ARTIFICIAL INTELLIGENCE (CS-412) PROJECT REPORT
COMPUTER & INFORMATION SYSTEMS ENGINEERING
FALL SEMESTER 2020

Lung problem identification system using machine learning targeting COVID-19

Muhammad Taha	| CS-17020
Abdul Ahad	| CS-17022
Syedah Nawal Munif	| CS-17024
Muhammad Mustafa Usmani	| CS-17034

Batch 2017 – Section A

Submitted to: Dr. Saad Qasim Khan

Table of Contents

Introduction

Machine learning (ML) based strategies have appeared to be a phenomenal victory within the reliable investigation and analysis of medical images. ML-based approaches are adaptable, automatable, and simple to implement and execute in a clinical atmosphere. Chest-X Ray (CXR) radiography is one of the foremost commonly utilized and available strategies for quick examination of the lung conditions. The accessibility of CXR radiography made it one of the primary imaging modalities to be utilized amid the later COVID-19 widespread. In expansion, the CXR turnaround was utilized by the radiology sectors in Italy and the U.K. to triage non-COVID-19 patients with pneumonia to distribute hospital assets productively. In any case, there are numerous common features between medical images of COVID-19 and pneumonia caused by other viral contaminants such as common flu. This likeness makes a differential diagnosis of COVID-19 cases by expert radiologists challenging. A solid automated algorithm for classification of COVID-19 and non-COVID-19 CXR pictures can speed up the triage process of non-COVID-19 cases and maximize the allotment of hospital assets to COVID-19 cases. Be that as it may, the similitude between highlights of CXR pictures of COVID-19 and pneumonia caused by other diseases make the differential diagnosis by radiologists challenging. We hypothesized that machine learning-based classifiers can reli our examinations giving solid confirmation of the concept that basic ML-based classification can be productively implemented as an aide to other tests to encourage differential conclusion of CXR of COVID-19 patients.

Background Research

The term “medical imaging” (aka “medical image analysis”) is used to describe a wide variety of techniques and processes that create a visualization of the body’s interiors in general and also specific organs or tissues.

Overall, medical imaging covers such disciplines as:

- X-ray radiography;
- Magnetic Resonance Imaging (MRI);
- Ultrasound;
- Endoscopy;
- Thermography;
- Medical Photography in general and a lot more.

Healthcare providers generate and capture enormous amounts of data containing extremely valuable signals and information, at a pace far surpassing what “traditional” methods of analysis can process. Machine learning therefore quickly enters the picture, as it is one of the best ways to integrate, analyze and make predictions based on large, heterogeneous data sets.

Healthcare applications of deep learning range from one-dimensional biosignal analysis and the prediction of medical events, e.g. seizures and cardiac arrests , to computer-aided detection and diagnosis supporting clinical decision making and survival analysis, to drug discovery and as an aid in therapy selection and pharmacogenomics, to increased operational efficiency, stratified care delivery, and analysis of electronic health records.

The implementation of deep learning into medical image analysis can improve on the main requirements for the proceedings [2]. Here is how:

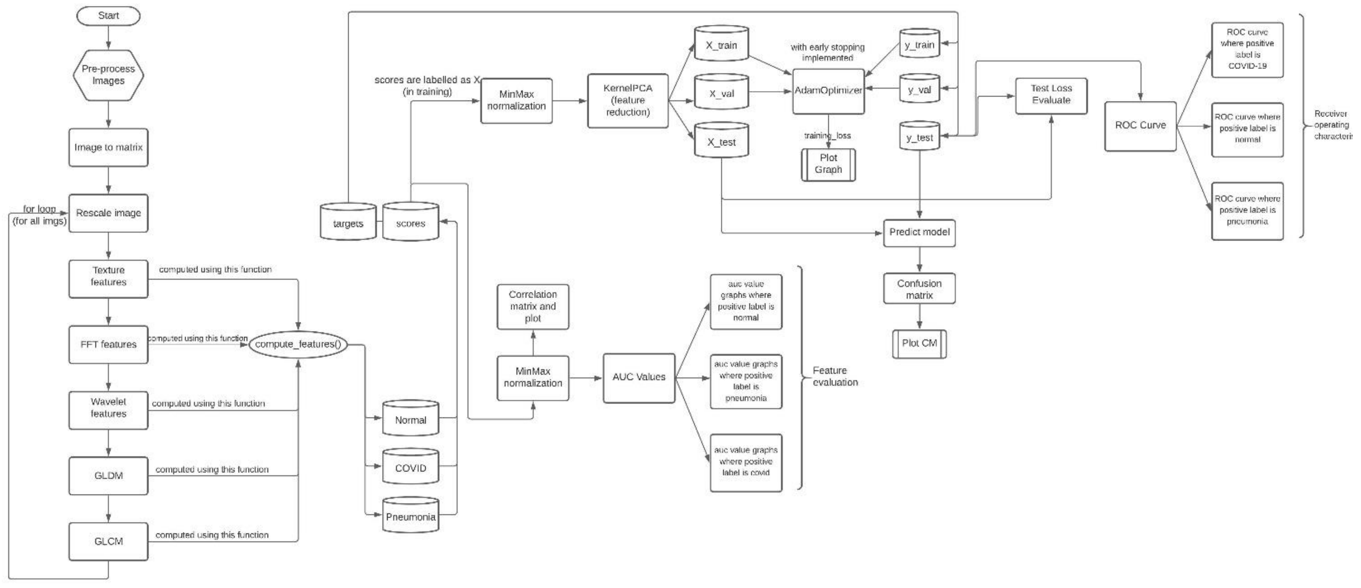
- Provide high accuracy image processing;
- Enable input images analysis with an appropriate level of sensibility to certain field-specific aspects (depends on the use case. For example, bone fracture analysis).

The primary operations handled by deep learning medical imaging application in our project are:

- **Diagnostic image classification** — involves the processing of the examination images, comparison of different samples. It is primarily used to identify objects and lesions into specific classes based on local and global information about the object's appearance and location.
- **Lesion segmentation** — combines object detection and organ / substructure segmentation.
- **Organ/substructure segmentation** — involves identifying a set of pixels that define contour or object of interest. This process allows quantitative analysis related to shape, size, and volume.
- **Spatial alignment** — involves the transformation of coordinates from one sample to another. It is mainly used in clinical research.
- **Content-based image retrieval** — used for data retrieval and knowledge discovery in large databases. One of the critical tools for navigation in numerous case histories and understanding of rare disorders.

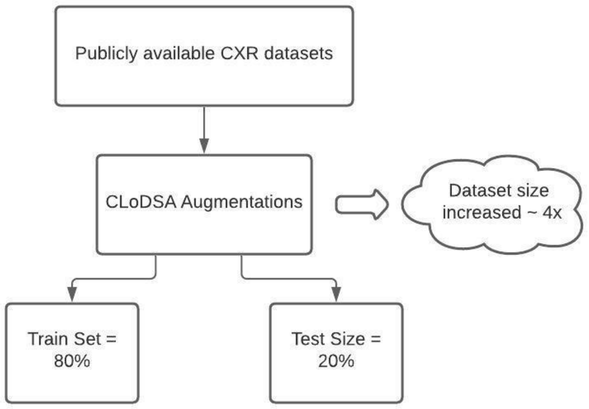
In this study, we demonstrated that an efficient machine learning classifier can accurately distinguish COVID-19 CXR images from normal cases and also pneumonia caused by other viruses. Although different imaging modalities have been applied for lung screening, X-ray remains the fastest and widely used tool for population-based lung disease screening. However, a large number of suspicious lung lesions can result in misclassification of cases[1]. Thus, the development of new approaches to facilitate the classification of different types of lung conditions is crucial to improve the efficacy of lung screening and analysis. In this study, we developed a novel machine learning scheme utilizing the global image features to predict the probability of the testing cases being COVID-19 without lesion segmentation.

Data Flowchart


Start
Pre-process Images
Image to matrix
for loop (for all imgs)
Rescale image
Texture features
FFT features
Wavelet features
GLDM
GLCM
computed using this function
compute_features()
Normal
COVID
Pneumonia
targets
scores
scores are labelled as X (in training)
MinMax normalization
KernelPCA (feature reduction)
X_train
X_val
X_test
with early stopping implemented
AdamOptimizer
training_loss
Plot Graph
y_train
y_val
y_test
Test Loss Evaluate
Predict model
Confusion matrix
Plot CM
ROC Curve
ROC curve where positive label is COVID-19
ROC curve where positive label is normal
ROC curve where positive label is pneumonia
Receiver operating characteri
Correlation matrix and plot
MinMax normalization
AUC Values
auc value graphs where positive label is normal
auc value graphs where positive label is pneumonia
auc value graphs where positive label is covid
Feature evaluation

Dataset and Dataset Source

We have acquired the dataset for this project from:
https://github.com/arunsharma8osdd/covidpred
The dataset includes 2-D X-ray images, in the Posteroanterior (P.A.) chest view, classified by valid tests to three predefined categories of Normal, Pneumonia, and COVID-19. The dataset is open-source. We are using augmented (flipped and rotated) dataset because augmentation, can prevent the neural network from learning irrelevant patterns, essentially boosting overall performance. The dataset is augmented using open-source augmentation rule CLoDSA.



Data Imbalance

Training a machine learning model on an imbalanced dataset can introduce unique challenges to the learning problem. Imbalanced data typically refers to a classification problem where the number of observations per class is not equally distributed; often you'll have a large amount of data/observations for one class (referred to as the majority class), and much fewer observations for one or more other classes (referred to as the minority classes). Here the COVID-19 class is the majority class and the pneumonia and normal classes are the minority.
For this classification problem there's a clear separation in the data, class imbalance doesn't impede on the model's ability to learn effectively. We're updating a parameterized model by Adam optimizer to minimize our cross-entropy loss function, we'll be spending most of our updates changing the parameter values in the direction which allow for correct classification of the majority class.

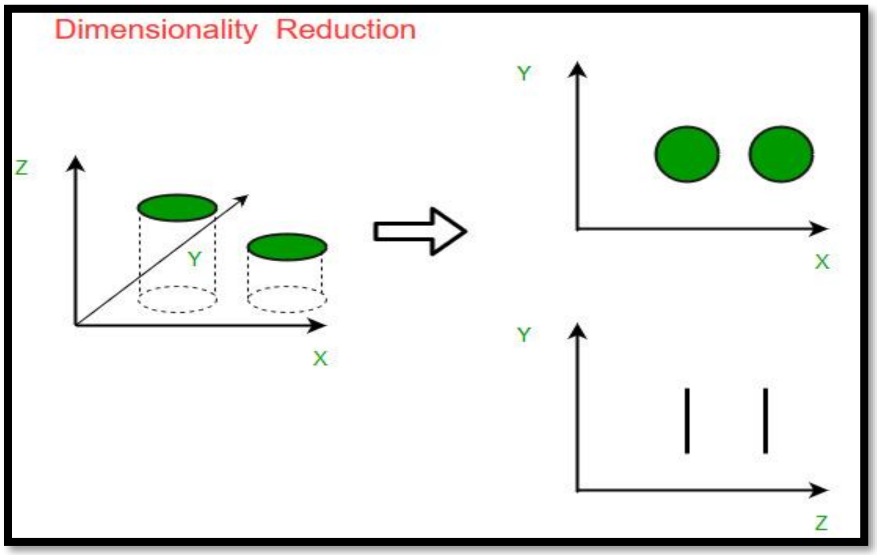
Feature Extraction

Introduction

Feature extraction is one of the fundamental factors in Machine Learning especially when we are dealing with datasets having an abundance of characteristics or ‘features’ in them. Feature extraction methods really shine when we have a large dataset and need to reduce the number of resources w/o losing any relevant information which gives the data its main characteristics.
So before diving deep into the importance of Feature Extraction and how it is being used in our project we have to first understand what *exactly is* feature extraction.

Definition and Its Importance

Feature extraction is actually a subdivision of a much renowned process which is *Dimensionality Reduction* process. In simple explanation, we can understand that Dimensionality reduction is a process where a large set of data is divided or reduced into more efficient groups because as the data gets bigger, it gets harder and harder to work on it for the training set phase and this is where dimensionality reduction is really helpful as it reduces the number of random variables and defines a set of fundamental variables or features. **Feature Selection** and **Feature Extraction** comes under the umbrella of Dimensionality reduction process. Feature Extraction is where we reduce the redundant or correlated characteristics of data by decreasing the data from a high dimension space to a lower dimension space i.e. a data with lesser but fundamental characteristics.



Visual representation of feature extraction from higher dimension data to lower dimension

The reduction of the data done in Feature Extraction helps us to build a model with less stress of machine and more efficient in terms of learning.

Implementation of Feature Extraction in COVID-19 Classifier

In our project, we have used dimensionality reduction with feature extraction to generate a set of optimal features of Chest X-Ray or CXR images to build an efficient classifier that can distinguish between COVID and non-COVID patient's CXR. In order to achieve this we have used a scheme to compute a total of 252 features in both, spatial and frequency domain because many ML models only focuses on computing spatial domain features and due to our usage of features from both the domains, we saw the significance of acquiring such frequency domain features hand-in-hand with spatial domain features that implies the great relevance of those features for the detection of COVID-19 infection in the CXR image.
Furthermore, the 252 features that are computed in both spatial and frequency domains are then categorized into five groups that are:

- Histogram based Statistical Texture Features
- Fast Fourier Transform (FFT) Features
- Gray-Level Co-Occurrence Matrix (GLCM) Features
- Gray-Level Difference Method (GLDM) Features
- Wavelet Transform Features

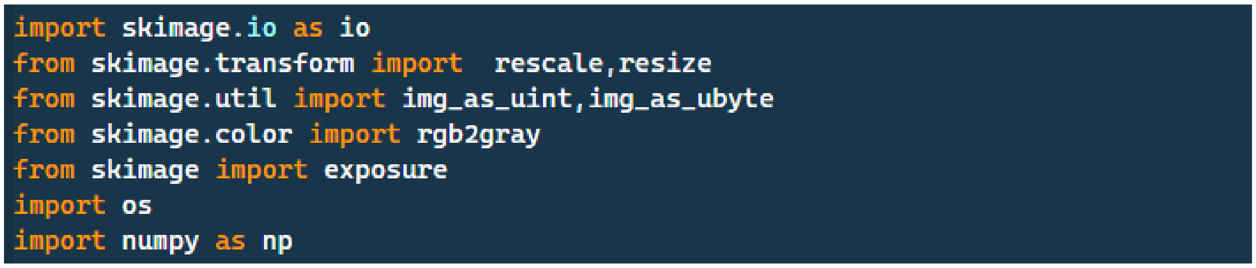
All these 252 features are divided into these groups like 14 features from Histogram based Texture Features and FFT both, 56 features from GLCM and GLDM both and 112 features from Wavelet transform.

Converting images to their statistics data for processing

Till now, we understood that what are the groups of features we are using and further along the report we will also discuss *why* each group that is been categorized is used in our classification model. Now we will understand how we are extracting those features from the CXR data set.
In order to understand clearly, we will go through some code that helps in sorting data and get it ready in order to get the features extracted out of it.

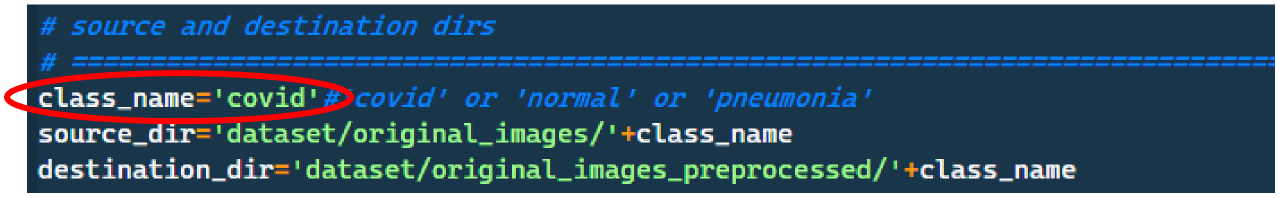
Pre-Processing of images

Before importing images and getting them ready for extraction, we will import all the necessary libraries which are as follows:



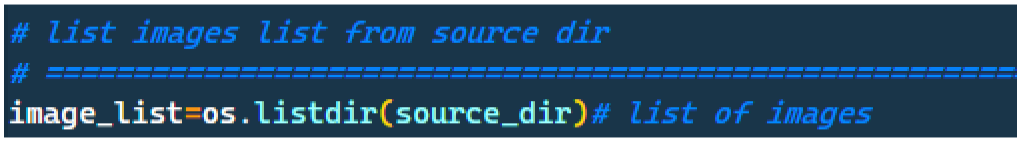
```
import skimage.io as io
from skimage.transform import  rescale,resize
from skimage.util import img_as_uint,img_as_ubyte
from skimage.color import rgb2gray
from skimage import exposure
import os
import numpy as np
```

Here the most important libraries that needs some explanation are *skimage* and *numpy. Skimage* or *Scikit-image* is an open-source library which is heavily used for image processing in python. The reason of using it is to rescale all the images at a reasonable size i.e 512x512 and also for other resources such as for converting colorful images to grayscale if there are any and many other resources that we will discuss shortly. On the other hand, *NumPy* is also an open source and highly recognized library for scientific computing in Python.
After doing all the importing, we initialized the directories from the data will be coming and the where the data will be going after prepping it.



```
# source and destination dirs
# ====================================================================
class_name='covid'#'covid' or 'normal' or 'pneumonia'
source_dir='dataset/original_images/'+class_name
destination_dir='dataset/original_images_preprocessed/'+class_name
```

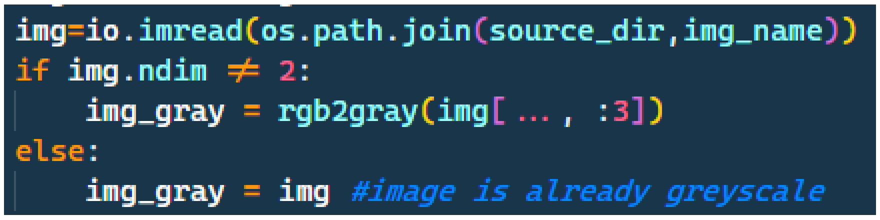
In the above code, as we want every CXR in its respective folder, so we initialized the folder name separately so when we need to preprocess normal CXR we will just change the "class_name" variable instead of changing the whole source and destination directory variable.



```
# list images list from source dir
# ======================================================
image_list=os.listdir(source_dir)# list of images
```

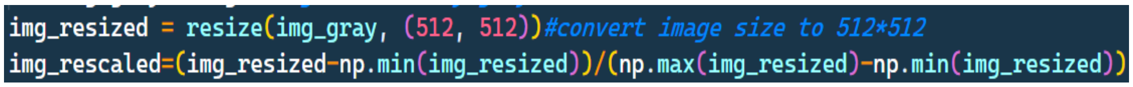
Now we made another variable in order to get all the list of images so that we can use all the images in the list. After that we, one by one, take one image and do the following task on it.

- We first convert the image into grayscale from rgb if it's not already in grayscale



```
img=io.imread(os.path.join(source_dir,img_name))
if img.ndim != 2:
    img_gray = rgb2gray(img[ ... , :3])
else:
    img_gray = img #image is already greyscale
```

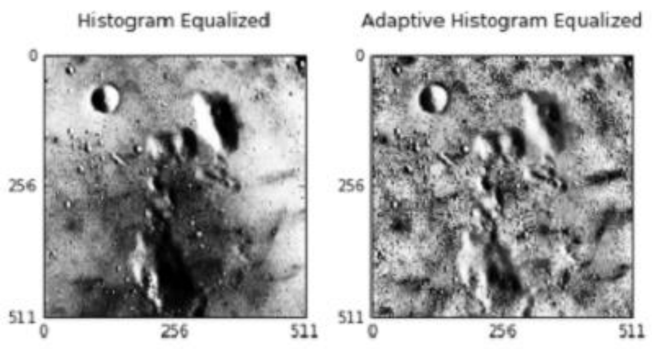
- Then we resize all the images to 512x512 and also rescale them using *min-max normalization*



```
img_resized = resize(img_gray, (512, 512))#convert image size to 512*512
img_rescaled=(img_resized-np.min(img_resized))/(np.max(img_resized)-np.min(img_resized))
```

This is how we are using the min-max normalization and the reason why we are using it is that it helps to understand the data more easily and scales the data between 0 and 1 (it gives the data an upper and lower boundary)

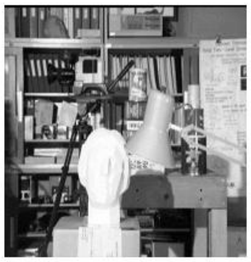
- Now we use an important technique to enhance the exposure in every image and that it CLAHE or Contrast Limited Adaptive Histogram Equalization. An Adaptive Histogram Equalization or AHE is a method in computer image processing used to improve contrast in images. An AHE computes several histograms, each for a distinct section of the image and uses all of them to redistribute the lightness and darkness value of the image hence more suitable than normal Histogram Equalization. But the reason we have used CLAHE is that standard AHE, in some cases, also introduces noise in the enhanced image and in our dataset it is very critical not to have any sort of noise as it associated with medical purposes.

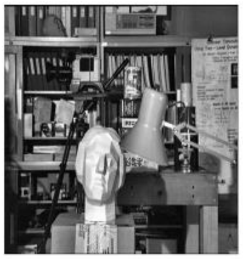


AHE performance in comparison of normal HE but also an increase in noise

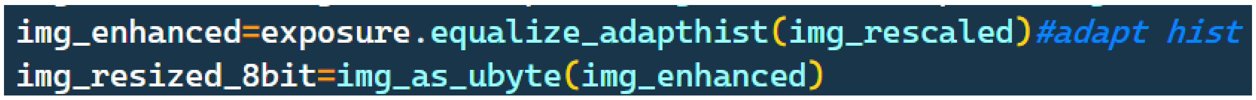
So, a CLAHE method works like AHE but prevents the over amplification of noise.

Initial Image

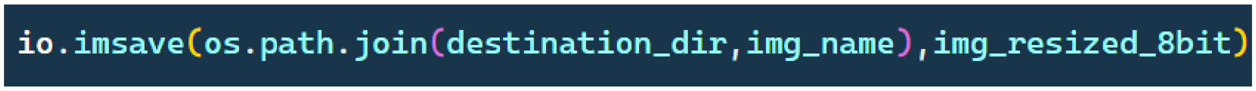

Histogram Equalized Image


Contrastive Limited Adaptive Equalized Image

The code for the CLAHE is of only one because everything is being handled by the scikit-image library, we just have to insert the image in the predefined function which is as follows:


```
img_enhanced=exposure.equalize_adapthist(img_rescaled)#adapt hist
img_resized_8bit=img_as_ubyte(img_enhanced)
```

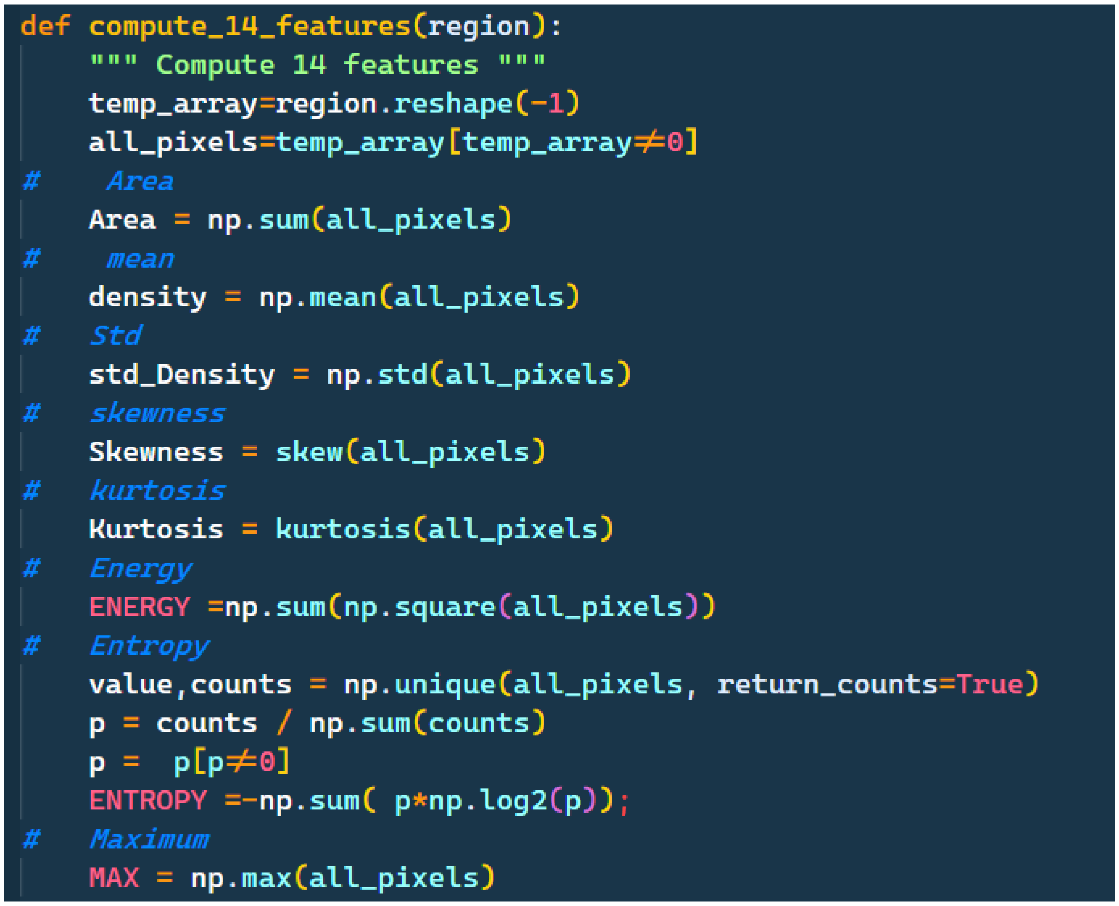
- After that, we just saved the enhanced images to the destination directory and the process repeats again for all the images


```
io.imsave(os.path.join(destination_dir,img_name),img_resized_8bit)
```

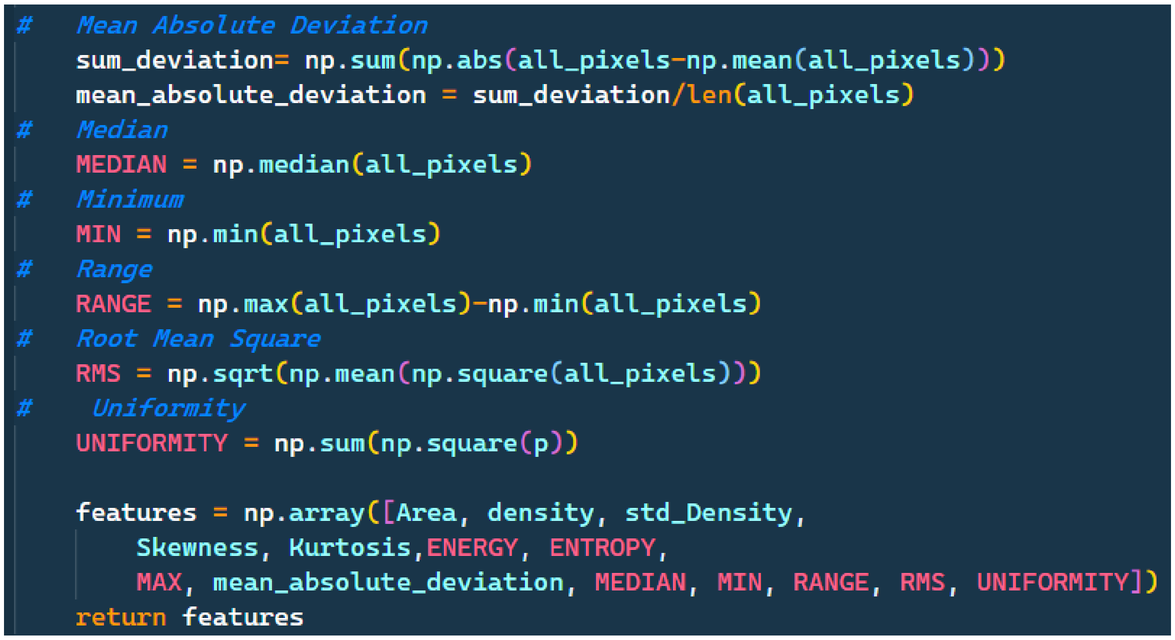
Global Image Features Extraction

Now comes the most important of our Feature Extraction process which is sort of like building the foundation for the groups of features that have been categorized above. Global features are those features which are not confined to a segmented area but rather available on the whole data, in our case, on the whole image. Those features are more like the statistical features of the image like Mean, Standard Deviation, Energy, Entropy, as well as Skewness and Kurtosis.
We will see how we have obtained those features and why they are useful as we move forward. Going back to the code now we will import all the libraries necessary for the

extraction, which in this case is only the *NumPy* library. After that, we will one by one initialize variables and assign them one feature each; as shown in code below:



```
def compute_14_features(region):
    """ Compute 14 features """
    temp_array=region.reshape(-1)
    all_pixels=temp_array[temp_array!=0]
#     Area
    Area = np.sum(all_pixels)
#     mean
    density = np.mean(all_pixels)
#    Std
    std_Density = np.std(all_pixels)
#    skewness
    Skewness = skew(all_pixels)
#    kurtosis
    Kurtosis = kurtosis(all_pixels)
#    Energy
    ENERGY =np.sum(np.square(all_pixels))
#    Entropy
    value,counts = np.unique(all_pixels, return_counts=True)
    p = counts / np.sum(counts)
    p =  p[p!=0]
    ENTROPY =-np.sum( p*np.log2(p));
#    Maximum
    MAX = np.max(all_pixels)
```



```
#    Mean Absolute Deviation
    sum_deviation= np.sum(np.abs(all_pixels-np.mean(all_pixels)))
    mean_absolute_deviation = sum_deviation/len(all_pixels)
#    Median
    MEDIAN = np.median(all_pixels)
#    Minimum
    MIN = np.min(all_pixels)
#    Range
    RANGE = np.max(all_pixels)-np.min(all_pixels)
#    Root Mean Square
    RMS = np.sqrt(np.mean(np.square(all_pixels)))
#     Uniformity
    UNIFORMITY = np.sum(np.square(p))

    features = np.array([Area, density, std_Density,
        Skewness, Kurtosis,ENERGY, ENTROPY,
        MAX, mean_absolute_deviation, MEDIAN, MIN, RANGE, RMS, UNIFORMITY])
    return features
```

Please note that all the variables or 'features' that are explained above and below are all inside a function so that when it's time to use these global features by other groups like Texture and FFT, we can simply call the function for all the features.

