Case Studies using the R Programming Language

Katherine Bennett, Miguel de los Reyes, Raymond Gao,

Sophia Hurr, Hannah Gahagan, Nikhil Miland, Mridu Nanda,

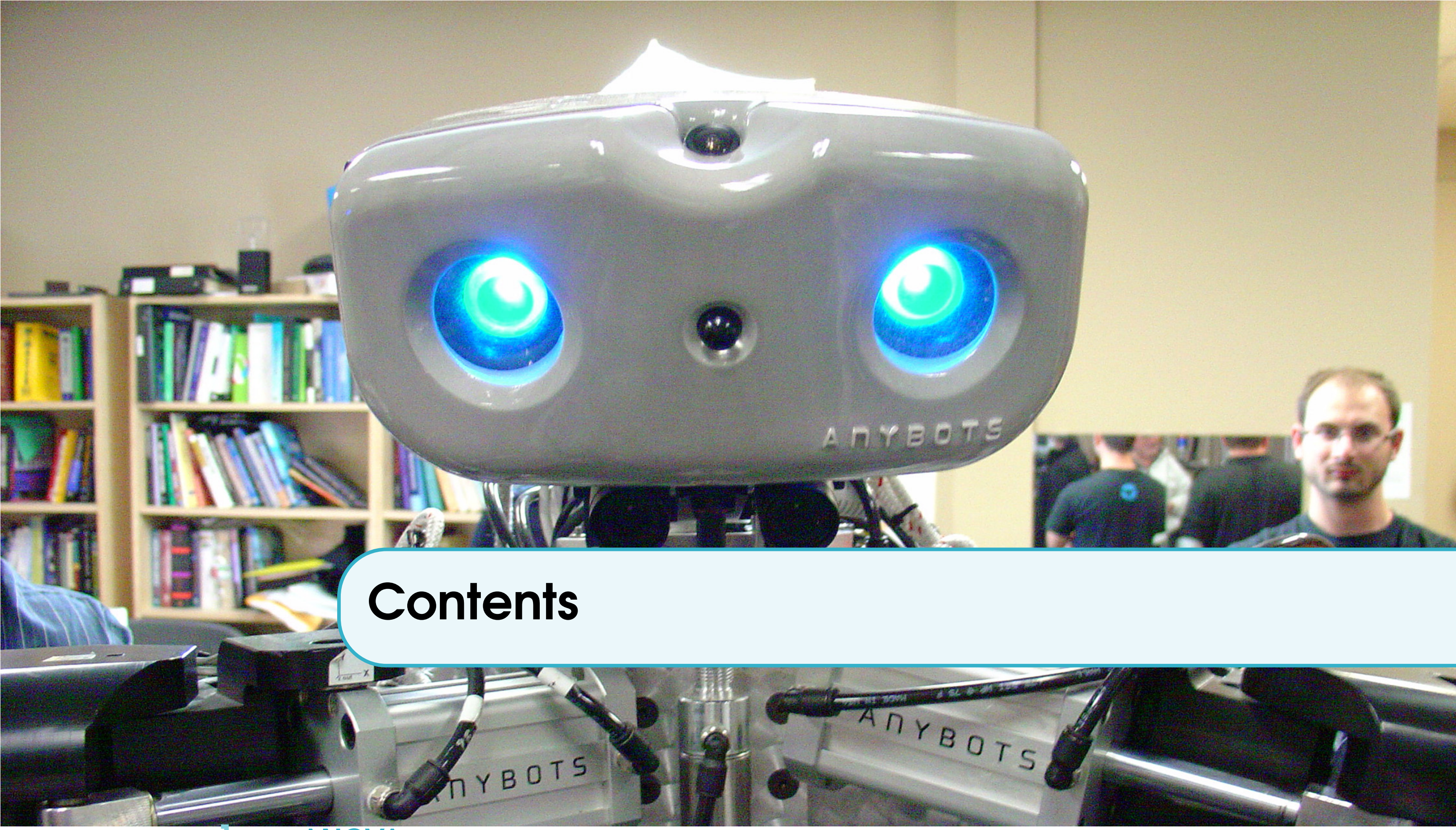
Christa Parrish, Ishaan Rao, Grayson York

RESEARCH IN COMPUTATIONAL SCIENCE

NORTH CAROLINA SCHOOL OF SCIENCE AND MATHEMATICS, DURHAM, NC

This case studies manual was prepared under the direction of Mr. Robert Gotwals, faculty mentor, The North Carolina School of Science and Mathematics, Durham NC.

First release, November 2016



Contents

Introduction

Gotwals will write an Intro.

