Contents

Welcome to R!

Welcome to using R for predictive analytics. This document will walk you through several exercises to demonstrate the most commonly used functions in R. In the text below, we have written code for you to use to explore what R can do. The R code has a gray background; selected output is printed in indented sections. We recommend creating a script for yourself as you walk through each line of code. In RStudio, the keyboard shortcut Ctrl + shift + N will create an empty script for you to start with. Storing code in a script allows you to repeat analysis without repeating work. To run the code directly from your script, highlight it and use the keyboard shortcut Ctrl + Enter.

Macintosh users will have the same keyboard shortcut options, but with different key combinations.

Introduction

Calculations, variables, and vectors

You can do basic arithmetic in R.

```
1 + 1
```

You can assign values and computed values to variables and return them to the console.

```
a <- 2
a <- 1 + 1
a
```

Note: You have the option to use "=" to assign values instead of "<-", however most R coding standards suggest the arrow is better because it is more clear about the direction of the assignment. Additionally, some functions allow for "<-" assignments as inputs, and in these cases the "=" sign will likely return an error. For example, which variable in the second line below is getting a new value?

```
b = 3
a = b
a
b
```

You can use variables to do computations.

```
x <- 5
y <- 2
z <- x*y
z
```

You can add comments to your code without changing what happens.

```
z # This will output the contents of the "z" object
```

There are many mathematical functions built into the base R package. Here are a few.

```
x <- -2
abs(x)
y <- exp(x)
y
log(y)
log(y, base = 10)
```

If you ever forget what a function requires as input, or what it outputs, you can ask R to remind you. *Utilizing R's help functions may be one of the most important tools we use on a regular basis.*

```
?runif()
help(seq)
```

R follows a standard order of operations like most graphing calculators.

```
((x-y)^z/z^x)/pi
```

R has many ways to create and store data in vectors. Use the following code to create ten vectors, then check what each looks like. How many are the same?

```
x1 <- c(1, 2, 3)
x2 <- 1:3
x3 <- c(1:3)
x4 <- seq(1, 3)
x5 <- seq(1, 3, 1)
x6 <- seq(1, 3, .5)
y1 <- c(1, 1, 1)
y2 <- rep(1, 3)
z1 <- c(rep(1, 3), seq(1, 3))
z2 <- c(y2, x4)
```

If you want to create, store, and show something all at once, wrap your code with parentheses.

```
(x7 <- c(1, 2, 3))
```

R can easily create vectors of random numbers from selected distributions.

```
a1 <- runif(5, 2, 4)
a2 <- rbeta(5, 1, 1)
a3 <- rnorm(5, 2, 4)
a4 <- rexp(5, 1)
```

Functions designed for single-value inputs can typically produce n outputs for n inputs.

```
a1^2
c(9, 15, 35, 22, 0)*9/5 + 32
```

Some functions take an input vector and produce a single output value. At times, these functions will produce an error or NA value when used on a single value rather than a vector.

```
sd(5)
sd(c(1,2,3))
max(a3, 0)
pmax(a3, 0) # See the difference between the max functions?
mean(na.rm = TRUE, x = a1) # You can rearrange input order if inputs are labeled.
```

Matrices and dataframes

Try using the following code to create two-dimensional objects. These are typically of the R class "matrix" or "data.frame".

```
m1 <- cbind(y2, x4)
m2 <- rbind(y2, x4)
m3 <- matrix(rep(1, 25), 5, 5)
(df1 <- data.frame(Column1 = x1, Column2 = x2, x3, x4, x5))
```

R uses the names() function to describe the pieces of an object, and to allow the user to supply meaningful names.

```
names(m3)
names(df1)
names(df1) <- c("test1", "test2", "test3", "test4", "test5")
```

Take another look at the dataframe with its new names.

```
df1
```

```
  test1 test2 test3 test4 test5
1     1     1     1     1     1
2     2     2     2     2     2
3     3     3     3     3     3
```

Now that you've got a table, how can you pull values out of it? There are many options for indexing in R. Each of the following indexing techniques can also be used to put a value into a cell of a matrix or dataframe (or vector or variable).

```
x1[1]
x1[1:2]
x1[c(1,2)]
df1[1]
df1[, 1]
df1$test1
df1[1, ]
df1[3, 1]
df1$test1[3]
df1$test1[3] <- 3
```

Basic coding skills

One of the most important skills to learn in any programming language is how to iteratively perform a predefined task. R has two simple methods for this, the For loop and the While, or Conditional, loop.

For loop

```
for(i in 1:10){
  print((i*2 + 10)/2 - i)
}
```

Conditional loop; note that both outputs are the same, a vector of fives.

```
k = 1
while(k <= 10){
  print((k*2 + 10)/2 - k)
```

```
  k <- k + 1
}
```

Workspace maintenance

R is not aware of your intentions, so it may keep more in memory than is necessary. If you are working with large datasets, this can quickly become a problem, causing R to crash when it reaches your computer's capacity. Cleaning up your workspace with the following steps from time to time can help.

First, find out what objects are in memory.

```
ls()
```

Second, check how much memory you are using and your limit. These outputs are in megabytes.

```
memory.size()
memory.limit()
```

Third, remove unneeded objects from memory, and clean up memory

```
rm(x, y, z, a, a1, a2, a3, a4, df1, i, k, m1, m2, m3,
   x1, x2, x3, x4, x5, x6, y1, y2, z1, z2)
```

Finally, clean up your memory.

```
gc()
memory.size()
```

Compare the resulting output to that of the prior memory.size() call to start getting a feel for how R handles storage. Be sure to try this again later when working with larger datasets.

Data exploration

Dataframe summary statistics

Let's pull in some data.

R has many built-in datasets in a package called datasets. There are several features in R to help you investigate what's available. In addition to the coding options below, you can also use the Packages screen at the bottom right of RStudio, scroll down to System Library, and click on datasets.

```
?datasets
library(help = "datasets")
??iris
?iris
```

To take a quick look at what's in the data, the following functions are useful:

- names() returns the field names in the dataset
- head() returns the first several rows of the dataset
- pairs() creates a scatter plot for each pair of fields in the dataset. Be careful not to use this with too many fields!
- class() returns the class of any object or any piece of an object
- summary() returns a summary of each field: quartiles for numeric fields and counts for the top few levels of factor fields
- str() gives a combination of information from class() and head()

