Predicting Pediatric Pneumonia

COSC 522: Machine Learning
Group 7: Rekha Bhupatiraju, Owen Queen, Andrew Strick, Sai Thatigotla

Summary: In this project, we explore classifying chest X-ray images in order to detect the presence of pneumonia in pediatric patients. A variety of machine models are trained on the image data in order to achieve this goal. We then compare all of our findings in a table at the end of this report.

Contents

Problem Statement

Pneumonia is a respiratory condition caused by inflammation of the lungs. This condition is typically diagnosed through the use of X-Ray images, which can reveal white spots on the lungs, as shown below in two X-Ray images of the lungs:

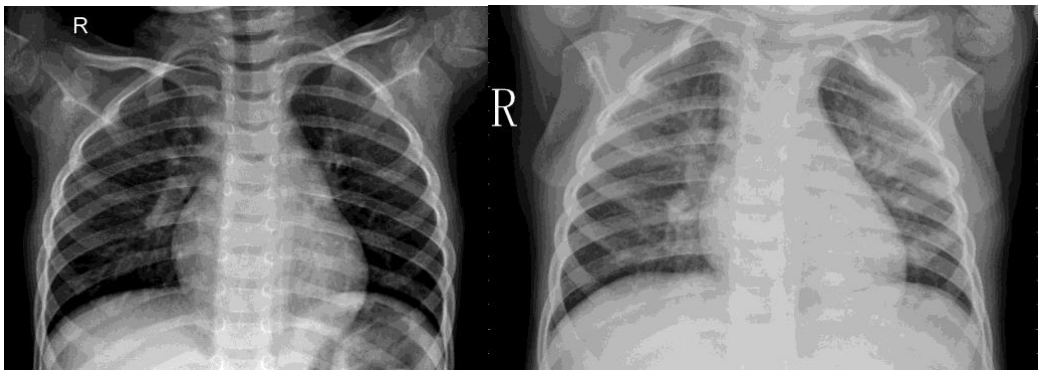


Figure 1: The left image shows the lungs of a patient who does not have pneumonia, and the right image shows a patient with pneumonia. Pneumonia is characterized by the white spots in the lungs, revealing inflammation caused by infection by a bacterial or viral source. Both images are from [1].

Accurately diagnosing the presence of pneumonia in a patient is critical because this condition can lead to further respiratory complications, including death. Chest X-Rays are the most common method of diagnosing pneumonia. It often takes a physician to be able to diagnose pneumonia from an X-Ray, but being able to predict pneumonia in the absence of a physician could aid various medical clinics around the world who treat cases of pneumonia. Our goal is to create a model that predicts the presence of pneumonia given a chest X-Ray image such as one of those above.

The Dataset

Our dataset, downloaded from Kaggle [1], contains 5856 samples of chest X-ray images from pediatric patients in Guangzhou, China. Each sample is an image in RGB jpeg format. On Kaggle, the train, validation, testing split had been predetermined by the user who uploaded the dataset, and we decided to stick to these splits in order to remain consistent across all of our models and allow for more accurate comparison to other notebooks on Kaggle.

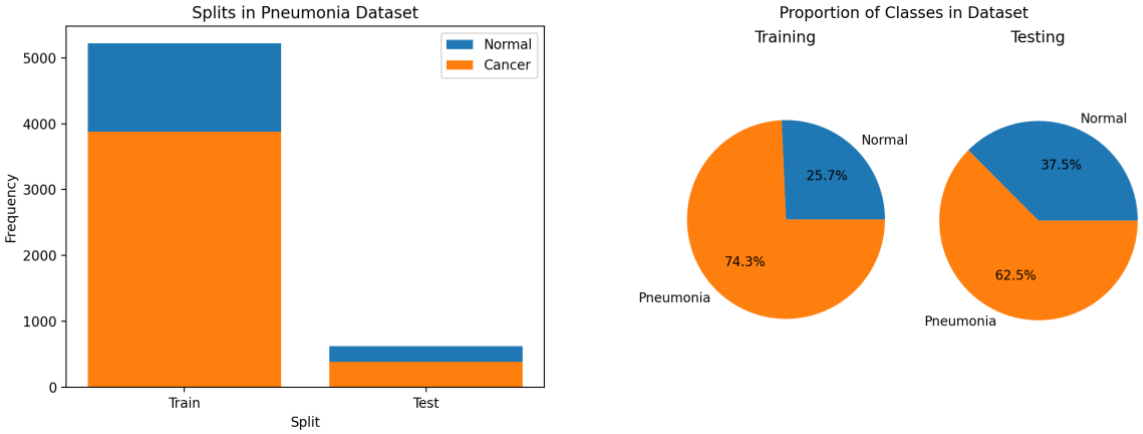


Figure 2: These charts show the imbalance in the overall dataset. This imbalance would plague our training process, especially with the convolutional neural network. Please note that the validation set is not shown here because it only consisted of 16 samples - 8 with pneumonia and 8 without pneumonia.

Each of the images were also of varying sizes. The largest images were over 1000 pixels tall and wide, and the smallest were around 100 pixels wide and 300 pixels tall. The sizes would have to be standardized before training the models.

Preprocessing

First, each image had to be standardized to a common size. This was done through the use of the Python Imaging Library, and the images were resized to (700, 1000, 1), a size slightly less than the mean size of images in the dataset. In order to use these samples practically in our models, we first had to preprocess the images through normalization and dimensionality reduction.

Data Normalization

Normalization for these images was performed by converting each image to grayscale. This brings each pixel value to an integer between 0 and 255.

For the models where more standardized pixels performs better, such as CNN, the following transformation was applied to each pixel *a* in the images:

$$a_{ij} = \frac{a_{ij}}{255}$$

This forced the largest pixel to be 1 and the smallest pixel to be 0.

Dimensionality Reduction

For images, the typical dimensionality reduction techniques that we have learned in this course, principal component analysis (PCA) and Fisher's linear discriminant (FLD), are not very commonly used. Other methods such as singular value decomposition (SVD) are popular for preprocessing and t-SNE is popular for visualization. However, in this project, we decided to stay simple and use max pooling instead of these other methods.

Max pooling reduces the dimension of an image by taking the maximum pixel value in a neighborhood of pixels and reducing that neighborhood to only one pixel - the value of the computed maximum.

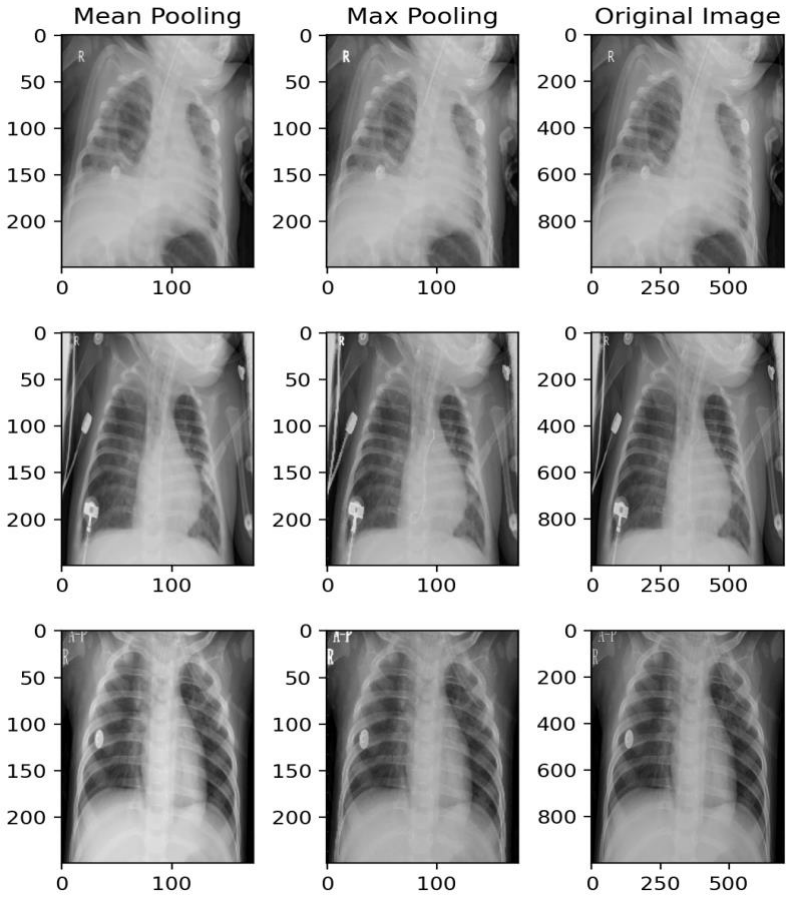


Figure 3: Different pooling methods compared to the original image. Mean pooling is similar to max pooling, but it takes the mean of the pixels in a neighborhood instead of the maximum. Max pooling seems to be a popular choice in practice, so we chose max pooling to create our final images.