1. ANOVA

1.1 Introductory Reading

The most common type of linear models are analysis of variance (ANOVA) and linear regression models. R is built to handle linear models, and as such it is easy to work with ANOVA. ANOVA is a statistical method used to compare two or more means. These values can be used to determine whether a significant correlation exists between variables. We will be using one-way ANOVA, which compares the means between the groups of interest, and determines whether those means are significantly different from each other. If one-way ANOVA returns a significant result, there are at least two group means that are significantly different from each other. It is important to note that ANOVA is an omnibus test statistic, meaning that although ANOVA tells the user is there is a significant difference in means, it is unable to show which means in particular differ. ANOVA can only be used with certain types of data configurations. To perform ANOVA, data must have a continuous response variable and at least one categorical factor with two or more levels, in lay terms, ANOVA can only be used with numeric values that can be ordered sequentially, with a certain number of possible responses. (i.e. data comparing object's weights, and each new weight is a level) ANOVA is easier to use if data is from from approximately normally distributed populations with equal variances between factor levels, however, as R is built to work with statistics, ANOVA procedures will generally work without incident unless one or more of the distributions or variances are highly skewed. A basic understanding of statistics how to use R is recommended before performing ANOVA. The ANOVA user will create and order factors, make box plots, and combine and stack data. [SL07] The following functions will be useful in using ANOVA.

1. `aov`
2. `as.factor`
3. `ls.str`
4. `data.frame`
5. `stack`

6. TukeyHSD
7. levels

1.2 Objectives

In this assignment, we will use ANOVA to analyze data from ELISA HIV Optical Density Readings. ANOVA identifies the causes of variation, and sorts out the corresponding components of variation with associated degrees of freedom. In this case, we are interested in seeing how HIV Optical Density Readings are related to their lot. The type of ANOVA we are using is one-way between groups, as we are comparing one grouping (the optical density readings) to define the groups (lots).

By the end of this lesson, the reader should be able to:

1. Understand the logic behind one-way analysis of variance.
2. Perform one-way analysis of variance in R for any data.
3. Appropriately interpret results of analysis of variance tests.

1.2.1 ANOVA

We will look at variances in data using R's ANOVA functions.

1.2.2 Visualization

We will also use our fitted models and our ANOVA data to create box plots and line plots. Much of the information gleaned from ANOVA is presented via numeric such as the sum of square, degrees of freedom, and the mean. ANOVA presents the null hypothesis that there is no difference in means of the treatments, and once this hypothesis is proven incorrect, the question arises of how the treatments differ. The post-hoc test also allows the user to find the differences in means, and specifically categorizes the lower and upper means of the data.

Figure 1.1 shows a box plot of the data before running ANOVA or TukeyHSD. Constructing a box plot before analyzing the data may prove helpful in deciding what kind of analysis would be preferable.

1.3 Building the Model

Looking at the data set of interest for ANOVA, note that that data may be in numeric form. Use the function is.numeric to review the data. If this function prints TRUE, use the function as.factor to change data from numeric to factor. In order to make it easier to manipulate data later on, it is recommended that the factored data is renamed.

```
lot = as.factor(elisa$Lot)
summary(lot)
levels(as.factor(elisaOptical))
optical = (as.factor(elisa$Optical))
levels(as.factor(elisa$Run))
run = (as.factor(elisa$Run))
```

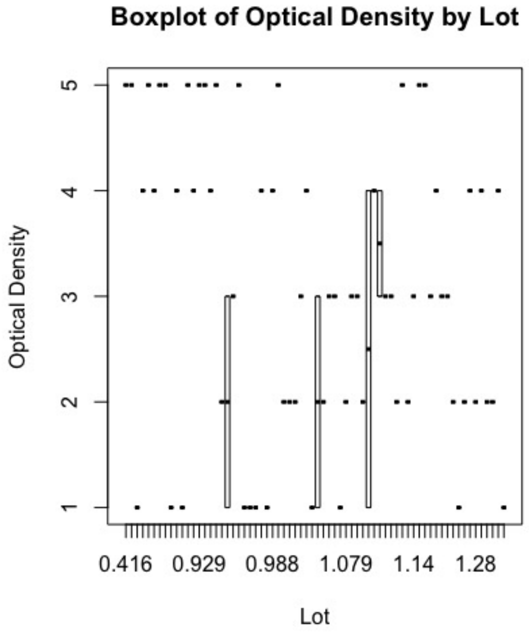


Figure 1.1: Boxplot

Now your factor data type is in non-numerical variables. Each different variable of the factor is called a level. For example, our factor type data for the lots of various ELISA HIV Optical Density Readings has five levels, 1, 2, 3, 4 and 5. Moving on, we can make a box plot of our data, so that we can visually compare data. Making a summary of this data allows us to look at the residuals and standard error of the plot. As mentioned before, this type of comparison will work better with data that has a relatively normal distribution. Factors can also be created within factors. We used our factor Optical to categorize and label by the density. For each level within the main factor, create labels. These labels will designate unordered factors. To make this data easier to read, unordered factors should be ordered. Choose ways to order the data that is relative to the averages in the data. For example, some levels were Low density(below 1) and some were High density,(above 1.5) so we ordered the data starting at 0, then continuing with 1, 1.5, 2. Choose labels that reflect the data, noting that the first label will relate to the lower numbers. Next, classify this newly ordered factor, and now label the data in lay terms.

```
elisa$Optical.type<-ordered(cut(elisa$Optical , c(0,1,1.5,2), labels = c("Low","M
class(elisa$Optical.type)
elisa$Optical.type
```

Note, that if you wish to remove one group of subjects (i.e. Lot 1) R will keep this removed group as a level, possibly skewing your data. Use the function drop levels to remove a selected group as a level. Rename this reconfigured data as to not overwrite your previous data. In order to use ANOVA, the data must be in a specific format. To properly configure data, combine the data from the two factors intended to be compared with the function data.frame. Next, stack these combined groups.

