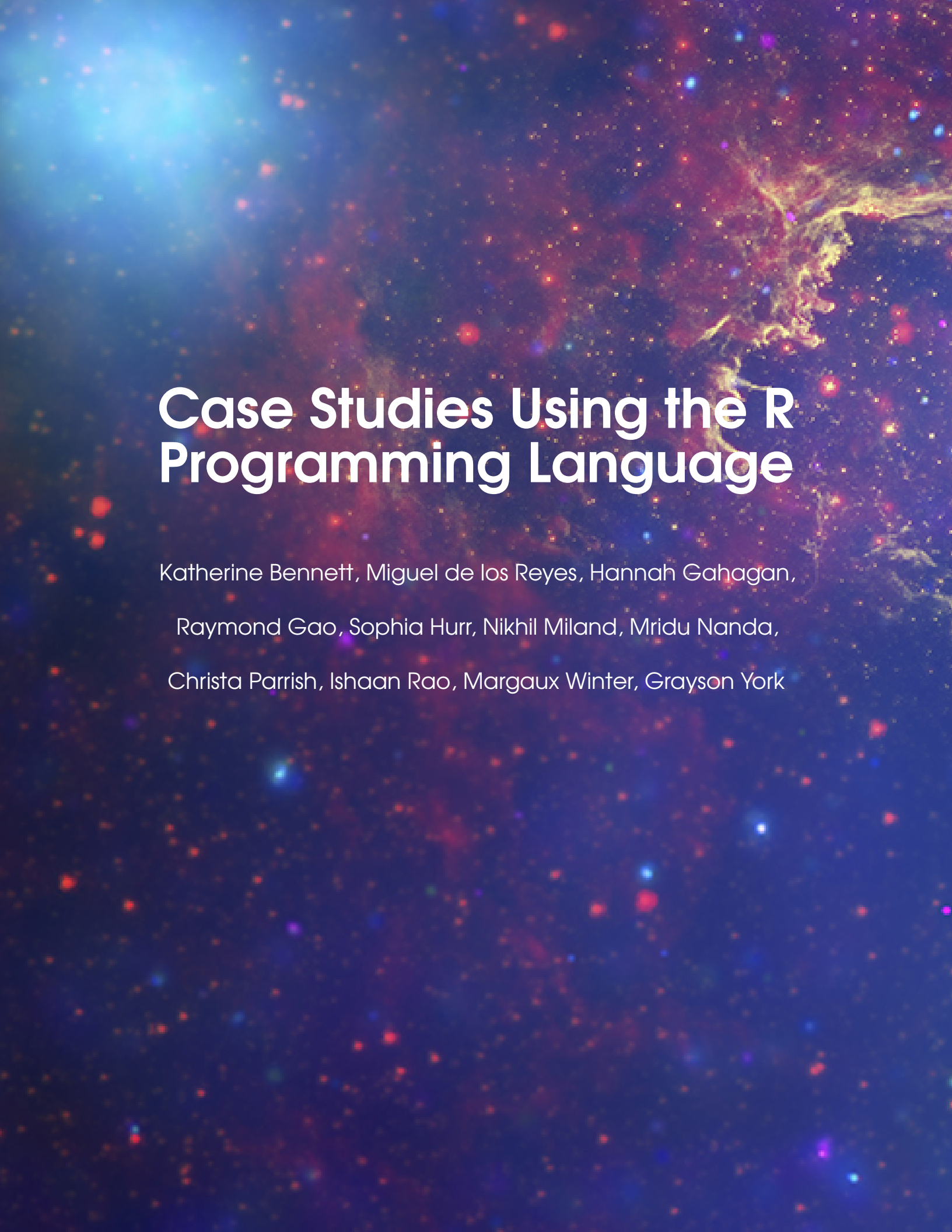


Case Studies Using the R Programming Language

Katherine Bennett, Miguel de los Reyes, Hannah Gahagan,

Raymond Gao, Sophia Hurr, Nikhil Miland, Mridu Nanda,

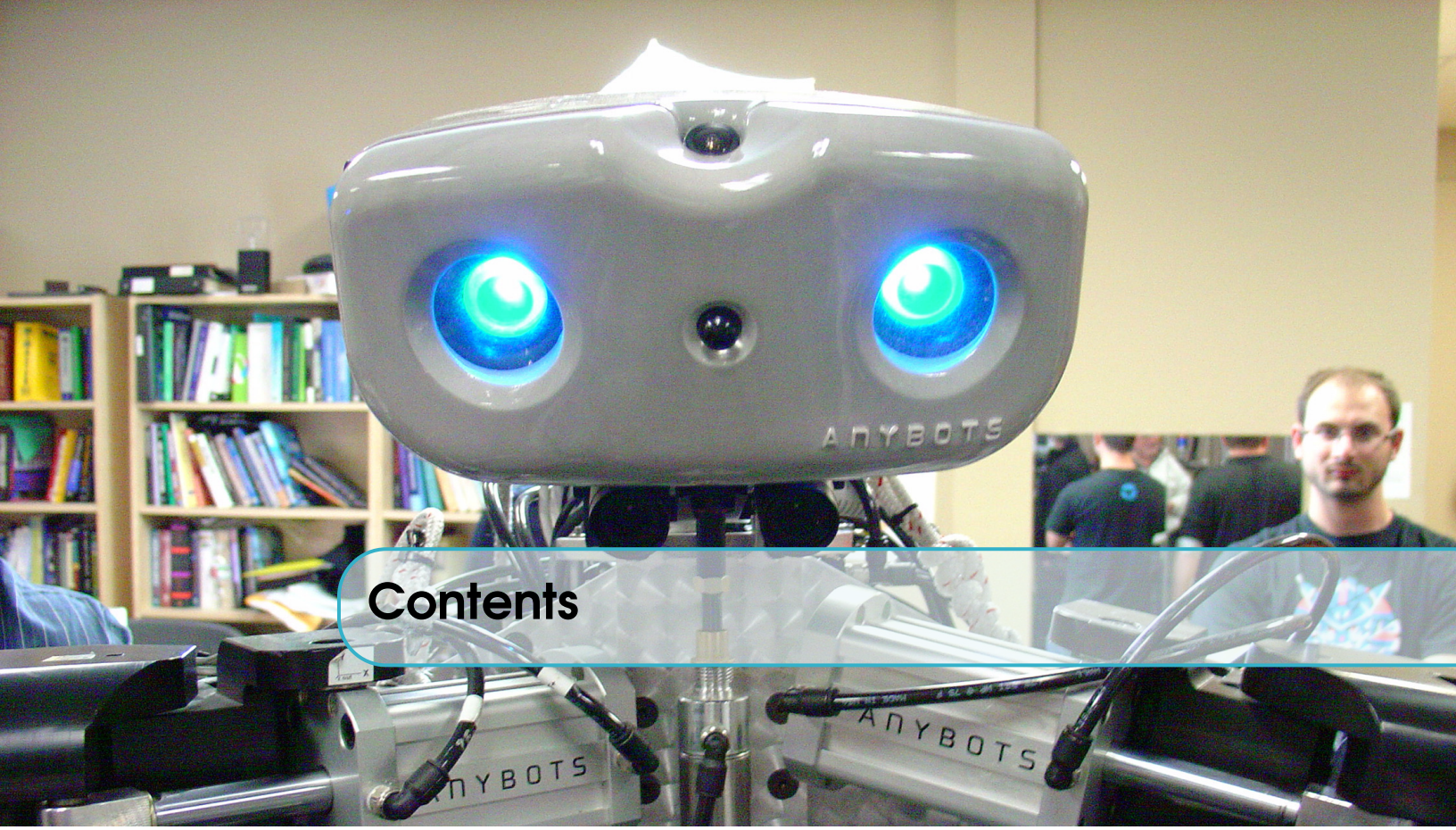
Christa Parrish, Ishaan Rao, Margaux Winter, Grayson York

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.
Editing and typesetting performed by Grayson York using LaTeX 2_ε.
Graphics designed by Tracy Telenko, Creative Services Group, The North Carolina School of Science and Mathematics. Template created by Mathias Legrand.

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, so 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.(9) 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 be renamed.

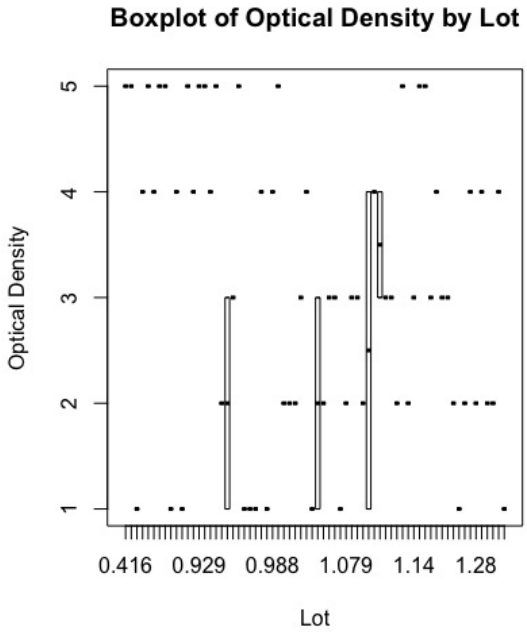


Figure 1.1: Boxplot

```
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))
```

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
    ","Medium","High")))
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.

```
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 be 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.

```
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.

```
# 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

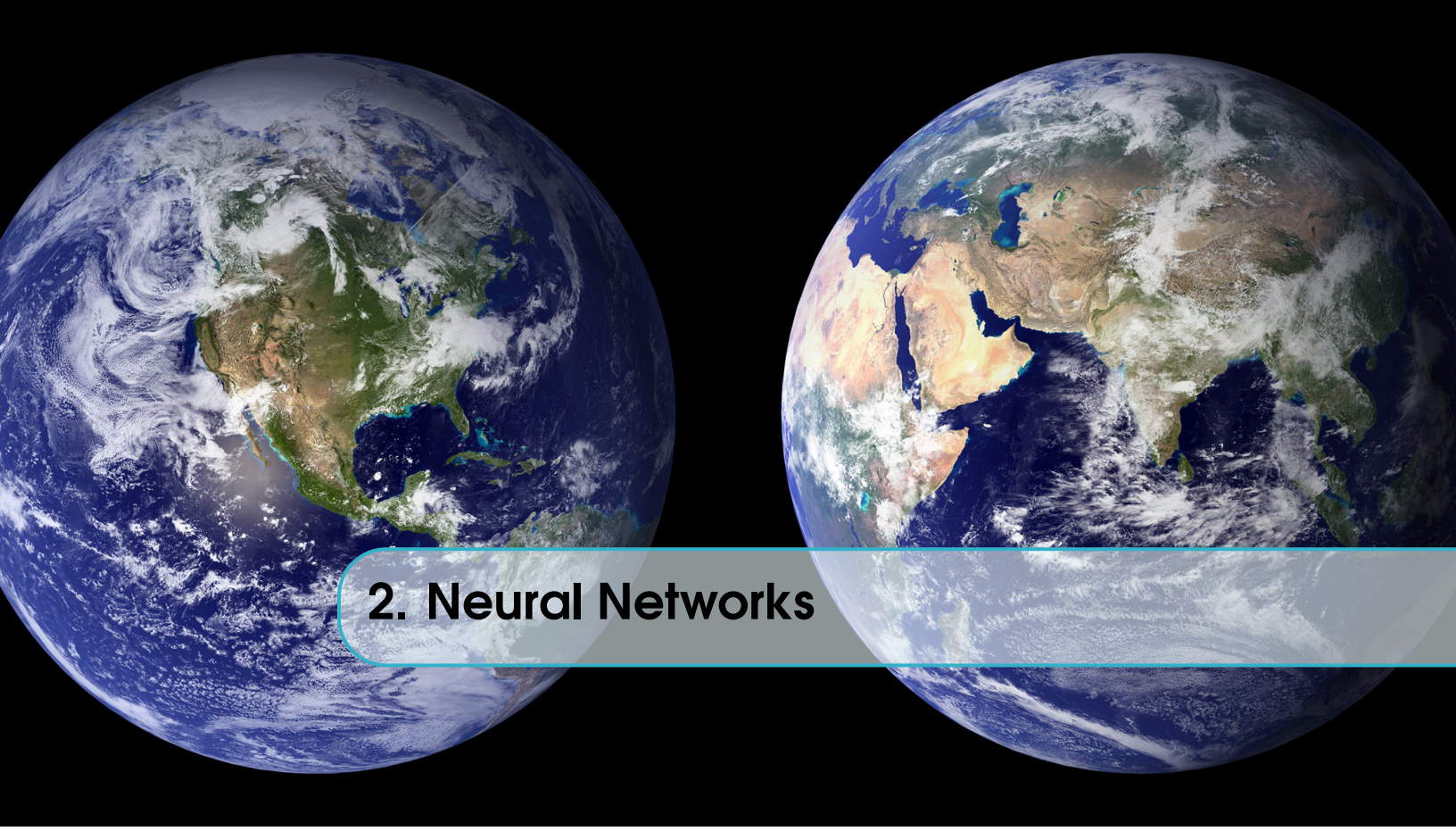
```
# KEY
```

```r
# 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. (6)

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

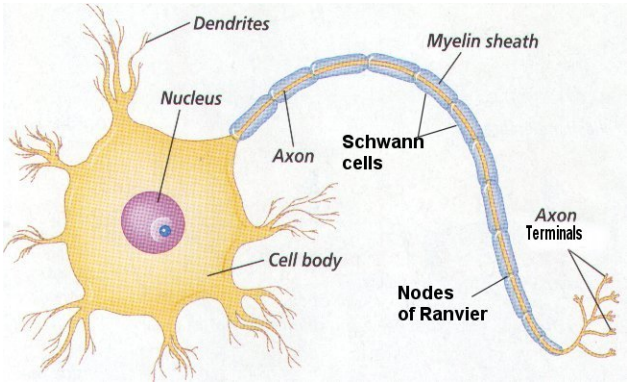


Figure 2.1: The Parts of an Anatomical Neuron

weights is used to calculate if the perceptron in question will fire an output to the next layer of perceptrons.

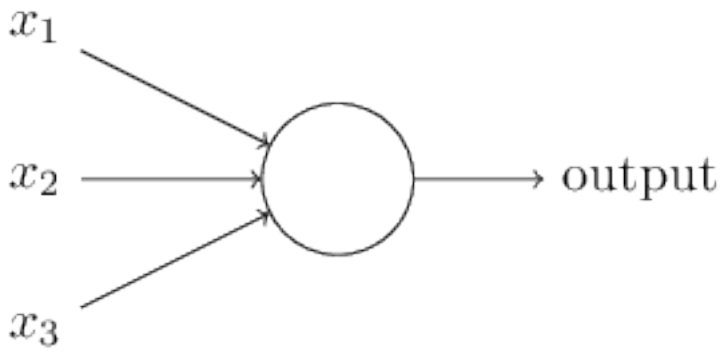


Figure 2.2: A neuron in a neural network, with binary inputs x_1, x_2, and x_3.

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

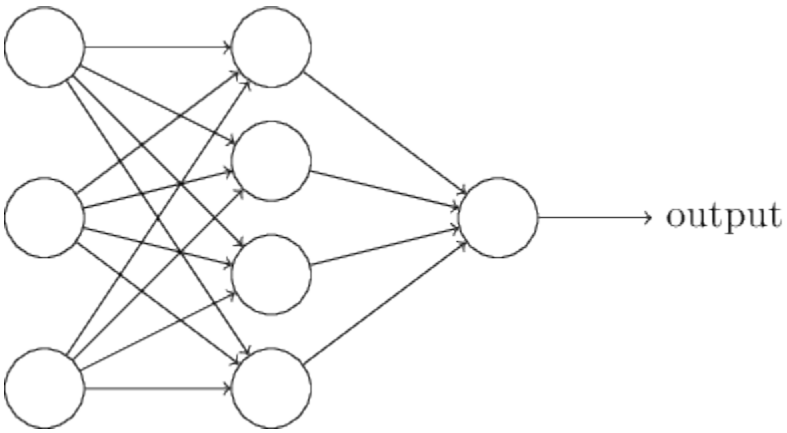


Figure 2.3: A neural network with three layers: an input layer, a hidden layer, and an output layer.

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.

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:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$O(W, X) = \sigma\left(\sum_{i}^{n} w_i x_i + b\right)$$

$$O(W, X) = \sigma\left(W \cdot X + b\right)$$

$$O(W, X) = \frac{1}{1 + e^{-W \cdot X - b}}$$

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.

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.

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)
```

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
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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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:

```
# 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`.

