```
#Summary Statistics and Various Plots
plot(AUC, main = "AUC over 100 Iterations \n(Naive Bayes Classifier)",
     xlab = "Iterations", ylab = "AUC", type = "l", col = "cadetblue")
hist(AUC, main = "Histogram for AUC \n(Naive Bayes Classifier)",
     xlab = "AUC Value", ylab = "Frequency", col = "firebrick3")
```

We follow the same intuition as when tuning the machine learning algorithms, by collecting the AUC. The nature of the logistic regression is such that it fits a model upon each iteration rather than choosing the most optimal regression solution, as some algorithms do. When plotting the AUC vector with respect to the iterations over time and plotting a histogram of the AUC vector, we observe the results shown in Figures 10-12 and 10-13.

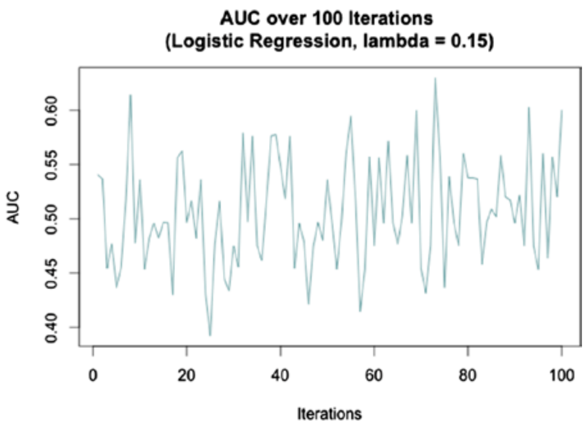


Figure 10-12. *Logistic regression AUC over 100 iterations*

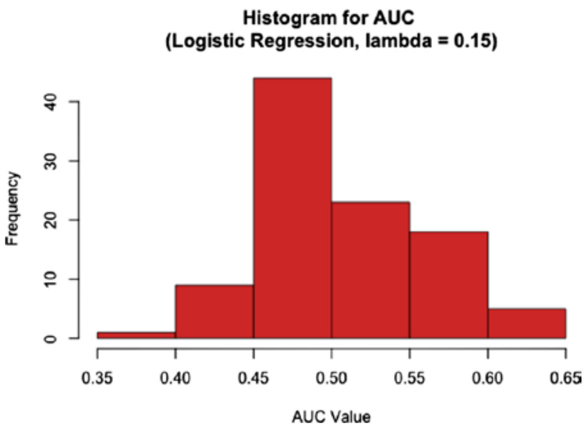


Figure 10-13. *Logistic regression AUC histogram over 100 iterations*

Numerically, we can summarize this vector using the following function:

```
summaryStatistics(AUC)

Mean           Std.Dev       Min          Max          Range
1 0.5063276    0.04964798    0.3920711    0.6297832    0.2377121
```

We'll keep these values in mind moving forward. When analyzing them as is, logistic regression is an insufficient classifier. Typically, we would like to see AUC scores be at least .70, because a score of .50 indicates that the model is correct only 50% of the time. Less than .50 is not optimal and arguably means that we should consider this classifier insufficient.

Method 3: K-Nearest Neighbors (KNN)

This is a fairly simple classification algorithm described in detail in Chapter 3. The purpose in picking this algorithm relative to another probabilistic algorithm is to create a diverse algorithm portfolio such that we can infer which types of algorithms are best suited to this task. As a note to the reader, the K-NN algorithm in the class package yields the classifications from the test data. To train your algorithm on the training data only, use the same data that you assign to the "train" argument:

```
#Method 3: K-Nearest Neighbor
#Tuning K Parameter (Number of Neighbors)
K <- seq(1, 40, 1)
AUC <- c()
for (i in 1:length(K)){
  rows <- sample(1:nrow(processedData), nrow(processedData)/2)
  y_h <- knn(train = processedData[rows, ], test = processedData[rows,],
  cl = second_date[rows], k = K[i], use.all = TRUE)
  AUC <- append(roc(y_h, as.numeric(second_date[rows]))$auc, AUC)
}

#Summary Statistics and Various Plots
plot(AUC, main = "AUC over K Value \n(K Nearest Neighbor)",  xlab = "K",
ylab = "AUC", type = "l", col = "cadetblue")
```

When looking at the plot of the AUC over K-value chart, we see the results shown in Figure 10-14.

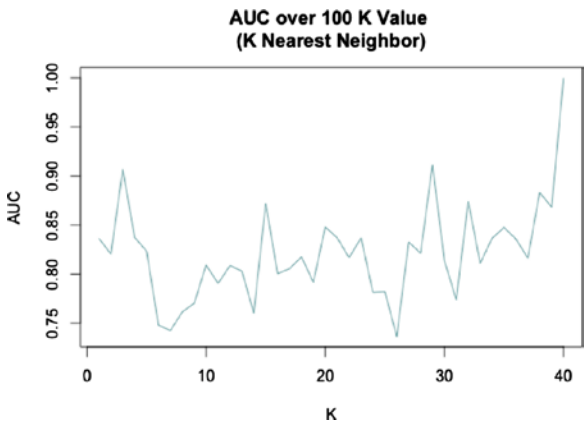


Figure 10-14. *KNN classifier AUC over 100 iterations*

The AUC score in the training phase is generally impressive for all the values, but it's reasonable to choose a lower K value than a large one to prevent overfitting. As such, we will choose a K of 3. Let's observe the AUC scores on the test set with our tuned model, as shown in Figures 10-15 and 10-16.

