Face Mask Detection (COSC 522)

Andrei Cozma [1] **Sabrullah Deniz** [1] **Hunter Price** [1] **Isaac Sikkema** [1] **Nan Tang** [1]

Abstract

Being able to recognize whether an individual is wearing a mask or not wearing a mask has become an increasingly popular topic, especially in the past year since the COVID-19 pandemic. There are a multitude of applications where such a system could be employed such as medical environments, hospitals, surgery rooms, health clinics, among other commercial use-cases. Feeding images through a custom-designed convolutional network architecture allowed us to employ various classification techniques in order to predict whether an individual is wearing a mask or not wearing a mask to an outstanding level of accuracy. In this paper we describe the implementation and evaluate various models used for the prediction of the convoluted images, from Back-Propagation Neural Network, Support Vector Machine, Decision Tree, as well as kNN and Bayesian Methods.

1. Introduction

1.1. Background

As part of our Computer Science 522: Machine Learning final project, the group decided to perform image classification on a dataset consisting of 992 training images, 10,000 testing images, and 800 validation images. These were all images of faces with or without a mask, respectively labeled as 1 or 0. The dataset was taken from a Kaggle competition titled "Face Mask Detection 12K Images Dataset" (Jangra, 2020).

1.2. Motivation

With the advent of COVID-19, the benefits of wearing a mask to prevent the spread of an airborne disease are clear, especially in high-risk areas like hospitals, health clinics, or nursing homes. Automatic face mask detection can help in the effort to lower and prevent the spread of pathogens in these areas. For example, in a secured area where a mask is absolutely required, administration could be alerted when an individual is detected not wearing a mask. Further, such a model can be employed in systems where entry in places like surgery rooms or treatment areas can be accepted or denied in order to ensure that all authorized personnel is wearing the appropriate personal protective equipment (PPE).

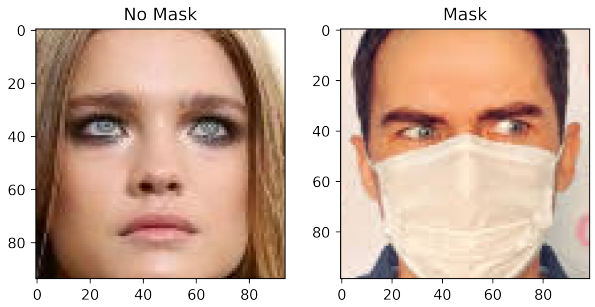

Figure 1. No Masked Image vs Masked Image

1.3. Challenge

Using the above dataset, we challenged ourselves to develop a custom feature extraction architecture based on image convolutions. With this approach we set out to find a design which allows the processing of the original images into effective feature sets which can be used as input for a multitude of classification methods such as Back-Propagation Neural Network, Support Vector Machine, Decision Tree, as well as kNN and other Bayesian Methods. Finally, these models should be able to classify these images to a high degree of accuracy without human intervention. With this algorithm, one would be able to address if not completely solve the problem motivation described in the section above.

1.4. Summary of proposed approach or contribution

1.5. Task assignment

Andrei Cozma: Structured the dataset read-in and preprocessing functionality in an object-oriented manner in

[*]Equal contribution [1]Department of Computation, University of Tennessee, Knoxville, Tennessee, USA. Correspondence to: Anon Anon <anon>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

order to facilitate usage and implementations of our work in the Jupyter notebook. Worked on designing and engineering the kernel filter matrices for the convolution and pooling layers with Hunter in order to efficiently extract the best features for improved accuracy and run-time. Implemented and trained the Back Propagation Neural Network (BPNN) models and tuned the hyperparameters with a custom method similar to the Hyperband tuning approach. Finally, generated graphs for the effects of different hyperparameters on accuracy of the BPNN, as well as visualization of images throughout each layer of the convolutions, and overall evaluation graphs.

Sabrullah Deniz: For nonparametric learning, k-nearest neighbors (kNN) was implemented to face-mask dataset. Different k values were also applied to find the right k value that is not overfitting or underfitting the data. Confusion matrix for testing and validation test data were compared.

Hunter Price: Defined the programmatic structure of the Convolution Network structure and implemented all functional layers of the network. Worked on the engineering of the preprocessing feature extraction of the network with Andrei. Designed the initial image preprocessing that was later refactored including: image resizing, gray-scaling, and min-max scaling. Implemented the support vector machine with the Scikit learn library. Implemented Behavioral Knowledge Space (BKS) model fusion. Implemented the Kfold cross-validation function. Implemented evaluation code to calculate class-wide accuracy, overall accuracy, and a confusion matrix. Implemented code to plot confusion matrices.

Isaac Sikkema: Implemented the code for Maximum Posterior Probability (MPP) classifiers as well as kMeans and WTA clustering classifiers. Implemented a Naive Bayes (NB) fusion solution for an arbitrary number of classifiers. Implemented the code to plot a 3D confusion hypermatrix.

Nan Tang: Working on decision tree classifier and PCA dimensionality reduction method. At the First stage, use a self-built decision tree model to train the convoluted data and apply PCA to deduct dimensionality (remain 100 features), which achieves 0.51 accuracy. PCA is also applied to generate eigen faces. Later use a sklearn model to train the data, which gains 0.87 overall accuracy.

2. Related Work

Most previous implementations submitted on Kaggle for the dataset use Haar Cascade to detect faces as the first step(Chaitanya, 2021), and employ different architectures of DCNNs here used for mask detection as the second step. This first step is with the assumption that some of the images included multiple instances of people with or without masks in the same image. However, it appears that the actual dataset had been revised to already crop the pictures on individual steps. Therefore, we will not need to perform this initial step.

While several submissions include their own implementations of Convolutional Neural Network(Holla, 2021), most of the submission implementations employ different architectures of DCNNs here used for mask detection as the fine-tuning step through transfer learning. Some of these architectures of pre-trained models used in previous submissions include the VGG19, Xception, ConvNet, InceptionV3, MobileNet, MobileNetV2, and DenseNet201 models. Then, they employ transfer learning to fine-tune them for specifically detecting the existence of masks in this dataset.