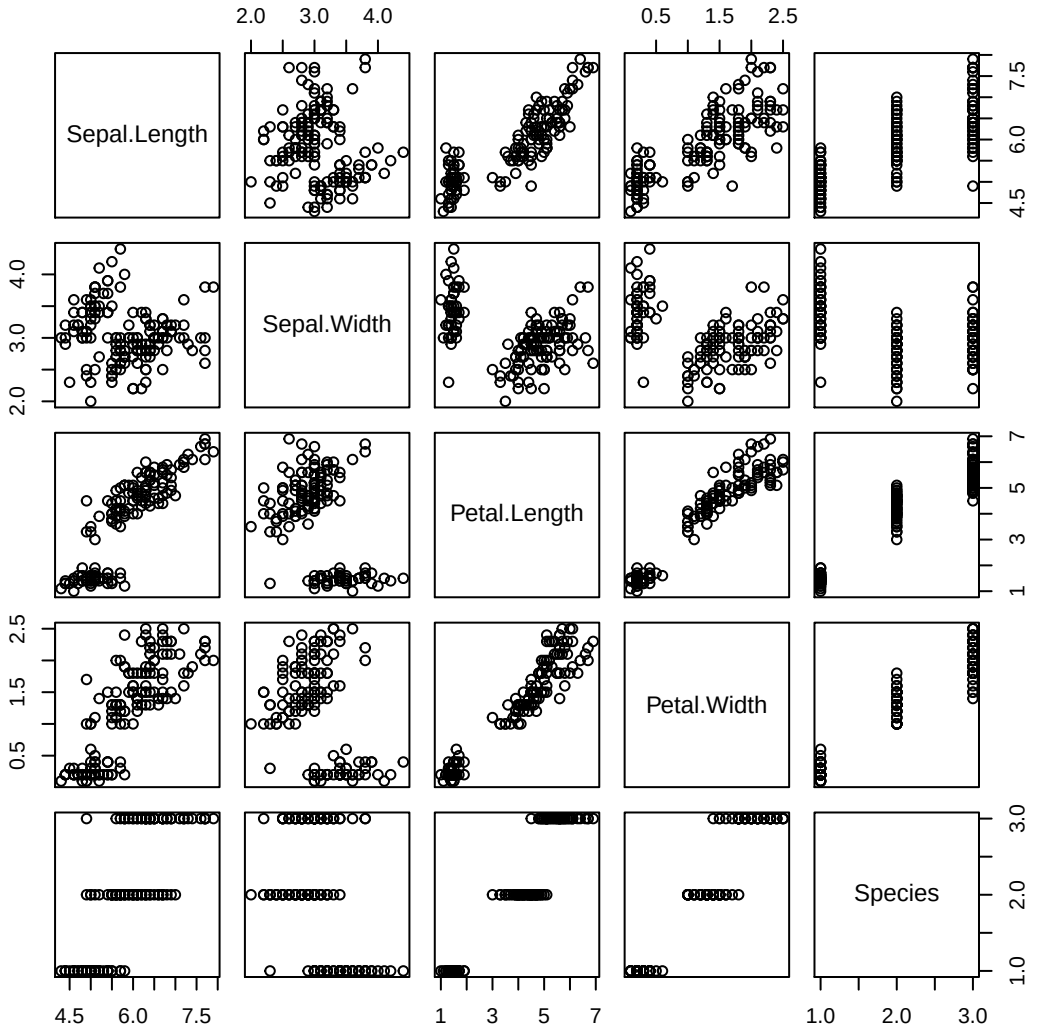
```
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
pairs(iris)
```




```
class(iris)
```

```
[1] "data.frame"
```

```
c(class(iris$Sepal.Length), class(iris$Sepal.Width),
  class(iris$Petal.Length), class(iris$Petal.Width),
  class(iris$Species))
```

```
[1] "numeric" "numeric" "numeric" "numeric" "factor"
```

```
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

```
str(iris)
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

##Defaults You may have noticed that R has some default formats for displaying the summary statistics above. These can be changed. It's good practice to store the defaults for later use in case you regret the changes you make.

For example, the following code shows how to save defaults, change output to display 1 significant digit, then reset defaults to their original values.

```
?options
```

```
op.default <- options()
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

```
options(digits = 1)
summary(iris)
```

```
 Sepal.Length  Sepal.Width  Petal.Length  Petal.Width        Species
Min.   :4     Min.   :2    Min.   :1     Min.   :0.1   setosa    :50
1st Qu.:5     1st Qu.:3    1st Qu.:2     1st Qu.:0.3   versicolor:50
Median :6     Median :3    Median :4     Median :1.3   virginica :50
Mean   :6     Mean   :3    Mean   :4     Mean   :1.2
3rd Qu.:6     3rd Qu.:3    3rd Qu.:5     3rd Qu.:1.8
Max.   :8     Max.   :4    Max.   :7     Max.   :2.5
```

```
options(op.default)
```

##Variable summary statistics Factor and numeric variables receive different treatment in R. The summary() function returns what R believes is the most relevant information for each. If you are interested in more information about a selected field, there are additional functions that can help.

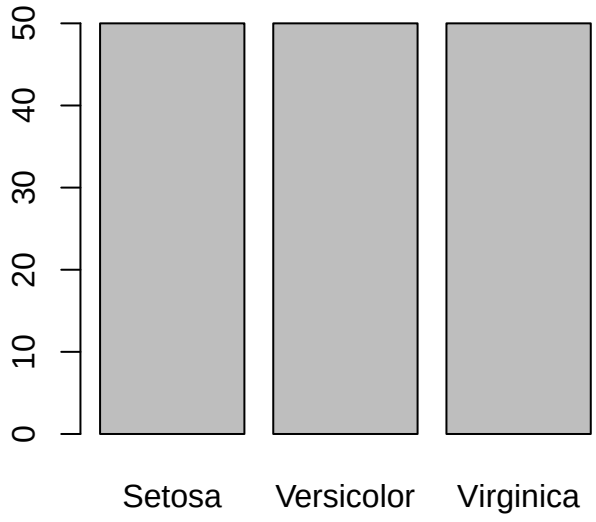
For factor variables, try these functions.

```
unique(iris$Species)
levels(iris$Species)
summary(iris$Species)
```

```
table(iris$Species, useNA = "ifany")
```

```
    setosa versicolor  virginica
        50         50         50
```

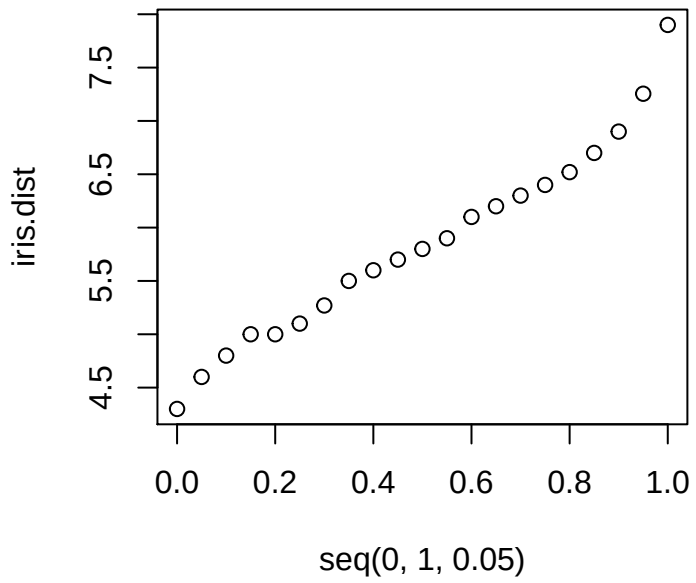
```
barplot(table(iris$Species, useNA = "ifany"),
        names.arg = c("Setosa", "Versicolor", "Virginica"))
```



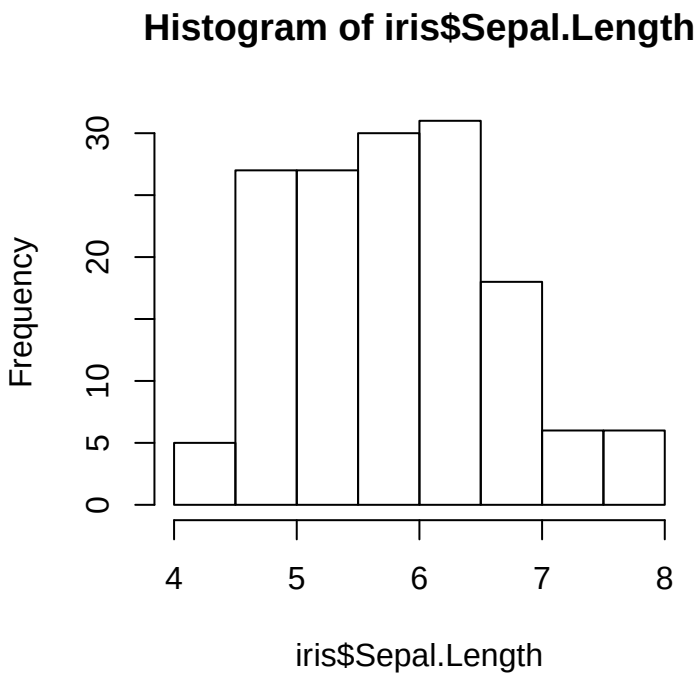
For numeric variables, you may want to look at quantiles or histograms.

