# FDA  Submission

Mark Abrenio

DIGITAL Pneumonia Detector

## Algorithm Description

### 1. General Information

**Intended Use Statement:**

This software can be utilized to prescan XRAY images, to help give a clinical physician some early indication that the patient might be suffering from Pneumonia.

**Indications for Use:**

This software is not to be utilized as a way of definitively determining an XRAY image does or does not contain Pneumonia. It is simply to be utilized as a diagnostic assistant to the clinician.

**Device Limitations:**

The accuracy rate of the algorithm is not high enough. We need to develop an accuracy rate of finding Pneumonia of at least 90 percent or higher. This is not possible with this data set because it is not very clean and there are not enough high quality Pneumonia XRAYs and clear XRAYs.

**Clinical Impact of Performance:**

This algorithm was never designed to replace a physician. Simply to give them a hint of what to look for with a specific patient. Just as a high bp reading does not immediately mean a clinician diagnoses hypertension, a positive Pneumonia notification from this algorithm would not mean a clinician would automatically diagnose pneumonia. This is simply a diagnostic aid. However, an accuracy rate of over 90 percent would greatly assist a clinician in accurately diagnosing Pneumonia.

### 2. Algorithm Design and Function

<< Insert Algorithm Flowchart >>

**DICOM Checking Steps:**

Our algorithm can utilize the following function to verify a DICOM file. We read the dicom and parse it using the pydicom library. Then verify the modailty is XRAY and the body part examined is the chest of the patient.

```
def check_dicom(filename):

    ds = pydicom.dcmread(filename)
    mod = ds[0x0008, 0x0060].value
    b_part_examined = ds[0x0018, 0x0015].value
    patient_position = ds[0x0018, 0x5100].value
    acceptance = 1

    print("Modality: " + mod)
    print("Body Part Exmained: " + b_part_examined)
    print("Patient Position: " + patient_position)

    if patient_position != "AP" and patient_position != "PA":
        acceptance = 0

    if b_part_examined != "CHEST":
        acceptance = 0

    if mod != "DX":
        acceptance = 0

    img = ds.pixel_array

    if acceptance:
        print("Image Accepted")
        return img

    if not acceptance:
        print("Image Rejected")
        return 0

for idx in test_dicoms:
    img = check_dicom(idx)
    print("\n")
```

**Preprocessing Steps:**

We use the TensorFlow ImageDataGenerator class to create an easy to use iterable data loader for our XRAY images. Some effective augmentations we use are samplewise_std_normalization = True, rotation_range = 15, zoom_range = 0.25,brightness_range=[0.35, 1.1], and samplewise_center = True.

**CNN Architecture:**

This model uses the VGG 16 architecture.

### 3. Algorithm Training

**Parameters:**
* Types of augmentation used during training
* Batch size
* Optimizer learning rate
* Layers of pre-existing architecture that were frozen
* Layers of pre-existing architecture that were fine-tuned
* Layers added to pre-existing architecture