Finally use this stacked data and the function aov to perform ANOVA. Make a summary of these results.

```r
finalelisa <- droplevels(newelisa)
summary(finalelisa$Lot)
Combined_Groups <- data.frame(cbind(lot, optical))
Combined_Groups
Stacked_Groups <- stack(Combined_Groups)
Stacked_Groups
Anova_Results <- aov(values ~ ind, data = Stacked_Groups)
```

This method can me used for a simple one-way ANOVA. However some data can compare multiple sets of data against each other. To do this, we can use a post-hoc test. Post-hoc tests compare outcome measurements between multiple groups. With post-hoc analysis the reader can examine differences between pairs of groups after global analysis. Use the function TukeyHSD to perform a pairwise post-hoc analysis on the ANOVA results.

```r
summary(Anova_Results)
TukeyHSD(Anova_Results)
summary(TukeyHSD(Anova_Results))
```

1.3.1 Programming Hints

It is recommended to look at data before analysis in order to better understand the levels and attributes of data. In addition, we recommend the use of `summary()` and the console to view the attributes of objects.

1.4 Deliverable

Using ANOVA, students should be able to take data with sequentially ordered number data, make a box plot, factor the data as unordered and ordered factors, configure the data such that the function aov can be performed, and perform TukeyHSD. Using the box plot, students should be able to identify residuals, coefficients of the linear regression, and residual standard error. Using aov, students should be able to identify degrees of freedom, the sum of the squares, the mean of the squares, and the F ratio (the ratio of two mean square values) for their data.

1.5 Teaching Code

Begin ANOVA by formatting the data in a `.csv` file, then uploading it to RStudio. This script is set up to analyze publicly available data about HIV Optical density readings, but can be adapted for any properly set up data. As the reader works his or her way through the script, be sure to liberally comment the purpose of each command.

```r
# Your name here
# Date
# ANOVA - ELISA HIV Optical density readings data

# Clean up and read in data
```

```
rm(list=ls())
setwd("C:/Your/Directory")
elisa = read.csv("ELISAHIV.csv")

# Create levels for lot, optical, and run
# Make sure to view/check your data
levels(as.factor(elisa$Lot))
lot=as.factor(elisa$Lot)
# etc.


# Create a boxplot showing optical density by lot
# Make sure to label your axes!
boxplot(..., ...)


# Perform a linear regression of optical density by lot
summary(lm(..., data=elisa))

# Create an ordered factor using Optical.type and verify its type
elisa$Optical.type<-ordered(...)
class(elisa$Optical.type)

# Remove 1 as a possible level
# Note: even if you eliminate a group of subjects (ex. 1),
# because it's a factor, R keeps 1 as a possible level for lot

# Use droplevels() to remove 1

# Create stacked results to run aov()

# Run TukeyHSD on the ANOVA results
```

