Introduction to R Programming

Aziz Nanthaamornphong
College of Computing
Prince of Songkla University, Phuket Campus
E-mail: aziz.n@phuket.psu.ac.th


Map
Satellite
Arden St
Auction Rooms
Seven Seeds
Specialty Coffee
Melbourne Museum
Royal Exhibition
Building
Carlton
Gardens
Victoria St
North Melbourne
Queen Victoria Market
St Vincent's
Hospital Melbourne
Melbourne Netball Pty
State Library Victoria
St Patrick's Cathedral
Festival Hall
Melbourne Central
Parliament House
Melbourne Star
Observation Wheel
Chinatown Melbourne
The District Docklands
Flagstaff
CBD
Target
Parliament
Fitzroy Gardens
Grand Hyatt Melbourne
NEWQUAY
Cooks' Cottage
Waterfront City
Melbourne Town Hall
Forum Melbourne
Skybus
St Paul's Cathedral,
Melbourne
Jolimont
Docklands
Federation Square
Southern Cross
Flinders Street
Yarra River
Birrarung
Marr
Immigration Museum
Docklands
Park
Alexandra Gardens
Rod Laver Arena
SEA LIFE
Melbourne Aquarium
Arts Centre Melbourne
Lorimer St
Yarra River
National Gallery
of Victoria
Crown Melbourne
Melbourne Arena
Crown Casino and
Entertainment Complex
Kings Domain
DFO South Wharf
AAMI Park

Companies, Officials and NGO Using R

https://github.com/ThinkR-open/companies-using-r/blob/master/README.md

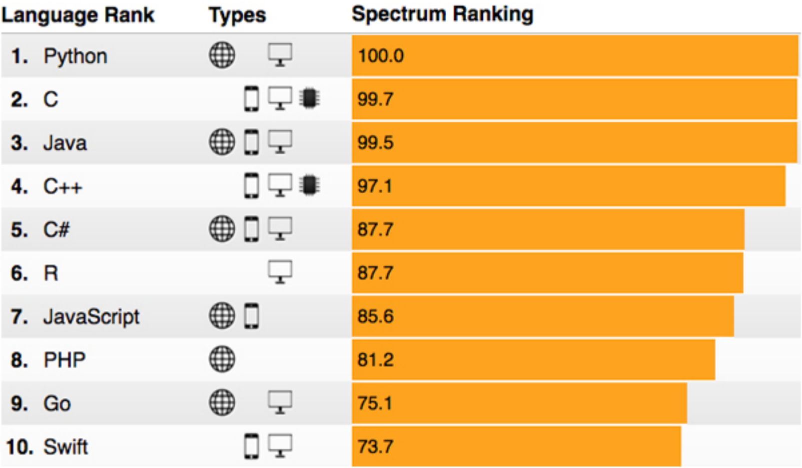
Outline

- Overview
- Foundation
- Data Structure
- Control Structure
- Function
- Statistics
- Data Visualization
- Text Mining Application
- Machine Learning

