Causes of Pneumonia

Bacteria:

- Common in adults; Streptococcus pneumoniae is a frequent cause in the US.
- Some bacteria cause atypical pneumonia, e.g., Mycoplasma pneumoniae ("walking pneumonia") and Legionella pneumophila (Legionnaires' disease).
- Can develop independently or after a cold/flu.

Viruses:

- Lung-infecting viruses like influenza and rhinovirus cause viral pneumonia.
- Respiratory syncytial virus (RSV) leads to viral pneumonia in young children.
- Other viruses, including SARS-CoV-2 (COVID-19), can cause pneumonia.

Fungi:

- Pneumocystis jirovecii causes pneumonia, especially in weakened immune systems.
- Some soil-based fungi can also trigger pneumonia.
-

Risk Factors:

Age:

- Higher risk for babies (<2 years old) due to developing immune systems.
- Older adults (≥65) at risk due to weakened immunity and chronic health conditions.

Environment/Occupation:

- Crowded places like military barracks, shelters, nursing homes increase risk.
- Air pollution and toxic fumes exposure heightens risk.
- Working with animals (veterinary, pet shops) or in poultry processing centers increases risk.

Lifestyle Habits:

- Smoking weakens airway defense.
- Drug/alcohol use weakens the immune system.

Other Medical Conditions:

- Brain disorders affecting cough/swallowing raise aspiration risk.

- Immune-weakening conditions (HIV/AIDS, transplants) increase susceptibility.
- Critical illnesses, hospitalization, and ventilator use heighten risk.
- Lung diseases (asthma, COPD) raise pneumonia risk.
- Serious conditions (diabetes, heart failure) are additional risk factors.

MILESTONE 1

1: Import the data

Download and install the `pydicom` library, which can be used in the code to work with DICOM files

```
!pip install pydicom
```

```
Requirement already satisfied: pydicom in /opt/conda/lib/python3.10/site-packages (2.4.1)
```

Import necessary libraries to be use in this Project

```
#Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder

import tensorflow as tf
from keras.optimizers import Adam
import scipy
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Convolution2D, Activation, SpatialDropout2D
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D, MaxPooling2D, MaxPool2D, GlobalMaxPooling2D, BatchNormalization
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, f1_score, precision_recall_curve, auc,classification_report, roc_curve
from tensorflow.keras import optimizers
from tensorflow.keras.applications.inception_v3 import InceptionV3
from keras import regularizers
from keras.optimizers import SGD, Adam
from tensorflow.keras import backend
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
from keras.wrappers.scikit_learn import KerasClassifier
from keras import optimizers
from keras.utils import np_utils
from sklearn.preprocessing import StandardScaler
from glob import glob
import cv2
from collections import Counter
from sklearn.model_selection import train_test_split
import os
from zipfile import ZipFile
from pathlib import Path
from tqdm import tqdm_notebook
from matplotlib.patches import Rectangle
from keras.layers import Input, Conv2D, UpSampling2D
from keras.models import Model
import random
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.utils import Sequence
```

Read the following:

1. Class information from the CSV file

2. Training labels from the CSV file
3. Define the path to the directory containing training images
4. Define the path to the directory containing test images
5. Create a Path object for the sample submission CSV file

```
classInfo = pd.read_csv('/kaggle/input/pneumonia-detection/stage_2_detailed_class_info.csv')
trainlabels = pd.read_csv('/kaggle/input/pneumonia-detection/stage_2_train_labels.csv')
trainImagesPath = "/kaggle/input/pneumonia-detection/stage_2_train_images/stage_2_train_images"
testImagesPath = "/kaggle/input/pneumonia-detection/stage_2_test_images/stage_2_test_images"
sampleSubPath = Path('/kaggle/input/pneumonia-detection/stage_2_sample_submission.csv')
```

Read the first 5 rows of the class information data frame to understand the structure.

```
classInfo.head()
```

	patientId	class
0	0004cfab-14fd-4e49-80ba-63a80b6bddd6	No Lung Opacity / Not Normal
1	00313ee0-9eaa-42f4-b0ab-c148ed3241cd	No Lung Opacity / Not Normal
2	00322d4d-1c29-4943-afc9-b6754be640eb	No Lung Opacity / Not Normal
3	003d8fa0-6bf1-40ed-b54c-ac657f8495c5	Normal
4	00436515-870c-4b36-a041-de91049b9ab4	Lung Opacity

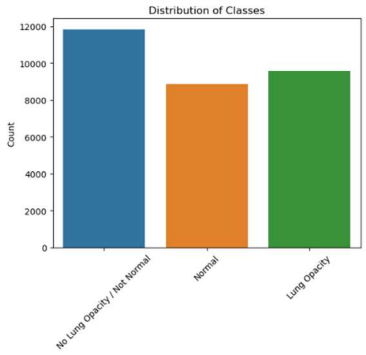
Read the class info to understand number of rows and columns. ‘classinfo’ dataset has 2 columns and 30227 rows.

```
classInfo.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30227 entries, 0 to 30226
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   patientId  30227 non-null  object
 1   class      30227 non-null  object
dtypes: object(2)
memory usage: 472.4+ KB
```

Displaying countplot to understand the number of classes and its counts.

```
sns.countplot(x='class',data=classInfo);

# Displaying labels, title, and rotate the x-axis labels for better visualization
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Distribution of Classes')
plt.xticks(rotation=45)

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



Define a function 'get_feature_distribution' to get actual counts and % of different classes in the dataset. From the result below, it is found that there are 3 classes: No lung Opacity / Not Normal, Lung Capacity & normal.
Counts and % for each of the class is provided in the result below after calling the function feeding the variables 'classInfo' (data frame name) and 'class' (target column name).

```
def get_feature_distribution(data, feature):
    # Get the count for each label
    label_counts = data[feature].value_counts()

    # Get total number of samples
    total_samples = len(data)

    # Count the number of items in each class
    print("Feature: {}".format(feature))
    for i in range(len(label_counts)):
        label = label_counts.index[i]
        count = label_counts.values[i]
        percent = int((count / total_samples) * 10000) / 100
        print("{:<30s}:   {} or {}%".format(label, count, percent))

get_feature_distribution(classInfo, 'class')
```

```
Feature: class
No Lung Opacity / Not Normal  :   11821 or 39.1%
Lung Opacity                  :   9555 or 31.61%
Normal                        :   8851 or 29.28%
```

Check for duplicate rows in the 'classinfo' dataset. 3543 duplicate rows has been identified from the below result.

```
classInfo[classInfo.duplicated()]
```

	patientId	class
5	00436515-870c-4b36-a041-de91049b9ab4	Lung Opacity
9	00704310-78a8-4b38-8475-49f4573b2dbb	Lung Opacity
15	00aecb01-a116-45a2-956c-08d2fa55433f	Lung Opacity
17	00c0b293-48e7-4e16-ac76-9269ba535a62	Lung Opacity
20	00f08de1-517e-4652-a04f-d1dc9ee48593	Lung Opacity
...	...	...
30209	c18d1138-ba74-4af5-af21-bdd4d2c96bb5	Lung Opacity
30215	c1cddf32-b957-4753-acaa-472ab1447e86	Lung Opacity
30220	c1e73a4e-7afe-4ec5-8af6-ce8315d7a2f2	Lung Opacity
30222	c1ec14ff-f6d7-4b38-b0cb-fe07041cbdc8	Lung Opacity
30226	c1f7889a-9ea9-4acb-b64c-b737c929599a	Lung Opacity

3543 rows × 2 columns

We shall check the shape of the duplicate rows and also a sample ‘patientId’ to understand the difference.

The below sample denotes that the same record is created twice for the same patient.

```
classInfo[classInfo.duplicated()].shape
```

```
(3543, 2)
```

```
classInfo[classInfo.patientId=='c1f7889a-9ea9-4acb-b64c-b737c929599a']
```

	patientId	class
30225	c1f7889a-9ea9-4acb-b64c-b737c929599a	Lung Opacity
30226	c1f7889a-9ea9-4acb-b64c-b737c929599a	Lung Opacity

Checking for Missing Values

+ Code + Markdown

```
# Defining a function to check for missing values in each column of a DataFrame
def missing_check(df):
    # Calculate the total number of missing values in each column and sort in descending order
    total = df.isnull().sum().sort_values(ascending=False)

    # Calculate the percentage of missing values in each column and sort in descending order
    percent = (df.isnull().sum() / df.isnull().count()).sort_values(ascending=False)

    # Combine the 'total' and 'percent' Series into a new DataFrame named 'missing_data'
    missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])

    # Return the DataFrame containing information about missing values
    return missing_data
```

```
# Call the missing_check function to check for missing values in each column of the DataFrame classInfo
missing_data = missing_check(classInfo)


# Print the DataFrame containing information about missing values
print(missing_data)
```

```
           Total  Percent
patientId      0      0.0
class          0      0.0
```

No Missing Values found in the dataset

To understand better, we checked a particular record which had duplicate rows

```
# It selects rows from the DataFrame where the 'patientId' column has the value 'c1f7889a-9ea9-4acb-b64c-b737c929599a'.
selected_data = trainlabels[trainlabels.patientId == 'c1f7889a-9ea9-4acb-b64c-b737c929599a']

# Set the option to display all columns in a single line
pd.set_option('display.expand_frame_repr', False)

# Print the DataFrame containing the selected rows
print(selected_data)
```

```
                          patientId      x      y  width  height  Target
30225  c1f7889a-9ea9-4acb-b64c-b737c929599a  570.0  393.0  261.0   345.0       1
30226  c1f7889a-9ea9-4acb-b64c-b737c929599a  233.0  424.0  201.0   356.0       1
```

From the above result, it is observed that 2 Records exists for same patient with different dimension

This may be due to in the X-ray opacity has been deteched at multiple locations

+ Code + Markdown

```
# It selects rows from the DataFrame where all columns have duplicate values, indicating duplicate rows.
duplicated_rows = trainlabels[trainlabels.duplicated()]

# Print the DataFrame containing the duplicate rows
print(duplicated_rows)
```

```
Empty DataFrame
Columns: [patientId, x, y, width, height, Target]
Index: []
```

No Duplicate Records found from the above result

```
plt.figure()  # Create a new figure to ensure the previous plot doesn't overlap
sns.countplot(x='Target', data=trainlabels, color='skyblue',orient='h')
plt.title('Distribution of Target Column')
plt.xlabel('Target Values')
plt.ylabel('Count')

# Display all the countplots
plt.show()
```

There are 2 values in Target column 0 & 1

In classInfo Dataframe, we could see 3 different classes

Lets proceed further to check how Target and Class columns are related.

The data looks imbalanced. We shall perform balancing data by up sampling in Milestone 2

2.Map training and testing images to its classes

Next, we explored the training labels dataset. This ha 6 columns and 26684 dataset. Our objective is to align the number of rows in the 'classInfo' dataset with the 'trainlabels' dataset. So, we explored more to remove the duplicate records in the 'classinfo' dataset.

```
trainlabels.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30227 entries, 0 to 30226
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   patientId  30227 non-null  object
 1   x          9555 non-null   float64
 2   y          9555 non-null   float64
 3   width      9555 non-null   float64
 4   height     9555 non-null   float64
 5   Target     30227 non-null  int64
dtypes: float64(4), int64(1), object(1)
memory usage: 1.4+ MB
```

```
trainlabels.patientId.nunique()
```

```
26684
```

More analysis for missing values:

```
# Call the missing_check function to check for missing values in each column of the DataFrame 'trainlabels'
missing_data_trainlabels = missing_check(trainlabels)

# Print the DataFrame containing information about missing values
print(missing_data_trainlabels)
```

```
           Total   Percent
x          20672  0.683892
y          20672  0.683892
width      20672  0.683892
height     20672  0.683892
patientId      0  0.000000
Target         0  0.000000
```

```
# Select rows where the 'x' column has missing values (NaN)
missing_x_rows = trainlabels[trainlabels.x.isna()]

# Print the DataFrame containing rows with missing values in the 'x' column
print(missing_x_rows)
```

```
                              patientId   x   y  width  height  Target
0      0004cfab-14fd-4e49-80ba-63a80b6bddd6 NaN NaN    NaN     NaN       0
1      00313ee0-9eaa-42f4-b0ab-c148ed3241cd NaN NaN    NaN     NaN       0
2      00322d4d-1c29-4943-afc9-b6754be640eb NaN NaN    NaN     NaN       0
3      003d8fa0-6bf1-40ed-b54c-ac657f8495c5 NaN NaN    NaN     NaN       0
6      00569f44-917d-4c86-a842-81832af98c30 NaN NaN    NaN     NaN       0
...                                     ...  ..  ..    ...     ...     ...
30216  c1cf3255-d734-4980-bfe0-967902ad7ed9 NaN NaN    NaN     NaN       0
30217  c1e228e4-b7b4-432b-a735-36c48fdb806f NaN NaN    NaN     NaN       0
30218  c1e3eb82-c55a-471f-a57f-fe1a823469da NaN NaN    NaN     NaN       0
30223  c1edf42b-5958-47ff-a1e7-4f23d99583ba NaN NaN    NaN     NaN       0
30224  c1f6b555-2eb1-4231-98f6-50a963976431 NaN NaN    NaN     NaN       0

[20672 rows x 6 columns]
```

Some of X,Y values found to have missing for few records . This can be due to the fact that for a normal patient these values could be not applicable.

```
# Select rows where the 'x' column has missing values (NaN)
# The expression 'trainlabels.x.isna()' returns a boolean mask with 'True' for rows where 'x' is NaN and 'False' otherwise.
# This boolean mask is then used as an index to select specific rows from the DataFrame 'trainlabels'.
# The result is a new DataFrame containing only the rows where the 'x' column has missing values.
missing_x_rows = trainlabels[trainlabels.x.isna()]

# Retrieve the unique values in the 'Target' column for the selected rows
# The 'Target' column of the DataFrame 'missing_x_rows' is accessed using ['Target'].
# The 'unique()' method is then called on this column to retrieve the unique values in the 'Target' column.
unique_target_values = missing_x_rows['Target'].unique()

# Print the unique values in the 'Target' column for the selected rows
print(unique_target_values)
```

```
[0]
```

From the above output, it is confirmed that for normal patients dimensions are not available.

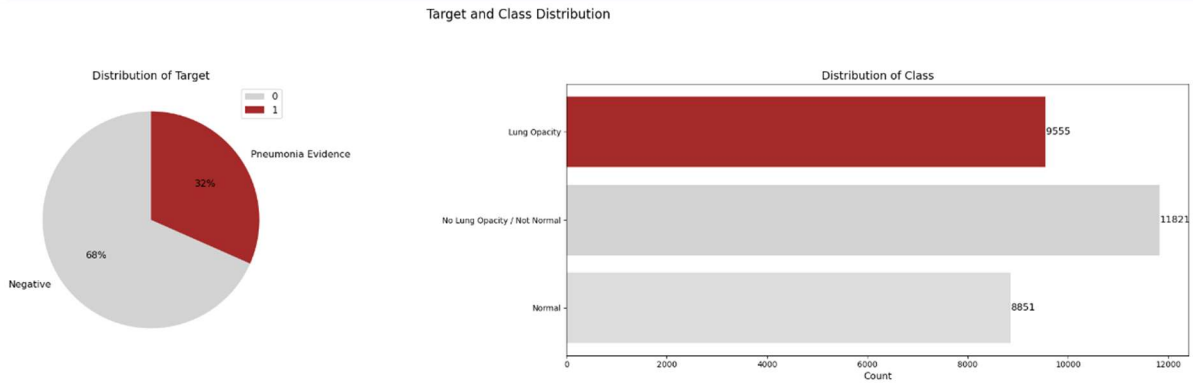
```
# The code calls the 'describe()' method on the DataFrame 'trainlabels' to get summary statistics of the numerical columns.

# Display summary statistics of the DataFrame 'trainlabels'
# The 'describe()' method computes various summary statistics for each numerical column in the DataFrame.
# The statistics include count, mean, standard deviation, minimum value, 25th percentile (Q1), median (50th percentile or Q2),
# 75th percentile (Q3), and maximum value.
# The output will show these statistics for each numerical column in 'trainlabels'.
summary_stats = trainlabels.describe()

# Print the summary statistics of the DataFrame 'trainlabels'
print(summary_stats)
```

	x	y	width	height	Target
count	9555.000000	9555.000000	9555.000000	9555.000000	30227.000000
mean	394.047724	366.839560	218.471376	329.269702	0.316108
std	204.574172	148.940488	59.289475	157.750755	0.464963
min	2.000000	2.000000	40.000000	45.000000	0.000000
25%	207.000000	249.000000	177.000000	203.000000	0.000000
50%	324.000000	365.000000	217.000000	298.000000	0.000000
75%	594.000000	478.500000	259.000000	438.000000	1.000000
max	835.000000	881.000000	528.000000	942.000000	1.000000

Looks , 75% of the data represents target value 1. Count plot also shows the same. Hence,it is an imbalanced data set.

To understand the distribution of Target variables, we plot them in a pie chart & bar chart.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 7))

# Plot the percentage distribution of 'Target' column using a pie chart
target_counts = trainlabels['Target'].value_counts()
ax1.pie(target_counts, labels=['Negative', 'Pneumonia Evidence'], colors=['lightgray', 'brown'],
        autopct='%.0f%%', startangle=90, textprops={'fontsize': 12})
ax1.set_title('Distribution of Target', fontsize=14)

# Plot the percentage distribution of 'class' column using a horizontal bar plot
class_counts = classInfo['class'].value_counts().sort_index(ascending=False)
ax2.barh(class_counts.index, class_counts, color=['gainsboro', 'lightgray', 'brown'])
ax2.set_xlabel('Count', fontsize=12)
ax2.set_title('Distribution of Class', fontsize=14)

# Add data labels to the bar plot
for i, v in enumerate(class_counts):
    ax2.text(v + 10, i, str(v), fontsize=12, color='black', va='center')

# Add a legend for the pie chart
ax1.legend(target_counts.index, loc='best', fontsize=12)

# Adjust the layout for subplots and add an overall title for the figure
plt.suptitle('Target and Class Distribution', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.9])  # Adjust the 'rect' parameter to control the figure layout

# Show the plots
plt.show()
```



Lets concatenate classInfo and trainlabels . Before concatinating them lets remove duplicate records from class info

```
# Drop duplicate rows from the DataFrame 'classInfo'
classInfo.drop_duplicates(inplace=True)

# Print the DataFrame after dropping duplicates
print(classInfo)
```

```
                              patientId                          class
0      0004cfab-14fd-4e49-80ba-63a80b6bddd6  No Lung Opacity / Not Normal
1      00313ee0-9eaa-42f4-b0ab-c148ed3241cd  No Lung Opacity / Not Normal
2      00322d4d-1c29-4943-afc9-b6754be640eb  No Lung Opacity / Not Normal
3      003d8fa0-6bf1-40ed-b54c-ac657f8495c5                      Normal
4      00436515-870c-4b36-a041-de91049b9ab4                Lung Opacity
...                                     ...                         ...
30219  c1e73a4e-7afe-4ec5-8af6-ce8315d7a2f2                Lung Opacity
30221  c1ec14ff-f6d7-4b38-b0cb-fe07041cbdc8                Lung Opacity
30223  c1edf42b-5958-47ff-a1e7-4f23d99583ba                      Normal
30224  c1f6b555-2eb1-4231-98f6-50a963976431                      Normal
30225  c1f7889a-9ea9-4acb-b64c-b737c929599a                Lung Opacity

[26684 rows x 2 columns]
```

```
# The code performs an inner join (merge) between the 'trainlabels' and 'classInfo' DataFrames based on the 'patientId' column.

# Merge the DataFrames 'trainlabels' and 'classInfo' based on the common 'patientId' column.
# The 'left_on' parameter specifies the column in the left DataFrame (trainlabels) to use for the merge.
# The 'right_on' parameter specifies the column in the right DataFrame (classInfo) to use for the merge.
# The 'how' parameter is set to 'inner', which means only the rows with matching 'patientId' values in both DataFrames will be included in the result.

traindf = trainlabels.merge(classInfo, left_on='patientId', right_on='patientId', how='inner')
```

```
traindf.head() #Top 5 rows for traiing dataset
```

	patientId	x	y	width	height	Target	class
0	0004cfab-14fd-4e49-80ba-63a80b6bddd6	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal
1	00313ee0-9eaa-42f4-b0ab-c148ed3241cd	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal
2	00322d4d-1c29-4943-afc9-b6754be640eb	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal
3	003d8fa0-6bf1-40ed-b54c-ac657f8495c5	NaN	NaN	NaN	NaN	0	Normal
4	00436515-870c-4b36-a041-de91049b9ab4	264.0	152.0	213.0	379.0	1	Lung Opacity

+ Code + Markdown

Lets check for specific patient how data has been concatenated.

```
# The code filters the DataFrame 'traindf' to select rows where the 'patientId' column has the value 'c1f7889a-9ea9-4acb-b64c-b737c929599a'.

# Select rows from the DataFrame 'traindf' where the 'patientId' column is equal to 'c1f7889a-9ea9-4acb-b64c-b737c929599a'.
filtered_traindf = traindf[traindf.patientId == 'c1f7889a-9ea9-4acb-b64c-b737c929599a']

# Print the output to see the DataFrame with filtered rows
print(filtered_traindf)
```

```
                              patientId      x      y  width  height  Target         class
30225  c1f7889a-9ea9-4acb-b64c-b737c929599a  570.0  393.0  261.0   345.0       1  Lung Opacity
30226  c1f7889a-9ea9-4acb-b64c-b737c929599a  233.0  424.0  201.0   356.0       1  Lung Opacity
```

Good, same patient has different dimension values with same target value

3. Map training and testing images to its annotations

Now lets see how class and target are related to each other

```
# The code retrieves the unique values from the 'Target' column in the DataFrame 'traindf'.

# Get the unique values from the 'Target' column in the DataFrame 'traindf'.
unique_targets = traindf.Target.unique()

# Print the unique values to see the distinct target values in the 'traindf' DataFrame.
print(unique_targets)
```

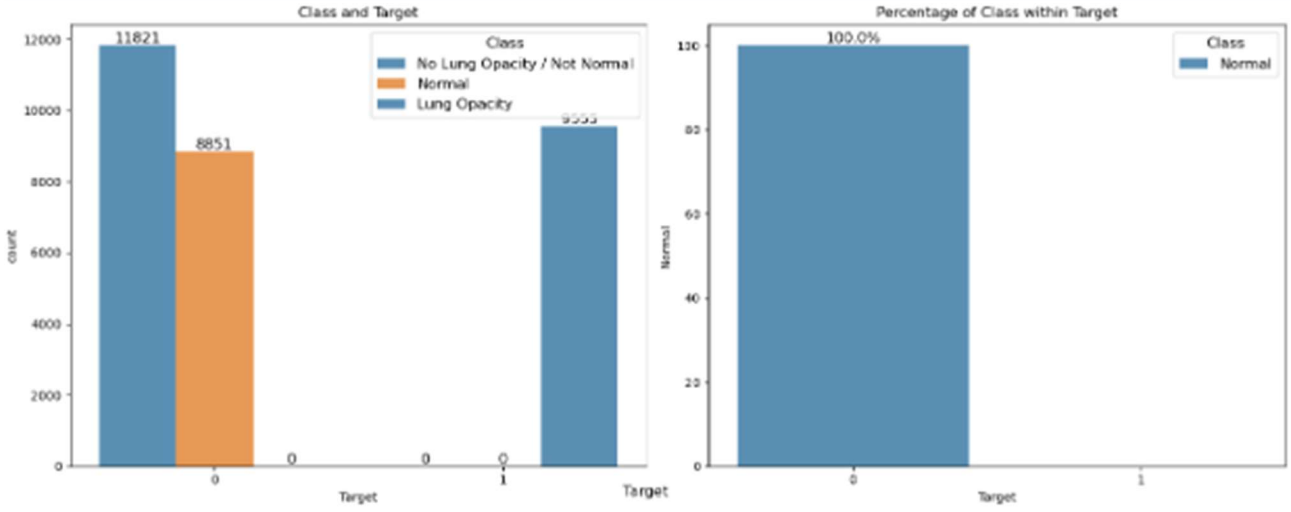
```
[0 1]
```

```
# The code groups the DataFrame 'traindf' by the columns 'class' and 'Target', and then calculates the size of each group (i.e., the number of patients) within each com

# Group the DataFrame 'traindf' by the columns 'class' and 'Target', and calculate the size of each group (number of patients).
grouped_traindf = traindf.groupby(['class', 'Target']).size().reset_index(name='Patient Count')

# Print the output to see the DataFrame with the grouped results and patient counts.
print(grouped_traindf)
```

```
                        class  Target  Patient Count
0                Lung Opacity       1           9555
1  No Lung Opacity / Not Normal       0          11821
2                      Normal       0           8851
```

4: Preprocessing and Visualization of different classes

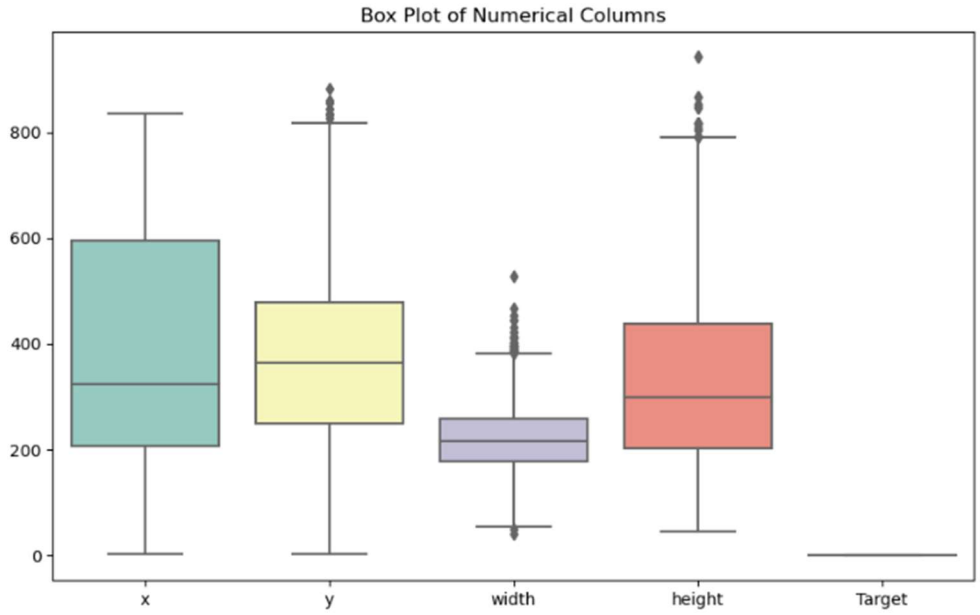
```
# Create a figure with two subplots side by side and set the figure size to (15, 6).
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(15, 6))

# Group the DataFrame 'traindf' by 'Target' and 'class', and calculate the count of each unique combination.
tmp = traindf.groupby(['Target', 'class']).size().reset_index(name='count')

# Create a bar plot using Seaborn's 'sns.barplot'.
# 'x' represents the 'Target' column, 'y' represents the 'count' column (count of each class per Target),
# and 'hue' represents the 'class' column to distinguish the classes in the plot.
# 'palette' sets the color palette for the bars.
# 'alpha' adjusts the transparency of the bars to improve visibility.
sns.barplot(ax=ax1, x='Target', y='count', hue='class', data=tmp, palette=['#1f77b4', '#ff7f0e'], alpha=0.8)

# Add data labels to the bars to display the exact count values.
for p in ax1.patches:
    height = p.get_height()
    if pd.notnull(height):  # Check if height is not NaN before converting to integer
        ax1.annotate(f'{int(height)}', (p.get_x() + p.get_width() / 2, height), ha='center', va='bottom', fontsize=12)
    else:  # Handle NaN values and display "0" for zero counts
        ax1.annotate('0', (p.get_x() + p.get_width() / 2, 0), ha='center', va='bottom', fontsize=12)

# Set a title for the first plot.
ax1.set_title("Class and Target")

# Set the legend title and position for better readability in the first plot.
ax1.legend(title='Class', title_fontsize=12, fontsize=12, loc='upper right')

# Calculate the percentage of each class within each 'Target' group.
percentage_df = tmp.pivot_table(index='Target', columns='class', values='count', aggfunc=lambda x: x / x.sum() * 100).reset_index()

# Create a percentage bar chart using Seaborn's 'sns.barplot' for the second subplot.
sns.barplot(ax=ax2, x='Target', y='Normal', data=percentage_df, color='#1f77b4', label='Normal', alpha=0.8)

# Check if the 'Pneumonia' class is present in 'percentage_df'.
if 'Pneumonia' in percentage_df.columns:
    sns.barplot(ax=ax2, x='Target', y='Pneumonia', data=percentage_df, color='#ff7f0e', label='Pneumonia', alpha=0.8)
else:
    percentage_df['Pneumonia'] = 0

# Add data labels to the percentage bars to display the exact percentage values.
for p in ax2.patches:
    height = p.get_height()
    if pd.notnull(height):  # Check if height is not NaN before displaying data labels
        ax2.annotate(f'{height:.1f}%', (p.get_x() + p.get_width() / 2, height), ha='center', va='bottom', fontsize=12)

# Set a title for the second plot.
ax2.set_title("Percentage of Class within Target")

# Set the legend title and position for better readability in the second plot.
ax2.legend(title='Class', title_fontsize=12, fontsize=12, loc='upper right')

# Set common y-axis label for both subplots.
fig.text(0.5, 0.04, 'Target', ha='center', fontsize=12)

# Show the plots.
plt.tight_layout()
plt.show()
```



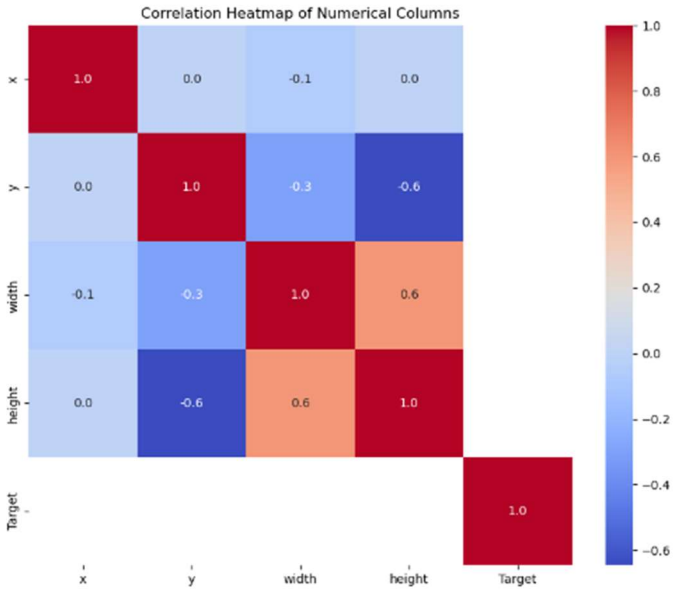
As suspected, class with Normal and No Lung Opacity / Not Normal has been classified into single target value that is 'O'. So we can summarize that the prediction which we have to do is like Patient has Lung Opacity or not.
Because Normal and not normal patients are combines in same Target.