In total these are 14 global features that we are using as the statistical features of our dataset. Every feature defines different characteristics of the image:

- Mean: Indicates how bright or dark the image is.

- Median: Separates high intensity level of pixel from low intensity value pixels.

- Min and Max: Finding out minimum and maximum value of the image. Min and Max filters are also used as Morphological filters.

- Standard Deviation: Indicates how much each value disperse from the mean value.

- Skewness: Skewness is the measure of the asymmetry of the probability distribution of random variable. In terms of image processing, it can be used to indicate darker and lighter, and glossier or matte surfaces.

- Kurtosis: Kurtosis is the measure of whether the data is peaked or flat relative to a normal distribution. In DIP, kurtosis plays part in the noise and resolution measurement.

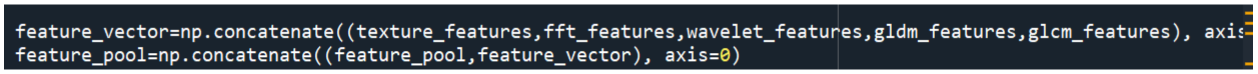
- Entropy: Entropy is a measure of image information content, which is interpreted as the average uncertainty of information source. The entropy value is used as it provides better comparison of the image details.

- Energy: Energy shows how the gray levels are distributed. Low gray level means the energy is high. Basically we can say that energy is defined on a normalized histogram of the image.

These and more other features like RMS, Uniformity also play statistical part in image processing and really helps in computing other features in spatial and frequency domains which we will discuss more as we go through the other aspects of our projects which includes Prime Features, Feature Evaluation, etc.

Prime Features:

We used a scheme to compute a total of 252 texture features in both the spatial and frequency domain. We categorized them into five groups; Histogram based Texture Features, Gray-Level Co-Occurrence Matrix (GLCM) , Gray Level Difference Method (GLDM)], Fast Fourier Transform (FFT), and Wavelet transform. We implemented GLCM and GLDM methods in four different directions, and Wavelet transforms in eight sub-bands. For each group or each subsection, we computed 14 features by applying the same statistical measures. The 14 features we measured in every group consisted of Mean, Std, Skewness, Kurtosis, Energy, Entropy, Max, Min, Mean Deviation, Median, Range, RMS, Uniformity, MeanGradient, and StdGradient. The feature extraction scheme resulted in 252 features for each X-ray image in total (14 features from Histogram based Texture, 14 features from FFT, 56 features from GLCM, 56 features from GLDM, and 112 features from Wavelet).



```
feature_vector=np.concatenate((texture_features,fft_features,wavelet_features,gldm_features,glcm_features), axis
feature_pool=np.concatenate((feature_pool,feature_vector), axis=0)
```

Features in the Frequency Domain

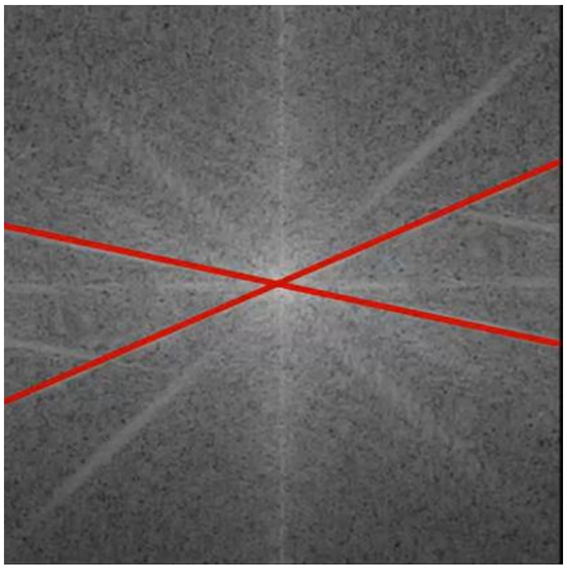
Basically frequency domain represents the rate of change of spatial pixels and hence gives an advantage when the problem you are dealing with relates to the rate of change of pixels which is very important in image processing. For example: high frequency in the frequency domain represents rapidly or sharply changing pixels such as boundaries or edges in an image. A high pass filter can be extremely helpful in identifying or removing these edges easily but the same problem is much more difficult in spatial domain (x-y domain). Similarly, a simple low pass filter can be used to get a smoother image.
It will be better for the model to analyze image in the frequency domain rather than time domain.

Fast Fourier Transform:

Fourier Transform decomposes an image into its real and imaginary components which is a representation of the image in the frequency domain. If the input signal is an image, then the number of frequencies in the frequency domain is equal to the number of pixels in the image or spatial domain. The inverse transform re-transforms the frequencies to the image in the spatial domain.
The reason of implementing FFT in digital images in simple terms is simply because computer can understand better in frequency domain than in time domain.

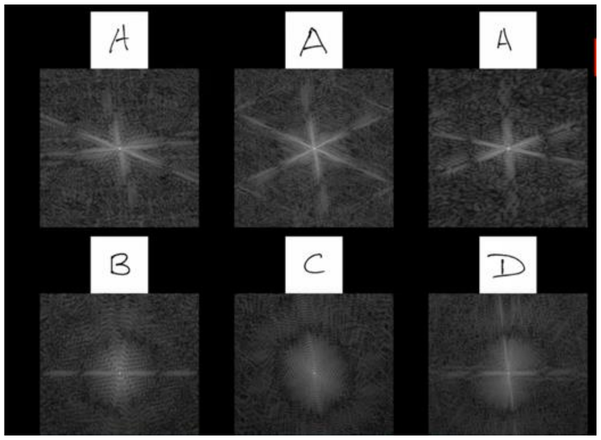
Considering an example:

Looking at this image, we humans can easily see that there’s a person standing taking a picture of something, but computer doesn’t work like humans do. Computer has its own level of understanding which is far different than how humans performs. Computer can perform better in frequency domain, if we apply FFT to the above image we obtain result which is something like this.

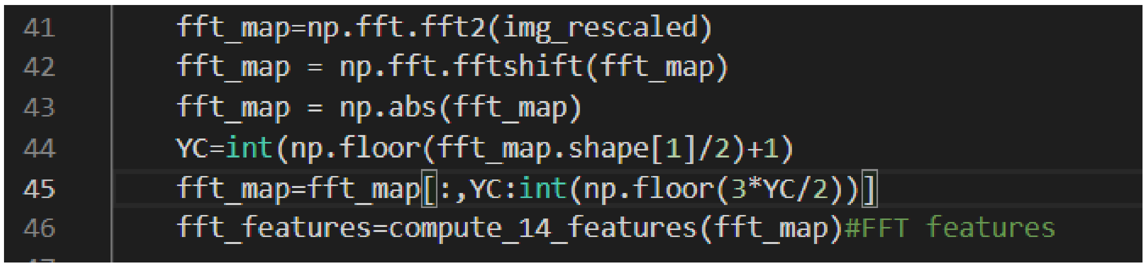


Here this image is really difficult for us to process because this is in frequency domain but for computer it’s relatively easy to interpret and find the patterns in the image. FFT is mainly used in image processing to process images in frequency domain so that AI models can work better on that.

Some more examples of pre-processing in FFT is text recognition (OCR)

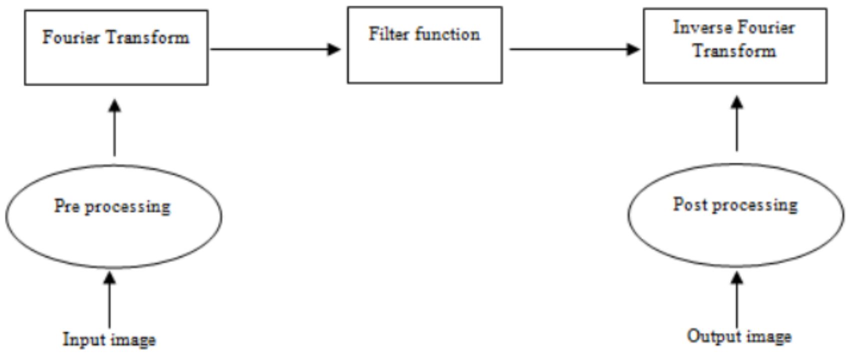


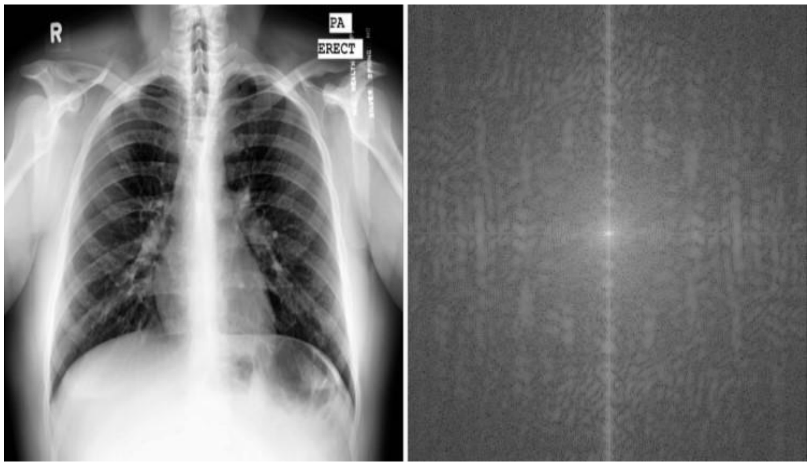
It will be much easier for neural model to interpret and play with this dataset rather than in time domain.



```
    fft_map=np.fft.fft2(img_rescaled)
    fft_map = np.fft.fftshift(fft_map)
    fft_map = np.abs(fft_map)
    YC=int(np.floor(fft_map.shape[1]/2)+1)
    fft_map=fft_map[:,YC:int(np.floor(3*YC/2))]
    fft_features=compute_14_features(fft_map)#FFT features
```

Here in this part of code in extract features file, we are applying FFT by using numpy library and then shifting the fourier frequency. Applying absolute on shifted frequency eliminates the negative noise from the image, which is useless for processing, so we get the desired output image we intend to work with. Filters are also be applied to reduce noise, this can also be done using FFT and will be very helpful for computers. Then we pass these outputs to compute_14_features which is explained before.





FOURIER TRANSFORM ON RIGHT

Wavelet

The conventional discrete wavelet transforms (DWT) may be regarded as equivalent to filtering the input signal with a bank of bandpass filters, whose impulse responses are all approximately given by scaled versions of a mother wavelet.

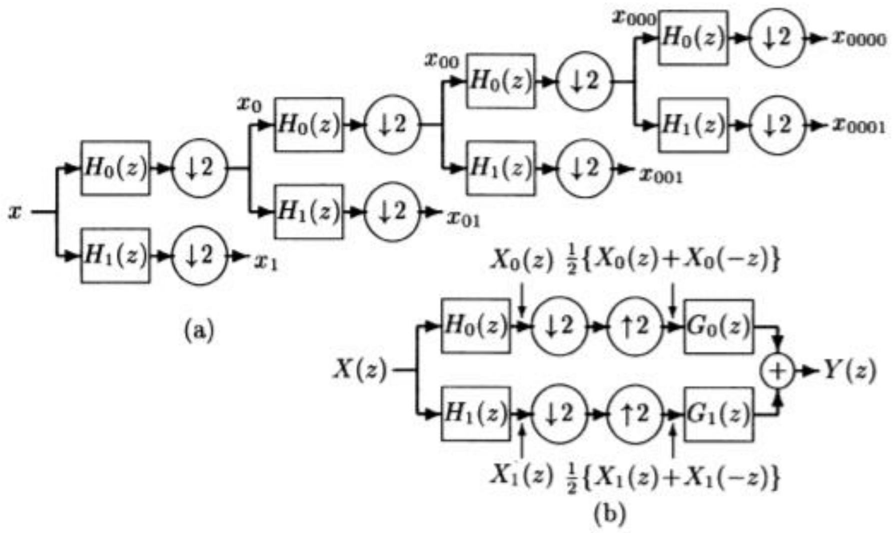
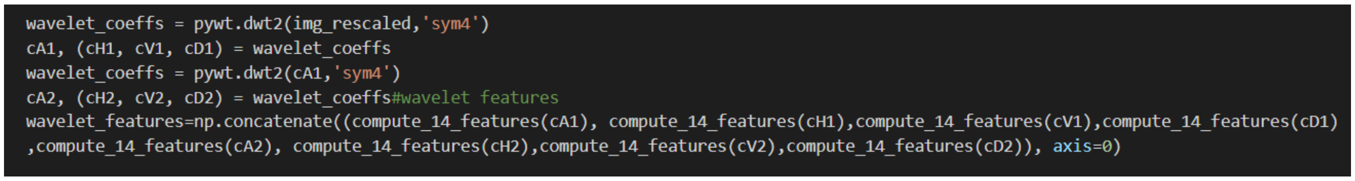


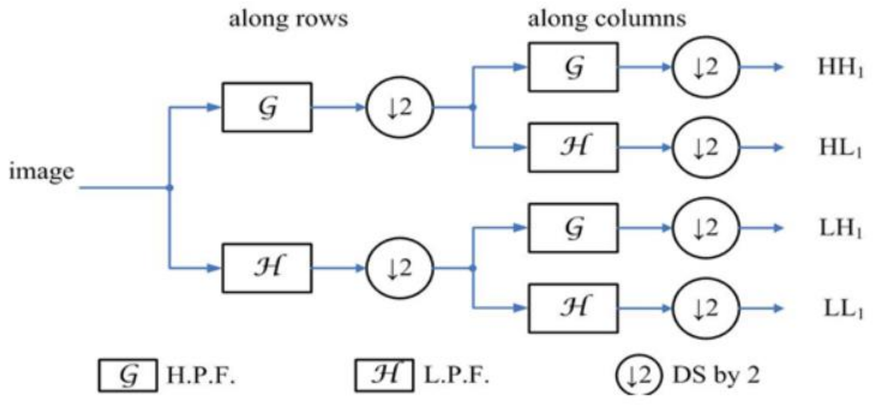
FIGURE 2.1. (a) Binary wavelet tree of lowpass (H_0) and highpass (H_1) filters; and (b) the 2-band reconstruction block

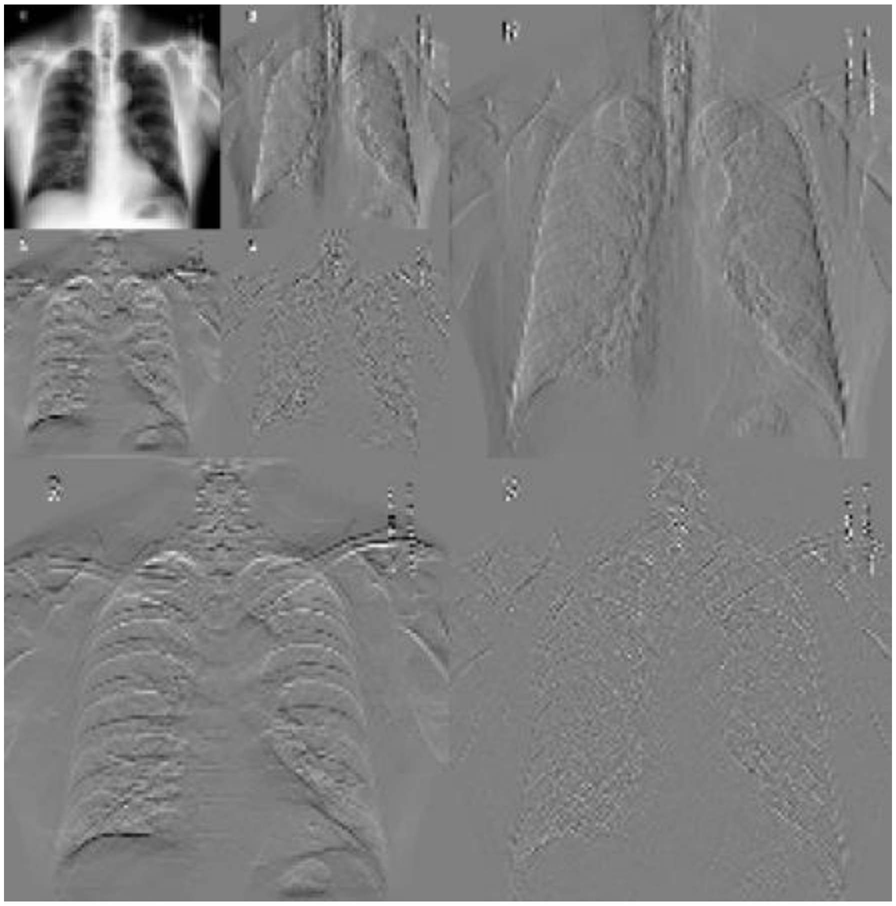
This picture is cited from Signal Analysis and Prediction pp 27-46



```
wavelet_coeffs = pywt.dwt2(img_rescaled,'sym4')
cA1, (cH1, cV1, cD1) = wavelet_coeffs
wavelet_coeffs = pywt.dwt2(cA1,'sym4')
cA2, (cH2, cV2, cD2) = wavelet_coeffs#wavelet features
wavelet_features=np.concatenate((compute_14_features(cA1), compute_14_features(cH1),compute_14_features(cV1),compute_14_features(cD1)
,compute_14_features(cA2), compute_14_features(cH2),compute_14_features(cV2),compute_14_features(cD2)), axis=0)
```

Here image is fed into the wavelet function and on the basis of high pass filter and low pass filter it's generating output values which is then inserted in compute_14_features which is explained above. The working of how filters are applied can be shown in the image below.





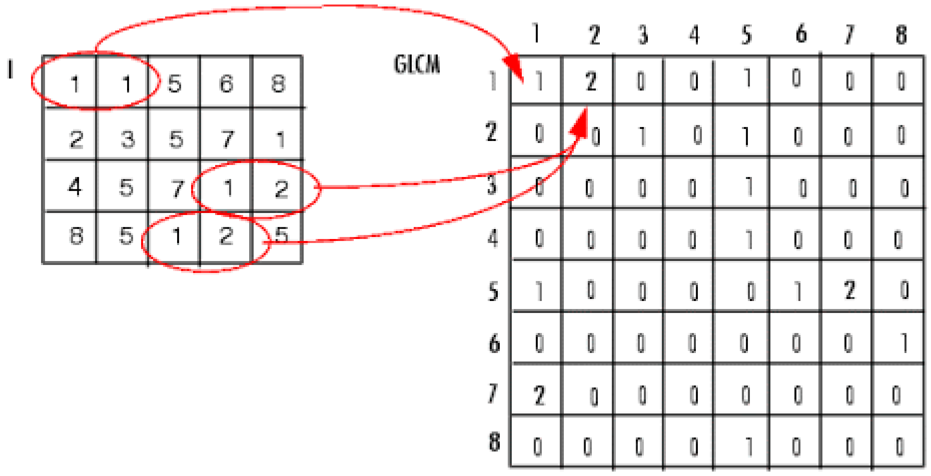
WAVELET TRANSFORMATION ANALYSIS

Features in the Spatial Domain

For simplicity, assume that the image I being considered is formed by projection from scene S (which might be a two- or three-dimensional scene, *etc.*). The *spatial domain* is the normal image space, in which a change in position in I directly projects to a change in position in S. Distances in I (in pixels) correspond to real distances (*e.g.* in meters) in S. This concept is used most often when discussing the frequency with which image values change, that is, over how many pixels does a cycle of periodically repeating intensity variations occur. One would refer to the number of pixels over which a pattern repeats (its periodicity) in the spatial domain. We have calculated texture-based image features in the spatial domain which include GLDM and GLCM methods. In spatial domain, we deal with images as it is. The value of the pixels of the image change with respect to scene. Whereas in frequency domain, we deal with the rate at which the pixel values are changing in spatial domain.

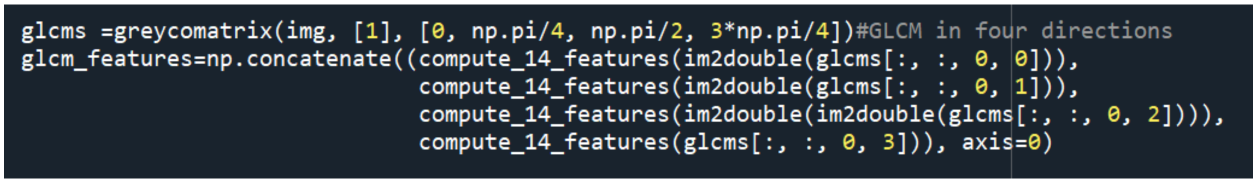
GLCM

Gray level co-occurrence matrix (GLCM) is a popular texture-based feature extraction method. The GLCM determines the textural relationship between pixels by performing an operation according to the second-order statistics in the images. Usually two pixels are used for this operation. The GLCM determines the frequency of combinations of these pixel brightness values determined. That is, it represents the frequency formation of the pixel pairs. The GLCM properties of an image are expressed as a matrix with the same number of rows and columns as the gray values in the image. The elements of this matrix depend on the frequency of the two specified pixels. Both pixel pairs can vary depending on their neighborhood.
Below is an example of the co-occurence matrix:



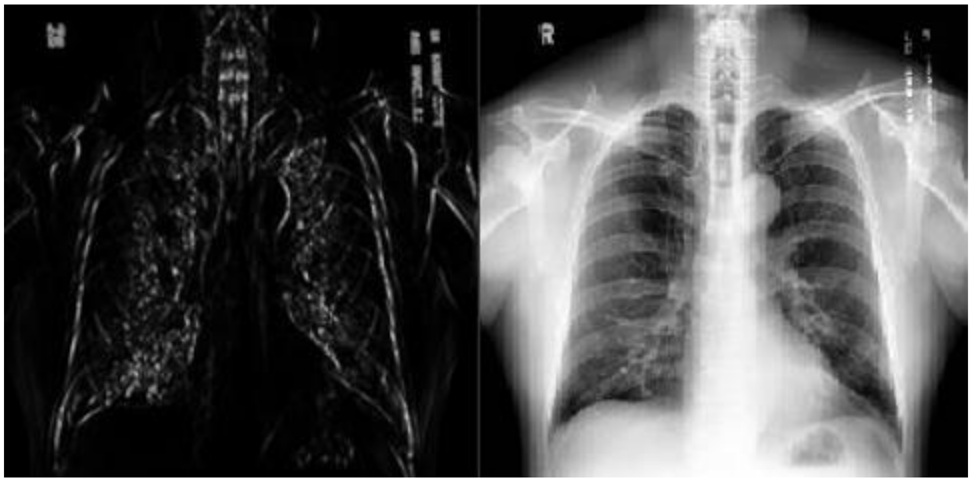
Prediction performance of all the 14 features were assessed in GLCM but none of them made it to the top predictors.
A set of 14 descriptors were calculated from each co-occurrence matrix evaluated at h={0 degree, 45 degree, 90 degree, 135 degree} and with distance d=[1]. A descriptor is obtained

by concatenating the features extracted for each distance and orientation value to prevent directional bias.



```
glcms =greycomatrix(img, [1], [0, np.pi/4, np.pi/2, 3*np.pi/4])#GLCM in four directions
glcm_features=np.concatenate((compute_14_features(im2double(glcms[:, :, 0, 0])),
                              compute_14_features(im2double(glcms[:, :, 0, 1])),
                              compute_14_features(im2double(im2double(glcms[:, :, 0, 2]))),
                              compute_14_features(glcms[:, :, 0, 3])), axis=0)
```

greycomatrix(image, distances, angles) calculates the grey-level co-occurrence matrix. distances List of pixel pair distance offsets.
angles : List of pixel pair angles in radians.

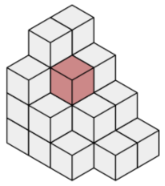


AN IMAGE WITH ITS GLCM ON LEFT

GLDM

GLDM (Gray Level Dependence Matrix) calculates the Gray level Difference Method Probability Density Functions for the given image. It quantifies gray level dependencies in an image. This technique is usually used for extracting statistical texture features of a digital mammogram.

