2. Learning R

Why R?

R is simple enough for programming novices to learn and a breeze for computer scientists. Everything in R is built in. You can switch seamlessly from data cleaning to data analysis to creating plots to running machine learning algorithms to performing statistical analysis on your results. R is open source and supported by a core group of contributors and there is lots of online help at sites such as `https://stackoverflow.com`. R gives us the ability to get work done quickly with minimal learning curves. R is the *lingua franca* in academia among data scientists, statisticians, and machine learning specialists. R is also used widely in industry by companies with a lot of data like Google, Microsoft, Amazon, as well as traditional industries. R also gives you great tools for easily sharing and communicating your results with html or pdf reports as well as interactive web sites.

In the next few sections you will learn:

- Where to find R and RStudio installation links and instructions
- R data structures and how to manipulate them in R syntax
- R data exploration functions and techniques
- R data visualization functions and techniques
- How to install and use R packages
- How to use R notebooks which combine text and code
- R control structures for more complex code
- Recommendations for R style

2.1 Installing R, RStudio

You will need to first install R and then install RStudio, a beautiful and powerful IDE for R. Both are available for Windows, Mac and linux.

Link to download and install R: `https://www.r-project.org/`

Link to download and install RStudio: `https://www.rstudio.com/`

Choose RStudio's free community edition. While you are on the site take a look around at some demos and tutorials.

2.1.1 Getting Started in R

If you open up RStudio you will see a couple of panes on the right, and one large pane on the left, similar to Figure 2.1. This pane on the left is the console. At the top of the console screen you will see some information about your version of R, then the interactive prompt > waiting for you to do something. In the figure, we've typed in a simple expression: 3+4.

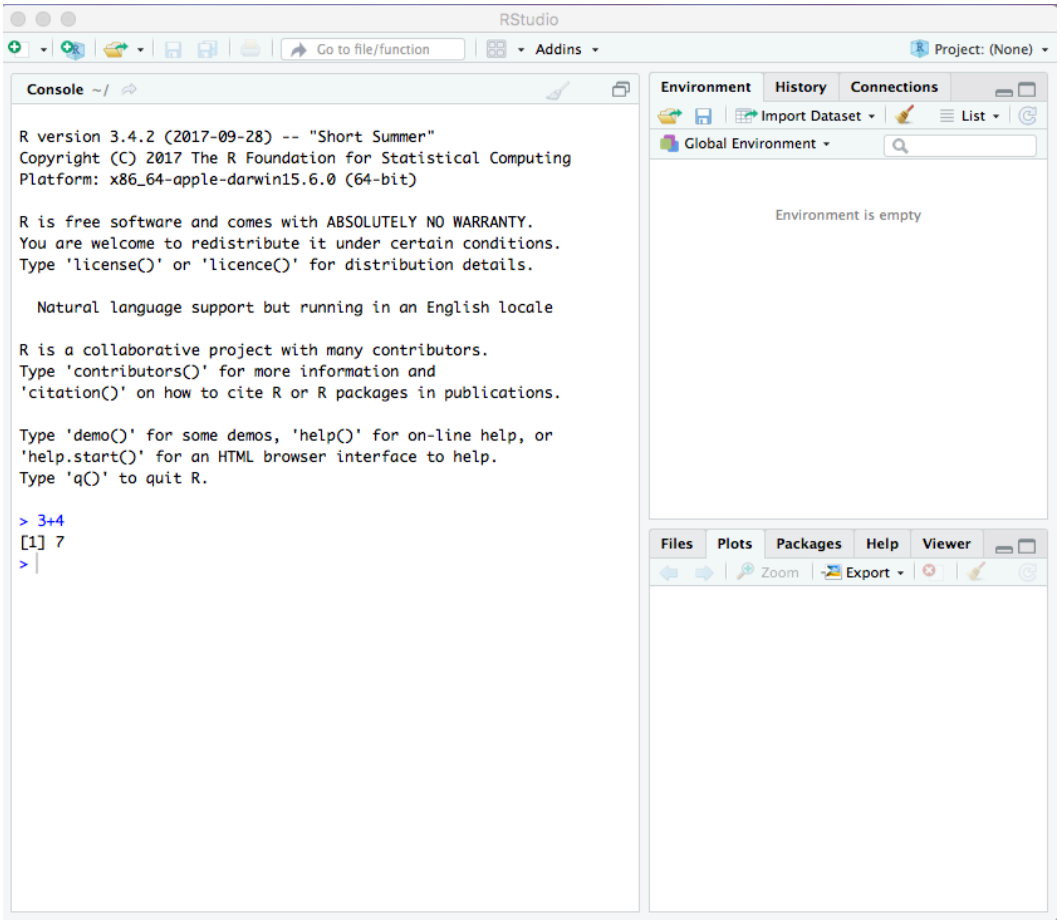

Figure 2.1: RStudio IDE

R is an interpreted language which makes it easy to experiment with ideas at the console. You can also save your code in notebooks or scripts, to run again and again. Although R is an interpreted language, most of the code is in C/C++ so it is really fast.