The below boxplot denotes spread of data in each variable in ‘traindf’ data frame.

```
# Set the figure size to (10, 6) for better visibility.
plt.figure(figsize=(10, 6))

# Create the box plot using Seaborn.
sns.boxplot(data=traindf, palette='Set3')

# Add a title to the plot.
plt.title("Box Plot of Numerical Columns")

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



```
# Let's compute the correlation matrix for the 'traindf' DataFrame:
correlation_matrix = traindf.corr()

# Create a heatmap to visualize the correlation matrix using Seaborn's 'sns.heatmap' function.

# Set the figure size to (10, 8) for better visibility.
plt.figure(figsize=(10, 8))

# Create the heatmap using Seaborn.
# 'annot' parameter displays the correlation coefficient values on the heatmap.
# 'fmt' parameter sets the format of the annotation to show correlation coefficients with one decimal point.
# 'cmap' parameter sets the color map for the heatmap.
sns.heatmap(correlation_matrix, annot=True, fmt=".1f", cmap="coolwarm")

# Add a title to the plot.
plt.title("Correlation Heatmap of Numerical Columns")

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

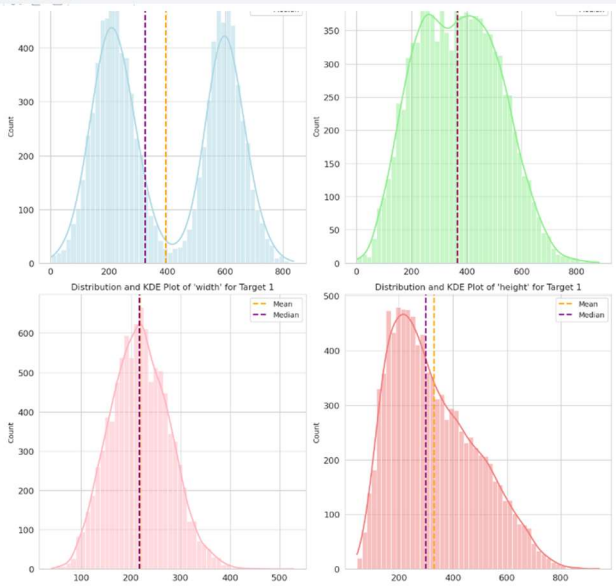


The correlation data is shown as heatmap. The following inferences are made from them:

1. ‘width’ has highest postive correlation with ‘height’
2. ‘y’ has highest negative correlation with ‘height’

We have plot a histogram to understand the data distribution of each of the feature in traindf data frame for target values = 1.

```
# Create a figure and subplots with a size of (12, 12).
fig, ax = plt.subplots(2, 2, figsize=(12, 12))

# Plot the distribution of 'x' with a histogram and a KDE plot on the first subplot.
sns.histplot(target1['x'], bins=50, color="lightblue", ax=ax[0, 0], kde=True, line_kws={'color': 'red', 'lw': 2})
mean_x = target1['x'].mean()
median_x = target1['x'].median()
ax[0, 0].axvline(mean_x, color='orange', linestyle='dashed', linewidth=2, label='Mean')
ax[0, 0].axvline(median_x, color='purple', linestyle='dashed', linewidth=2, label='Median')
ax[0, 0].legend()
ax[0, 0].set_title("Distribution and KDE Plot of 'x' for Target 1", fontsize=12)

# Plot the distribution of 'y' with a histogram and a KDE plot on the second subplot.
sns.histplot(target1['y'], bins=50, color="lightgreen", ax=ax[0, 1], kde=True, line_kws={'color': 'blue', 'lw': 2})
mean_y = target1['y'].mean()
median_y = target1['y'].median()
ax[0, 1].axvline(mean_y, color='orange', linestyle='dashed', linewidth=2, label='Mean')
ax[0, 1].axvline(median_y, color='purple', linestyle='dashed', linewidth=2, label='Median')
ax[0, 1].legend()
ax[0, 1].set_title("Distribution and KDE Plot of 'y' for Target 1", fontsize=12)

# Plot the distribution of 'width' with a histogram and a KDE plot on the third subplot.
sns.histplot(target1['width'], bins=50, color="lightpink", ax=ax[1, 0], kde=True, line_kws={'color': 'red', 'lw': 2})
mean_width = target1['width'].mean()
median_width = target1['width'].median()
ax[1, 0].axvline(mean_width, color='orange', linestyle='dashed', linewidth=2, label='Mean')
ax[1, 0].axvline(median_width, color='purple', linestyle='dashed', linewidth=2, label='Median')
ax[1, 0].legend()
ax[1, 0].set_title("Distribution and KDE Plot of 'width' for Target 1", fontsize=12)

# Plot the distribution of 'height' with a histogram and a KDE plot on the fourth subplot.
sns.histplot(target1['height'], bins=50, color="lightcoral", ax=ax[1, 1], kde=True, line_kws={'color': 'blue', 'lw': 2})
mean_height = target1['height'].mean()
median_height = target1['height'].median()
ax[1, 1].axvline(mean_height, color='orange', linestyle='dashed', linewidth=2, label='Mean')
ax[1, 1].axvline(median_height, color='purple', linestyle='dashed', linewidth=2, label='Median')
ax[1, 1].legend()
ax[1, 1].set_title("Distribution and KDE Plot of 'height' for Target 1", fontsize=12)

# Remove x-axis labels from all plots to avoid clutter.
for i in range(2):
    for j in range(2):
        ax[i, j].set_xlabel('')

# Adjust the tick label font size for better readability.
for ax_row in ax:
    for ax_col in ax_row:
        ax_col.tick_params(axis='both', which='major', labelsize=12)

# Adjust the layout to avoid overlapping titles and labels.
plt.tight_layout()

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



From the above results, following inferences are made:

1. ‘width’ represents a bimodal distribution with 2 peaks above 0.0025
2. ‘y’ & ‘height’ are right skewed with ‘height’ being predominant.
3. ‘width’ represents a normal distribution.

Define the path to the directory containing train & test images:

```
# Define the file paths for the train and test images directories using the 'Path' class.

# File path for the directory containing the train images.
trainImagesPath1 = Path("/kaggle/input/pneumonia-detection/stage_2_train_images/stage_2_train_images")

# File path for the directory containing the test images.
testImagesPath1 = Path("/kaggle/input/pneumonia-detection/stage_2_test_images/stage_2_test_images")
```

```
# 'trainImagesPath1' is the directory path containing the train images.
# 'testImagesPath1' is the directory path containing the test images.

# List all the filenames (images) present in the 'trainImagesPath1' directory.
image_train_path = os.listdir(trainImagesPath1)

# List all the filenames (images) present in the 'testImagesPath1' directory.
image_test_path = os.listdir(testImagesPath1)

# Print the number of images in the train set and the test set.
print("Number of images in train set:", len(image_train_path), "\nNumber of images in test set:", len(image_test_path))
```

```
Number of images in train set: 26684
Number of images in test set: 3000
```

Train images length matched with unique patient id's in traindf.

samplePatientID is extracted from the first row of the ‘traindf’ data frame. ‘dcm’ contain the DICOM metadata and pixel data from the selected file.

```
# Get the patient ID from the first three rows of the 'traindf' DataFrame and convert it to a list.
# Then, select the first element (the patient ID) from the list.
samplePatientID = list(traindf[:3].T.to_dict().values())[0]['patientId']

# Create a file path to the DICOM file corresponding to the selected patient ID.
# The 'trainImagesPath1' variable should contain the path to the directory containing the DICOM images.
dcm_path = trainImagesPath1 / samplePatientID

# Change the file extension of the file path to '.dcm' (assuming the original file has a different extension).
dcm_path = dcm_path.with_suffix(".dcm")

# Read the DICOM file using the 'pydicom.read_file()' function and store it in the 'dcm' variable.
dcm = pydicom.read_file(dcm_path)

# Print the DICOM data.
print(dcm)
```

```
Dataset.file_meta -------------------------------
(0002, 0000) File Meta Information Group Length  UL: 202
(0002, 0001) File Meta Information Version       OB: b'\x00\x01'
(0002, 0002) Media Storage SOP Class UID         UI: Secondary Capture Image Storage
(0002, 0003) Media Storage SOP Instance UID      UI: 1.2.276.0.7230010.3.1.4.8323329.28530.1517874485.775526
(0002, 0010) Transfer Syntax UID                 UI: JPEG Baseline (Process 1)
(0002, 0012) Implementation Class UID            UI: 1.2.276.0.7230010.3.0.3.6.0
(0002, 0013) Implementation Version Name         SH: 'OFFIS_DCMTK_360'
-------------------------------------------------
(0008, 0005) Specific Character Set              CS: 'ISO_IR 100'
(0008, 0016) SOP Class UID                       UI: Secondary Capture Image Storage
(0008, 0018) SOP Instance UID                    UI: 1.2.276.0.7230010.3.1.4.8323329.28530.1517874485.775526
(0008, 0020) Study Date                          DA: '19010101'
(0008, 0030) Study Time                          TM: '000000.00'
(0008, 0050) Accession Number                    SH: ''
(0008, 0060) Modality                            CS: 'CR'
(0008, 0064) Conversion Type                     CS: 'WSD'
(0008, 0090) Referring Physician's Name          PN: ''
(0008, 103e) Series Description                  LO: 'view: PA'
(0010, 0010) Patient's Name                      PN: '0004cfab-14fd-4c49-80ba-63a80b6bddd6'
(0010, 0020) Patient ID                          LO: '0004cfab-14fd-4c49-80ba-63a80b6bddd6'
(0010, 0030) Patient's Birth Date                DA: ''
(0010, 0040) Patient's Sex                       CS: 'F'
(0010, 1010) Patient's Age                       AS: '51'
(0018, 0015) Body Part Examined                  CS: 'CHEST'
(0018, 5101) View Position                       CS: 'PA'
(0020, 000d) Study Instance UID                  UI: 1.2.276.0.7230010.3.1.2.8323329.28530.1517874485.775525
(0020, 000e) Series Instance UID                 UI: 1.2.276.0.7230010.3.1.3.8323329.28530.1517874485.775524
(0020, 0010) Study ID                            SH: ''
(0020, 0011) Series Number                       IS: '1'
(0020, 0013) Instance Number                     IS: '1'
(0020, 0020) Patient Orientation                 CS: ''
(0028, 0002) Samples per Pixel                   US: 1
(0028, 0004) Photometric Interpretation          CS: 'MONOCHROME2'
(0028, 0010) Rows                                US: 1024
(0028, 0011) Columns                             US: 1024
(0028, 0030) Pixel Spacing                       DS: [0.143000000000000002, 0.143000000000000002]
(0028, 0100) Bits Allocated                      US: 8
(0028, 0101) Bits Stored                         US: 8
(0028, 0102) High Bit                            US: 7
(0028, 0103) Pixel Representation                US: 0
(0028, 2110) Lossy Image Compression             CS: '01'
(0028, 2114) Lossy Image Compression Method      CS: 'ISO_10918_1'
(7fe0, 0010) Pixel Data                          OB: Array of 142006 elements
```

+ Code + Markdown

We have available some useful information in the DICOM metadata with predictive value, for example: Patient sex; Patient age; Modality; Body part examined; View position; Rows & Columns; Pixel Spacing.

5: Display images with bounding box

We have created a function ‘show_dicom_images’ to visualize DICOM images along with associated metadata.

```
# Define a function 'show_dicom_images' that takes 'data' as input (DataFrame containing DICOM information).
def show_dicom_images(data):
    # Convert the DataFrame 'data' to a list of dictionaries and store it in 'img_data'.
    img_data = list(data.T.to_dict().values())

    # Create a 3x3 grid of subplots using matplotlib with a figure size of 16x18.
    f, ax = plt.subplots(3, 3, figsize=(16, 18))

    # Loop through each item in 'img_data' and extract DICOM information.
    for i, data_row in enumerate(img_data):
        # Create a file path to the DICOM file corresponding to the patient ID.
        dcm_path = trainImagesPath1 / data_row['patientId']
        dcm_path = dcm_path.with_suffix(".dcm")

        # Read the DICOM file using 'pydicom.read_file()' and store the DICOM data in 'data_row_img_data'.
        data_row_img_data = pydicom.read_file(dcm_path)

        # Extract relevant metadata from the DICOM data.
        modality = data_row_img_data.Modality
        age = data_row_img_data.PatientAge
        sex = data_row_img_data.PatientSex

        # Display the DICOM image using 'imshow' on the appropriate subplot and set axis off.
        ax[i // 3, i % 3].imshow(data_row_img_data.pixel_array, cmap=plt.cm.bone)
        ax[i // 3, i % 3].axis('off')

        # Set the title of the subplot with patient information and other details.
        ax[i // 3, i % 3].set_title('ID: {}\nModality: {} Age: {} Sex: {} Target: {}\nClass: {}\nWindow: {}:{}:{}:{}'.format(
                data_row['patientId'],
                modality, age, sex, data_row['Target'], data_row['class'],
                data_row['x'], data_row['y'], data_row['width'], data_row['height']))

    # Display the grid of subplots containing the DICOM images and their information.
    plt.show()
```

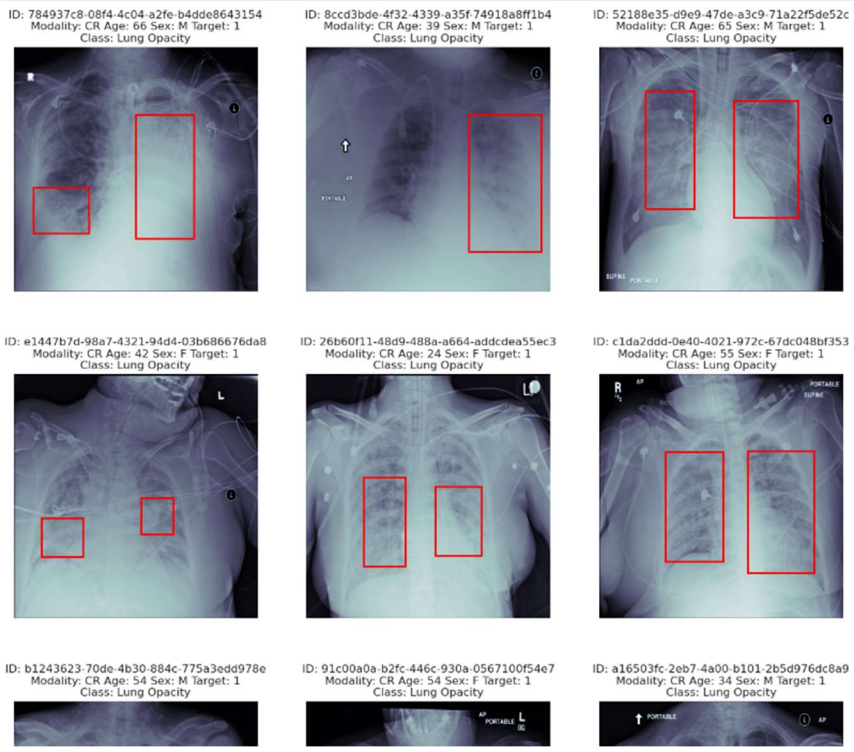
```
# Display a sample of 9 DICOM images from the 'traindf' DataFrame where 'Target' is equal to 1.
show_dicom_images(traindf[traindf['Target'] == 1].sample(9))
```

‘show_dicom_images_with_boxes’ function visualizes DICOM images along with bounding box annotations

```
# Define a function 'show_dicom_images_with_boxes' that takes 'data' as input (DataFrame containing DICOM information).
def show_dicom_images_with_boxes(data):
    # Convert the DataFrame 'data' to a list of dictionaries and store it in 'img_data'.
    img_data = list(data.T.to_dict().values())

    # Create a 3x3 grid of subplots using matplotlib with a figure size of 16x18.
    f, ax = plt.subplots(3, 3, figsize=(16, 18))

    # Loop through each item in 'img_data' and extract DICOM information.
    for i, data_row in enumerate(img_data):
        # Create a file path to the DICOM file corresponding to the patient ID.
        dcm_path = trainImagesPath1 / data_row['patientId']
        dcm_path = dcm_path.with_suffix(".dcm")

        # Read the DICOM file using 'pydicom.read_file()' and store the DICOM data in 'data_row_img_data'.
        data_row_img_data = pydicom.read_file(dcm_path)

        # Extract relevant metadata from the DICOM data.
        modality = data_row_img_data.Modality
        age = data_row_img_data.PatientAge
        sex = data_row_img_data.PatientSex

        # Display the DICOM image using 'imshow' on the appropriate subplot and set axis off.
        ax[i // 3, i % 3].imshow(data_row_img_data.pixel_array, cmap=plt.cm.bone)
        ax[i // 3, i % 3].axis('off')

        # Set the title of the subplot with patient information and class details.
        ax[i // 3, i % 3].set_title('ID: {}\nModality: {} Age: {} Sex: {} Target: {}\nClass: {}'.format(
                data_row['patientId'],modality, age, sex, data_row['Target'], data_row['class']))

        # Filter the DataFrame 'traindf' to extract rows with the same patient ID as the current data_row.
        rows = traindf[traindf['patientId'] == data_row['patientId']]

        # Convert the filtered DataFrame 'rows' to a list of dictionaries and store it in 'box_data'.
        box_data = list(rows.T.to_dict().values())

        # Loop through each item in 'box_data' and draw a red rectangle (box) on the corresponding subplot.
        for j, row in enumerate(box_data):
            ax[i // 3, i % 3].add_patch(Rectangle(xy=(row['x'], row['y']),
                                        width=row['width'], height=row['height'],
                                        linewidth=2, edgecolor='r', facecolor='none'))

    # Display the grid of subplots containing the DICOM images with bounding boxes.
    plt.show()
```

By calling the function, the output displays dicome images associated with metadata and bounding boxes.

```
# Display a sample of 9 DICOM images with bounding boxes from the 'traindf' DataFrame where 'Target' is equal to 1.
show_dicom_images_with_boxes(traindf[traindf['Target'] == 1].sample(9))
```



We have created a function ‘process_dicom_data’ which processes DICOM metadata from a given directory and populates the relevant columns in the DataFrame

```
# List of DICOM metadata variables to process
vars = ['Modality', 'PatientAge', 'PatientSex', 'BodyPartExamined', 'ViewPosition',
        'ConversionType', 'Rows', 'Columns', 'PixelSpacing']

def process_dicom_data(data_df, imagesPath):
    # Initialize DICOM metadata columns with None
    for var in vars:
        data_df[var] = None

    # Get a list of image names in the specified directory
    image_names = os.listdir(imagesPath)

    # Loop through image names and process DICOM metadata
    for i, img_name in tqdm_notebook(enumerate(image_names)):
        # Construct the path to the DICOM file using image name
        dcm_path = imagesPath / img_name
        dcm_path = dcm_path.with_suffix(".dcm")

        # Read DICOM data from the file
        data_row_img_data = pydicom.read_file(dcm_path)

        # Identify rows corresponding to the patient ID in data_df
        idx = (data_df['patientId'] == data_row_img_data.PatientID)

        # Populate DICOM metadata values into data_df
        data_df.loc[idx, 'Modality'] = data_row_img_data.Modality
        data_df.loc[idx, 'PatientAge'] = pd.to_numeric(data_row_img_data.PatientAge)
        data_df.loc[idx, 'PatientSex'] = data_row_img_data.PatientSex
        data_df.loc[idx, 'BodyPartExamined'] = data_row_img_data.BodyPartExamined
        data_df.loc[idx, 'ViewPosition'] = data_row_img_data.ViewPosition
        data_df.loc[idx, 'ConversionType'] = data_row_img_data.ConversionType
        data_df.loc[idx, 'Rows'] = data_row_img_data.Rows
        data_df.loc[idx, 'Columns'] = data_row_img_data.Columns
        data_df.loc[idx, 'PixelSpacing'] = str.format("{:4.3f}", data_row_img_data.PixelSpacing[0])
```

We called this function to process DICOM metadata from ‘trainImagesPath1’. The ‘traindf’ dataset has 26684 Records

```
# Process DICOM metadata for the DataFrame 'traindf'.
process_dicom_data(traindf, trainImagesPath1)
```

```
26684/? [05:18<00:00, 82.38it/s]
```

We call the same function for ‘testImagesPath1’ path for test data

```
# Read the CSV file named 'sampleSubPath' into a pandas DataFrame and store it in the variable 'testdf'.
# The CSV file likely contains the submission data for a machine learning model, which typically includes
# the predictions or probabilities for each test sample in a competition or evaluation task.
# Note that 'sampleSubPath' should be the path to the CSV file containing the submission data.
testdf = pd.read_csv(sampleSubPath)
```

```
# Drop the 'PredictionString' column from the DataFrame 'testdf' using the 'drop' method with axis=1.
# The 'PredictionString' column likely contains the predictions or detection information made by the model,
# which is no longer needed for further processing.
testdf = testdf.drop('PredictionString', 1)

# Call the function 'process_dicom_data' with 'testdf' and 'testImagesPath1' as arguments.
# The function 'process_dicom_data' processes the DICOM data in 'testdf' by extracting relevant information
# from the DICOM files located in the directory specified by 'testImagesPath1'.
# It populates specific columns in 'testdf' with metadata information, such as 'Modality', 'PatientAge',
# 'PatientSex', 'BodyPartExamined', 'ViewPosition', 'ConversionType', 'Rows', 'Columns', and 'PixelSpacing'.
process_dicom_data(testdf, testImagesPath1)
```

```
3000/? [00:23<00:00, 203.37it/s]
```

Identify top 5 rows of 'testdf' dataframe

```
testdf.head() #Display top 5 rows of Test dataframe
```

	patientId	Modality	PatientAge	PatientSex	BodyPartExamined	ViewPosition	ConversionType	Rows	Columns	PixelSpacing
0	0000a175-0e68-4ca4-b1af-167204a7e0bc	CR	46	F	CHEST	PA	WSD	1024	1024	0.194
1	0005d3cc-3c3f-40b9-93c3-46231c3eb813	CR	22	F	CHEST	PA	WSD	1024	1024	0.143
2	000686d7-f4fc-448d-97a0-44fa9c5d3aa6	CR	64	M	CHEST	PA	WSD	1024	1024	0.143
3	000e3a7d-c0ca-4349-bb26-5af2d8993c3d	CR	75	F	CHEST	PA	WSD	1024	1024	0.143
4	00100a24-854d-423d-a092-edcf6179e061	CR	66	F	CHEST	AP	WSD	1024	1024	0.139

```
# Access the 'Modality' column in the DataFrame 'traindf' and call the 'unique()' method on it.
# The 'unique()' method returns an array containing all the unique values present in the 'Modality' column.
# This operation provides insights into the different modalities used in the DICOM data.
traindf.Modality.unique()
```

```
array(['CR'], dtype=object)
```

```
# Access the 'Modality' column in the DataFrame 'testdf' and call the 'unique()' method on it.
# The 'unique()' method returns an array containing all the unique values present in the 'Modality' column.
# This operation provides insights into the different modalities used in the DICOM data.
testdf.Modality.unique()
```

```
array(['CR'], dtype=object)
```

The meaning of this modality is CR - Computer Radiography

Check Patient Age value counts:

```
# Access the 'PatientAge' column in the DataFrame 'traindf' and call the 'value_counts()' method on it.
# The 'value_counts()' method returns a Series containing counts of unique values in the 'PatientAge' column.
# It provides a distribution of the number of occurrences for each unique patient age in the dataset.
traindf['PatientAge'].value_counts()
```

```
58     955
56     869
52     791
55     767
54     717
      ...
148      1
151      1
153      1
150      1
155      1
Name: PatientAge, Length: 97, dtype: int64
```