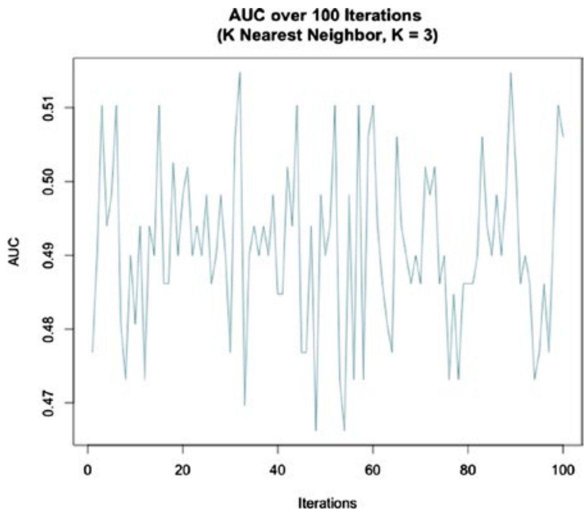


Figure 10-15. *KNN AUC over 100 iterations on test set*

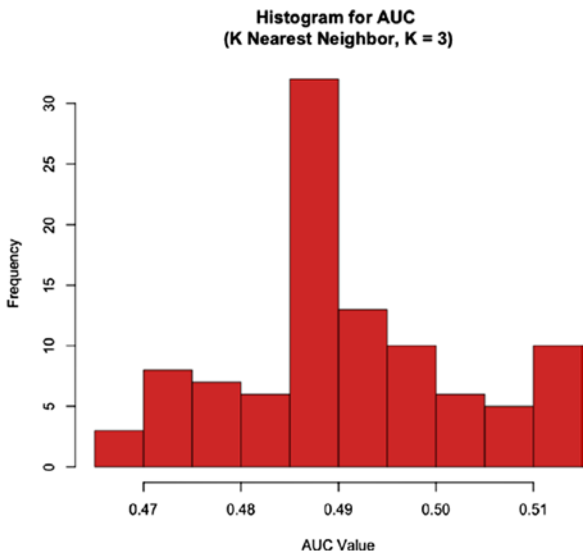


***Figure 10-16.** KNN AUC over 100 iterations on test set histogram*

Numerically, we evaluate the AUC vector as the following:

```
summaryStatistics(AUC)

Mean           Std.Dev       Min          Max          Range
1 0.445006     0.01126862    0.4257075    0.4663915    0.04068396
```

Finally, we predict out of sample and observe the following results:

```
#Predicting out of Sample
y_h <- knn(train = processedData[rows, ], test = processedData[-rows, ],
cl = second_date[-rows])
roc(y_h, as.numeric(second_date[-rows]))$auc
```

Area under the curve: 0.4638

We see a stark drop-off from the training set to the test set, in addition to the test set performance being objectively poor.

Method 2: Bayesian Classifier

I suspect that occurrence of a second date can be modeled by Bayesian estimators, so the first model we'll begin with is the Bayesian classifier. In the following code, first we perform two-fold cross-validation on the data set so that we evaluate the performance on the training set. In this particular model, very little tuning needs to occur, so we'll just observe the performance of the model over 100 iterations:

```
#Method 1: Bayesian Classifier
AUC <- c()
for (i in 1:100){
  rows <- sample(1:nrow(processedData), 92)
  bayesClass <- naiveBayes(y = as.factor(second_date[rows]),
  x = processedData[rows, ], data = processedData)
  y_h <- predict(bayesClass, processedData[rows, ], type = c("class"))
  AUC <- append(roc(y_h, as.numeric(second_date[rows]))$auc, AUC)
}

#Summary Statistics and Various Plots
plot(AUC, main = "AUC over 100 Iterations \n(Naive Bayes Classifier)",
     xlab = "Iterations", ylab = "AUC", type = "l", col = "cadetblue")

hist(AUC, main = "Histogram for AUC \n(Naive Bayes Classifier)",
     xlab = "AUC Value", ylab = "Frequency", col = "cadetblue")

summaryStatistics(AUC)
```

When executing the code, we append the AUC score to the vector AUC, as shown in the preceding code that is looped over for 100 iterations. A line plot and histogram of this vector is shown in Figures 10-17 and 10-18.

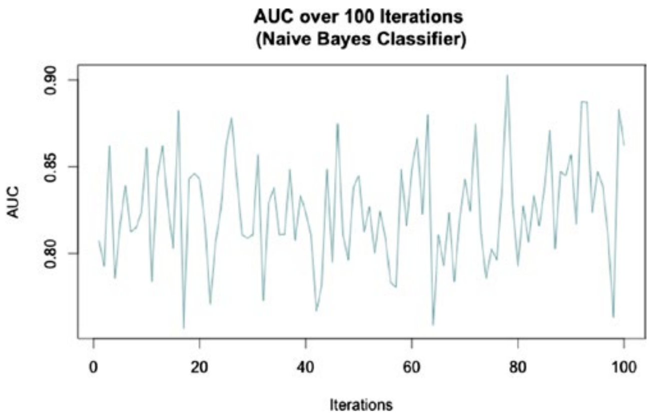


Figure 10-17. *Bayes classifier AUC performance over 100 iterations*

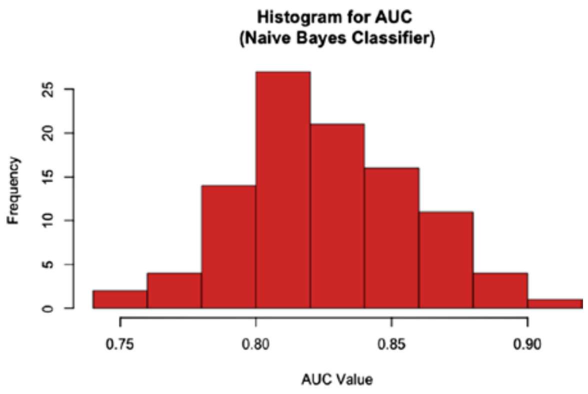


Figure 10-18. *Bayes classifier AUC histogram over 100 iterations*

We observe that the AUC scores have a slight right skew in their distribution and that the majority of the AUC scores are distributed within a relatively tight band of one another. When looking at the raw numerical data, we observe the following:

```
Mean            Std.Dev         Min           Max           Range
1 0.8251087     0.03142345      0.7567568     0.9027778     0.146021
```

These AUC scores yielded are more than generally acceptable for a model we choose, though we should still evaluate the performance of the model out of sample to be certain of how stable this process is:

```
#Predicting out of Sample
y_h <- predict(bayesClass, processedData[-rows, ], type = c("class"))
roc(y_h, as.numeric(second_date[-rows]))$auc
```

After executing the following code, we observe the following AUC score: area under the curve: 0.8219. This is acceptable within the distribution of the data yielded from the training set, with this AUC score trending towards the mean of the data.

When evaluating the solutions chosen, I strongly suggest choosing the Bayesian classifier given its stability from the training to the test set and superior AUC score above all other methods. In a practical setting, we would use the predictions out of the sample data to help influence our decision-making processes. In a professional context, this might include targeted marketing or recommendations to different users based on their dating profiles.