Type along with these examples in the console window as we go. Below, we typed an arithmetic expression at the prompt, hit enter and we see the results on the next line. There should be no surprises here, it's the usual operators with the usual order of operations.

```
> 3 + 4 - 1 * 8 / 2
[1] 3
```

But what is that [1] doing there in front of the output 3? That's just telling your that your result is a one-dimensional object. It's actually a vector of length 1. In R, pretty much everything is an object. Let's type in the same expression as above but this time save it to variable x. By the way, the up arrow displays the previous command, just like in a unix console. The less-than-dash, < –,

is the assignment operator in R. We did not have to declare variable x. R figured out what type it is by what was stored in it. R has dynamic typing; if we change what type of object is in x later, R will not complain.

```
> x <- 3 + 4 - 1 * 8 / 2
> x
[1] 3
```

Now when we type in x at the console, R echoes back its contents. What is x? Let's ask R about it. There are many functions we can use to inquire about an object:

```
> is.numeric(x)
[1] TRUE
> class(x)
[1] "numeric"
> typeof(x)
[1] "double"
```

Note that numbers are stored as doubles by default. If we specifically want to store an integer we use the L indicator:

```
> x <- c(1L, 2L, 3L)
> typeof(x)
[1] "integer"
```

The c() function combines elements into a vector. It will coerce them to a common type, the most inclusive type in the vector:

```
> x <- c(1L, 1.2, "hi")
> typeof(x)
[1] "character"
```

2.2 R Data

We have seen in the examples above that data in R can be logical, numeric (integer or floating point), or character. Data is organized into objects of varying types. In this section we learn more about R objects that hold and organize data. The following table organizes the R data structures by their dimensions and whether or not all elements have to be of the same type.

Dimension	Homogeneous	Heterogeneous
1d	atomic vector	list
2d	matrix	data frame
nd	array	

Table 2.1: R Data Structures

2.2.1 Vector

A vector is a sequential structure with one or more elements of the same type. There isn't really a scalar in R, it's just a vector of length 1. Follow along with this code:

```
> x <- 1:10
> x*5
 [1]  5 10 15 20 25 30 35 40 45 50
>length(x)
[1] 10
>sum(x)  # try mean(), median(), range(), max(), min()
[1] 55
```

In the first line, x is assigned to be a vector from 1 to 10 with the sequence operator ":". In the second line we multiply each element by 5. In most programming languages you would need a loop to do this but note the simple syntax that allows us to do this in one line. There are many built-in functions in R. The next line shows length(). The last line returns the sum of each element of x. You may be wondering why it is 55 since we multiplied each element by 5 above. The reason is that we did not assign it back to x as in `x <- x*5`.

The vector x is numeric but we can have other types of vectors such as character or logical. When we type "x > 5" at the console, a logical vector returns with TRUE or FALSE for each element. If we `sum(x>5)`, the TRUE values count as 1 and the FALSE values as 0 so we get a count of how many elements are greater than 5. Notice in the last line below that we can select those instances greater than 5 and replace only those with 0. Such power with such simple syntax!

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> sum(x>5)
[1] 5
> x[x>5] <- 0
> x
 [1] 1 2 3 4 5 0 0 0 0 0
```

Subsetting vectors requires first learning one strange thing about R. It starts indexing at 1, not 0 like other programming languages. If you type `x[0]` at the console you won't get an error but some information about the vector, so it looks like that space is put to good use. Here are some indexing examples:

```
> x <- 1:10
> x[0]
integer(0)
> x[2:4]      # use : for a range of elements
[1] 2 3 4
> x[c(2:4,8)] # use c() for noncontiguous elements
[1] 2 3 4 8
> x[11]
[1] NA
```

Notice above that we did not get an error when we had an index out of bounds, it just gave us an NA, for "not available".

To check the type of a vector you can use:

- typeof(x) # returns type
- is.integer(x) # returns boolean
- is.double(x) # returns boolean
- is.numeric(x) # returns TRUE for integer or double
- is.character(x) # returns boolean
- is.logical(x) # returns boolean

2.2.2 Lists

A list is an ordered collection of objects not neccesarily of the same type. Lists can contain other lists as elements. Lists are often used as holders for things returned from functions. List elements are selected with double square brackets:

```
z <- list(1,2,3)
z[[1]]
[1] 1
y <- list('a', TRUE, z)
> typeof(y)
[1] "list"
> length(y)       # y is a list of length 3
[1] 3
> length(y[[3]])  # the 3rd element is also a list of length 3
[1] 3
```

Caution 2.1 — Lists and Vectors. A list is a generic vector where elements do not have to be of the same type. For this reason, to check if an object is a vector, don't use `is.vector()` but the following:

```
x <- 1:5
is.atomic(x)  # check if x is a vector
[1] TRUE
```

■

You can check if a list has lists as elements with the `is.recursive()` function. The `unlist()` function converts a list into a 1-dimensional atomic vector, coercing all elements into the most general type.