```
traindf['PatientAge'].max()
```

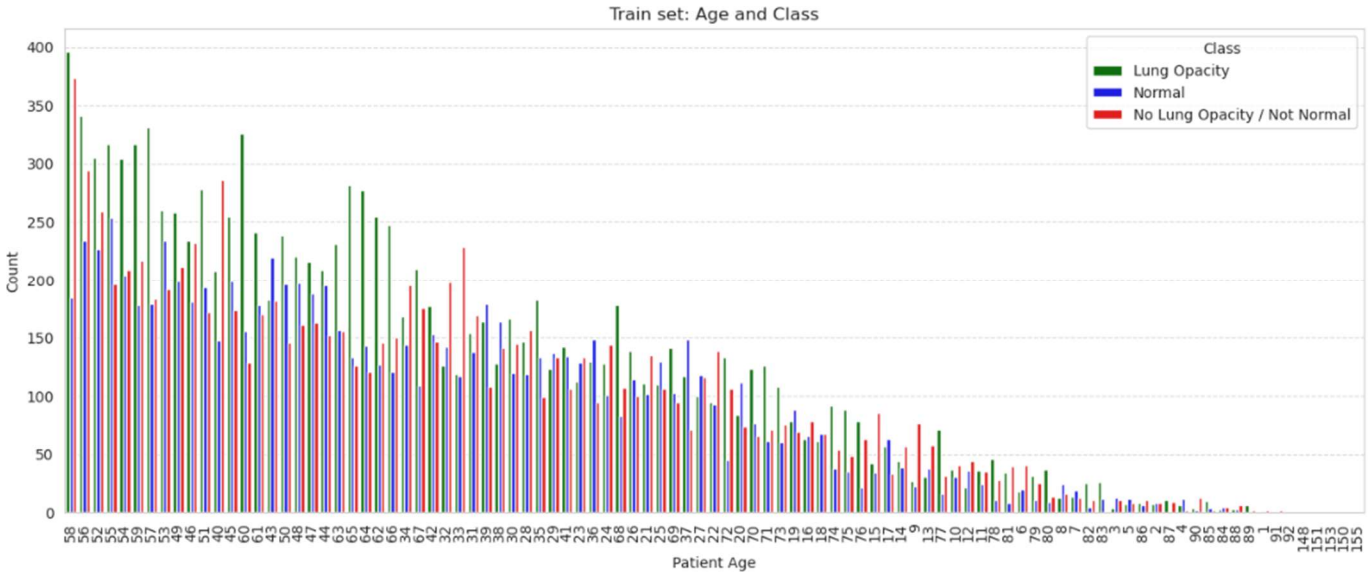
```
155
```

155 could be a typo error during data collection

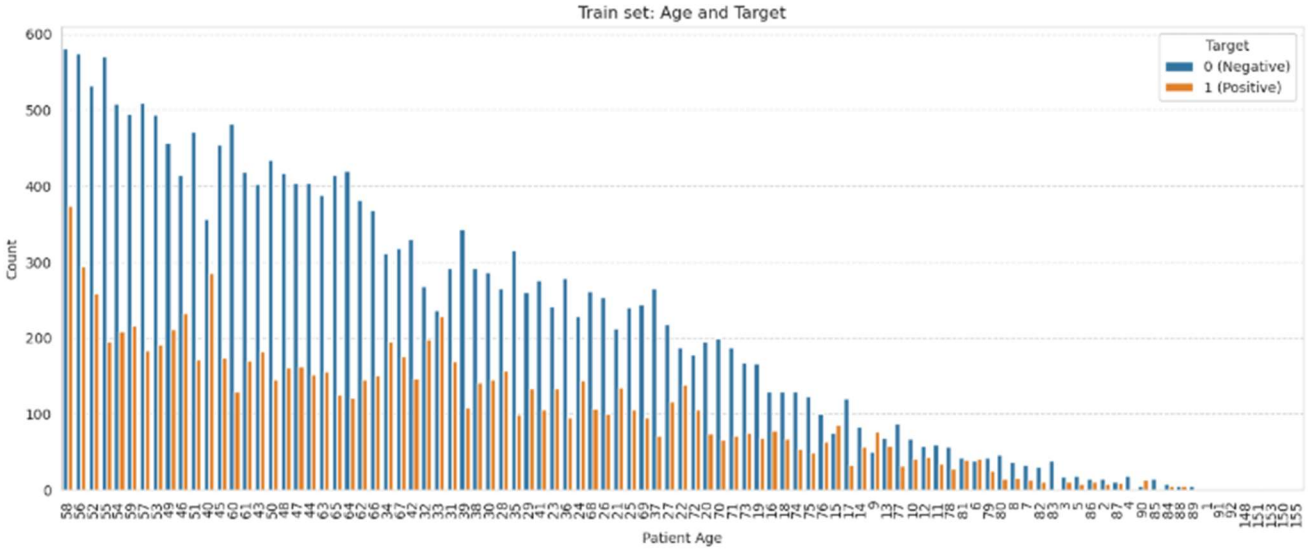
```
traindf['PatientAge'].min()
```

```
1
```

```
# Create a figure and a single subplot using 'plt.subplots' function.
# 'fig' will store the figure object, and 'ax' will store the axes object.
# Set 'nrows=1' to create a single row for subplots, and 'figsize=(16, 6)' to adjust the figure size.
fig, ax = plt.subplots(nrows=1, figsize=(16, 6))

# Create a count plot using Seaborn's 'sns.countplot' function.
# Set 'ax=ax' to specify the axes object to draw the plot on.
# Use 'x' for the 'PatientAge' column on the x-axis and 'hue' for the 'class' column to distinguish classes.
# 'data=traindf' specifies the DataFrame containing the data to be plotted.
# 'order=traindf['PatientAge'].value_counts().index' ensures the bars are ordered based on the count of ages.
# 'palette' argument is used to provide custom colors for each class.
sns.countplot(ax=ax, x='PatientAge', hue='class', data=traindf, order=traindf['PatientAge'].value_counts().index,
              palette={"Lung Opacity": "red", "Normal": "blue", "No Lung Opacity / Not Normal": "green"})

# Set the title for the plot.
plt.title("Train set: Age and Class")

# Rotate the x-axis labels by 90 degrees to improve readability.
plt.xticks(rotation=90)

# Set labels for x and y-axis.
plt.xlabel("Patient Age")
plt.ylabel("Count")

# Display a legend on the plot.
plt.legend(title='Class', loc='upper right', labels=['Lung Opacity', 'Normal', 'No Lung Opacity / Not Normal'])

# Add grid lines to the plot for better readability.
plt.grid(axis='y', linestyle='--', alpha=0.7)

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



```
# Create a figure and a single subplot using 'plt.subplots' function.
# 'fig' will store the figure object, and 'ax' will store the axes object.
# Set 'nrows=1' to create a single row for subplots, and 'figsize=(16, 6)' to adjust the figure size.
fig, ax = plt.subplots(nrows=1, figsize=(16, 6))

# Create a count plot using Seaborn's 'sns.countplot' function.
# Set 'ax=ax' to specify the axes object to draw the plot on.
# Use 'x' for the 'PatientAge' column on the x-axis and 'hue' for the 'Target' column to distinguish the target values.
# 'data=traindf' specifies the DataFrame containing the data to be plotted.
# 'order=traindf['PatientAge'].value_counts().index' ensures the bars are ordered based on the count of ages.
sns.countplot(ax=ax, x='PatientAge', hue='Target', data=traindf, order=traindf['PatientAge'].value_counts().index)

# Set the title for the plot.
plt.title("Train set: Age and Target")

# Rotate the x-axis labels by 90 degrees to improve readability.
plt.xticks(rotation=90)

# Set labels for x and y-axis.
plt.xlabel("Patient Age")
plt.ylabel("Count")

# Display a legend on the plot.
plt.legend(title='Target', loc='upper right', labels=['0 (Negative)', '1 (Positive)'])

# Add grid lines to the plot for better readability.
plt.grid(axis='y', linestyle='--', alpha=0.7)

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



Most of the data has been captured for age group between 40 to 50. There is an outlier with age 151. There are very few data points for age group between 1 to 5 and 80 to 90.

Let us check Patient Sex feature:

```
traindf['PatientSex'].value_counts()
```

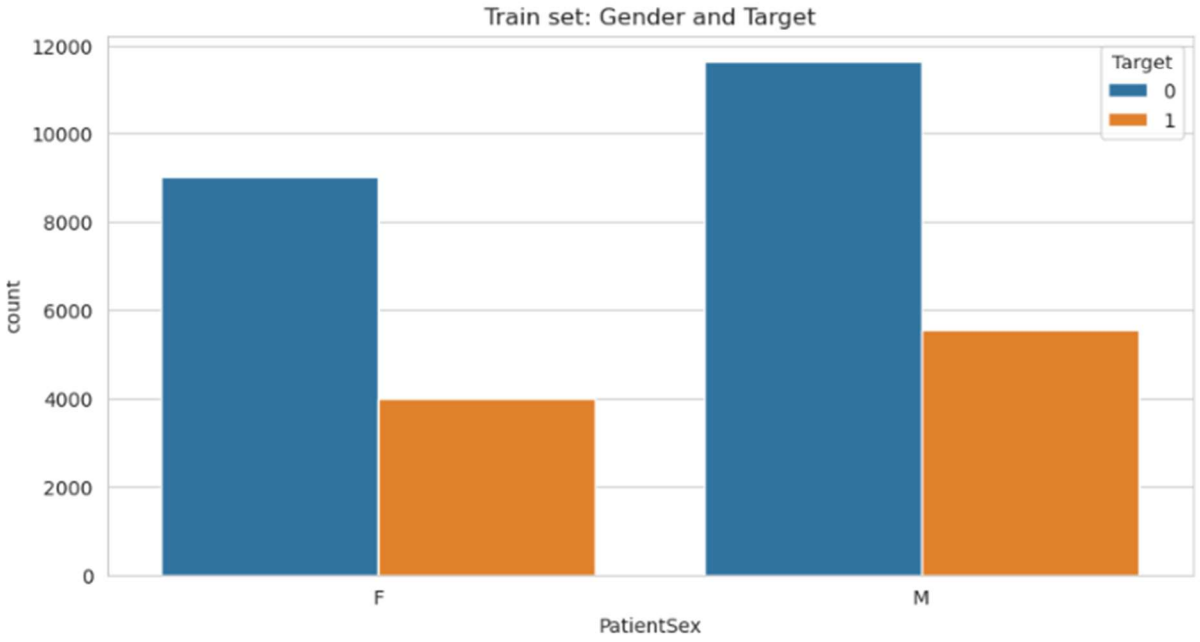
```
M    17216
F    13011
Name: PatientSex, dtype: int64
```

```
testdf['PatientSex'].value_counts()
```

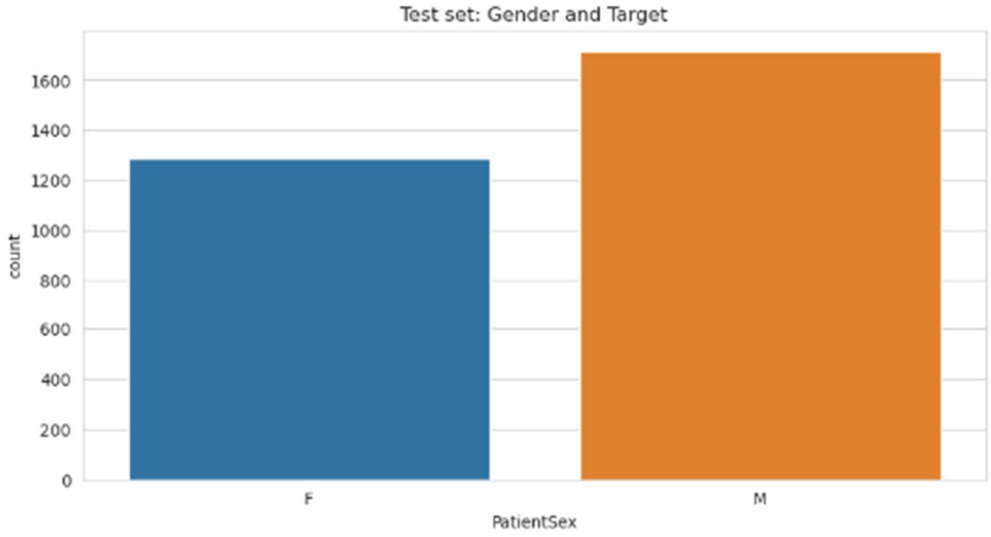
```
M    1714
F    1286
Name: PatientSex, dtype: int64
```

Most of the data points for Male gender both in traing and testing set .

```
# The code will create a single subplot figure with 1 row and 1 column, using 'plt.subplots()'.
# The subplot figure will have a size of 10 inches in width and 5 inches in height.
fig, (ax) = plt.subplots(nrows=1, figsize=(10, 5))

# The 'sns.countplot()' function is used to create a countplot on the subplot.
# The 'x' parameter is set to 'PatientSex', which represents the column in the 'traindf' DataFrame that contains gender information.
# The 'hue' parameter is set to 'Target', which represents the column that contains the target variable (0 or 1) for each patient.
# The 'data' parameter is set to 'traindf', indicating that the data for the plot will be taken from the 'traindf' DataFrame.
sns.countplot(ax=ax, x='PatientSex', hue='Target', data=traindf)

# The 'plt.title()' function is used to set the title of the plot to "Train set: Gender and Target".
plt.title("Train set: Gender and Target")

# Finally, 'plt.show()' is called to display the plot.
plt.show()
```



```
# The code will create a single subplot figure with 1 row and 1 column, using 'plt.subplots()'.
# The subplot figure will have a size of 10 inches in width and 5 inches in height.
fig, (ax) = plt.subplots(nrows=1, figsize=(10, 5))

# The 'sns.countplot()' function is used to create a countplot on the subplot.
# The 'x' parameter is set to 'PatientSex', which represents the column in the 'testdf' DataFrame that contains gender information.
# The 'data' parameter is set to 'testdf', indicating that the data for the plot will be taken from the 'testdf' DataFrame.
sns.countplot(ax=ax, x='PatientSex', data=testdf)

# The 'plt.title()' function is used to set the title of the plot to "Test set: Gender and Target".
plt.title("Test set: Gender and Target")

# Finally, 'plt.show()' is called to display the plot.
plt.show()
```



Let us check Body Part and Conversion Type features:

Let us examine BodyPartExamined feature

```
traindf['BodyPartExamined'].value_counts()
```

```
CHEST    30227
Name: BodyPartExamined, dtype: int64
```

```
testdf['BodyPartExamined'].value_counts()
```

```
CHEST    3000
Name: BodyPartExamined, dtype: int64
```

Unique values found for this column

Let us check ConversionType feature

```
traindf['ConversionType'].value_counts()
```

```
WSD    30227
Name: ConversionType, dtype: int64
```

```
testdf['ConversionType'].value_counts()
```

```
WSD    3000
Name: ConversionType, dtype: int64
```

Unique values found for this column

Let us check Rows and Columns features:

Let us review Rows feature

```
traindf['Rows'].value_counts()
```

```
1024    30227
Name: Rows, dtype: int64
```

Let us review Columns feature

```
traindf['Columns'].value_counts()
```

```
1024    30227
Name: Columns, dtype: int64
```

```
testdf['Rows'].value_counts(),testdf['Columns'].value_counts()
```

```
(1024    3000
 Name: Rows, dtype: int64,
 1024    3000
 Name: Columns, dtype: int64)
```

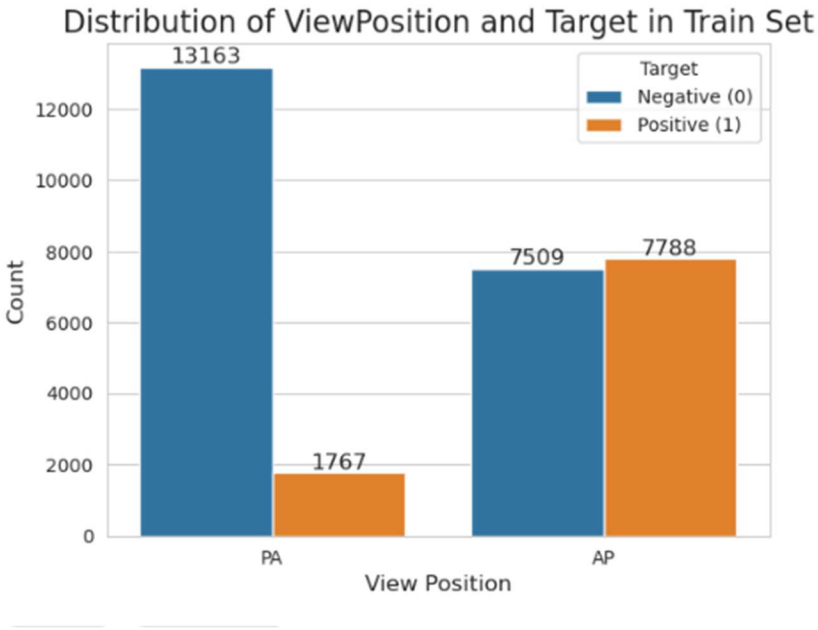
Unique values found.

Let us check ViewPosition feature:

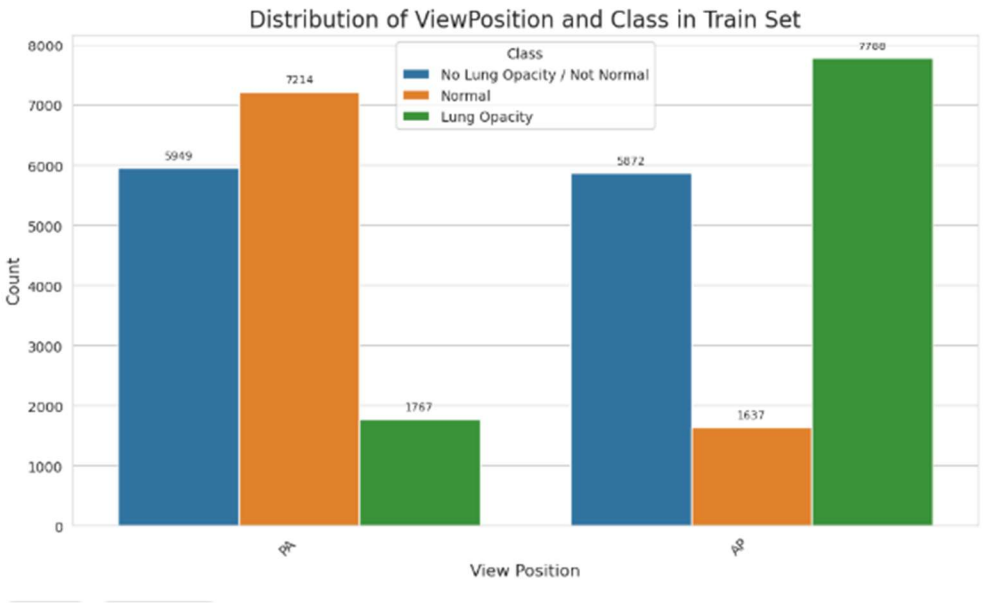
```
traindf['ViewPosition'].value_counts()
```

```
AP    15297
PA    14930
Name: ViewPosition, dtype: int64
```

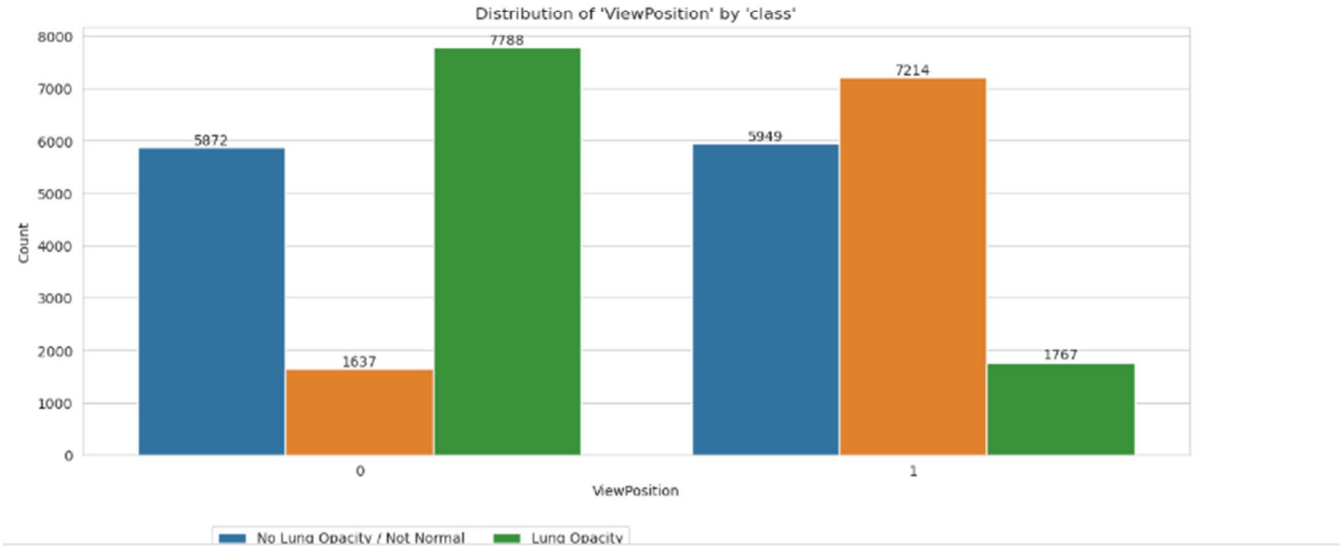
```
# Create a countplot to visualize the distribution of 'ViewPosition' column in the 'traindf' DataFrame.
# The countplot will be color-coded by the 'Target' column, representing the target label (0 or 1) in the dataset.
# The 'x' parameter is set to 'ViewPosition', indicating that the values of the 'ViewPosition' column will be displayed on the x-axis.
# The 'hue' parameter is set to 'Target', indicating that the color of each bar will be determined by the corresponding 'Target' value.
# The 'data' parameter is set to 'traindf', specifying that the data for the plot will be taken from the 'traindf' DataFrame.
ax = sns.countplot(x='ViewPosition', hue='Target', data=traindf)

# Customize the appearance of the plot
plt.title("Distribution of ViewPosition and Target in Train Set", fontsize=16)
plt.xlabel("View Position", fontsize=12)
plt.ylabel("Count", fontsize=12)

# Add annotations above each bar to display the exact count for each category and target class.
# The code iterates through each bar and its associated data to add text annotations above the bars.
for p in ax.patches:
    height = p.get_height()
    ax.annotate(f'{int(height)}', (p.get_x() + p.get_width() / 2, height), ha='center', va='bottom', fontsize=12)

# Add a legend to describe the color coding for the 'Target' classes.
plt.legend(title="Target", labels=['Negative (0)', 'Positive (1)'])

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



```
# The 'data' parameter is set to 'traindf', specifying that the data for the plot will be taken from the 'traindf' DataFrame.
fig, ax = plt.subplots(figsize=(10, 6))
sns.countplot(x='ViewPosition', hue='class', data=traindf, ax=ax)

# Customize the appearance of the plot
plt.title("Distribution of ViewPosition and Class in Train Set", fontsize=16)
plt.xlabel("View Position", fontsize=12)
plt.ylabel("Count", fontsize=12)

# Rotate the x-axis labels to avoid overlapping
plt.xticks(rotation=45, ha='right')

# Add annotations above each bar to display the exact count for each category and class.
# The code iterates through each bar and its associated data to add text annotations above the bars.
for p in ax.patches:
    height = p.get_height()
    ax.annotate(f'{int(height)}', (p.get_x() + p.get_width() / 2, height), ha='center', va='bottom', fontsize=8, xytext=(0, 5), textcoords='offset points')

# Add a legend to describe the color coding for the 'class' labels.
plt.legend(title="Class", fontsize=10)

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



```
# Increase the figure size for better visualization
plt.figure(figsize=(12, 6))

# Create the countplot with 'ViewPosition' on the x-axis and 'class' on the hue for color differentiation.
sns.countplot(x='ViewPosition', hue='class', data=traindf)

# Set a title for the plot
plt.title("Distribution of 'ViewPosition' by 'class'")

# Move the legend to the middle to avoid overlapping with the bars
plt.legend(loc='upper right', bbox_to_anchor=(0.5, -0.15), ncol=2)

# Add annotations for the counts above the bars
for p in plt.gca().patches:
    height = p.get_height()
    plt.gca().annotate(f'{int(height)}', (p.get_x() + p.get_width() / 2, height),
                       ha='center', va='bottom', fontsize=10)

# Set labels for x and y axes
plt.xlabel('ViewPosition')
plt.ylabel('Count')

# Adjust the layout to avoid overlapping
plt.tight_layout()

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



AP-AnteriorPosterior=0 PA-PostereiorAterior=1

```
testdf['ViewPosition'].value_counts()
```

```
PA    1618
AP    1382
Name: ViewPosition, dtype: int64
```

```
# Create a label_encoder object, which is used to encode categorical labels as integer values.
label_encoder = preprocessing.LabelEncoder()

# Use label_encoder to transform the 'ViewPosition' column in the test dataframe from categorical labels to numerical values.
# This step is necessary for many machine learning algorithms that cannot handle categorical data directly.
testdf['ViewPosition'] = label_encoder.fit_transform(testdf['ViewPosition'])

# Print the classes that have been encoded to understand the mapping of numerical values to original categories.
print(label_encoder.classes_)

# Display the unique numerical values of the 'ViewPosition' column after label encoding.
testdf['ViewPosition'].unique()
```

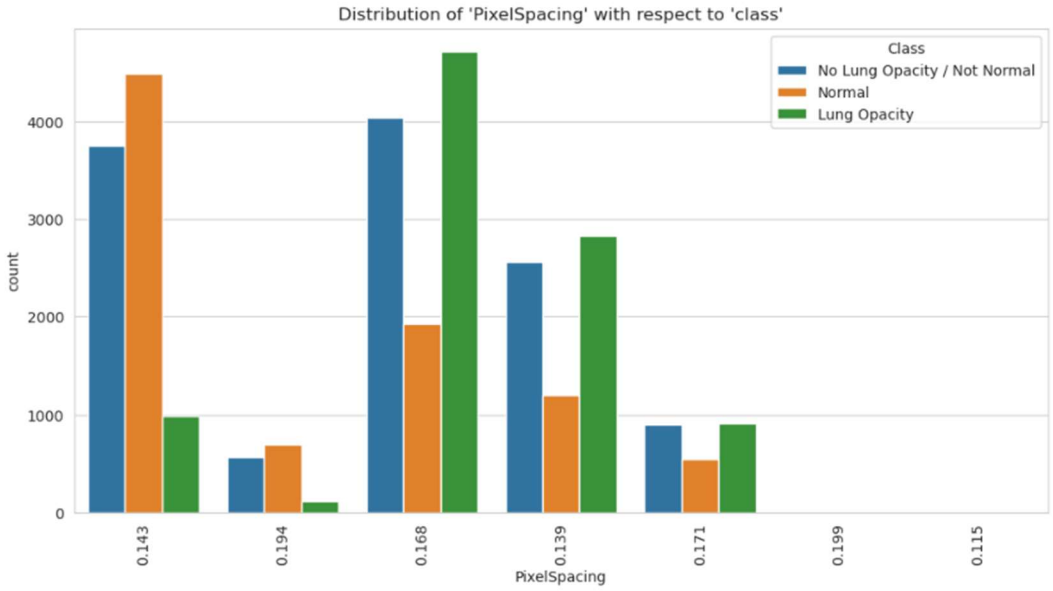
```
['AP' 'PA']
array([1, 0])
```

Let us explore PixelSpacing feature:

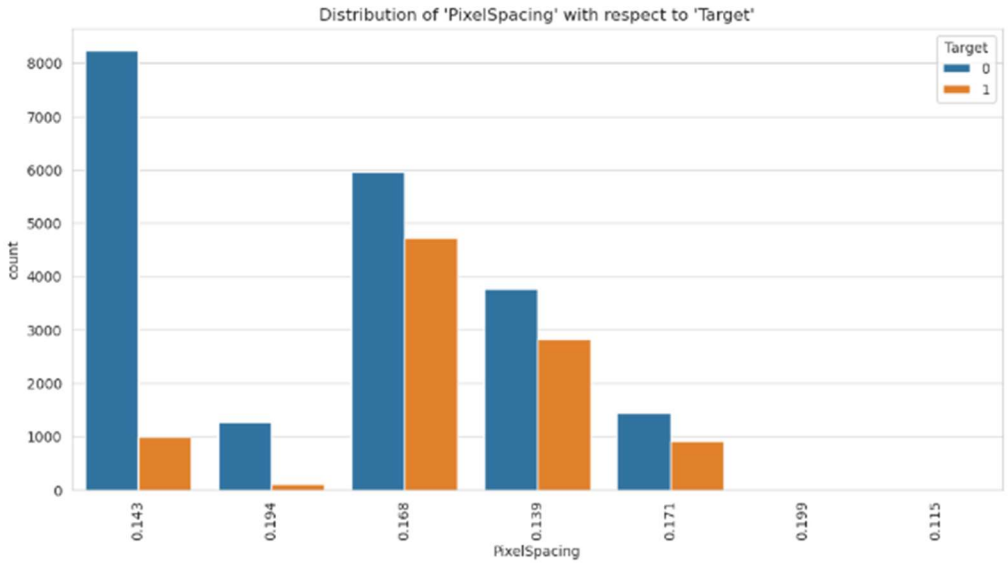
```
traindf['PixelSpacing'].value_counts()
```

```
0.168    10677
0.143     9221
0.139     6585
0.171     2351
0.194     1380
0.115        8
0.199        5
Name: PixelSpacing, dtype: int64
```

```
# Create a count plot for 'PixelSpacing' column with 'class' as hue using Seaborn's 'countplot' function.
# Set the figure size to (12, 6) to make the plot larger and avoid overlapping of labels.
fig, ax = plt.subplots(figsize=(12, 6))

# Use Seaborn's 'countplot' function to create the count plot.
sns.countplot(x='PixelSpacing', hue='class', data=traindf, ax=ax)

# Set the title for the plot.
plt.title("Distribution of 'PixelSpacing' with respect to 'class'")

# Rotate the x-axis labels to prevent overlapping.
plt.xticks(rotation=90)

# Add a legend to the plot to show the mapping of hue colors to 'class' labels.
plt.legend(title='Class', loc='upper right')

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



```
# Create a count plot for 'PixelSpacing' column with 'Target' as hue using Seaborn's 'countplot' function.
# Set the figure size to (12, 6) to make the plot larger and avoid overlapping of labels.
fig, ax = plt.subplots(figsize=(12, 6))

# Use Seaborn's 'countplot' function to create the count plot.
sns.countplot(x='PixelSpacing', hue='Target', data=traindf, ax=ax)

# Set the title for the plot.
plt.title("Distribution of 'PixelSpacing' with respect to 'Target'")

# Rotate the x-axis labels to prevent overlapping.
plt.xticks(rotation=90)

# Add a legend to the plot to show the mapping of hue colors to 'Target' labels.
plt.legend(title='Target', loc='upper right')

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



0.115 Pixel spacing is for age between 2-11 years. and 0.199 is from 20-35 age group. we have very less data points for this.

```
traindf[traindf['PixelSpacing']=='0.199']
```

	patientId	x	y	width	height	Target	class	Modality	PatientAge	PatientSex	BodyPartExamined	ViewPosition	ConversionType	Rows	Columns	PixelSpacing
673	09714ab6-5dce-4ded-94cc-f79a47b4b171	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal	CR	25	M	CHEST	0	WSD	1024	1024	0.199
16521	9fae656b-cf66-4c4b-a5ed-1962ebcafb7d	651.0	346.0	275.0	210.0	1	Lung Opacity	CR	38	M	CHEST	0	WSD	1024	1024	0.199
16522	9fae656b-cf66-4c4b-a5ed-1962ebcafb7d	262.0	354.0	240.0	156.0	1	Lung Opacity	CR	38	M	CHEST	0	WSD	1024	1024	0.199
20838	bda68f49-6eb2-4afa-86bf-89a3d4378100	634.0	436.0	217.0	167.0	1	Lung Opacity	CR	23	F	CHEST	0	WSD	1024	1024	0.199
20839	bda68f49-6eb2-4afa-86bf-89a3d4378100	389.0	395.0	142.0	181.0	1	Lung Opacity	CR	23	F	CHEST	0	WSD	1024	1024	0.199

```
traindf[traindf['PixelSpacing']=='0.115']
```

	patientId	x	y	width	height	Target	class	Modality	PatientAge	PatientSex	BodyPartExamined	ViewPosition	ConversionType	Rows	Columns	PixelSpacing
9834	6a8a2a9b-b22e-471a-8b90-3c5f6496cee6	NaN	NaN	NaN	NaN	0	Normal	CR	6	M	CHEST	1	WSD	1024	1024	0.115
15480	9746c15f-94d1-4528-aff0-8ef02ccccef0	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal	CR	2	F	CHEST	0	WSD	1024	1024	0.115
17741	a8f16478-b36a-4bc9-ac8a-75477d8ea168	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal	CR	5	F	CHEST	1	WSD	1024	1024	0.115
21089	bf494059-4fe2-4f8f-a503-c340105e56ce	NaN	NaN	NaN	NaN	0	Normal	CR	4	M	CHEST	1	WSD	1024	1024	0.115
22215	c908770d-5bb8-479e-9112-ff9c47d07277	NaN	NaN	NaN	NaN	0	No Lung Opacity / Not Normal	CR	11	F	CHEST	0	WSD	1024	1024	0.115

Dropping features in ‘traindf’ & ‘trestdf’ dataframes which are not necessary for out futher analysis

```
# Drop specified columns from the 'traindf' DataFrame.
# The columns to be dropped are 'Modality', 'ConversionType', 'BodyPartExamined', 'Rows', and 'Columns'.
# The DataFrame will be updated in place, and the specified columns will be removed.

traindf = traindf.drop(['Modality', 'ConversionType', 'BodyPartExamined', 'Rows', 'Columns'], axis=1)
```

```
# Drop specified columns from the 'testdf' DataFrame.
# The columns to be dropped are 'Modality', 'ConversionType', 'BodyPartExamined', 'Rows', and 'Columns'.
# The DataFrame will be updated in place, and the specified columns will be removed.

testdf = testdf.drop(['Modality', 'ConversionType', 'BodyPartExamined', 'Rows', 'Columns'], axis=1)
```

```
traindf.info()
```

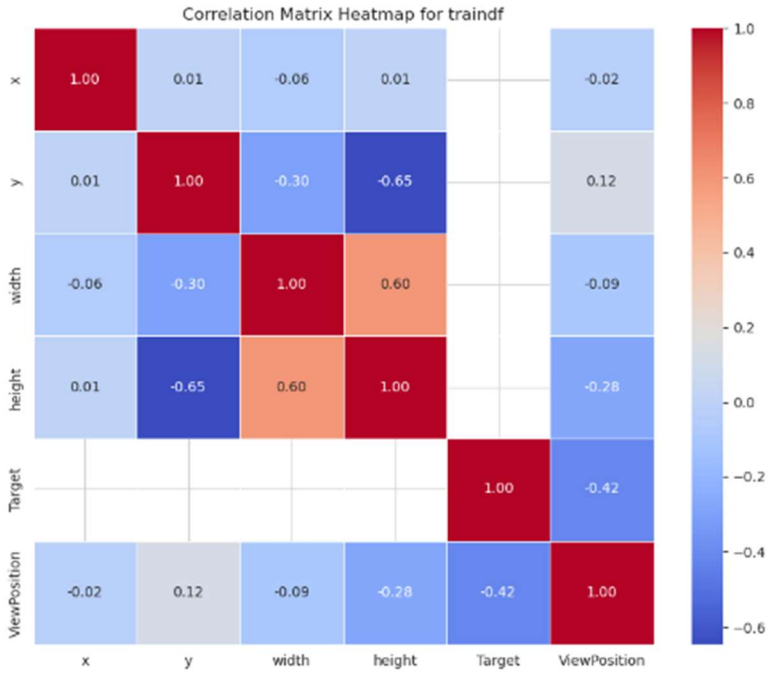
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30227 entries, 0 to 30226
Data columns (total 11 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   patientId     30227 non-null  object
 1   x             9555 non-null   float64
 2   y             9555 non-null   float64
 3   width         9555 non-null   float64
 4   height        9555 non-null   float64
 5   Target        30227 non-null  int64
 6   class         30227 non-null  object
 7   PatientAge    30227 non-null  object
 8   PatientSex    30227 non-null  object
 9   ViewPosition  30227 non-null  int64
 10  PixelSpacing  30227 non-null  object
dtypes: float64(4), int64(2), object(5)
memory usage: 2.8+ MB
```

```
traindf[traindf['PatientAge'].isna()]
```

patientId	x	y	width	height	Target	class	PatientAge	PatientSex	ViewPosition	PixelSpacing

Checking Correlation matrix to understand the correlation between the features in the ‘traindf’ dataset.

```
# Calculate the correlation matrix for the 'traindf' DataFrame.
# The correlation matrix shows the pairwise correlation between numerical columns.
correlation_matrix = traindf.corr()
# Set up the figure and axis for the heatmap plot.
fig, ax = plt.subplots(figsize=(10, 8))
# Create the heatmap using the correlation matrix.
# The 'cmap' parameter sets the color map for the heatmap (e.g., 'coolwarm', 'viridis', 'RdBu_r', etc.).
# The 'annot' parameter is set to True to display the correlation coefficients on the heatmap.
# The 'fmt' parameter is used to specify the number format for the annotations (e.g., '.2f' for two decimal places).
# The 'linewidths' parameter controls the width of the lines separating each cell.
# The 'cbar' parameter is set to True to display the color bar on the side of the heatmap.
# The 'cbar_kws' parameter is used to customize the color bar appearance (e.g., orientation, label, etc.).
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=True, fmt='.2f', linewidths=0.5, cbar=True, cbar_kws={'orientation': 'vertical'})

# Set the title for the heatmap plot.
plt.title('Correlation Matrix Heatmap for traindf')
# Display the heatmap.
plt.show()
```



+ Code + Markdown

y & height has highest negative correlation width & height has highest positive correlation

6: Design, train and test basic CNN models for classification

Split data into train and validation

```
# Set the path to the train images directory and the directory where the CSV file will be saved
TRAIN_PATH = trainImagesPath1
SAVE_PATH = "/kaggle/working/"

# Create an empty list to store the filenames of the images
filenames = []

# Read the directory and save the filenames to a list
filenames = os.listdir(TRAIN_PATH)

# Save the filenames to a CSV file
pd.DataFrame(filenames).to_csv(SAVE_PATH + 'train_path_listdir.csv', index=False)

# Specify the percentage of data used for training (e.g., 100% of the data)
percentage_data_used = 100

# Calculate the number of files to use based on the specified percentage
file_count = int(len(filenames) * percentage_data_used / 100)

# Print the total number of files available
print("Total files available:", file_count)

# Shuffle the list of filenames randomly to introduce randomness in the data
random.shuffle(filenames)

# Calculate the number of validation samples based on the specified percentage (30% in this case)
n_valid_samples = int(file_count * 0.3)

# Split the filenames into train and validation sets based on the number of validation samples
train_filenames = filenames[n_valid_samples:file_count]
valid_filenames = filenames[:n_valid_samples]

# Print the number of train and validation samples
print('n train samples:', len(train_filenames))
print('n valid samples:', len(valid_filenames))

# Calculate the number of train samples
n_train_samples = len(filenames) - n_valid_samples

# Set the image dimension to be used (e.g., 128x128)
image_dimension = 128

# Print the chosen image dimension and the filename of a sample image
print('Image Dimension to use:', image_dimension)
print('sample file:', filenames[0])
```

```
Total files available: 26684
n train samples: 18679
n valid samples: 8005
Image Dimension to use: 128
sample file: c77a7c87-8989-4dcc-8dc4-2c7cc98a5cc5.dcm
```

Create a dictionary of pneumonia locations in one place.

```
# Define the project path where the CSV file is located
PROJECT_PATH = "/kaggle/input/pneumonia-detection"

# Create an empty dictionary to store pneumonia locations for each image
pneumonia_locations = {}

# Load the CSV file that contains pneumonia labels and image locations
with open(os.path.join(PROJECT_PATH, 'stage_2_train_labels.csv'), mode='r') as infile:
    # Create a CSV reader to read the contents of the file
    reader = csv.reader(infile)
    # Skip the header row of the CSV file
    next(reader, None)
    # Loop through each row in the CSV file
    for rows in reader:
        # Retrieve the filename and pneumonia location information from the current row
        filename = rows[0]
        location = rows[1:5]
        pneumonia = rows[5]

        # Check if the current row represents a pneumonia case (indicated by pneumonia=1)
        if pneumonia == '1':
            # Convert the location values from string to float and then to integer
            location = [int(float(i)) for i in location]

            # Save the pneumonia location in the dictionary
            # If the filename is already present in the dictionary, append the location to the existing list
            # If the filename is not present in the dictionary, create a new entry with the location list
            if filename in pneumonia_locations:
                pneumonia_locations[filename].append(location)
```

Creation of Generator class

```
# Define the custom data generator class
class generator(keras.utils.Sequence):
    def __init__(self, folder, filenames, pneumonia_locations=None, batch_size=32, image_size=256, shuffle=True, augment=False, predict=False):
        # Initialize the generator with required parameters
        self.folder = folder
        self.filenames = filenames
        self.pneumonia_locations = pneumonia_locations
        self.batch_size = batch_size
        self.image_size = image_size
        self.shuffle = shuffle
        self.augment = augment
        self.predict = predict
        # Execute this method after every epoch
        self.on_epoch_end()

    def __load__(self, filename):
        # Load DICOM file as a numpy array
        img = pydicom.dcmread(os.path.join(self.folder, filename)).pixel_array
        # Create an empty mask with zeros
        msk = np.zeros(img.shape)
        # Get the filename without extension
        filename = filename.split('.')[0]
        # Check if the image contains pneumonia
        if filename in self.pneumonia_locations:
            # Loop through each pneumonia location and set corresponding pixels to 1
            for location in self.pneumonia_locations[filename]:
                x, y, w, h = location
                msk[y:y+h, x:x+w] = 1
        # Resize both image and mask to the specified image size
        img = resize(img, (self.image_size, self.image_size), mode='reflect')
        msk = resize(msk, (self.image_size, self.image_size), mode='reflect') > 0.5
        # If augmentation is enabled, horizontally flip the images and masks half the time
        if self.augment and random.random() > 0.5:
            img = np.fliplr(img)
            msk = np.fliplr(msk)
        # Add trailing channel dimension to the images and masks
        img = np.expand_dims(img, -1)
        msk = np.expand_dims(msk, -1)
        return img, msk

    def __loadpredict__(self, filename):
        # Check if the file is a DICOM file
        if filename.lower().endswith(".dcm"):
            # Load the DICOM file as a numpy array
            img = pydicom.dcmread(os.path.join(self.folder, filename)).pixel_array
        else:
            # Load the image file using PIL
            img = Image.open(os.path.join(self.folder, filename))
            # Convert the image to a numpy array
            img = np.array(img)

        # Resize the image to the specified image size
        img = resize(img, (self.image_size, self.image_size), mode='reflect')

        # Normalize pixel values to a range between 0 and 1
        img /= 255.

        return img

    def __getitem__(self, index):
        # Select a batch of filenames based on the index
        filenames = self.filenames[index*self.batch_size:(index+1)*self.batch_size]
        # If in prediction mode, return images and filenames
        if self.predict:
            # Load files and create a numpy batch of images
            imgs = [self.__loadpredict__(filename) for filename in filenames]
            imgs = np.array(imgs)
            return imgs, filenames
        # In training mode, return images and masks
        else:
            # Load files and create a numpy batch of images and masks
            items = [self.__load__(filename) for filename in filenames]
            imgs, msks = zip(*items)
            imgs = np.array(imgs)
            msks = np.array(msks)
            return imgs, msks

    def on_epoch_end(self):
        # If shuffle is enabled, shuffle the list of filenames after every epoch
        if self.shuffle:
            random.shuffle(self.filenames)

    def __len__(self):
        # Determine the number of batches based on the predict mode
        if self.predict:
            # Return the total number of batches to predict everything
            return int(np.ceil(len(self.filenames) / self.batch_size))
        else:
            # Return the total number of full batches for training
            return int(len(self.filenames) / self.batch_size)
```

The code below provides components of a U-Net architecture for image segmentation. Here's a breakdown of what each part does:

IoU (Intersection over Union) Loss Function (iou_loss):

This function calculates the IoU loss between the predicted and true masks. It's a measure of the similarity between two binary masks, where higher values indicate better overlap between the masks.

Combining Binary Cross-Entropy Loss and IoU Loss (iou_bce_loss):

This function combines the binary cross-entropy loss and the IoU loss. It's a custom loss function that aims to balance the importance of classifying each pixel and ensuring a good mask overlap.

Mean IoU Metric (mean_iou):

This function calculates the mean IoU metric for evaluating the performance of the segmentation model. It computes the average IoU across all predictions compared to the true masks.

Downsample Layer (create_downsample):

This function creates a downsample layer using Batch Normalization, LeakyReLU activation, and a 1x1 Convolution operation. It's used to reduce the spatial dimensions of the feature maps while increasing the number of channels.

Residual Block (create_resblock):

This function creates a residual block with two Convolution layers. Residual blocks help the network capture complex features while maintaining gradient flow during training.

U-Net Architecture (create_network):

This function defines the U-Net architecture. U-Net is a popular architecture for image segmentation. It consists of a contracting path (encoder) and an expansive path (decoder). The encoder captures context and features, while the decoder recovers spatial information.

The input size is specified. The initial Conv2D layer processes the input. The architecture has multiple levels of depth (controlled by depth) where downsampling is followed by residual blocks.
At each depth level, n_blocks number of residual blocks are applied. The final layer is a Conv2D layer with a sigmoid activation, which produces mask-like outputs. UpSampling2D layers in the decoder part increase the spatial dimensions to match the input size.
This code outlines the construction of a U-Net-based neural network architecture for image segmentation.

```python
# Define the IoU (Intersection over Union) or Jaccard loss function.
# This loss function calculates the similarity between the predicted and true masks.
def iou_loss(y_true, y_pred):
    y_true = tf.cast(y_true, tf.float32)
    y_pred = tf.cast(y_pred, tf.float32)
    y_true = tf.reshape(y_true, [-1])
    y_pred = tf.reshape(y_pred, [-1])
    intersection = tf.reduce_sum(y_true * y_pred)
    score = (intersection + 1.) / (tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) - intersection + 1.)
    return 1 - score
# Combine binary cross-entropy loss and IoU loss to create a new loss function.
def iou_bce_loss(y_true, y_pred):
    return 0.5 * keras.losses.binary_crossentropy(y_true, y_pred) + 0.5 * iou_loss(y_true, y_pred)
# Define the mean IoU (Intersection over Union) as a metric.
# This metric calculates the average IoU for all predictions compared to the true masks.
def mean_iou(y_true, y_pred):
    y_pred = tf.round(y_pred)
    intersect = tf.reduce_sum(y_true * y_pred, axis=[1, 2, 3])
    union = tf.reduce_sum(y_true, axis=[1, 2, 3]) + tf.reduce_sum(y_pred, axis=[1, 2, 3])
    smooth = tf.ones(tf.shape(intersect))
    return tf.reduce_mean((intersect + smooth) / (union - intersect + smooth))
# Create a downsample layer using Batch Normalization, LeakyReLU activation, and 1x1 Convolution.
def create_downsample(channels, inputs):
    x = keras.layers.BatchNormalization(momentum=0.9)(inputs)
    x = keras.layers.LeakyReLU(0)(x)
    x = keras.layers.Conv2D(channels, 1, padding='same', use_bias=False)(x)
    x = keras.layers.MaxPool2D(2)(x)
    return x
# Create a residual block with two Convolution layers.
def create_resblock(channels, inputs):
    x = keras.layers.BatchNormalization(momentum=0.9)(inputs)
    x = keras.layers.LeakyReLU(0)(x)
    x = keras.layers.Conv2D(channels, 3, padding='same', use_bias=False)(x)
    x = keras.layers.BatchNormalization(momentum=0.9)(x)
    x = keras.layers.LeakyReLU(0)(x)
    x = keras.layers.Conv2D(channels, 3, padding='same', use_bias=False)(x)
    return keras.layers.add([x, inputs])
# Create the U-Net architecture-based neural network.
def create_network(input_size, channels, n_blocks=2, depth=4):
    # Input layer
    inputs = keras.Input(shape=(input_size, input_size, 1))
    x = keras.layers.Conv2D(channels, 3, padding='same', use_bias=False)(inputs)
    # Residual blocks
    for d in range(depth):
        channels = channels * 2
        x = create_downsample(channels, x)
        for b in range(n_blocks):
            x = create_resblock(channels, x)
    # Output layer
    x = keras.layers.BatchNormalization(momentum=0.9)(x)
    x = keras.layers.LeakyReLU(0)(x)
    x = keras.layers.Conv2D(1, 1, activation='sigmoid')(x)
    outputs = keras.layers.UpSampling2D(2**depth)(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

Setting Batch Size and Image Size:

The batch size determines how many samples are processed in each training iteration, and the image size is the size of input images that the U-Net model will expect.

Creating U-Net Model:

Using the create_network function defined earlier, U-Net model is created with the specified parameters:

input_size: The size of input images (both width and height) for the model.

channels: The number of channels in the initial convolutional layer of the U-Net.

n_blocks: The number of residual blocks to include in each downsampling stage of the U-Net.
depth: The number of downsampling stages in the U-Net model.

Compiling the Model:

Compiling for training using the compile method with the following:

optimizer: The optimization algorithm used during training. In this case, it's 'adam', which is a popular adaptive learning rate optimization algorithm.
loss: The loss function used during training is iou_bce_loss, which combines the binary cross-entropy loss and the IoU loss. This helps balance segmentation performance.
metrics: During training, the model will calculate and display the accuracy and mean IoU metrics to monitor its performance.
With this code, the U-Net model architecture is setup, defined the training parameters, and compiled the model.

The next step is using this model to train it ‘traindf’ using the Keras framework.

```
# Set the batch size to 128, which determines the number of samples processed in each training iteration.
BATCH_SIZE = 128

# Set the image size to 128x128 pixels, which will be used as the input size for the U-Net model.
IMAGE_SIZE = 128

# Create the U-Net model using the 'create_network' function with the specified parameters:
# input_size: The size of the input images (both width and height) for the U-Net model.
# channels: The number of channels in the initial convolutional layer of the U-Net model.
# n_blocks: The number of residual blocks to include in each downsampling stage of the U-Net.
# depth: The number of downsampling stages in the U-Net model.
model = create_network(input_size=IMAGE_SIZE, channels=32, n_blocks=2, depth=4)

# Compile the U-Net model for training:
# optimizer: 'adam' is used as the optimization algorithm, which is an adaptive learning rate optimization algorithm.
# loss: The loss function used during training is 'iou_bce_loss', which combines the binary cross-entropy loss and
#       the IoU (Intersection over Union) loss to balance the segmentation performance.
# metrics: During training, the model will calculate and display the accuracy and mean IoU metrics to monitor its performance.
```

Defining Cosine Annealing Learning Rate Function (cosine_annealing):

This function implements a learning rate annealing schedule based on the cosine function. The function takes the current epoch number (x) as an argument and returns the updated learning rate based on the cosine annealing formula.

Creating LearningRateScheduler Callback (learning_rate):

Here, LearningRateScheduler callback named learning_rate is created. This callback adjusts the learning rate during training based on the cosine_annealing function. The learning rate will be updated at the end of each epoch.

Creating Data Generators (train_gen and valid_gen):

Creating train and validation data generators using the generator class defined earlier. These generators will provide data batches for training and validation during the training process. You pass in various parameters, including the folder path, file names, pneumonia locations, batch size, image size, and more, to configure the behavior of the generators.

Printing Model Summary:

printing a summary of the U-Net model using the model.summary() function. This summary provides an overview of the model's architecture, layer information, and the number of trainable parameters in each layer.

The code sets up the necessary components for training the U-Net model, including the learning rate annealing schedule, data generators, and model architecture.

```python
# Define a function 'cosine_annealing' that implements a learning rate annealing schedule based on the cosine function.
# The function takes a single argument 'x', which represents the current epoch number during training.
def cosine_annealing(x):
    lr = 0.0001  # The initial learning rate.
    epochs = 2   # The total number of epochs in the training process.
    # Calculate the learning rate using the cosine annealing formula.
    # The learning rate will start from 'lr' and decay in a cosine manner over the given number of 'epochs'.
    return lr * (np.cos(np.pi * x / epochs) + 1.) / 2

# Create a LearningRateScheduler callback, 'learning_rate', that will adjust the learning rate during training.
# The learning rate will be updated at the end of each epoch based on the 'cosine_annealing' function.
learning_rate = tf.keras.callbacks.LearningRateScheduler(cosine_annealing)

# Assuming 'folder' is the path to the directory containing the training images.
folder = trainImagesPath1

# Create train and validation data generators using the 'generator' class.
# The 'generator' class provides data batches for training and validation during the training process.
# 'train_gen' will be used for training, and 'valid_gen' will be used for validation.
# The generators will provide data in batches of size 'BATCH_SIZE' and resize images to 'IMAGE_SIZE'.
# 'pneumonia_locations' is the dictionary containing pneumonia locations in the training data.
# 'shuffle=True' randomizes the order of data samples during training, while 'shuffle=False' keeps the validation data order unchanged.
# 'augment=False' indicates that data augmentation will not be used during training.
train_gen = generator(folder, train_filenames, pneumonia_locations, batch_size=BATCH_SIZE, image_size=IMAGE_SIZE, shuffle=True, augment=False, predict=False)
valid_gen = generator(folder, valid_filenames, pneumonia_locations, batch_size=BATCH_SIZE, image_size=IMAGE_SIZE, shuffle=False, predict=False)

# Print the summary of the U-Net model to get an overview of its architecture and number of parameters.
print(model.summary())
```

```
batch_normalization_18 (BatchN  (None, 8, 8, 512)   2048        ['add_6[0][0]']
ormalization)

leaky_re_lu_18 (LeakyReLU)     (None, 8, 8, 512)    0           ['batch_normalization_18[0][0]']

conv2d_19 (Conv2D)             (None, 8, 8, 512)    2359296     ['leaky_re_lu_18[0][0]']

batch_normalization_19 (BatchN  (None, 8, 8, 512)   2048        ['conv2d_19[0][0]']
ormalization)

leaky_re_lu_19 (LeakyReLU)     (None, 8, 8, 512)    0           ['batch_normalization_19[0][0]']

conv2d_20 (Conv2D)             (None, 8, 8, 512)    2359296     ['leaky_re_lu_19[0][0]']

add_7 (Add)                    (None, 8, 8, 512)    0           ['conv2d_20[0][0]',
                                                                 'add_6[0][0]']

batch_normalization_20 (BatchN  (None, 8, 8, 512)   2048        ['add_7[0][0]']
ormalization)

leaky_re_lu_20 (LeakyReLU)     (None, 8, 8, 512)    0           ['batch_normalization_20[0][0]']

conv2d_21 (Conv2D)             (None, 8, 8, 1)      513         ['leaky_re_lu_20[0][0]']

up_sampling2d (UpSampling2D)   (None, 128, 128, 1)  0           ['conv2d_21[0][0]']

==================================================================================================
Total params: 12,727,969
Trainable params: 12,718,305
Non-trainable params: 9,664
__________________________________________________________________________________________________
None
```

Next step is to proceed with training the model using these components.