A gray level dependency is defined as the number of connected voxels within distance d that are dependent on the center voxel. A neighbouring voxel with gray level j is considered dependent on center voxel with gray level i, if $|i-j| \le \alpha$. In a gray level dependence matrix P(i,j) the (i,j)th element describes the number of times a voxel with gray level i with j dependent voxels in its neighbourhood appears in image.



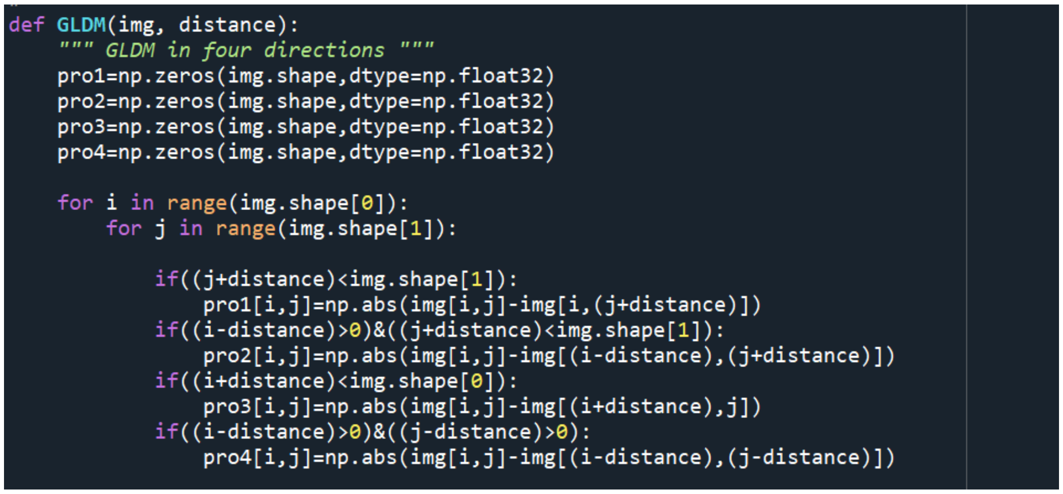
A series of voxels in a stack, with a single voxel shaded

As a two dimensional example, consider the following 5x5 image, with 5 discrete gray levels:

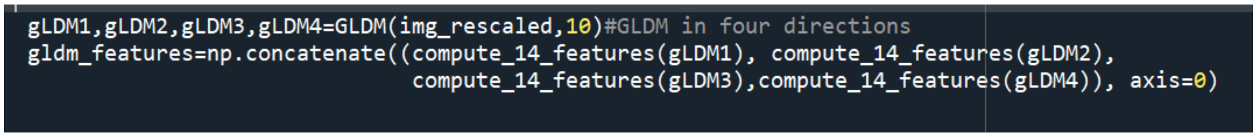
$$\mathbf{I} = \begin{bmatrix} 5 & 2 & 5 & 4 & 4 \\ 3 & 3 & 3 & 1 & 3 \\ 2 & 1 & 1 & 1 & 3 \\ 4 & 2 & 2 & 2 & 3 \\ 3 & 5 & 3 & 3 & 2 \end{bmatrix}$$

For $\alpha=0$ and d=1, the GLDM then becomes:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 1 & 2 & 3 & 0 \\ 1 & 4 & 4 & 0 \\ 1 & 2 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$



```python
def GLDM(img, distance):
    """ GLDM in four directions """
    pro1=np.zeros(img.shape,dtype=np.float32)
    pro2=np.zeros(img.shape,dtype=np.float32)
    pro3=np.zeros(img.shape,dtype=np.float32)
    pro4=np.zeros(img.shape,dtype=np.float32)

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):

            if((j+distance)<img.shape[1]):
                pro1[i,j]=np.abs(img[i,j]-img[i,(j+distance)])
            if((i-distance)>0)&((j+distance)<img.shape[1]):
                pro2[i,j]=np.abs(img[i,j]-img[(i-distance),(j+distance)])
            if((i+distance)<img.shape[0]):
                pro3[i,j]=np.abs(img[i,j]-img[(i+distance),j])
            if((i-distance)>0)&((j-distance)>0):
                pro4[i,j]=np.abs(img[i,j]-img[(i-distance),(j-distance)])
```



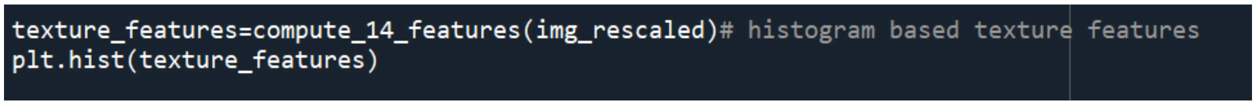
```python
gLDM1,gLDM2,gLDM3,gLDM4=GLDM(img_rescaled,10)#GLDM in four directions
gldm_features=np.concatenate((compute_14_features(gLDM1), compute_14_features(gLDM2),
                              compute_14_features(gLDM3),compute_14_features(gLDM4)), axis=0)
```

A set of 14 descriptors were calculated from each dependence matrix evaluated at h={0 degree, 45 degree, 90 degree, 135 degree} and with distance d=[10], this specifies the distances between the center voxel and the neighbor.A descriptor is obtained by

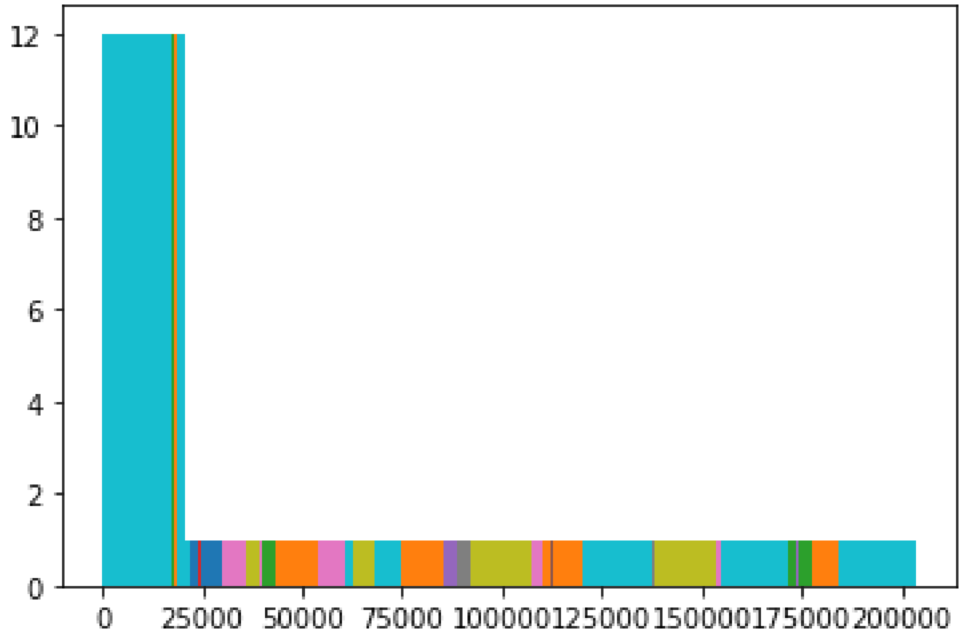
concatenating the features extracted for each distance and orientation value to prevent directional bias.
MeanDeviation_GLDM made to the top predictors.

Histogram based Texture Features

Histogram features are first-order statistics based features, which are used to represent surface texture. According to Srinivasan and Shobha [10], histogram based features represent intensity concentration on all parts of the image. The statistical texture features are considered useful for classification and retrieval of similar images. These texture features provide information about the properties of the intensity level distribution in the image like uniformity, smoothness, flatness and contrast. The statistical texture features of mean, standard deviation, skewness, kurtosis, energy, entropy and smoothness are calculated by using the probability distribution of the intensity levels in the histogram bins of the histograms.

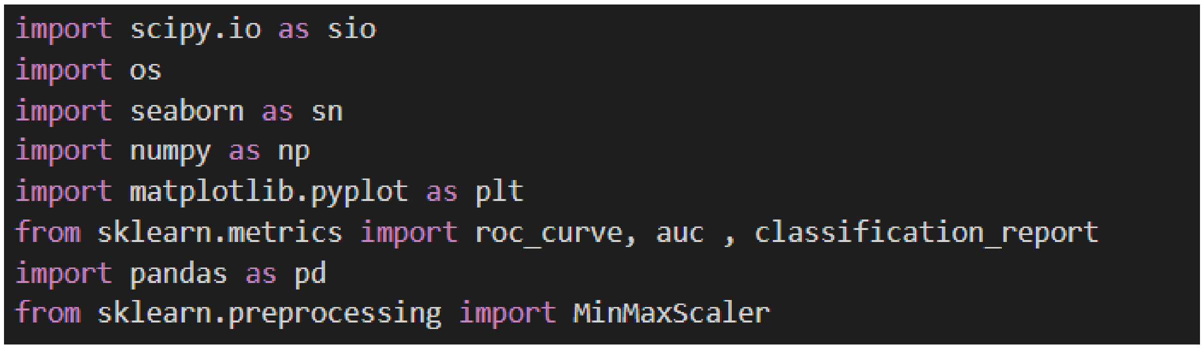


```
texture_features=compute_14_features(img_rescaled)# histogram based texture features
plt.hist(texture_features)
```



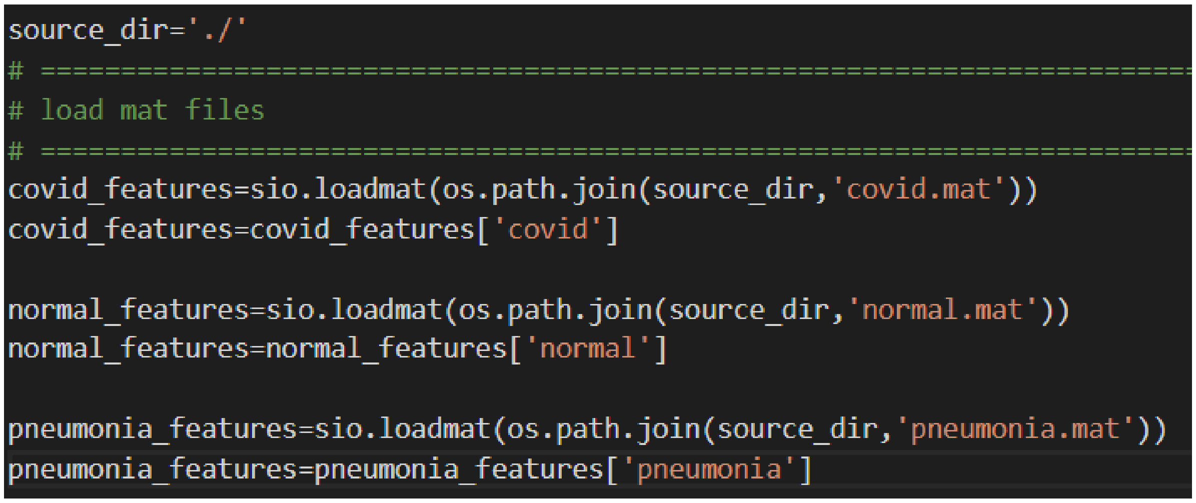
The x-axis of the histogram denotes the number of bins while the y-axis represents the frequency of a particular bin.

Feature Evaluation



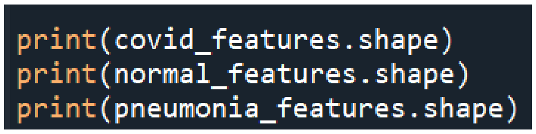
```
import scipy.io as sio
import os
import seaborn as sn
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc , classification_report
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
```

Firstly, we import the relevant libraries we need to use in order to evaluate features for our model from different images. We'll dive deeper into what is used where and why in their own respective sections.

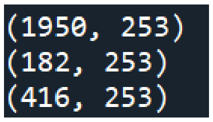


```
source_dir='./'
# ========================================================================
# load mat files
# ========================================================================
covid_features=sio.loadmat(os.path.join(source_dir,'covid.mat'))
covid_features=covid_features['covid']

normal_features=sio.loadmat(os.path.join(source_dir,'normal.mat'))
normal_features=normal_features['normal']

pneumonia_features=sio.loadmat(os.path.join(source_dir,'pneumonia.mat'))
pneumonia_features=pneumonia_features['pneumonia']
```

Here in this dataset we can see that we have 1950 x 253 matrix for covid, 182 x 253 for normal and 416 x 253 for pneumonia class. Clearly, the rows represent number of training examples and the columns represents number of features.

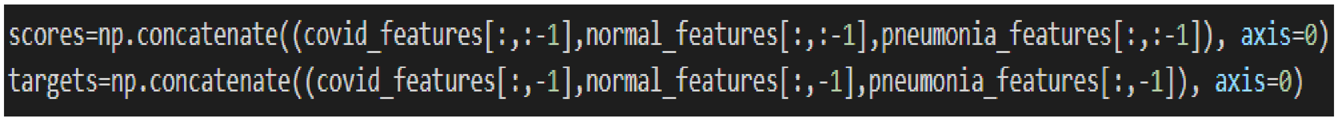


```
print(covid_features.shape)
print(normal_features.shape)
print(pneumonia_features.shape)
```



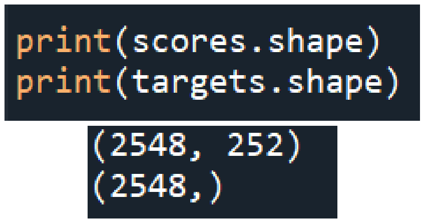
```
(1950, 253)
(182, 253)
(416, 253)
```

We get three arrays with their respective dimensions.



```
scores=np.concatenate((covid_features[:,:-1],normal_features[:,:-1],pneumonia_features[:,:-1]), axis=0)
targets=np.concatenate((covid_features[:,-1],normal_features[:,-1],pneumonia_features[:,-1]), axis=0)
```

To be precise, what we've calculated is the entire dataset of features. From this we've to separate labels or targets and features of input. For this, we know out of 253 columns for each training example, the last column is the labels column for this training set. What we do is we separate the last column of all three arrays into a single array and make that target for our dataset. The rest of the dataset is concatenated with the exclusion of last column. Hence, we get the following shapes of our features (variable's named as scores in code) and targets.



```
print(scores.shape)
print(targets.shape)
```

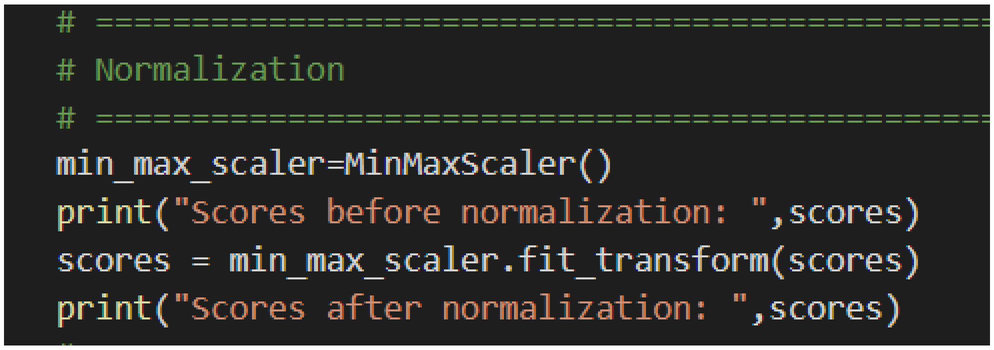
```
(2548, 252)
(2548,)
```

Normalization

Variables that are measured at different scales do not contribute equally to the model fitting, so we need to normalize features for each feature to be on the same scale. For example: We are training for Real Estate Prediction model (the most basic problem in machine learning), theoretically considering, we have two features:

1. Year in which constructed.
2. Price

The scale of these two features are far apart from each other. The year will be in the scale of (1000s) and price will most likely to be in (1,000,000) or hundreds of millions. For this problem to cater, we need to have on one page for all features or in simple terms, in one range for all features. For this purpose, we will use normalization.

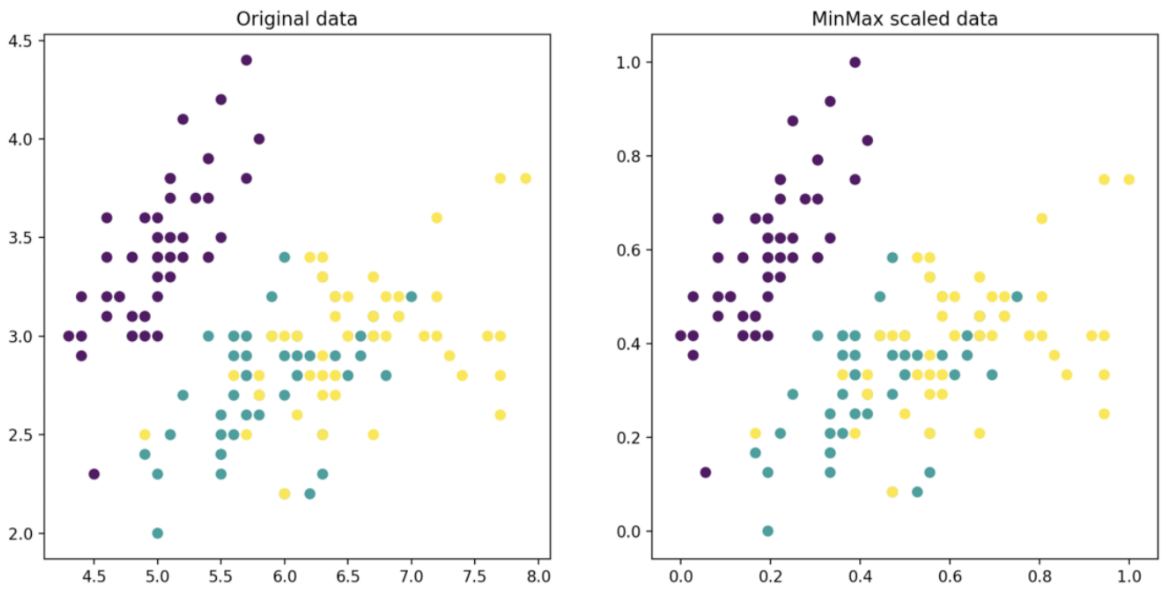


```
# ===============================================
# Normalization
# ===============================================
min_max_scaler=MinMaxScaler()
print("Scores before normalization: ",scores)
scores = min_max_scaler.fit_transform(scores)
print("Scores after normalization: ",scores)
```

We are using MinMaxScalar**()** here. By doing so, all features will be transformed into the range [0,1] meaning that the minimum and maximum value of a feature/variable is going fto be 0 and 1, respectively.

$$x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

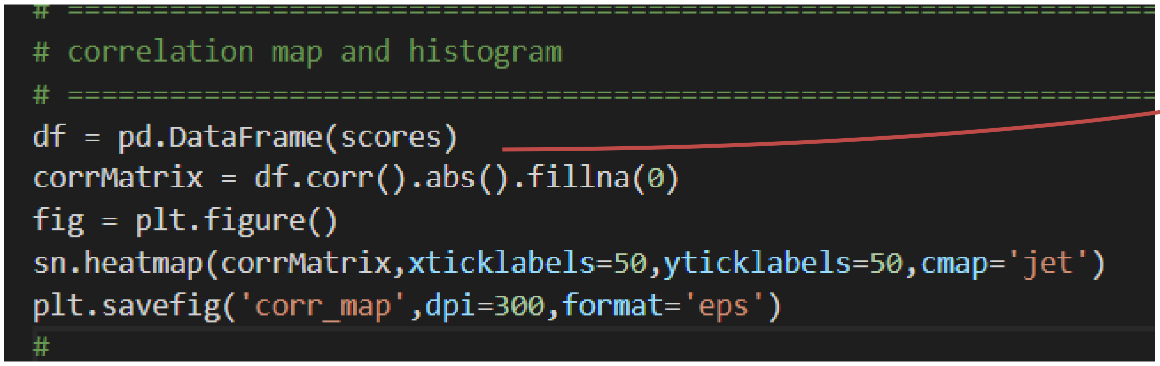
Mathematical formulation



The MinMax scaling effect on the first 2 features of the Iris dataset. Figure produced by the author in Python [1]

Correlation and Histogram

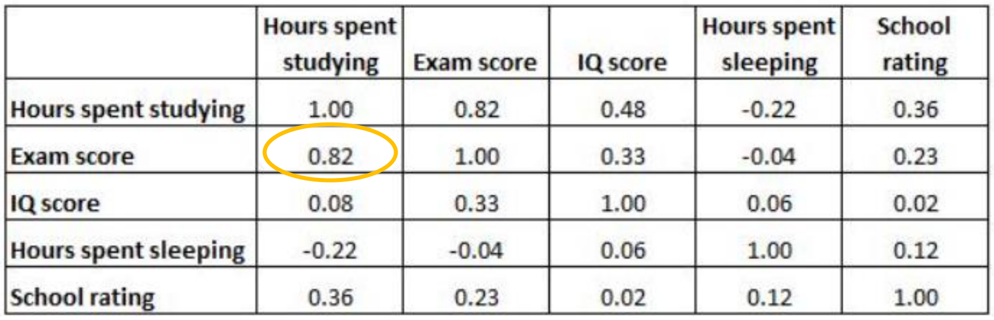
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. We have converted the scores data into tabular form using **pandas**.



```
# ==================================================================
# correlation map and histogram
# ==================================================================
df = pd.DataFrame(scores)
corrMatrix = df.corr().abs().fillna(0)
fig = plt.figure()
sn.heatmap(corrMatrix,xticklabels=50,yticklabels=50,cmap='jet')
plt.savefig('corr_map',dpi=300,format='eps')
#
```

Correlation

A correlation matrix is simply a table which displays the correlation coefficients for different variables. The matrix depicts the correlation between all the possible pairs of values in a table. It is a powerful tool to summarize a large dataset and to identify and visualize patterns in the given data.

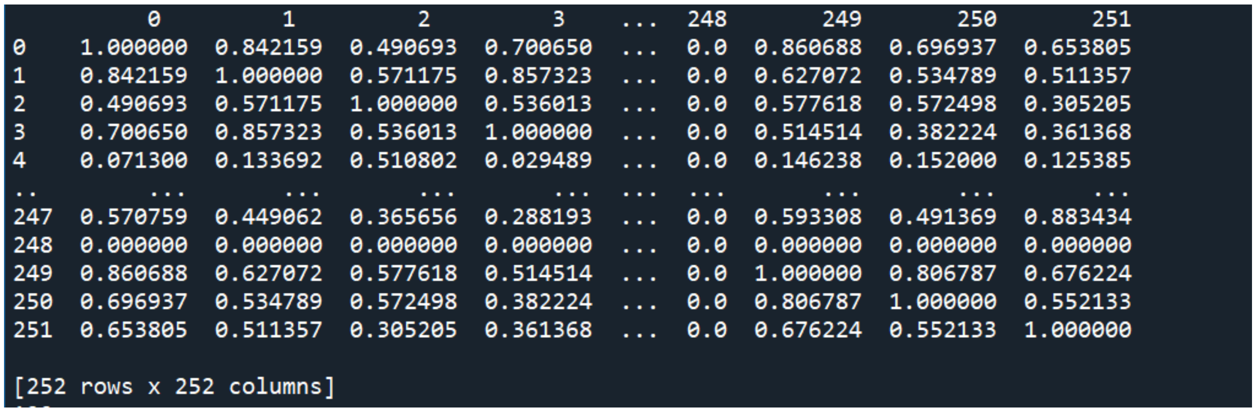


	Hours spent studying	Exam score	IQ score	Hours spent sleeping	School rating
Hours spent studying	1.00	0.82	0.48	-0.22	0.36
Exam score	0.82	1.00	0.33	-0.04	0.23
IQ score	0.08	0.33	1.00	0.06	0.02
Hours spent sleeping	-0.22	-0.04	0.06	1.00	0.12
School rating	0.36	0.23	0.02	0.12	1.00

Here the correlation between "hours spent studying" and "exam score" is 0.82, which indicates that they're strongly positively correlated. More hours spent studying is strongly related to higher exam scores. [2]

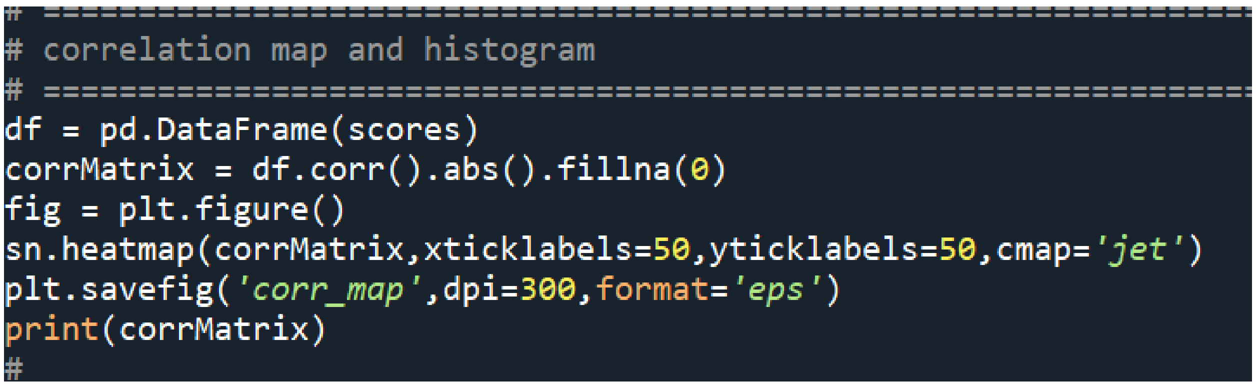
This is the simple example of human understandable data but when it comes to image processing, it's not feasible for humans to understand and analyze the pixels of image values and correlated them that's why a much simpler example is used here to define why correlation is being used.

In this code, correlated matrix describes how each feature is related with other features.