1.6 Example Student Code

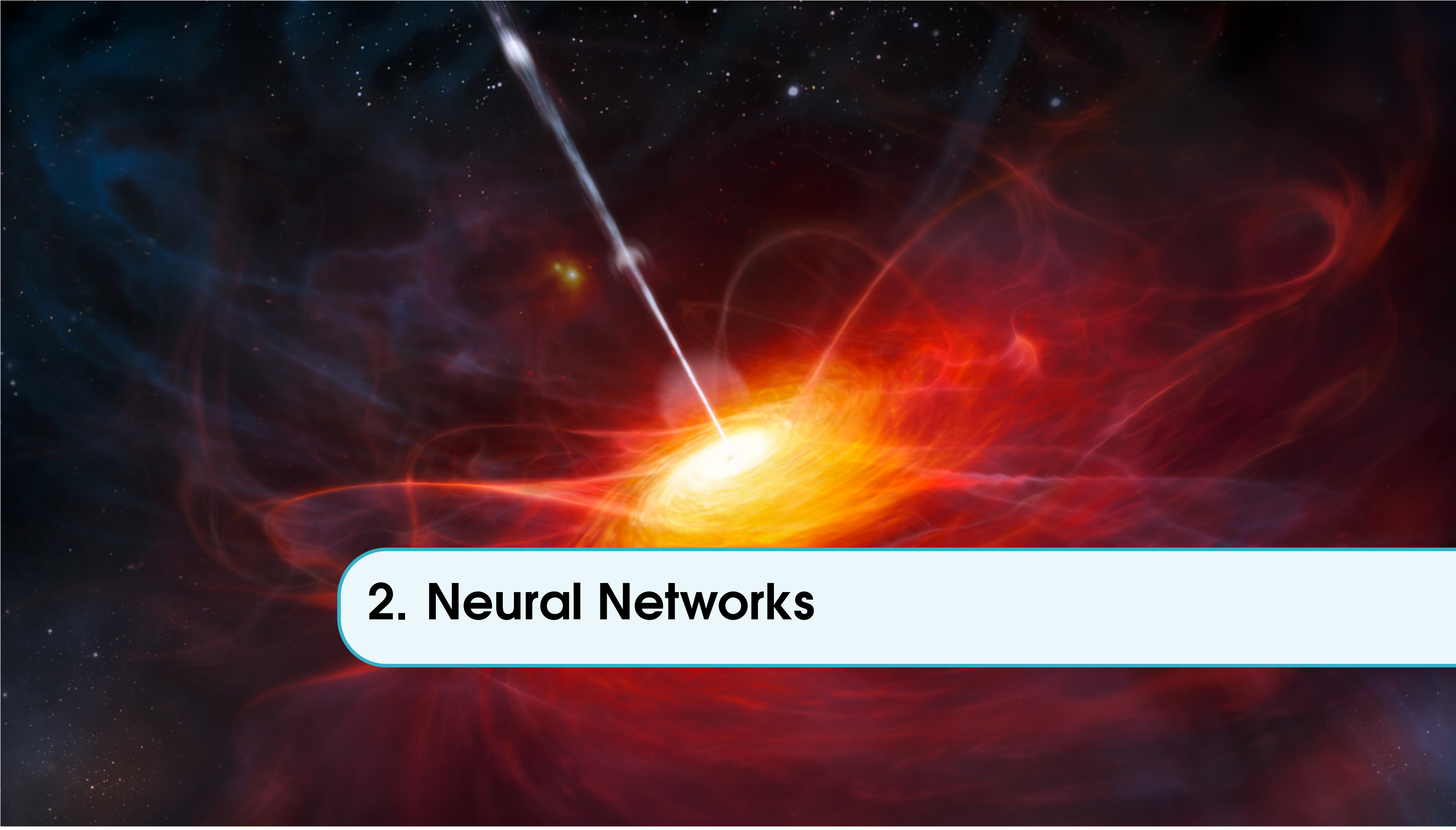
```
# KEY
# ANOVA - ELISA HIV Optical density readings data

# Clean up and read in data
rm(list=ls())
setwd("C:/Example/Code")
elisa = read.csv("ELISAHIV.csv")

# Look at variables and create levels
ls.str(elisa)
levels(as.factor(elisa$Lot))
lot=as.factor(elisa$Lot)
summary(lot)
levels(as.factor(elisa$Optical))
optical = (as.factor(elisa$Optical))
levels(as.factor(elisa$Run))
run = (as.factor(elisa$Run))

```

```
# Create a boxplot
boxplot(elisa$Lot~elisa$Optical, xlab="Lot", ylab="Optical Density",
    main="Boxplot of Optical Density by Lot")
summary(lm(elisa$Optical~elisa$Lot, data=elisa))

# Perform regression
summary(optical)
elisa$Optical.type<-ordered(cut(elisa$Optical, c(0,1,1.5,2),
    labels=c("Low","Medium","High")))
class(elisa$Optical.type)
elisa$Optical.type

# Remove 1 as a level
# We've verified that optical is an ordered factor
# Note: even if you eliminate a group of subjects (ex. 1),
# because it's a factor, R keeps 1 as a possible level for lot
newelisa <- elisa[1:2,]
summary(newelisa$Lot)
# We remove 1 as a possible level using droplevels()
finalelisa <-droplevels(newelisa)

# We create stacked results to run aov()
summary(finalelisa$Lot)
Combined_Groups <- data.frame(cbind(lot,optical))
Combined_Groups
Stacked_Groups <- stack(Combined_Groups)
Stacked_Groups
Anova_Results <- aov(values ~ ind, data = Stacked_Groups)
summary(Anova_Results)

# We also run TukeyHSD on the ANOVA results
TukeyHSD(Anova_Results)
summary(TukeyHSD(Anova_Results))
```

1.7 Further Readings

1. Seefeld, Kim and Linder, Ernst. *Statistics Using R with Biological Examples*, University of New Hampshire, Durham, NH Department of Mathematics and Statistics(2007)

2. Julian J. Faraway. *Practical Regression and Anova using R*, University of Michigan, (2002)

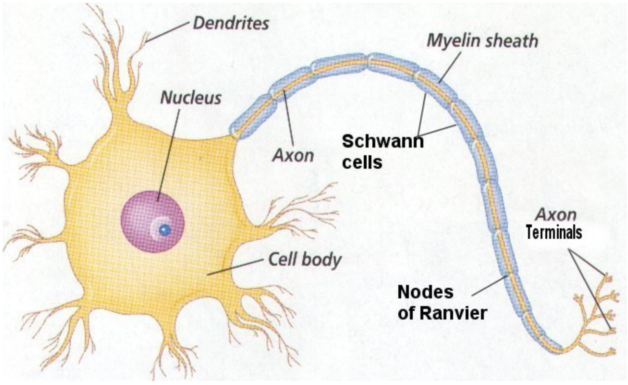


2. Neural Networks

2.1 Introduction to Neural Networks

In the field of computational research, the use of computers is generally restricted to discreet, exact commands that are meant to be executed in a certain manner, which provide the computational power required for complex calculations. In contrast, neural networks are an example of machine learning (ML), wherein the machine is capable of learning behavior or interpreting patterns from a given data set. The field of predictive analysis uses this paradigm of computational interpretation. For example, it is extremely challenging to develop a system that is purely programmed to recognize handwriting. Yet, computers at the United States Postal Service are able to decipher human words and reroute letters and packages without the need for humans. These computers contain a neural network that has been trained to recognize handwriting and predict characters as they appear on mail. [Nie15]