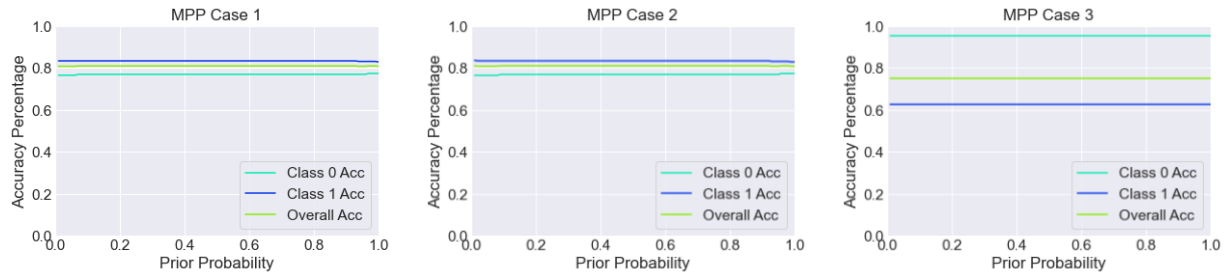
For pooling, we used two pooling layers of 2x2 filters with stride 2. This had the effect of greatly reducing the storage and computational power needed to process and run machine learning

models with these images. We were able to reduce images with an initial size of (700, 1000) to a size of (175, 250), a 94% reduction in the total number of pixels in each image.

MPP (Maximum Posterior Probability)

Figures 4-6

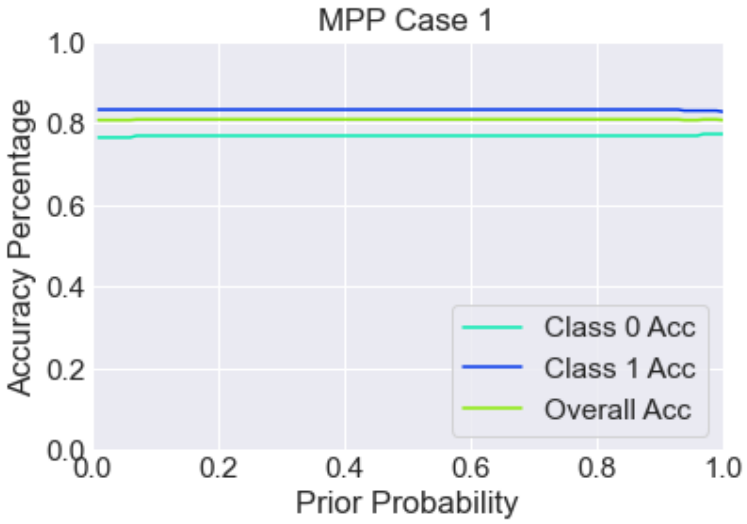
Comparison of Accuracy Scores with Changes to Prior Probability on the Testing Dataset



As for the comparison of classwise and overall accuracy, changes to prior probability resulted in almost no differences across the each case of MPP. There were exactly no changes in MPP Case 3 no matter the variance of prior probability input. Additionally MPP Case 1 and Case 2 resulted in the same accuracy outcomes which differed from Case 3.

Figure 7

Comparison of Accuracy scores in MPP Case 1

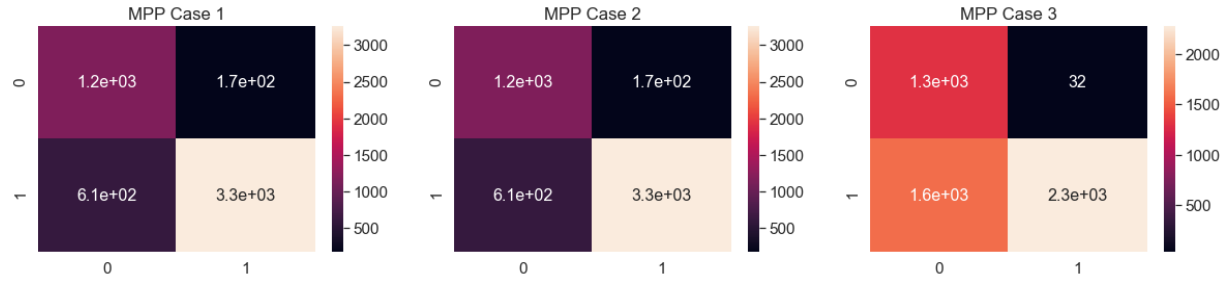


For MPP Case 1 (see Figure 7) and MPP Case 2 (not seen here) you can see small deviations or changes in accuracy scores near the extremes of prior probabilities. For instance, notice a small increase in Class 0 accuracy (normal, pneumonia free) as prior probability increases beyond roughly five percent and again increases at roughly 95 percent. There is a small decrement in class 1 (pneumonia) accuracy performance at roughly 92 percent.

Taken together, changes in prior probability did not play a significant role in determining accuracy scores across three separate cases of maximum posterior probability (MPP).

Figures 8-10

Comparison of Confusion Matrices on 5-fold Cross Validation for three Separate Cases of MPP.



Note. MPP = Maximum Posterior Probability.

5-fold cross validation was performed in order to verify the validity of the accuracy scores so the model could be evaluated generally instead of specifically to the training and testing splits. MPP Case 1 and Case 2 did not differ as the assumptions changed from classification using minimum euclidean distance (Case 1; $\sum_i = \sigma I$) compared to using minimum mahalanobis distance (Case 2; $\sum_i = \sum$). Changing of distance metrics between Case 1 and Case 2 led to no changes in resultant classifications while performing 5-fold cross validation.

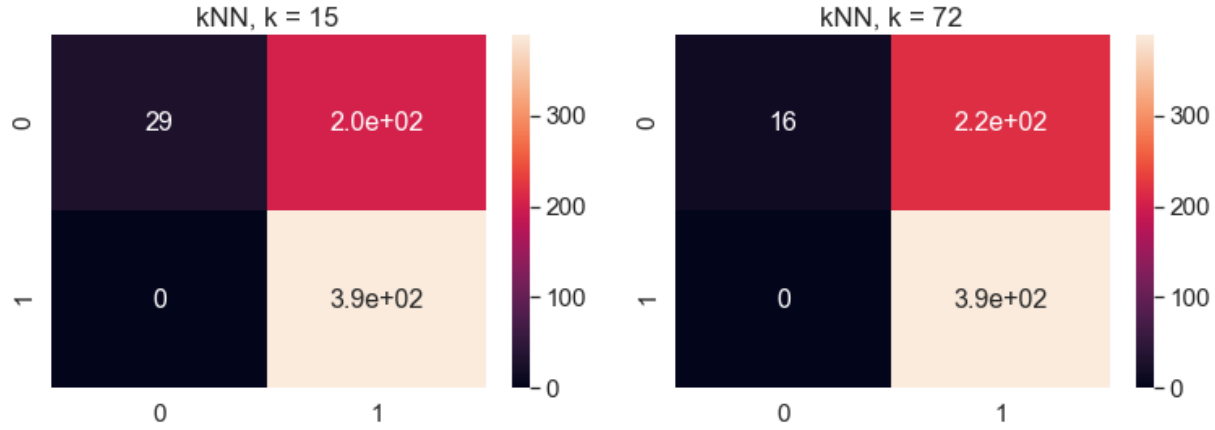
MPP Case 3 did differ from both MPP Case 2 and MPP Case 1. From looking at the confusion matrix for Case 3 there was a dramatic reduction in the number of false positives (n = 32; where n is the number of classifications) and dramatic increase in the number of false negatives ($n = 1.6 * 10^3$) in comparison Case 2 and Case 3. There was also a slighter changes with a small increase in the number true positives in Case 3 ($n = 1.3 * 10^3$) and a small decrease in the number of true negatives ($n = 2.3 * 10^3$) relative to the other two cases (see Figures 8-10).