```
summary(iris$Sepal.Length)
quantile(iris$Sepal.Length, probs = 0.25)
quantile(iris$Sepal.Length, probs = seq(0, 1, .05))
iris.dist <- quantile(iris$Sepal.Length, probs = seq(0, 1, .05))
```

```
plot(seq(0, 1, .05), iris.dist)
```




```
hist(iris$Sepal.Length)
```



Import and export

R selects a directory in which it will assume you are working.

```
getwd()
orig.wd <- getwd()  # Save it for later.
```

Set a new working directory for the current project. Ours lives in the C drive, under "Work" and then "SOASeminar". R lets you name directories with two different syntaxes, neither of which tie to normal directory syntax in Windows.

```
setwd("C:\\Work\\SOASeminar")
setwd("C:/Work/SOASeminar")
```

In case your coworker insists on using Excel, you can write out a dataset from the dataset package.

```
write.table(iris, "iris.csv", row.names = FALSE, sep = ",")
write.csv(iris, "iris.csv", row.names = FALSE)
```

If you want to show you can do all the same work in R, you'll need to be able to read data into R's environment. We've provided a comma-separated file for you to practice with. Replace the file path in this code with your selected location. For .csv files, either command below will work. read.table(), like write.table(), is more flexible, which may be necessary if your data is in another format.

```
iris2 <- read.table("iris2.csv", header = TRUE, sep = ",")
iris2 <- read.csv("iris2.csv", header = TRUE)
summary(iris2)
```

```
      Species     petalwidth     Petal.Size
 setosa    : 6   Min.   :0.100   large :10
 versicolor: 9   1st Qu.:1.050   medium: 7
 virginica :12   Median :1.500   small :10
```

```
Mean   :1.411
3rd Qu.:1.850
Max.   :2.500
```

Important R packages

The instructors of the 2020 Practical Predictive Analytics Seminar especially like the following packages: dplyr, tidyr, zoo, lubridate, ggplot2, shiny, caret, survival, xlsx, mass, mgcv, car.

If you don't already have them installed, this is the syntax to install packages one at a time:

```
install.packages("dplyr")
```

Once you've installed a package, you must load it to use its functions:

```
library(dplyr)
packageVersion("dplyr")
```

```
## [1] '0.8.5'
```

Data manipulation

Base R has some useful functions for manipulating data, and as with everything else, there are multiple ways to accomplish whatever tasks you dream up.

Creating fields

```
iris <- data.frame(iris, Sepal.Area = iris$Sepal.Length*iris$Sepal.Width)
```

The dplyr package has a few more useful functions, and uses considerably less memory. The mutate function is used for generating new columns (or modifying existing columns*).

```
iris <- iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length)
```

Note: mutate() will copy over column names that already exist.

Joining data sources

There are functions to join tables, with inputs that allow you to match columns (which do not need to have the same column names).

```
iris.combined <- iris %>%
  left_join(iris2,
            by = c("Species" = "Species", "Petal.Width" = "petalwidth"))
```

The third column from iris2.csv is now joined onto the existing iris dataset, matched by the species and petal width columns.

More on creating fields

There is always more than one way to define the same field. Be careful when adding new fields that you use "vectorized" functions as needed. The difference between the functions max() and pmax(), a vectorized version of max(), is exhibited below.

```
iris.combined <- iris.combined %>%
  mutate(Petal.Area = Petal.Width * Petal.Length,
         SP.Ratio = Sepal.Area/Petal.Area,
         SP.Ratio.Floor = pmax(SP.Ratio, 5),
```

```
         SP.Ratio.BadFloor = max(SP.Ratio, 5),
         SP.Ratio.Floor2 = ifelse(SP.Ratio > 5, SP.Ratio, 5))
iris.combined$SP.Ratio.BadFloor # Oops!
```

More on dropping fields

Subsetting a dataset can be done equivalently with either base R functions or a dplyr function. Again, dplyr's coding syntax is cleaner.

```
iris.Big <- iris.combined[iris.combined$Petal.Area > 1,]
iris.Big2 <- iris.combined %>%
  filter(Petal.Area > 1)  # This is a row filter
iris.Skinny <- iris.combined[,names(iris.combined) %in% c("Species", "Petal.Length", "Petal.Width")]
iris.Skinny2 <- iris.combined %>%
  select(Species, Petal.Length, Petal.Width) # This is a column "filter"
```

Note, however, that these have not automatically updated what R thinks is important to know about the field. A factor column that lost some levels due to filtering will still believe that it has those levels. We can fix that if needed.

```
levels(iris.Big$Species)
levels(iris.Big2$Species)
unique(iris.Big$Species)
unique(iris.Big2$Species)
iris.Big2 <- iris.Big2 %>%
  mutate(Species = factor(Species)) # Re-level the factor variable
levels(iris.Big2$Species)
```

More on data summaries

dplyr functions can also be used for pivot table-like functionality. We briefly show a more compact way that dplyr functions can be used, but then we return to the more intuitive syntax using "piping" (%>%). Piping is considered a best practice when it comes to dplyr coding.

```
summarize(group_by(iris.combined, Species, Petal.Size),
          NumObs = n(),
          AvgArea.Sepal = mean(Sepal.Area),
          AvgArea.Petal = mean(Petal.Area))

# Same output
iris.combined %>%
  group_by(Species, Petal.Size) %>%
  summarize(NumObs = n(),
            AvgArea.Sepal = mean(Sepal.Area),
            AvgArea.Petal = mean(Petal.Area))
```

The tables can be stored for use later