Setting the Number of Epochs:

Number of epochs for training is set as 2 (EPOCHS = 2). The training process will iterate over the entire dataset for this number of epochs.

Training the Model:

model.fit_generator() function is used to train the model. This function takes the following arguments:

train_gen: The training data generator provides data batches during training.

validation_data: The validation data generator provides data batches during validation.

callbacks: A list of callbacks to be used during training. Here, learning_rate callback is included that adjusts the learning rate based on the cosine annealing schedule.

epochs: The number of epochs to train the model.

```
EPOCHS=2

history = model.fit_generator(train_gen, validation_data=valid_gen, callbacks=[learning_rate], epochs=EPOCHS)
```

```
Epoch 1/2
145/145 [==============================] - 2518s 17s/step - loss: 0.5222 - accuracy: 0.9343 - mean_iou: 0.6338 - val_loss: 0.4507 - val_accuracy: 0.9637 - val_mean_iou: 0.6920 - lr: 1.0000e-04
Epoch 2/2
145/145 [==============================] - 2443s 17s/step - loss: 0.4437 - accuracy: 0.9673 - mean_iou: 0.7150 - val_loss: 0.4342 - val_accuracy: 0.9679 - val_mean_iou: 0.7326 - lr: 5.0000e-05
```

From the above code, we could not print Confusion Matrix as the shape of y_true & y_pred #are different.

We shall continue to conduct testing on other models and check accuracy of different classes with confusion matrix

MILESTONE 2

Input:

Preprocessed output from Milestone-1

Process:

- Step 1: Fine tune the trained basic CNN models for classification.
- Step 2: Apply Transfer Learning model for classification
- Step 3: Design, train and test RCNN & its hybrids based object detection models to impose the bounding box or mask over the area of interest.
- Step 4: Pickle the model for future prediction
- Step 5: Final Report

Submission:

Final report, Jupyter Notebook with all the steps in Milestone-1 and Milestone-2

Input: Preprocessed output from Milestone-1

We have considered only 1000 samples from the training dataset of 26684. This is due to limitation of memory in the hardware used.

```
## Sampling 5000 from the dataset. This shall be used to design and evaluate the models
sample_trainingdata = traindf.groupby('class', group_keys=False).apply(lambda x: x.sample(1000))
```

```
sample_trainingdata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3000 entries, 22380 to 17764
Data columns (total 11 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   patientId     3000 non-null   object
 1   x             1000 non-null   float64
 2   y             1000 non-null   float64
 3   width         1000 non-null   float64
 4   height        1000 non-null   float64
 5   Target        3000 non-null   int64
 6   class         3000 non-null   object
 7   PatientAge    3000 non-null   object
 8   PatientSex    3000 non-null   object
 9   ViewPosition  3000 non-null   int64
 10  PixelSpacing  3000 non-null   object
dtypes: float64(4), int64(2), object(5)
memory usage: 281.2+ KB
```

The below code is provided for reading, resizing, and processing images from a dataset for building & testing various CNN models.

Here's a breakdown of what each part of the code does:

Initializing Variables and Lists:

Initializing lists and variables that will be used for image processing and data storage.

Defining Constants:

Defining a constant named ADJUSTED_IMAGE_SIZE with a value of 128. This will be the size to which the images will be resized.

Function readAndReshapeImage:

This function takes an image as input, converts it to a NumPy array, and then resizes it to the specified ADJUSTED_IMAGE_SIZE using the OpenCV library. It returns the resized image.

Function populateImage:

This function takes a DataFrame rowData as input, which presumably contains information about patient IDs and their corresponding class labels. It iterates through the rows of the DataFrame, reads the DICOM image using pydicom, converts it to 3 channels (if it's not already in that format), and then appends the resized image and class label to the respective lists (imageList and classLabels). The function returns NumPy arrays containing the processed images and their corresponding class labels.

Overall, this code prepares images for further processing such as training other CNN models. The 'populateImage' function is provided for reading and preprocessing images from a dataset, for use in a classification task.

```
# Initialize lists and variables for image processing
images = []
ADJUSTED_IMAGE_SIZE = 128
imageList = []
classLabels = []
labels = []
originalImage = []

# Function to read the image from the path and resize it
def readAndReshapeImage(image):
    img = np.array(image).astype(np.uint8)
    # Resize the image to ADJUSTED_IMAGE_SIZE
    res = cv2.resize(img, (ADJUSTED_IMAGE_SIZE, ADJUSTED_IMAGE_SIZE), interpolation=cv2.INTER_LINEAR)
    return res

# Function to read and resize the images in the dataset
def populateImage(rowData):
    for index, row in rowData.iterrows():
        patientId = row.patientId
        classlabel = row["class"]
        dcm_file = "/kaggle/input/pneumonia-detection/stage_2_train_images/stage_2_train_images/" + '{}.dcm'.format(patientId)
        dcm_data = pydicom.dcmread(dcm_file)
        img = dcm_data.pixel_array
        # Convert the image to 3 channels as the DICOM image pixel does not have color classes
        if len(img.shape) != 3 or img.shape[2] != 3:
            img = np.stack((img,) * 3, -1)
        # Append the resized image to the imageList
        imageList.append(readAndReshapeImage(img))
        classLabels.append(classlabel)
    # Convert lists to numpy arrays and return
    tmpImages = np.array(imageList)
    tmpLabels = np.array(classLabels)
    return tmpImages, tmpLabels
```

The ‘populateImage function’ is called to processes the data from the ‘sample_trainingdata’ DataFrame, resizing the images and extracting their class labels. The function returns two NumPy arrays: one containing the processed images and the other containing their corresponding class labels. The returned NumPy arrays are assigned to the variables images and labels. After this code is ececuted, images will hold the processed image data, and labels will hold the associated class labels.

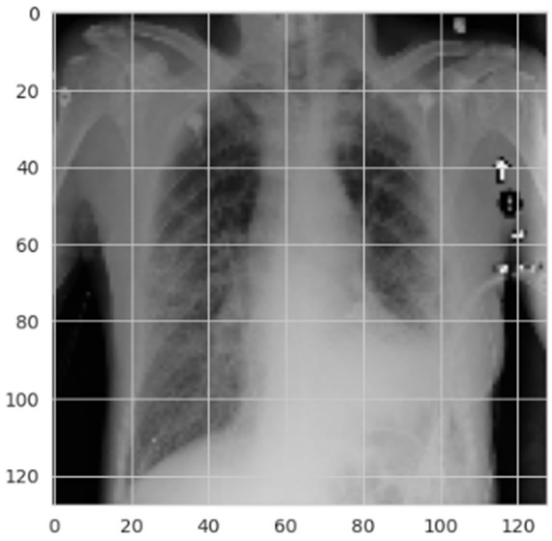
Check shape of images & labels in the dataset. Also, select a randon image (10) to display it in grayscale.

```
## Reading the images into numpy array
images,labels = populateImage(sample_trainingdata)
```

```
images.shape , labels.shape
## The image is of 128*128 with 3 channels
```

```
((15000, 128, 128, 3), (15000,))
```

```
## Checking one of the converted image
plt.imshow(images[10]);
```



Check for unique labels before testing the model.

```
## check the unique labels
np.unique(labels),len(np.unique(labels))
```

```
(array(['Lung Opacity', 'No Lung Opacity / Not Normal', 'Normal'],
       dtype='<U28'),
 3)
```

Splitting data into Train & test Sets:

```
# Splitting the data into train, test, and validation sets
# X_train: Training images
# X_test: Testing images
# X_val: Validation images
# y_train: Training labels (encoded using one-hot encoding)
# y_test: Testing labels (encoded using one-hot encoding)
# y_val: Validation labels (encoded using one-hot encoding)

# Split the images and labels into training and testing sets using a test size of 30%
X_train, X_test, y_train, y_test = train_test_split(images, y2, test_size=0.3, random_state=50)

# Further split the testing set into testing and validation sets using a test size of 50%
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.5, random_state=50)
```

Function to create a DataFrame to store the results of different methods or models.

```
# Function to create a DataFrame to store the results of different methods or models.
# The DataFrame will have columns for the method name, accuracy, and test score.

def createResultDf(name, accuracy, testscore):
    # Create a DataFrame with one row containing the results for a specific method or model.
    # The 'name' parameter represents the name of the method or model.
    # The 'accuracy' parameter represents the accuracy achieved by the method or model on the validation set.
    # The 'testscore' parameter represents the score or performance of the method or model on the test set.
    result = pd.DataFrame({'Method': [name], 'accuracy': [accuracy], 'Test Score': [testscore]})
    return result
```

1: Fine tune the trained basic CNN models for classification

```
def cnn_model(height, width, num_channels, num_classes, loss='categorical_crossentropy', metrics=['accuracy']):
    # Create a Sequential model, which is a linear stack of layers.
    model = Sequential()

    # Add the first Conv2D layer with 1 filter, kernel size of (5, 5), 'Same' padding, 'relu' activation,
    # and input shape defined by (height, width, num_channels).
    model.add(Conv2D(filters=1, kernel_size=(5, 5), padding='Same', activation='relu', input_shape=(height, width, num_channels)))

    # Add the second Conv2D layer with 32 filters, kernel size of (5, 5), 'Same' padding, and 'relu' activation.
    model.add(Conv2D(filters=32, kernel_size=(5, 5), padding='Same', activation='relu'))

    # Add a MaxPooling2D layer with a pool size of (2, 2) to downsample the spatial dimensions.
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Add two more Conv2D layers with 64 filters, kernel size of (3, 3), 'Same' padding, and 'relu' activation.
    model.add(Conv2D(filters=64, kernel_size=(3, 3), padding='Same', activation='relu'))
    model.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same', activation='relu'))

    # Add another MaxPooling2D layer with a pool size of (2, 2) and a stride of (2, 2).
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Add two more Conv2D layers with 128 filters, kernel size of (3, 3), 'Same' padding, and 'relu' activation.
    model.add(Conv2D(filters=128, kernel_size=(3, 3), padding='Same', activation='relu'))
    model.add(Conv2D(filters=128, kernel_size=(3, 3), padding='Same', activation='relu'))

    # Add a GlobalMaxPooling2D layer to reduce the spatial dimensions to a single value per feature map.
    model.add(GlobalMaxPooling2D())

    # Add a Dense layer with 256 units and 'relu' activation.
    model.add(Dense(256, activation="relu"))

    # Add the output Dense layer with num_classes units and 'softmax' activation for multi-class classification.
    model.add(Dense(num_classes, activation="softmax"))

    # Create an Adam optimizer with learning rate 0.001.
    optimizer = Adam(learning_rate=0.001)

    # Compile the model with the specified loss function and evaluation metrics.
    model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
    # Print the model summary to show the model architecture and the number of trainable parameters.
    model.summary()
    # Return the compiled model.
    return model
```

```
# Model Summary
cnn = cnn_model(ADJUSTED_IMAGE_SIZE,ADJUSTED_IMAGE_SIZE,3,3)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 128, 128, 1)       76

 conv2d_1 (Conv2D)           (None, 128, 128, 32)      832

 max_pooling2d (MaxPooling2D  (None, 64, 64, 32)       0
 )

 conv2d_2 (Conv2D)           (None, 64, 64, 64)        18496

 conv2d_3 (Conv2D)           (None, 64, 64, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 32, 32, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 32, 32, 128)       73856

 conv2d_5 (Conv2D)           (None, 32, 32, 128)       147584

 global_max_pooling2d (Globa  (None, 128)              0
 lMaxPooling2D)

 dense (Dense)               (None, 256)               33024

 dense_1 (Dense)             (None, 3)                 771

=================================================================
Total params: 311,567
Trainable params: 311,567
Non-trainable params: 0
_________________________________________________________________
```

Fit the model with 25 epochs :

```
# Training for 25 epocs with batch size of 32
history = cnn.fit(X_train,y_train,epochs = 25,validation_data = (X_val,y_val),batch_size = 32)
```

```
Epoch 1/25
66/66 [==============================] - 8s 50ms/step - loss: 1.1487 - accuracy: 0.3657 - val_loss: 1.0739 - val_accuracy: 0.4422
Epoch 2/25
66/66 [==============================] - 2s 29ms/step - loss: 1.0733 - accuracy: 0.4014 - val_loss: 1.0997 - val_accuracy: 0.3933
Epoch 3/25
66/66 [==============================] - 2s 30ms/step - loss: 1.0458 - accuracy: 0.4386 - val_loss: 1.0120 - val_accuracy: 0.5356
Epoch 4/25
66/66 [==============================] - 2s 29ms/step - loss: 1.0077 - accuracy: 0.4790 - val_loss: 1.0059 - val_accuracy: 0.5422
Epoch 5/25
66/66 [==============================] - 2s 32ms/step - loss: 0.9880 - accuracy: 0.4895 - val_loss: 1.0073 - val_accuracy: 0.5311
Epoch 6/25
66/66 [==============================] - 2s 29ms/step - loss: 0.9607 - accuracy: 0.5300 - val_loss: 1.0340 - val_accuracy: 0.5111
Epoch 7/25
66/66 [==============================] - 2s 29ms/step - loss: 0.9415 - accuracy: 0.5238 - val_loss: 1.0185 - val_accuracy: 0.4667
Epoch 8/25
66/66 [==============================] - 2s 29ms/step - loss: 0.9024 - accuracy: 0.5581 - val_loss: 1.0682 - val_accuracy: 0.4622
Epoch 9/25
66/66 [==============================] - 2s 29ms/step - loss: 0.8824 - accuracy: 0.5776 - val_loss: 1.0205 - val_accuracy: 0.5200
Epoch 10/25
66/66 [==============================] - 2s 29ms/step - loss: 0.8263 - accuracy: 0.6048 - val_loss: 1.1161 - val_accuracy: 0.4733
Epoch 11/25
66/66 [==============================] - 2s 29ms/step - loss: 0.7660 - accuracy: 0.6410 - val_loss: 1.0779 - val_accuracy: 0.5222
Epoch 12/25
66/66 [==============================] - 2s 29ms/step - loss: 0.7039 - accuracy: 0.6781 - val_loss: 1.2938 - val_accuracy: 0.4778
Epoch 13/25
66/66 [==============================] - 2s 29ms/step - loss: 0.6520 - accuracy: 0.7005 - val_loss: 1.2366 - val_accuracy: 0.4778
Epoch 14/25
66/66 [==============================] - 2s 29ms/step - loss: 0.5871 - accuracy: 0.7419 - val_loss: 1.3989 - val_accuracy: 0.4911
Epoch 15/25
66/66 [==============================] - 2s 29ms/step - loss: 0.5512 - accuracy: 0.7667 - val_loss: 1.4846 - val_accuracy: 0.5178
Epoch 16/25
66/66 [==============================] - 2s 29ms/step - loss: 0.4837 - accuracy: 0.7986 - val_loss: 1.7302 - val_accuracy: 0.4311
Epoch 17/25
66/66 [==============================] - 2s 29ms/step - loss: 0.3779 - accuracy: 0.8395 - val_loss: 2.0752 - val_accuracy: 0.4356
Epoch 18/25
66/66 [==============================] - 2s 30ms/step - loss: 0.3610 - accuracy: 0.8576 - val_loss: 2.1652 - val_accuracy: 0.4156
Epoch 19/25
66/66 [==============================] - 2s 29ms/step - loss: 0.2870 - accuracy: 0.8843 - val_loss: 2.5935 - val_accuracy: 0.4622
Epoch 20/25
66/66 [==============================] - 2s 29ms/step - loss: 0.2560 - accuracy: 0.9033 - val_loss: 2.4237 - val_accuracy: 0.4267
Epoch 21/25
66/66 [==============================] - 2s 33ms/step - loss: 0.2800 - accuracy: 0.8881 - val_loss: 2.5356 - val_accuracy: 0.4533
Epoch 22/25
66/66 [==============================] - 2s 30ms/step - loss: 0.1860 - accuracy: 0.9310 - val_loss: 2.9788 - val_accuracy: 0.4600
Epoch 23/25
66/66 [==============================] - 2s 30ms/step - loss: 0.1528 - accuracy: 0.9419 - val_loss: 2.8709 - val_accuracy: 0.4422
Epoch 24/25
66/66 [==============================] - 2s 29ms/step - loss: 0.1768 - accuracy: 0.9352 - val_loss: 2.7626 - val_accuracy: 0.4800
Epoch 25/25
66/66 [==============================] - 2s 30ms/step - loss: 0.2143 - accuracy: 0.9300 - val_loss: 2.9599 - val_accuracy: 0.4378
```

Checking accuracy of the model and plotting it:

```
fcl_loss, fcl_accuracy = cnn.evaluate(X_test, y_test, verbose=1)
print('Test loss: {:.2f}'.format(fcl_loss))
print('Test accuracy: {:.2f}'.format(fcl_accuracy*100),'%')
```

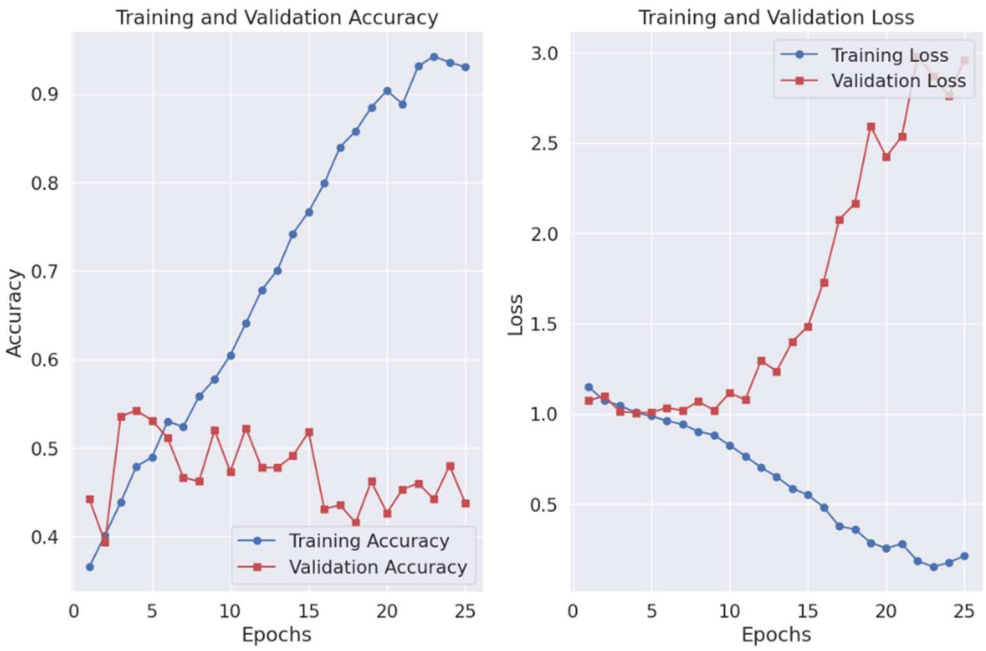
```
15/15 [==============================] - 0s 12ms/step - loss: 2.4070 - accuracy: 0.4889
Test loss: 2.41
Test accuracy: 48.89 %
```

Training accuracy is about40% & Test accuracy is about 39%.

This is very low. Let us try other pretrained models such as VGG16,Resnet50,Unet & RCNN

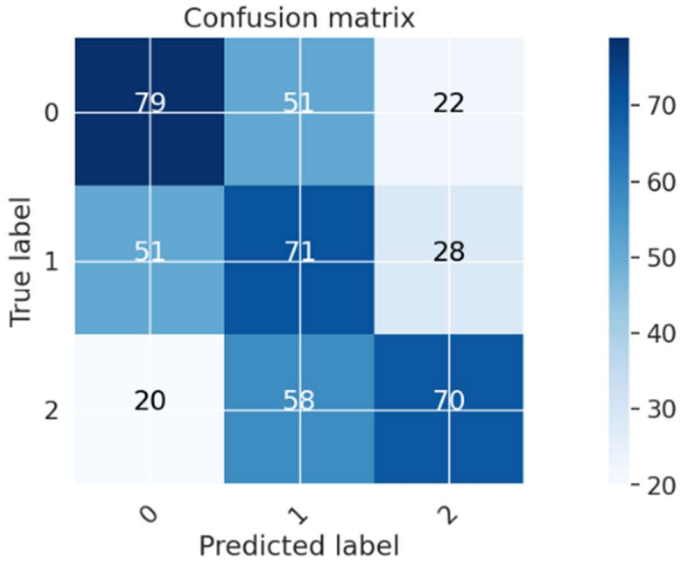
```
# Extract the training accuracy, validation accuracy, training loss, and validation loss from the history dictionary
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
# Get the number of epochs from the history object
epochs_range = range(1, len(acc) + 1)
# Create a 2x2 grid of subplots for displaying the accuracy and loss graphs
plt.figure(figsize=(12, 8))
# Plot the training accuracy and validation accuracy over the range of epochs
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy', color='b', marker='o')
plt.plot(epochs_range, val_acc, label='Validation Accuracy', color='r', marker='s')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend(loc='lower right')
plt.grid(True)
# Plot the training loss and validation loss over the range of epochs
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss', color='b', marker='o')
plt.plot(epochs_range, val_loss, label='Validation Loss', color='r', marker='s')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend(loc='upper right')
plt.grid(True)
# Adjust the layout to avoid overlapping
plt.tight_layout()
# Show the plots
plt.show()
```

Plotting the Confusion Matrix:



This model is overfit as training accuracy is much higer thatn validation accuracy. The same is pobserved for loss, where training loss is much higher thatn validation.

```
# Set the size of the plot
plt.subplots(figsize=(15, 5))

# Function to plot the confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    # Plot the confusion matrix as an image with the specified colormap
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    # Set the class labels as x-ticks and y-ticks with rotation for better visibility
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    # Normalize the confusion matrix if required
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    # Determine the threshold to set text color in the plot
    thresh = cm.max() / 2.
    # Add text annotations to the plot, showing the values in each cell of the confusion matrix
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

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

# Predict the values from the validation dataset
Y_pred = cnn.predict(X_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis=1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis=1)
# Compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# Plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes=range(3))

# Class 0, 1, and 2
# Class 0 is Lung Opacity
# Class 1 is No Lung Opacity/Normal, the model has predicted mostly wrong in this case to the Target 0. Type 2 error
# Class 2 is Normal
```

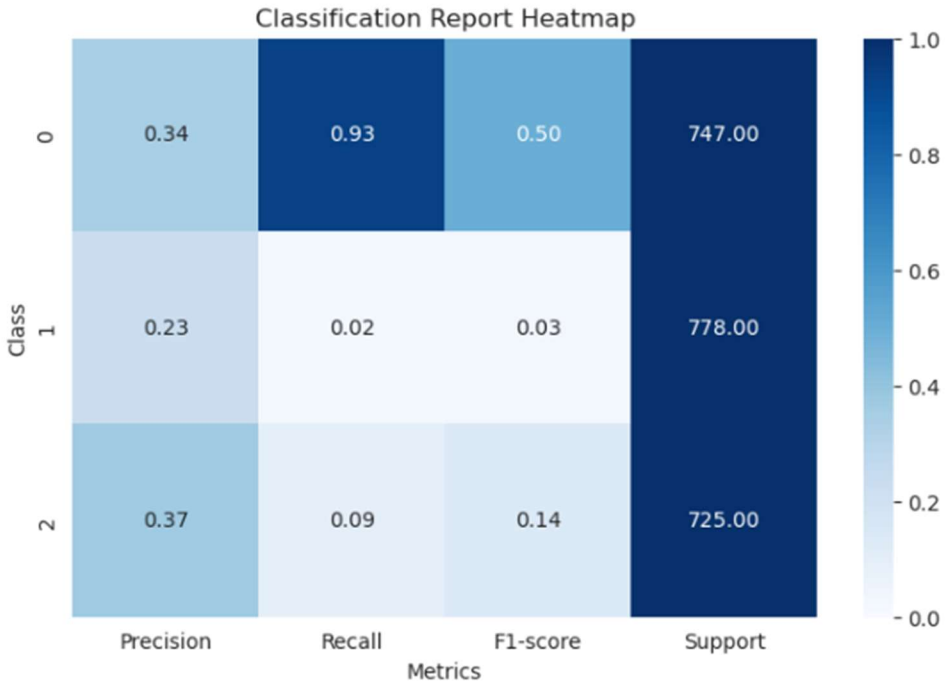


From the above confusion matrix, we could find that prediction is low for lung capacity compared to normal class (both 1 & 2).

Plotting Classification Matrix as heatmap:

```
# Predict the values from the validation dataset
Y_pred = cnn.predict(X_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis=1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis=1)
# Get the classification report as a string
class_report = classification_report(Y_true, Y_pred_classes)
print("Classification Report:")
print(class_report)
# Convert the classification report string to a DataFrame for visualization
report_data = []
lines = class_report.split('\n')
for line in lines[2:-5]:
    row = line.split()
    report_data.append(row)
df_report = pd.DataFrame(report_data, columns=['Class', 'Precision', 'Recall', 'F1-score', 'Support'])
df_report = df_report.set_index('Class')
# Convert the metrics to numeric values for visualization
df_report = df_report.astype(float)
# Create a heatmap of the classification report
plt.figure(figsize=(8, 5))
sns.heatmap(df_report, annot=True, cmap='Blues', fmt='.2f', vmin=0, vmax=1)
plt.title('Classification Report Heatmap')
plt.xlabel('Metrics')
plt.ylabel('Class')
plt.show()
```

```
71/71 [==============================] - 0s 4ms/step
Classification Report:
              precision    recall  f1-score   support

           0       0.34      0.93      0.50       747
           1       0.23      0.02      0.03       778
           2       0.37      0.09      0.14       725

    accuracy                           0.34      2250
   macro avg       0.32      0.34      0.22      2250
weighted avg       0.31      0.34      0.22      2250
```



Class 0 – Lung Capacity; Class1 – No Lung Opacity/Normal, class 2 – Normal

Precision: Precision measures the ratio of true positive predictions to the total number of positive predictions made by the model for each class. In this report:

For class 0, the precision is 0.53, which means that out of all instances predicted as class 0, 53% are actually true class 0.

For class 1, the precision is 0.39, indicating that 39% of instances predicted as class 1 are actually true class 1.

