

# Using Built-in nnet Function

Michael Yoensky

December 17, 2009

---

## Contents

|          |                                    |           |
|----------|------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                | <b>1</b>  |
| <b>2</b> | <b>Motivation</b>                  | <b>1</b>  |
| <b>3</b> | <b>Methods</b>                     | <b>2</b>  |
| <b>4</b> | <b>Performance Analysis</b>        | <b>2</b>  |
| <b>5</b> | <b>Analysis of Parameters</b>      | <b>12</b> |
| <b>6</b> | <b>Analysis of Implementations</b> | <b>12</b> |
| <b>7</b> | <b>Discussion</b>                  | <b>13</b> |
| <b>8</b> | <b>Conclusions</b>                 | <b>13</b> |

---

## 1 Introduction

This assignment was a student created assignment to enable each individual to further pursue an area of machine learning that interested him or her throughout the semester. This particular project was chosen to find the differences and advantages or disadvantages of the custom function to handle neural networks that was used throughout the semester, and the similar function that is built into the standard R libraries. [4]

Several methods of comparing the two functions are used throughout this report. The first comparisons were made by using the functions to analyze the same data, and determine the differences in analysis. Next the parameters available for each implementation are compared, the matching parameters are mentioned, and any unique parameters to either implementation are also specified. Finally, the code of the built in library is compared to the code of the custom developed neural network function written for use by this class.

In section 2 the reasons for selecting this particular topic are summarized. Section 3 describes any specific methods used to perform this assignment. The next three sections lay out in detail the exact methods used to compare the two implementations and is broken down into three distinct sections to focus on performance analysis in section 4, analysis of parameters in section 5, and analysis of the implementations in section 6. The results of the experiments are discussed in general in section 7 and conclusions about the results are discussed in section 8.

## 2 Motivation

Neural networks are a very useful way to perform classification and regression on a given set of data. Many machine learning algorithms, such as logistic regression and nonlinear logistic regression can be implemented

using neural networks. [1]

This course has thus far required development of code to implement neural networks of varying complexity throughout the semester. They have been used to predict the average miles-per-gallon for a car. They were also used to identify hand-drawn digit images and to learn to balance the ball on the center of a beam. [3]

It is interesting to determine how the custom functions written in R to handle neural networks compare to the `nnet` function built into R. There is value in understanding which implementation can lead to the most accurate predictions, which implementation requires the fewest number of epochs, and which implementation takes the least amount of time to run with comparable input parameters. In addition to these items, this work will also investigate the algorithm used by the built-in `nnet` function as well as identify and discuss any limitations of the built in `nnet` function calls.

### 3 Methods

For this project there were two functions implementing neural networks that were used. The first implementation was the one previously used in class, and the second is the implementation built into R. A wine dataset was chosen because of its ease of use and known results on the currently available custom neural network implementation. [6]

Once the dataset was selected it was analyzed with the original custom neural network implementation using the built-in `optim` function with the conjugate gradient method. The dataset was also analyzed with the built-in neural network implementation. The results were compared and noted to be similar but different. At this point it was required that the experiment be repeated many more times to determine which function worked better with the default settings. Some additional parameter settings, that were common between the two implementations were used to determine the better overall function.

Data was gathered throughout the process for presentation including the runtime of each method with a default parameter set and identical hidden units count. Additionally, the results of classification were gathered and discussed. Finally, it was also planned to gather the number of epochs required.

After analyzing the performance of the built-in `nnet` function, the parameters available were analyzed and compared with the custom neural network implementation. The number of parameters for each implementation was determined, and any matching parameters were identified.

Finally the source code implementation of the built-in `nnet` implementation was analyzed. This includes determining how the implementation is expected to work when compared with the custom implementation in R. It should be noted that the built-in `nnet` implementation is written using C, and the custom implementation created throughout the course is written in R, making a direct source code comparison not possible, so this comparison will be done a higher level discussing the basic high level algorithm used only.