```
iris.summary <- iris.combined %>%
  group_by(Species, Petal.Size) %>%
  summarize(NumObs = n(),
            AvgArea.Sepal = mean(Sepal.Area),
            AvgArea.Petal = mean(Petal.Area))
```

Visualization

Warning: This is where we pick up the pace and start using R's flexibility to our advantage. *This is a good time to remind you to use the "?" and "help" functions to explore how other functions work.*

We've demonstrated histograms for getting a quick understanding of data distributions.

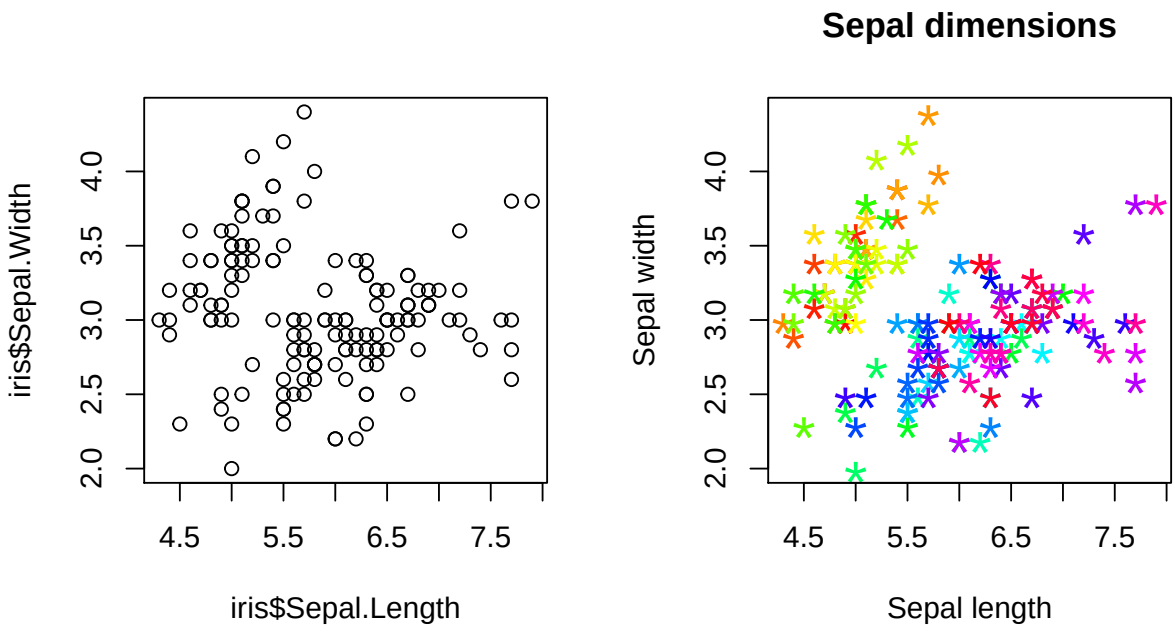
```
hist(iris$Sepal.Length)
```

Let's take this a bit further. Store a histogram into a variable, then look at all of the pieces R has created. You can store these pieces and use them to create custom graphs or tables.

```
SL.hist <- hist(iris$Sepal.Length, plot = FALSE)
names(SL.hist)
data.frame(Break = SL.hist$breaks[-1], Count = SL.hist$counts)
```

Every plot has many features that can be overwritten from their default values. Even the plot window itself can be manipulated to contain multiple plots, and here we use the graphical parameters function "par" and the opton "mfrow" to do just that. In the second plot, we also include labels (xlab, ylab, main), colors (col), a new point symbol (pch), and a new point size (cex).

```
par(mfrow = c(1, 2))
plot(iris$Sepal.Length, iris$Sepal.Width)
plot(iris$Sepal.Length, iris$Sepal.Width,
     xlab = "Sepal length",
     ylab = "Sepal width",
     main = "Sepal dimensions",
     col = rainbow(150),
     pch = "*",
     cex = 2)
```



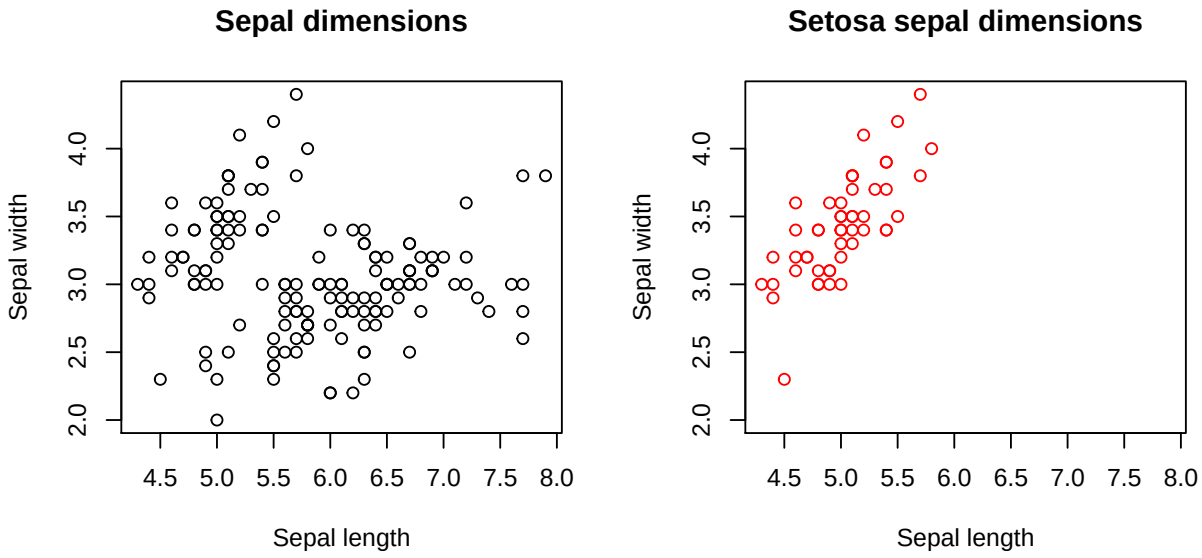
Note: there are plotting examples from the ggplot2 package later in this section.

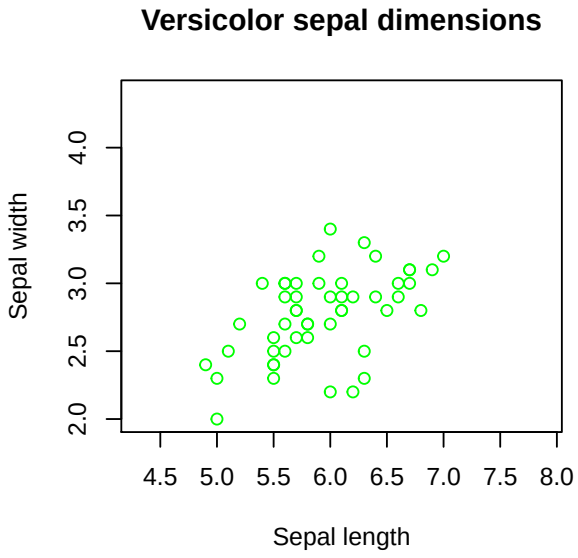
You can create a vector to set a color based on values in the data, such as Species in the following example.

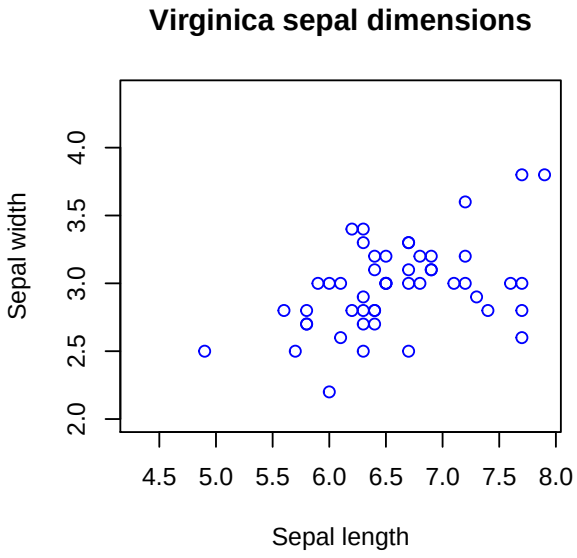
```
iris <- iris %>%
  mutate(iris.col = ifelse(Species == "setosa", "red",
                           ifelse(Species == "versicolor", "green", "blue")))
```

If you want to save a plot from RStudio for use later, use the functions "windows", "savePlot", and "dev.off". We have included those functions below as an example, but commented them out because they are not made for rmarkdown documents like this one. There are "point-and-click" options, as well, to save plots in R Studio.

