datascienist98
Apr 30 · 13 min read · Listen

Detecting the presence of Pneumonia using Machine Learning

In this post, we will explore the machine learning algorithms used to classify patients based on the presence of pneumonia using their chest X-rays.

Pneumonia is a lung infection that ranges from mild to life-threatening consequences. In order to identify the infection, radiologists examine chest X-ray images to identify possible infection. By analyzing the "Chest X-Ray Images (Pneumonia)" dataset on Kaggle, we aim to use machine learning algorithms to help identify the presence of Pneumonia in patients and to facilitate diagnosis at a larger scale.

The dataset consists of roughly 6,000 chest x-rays taken from pediatric patients in Guangzhou, China with each x-ray labeled "Pneumonia" or "Normal" as identified by two expert physicians. The types of models we considered include Logistic Regression, Support Vector Machines and Convolutional Neural Networks. In addition, we aimed to incorporate some feature engineering to augment these models.

We start by making the necessary imports:

```
import pandas as pd
import numpy as np

from torchvision import datasets, transforms
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
import cv2
import os
import pathlib

from skimage.io import imread
from skimage.transform import resize
from skimage.color import rgb2gray

import matplotlib.pyplot as plt
import seaborn as sns
import random
from sklearn.utils import resample

# setting device to GPU if available (for the neural networks)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

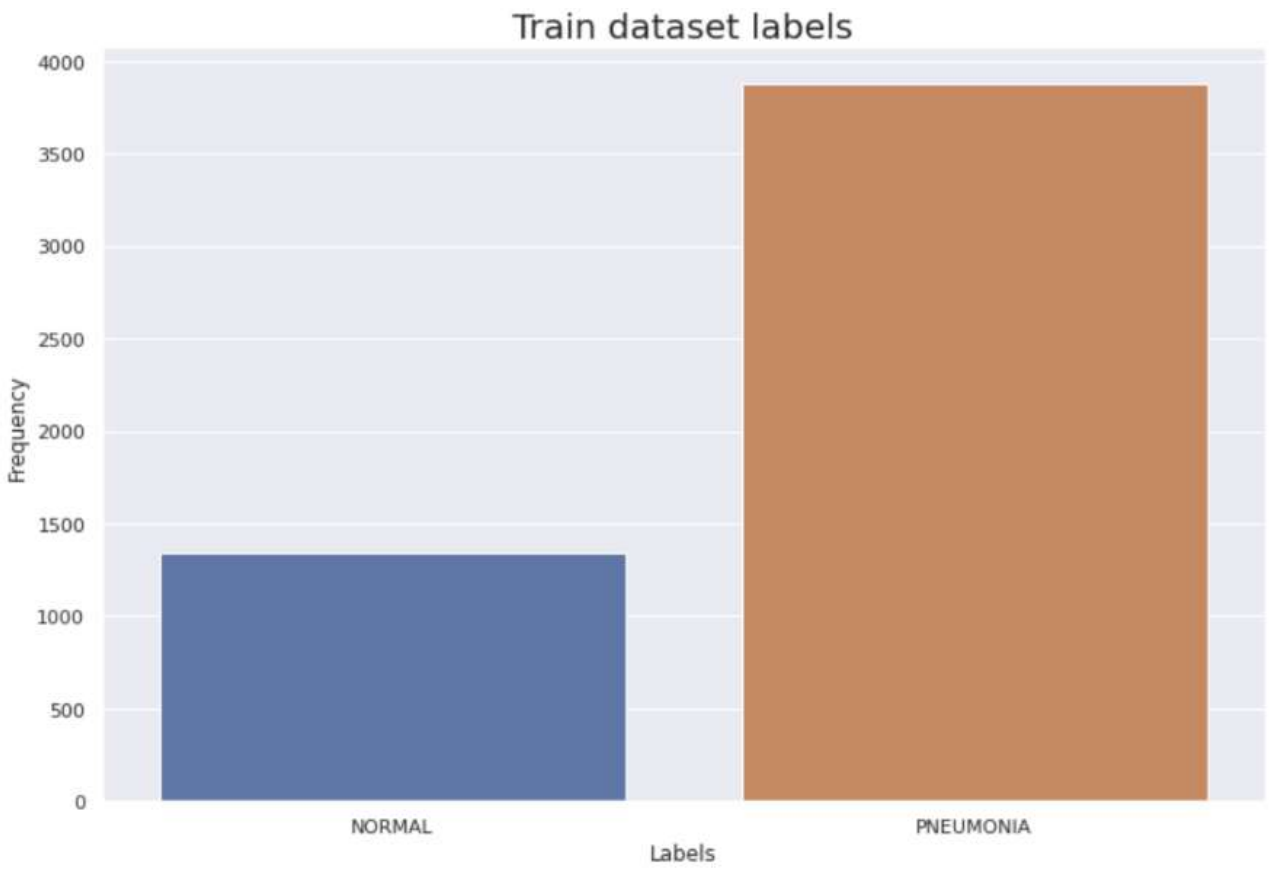
Then, let's extract the dataset from Kaggle:

```
! pip install kaggle
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
!kaggle datasets download -d paultimothymooney/chest-xray-pneumonia
! unzip /content/chest-xray-pneumonia.zip
```

This will create a zip file containing the training, validation and testing images present in the dataset.

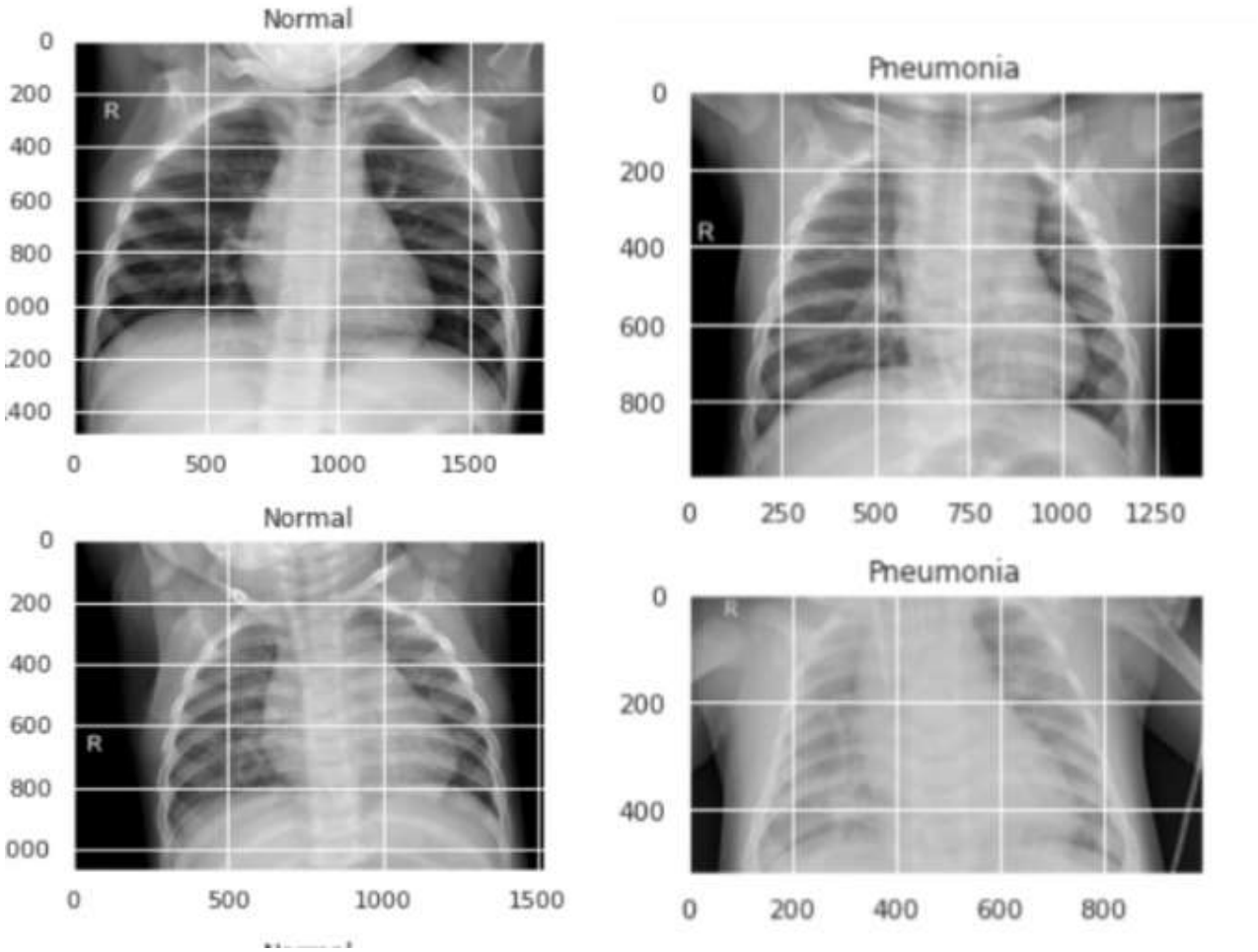
As part of the exploratory data analysis, we looked at the makeup of the training, validation, and testing datasets, including the labels and frequency of each. The training dataset had 1,341 normal images and 3,875 pneumonia images; the validation dataset had 8 normal images and 8 pneumonia images; and the testing dataset had 234 normal images and 390 pneumonia images. Shown below is a visualization of the training dataset distribution.



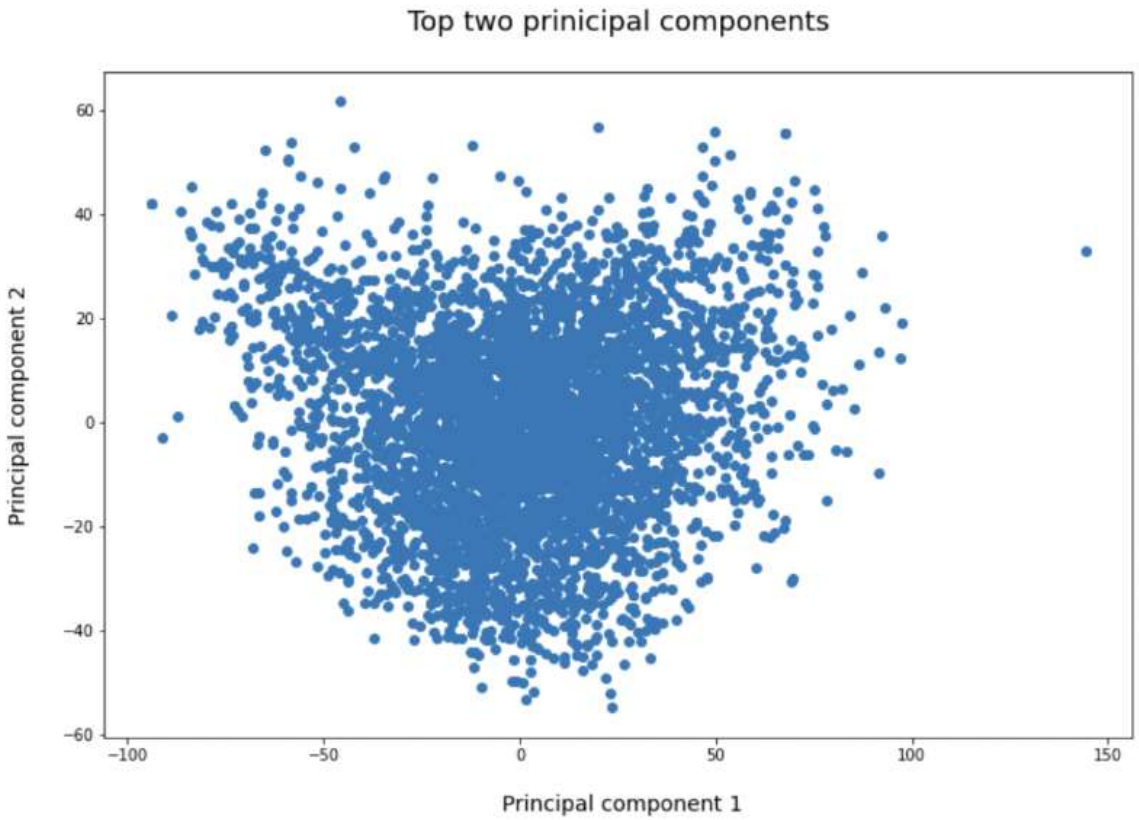
The distribution of labels in the training set

We then visualized some x-ray images from the training dataset with their corresponding labels using plt.imshow().



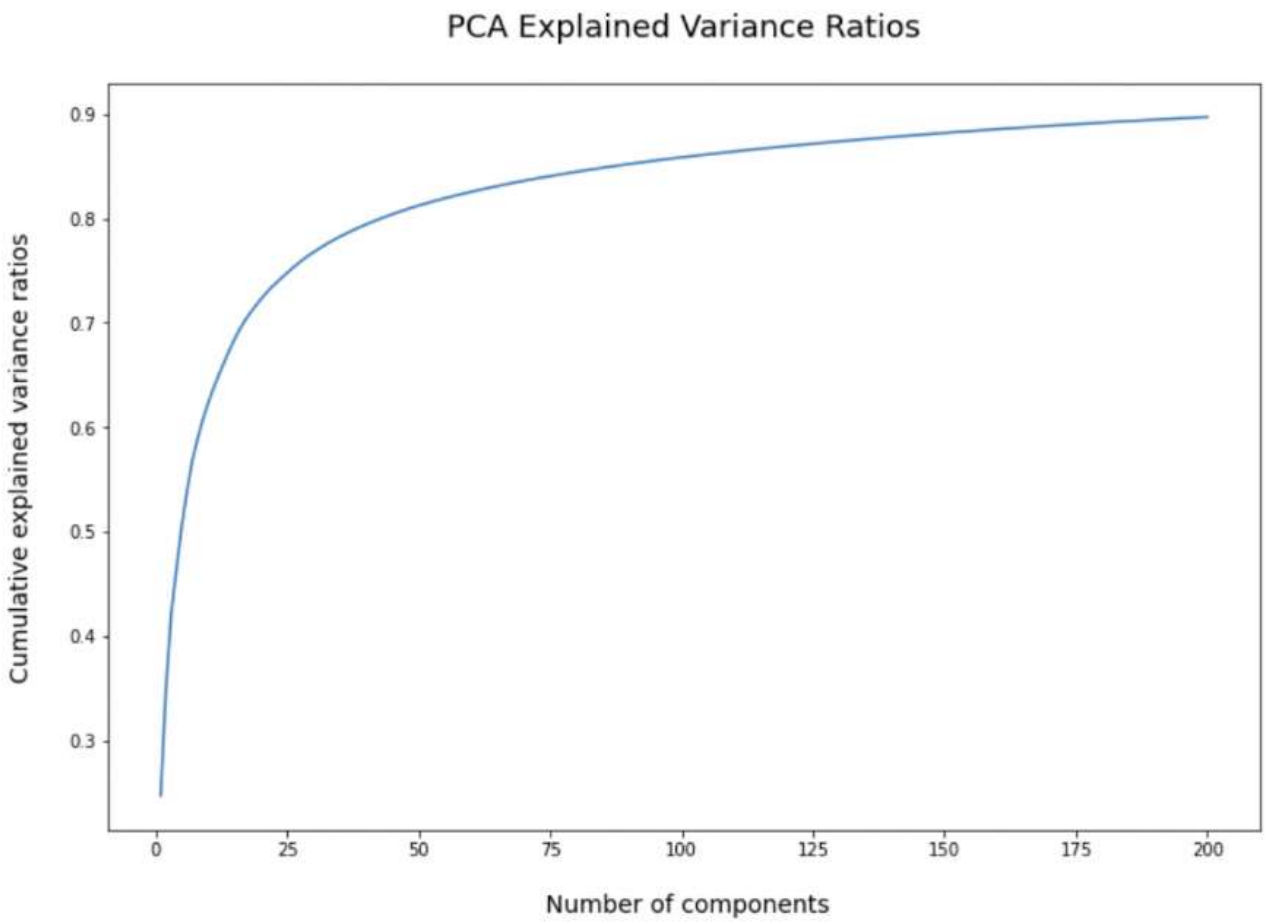
Visualizing the X-rays and their respective labels

After EDA, we conducted Principal Component Analysis to reduce the number of dimensions of our images while keeping as much variability in the original data as possible. First, we visualized two components of PCA from scikit-learn when applied to the training data as shown below.



Principal Component Analysis

Then, we conducted an initial PCA on our training dataset and found the explained variance ratios and cumulative explained variance ratios of the PCA. Shown below is a visualization of the cumulative explained variance ratios plotted against the number of components. Using 80% as the threshold for total variance in our dataset and noticing that this is the point at which the curve flattens out, we concluded that 50 components made the most sense for our training dataset.



Plot of explained variance ratios to determine the optimum number of components

Let's start with the algorithms -

We have two categories or labels in our dataset, "NORMAL" for patients with X-rays not having Pneumonia and "PNEUMONIA" for patients with X-rays having Pneumonia. So we can initialize a list to store our labels.