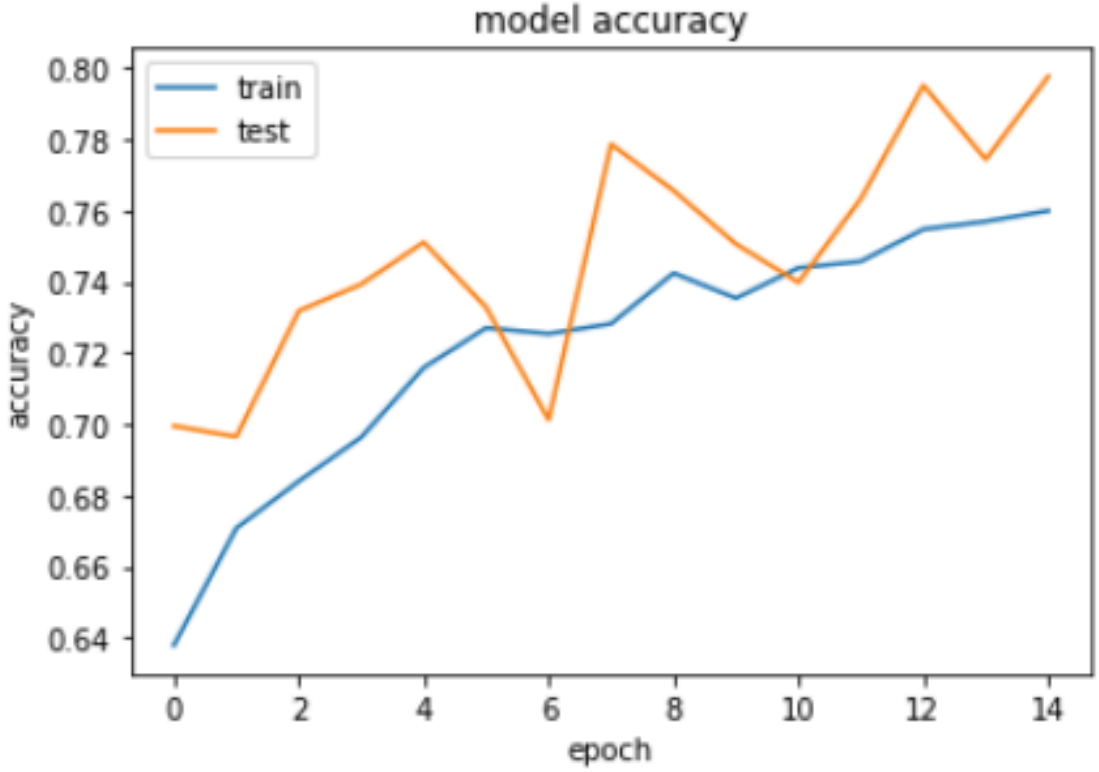
We primarily use the function below to load the pertained VGG16 model for training. The only thing we really do to the model is change the input layer to accept 224 x 224 images, and then set the entire model except for the output layer to non trainable so we don't have to train the entire model again. It takes too long to train the entire model and we are limited in GPU speed.

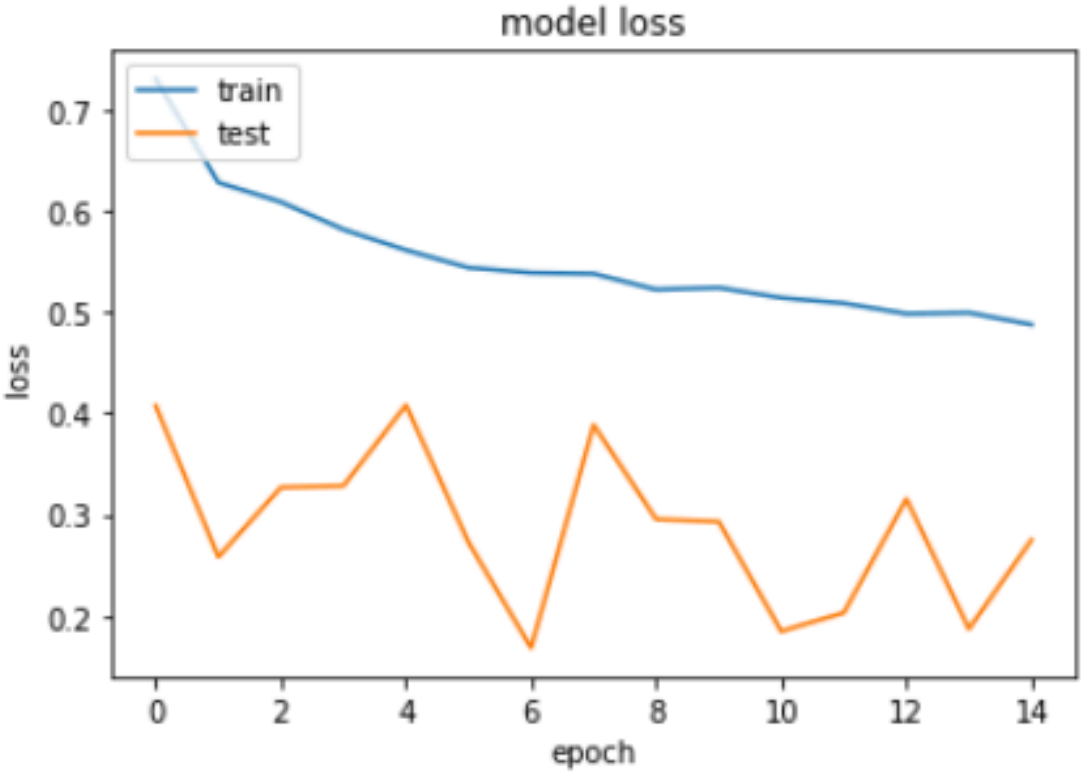
```
def load_pretrained_model(number_output, load_old_model=0, file_name_json=0, file_name_

    # model = VGG16(include_top=True, weights='imagenet')
    # transfer_layer = model.get_layer(lay_of_interest)
    # vgg_model = Model(inputs = model.input, outputs = transfer_layer.output)

    # Todo
    from keras.optimizers import Adam
    number_output = 1 #**need to whittle it down to only look for pneumonia!

    print(device_lib.list_local_devices()) #show if we have GPU ready or not!

    if load_old_model:
        try:
            print("Loading Saved Model")
            model = load_model1(file_name_json, file_name_weights)
            print("SUCCESS: Returning saved model!")
            from datetime import datetime
            from keras.callbacks import ModelCheckpoint, LearningRateScheduler
            from keras.callbacks import ReduceLROnPlateau
            lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                                           cooldown=0,
                                           patience=5,
                                           min_lr=0.5e-6)

            return model, lr_reducer
        except Exception as e:
            print("ERROR: Could Not Load Saved Model. TERMINATING")
            print(e)
            sys.exit(1)

    IMAGE_SIZE = [224, 224] #**use a little bigger images, to get more xray data!
    vgg = VGG16(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False)

    # don't train existing weights GPU TOO SLOW
    for layer in vgg.layers:
        layer.trainable = False

    x = Flatten()(vgg.output)
    prediction = Dense(number_output, activation='sigmoid')(x)
    model = Model(inputs=vgg.input, outputs=prediction)

    model.compile(optimizer = Adam(lr=0.0007), loss='binary_crossentropy', metrics=['bi
    model.summary()

    from datetime import datetime
    from keras.callbacks import ModelCheckpoint, LearningRateScheduler
    from keras.callbacks import ReduceLROnPlateau
    lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                                   cooldown=0,
                                   patience=5,
                                   min_lr=0.5e-6)
```