Summary

You now have a brief but comprehensive view into how I would recommend applying the concepts I've explained in the previous chapters. You should also note that although I've had success in implementing machine learning algorithms using this general process/methodology, this isn't the only way of training/tuning machine learning models. Nevertheless, I strongly emphasize the use of metrics and plotting the performance of the models with respect to these metrics when tuning different parameters. Chapter 11 will look at use examples of how to implement and use various deep learning models.

CHAPTER 11

Deep Learning and Other Example Problems

Now that I've sufficiently covered how to use and apply machine learning concepts, we should finally dive into applying and coding deep learning models using R. This can seem like a daunting task, but don't be intimidated. If you have been able to code everything successfully in this book, it's just a matter of adjusting to new packages. We will discuss a variety of deep learning examples, but will begin by dealing with simpler models and then eventually going on to more complex models. The purpose of these exercises is twofold:

- To show how to construct these models or access them from various packages
- To give examples of how they could be used in a practical concept

Autoencoders

Many of the other models described in the deep learning chapters of the book are relatively straightforward when it comes to how to use them, but I have found that the use of autoencoders does *not* become automatically clear. Therefore, I want to explore a use case in which the use of autoencoders is made abundantly clear in a practical context. Let's consider a case in which we would like to use an autoencoder to improve the performance of a classification algorithm from Chapter 10. Specifically, I mean the classification problem we walked through, in which we were trying to determine whether a pair of individuals will go on a second date or not based on several features. Let's begin by working with the Bayesian classifier:

```
#Bayes Classifier
#Bayes Classifier
AUC <- c()
for (i in 1:100){
  rows <- sample(1:nrow(processedData), 92)
  bayesClass <- naiveBayes(y = as.factor(second_date[rows]),
x = processedData[rows, ], data = processedData)
```

© Taweh Beysolow II 2017

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_11

```
  y_h <- predict(bayesClass, processedData[rows, ], type = c("class"))
  AUC <- append(roc(y_h, as.numeric(second_date[rows]))$auc, AUC)
}

summaryStatistics(AUC)
curve <- roc(y_h, as.numeric(second_date[rows]))
plot(curve, main = "Bayesian Classifier ROC")
```

When executing the preceding code, it yields what is shown in Figure 11-1.

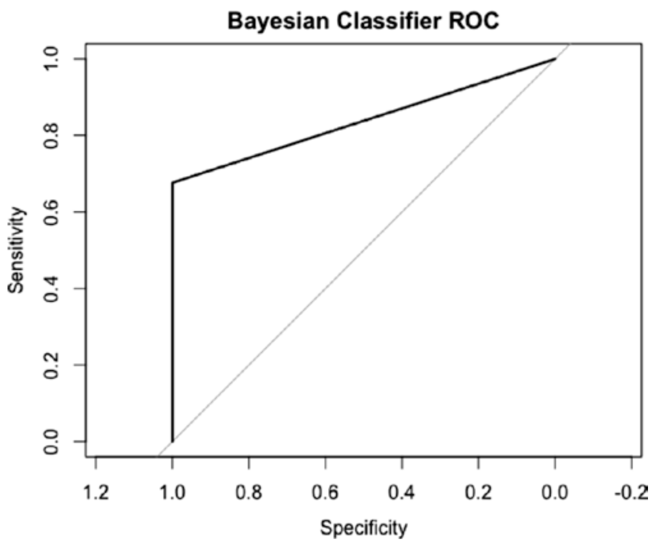


Figure 11-1. *ROC plot for Bayesian classifier*

We observe AUC scores of this model when collecting sample statistics:

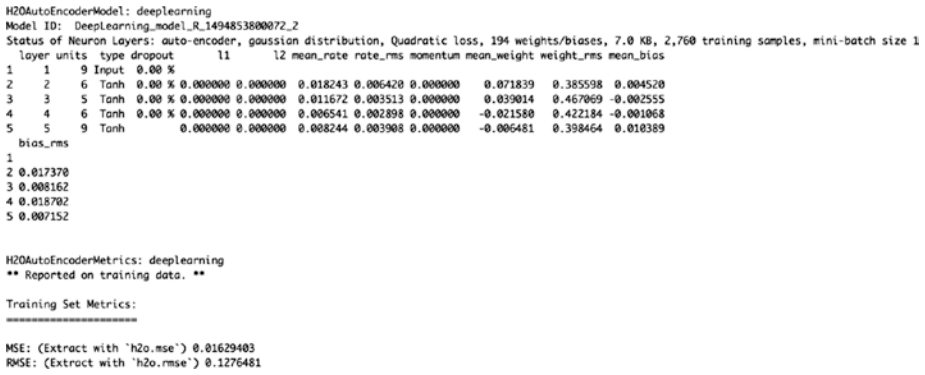
```
Mean       Std.Dev     Min        Max    Range
0.8210827  0.02375922  0.7571429  0.875  0.1178571
```

These are objectively good scores. However, for the purpose of this example, we're going to use an autoencoder to help improve the performance of this model even further. This is where I introduce *h2o*. h2o produces a deep learning framework for R (along with other languages) that you will find useful for implementing many models. I encourage you to search through the documentation, because some implementations of deep learning models are hard to find (not to mention finding robust implementations).
So let's initialize h2o and use an autoencoder:

```
#Autoencoder
h2o.init()
training_data <- as.h2o(processedData, destination_frame = "train_data")
```

```
autoencoder <- h2o.deeplearning(x = colnames(processedData),
 training_frame = training_data, autoencoder = TRUE, activation = "Tanh",
 hidden = c(6,5,6), epochs = 10)
autoencoder
```

h2o is similar to TensorFlow in that each session must be initialized. After this is initialized, whatever data passes through the models used must be transformed into an h2o-friendly format. We perform that transformation on our training data. Our autoencoder has three hidden layers, each of which has six, five, and six respective neurons within the given layers (denoted by the "hidden" argument within the `h2o.deeplearning()` function. We use tanh as our activation function. Upon executing the following code, we see what is shown in Figure 11-2.



```
H2OAutoEncoderModel: deeplearning
Model ID:  DeepLearning_model_R_1494853800072_2
Status of Neuron Layers: auto-encoder, gaussian distribution, Quadratic loss, 194 weights/biases, 7.0 KB, 2,760 training samples, mini-batch size 1
  layer units  type dropout       l1       l2 mean_rate rate_rms momentum mean_weight weight_rms mean_bias
1     1     9 Input  0.00 %
2     2     6  Tanh  0.00 % 0.000000 0.000000  0.018243 0.006420 0.000000    0.071839   0.385598  0.004520
3     3     5  Tanh  0.00 % 0.000000 0.000000  0.011672 0.003513 0.000000    0.039014   0.467069 -0.002555
4     4     6  Tanh  0.00 % 0.000000 0.000000  0.006541 0.002898 0.000000   -0.021580   0.422184 -0.001068
5     5     9  Tanh         0.000000 0.000000  0.008244 0.003908 0.000000   -0.006481   0.398464  0.010389
  bias_rms
1
2 0.017370
3 0.008162
4 0.018702
5 0.007152


H2OAutoEncoderMetrics: deeplearning
** Reported on training data. **

Training Set Metrics:
=====================

MSE: (Extract with `h2o.mse`) 0.01629403
RMSE: (Extract with `h2o.rmse`) 0.1276481
```