From the confusion matrices seen in Figures 5-7 it can be deduced that using a quadratic classifier (Case 3; arbitrary $\sum_i$) lead to increases in classifying the number of true negatives but at the detriment of increases the number of false negative classifications, and, more importantly, led to decreases to the number of true positives. Although Case 3 would be better at classifying those that do not have pneumonia, it would also misclassify more pneumonia cases which could lead to putting more patients at risk in comparison to Case 1 and Case 2. Additionally, it worth noting that there were no differences observed between Case 1 and Case 2 in any area of classification, which means the changing distance metrics from euclidean to mahalanobis distance did not lead to any meaningful differences in resultant accuracy scores.

kNN (k-Nearest Neighbor)

Figures 11-12

Comparison of Confusion Matrices for two different k's on the Testing Dataset.



Note. kNN = k-Nearest Neighbor, where k = the number of nearest neighbors. Also, it took 41.8 and 46.3 hours to run kNN on the dataset which is why there are only two matrices here.

Confusion matrices were created in order to compare two different k values when using the k-Nearest Neighbor classification method. The important similarities between both k values is that they both showed zero false negative classifications, which means that there were zero patients that had pneumonia classified as normal. Also, differences in k value did not reveal any changes in the number total true positive classifications. This means that all pneumonia x-rays were correctly classified regardless of changes in the number of nearest neighbors.

Table 1

Comparison of Accuracy Metrics between two Different k Values.

	kNN, k = 15	kNN, k = 72
Sensitivity	100%	100%
Specificity	12.4%	6.8%
Overall Accuracy	67.1%	65.0%

Note. kNN = k-Nearest Neighbor, where k = the number of nearest neighbors.

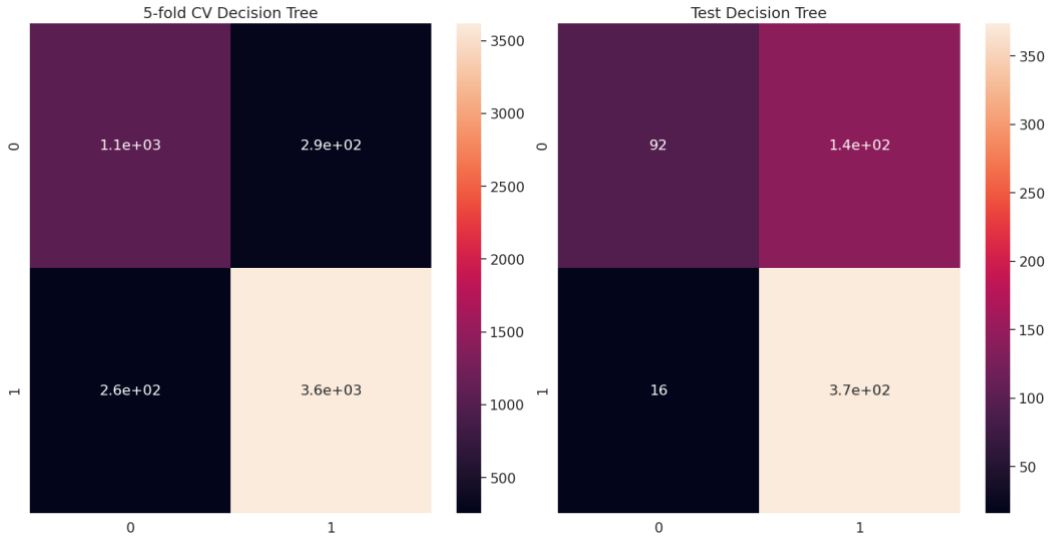
There are important differences between the changes of k, specifically that as k increased the total number of true negatives decreased. This would suggest that any further increases in the number of nearest neighbors would actually decrease the classification accuracy of non-

pneumonia x-rays. Furthermore, the specificity decrease found as the number of nearest neighbors increases lead to an decrease in overall accuracy, with no other changes found between the two (see Table 1).

Decision Tree

Figure 13-14

Confusion matrix of Decision Tree on 5-Fold CV (left) and on Test (Right)



The above two figures are the confusion matrices for Decision Tree on 5-fold CV and on the held-out test set. In both cases, the Gini Impurity was used to optimize the tree. During training (5-fold CV), the tree achieved an overall accuracy of 89.5%, Sensitivity of 93.4%, and specificity of 78.4%. During the testing set, it achieved overall of 74.7%, a sensitivity of 95.9% and a specificity of 39.7%. The model was clearly overfitting, which makes sense given that the imbalance nature of the dataset makes it harder to generalize.

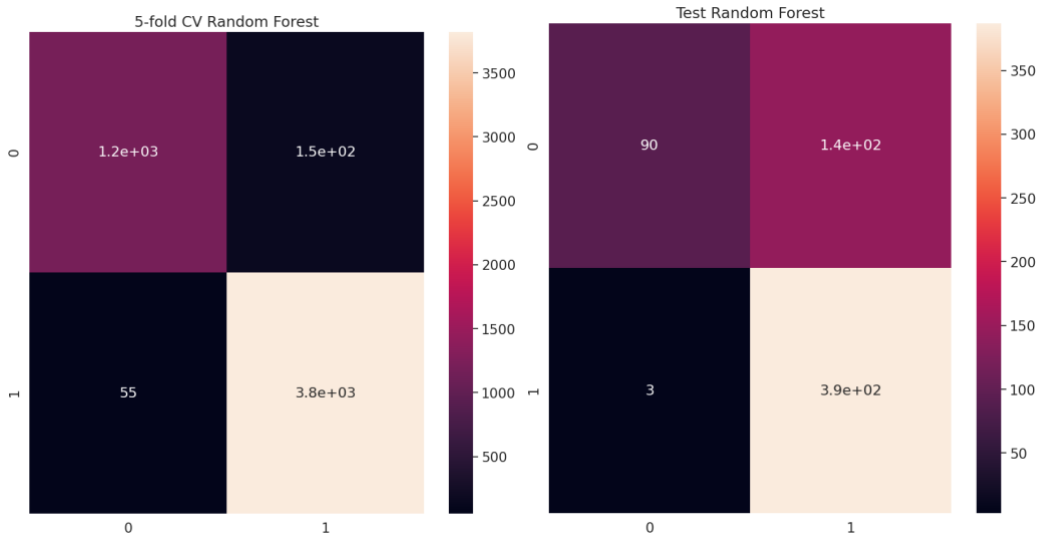
The following figure shows the decision tree that was learned. It is not as interpretable for this problem as the column values, which would normally be specific features, are instead pixel columns.

Figure 15

Random Forest

Figure 16-17

Confusion matrix of Random Forest on 5-Fold CV (left) and on Test (Right)

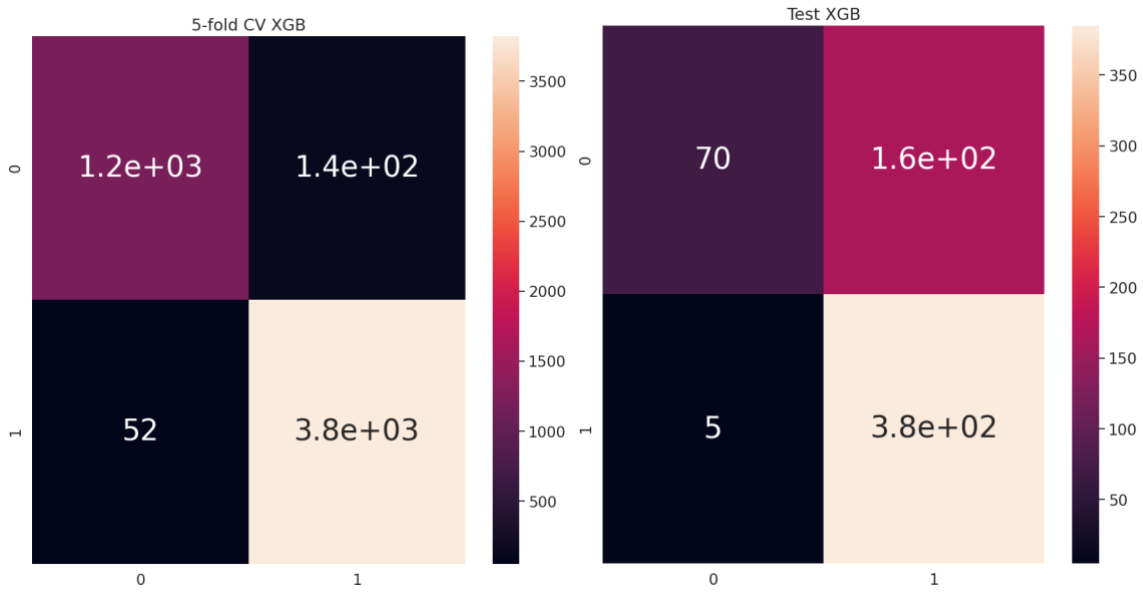


The above two figures are the confusion matrices for the Random Forest classifier. Like with the previous models, the left is the 5-fold CV on the train set and the right is the matrix for the test set. They both included 100 trees. For the training set, the overall accuracy is 96.1%, sensitivity of 98.6%, and specificity of 88.8%. For the test set, the overall accuracy is 76.4%, sensitivity of 99.2%, and specificity of 39.1%. Like with Decision Tree, Random Forest also used Gini Impurity to optimize their trees. Also, this model overfits like with Decision Tree with the specificity dropping from 88.8% to 39.1%. However, while the training accuracy was lower tha decision tree, the sensitivity was higher in the random forest in the test set suggesting that random forest was better at generalizing.