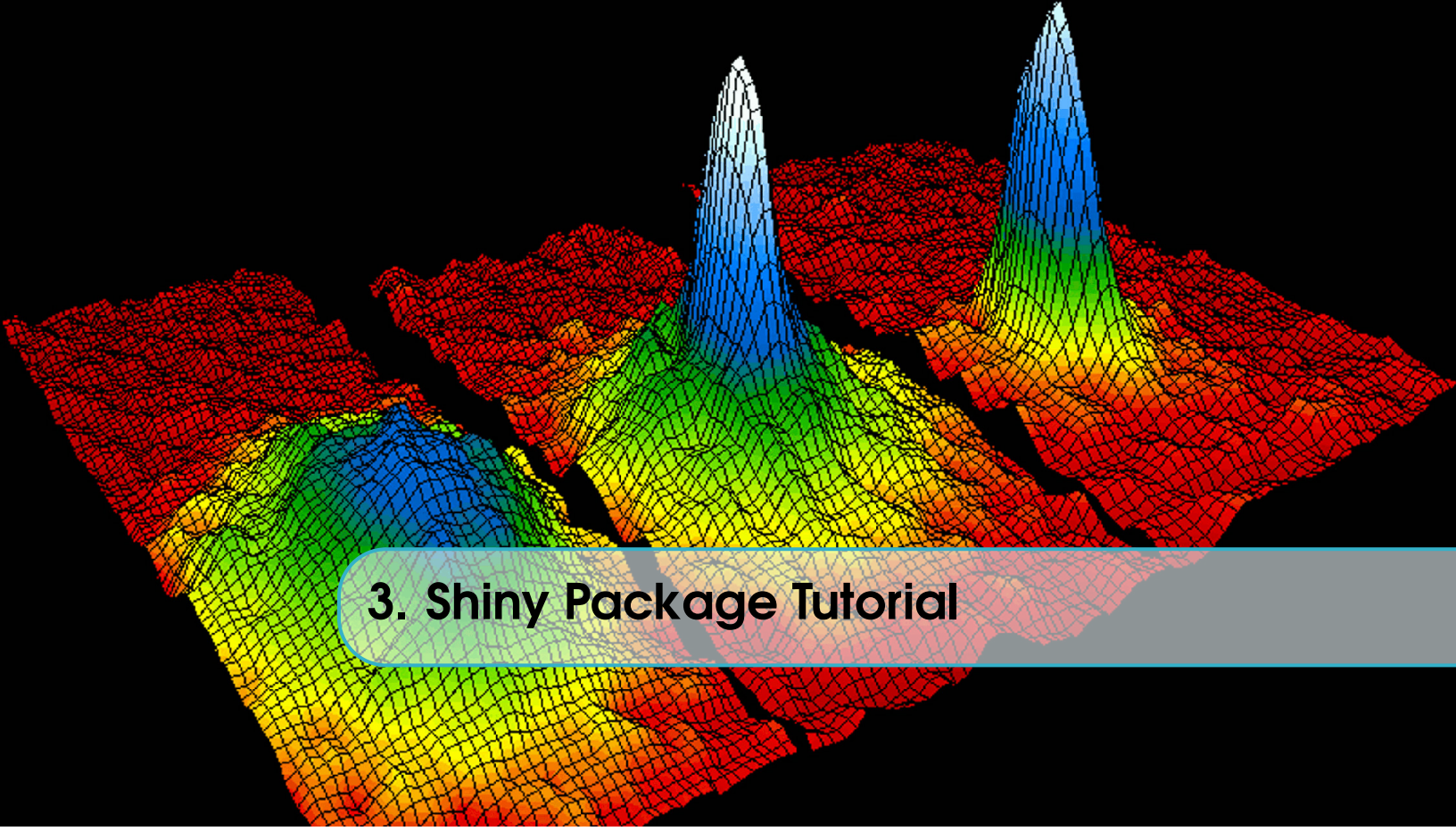
```
# Clear environment variables
rm(list = ls())

# 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)
```

```
14
15 # Ensure that the data has no NaN values
16 apply(data, 2, function(x) sum(is.na(x)))
17
18 # Split the data into 75% and 25%
19 # One dataset is for training and the other is for testing
20 index <- sample(1:nrow(data), round(0.75 * nrow(data)))
21 train <- data[index,]
22 test <- data[-index,]
23
24 # Create a generalized linear fit to compare with the neural network
25 lm.fit <- glm(Prog~., data=train)
26 # Print the summary of the fit
27 summary(lm.fit)
28 # Predict values using the fit
29 lm.pr <- predict(lm.fit, test)
30 # Calculate the Mean-Squared Error of the model
31 lm.MSE <- sum((lm.pr - test$Prog)^2) / nrow(test)
32
33 # Calculate mins and maxes for scaling
34 maxs <- apply(data, 2, max)
35 mins <- apply(data, 2, min)
36 # Scale using the range of the data
37 scaled <- as.data.frame(scale(data, center=mins, scale=maxs-mins))
38 # Create the scaled training and test data sets
39 train_ <- scaled[index,]
40 test_ <- scaled[-index,]
41
42 # Create the neural network
43 n <- names(train_)
44 f <- as.formula(paste("Prog ~", paste(n[!n %in% "Prog"], collapse=" + ")))
45 nn <- neuralnet(f, data=train_, hidden=hiddenLayers, liner.output=T)
46 plot(nn)
47
48 # Calculate the scaled predictions and errors
49 nn.pr_ <- compute(nn, test_[,1:11])
50 nn.pr <- nn.pr_$net.result * (max(data$Prog) - min(data$Prog)) + min(data$
       Prog)
51 test.r <- (test_$Prog) * (max(data$Prog) - min(data$Prog)) + min(data$Prog)
52 nn.MSE <- sum((test.r - nn.pr)^2) / nrow(test_)
53
54 # Print the resulting mean-squared errors
55 print(paste(lm.MSE, nn.MSE))
56
57 # Generate the graphics for displaying data
58 par(mfrow=c(1,2))
59 plot(test$Prog, nn.pr, col="red", main="Real vs. Predicted NN", pch=18, cex
       =0.7, xlab="Actual Value", ylab="Neural Network")
60 abline(0, 1, lwd=2)
61 legend("bottomright", legend="LM", pch=18, col="red", bty="n", cex=0.95)
62 plot(test$Prog, lm.pr, col="blue", main="Real vs. Predicted LM", pch=18, cex
       =0.7, xlab="Actual Value", ylab="Linear Fit")
63 abline(0, 1, lwd=2)
64 legend("bottomright", legend="LM", pch=18, col="blue", bty="n", cex=0.95)
65 par(mfrow=c(1,1))
66 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.



3. Shiny Package Tutorial

3.1 Introductory Reading

Shiny is a package used in R, that enables the user to easily create interactive applications. Shiny can be used to create both simple histograms of disease data or interactive maps of the crime rate in different cities. With Shiny the possibilities are endless. In this lab, you will learn how to create a basic Shiny application. Look at the picture in Figure 3.1. (11) This is a sample Shiny app depicting a graph of many famous movies. The graph plots movies based on their rating and the number of reviews it has on Rotten Tomatoes. On the left of the graph you find various inputs, such as Oscars and genres, that can customize the output of the graph. These are the basic components of a Shiny app: inputs and outputs. Here are some examples of Shiny input functions:

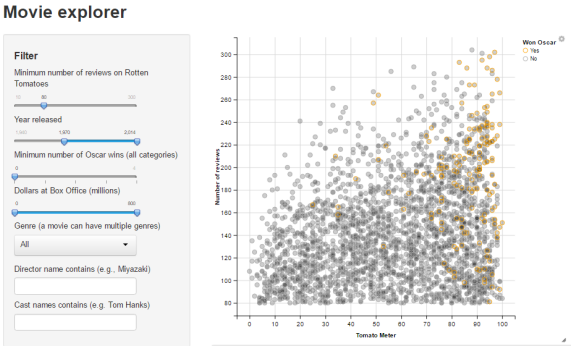


Figure 3.1: Sample Shiny app

- checkboxInput: a single check box
- dateInput: a calendar
- fileInput: file upload
- numericInput: enter number
- radioButtons: set of radio buttons
- selectInput: box with multiple choices
- sliderInput: a slider bar
- textInput: enter text

The users of a Shiny app will interact solely with the inputs that you, the developer, will allow them to. The above inputs can collect various types of data input from the user, to determine many different types of outputs. Here are some of the output functions in Shiny:

- imageOutput: creates image
- plotOutput: creates a plot
- tableOutput: creates a table
- textOutput: creates text

Understanding how inputs and outputs relate to each other is a key requirement for developing a Shiny app. In the model students will create today, the inputs will be dropdown boxes (selectInput) and the output will be a simple scatterplot (plotOutput).

3.2 Objectives

Students will be expected to learn the basic components of a Shiny App. Students will create their own application after learning the fundamentals of the package. This application will illustrate the progression of Olympic swimming medalists' times since the modern Olympics began.

- By the end of this lesson the student will understand the various parts of a Shiny app, while building a simple population prediction application.
- Students will also be able to create their own Shiny apps from scratch, and will be expected to create a Shiny app displaying Olympic swimming times over various events and years.

3.3 Building the Model

Shiny applications have a wide variety of uses. Today students will learn how to create a basic population prediction model.
To begin the application, open a new R script file in RStudio, clear the environment, set a working directory, and import the libraries necessary for this project. To clear the environment and set a directory use the following functions. Make sure you set a directory on your computer that the dataset is located in.

```
rm(list=ls())
setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")
```

The libraries are as follows:

```
library(ggplot2)
library(scales)
library(ggmap)
library(dplyr)
library(car)
library(gcookbook)
library(shiny)
```

If you don't have one of these libraries installed on your computer, you can simply type

```
install.packages("insertlibraryname")
```

in the console to download them.
Next, we need to import the dataset of all the population data in NC for the next five years into a variable of your choice (I used popData). This can be done with the read `.csv` function. Make sure the data file is in directory you specified above.

```
popData <- read.csv("Population.csv")
```

Shiny applications have two components: a user-interface definition and a server script. The user-interface is used to control the design and layouts of the Shiny application. The server script for the Shiny application controls the algorithms and code that tell the Shiny app how to manipulate the inputs to produce outputs.(14) Three functions — headerPanel, sidebarPanel, and mainPanel — define the various regions of the user-interface. The application will be called "Population Statistics" so we specify that as the title when we create the header panel. The other panels are empty for now. We create a UI function with the aforementioned components, like this:

```
ui <- shinyUI(fluidPage(
  headerPanel("Population Statistics"),
  sidebarPanel(),
```

```
  mainPanel()
))
```

Next, define a skeletal server implementation. This can be done by creating a function called server with two parameters: input and output.

```
server <- function(input, output) {

}
```

The server function will be empty to begin with, but it will eventually be used to set relationships between the inputs and outputs of the application. At the end of your code, we need to call the shiny function so the application will run. To do that use the following syntax:

```
shinyApp(ui = ui, server = server)
```

The skeletal structure of the simplest Shiny application possible is shown in figure 3.2. Run the Shiny application by selecting all of your code and clicking the run button in the top right corner of RStudio. The blank Shiny application should open in a new window as shown in figure 3.2. To begin the model, we need to display the inputs

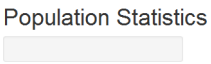


Figure 3.2: Blank Shiny App

of our program in the sidebar panel of the application. Our program is taking age group as an input and displaying a plot showing the predicted populations of that age group in North Carolina over the next five years. To do this, we will create a selectInput with all of the age groups as choices, like so:

```
sidebarPanel(
    selectInput(
        inputId = "age",
        label = "Age:",
        choices = c(
            "Total", "0-4 years", "5-9 years",
            "10-14 years", "15-19 years", "20-24 years",
            "25-29 years", "30-34 years", "35-39 years",
            "40-44 years", "45-49 years", "50-54 years",
```

```
            "55-59 years", "60-64 years", "65-69 years",
            "70-74 years", "75-79 years", "80-84 years",
            "85+ years")
    )
  )
```

The inputId is simply a name that we give to each input, so we can access them in the server function. The label is the text that will appear on the application above the input box.

We also need to tell the application that our plot will be displayed in the main panel of the UI. This involves creating a plotOuput, like so:

```
mainPanel(
   plotOutput(outputId = "myPlot")
)
```

Just like in `selectInput`, we need to give the output an ID, so we can use that name to access it in the server function. Once the model is run, the application should look like figure 3.3. Next we need to define the server-side of the application which

Figure 3.3: Adding Inputs

will accept inputs and compute outputs. Our server function will contain the code necessary to do this. We first tell the function that we are creating a plot as an output, and we can create the plot using the renderPlot command. The command renderPlot is what makes the application react. This means when a user changes the input, the output will instantly change. This is what makes Shiny so interactive.

Within the renderPlot function we first select what data we will use and then plot that data using the ggplot function. To select our data, create a variable called `dataInput`, and take a subset of the `.csv` file we imported into R. We want to take only the data that the user specified, so instead of importing all the age group data, we only import the age group that the user selected. We can access that specific age group using `input$age`. The `ggplot` function takes in various inputs, such as the data, the variables, and the title. The `aes` function within `ggplot` indicates which axis is given to which variable, while the `geom_point` function draws the points on the graph. The `ggtitle` function creates a title over the plot. The completed `server` function should look like this:

```
server <- function(input, output) {
  output\$myPlot <- renderPlot({
    dataInput <- subset(popData, (Age==input\$age), select=c(Year,
        Population, State))

    ggplot(
        data = dataInput,
        aes(x=Year, y=Population)
    ) + geom_point() + ggtitle("Population")

 })

}
```

Finally, run the entire code including the `shinyApp` command. Your screen should look like the screen displayed in Figure 3.4. The application should have a working select box, a title, and a plot displaying the population data.

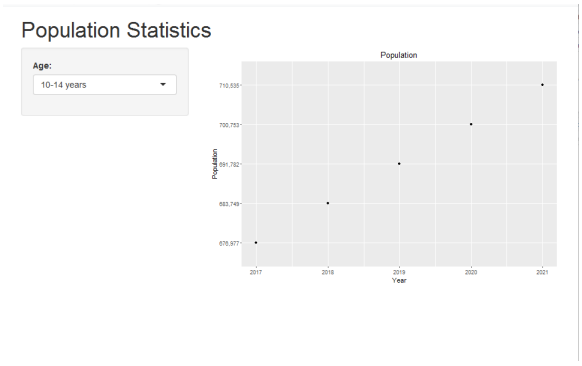


Figure 3.4: Final Model

3.4 Deliverable

You are expected to deliver properly formatted code written in the R language, using the Shiny package. Additionally, you will need to provide screenshots of various input combinations to demonstrate that your application is fully functional. This model will display Olympic swimming times over the modern Olympics. The variables for this model include time, year, and medal. The inputs are the event and gender of the swimmer. The graph should display the medaling times for each event and gender for each year of the modern Olympics. The data for this project is found within the `SwimmingExampleM` dataset provided.

3.4.1 Programming Hints

In the deliverable you will need to be able to add color to the model. It is quite simple. In the previous section we explained that the `ggplot` command makes the

actual plot. In order to create color we just add `color = variablename`, inside the `aes()` function.

```
aes(x=Time, y=Year, color=Medal)
```

The final shiny app screen should look like figure 3.5

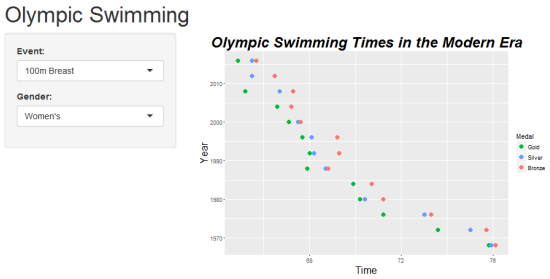


Figure 3.5: Deliverable

3.5 Teaching Code

```
#Ishaan Rao
#September 1, 2016
#pop.R

#clean up and set the directory
rm(list=ls())
setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")

#load libraries
library(ggplot2)
library(scales)
library(ggmap)
library(dplyr)
library(car)
library(gcookbook)
library(shiny)

#read data file
popData <- read.csv("Population.csv")

#setting the server, code instructing R what to do with inputs
#output\$my plot is what will be created when the app is run
#renderPlot allows Shiny to adjust when the user changes an option
#By defining the subset we indicate which inputs we want to be selected by
    the user
#select are the values in the dataset that are going to be plotted
#ggplot give direction to create plot, aes() defines the axes, ggtitle
    creates
title and geom_point creates the scatter plot
server <- function(input, output) {
```

```
  output\$myPlot <- renderPlot({
    dataInput <- subset(popData, (Age==input\$age), select=c(Year,
        Population, State))

    ggplot(data = dataInput, aes(x=Year, y=Population))
    + geom_point() + ggtitle("Population")

  })

}
# Define UI for application that draws a scatterplot
#headerPanel creates the title
#sidebarPanel creates the user entered section
#selectInputs is where you insert the options that the user can select
#mainPanel outputs the app using all of the previous code we inputed
ui <- shinyUI(fluidPage(
  headerPanel("Population Statistics"),
  sidebarPanel(
    selectInput(
        inputId = "age",
        label = "Age:",
        choices = c(
            "Total", "0-4 years", "5-9 years",
            "10-14 years", "15-19 years", "20-24 years",
            "25-29 years", "30-34 years", "35-39 years",
            "40-44 years", "45-49 years", "50-54 years",
            "55-59 years", "60-64 years", "65-69 years",
            "70-74 years", "75-79 years", "80-84 years",
            "85+ years")
    )
  ),
  mainPanel(
    plotOutput(outputId = "myPlot")
  )
))
#this runs the Shiny App
shinyApp(ui = ui, server = server)
```

3.6 Example Student Code

```
#Ishaan Rao
#September 1, 2016
#olympic.R

#load libraries
library(ggplot2)
library(scales)
library(ggmap)
library(dplyr)
library(car)
library(gcookbook)
library(shiny)
```

```
13
14 rm(list=ls())
15 setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")
16
17 swimData <- read.csv("SwimExampleM.csv")
18 #setting the server, code instructing R what to do with inputs
19 #output\$my plot is what will be created when the app is run
20 #renderPlot allows Shiny to adjust when the user changes an option
21 #By defining the subset we indicate which inputs we want to be selected by
       the user
22 #select are the values in the dataset that are going to be plotted
23 #ggplot give direction to create plot, aes() defines the axes, ggtitle
       creates
24 title and geom_point creates the scatter plot
25 #scale_color_hue creates different colors for each of the three models\\
26 server <- function(input, output) {
27
28   output\$myPlot <- renderPlot({
29
30     dataInput <- subset(swimData,
31              (Event==input\$event & Gender==input\$gender),
32                 select=c(Time, Year, Medal, Name))
33
34     ggplot(data = dataInput, aes(x=Time, y=Year, color=Medal)) +
35         geom_point(size=3) +
36         ggtitle("Olympic Swimming Times in the Modern Era") +
37         theme(
38             plot.title = element_text(color="black", size=24,
39             family="Times", face="bold.italic"), axis.title = element_text(
                   size=16)
40         ) +
41         scale_color_hue(breaks=c("Gold","Silver","Bronze"))
42   })
43 }
44 # Define UI for application that draws a scatterplot
45 #headerPanel creates the title
46 #sidebarPanel creates the user entered section
47 #selectInputs is where you insert the options that the user can select
48 #In the model we have two different inputs one for event and one for gender,
       we
49 create two different selectInput commands to account for the options
50 #mainPanel outputs the app using all of the previous code we inputted
51 ui <- shinyUI(fluidPage(
52   headerPanel("Olympic Swimming"),
53   sidebarPanel(
54     selectInput(
55         inputId = "event",
56         label = "Event:",
57         choices = c(
58             "50m Free", "100m Free", "100m Back", "100m Breast",
59             "100m Butterfly", "200m Free", "200m Back", "200m Breast",
60             "200m Butterfly", "200m IM", "400m IM"
61         ),
62         selected = "50m Free"
63     ),
64     selectInput(
```