```
> w <- unlist(y)
> w
[1] "a"    "TRUE" "1"    "2"    "3"
> typeof(w)
[1] "character"
```

2.2.3 Matrix

A matrix is a 2-dimensional object with elements of the same type. The code below creates a sequence from 1:10 and stores it in a matrix of 2 rows.

```
> m <- matrix(1:10, nrow=2)
> m
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Notice the handy reminders showing how to index at the top of the columns and the left of the rows. Practice indexing matrix m. Indices are `[row, col]`.

We will mainly use vectors and data frames (discussed below) in machine learning but we will use matrices from time to time.

2.2.4 Arrays

Arrays are similar to matrices but can have more than 2 dimensions. Like matrices, all elements must be of the same type.

2.2.5 Data Frames

A data frame is a 2-dimensional structure where each column can be of a different type. Most of the time we will load a data frame from disk but we can create one manually by creating 3 vectors that we combine with cbind (column bind):

```
> x <- c(1,2,3)
> y <- c(1.1, 2.2, 3.3)
> z <- c('a','b','c')
> df <- data.frame(cbind(x,y,z))
> df
  x   y z
1 1 1.1 a
2 2 2.2 b
3 3 3.3 c
```

In a data frame, each column is a vector of a specific type. All the columns are of the same length. Observe in the code below that we can access these column vectors with the dollar operator and that we can also change the column names.

```
> df$x
[1] 1 2 3
Levels: 1 2 3
> colnames(df) <- c('Ticket', 'Discount', "Section")
> df
  Ticket Discount Section
1      1      1.1       a
2      2      2.2       b
3      3      3.3       c
```

We can read in a text file or csv file from disk as shown below. If you want to know more about the options for read.csv use the command at the bottom of the following code example. The str() functions tells you about the structure of the data frame, its columns, and what type of data it contains, as well as how many observations and variables.

```
> df <- read.csv("titanic3.csv")
> str(df)
'data.frame': 1310 obs. of  14 variables:
```

```
 $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : Factor w/ 1308 levels "","Abbing, Mr. Anthony",...
 $ sex      : Factor w/ 3 levels "","female","male": 2 3 2 3 2 3 2 3 2 3 ...
 $ age      : num  29 0.917 2 30 25 ...
 $ sibsp    : int  0 1 1 1 1 0 1 0 2 0 ...
 $ parch    : int  0 2 2 2 2 0 0 0 0 0 ...
 $ ticket   : Factor w/ 930 levels "","110152","110413",...
 $ fare     : num  211 152 152 152 152 ...
 $ cabin    : Factor w/ 187 levels "","A10","A11", ...
 $ embarked : Factor w/ 4 levels "","C","Q","S": 4 4 4 4 4 4 4 4 4 2 ...
 $ boat     : Factor w/ 28 levels "","1","10","11", ...
 $ body     : int  NA NA NA 135 NA NA NA NA NA 22 ...
 $ home.dest: Factor w/ 370 levels "","?Havana, Cuba", ...
> ?read.csv  # ? is the help command
```

There are also many built-in data sets in R. You can find out about them by typing data() at the console.

2.3 Data Exploration

You can load a built-in data set with the data() functions and then look at a few rows with head() or tail(). Comments start with # so you don't need to type those in when you are working at the console.

```
> data(airquality)
> head(airquality, n=2) # see the first two rows
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
```

Here are several data explorations functions you should explore at the console, replacing "df" with the name of your data frame, such as "airquality":

- names(df) lists the columns
- dim(df) gives the row, col dimensions
- summary(df) gives summary statistics for each column
- str(df) gives row and column counts, information for each column
- head(df) and tail(df) give the first and last 6 rows by default

Missing items in a data frame are typically encoded as NA. This can cause some problems for built-in functions as shown below. Notice the sum() function would not work until we told it to ignore NAs.

```
> sum(is.na(airquality$Ozone))  # count the NAs
[1] 37
> mean(airquality$Ozone)
[1] NA
> mean(airquality$Ozone, na.rm=TRUE)
[1] 42.12931
```

A trickier problem is deciding what to do with NAs before we run data sets through machine learning algorithms. Some algorithms may not work and others may give less than optimal results

with NAs. One option is to delete rows that have NAs but this could be a problem if your data set is already small. Another option is to replace NAs with a mean or median value as shown next. First we copy the data set to another variable so we won't alter the original.

```
> df <- airquality[]
> df$Ozone[is.na(df$Ozone)] <- mean(df$Ozone, na.rm=TRUE)
> mean(df$Ozone)
[1] 42.12931
```

2.4 Visual Data Exploration

In addition to exploring data with the R commands described in the last section, and exploring the data with built-in statistical functions, we can also explore data with graphs. Type `demo(graphics)` at the console to see the variety of graphs you can create in R. Next we continue looking at airquality with R functions.