XGBoost

Figure 18-19

Confusion matrix of Random Forest on 5-Fold CV (left) and on Test (Right)

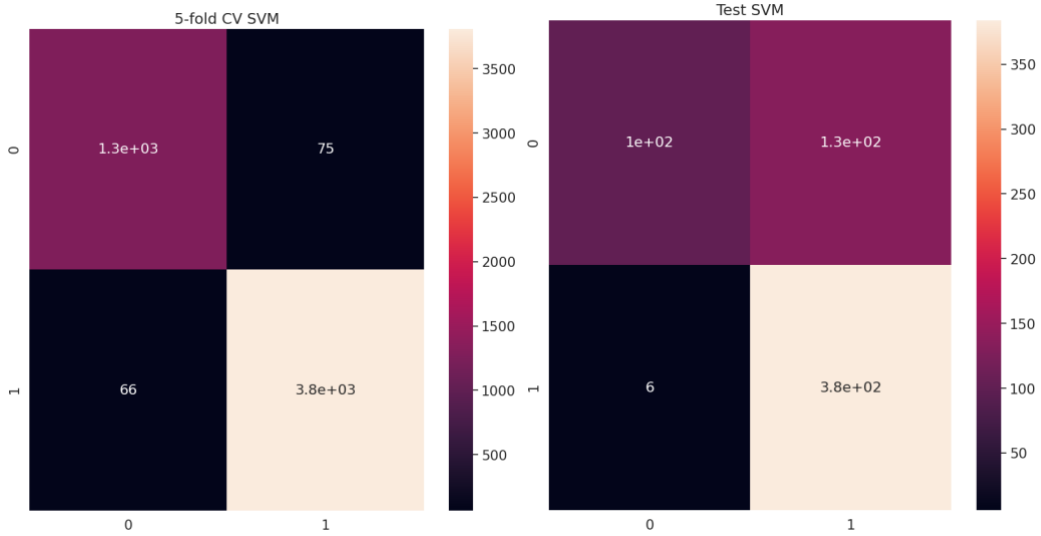


The above two graphs are the confusion matrices for XGBoost with the left for 5-fold CV (train) and the right for the test set. For the train, the overall accuracy is 96.4%, the sensitivity is 98.7%, and the specificity is 89.8%. For the test, the overall accuracy is 73.2%, the sensitivity is 98.7% and the specificity is 30.4%. The learning rate was 0.1, max depth was 3 and the number of estimators was 100. This model overfit like the previous models, but it was a little surprising that it did not outperform Random Forest or decision tree. However, hyperparameter tuning and/or weighing the positive class less might help with generalization.

SVM

Figure 20-21

Confusion matrix of SVM on 5-Fold CV (left) and on Test (Right)



The above two graphs are the confusion matrices for SVM with the 5-fold CV on the training set on the left and the test set on the right. This SVM model used the Radial-Basis Function Kernel (RBF). For the training set, the overall accuracy is 97.3%, the sensitivity is 98.3% and the specificity of 94.4%. For the testing set, the overall accuracy is 77.7%, the sensitivity is 98.5%, and the specificity is 43.5%. While this model also overfit (overall went from 97.3% to 77.7%), it generalized better than some of the previous models as its sensitivity stayed the same, but its specificity dropped less than for the other models such as decision tree and random forest.

Clustering

The clustering algorithms applied to this dataset were Agglomerative Clustering and Kmeans. Initial tests with two clusters to align with the number of classes resulted in low accuracies - around 74%. Accuracy was greatly improved by an approach that created more clusters and then classified those. Both algorithms showed great increases in accuracy up to around 40 clusters, at which point they plateaued out in their improvement.
Because these algorithms are unsupervised, they were used just to classify the training data set, leading to a higher accuracy than other algorithms. Both algorithms had higher sensitivity than specificity, because the training data was largely class one.

Agglomerative Clustering

Agglomerative clustering doesn't have any significant time cost as number of clusters increases. It does have a reduction of improvement at 30 clusters.

Figure 22-23
Overall accuracy (left) and time cost (right) in agglomerative clustering for different numbers of clusters - we tested 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, and 90

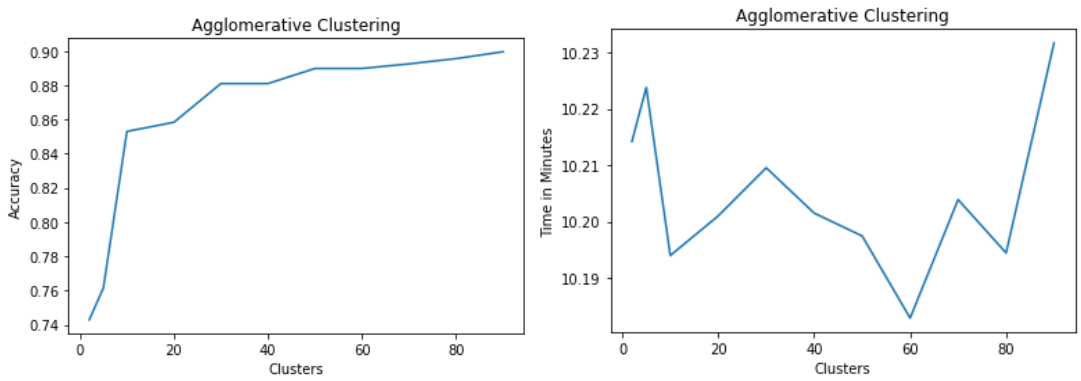


Table 2
Confusion matrix for 50 clusters (10:11.84 minutes):

Predictions >	0	1	Accuracy: 0.8898
0	970	371	Specificity: 0.7233
1	204	3671	Sensitivity: 0.9473

KMeans

Kmeans, unlike agglomerative clustering, had a steadily increasing time cost to adding more clusters, so it is more important that a point is found where additional improvements in accuracy are no longer dramatic. That point seems to be when it reaches about 88% accuracy around 40 clusters. This corresponds with a runtime of sixteen and a half minutes.

Figure 24-25

Accuracy (left) and time cost (right) for different numbers of clusters in KMeans. We tested the same values as with Agglomerative clustering.

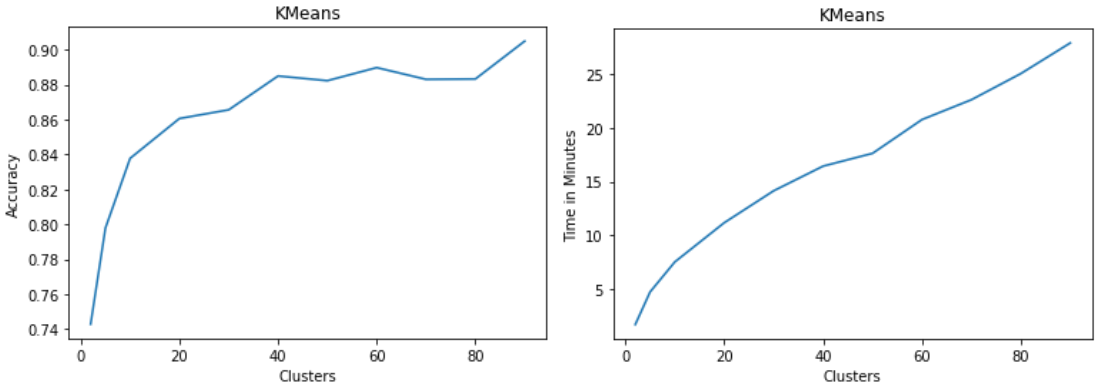


Table 3

Confusion matrix for 50 clusters (17:37.96 minutes):

Predictions >	0	1	Accuracy: 0.8823
0	1004	331	Specificity: 0.7635
1	277	3598	Sensitivity: 0.9285

Comparison

Agglomerative clustering and KMeans have comparable accuracy at all levels. If smaller numbers of clusters are accurate or lower accuracy is acceptable, then KMeans has a much shorter runtime. For this dataset, KMeans has better time cost until around 86% percent accuracy. It may be the better choice for larger datasets.