```
        inputId = "gender",
        label = "Gender:",
        choices = c("Men's", "Women's"),
        selected = "Women's"
    )
  ),
  mainPanel(
    plotOutput("myPlot")
  )
))
#runs application
shinyApp(ui = ui, server = server)
```

3.7 Further Readings

- For a full tutorial on creating a Shiny app, visit this tutorial on the Shiny website: `http://shiny.rstudio.com/tutorial/`.
- To learn more advanced Shiny topics, functions, and capabilities visit: `http://shiny.rstudio.com/articles/`.
- To see sample Shiny apps in action visit Shiny's gallery: `http://shiny.rstudio.com/gallery/`

4. deSolve

4.1 Introduction to Computational Differential Solutions

Many of the systems studied in life sciences, environmental sciences, and physical sciences can be modeled with differential equations. Differential equations serve to model rates of change, which are often easier to measure than every interaction in the system. A differential equation is an equation for a function that relates the values of the function to the values of its derivatives. An ordinary differential equation (ODE) is a differential equation for a function of a single variable (such as $x(t)$), while a partial differential equation (PDE) is a differential equation for a function of several variables (such as $v(x, y, z, t)$). An ODE contains ordinary derivatives and a PDE contains partial derivatives. Typically,PDE's are much harder to solve than ODE's (3).

Initial Value Problems (IVPs) involve differential equations where certain initial values in the model are known. The `deSolve` package uses computational means to analyze and compute the values of states in such models. In a differential equation, the state is the variable that is varied with respect to the independent variable in the system. Parameters, on the other hand, are values that signify some characteristic or attribute of the system. For example, in the given system of equations:

$$\frac{dY}{dt} = aY$$

The state of the differential equation is Y. The independent variable in this system is t. Finally, a is a parameter that can be varied to vary the solution. In this lab, we will analyze systems of ordinary first-order differential equations, which relate the first derivative to states and parameters.

4.1.1 Examples of Differential Equations

The *Malthusian law of population growth* is used to model the populations of certain kinds of organisms living in ideal environments for limited lengths of time t. It gives the rate at which a population p changes with respect to t. The value of the constant r depends on the organism.

$$\frac{dp}{dt} = rp$$

This *second-order reaction rate* law gives the rate at which a single chemical species combines to produce a new species, such as methyl radicals combining in a gas to form ethane molecules.

$$\frac{dx}{dt} = k(A - x)^2$$

This equation models the motion of a damped mass-spring system subjected to a time-dependent force $F(t)$.

$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + kx = F(t)$$

4.2 Installation

To install the `deSolve` package in R, enter the following code into the command line. Installation may produce warnings that are generally inconsequential to the overall usage of the package. However, if the package does not install correctly, contact the package developers.

```
> install.packages("deSolve")
> library(deSolve)
```

4.3 Objectives of Case Study

There are certain objectives that students should focus to accomplish while completing this activity.

- Know what ordinary differential equations and partial differential equations are. For the case studies that follow, it is inconsequential for students to understand the structure and function of ordinary differential equations.
- Understand how initial value problems arise in ordinary differential equation systems in the natural sciences. That is, use initial values found in nature and differential equation models to produce valid solutions.

- Know how to set up a differential equation given a system and the relationship between rates. This is important for modeling a given system.

- Know how to use `deSolve` to computationally solve systems of ordinary first-order differential equations. This R package allows for computational solutions that effectively model the changes in the dependent variables of the system.

4.4 Case Study - Dampened Mass-Spring System

As discussed in the examples section, a well-known second-order differential equation is used to analyze the motion of an oscillating mass-spring.

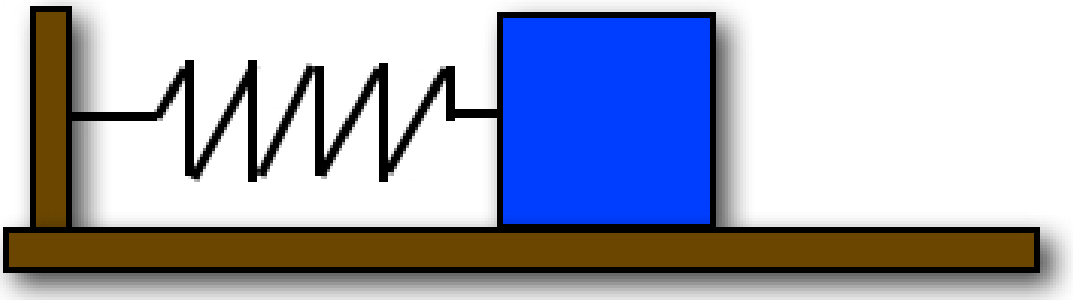


Figure 4.1: A mass-spring system

A mass-spring system is often called a harmonic oscillator. In an idealized system where no heat loss occurs, the mass-spring oscillator will continue oscillating forever.

The nature of a spring can be described using a single parameter, often noted as k. The spring constant k is used primarily in Hooke's Law, which relates the amount a spring stretches from rest, x, to the amount of force F that is applied on the mass m. Hooke's Law states that:

$$F = -kx$$

By Newton's Second Law, we also know that:

$$F = m\frac{d^2x}{dt^2}$$

Therefore, we can create a second-order differential equation for the idealized mass-spring oscillator:

$$m\frac{d^2x}{dt^2} = -kx$$
$$\boxed{m\frac{d^2x}{dt^2} + kx = 0}$$

Unfortunately, due to the complexity of behaviors and solutions in second-order differential equations, it is computationally easier to create a system of differential equations $x_1', x_2', \ldots, x_n'$ that model the system.

We begin with a few basic states that will represent the differential equations in our system.

$$x_1 = x$$
$$x_2 = \frac{dx}{dt}$$

We compute the derivatives of these states:

$$x_1' = x' = x_2$$
$$x_2' = x''$$

By solving the second-order differential equation in terms of the second derivative x'':

$$m\frac{d^2x}{dt^2} + kx = 0$$
$$mx'' + kx = 0$$
$$x'' = -\frac{k}{m}x = -\frac{k}{m}x_1$$

Therefore, the system of ordinary first-order differential equations that we are trying to solve is:

$$\frac{dx_1}{dt} = x_2$$
$$\frac{dx_2}{dt} = -\frac{k}{m}x_1$$

4.4.1 Computational Solutions

We begin by setting up the environment to begin using the `deSolve` package. We import the library and clear the environment variables.

```
1 # Author:       First Last
```

```
# Date:          Month Date Year
# Name:          DiffEq.R

# Import the library into R
library(deSolve)

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

As stated in the introduction, we are solving an initial value problem. In this case, we use arbitrary values for states and parameters. We initialize $x_1 = 100$ to suggest that the spring is stretched 100 meters past it's rest phase. Furthermore, $x_2 = 0$ to suggest that the motion begins from rest, when the rate of change is 0.

```
# m is the mass of the object (1 kg)
# k is the spring constant (50 kg / s^2)
parameters <- c(m = 1, k = 50)

# X is the first differential equation (100 m)
# Y is the second differential equation (0 m/s)
state <- c(X = 100, Y = 0)
```

We can define a function that represents the system of differential equations we have defined. Such a function will effectively relate the state and parameters to the derivatives in question.

```
# Function that defines the system
system <- function(t, state, parameters) {
    with(as.list(c(state, parameters)), {
        dX <- Y
        dY <- -1*(k/m)X

        list(c(dX, dY))
    })
}
```

We run the simulation over a certain time interval with discreet differences between every time value. Finally, we run the `ode()` function to produce the discreet values.

```
# The sequence of time run in the simulation
times <- seq(0, 10, by=0.01)

# Generate curves that are solutions to the system of differential equations
out <- ode(y=state, times=times,func=system, parms=parameters)
```

Running the command above will generate data that is stored in `out`. We can plot this data out. From our derivation, we know that the value of $x_1 = x$. Therefore,

the solution to x_1 is equal to the solution of the second-order differential equation. We also plot how the two variables, x_1 and x_2, vary in comparison to each other. We plot the following:

```
plot(out[,"time"], out[,"X"], xlab="time", ylab="displacement", pch=19)
plot(out[,"X"], out[,"Y"], xlab="displacement", ylab="velocity", pch=19)
```

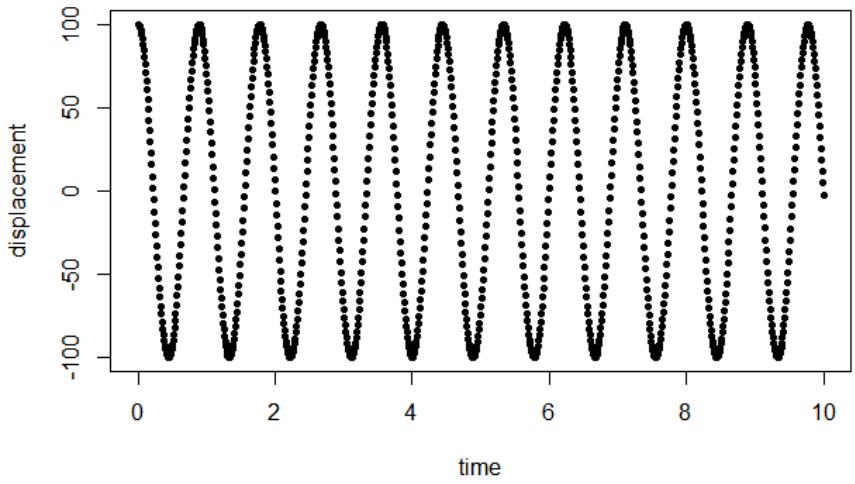


Figure 4.2: The displacement of the spring over time

4.4.2 Improving the System

However, this idealized system does not capture the motion of a spring that is dampened by other factors, such as heat loss by friction. Such a system is known as a **Dampened Mass-Spring Oscillator**. The spring looses momentum with relation to its velocity, which is the derivative of the position x. Therefore, we can add a new parameter to the second-order differential equation, b, that accounts for the dampening of the motion of the spring.

$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + kx = 0$$

Using the method described above to convert the second-order differential equation into a system of first-order differential equations, we get:

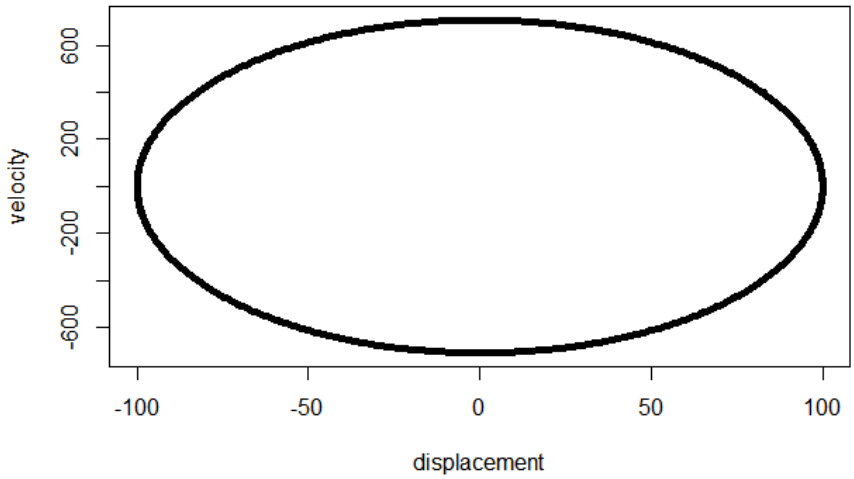


Figure 4.3: The relationship between x_1 and x_2

$$\frac{dx_1}{dt} = x_2$$
$$\frac{dx_2}{dt} = -\frac{k}{m}x_1 - \frac{b}{m}x_2$$

We modify our systems function in R to account for the velocity-dependent dampening of the spring as follows:

```
# m is the mass of the object (1 kg)
# b is the dampening of the object (0.25)
# k is the spring constant (50 kg / s^2)
parameters <- c(m = 1, b = 0.25, k = 50)

# Function that defines the system
system <- function(t, state, parameters) {
    with(as.list(c(state, parameters)), {
        dX <- Y
        dY <- -1*(k/m)X - (b/m)Y
    })
}
```

Running the simulation again, we get the graphs shown in figures 4.4 and 4.5. Notice that the range of the motion of the spring-mass system continually decreases as time passes. This accounts for much of the heat loss that a dampened spring experiences.

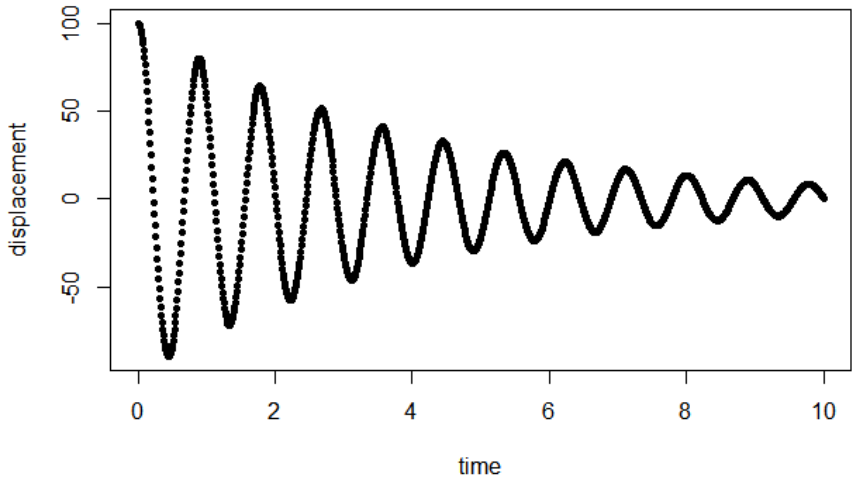


Figure 4.4: The displacement of the spring over time

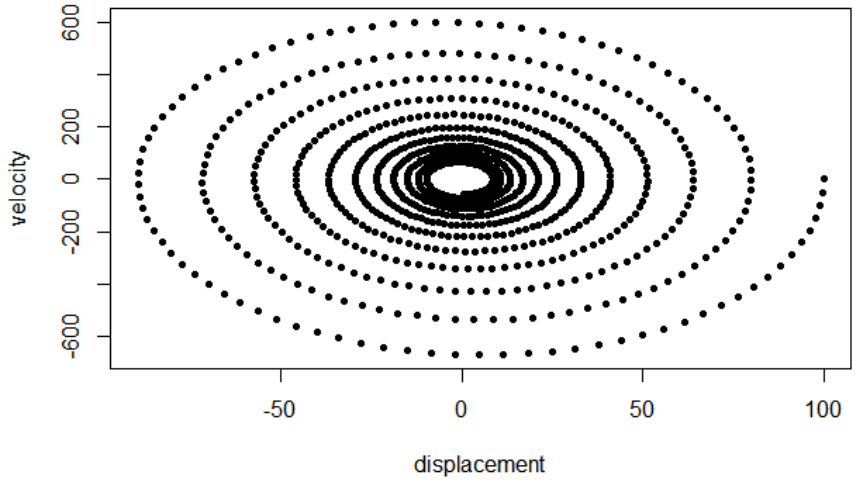


Figure 4.5: The relationship between x_1 and x_2

4.5 Student Assignment - Modelling of Genetic Processes

Many of the genetic processes that are studied in the field of molecular biology can be modeled using systems of differential equations. The student is tasked with understanding many of theses systems and modeling them. It is important for us to go over the *central dogma of molecular biology* before proceeding any further.

The three main components of this dogma are DNA, mRNA, and proteins. Proteins are generally the effectors in our cells, responsible for life-critical tasks, enzymatic processes, and regulation. Every protein has its own function and specificity. Ribosomal subunits, responsible for synthesizing proteins, get instructions for synthesis from polymer strands made of nucleotides known as mRNA. mRNA strands, in turn, are synthesized using instructions from another polymer known as DNA, which is the only heritable molecule present in the cell (as long as we ignore mitochondrial DNA that is passed down from a maternal lineage). We will also quickly review the processes of transcription and translation in depth. However, a strong foundation in biology is not necessary for modelling the problems that will follow.

- **Transcription:** Transcription is the process by which the information contained in a section of DNA (most often a gene) is transferred to a newly assembled piece of messenger RNA (mRNA). Transcription is triggered by the binding of RNA Polymerase to specific regions on the DNA called promoters, and is enabled/facilitated or impeded by a range fo promoter-specific proteins called transcription factors (13).

- **Translation:** During translation, proteins are produced by ribosomes that bind to a specific site of the mRNA (the ribosome binding site - RBS) and then move along the mRNA chain converting triplets of bases or "codons" on the mRNA into the appropriate peptide chain of amino acids defining the desired protein. The mRNA is read by the ribosome as triplet of bases, usually beginning with an AUG, or an initiator methionine codon downstream of the ribosome binding site. Complexes of initiation factors and elongation factors bring aminoacylated transfer RNAs (tRNAs) into the riboosome-mRNA complex, matching the codon in the mRNA to the anti-codon in the tRNA, thereby adding the correct amino acid in the growing peptide sequence encoding the emerging protein. As the amino acids are linked together into the growing peptide chain, they begin folding into the correct conformation. This folding continues until the nascent polypeptide chain is released from the ribosome as a mature protein (13).

4.5.1 Constitutive Gene Expression

Problem Statement: Model the system of constitutive gene expression in terms of the central dogma of molecular biology, mRNA degradation, and protein degradation.