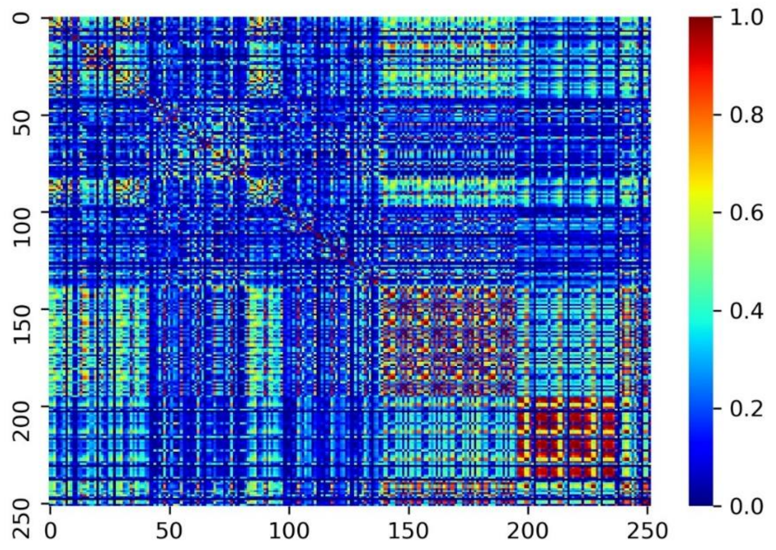


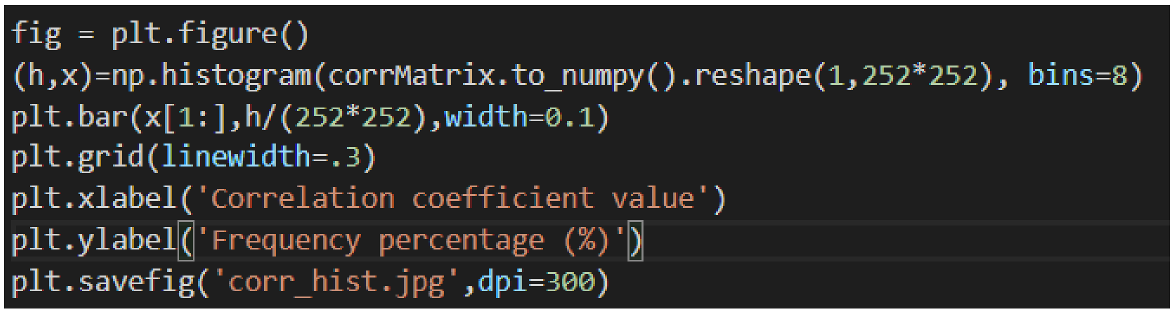
```
          0         1         2         3   ...  248       249       250       251
0    1.000000  0.842159  0.490693  0.700650  ...  0.0  0.860688  0.696937  0.653805
1    0.842159  1.000000  0.571175  0.857323  ...  0.0  0.627072  0.534789  0.511357
2    0.490693  0.571175  1.000000  0.536013  ...  0.0  0.577618  0.572498  0.305205
3    0.700650  0.857323  0.536013  1.000000  ...  0.0  0.514514  0.382224  0.361368
4    0.071300  0.133692  0.510802  0.029489  ...  0.0  0.146238  0.152000  0.125385
..        ...       ...       ...       ...  ...  ...       ...       ...       ...
247  0.570759  0.449062  0.365656  0.288193  ...  0.0  0.593308  0.491369  0.883434
248  0.000000  0.000000  0.000000  0.000000  ...  0.0  0.000000  0.000000  0.000000
249  0.860688  0.627072  0.577618  0.514514  ...  0.0  1.000000  0.806787  0.676224
250  0.696937  0.534789  0.572498  0.382224  ...  0.0  0.806787  1.000000  0.552133
251  0.653805  0.511357  0.305205  0.361368  ...  0.0  0.676224  0.552133  1.000000

[252 rows x 252 columns]
```

Computers will do much better job in understanding this correlated matrix than humans.



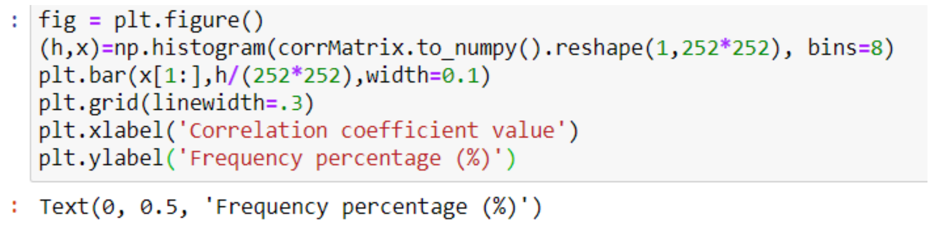
```
# =================================================================
# correlation map and histogram
# =================================================================
df = pd.DataFrame(scores)
corrMatrix = df.corr().abs().fillna(0)
fig = plt.figure()
sn.heatmap(corrMatrix,xticklabels=50,yticklabels=50,cmap='jet')
plt.savefig('corr_map',dpi=300,format='eps')
print(corrMatrix)
#
```



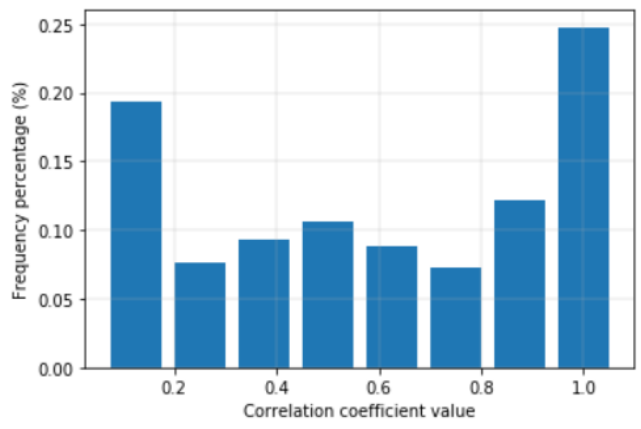


```
fig = plt.figure()
(h,x)=np.histogram(corrMatrix.to_numpy().reshape(1,252*252), bins=8)
plt.bar(x[1:],h/(252*252),width=0.1)
plt.grid(linewidth=.3)
plt.xlabel('Correlation coefficient value')
plt.ylabel('Frequency percentage (%)')
plt.savefig('corr_hist.jpg',dpi=300)
```

Now we are plotting the graph of correlation and we see how many percentages of features are dependent on other features.



```
: fig = plt.figure()
  (h,x)=np.histogram(corrMatrix.to_numpy().reshape(1,252*252), bins=8)
  plt.bar(x[1:],h/(252*252),width=0.1)
  plt.grid(linewidth=.3)
  plt.xlabel('Correlation coefficient value')
  plt.ylabel('Frequency percentage (%)')

: Text(0, 0.5, 'Frequency percentage (%)')
```



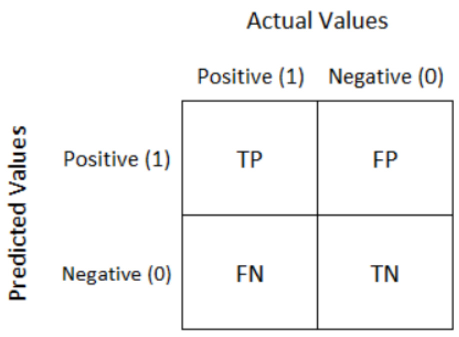
Here we can conclude that 25% of the features out of the pool of features are totally dependent on other feature meaning that tweaking this factor can through its effect on 25% of features. Similarly, all the appropriate % effect values are given in this graph.

AUC: Area Under the ROC Curve

Before diving into AUC, we need to explore some pre-requisites evaluation metrics terminologies and their definitions and how they work in order to better grasp the concept of AUC.

What is Confusion Matrix and why you need it?

It is a performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.

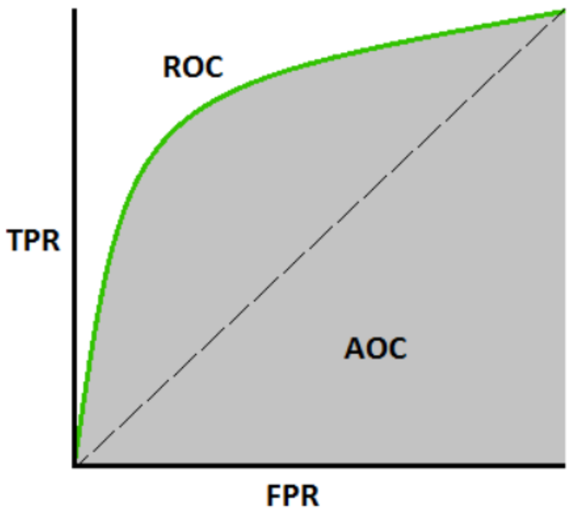


		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

True Positive: The model predicts positive and it’s true.
False Positive: The model predicts positive and it’s false.
True Negative: The model predicts negative and it’s true.
False Negative: The model predicts negative and it’s false.
It is tremendously useful for measuring Recall, Precision, Accuracy and most importantly AUC-ROC Curve

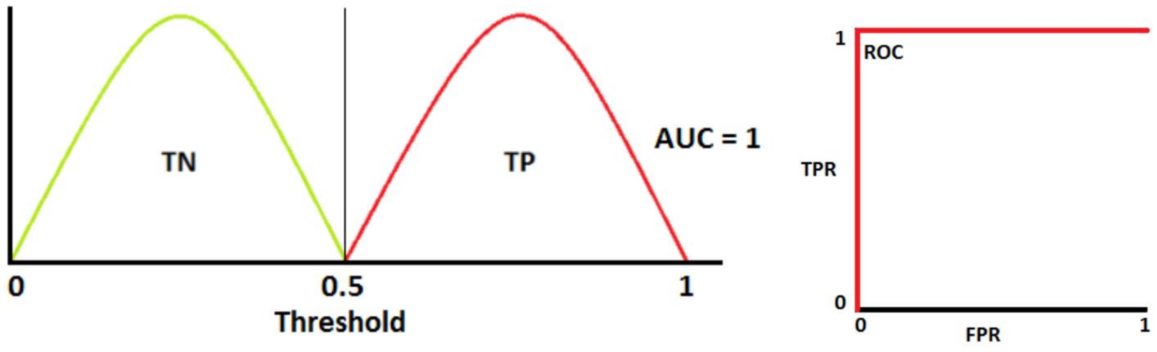
AUC-ROC Curve:

AUC - ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represent degree or measure of separability. It tells how much model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s.

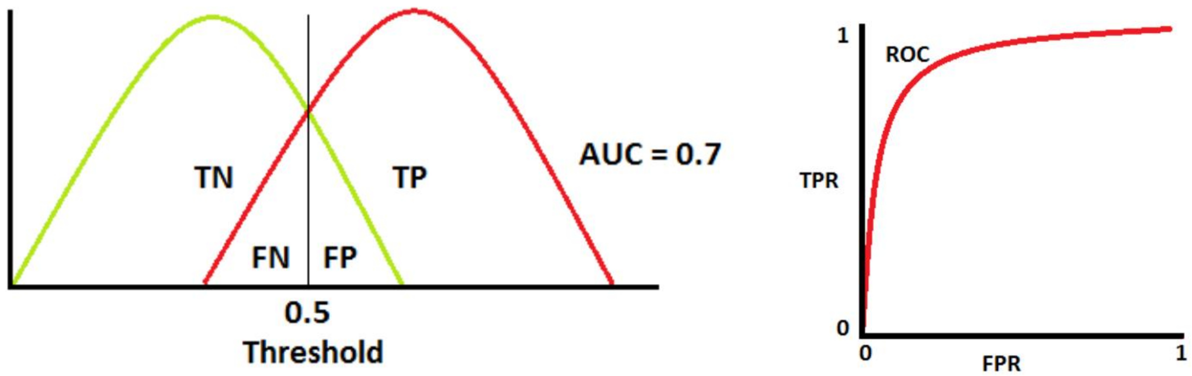


AUC - ROC Curve [3]

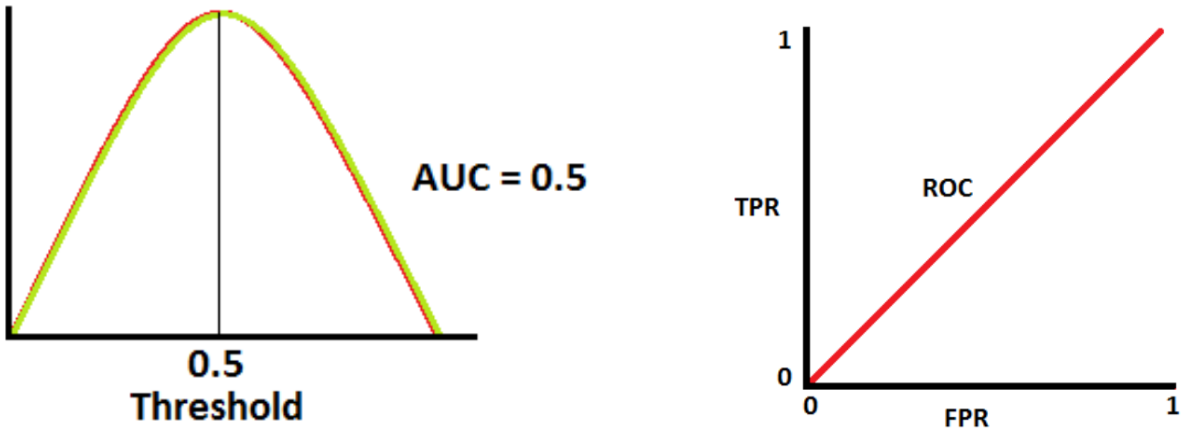
So this is the evaluation metric between true positive rates and false positive rates and the area under the curve between these two relations is called as AUC (Area under the curve)
As we know, ROC is a curve of probability. So, let's plot the distributions of those probabilities:
Note: Red distribution curve is of the positive class (Covid) and the green distribution curve is of the negative class(normal or pneumonia).



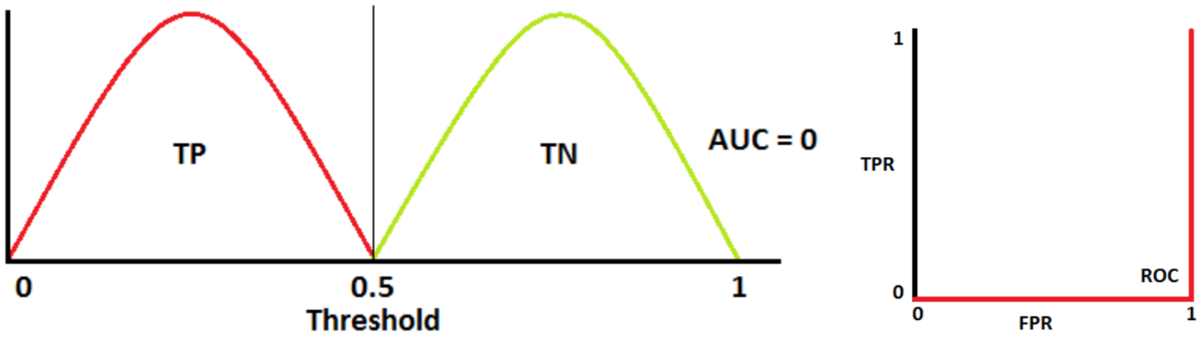
This is an ideal situation. When two curves don't overlap at all means model has an ideal measure of separability. It is perfectly able to distinguish between positive class and negative class.



When AUC is 0.7, it means there is a 70% chance that the model will be able to distinguish between positive class and negative class.

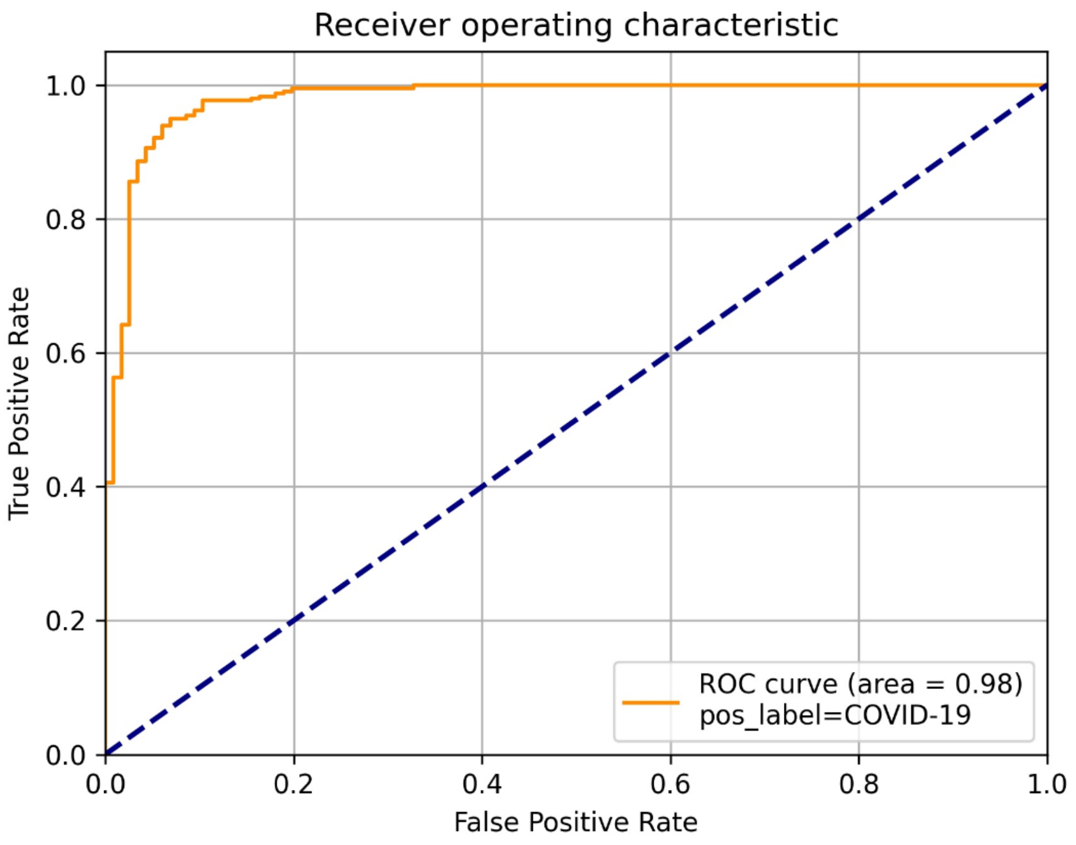


This is the worst situation. When AUC is approximately 0.5, the model has no discrimination capacity to distinguish between positive class and negative class.



When AUC is approximately 0, the model is actually reciprocating the classes. It means the model is predicting a negative class as a positive class and vice versa.[3]

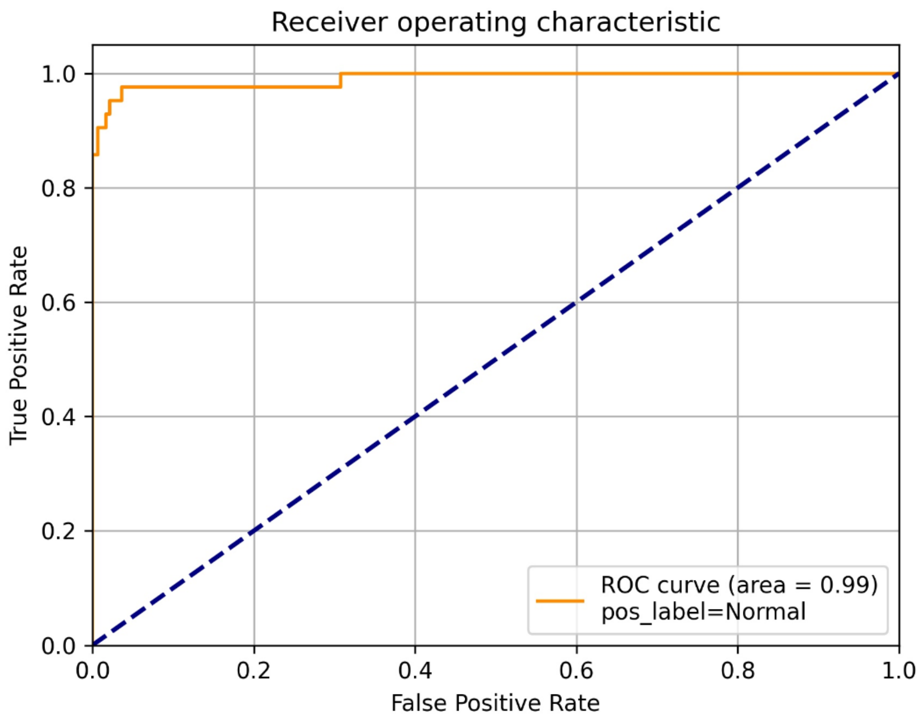
In our multiclass classifier, to study ROC curve we have to make each one positive just for the sake of AUC-ROC graph since it's multiclass classifier.



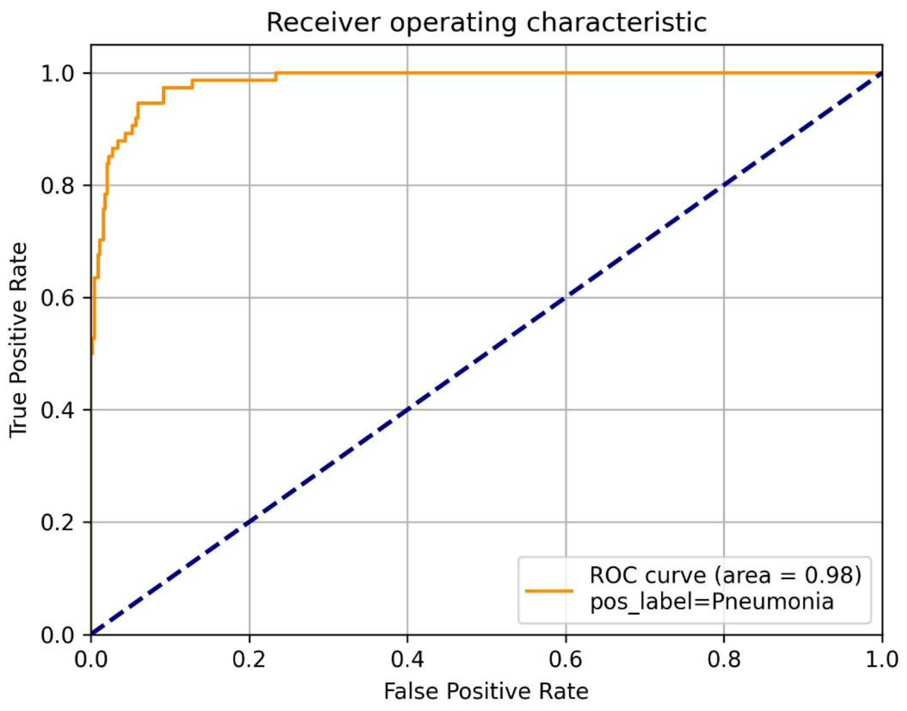
When positive label is COVID-19

Here in this graph, the area under the curve is 0.98 which is represented in the yellow color which shows that 98% of the time the model is correctly classifying features of COVID-19 and the blue line indicates that this is the bare minimum the model can reach(and it's predefined in sklearn library and is set to 0.5), this shows that any curve below this line will be considered as worst classifier because it will not have discrimination capacity to distinguish.

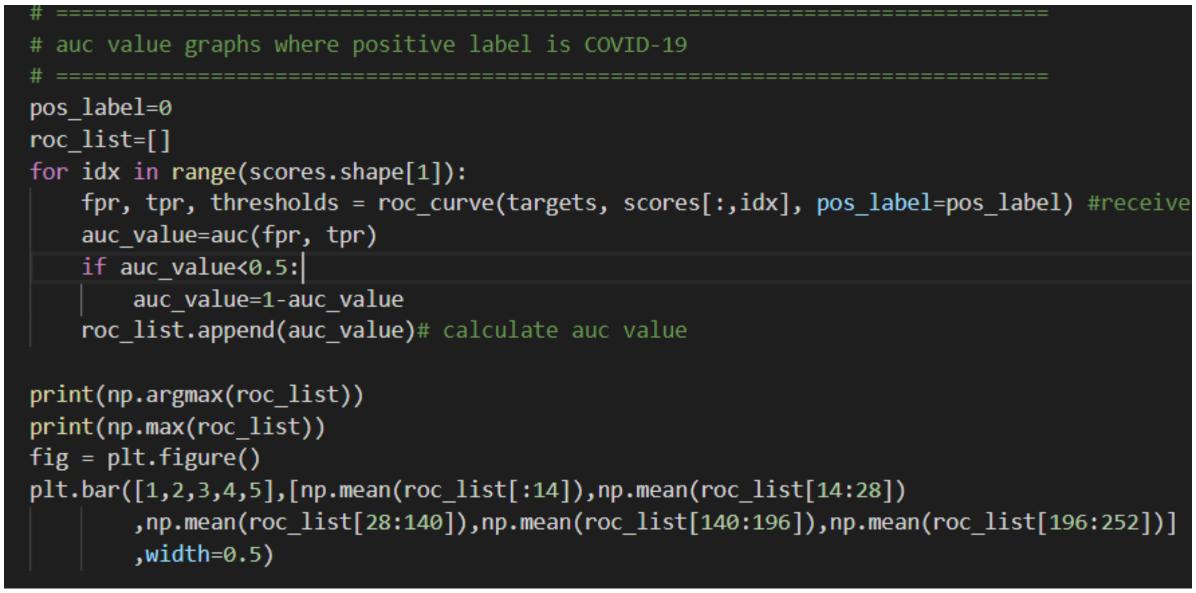
Similarly, we can justify this by showing graphs when positive label is normal and pneumonia.