Back-propagation Neural Network (BPNN)

In addition to the other models, we also trained a BPNN model on the training dataset. CNN's involve a lot of separate hyperparameters, so we had to construct experiments to find the optimal hyperparameter for this dataset. These experiments all consisted of cross-validation techniques, but we used the entire dataset - training, testing, and validation - to validate the choices of hyperparameters. The exploration of hyperparameters is shown below:

Architecture

This BPNN structure is based off of CNN structures found in the Kaggle notebooks. A visual depiction of the network is shown below:

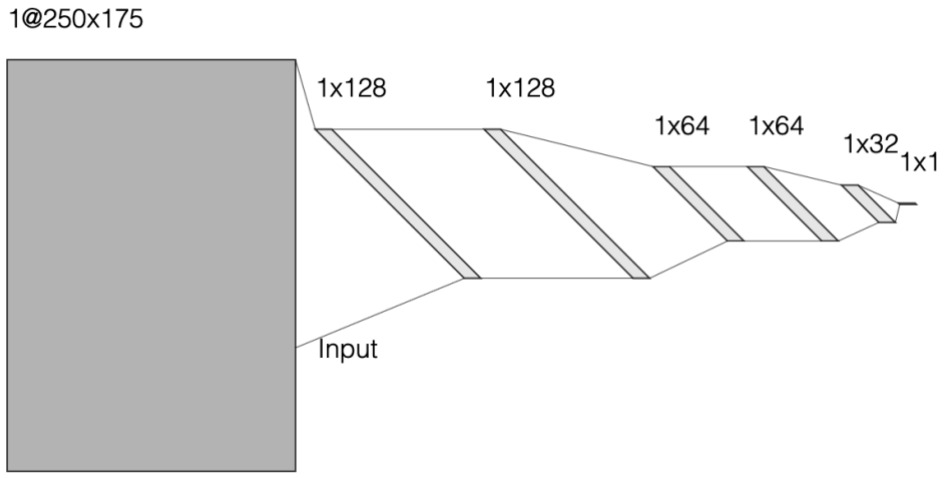


Figure #26: The image is flattened before feeding it into the first layer. The network also uses batch normalization layers to normalize the inputs to successive layers to control feedforward error propagation in the network.

We also made the following hyperparameter choices:

Loss Function: Cross Entropy

- This popular error function in deep learning imparts more weight to samples that are confidently misclassified (i.e. when the probability output by the network is severely wrong from the correct label).

Optimizer: Stochastic Gradient Descent

- I stuck to this choice because it is what we learned in class.
- The learning rate for this function is discussed below.

Activation Function: ReLU and Sigmoid

- ReLU is used for each layer in the network that is not the final layer.
- Sigmoid is used for the activation layer (last layer).

Since the datasets are imbalanced and neural networks perform better with balanced data, I weighted each class when feeding the training data into the neural network. In theory, this would offset the imbalance in the training data. Details can be found in the notebook file.

Learning Rate

The below plots show the training routines with several learning rates for the stochastic gradient descent optimizer used on the network.

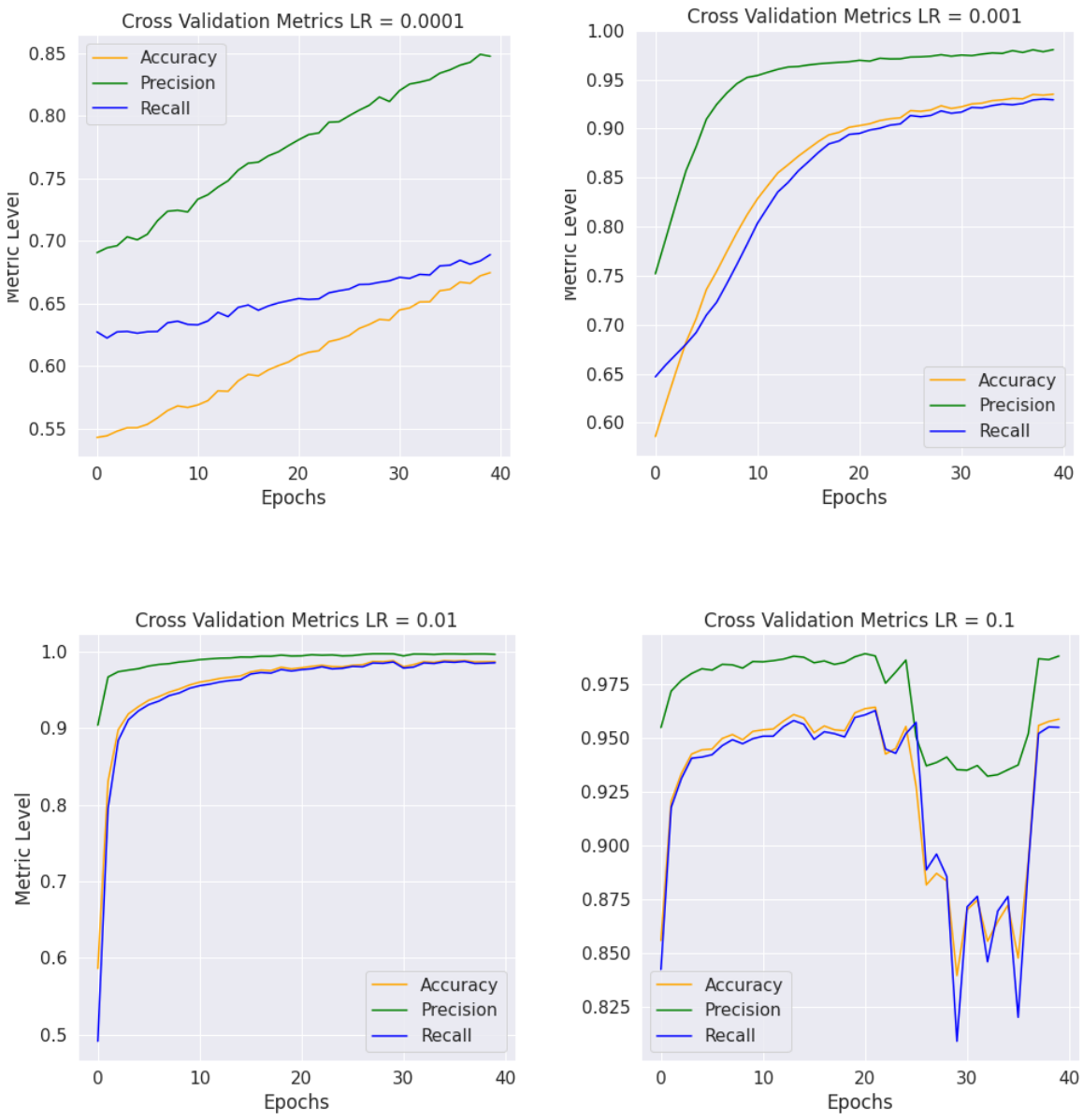


Figure 27-30: The learning rate was tested at 4 different values with a fixed batch size of 32.

From these plots, it appears that a learning rate of 0.01 would be optimal for this problem because it reaches the highest accuracy, precision, and recall on the training data. The other rates either lead to underfitting (in the case of 0.0001 and 0.001) or overfitting (in the case of 0.1). Also, this learning rate seems to level off after 20 epochs, so 20 epochs will be used for analyzing optimal batch size and for training.

Batch Size

The below plots show several batch sizes that were analyzed with the learning rate 0.01, which was found to work optimally in the analysis of different learning rate values.