### 4 Performance Analysis

The wine data set was obtained from UCI and it was randomized, standardized, and indicator variables were created. [6] This was to prepare the data for use in the neural network code. Through this process several variables for accessing the data were created, and all input data was given standard labels to maximize code reuse in the future with various datasets.

```
library(nnet)
source("mlUtilities.R")
source("nnUtils.R")
```

```
# This is a function that creates a table of values and was
# written by Brian Ripley as part of the 'nnet' package documentation
# It is used here as a utility to display the results as a table.
```

```
test.cl <- function(true, pred)
{
  true <- max.col(true)
```

```

    cres <- max.col(pred)
    table(true, cres)
}

# read the data
data <- read.table("wine.data", sep=" ",)

# label the data
headings <- c("Class", "Alcohol", "Malic Acid", "Ash", "Alcalinity of Ash",
             "Magnesium", "Total phenols", "Flavanoids",
             "Nonflavanoid phenols", "Proanthocyanins", "Color Intensity",
             "Hue", "OD280/OD315 of diluted wines",
             "Proline")
colnames(data) <- headings

### Convert to numbers and remove all samples that have at least one "?"
data <- apply(data, 2, as.numeric)
keepRows <- apply(data != "?", 1, all)
data <- data[keepRows,]

# Set the training fraction to 70%(training)/30%(test)
trainf <- 0.70

# Seperate out the class
class <- data[, "Class"]
data <- data[, 2:14 ]

class1Data <- data[class==1,]
class2Data <- data[class==2,]
class3Data <- data[class==3,]

nClass1Data <- nrow(class1Data)
nClass2Data <- nrow(class2Data)
nClass3Data <- nrow(class3Data)

### Randomly pick samples for training partition
randorder1 <- sample(nClass1Data)
randorder2 <- sample(nClass2Data)
randorder3 <- sample(nClass3Data)

# Divide the data up into trainf training data and 1-trainf testing data
class1TrainDiv <- round(floor(trainf*nClass1Data))
class2TrainDiv <- round(floor(trainf*nClass2Data))
class3TrainDiv <- round(floor(trainf*nClass3Data))

class1TrainRows <- randorder1[1:class1TrainDiv]
class2TrainRows <- randorder2[2:class2TrainDiv]
class3TrainRows <- randorder3[3:class3TrainDiv]
class1TestRows <- randorder1[(class1TrainDiv+1):length(randorder1)]
class2TestRows <- randorder2[(class2TrainDiv+1):length(randorder2)]
class3TestRows <- randorder3[(class3TrainDiv+1):length(randorder3)]

```

```

# Assign the test and training data
Xtrain <- rbind(class1Data[class1TrainRows,],
               class2Data[class2TrainRows,],
               class3Data[class3TrainRows,])
Ttrain <- matrix(c(rep(1,length(class1TrainRows)),
                  rep(2,length(class2TrainRows)),
                  rep(3,length(class3TrainRows))))
Xtest <- rbind(class1Data[class1TestRows,],
               class2Data[class2TestRows,],
               class3Data[class3TestRows,])
Ttest <- matrix(c(rep(1,length(class1TestRows)),
                  rep(2,length(class2TestRows)),
                  rep(3,length(class3TestRows))))

```

```

# Standardize the data
standardize <- makeStandardizeF(Xtrain)
Xtrains <- standardize(Xtrain)
Xtests <- standardize(Xtest)

```

```

# Make indicator variables
TtrainI <- makeIndicatorVars(Ttrain)
TtestI <- makeIndicatorVars(Ttest)

```

The first performance measurement performed was a runtime performance measurement. To use the built-in neural network implementation in `nnet` some default parameters were used. These were observed to require the `rang` field to be set to 0.01 to match the hard-coded value within the custom neural network code. It was also required that the hidden units and maximum number of iterations be set to the same values that were being passed into the custom `nnet` function.

The runtime of both algorithms was measured to compare which one seemed to execute the fastest with various iteration counts. Using the default settings for both, it was found that the built-in neural net function was taking a much lower amount of time to execute when compared with the custom implementation. The code used to take these measurements can be found below.

```

assortedIterationsCnt <- c(200,2000,20000,200000,2000000)
customTimes <- c()
builtinTimes <- c()
repeatCnt <- 10

for (tempReps in assortedIterationsCnt)
{
  tempTimeCust <- 0
  tempTimeBuiltin <- 0

  for (tempval in 1:repeatCnt)
  {
    tempTimeCust <- tempTimeCust + system.time(nnet_original <-
                                                makeNNOptim(Xtrain, TtrainI, hiddenUnits, lambda,
                                                            nIterations=tempReps, method="BFGS"))[3]
    tempTimeBuiltin <- tempTimeBuiltin+ system.time(nnet_builtin <-
                                                    nnet(Xtrains, TtrainI, size=hiddenUnits,
                                                        maxit=tempReps, rang=0.01))[3]
  }
}