The advantage for these pre-trained models is that they were trained on very large datasets, such as ImageNet. Because of this they are able to take advantage of millions of data already available. On the other hand, one major downside of these approaches is that, while they are able to achieve impressive accuracies, the models are very heavy and complex, often hundreds of megabytes in size. As described in the section about the main challenges, this can be a potential downside which could limit their potential use-cases in various applications, such as running directly on embedded systems. Certain systems may not have may not have the computational capacity to run these models, especially in real time.

3. Technical Approaches

3.1. Convolutional Network

In this project we implemented a feature extraction and dimensionality reduction technique called a convolutional network. Our convolutional network consists of multiple types of layers. An input image is passed through the network and in each layer the image is transformed and outputted. The layers implemented in this project are:

- Convolution Layer
- Pooling Layer
- Activation Layer
- Flatten Layer

3.1.1. Convolution Layer

A Convolution Layer consists of applying a filter or kernel to an image of m by n pixels. The filter is applied to every segment of the image. The segments can be created by creating a window of the same size as the filter on the image. We start at the top-left of the image, apply the filter, then shift the window to the left by the specified stride length and continue. When we reach the right side of the image, we shift the window back to the left and move the filter down

by a the specified stride length. To apply a filter to a portion of an image you perform an element wise multiplication between the filter and the window, then sum the resulting matrix to get the new pixel value. The technical implementation was adapted from a medium article by Samrat Sahoot (Sahoo, 2020).

An example of this is given some image I of the letter T shown in figure 2, a vertical line filter F shown in figure 3, and a stride length of 1.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 2. Image I

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

Figure 3. Vertical Line Filter F

We can begin the convolution by applying the filter to the first top-left segment of the image shown in Figure 4.

$$sum(\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \odot \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}) = 0$$

Figure 4. Applying The Filter To The First Window

To continue the convolutional process, we shift the window to the right by one index the apply the filter once more shown in Figure 5.

This process continues until you have applied the filter to all possible windows of the image. This will give us a resulting image of the shape $n - 1$ by $m - 1$ pixels. The image now has been filtered for vertical lines. This can be seen in Figure 6. There is no longer the horizontal line of the top of the capital T.

This layer gives us incredible flexibility as we can now filter images for specific features. Filters or Kernels can be experimented with to get the desired result.

3.1.2. Pooling Layer

A Pooling Layers serves multiple purposes in our implementation. First, it serves as a dimensionality reduction technique. A single pooling layer significantly reduces the number of features in the image. Second, we use it to keep our significant features while ignoring the irrelevant ones; this is discussed in further detail in the Feature Engineering section. Lastly, it generalizes the image by down sampling. Images are never the exact same and contain a large number

$$sum(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \odot \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}) = 1$$

Figure 5. Applying The Filter To The Next Window

$$\begin{bmatrix} 0 & 1 & -1 & 0 \\ 0 & 2 & -2 & 0 \\ 0 & 2 & -2 & 0 \end{bmatrix}$$

Figure 6. Resulting Image

of patterns that differ image to image, and therefore, we need to have a way to generalize the images. The technical implementation was adapted from a Stack Overflow post (user10030086 & K., 2019).

Pooling works by applying a non-overlapping window to an image of the size $pool\_size$ by $pool\_size$. This value is given by the user. Similar to convolution, in each window we apply a pool function to the segment. There are 3 main pool functions: maximum, minimum, and average. The maximum pool takes the largest pixel value of the segment as the next pixel, the minimum pool takes the smallest pixel value of the segment as the next pixel, and the average pool takes the average pixel value of the segment as the next pixel.

3.1.3. Activation Layer

An Activation Layer simply applies an activation function to the entirety of an image. The specific activation function we used in this project is the Rectified Linear Unit (ReLU) show in Figure 7. This function takes the max of the inputted value and 0. It is plotted in Figure 8. This function is very useful when you need to filter out negative values from an image.

$$f(u) = max(0, u)$$

Figure 7. ReLU Activation Function

3.1.4. Flatten Layer

A Flatten Layer changes the shape of the image. The layer simple flattens the 2 or 3 dimensional image to a singular array. This layer is used as the final layer of the network. The output of this layer is used as input to the classification models detailed later in later sections.

3.2. Hyperband Tuning

A custom hyperparameter tuning method was employed, based vaguely on the Hyperband tuning approach (Li et al., 2018). This method was used for tuning the hyperparameters of the Back Propagation Neural Network (BPNN). The goal is to more efficiently converge onto an optimal set of hyperparameters as compared to other approaches such as

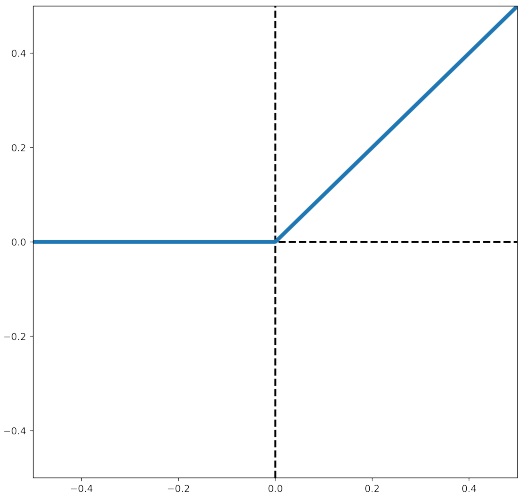

Figure 8. ReLU Activation Function

Grid Search, Random Search, or Bayesian Optimization methods.

The process starts with an initial set of hyperparameters which can be chosen either randomly or through limited manual experimental tuning. Additionally, the tuner is provided a maximum number of epochs to train and a value (Eta) that defines the level of hyperparameter generation to be performed as well as the steps for the training epochs for each stage of the tuning process. For the tuning of the BPNN, we used an Eta parameter with a value of 3 and maximum number of training epochs of 60.

Given these values, the tuner trains the model with different parameters in multiple stages. Each step of the iterative process of the tuner begins by generating a set of hyperparameters based on the initial values, or the best set of values found from the previous iteration. The tuner then trains the model on all members of the generated set of hyperparameters and determines which hyperparameter set yielded the best results. These are to be used in the next iteration of the process.

The tuner then uses these best values to generate another set of hyperparameters and repeats the same process to find the best set for a total of 9 training epochs. Each stage increases the number of training epochs by a factor of Eta. For example, suppose a value of 3 was chosen for Eta and a maximum number of epochs of 60. In that case, the first stage trains for 3 epochs, the second stage trains for 9 epochs, the third for 27, and the fourth for 54, where it reaches the stopping point and the overall best hyperparameters as well as the best trained model are returned.