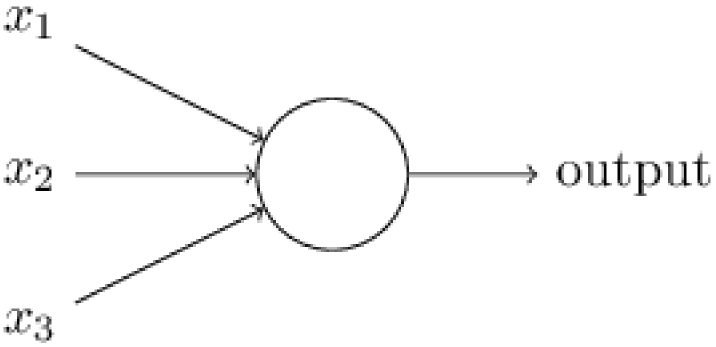
Figure 2.1: The Parts of an Anatomical Neuron



Neural networks were designed using the human brain as an inspiration. The neural network is composed of perceptrons, which are similar to the smallest functional units of the brain: the neuron.

Neurons act as the wiring of the brain by passing along signals that they receive to other neurons. When a neuron communicates with another neuron, it uses neurotransmitters to either increase or decrease the probability of the next neuron firing. Thus, a neuron can be excitatory (increases the chance of the next neuron firing) or inhibitory (decreases the chance of the next neuron firing). Similarly, neural networks utilize perceptrons that can receive multiple inputs and produce a single output. The inputs are binary values (0 or 1), and affect the perceptron differently depending on their excitatory or inhibitory ability. The amount of excitation or inhibition is captured by weights, or values that the inputs are multiplied by. Finally, the aggregate sum of these inputs and their weights is used to calculate if the perceptron in question will fire an output to the next layer of perceptrons.

Figure 2.2: A neuron in a neural network, with binary inputs x_1, x_2, and x_3.

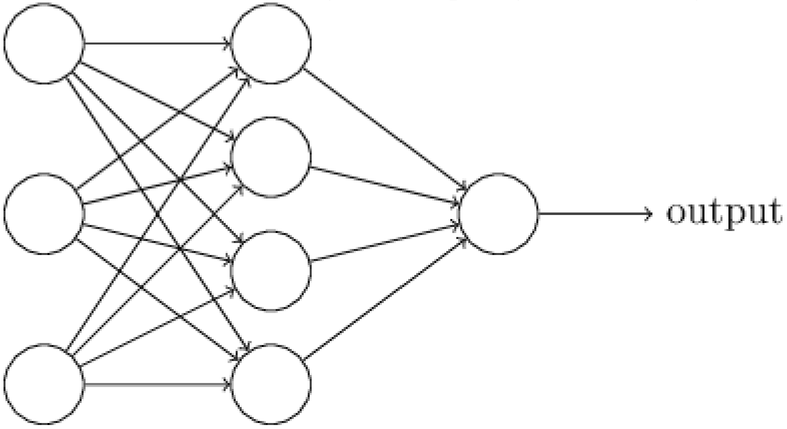


Neurons that only utilize binary inputs, however, are not efficient in learning. For example, neurons in the brain release different amount of neurotransmitters to indicate different levels of excitation. Similarly, most modern neural networks generate an output that is between 0 and 1, inclusive. This type of function is known as the sigmoid function, and the resulting neuron is known as the sigmoid neuron.

2.1.1 Neural Networks with Sigmoid Neurons

With practice, humans can arrange sigmoid neurons to generate a functional unit that can take inputs and produce a certain computational output. Functionally, sigmoid neurons act as NAND gates, which are universal to computational procedures. However, the true value of the neural network lies in data and input values that do not have any discernible pattern. In such cases, neural networks are capable of learning patterns by themselves. Therefore, neural networks can be trained with an initial data set, after which they can predict the output values of new information. Although a single sigmoid neurons can be trained, it is not adequate to use such a unit to capture the entirety of complex patterns found in data sets. Rather, a network of sigmoid neurons is created to assess input values and generate proper outputs. Neural networks are most often arranged in layers. The first layer is known as the input layer, whilst the last layer is known as the output layer. All the neural layers in the middle are known as hidden layers. The sample neural network shown in figure 2.3 takes four inputs in the first layer and produces a singular output. Since all the components involved are sigmoid neurons, the value of the output of the neural network can take any value between 0 and 1.