Figure 11-2. *Summary of autoencoder function*

Note the MSE values. Because we're trying to recreate inputs of a function, this becomes a regression task. So we evaluate the effectiveness of this algorithm using the traditional regression statistics (MSE and RSME). Let's take a close look at the MSE yielded here and view the MSE with respect to the index of the data frame that holds the training data:

```
#Reconstruct Original Data Set
syntheticData <- h2o.anomaly(autoencoder, training_data, per_feature = FALSE)
errorRate <- as.data.frame(syntheticData)

#Plotting Error Rate of Feature Reconstruction
plot(sort(errorRate$Reconstruction.MSE), main = "Reconstruction Error Rate")
```

The `h2o.anomaly()` function uses the autoencoder to detect *anomalies*, which statistically we define as observations whose MSE during the reconstruction process are significantly higher than others. When executing the preceding code, we yield Figure 11-3.

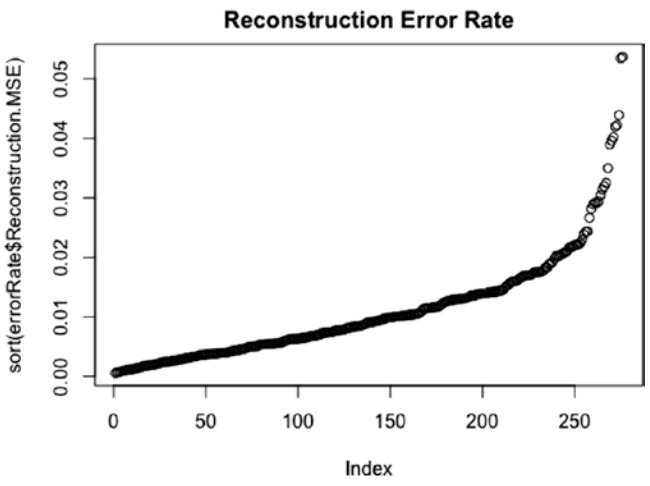


Figure 11-3. *Plot of reconstruction error*

We can see that there is a steady increase of the MSE but also a sharp increase from the index level 225 through the end of the training data. We can reasonably state that the outliers are generally these last inputs. With this in mind, we'll use a threshold determined by the MSE when segregating outliers from non-outliers into their respective subsets. We seek to train our Bayesian classifier by fitting our model to these subsets and seeing how the performance of the model, with respect to the AUC score, improves (or doesn't):

```
#Removing Anomolies from Data
train_data <- processedData[errorRate$Reconstruction.MSE < 0.01, ]

#Bayes Classifier
AUC <- c()
for (i in 1:100){
  rows <- sample(1:nrow(processedData), 92)
  bayesClass1 <- naiveBayes(y = as.factor(second_date[rows]), x =
  processedData[rows, ], data = processedData)
  y_h <- predict(bayesClass1, processedData[rows, ], type = c("class"))
  AUC <- append(roc(y_h, as.numeric(second_date[rows]))$auc, AUC)
}

#Summary Statistics
summaryStatistics(AUC)
```

We follow the same general steps we followed in Chapter 10 with respect to model training, collecting samples of the AUC statistic over 100 trials. The only difference here is that we're using a subset of the data with respect to the index values that fall below the MSE threshold. When looking at the summary statistics, we observe the following:

```
Mean       Std.Dev     Min   Max        Range
0.8274664  0.03076285  0.75  0.9117647  0.1617647
```

When comparing the distribution of our results to the original model, we observe a slightly higher mean, a higher max. However, we also observe a lower minimum. Therefore, the range and standard deviation of our results increase. Let's evaluate our results when we only look at anomalies:

```
###########################################################################
#Using only Anomalies in Data Set
train_data <- processedData[errorRate$Reconstruction.MSE >= 0.01, ]

#Bayes Classifier
AUC <- c()
for (i in 1:100){
  rows <- sample(1:nrow(processedData), 92)
  bayesClass2 <- naiveBayes(y = as.factor(second_date[rows]),
  x = processedData[rows, ], data = processedData)
  y_h <- predict(bayesClass2, processedData[rows, ], type = c("class"))
  AUC <- append(roc(y_h, as.numeric(second_date[rows]))$auc, AUC)
}

#Summary Statistics
summaryStatistics(AUC)
```

When executing the preceding code, we see the following results:

```
Mean       Std.Dev     Min        Max        Range
0.8323727  0.03168166  0.7692308  0.9107143  0.1414835
```

Here we observe that this distribution contains the highest mean and minimum, with moderate results with respect to range and standard deviation. When choosing between the two data sets, I would argue for using the second subset in this instance due to the superior AUC score performance on average—and given the fact that at a minimum, we can still expect a higher score.