The hyperparameter generation works as follows. For each individual parameter, an 3 additional values bigger than the initial and 3 additional values smaller than the initial are added to the set of hyperparameters to train on. For example, suppose an initial batch size of 20 is chosen as one of the hyperparameters. The generated set of hyperparameters will include all combinations possible with batch size of 20, the batch sizes bigger than 20 being 26 and 33, as well as the batch sizes smaller than 20 being 13 and 6. These numbers are based on the Eta parameter dictated by the following equations on a range from i to Eta:

$$hyper := hyper - hyper * (i/Eta)$$
$$hyper := hyper + hyper * (i/Eta)$$

Figure 9. Equations for Hyperparameter Generation

4. Experiments and Results

4.1. Standardization

The training, validation, and testing data all required significant pre-processing. The images in the dataset were given in a 3-channel red-green-blue (RGB) format and differed in both height and width. The first standardization technique applied to the data was the resizing of the images. Each image was resized to 128 by 128 pixels using a high-quality Lanczos filter provided by the Pillow library. Once the images were resized they were grayscaled. This process involves transforming the 3-channel images to a single channel where each pixel value represents how light or dark a pixel is displayed. To perform this transformation we used the Pillow library which contains a grayscale function. Finally, we need to normalize the values in the images to improve the performance of the classification algorithms later down the pipeline. In image pre-processing it is standard to min-max scale the images rather than standardize the values. This process changes the values that range from 0 to 255 to a scale of 0 to 1. Figure 10 displays this transformation.

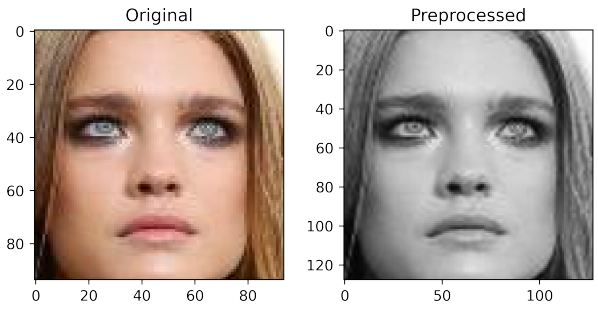

Figure 10. Original Image vs Preprocessed Image

4.2. Dimensionality Reduction

The only explicit dimensionality reduction algorithm we implemented was a Principle Component Analysis (PCA). It was used in our decision tree experiment. However, we ultimately decided to rely only on the reduction effects of our feature engineering work.

4.3. Feature Engineering

In this work, a convolutional network was created to extract specific features from the dataset. Feature extraction serves as an additional pre-processing step that reduces the dimensionality of the dataset while only keeping the specific features that are desired. Various designs for the extraction of features through the convolutional network were implemented and tested. The choices made in the kernel matrices for the convolution layers, the parameters for the pooling layers, as well as the choice of activation layers play a vital role in the effective extraction of features for maximized performance and training time of the classification models. The best architecture we engineered is designed is described in this section, which allowed for an average of 10-15% accuracy improvement for all models on average as compared to some of our previous designs, and training time was reduced by a factor of 20. The first step was to decide on the importance of features present in the sample images, and we decided were important in our images were the nose and mouth. This is due to the fact that these specific features will be missing in images where the person is wearing a mask. To single out these specific features we engineered a convolutional network consisting of 6 layers. The first of these layers (Layer 1) was a convolution layer using the horizontal filter shown in Figure 11.

$$\begin{bmatrix} -1 & -1 & -1 \\ 1.15 & 1.15 & 1.15 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 11. Horizontal Line Filter

This filter extracts the horizontal lines that occur in an image. We felt this was important as the nose and mouth typically have horizontal features which we can be accentuated by the filter. Additionally masks, will create a horizontal line across the face which will be drawn out by the filter. The size of the resulting image is 126 by 126 pixels. Figure 12 shows this filter applied to the example image.

The second layer (Layer 2) was a max pooling layer. This layer serves two purposes. The first is dimensionality reduction. This reduces the size of the images to 63 by 63 pixels. The second purpose is to keep the key features which we have highlighted with the first layer. The result of this layer is shown in Figure 13.

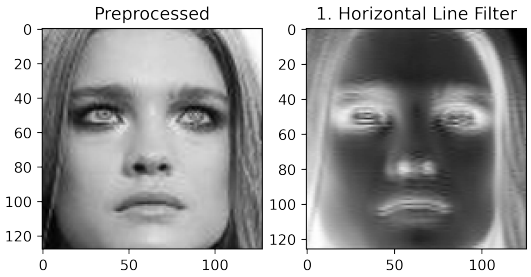

Figure 12. Preprocessed Image vs Layer 1

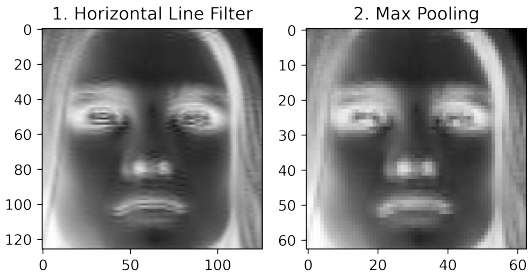

Figure 13. Layer 1 vs Layer 2

The third layer (Layer 3) was another convolution layer using the outline filter shown in Figure 14. This filter aims to convert unimportant features to a negative value which will then be converted to 0 in the following layer. This is important as the previous layers have made the eyes, nose, mouth, and mask features high values and the other features lower values. When the filter is applied the difference between these features is made greater. The output of this layer is show in Figure 15. The resulting image is of the shape 61 by 61 pixels.

The fourth layer (Layer 4) was a ReLU activation layer. This layer will convert all negative values to 0 and keep all the positive values the same. This will effectively eliminate all of the unimportant features, now negative numbers, from the image. The output of this layer is shown in Figure 16. Notice how the majority of the image is now completely black except for the features we wanted to keep. The resulting image is 61 by 61 pixels

The fifth layer (Layer 5) was another max pooling layer. This layer further reduces the dimensions of the image to 30 by 30 pixels whilst keeping our most important features. The resulting output of the layer is shown in Figure 17.