Type the following at the console to see the graphs:

```
hist(airquality$Temp)
plot(airquality$Temp)
plot(airquality$Temp, airquality$Ozone)
```

Here are some useful arguments for modifying graphs:

- pch - specifies the symbol to use for points; 1 is the default
- cex - specifies symbol size, 1 is the default, 1.5 is larger, 0.5 is smaller
- lty - specifies line type, default is 1
- lwd - specifies line width, default is 1
- col - color of graph element; colors can be specified by name, number, hex; col=2 and col="red" are the same
- xlab - label for x axis
- ylam - label for y axis
- main - main label

We will learn more about graphs in R as we go. The following link provides a good overview of R graph parameters: `https://www.statmethods.net/advgraphs/parameters.html` Below is an example of using some of these parameters in code. Figure 2.2 shows the resulting plot.

```
plot(airquality$Ozone, airquality$Temp, pch=16, col="blue", cex=1.5,
    main="Airquality", xlab="Ozone", ylab="Temperature")
```

Often we want to know if columns are correlated with other columns. Correlation means that if you plotted the vectors as lines they would be somewhat parallel. Vectors with positive correlation point in the same direction; those with negative correlation point in opposite directions. Correlation is quantified on a scale of -1 to $+1$, with values closer to the 1s indicating strong positive or negative correlation and values closer to 0 indicating weak or no correlation. The code below shows how to generate a correlation matrix or a graph. See if you can identify correlations in the graphs.

```
> cor(airquality[1:4], use="complete")
             Ozone    Solar.R       Wind       Temp
Ozone    1.0000000  0.3483417 -0.6124966  0.6985414
Solar.R  0.3483417  1.0000000 -0.1271835  0.2940876
Wind    -0.6124966 -0.1271835  1.0000000 -0.4971897
Temp     0.6985414  0.2940876 -0.4971897  1.0000000
pairs(airquality[1:4])
```

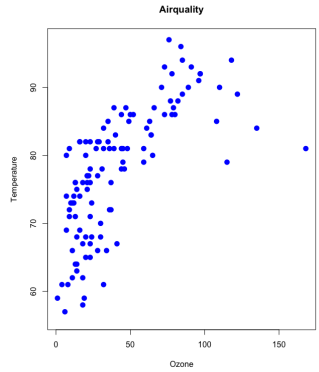

Figure 2.2: Airquality

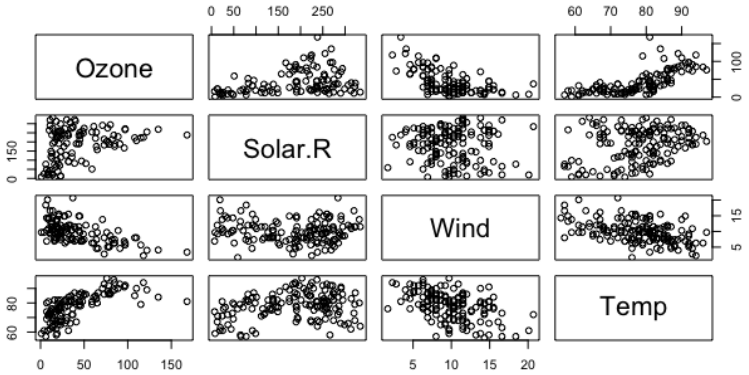

Figure 2.3: Correlation Pairs for Airquality

2.5 Factors

We have seen vectors of various types: integer, numeric, character, and logical. There is one more type of vector that will be important to us. Factors are used to encode qualitative data. A factor vector is a vector of integers with an associated vector of labels. Internally it is stored as integers but for our benefit they display as labels. Let's take another look at the Titanic data.

```
> df <- read.csv("titanic3.csv")
> str(df)
'data.frame': 1310 obs. of  14 variables:
 $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : Factor w/ 1308 levels "","Abbing, Mr. Anthony",...
 $ sex      : Factor w/ 3 levels "","female","male": 2 3 2 3 2 3 2 3 2 3 ...
 $ age      : num  29 0.917 2 30 25 ...
 $ sibsp    : int  0 1 1 1 1 0 1 0 2 0 ...
 $ parch    : int  0 2 2 2 2 0 0 0 0 0 ...
 $ ticket   : Factor w/ 930 levels "","110152","110413",..: 189 51 51 51 51 ...
 $ fare     : num  211 152 152 152 152 ...
. . .
```