```

Table 1: Iteration Count vs. Execution Time (seconds)

| Iterations | Custom | Builtin |
|------------|--------|---------|
| 200        | 1.0229 | 0.1715  |
| 2000       | 4.1061 | 0.1581  |
| 20000      | 2.4384 | 0.1511  |
| 200000     | 1.6544 | 0.2059  |
| 2000000    | 1.9264 | 0.1791  |

```

customTimes <- rbind(customTimes, (tempTimeCust/repeatCnt))
builtinTimes <- rbind(builtinTimes, (tempTimeBuiltin/repeatCnt))
}

pdf("timedata.pdf")
ptemp <- par(bty="n")

legendTags <- c("Custom nnet", "Builtin nnet")
legendColors <- c("orange", "maroon")
matplot(assortedIterationsCnt, customTimes, pch=1, lty=1,
        xlab="Number of Iterations", ylab="Elapsed Time",
        col=legendColors[1], type='l',
        main="Time Consumed by Creating nnet")
matlines(assortedIterationsCnt, builtinTimes, type='l', pch=1, lty=1,
        col=legendColors[2])
legend(x="topleft", legend = legendTags, col=legendColors, lty=1)

dummy <- dev.off()
par(ptemp)

```

In table 1 the raw time data from execution time can be observed. This data was used to create the graph in figure 1. In this chart you can see that the custom function does not appear to take a linear increase in execution time with respect to the iteration count, and it also does not appear to have much of an effect on the execution time of the built in function. One important observation about this data is that the execution time for the built-in `nnet` function with the same default parameters is significantly better than the execution time for the custom function. That is likely because the built in function is implemented in C, providing a more efficient implementation for the underlying algorithm. It is also likely that another, hard-coded and hidden setting could be impacting this, and the full number of iterations never needs to execute because of this difference. It therefore appears that the builtin neural network code will excute much faster and thus has the better performance.

After the execution time was measured, the performance of the two methods themselves were observed. The wine data set was used to generate a neural network, and then that neural network was used to make predictions on some test data. The predictions generated by both neural networks were compared to determine which implementation performed better. The code to create the graphs displayed in figure 2, figure 3, and figure 4 can be found below.

```

# Train the neural network - Our implementation
hiddenUnits <- 10
lambda <- 0
nReps <- 200000

timeCust <- system.time(nnet_original <- makeNNoptim(Xtrain, TtrainI,
            hiddenUnits, lambda, nIterations=nReps, method="BFGS"))