```
categories = ['NORMAL', 'PNEUMONIA']
```

Next, its time to extract the images from the zip file we created:

```
val_pneumonia_dir= base_dir+'val/PNEUMONIA/'
val_normal_dir= base_dir+'val/NORMAL/'
train_pn = [train_pneumonia_dir+"{}".format(i) for i in
os.listdir(train_pneumonia_dir) ]
train_normal = [train_normal_dir+"{}".format(i) for i in
os.listdir(train_normal_dir) ]
test_normal = [test_normal_dir+"{}".format(i) for i in
os.listdir(test_normal_dir)]
test_pn = [test_pneumonia_dir+"{}".format(i) for i in
os.listdir(test_pneumonia_dir)]
val_pn= [val_pneumonia_dir+"{}".format(i) for i in
os.listdir(val_pneumonia_dir) ]
val_normal= [val_normal_dir+"{}".format(i) for i in
os.listdir(val_normal_dir) ]
print ("Total
images:",len(train_pn+train_normal+test_normal+test_pn+val_pn+val_norm
al))
print ("Total pneumonia images:",len(train_pn+test_pn+val_pn))
print ("Total Normal
images:",len(train_normal+test_normal+val_normal))
```

We can now see that we have a total of 5856 images out of which there are 4273 images of X-rays with Pneumonia and 1583 X-rays without Pneumonia. So there is a huge class imbalance.

Let's combine the pneumonia and normal images and then recreate the train, validation and test sets by making a custom split. 80% of images now belong to the training set, 15% are used for validation and 5% are used for testing. We will also randomly shuffle the images in their respective datasets and set image size to 224.

```
pn = train_pn + test_pn + val_pn
normal = train_normal + test_normal + val_normal

train_imgs = pn[:3418]+ normal[:1224]
test_imgs = pn[3418:4059]+ normal[1224:1502]
val_imgs = pn[4059:] + normal[1502:]

random.shuffle(train_imgs)
random.shuffle(test_imgs)
```

Next, we define a function to preprocess the images. This function takes as input a list of images. We will load the images in grayscale, and resize the images. After stacking the images, we will normalize the images by dividing by 255 and end with flattening them. To create lists of labels, we will append 0 if the word 'NORMAL' appears in the image title and if we see 'IM', 'virus' or 'bacteria' in the image title, we will append 1 to the label list.

```
def preprocess_image(image_list):
 X = []
 y = []
 for image in image_list:
  img = cv2.imread(image,cv2.IMREAD_GRAYSCALE)
  img=cv2.resize(img,
(img_size,img_size),interpolation=cv2.INTER_CUBIC)
  img = np.dstack([img, img, img])
  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  img = img.astype(np.float32)/255.
  img = img.flatten()
  X.append(img)
  if 'NORMAL' in image:
   y.append(0)
  elif 'IM' in image:
   y.append(0)
  elif 'virus' or 'bacteria' in image:
   y.append(1)
return X, y
```

Then we create the required NumPy arrays for our machine learning models by passing the training, validation and image lists to the preprocess_image function and converting the results to NumPy.

```
X_train, y_train = preprocess_image(train_imgs)
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)