Figure 2.3: A neural network with three layers: an input layer, a hidden layer, and an output layer.



2.1.2 Training in a Neural Network

In general, every sigmoid neuron has a certain set of inputs $X = \{x_1, x_2, \ldots, x_n\}$. Each input is associated with a weight, or the amount of excitation or inhibition each input causes, which are expressed as the weight set $W = \{w_1, w_2, \ldots, w_n\}$. The value of the previous inputs is also influenced by a neuron-specific bias b, which acts as a value of how reactive the neuron is to the inputs. Finally, the sigmoid neuron produces an output O between 0 and 1 depending on the influence of previous inputs as follows:

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
O(W,X) &= \sigma\left(\sum_i^n w_i x_i + b\right) \\
O(W,X) &= \sigma\left(W \cdot X + b\right) \\
O(W,X) &= \frac{1}{1+e^{-W \cdot X - b}}
\end{aligned}
$$

Therefore, by computing the dot product of set W and set X, offsetting the dot product by the neural bias, and then passing that value through the sigmoid function, we generate an output between 0 and 1. If the sigmoid neuron in question is part of a hidden layer, it will pass on its output to other neurons in the next lay.

To train a network, we begin with a neural network that is set up with well-chosen (but random) weights and begin sending input values through it. The output is then compared to the value we were expecting, and computational corrections are made in increments to the weights and biases in the neural network using gradient descent algorithms. Over time, the neural network becomes more accurate and has the ability to predict future output values from input values.

2.1.3 Measuring Predictive Error

To train our neural network computationally, we need to develop a cost function, that quantifies the amount of training our neural network has undergone. That is, this function has to return a value that quanitifies the difference between prediction and reality. During training, every predicted value is compared to the actual value of the data through subtraction and squared. Thus, for a prediction y and actual value $\hat{y}$.

$$C(y,\hat{y}) = (\hat{y}-y)^2$$

The cost function is computed for every data point used to train the neural network. The overall error of the neural network is simply the average of all the cost function values for every data point that was used to train the network. However, since we are squaring the cost function, we divide the average by 2 to decrease the severity of predictive error. This error function is most commonly known as the Mean-Squared Error function, and is predominantly used to train neural networks. For a set of predictions $Y = \{y_1, y_2, \ldots, y_n\}$ and a set of actual values $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n\}$:

$$J(Y,\hat{Y}) = \sum_{i=0}^{n} \frac{C(y_i,\hat{y}_i)}{2n}$$
$$J(Y,\hat{Y}) = \sum_{i=0}^{n} \frac{(\hat{y}_i - y_i)^2}{2n}$$

2.2 Installation

The neuralnet package can be found at `https://cran.r-project.org/web/packages/neuralnet/index.html`. Download the package that works for your operating system and CPU architecture. The following code can be used to install the package from the R command line. This example illustrates the installation of a windows system binary file. However, all other installations will also be the same, with `FILE_PATH` leading to the package downloads. Alternatively, a script file can be generated to install the neuralnet package.

Figure 2.4: Code for Windows

```
install.packages("FILE_PATH.zip", repos=NULL)
package 'neuralnet' successfully unpacked with MD5 sums checked
library(neuralnet)
```

Figure 2.5: Code for Mac / OSX

```
install.packages("FILE_PATH.tgz")
package 'neuralnet' successfully unpacked with MD5 sums checked
library(neuralnet)
```

Note that safari automatically decompresses `.tgz` files to tar files, and this will cause errors when running the above code.

The installation may throw a warning reminding the user that the package may be out-of-date when compared to the R version being used.

2.3 Objectives of Case Study

There are multiple objectives that the student must accomplish over the course of this neural networks case study.

1. Understand neural networks, their functional units, and their overall purpose in Machine Learning. It is important to understand how neural networks work, beginning from an understanding of sigmoid neurons and culminating in an overall appreciation for the function of neural networks in predictive analysis.

2. Have the ability to prepare data for use in neural networks in R. This objective includes the ability to identify and remove rows with empty data, normalizing the data to allow for learning over a smaller number of iterations, and fitting data curves to create the predictive function.

3. Be able to implement the neuralnet package in R. An understanding of the functionality that the neuralnet package provides, including creating the neural network, training it with a data set, and using it for predictive analysis, is imperative to this objective.

4. Perform cross-validation on the neural network to confirm understand its learning progress. This involves using statistical measures to see how the neural network is functioning during training and prediction.

2.4 Case Study - Boston Housing Database

The Boston Housing Database is a databaase that contains the housing data from the boston area. It contains data on housing sales, including the area of the house sold, the location of the house, and other price-determining factors.