Constitutive gene expression is the process that occurs when a gene is always on. That is, the gene is not regulated in any manner and simply follows the central

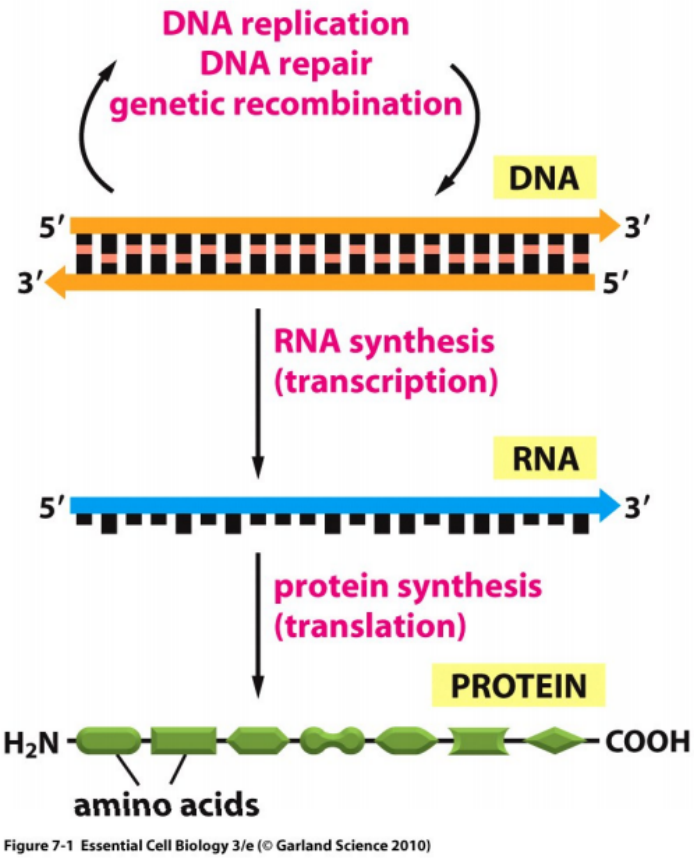


Figure 4.6: The Central Dogma of Molecular Biology

dogma of biology. In such a case, we have to consider a few rates that occur in the system.

- $\frac{dm}{dt}$ is the rate at which mRNA is synthesized from DNA. It will be one of the differential equations students will use in their system.
- $\frac{dp}{dt}$ is the rate at which the protein in question is synthesized from DNA. It will be the other differential equation students will consider in their system.
- k_m is the constitutive transcription rate. It is considered to be constant, and it represents the number of mRNA molecules produced per gene, per unit of time. We assume that there is only one copy of the gene in the cell. In the case of bacterial cells, where plasmid-located genes occur in multiple copies in different plasmids, we would multiply the rate k_1 by the number of copies N in the cell to obtain a proper rate.
- k_p is the translation rate. It is considered to be constant, and it represents the number or protein molecules produced per mRNA molecule, per unit of time.
- d_m is the mRNA degradation rate, measured as the frequency at which mRNA molecules degrade.
- d_p is the protein degradation rate, measured as the frequency at which protein molecules degrade.

Sample Student Solution

The student should be capable of modeling the system using the rates provided above. The system we are considering should account for transcription, translation, and the degradation of components in the process.

$$\text{Gene} \xrightarrow[\text{Transcr.}]{} \text{mRNA} \xrightarrow[\text{Transl.}]{} \text{Protein}$$

$$\text{mRNA} \xrightarrow[\text{Degrad.}]{} \emptyset$$

$$\text{Protein} \xrightarrow[\text{Degrad.}]{} \emptyset$$

Figure 4.7: The constitutive gene expression system

The rate at which mRNA is transcribed is equal to the rate provided to us, k_m. Furthermore, we account for the rate of degradation by multiplying the frequency rate d_m by the number of mRNA molecules in the system. The rate at which protein molecules are translated is equal to the rate k_p. Similar to the mRNA system, the degradation occurs as a product of d_p and the amount of protein produced.

$$\frac{dm}{dt} = k_m - d_m m$$
$$\frac{dp}{dt} = k_p m - d_p p$$

The student simply has to change certain aspects of the program written by the instructor. In this case, the student must define a new set of states, parameters, and a new system function.

```
library(deSolve)

parameters <- c(km = 10, kp = 5, dm = 1, dp = 3)
state <- c(M = 0, P = 0)

system <- function(t, state, parameters) {
    with(as.list(c(state, parameters)), {
        dM <- km - dm*M
        dP <- dp*M - dp*P

        list(c(dM, dP))
    })
}

times <- seq(0, 10, by=0.01)

out <- ode(y = state, times = times, func = system, parms = parameters)

# Plotting Functions
xaxis.range <- range(times)
yaxis.range <- range(min(range(out[,"M"]), range(out[,"P"])), max(range(out
    [,"M"]), range(out[,"P"])))
plot(out[,"time"], out[,"M"], xlim=xaxis.range, ylim=yaxis.range, col="red",
     xlab="Time", ylab="Concentration", main="System", pch=19)
points(out[,"time"], out[,"P"], col="blue", pch=19)
legend("bottomright", c("mRNA", "Protein"), pch=c(19, 19), col=c("red", "
    blue"))
```

We see that the concentration of proteins and mRNA in the system reaches a steady state with the initial values supplied. It is important for the student to understand that varying the parameters can change the solution to the system. For example, if the rate of degradation of a protein is much larger than the rate at which it is produced, we expect to see no overall protein production. Therefore, it is just as important to choose appropriate initial values as it is to produce a sound differential model. Although arbitrary values are chosen for these samples, students are encouraged to read literature in their field of research or study to find common values for such problems. For instance, the average half life of mRNA in the model system *E.coli* is around 5 minutes. Such information is inconsequential to producing an appropriate model.

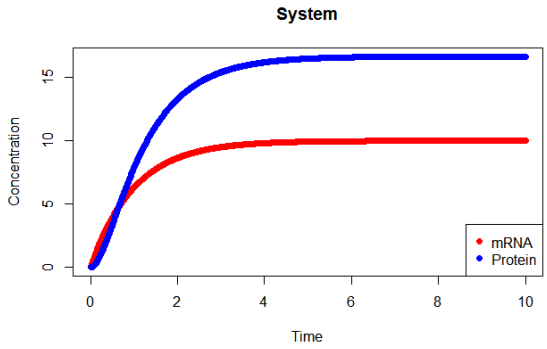


Figure 4.8: Constitutive gene expression concentrations in the system

4.5.2 Gene Transcription Regulation

Interestingly, most genes that are known today do not follow the constitutive gene expression model. The constant production of proteins can be unnecessary, if not harmful, to the proper function of the organism. Therefore, multiple layers of regulation occur to ensure that the gene produces proteins at the correct time and the proper rate. There are two major arguments for the necessity of gene regulation in cellular and molecular biology. Firstly, all cells carry the same DNA (and therefore, the same heritable information) but have different developmental stages and functions. Secondly, cells are subject to many environmental perturbations, which can only be dealt with by regulation.

Proteins known as transcription factors are generally responsible for the regulation of genes. Transcription factors attach to a region upstream of the gene known as the promoter. The transcription factor can either be an activator (promotes the transcription of the gene) or a repressor (inhibits the transcription of the gene).

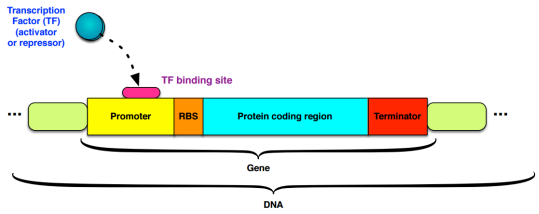


Figure 4.9: Action of regulative transcription factors on protein-encodoing DNA regions

Activators and Repressors

Problem Statement: Model the case of a gene whose transcription is activated by the binding of an activator/repressor to its transcription factor binding site.

A gene begins transcription once an activator binds to its corresponding transcription factor binding site. Once activated, the gene goes through the same process that it would during constitutive gene expression. These processes are

inhibited once a repressor binds to the transcription factor binding site. This process as a whole is the basis of gene regulation. Because of the fact that, save for the introduction of activators and repressors, the process that the gene undergoes is identical to constitutive gene expression, many of the rates from the constitutive gene expression section are the same in this case. These include $\frac{dm}{dt}$, $\frac{dp}{dt}$, k_m, k_p, d_m, and d_p. With the introduction of a new parameter (an activator or a repressor), there are subsequently new variables added to this model. The repressor and activator cannot occur at the same time, as the gene can only be activated or inhibited at a certain point in time. These are as follows.

- A represents the activator.
- R represents the repressor.
- K represents the activation or repression coefficient.
- n represents the Hill coefficient, or the number of activators/repressors that need to cooperatively bind to the promoter (transcription factor binding site) in order for gene expression to be activated/inhibited.

Sample Student Solution

The student should be capable of modeling the system using the rates provided in the constitutive gene expression section, as well as using the new variables provided above. The system we are considering should account for transcription, translation, and the degradation of components in the process.

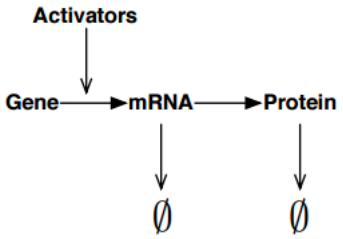


Figure 4.10: The activation and subsequent activities of a gene

The differential equations used in this model are almost identical to the differential equations used in previous models (the constitutive gene expression model). The only difference is the introduction of the activator/repressor and its coefficients, which is then multiplied by k_m.

These are the new differential equations with the introduction of the activator:

$$\frac{dm}{dt} = k_m \frac{A^n}{K^n + A^n} - d_m m$$

$$\frac{dp}{dt} = k_p m - d_p p$$

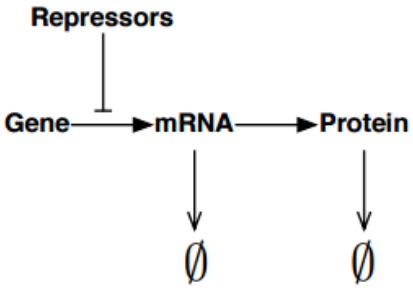


Figure 4.11: The inhibition and subsequent activities of a gene

These are the new differential equations with the introduction of the repressor:

$$\frac{dm}{dt} = k_m \frac{K^n}{K^n + R^n} - d_m m$$
$$\frac{dp}{dt} = k_p m - d_p p$$

As with other cases, the student must define a new set of parameters and system functions in the code. Most of the code remains the same. The following shows the code when using the activator differential equations.

```
library(deSolve)

parameters <- c(km = 10, kp = 5, dm = 4, dp =  4, A = 0.25, K = 0.01, n = 4)
state <- c(M = 0, P = 0)

system <- function(t, state, parameters) {
    with(as.list(c(state, parameters)), {
        dM <- km*(A^n/(K^n+A^n)) - dm*M
        dP <- dp*M - dp*P

        list(c(dM, dP))
    })
}
```

4.5.3 Regulation

Unfortunately, by simply solving each system individually, it is hard to tell how activation and inhibition affect gene expression models. With the ability to model constitutive gene expression along with activation and regulation, we can compare models where activation and regulation does occur. Using the following values, we can create three concurrent systems and compare them:

```
# New parameters in the code
parameters <- c(km = 0.346, kp = 6.931, dm = 0.138, dp = 0.017, A = 200, R =
    200, K = 100, n = 2)
state <- c(M = 0, P = 0)
```

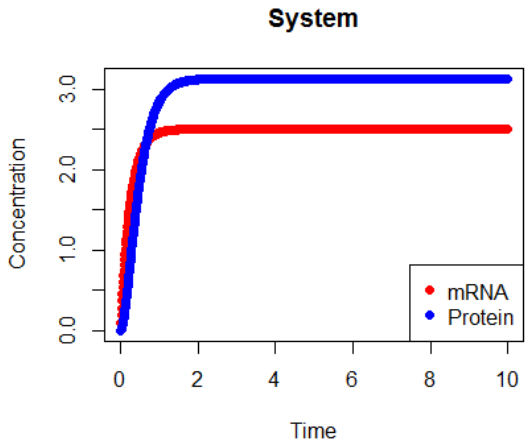


Figure 4.12: Gene expression in a system containing activators

```
4
5  # Constitutive Gene Expression Model
6  system.const <- function(t, state, parameters) {
7      with(as.list(c(state, parameters)), {
8          dM <- km - dm*M
9          dP <- kp*M - kp*P
10
11         list(c(dM, dP))
12     }
13 }
14
15 # Activation in Gene Expression
16 system.act <- function(t, state, parameters) {
17     with(as.list(c(state, parameters)), {
18         dM <- dm*(A^n / (K^n + A^n)) - dm*M
19         dP <- kp*M - dp*P
20
21         list(c(dM, dP))
22     })
23 }
24
25 # Inhibition in Gene Expression
26 system.rep <- function(t, state, parameters) {
27     with(as.list(c(state, parameters)), {
28         dM <- kM*(K^n / (K^n + R^n)) - dm*M
29         dP <- kp*M - dp*P
30
31         list(c(dM, dP))
32     })
33 }
34
35 # Independent variable
36 times <- seq(0, 400, by=1)
37
38 # Computing the solution to all three models
```

```
39 out.const <- ode(y = state, times = times, func = system.const, parms =
       parameters)
40 out.act <- ode(y = state, times = times, func = system.act, parms =
       parameters)
41 out.rep <- ode(y = state, times = times, func = system.rep, parms =
       parameters)
42
43 # Plotting functions
44 plot(out.const[,"time"], out.const[,"P"], col="red", xlab="Time", ylab="
       Concentration", main="System", pch=19)
45 points(out.act[,"time"], out.act[,"P"], col="green", pch=19)
46 points(out.rep[,"time"], out.rep[,"P"], col="yellow", pch=19)
47 legend("bottomright", c("Protein CONST", "Protein ACT", "Protein REP"), pch=
       c(19, 19, 19), col=c("red", "green", "yellow"))
```

The code will produce the graphics shown in figure 4.13. Notice that the constitutive protein expression runs unaffected. The activation gene expression curve takes longer to achieve expression because it takes time for activators to act as transcription factors before transcription can take place. Furthermore, the repressed curve is much lower than both, as is expected from such systems.

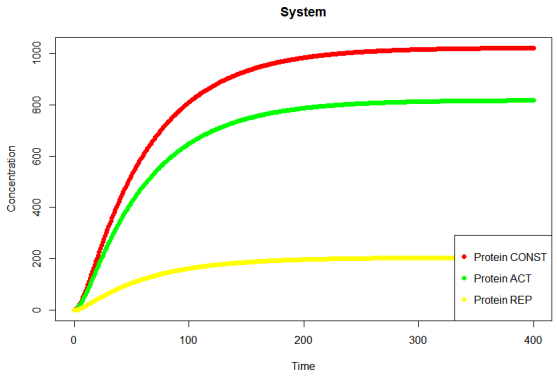


Figure 4.13: Comparing constitutive, activation, and inhibition in gene expression.

4.6 Conclusion

Differential equations are used in a variety of different areas to model a wide range of functions, such as the rate that a protein is transcribed or the rate that a population grows. The main function of a differential equation is to model a rate of change. Some of these equations are easily solved by hand, whereas others are more complicated; the `deSolve` package in R serves to solve and graph a system of differential equation once certain parameters and initial values are entered. Through learning and using `deSolve`, it is possible to easily solve a system of complex differential equations, as well as view the results graphically and be able to change parameters and values as needed. This streamlines a process of solving equations that occur within all types of science, math, and other processes.

4.7 Acknowledgements

We would like to thank the creators of the `deSolve` package, Karline Soetaert, Thomas Petzoldt, and R. Woodrow Setzer of the Royal Netherlands Institute of Sea Research. We would also like to thank Mr. Robert Gotwals and the North Carolina School of Science and Mathematics (NCSSM).

5. Clustering

Often, we are not only interested in specific data points, but also in the relationships between those data points. Cluster analysis refers to the grouping of data points into *clusters* based on similar characteristics. R has variety of libraries that support cluster analysis. We will use `cluster`. `datasets` for this lesson, in which we will analyze clustering in data about crime rates in US cities. (7)

Here are some typical commands used in cluster analysis:

1. `scale`
2. `k.means.fit`
3. `attributes`
4. `wssplot`
5. `clusplot`
6. `dist`
7. `hclust`
8. `groups`
9. `rect.hclust`
10. `table`

For this study, you will learn a basic framework for cluster analysis using two different methods, K-means and hierarchical clustering. Each method produces a different type of cluster graph of the same data.

- Your first task is to graph a set of data using the K-means clustering method.
- Your second task is to graph a set of data using the hierarchical clustering method.

5.1 Building the Model

You should begin by copying and pasting the teaching code into RStudio. This will serve as a foundation upon which to build your code. We have included comments to guide you as you complete the code by adding commands. When you successfully complete the code, it should output two graphics, shown in figures 5.1 and 5.2.

5.1.1 Programming Hints

1. Different datasets will often require different clustering methods. Therefore, not all clustering methods will work (or at least, work well) for every dataset! Be prepared to use a variety of clustering methods in the future.
2. When installing packages, install them in the "console" window of RStudio.
3. If a piece of code is indented in the teaching or student code, it must also be indented in the code that you write. This is important!

5.2 Deliverable

The final product that you create should consist of one K-means cluster graph composed of groups of data points embedded in ovals and one hierarchical tree plot.