The final layer (Layer 6) is a flatten layer. This layer flattens the image into a singular array with a length of 900. This array is the final list of features that we use in our

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 14. Outline Filter

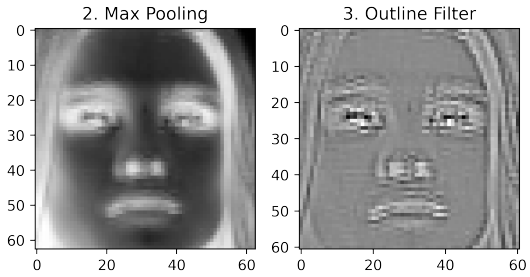

Figure 15. Layer 2 vs Layer 3

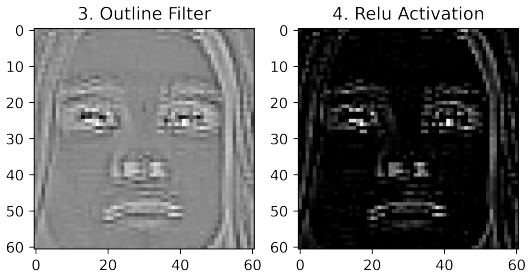

Figure 16. Layer 3 vs Layer 4

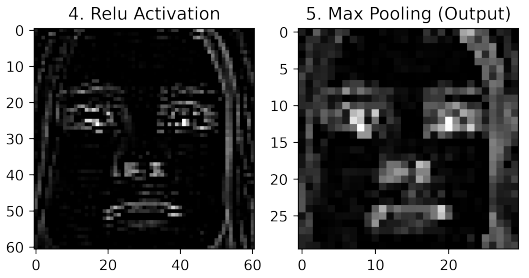

Figure 17. Layer 4 vs Layer 5

classification models.

The ultimate goal of our convolutional network is to have the major eyes, nose, mouth, and mask features apparent as positive numbers while the rest of the array is filled with 0's. Figures 18 and 19 show this to be the case.

4.4. Model Evaluation

4.4.1. MPP

Maximum Posterior Probability (MPP) methods were implemented in the standard manner. Case 1 got an overall accuracy of 86.90%. Case 2 got 90.83%, and case 3 got 84.38%. We were surprised to see the accuracy with only case 1 was so high. We believe this is due to the initial feature engineering work done on the dataset. The confusion matrices can be seen in figures 20, 21, and 22. It may be noted that the accuracy of case 3 is lower than the others. This is simply because the training dataset is not a perfect representation of the testing dataset, and the case 3 model has overfitted for the training set.

4.4.2. KNN

Different k values were implemented and figure 23 shows the plots of classification accuracies for different k values (ranging from 1 to 50). The highest classification rate on the dataset was achieved by k = 4 at 88.36% overall accuracy. We also tested higher k values and the accuracy result dropped sharply which caused to underfitting problem. From the plot, we can see that k = 4 has overall good accuracy and the higher we choose it may cause underfitting.

In Figure 24, we also created confusion matrix of validation and test dataset by k = 4. The overall accuracy is higher in our test set than validation set.

4.4.3. DECISION TREE

At the first stage, we use a self-written decision tree code and use PCA to keep 100 dimensions, but the accuracy is only 0.5. This low accuracy may be due to the fewer features we keep. Later, we used Sklearn to run the decision tree. Fixing the random state equals to 30, we change maximum depth from 0 to 30, and find the maximum accuracy of 0.87.

4.4.4. CLUSTERING

Both kMeans and WTA clustering approaches were tested. However, neither gave any results significantly better than random guessing and so they are omitted here.

4.4.5. SUPPORT VECTOR MACHINE

The Support Vector Machine was implemented with the Sklearn library. We used k-fold cross validation to determine the best hyperparameters. Ultimately, we found that the best hyperparameters were the Radial Basis Function (rbf) kernel, a gamma of 0.001, and a C of 100. With these parameters we were able to get a validation accuracy of 91.75% and a test accuracy of 93.25%. The confusion matrix on the test dataset is shown in Figure 25.

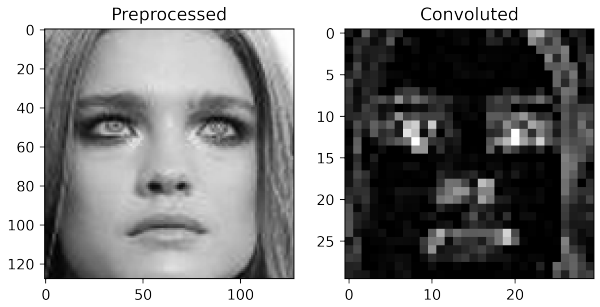

Figure 18. Preprocessed Image vs Convoluted Image - No Mask

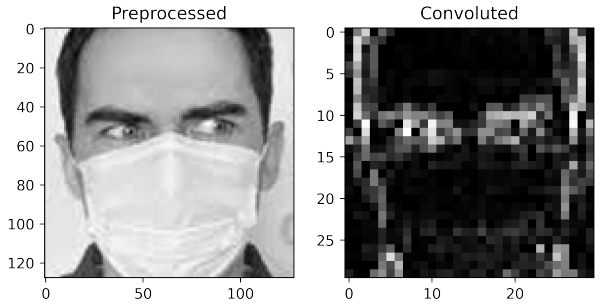

Figure 19. Preprocessed Image vs Convoluted Image - Mask

4.4.6. BPNN

The Back-Propagation Neural Network (BPNN) approach was implemented based on the descriptions and examples shown by Michael A. Nielsen in his book "Neural Networks and Deep Learning" (Nielsen, 2015).

The neural network implementation allows for a configurable number of hidden layers, random normal and random uniform initialization options for weights and biases, as well as configurable batch size, learning rate, and training epochs. Additionally, the neural network allows for training on the GPU for much improved training times. This is achieved through the use of the CuPy Python library which implements Numpy array manipulation while leveraging the Nvidia CUDA library for GPU acceleration (Okuta et al., 2017). The GPU used for the experimentation with BPNN outlined below was an Nvidia RTX3070.