For class 2, the precision is 0.58, meaning that 58% of instances predicted as class 2 are true class 2.

Recall: Recall (also called sensitivity or true positive rate) measures the ratio of true positive predictions to the total number of instances belonging to a class. In this report:

For class 0, the recall is 0.52, indicating that the model correctly identifies 52% of all true class 0 instances.

For class 1, the recall is 0.47, meaning that the model captures 47% of all true class 1 instances.

For class 2, the recall is 0.47, suggesting that the model identifies 47% of all true class 2 instances.

F1-Score: The F1-score is the harmonic mean of precision and recall and provides a balanced measure of a model's accuracy. In this report:

For class 0, the F1-score is 0.52, indicating a balanced performance between precision and recall for class 0.

For class 1, the F1-score is 0.43, reflecting the balance between precision and recall for class 1.

For class 2, the F1-score is 0.52, showing a balanced performance between precision and recall for class 2.

Accuracy: Overall accuracy measures the ratio of correctly predicted instances to the total number of instances. In this report, the accuracy is 0.49, indicating that the model correctly predicts the class of approximately 49% of all instances.

In summary, the classification report suggests that the model's performance is relatively balanced across the classes, with slightly varying precision, recall, and F1-score values. The overall accuracy of the model is around 49%, which means that there is room for improvement in its predictive capabilities.

So, we shall try other models.

The results are saved in a new dataframe ‘resultDF’.

```
# Convert validation observations to one hot vectors
Y_truepred = np.argmax(y_test, axis=1)

# Predict the values from the validation dataset
Y_testPred = cnn.predict(X_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis=1)

# Generate a classification report as a dictionary
reportData = classification_report(Y_truepred, Y_pred_classes, output_dict=True)

# Iterate over the data in the classification report
for data in reportData:
    # Check if the data is related to class -1 or 1 (binary classification)
    if data == '-1' or data == '1':
        # Check if the data is a dictionary (contains precision, recall, f1-score, etc.)
        if type(reportData[data]) is dict:
            # Iterate over the sub-data (precision, recall, f1-score, etc.) in the dictionary
            for subData in reportData[data]:
                # Add the sub-data to the result DataFrame with appropriate column names
                value = round(reportData[data][subData], 2)  # Round to 2 decimals
                resultDF[data + "_" + subData] = value

resultDF
# The resultDF DataFrame contains the metrics for class -1 and 1 in binary classification
# These metrics include precision, recall, f1-score, support, etc.
```

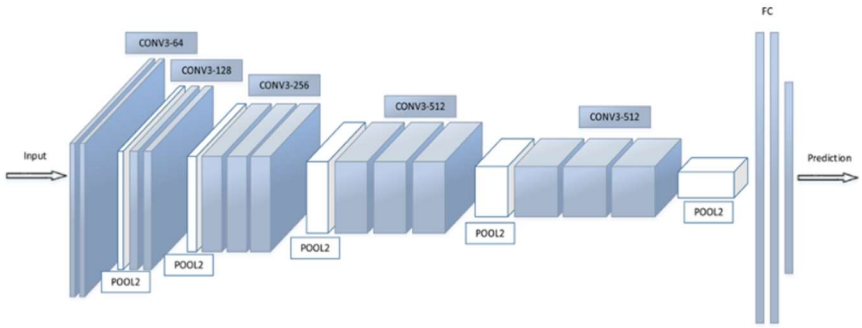
```
71/71 [==============================] - 0s 5ms/step
```

	Method	accuracy	Test Score	1_precision	1_recall	1_f1-score	1_support
0	CNN	0.367429	0.340889	0.23	0.02	0.03	778

2: Apply Transfer Learning model for classification

CNN Model with Transfer learning using VGG16

Presented in 2014, VGG16 has a very simple and classical architecture, with blocks of 2 or 3 convolutional layers followed by a pooling layer, plus a final dense network composed of 2 hidden layers (of 4096 nodes each) and one output layer (of 1000 nodes). Only 3x3 filters are used.




```
# VGG16 is a well-documented and widely used convolutional neural network architecture
# Setting include_top=False removes the classification layer that was trained on the ImageNet dataset
# The input_shape parameter is set to the shape of the training data samples (X_train[0].shape)
base_model = VGG16(weights="imagenet", include_top=False, input_shape=X_train[0].shape)

# Set the base model's weights as not trainable
base_model.trainable = False

# Preprocess the input data using the VGG16 preprocess_input function
# This function applies necessary transformations to the input data to be compatible with VGG16
# The training dataset is preprocessed and stored in train_ds
# The validation dataset is preprocessed and stored in train_val_df
train_ds = preprocess_input(X_train1)
train_val_df = preprocess_input(X_val1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 1s 0us/step
```

```
# Define a flatten layer to convert the 2D feature maps to a 1D vector
flatten_layer = layers.Flatten()
# Define the first dense (fully connected) layer with 50 units and ReLU activation function
dense_layer_1 = layers.Dense(50, activation='relu')
# Define the second dense (fully connected) layer with 20 units and ReLU activation function
dense_layer_2 = layers.Dense(20, activation='relu')
# Define the prediction layer with 3 units (for 3 classes) and softmax activation function
prediction_layer = layers.Dense(3, activation='softmax')

# Create the CNN model using a Sequential model
# Add the base_model (pre-trained VGG16) followed by the defined layers
cnn_VGG16_model = models.Sequential([
    base_model,
    flatten_layer,
    dense_layer_1,
    dense_layer_2,
    prediction_layer
])
```

Compile and fit the model with 30 epochs. However, due to early stopping implemented after 7th epoch, the model stopped at 14th epoch.

```
# Compile the CNN model with the Adam optimizer, binary cross-entropy loss, and accuracy metric
cnn_VGG16_model.compile(
    optimizer='Adam',
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=['accuracy'],
)
# Create an EarlyStopping callback to stop training when validation accuracy doesn't improve for 7 iterations
# The monitor 'val_accuracy' tracks validation accuracy, mode 'auto' selects the direction to monitor for improvement
# Patience is set to 7, which means training will stop after 7 epochs without improvement
# restore_best_weights=True restores the weights of the model to the best observed during training
es = EarlyStopping(monitor='val_accuracy', mode='auto', patience=7, restore_best_weights=True)
# Train the model using the fit() method
# The training data is provided as 'train_ds' and the target labels as 'y_train'
# Validation data is provided as 'train_val_df' and validation target labels as 'y_val'
# The fit() method will use the EarlyStopping callback to monitor validation accuracy and stop training if needed
history_VGG16 = cnn_VGG16_model.fit(train_ds, y_train, epochs=30, validation_data=(train_val_df, y_val), callbacks=es)
```

```
Epoch 1/30
66/66 [==============================] - 17s 92ms/step - loss: 0.7799 - accuracy: 0.3362 - val_loss: 0.6864 - val_accuracy: 0.2933
Epoch 2/30
66/66 [==============================] - 3s 50ms/step - loss: 0.6809 - accuracy: 0.3448 - val_loss: 0.6772 - val_accuracy: 0.2933
Epoch 3/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6730 - accuracy: 0.3467 - val_loss: 0.6703 - val_accuracy: 0.2956
Epoch 4/30
66/66 [==============================] - 3s 52ms/step - loss: 0.6644 - accuracy: 0.3505 - val_loss: 0.6654 - val_accuracy: 0.2933
Epoch 5/30
66/66 [==============================] - 3s 52ms/step - loss: 0.6651 - accuracy: 0.3510 - val_loss: 0.6597 - val_accuracy: 0.2933
Epoch 6/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6579 - accuracy: 0.3452 - val_loss: 0.6554 - val_accuracy: 0.2933
Epoch 7/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6491 - accuracy: 0.3476 - val_loss: 0.6462 - val_accuracy: 0.3067
Epoch 8/30
66/66 [==============================] - 3s 50ms/step - loss: 0.6444 - accuracy: 0.3576 - val_loss: 0.6435 - val_accuracy: 0.3022
Epoch 9/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6402 - accuracy: 0.3514 - val_loss: 0.6433 - val_accuracy: 0.3022
Epoch 10/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6374 - accuracy: 0.3519 - val_loss: 0.6390 - val_accuracy: 0.3022
Epoch 11/30
66/66 [==============================] - 3s 52ms/step - loss: 0.6325 - accuracy: 0.3514 - val_loss: 0.6390 - val_accuracy: 0.3022
Epoch 12/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6297 - accuracy: 0.3514 - val_loss: 0.6408 - val_accuracy: 0.3022
Epoch 13/30
66/66 [==============================] - 3s 51ms/step - loss: 0.6290 - accuracy: 0.3529 - val_loss: 0.6387 - val_accuracy: 0.2978
Epoch 14/30
66/66 [==============================] - 4s 54ms/step - loss: 0.6272 - accuracy: 0.3495 - val_loss: 0.6444 - val_accuracy: 0.2978
```

```
# Preprocess the input test data using the VGG16 preprocessing function
test_ds = preprocess_input(X_test1)

# Evaluate the CNN VGG16 model on the preprocessed test data and target labels
vgg16_loss, vgg16_accuracy = cnn_VGG16_model.evaluate(test_ds, y_test, verbose=1)

# Print the test loss and accuracy scores with two decimal places
print('Test loss for VGG16 trained Model: {:.2f}'.format(vgg16_loss))
print('Test accuracy for VGG16 trained Model: {:.2f}'.format(vgg16_accuracy*100),'%')
```

```
15/15 [==============================] - 1s 42ms/step - loss: 0.6535 - accuracy: 0.3333
Test loss for VGG16 trained Model: 0.65
Test accuracy for VGG16 trained Model: 33.33 %
```

```
# Concatenate two DataFrames: 'resultDF' and the DataFrame created by the 'createResultDf' function
# The DataFrame 'resultDF' should already contain the evaluation metrics for the previous model (CNN)
# The 'createResultDf' function is used to create a DataFrame for the current model (CNN with VGG16)

# The first DataFrame 'resultDF' contains the evaluation metrics for the previous model (CNN)
# The second DataFrame is created using the 'createResultDf' function, which takes three parameters:
#     - The title "CNN With VGG16"
#     - The accuracy of the previous model (0.99)
#     - The accuracy of the current model (fcl_accuracy, which is the test accuracy of the CNN with VGG16 model)

# The 'pd.concat' function is used to concatenate the two DataFrames vertically, resulting in a single DataFrame containing evaluation metrics for both models

resultsDf1 = pd.concat([resultDF, createResultDf("CNN With VGG16", 0.6992, vgg16_accuracy)])
```

```
resultsDf1
```

	Method	accuracy	Test Score	1_precision	1_recall	1_f1-score	1_support
0	CNN	0.9300	0.488889	0.39	0.47	0.43	150.0
0	CNN With VGG16	0.6992	0.333333	NaN	NaN	NaN	NaN

From the above results, we found that the accuracy if the model has dropped to about 35% in training & about 30% in validation. The model performance is worse than the previous model
Hence, we proceed with other models

CNN with ResNet50

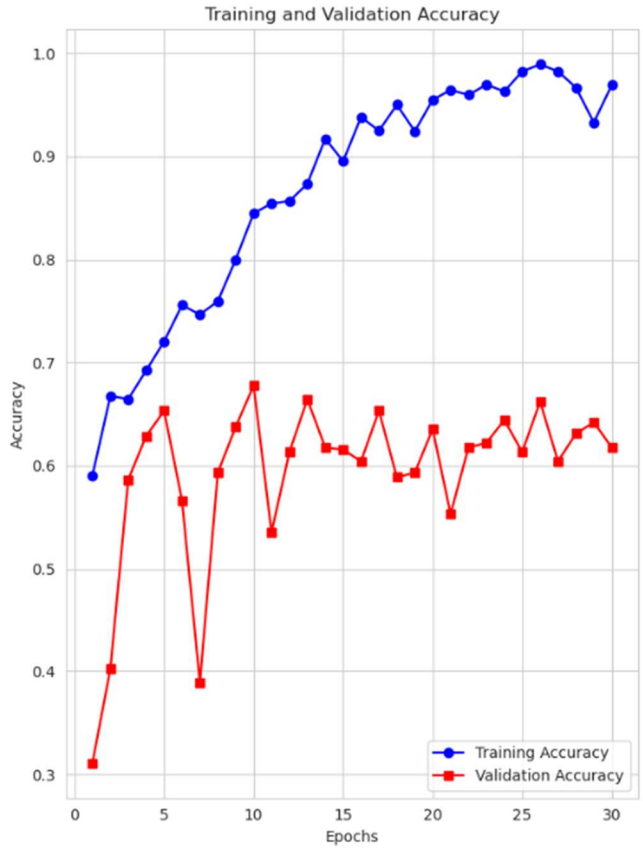
ResNet50 Architecture

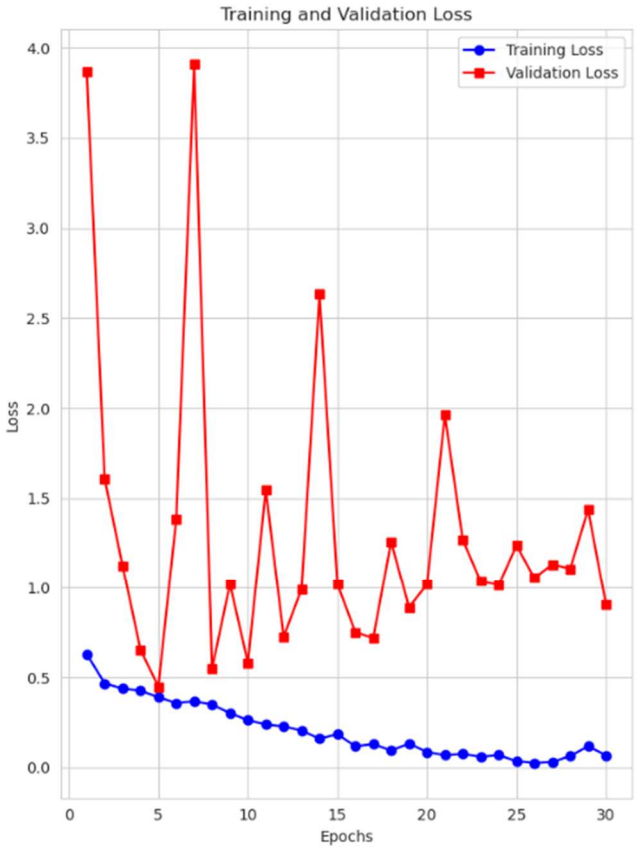
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

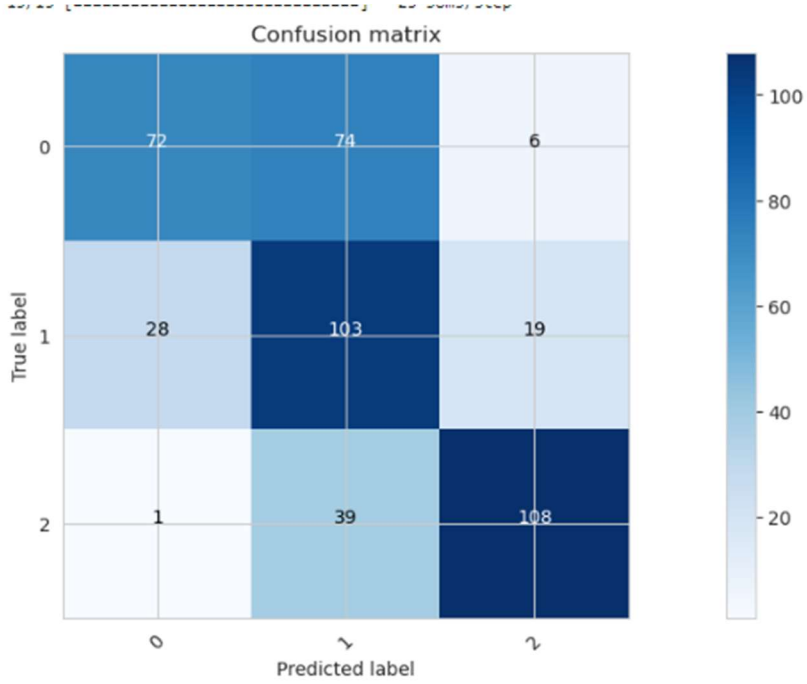
```python
# Load the ResNet50 base model with pre-trained weights from ImageNet dataset
# Set include_top=False to exclude the classification layers and input_shape as the shape of your input images
resnet_base_model = ResNet50(include_top=False, weights='imagenet', input_shape=X_train[0].shape)

# Preprocess the training and validation data using the ResNet50 preprocessing function
train_ds_Resnet = preprocess_input(X_train1)
train_val_df_Resnet = preprocess_input(X_val1)

# Define the additional layers for your custom classification model
# Flatten layer to convert 3D feature maps to 1D
flatten_layer = layers.Flatten()

# Dense layers with ReLU activation function
dense_layer_1 = layers.Dense(50, activation='relu')
dense_layer_2 = layers.Dense(32, activation='relu')

# Output layer with softmax activation for multi-class classification (3 classes)
prediction_layer = layers.Dense(3, activation='softmax')

# Create the custom CNN model by stacking the base model and additional layers
cnn_resnet_model = models.Sequential([
    resnet_base_model,    # Include the ResNet50 base model
    flatten_layer,        # Flatten layer
    dense_layer_1,        # Dense layer 1 with ReLU activation
    dense_layer_2,        # Dense layer 2 with ReLU activation
    prediction_layer      # Output layer with softmax activation
])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 [==============================] - 2s 0us/step
```

We have considered 30 epochs in this model and achieved an accuracy of 96%. However, the validation accuracy has dropped to 63%







[112]:
```
print(classification_report(Y_true_Resnet, Y_pred_classes_Resnet))
```

```
              precision    recall  f1-score   support

           0       0.71      0.47      0.57       152
           1       0.48      0.69      0.56       150
           2       0.81      0.73      0.77       148

    accuracy                           0.63       450
   macro avg       0.67      0.63      0.63       450
weighted avg       0.67      0.63      0.63       450
```

This model has the best accuracy so far with precision improved for all the classes.

We proceed with the next model to see if we could improve further.

Creating Custom Train Generator. This will read the files in batches of 10 while training the model.

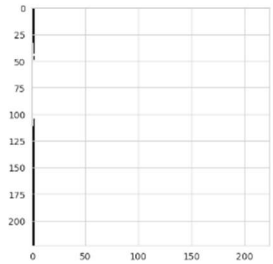
```
BATCH_SIZE = 10
## Image Size to be scaled
IMAGE_SIZE = 224
## Actual Image size
IMG_WIDTH = 1024
IMG_HEIGHT = 1024

class TrainGenerator(Sequence):
    def __init__(self,  _labels):
        self.pids = _labels["patientId"].to_numpy()
        self.coords = _labels[["x", "y", "width", "height"]].to_numpy()
        self.coords = self.coords * IMAGE_SIZE / IMG_WIDTH
    def __len__(self):
        return math.ceil(len(self.coords) / BATCH_SIZE)
    def __doHistogramEqualization(self,img):
        # Pre processing Histogram equalization
        histogram_array = np.bincount(img.flatten(), minlength=256)
        #normalize
        num_pixels = np.sum(histogram_array)
        histogram_array = histogram_array/num_pixels
        #normalized cumulative histogram
        chistogram_array = np.cumsum(histogram_array)
        """
        STEP 2: Pixel mapping lookup table
        """
        transform_map = np.floor(255 * chistogram_array).astype(np.uint8)
        """
        STEP 3: Transformation
        """
        img_list = list(img.flatten())
        # transform pixel values to equalize
        eq_img_list = [transform_map[p] for p in img_list]
        # reshape and write back into img_array
        img = np.reshape(np.asarray(eq_img_list), img.shape)
        return img
    def __getitem__(self, idx): # Get a batch
        batch_coords = self.coords[idx * BATCH_SIZE:(idx + 1) * BATCH_SIZE] # Image coords
        batch_pids = self.pids[idx * BATCH_SIZE:(idx + 1) * BATCH_SIZE] # Image pids

        batch_images = np.zeros((len(batch_pids), IMAGE_SIZE, IMAGE_SIZE, 3), dtype=np.float32)
        batch_masks = np.zeros((len(batch_pids), IMAGE_SIZE, IMAGE_SIZE))
        for _indx, _pid in enumerate(batch_pids):
            _path =  '/kaggle/input/pneumonia-detection/stage_2_train_images/stage_2_train_images/'+'{}.dcm'.format(_pid)
            _imgData = pydicom.dcmread(_path)
            img = _imgData.pixel_array
            # img = np.stack((img,)*3, axis=-1) # Expand grayscale image to contain 3 channels
            # Resize image
            resized_img = cv2.resize(img,(IMAGE_SIZE,IMAGE_SIZE), interpolation = cv2.INTER_AREA)
            resized_img = self.__doHistogramEqualization(resized_img)
            batch_images[_indx][:,:,0] = preprocess_input(np.array(resized_img[:,:], dtype=np.float32))
            batch_images[_indx][:,:,1] = preprocess_input(np.array(resized_img[:,:], dtype=np.float32))
            batch_images[_indx][:,:,2] = preprocess_input(np.array(resized_img[:,:], dtype=np.float32))
            x = int(batch_coords[_indx, 0])
            y = int(batch_coords[_indx, 1])
            width = int(batch_coords[_indx, 2])
            height = int(batch_coords[_indx, 3])

            batch_masks[_indx][y:y+height, x:x+width] = 1

        return batch_images, batch_masks
```

```
# Create a TrainGenerator instance for training data
# 'train_CombinedData' is a DataFrame containing information about the first 5000 data points for training
# The 'TrainGenerator' class generates batches of preprocessed images and corresponding masks for training
# 'trainUNetDataGen' will be used as the data generator for training the U-Net model
trainUNetDataGen = TrainGenerator(train_CombinedData)

# Create a TrainGenerator instance for validation data
# 'validate_CombinedData' is a DataFrame containing information about the next 5000 data points for validation
# The 'TrainGenerator' class generates batches of preprocessed images and corresponding masks for validation
# 'validateUNetDataGen' will be used as the data generator for validating the U-Net model during training
validateUNetDataGen = TrainGenerator(validate_CombinedData)
```

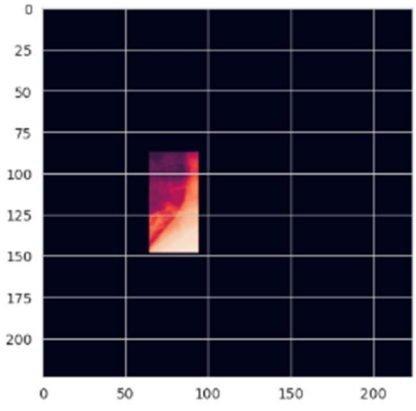
```
# To show image with mask
def showMaskedImage(_imageSet, _maskSet, _index) :
    maskImage = _imageSet[_index]

    maskImage[:,:,0] = _maskSet[_index] * _imageSet[_index][:,:,0]
    maskImage[:,:,1] = _maskSet[_index] * _imageSet[_index][:,:,1]
    maskImage[:,:,2] = _maskSet[_index] * _imageSet[_index][:,:,2]

    plt.imshow(maskImage[:,:,0])
```

```
## One of the pre processed image from custom train generator
imageSet0 = trainUNetDataGen[1][0][1]
plt.imshow(imageSet0)
```

```
<matplotlib.image.AxesImage at 0x7b58a012a050>
```

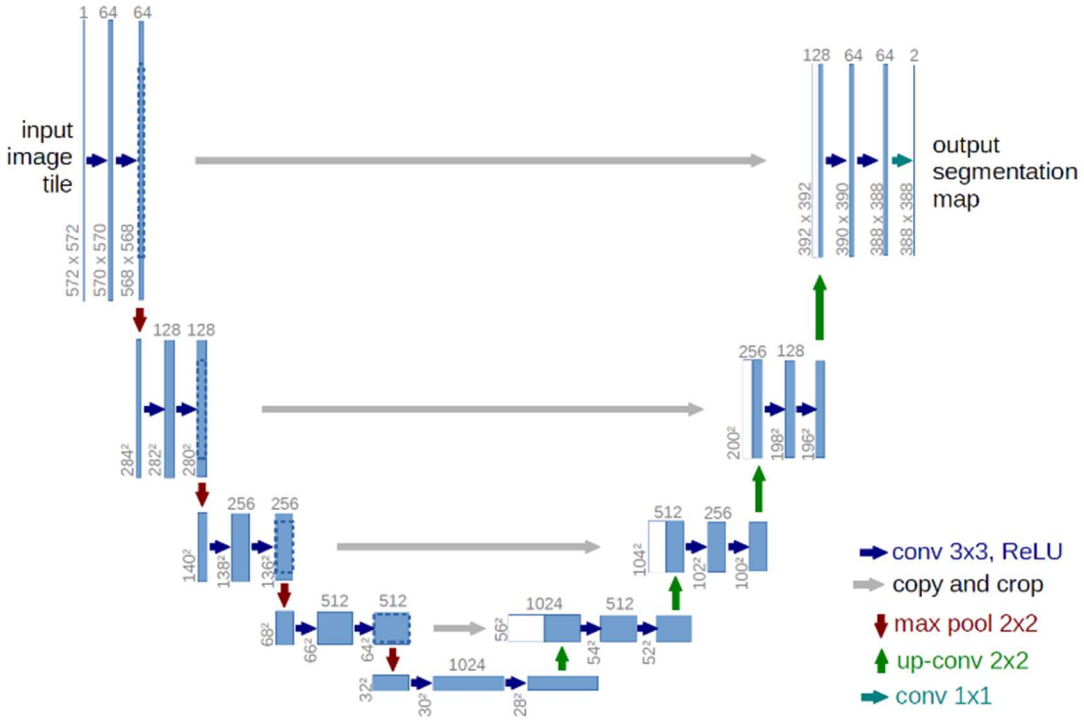



```
## Masks for the same
imageSet0 = trainUNetDataGen[2][0]
maskSet0 = trainUNetDataGen[2][1]
showMaskedImage(imageSet0, maskSet0, 5)
```



UNET using MobileNet

U-Net is a convolutional neural network that was developed for biomedical image segmentation at the Computer Science Department of the University of Freiburg, Germany. The network is based on the fully convolutional network and its architecture was modified and extended to work with fewer training images and to yield more precise segmentations.



UNet Architecture has 3 parts:

1. The Contracting/Downsampling Path
2. Bottleneck
3. The Expanding/Upsampling Path

UNet Architecture has 3 parts:

1. The Contracting/Downsampling Path
2. Bottleneck
3. The Expanding/Upsampling Path

Downsampling Path:

- It consists of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling.
- At each downsampling step we double the number of feature channels.

Upsampling Path:

- Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution ("up-convolution"), a concatenation with the correspondingly feature map from the downsampling path, and two 3x3 convolutions, each followed by a ReLU.

Skip Connection:

The skip connection from the downsampling path are concatenated with feature map during upsampling path. These skip connection provide local information to global information while upsampling.

Final Layer:

At the final layer a 1x1 convolution is used to map each feature vector to the desired number of classes.