X_val, y_val = preprocess_image(val_imgs)
X_val = np.asarray(X_val)
y_val = np.asarray(y_val)
```




We have used the following evaluation metrics from the sklearn library:

1. Accuracy
2. Recall
3. Precision
4. F1-Score
5. Confusion Matrix

We will also import seaborn library for visualization

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score #weighted for imbalanced dataset
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

At the core of our project, our goal was to compare and contrast different Machine Learning techniques and analyze the results to determine which technique is the most suitable for detecting the presence of Pneumonia in chest X-rays

Thus, the supervised Machine Learning techniques that we have used are:

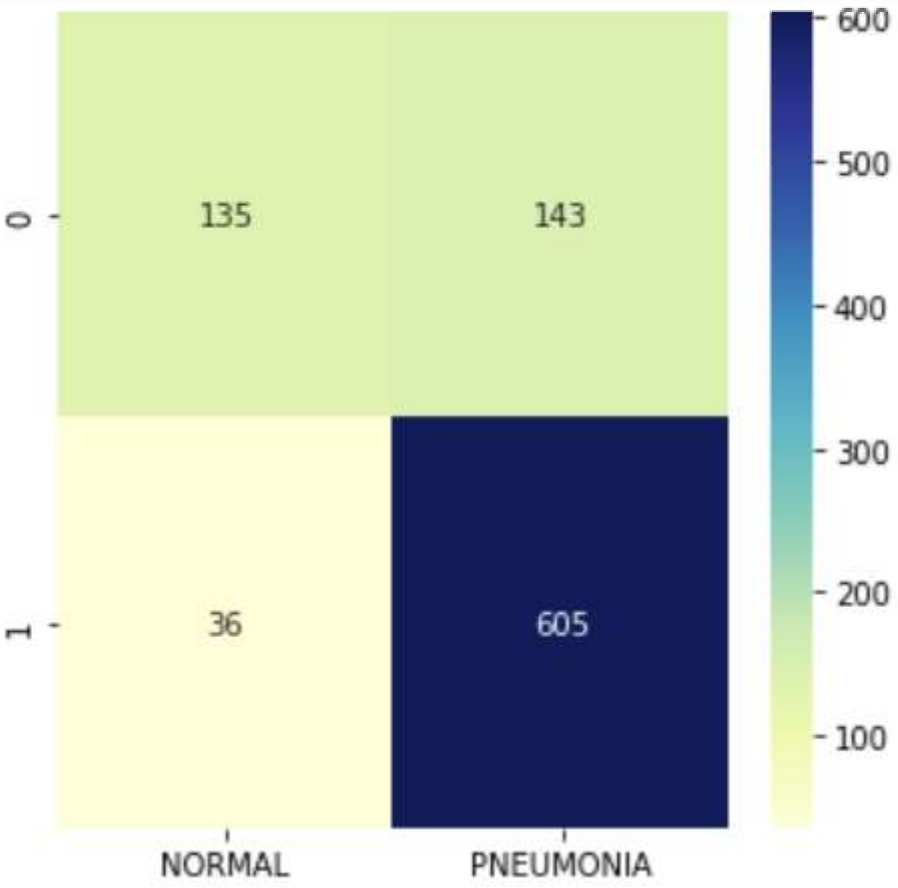
1. Decision Trees
2. Random Forest




```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("Accuracy is: ", accuracy_score(y_test, y_pred))
print("Balanced Accuracy is: ", balanced_accuracy_score(y_test,
y_pred))
print("Recall is: ",recall_score(y_test, y_pred))
print("Precision is: ",precision_score(y_test, y_pred))
print("F1 Weighted Score is: ",f1_score(y_test, y_pred, average =
'weighted'))

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix: ", cm)
tn, fp, fn, tp = cm.ravel()
print("True Negative: ", tn, "False Positive: ", fp, "False Negative:
", fn, "True Positive: ", tp)
```

The results of Decision Tree were as follows:

```
Accuracy is:  0.8052230685527747
Balanced Accuracy is:  0.7147246321507537
Recall is:  0.9438377535101404
Precision is:  0.8088235294117647
F1 Weighted Score is:  0.7895168886209192

Confusion Matrix:  [[135 143]
[ 36 605]]
True Negative:  135 False Positive:  143 False Negative:  36 True
Positive:  605
```



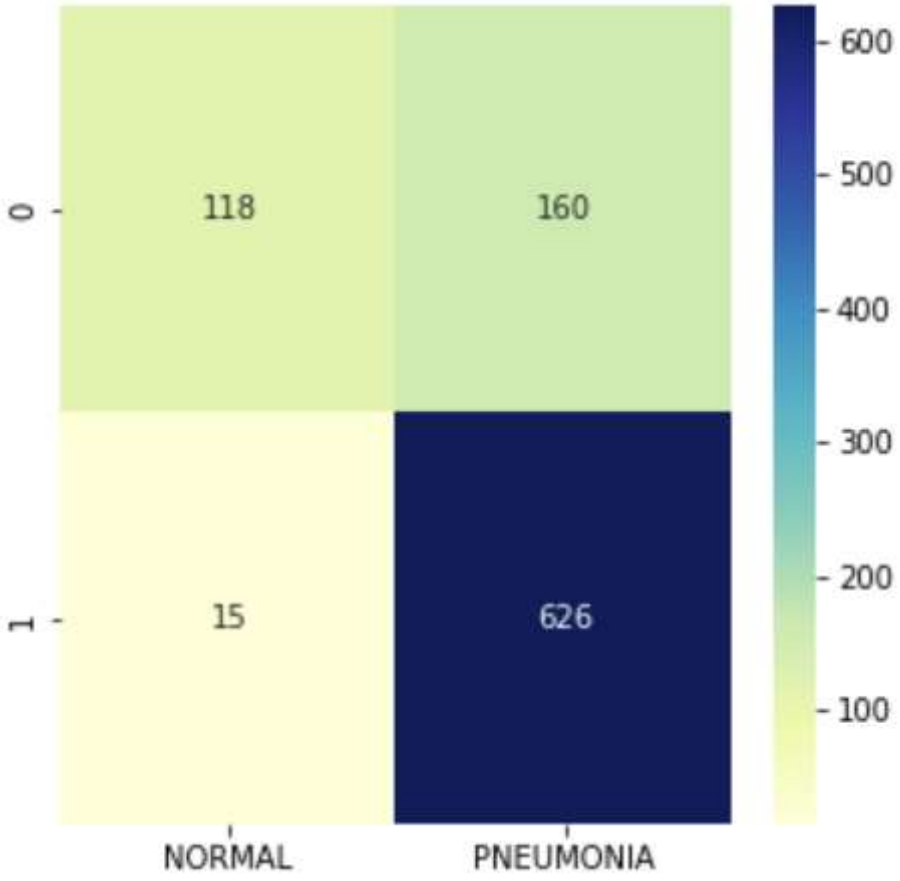
Confusion Matrix based on the output of Decision Tree

Implementation of Random Forest:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("Accuracy is: ", accuracy_score(y_test, y_pred))
print("Balanced Accuracy is: ", balanced_accuracy_score(y_test,
y_pred))
print("Recall is: ",recall_score(y_test, y_pred))
print("Precision is: ",precision_score(y_test, y_pred))
print("F1 Weighted Score is: ",f1_score(y_test, y_pred, average =
'weighted'))

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix: ", cm)
```

The results of Random Forest were as follows:

```
Accuracy is:  0.809575625680087
Balanced Accuracy is:  0.7005297478086174
Recall is:  0.9765990639625585
Precision is:  0.7964376590330788
F1 Weighted Score is:  0.7856596305217063

Confusion Matrix:  [[118 160]  [ 15 626]]
True Negative:  118 False Positive:  160 False Negative:  15 True
Positive:  626
```