The code below displays the augmentations we used for the XRAY images. The varying brightness range and slight in and out zoom seem to help the model learn the XRAYs a lot better. With these simple augmentation the VGG16 model learns the XRAYs pretty well. But it could learn them better if the data was a little cleaner. Some of the Pneumonia XRAYs don't look very clear, and I think it confuses the model some.

```
def make_train_gen(class_column_list, patched_df): #**lets provide the dataframe here

    class_column_list = ["Infiltration", "Pneumonia"] #*just narrow it down to notice
                                                      #*Pneumonia images!

    batch_size = 64

    datagen = ImageDataGenerator(samplewise_std_normalization = True,
                                       rotation_range = 15,
                                       zoom_range = 0.25,
                                       brightness_range=[0.35, 1.1],
                                       samplewise_center = True
                                      )

    train_generator=datagen.flow_from_dataframe(
    dataframe=patched_df,
    directory="None",
    x_col="path",
    y_col="Pneumonia_or_Infiltration",
    subset="training",
    batch_size=batch_size,
    seed=42,
    shuffle=True,
    class_mode="binary",
    target_size=(224,224)) #**use bigger images to get more xray data! :)

    print("Train Loader Ready")

    print("Number of Batches of Train_Generator: " + str(len(train_generator)))

    return train_generator#, valid_generator
```