The implemented model was trained and evaluated on various network configurations, batch sizes, and learning rates. The first step of the process was to individually assess the effect of each hyperparameter to the overall validation accuracy of the model. Higher number of training epochs were used at first with each of the experimentation stages described below. The optimal maximum number of training epochs needed for convergence based on our tests was from 30 to 60. For the graphs shown in our discussion below,

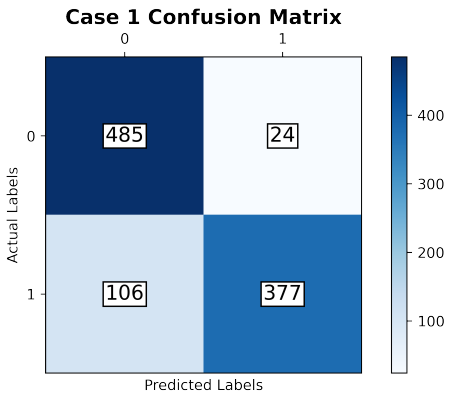

Figure 20. MPP Case 1 Confusion Matrix For Test Dataset

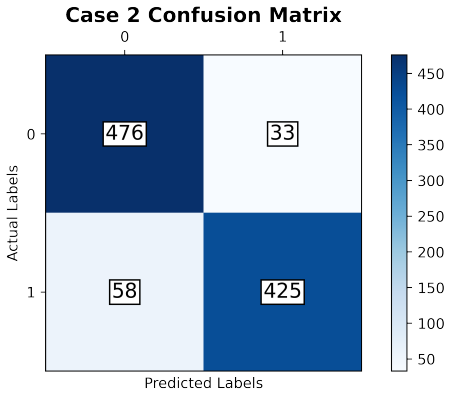

Figure 21. MPP Case 2 Confusion Matrix For Test Dataset

below we trained the models up to 50 epochs.

For network configuration, various numbers of hidden layers were assessed and with our convolutional network design the best performing number of hidden layers based on training time and accuracy was one. The effect of differing number of neurons in the hidden layer is shown in Figure 26. Results show that the optimal number of hidden layer neurons is around 190.

Various mini-batch sizes were evaluated on a range from 5 to 40, and the results are outline in Figure 27. The optimal value for this set of mini-batch sizes was 5.

Lastly, learning rates were chosen on a range between 0.001 and 2. The results shown in Figure 28 show a tie between a learning rate of 2 and learning rate of 0.5.

After the initial look into the effects of the various hyperparameters, we utilized the custom implementation of the Hyperband tuning method described earlier in order to converge onto a more stable set of hyperparameters. The best

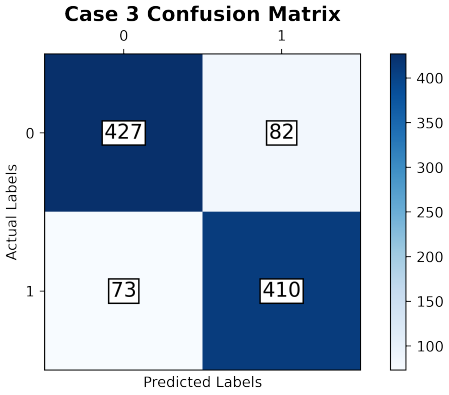

Figure 22. MPP Case 3 Confusion Matrix For Test Dataset

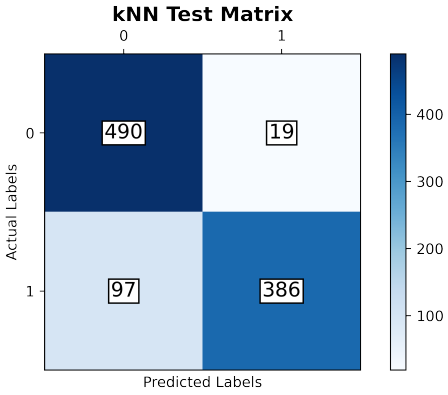

Figure 24. kNN Confusion Matrix for Test Dataset

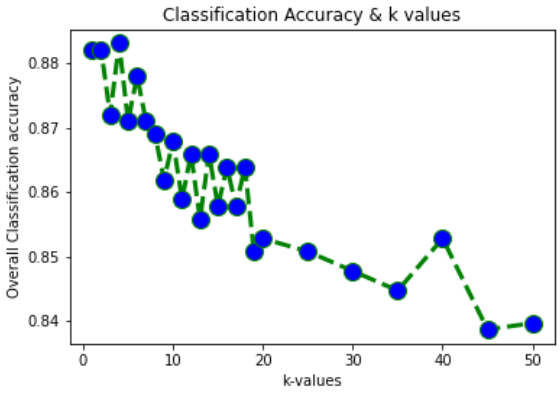

Figure 23. kNN Accuracy vs. K

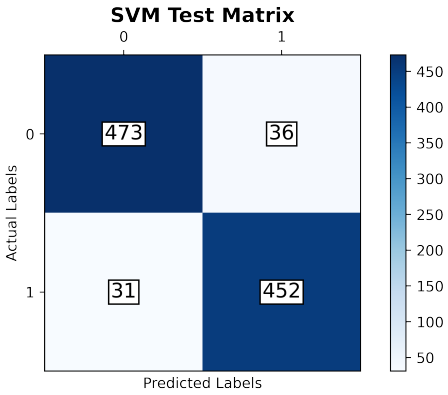

Figure 25. SVM Confusion Matrix on Test Dataset

results found earlier were used for the initial hyperparameters input into the tuner. This tuning method proved to be efficient and effective, as the tuning process finds the best set of hyperparameters at each stage starting from a low number of training epochs. In this case, the Eta parameter value chosen was 3. As the number of training epochs increases, the tuner begins to converge onto better and better sets of hyperparameters at each stage.

The results of the Hyperband tuning method converged onto the following set of hyperparameters. For the number of nodes in the hidden layer, the optimal value was 253. For the best batch size, the optimal number came down to 1. Finally, the optimal learning rate was found to be 0.296296.

With these hyperparameters, the trained model was able to achieve a validation accuracy of 0.95625 (95.62%), and a slightly lower testing accuracy of 0.94254 (94.25%). The confusion matrices for testing and validation are shown in Figures 29 and 30. Interestingly, the class-wise validation results show that the model was able to achieve an impressive 97% accuracy for Class 1 (wearing a mask) also depicted in Figure 29. However, the classwise testing results were much closer to eachother, with Class 0 (not wearing a mask) achieving an accuracy of 94.11% and Class 1 (wearing a mask) an accuracy of 94.41%. Finally, a comparison between the testing and validation results showing overall as well as class-wise accuracies are shown in Figure 31.