```

### Time Consumed by Creating nnet

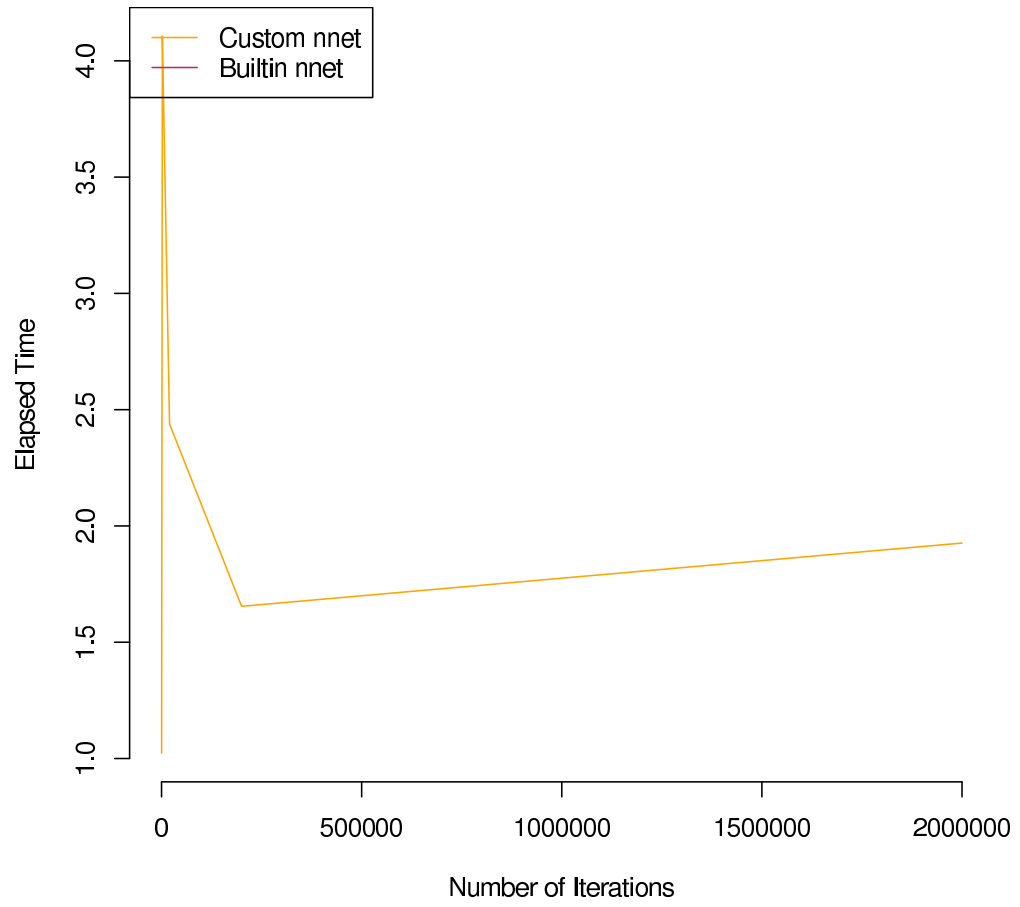


Figure 1: Parameter Status After Training Phase

```

custNNResults <- useNN(nnet_original, Xtrain)
custNNResultsTest <- useNN(nnet_original, Xtest)

# It was discovered during implementation that the nnet
# function that is built in
# does not standardize the data automatically. Our custom function does.
timeBuiltIn <- system.time(nnet_builtin <- nnet(Xtrains, TtrainI,
      size=hiddenUnits, maxit=nReps, rang=0.01))
builtinNNResults <- predict(nnet_builtin, Xtrains)
builtinNNResultsTest <- predict(nnet_builtin, Xtests)

print(paste("Custom Runtime was: ", timeCust[3]))
print(paste("Built-in Runtime was: ", timeBuiltIn[3]))

# There is now some accuracy data available, compile it
# Ttrain and Ttest are both the value expected responses
trainResultsCustom <- max.col(custNNResults)
trainResultsBuiltin <- max.col(builtinNNResults)

testResultsCustom <- max.col(custNNResultsTest)
testResultsBuiltin <- max.col(builtinNNResultsTest)

pdf("custNerualNet.pdf")
ptemp <- par(mfrow=c(2,2), bty="n")

legendTags <- c("Class 1", "Class 2", "Class 3")
legendColors <- c("red", "green", "blue")

# 1. the training data, as its x value versus the class (1, 2, or 3)
matplot(1:length(Ttrain), Ttrain, pch=1, lty=1,
  xlab="X Value of Training Data", ylab="Class",
  main="1. Training Data")

# 2. the three curves for  $p(C=k | x)$  for  $k=1,2$ , and  $3$ , for  $x$  values in a set
# of test data
matplot(1:length(Ttrain), custNNResults[,1], type='l', pch=1, lty=1, col="red",
  xlab="X Value of Test Data", ylab="Probability",
  main="2.  $p(C=k | x)$  for  $k=1,2$ , and  $3$ ")
matlines(1:length(Ttrain), custNNResults[,2], type='l', pch=1, lty=1, col="green")
matlines(1:length(Ttrain), custNNResults[,3], type='l', pch=1, lty=1, col="blue")

# 3. Plot the class predicted by the Logistic
# Regression classifier for the test data
legendTags <- c("Predicted Classes", "Actual Classes")
legendColors <- c("red", "green", "blue")
matplot(1:length(Ttest), testResultsCustom, pch=1, lty=1,
  xlab="X Value of Test Data", ylab="Class",
  col=legendColors[1],
  main="3. Test Data Class Predictions")
matlines(1:length(Ttest), Ttest, type='l', pch=1, lty=1,
  col=legendColors[2])
legend(x="topleft", legend = legendTags, col=legendColors, lty=1)