2.4.1 Loading and Separating the Data

Create a R script file and enter the following code. This code will set up the environment for the neural network. We load the data into our scripting environment, modify it to fit the neuralnet package, and split it into a training and testing dataset.

```
# Name:       neural.R
# Author:     First Last
# Date:       Month Date Year

# Load necessary libraries into R
library(neuralnet) # Contains the neuralnet package
library(MASS)      # Contains the database we will use

# Clear the current environmental variables
rm(list=ls())

# We set the seed for the random generator
# We use random data values for training and for testing
# Use this seed to replicate the results in this chapter
```

```r
set.seed(500)

# Import the database into R from the MASS library
data <- Boston

# Define the number of neurons in each hidden layer
# The input layer depends on the number of columns in the data frame
# The output layer depends on variables we ask the network to predict
hiddenLayers = c(5, 3)
```

Unfortunately, the neuralnet package cannot handle NULL values in the data. Therefore, we must remove such extraneous data from the dataset.

```r
# Unlike most R libraries, neuralnet is unable to parse NA values
# We check for NA values in the data set and print their amount
# If we have rows with missing data, we will have to remove them
apply(data, 2, function(x) sum(is.na(x)))
```

We will use 75% of the data to train the network and 25% to test the efficacy of the neural network. We split the dataset using the following code.

```r
# Generate a list of indices that are randomly sampled
# The indices point to rows in the data
# 75 percent of the data set indices is selected in this list
index <- sample(1:nrow(data), round(0.75 * nrow(data)))

# The training data set is composed of 75 percent of the test data
train <- data[index,]

# The testing data set is composed of 25 percent of the test data
test <- data[-index,]
```

2.4.2 Linear Model for Comparison

First, we run the data through a simple linear model to see how well a linearly scaled system can predict the value of interest in the data set. Our data frame is constructed so that the last column, named `data$medv`, is what is predicted. Every other column is the value of a pixel in the image.

```r
# Create a regression line using prog as the prognostic variable
lm.fit <- glm(medv~., data=train)
# Print the summary of our fit
summary(lm.fit)
# Predict values using the fit and the testing data set
lm.pr <- predict(lm.fit, test)
# Calculate the Mean-Square Error of the linear model
lm.MSE <- sum((lm.pr - test$medv)^2) / nrow(test)
```

2.4.3 Neural Network Setup and Execution

We will compare the linear model to the neural network graphically after generating results from the neural network. We now work to set up the data for the neural network. Neural networks generally do not work well with non-uniform distributions of data. Therefore, we will scale every column to distribute the values correctly. When we scale the data set, we will lose the true values. Therefore, it is important to ”unscale" the data once we generate output from the network.

```r
# Create a list of maximums from each data column
maxs <- apply(data, 2, max)
# Create a list of minimums from each data column
mins <- apply(data, 2, min)

# Scale the values in each column to its respective limits
scaled <- as.data.frame(scale(data, center=mins, scale=maxs - mins))

# Redefine a training data set using the scaled values
train_ <- scaled[index,]
# Redefine a testing data set using the scaled values
test_ <- scaled[-index,]
```

At this point, we can generate a neural network and execute it. We begin by constructing a rule for the neural network to follow. This rule simply describes the output variables and the input variables that are thought to influence the output.

```r
# Every column affects the prognostic variable
# We extract the name of every column in the data set
n <- names(train_)

# Using the normal fit construct (prog ~) does not work in this package
# Instead, we generate the command using the following code
# Will generate something like this: prog ~ c1 + c2 + c3 + ...
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse=" + ")))

# The following code will generate the neural network
# This step can take some time
nn <- neuralnet(f, data=train_, hidden=hiddenLayers, linear.output=T)

# We can plot the neural network to view its structure
plot(nn)
```

Just like the linear model that we use to compare, we will calculate the Mean-Squared Error of the neural network. For this, we extract the predictions of the neural network and ”unscale" them.

```r
# Compute predictions from our neural network
nn.pr_ <- compute(nn, test_[,1:13])

# Unscale the data output from the neural network
nn.pr <- nn.pr_$net.result * (max(data$medv) - min(data$medv)) + min(data$medv)
test.r <- (test_$medv) * (max(data$medv) - min(data$medv)) + min(data$medv)

# Root Mean Square Error computation for the neural network
nn.MSE <- sum((test.r - nn.pr)^2) / nrow(test_)
```

2.4.4 Graphical Representation

Now, we simply use descriptive and graphical methods to highlight the difference between a linear fit model and the neural network.

```r
# A higher number will suggest a lower correlation between the prediction and reality
print(paste(lm.MSE, nn.MSE))

# Set up a plot to hold two graphs
par(mfrow=c(1, 2))

# Plot the neural network's real vs. predicted values
plot(test$medv, nn.pr, col="red", main="Real vs. Predicted NN", pch=18, cex=0.7, xlab="Actual Value", ylab="Neural Network")
# Generate a line and legend for the plot
abline(0, 1, lwd=2)
legend("bottomright", legend="LM", pch=18, col="red", bty="n", cex=0.95)

# Plot the linear model's real vs. predicted values
plot(test$prog, lm.pr, col="blue", main="Real vs. Predicted LM", pch=18, cex=0.7, xlab="Actual Value", ylab="Linear Fit")
# Generate a line and legend for the plot
abline(0, 1, lwd=2)
legend("bottomright", legend="LM", pch=18, col="blue", bty="n", cex=0.95)

# Generate a plot with both scatter plots for better comparison
par(mfrow=c(1, 1))
plot(test$medv, nn.pr, col="red", main="Real vs. Predicted", pch=18, cex=0.7, xlab="Actual Value", ylab="Predicted Value")
points(test$medv, lm.pr, col="blue", pch=18, cex=0.7)
abline(0, 1, lwd=2)
legend("bottomright", legend=c("NN", "LM"), pch=18, col=c("red", "blue"))
```

A proper execution of the neural network should show that the Mean-Squared Error of the generalized linear fit model is much higher than that of the neural network. Furthermore, the graphs should portray a tighter clustering of the neural network data points around the linear fit.