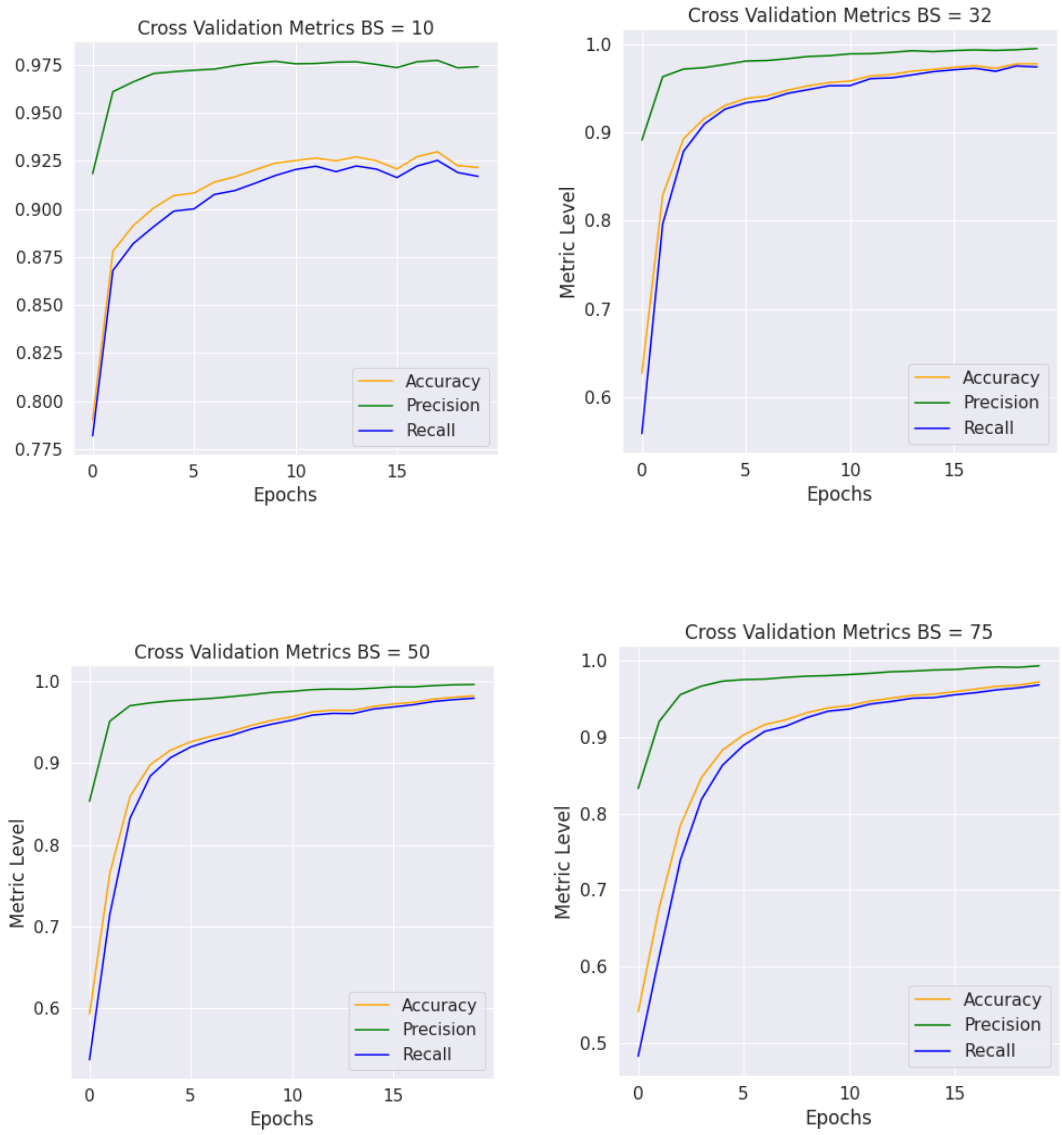


Figure #31-34: The batch sizes were varied with a fixed learning rate of 0.01. These batch sizes directly affect the frequency at which the optimizer updates the weights of the network.

The optimal batch size in this instance appears to be 50, as it results in the highest accuracy and recall at the end of 20 epochs.

The confusion matrices for the model generated by cross validation (left) and the testing dataset (right) are shown below:

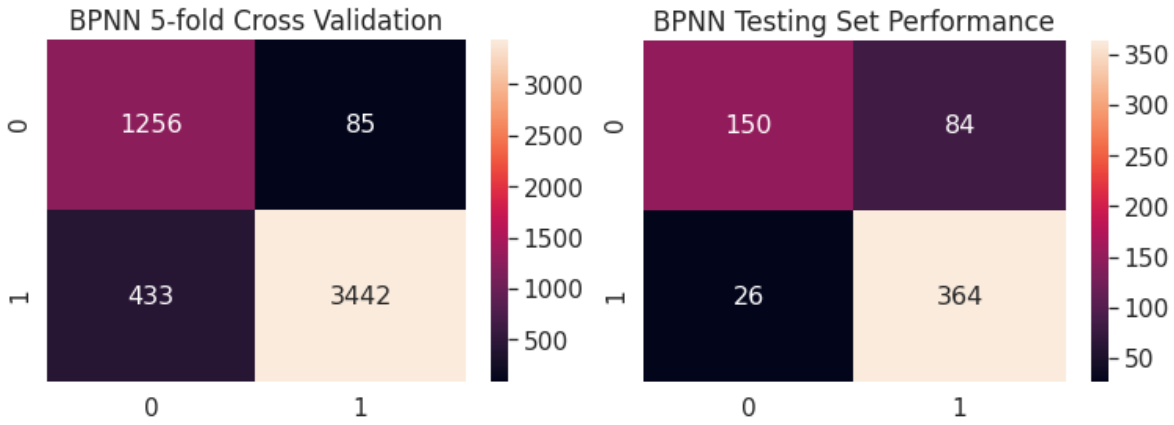


Figure 35, 36: This was a general trend we saw with the BPNN (and CNN): the performance in cross validation was very different from testing. From a cross validation point of view, BPNN performed very well and achieved a very high accuracy. However, in testing, that accuracy fell, mainly due to large numbers of false positives. This may be due to BPNN overfitting the data, so more sophisticated techniques would need to be introduced to avoid this phenomenon. Full accuracy results are available in Table 4.

Convolutional Neural Network (CNN)

CNN's are used in many modern image processing and classification tasks. We applied a CNN architecture by using TensorFlow's Keras library in Python. CNN's involve a lot of separate hyperparameters, so we had to construct experiments to find the optimal hyperparameter for this dataset. These experiments all consisted of cross-validation techniques, but we used the entire dataset - training, testing, and validation - instead of just the training set to measure these hyperparameter choices.

Architecture

The architecture of the CNN is based off of similar architectures found in the Kaggle notebooks that performed well on this dataset. I have included a visual depiction of the structure below (very small because the network is large):

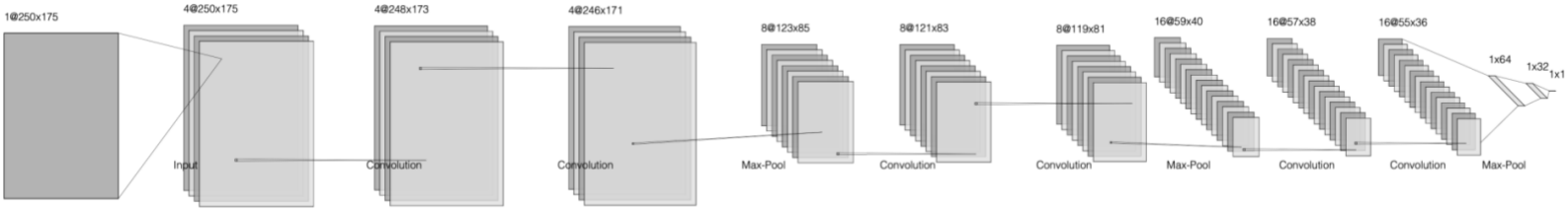


Figure 37: This network contains 6 convolutional layers and 2 fully-connected layers. It also utilizes batch normalization layers, dropout layers, and max pooling layers. See the CNN notebook for the full architecture.

We also made the following hyperparameter choices:

Loss Function: Mean-Squared Error
- In an ad-hoc analysis, this error function performed better than other options.

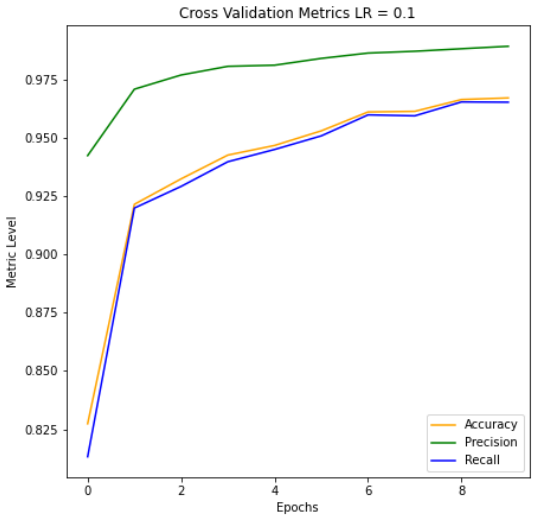
Optimizer: Stochastic Gradient Descent
- I stuck to this choice because it is what we learned in class.
- The learning rate for this function is discussed below.