The importance of this technique lies in the fact that it is an effective method by which you can fit superior models on subsets of data. This will be extremely handy if you find you have a data set that is smaller than you would like. There are times when you can find yourself stuck trying to tweak a model whose performance is slightly unsatisfactory, despite using proper cross-validation techniques, data preprocessing techniques, and parameter tuning techniques. In instances where this is due to lack of data, this technique would be the first I tried to use prior to trying to acquire more data. As for the final step in our experiment, let's use the fitted models and see how they perform out of sample:

```
#Fitted Models and Out of Sample Performance
AUC1 <- AUC2 <- c()

for (i in 1:100){
  rows <- sample(1:nrow(processedData), 92)
  y_h1 <- predict(bayesClass1, processedData[-rows,], type = c("class"))
  y_h2 <- predict(bayesClass2, processedData[-rows,], type = c("class"))
  AUC1 <- append(roc(y_h1, as.numeric(second_date[-rows]))$auc, AUC1)
  AUC2 <- append(roc(y_h2, as.numeric(second_date[-rows]))$auc, AUC2)
}
summaryStatistics(AUC1)
summaryStatistics(AUC2)
```

When executing the preceding code, we see the results for the model fitted against the subset without and with only anomalies respectively in Figures 11-4 and 11-5:

```
Mean       Std.Dev     Min        Max        Range
0.7890102  0.01468805  0.75       0.8194444  0.06944444
Mean       Std.Dev     Min        Max        Range
0.8303613  0.01506222  0.7957983  0.8688836  0.07308532
```

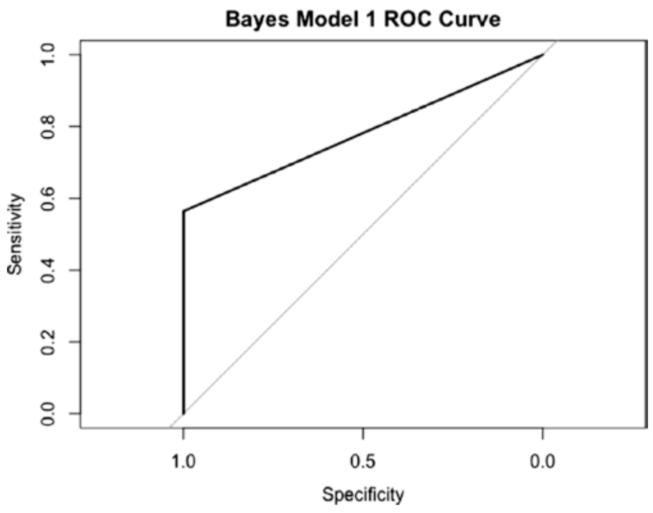


Figure 11-4. *ROC curve for Bayes model without anomalies (AUC : 0.7821)*

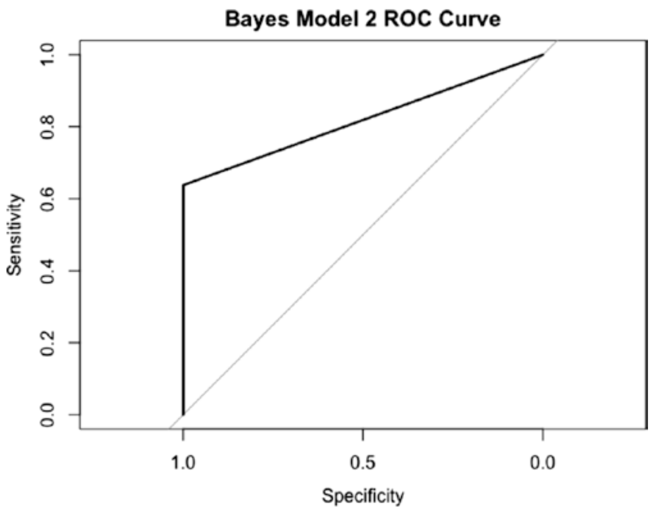


Figure 11-5. *ROC curve for Bayes model w/o anomalies (AUC: 0.8188)*

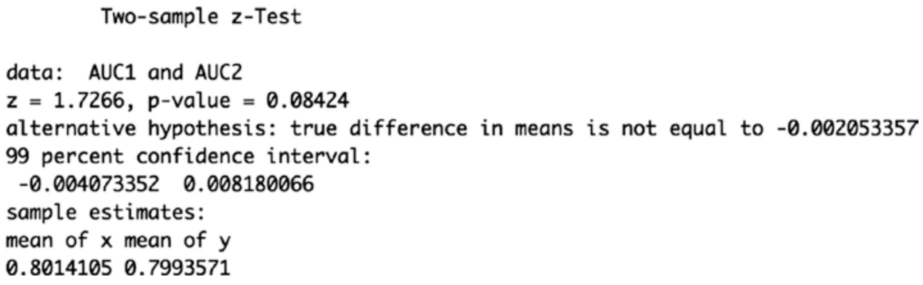
When reviewing the results from our experiment, it has become abundantly clear that the second model, fitted with only anomalies, produces a markedly better model than the model fit with observations that aren't anomalies. But before we become entirely convinced that we should use the second model, let's quickly perform a two-sided hypothesis test on using data from both of these models.

Being that we sampled our results 100 times, we can safely use a Z-test. As such, we set the Z-test parameters as shown in the following code:

```
#Two Sided Hypothesis Test
require(BSDA)

z.test(x = AUC1, y = AUC2, alternative = "two.sided", mu = mean(AUC2) - mean(AUC1),
                 conf.level = 0.99, sigma.x = sd(AUC1), sigma.y = sd(AUC2))
```

When executing the preceding function, it yields the output shown in Figure 11-6.



```
        Two-sample z-Test

data:  AUC1 and AUC2
z = 1.7266, p-value = 0.08424
alternative hypothesis: true difference in means is not equal to -0.002053357
99 percent confidence interval:
 -0.004073352  0.008180066
sample estimates:
mean of x mean of y
0.8014105 0.7993571
```

Figure 11-6. *Two-sided hypothesis test results*

Statistically, within a 99% confidence interval, we have determined that the results of the two models are statistically different from one another and therefore we can confidently choose the second Bayesian model fitted, knowing that it is the superior model.

Convolutional Neural Networks

When I discussed CNNs in Chapter 5, I showed the power of this model by discussing the MNIST digit recognition use case. Although that was at one point the primary use case of CNNs, they are now currently being used for increasingly more difficult and complex tasks. Now I'd like to explore a use case in which we're trying to distinguish between different objects of significantly more complexity than handwritten digits. In this tutorial, we'll be using the Caltech 101 dataset, which contains 101 object categories with between 60 and 800 images in each category. We'll take various images from each category, doing so in such a way that we get diversity of images without picking starkly different pictures. We'll be choosing between images of guitars and laptops. Sample of theses photos are shown in Figures 11-7 and 11-8.