```

```

#dummy <- dev.off()
par(ptemp)

pdf("builtinNerualNet.pdf")
ptemp <- par(mfrow=c(2,2), bty="n")

legendTags <- c("Class 1", "Class 2", "Class 3")
legendColors <- c("red", "green", "blue")

# 1. the training data, as its x value versus the class (1, 2, or 3)
matplot(1:length(Ttrain), Ttrain, pch=1, lty=1,
        xlab="X Value of Training Data", ylab="Class",
        main="1. Training Data")

# 2. the three curves for  $p(C=k | x)$  for  $k=1,2$ , and  $3$ , for  $x$  values in a set
# of test data
matplot(1:length(Ttrain), builtinNNResults[,1], type='l', pch=1,
        lty=1, col="red",
        xlab="X Value of Test Data", ylab="Probability",
        main="2.  $p(C=k | x)$  for  $k=1,2$ , and  $3$ ")
matlines(1:length(Ttrain), builtinNNResults[,2], type='l',
        pch=1, lty=1, col="green")
matlines(1:length(Ttrain), builtinNNResults[,3], type='l',
        pch=1, lty=1, col="blue")

# 3. Plot the class predicted by the Logistic
# Regression classifier for the test data
legendTags <- c("Predicted Classes", "Actual Classes")
legendColors <- c("red", "green", "blue")
matplot(1:length(Ttest), testResultsBuiltin, pch=1, lty=1,
        xlab="X Value of Test Data", ylab="Class",
        col=legendColors[1],
        main="3. Test Data Class Predictions")
matlines(1:length(Ttest), Ttest, type='l', pch=1, lty=1,
        col=legendColors[2])
legend(x="topleft", legend = legendTags, col=legendColors, lty=1)

#dummy <- dev.off()
par(ptemp)

pdf("compareNNetImpl.pdf")
ptemp <- par(bty="n")

legendTags <- c("Class 1", "Class 2", "Class 3")
legendColors <- c("red", "green", "blue")

# 3. Plot the class predicted by the Logistic Regression classifier
# for the test data
legendTags <- c("Actual Data", "Built In nnet", "Custom nnet")