5.3 Teaching Code

```
1
2  # Your Name
3  # Date
4  # Script name: dataset.R
5  # Objective of script
6  #
7  # set working directory where results will be saved
8  setwd("file location")
9  #
10 # install the necessary package and datasets, then read in library
11 install.packages("data package")
```

```
# this particular step should be completed in the console window
library(data package)
data(dataset, package = 'data package')
# "dataset" is the name that you choose to call your specific dataset
head(dataset)
#
# now we will begin to complete the first objective
#
# begin by standardizing the variables
dataset <- scale(original.dataset.name[-1])
#
# this next piece of code is the K-means function, where the number
# # symbol denotes the number of cluster groups
k.means.fit <- kmeans(dataset, #)
#
# this code segment includes all of the outputs of the k.means.fit function
attributes((k.means.fit))
#
# this next piece of code denotes the centroids
k.means.fit$centers # a nice table should result
#
# this piece of code shows the clusters
k.means.fit$cluster
#
# this code string denotes the size of the clusters
k.means.fit$size
#
# now we will begin to complete the first objective
#
# load the cluster library
library(cluster)
#
# this next section of code creates the a cluster graph using the K-means
    method
clusplot(dataset, k.means.fit$cluster, main = "Name of Cluster Graph", color
     = TRUE,
shade = TRUE, labels = 2, lines = 0) # look at those nice colorful ovals!
#
# this next line creates a confusion matrix which evaluates the performance
    of the clustering
table(dataset[,1],k.means.fit$cluster)
#
# Okay, now that you've mastered k-means clustering,
# let's move on to hierarchical clustering
#
# this code segments defines the distance "d", which in
# this case is Euclidean, for the clustering algorithm
d <- dist(us.crime, method = "euclidean")
#
H.fit <- hclust(d, method = "ward.D")
#
# this displays the dendrogram
plot(H.fit)
#
# this separates the diagram into 4 groups...
groups <- cutree(H.fit, k=4)
```

```
#
# ... while this piece of code creates a red border around those same groups
rect.hclust(H.fit, k=4, border = "blue")
#
# lastly, another confusion matrix is used to evaluate the clustering
table(us.crime[,1],groups)
#
# end code
```

5.4 Example Student Code

```

# Katherine L. Bennett
# September 1, 2016
# Script name: crime.R
# The purpose of this script is to analyse a sample of U.S. city crime
# using two cluster analysis methods: K-means and Hierarchical
#
setwd("/Users/Katherine/Desktop/RFiles")
#
library(cluster.datasets)
data(us.crime, package = 'cluster.datasets')
head(us.crime)
#
us.crime <- scale(sample.us.city.crime.1970[-1])
#
k.means.fit <- kmeans(us.crime, 4)
#
attributes((k.means.fit))
k.means.fit$centers
k.means.fit$cluster
k.means.fit$size
#
library(cluster)
#
# the following two lines should be one line in your code
# we have separated them in this guide for the reader's convenience
#
clusplot(us.crime, k.means.fit$cluster, main = "U.S. City Crime Cluster
    Representation", color = TRUE,
shade = TRUE, labels = 2, lines = 0)
#
table(us.crime[,1],k.means.fit$cluster)
#
# hierarchical cluster method
#
d <- dist(us.crime, method = "euclidean")
H.fit <- hclust(d, method = "ward.D")
plot(H.fit)
groups <- cutree(H.fit, k=4)
rect.hclust(H.fit, k=4, border = "blue")
table(us.crime[,1],groups)
```

```
#
# end code
```

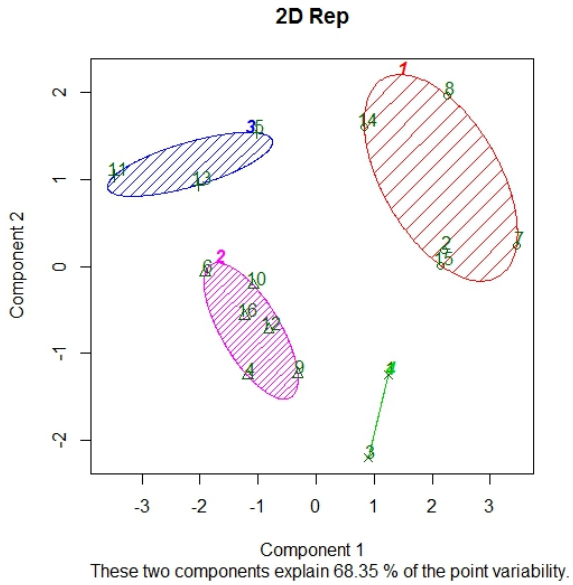
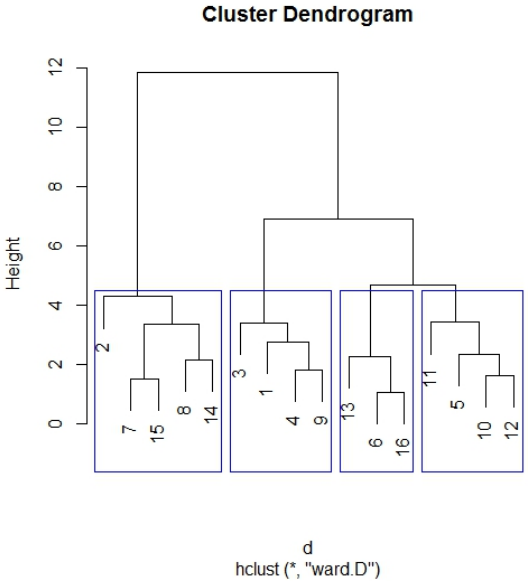


Figure 5.1: K-means Clustering Method Result

Figure 5.2: Hierarchical Clustering Method Result



5.5 Further Readings

There are a wide variety of sources available that cover cluster analysis. We recommend chapter 8 of *Introduction to Data Mining* by Tan, Steinbach, and Kumar (2006). A link to the chapter is provided below:

`https://www-users.cs.umn.edu/~kumar/dmbook/ch8.pdf`

6. Microseq

6.1 Introductory Reading

The conversion of working DNA to proteins is one of the most important processes in biology. After DNA undergoes transcription and becomes an mRNA sequence, it can be turned into a string of amino acids via sequences known as codons. Each codon consists of a combination of three nucleotides, and there are twenty amino acids that match with these combinations.

DNA and all DNA processes are integral to scientists' understanding of biology. Studying this molecule is a fascinating and essential task, but because of the large quantities of data that genetics studies can obtain, manually analyzing a sequence can prove to be insignificant. Bioinformatics is a relatively recent field that combines computational science, statistics, engineering, and biology in order to interpret this type of data. Being able to use technology to aid in the analysis of different sequences can greatly lessen the tediousness of translating sequences and manually visualizing data. It also eliminates the possibility of human error, and allows genetics research to move much faster.

For example, existing algorithms, such as JASPAR and STARR-seq, are databases or functional tools that allow bioinformaticians to analyze sequence data. These programs can do things such as identify even remotely homologous sequences from a DNA multiple sequence alignment, or identify *cis*-regulatory regions throughout the genome. (5) The focus of this case study are R packages known as Biostrings and microseq, which, in conjuction, allow for sequence data analysis.

6.2 Objectives of the Case Study

The objective of this study is to use the Biostrings and microseq packages in conjunction to translate a plain sequence format of DNA.

6.2.1 Objective 1

Translation of DNA to RNA is a vital biological process, and using the Biostrings and microseq functions will allow for a significantly streamlined process that will aid projects or studies in need of translated DNA. The objective is to obtqain an amino acid sequence translated automatically from the given DNA sequence.

6.3 Building the Model

Tips for building the model are located in the comment sections of the code.

If the code is written properly, the output should be a list with the frequencies of each amino acid in the DNA sequence. The output should look similar to figure 6.1.

```
> alphabetFrequency(dn, as.prob=TRUE)
              A          R          N          D          C          Q          E          G          H
[1,] 0.03918031 0.04961676 0.03601664 0.023308 0.03593311 0.03796178 0.03690582 0.05007206 0.03404047
              I          L          K          M          F          P          S          T          W
[1,] 0.05424414 0.1079028 0.05324396 0.01835742 0.05273559 0.04981224 0.08967853 0.05192683 0.01759073
              Y          V U O B J Z          X          * - + . other
[1,] 0.03112627 0.0520845 0 0 0 0 0 0 0.02775455 0.05050743 0 0 0     0
```

Figure 6.1: Frequency of amino acids

6.4 Deliverable

The student should submit a list containing the frequencies of each amino acid in the sequence analyzed. This list can be compared to the instructor's frequencies list to check for accuracy.

6.5 Teaching Code

You should also put any notes, teaching suggestions, etc. in this space!

```
# Name
# Date
# Name of script
# Objective of script

# clean up and set working directory
rm(list=ls())
setwd()

### download data set from
ftp://ftp.ensembl.org/pub/release-85/fasta/mus_musculus/dna/

### install Biostrings from
```

```
https://bioconductor.org/packages/release/bioc/html/Biostrings.html
### then install microseq

#load  microseq and Biostrings library
library()
library()

# view accepted IUPAC DNA letters
# these are the values that the functions will accept and put out
DNA_BASES
DNA_ALPHABET

# obtain plain sequence format from .txt file on desktop or source
# turn fasta into string
d <- readDNAStringSet("data set")
# translate object to amino acid sequence
dn <- translate(data set, genetic.code=GENETIC_CODE, if.fuzzy.codon = "solve
    ")
#"solve" allows unidentifiable codons to be represented as *

#characteristics of DNA strings
d
summary(data set)
dn
summary(translated data set)

#frequency of each letter or present symbol
alphabetFrequency(translated data set)

#frequency values
alphabetFrequency(translated data set, as.prob=TRUE)

### EXTRA CREDIT!
letterFrequency(translated data set, "A", as.prob=TRUE)
letterFrequency(data set[[1]], letters="ACGT", OR=0)

# What do these two functions show?
#show frequency of base "A" in translated data set, and quantity of "ACGT"
    in
original data set
```

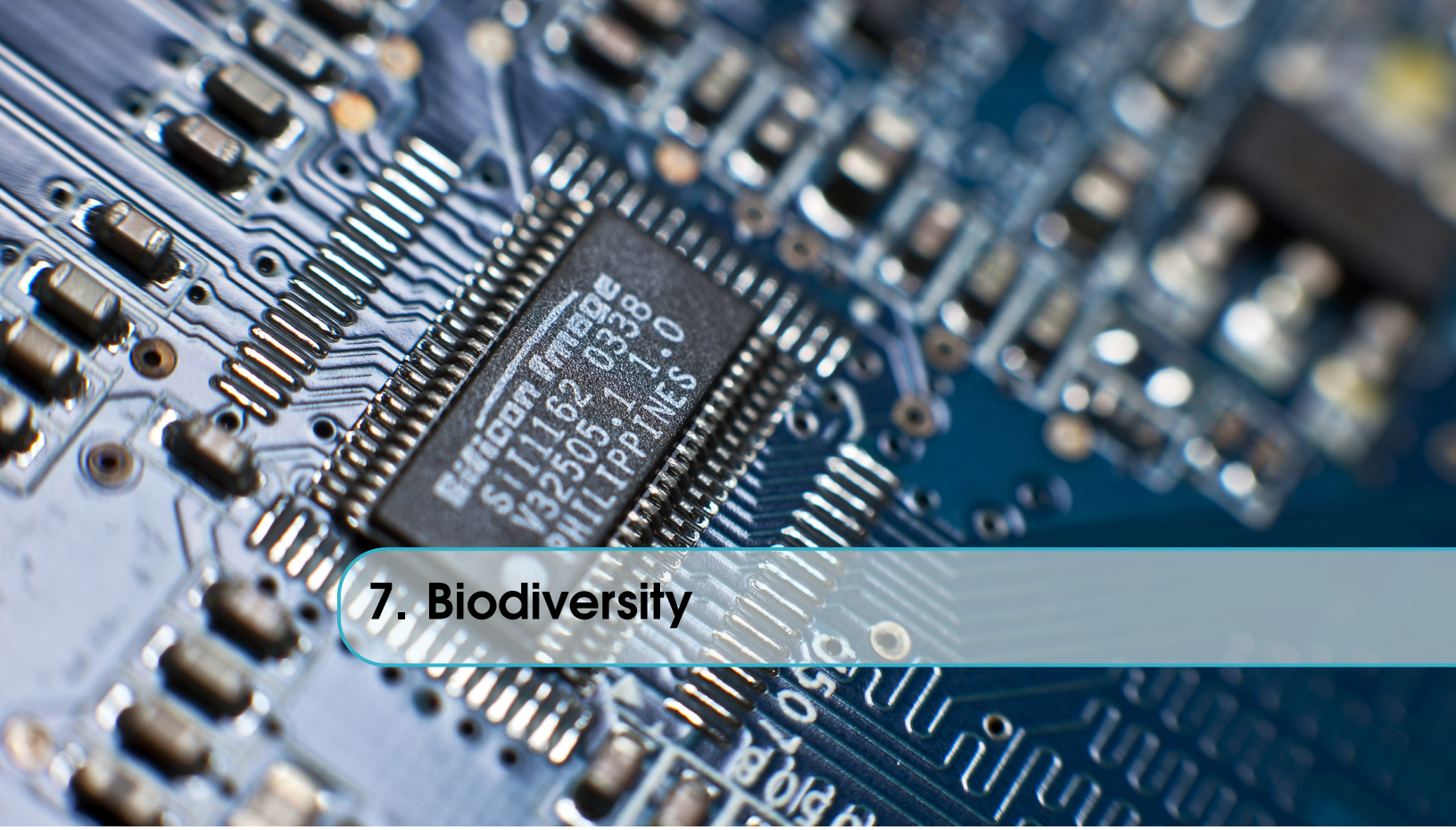
6.6 Example Student Code

```
# Christa Parrish
# September 8, 2016
# translate.R
# this script translates DNA sequence to Amino Acid sequence and observes
    frequencies
of letters

# clean up and set working directory
rm(list=ls())
```

```
setwd("~/Desktop/R")

### download data set from ftp://ftp.ensembl.org/pub/release-85/fasta/mus_
    musculus/dna/

#load  microseq and Biostrings library
library(microseq)
library(Biostrings)

#view accepted IUPAC DNA letters
DNA_BASES
DNA_ALPHABET

# obtain plain sequence format from .txt file on desktop or source,
    whichever works
# turn fasta into object
d <- readDNAStringSet("MusMusculus.fa")
# translate object
dn <- translate(d, genetic.code=GENETIC_CODE, if.fuzzy.codon = "solve")
#"solve" allows unidentifiable codons to be represented as *

#characteristics of DNA strings
d
summary(d)
dn
summary(dn)

#frequency of each letter or present symbol
alphabetFrequency(dn)

#frequency values
alphabetFrequency(dn, as.prob=TRUE)

### EXTRA CREDIT!
letterFrequency(dn, "A", as.prob=TRUE)
letterFrequency(d[[1]], letters="ACGT", OR=0)

# What do these two functions show?
```

6.7 Further Readings

By looking at the package manual online, students can read about the other functions of the microseq and Biostrings packages.



7. Biodiversity

7.1 Introduction to Biodiversity

The 21st century marks the era of big data. This increase of data has effected all spheres of science, especially the study of biodiversity. Biodiversity data covers an increasingly wide range of areas including bio-geography, ecology, invasive species biology, and climate change. The information collected in biodiversity informatics projects has been used for studies relating to food security, control of disease vectors, and marine productivity (2).

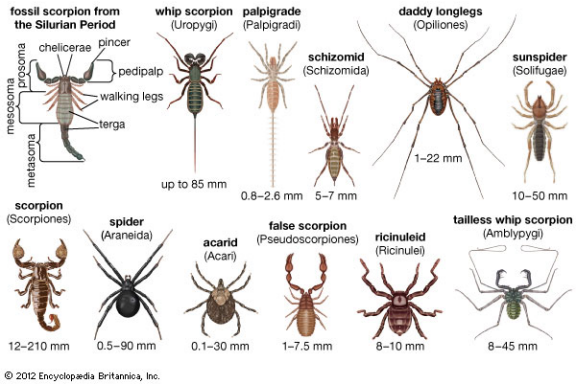


Figure 7.1: Example of biodiversity in arachnids

Primary biodiversity data is one of the most important biodiversity statistics; it documents a species' occurrences in time and space. Associated metadata, or data that describes statistics given in another set, may include information about who identified and recorded the species, who verified the identification, climatic

parameters, micro-habitat information, number of individuals, sex, size, etc (2). When these two types of data are formatted in universally accepted formats, published openly and integrated with other such data streams, they are known "Digitally Accessible Knowledge" or DAK.

There are many sources of biodiversity data including: directed surveys, broad-scale surveys, and biological collections.

7.1.1 Direct Surveys

Directed surveys are often used where prior knowledge of a given system or biological mechanism exists. Surveys for this type of experiment are designed to control for known sources of variation(8). Direct surveys aim to establish a casual relationship between some experimental treatment and its effect and are widely accepted in the scientific community. Regardless, the data collected in this type of survey is extremely expensive. Additionally, the data is aggregated by many researchers working independently in small areas over small time scales, which creates a network of heterogeneous data repositories with little opportunity for integration and eventual data degradation.

7.1.2 Broad Surveys

Broad-scale surveys involve over ten million observations annually. This data is used to generate probabilistic estimates of species occurrence, which can then be used to determine general patterns over broad geographic regions and time scales (8). Data for broad surveys can be collected by anyone, ranging from trained observers to interested citizens. As a result, these surveys are relatively inexpensive to operate; however, the regulations behind these surveys are more relaxed, resulting in more potential for bias. Nonetheless, these studies provide the bulk of non-specimen observational data available via the Internet (10).

7.1.3 Biological Collections

Biological collections are data sets of specimens. This type of data is primarily assembled to document the phenotypes and genotypes of the biotas worldwide (2). Biological collections are typically considered as "high-quality, low-volume" sources of data on biological diversity (2).

Given all the different ways to collect data, each with their own pros and cons, it is important for researchers utilizing biodiversity data to determine the completeness and accuracy of their data sets. Data visualization is one technique to observe patterns within very large data sets that are extremely difficult to analyze manually. The goal of this study is to learn how analyze biodiversity data sets and visualize these sets in order to paint a comprehensive representation of the data.

7.2 Objectives of the Case Study

The bdvis packages allows users to create models that visualize the biodiversity of the data collected. Using data that catalogs the occurrences of the species in

question, bdvis creates a heat-map with geography superimposed for every point on the globe where the species was recognized, create a temporal visualization of the data by day, week, or month, and create a chronohorogram which plots the number of records on each day with colors indicating the value, and concentric circles for each year. The models that are created not only show the user if there is a lack of biodiversity, or lack of data, but also exactly where this lack occurs. The user can also see variations in the data on different months, and days. This variation in data may be due to differences in temperature.