The above code assumes that you have the csv file in the same directory you are working in. The file can be downloaded from the web. This data is a bit messy because some things that should be factors, like pclass or survived, are not, and some things are factors that should not, like fare, cabin, etc. We can convert to another data type as follows:

```
df$pclass <- as.factor(df$pclass)
df$ticket <- as.integer(df$ticket)
```

We can see the encoding with the contrasts() function.

```
contrasts(df$sex)
       female male
            0    0
female      1    0
male        0    1

> contrasts(df$pclass)
  2 3
1 0 0
2 1 0
3 0 1
```

The contrasts() for sex shows a problem, apparently we have some passengers for whom we don't have data on sex. The contrasts for pclass shows that we need 2 variables to encode 3 classes. The base case will be class 1. R will create 2 *dummy variables* for classes 2 and 3. We will see the importance of these when we get to machine learning.

2.5.1 Adding a Factor Column to a Data Frame

The following code adds a new column to our data frame for airquality. First it makes all items = FALSE, then makes those with a temperature over 80 to be TRUE. Finally we coerce it to be a factor. The first line has a comment which starts with a hash tag.

```
> df <- airquality[]  # copy the data set
> df$Hot <- FALSE
> df$Hot[df$Temp>89] <- TRUE
> df$Hot <- factor(df$Hot)
> df$Hot[40:46]
[1] TRUE  FALSE TRUE  TRUE  FALSE FALSE FALSE
Levels: FALSE TRUE
>plot(df$Hot)
```

Here are some plots of the Hot column, arranged in a 1x3 grid with the par() function. Plotting the column by itself just gives us a visual of the distribution of Hot and not Hot. The cdplot() gives a conditional density of Hot (light grey) and not (black) across the x axis which represents temperature. The third plot is a box plot in which the heavy middle line in the box denotes the median, the box itself indicates the IQR and the horizontal lines at either end of the dashed line indicate min and max, excluding suspected outliers which are dots beyond that line.

```
> par(mfrow=c(1,3))
> plot(df$Hot)
> cdplot(df$Temp, df$Hot)
> plot(df$Hot, df$Temp)
```

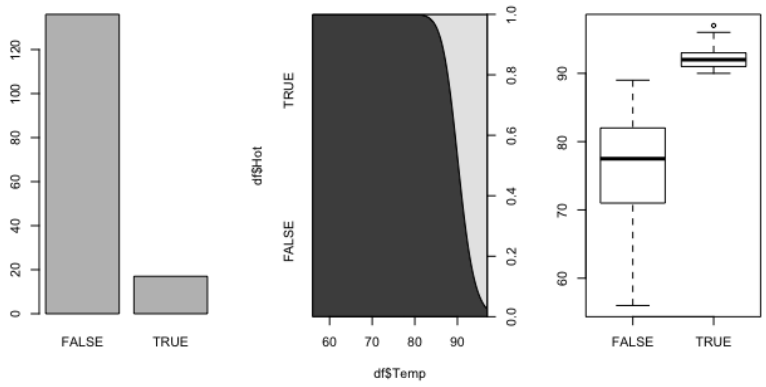

Figure 2.4: Airquality

The graphs shown in this chapter are rather plain. We can add headings, labels, color and more. Read more about graphical parameters here: `https://www.statmethods.net/advgraphs/parameters.html`

2.6 R Notebooks

Here are a few things to keep in mind about R:

- R is case sensitive
- The period has no special meaning unlike many other languages
- The dollar sign acts somewhat like the period in other languages
- The hashtag starts comments
- White space is ignored
- The R workspace stores all the objects you create in a session. You can save it, but for now you should always choose No when it asks you to save the workspace.

R is a powerful language in its own right. However it becomes truly awesome when we load packages built for specific purposes. These packages are available on CRAN, the Comprehensive R Archival Network. We install packages at the console with the install.packages() command. You only have to install once. We load packages into our working environment only once per script with either the library() or require() functions.

In RStudio, go to File, New File, then R Notebook. Save your file. I recommend creating a folder to keep all your notebooks in, then always loading from that folder by double-clicking on it.

Before we get started, we should note that you can also create a simple R script. These are text files that end in .R and can be run from the console or from within RStudio. We are going to focus on R notebooks which are similar to Python notebooks, interspersing text commentary and code.

Once you create the R notebook you will see some YAML code at the top. You can also add an author: line and you should change the title. YAML is a recursive acronym for YAML Ain't Markup Language. This YAML tells RStudio to what kind of output to create, in this case it will make an html notebook that you can upload to your website or github.

```
---
title: "R Notebook"
output: html_notebook
---
```

You'll notice white portions for text and grey portions for code. Read through the standard text which provides some useful tips on things like how to add new code chunks and get started with markdown text. You can type regular text in the white portions but markdown is worth learning because you will encounter it in many situations beyond R. There is a nice cheat sheet for markdown in RStudio's web site. The really nice thing about markdown is that you can format as you type without taking your hands off the keyboard.