2.5 Student Assignment - Temperature Dataset

The student is tasked with creating a neural network that interprets a dataset that contains the monthly temperatures from January to November over multiple years. The goal of the network is to predict the temperatures of the month of December. The following code should be used to parse the data:

```r
# Name:          neuralStudent.R
# Author:        First Last
# Date:          Month Date Year

### PARSING ###
```

```

monthlyavg <- c(-4.07, -1.75, 2.17, 8.06, 13.90, 18.77, 21.50, 20.50, 16.27, 9.74, 2.79, -2.36)
rawdata <- read.csv("berkeleyusdata.txt", header = FALSE)
tempdevs <- rawdata$V4
temps <- rawdata$V4 + monthlyavg
months <- rawdata$V3
mat <- matrix(temps,ncol=12)
datafornnet <- data.frame(mat)
colnames(datafornnet) <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Prog")
data <- datafornnet
rm(monthlyavg, rawdata, tempdevs, temps, months, mat, datafornnet)
```

Thus, the data set is stored in a variable called `data`. The neural network can now be built by the students. The name of the variable being predicted is `Prog`.

```
# Clear environment variables
rm(list = ls())

# Import the appropriate libraries
library(neuralnet)

# We do not need to set a seed
# We can if the pseudo-random system creates better results
# set.seed(500)

# Vary the number of neurons in the hidden layers to get better results
# This is an example a student may use
hiddenLayers = c(5, 3)

# Ensure that the data has no NaN values
apply(data, 2, function(x) sum(is.na(x)))

# Split the data into 75% and 25%
# One dataset is for training and the other is for testing
index <- sample(1:nrow(data), round(0.75 * nrow(data)))
train <- data[index,]
test <- data[-index,]

# Create a generalized linear fit to compare with the neural network
lm.fit <- glm(Prog~., data=train)
# Print the summary of the fit
summary(lm.fit)
# Predict values using the fit
lm.pr <- predict(lm.fit, test)
# Calculate the Mean-Squared Error of the model
lm.MSE <- sum((lm.pr - test$Prog)^2) / nrow(test)

# Calculate mins and maxes for scaling
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)
# Scale using the range of the data
scaled <- as.data.frame(scale(data, center=mins, scale=maxs-mins))
# Create the scaled training and test data sets
```

```
train_ <- scaled[index,]
test_ <- scaled[-index,]

# Create the neural network
n <- names(train_)
f <- as.formula(paste("Prog ~", paste(n[!n %in% "Prog"], collapse=" + ")))
nn <- neuralnet(f, data=train_, hidden=hiddenLayers, liner.output=T)
plot(nn)

# Calculate the scaled predictions and errors
nn.pr_ <- compute(nn, test_[,1:11])
nn.pr <- nn.pr_$net.result * (max(data$Prog) - min(data$Prog)) + min(data$Prog)
test.r <- (test_$Prog) * (max(data$Prog) - min(data$Prog)) + min(data$Prog)
nn.MSE <- sum((test.r - nn.pr)^2) / nrow(test_)

# Print the resulting mean-squared errors
print(paste(lm.MSE, nn.MSE))

# Generate the graphics for displaying data
par(mfrow=c(1,2))
plot(test$Prog, nn.pr, col="red", main="Real vs. Predicted NN", pch=18, cex=0.7,
     xlab="Actual Value", ylab="Neural Network")
abline(0, 1, lwd=2)
legend("bottomright", legend="LM", pch=18, col="red", bty="n", cex=0.95)
plot(test$Prog, lm.pr, col="blue", main="Real vs. Predicted LM", pch=18, cex
    =0.7, xlab="Actual Value", ylab="Linear Fit")
abline(0, 1, lwd=2)
legend("bottomright", legend="LM", pch=18, col="blue", bty="n", cex=0.95)
par(mfrow=c(1,1))
plot(test$Prog, nn.pr, col="red", main="Real vs. Predicted", pch=18, cex=0.7,
    xlab="Actual Value", ylab="Predicted Value")
points(test$Prog, lm.pr, col="blue", pch=18, cex=0.7)
abline(0, 1, lwd=2)
legend("bottomright", legend=c("NN", "LM"), pch=18, col=c("red", "blue"))
```

2.6 Downloading The Dataset

Download the file located at `https://github.com/nosyarg/textbookdata/blob/master/berkeleyusdata.txt`. This data comes from the Berkeley Earth Project. Once you have downloaded the data, run the following code: as relevant, and make December the prognosis variable