```

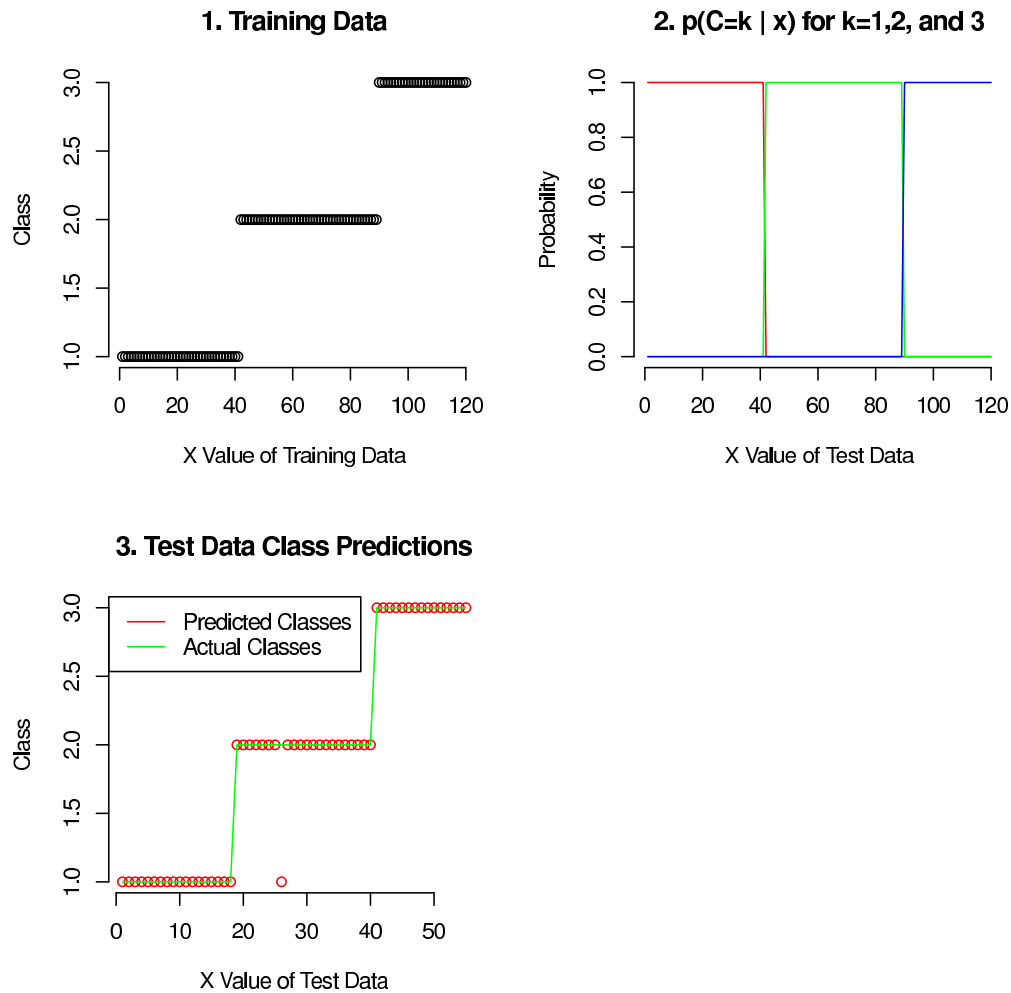


Figure 2: Custom Neural Network Predictions

```

legendColors <- c("red", "green", "blue")
matplot(1:length(Ttest), Ttest, pch=1, lty=1, type='p',
        xlab="X Value of Test Data", ylab="Class",
        col=legendColors[1],
        main="3. Test Data Class Predictions")
matlines(1:length(Ttest), testResultsBuiltin, type='l', pch=1, lty=1,
        col=legendColors[2])
matlines(1:length(Ttest), testResultsCustom, type='l', pch=1, lty=1,
        col=legendColors[3])
legend(x="topleft", legend = legendTags, col=legendColors, lty=1)

dummy <- dev.off()
par(ptime)

```

From looking at the graphs in figure 2 and figure 3 it is clear that in both cases the neural networks will accurately create a classification of test data. This can be seen for an easy comparison in figure 4. It looks like both implementations create an equally accurate classifier for the wine data.