When positive label is Normal

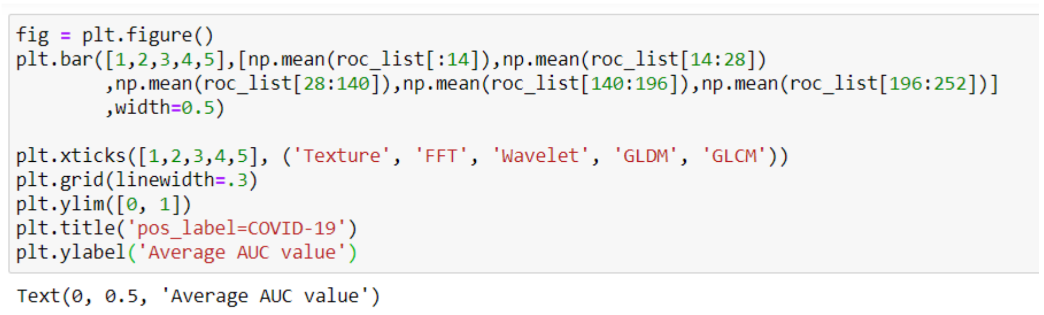


When positive label is Pneumonia



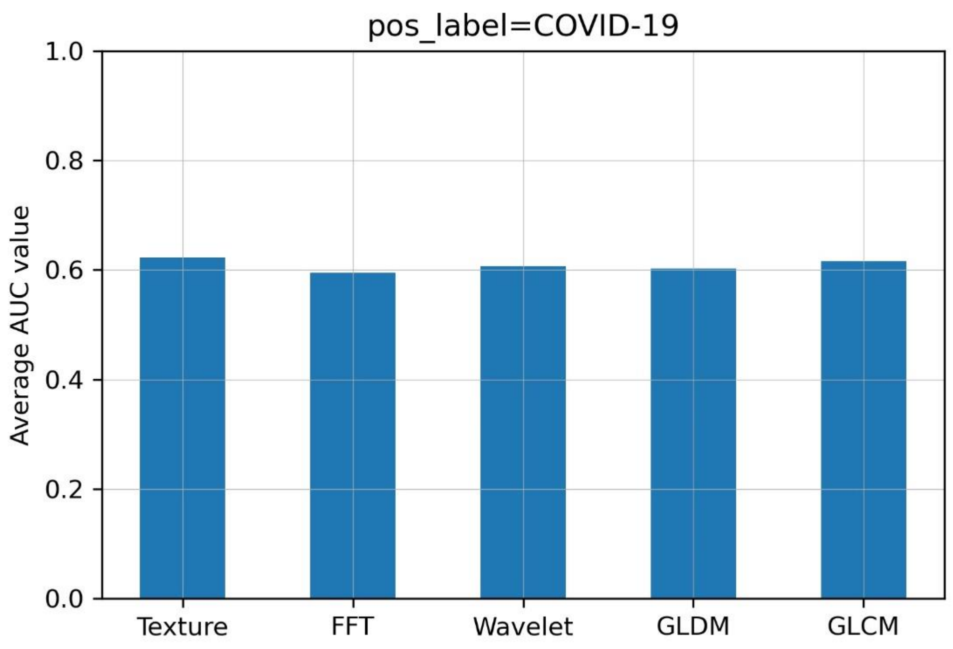
```
# ============================================================================
# auc value graphs where positive label is COVID-19
# ============================================================================
pos_label=0
roc_list=[]
for idx in range(scores.shape[1]):
    fpr, tpr, thresholds = roc_curve(targets, scores[:,idx], pos_label=pos_label) #receive
    auc_value=auc(fpr, tpr)
    if auc_value<0.5:
        auc_value=1-auc_value
    roc_list.append(auc_value)# calculate auc value

print(np.argmax(roc_list))
print(np.max(roc_list))
fig = plt.figure()
plt.bar([1,2,3,4,5],[np.mean(roc_list[:14]),np.mean(roc_list[14:28])
        ,np.mean(roc_list[28:140]),np.mean(roc_list[140:196]),np.mean(roc_list[196:252])]
        ,width=0.5)
```

Firstly, we are taking positive label as COVID-19 which means true positive rate and false positive rate is to be taken from COVID category only. Rest to be followed.
Here we are calculating AUC values for all the features and appending the result in roc_list and then have collection of 252 AUC values. AUC values are calculated by the function defined by sklearn library in python, for understanding purposes auc values can be calculated at different values of FPR, TPR in the graph. Following this, we are combining values into 5 pairs by taking mean of the AUC values of different collection. Like for the first plot bar, we are going to take 15 values and so it goes. The graph obtained is shown below:

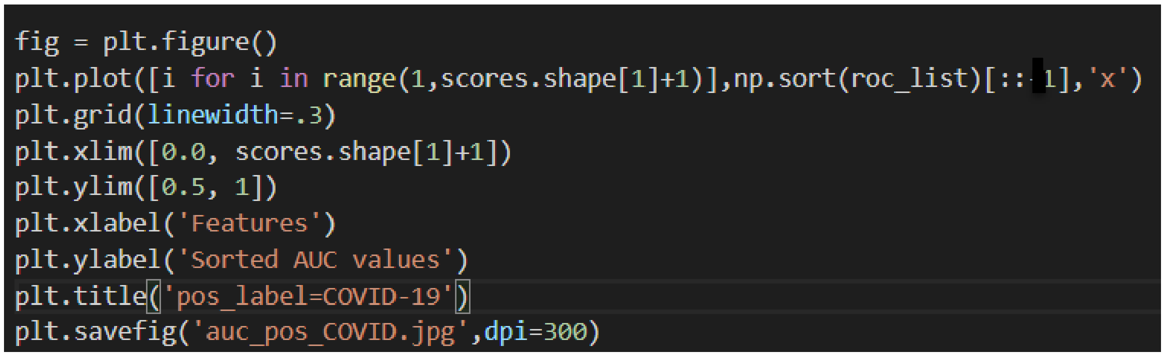


```
fig = plt.figure()
plt.bar([1,2,3,4,5],[np.mean(roc_list[:14]),np.mean(roc_list[14:28])
        ,np.mean(roc_list[28:140]),np.mean(roc_list[140:196]),np.mean(roc_list[196:252])]
        ,width=0.5)

plt.xticks([1,2,3,4,5], ('Texture', 'FFT', 'Wavelet', 'GLDM', 'GLCM'))
plt.grid(linewidth=.3)
plt.ylim([0, 1])
plt.title('pos_label=COVID-19')
plt.ylabel('Average AUC value')
```

```
Text(0, 0.5, 'Average AUC value')
```

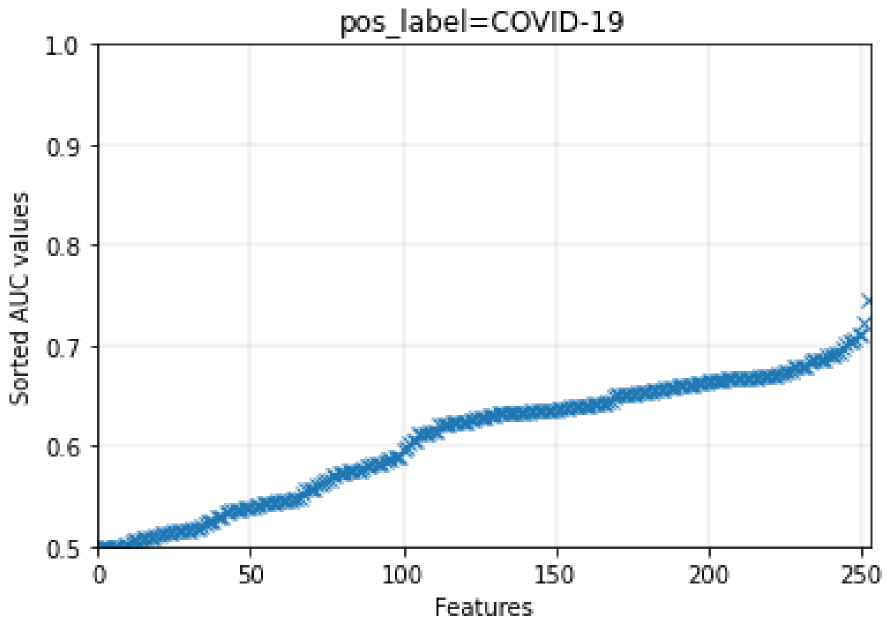


This tells us that most of the mean of auc value are around 0.6 which is a good sign. Since higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s.

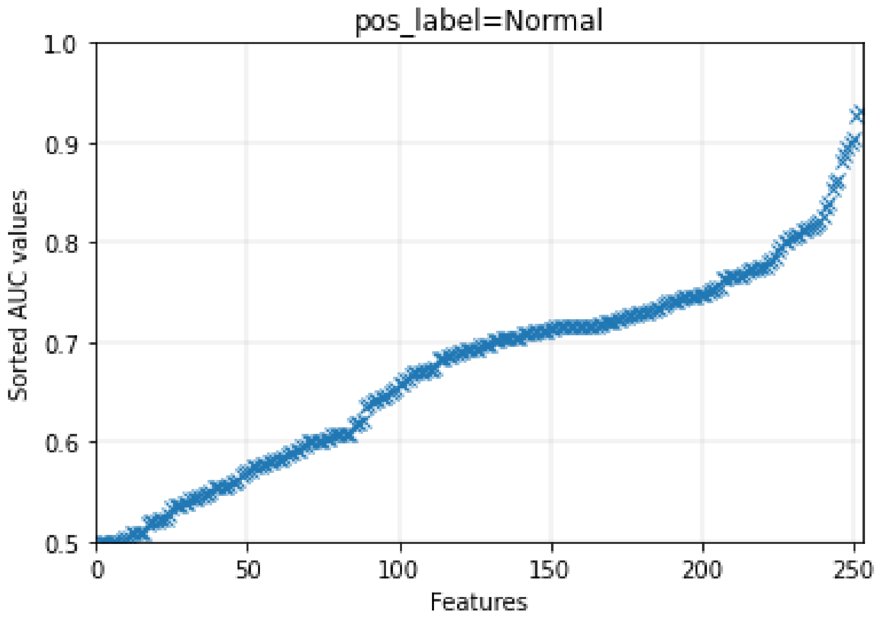


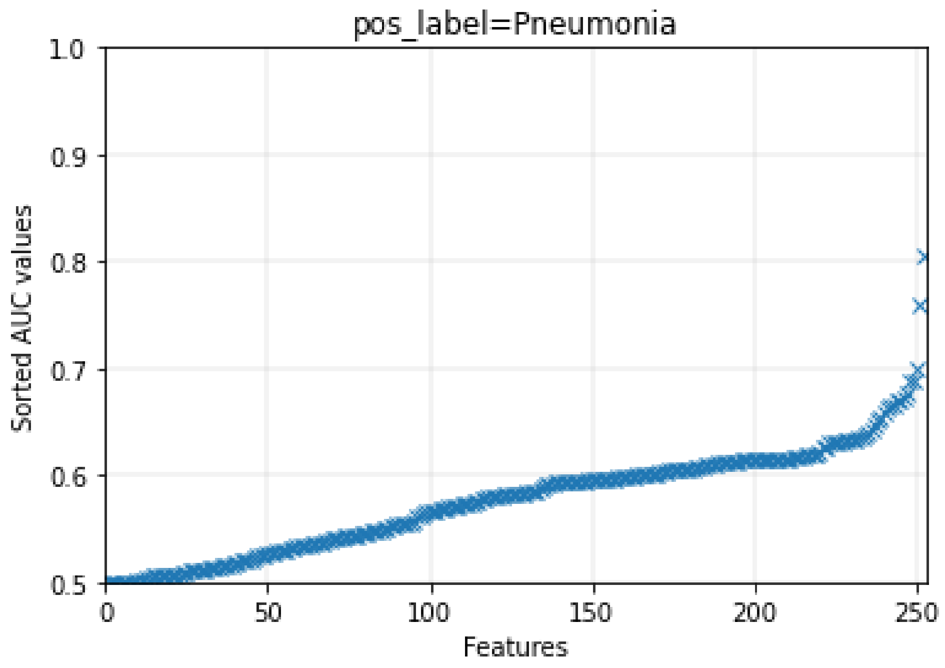
```
fig = plt.figure()
plt.plot([i for i in range(1,scores.shape[1]+1)],np.sort(roc_list)[::-1],'x')
plt.grid(linewidth=.3)
plt.xlim([0.0, scores.shape[1]+1])
plt.ylim([0.5, 1])
plt.xlabel('Features')
plt.ylabel('Sorted AUC values')
plt.title('pos_label=COVID-19')
plt.savefig('auc_pos_COVID.jpg',dpi=300)
```

Now we are going to plot the graph between AUC values and features. Remember that in previous part of the code, to obtain AUC values for the assignment of threshold to this algorithm we worked in incrementing order meaning that first we calculate auc value for first feature, then for first two features and then first three features and so on resulting in the last auc value we calculate is for the whole 252 features.



Here is the graph for AUC values throughout the features, so the cutoff is sorted out to be 0.75 started from 0.5 approximately when AUC value is calculated on first feature and the rest is assured by this graph.
This has done for the part where we considered COVID-19 as positive label and the rest two as negatives. Now we've to consider for each class, their AUC_ROC curve and its details.
The below are the graphs when normal and pneumonia is taken as positive label.

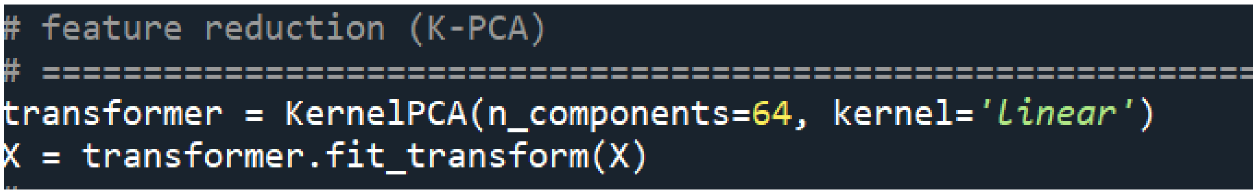



pos_label=Pneumonia
Sorted AUC values
Features
1.0
0.9
0.8
0.7
0.6
0.5
0
50
100
150
200
250

Feature Reduction

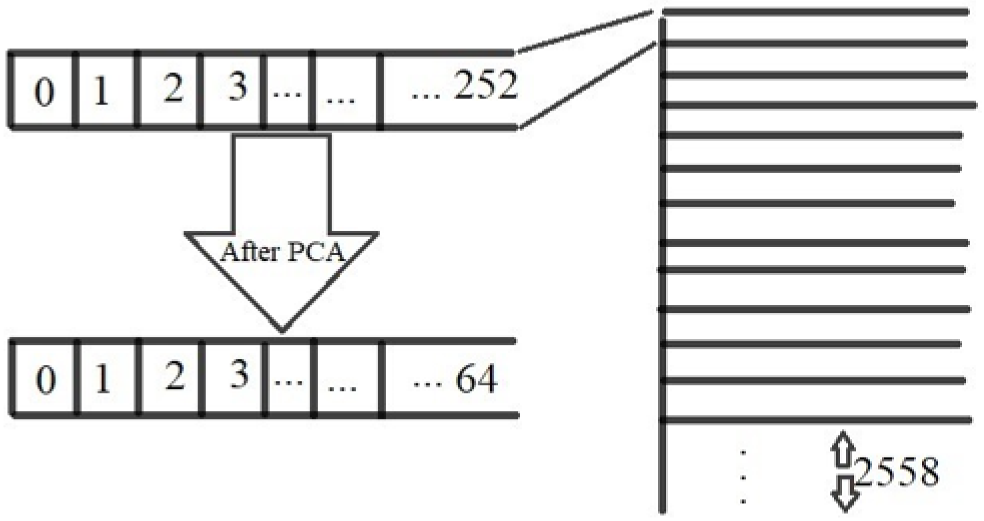
Kernel PCA

PRINCIPAL COMPONENT ANALYSIS: is a tool which is used to reduce the dimension of the data. It allows us to reduce the dimension of the data without much loss of information. PCA reduces the dimension by finding a few orthogonal linear combinations (principal components) of the original variables with the largest variance.PCA is a linear method. That is it can only be applied to datasets which are linearly separable. But, if we use it to non-linear datasets, we might get a result which may not be the optimal dimensionality reduction. Kernel PCA uses a kernel function to project dataset into a higher dimensional feature space, where it is linearly separable.



```
# feature reduction (K-PCA)
# ===============================================================
transformer = KernelPCA(n_components=64, kernel='linear')
X = transformer.fit_transform(X)
```

Where fit_transform(*X*) fits the model from data in X(training vector) and transform X.



FEATURE POOL AFTER KERNEL PCA

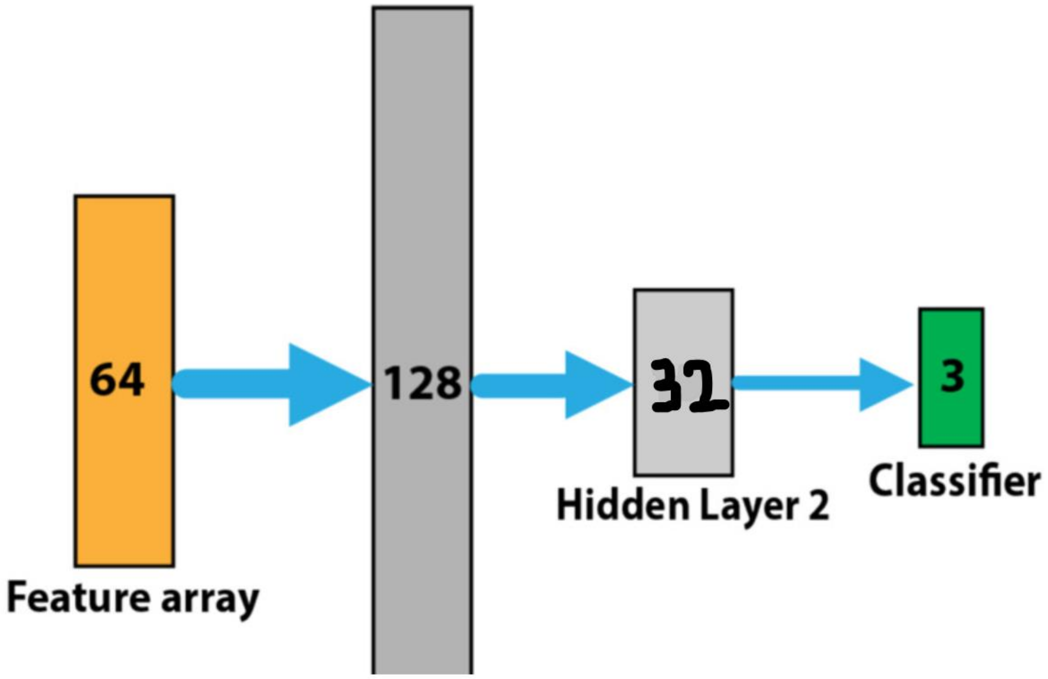
Building the Model

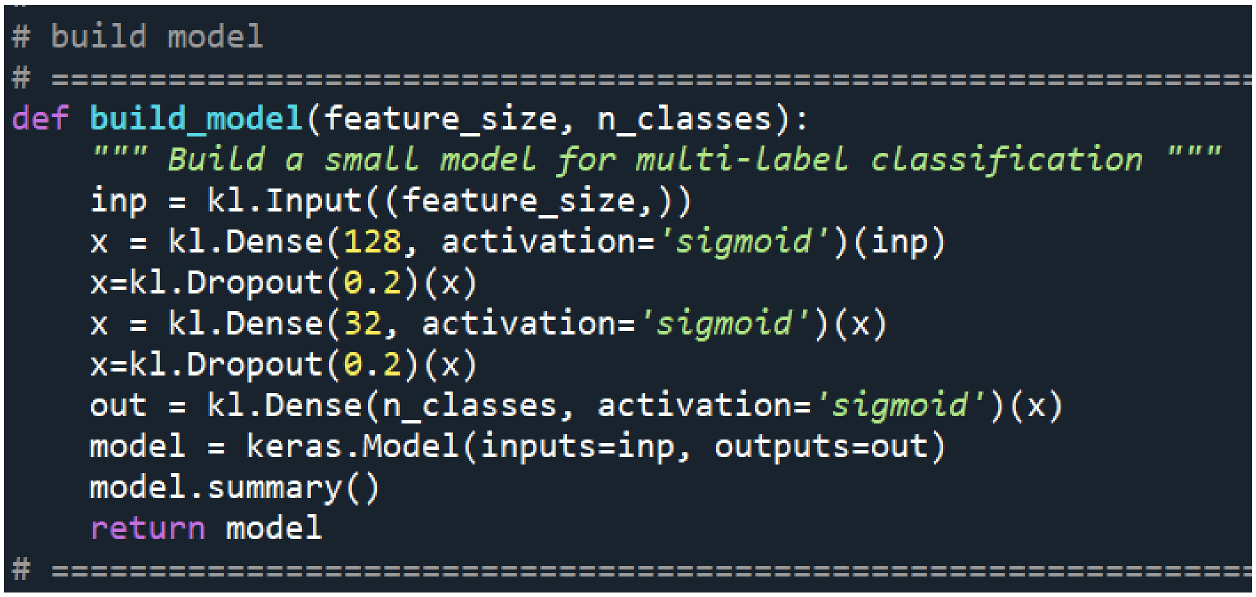
We have deployed a multi-layer neural network to classify the COVID-19 CXR images from non-Covid ones.

The Neural Network Layers:

Multi-label classification is a generalization of multiclass classification, which is the single-label problem of categorizing instances into precisely one of more than two classes; in the multi-label problem there is no constraint on how many of the classes the instance can be assigned to. Formally, multi-label classification is the problem of finding a model that maps inputs x to binary vectors y.

To design our COVID classifier, we have used a multi-layer neural network consisting of two hidden layers. The first hidden layer has 128 neurons and the second hidden layer has 32 neurons. The final layer i.e. the classifier has three neurons to categorize normal, COVID-19, non-COVID-19 pneumonia X-rays.



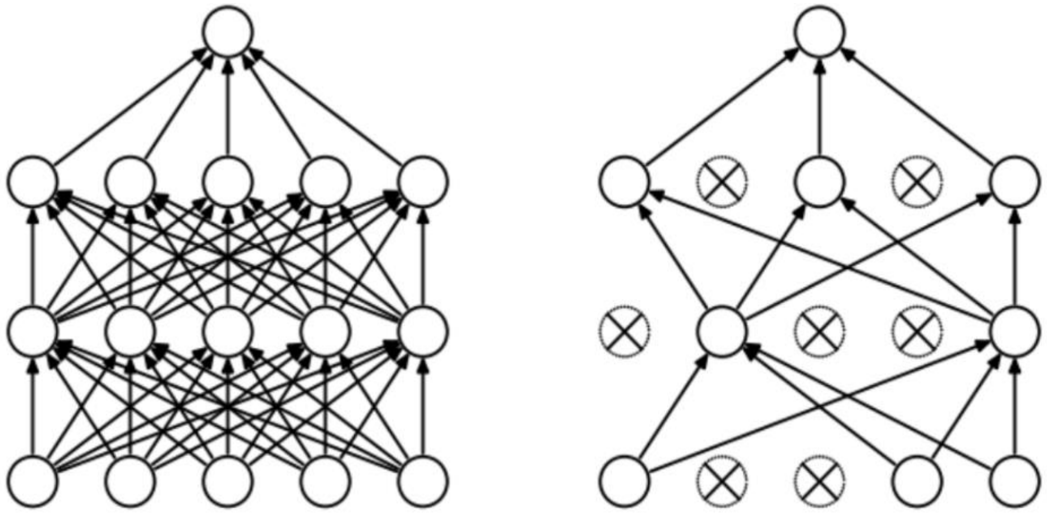


```
# build model
# ================================================================
def build_model(feature_size, n_classes):
    """ Build a small model for multi-label classification """
    inp = kl.Input((feature_size,))
    x = kl.Dense(128, activation='sigmoid')(inp)
    x=kl.Dropout(0.2)(x)
    x = kl.Dense(32, activation='sigmoid')(x)
    x=kl.Dropout(0.2)(x)
    out = kl.Dense(n_classes, activation='sigmoid')(x)
    model = keras.Model(inputs=inp, outputs=out)
    model.summary()
    return model
# ================================================================
```

- Sigmoid activation function,

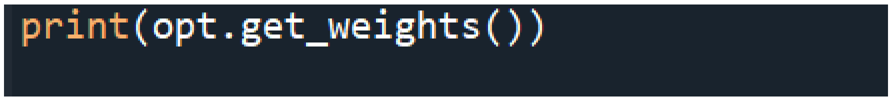
  sigmoid(x) = 1 / (1 + exp(-x))

  is applied.
- A dense layer is simply a layer where each unit or neuron is connected to each neuron in the next layer.
- Dropout is a regularization technique, which aims to reduce the complexity of the model with the goal to prevent overfitting. Using "dropout", certain neurons are randomly deactivated in a layer. The dropout rate i.e. the frequency with which the neurons are set to 0 at each step during training time, is set to 0.2.

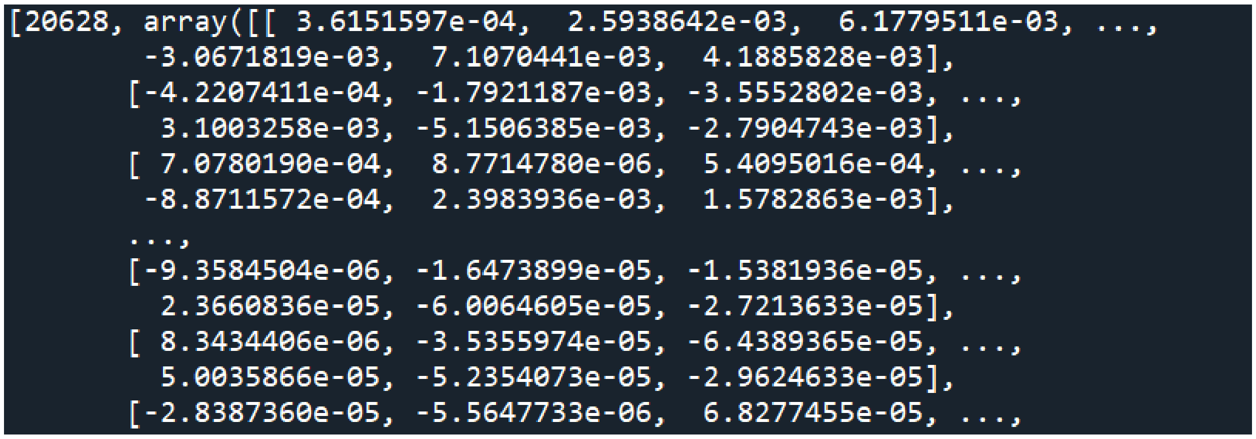


(a) Standard Neural Net (b) After applying dropout.