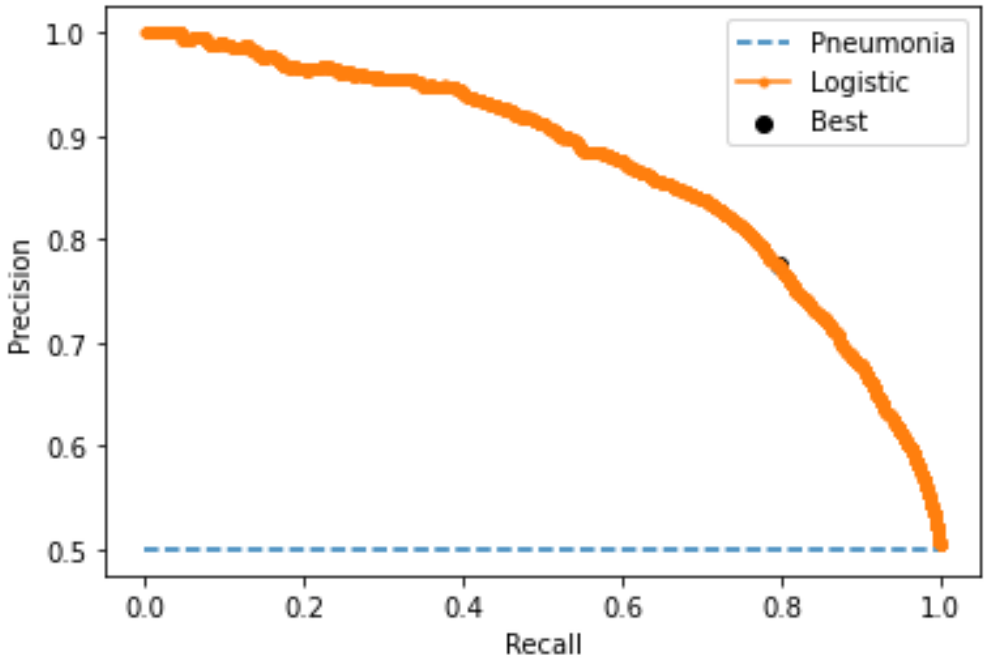
<< Insert algorithm training performance visualization >>


model accuracy
train
test
accuracy
epoch
0.80
0.78
0.76
0.74
0.72
0.70
0.68
0.66
0.64
0
2
4
6
8
10
12
14


model loss
train
test
loss
epoch
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
14

<< Insert P-R curve >>

```
Best Threshold=0.392611, F-Score=0.786
0.39261103
```




We can see from the illustration below that our model for the XRAYs is not multi class, but rather based on binary cross entropy — either Pneumonia or not Pneumonia. Also we use an ADAM optimizer with a learning rate of 0.0007.

```
    model.compile(optimizer = Adam(lr=0.0007), loss='binary_crossentropy', metrics=['binary_accuracy'])
    model.summary()

    from datetime import datetime
    from keras.callbacks import ModelCheckpoint, LearningRateScheduler
    from keras.callbacks import ReduceLROnPlateau
    lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                                   cooldown=0,
                                   patience=5,
                                   min_lr=0.5e-6)

    print("SUCCESS: Returning built model!")

    return model, lr_reducer
```

**Final Threshold and Explanation:**

I utilized information from this source to find the optimal threshold: https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/
Specific information from this site is illustrated below:

There are many ways we could locate the threshold with the optimal balance between false positive and true positive rates.

Firstly, the true positive rate is called the Sensitivity. The inverse of the false-positive rate is called the Specificity.

- Sensitivity = TruePositive / (TruePositive + FalseNegative)
- Specificity = TrueNegative / (FalsePositive + TrueNegative)

Where:

- Sensitivity = True Positive Rate
- Specificity = 1 – False Positive Rate

The Geometric Mean or G-Mean is a metric for imbalanced classification that, if optimized, will seek a balance between the sensitivity and the specificity.

- G-Mean = sqrt(Sensitivity * Specificity)

One approach would be to test the model with each threshold returned from the call roc_auc_score() and select the threshold with the largest G-Mean value.

Given that we have already calculated the Sensitivity (TPR) and the complement to the Specificity when we calculated the ROC Curve, we can calculate the G-Mean for each threshold directly.

```
...
# calculate the g-mean for each threshold
gmeans = sqrt(tpr * (1-fpr))
```

Once calculated, we can locate the index for the largest G-mean score and use that index to determine which threshold value to use.

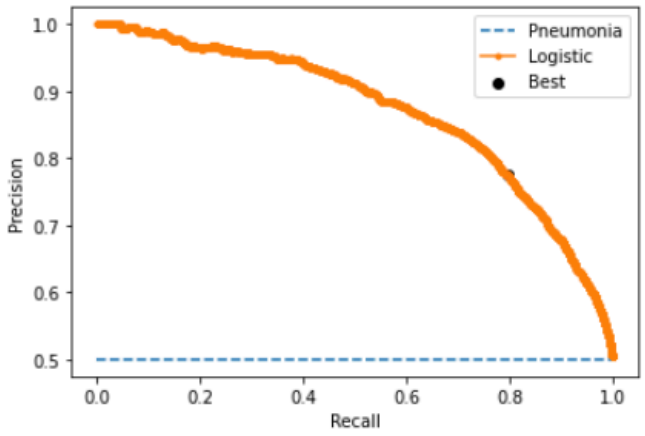
```
...
# locate the index of the largest g-mean
ix = argmax(gmeans)
print('Best Threshold=%f, G-Mean=%.3f' % (thresholds[ix], gmeans[ix]))
```

We can also re-draw the ROC Curve and highlight this point.