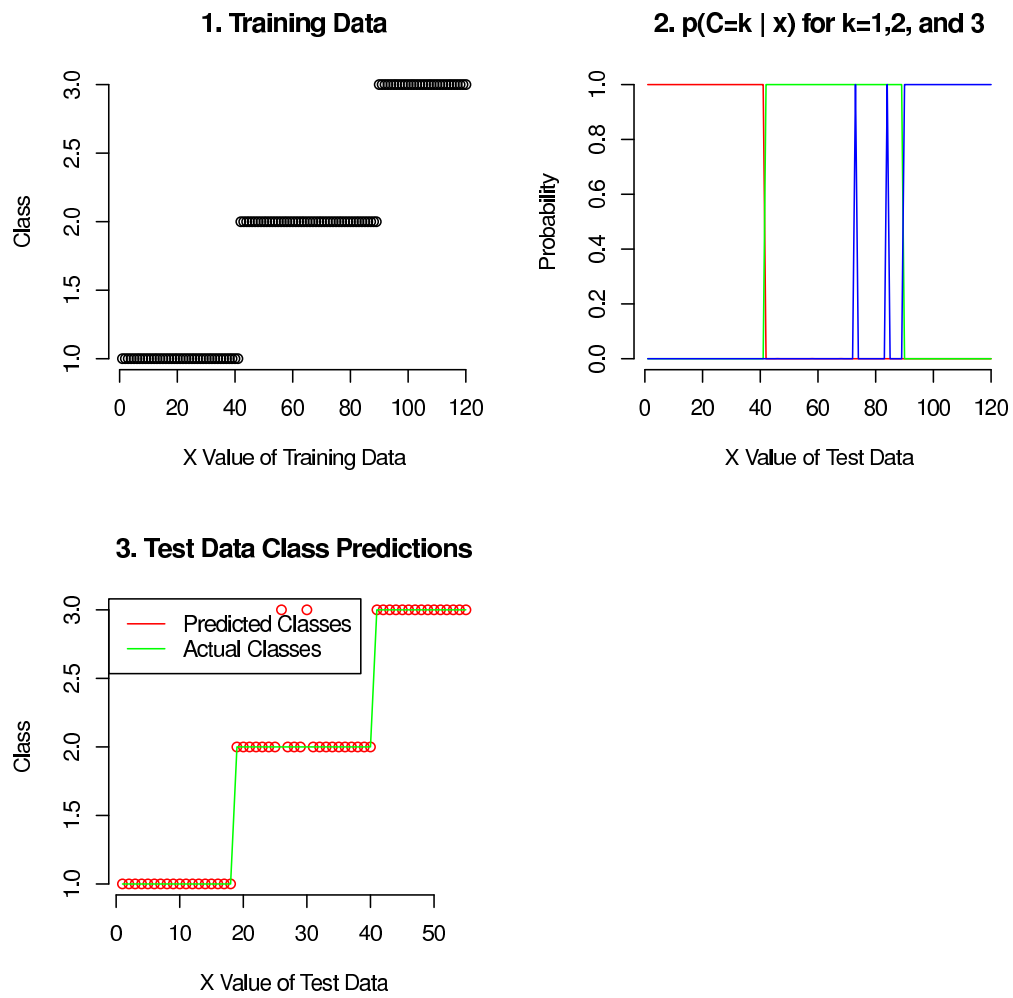


Figure 3: Built-in Neural Network Predictions

### 3. Test Data Class Predictions

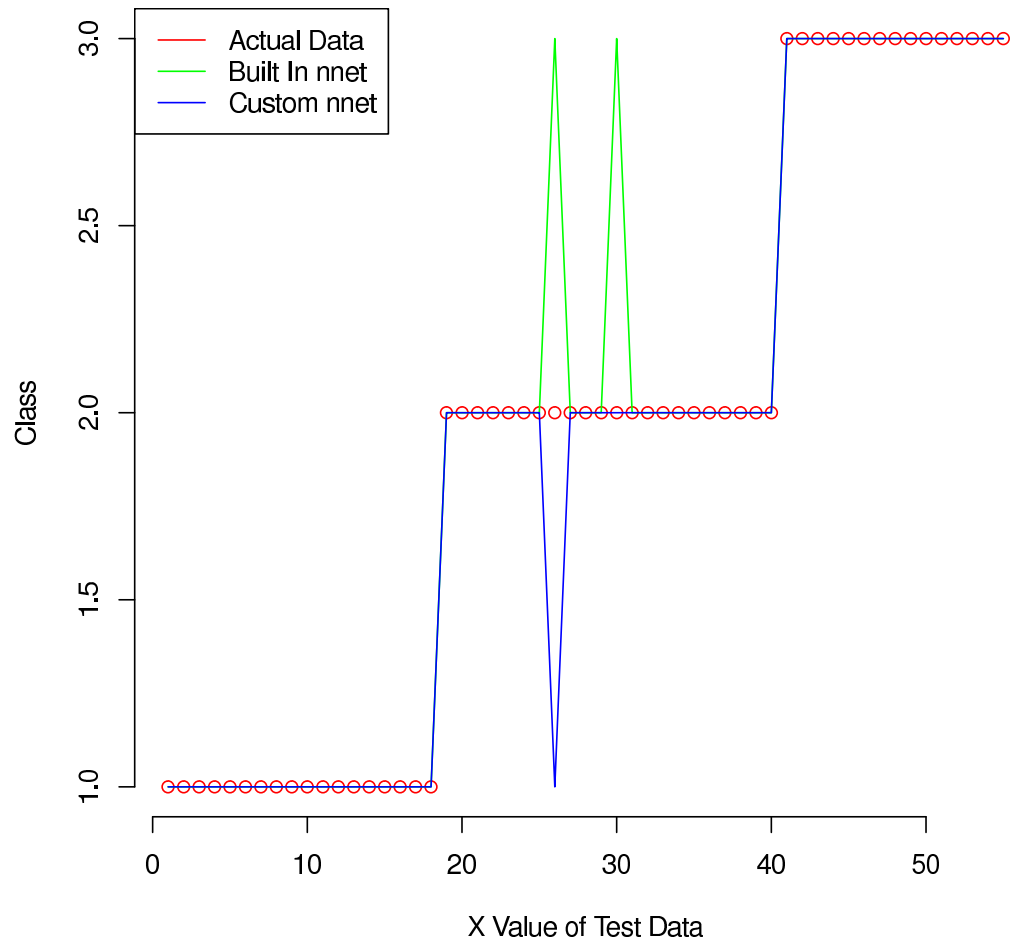


Figure 4: Compare Custom and Builtin

Table 2: Parameters for `nnet`

| Builtin   | Custom         | Comments   |
|-----------|----------------|--|
| formula   |                | Builtin <code>nnet</code> supports a formula or X and Y variables.                     |
| x         | Xtrain         |  |
| y         | Ttrain         |  |
| weights   | -              | In our implementation this is hard-coded.  |
| size      | numHiddenUnits |  |
| data      | -              | Used with formula above in built-in function only.                                     |
| subset    | -              | Select a subset of the input data to train with.                                       |
| na.action | -              | Will allow the user to specify if NAs in data should be ignored.                       |
| contrasts | -              |  |
| Wts       | -              | Optionally initialize the initial weights, hard coded to random in our implementation. |
| mask      | -              | Mask to selectively optimize only certain parameters.                                  |
| linout    | -              | Part 1 of 4 of a selection algorithm (may only choose 1.)                              |
| entropy   | -              | Part 2 of 4 of a selection algorithm (may only choose 1.)                              |
| softmax   | -              | Part 3 of 4 of a selection algorithm (may only choose 1.)                              |
| censored  | -              | Part 4 of 4 of a selection algorithm (may only choose 1.)                              |
| skip      | -              |  |
| rang      | -              | This is hard-coded to 0.01 in our implementation.                                      |
| decay     | lambda         | This is the weight penalty in our implementation.                                      |
| maxit     | nIterations    |  |
| Hess      | -              |  |
| trace     | -              |  |
| MaxNWts   | -              |  |
| abstol    | -              | This is similar to the custom precision fields, but combined.                          |
| reitol    | -              | This is useful to stop the optimizer sooner than normal.                               |
| -         | xPrecision     |  |
| -         | fPrecision     |  |
| -         | method         | The method in the builtin function is hard-coded to “BFGS.”                            |