```
#windows()
par(mfrow = c(2, 2))
plot(iris$Sepal.Length, iris$Sepal.Width,
     xlab = "Sepal length", ylab = "Sepal width", main = "Sepal dimensions")

plot(iris$Sepal.Length[which(iris$Species == "setosa")],
     iris$Sepal.Width[which(iris$Species == "setosa")],
     xlab = "Sepal length", ylab = "Sepal width", main = "Setosa sepal dimensions",
     xlim = c(min(iris$Sepal.Length), max(iris$Sepal.Length)),
     ylim = c(min(iris$Sepal.Width), max(iris$Sepal.Width)),
     col = iris$iris.col[which(iris$Species == "setosa")])

plot(iris$Sepal.Length[which(iris$Species == "versicolor")],
     iris$Sepal.Width[which(iris$Species == "versicolor")],
     xlab = "Sepal length", ylab = "Sepal width", main = "Versicolor sepal dimensions",
     xlim = c(min(iris$Sepal.Length), max(iris$Sepal.Length)),
     ylim = c(min(iris$Sepal.Width), max(iris$Sepal.Width)),
     col = iris$iris.col[which(iris$Species == "versicolor")])

plot(iris$Sepal.Length[which(iris$Species == "virginica")],
     iris$Sepal.Width[which(iris$Species == "virginica")],
     xlab = "Sepal length", ylab = "Sepal width", main = "Virginica sepal dimensions",
     xlim = c(min(iris$Sepal.Length), max(iris$Sepal.Length)),
     ylim = c(min(iris$Sepal.Width), max(iris$Sepal.Width)),
     col = iris$iris.col[which(iris$Species == "virginica")])
```







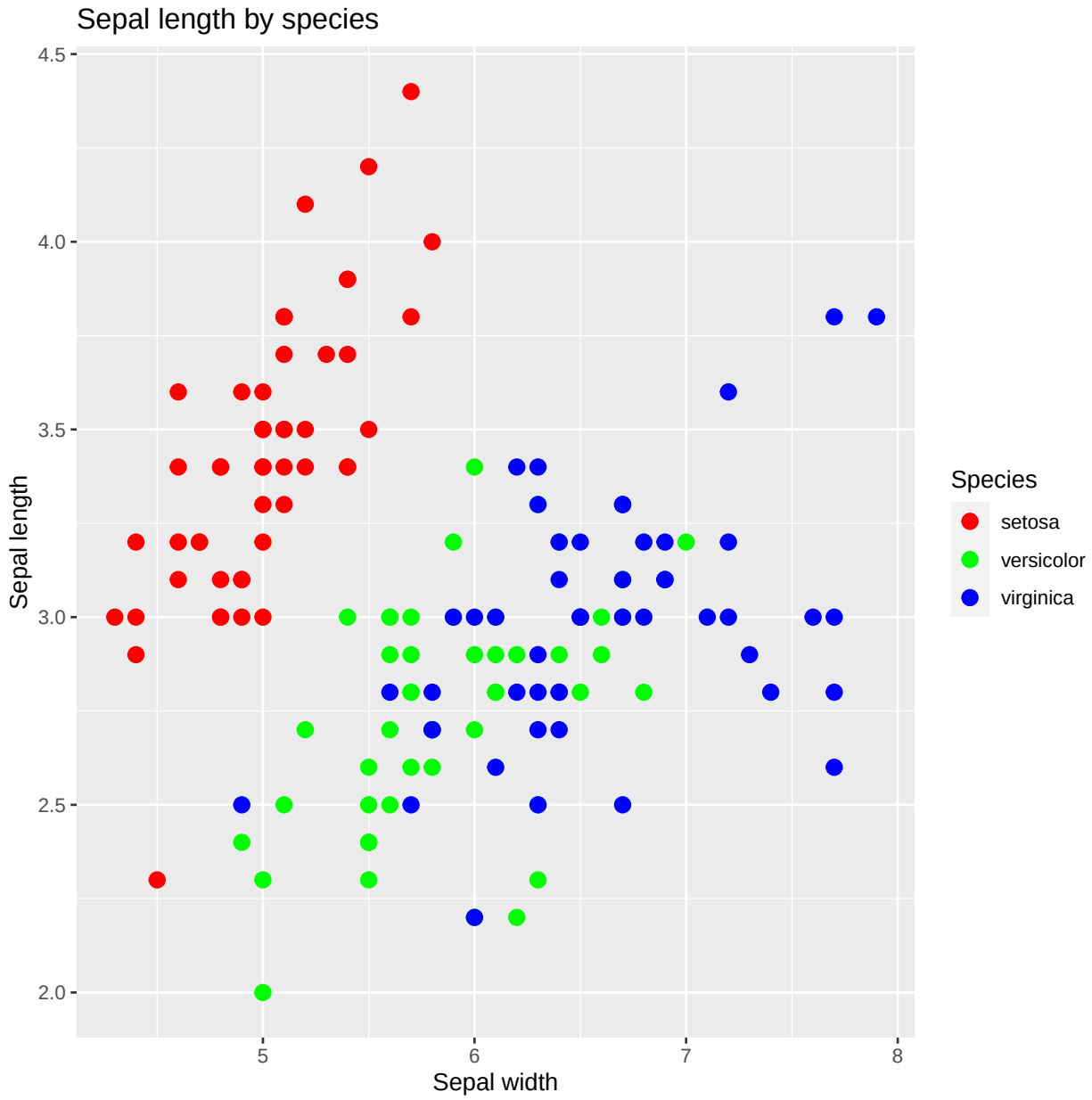