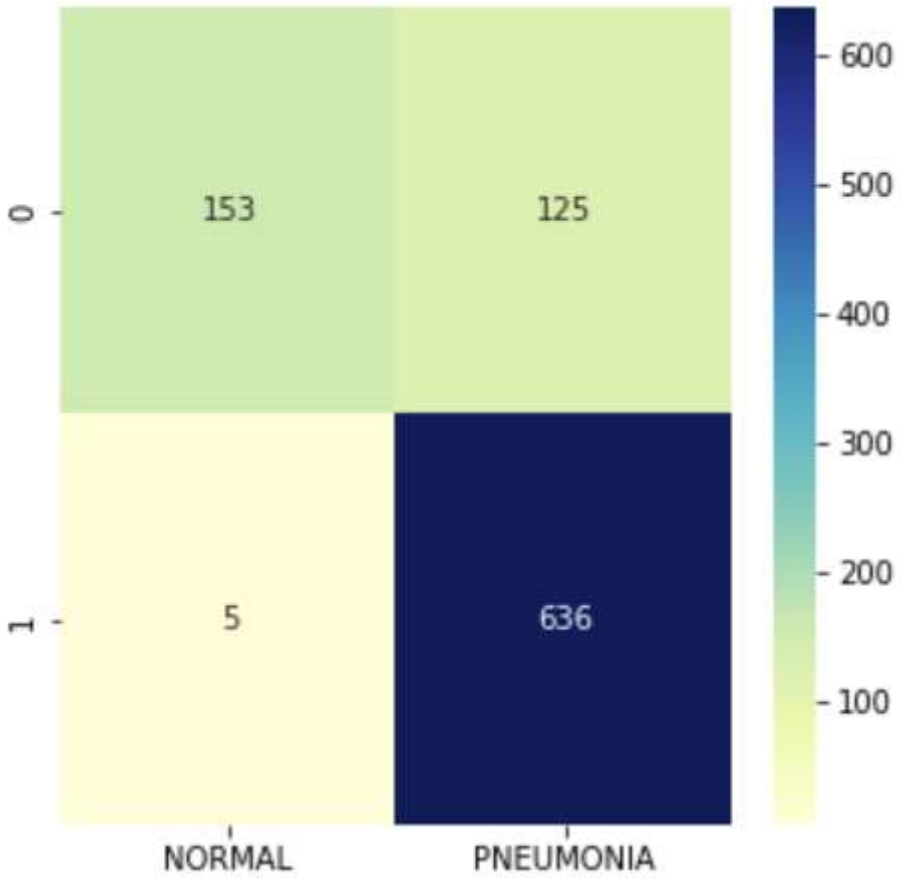
Confusion Matrix based on the output of Random Forest

Implementation of K-Nearest Neighbors:

```
knn_model.fit(X_train,y_train)
y_pred = knn_model.predict(X_test)

print("Accuracy is: ", accuracy_score(y_test, y_pred))
print("Balanced Accuracy is: ", balanced_accuracy_score(y_test,
y_pred))
print("Recall is: ",recall_score(y_test, y_pred))
print("Precision is: ",precision_score(y_test, y_pred))
print("F1 Weighted Score is: ",f1_score(y_test, y_pred, average =
'weighted'))

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix: ", cm)
tn, fp, fn, tp = cm.ravel()
print("True Negative: ", tn, "False Positive: ", fp, "False Negative:
", fn, "True Positive: ", tp)
```

The results for KNN were as follows:

```
Accuracy is:  0.8585418933623504
Balanced Accuracy is:  0.7712797001088676
Recall is:  0.9921996879875195
Precision is:  0.835742444152431
F1 Weighted Score is:  0.845129023378582

Confusion Matrix:  [[153 125]  [  5 636]]
True Negative:  153 False Positive:  125 False Negative:  5 True
Positive:  636
```



Confusion Matrix based on the output of K-Nearest-Neighbors

Thus, we can observe that so far, KNN has the best performance as measured by evaluation metrics. It is also worthwhile to mention that it also has the minimum False Negatives which is a primary concern when we build algorithms for medical diagnosis. The reason behind this is that we would want to prioritize being able to perform well when it comes to minimize the number of patients having Pneumonia being classified as Normal.

Now, we approach the problem of classification using CNN's. For the CNN's we decided to use a separate channel to import the data using ImageDataGenerators() given that we decided to use the Keras package to build our CNN.

```
train_clean = ImageDataGenerator()
size = (64,64)
```

```
shuffle=True,
seed=1,
batch_size = 32)

validation_data =
train_clean.flow_from_directory('../content/chest_xray/val/',
target_size=size,
color_mode="grayscale",
shuffle=True,
seed=1,
batch_size=16)

test_data =
train_clean.flow_from_directory('../content/chest_xray/test/',
target_size=size,
shuffle=False,
color_mode="grayscale",
batch_size=16)
```

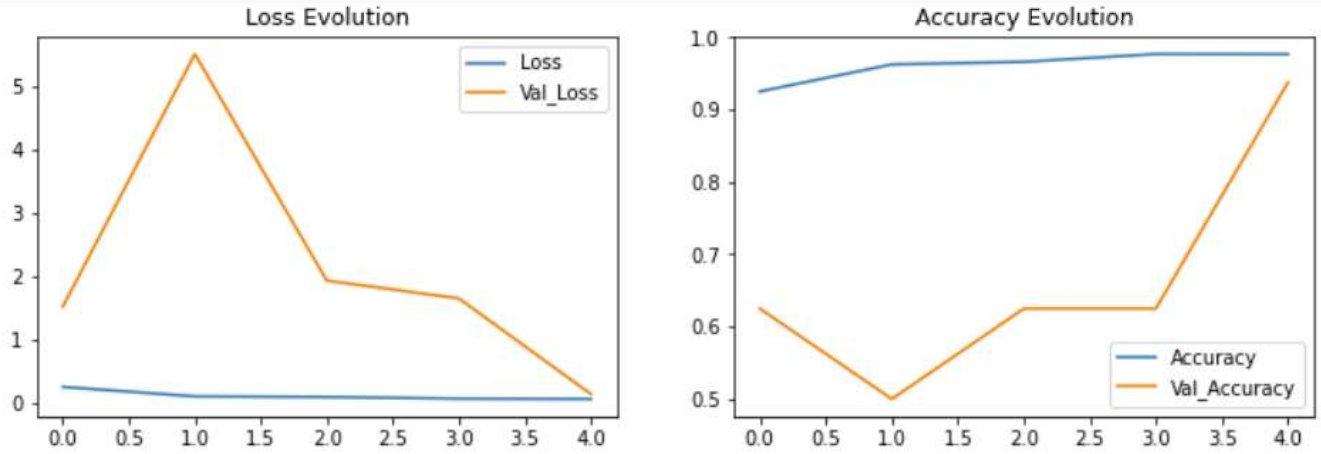
This means that we don't have an exact comparison with the baseline models, but we suspect that this will have an only minimal effect. For the CNN itself, we built it with two computational layers, both using 3x3 window sizes. The first layer has 32 filters, and the second has 64. We also included various dropout, batch normalization, and pooling layers as we were worried about overfitting given the limited size of the data.




```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 64, 64, 32)        320

 batch_normalization (BatchN  (None, 64, 64, 32)       128
 ormalization)

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

 dropout (Dropout)           (None, 32, 32, 32)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 64)        18496

 batch_normalization_1 (Batc  (None, 32, 32, 64)       256
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 16, 16, 64)        0

 flatten (Flatten)           (None, 16384)             0

 dense (Dense)               (None, 1024)              16778240

 batch_normalization_2 (Batc  (None, 1024)             4096
 hNormalization)

 dropout_2 (Dropout)         (None, 1024)              0

 dense_1 (Dense)             (None, 2)                 2050

=================================================================
Total params: 16,803,586
Trainable params: 16,801,346
Non-trainable params: 2,240
_________________________________________________________________
```