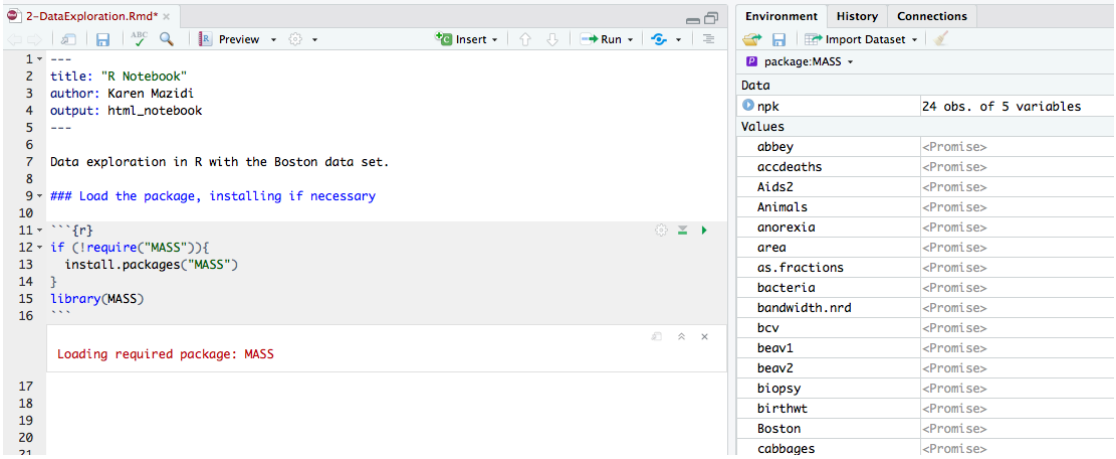

Figure 2.5: RStudio Environment

Figure 2.5 shows the RStudio environment with the white portions for text markdown and the grey portions for code. The full notebook is available on the github site for this book. You should practice making your own notebook similar to the sample. Let's point out a couple of things before we move on. First, once the MASS package was loaded, you can see it in the environment in the upper right. This environment is a great way to see your variables as you go. It is a great help in debugging. Second, notice the Preview button above the notebook. You can use this to "knit" or render your file into html or pdf or other formats. This option is also available under the File menu.

2.7 Control structures

We have seen earlier that a lot of things we might use a loop for in other languages can be done in simple R syntax. However there will be times when we want to code loops, conditionals, and functions. We describe those in this section, in an example getting you started using an R notebook.

This section uses more advanced R code so it is fine to skip this section and come back to it when you are more familiar with R. The notebook is in the github site so you can refer to it as you need it.

If you wish to go through this section now, open RStudio and go to File->New File->R Notebook. Get rid of the text in the white areas. Change the title at the top of the file. Now we are going to load package mlbench, which contains several benchmark ML data sets. In the grey code box, remove the code that is there and type "library(mlbench)". If you run this chun of code either with the green arrow on right of the box or just hitting ctrl-enter on the line, you will probaby get an error since you probably don't have this package installed. To install the package, type the following down at the console, not in your file:

```
install.packages("mlbench")
```

This should just take a minute. If R asks you for a mirror site in a pop-up, just pick one. Once

the package is installed, from then on you just load the package into memory with library() or require(). Now type the following at the console:

```
data(package='mlbench')
```

This will pop up a tab listing the data in the package. Once you have installed mlbench, add the following commands to the code chunk so that it looks like the following and rerun the code.

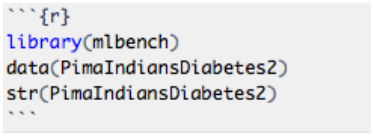

Figure 2.6: First Code Chunk

In the upper right pane of RStudio, click on Environment then Global Environment. You should see that the data has been loaded into memory. You should see that it has 768 observations and 9 variables. The str() command you ran above will output information about each variable. You can learn more about this data set by typing `?PimaIndiansDiabetes2` at the console.

From the str() function above, you can see that there are a lot of missing values denoted by NA. Create a new code chunk at the botton of your notebook by clicking on Insert->R at the top of the notebook window. Type in the following code and see the results. The sapply function applies a function to data. Here we have an anonymous function coded on the fly to sum NAs. The sapply() function will apply this to each column.