This blog resulted in me utilizing the following code in the project, to find a good threshold to use for the model:

```
precision, recall, thresholds = metrics.precision_recall_curve(valY_val, pred_Y_val)
# convert to f score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

best_threshold = thresholds[ix]
print(best_threshold)

# plot the roc curve for the model
no_skill = len(valY_val[valY_val==1]) / len(valY_val)
pyplot.plot([0,1], [no_skill,no_skill], linestyle='--', label='Pneumonia')
pyplot.plot(recall, precision, marker='.', label='Logistic')
pyplot.scatter(recall[ix], precision[ix], marker='o', color='black', label='Best')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
pyplot.legend()
# show the plot
pyplot.show()
```

```
Best Threshold=0.392611, F-Score=0.786
0.39261103
```

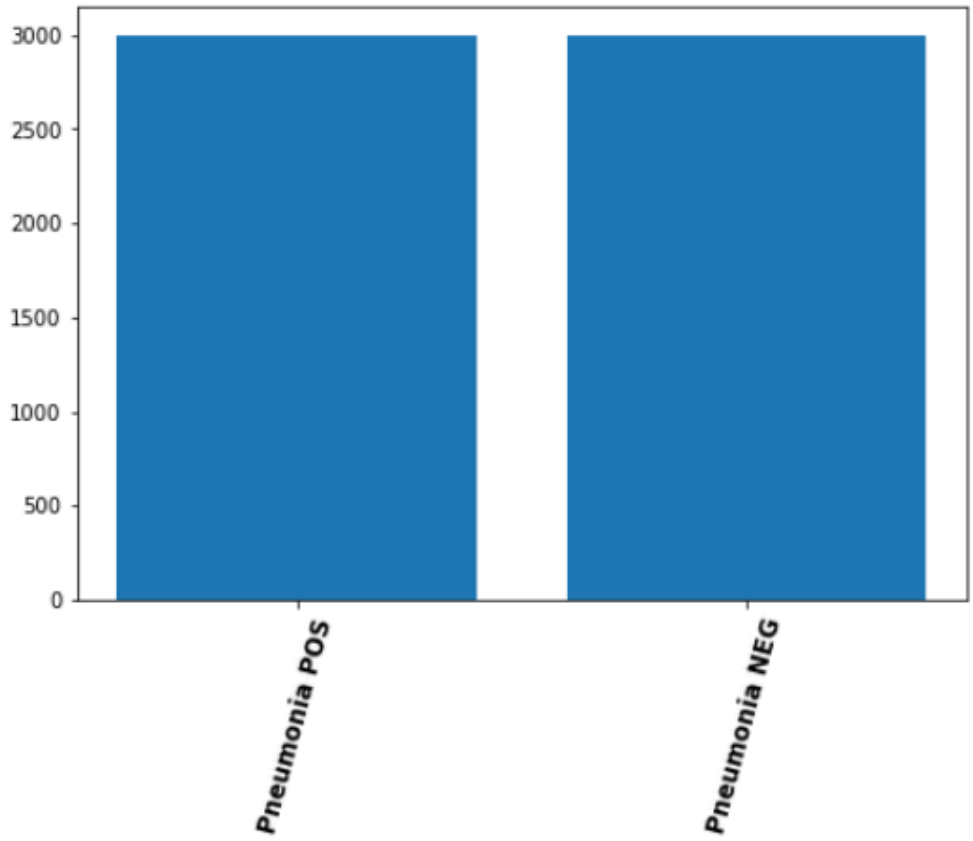



### 4. Databases
(For the below, include visualizations as they are useful and relevant)

**Description of Training Dataset:**

For the dataset I grouped infiltration and pneumonia together, since the XRAY images look similar in nature. Both tend to show a highlighted blob within the lung, and there were not very many Pneumonia XRAY images to begin with. The training set is balanced and plotted below. There are more images available to use since we group Pneumonia and Infiltration together, but the GPU we have for this project is too slow. So I am only using 6 thousand total images for training.