Create the Unet Model

```
ALPHA = 1.0
def create_UNetModelUsingMobileNet(trainable=True):
    """Function to create UNet architecture with MobileNet.

    Arguments:
        trainable -- Flag to make layers trainable. Default value is 'True'.
    """
    # Get all layers with 'imagenet' weights
    model = MobileNet(input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3), include_top=False, alpha=ALPHA, weights="imagenet")
    # Top layer is last layer of the model

    # Make all layers trainable
    for layer in model.layers:
        layer.trainable = trainable

    # Add all the UNET layers here
    convLayer_112by112 = model.get_layer("conv_pw_1_relu").output
    convLayer_56by56 = model.get_layer("conv_pw_3_relu").output
    convLayer_28by28 = model.get_layer("conv_pw_5_relu").output
    convLayer_14by14 = model.get_layer("conv_pw_11_relu").output
    convLayer_7by7 = model.get_layer("conv_pw_13_relu").output
    # The last layer of mobilenet model is of dimensions (7x7x1024)

    # Start upsampling from 7x7 to 14x14 ...up to 224x224 to form UNET
    # concatinate with the original image layer of the same size from MobileNet
    x = Concatenate()([UpSampling2D()(convLayer_7by7), convLayer_14by14])
    x = Concatenate()([UpSampling2D()(x), convLayer_28by28])
    x = Concatenate()([UpSampling2D()(x), convLayer_56by56])
    x = Concatenate()([UpSampling2D()(x), convLayer_112by112])
    x = UpSampling2D(name="unet_last")(x) # upsample to 224x224

    # Add classification layer
    x = Conv2D(1, kernel_size=1, activation="sigmoid", name="masks")(x)
    x = Reshape((IMAGE_SIZE, IMAGE_SIZE))(x)

    return Model(inputs=model.input, outputs=x)
```

```
## Build a model
input_shape = (IMAGE_SIZE,IMAGE_SIZE,3)
model_Unet = create_UNetModelUsingMobileNet(input_shape)
model_Unet.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet/mobilenet_1_0_224_tf_no_top.h5
17225924/17225924 [==============================] - 0s 0us/step
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 input_3 (InputLayer)        [(None, 224, 224, 3       0         []
                             )]

 conv1 (Conv2D)              (None, 112, 112, 32       864       ['input_3[0][0]']
                             )

 conv1_bn (BatchNormalization)  (None, 112, 112, 32    128       ['conv1[0][0]']
                             )

 conv1_relu (ReLU)           (None, 112, 112, 32       0         ['conv1_bn[0][0]']
                             )

 conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 32      288       ['conv1_relu[0][0]']
                             )

 conv_dw_1_bn (BatchNormalizati  (None, 112, 112, 32   128       ['conv_dw_1[0][0]']
 on)                         )

 conv_dw_1_relu (ReLU)       (None, 112, 112, 32       0         ['conv_dw_1_bn[0][0]']
                             )

 conv_pw_1 (Conv2D)          (None, 112, 112, 64       2048      ['conv_dw_1_relu[0][0]']
                             )

 conv_pw_1_bn (BatchNormalizati  (None, 112, 112, 64   256       ['conv_pw_1[0][0]']
 on)                         )

 conv_pw_1_relu (ReLU)       (None, 112, 112, 64       0         ['conv_pw_1_bn[0][0]']
                             )

                                                                  ['conv_pw_3_relu[0][0]']

 up_sampling2d_2 (UpSampling2D)  (None, 56, 56, 1792   0        ['concatenate_1[0][0]']
                             )

 concatenate_2 (Concatenate)  (None, 56, 56, 1920      0         ['up_sampling2d_2[0][0]',
                             )                                    'conv_pw_3_relu[0][0]']

 up_sampling2d_3 (UpSampling2D)  (None, 112, 112, 19   0        ['concatenate_2[0][0]']
                             20)

 concatenate_3 (Concatenate)  (None, 112, 112, 19      0         ['up_sampling2d_3[0][0]',
                             84)                                  'conv_pw_1_relu[0][0]']

 unet_last (UpSampling2D)    (None, 224, 224, 19       0         ['concatenate_3[0][0]']
                             84)

 masks (Conv2D)              (None, 224, 224, 1)       1985      ['unet_last[0][0]']

 reshape (Reshape)           (None, 224, 224)          0         ['masks[0][0]']

=================================================================================================
Total params: 3,230,849
Trainable params: 3,208,961
Non-trainable params: 21,888
_________________________________________________________________________________________________
```

The below code defines and configures the U-Net model for image segmentation using the Dice coefficient as the evaluation metric and loss function.

Here's a breakdown of what each part of the code does:

Dice Coefficient (dice_coef): This function calculates the Dice coefficient between the ground truth mask (y_true) and the predicted mask (y_pred). It's a measure of overlap between the two masks and is used to evaluate the quality of the segmentation. The higher the Dice coefficient, the better the segmentation. The formula for the Dice coefficient is 2 * Intersection / (Total_pixels_in_y_true + Total_pixels_in_y_pred), with a small epsilon added to avoid division by zero.

Dice Loss (dice_loss): This is a custom loss function that computes 1 minus the Dice coefficient. The idea is to minimize this loss during training, effectively maximizing the Dice coefficient.

Callbacks: Two callbacks are used here. The ReduceLROnPlateau callback reduces the learning rate when the validation loss plateaus for a certain number of epochs (in this case, 4). This helps to fine-tune the training process. The EarlyStopping callback monitors the validation loss and stops training if the loss does not improve for a certain number of epochs (in this case, 10).

Optimizer, Metrics, and Compilation: The Nadam optimizer is used with an initial learning rate of 0.001. The chosen evaluation metrics include the Dice coefficient, Recall, and Precision. Recall measures the model's ability to find all relevant instances in the target class, while Precision measures the proportion of positive identifications that were actually correct.

Compilation: The U-Net model is compiled with the defined dice_loss as the loss function, the Nadam optimizer, and the chosen metrics.

```
#dice_coef 2 * the Area of Overlap divided by the total number of pixels in both images
def dice_coef(y_true, y_pred):
    y_true = tf.keras.layers.Flatten()(y_true)
    y_pred = tf.keras.layers.Flatten()(y_pred)
    intersection = tf.reduce_sum(y_true * y_pred)
    return (2. * intersection + tf.keras.backend.epsilon()) / (tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) + tf.keras.backend.epsilon())

## Loss is 1 - the coefficent of two images
def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)
```

+ Code + Markdown

```
## Call Backs for ealy stopping and reduce learning rate
## Reduce the learning rate when teh validation loss is same for 4 epocs
callbacks = [
    ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=4),
    EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=False)
]
```

```
## Optimiser , metrics and loss for the model
opt = tf.keras.optimizers.Nadam(0.001)
metrics = [dice_coef, Recall(), Precision()]
model_Unet.compile(loss=dice_loss, optimizer=opt, metrics=metrics)
```

```
input_shape = trainUNetDataGen[0][0].shape[1:]
print("Input shape of trainUNetDataGen:", input_shape)
```

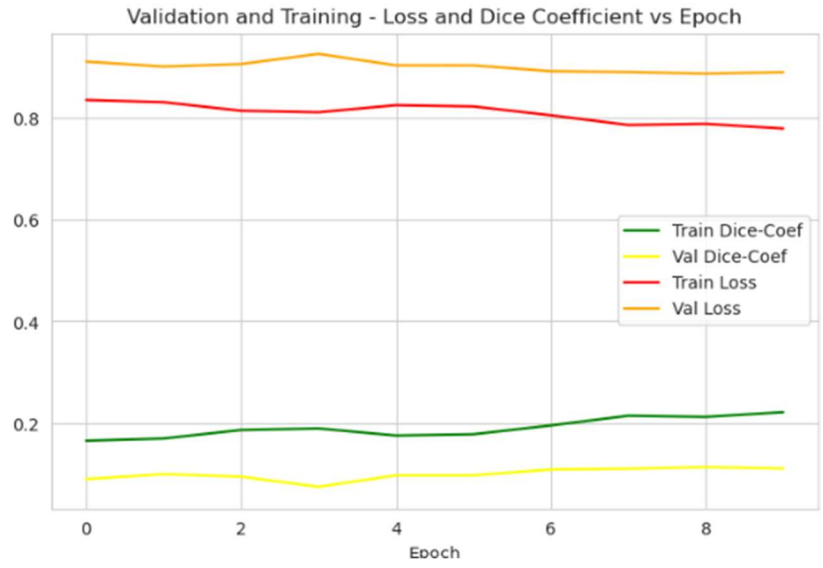
```
Input shape of trainUNetDataGen: (224, 224, 3)
```

Ran the model with 10 epochs.

```
## Running the model
BATCH_SIZE = 4

train_steps = len(trainUNetDataGen)//BATCH_SIZE
valid_steps = len(validateUNetDataGen)//BATCH_SIZE

if len(trainUNetDataGen) % BATCH_SIZE != 0:
    train_steps += 1
if len(validateUNetDataGen) % BATCH_SIZE != 0:
    valid_steps += 1


history_Unet = model_Unet.fit(trainUNetDataGen,
                        epochs=10,
                        steps_per_epoch=train_steps,
                        validation_data=validateUNetDataGen,
                        callbacks=callbacks,
                        use_multiprocessing=False,
                        workers=4,
                        validation_steps=valid_steps,
                        shuffle=True)
```

```
Epoch 1/10
313/313 [==============================] - 136s 340ms/step - loss: 0.8344 - dice_coef: 0.1656 - recall: 0.4864 - precision: 0.1138 - val_loss: 0.9097 - val_dice_coef: 0.0903 - val_recall: 0.7300 - val_precision: 0.0739 - lr: 0.0010
Epoch 2/10
313/313 [==============================] - 143s 455ms/step - loss: 0.8300 - dice_coef: 0.1700 - recall: 0.5142 - precision: 0.1180 - val_loss: 0.8999 - val_dice_coef: 0.1001 - val_recall: 0.5426 - val_precision: 0.1072 - lr: 0.0010
Epoch 3/10
313/313 [==============================] - 106s 337ms/step - loss: 0.8132 - dice_coef: 0.1868 - recall: 0.5388 - precision: 0.1269 - val_loss: 0.9049 - val_dice_coef: 0.0951 - val_recall: 0.7766 - val_precision: 0.0691 - lr: 0.0010
Epoch 4/10
313/313 [==============================] - 107s 342ms/step - loss: 0.8102 - dice_coef: 0.1898 - recall: 0.5681 - precision: 0.1294 - val_loss: 0.9248 - val_dice_coef: 0.0752 - val_recall: 0.4374 - val_precision: 0.0532 - lr: 0.0010
Epoch 5/10
313/313 [==============================] - 107s 341ms/step - loss: 0.8242 - dice_coef: 0.1758 - recall: 0.5628 - precision: 0.1219 - val_loss: 0.9023 - val_dice_coef: 0.0977 - val_recall: 0.5651 - val_precision: 0.1126 - lr: 0.0010
Epoch 6/10
313/313 [==============================] - 106s 339ms/step - loss: 0.8216 - dice_coef: 0.1784 - recall: 0.4750 - precision: 0.1289 - val_loss: 0.9022 - val_dice_coef: 0.0978 - val_recall: 0.8239 - val_precision: 0.0813 - lr: 0.0010
Epoch 7/10
313/313 [==============================] - 143s 456ms/step - loss: 0.8043 - dice_coef: 0.1957 - recall: 0.6010 - precision: 0.1374 - val_loss: 0.8908 - val_dice_coef: 0.1092 - val_recall: 0.7709 - val_precision: 0.0930 - lr: 1.0000e-04
Epoch 8/10
313/313 [==============================] - 143s 456ms/step - loss: 0.7853 - dice_coef: 0.2147 - recall: 0.5833 - precision: 0.1537 - val_loss: 0.8892 - val_dice_coef: 0.1108 - val_recall: 0.6989 - val_precision: 0.1132 - lr: 1.0000e-04
Epoch 9/10
313/313 [==============================] - 107s 341ms/step - loss: 0.7876 - dice_coef: 0.2124 - recall: 0.5482 - precision: 0.1607 - val_loss: 0.8861 - val_dice_coef: 0.1139 - val_recall: 0.7264 - val_precision: 0.1080 - lr: 1.0000e-04
Epoch 10/10
313/313 [==============================] - 107s 341ms/step - loss: 0.7783 - dice_coef: 0.2217 - recall: 0.5673 - precision: 0.1634 - val_loss: 0.8889 - val_dice_coef: 0.1111 - val_recall: 0.7304 - val_precision: 0.0968 - lr: 1.0000e-04
```

```
## PLot Matrix between training and validation data
plt.figure(figsize=(8, 5))
plt.grid(True)
plt.plot(history_Unet.history['dice_coef'],     label='Train Dice-Coef',      color = "green" );
plt.plot(history_Unet.history['val_dice_coef'], label='Val Dice-Coef',  color = "yellow");
plt.plot(history_Unet.history['loss'],         label='Train Loss',          color = "red"    );
plt.plot(history_Unet.history['val_loss'],     label='Val Loss',      color = "orange");
plt.title("Validation and Training - Loss and Dice Coefficient vs Epoch")
plt.xlabel("Epoch")
plt.legend();
```



val_dice-oefficient value is very low and pretty much a flat curve, indicating underfitting indicating model has not learnt sufficiently. It is steadily increasing, not sufficient training(more epochs needed).We have used an image size of 224x224 as against the original size of 1024x1024. Using a higher resolution, could also improve training capacity. Hyper-parameter tuning, image_augmentation, using different architectures will help in increasing model performance and generalization.

```
##Preparing test data , picked up random 20 images
test_CombinedData = labels[15000:15020]
test_CombinedData.fillna(0, inplace=True)
```

```
## Check target distrubution in test dataset, there are both the classes available with equal sdistrubution
test_CombinedData.Target.value_counts()
```

```
0    10
1    10
Name: Target, dtype: int64
```

```
## Setting the custom generator for test data
testUNetDataGen = TrainGenerator(test_CombinedData)
```

```
## evaluating the model
test_steps = (len(testUNetDataGen)//BATCH_SIZE)
if len(testUNetDataGen) % BATCH_SIZE != 0:
    test_steps += 1

model_Unet.evaluate(testUNetDataGen)
```

```
5/5 [==============================] - 1s 161ms/step - loss: 0.6297 - dice_coef: 0.3703 - recall_1: 0.5711 - precision_1: 0.3205
[0.629738450050354,
 0.3702615201473236,
 0.5711286067962646,
 0.32047128677368164]
```

Model evaluation , dice coef = 37% , recall is less at 57% , precission is very low , of about 32%

```
## Precidt the test data that we have
pred_mask = model_Unet.predict(testUNetDataGen)
```

```
5/5 [==============================] - 1s 150ms/step
```

+ Code + Markdown

```
test_CombinedData = test_CombinedData.reset_index()
```

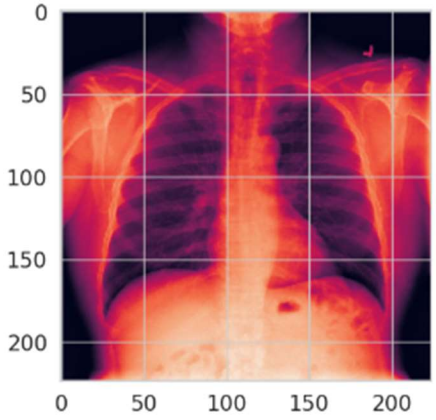
Create functions for getting predicrions and confusion matrix report

```
from sklearn.metrics import confusion_matrix
y_pred = []
y_True = []
imageList = []
predMaskTemp = []
IMAGE_HEIGHT = 224
IMAGE_WIDTH = 224
def getPredictions(test_CombinedData):
    masks = np.zeros((int(test_CombinedData.shape[0]), IMAGE_HEIGHT, IMAGE_WIDTH))
    for index, row in test_CombinedData.iterrows():
        patientId = row.patientId
#         print(patientId)

        classlabel = row["Target"]
        dcm_file = '/kaggle/input/pneumonia-detection/stage_2_train_images/stage_2_train_images/'+'{}.dcm'.format(patientId)
        dcm_data = pydicom.dcmread(dcm_file)
        img = dcm_data.pixel_array
        resized_img = cv2.resize(img,(IMAGE_HEIGHT,IMAGE_WIDTH), interpolation = cv2.INTER_LINEAR)
        predMaskTemp.append(pred_mask[index])
        iou = (pred_mask[index] > 0.5) * 1.0
        y_pred.append((1 in iou) * 1)
        imageList.append(resized_img)
        y_True.append(classlabel)
        x_scale = IMAGE_HEIGHT / 1024
        y_scale = IMAGE_WIDTH / 1024

        if(classlabel == 1):
            x = int(np.round(row['x'] * x_scale))
            y = int(np.round(row['y'] * y_scale))
            w = int(np.round(row['width'] * x_scale))
            h = int(np.round(row['height'] * y_scale))
            masks[index][y:y+h, x:x+w] = 1
    tmpImages = np.array(imageList)
    tmpMask = np.array(predMaskTemp)
    originalMask = np.array(masks)
    return (y_True,y_pred,tmpImages,tmpMask ,originalMask)

def print_confusion_matrix(y_true, y_pred):
    '''Function to print confusion_matrix'''
    # Get confusion matrix array
    array = confusion_matrix(y_true, y_pred)
    df_cm = pd.DataFrame(array, range(2), range(2))
    print("Total samples = ", len(test_CombinedData))
    # Plot heatmap and get sns heatmap values
    sns.set(font_scale=1.4); # for label size
    result = sns.heatmap(df_cm, annot=True, annot_kws={"size": 16}, fmt='g', cbar=False);
    # Add labels to heatmap
    labels = ['TN=','FP=','FN=','TP=']
    i=0
    for t in result.texts:
        t.set_text(labels[i] + t.get_text())
        i += 1
    plt.xlabel("Predicted Values")
    plt.ylabel('True Values')
    plt.show()
    return
```

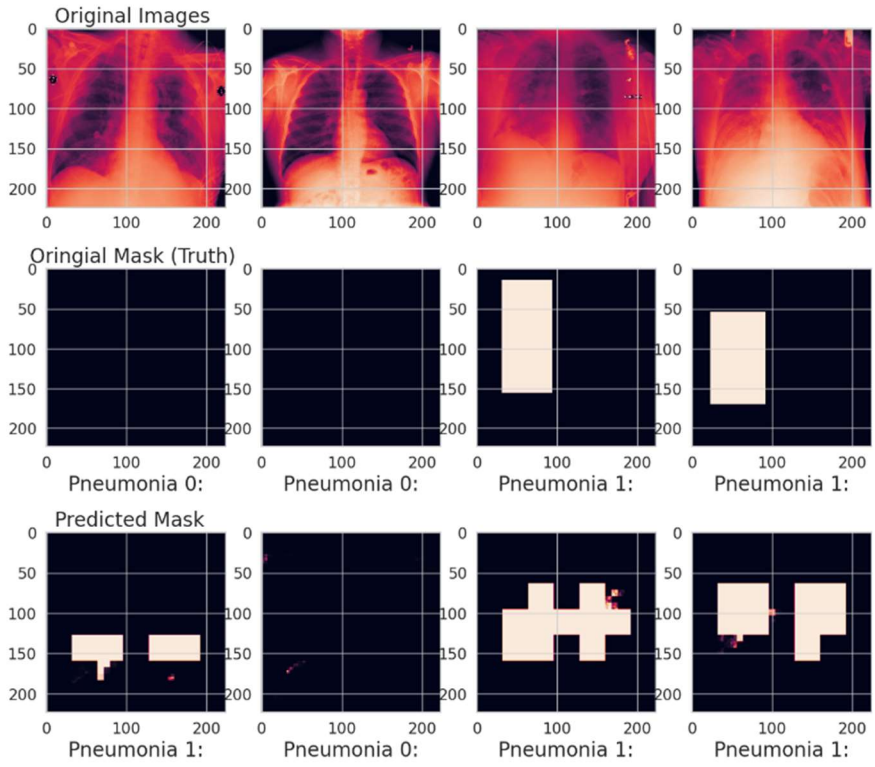
```
## Create predictions map
y_true,y_pred ,imagelist , maskList , originalMask = getPredictions(test_CombinedData)
```

+ Code + Markdown

```
plt.imshow(imagelist[12]);
```



```
## Visualising the train and output data

fig = plt.figure(figsize=(15, 15))
a = fig.add_subplot(1, 4, 1)
imgplot = plt.imshow(imagelist[1])
a.set_title('Original Images ',fontsize=20)

a = fig.add_subplot(1, 4, 2)
imgplot = plt.imshow(imagelist[12])

a = fig.add_subplot(1, 4, 3)
imgplot = plt.imshow(imagelist[13])

a = fig.add_subplot(1, 4, 4)
imgplot = plt.imshow(imagelist[15])

fig = plt.figure(figsize=(15, 15))
a = fig.add_subplot(1, 4, 1)
imgplot = plt.imshow(originalMask[1])
a.set_title('Oringial Mask (Truth) ',fontsize=20)

a.set_xlabel('Pneumonia {}:'.format(y_true[1]), fontsize=20)

a = fig.add_subplot(1, 4, 2)
imgplot = plt.imshow(originalMask[12])
a.set_xlabel('Pneumonia {}:'.format(y_true[12]), fontsize=20)

a = fig.add_subplot(1, 4, 3)
imgplot = plt.imshow(originalMask[13])
a.set_xlabel('Pneumonia {}:'.format(y_true[13]), fontsize=20)

a = fig.add_subplot(1, 4, 4)
imgplot = plt.imshow(originalMask[15])
a.set_xlabel('Pneumonia {}:'.format(y_true[15]), fontsize=20)

fig = plt.figure(figsize=(15, 15))
a = fig.add_subplot(1, 4, 1)
a.set_title('Predicted Mask  ',fontsize=20)
imgplot = plt.imshow(maskList[1])
a.set_xlabel('Pneumonia {}:'.format(y_pred[1]), fontsize=20)

a = fig.add_subplot(1, 4, 2)
imgplot = plt.imshow(maskList[12])
a.set_xlabel('Pneumonia {}:'.format(y_pred[12]), fontsize=20)

a = fig.add_subplot(1, 4, 3)
imgplot = plt.imshow(maskList[13])
a.set_xlabel('Pneumonia {}:'.format(y_pred[13]), fontsize=20)

a = fig.add_subplot(1, 4, 4)
imgplot = plt.imshow(maskList[15])
a.set_xlabel('Pneumonia {}:'.format(y_pred[15]), fontsize=20)
```



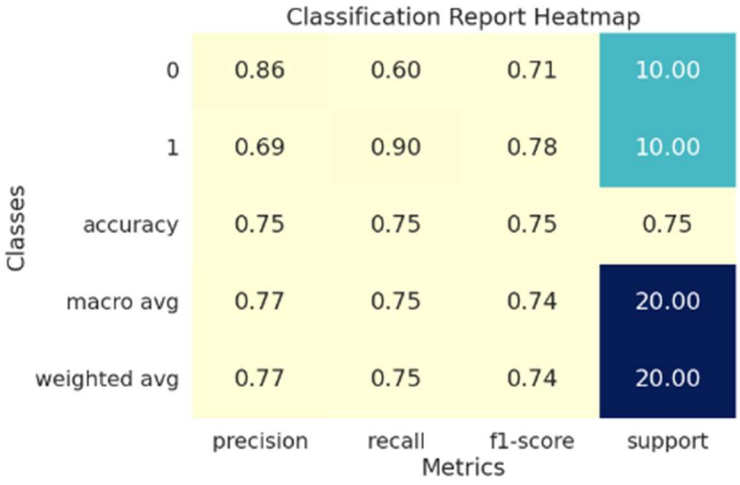
we could see that the first one is mis-classifed

Second one is classified correctly , there are no masks

Thrid is predcited correctly and the bounding box is also almost at the same position

fourth is predicted correctly but there are two boudning boxes, this culd be because there are duplicate patient id and we are picking only one mask to display inte Truth

[1]:
```
def print_classification_report_heatmap(y_true, y_pred):
    # Get the classification report as a dictionary
    report_dict = classification_report(y_true, y_pred, output_dict=True)

    # Convert the dictionary to a DataFrame
    report_df = pd.DataFrame(report_dict).transpose()

    # Plot the heatmap using seaborn
    sns.set(font_scale=1.4)  # for label size
    plt.figure(figsize=(7, 5))
    result = sns.heatmap(report_df, annot=True, cmap="YlGnBu", fmt='.2f', cbar=False)

    # Add labels to the heatmap
    plt.xlabel("Metrics")
    plt.ylabel('Classes')
    plt.title("Classification Report Heatmap")

    # Show the plot
    plt.show()

print_classification_report_heatmap(y_true, y_pred)
```



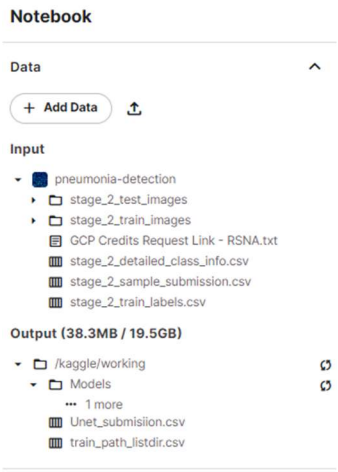
4

4: Pickle the model for future prediction

Pickle the model and save it

```
import pickle

# DDirectory path to save the model
save_dir = "/kaggle/working/Models/"

# Create the directory if it doesn't exist
import os
os.makedirs(save_dir, exist_ok=True)

## Save the model to the directory
model_path = os.path.join(save_dir, "model_Unet.pkl")
with open(model_path, "wb") as f:
    pickle.dump(model_Unet, f)

print(f"Model saved to {model_path}")
```

```
Model saved to /kaggle/working/Models/model_Unet.pkl
```



Save ‘csv’ file for predicted 20 images.

```
# Create a DataFrame to hold the submission data
submission_df = pd.DataFrame({
    'patientId': test_CombinedData['patientId'],
    'Prediction': y_pred
})

# Export the DataFrame to a CSV file
submission_filename = 'Unet_submisiion.csv'
submission_df.to_csv(submission_filename, index=False)

print(f'CSV file "{submission_filename}" created.')
```

```
CSV file "Unet_submisiion.csv" created.
```

3: Design, train and test RCNN & its hybrids-based object detection models

Faster RCNN

Import Data and labels and verify the number of records

```
seed = 42
num_classes = 2
batch_size = 10
train_img_size = 256
origin_img_size = 1024
scale_factor = train_img_size / origin_img_size
np.random.seed(seed)

TRAIN_DIR = "Train images/stage_2_train_images/"
TEST_DIR = "Test images/stage_2_test_images/"
ROOT_DIR = "/content/drive/My Drive/Colab Notebooks/My Python Projects/CAPSTONE PROJECT/"
LABELS_FILE = "CV capstone/stage_2_train_labels.csv"
SUBMISSION_FILE = "CV capstone/stage_2_sample_submission.csv"
rcnn_losses = ["loss_objectness", "loss_box_reg", "loss_rpn_box_reg"]
```

```
train_imgs = os.listdir(os.path.join(ROOT_DIR, TRAIN_DIR))
test_imgs =  [patienId + ".dcm" for patienId in pd.read_csv(os.path.join(ROOT_DIR, SUBMISSION_FILE)).patientId]
```

```
from sklearn.model_selection import train_test_split

train_imgs, valid_imgs = train_test_split(train_imgs, test_size=0.25, random_state=seed)

print(f"Number of training samples: {len(train_imgs)}")
print(f"Number of validation samples: {len(valid_imgs)}")
```

```
Number of training samples: 20013
Number of validation samples: 6671
```

```
train_label_df = pd.read_csv(os.path.join(ROOT_DIR, LABELS_FILE))
train_label_df.head()
```

	patientId	x	y	width	height	Target
0	0004cfab-14fd-4e49-80ba-63a80b6bddd6	NaN	NaN	NaN	NaN	0
1	00313ee0-9eaa-42f4-b0ab-c148ed3241cd	NaN	NaN	NaN	NaN	0
2	00322d4d-1c29-4943-afc9-b6754be640eb	NaN	NaN	NaN	NaN	0
3	003d8fa0-6bf1-40ed-b54c-ac657f8495c5	NaN	NaN	NaN	NaN	0
4	00436515-870c-4b36-a041-de91049b9ab4	264.0	152.0	213.0	379.0	1

Our model require box coordinates in format X0, X1. We also multiply bounding box coordinates with scale_factor since we will train our model on images of size 256 instead of their original size 1024. R-CNN also require area of bounding box as input.