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

1. Create a map of completeness of biodiversity
2. Visualize geographical and temporal coverage
3. Visualize gaps in data

7.3 Building the Model

Before loading you data into RStudio, it is necessary to install the packages: `maps`, `sqldf`, `plotrix`, `treemap`, `plyr`, `ggplot2`, `grid`, `lattice`, `chron`, `devtools`, `bdvis`, and `rinat`. After installing these packages, load them into your library.

```
#make sure to include quotation marks when installing packages
install.packages("maps")
```

```
#leave off quotation marks when loading packages into library
library(maps)
```

Download the "Historical bird ringing records" from: `http://www.gbif.org/dataset/b4ae1720-1431-49ee-bfeb-8146fc42b1a3`. After data is configured to fit the bdvis package, set the working directory to the location of the data on your laptop. Next, read the data from the file, making sure to set `stringsAsFactors` as false.

```
#set working directory to where data is located
setwd("/Users/margauxwinter/Desktop/Rfolder/dwca-safring-v1.0")

#read your data into RStudio
birds <- read.delim('occurrence.txt', quote='', stringsAsFactors=FALSE)
```

You may begin using functions that create visual models of the data. The first model used in this example is a graph of completeness vs. number of species. To create this graph use the function bdcomplete, which divides the extent of the dataset in cells, via the getcellid function. Next, the function calculates the Chao2 estimator of species richness, which uses occurrence data from many samples to estimate the entirety of the species diversity. This function requires a minimum

number of records that must be present on each cell to properly compute the index. If there are too few records in the cells, the function is unable to finish, an error will result. Use the following code to generate the plot shown in figure 7.2.

```
comp=bdcomplete(birds)
```

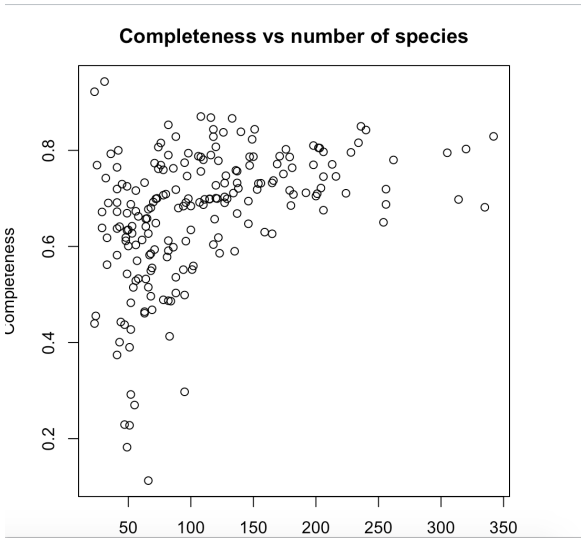


Figure 7.2: Completeness

Next use the function mapgrid, which builds a grid map colored according to the density of records in each cell. Mapgrid allows the user to visually see gap and blank spaces in data. Grids are 1-degree cells, build with the getcellid function. Record-density maps apply a color gradient according to the number of records in the cell. Species-density maps apply a color gradient according to the number of different species in the cell, disregarding the number of records. Completeness maps apply a color gradient according to the completeness index, from 0 (incomplete) to 1 (complete). Use the following function to generate a map grid as shown in figure 7.3

```
mapgrid(comp,ptype='complete')
```

In order to create a chronohorogram of the dates in the provided data set, use the function chronohorogram. A chronohorogram plots the number of records on each day with colors indicating the value and concentric circles for each year. The following code is used to create a chronohorogram like the one shown in figure 7.4.

```
chronohorogram(birds)
```

Use the function distrigraph to build plots displaying distribution of biodiversity records among user-defined features. The next four distrographs all use different

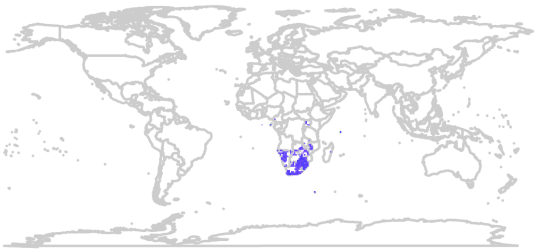
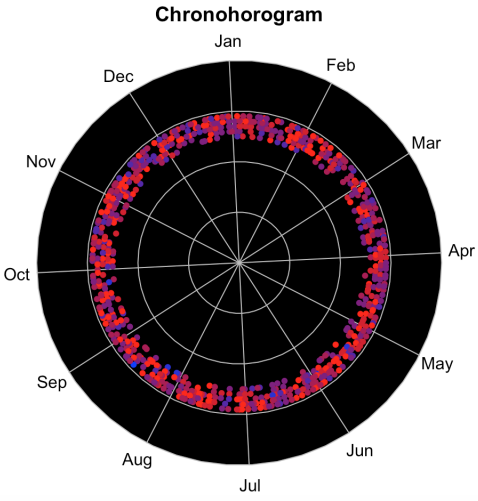


Figure 7.3: Mapgrid

Figure 7.4: Chronohorogram



p-types (feature to represent) allowing each graphic to depict different aspects of the data. The following code is used to create all four distirgraph respectively as shown in figures 7.5 and 7.6.

```
distrigraph(birds,ptype="cell",col="tomato")
distrigraph(birds,ptype="species",ylab="Species")
distrigraph(birds,ptype="efforts",col="red")
distrigraph(birds,ptype="efforts",col="red",type="s")
```

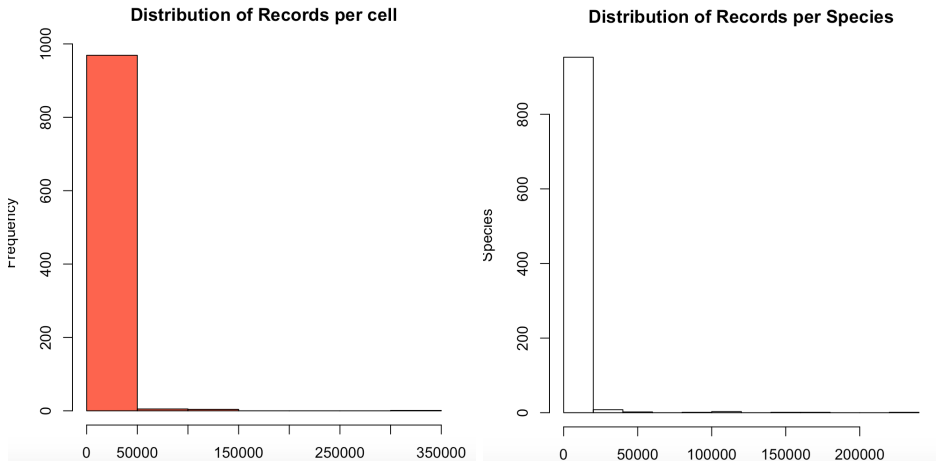


Figure 7.5: Left: Distrigraph 1, Right: Distrigraph 2

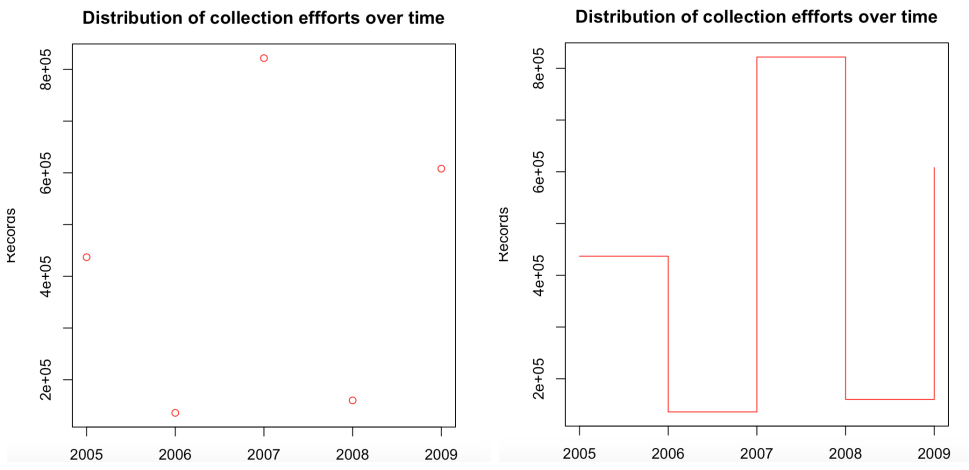


Figure 7.6: Left: Distrigraph 3, Right: Distrigraph 4

Bdcalendarheat create a calendar heat map in which a calendar is structured as a matrix-like plot where each cell represents a unique date. The cells are colored as to show the density of records for that date. Use the following function to create the heat map shown in figure 7.7.

```
bdcalendarheat(birds)
```

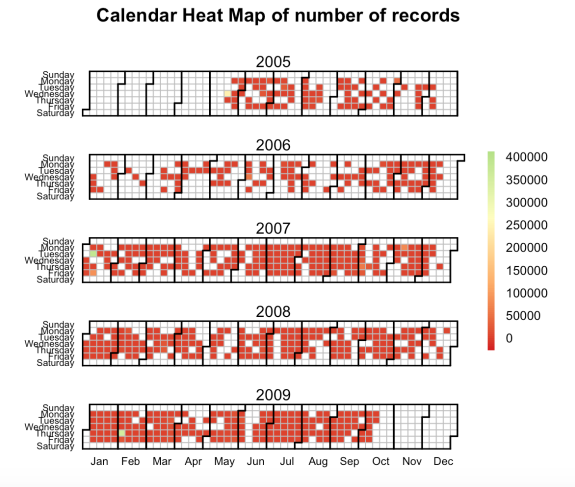


Figure 7.7: Heat Calendar

7.3.1 Programming Hints

Some functions, such as mapgrid, are customizable. Arguments within the function, including `indf`, `ptype`, `title`, `bbox`, `legscale`, `collow`, `colhigh`, `mapdatabase`, `region`, and `customize`, allow the map to be configured to fit the data as desired. Using the function `help()` for mapgrid, and other functions, within the console of RStudio gives a more comprehensive overview of these functions.

Other functions can be used within the package bdvis. These functions include `tempolar`, `gettaxo`, and `taxotree`. Each function provides a different view of the data by compiling data that are not used in the other models, and by allowing users to view the same data through different means.

7.4 Teaching Code

This is the code shown in the above examples:

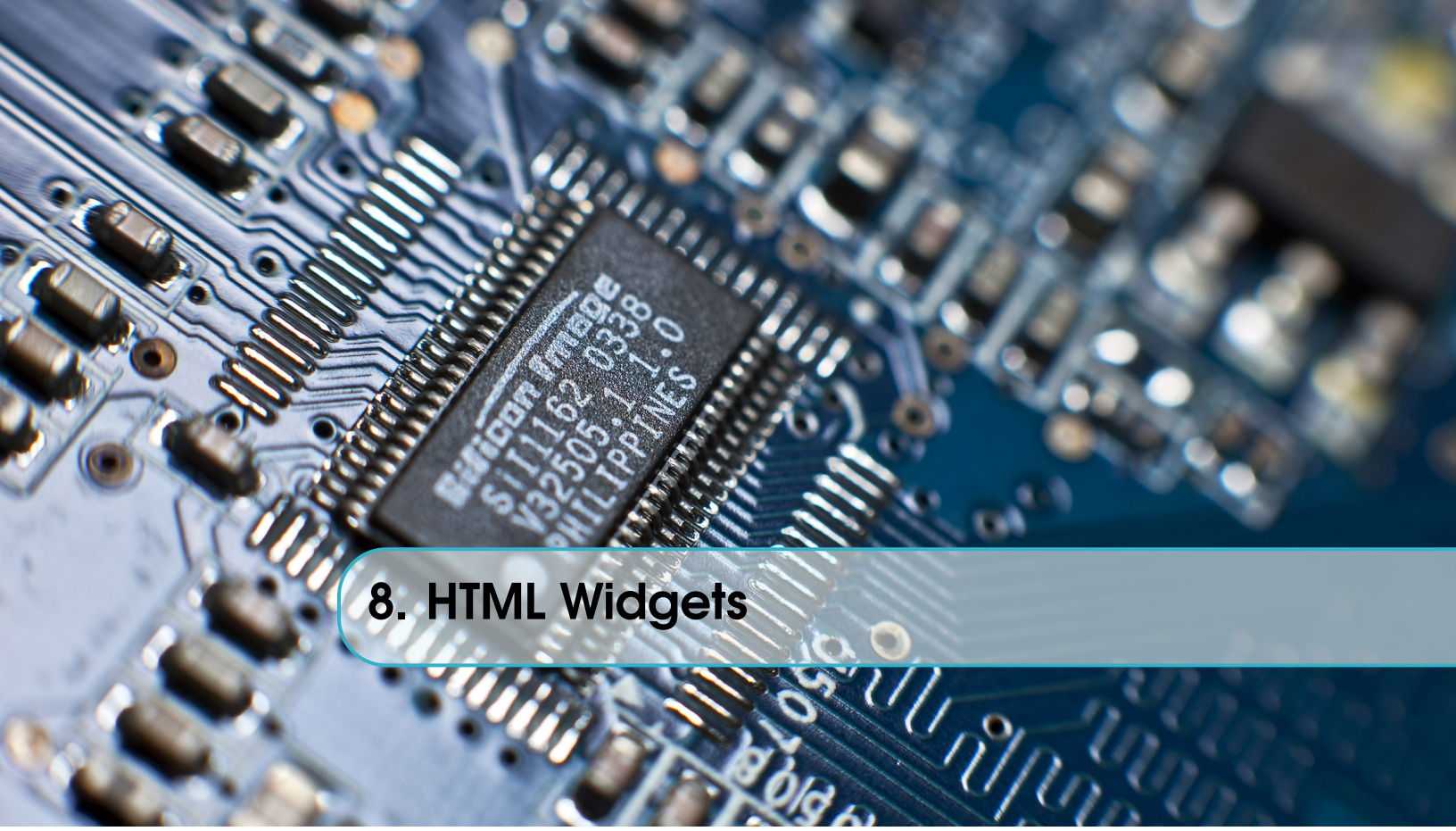
```
#set working directory to data location
setwd("/Users/margauxwinter/Desktop/Rfolder/dwca-safring-v1.0")

#read data into RStudio
birds <- read.delim('occurrence.txt', quote='', stringsAsFactors=FALSE)

#configure data
conf <- list(Latitude='decimalLatitude',Longitude='decimalLongitude',
Date_collected='eventDate',Scientific_name='specificEpithet')
birds <- format_bdvis(birds, config=conf)
comp=bdcomplete(birds)
```

```
12
13 #visualize biodiversity data on the world map
14 mapgrid(comp,ptype='complete')
15
16 #create chronohorogram
17 chronohorogram(birds)
18 comp=bdcomplete(birds,recs=5)
19
20 #create various distrigraphs
21 distrigraph(birds,ptype="cell",col="tomato")
22 distrigraph(birds,ptype="species",ylab="Species")
23 distrigraph(birds,ptype="efforts",col="red")
24 distrigraph(birds,ptype="efforts",col="red",type="s")
25
26 #create heat calendar
27 bdcalendarheat(birds)
```

7.5 Example Student Code

The final deliverable should be submitted as visual models of the data in the form of distropraphs, maps, chronohorograms, and plots. Use the arachnid data set downloaded from: `http://www.gbif.org/occurrence/download/0012867-160910150852091`

```
1 #set working directory
2 setwd("/Users/margauxwinter/Desktop/Rfolder/
3 dwca-arc_national_survey_of_arachnida-v1.1")
4
5 #read in data set
6 spider <- read.delim('occurrence.txt', quote='', stringsAsFactors=FALSE)
7
8 #configure data to make it readable for bdvis package
9 conf <- list(Latitude='decimalLatitude',Longitude='decimalLongitude',
10 Date_collected='eventDate',Scientific_name='specificEpithet')
11 spider <- format_bdvis(spider, config=conf)
12 comp=bdcomplete(spider)
13
14 #begin functions for data visualization
15 mapgrid(comp,ptype='complete')
16 tempolar(spider, color="green", title="Arachnida daily", plottype="r",
    timescale="d")
17 tempolar(spider, color="blue", title="Arachnida weekly", plottype="p",
    timescale="w")
18 tempolar(spider, color="red", title="Arachnida monthly", plottype="r",
    timescale="m")
19 chronohorogram(spider)
20 comp=bdcomplete(spider,recs=5)
21 distrigraph(spider,ptype="cell",col="tomato")
22 distrigraph(spider,ptype="species",ylab="Species")
23 distrigraph(spider,ptype="efforts",col="red")
24 distrigraph(spider,ptype="efforts",col="red",type="s")
```



8. HTML Widgets

8.1 Introduction

The ability to display statistical data is important in visualizing the information that is generated in scientific studies and research. Although R provides a method for visualizing data using graphs and plots, it is often challenging to represent demographic data. `htmlwidgets`, especially the `leaflet` package, is designed to address this concern. Furthermore, `leaflet` and any other `htmlwidgets` package can be integrated with `shiny` to produce a web application that fluidly portrays pertinent information.

8.2 Installation

The following code can be used to install the libraries that will be used in the demonstrations that follow.

```
1                > install.packages("shiny")
2                > install.packages("leaflet")
```

If the installation results in warnings about the R version you are using, please consider updating for the latest functionality.

The installation, usage, and functionality of `ShinyApp` is expanded upon separately in another chapter of this R manual. Please consult that section of the book before continuing with the demonstrations outlined in this chapter. Although `leaflet` can be used in R without `ShinyApp` integration, the accessibility and functionality of these widgets is best used as a Web application.

8.3 Leaflet

Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. It is used by websites ranging from The New York Times and The Washington Post to GitHub and Flickr, as well as GIS specialists like OpenStreetMap, Mapbox, and CartoDB (4). The Leaflet libraries have been integrated fluidly into a R widget package form. We will learn the basics about this widget by creating some simple demonstrations.

8.3.1 Leaflet Demonstration 1 - A Simple Map

We can develop a simple user interface for our `ShinyApp` integration. Include a title panel, and create a sidebar layout. For this demonstration, we will simply have two numeric input controls that allow the user to enter the latitude and longitude. Our `ShinyApp` server will automatically render the new position onto the Leaflet map.

```
        library(shiny)
        library(leaflet)

        ### USER INTERFACE ###

        ui <- fluidPage(
                # Title of the ShinyApp
                titlePanel("Leaflet Demonstration 1"),
                # A layout with a sidebar
                sidebarLayout(
                        # Content of the sidebar panel
                        sidebarPanel(
                                # Two input controls
                                numericInput("lat", "Latitude:", min=-90,
                                    max=90, value=44.366002, step=0.001),
                                numericInput("lon", "Longitude:", min=-180,
                                    max=180, value=-68.196409, step=0.001)
                        ),
                        # Content of the main panel
                        mainPanel(
                                # The leaflet widget UI
                                leafletOutput("map")
                        )
                )
        )
```

For the server-side logic, we simply render the leaflet using a reactive system so that every time the latitude or longitude is updated, the map renders itself again. We generate a `leaflet()` object and assign it certain attributes from functions that are provided by the `leaflet` package. The `addTiles()` command ensures that the map will use the default map tiles that are provided in the package. The `addMarkers()` command adds a marker on the map that points to the location that has been chosen. This command also supports multiple markers in one command.

```
        ### SERVER LOGIC ###

```

```
        server <- function(input, output) {
                # Set the output map using a rendering function
                output$map <- renderLeaflet({
                        # The leaflet object
                        leaflet() %>%
                                addTiles() %>%
                                addMarkers(lng=input$lon, lat=input$lat,
                                    popup="The Location")
                })
        }
```

Finally, we run the server using the `shinyApp()` command. We pass both the user interface and the server logic as arguments. The server will run on a local address, such as `http://127.0.0.1:7591`.

```
        shinyApp(ui=ui, server=server)
```

Running the server produces an interactive map with a marker at the location that has been specified with the latitude and longitude in the sidebar. By changing the values, we can change the location reactively. Furthermore, we can zoom in and out of the map using the scroll function or the buttons provided on the map. Finally, we can also drag around the map by clicking inside the map control.