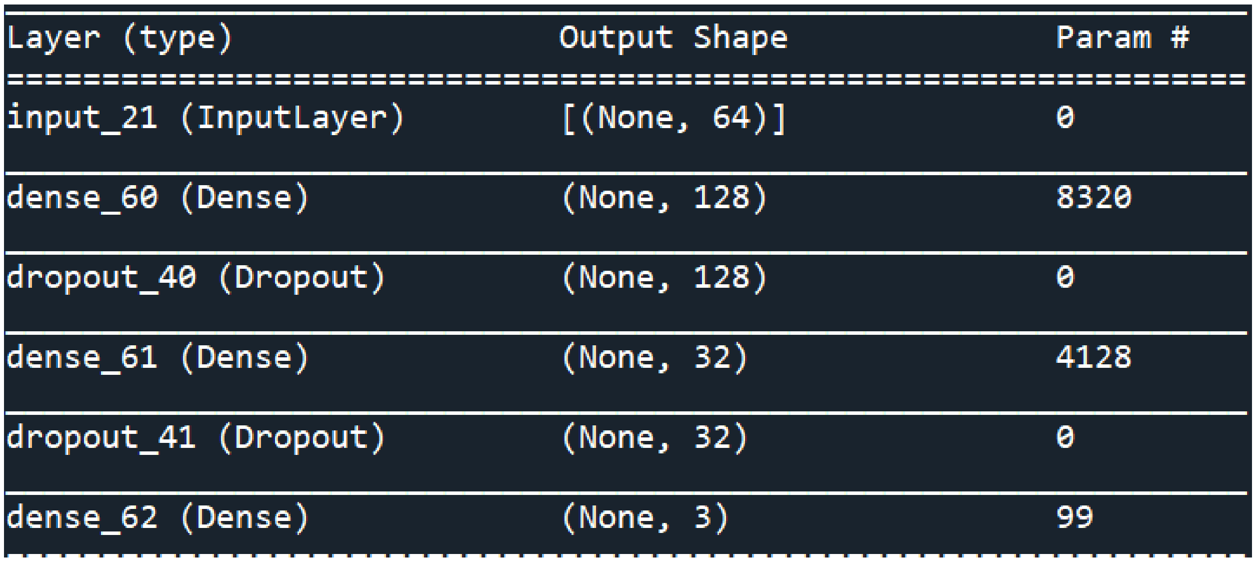
- Each layer has its own default value for initializing the weights. For most of the layers, such as Dense,, the default kernel initializer is 'glorot_uniform' and the default bias intializer is 'zeros' .
- The Glorot uniform initializer is also called Xavier uniform initializer. It draws samples from a uniform distribution within [-limit, limit], where limit = sqrt(6 / (fan_in + fan_out)) (fan_in is the number of input units in the weight tensor and fan_out is the number of output units).



```
print(opt.get_weights())
```

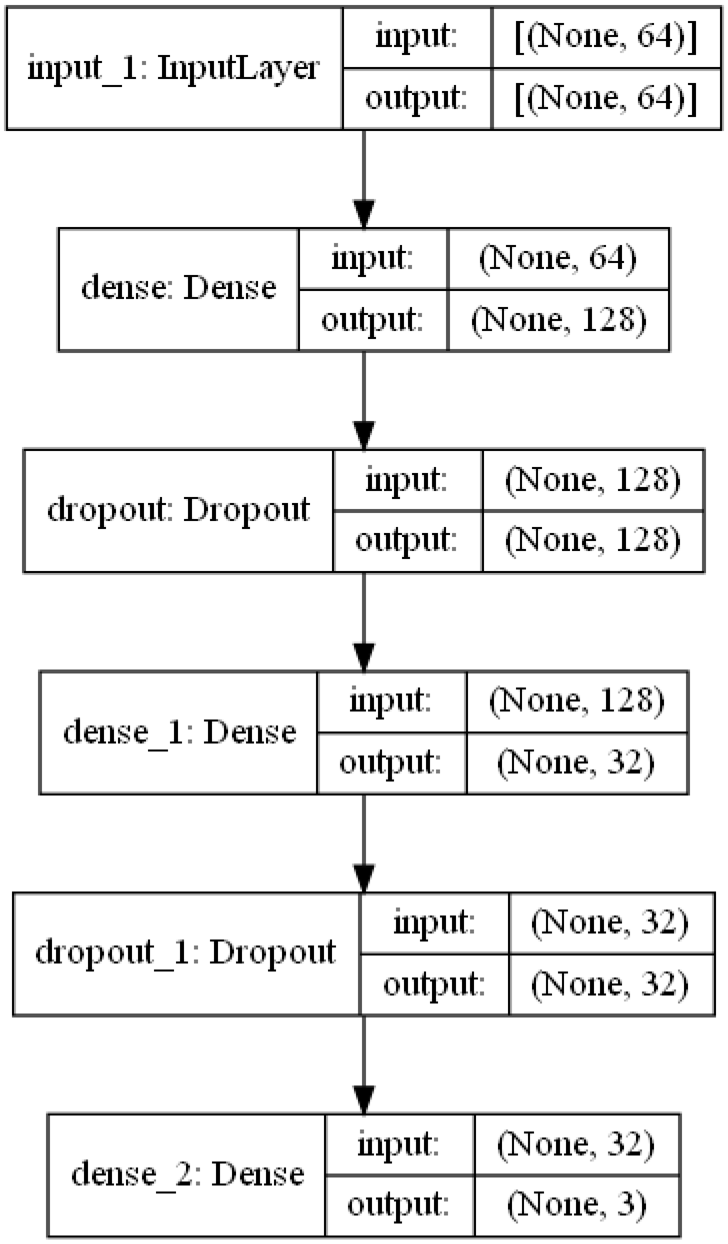


```
[20628, array([[ 3.6151597e-04,  2.5938642e-03,  6.1779511e-03, ...,
        -3.0671819e-03,  7.1070441e-03,  4.1885828e-03],
       [-4.2207411e-04, -1.7921187e-03, -3.5552802e-03, ...,
         3.1003258e-03, -5.1506385e-03, -2.7904743e-03],
       [ 7.0780190e-04,  8.7714780e-06,  5.4095016e-04, ...,
        -8.8711572e-04,  2.3983936e-03,  1.5782863e-03],
       ...,
       [-9.3584504e-06, -1.6473899e-05, -1.5381936e-05, ...,
         2.3660836e-05, -6.0064605e-05, -2.7213633e-05],
       [ 8.3434406e-06, -3.5355974e-05, -6.4389365e-05, ...,
         5.0035866e-05, -5.2354073e-05, -2.9624633e-05],
       [-2.8387360e-05, -5.5647733e-06,  6.8277455e-05, ...,
```



```
Layer (type)                 Output Shape              Param #
=================================================================
input_21 (InputLayer)        [(None, 64)]              0

dense_60 (Dense)             (None, 128)               8320

dropout_40 (Dropout)         (None, 128)               0

dense_61 (Dense)             (None, 32)                4128

dropout_41 (Dropout)         (None, 32)                0

dense_62 (Dense)             (None, 3)                 99
```

- None means this dimension is variable. The first dimension in a keras model is always the batch size. The following figure shows the classification model layers and their neurons:


input_1: InputLayer
input:
[(None, 64)]
output:
[(None, 64)]
dense: Dense
input:
(None, 64)
output:
(None, 128)
dropout: Dropout
input:
(None, 128)
output:
(None, 128)
dense_1: Dense
input:
(None, 128)
output:
(None, 32)
dropout_1: Dropout
input:
(None, 32)
output:
(None, 32)
dense_2: Dense
input:
(None, 32)
output:
(None, 3)

Choosing the best optimizer

Optimizers are basically algorithms or methods used to change the attributes of the neural network such as weights and learning rates, in order to reduce the losses.

Optimization functions usually calculate the gradient i.e. the partial derivative of loss function with respect to weights, and the weights are modified in the opposite direction of the calculated gradient. This cycle is repeated until we reach the minima of loss function.

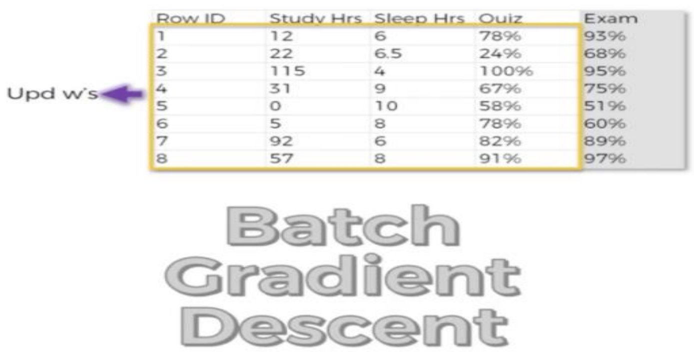
$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \frac{\partial}{\partial \mathbf{W}^{(k)}} J(\mathbf{W})$$

Thus, the components of a neural network model i.e. the activation function, loss function and optimization algorithm play a very important role in efficiently and effectively training a Model and producing accurate results. Different tasks require a different set of such functions to give the most optimum results.

A brief comparison is explained among different optimizers tested in regard to the COVID classifier.

1. Batch Gradient Descent (BGD)

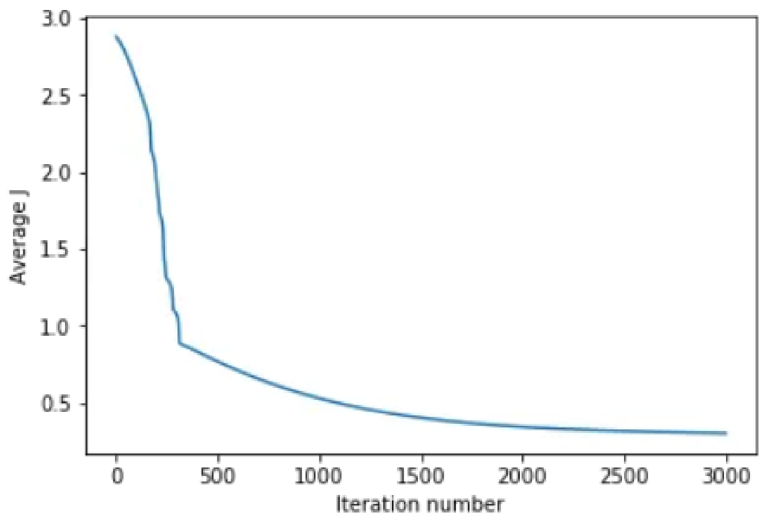
   Gradient update rule:
   BGD uses the data of the entire training set to calculate the gradient of the cost/lost function to the parameters.



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

Upd w's
Batch Gradient Descent

Cost Function:
Batch gradient descent is good because the training progress is nice and smooth – if we plot the average value of the cost function over the number of iterations / epochs it will look something like this:



Example batch gradient descent progress

As it can be seen, the line is mostly smooth and predictable. However, a problem with batch gradient descent in neural networks is that for every gradient descent update in the weights, we have to cycle through every training sample. For big data sets i.e. > 50,000 training samples as of COVID-19, this can be time prohibitive.

Disadvantage:
Because this approach calculates the gradient for the entire data set in one update, the calculation is very slow, it will be very problematic to encounter a large number of data sets as of COVID-19 classifier, and hence making it difficult to invest in new data to update the model in real-time.

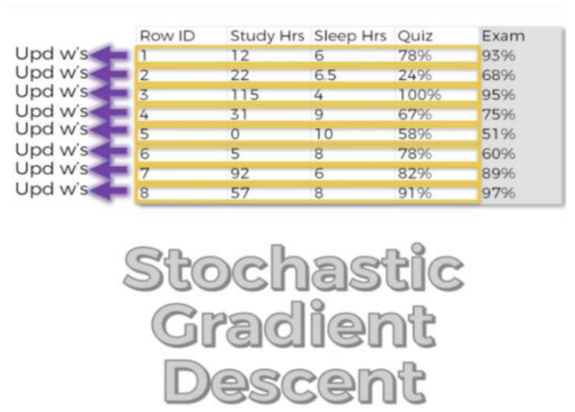
2. <u>Stochastic gradient descent (SGD)</u>

Gradient update rule:
The word '*stochastic*' means a system or a process that is linked with a random probability.

In this method one training sample (example) is passed through the neural network at a time and the parameters (weights) of each layer are updated with the computed gradient. So, at a time a single training sample is passed through the network and its corresponding loss is computed. The parameters of all the layers of the network are updated after every training sample. In other words, SGD updates the gradient of each sample with each update unlike BGD.

x +=-learning_rate * dx

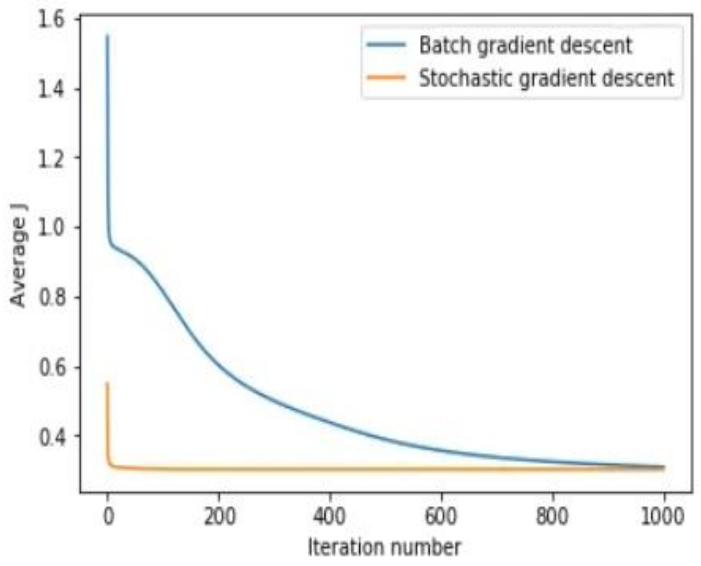
where x is a parameter, dx is the gradient and learning rate is constant



	Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
Upd w's	1	12	6	78%	93%
Upd w's	2	22	6.5	24%	68%
Upd w's	3	115	4	100%	95%
Upd w's	4	31	9	67%	75%
Upd w's	5	0	10	58%	51%
Upd w's	6	5	8	78%	60%
Upd w's	7	92	6	82%	89%
Upd w's	8	57	8	91%	97%

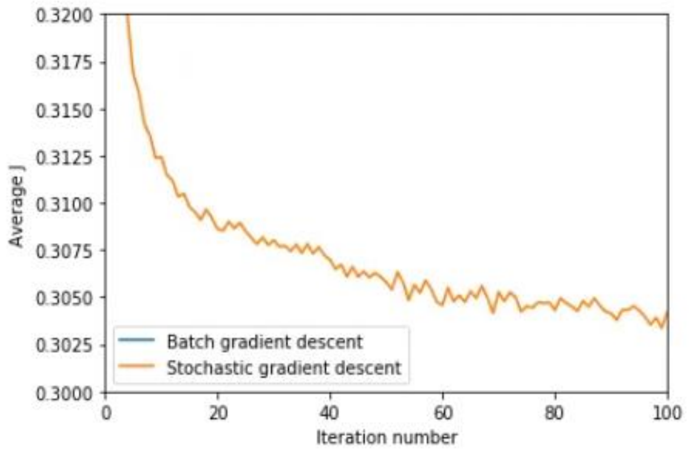
For large data sets, there may be similar samples, so BGD calculates the gradient. There will be redundancy, and SGD is updated only once, there is no redundancy, it is faster, and new samples can be added.

Cost Function:
The plot below shows the average cost versus the number of training epochs / iterations for batch gradient descent and SGD on the scikit-learn MNIST dataset. Note that both of these are operating off the same optimised learning parameters



Batch gradient descent versus SGD

Some interesting things can be noted from the above figure. First, SGD converges much more rapidly than batch gradient descent. In fact, SGD converges on a minimum *J* after < 20 iterations. Secondly, despite what the average cost function plot says, batch gradient descent after 1000 iterations *outperforms* SGD. On the MNIST test set, the SGD run has an accuracy of 94% compared to a BGD accuracy of 96%.



Noisy SGD

As it can be seen in the figure above, SGD is *noisy*. That is because it responds to the effects of each and every sample, and the samples themselves will no doubt contain an element of noisiness. While this can be a benefit in that it can act to "kick" the gradient descent out of local minimum values of the cost function, it can also hinder it settling down into a good minimum. This is why, eventually, batch gradient descent has outperformed SGD after 1000 iterations.

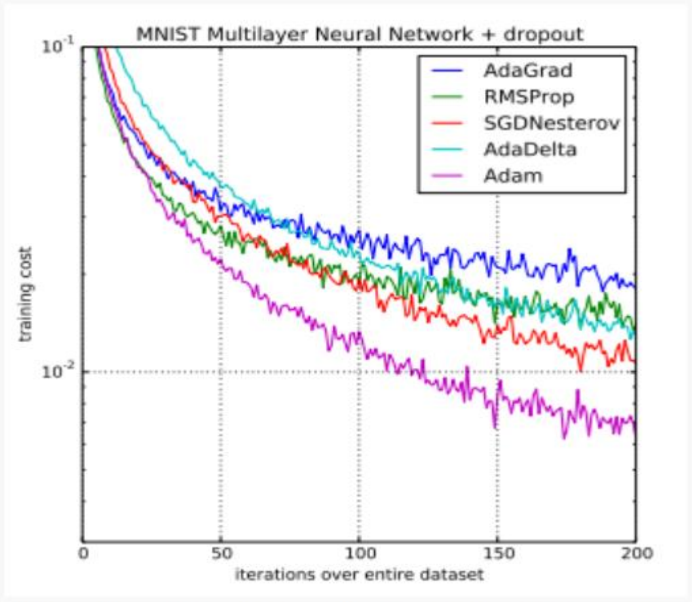
Disadvantages:
Due to frequent updates the steps taken towards the minima were very noisy. Also, due to noisy steps it took longer to achieve convergence to the minima of the loss function. Majorly, in this method the frequent updates are computationally expensive due to using all resources for processing one training sample at a time.

3. Adam Optimizer

Gradient Update Rule:
Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. Adam is an adaptive learning rate method, which means it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

Cost Function:
Adam was applied to the logistic regression algorithm on the MNIST digit recognition and IMDB sentiment analysis datasets, a Multilayer Perceptron algorithm on the MNIST dataset and Convolutional Neural Networks on the CIFAR-10 image recognition dataset. They conclude:

Using large models and datasets, we demonstrate Adam can efficiently solve practical deep learning problems.



Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron
Taken from Adam: A Method for Stochastic Optimization, 2015.

Disadvantages:
Adam can suffer a weight decay problem (after each update, the weights are multiplied by a factor slightly less than 1. This prevents the weights from growing too large).

<u>Why Adam Optimizer?</u>

Although SGD and other gradient descent variants are a good bet but we have to face certain challenges. Let say we have a sparse data set(in actual the COVID-19 data set used is not sparse) where some features are frequently occurring and others are rare, then opting for a same learning rate for all the parameters will not be a good idea. We would want to make a larger update for the rarely occurring ones as compared to the frequently occurring features. Another challenge is to choose a proper learning rate. If we choose a very large learning rate, we try to dwindle around the minimum and if we choose a very small learning rate the convergence gets really slow. Adaptive Moment Estimation (Adam) is the best optimizer to address these challenges which inherits itself from Adagrad and RMSProp.

Adagrad:

Adagrad optimizer helps us in solving one of the challenges we saw above connected to the sparse data set, by adapting the learning rate to the parameters, by having a low learning rate for the parameters associated to frequently occurring features and larger updates to the ones with infrequent features. So how it differs from the trivial SGD is that, in SGD we have a common learning rate for all the parameters and thus we are able to update all the parameters θ at once. But in Adagrad we try to have a different learning rate for each parameter θ at every timestamp. So the procedure is to have a per-parameter update and vectorize it.

$g_i = \nabla J(\theta_i)$, where g is the gradient wrt to each parameter θ_i.

How Adagrad is different is that it modifies the learning rate α for every parameter θ_i based on all the past gradients that have been calculated so far for each θ_i.

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{G_{ii} + \epsilon}} \cdot g_i$$

Here G_{ii} is an i*i dimensional diagonal matrix where the diagonal terms (i,i) represent the sum of the squares of the gradients w.r.t θ_i until that timestamp and ϵ is the smoothing out constant to avoid the division by 0 and has a value of 10^(-8).

Thus we have a method to automatically tune in the learning rate. But we have a disadvantage here that as the G is positive cause of the squared sum being added in all the diagonals, the accumulated sum keeps on increasing during the training and thus the learning rate becomes infinitesimally small. Thus we resort to another RMSprop method in order to solve the Adagrad's diminishing learning rate.

RMSprop with Adagrad:

In order to decrease the monotonically learning rate in the above Adagrad algorithm, we try to take a decaying average of all the past squared gradients in a recursive fashion. Thus the running average of the squared gradients depends only on the previous average and the current gradient, the running gradient being denoted by E[g^2]т at time t.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

As we see that the parameter update term in the Adagrad is of the form:

$$\theta = \theta - \frac{\eta}{\sqrt{G+\epsilon}} \odot g$$

We replace the G term here with the above E[g^2]т which is nothing but root mean squared error-criterion of the gradient.

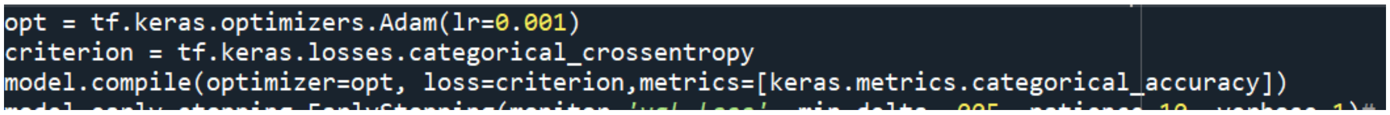
$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t{-}1) + 0.1\left(\partial E/_{\partial w}(t)\right)^2$$

For the learning to work much better, we set the above values for the parameters and divide the gradient by square root of the above term. We get the value of γ to be 0.9 thus the parameter update term becomes

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} g_t$$

Code:

The Adam Optimizer is implemented in training.py module through the TensorFlow API:



```
opt = tf.keras.optimizers.Adam(lr=0.001)
criterion = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=opt, loss=criterion,metrics=[keras.metrics.categorical_accuracy])
```

As seen in the code, Adam optimizer is first instantiated before passing it to model.compile and the learning rate provided is 0.001

Deciding on a loss function:

Error and Loss Function: In most learning networks, error is calculated as the difference between the actual output and the predicted output.

$$J(w) = p - \widehat{p}$$

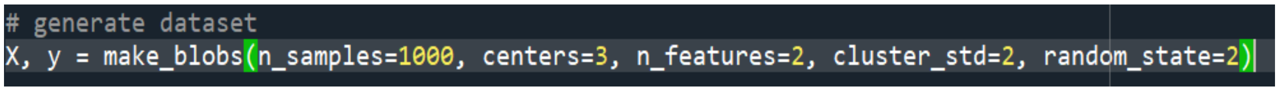
The function that is used to compute this error is known as Loss Function J(.). Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model.

Multi-Class Classification Loss Functions

Multi-Class classification are those predictive modeling problems where examples are assigned one of more than two classes.

The problem is often framed as predicting an integer value, where each class is assigned a unique integer value from 0 to (num_classes – 1). The problem is often implemented as predicting the probability of the example belonging to each known class.

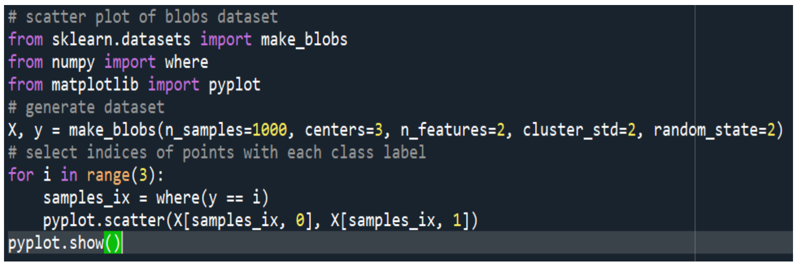
We will use the blobs problem as the basis for the investigation. The make_blobs() function provided by the scikit-learn provides a way to generate examples given a specified number of classes and input features. We will use this function to generate 1,000 examples for a 3-class classification problem with 2 input variables. The pseudorandom number generator will be seeded consistently so that the same 1,000 examples are generated each time the code is run.



```
# generate dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

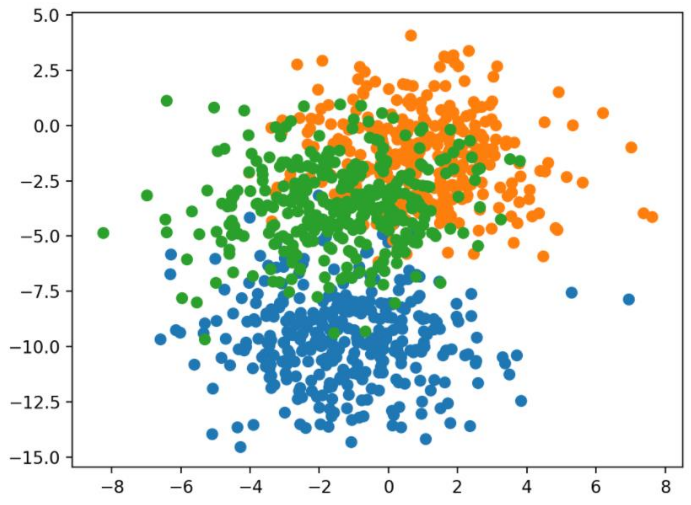
The two input variables can be taken as x and y coordinates for points on a two-dimensional plane.

The example below creates a scatter plot of the entire dataset coloring points by their class membership.



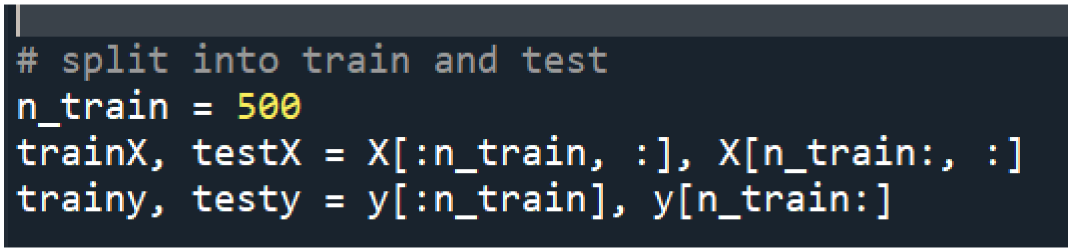
```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from numpy import where
from matplotlib import pyplot
# generate dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# select indices of points with each class label
for i in range(3):
    samples_ix = where(y == i)
    pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1])
pyplot.show()
```

Running the example creates a scatter plot showing the 1,000 examples in the dataset with examples belonging to the 0, 1, and 2 classes colors blue, orange, and green respectively.



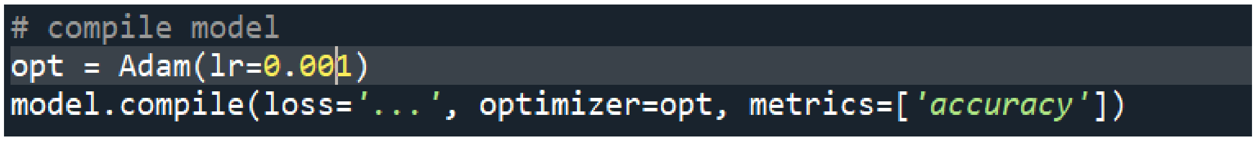
Scatter Plot of Examples Generated from the Blobs Multi-Class Classification Problem

The dataset will be split evenly between train and test sets.



```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

A small MLP model will be used as the basis for exploring loss functions. The model expects two input variables, has 50 nodes in the hidden layer and the rectified linear activation function, and an output layer that must be customized based on the selection of the loss function. The model is fit using adam optimizer.



```
# compile model
opt = Adam(lr=0.001)
model.compile(loss='...', optimizer=opt, metrics=['accuracy'])
```

Now that we have the basis of a problem and model, we can take a look evaluating common loss function that is appropriate for a multi-class classification modeling problem.

Multi-Class Classification Loss Functions are listed below and would be discussed in regard to this COVID-19 classifier

1. Kullback Leibler Divergence Loss
2. Multi-Class Cross-Entropy Loss

Multi-Class Cross-Entropy Loss

Cross-entropy is the default loss function to use for multi-class classification problems. It is intended for use with multi-class classification where the target values are in the set {0, 1, 3, …, n}, where each class is assigned a unique integer value.

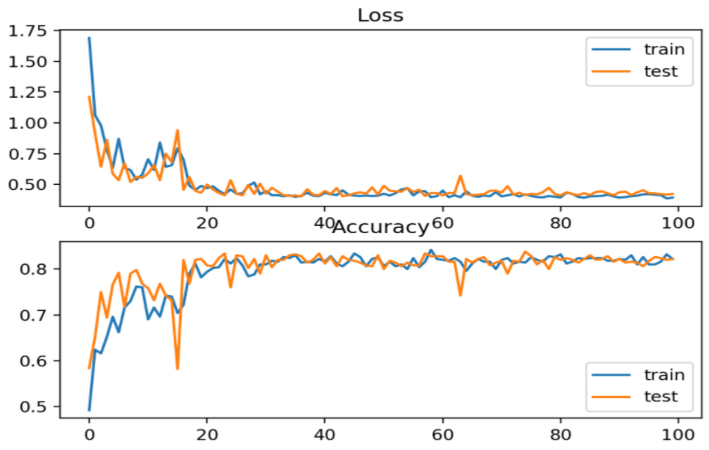
It is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if we have a good reason. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0. The categorical crossentropy loss function calculates the loss of an example by computing the following sum:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

where *ŷi* is the *i*-th scalar value in the model output, *yi* is the corresponding target value, and output size is the number of scalar values in the model output. This loss is a very good measure of how distinguishable two *discrete probability distributions* are from each other. In this context, *yi* is the probability that event *i* occurs and the sum of all *yi* is 1, meaning that exactly one event may occur(i.e. an example may belong to only one class). The minus sign ensures that the loss gets smaller when the distributions get closer to each other. If an example exactly belongs to a class it will have a target probability of 1, the other classes 0.

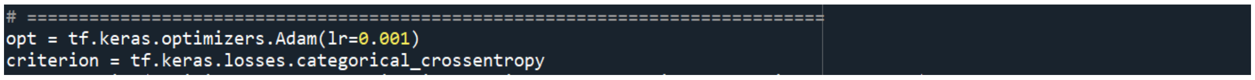
In the considered example, the model performed well, achieving a classification accuracy of about 84% on the training dataset and about 82% on the test dataset. A figure is also created showing two line plots, the top with the cross-entropy loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. The plot shows the model seems to have converged. The line plots for both cross-entropy and accuracy show good convergence behavior, although somewhat bumpy.

The model may be well configured given no sign of over or under fitting. The learning rate or batch size may be tuned to even out the smoothness of the convergence in this case.