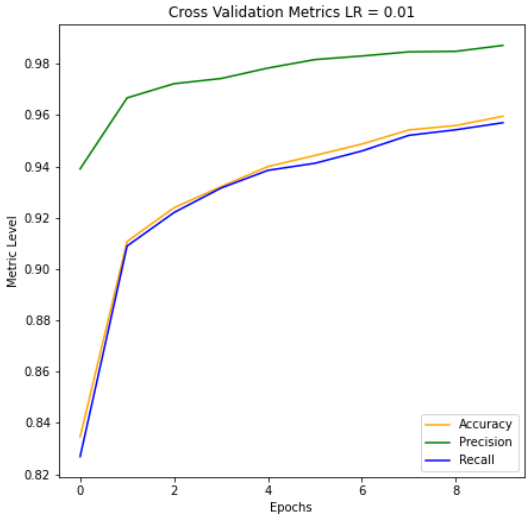
Activation Function: ReLU and Sigmoid
- ReLU is used for each layer in the network that is not the final layer.
- Sigmoid is used for the activation layer (last layer).

We also weighted the samples during training according to class size, as with the BPNN.

Learning Rate

Similar to the analysis of BPNN, I chose to analyze the learning rate and how it affected overall accuracy.





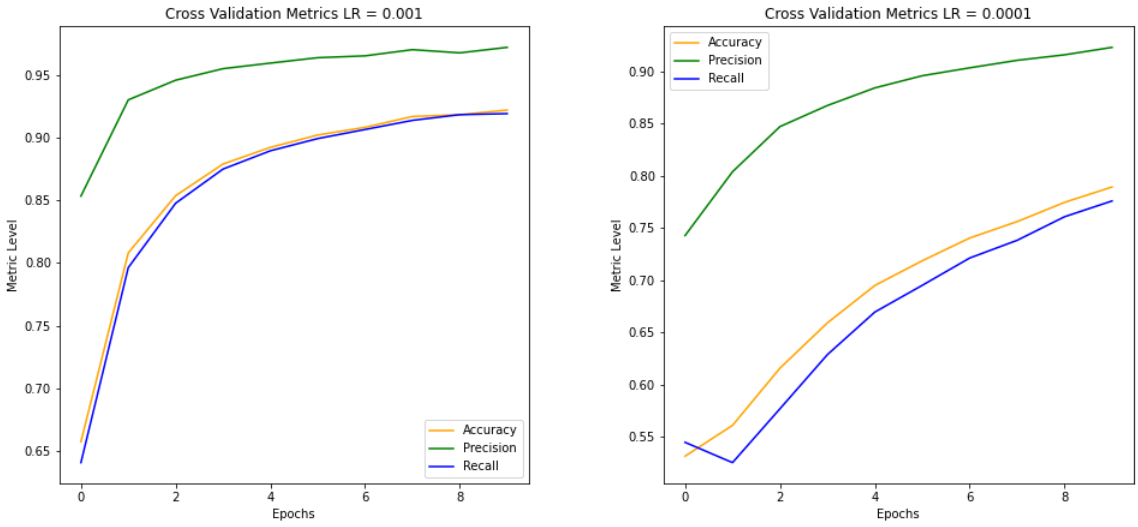
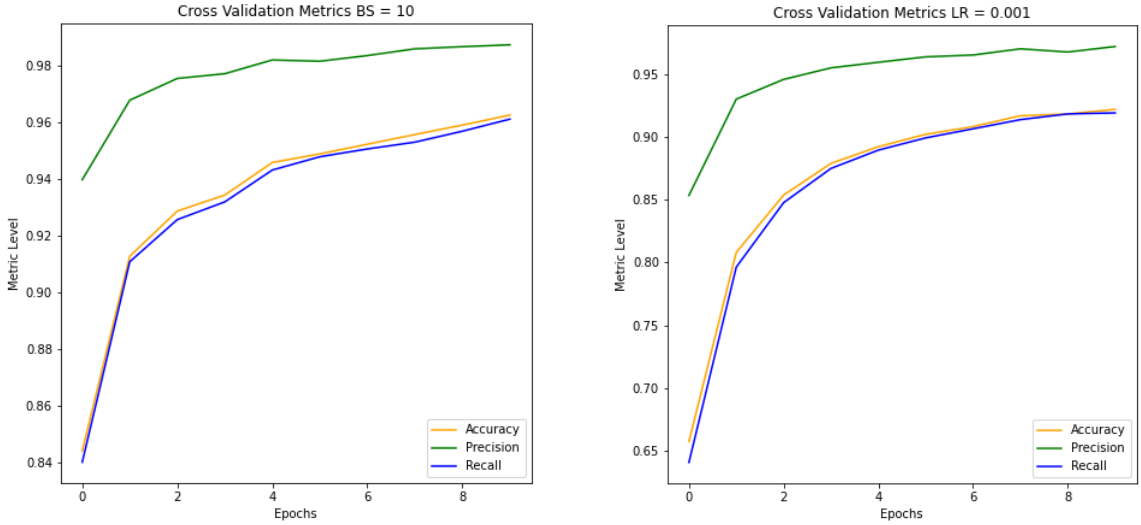


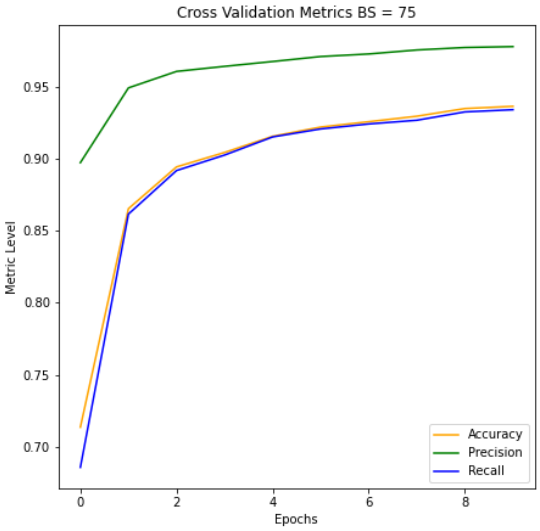
Figure 38-41: From these plots, it appears that a higher learning rate affects the accuracy of the model.

From this analysis, there are two viable options: 0.01 and 0.001 (or something in between). I chose to use 0.001 to prevent overfitting in the model. When testing the model in the future, this seemed to be a good choice because the CNN was very prone to overfitting the training dataset.

Batch Size

I then tested several different batch sizes with a learning rate of 0.001.



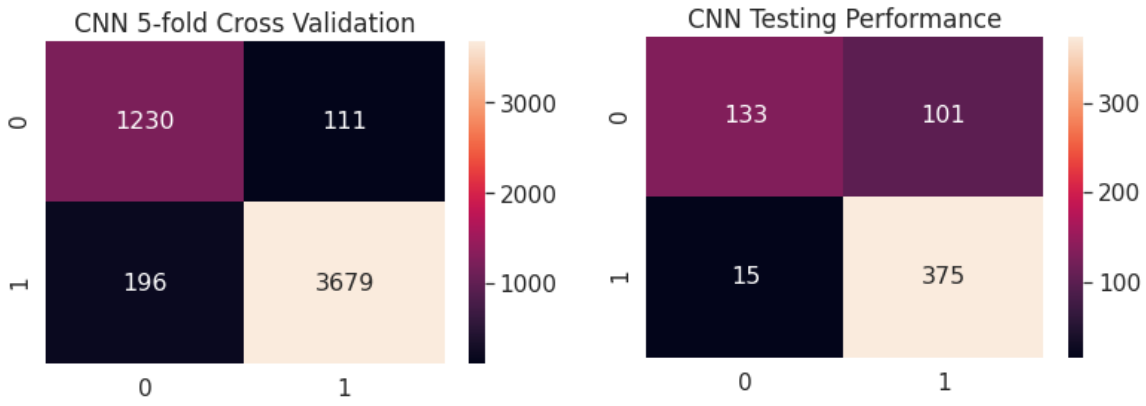


Figures 42-44: Note that Figure 43 has a batch size of 32; I reused the same plot from the previous part for consistency. From these experiments, we can see that there is not much difference between using a batch size of 32 and a batch size of 75. 10 is simply too small because we want to avoid overfitting the dataset.

Therefore, the batch size was chosen to be 32; this was partially based on other Kaggle notebooks that achieved good results.

Accuracy Results:

The results of the model are shown below on both cross validation of the training dataset and on the testing dataset. Note that for evaluating the testing dataset, the CNN was trained on only the training dataset.