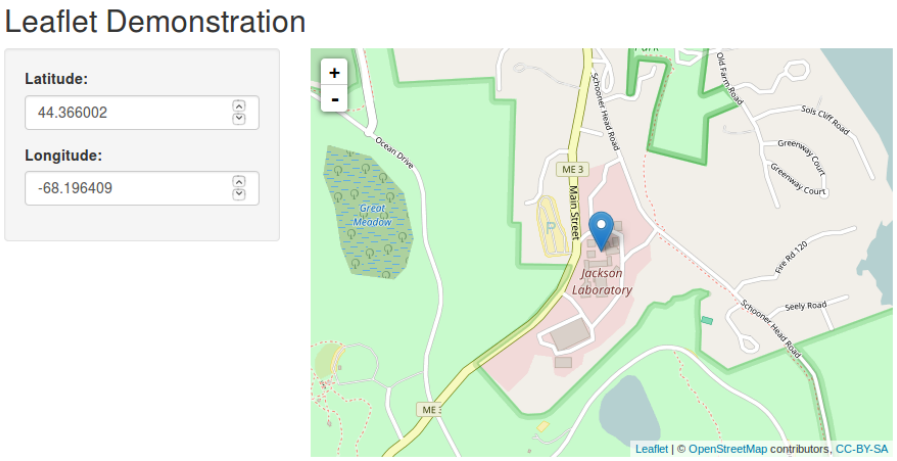


Figure 8.1: The Result of our First Demonstration

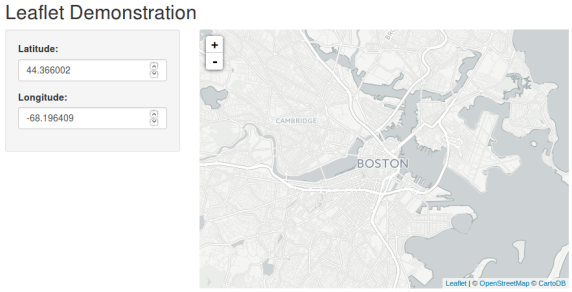
8.3.2 Leaflet Demonstration 2 - Map Tiles

In this section, we will explore the different views and tiles that are provided by third-party developers. We will only be changing the part of the code that renders the `leaflet()` in the server logic. The `setView()` command allows us to choose the location of the initial view and the level of zoom.

```
        # Stamen.Toner
        output$map <- renderLeaflet({
                leaflet() %>%
                        addProviderTiles("Stamen.Toner") %>%
                        setView(lng=-71.0589, lat=42.3601, zoom=12)
        })
        # CartoDB.Positron
        output$map <- renderLeaflet({
                leaflet() %>%
                        addProviderTiles("CartoDB.Positron") %>%
                        setView(lng=-71.0589, lat=42.3601, zoom=12)
        })
```

Figure 8.2: The Toner Tiles

Figure 8.3: The Positron Tiles



Web Map Service Tiles

We can use interesting commands to overlay the graphics from other web map services onto our tiles. For example, we will create a map that shows the intensity of precipitation by using the `addWMSTiles()` command.

```
        output$map <- renderLeaflet({
                leaflet() %>%
                        addTiles() %>%
                        setView(-93.65, 42.0285, zoom=4) %>%
                        addWMSTiles(
```

```
                                "http://mesonet.agron.iastate.edu/cgi-bin/
                                    wms/nexrad/n0r.cgi",
                                layers = "nexrad-n0r-900913",
                                options = WMSTileOptions(format = "image/png
                                    ", transparent = TRUE),
                                attribution = "Weather data Copyright 2012
                                    IEM Nexrad"
                            )
        })
```

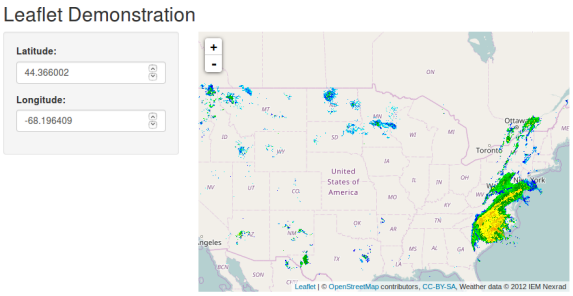


Figure 8.4: The Weather Data for Precipitation

8.3.3 Leaflet Student Demonstration - Markers

One of the most important functions of this widget, and this chapter in general, is the ability to correctly visualize data. To accomplish this, we utilize markers, which are inconsequential to visualizing data on maps. We will use population data from some major cities in America and use different markers to visualize the data.
Student Prompt: Use the following code that provides population data to develop a map that accurately demonstrates the size of each city mentioned in the dataset.

```
                population <- read.csv(textConnection(
                "City,Population,Lat,Lon,
                        Nashville,678889,36.1627,-86.7816
                        Asheville,87236,35.5951,-82.5515
                        Portland,609456,45.5231,-122.6765
                        Denver,649495,39.7392,-104.9903
                        Kansas City,467007,39.0997,-94.5786
                        Seattle,652405,47.6062,-122.3321
                        New York City,8406000,40.7128,-74.0059"
                ), header=TRUE)
```

Solution

In general, after using the documentation found on the web, most students will create a solution as follows:

```
        server <- function(input, output) {

```

```
                output$map <- renderLeaflet({
                        leaflet(data=population) %>%
                                addTiles() %>%
                                addMarkers(~Lon, ~Lat, popup=~paste(sep="<br
                                    />", paste(sep="", "<b>", as.character(
                                    City), "</b>"), as.character(Population)
                                    ))
                })
        }
```

The code will generate the following maps. Unfortunately, this type of visualization, while informative about the location of every city, does not help visualize any part of the population in every location. We will instead try to create a visual representation of the population in every location by generating circles with an area proportional to the population of the city.

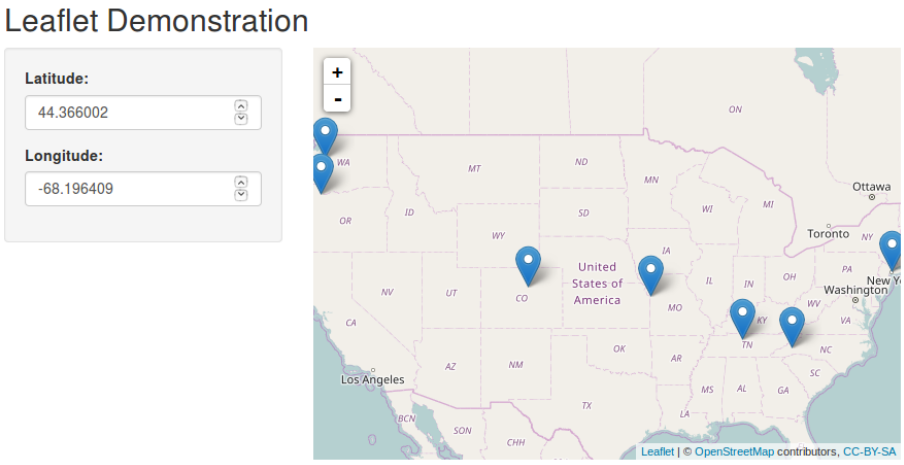


Figure 8.5: The Markers Placed at Every Location

The `addCircleMarkers()` allows us to add circles that have custom properties depending on the data point. We will use the `radius` attribute to make the circle larger or smaller depending on the population of the city.

```
        output$map <- renderLeaflet({
                leaflet(data=population) %>%
                        addTiles() %>%
                        addCircleMarkers(~Lon, ~Lat, popup=~paste(sep="<br/>
                            ", paste(sep="", "<b>", as.character(City), "</b
                            >"), as.character(Population)), radius=~sqrt(
                            Population/1000))
        })
```

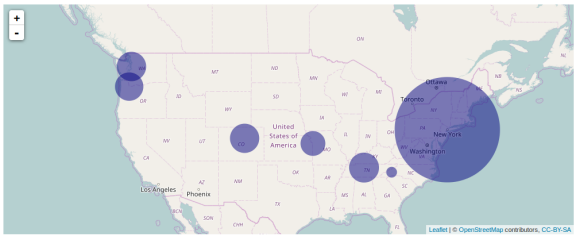
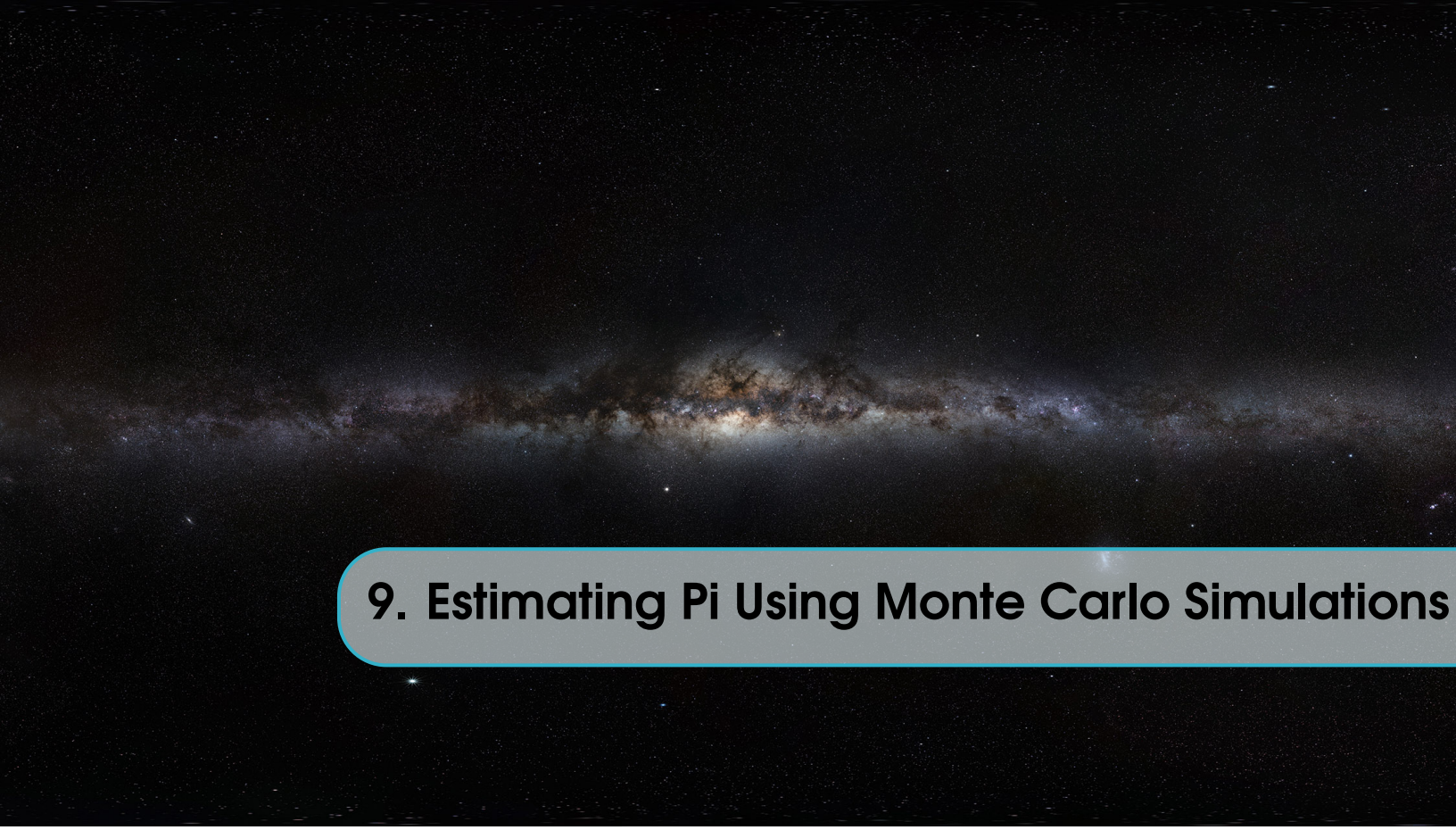


Figure 8.6: The Markers Placed at Every Location



9. Estimating Pi Using Monte Carlo Simulations

9.1 Introductory Reading

π (pi) denotes the ratio of the circumference of a circle to its diameter and is of great importance in mathematics. The first to attempt a theoretical calculation of its value was Archimedes (12), who was able to prove the following inequality:

$$\frac{223}{71} < \pi < \frac{22}{7}$$

In recent years, supercomputers have been able to calculate pi to 13.3 trillion digits (15). In this lesson, we investigate a computational method that can be used roughly determine the value of π: the Monte Carlo Method (1).
The Monte Carlo method is based on geometric probability. A circle inscribed in a square of side length 2 will have an area of π. Thus, the probability that any random point in the square is also inside the circle will be $\frac{\pi}{4}$. We can approximate π by asking the computer to pick a large number of random points inside the square, and seeing how many are also inside the circle.
We will use the Shiny package to display a plot with the circle inscribed in a square. We will also output the numerical approximation of π.

9.2 Objectives of the Case Study

- Students will use the Shiny package to create a Monte Carlo simulation in R.
- Students will understand the mathematical basis behind the estimation of π program.

- Students will use their Monte Carlo program to approximate the value of π.

9.3 Building the Model

The Monte Carlo simulation will be built using the Shiny package in R. This package is used to interactively display outputs such as plots and histograms. Today, we will be using it to display a plot of the circle inscribed in the square, as well as outputting the value of the π approximation.
To begin the application, open a new R script file in RStudio, clear the environment, set a working directory, and import the libraries necessary for this project. To clear the environment and set a directory use the following functions. Make sure you set a directory on your computer that the dataset is located in.

```
rm(list=ls())
setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")
```

The libraries are as follows:

```
library(shiny)
library(plotrix)
```

If you don't have one of these libraries installed on your computer, you can simply type `install.packages("insertlibraryname")` in the console to download them.
To begin, we will initialize the basic Shiny outline with the following code. Shiny relies on developing outputs based on any user input.

```
    server <- function(input, output) {

    }

    ui <- shinyUI(fluidPage(
        headerPanel("Estimating Pi"),
        sidebarPanel(),
        mainPanel()
    ))

    shinyApp(ui = ui, server = server)
```

This is the skeletal structure of the simplest Shiny application possible. Run the Shiny application by selecting all of your code and clicking the run button in the top right corner of RStudio. The blank Shiny application should open in a new window and look like figure 9.1.
To begin the model, we need to display the inputs of our program in the sidebar panel of the application. Our program is only taking in one input: number of simulated points. This input will define how many points the Monte Carlo method will simulate on the plot. We can do this like so:

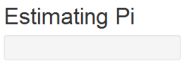


Figure 9.1: Blank Shiny App

```
    sidebarPanel(
        numericInput(inputId = "simulate", "Number of Simulated Points:",
        100, min = 100, max = 10000)
    )
```

This block of code will take a numerical value between 100 and 10000 as the number of simulated points. We want to set a maximum of 10000, because if we run more than 10000 simulations, the code because very computationally intensive and could result in RStudio crashing or your computer freezing. The `inputId` is simply a name we give the value of the input for future reference.
We also need to tell the application that our plot will be displayed in the main panel of the user interface. We also want the value of π that we have estimated to appear on the page. This involves creating a `plotOutput` and a `textOutput`, like so:

```
 mainPanel(
    textOutput(outputId = "estimate"),
    plotOutput(outputId = "monte", width = "400px", height = "400px")
 )
```

Just like in `selectInput`, we need to give the outputs an ID, so we can use that name to access it in the server function. Once the model is run, the application should look like figure 9.2.
Next we need to define the server-side of the application which will accept inputs and compute outputs. Our server function will contain the code necessary to do this. We first tell the function that we are creating a plot as an output, and we can create these outputs using the `renderPlot` and `renderText` command.

```
    output\$monte <- renderPlot({

    })

    output\$estimate <- renderText({

    })
```

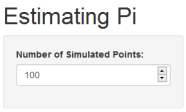


Figure 9.2: Shiny App with Inputs

The goal of this program is to simulate hundreds points on a plot of a unit circle circumscribed in a square. We need to generate x and y coordinates for these points, and then check if they fall within the circle. To do this we will use the *runif* command. This command generates random deviates within a range of values for a certain number of observations.
We will set two variables, `x` and `y`, to store the `x` and `y` coordinates of the generated points. Remember we want these values to lie between -1 and 1 on the coordinate grid. We want to run this simulation within the `renderPlot` and `renderText` functions so both outputs can access the result. This can be done like so:

```
    output\$monte <- renderPlot({
        x <- runif(input\$simulate, min = -1, max = 1)
        y <- runif(input\$simulate, min = -1, max = 1
    })

    output\$estimate <- renderText({
        x <- runif(input\$simulate, min = -1, max = 1)
        y <- runif(input\$simulate, min = -1, max = 1
    })
```

The `input$simulate` term is the number of simulations that the user inputs into the model. This function will generate values between -1 and 1 and store them in the `x` variable. Now we have to test if these points lie within the circle. The equation of a circle is $x^2 + y^2 = r^2$. This can be done using the `s.inside` command, which returns a 1 or 0 depending on if the point is inside the given boundary. Then we want to find the ratio of the number of points within the circle to the number of total points and multiply this by 4 to get an estimation of π.

```
     output\$monte <- renderPlot({
        x <- runif(input\$simulate, min = -1, max = 1)
        y <- runif(input\$simulate, min = -1, max = 1
        is.inside <- (x^2 + y^2) <= R^2
```

```
        estimation <- 4*sum(is.inside) / N
    })

    output\$estimate <- renderText({
        x <- runif(input\$simulate, min = -1, max = 1)
        y <- runif(input\$simulate, min = -1, max = 1
        is.inside <- (x^2 + y^2) <= R^2
        estimation <- 4*sum(is.inside) / N
    })
```

All we need to do now is to output the value of our estimation as well as the plot of the circle and the square. To output the text, we can simply use a return function within the renderText environment under the rest of our code like so:

```
    return(c("Estimate of Pi:", estimation))
```

To plot the circle and the points, we use the `plot` command to place the randomly generated points on a grid, and use the `draw.circle` command to draw a circle. To highlight which points are inside the circle we can color them blue, and the ones not inside will be colored red.

```
    plot(x, y, asp=1, xlim = c(-1,1), ylim = c(-1,1))
    draw.circle(0,0,1,nv=1000,lty=1,lwd=1)
    points(x[ is.inside], y[ is.inside], pch = 19, col = "blue")
    points(x[!is.inside], y[!is.inside], pch = 19, col = "red")
```

The above code should go in the `renderPlot` command under the previous code. Now when the Shiny app is run your final project should like figure 9.3.