4.5. Classifier Fusion

4.5.1. Naive Bayes

We implemented Naive Bayes (NB) fusion in a manner which allowed us to fuse the classification results of an arbitrary number of classifiers. Instead of outputting another square matrix, for the fusion of two confusion matrices, we outputted a 3D hypermatrix, a model of which can be seen in figure 32. For the fusion of n confusion matrices, we output an $(n+1)$D hypermatrix. To find the calculated classification, we index into the hypermatrix at each classifiers prediction. This returns a list of probabilities, and we

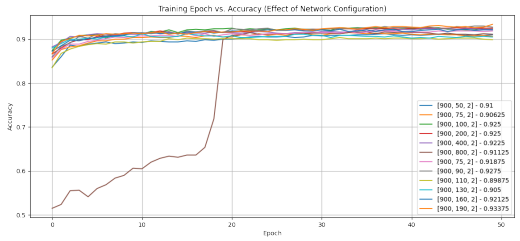

Figure 26. BPNN Effect of Network Configuration

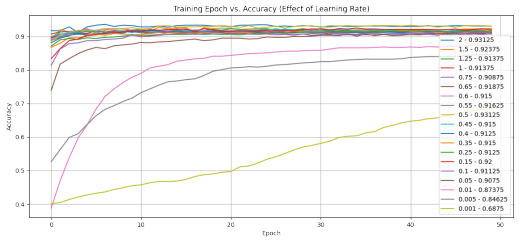

Figure 28. BPNN Effect of Learning Rate

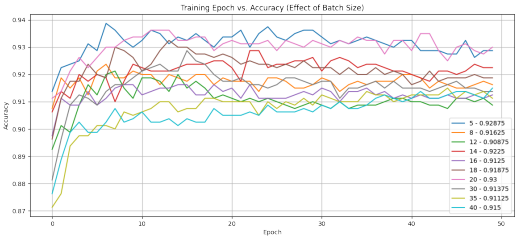

Figure 27. BPNN Effect of Mini-Batch Size

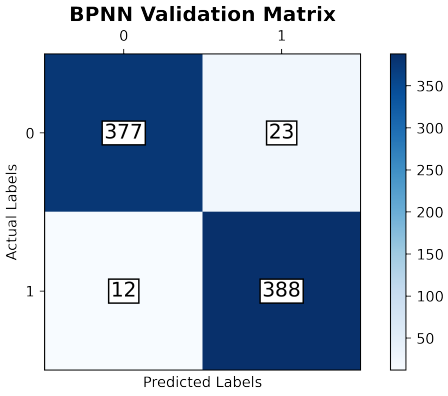

Figure 29. BPNN Confusion Matrix Validation

select the label with the highest probability. For instance, if classifier A predicts 1 and classifier B predicts 0, our fused classification is argmax(hypermatrix[1][0]). When we merged all cases of MPP we achieved an accuracy of 88.51%. The resulting confusion matrix is shown in Figure 33. When we combined the MPP cases, the SVM, and the BPNN we got an accuracy of 94.05%. The confusion matrix is shown in Figure 34. Finally, when we combined only the SVM and the BPNN we achieved our maximum accuracy of 94.25% on the test set. The confusion matrix is show in Figure 35.

4.5.2. Behavior Knowledge-Space

Behavioral Knowledge-Space (BKS) performed excellent when combining models with sub-optimal accuracy's. In each instance the model fusion from BKS performed the same or better than Naive Bayes. When we merged all cases of MPP we achieved an accuracy of 90.82%. The resulting confusion matrix is shown in Figure 36. When we combined the MPP cases, the SVM, and the BPNN we got an accuracy of 93.15%. The confusion matrix is shown in Figure 37. Finally, when we combined only the SVM and the BPNN we achieved our maximum accuracy of 94.14% on the test set. The confusion matrix is show in Figure 38.

4.6. Overall

See Figures 39, 40, and 41 for our final evaluation comparisons.

4.7. Cross Validation

We implemented a multi-threaded K-fold cross validation algorithm that performs a hyperparameter grid search with K sets of validation data from a combined set of the training and validation data. This performed very well when tuning hyperparameters for the majority of our models. The API functions similar to Sklearn's implementation of GridSearchCV.

5. Discussion

In this face mask detection project, we have firstly designed a convoluted network structure for imaging processing, which includes image resizing, gray-scaling and min-max scaling. Next, the codes of classifiers BPNN, SVM, MPP (case 1, 2 and 3), kNN and decision tree are prepared while the NB and BKS methods are also applied for classifier fusion, which includes NB-(SVM+BPNN; MPP+SVM+BPNN; MPP+SVM; MPP) and BKS-(SVM+BPNN; MPP+SVM+BPNN; MPP+SVM; MPP). Finally, we apply these classifiers to the convoluted data and predict the test samples, generating the confusion matrix, respectively. All of these classifiers present well prediction of the test sample with accuracy above 0.84. Notably,

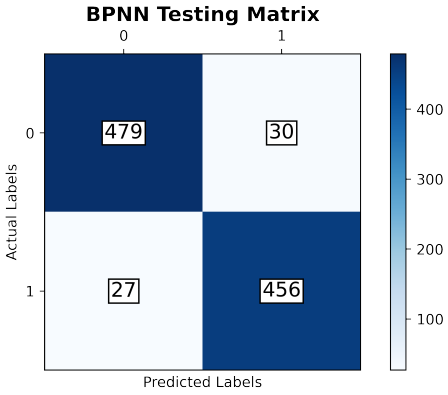

Figure 30. BPNN Confusion Matrix Testing

the BPNN, NB-(SVM+BPNN) and BKS-(SVM+BPNN) have the best performance with the overall accuracy of 0.94. This high accuracy may be due to two reasons. First, the convoluted network captures the main features after dimensionality reduction of each class; non-face-mask pictures have a sharp feature of nose and mouth while it is blur in face-mask pictures. Second, BPNN considers the picture is a linear combination of different eigen vectors. Assuming some eigen vectors effectively correspond to the mouth and nose area, the value of linear combination of basis function for the mask on and off pictures can be largely different. Therefore, applying the sigmoidal function can significantly separate these pictures. On the other hand, SVM can largely optimize the boundary of two classes; if mask on and off features are well separated, this boundary may be distinctive. These two reasons may demonstrate the high performance of BPNN and SVM while applying in predicting the test samples.