```
#savePlot("C:\\Work\\SOASeminar\\SplitSepalDim.jpg", type = "jpg")
#dev.off()
```

Note that we used indexing within the plot function to display distinct subsets of our data, and utilized our color scheme from earlier.

The ggplot2 package can be very useful when dealing with grouped data. Let's create a plot using ggplot that gives the same information as in the previous set of plots.

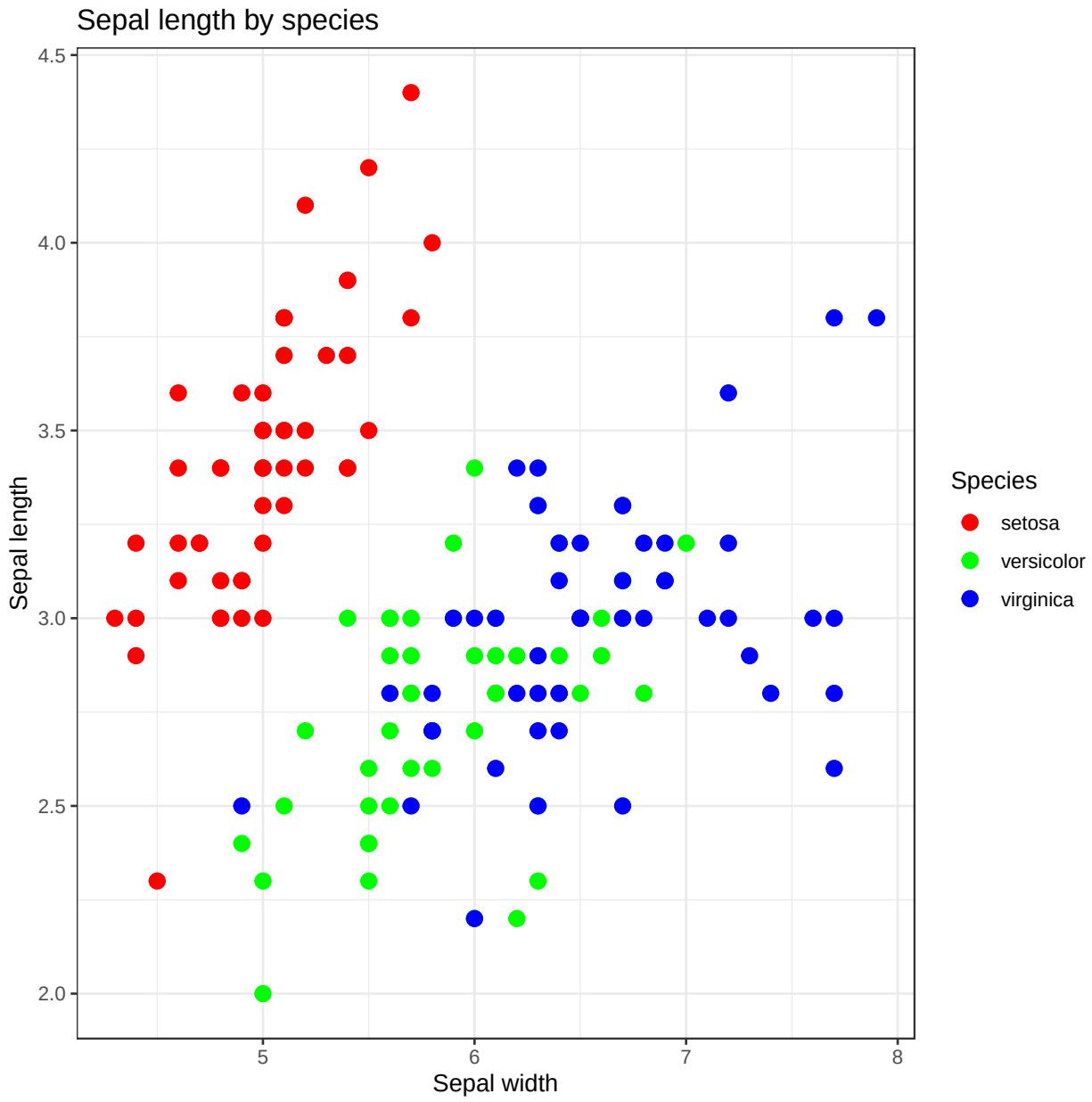
```
library(ggplot2)

FlowerPlot <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, colour = Species)) +
     geom_point(size = 3) +
     labs(x = "Sepal width", y = "Sepal length", title = "Sepal length by species") +
     scale_colour_manual(values = c("setosa" = "red", "versicolor" = "green", "virginica" = "blue"))
```

```
FlowerPlot
```



Note that we've stored the ggplot output as a variable. This is not necessary but does allow us to easily make edits to the plot. For example, we can replace the grey background with a white background as shown below.

```
FlowerPlot + theme_bw()
```



R Projects

At this point, it is probably a good idea to mention R Projects. R Projects are designed to streamline some of the nitty gritties that become annoying when you stop and start the same work over a period of time. Working within an R Project automates features like the following every time you open the project:

- Clears current R session
- Sets working directory to that where the project is located
- Loads selected data, console history, and options
- Allows for version control using Git and GitHub, and connects to a selected GitHub repository
  - For more information on using Git in RStudio:
  - https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN

To create a new R Project, just select "New Project..." under the "File" dropdown in RStudio. You will then get to choose between a new directory, an existing directory, or cloning a project from an existing Git repository. For a more detailed look at projects:

- https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects

Analysis

Fitting a linear model

Finally! Fitting a linear model is very easy in R.

From the prior plots, it's apparent that there could be a relationship between sepal length and sepal width, which could be different among the species. it's less clear that there is a universally applicable relationship. Let's investigate whether the relationship is statistically signifcant, starting with the relationship as a global phenomenon. The syntax is somewhat similar to the plot function.

```
lm00 <- lm(iris$Sepal.Width ~ iris$Sepal.Length)
summary(lm00)
```

```

Call:
lm(formula = iris$Sepal.Width ~ iris$Sepal.Length)

Residuals:
    Min      1Q  Median      3Q     Max
-1.1095 -0.2454 -0.0167  0.2763  1.3338

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        3.41895    0.25356   13.48   <2e-16 ***
iris$Sepal.Length -0.06188    0.04297   -1.44    0.152
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4343 on 148 degrees of freedom
Multiple R-squared:  0.01382,   Adjusted R-squared:  0.007159
F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

P-values for each coefficient in the model are outputted by the summary function. This summary shows that the coefficient for Sepal.Length is not statistically significant at the 0.05 level, meaning that one may not be confident estimating/predicting sepal width from knowledge of sepal length, when we consider all flower species together.

What happens if we look at the three species separately? We can take a first step by looking at the interaction between Sepal.Length and Species.

```
lm01 <- lm(Sepal.Width ~ Sepal.Length:Species, data = iris)
# Providing the data frame as an input prevents having to use "iris$"
summary(lm01)
```

```

Call:
lm(formula = Sepal.Width ~ Sepal.Length:Species, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
```

```
-0.86711 -0.15751  0.00376  0.17614  0.60926

Coefficients:
                              Estimate Std. Error t value Pr(>|t|)
(Intercept)                    0.78852    0.27026   2.918  0.00408 **
Sepal.Length:Speciessetosa     0.52857    0.05430   9.735  < 2e-16 ***
Sepal.Length:Speciesversicolor 0.33370    0.04567   7.306 1.65e-11 ***
Sepal.Length:Speciesvirginica  0.33083    0.04109   8.052 2.62e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2785 on 146 degrees of freedom
Multiple R-squared:    0.6, Adjusted R-squared:  0.5917
F-statistic: 72.99 on 3 and 146 DF,  p-value: < 2.2e-16
```

This model fit a separate slope coefficient for each species. The summary shows that each slope coefficient is statistically significant at the 0.05 level, and thus Sepal.Length contains valuable information for estimating/predicting Sepal.Width when we look at the three species separately.