Line Plots of Cross Entropy Loss and Classification Accuracy over Training Epochs on the Blobs Multi-Class Classification Problem

Code:



```
# ============================================================================
opt = tf.keras.optimizers.Adam(lr=0.001)
criterion = tf.keras.losses.categorical_crossentropy
```

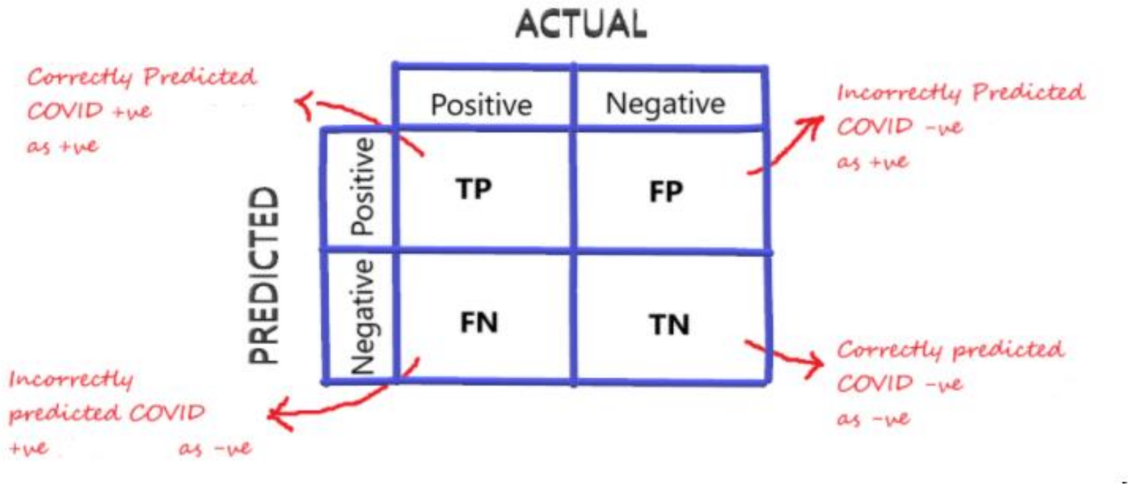
Tf.keras.losses.categorical_crossentropy computes the categorical crossentropy loss.

Deciding on the evaluation metrics:

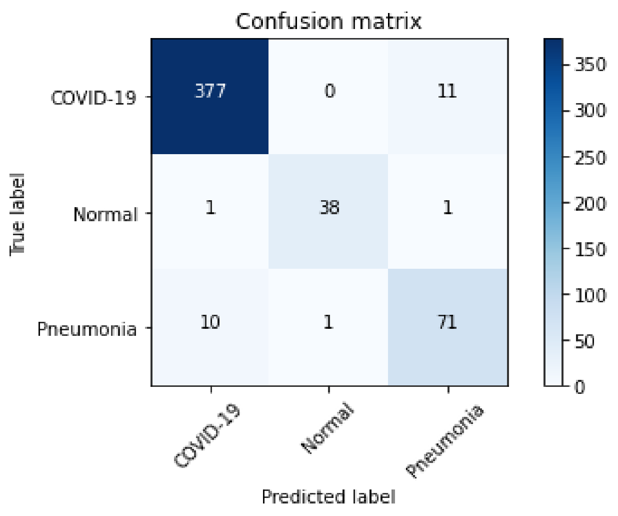
After doing the usual feature engineering, selection, implementing a model and getting some output in the form of a probability or a class, the next step is to find out how effective the model is based on some metric using test datasets. The metric explains the performance of a model. The model may give satisfying results when evaluated using a metric say accuracy score but may give poor results when evaluated against other metrics such as logarithmic loss or any other such metric. Hence, it is very much important to choose the right metric to evaluate the Machine Learning model.

Choice of metrics influences how the performance of machine learning algorithm is measured and compared. They influence how we weigh the importance of different characteristics in the results.

Confusion Matrix


ACTUAL
Positive
Negative
PREDICTED
Positive
Negative
TP
FP
FN
TN
Correctly Predicted COVID +ve as +ve
Incorrectly Predicted COVID -ve as +ve
Incorrectly predicted COVID +ve as -ve
Correctly predicted COVID -ve as -ve

The confusion matrix of our covid classifier is as follows:



Accuracy

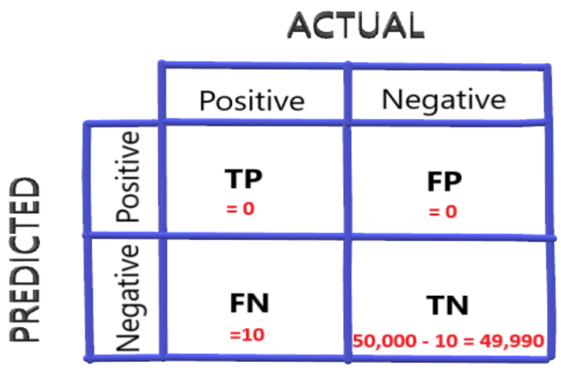
We want our model to focus on True positive and True Negative. Accuracy is one metric which gives the fraction of predictions our model got right. Formally, accuracy has the following definition:

- *Accuracy = Number of correct predictions / Total number of predictions.*

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TP}$$

Our COVID classifier has achieved categorical_accuracy of 0.9451.

Now, let’s consider 50,000 people get their Xrays done per day on an average. Out of which, 10 are actually COVID positive. One of the easy ways to increase accuracy is to classify every person as COVID negative. So our confusion matrix looks like:



PREDICTED \ ACTUAL	Positive	Negative
Positive	TP = 0	FP = 0
Negative	FN =10	TN 50,000 - 10 = 49,990

Accuracy for this case will be:

- Accuracy = 49,990/50,000 = 0.9998 or 99.98%

This concludes that accuracy alone cannot be used as an evaluation metric.

Recall (Sensitivity or True positive rate)

Recall gives the fraction correctly identified as positive out of all positives.

$$\text{Recall} = \frac{\overset{\text{Correctly predicted as COVID +ve}}{TP}}{\underbrace{TP + FN}_{\text{Total COVID +ve}}}$$

Now, this is an important measure. Out of all positive people what fraction is identified correctly. Adopting the strategy of labeling every person as negative that will give recall of Zero.

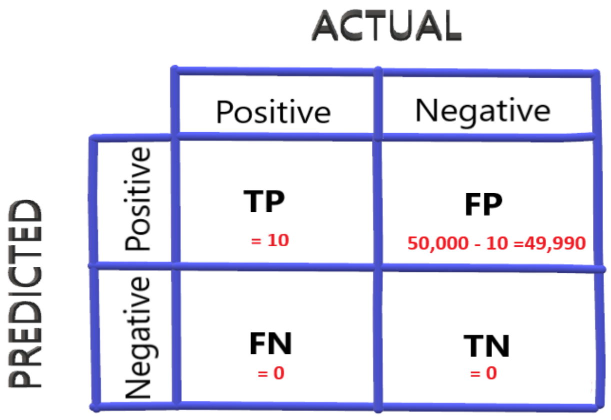
Recall = 0/10 = 0

So, in this context, Recall is a good measure. It says that the terrible strategy of identifying every person as COVID negative leads to zero recall. And we want to maximize the recall.

Is recall good enough to evaluate the performance of the classification model?

To answer the above question, consider another scenario of labeling every person as COVID positive. Labeling every person as positive is bad in terms of the amount of cost that needs to be spent in actually doing the Xray and investigating it.

The confusion matrix will look like:



Recall for this case would be:

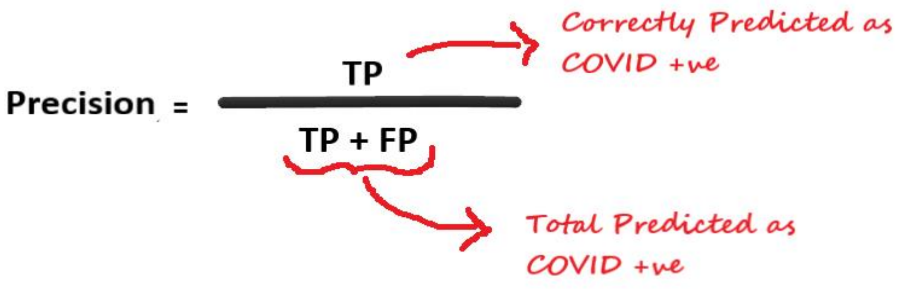
Recall = 10/(10+0) = 1

That's a huge problem. So concluding, Recall will be a good measure in this context but then realized that labeling everyone as positive will increase recall as well.

So recall independently is not a good measure. There is another measure called Precision

Precision

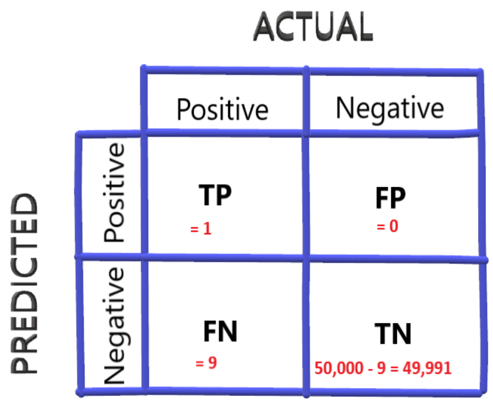
Precision gives the fraction of correctly identified as positive out of all predicted as positives.


Precision = TP / TP + FP
Correctly Predicted as COVID +ve
Total Predicted as COVID +ve

Considering our the strategy of labeling every person as positive, the precision would be :
Precision = 10 / (10 + 49990) = 0.0002 While this bad strategy has a good recall value of 1

but it has a terrible precision value of 0.0002. This clarifies that recall alone is not a good measure, we need to consider precision value.

Considering another case of labeling people with the highest probability of having COVID. Let’s say we got only one such Xray. The confusion matrix in this case will be:



Precision comes out to be: 1/ (1 + 0) = 1

Precision value is good in this case but let’s check for recall value once:

Recall = 1 / (1 + 9) = 0.1

Precision value is good in this context but recall value is low. So a high precision value would correspond to a low recall value if the Xrays are correctly identified.

F1 Score

It is defined as the harmonic mean of the model’s precision and recall.

$$\text{F1 Score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

We use Harmonic mean because it is not sensitive to extremely large values, unlike simple averages. Say, we have a model with a precision of 1, and recall of 0 gives a simple average as 0.5 and an F1 score of 0. If one of the parameters is low, the second one no longer matters in the F1 score. The F1 score favors classifiers that have similar precision and recall. Thus, the F1 score is a better measure to use if we are seeking a balance between Precision and Recall.

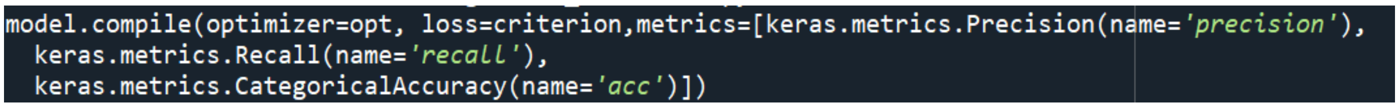
False Positive Rate (FPR)

Out of all the negative classes, how much we predicted wrong and is defined as follows:

$$FPR = \frac{FP}{FP + TN}$$

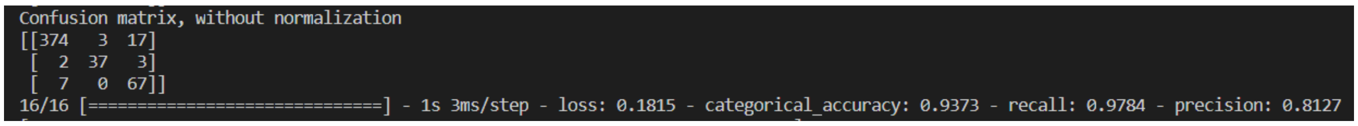
Code:

In some cases, we are pretty sure that we want to maximize either recall or precision at the cost of others. As in this case of labeling Xrays, we really want to get the predictions right for COVID positive people because it is really expensive to not predict the people right as it would result in increasing the spread. So, we are more interested in precision here.



```
model.compile(optimizer=opt, loss=criterion,metrics=[keras.metrics.Precision(name='precision'),
  keras.metrics.Recall(name='recall'),
  keras.metrics.CategoricalAccuracy(name='acc')])
```

The evaluation metrics for our model are:



```
Confusion matrix, without normalization
[[374   3  17]
 [  2  37   3]
 [  7   0  67]]
16/16 [==============================] - 1s 3ms/step - loss: 0.1815 - categorical_accuracy: 0.9373 - recall: 0.9784 - precision: 0.8127
```

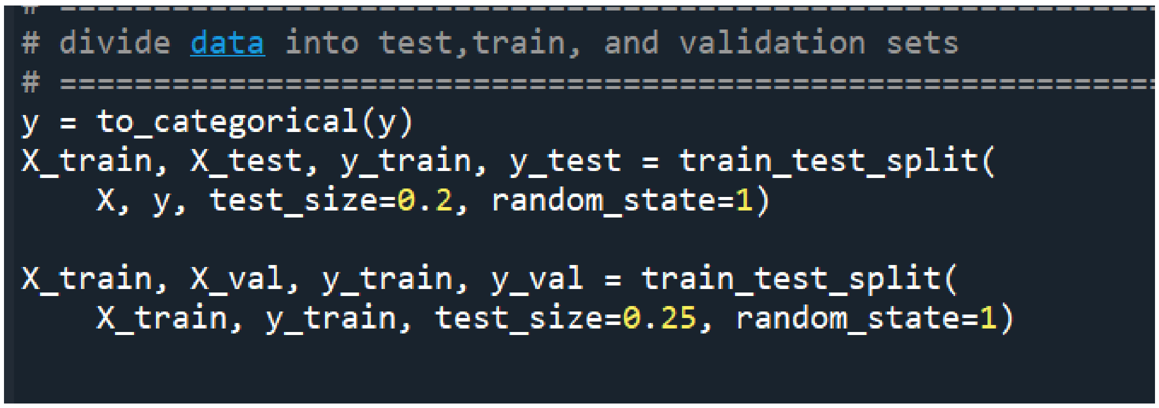
Training the Model

Split data into training and testing sets

The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model. The procedure involves taking a dataset and dividing it into two subsets.

- **Train Dataset**: Used to fit the machine learning model.
- **Test Dataset**: Used to evaluate the fit machine learning model.

The dataset is divided in into 80% for training and 20% for testing during model training and 75% for training and 25% for testing during validation.


```
# ==========================================================
# divide data into test,train, and validation sets
# ==========================================================
y = to_categorical(y)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=1)

X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.25, random_state=1)
```

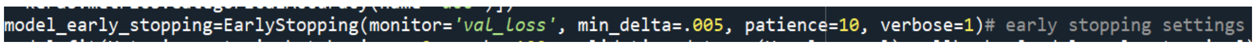
- We have used **to_categorical** function to convert a class vector (integers) to binary class matrix. for use with categorical_crossentropy.
- **sklearn.model_selection.train_test_split**(*arrays, test_size=None, train_size=None , random_state=None) split arrays or matrices into random train and test subsets. Where,
  - ***arrays** are a sequence of indexables with same length
  - **test_size**(float or int, default=None)
    If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If train_size is also None, it will be set to 0.25.
  - **train_size**(float or int, default=None)
    If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.
  - **random_state** controls the shuffling applied to the data before applying the split.

Early Stopping Parameters:

A problem with training neural networks is in the choice of the number of training epochs to use. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. **Early stopping** is a method that allows to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

Keras supports the early stopping of training via a callback called EarlyStopping. This callback allows to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process.

- The **“monitor”** allows to specify the performance measure to monitor in order to end training. The calculation of measures on the validation dataset will have the ‘val_‘ prefix, such as ‘val_loss‘ for the loss on the validation dataset. Training will stop when the chosen performance measure stops improving.
- To discover the training epoch on which training was stopped, the “**verbose”** argument can be set to 1. Once stopped, the callback will print the epoch number.
- Often, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the **“patience”** argument.
- By default, any change in the performance measure, no matter how fractional, will be considered an improvement. We may want to consider an improvement that is a specific increment, such as 1 unit for mean squared error or 1% for accuracy. This can be specified via the “**min_delta**” argument.

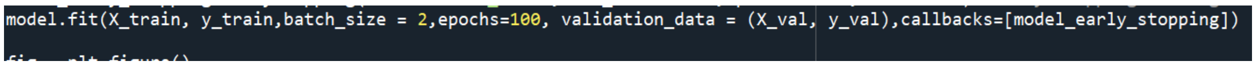


```
model_early_stopping=EarlyStopping(monitor='val_loss', min_delta=.005, patience=10, verbose=1)# early stopping settings
```

Fitting the model

Model fitting is a measure of how well a machine learning model generalizes to similar data to that on which it was trained. A model that is well-fitted produces more accurate outcomes. We call fit(), which will train the model by slicing the data into "batches" of size "batch_size", and repeatedly iterating over the entire dataset for a given number of "epochs".

- The **batch size** i.e. number of samples processed before the model is updated, is 2. And the number of **epochs** i.e. the number of complete passes through the training dataset is set to 100. They are defined in the fit method.
- **Callbacks** provide a way to execute code and interact with the training model process automatically. Callbacks can be provided to the fit() function via the “callbacks” argument.
- Early stopping requires that a validation dataset is evaluated during training. This can be achieved by specifying the **validation dataset** to the fit() function when training your model. Primarily through splitting the training data into a train and validation dataset and specifying the validation dataset to the fit() function via the validation_data argument.



```
model.fit(X_train, y_train,batch_size = 2,epochs=100, validation_data = (X_val, y_val),callbacks=[model_early_stopping])
```

Model Evaluation

Model evaluation aims to estimate the generalization accuracy of a model on future (unseen/out-of-sample) data. Methods for evaluating a model's performance are divided into 2 categories: namely, Holdout and Cross-validation.

Holdout

The purpose of holdout evaluation is to test a model on different data than it was trained on. This provides an unbiased estimate of learning performance.
In this method, the dataset is randomly divided into three subsets:

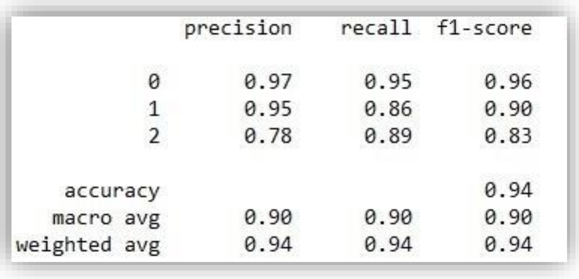
- Training set
- Validation set
- Test set

All these types of datasets in regard to this COVID-19 Classifier have been discussed under the sub-heading Split data into training and testing sets

Cross-Validation

Cross-validation is a technique that involves partitioning the original observation dataset into a training set, used to train the model, and an independent set used to evaluate the analysis.
In our model, we have used Repeated **Random Sub-Sampling Validation.** This method, also known as Monte Carlo cross-validation, creates multiple random splits of the dataset into training and validation data. For each such split, the model is fit to the training data, and predictive accuracy is assessed using the validation data. The results are then averaged over the splits. This is achieved through random_state in the sklearn.model_selection.train_test_split() function.

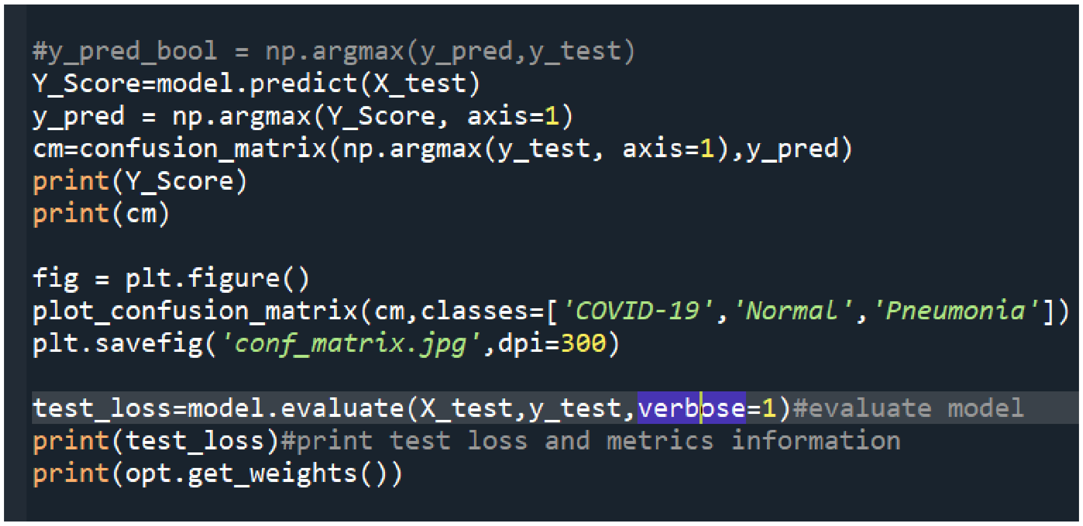
Model Evaluation Metrics



```
              precision    recall  f1-score

           0       0.97      0.95      0.96
           1       0.95      0.86      0.90
           2       0.78      0.89      0.83

    accuracy                           0.94
   macro avg       0.90      0.90      0.90
weighted avg       0.94      0.94      0.94
```

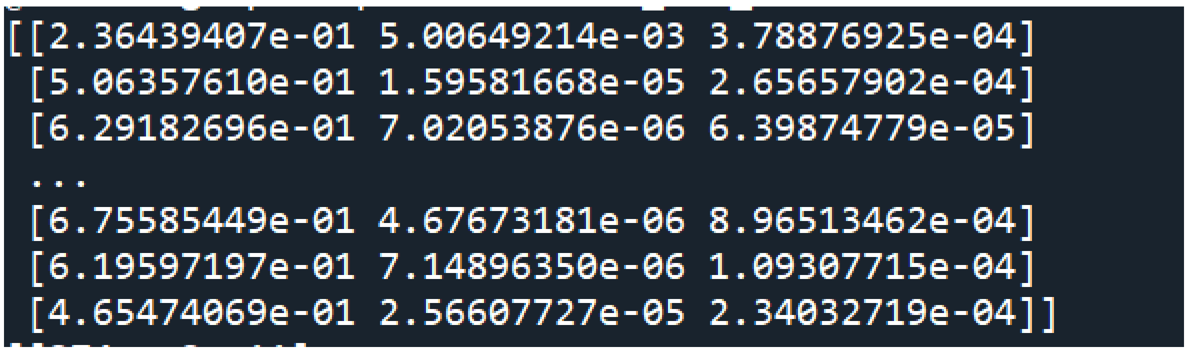
CLASSIFICATION REPORT

This section has been discussed in detailed under the heading Evaluate Metrics. However, this section will focus on the code.



```
#y_pred_bool = np.argmax(y_pred,y_test)
Y_Score=model.predict(X_test)
y_pred = np.argmax(Y_Score, axis=1)
cm=confusion_matrix(np.argmax(y_test, axis=1),y_pred)
print(Y_Score)
print(cm)

fig = plt.figure()
plot_confusion_matrix(cm,classes=['COVID-19','Normal','Pneumonia'])
plt.savefig('conf_matrix.jpg',dpi=300)

test_loss=model.evaluate(X_test,y_test,verbose=1)#evaluate model
print(test_loss)#print test loss and metrics information
print(opt.get_weights())
```

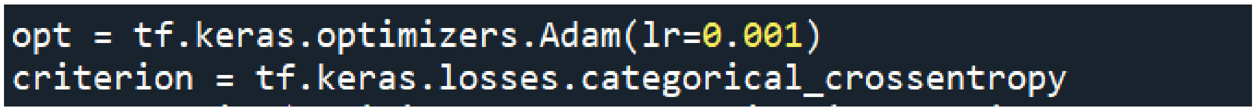
- **model.predict()** : given a trained model, predict the label of a new set of data. This method accepts one argument, the new data X_new (e.g. model.predict(X_new)), and returns the learned label for each object in the array.
- **np.argmax(Y_score, axis=1)**
  The Y_Score matrix is:



```
[[2.36439407e-01 5.00649214e-03 3.78876925e-04]
 [5.06357610e-01 1.59581668e-05 2.65657902e-04]
 [6.29182696e-01 7.02053876e-06 6.39874779e-05]
 ...
 [6.75585449e-01 4.67673181e-06 8.96513462e-04]
 [6.19597197e-01 7.14896350e-06 1.09307715e-04]
 [4.65474069e-01 2.56607727e-05 2.34032719e-04]]
```

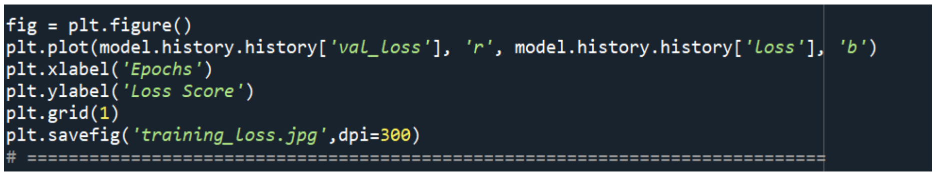
By adding the axis argument, numpy looks at the rows and columns individually. axis=1 means that the operation is performed across the rows of Y_Score.

- The cross-entropy loss function chosen to be optimized for our model is calculated at the end of each epoch.

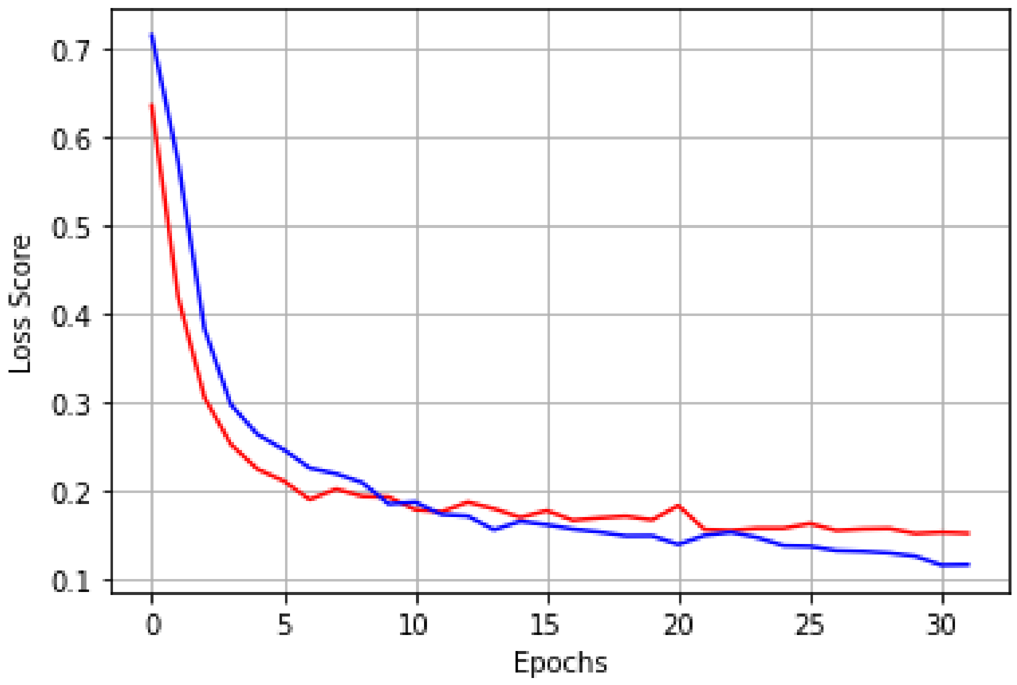


```
opt = tf.keras.optimizers.Adam(lr=0.001)
criterion = tf.keras.losses.categorical_crossentropy
```

Tf.keras.losses.categorical_crossentropy computes the categorical crossentropy loss. We can create plots from the collected history data. The returned "history"(**history.history**) from the fit() method holds a record of the loss values and metric values during training. From the plot of loss, we can see that the model has comparable performance on both train and validation datasets (labeled test).