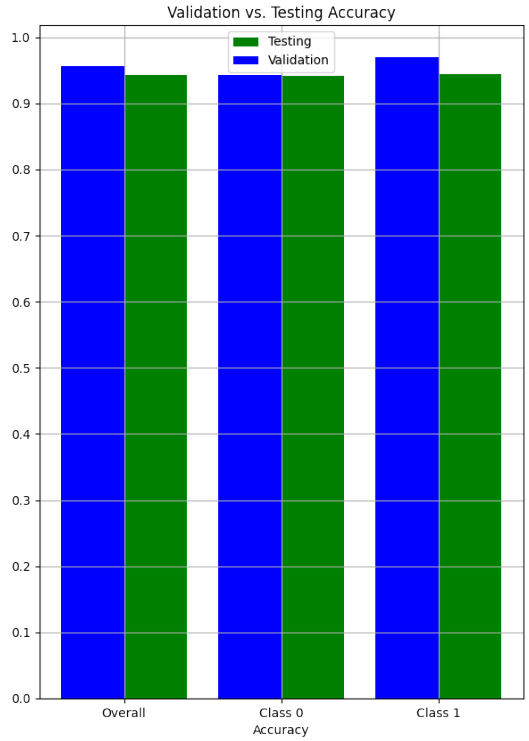

Figure 31. BPNN Validation vs. Testing

References

Chaitanya. Face mask detection — xception + haar cascade, 2021. URL https://www.kaggle.com/chaitanya99/face-mask-detection-xception-haar-cascade.

Holla, K. Facemask detection using tensorflow 2, 2021. URL https://www.kaggle.com/kaushikholla/facemask-detection-using-tensorflow-2.

Jangra, A. Face mask detection 12k images dataset, May 2020. URL https://www.kaggle.com/ashishjangra27/face-mask-12k-images-dataset.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization, 2018. URL https://jmlr.org/papers/volume18/16-558/16-558.pdf.

Nielsen, M. A. Neural networks and deep learning, 2015. URL http://neuralnetworksanddeeplearning.com/chap1.html.

Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations, 2017. URL http://learningsys.org/nips17/assets/papers/paper_16.pdf.

Sahoo, S. 2d convolution using python & numpy, Dec 2020. URL https://medium.com/analytics-vidhya/2d-convolution-using-python-numpy-43442ff5f381.

user10030086 and K., A. Implement max/mean pooling(with stride) with numpy, Mar 2019. URL https://stackoverflow.com/questions/54962004/implement-max-mean-poolingwith-stride-with-numpy.