Figures 45, 46: CNN, much like BPNN and the other models that we analyzed, performed very differently in cross validation than it did in testing. CNN was one of the most accurate models in

cross validation at 94.1%, but it ultimately resulted in a top accuracy of 81.4% on the testing dataset (when trained with the training split). This may be due to the CNN overfitting the data, one of the major issues with these types of neural networks.

Overall, the CNN architecture could be improved for this problem. The CNN is a very powerful model, and it can achieve very accurate results on this dataset. The network seems to be overfitting to the training set no matter what methods we used, so without some advanced optimization techniques, we could not fully harness the power of this popular tool.

Fusion

Adaboost

Each of the classifiers in Adaboost were decision trees of depth 1; these type of classifiers are known as “stumps” because they are short decision trees. This is a common method that is implemented when using Adaboost, and it was the default behavior in Sklearn.

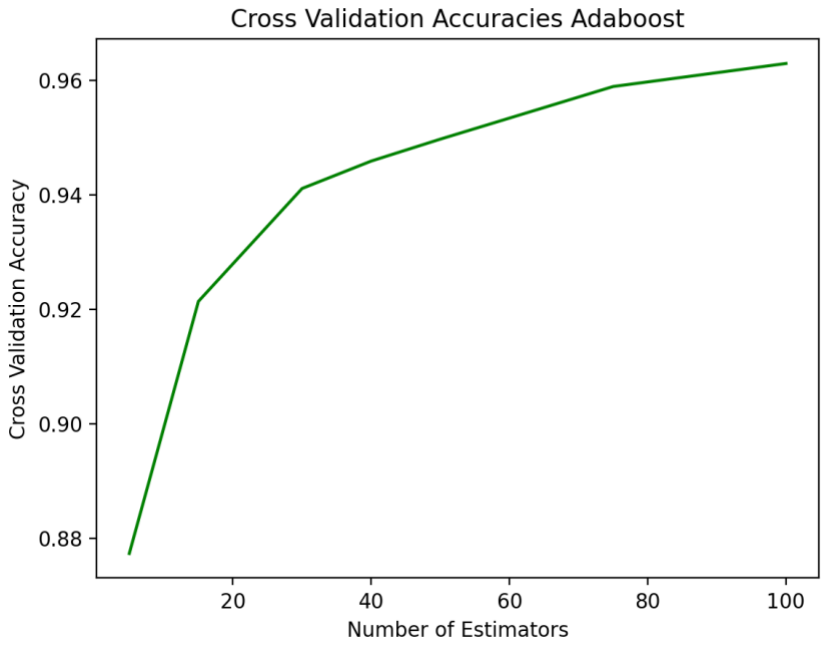


Figure 47: This table shows that accuracy increases as the number of estimators increases, but this also comes at a cost of runtime. Therefore, we want to choose a lower number of estimators for efficiency.

The training time for these methods was very long, and the runtime to train increased exponentially as the number of classifiers increased. Therefore, I decided to fix the number of classifiers at 30, which is where the elbow point on the above curve occurs. The cross-validation confusion matrix and confusion matrix for classification on the testing dataset is shown below:

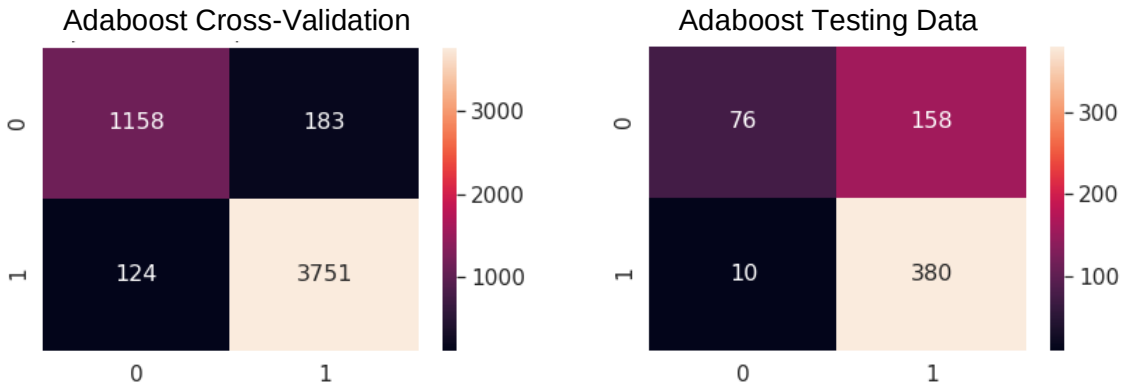


Figure 48, 49: Adaboost actually performed very well in the cross validation, but it suffered from poor specificity in classifying the testing data. This suggests that the testing dataset and the training dataset have very different distributions, so more generalizable methods are desirable for this problem.

Other Fusion Methods

In addition to Adaboost, Random Forest is another fusion method that was used in our analysis of this dataset.

Evaluation

Table 4
The following table compiles the model results and compares each of them:

Model	Testing Accuracy	Test Spec.	Test Sens.	CV Accuracy	CV Spec.	CV Sens.	Runtime
MPP Case 1	80.9%	76.9%	83.3%	85.0%	83.3%	87.1%	1.82 sec
MPP Case 2	80.9%	76.9%	83.3%	85.0%	83.3%	87.1%	2.06 sec
MPP Case 3	75.0%	95.3%	62.8%	68.9%	62.8%	97.6%	4.47 sec
kNN (k = 15)	67.1%	100%	12.4%	-	-	-	46.3 hrs
kNN (k = 72)	65.0%	100%	6.8%	-	-	-	41.8 hrs
Agglomerative Clustering (50)	89.0%	72.3%	94.7%	-	-	-	10.2 min
Kmeans (50)*	88.2%	76.4%	92.9%	-	-	-	17.6 min
Decision Tree*	74.7%	39.7%	95.9%	89.5%	78.4%	93.4%	4.46 min

Random Forest	76.4%	39.1%	99.2%	96.1%	88.8%	98.6%	1.23 min
XGBoost	73.2%	30.4%	98.7%	96.4%	89.8%	98.7%	10.5 min
SVM (RBF)	77.7%	43.5%	98.5%	97.3%	94.4%	98.3%	4.69 min
Adaboost	73.1%	32.48%	97.44%	94.1%	86.35%	96.80%	8.68 min
BPNN	82.37%	64.10%	93.33%	90.01%	93.66%	88.82%	15.5 sec
CNN	81.41%	56.84%	96.15%	94.11%	91.72%	94.94%	4.42 min**

* Clustering algorithms were used to classify the training data, not testing.
** CNN ran on GPU accessed with Google Colab

Performance Comparison

Since this dataset was posted on Kaggle, we were able to compare our models with the best accuracies achieved in the notebooks on Kaggle. This challenge did not feature a leaderboard as typical challenges do, but we did find some notebooks with very accurate models:

1. Username: madz2000 [2]

Metric	**Score**
Accuracy	0.9260
Sensitivity	0.9564
Specificity	0.8760

This user's main strategy was using data augmentation to generate a dataset that had a similar distribution to the original training dataset. Through use of Tensorflow's ImageGenerator class, he was able to generate new images that took random images in the original training dataset and performed small perturbations such as horizontally flipping the image, zooming in, or rotating the image.

2. Username: paultimothymooney [3]

Metric	**Score**
Accuracy	0.8910
Sensitivity	0.9829
Specificity	0.6111

This user, one of Kaggle's staff members, was the one who uploaded this dataset. He used pre-constructed transfer learning architectures with CNN's to achieve this result. Transfer learning would be a good approach for this application because the testing dataset seems to be so different from the training dataset. In order to make the model more generalizable, sophisticated methods such as transfer learning would be necessary when building the CNN.

Sources

[1] P. Mooney, *Chest X-Ray Images (Pneumonia)*, version 2, Kaggle.com, 2017. [Online]. Available: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia. [Accessed: Dec. 08, 2020].

[2] Madz2000, 'Pneumonia Detection using CNN (92.6% Accuracy)' *Kaggle.com*, 2020. [Online]. Available: https://www.kaggle.com/madz2000/pneumonia-detection-using-cnn-92-6-accuracy. [Accessed Dec. 08, 2020].

[3] P. Mooney (paultimothymooney), 'Detecting Pneumonia in X-Ray Images' *Kaggle.com*, 2017. [Online]. Available: https://www.kaggle.com/paultimothymooney/detecting-pneumonia-in-x-ray-images. [Accessed Dec. 08, 2020].