We can take this one step further. If we add both the interaction term and an intercept term for each species, we can allow for a fully different regression line for each species within a single linear model.

```
lm02 <- lm(Sepal.Width ~ Species + Sepal.Length:Species, data = iris)
summary(lm02)
```

```
Call:
lm(formula = Sepal.Width ~ Species + Sepal.Length:Species, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.72394 -0.16327 -0.00289  0.16457  0.60954

Coefficients:
                              Estimate Std. Error t value Pr(>|t|)
(Intercept)                   -0.56943    0.55386  -1.028 0.305622
Speciesversicolor              1.44158    0.71304   2.022 0.045056 *
Speciesvirginica               2.01574    0.68609   2.938 0.003848 **
Speciessetosa:Sepal.Length     0.79853    0.11037   7.235 2.55e-11 ***
Speciesversicolor:Sepal.Length 0.31972    0.07537   4.242 3.95e-05 ***
Speciesvirginica:Sepal.Length  0.23189    0.06118   3.790 0.000221 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2723 on 144 degrees of freedom
Multiple R-squared:  0.6227,    Adjusted R-squared:  0.6096
F-statistic: 47.53 on 5 and 144 DF,  p-value: < 2.2e-16
```

Success! Each of the coefficients is statistically significant at the 0.05 level, and this model is even better according to adjusted R-squared. For those less familiar with stastitical modeling, the adjusted R^2 value is a version of the more conventional R^2. The adjusted R^2 penalizes a model that has too many statistically insignificant predictor variables. As with the conventional R^2, an adjusted R^2 of zero would still suggest a poor model, and an adjusted R^2 of one would still suggest a very strong model. Unfortunately, when we get into general linearized models (GLM's), we won't have such simple measures of model performance.

There is a shortcut in R for the last step, using "*" in place of the ":" above. The model will be equivalent, but the form will appear slightly different. If you were to estimate the sepal width for a pre-selected

combination of species and sepal length, both models would estimate the same value.

```
lm03 <- lm(Sepal.Width ~ Sepal.Length*Species, data = iris)
summary(lm03)
```

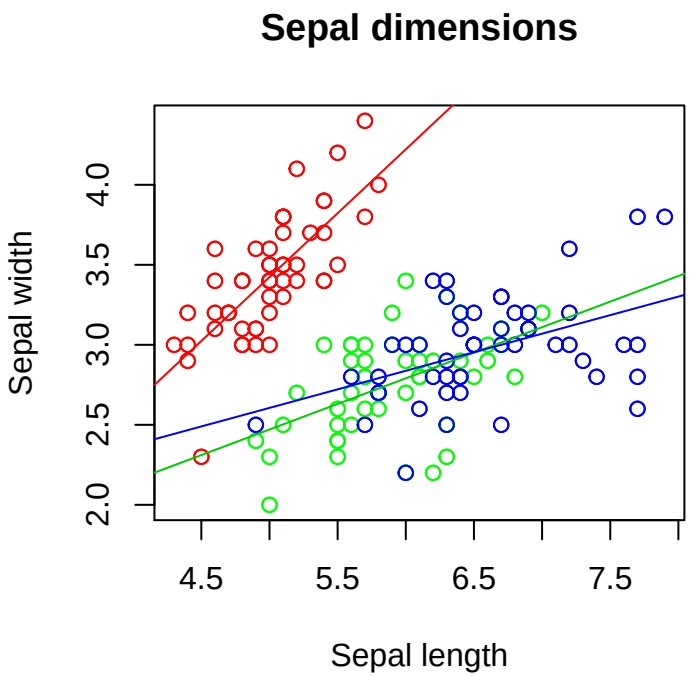
```
Call:
lm(formula = Sepal.Width ~ Sepal.Length * Species, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.72394 -0.16327 -0.00289  0.16457  0.60954

Coefficients:
                             Estimate Std. Error t value Pr(>|t|)
(Intercept)                   -0.5694     0.5539  -1.028 0.305622
Sepal.Length                   0.7985     0.1104   7.235 2.55e-11 ***
Speciesversicolor              1.4416     0.7130   2.022 0.045056 *
Speciesvirginica               2.0157     0.6861   2.938 0.003848 **
Sepal.Length:Speciesversicolor -0.4788     0.1337  -3.582 0.000465 ***
Sepal.Length:Speciesvirginica  -0.5666     0.1262  -4.490 1.45e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2723 on 144 degrees of freedom
Multiple R-squared:  0.6227,    Adjusted R-squared:  0.6096
F-statistic: 47.53 on 5 and 144 DF,  p-value: < 2.2e-16
```

Let's put what we've learned on a plot. We'll need the intercept and slope terms separately for the three species, along with the global intercept.

```
names(lm02)
lm02$coeff
lm02$coefficients
```

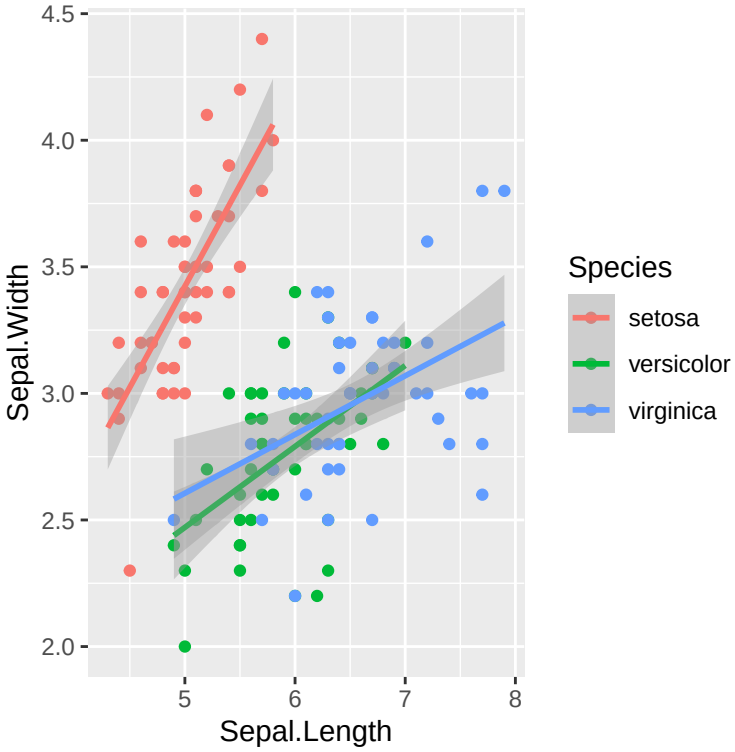
Use these to plot the relationship for each species, recalling the colors assigned earlier. A ggplot example is included.