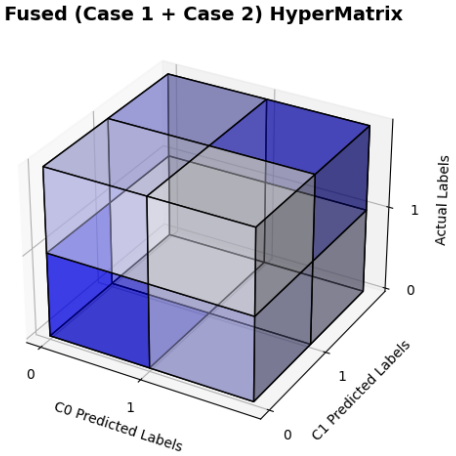

Figure 32. NB 3D Fused Hypermatrix

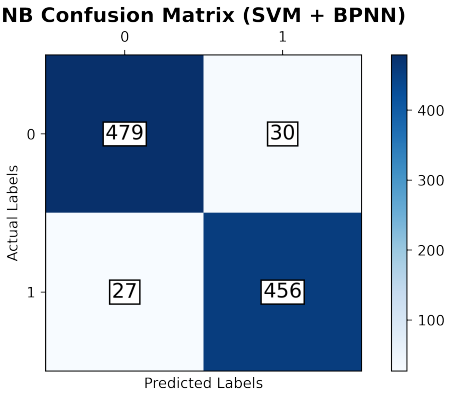

Figure 35. NB SVM with NN

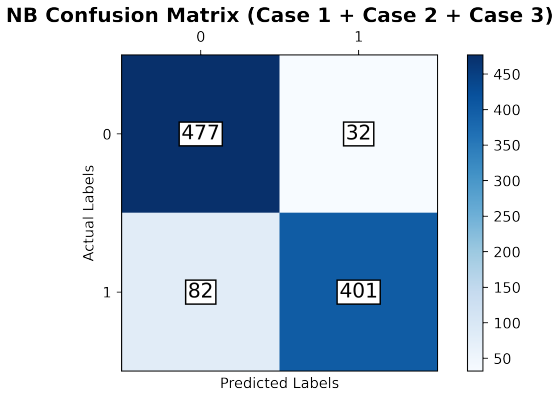

Figure 33. NB all cases of MPP

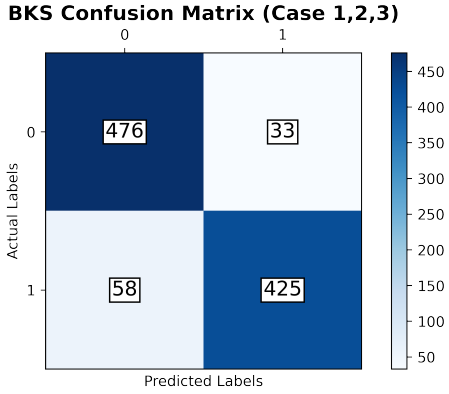

Figure 36. BKS all cases of MPP

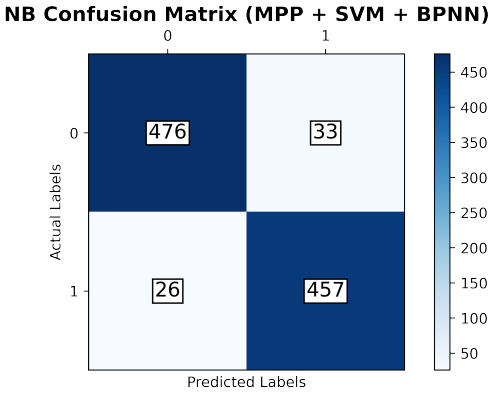

Figure 34. NB MPP, SVM, and NN

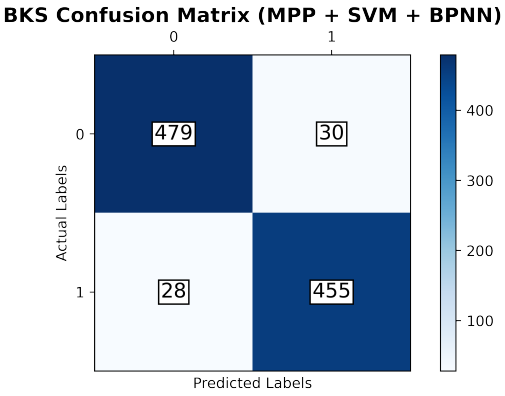

Figure 37. BKS MPP, SVM, and NN

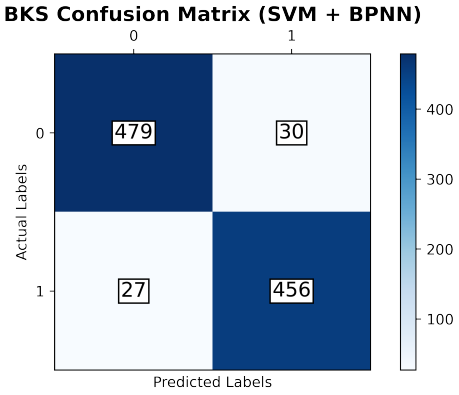

Figure 38. BKS SVM with NN

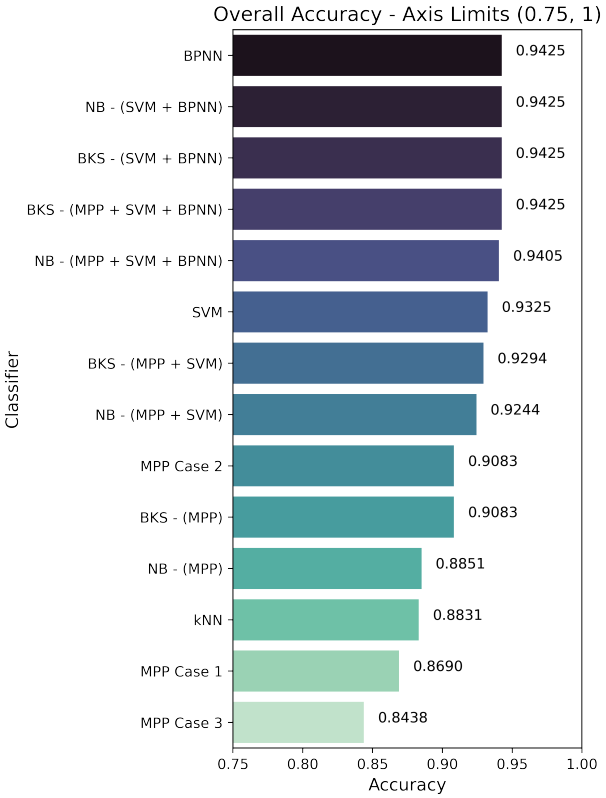

Figure 39. Overall Model Evaluation

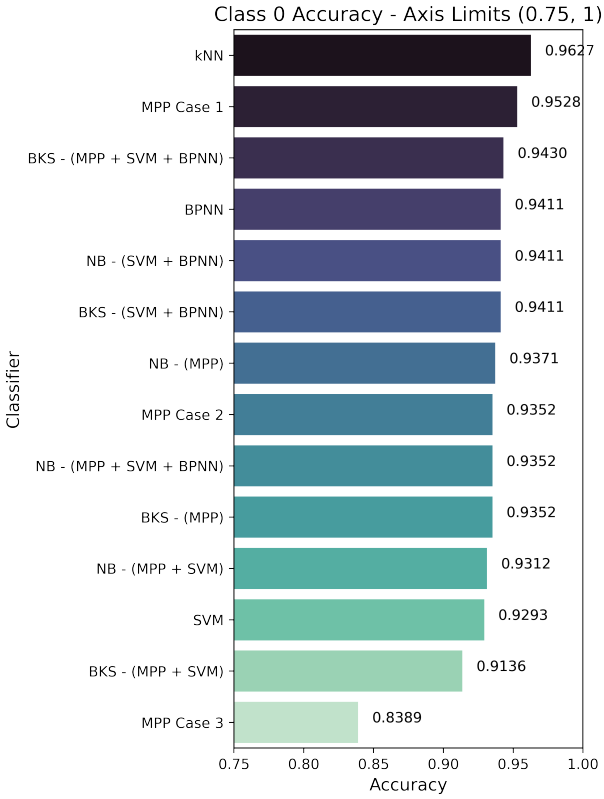

Figure 40. Class 0 Model Evaluation

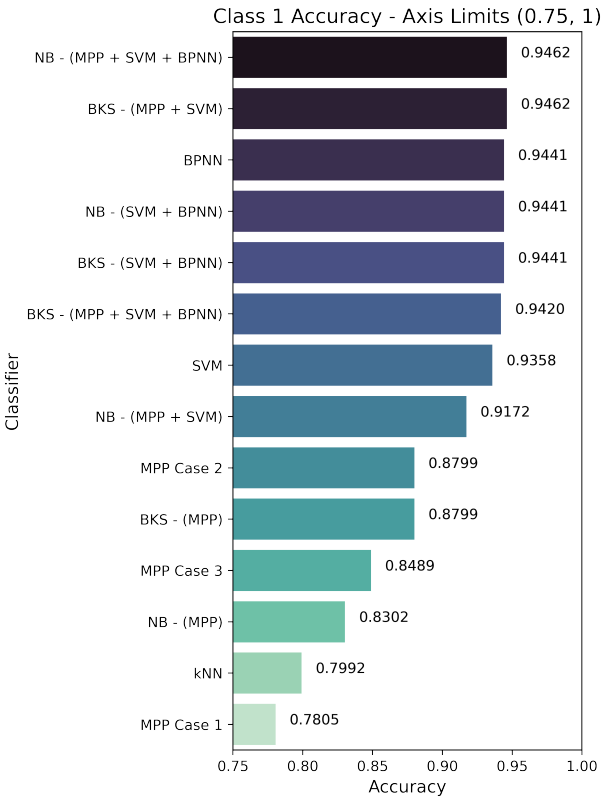

Figure 41. Class 1 Model Evaluation