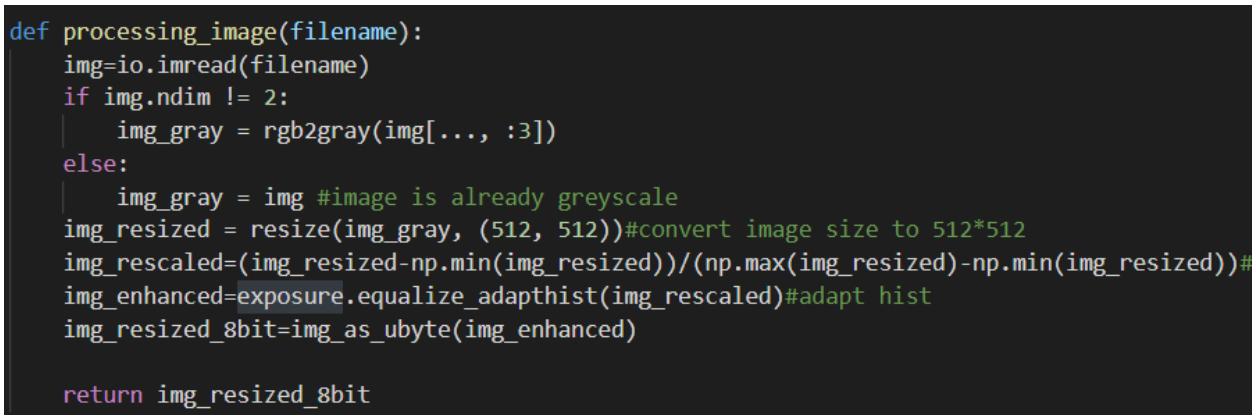
```
fig = plt.figure()
plt.plot(model.history.history['val_loss'], 'r', model.history.history['loss'], 'b')
plt.xlabel('Epochs')
plt.ylabel('Loss Score')
plt.grid(1)
plt.savefig('training_loss.jpg',dpi=300)
# ===========================================================================
```



During training our model, both training and validation sets reached ~ 0.142 loss score and 95% accuracy after ~30 epochs. The loss graph showed a good fit between validation and training curves, confirming that our model is not suffering from overfitting or underfitting.

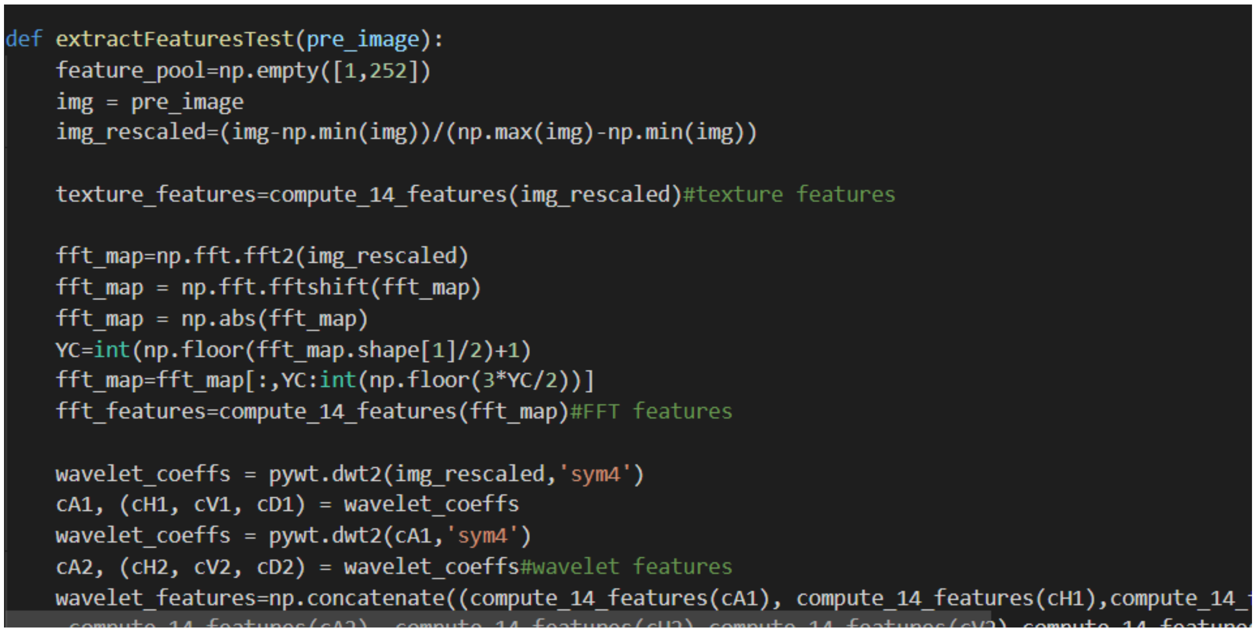
Model Prediction

So far, we have preprocessed, extract features from the image, get done with evaluation metrics of different types and also test our model on validation set and test set. Now we will move towards our last objective i.e. prediction and classification of image when user uploads a image, the model will tell you is it Covid? Or Pneumonia? Or are you okay?



```python
def processing_image(filename):
    img=io.imread(filename)
    if img.ndim != 2:
        img_gray = rgb2gray(img[..., :3])
    else:
        img_gray = img #image is already greyscale
    img_resized = resize(img_gray, (512, 512))#convert image size to 512*512
    img_rescaled=(img_resized-np.min(img_resized))/(np.max(img_resized)-np.min(img_resized))#
    img_enhanced=exposure.equalize_adapthist(img_rescaled)#adapt hist
    img_resized_8bit=img_as_ubyte(img_enhanced)

    return img_resized_8bit
```

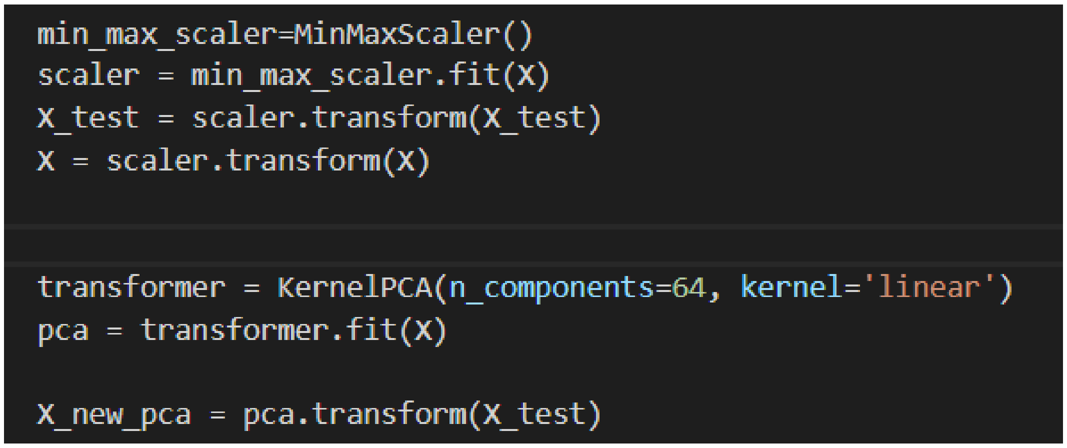
For this we have to read an image from PC which user uploads from the button. We have given this GUI interface for non-technical users to be more interactive with the process. The image is then being pre-processed of which the whole phenomenon has been described earlier in the report.



```python
def extractFeaturesTest(pre_image):
    feature_pool=np.empty([1,252])
    img = pre_image
    img_rescaled=(img-np.min(img))/(np.max(img)-np.min(img))

    texture_features=compute_14_features(img_rescaled)#texture features

    fft_map=np.fft.fft2(img_rescaled)
    fft_map = np.fft.fftshift(fft_map)
    fft_map = np.abs(fft_map)
    YC=int(np.floor(fft_map.shape[1]/2)+1)
    fft_map=fft_map[:,YC:int(np.floor(3*YC/2))]
    fft_features=compute_14_features(fft_map)#FFT features

    wavelet_coeffs = pywt.dwt2(img_rescaled,'sym4')
    cA1, (cH1, cV1, cD1) = wavelet_coeffs
    wavelet_coeffs = pywt.dwt2(cA1,'sym4')
    cA2, (cH2, cV2, cD2) = wavelet_coeffs#wavelet features
    wavelet_features=np.concatenate((compute_14_features(cA1), compute_14_features(cH1),compute_14_
     compute_14_features(cA2), compute_14_features(cH2),compute_14_features(cV2),compute_14_feature
```

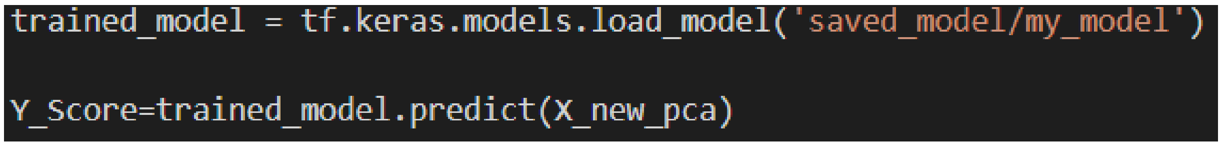
The preprocessed image is then passed into extractFeaturesTest() which will extract 252 features divided into 5 parts.
Texture = 14 features
FFT = 14 features
Wavelet = 8*14 = 112 features
GLDM = 4*14 = 56 features

GLCM = 4*14 = 56 features
So that makes total of 252 features.



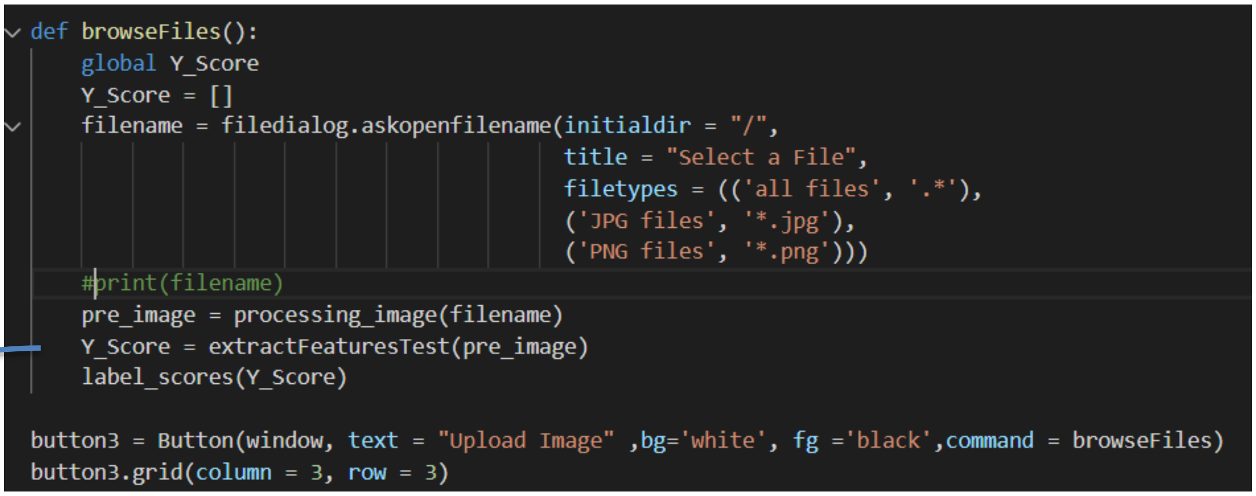
```
min_max_scaler=MinMaxScaler()
scaler = min_max_scaler.fit(X)
X_test = scaler.transform(X_test)
X = scaler.transform(X)


transformer = KernelPCA(n_components=64, kernel='linear')
pca = transformer.fit(X)

X_new_pca = pca.transform(X_test)
```

These features are then normalized and is saved in variable X_test. We have KernelPCA which is trained on saved model that we have saved after training the data into saved_model folder and will be loaded when we need to pass appropriate features. We then pass our test features into transformer into the same kernel on which we have trained the model. This will reduce the dimensionality of features from (1,252) to (1,64) so we now have 64 features to work with.



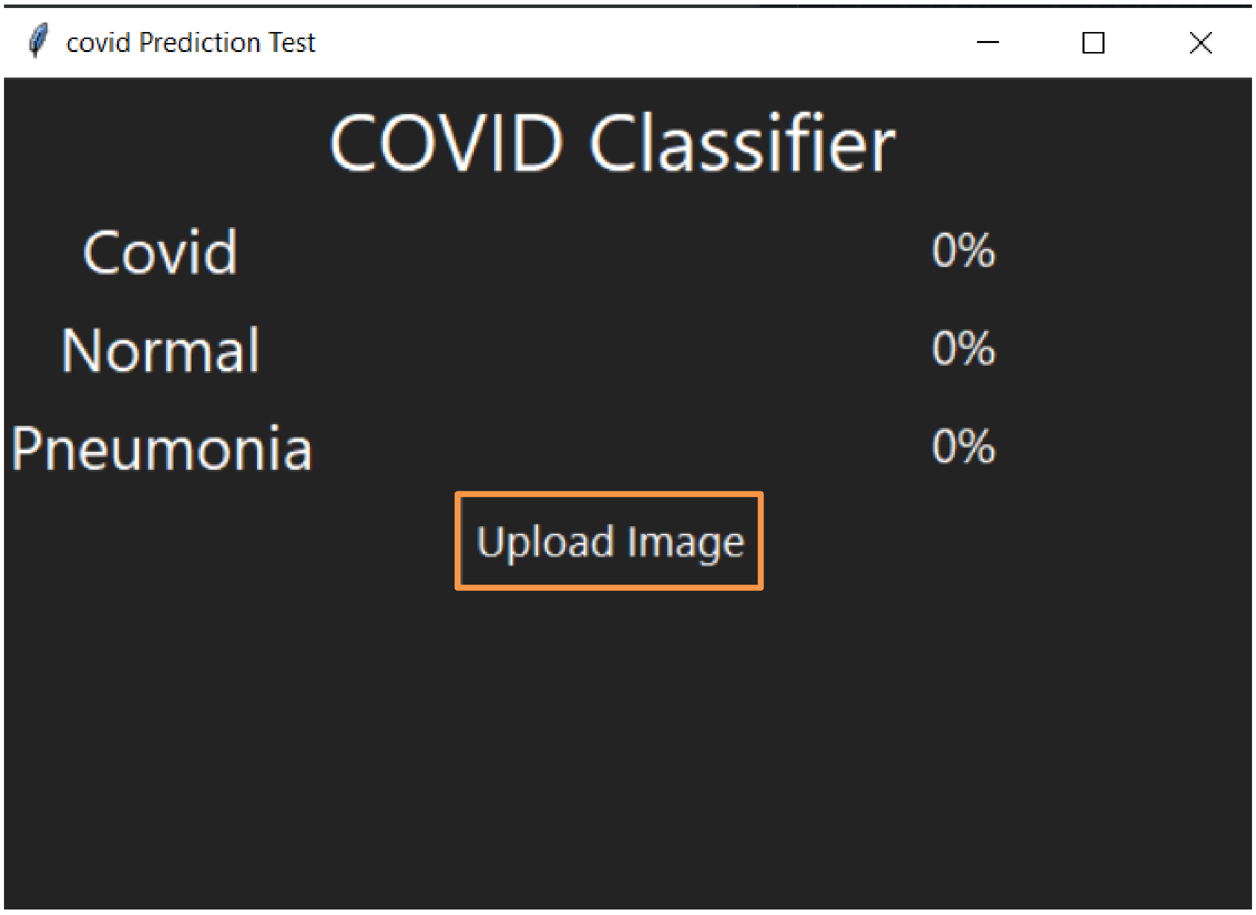
```
trained_model = tf.keras.models.load_model('saved_model/my_model')

Y_Score=trained_model.predict(X_new_pca)
```

We now obtain the saved model that we have saved when training the data and now we will pass these 64 features into our model and the model will predict the scores of our test example.
This whole working is of module predict_model.py.



```
def browseFiles():
    global Y_Score
    Y_Score = []
    filename = filedialog.askopenfilename(initialdir = "/",
                                          title = "Select a File",
                                          filetypes = (('all files', '.*'),
                                          ('JPG files', '*.jpg'),
                                          ('PNG files', '*.png')))
    #print(filename)
    pre_image = processing_image(filename)
    Y_Score = extractFeaturesTest(pre_image)
    label_scores(Y_Score)

button3 = Button(window, text = "Upload Image" ,bg='white', fg ='black',command = browseFiles)
button3.grid(column = 3, row = 3)
```

The module where our GUI resides execute this module by importing it in its module and show us the predictions.

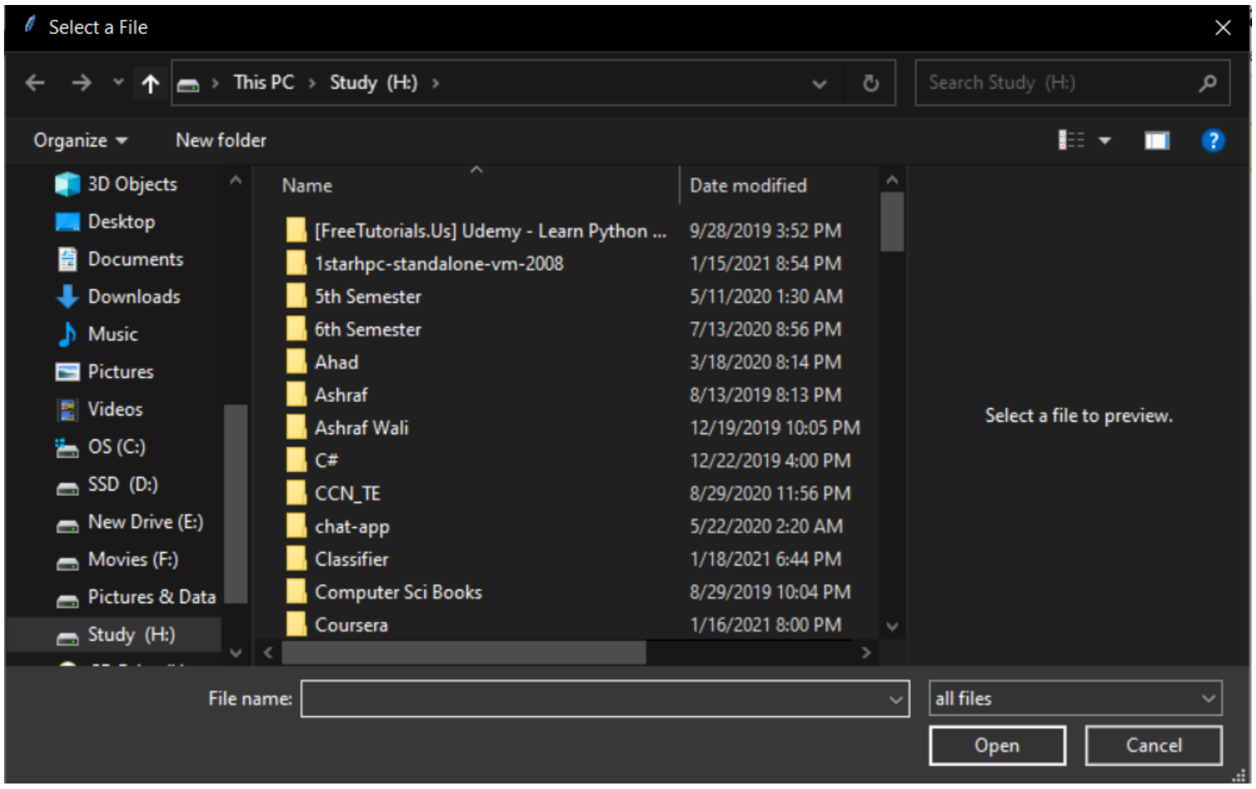


covid Prediction Test

COVID Classifier

```
Covid        0%
Normal       0%
Pneumonia    0%
```

Upload Image



Select a File

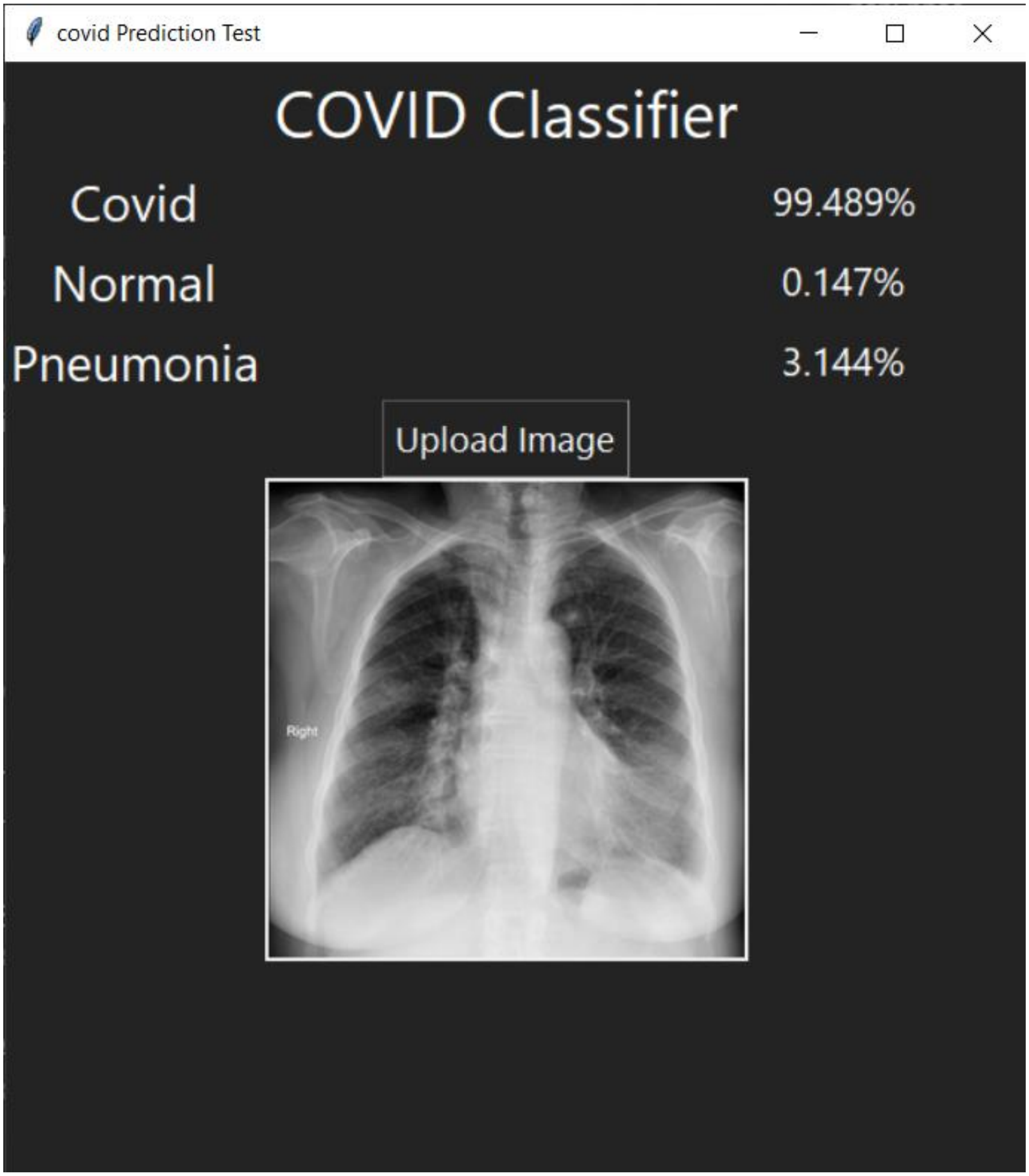
This PC › Study (H:) ›

Search Study (H:)

Organize ▾ New folder

Name	Date modified
[FreeTutorials.Us] Udemy - Learn Python ...	9/28/2019 3:52 PM
1starhpc-standalone-vm-2008	1/15/2021 8:54 PM
5th Semester	5/11/2020 1:30 AM
6th Semester	7/13/2020 8:56 PM
Ahad	3/18/2020 8:14 PM
Ashraf	8/13/2019 8:13 PM
Ashraf Wali	12/19/2019 10:05 PM
C#	12/22/2019 4:00 PM
CCN_TE	8/29/2020 11:56 PM
chat-app	5/22/2020 2:20 AM
Classifier	1/18/2021 6:44 PM
Computer Sci Books	8/29/2019 10:04 PM
Coursera	1/16/2021 8:00 PM

Upload the desired image to classify it.



covid Prediction Test

COVID Classifier

Covid	99.489%
Normal	0.147%
Pneumonia	3.144%

Individual Efforts

Members	Code Analysis	Report	Research Paper
Muhammad Taha CS-020	- utils.py: im2double() - extract_features.py: glcm - train_model.py: feature reduction, division of data into test and validation sets	- Dataset and data-source - Features in the Spatial Domain (GLCM) - Feature Reduction - Model Evaluation	- Materials and Methods - Dataset Transformation & Distribution - Feature Extraction (GLCM)
Abdul Ahad CS-022	extract_features.py: FFT, wavelet -evaluate_features.py - train_model.py: normalization, evaluate model(only CM, AUC, RUC) Predict_model.py GUI.py	- Dataflow Chart - Features in the Frequency Domain - Feature Evaluation - Model Prediction	- Feature Extraction (FFT, Wavelet, Correlation Coefficients) - Result (Confusion Matrix, classification report metrics)
Syedah Nawal Munif CS-024	- utils.py: GLDM(), build_model() - extract_features.py: gldm, texture train_model.py: build model, train model	- Features in the Spatial Domain (Texture, GLDM) - Building the Model - Fitting the Model	- Feature Extraction (Texture Analysis , GLDM, PCA) - Neural Network Model
Muhammad Mustafa Usmani CS-033	- utils.py: compute_14_features() - preprocess_images.py extract_features.py: feature_pool process_image_for_gui.py GUI.py	- Converting images to their statistics data for processing - Training the Model	- Data Preprocessing Techniques - Feature Extraction (statistical identifiers) - Result(Training and validation loss)

Bibliography

1. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7273278/#R18.
2. https://volodymyrbilyk.medium.com/computer-vision-opportunities-in-medical-imaging-explained-9e046f9e2d88.
3. https://en.wikipedia.org/wiki/Multiclass_classification
4. https://towardsdatascience.com/everything-you-need-to-know-about-min-max-normalization-in-python-b79592732b79 [1]
5. https://www.statology.org/how-to-read-a-correlation-matrix/ [2]
6. https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5 [3]
7. https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be
8. https://www.geeksforgeeks.org/dimensionality-reduction/
9. https://scikit-image.org/docs/stable/api/skimage.exposure.html
10. https://en.wikipedia.org/wiki/Adaptive_histogram_equalization
11. G. N. Srinivasan and G. Shobha, "Statistical texture analysis," *Proceedongs of World Academy of Science, Engineering and Technology*, vol. 36, pp. 1264–1269, 2008.