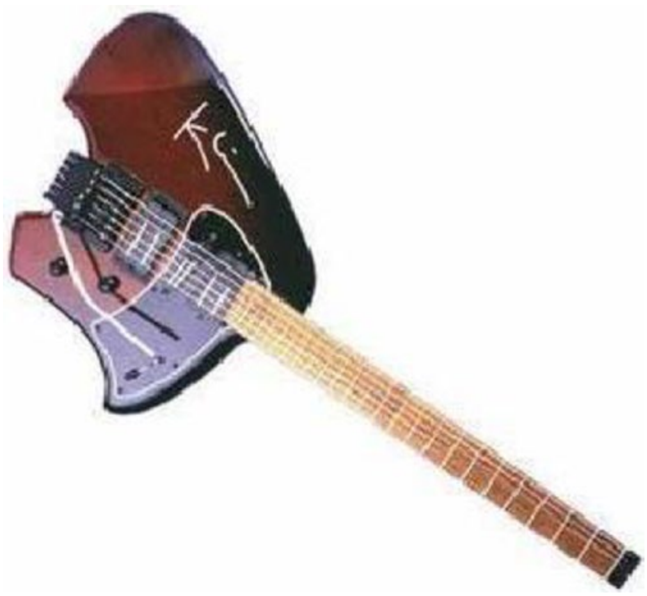


Figure 11-7. *Photo of guitar*

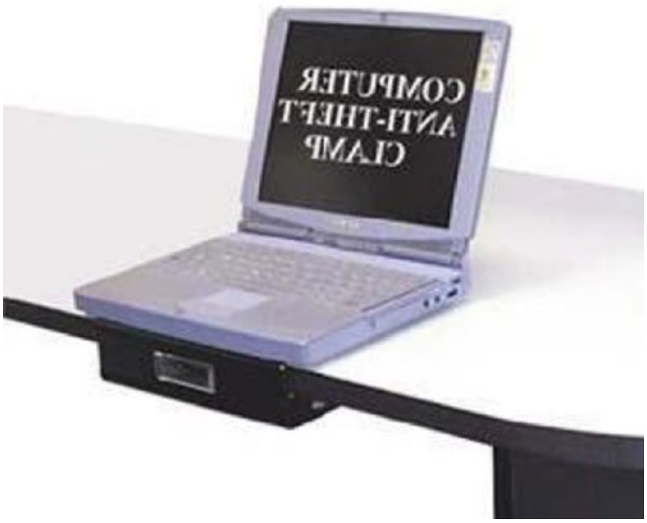


Figure 11-8. *Photo of laptop*

These images are pieces of technology, but they're distinctly different from each other in such a way that we would expect a human to be able to distinguish them. Let's now discuss how we should prepare our data for the CNNs.

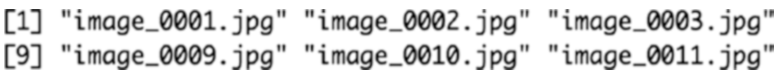
Preprocessing

Working with image files requires a particular type of preprocessing that we haven't discussed in detail yet, mainly because image recognition and computer vision is a very specific subfield of computer science. It would be wise to seek other texts to build upon your understanding of computer vision, but this passage will give you a basic overview. We're working with color images, each with dimension x, y, z, where x and y are specific to each photo but z is always 3. Image files, insofar as a computer understands them, are three layers of matrices stacked on top of each other, with each pixel being an individual entry in that matrix. For this task, I recommend you use the EBImage package so you can grayscale and resize images. To help with the training time of the neural network, we'll be resizing images so they're smaller, and therefore the neural network takes in less data. But let's walk through our preprocessing step by step:

```
#Loading required packages
require(mxnet)
require(EBImage)
require(jpeg)
require(pROC)

#Downloading the strings of the image files in each directory
guitar_photos <- list.files("/file/path/to/image")
laptop_photos <- list.files("/file/path/to/image")
```

The Caltech library is organized into directories with multiple levels, so be mindful when trying to access these images in an automated fashion. All the directories for each category have the same format for the filenames: the image file is denoted as image_000,X, where X is the number of the image in the directory. But each directory has a different number of files, so we should use the `list.files()` function to collect the names of all the image files within the directories. We use them in the following section of code. The contents of the guitar photos directory when using the `list.files()` function are shown in a truncated form in Figure 11-9.



```
[1] "image_0001.jpg" "image_0002.jpg" "image_0003.jpg"
[9] "image_0009.jpg" "image_0010.jpg" "image_0011.jpg"
```

Figure 11-9. *List of files from image directory*

Now that we have the names of the individual files, we can load them into the `img_data` data frame using the following process:

```
#Creating Empty Data Frame
img_data <- data.frame()

#Turning Photos into Bitmaps
#Guitar Bitmaps
for (i in 1:length(bass_photos)){
  img <- readJPEG(paste("/path/to/image/directory/", guitar_photos[i], sep = ""))
```

We use the `paste` function here to combine the directory with the image with the string such that it leads us to the data. Using the `readJPEG()` function from the jpeg package, we can read the image into a bitmap, as described earlier as the stack of matrices. Each dimension represents the three colors (red, blue, and green) that make up every color photo. But to reduce the complexity of the images we're working with, we're going to convert these images to greyscale (black and white). When working with black and white images, we assign the pixel values a number between 0 and 1, with 0 representing black and 1 representing white. The colors in between determine the degree of intensity toward either side of the spectrum a particular color:

```
#Reshape to 64x64 pixel size and grayscale image
img <- Image(img, dim = c(64, 64), color = "grayscale")

#Resizing Image to 28x28 Pixel Size
img <- resize(img, w = 28, h = 28)
img <- img@.Data
```

We perform the reshaping and resizing of various images using the `resize()` function provided in EBImage. If you're interested in viewing what images look like when they're grayscaled, feel free to experiment with the `display()` and `Image()` functions accordingly. After the image is resized, we take the bitmap and convert it into a vector for a better storage method. Finally, we must add a label to the vector of data for when we're creating and training a model. This will be useful when calculating the accuracy of our model. Specifically, guitars will be labeled as 1 and laptops will be labeled as 2:

```
  #Transforming to vector
  img <- as.vector(t(img))

  #Adding Label
  label <- 1

  img <- c(label, img)

  #Appending to List
 img_data <- rbind(img_data, img)

}
```