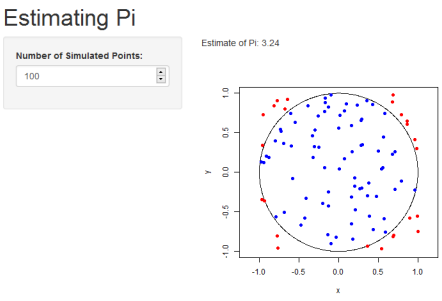


Figure 9.3: Complete Estimating Pi Application

Your application is now complete! Feel free to explore the approximations of π as you increase or decrease the number of simulated points.

9.4 Deliverable

For the final deliverable, the student should submit their code, along with a screenshot of the Shiny application using an instructor-defined number of simulated points.

9.5 Teaching Code

Students may use the framework below as a guide:

```
#Student name
#Date
#EstimatingPi.R

#clean up and set the directory


#load libraries


#setting the server, code instructing R what to do with inputs


    #initialize objects


    #create plot


  #set up estimate routine


#create graphics


#call shiny app
```

9.6 Example Student Code

The code that students submit should be similar to the example given below:

```
#Ishaan Rao
#September 1, 2016
#EstimatingPi.R

#clean up and set the directory
rm(list=ls())
setwd("C:/Users/Ishaan/Documents/NCSSM 2015-2016")

#load libraries
library(shiny)
library(plotrix)

```

```
#setting the server, code instructing R what to do with inputs
server <- function(input, output) {
  output\$monte <- renderPlot({

    N <- input\$simulate
    R <- 1
    x <- runif(N, min = -R, max = R)
    y <- runif(N, min = -R, max = R)
    is.inside <- (x^2 + y^2) <= R^2
    estimation <- 4*sum(is.inside) / N

    #plot.new()
    #plot.window(xlim = 1.1 * R * c(-1, 1), ylim = 1.1 * R * c(-1, 1))
    plot(x, y, asp=1, xlim = c(-1,1), ylim = c(-1,1))
    draw.circle(0,0,1,nv=1000,lty=1,lwd=1)
    points(x[ is.inside], y[ is.inside], pch = 19, col = "blue")
    points(x[!is.inside], y[!is.inside], pch = 19, col = "red")
  })

  output\$estimate <- renderText({
    N <- input\$simulate
    R <- 1
    x <- runif(N, min = -R, max = R)
    y <- runif(N, min = -R, max = R)
    is.inside <- (x^2 + y^2) <= R^2
    estimation <- 4*sum(is.inside) / N

    return(c("Estimate of Pi:", estimation))

  })
}

ui <- shinyUI(fluidPage(
  headerPanel("Estimating Pi"),
  sidebarPanel(
    numericInput("simulate", "Number of Simulated Points:", 100, min = 100,
        max = 10000)
  ),
  mainPanel(
    textOutput("estimate"),
    plotOutput("monte", width = "400px", height = "400px")
  )
))

shinyApp(ui = ui, server = server)
```

9.7 Further Readings

To learn more about what Monte Carlo simulations are and their applications to a wide variety of problems visit `https://en.wikipedia.org/wiki/Monte_Carlo_method` or `http://www.palisade.com/risk/monte_carlo_simulation.asp`.
To learn more about the mathematical theory behind estimating π visit `https://`

en.wikipedia.org/wiki/Approximations_of_%CF%80 or http://polymer.bu.edu/java/java/montepi/MontePi.html

10. Plotluck

10.1 Introduction

Coined as the "ggplot2 version of 'I'm feeling lucky!"', `plotluck` is designed to consider the characteristics of a data frame to decide, through a heuristic algorithm, the most appropriate plot to represent the data. The package has scatter, box, bar, violin, density, heat map, hexagon bin, and spine plots available. The ultimate goal of the package is to allow users to explore which type of plot would best fit their data, instead of becoming burdened with the creating code in R.

10.2 Objectives

In this assignment, we will use `plotluck` to explore rapid visualization of several data sets. The code created is a good place to start, but we encourage you to continue exploring the package. The website Gapminder has an extreme amount of data sets available for public use, and R has around 30 data sets built in.

10.3 Deliverable

As plot luck is created for finding and forming appropriate graphics for respective data sets, the desired deliverable will be dependent on the data set used.

For example, plots using ggplot2's `mpg` data set are shown in figures 10.1, 10.2, and 10.3.

More examples include figures 10.4 and 10.5.

10.4 Teaching Code

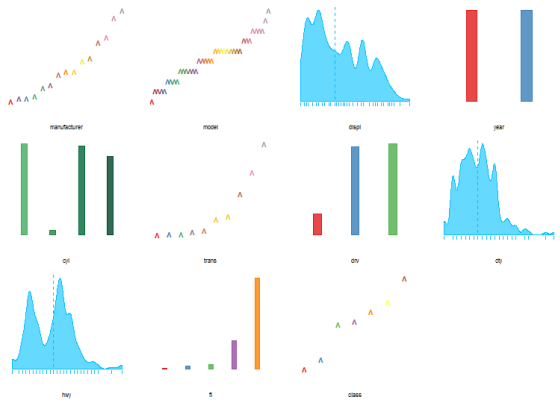


Figure 10.1: Plotting every variable in `mpg`

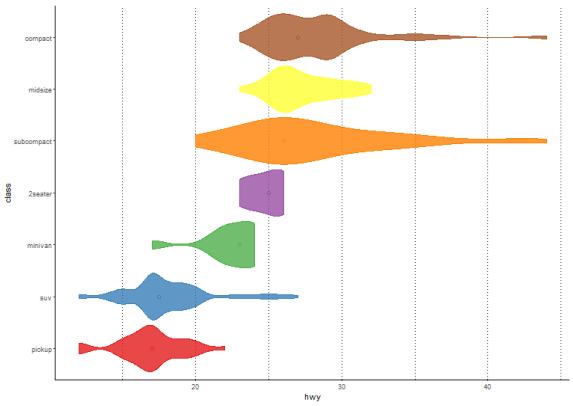


Figure 10.2: Highway mileage to class

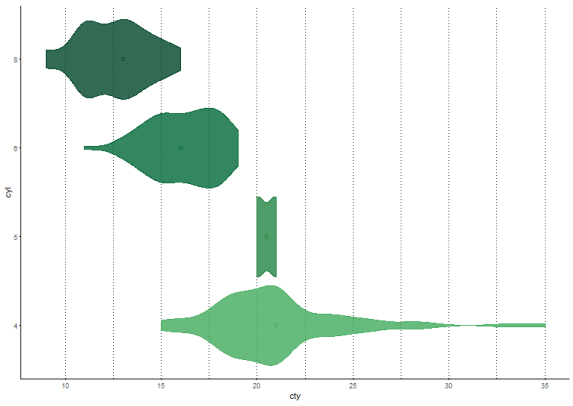


Figure 10.3: Cylinder count vs city mileage

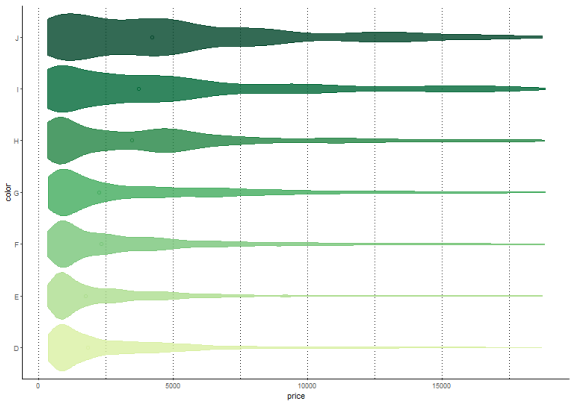


Figure 10.4: Price to color in the `diamonds` dataset

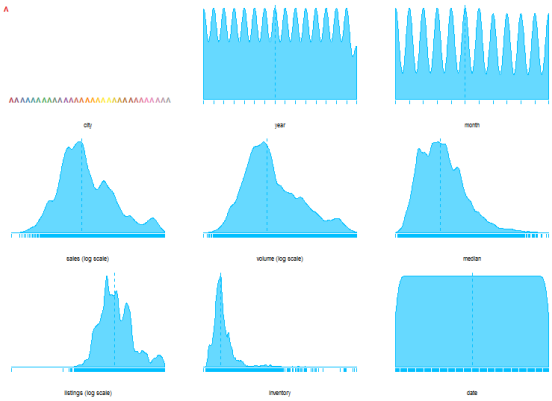


Figure 10.5: Plotting every variable in `txhousing`

```
# Example for Plotluck

# Clean up and import plotluck
rm(list=ls())
library(plotluck)
setwd("C:/Your/R/Directory")

# Let's work with the faithful data
# What's in this data set?
plotluck(.~1, faithful)

# Look at everything vs everything else:
plotluck(.~., faithful)

# Eruptions look interesting, what can that tell us?
plotluck(eruptions~., faithful)
plotluck(eruptions~waiting, faithful)

# Let's look at airquality
plotluck(.~1, data=airquality)

# We can generate some interesting plots because Month is discrete
plotluck(Temp~Month, airquality)
plotluck(Wind~Month, airquality)
plotluck(Ozone~Month, airquality)

# And some more interesting ones
plotluck(Temp~Solar.R+Ozone, airquality)
plotluck(Temp~Wind+Ozone, airquality)
plotluck(Temp~Solar.R|Month, airquality)

# Let's look at mtcars
# This tells us more about the variables
plotluck(.~1, mtcars)
# Here are some interesting examples
plotluck(vs~gear, mtcars)
plotluck(mpg~wt, mtcars)

# Let's explore random plotting!
data(diamonds, package='ggplot2')
set.seed(100)
# Run this several times to get random plots
sample.plotluck(diamonds)
# Note the following hex plots:
plotluck(depth~x, diamonds)
plotluck(price~carat, diamonds)
```

10.5 Example Student Code

```
# EXAMPLE KEY
# Date
# Plotluck Exercises - KEY
```

```
4
5 # Clean up and import plotluck
6 rm(list=ls())
7 library(plotluck)
8 setwd("D:/School/RCompSci/book/plotluck")
9
10 # Using the following command:
11 # data(dataset_name, package='ggplot2')
12 # Look at three of the following datasets using plotluck:
13 # mpg, msleep, diamonds, presidential, txhousing, seals
14
15 # Code goes here
```

10.6 Further Readings

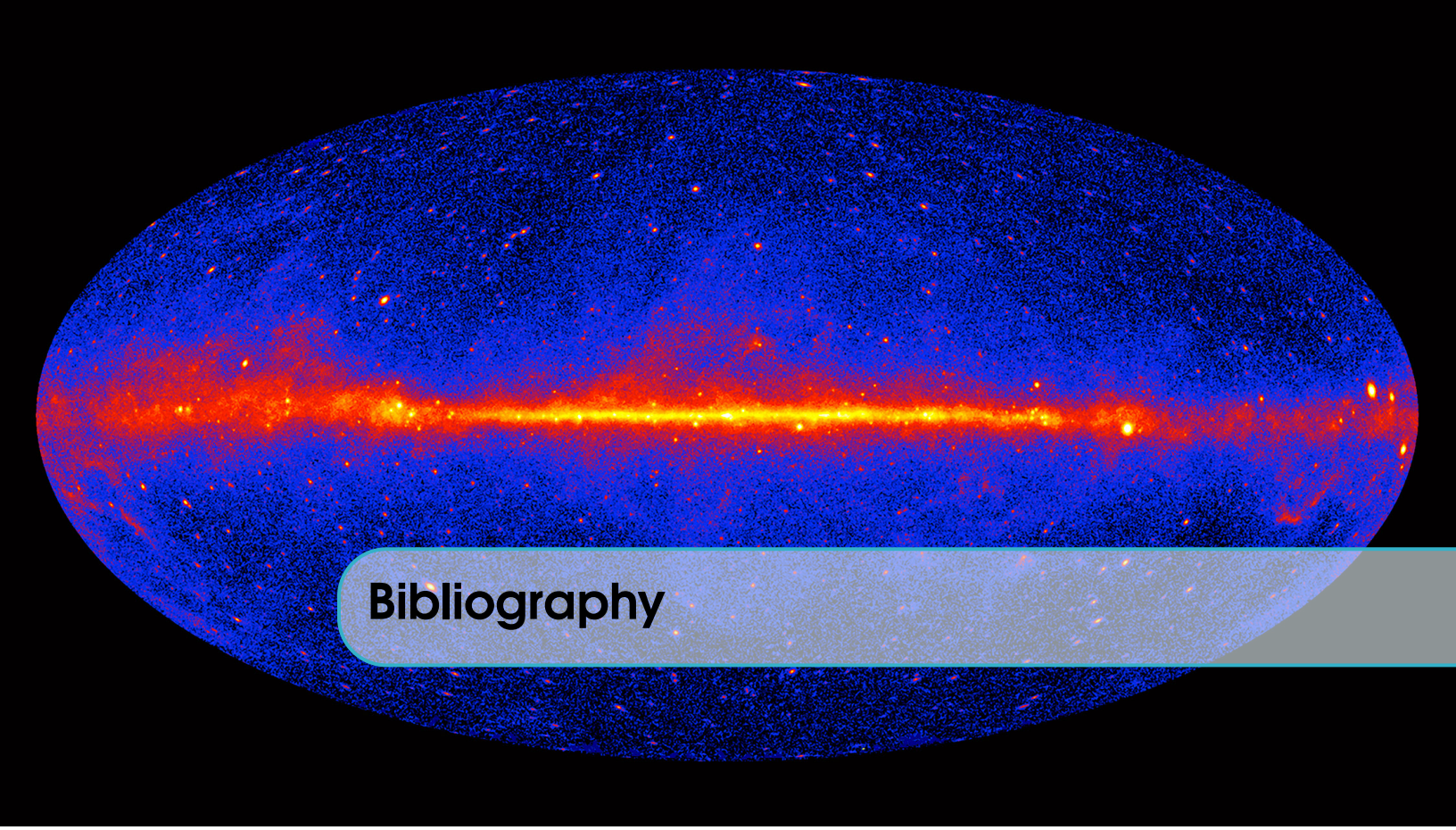
1. https://cran.r-project.org/web/packages/plotluck/index.html
2. https://cran.r-project.org/web/packages/plotluck/vignettes/plotluck.html

Bibliography

(1) https://cran.r-project.org/web/packages/plotluck/index.html

(2) https://cran.r-project.org/web/packages/plotluck/vignettes/plotluck.html

(3) https://www.gapminder.org/data/



Bibliography

(1) Eve Andersson. *Calculation of Pi Using the Monte Carlo Method* (cited on page 83).

(2) V. Barve. *Discovering and developing primary biodiversity data from social networking sites: A novel approach*. 2014. URL: http://doi.org/10.1016/j.ecoinf.2014.08.008 (cited on pages 67, 68).

(3) Jeffrey R. Chasnov. *Introduction to Differential Equations*. 2016. URL: https://www.math.ust.hk/~machas/differential-equations.pdf (cited on page 39).

(4) *Leaflet for R*. 2015. URL: http://rstudio.github.io/leaflet/ (cited on page 76).

(5) A. Mathelier, X. Zhao, and A. W. Zhang. *JASPAR 2014: an extensively expanded and updated open-access database of transcription factor binding profiles*. Volume 42. Jan. 2014 (cited on page 63).

(6) Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cited on page 17).

(7) Frederick Novomestky. *Cluster Analysis Data Sets*. 2013. URL: https://cran.r-project.org/web/packages/cluster.datasets/cluster.datasets.pdf (cited on page 57).

(8) *SAFRING: Historical Bird Ringing Records*. Oct. 9, 2016. URL: http://www.gbif.org/dataset/b4ae1720-1431-49ee-bfeb-8146fc42b1a3 (cited on page 68).

(9) Kim Seefield and Ernst Linder. *Statistics Using R with Biological Examples*. 2007 (cited on page 9).

(10) P. Shanmughavel. *An overview on biodiversity information in databases*. 2007. URL: http://www.ncbi.nlm.nih.gov/pubmed/17597923 (cited on page 68).

(11) *Shiny - Movie Explorer*. Sept. 8, 2016. URL: http://shiny.rstudio.com/gallery/movie-explorer.html (cited on page 29).

(12) Laura Smoller. *The Amazing History of Pi*. Feb. 6, 2001 (cited on page 83).

(13) Guy-Bart Stan. *Modelling in Biology*. 2010. URL: http://www.bg.ic.ac.uk/research/g.stan/2010_Course_MiB_article.pdf (cited on page 47).

(14) *Teach Yourself Shiny*. Sept. 8, 2016. URL: http://shiny.rstudio.com/tutorial/ (cited on page 31).

(15) Alexander J. Yee. *Y-cruncher - A Multi-Threaded Pi Program*. Sept. 19, 2016 (cited on page 83).