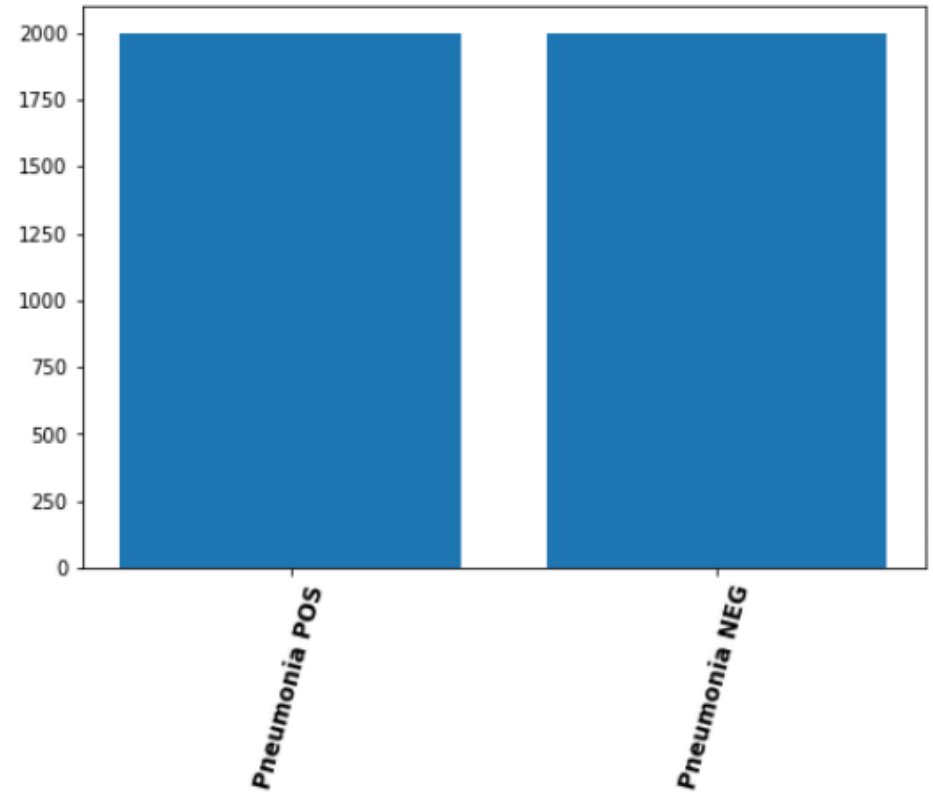
```
PLOT TRAINING DATA. POSITIVE AND NEGATIVE PNEUMONIA IMAGES (BALANCED)
Positive Images: 3000
Negative Images: 3000
```




**Description of Validation Dataset:**

I used the same methodology for the validation set. I counted all images that were either Pneumonia or Infiltration positive, as Pneumonia positive XRAY images. I have an approximate balance of Positive and Negative images to validate with. The plot of the positive and negative validation images is illustrated below.

```
PLOT VALIDATION DATA. POSITIVE AND NEGATIVE PNEUMONIA IMAGES (BALANCED)
Positive Images: 2000
Negative Images: 2000
```




### 5. Ground Truth

The ground truth I used was simply based on the truth labels provided in the NIH dataset. Although , I did notice that some of the images appeared to be false positives and some of the images were very low quality. If the dataset was cleaned up I think the model would be much more effective at generalizing on Pneumonia XRAY images.

### 6. FDA Validation Plan

**Patient Population Description for FDA Validation Dataset:**

Ideally, this model needs to be trained with thousands of known pneumonia images, and thousands of known clear images. The NIH dataset is not really clean. It has some very low quality XRAYs and some apparent false positives. Ideally, this algorithm would be continually trained alongside an active clinicians ongoing manually diagnoses of many patients, so it gets better the longer it works beside an actual physician.

**Ground Truth Acquisition Methodology:**

Ideally, this algorithm would be continually trained alongside an active clinicians ongoing manually diagnoses of many patients, so it gets better the longer it works beside an actual physician.

**Algorithm Performance Standard:**

This algorithm was never designed to replace a physician. Simply to give them a hint of what to look for with a specific patient. Just as a high bp reading does not immediately mean a clinician diagnoses hypertension, a positive Pneumonia notification from this algorithm would not mean a clinician would automatically diagnose pneumonia. This is simply a diagnostic aid. However, an accuracy rate of over 90 percent would greatly assist a clinician in accurately diagnosing Pneumonia.