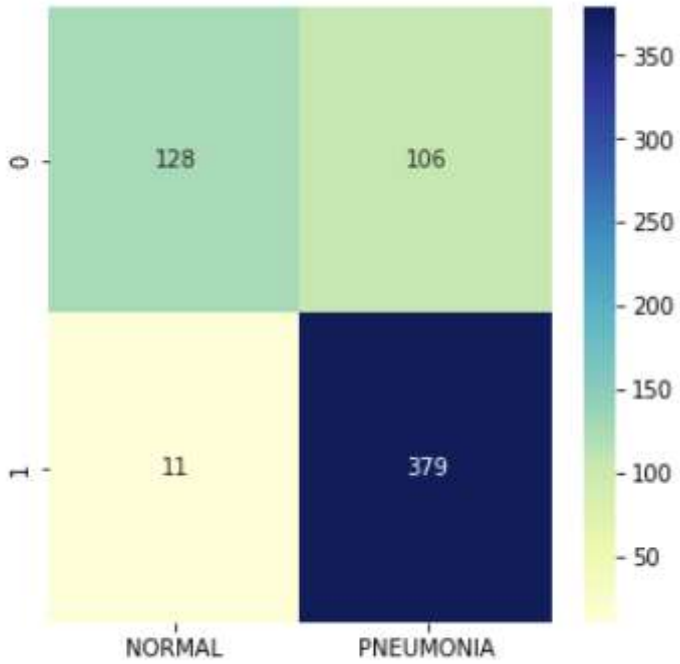
Model Architecture of CNN

We then proceeded to train our model for five epochs with a batch size of 32. Very quickly,



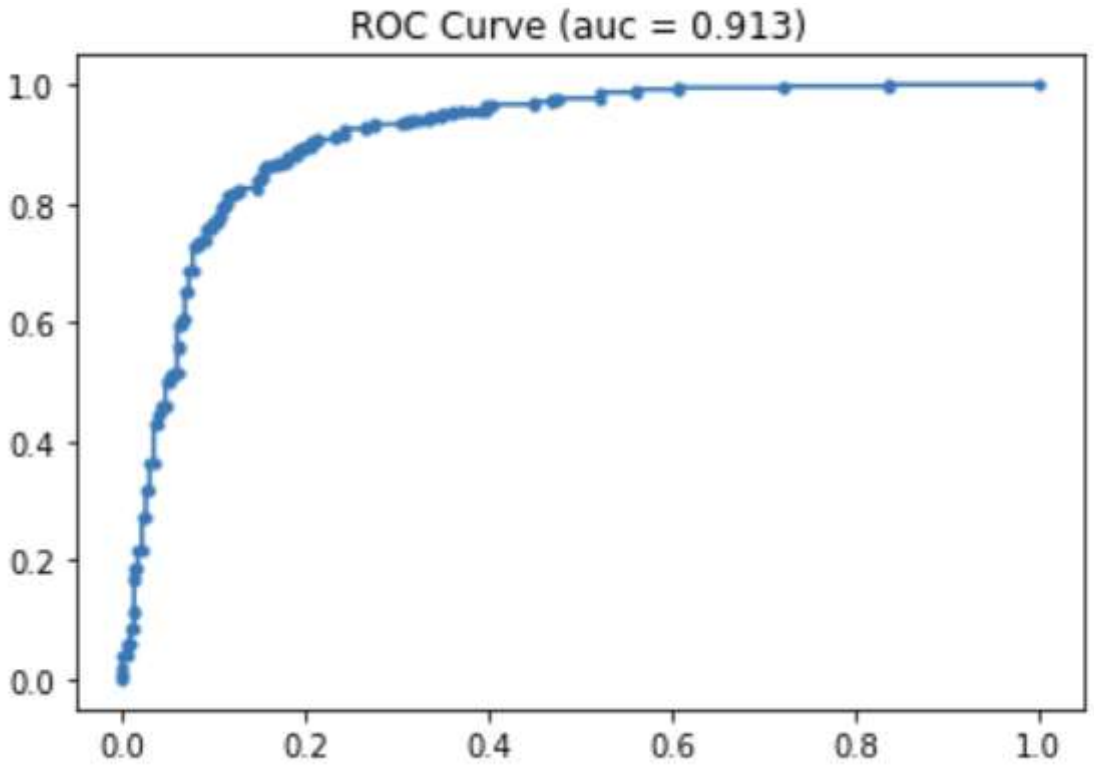
Loss and Accuracy plots of CNN

We evaluated the data on our testing set, and found quite nice results. The model had the following predictions at a 0.6 threshold:



Confusion Matrix based on the output of CNN

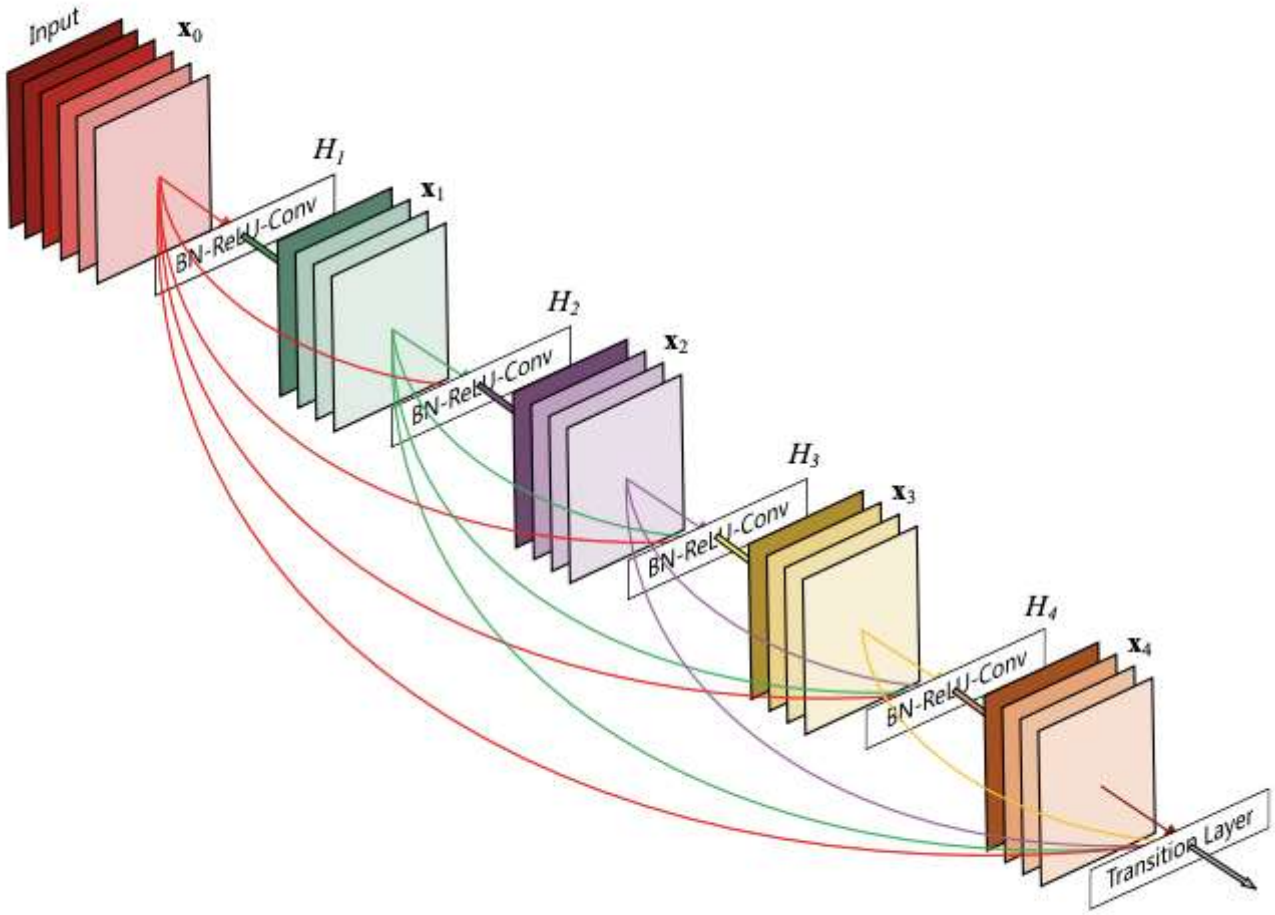
We decided on the 0.6 threshold after plotting an ROC curve for the data. The model had a quite high AUC at 0.913.