We repeat this process for the laptop images. If you want to use this structure of preprocessing and model evaluation, feel free to do so—or experiment with alternative preprocessing methods. Prior to creating the CNN model, we must ensure that the input format for the model is correct. MXNet and many neural network models have specific formats that you should be familiar with. The first step is to create a training and test set. For this example, we'll be splitting the data set such that we train against 75% of the data and test against the remaining 25%. We now will transform the data such that it was a matrix in which each row was a different image observation, with the label as the first column entry and the bitmap values as the successive column entries. We'll then strip the label from the X matrix and use this as the values in the corresponding order of observations for the y vector. We then perform cross-validation using the `sample()` function:

```
#Transforming data into matrix for input into CNN
training_set <- data.matrix(img_data)

#Cross Validating Results
rows <- sample(1:nrow(training_set), nrow(training_set)*.75)

#Training Set
x_train <- t(training_set[rows, -1])
y_train <- training_set[rows, 1]
dim(x_train) <- c(28,28, 1, ncol(x_train))
```

In the preceding code, it's important to point out a distinct detail that if omitted will prevent you from being able to execute your code. The MXNet CNN model *only* takes an X matrix that is 4 dimensions. Be *sure* to remember this—otherwise you'll waste time debugging this issue! We also alter the dimensions of the test set accordingly:

```
#Test Set
x_test <- t(training_set[-rows, -1])
y_test <- training_set[-rows, 1];
dim(x_test) <- c(28,28, 1, ncol(x_test))
```

Now that we've finished preprocessing our data, we can finally begin to build and train our model.

Model Building and Training

CNN models are built in such a way that the data passes through each layer, but the only layer that's actually inputted to the `FeedForward()` function is the final layer. So we build the model prior to it being activated here. Some packages might be more proprietary and require less architecture, but MXNet allows for a significant degree of customization that would be useful if you would like to construct different ConvNet structures, such as those elaborated upon in Chapter 5. If you would like to improve upon the results here, that may be a good use of your time.

Let’s move to the architecture. We’ll be using a generic LeNet architecture here, as is the standard for image recognition tasks. As such, we organize the layers in the same manner:

```
data <- mx.symbol.Variable('data')

#Layer 1
convolution_l1 <- mx.symbol.Convolution(data = data, kernel = c(5,5),
num_filter = 20)
tanh_l1 <- mx.symbol.Activation(data = convolution_l1, act_type = "tanh")
pooling_l1 <- mx.symbol.Pooling(data = tanh_l1, pool_type = "max", kernel =
c(2,2), stride = c(2,2))

#Layer 2
convolution_l2 <- mx.symbol.Convolution(data = pooling_l1, kernel = c(5,5),
num_filter = 20)
tanh_l2 <- mx.symbol.Activation(data = convolution_l2, act_type = "tanh")
pooling_l2 <- mx.symbol.Pooling(data = tanh_l2, pool_type = "max",
kernel = c(2,2), stride = c(2,2))
```

We first start by creating a dummy `data` variable that will be used to pass the x matrix values in a file format friendly to the ConvNet here. `data` passes through each layer, as discussed in Chapter 5, where the model builds from lower abstractions to higher abstractions of the data to make a determination. Here, we will use a stride of 2 as generally recommended, 20 filters in the first Conv layer, and 50 filters in the second Conv layer. As an activation function, we use tanh. This activation function will be held constant throughout the entire model with the exception of the output function:

```
#Fully Connected 1
fl <- mx.symbol.Flatten(data = pooling_l2)
full_conn1 <- mx.symbol.FullyConnected(data = fl, num_hidden = 500)
tanh_l3 <- mx.symbol.Activation(data = full_conn1, act_type = "tanh")

#Fully Connected 2
full_conn2 <- mx.symbol.FullyConnected(data = tanh_l3, num_hidden = 40)

#Softmax Classification Layer
CNN <- mx.symbol.SoftmaxOutput(data = full_conn2)
```

The data continues to pass to the fully connected layers. Respectively, there are 500 and 40 hidden neurons in the fully connected layers. Finally, the data reaches the last layer, where we have a softmax classifier to determine the class of the observations.

Before we make any predictions, though, we must train our parameters using the method suggested in the previous section. When possible, particularly in the case of neural networks, using a local search method for packages that support these functionalities is highly recommended. Specifically, h2o supports a grid search function to tune parameters. Although here we're using MXNet, it's useful for readers to be aware of packages that do provide these functionalities.

Let's begin by training the parameters:

```
#Learning Rate Parameter
AUC <- c()
learn_rate <- c(0.01, 0.02, 0.03, 0.04)
CPU <- mx.cpu()

for (i in 1:length(learn_rate)){
  cnn_model <- mx.model.FeedForward.create(CNN, X = x_train,
  y = y_train, ctx = CPU, num.round = 50, array.batch.size = 40,
learning.rate = learn_rate[i],
momentum = 0.9, eval.metric = mx.metric.accuracy,
epoch.end.callback = mx.callback.log.train.metric(100),
 optimizer = "sgd")
#Code redated partially, please check github!
```

Similar to other neural network models, the learning rate parameter determines the magnitude of the gradient in updating the weights connecting the layers to each other. We give an array and plot the AUC, with respect to the tuning parameter in Figure 11-10.

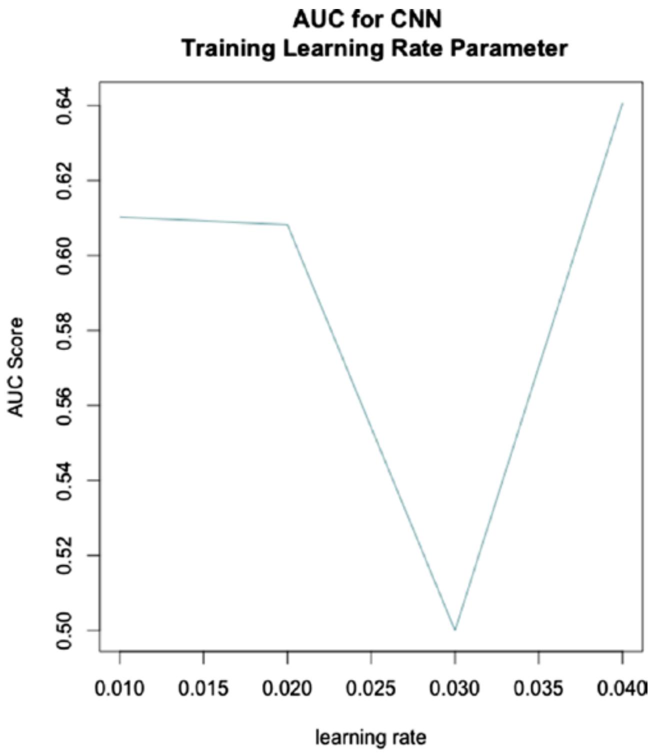


Figure 11-10. *AUC score over learning rate*

We can clearly see that a learning rate of 0.04 here is the most optimal because it yields the highest AUC score.