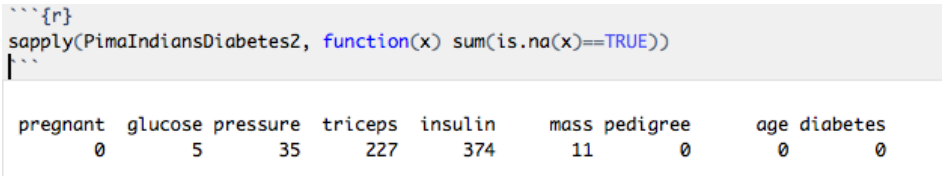

Figure 2.7: Second Code Chunk and Results

We have a few NAs for glucose, mass, and pressure, but a lot for tricepts and insulin. If we just omit all rows with NAs that will cut our data in half. An alternative to just deleting them is to fill them with either the mean or the median of the column. In the code section shown below we use an if-else to either calculate the mean or median of a vector. Then we write a function to fill NAs of a vector with either the mean or the median. There are several new things to talk about in this code.

2.7.1 if-else

Within the function is an if statement. The if statement in R has this form:

```
if (condition) {statements if true} else {statements if false}
```

The ( ) around the condition is required, as are curly braces around the statements. Statements should occur on individual lines for readability, although R will allow multiple statements separated by a semicolon. We will discuss R style preferences below, but note that the opening is at the end of a line and the closing is on a line by itself. Our if-else statement lets us use either mean or median, depending on the choice sent to the function.

We will see examples of the ifelse shortcut throughout the book. Here is the format:

```
ifelse(cond, true, false)
```

Code 2.7.1 — Function Example. Third Code Chunk

```
fill_NA <- function(mean_med, v){
  # fill missing values with either 1=mean or 2=median
  if (mean_med == 1){
    m <- mean(v, na.rm=TRUE)
  } else {
    m <- median(v, na.rm=TRUE)
  }
  v[is.na(v)] <- m
  v
}

# make a new data set with NA's filled
df <- PimaIndiansDiabetes2
df$triceps <- fill_NA(1, df$triceps)
df$insulin <- fill_NA(1, df$insulin)
df <- df[complete.cases(df),]
```

2.7.2 Defining and Calling Functions

The code above shows a user-defined function. The definition format is:

`name <- function(args) {statements}`

The next-to-last line of our function replaced all NAs in the vector with the mean (or median). Notice that there is no "return" statement. The result of the last statement in the function is what is returned, which in this case is our updated vector. Notice again that the opening curly brace for the function is at the end of the line and the closing curly brace is on a line by itself.

Our calling statements simply have the name of the function and the arguments. The result of the function replaces the original values of those vectors. The last line of the third code chunk handles the remaining missing value which are few. Our data set df will have 724 observations.

For an additional example of a function in R, let's look at a recursive function for the familiar Finonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...

Code 2.7.2 — A Recursive Function. Fibonacci Sequence

```
fib <- function(n){
  if (n <= 1){
    return(n)
  } else {
    return (fib(n-1) + fib(n-2))
  }
}

# print a sequence
for (i in 1:8)
    print(fib(i))
```

Try this out on your computer. Note in the above code that we used return(). However, the function would work correctly without the surrounding return(). For example, instead of `return(n)` just `n`.

2.7.3 Loops

R has for loops and while loops. The formats are:

```
for (condition) {statements}
while (condition) {statements}
```

Figure 2.11 shows a couple of examples of for loops. The first loop creates 3 linear models. The linear model is the first algorithm we will learn. The formula "glucose~df[,cols[i]]" will learn 3 models: glucose as a function of mass, age, and number of pregnancies because these are the columns selected in cols. These three models are stored in a list for display later.

First we plot mass on the x axis and glucose on the y axis. Then we plot 3 ablines, one for each stored model. The abline function can be used to plot the regression line as we are doing here, but can also be used to plot straight lines. We make each line a different color by just letting color =i, our index. Notice that we had to use the double square bracket to index the list items. Finally we add a legend using the same color codes as the ablines.

```r
cols <- c(6,8,1)
plot(df$mass, df$glucose, main="PimaIndianDiabetes2")
for (i in 1:3){
    model <- lm(glucose~df[,cols[i]], data=df)[1]
    abline(model, col=i)
}
legend("topright", title="Predictors", c("Mass", "Age", "Pregnant"), fill=c(1,2,3))
```