AUC ROC Curve for CNN

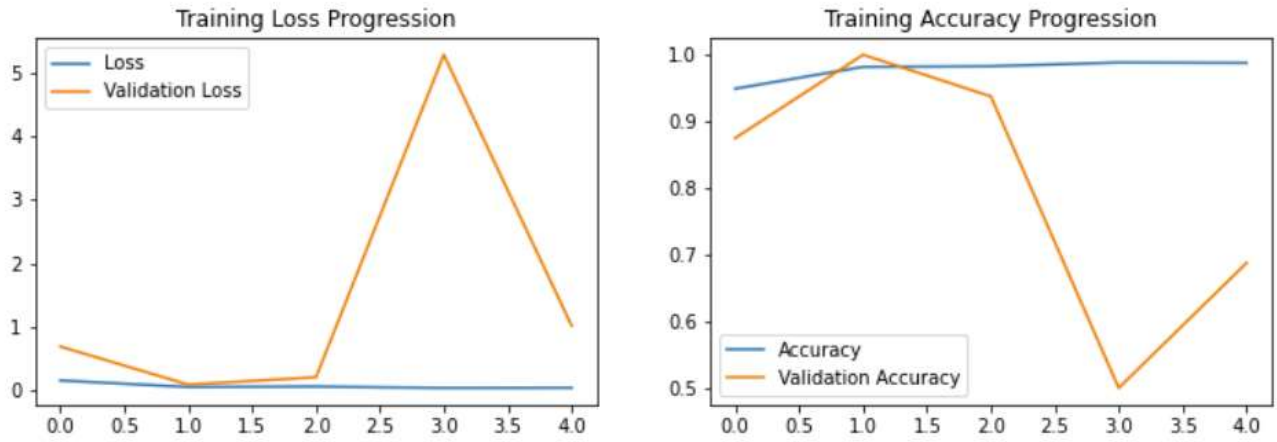
For comparison to the baseline networks, i.e. KNN and FCN, we also calculated an f1-score and the test accuracy. These were more disappointing in their results. The weighted F1 score was 0.799, while test accuracy was 0.77.

As the results of CNN were not as good as we expected, we decided to see if a transfer learning approach would work. Transfer learning is borrowing another pre-trained CNN and adapting to a new task. We wanted to stick with Keras given time and computing constraints, so we chose one of their available for download models: DenseNet121 trained on imagenet. DenseNet121 also has some special features (connections between non-sequential layers).



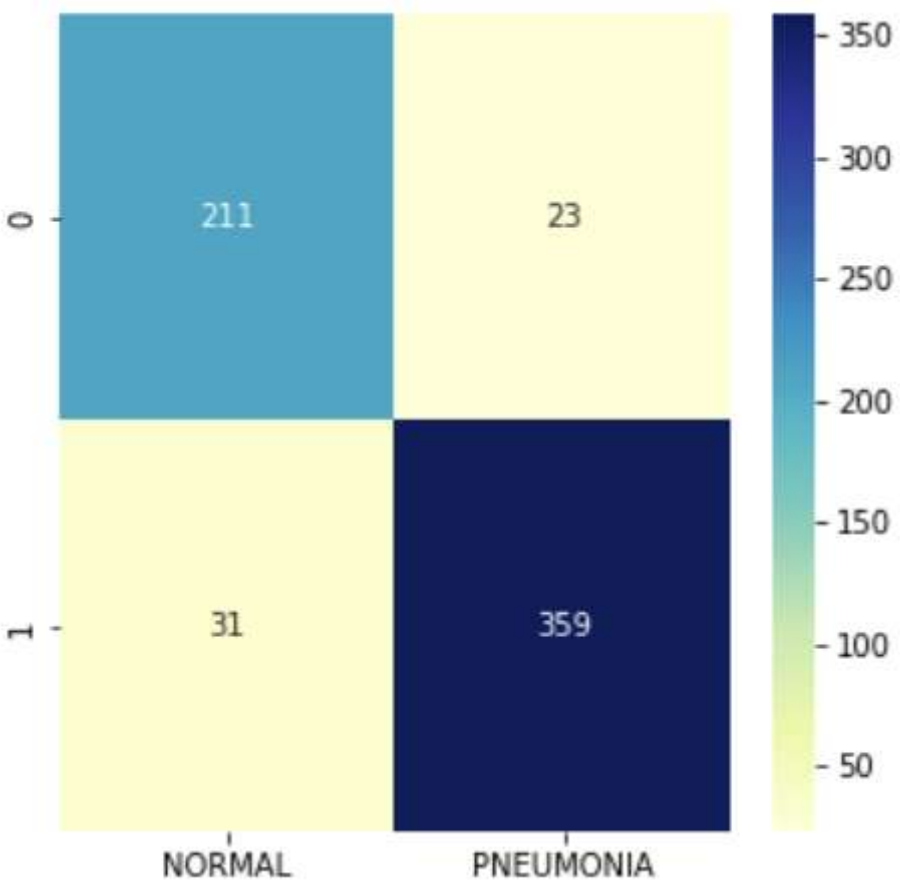
DenseNet121 (Source: https://pytorch.org/hub/pytorch_vision_densenet/)

DenseNet121 did require some adaptation to work for this particular dataset. We added on some layers so that its output would fit our task. We still used softmax and categorical cross entropy for loss as those worked well for our CNN. Further, we had to adapt our importation of data to have three channels due to DenseNet121 (this merely meant replicating the data three times, something which ImageDataGenerator does automatically if the “grayscale” option is not specified). We trained this network for five epochs.