Let's now train the momentum parameter:

```
AUC1 <- c()
mom <- c(0.5, 0.9, 1.5)
for (i in 1:length(mom)){
cnn_model <- mx.model.FeedForward.create(CNN, X = x_train, y = y_train, ctx
= CPU, num.round = 50, array.batch.size = 40, learning.rate = 0.04,
momentum = mom[i], eval.metric = mx.metric.accuracy,
epoch.end.callback = mx.callback.log.train.metric(100), optimizer = "sgd")
#Code redacted partially, please check github!
```

When we execute the preceding code, we receive the results shown in Figure 11-11.

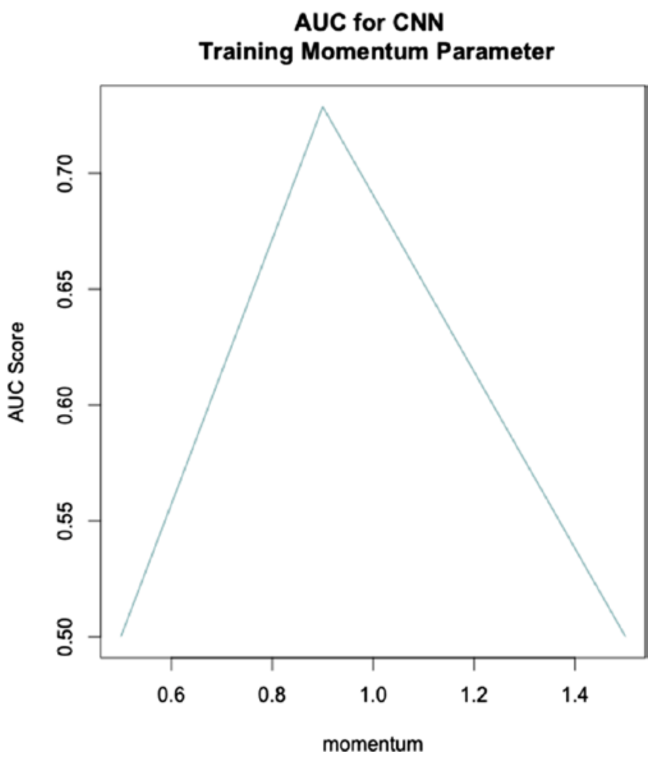


Figure 11-11. *AUC over momentum value*

When evaluating the results from the different parameters, we shall set the momentum value as 0.9. Now that we've tuned these two parameters, we can start training the tuned model in the final section and evaluating its performance on the test and training set:

```
#Fitted Model Training
cnn_model <- mx.model.FeedForward.create(CNN, X = x_train, y = y_train, ctx
= CPU, num.round = 150, array.batch.size = 40,
learning.rate = 0.04, momentum = 0.9, eval.metric = mx.metric.accuracy,
initializer = mx.init.normal(0.01) , optimizer = "sgd")

#Calculating Training Set Accuracy
y_h <- predict(cnn_model, x_train)
Labels <- max.col(t(y_h)) - 1
roc(as.factor(y_train), as.numeric(Labels))
curve <- roc(as.factor(y_train), as.numeric(Labels))
#Code partially redacted, please check github!
```

Before executing the code, I would like to point out one detail. Here, we have not enabled GPU training. If you want to decrease training time and improve computational performance, look into the necessary steps in the MXNet documentation to enable this feature. For this example, we'll be using CPU training. You should also be aware that the temptation to increase the `num.round` parameter will often be strong, as this will directly affect the accuracy of the model on the training set data. Beware that setting this parameter too high will cause overfitting, particularly on a data set the size of the one we're using in this example. When executing the preceding code, the user should see the terminal printing out the training accuracy in a format such as the following:

```
[184] Train-accuracy=0.708333333333333
[185] Train-accuracy=0.708333333333333
[186] Train-accuracy=0.708333333333333
[187] Train-accuracy=0.708333333333333
[188] Train-accuracy=0.708333333333333
```

The number on the left side of the words `Train-accuracy` represents the current iteration, which will run until the number indicated in the `num.round` parameter. The `accuracy` parameter used here is equivalent to the AUC score and is given by the `mx.metric.accuracy` object. As always, learning rates are difficult to approximate, but we can mitigate the loss of accuracy by adjusting the weights within the neural network using the stochastic gradient descent optimizer. When executing the code, we yield Figure 11-12.

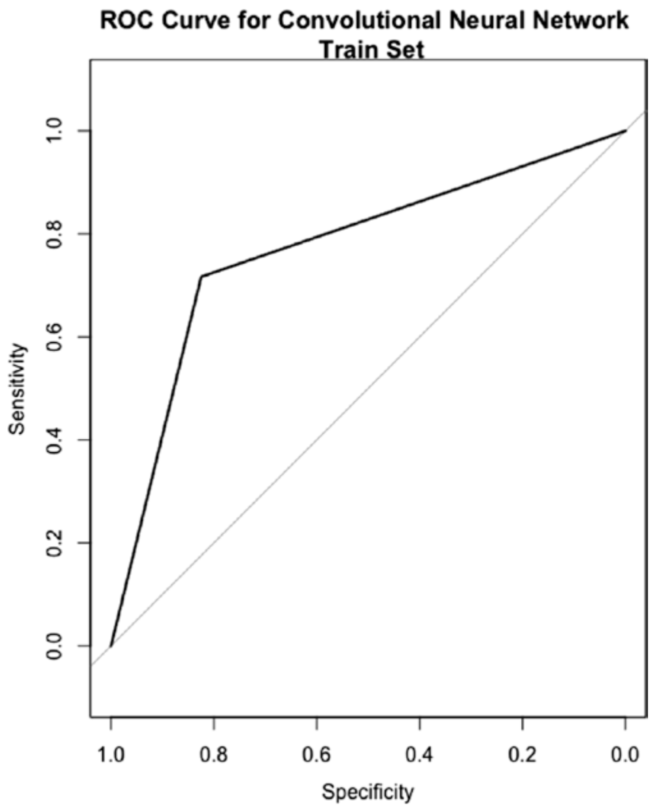


Figure 11-12. *ROC plot for CNN over training data*

This ROC plot has an AUC of 0.7706. When assessing the performance on the test data, Figure 11-13 and results are yielded.