After looking at the classification accuracy it was decided that we would take a look at the number of epochs required for both the custom neural network implementation and the builtin implementation. Unfortunately it turns out that the epoch values of the builtin function are not openly available through the R interface. To get these values would require modification of the C code that is used to implement the functionality. Without being able to compare the number of epochs from both, it was decided that this method of evaluation would not be used.

## 5 Analysis of Parameters

There are about twenty four parameters to the built in neural network function, and our custom R function only has eight. Most of these eight parameters have a directly related parameter in the built in neural network function. This leaves about sixteen parameters that do not have equivalents in our custom implementation that also exist in the built in neural network function. In table 2 the `nnet` function parameter, the custom neural network function parameters, and the description are listed. [2]

## 6 Analysis of Implementations

One major limitation with the built in neural network function is that it uses the builtin `optim` function but can only use the “BFGS” method within that function.[5] The custom neural network function that I was

using to compare, could access any method available on the `optim` function for this purpose.

Another implementation detail is that the built in `nnet`, like our custom implementation, only supports a single hidden layer. The implementation supports “Feed-forward Neural Networks and Multinomial Log-Linear Models.”[2]

After downloading the code for the neural network implementation it was determined that the implementation style and comments were far better in the custom version used in class. The custom implementation used in class was based in R, which is already more readable than native C code. The built in function was highly optimized C code, which unfortunately had very few comments throughout the file. There were about six hundred lines of code, and six comments within that code. Due to this limitation, for educational purposes the custom neural network code is much more user friendly and easier for a student to understand.

After some early issues with the implementations used for performance evaluation of the builtin code, it was found that the native `nnet` function does not implement a standardization of the data. In the custom implementation the neural network function will standardize the data, but in the built in function this was not the case. This led to unpredictable results, and took some time to find the root cause because it was assumed initially that the built in function would also standardize the input data. Once standardized data was used, the output data was much more accurate.

## 7 Discussion

This was a very interesting project for me to complete. I needed to learn how functions are natively implemented in R. I also learned a great deal about the built in `nnet` function, which implements most, but not all, of the features that our custom implementation had. It seems that our custom implementation would be much easier to troubleshoot in the future, but the built-in function will always be around without needing to search for our existing toolbox of functionality.

Going forward it would be interesting to add the functionality that our custom implementation had that the builtin version was lacking. This may be difficult for someone other than the original author however, because the existing implementation of the built in function is about fifteen thousand lines of C with little or no comments. If a new project was to be started to perform this task, significant time would need to be spent understanding the existing code, and also checking that current functionality was not changed in any way.

## 8 Conclusions

During this assignment it was found that the built in neural network function will be able to satisfy a majority of our needs for this type of function. There are a couple of features that are not present in the current built in implementation, and which due to several factors, will probably not be included anytime in the near future. Most of the functionality that is missing was useful for debugging the custom implementation itself, but some was a good way to fine tune the results.

Overall the built in function written in C executes much faster than the custom version written in R with very similar results. This code will exist on all R installations, since it is part of the standard R libraries. There are also a number of options that were not available in the custom implementation. The custom implementation has a lot of value as a learning tool, but if performance is critical, the built in neural network function should be used.

## References

- [1] Bishop, C., *Pattern Recognition and Machine Learning*, Springer Science, 2006.
- [2] Ripley, B., *Feed-forward Neural Networks and Multinomial Log-Linear Models*, <http://cran.r-project.org/web/packages/nnet/nnet.pdf>, 2009.
- [3] Anderson, C., *Lecture Notes 1 through 30, 12/10/2009*, <http://www.cs.colostate.edu/~anderson/cs545/>, 2009.

- [4] Anderson, C., *Homework Assignment 8, 12/16/2009*, <http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment6.html>, 2009.
- [5] Venables, W.N., and Ripley, B.D., *Modern Applied Statistics with S*, Springer Science, 2002.
- [6] Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository <http://www.ics.uci.edu/~mllearn/MLRepository.html>. Irvine, CA: University of California, School of Information and Computer Science.