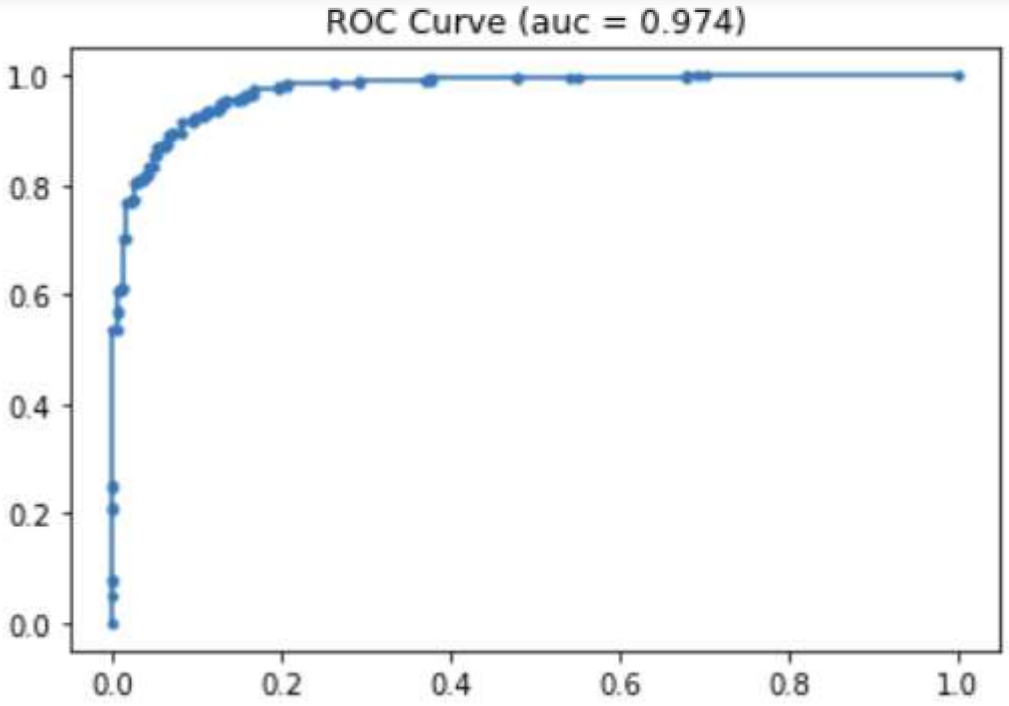


Loss and Accuracy plots for Transfer Learning Approach

Transfer learning performed really well and the results are as follows:



Confusion Matrix based on the output of Transfer Learning



AUC ROC curve of Transfer Learning

Finally, we calculated the F1 and test accuracy scores to compare with our previous methods. These were 0.914 and 0.910, respectively, scores which were quite a bit better than our baseline methods and our constructed CNN.

Let us perform further analysis on the task —

As we found out in the beginning, the original dataset had a huge class imbalance. Since this has an impact on our results, we decided to explore two strategies to balance the classes:

1. Upsampling
2. Downsampling

We chose the KNN model to test the effects of upsampling and downsampling as it had the best performance out of Decision Tree, Random Forest and KNN.

We will define a function which will perform upsampling or downsampling based on what

```
if strategy == "upsample":
upsampled_data = resample(minor_data, replace = True, n_samples =
len(major_data), random_state = 42)
return upsampled_data

if strategy == "downsample":
downsampled_data = resample(major_data, replace = True, n_samples =
len(minor_data), random_state = 42)
return downsampled_data
```

Since the Pneumonia class had more examples than the Normal class, let us downsample the Pneumonia class to see how the performance of KNN model is impacted:

```
pn_downsample = fix_class_imabalance(major_data = pn, minor_data =
normal, strategy = "downsample")

train_imgs_down = pn_downsample[:1224]+ normal[:1224]
test_imgs_down = pn_downsample[1224:1502]+ normal[1224:1502]
val_imgs_down = pn_downsample[1502:] + normal[1502:]
```

The rest of the preprocessing and modeling is done in the same way as before. The results were as follows:

```
Accuracy is:  0.8345323741007195
Balanced Accuracy is:  0.8345323741007193
Recall is:  0.9496402877697842
Precision is:  0.7719298245614035
F1 Weighted Score is:  0.8323105166535536

Confusion Matrix:  [[200  78]  [ 14 264]] True Negative:  200 False
Positive:  78 False Negative:  14 True Positive:  264
```

We can see that downsampling the Pneumonia class, slightly reduced the performance of the model. We think that it happened because when we downsampled the data, we lost information in the process which leaded to weaker results.




```
normal_upsample = fix_class_imabalance(major_data = pn, minor_data =
normal, strategy = "upsample")

train_imgs_up = normal_upsample[:3418]+ normal[:3418]
test_imgs_up = normal_upsample[3418:4059]+ normal[3418:4059]
val_imgs_up = normal_upsample[4059:] + normal[4059:]
```

After proprocessing and modeling, we observed that surprsingly the results were much worse than the results of downsampling. We believe that the model highly overfit to the training data, thus resulting in poor performance on the test data.

Another tricky and time-consuming part of any data science project is hyperparameter tuning. We used GridSearchCV by sklearn to find the best set of hyperparameters on the random forest model. The sole reason behind choosing Random Forest model for this analysis was the fact that it has more trainable parameters than KNN. The parameters we have tried different values for are: n_estimators, max_features, max_depth and criterion.

```
from sklearn.model_selection import GridSearchCV

clf = RandomForestClassifier(random_state=42)

param_grid = {

'n_estimators': [50,100],

'max_features': ['auto', 'sqrt'],

'max_depth' : [4,8],

'criterion' :['gini', 'entropy']

}

rf_grid = GridSearchCV(estimator=clf, param_grid=param_grid, cv=
3,verbose=3)

rf_grid.fit(X_train, y_train)
```

The model picked the following combination of parameters:

criterion=entropy, max_depth=8, max_features=sqrt, n_estimators=100

The results on test set were as follows:

```
Accuracy is:  0.8650707290533188
Balanced Accuracy is:  0.7932748964634844
Recall is:  0.9750390015600624
Precision is:  0.8526603001364257
F1 Weighted Score is:  0.856211403020235

Confusion Matrix:  [[170 108]  [ 16 625]]
True Negative:  170 False Positive:  108 False Negative:  16 True
Positive:  625
```

Therefore, we can observe that the performance of Random Forest model improved when it used the parameters obtained byGridSearchCV.

Based on the observations obtained by comparing the performance of different algorithms on our dataset, we can conclude that Transfer Learning outperformed the other methods with an F1-Score of 0.91.

The main challenges faced were due to computational limitations. A lot of the modeling that we could do was limited by the capacities of the Colab environment, especially given that we were using a free version of the software. For example, the reason for using Densenet121 instead of Densenet201 or even Densenet160 is due to worries about the time needed to train the latter two. Densenet itself was also chosen because of the limited number of parameters it has, only in the millions rather than the tens of millions. This was also the problem with our own CNN modeling. The amount of training epochs used was limited by worries of time and computing capacity. For the neural networks in particular, managing to get consistent results was also something of a challenge. Often, it would return satisfactory results, but fail completely when slight modifications were made or even when the code was shifted to a new notebook. Sometimes, even without changes

For next steps, there are several other models that we did not use in our analysis but would be informative to implement. For example, Keras also has weights for VGG, GooLeNet, etc. A future project could be implementing transfer learning with these networks and, then, comparing the results to our current implementation of transfer learning. Further, so far, we have been only using individual models. However, we could create an ensemble model using a variety of approaches. These ensemble models, when well constructed, often perform better than individual models. It would be interesting to see how they work for this example. This, however, is likely to be limited by the same technical challenges that we currently are facing with regards to the individual models. To combat that, and also see how our individual models would work without such limitations, a natural direction for the project is finding more computing power. This could come, perhaps, in paying for a better Colab tier. It could also be through moving to a different environment.

Please feel free to reach out to us with feedback!

The link to the entire project code and data can be found at this repository: https://github.com/datascience98/Pneumonia-Detection