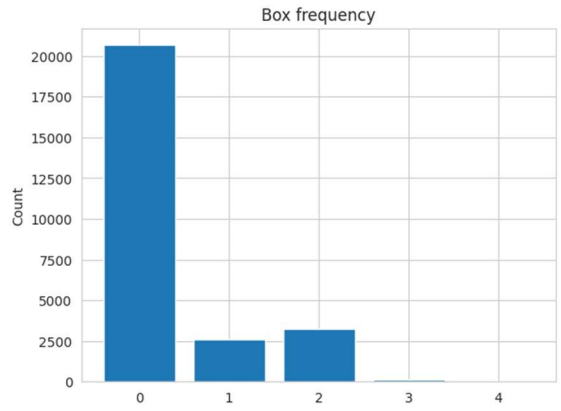
Preprocessing of ‘traindf’ dataframe:

```
isna_count = len(train_label_df[train_label_df.Target == 0]) # number of images without bounding box
train_label_df = train_label_df[train_label_df.Target == 1]
train_label_df.rename(columns={"x": "X0", "y": "Y0"}, inplace=True)
train_label_df["X1"] = train_label_df["X0"] + train_label_df["width"]
train_label_df["Y1"] = train_label_df["Y0"] + train_label_df["height"]
train_label_df[["X0", "X1", "Y0", "Y1"]] = train_label_df[["X0", "X1", "Y0", "Y1"]] * scale_factor
train_label_df["area"] = train_label_df["width"] * scale_factor * train_label_df["height"] * scale_factor
train_label_df.drop(["width", "height"], axis=1, inplace=True)
train_label_df.head()
```

	patientId	X0	Y0	Target	X1	Y1	area
4	00436515-870c-4b36-a041-de91049b9ab4	66.00	38.00	1	119.25	132.75	5045.4375
5	00436515-870c-4b36-a041-de91049b9ab4	140.50	38.00	1	204.50	151.25	7248.0000
8	00704310-78a8-4b38-8475-49f4573b2dbb	80.75	144.25	1	120.75	170.25	1040.0000
9	00704310-78a8-4b38-8475-49f4573b2dbb	173.75	143.75	1	214.25	178.00	1387.1250
14	00aecb01-a116-45a2-956c-08d2fa55433f	72.00	80.50	1	95.50	114.25	793.1250

Now let´s check how many images have zero, one or more bounding boxes

```
from collections import Counter

cnt = Counter(train_label_df.patientId)
sample_batch = [sample[0] + ".dcm" for sample in cnt.most_common(2)] # We will take two imgs with 4 boxes to display
counts = pd.Series(cnt.values()).value_counts()
counts[0] = isna_count
plt.title("Box frequency")
plt.ylabel("Count")
plt.xticks(counts.index)
plt.bar(counts.index, counts)
plt.show()
del cnt
```



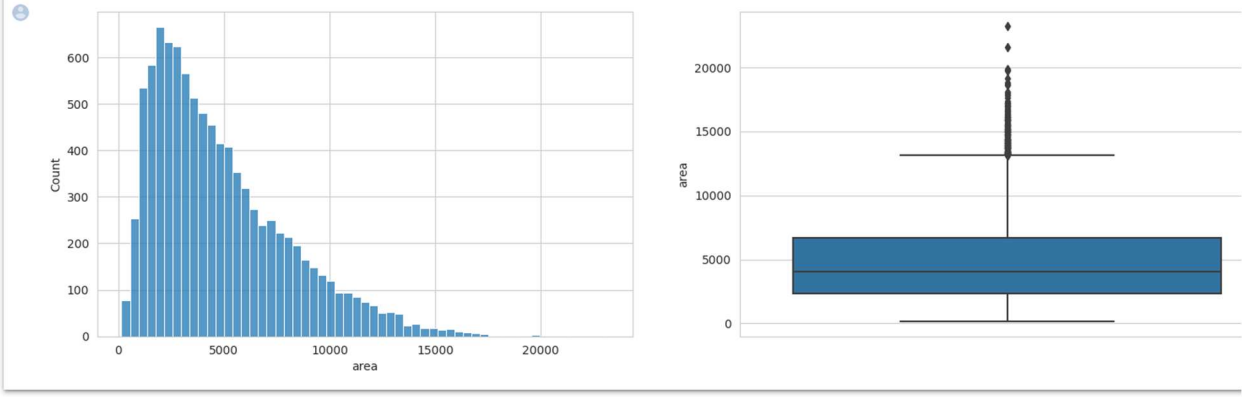

```
counts.sort_values(ascending=False)
```

```
0    20672
2     3266
1     2614
3      119
4       13
dtype: int64
```

We see that most of images do not have any bounding box. There is also a handful of images with three and four bounding boxes. Previously we calculated bounding boxes area so we can now display area distribution.

Let us see the distribution of data in a histogram plot.

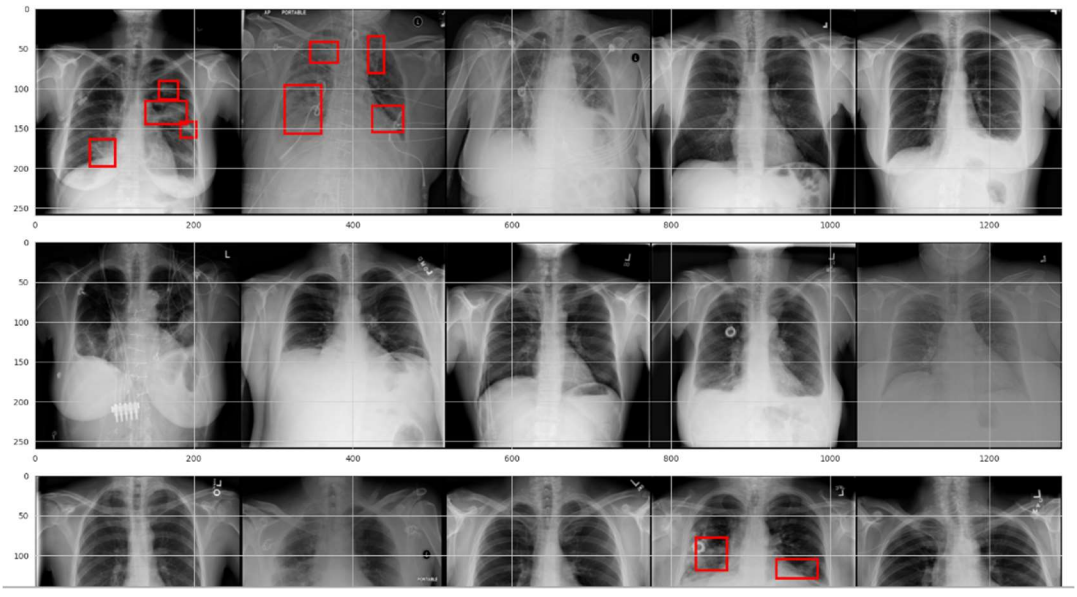
```
fig, axs = plt.subplots(1, 2, figsize=(18, 5))
sns.histplot(data=train_label_df, x="area", ax=axs[0])
sns.boxplot(y=train_label_df["area"])
plt.show()
```



We see that most of boxes have area smaller than 7000. There are also outliers with area greater than 15000. Now we will define some helper functions to display samples batch along with boxes.

Create functions to define several functions for reading and plotting image samples along with their bounding boxes

```
from torchvision.utils import draw_bounding_boxes, make_grid


def find_boxes(img_name, label_df):
    patient_id = img_name.split(".")[0]
    boxes = label_df[label_df.patientId == patient_id]
    boxes_coord = boxes[["X0", "Y0", "X1", "Y1"]].to_numpy()

    return boxes_coord, boxes.area.to_numpy()


def read_images(img_names, label_df, resize):
    batch = []
    for img_name in img_names:
        img_path = os.path.join(ROOT_DIR, TRAIN_DIR, img_name)
        img = PIL.Image.fromarray(pydicom.dcmread(img_path).pixel_array).convert("RGB")
        img = img.resize(resize)
        img = np.array(img)

        boxes, _ = find_boxes(img_name, label_df)

        img = torch.tensor(img, dtype=torch.uint8)
        boxes = torch.tensor(boxes, dtype=torch.int32)
        batch.append((img, boxes))
    return batch


def plot_samples(batch, n_rows=3, n_cols=5, box_color="red", titles=None, box_width=3, fig_size=(20,12)):
    assert len(batch) >= n_rows * n_cols, f"Not enough samples to display, required at least {n_rows * n_cols} samples"
    fig, axs = plt.subplots(n_rows, 1, figsize=fig_size)
    fig.tight_layout()
    imgs_with_boxes = []
    for sample in batch[:n_rows * n_cols]:
        scores = None
        if len(sample) == 2:
            image, boxes = sample
        else:
            image, boxes, scores = sample
            scores = [f"Score: {score:.2f}" for score in scores]

        img = draw_bounding_boxes(image=image.permute(2, 0, 1),
                                  boxes=boxes,
                                  colors=box_color,
                                  labels=scores,
                                  width=box_width)
        imgs_with_boxes.append(img)

    for i, ax in enumerate(axs):
        if titles:
            ax.set_title(titles[i], fontsize=18)
        img_with_boxes = make_grid(imgs_with_boxes[i * n_cols: (i + 1) * n_cols]).numpy()
        ax.imshow(np.transpose(img_with_boxes, (1, 2, 0)))
    plt.show()

sample_batch = np.concatenate([sample_batch, np.random.choice(train_imgs, size=13)])
sample_batch = read_images(sample_batch, train_label_df, (train_img_size, train_img_size))
plot_samples(sample_batch)
```



Some bounding boxes covers small part of the lung while others cover most of the lung.

Image augmentation:

Now we will use a nifty library for image augmentation imgaug. The library can perform geometric transformations along with bounding boxes, masks and key points. In addition we add some noice and blur to image.

Class called 'Augmentation' is used for applying image augmentations using the imgaug library. imgaug is a powerful library for image augmentation that supports a wide range of transformations.

```
# Define a class 'Augmentation' for applying image augmentations
class Augmentation:
    def __init__(self,
                 x_translation=(-0.05, 0.05),
                 rotate=(-5, 5),
                 scale=(0.9, 1.1),
                 noice=(0, 10),
                 blur=(0, 0.1)):

        # Define the augmentation transformations using imgaug.Sequential
        self.transform = iaa.Sequential([
            iaa.Affine(translate_percent={"x": x_translation}, scale=scale, rotate=rotate),
            iaa.AdditiveGaussianNoise(scale=noice),
            iaa.GaussianBlur(sigma=blur)])

    # Implement the __call__ method, which will be called when the class object is called as a function
    def __call__(self, image, boxes):
        # Convert the bounding boxes coordinates into imgaug format BoundingBoxesOnImage
        bbs = BoundingBoxesOnImage([BoundingBox(x1=box[0], y1=box[1], x2=box[2], y2=box[3])
                                    for box in boxes], shape=image.shape)

        # Apply the defined augmentations to the image and bounding boxes
        image, bbs = self.transform(image=image, bounding_boxes=bbs)

        # Convert the bounding boxes back to the original format (x1, y1, x2, y2) array
        return image, bbs.to_xyxy_array()

    # Worker initialization function for multi-processing (not used in this code snippet)
    def worker_init_fn(self, worker_id):
        """Set a unique seed for each worker to ensure different augmentations."""
        imgaug.seed(np.random.get_state()[1][0] + worker_id)
```

The code provided below defines a PyTorch dataset class called PneumoniaDataset, which is designed to handle loading images, applying transformations, and preparing target information (annotations) for training or inference.

```
from torchvision.transforms import PILToTensor, ConvertImageDtype

class PneumoniaDataset(torch.utils.data.Dataset):
    def __init__(self, root, transforms, img_names, train_img_size, labels=None):
        self.root = root
        self.transforms = transforms
        self.img_names = img_names
        self.resize = (train_img_size, train_img_size)
        self.labels = labels

    def __getitem__(self, idx):
        """Load image, boxes and process them"""
        img_name = self.img_names[idx]

        img_path = os.path.join(self.root, self.img_names[idx])
        img = PIL.Image.fromarray(pydicom.dcmread(img_path).pixel_array).convert("RGB")
        img = img.resize(self.resize)
        img = np.array(img)

        if self.labels is None:
            return torch.tensor(img / 255.).permute(2, 0, 1).float()

        boxes, area = find_boxes(img_name, self.labels)
        n_objects = len(boxes)
        if self.transforms is not None:
            img, boxes = self.transforms(img, boxes)

        img = torch.tensor(img / 255.).permute(2, 0, 1).float()

        target = {}
        target["image_id"] = torch.tensor([idx])
        target["boxes"] = torch.as_tensor(boxes, dtype=torch.float32)
        target["labels"] = torch.ones(n_objects, dtype=torch.int64)
        target["area"] = torch.as_tensor(area, dtype=torch.float32)
        target["iscrowd"] = torch.zeros(n_objects, dtype=torch.int32)
        return img, target

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

    def collate_fn(self, batch):
        return tuple(zip(*batch))
```

The code provided below sets up data loaders for training, validation, and testing using the defined PneumoniaDataset class and the Augmentation class.

```
from torch.utils.data import DataLoader

augmentation = Augmentation()
train_ds = PneumoniaDataset(os.path.join(ROOT_DIR, TRAIN_DIR), augmentation, train_imgs, train_img_size, train_label_df)
valid_ds = PneumoniaDataset(os.path.join(ROOT_DIR, TRAIN_DIR), None, valid_imgs, train_img_size, train_label_df)
test_ds = PneumoniaDataset(os.path.join(ROOT_DIR, TEST_DIR), None, test_imgs, train_img_size)

train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2, collate_fn=train_ds.collate_fn,
                          worker_init_fn=augmentation.worker_init_fn)
valid_loader = DataLoader(valid_ds, batch_size=batch_size, shuffle=False, num_workers=2, collate_fn=valid_ds.collate_fn)
test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)
```

Install & import imgaug library for image augmentation.

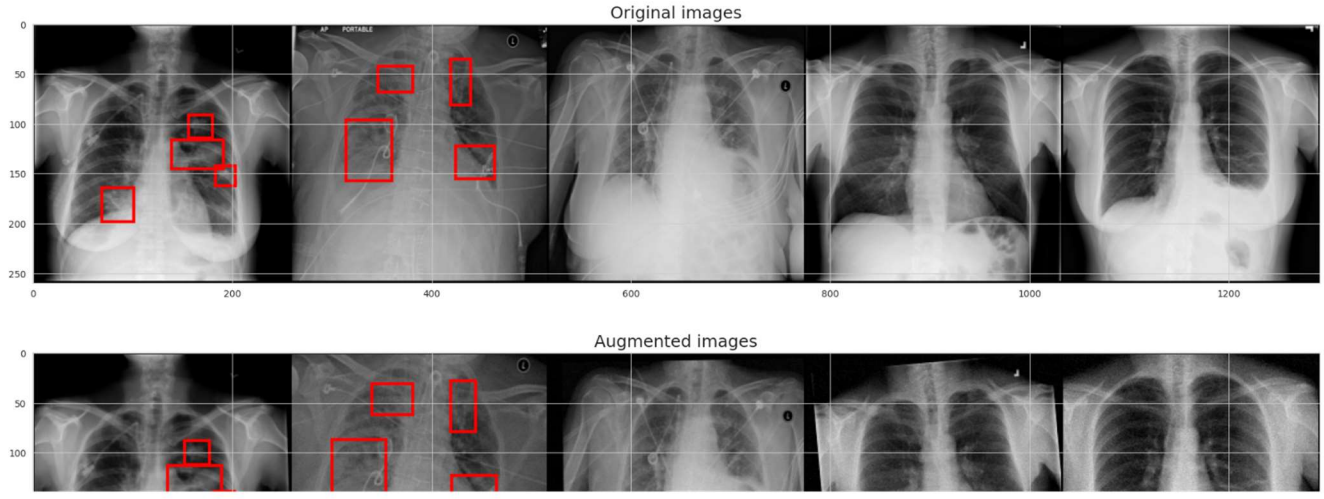
```
!pip install imgaug
```

```
Requirement already satisfied: imgaug in /usr/local/lib/python3.10/dist-packages (0.4.0)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from imgaug) (1.16.0)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.10/dist-packages (from imgaug) (1.22.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from imgaug) (1.10.1)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from imgaug) (9.4.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from imgaug) (3.7.1)
Requirement already satisfied: scikit-image>=0.14.2 in /usr/local/lib/python3.10/dist-packages (from imgaug) (0.19.3)
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/dist-packages (from imgaug) (4.7.0.72)
Requirement already satisfied: imageio in /usr/local/lib/python3.10/dist-packages (from imgaug) (2.25.1)
Requirement already satisfied: Shapely in /usr/local/lib/python3.10/dist-packages (from imgaug) (2.0.1)
Requirement already satisfied: networkx>=2.2 in /usr/local/lib/python3.10/dist-packages (from scikit-image>=0.14.2->imgaug) (3.1)
Requirement already satisfied: tifffile>=2019.7.26 in /usr/local/lib/python3.10/dist-packages (from scikit-image>=0.14.2->imgaug) (2023.7.18)
Requirement already satisfied: PyWavelets>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-image>=0.14.2->imgaug) (1.4.1)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from scikit-image>=0.14.2->imgaug) (23.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (1.1.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (4.41.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (1.4.4)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (3.1.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->imgaug) (2.8.2)
```

```
import imgaug
```

The code is used for applying augmentations to a batch of images using the Augmentation class and then plotting the original and augmented images side by side.

```
def process_batch(idx, sample):
    img, boxes = sample

    augmentation = Augmentation()
    augmentation.worker_init_fn(idx)
    img, boxes = augmentation(img.numpy(), boxes.numpy())
    img = torch.tensor(img, dtype=torch.uint8)
    boxes = torch.tensor(boxes, dtype=torch.float32)
    return img, boxes


augmented_batch = sample_batch[:5] + [process_batch(i, sample) for i, sample in enumerate(sample_batch[:5])]
plot_samples(augmented_batch, titles=["Original images", "Augmented images"], fig_size=(20, 10), n_rows=2)
del sample_batch, augmented_batch
```



Define a PyTorch Lightning module for object detection using Faster R-CNN

```
import pytorch_lightning as pl
from torchvision.ops import nms
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

class LitRCNN(pl.LightningModule):
    def __init__(self, num_classes):
        super().__init__()
        model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
        in_features = model.roi_heads.box_predictor.cls_score.in_features
        model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
        self.model = model
    def forward(self, x):
        self.model.eval()
        outputs =  self.model(x)
        preds = []
        for output in outputs:
            boxes = output["boxes"]
            scores = output["scores"]
            idx = nms(boxes,scores, 0.05)
            preds.append({"boxes": boxes[idx].cpu().detach().numpy(), "scores": scores[idx].cpu().detach().numpy()})
        return preds
    def training_step(self, batch, batch_idx):
        images, targets = batch
        losses = self.model(images, targets)
        loss = sum(loss for loss in losses.values())
        self.log_losses(loss, losses)
        return loss
    def validation_step(self, batch, batch_idx):
        self.model.train()
        images, targets = batch
        losses = self.model(images, targets)
        loss = sum(loss for loss in losses.values())
        self.log_losses(loss, losses, mode="val")
    def predict_step(self, batch, batch_idx):
        return self.forward(batch)
    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(),  lr=0.005, momentum=0.9, weight_decay=0.0005)
        scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.3)
        return {
            "optimizer": optimizer,
            "lr_scheduler": {
                "scheduler": scheduler,
                "interval": "epoch",
                "frequency": 1}}

    def log_losses(self, loss, losses, mode="train"):
        self.log_dict({f"{mode}_loss": loss,
                       f"{mode}_loss_box_reg": losses['loss_box_reg'],
                       f"{mode}_loss_objectness": losses['loss_objectness'],
                       f"{mode}_loss_rpn_box_reg": losses['loss_rpn_box_reg']}, on_step=False, on_epoch=True, batch_size=batch_size)

model = LitRCNN(num_classes)
```

```
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.cache/torch/hub/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
100%|██████████| 160M/160M [00:02<00:00, 77.8MB/s]
```

We define some callbacks for logging and model checkpoints.

```
from pytorch_lightning import Callback
from pytorch_lightning.callbacks import ModelCheckpoint


class MetricsCallback(Callback):
    """PyTorch Lightning metric callback."""

    def __init__(self, metrics):
        super().__init__()
        self.metrics = metrics
        self.training = {}
        self.validations = {}

    def on_train_epoch_end(self, trainer, pl_module):
        self.training[trainer.current_epoch] = {metric: trainer.callback_metrics["train_" + metric] for metric in self.metrics}

    def on_validation_end(self, trainer, pl_module):
        self.validations[trainer.current_epoch] = {metric: trainer.callback_metrics["val_" + metric] for metric in self.metrics}


checkpoint_callback = ModelCheckpoint(dirpath='checkpoints',
                                      filename='{epoch}-{val_loss:.4f}',
                                      every_n_epochs=1,
                                      monitor='val_loss',
                                      save_top_k=1,
                                      mode='min')

callbacks = [MetricsCallback(["loss"] + rcnn_losses), checkpoint_callback]
```

Training the Model:

```
device = "gpu" if torch.cuda.is_available() else "cpu"
trainer = pl.Trainer(accelerator=device, max_epochs=1, callbacks=callbacks)
trainer.fit(model=model, train_dataloaders=train_loader, val_dataloaders=valid_loader)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
WARNING:pytorch_lightning.loggers.tensorboard:Missing logger folder: /content/lightning_logs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type       | Params
-------------------------------------
0 | model | FasterRCNN | 41.3 M
-------------------------------------
41.1 M    Trainable params
222 K     Non-trainable params
41.3 M    Total params
165.197   Total estimated model params size (MB)
```

Epoch 0: 100%

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=1` reached.
```

Evaluation:

Now, we an check some prediction on validation set.

```
[ ] sample_batch_idx = np.random.choice(len(valid_imgs), size=5)
    sample_batch = read_images(np.array(valid_ds.img_names)[sample_batch_idx], train_label_df, (train_img_size, train_img_size))

    preds = model([valid_ds[i][0] for i in sample_batch_idx])
    for i, pred in enumerate(preds):
        sample_batch.append((sample_batch[i][0],
                             torch.tensor(pred["boxes"], dtype=torch.int32),
                             pred["scores"]))

    plot_samples(sample_batch, titles=["Validation", "Predictions"], fig_size=(20, 10), n_rows=2)
```

```
[ ]
    sns.set_style("whitegrid")
    training = pd.DataFrame.from_dict(callbacks[0].training, orient="index").applymap(lambda x: x.cpu().numpy())
    validations = pd.DataFrame.from_dict(callbacks[0].validations, orient="index").applymap(lambda x: x.cpu().numpy())

    fig, axs = plt.subplots(2, 2, figsize=(16, 8))

    for ax, metric in zip(axs.flat, callbacks[0].metrics):
        ax.set_title(metric)

        # Check if the metric key exists in the DataFrames
        if metric in training.index and metric in validations.index:
            # Extract the numeric values for the specific metric
            training_metric_values = training.loc[metric].values
            validations_metric_values = validations.loc[metric].values

            # Use the numeric values for the lineplot
            g = sns.lineplot(data=training_metric_values, ax=ax)
            g = sns.lineplot(data=validations_metric_values, ax=ax)
            g.set(xlabel="Epoch", ylabel=None)
            g.set_xticks(range(1, len(training_metric_values) + 1))  # Assuming epoch numbers start from 1
            ax.legend(labels=["Training", "Validation"])
        else:
            print(f"Warning: Metric '{metric}' not found in the DataFrames.")

    fig.tight_layout()
    plt.show()
```

```
Warning: Metric 'loss' not found in the DataFrames.
Warning: Metric 'loss_objectness' not found in the DataFrames.
Warning: Metric 'loss_box_reg' not found in the DataFrames.
Warning: Metric 'loss_rpn_box_reg' not found in the DataFrames.
```

Validation parameters:

```
[ ] history = trainer.validate(model, dataloaders=valid_loader, verbose=False)
    pd.DataFrame(history, index=["Validation"]).T
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Validation DataLoader 0: 100%
```

	Validation
val_loss	0.073671
val_loss_box_reg	0.034431
val_loss_objectness	0.004091
val_loss_rpn_box_reg	0.001276

Submission of Prediction ‘csv’ file

```
[ ] preds = trainer.predict(model, test_loader)

    outputs = []
    for batch_pred in preds:
        for sample_pred in batch_pred:
            scores, boxes = sample_pred["scores"], sample_pred["boxes"]
            if len(scores) == 0:
                outputs.append(np.nan)
            else:
                label = ""
                boxes = boxes / scale_factor
                for score, box in zip(scores, boxes):
                    label += f"{score:.2f} {box[0]:.1f} {box[1]:.1f} {(box[2]-box[0]):.1f} {(box[3]-box[1]):.1f} "
                outputs.append(label.strip())

    submission = pd.read_csv(os.path.join(ROOT_DIR, SUBMISSION_FILE))
    submission.PredictionString = outputs
    submission.to_csv("submission.csv", header=True, index=False)
    submission
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Predicting DataLoader 0: 100%
```

	patientId	PredictionString
0	0000a175-0e68-4ca4-b1af-167204a7e0bc	NaN
1	0005d3cc-3c3f-40b9-93c3-46231c3eb813	0.25 314.2 203.2 130.3 179.1 0.11 660.3 479.5 ...
2	000686d7-f4fc-448d-97a0-44fa9c5d3aa6	0.08 179.2 495.9 274.8 258.8 0.06 758.6 570.1 ...
3	000e3a7d-c0ca-4349-bb26-5af2d8993c3d	0.11 230.5 407.1 223.1 313.8
4	00100a24-854d-423d-a092-edcf6179e061	0.57 251.1 599.2 148.0 193.5 0.56 515.2 345.1 ...
...	...	...
2995	c1e88810-9e4e-4f39-9306-8d314bfc1ff1	0.42 338.6 146.0 191.3 382.8 0.29 621.2 586.1 ...
2996	c1ec035b-377b-416c-a281-f868b7c9b6c3	0.29 253.3 580.6 203.3 191.0

SCORE FOR MODEL IMPROVEMENTS

- The models trained on sampled data. Increasing the size of sample could help in providing more variation in data and improving the model.
- Models can be further explored to be trained with Chesnet,YOLO, EfficientNet, etc and other similar models to get better accuracy.
- Image size reduction, more augmentation including rotation of the images at differert angles, moving pixel positions, flipping, etc.. could be done in training.
- Reduction in size of the images could help in improving memory utilization.