Figure 2.8: Fourth Code Chunk

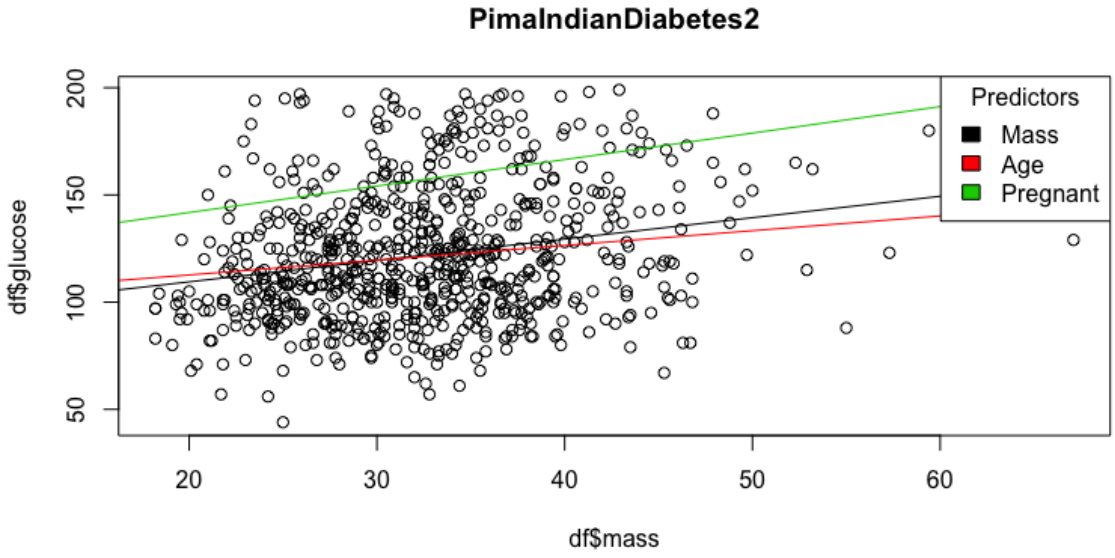

Figure 2.9: Graph with Regression Lines

Finally, we want to demonstrate the ifelse() shortcut. The ifelse() below checks if insulin is greater than 155. If this is true, the element will be 1 otherwise it will be 0. The ifelse() creates an integer vector, which is converted to a factor vector by factor(). Next we plot the observations

again, this time color coding those with high insulin as red, and those that do not have high insulin as blue.

```
```{r}
df$large <- factor(ifelse(df$insulin>155,1,0))
plot(df$mass, df$glucose, pch=21, bg=c("blue","red")[unclass(df$large)])
```
```

Figure 2.10: Fifth Code Chunk

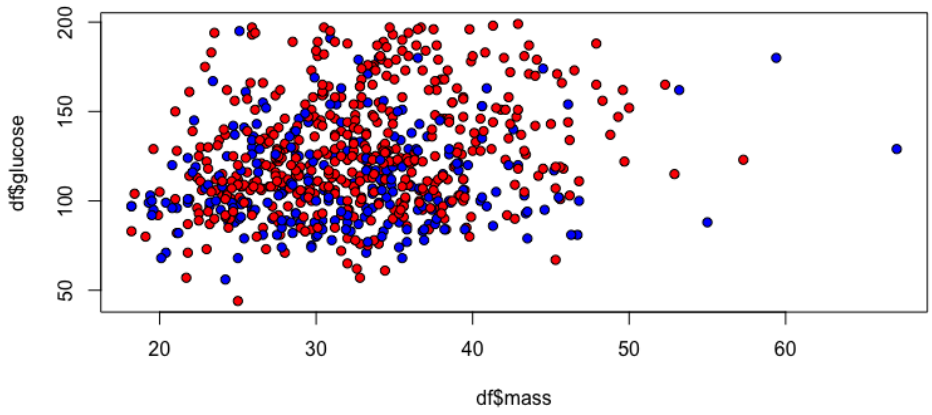

Figure 2.11: Graph with Color Coding

2.8 R Style

Google has its own R style guide available here: `https://google.github.io/styleguide/Rguide.xml`. Another style guide, written by Hadley Wickham, is available here: `http://adv-r.had.co.nz/Style.html`. Hadley Wickham is the Chief Scientist at RStudio, and an influential R developer, having created several amazing open-source packages such as ggplot2, dplyr and more. The point is not to follow one style guide rather than the other, but to be consistent in your code. It makes it easier to read, even for you.

Here are a few recommendations from Hadley Wickham's style guide:

- variable and function names should be lowercase, using underscore to separate names
- variable names generally should be nouns and function names should be verbs
- try to use names that are concise but meaningful
- don't use names of existing functions; R will let you override them
- put a space around operators, except :
- otherwise don't add unnecessary spaces
- opening curly braces should not be on their own lines
- closing curly braces should go on its own line unless it's followed by else
- indent with 2 spaces, exception: function arguments

2.9 Summary

This chapter provided a fast-paced introduction to R. Don't expect to retain everything in the chapter right now, you can refer back to it as you need it. You will learn more R as we go, but now you already know enough to get started with machine learning in the next chapter. The github site has an R cheat sheat that will be helpful as you start coding your own scripts.

Also in the github are full notebooks of the data exploration and R control structure examples in this chapter. Make sure you understand the R code in these notebooks. Link: `https://github.com/kjmazidi/Learning_from_Data`