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     xlab = "Sepal length", ylab = "Sepal width", main = "Sepal dimensions",
     col = iris$iris.col)
points(iris$Sepal.Length, iris$Sepal.Width,col = iris$iris.col)
abline(a = -.56943 + 0, b = 0.79853, col = 2)  #setosa in red
abline(a = -.56943 + 1.44158, b =  0.31972, col = 3)  #versicolor in green
abline(a = -.56943 + 2.01574, b = 0.23189, col = 4)  #virginica in blue
```




```
# ggplot example
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
## `geom_smooth()` using formula 'y ~ x'
```

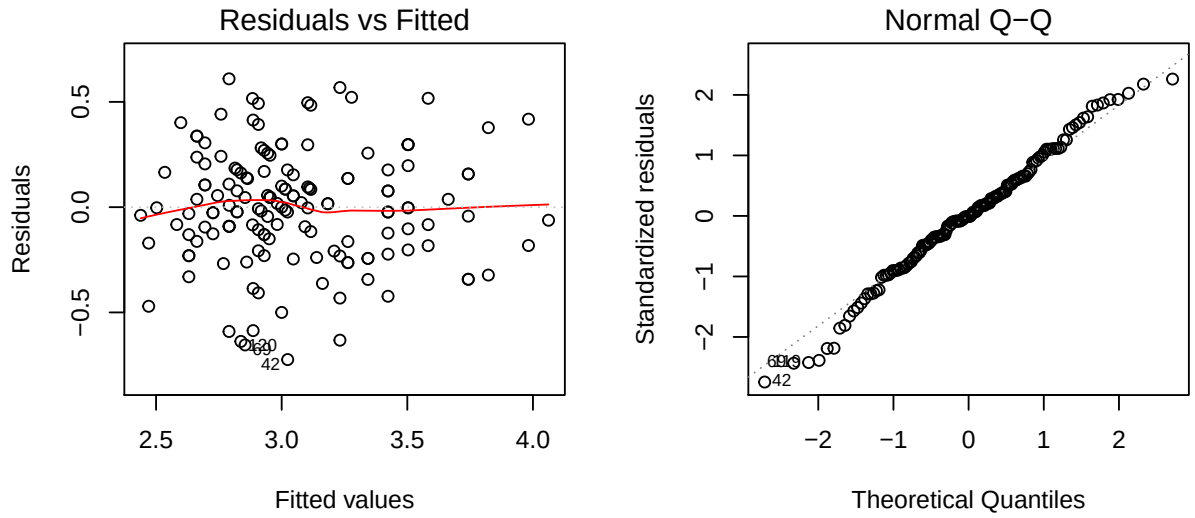


Let's plot residuals and generate predictions.

Plot residuals

The plot() function can be called iteratively to show several diagnostic plots about the selected model. We will discuss these more during the seminar.

```
par(mfrow = c(2, 2))
plot(lm02)
plot(lm02)
```




```
plot(lm02)
plot(lm02)
```

Generate predictions

Predictive analytics is all about predicting, so this last step is the end goal: what does our model predict as the value of the response variable, given what we know about the predictor variables? The following code will add predictions and their standard errors onto the working dataset.

```
iris <- iris %>%
  mutate(pred.SW = predict(lm02, iris),
         pred.SW.se = predict(lm02, iris, se.fit = TRUE)$se.fit)
```

Advanced R coding

This section provides code to help you explore more of what R can do, and more options in R for completing the same computations performed above.

Basic functions

```
tapply(iris$Petal.Width, iris$Species, mean) # One-variable pivot table
iris %>% group_by(Species) %>% summarize(Petal.Width.mean = mean(Petal.Width)) # same information as ab
sapply(1:150, function(x) (x+1)*2/x) # clean alternative to using loops
```

Matrix/vector functions

```
mean(iris$Sepal.Length)
apply(iris[,1:4], 2, mean) # The "2" input applies to columns
iris %>% # dplyr alternative
  summarize_at(.vars = vars(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
                       .funs = mean)

sd(iris$Sepal.Length)
apply(iris[,1:4], 2, sd)
iris %>% # dplyr alternative
  summarize_at(.vars = vars(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
                       .funs = sd)

quantile(iris$Sepal.Length, .9)
apply(iris[,1:4], 2,
      function(x){return(quantile(x, .9))}) # Slightly different function syntax here because additiona
iris %>% # dplyr alternative
  summarize_at(.vars = vars(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
                       .funs = function(x){return(quantile(x, .9))})

cor(iris$Sepal.Length, iris$Sepal.Width)
cor(iris[,1:4])
```

```
sapply(iris, class) # Applied to a data.frame/matrix
iris %>% # dplyr alternative
  summarize_all(.funs = class)

apply(iris[, sapply(iris, class) %in% c("numeric", "integer")], 2, sum)

apply(iris[, sapply(iris, class) %in% c("numeric", "integer")], 1, # The "1" input applies to rows
      function(x) x[1] - x[2] + x[3]/x[4])
```

Indexing with special characters

```
# | denotes "or"
iris %>%
  filter(Sepal.Width > quantile(Sepal.Width, .9) |
         Sepal.Width < quantile(Sepal.Width, .1))
# & denotes "and"
iris %>%
  filter(Species == "virginica" & Petal.Length > 6) %>%
```

```
  select(Petal.Length, Petal.Width)
# Use %in% for set membership
head(iris %>%
       filter(Species %in% c("setosa", "versicolor")))
```

Removing objects

```
rm(list = ls()) # Careful! This removes all objects from memory.
rm(list = ls()[ls() != "df1" & ls() != "a2"])
rm(list = ls()[!(ls() %in% c("df1", "a2"))])
```

Filter dataframes for specific records

```
iris.Big <- iris.combined[iris.combined$Petal.Area > 5
                          & iris.combined$Sepal.Area > 17,]
iris.Big <- filter(iris.combined, Petal.Area > 5 & Sepal.Area > 17)

table(iris.Big$Species)
# No setosa in the subset of the data
levels(iris.Big$Species)
# The Species factor preserves the original levels
# of the larger dataset, even if there are 0 of them

newspecies <- factor(iris.Big$Species)
levels(newspecies)
```

The factor() function resets the levels to what actually exists in the current iteration of the vector.

Extra Credit

Loops In Parallel

```
install.packages("doParallel")
library(doParallel)

registerDoParallel(2)
getDoParWorkers()
Sepal.Area <- foreach(i = 1:150, .inorder = T, .combine = c) %dopar% {
  iris$Sepal.Length[i]*iris$Sepal.Width[i]
}
```

Use .combine = c to append results to a vector/array

Time comparison

```
iris.bootstrap1 <- data.frame()
system.time(for(i in 1:10000){
  iris.bootstrap1 <- rbind(iris.bootstrap1, sample_n(iris, 1))
})

registerDoParallel(3)
system.time(iris.bootstrap2 <- foreach(i = 1:10000,
```

```
                                        .inorder = F,
                                        .combine = rbind,
                                        .packages = c("dplyr"))
            %dopar% {
  sample_n(iris, 1)
})
```

Use .combine = rbind to append a row to a dataframe

Note: Packages loaded into the global environment don't always carry over into your parallelization, so be sure to input a vector of packages you want in the loop.

Directory manipulation

Suppose you have Dropbox, Google Drive or some other shared drive on multiple machines with distinct filepaths. But it's worth pointing out that file path issues in shared directories can be avoided by using the aforementioned R projects.

```
if(file.exists("C:/Users/MachineName1")){
  setwd("C:/Users/MachineName1")
}else{
  setwd("C:/Users/MachineName2")
}
```

Now you could run a script regardless of which machine you're working on without editing the filepaths every time!

About this document

This document was created with Rmarkdown, a convenient tool for documenting what you've done. At the end, it prints out an html (or pdf or word) file that documents whichever code and output you request. An Rmarkdown script creates two environments within your R session: the global environment, and the Rmarkdown environment. This is beyond the beginner level in R, so we will distribute the Rmarkdown file at the end of the seminar.

Several additional packages may be needed in order to use Rmarkdown as well as an installation of LaTeX if you wish to produce a pdf.

Here are sites that will demonstrate more options:

- http://rmarkdown.rstudio.com/html_document_format.html
- https://github.com/jimhester/knitrBootstrap