Overview

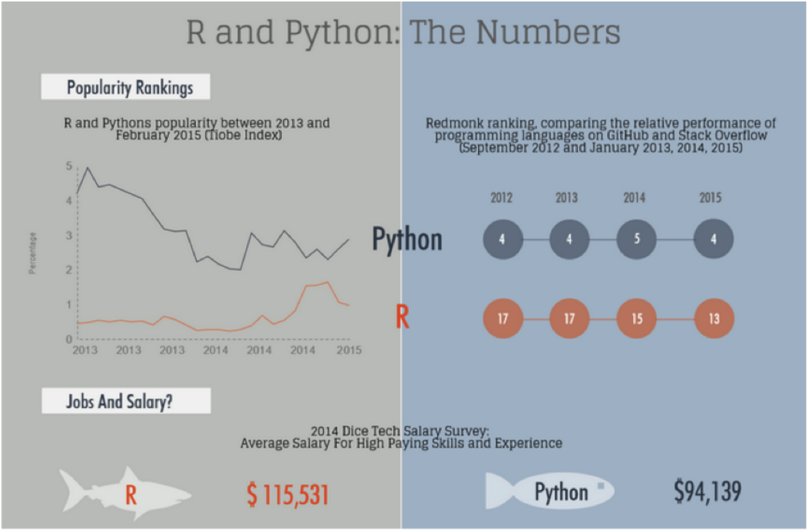
R Programming

The 2017 Top Programming Languages



Source: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

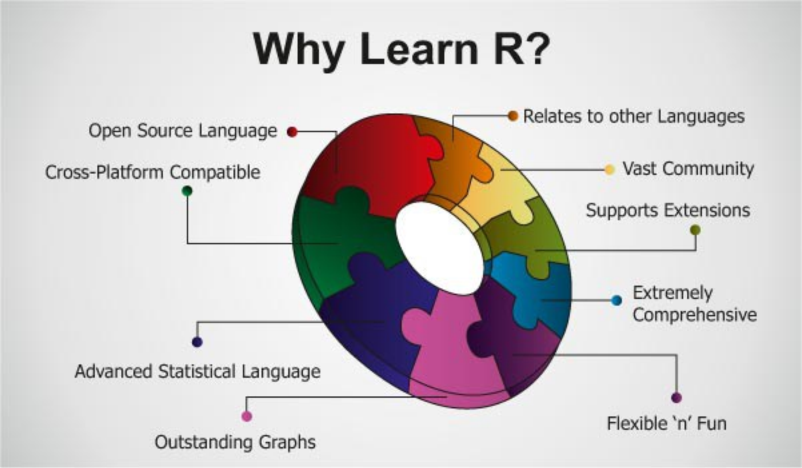
R vs. Python



source: https://blog.dominodatalab.com/comparing-python-and-r-for-data-science/

Why Use R?

- It is defacto standard among professional statisticians.
- It is available for the Windows, Mac, and Linux operating systems.
- R is a general purpose programming language, so you can use it to automate analyses and create new functions that extend the existing language features.
- Because R is open source software, it's easy to get help from the user community. Also, a lot of new functions are contributed by users, many of whom are prominent statisticians.



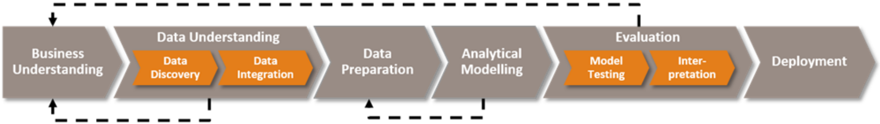
Why Learn R?
Open Source Language
Relates to other Languages
Cross-Platform Compatible
Vast Community
Supports Extensions
Extremely Comprehensive
Advanced Statistical Language
Outstanding Graphs
Flexible 'n' Fun

Skills of Data Science

- Data wrangling - 80% of the work in data science is data manipulation.
- Data visualization - *ggplot2* is one of the best data visualization tool.
- Machine learning - when you're ready to start using (and learning) machine learning, R has some of the best tools and resources.

"Spending 100 hours on R will yield vastly better than spending 10 hours on 10 different tools."

The Data Science Process



Data Understanding

Package name	*Description*
1. gridExtra	Grid plotting functions (very useful to plot grids of plots or tables)
2. corrplot	Nice plots of correlation matrices (see screenshot in Fig. 1)
3. ggplot2	Advanced plotting library (exceeds any other library in terms of customizing figures)
4. MASS	A wide range of statistical functions
5. matlab	Use real Matlab code in R (useful for Matlab to R transitioners)
6. iterator	Very useful to read files line by line that are larger than the RAM of my machine
7. dplyr	All kinds of data manipulation

Data Preparation

Package name	Description
8. compiler	Compile functions for faster execution (increase in speed by up to a factor of two depending on use case)
9. foreach	Parallelization of loops, in my opinion inferior to the parSapply function in the doParallel package
10. doParallel	Improved parallel computing, can speed up things up to a factor of ten depending on the use case (generally, R does not use more than one processor core)

Analytical Modelling

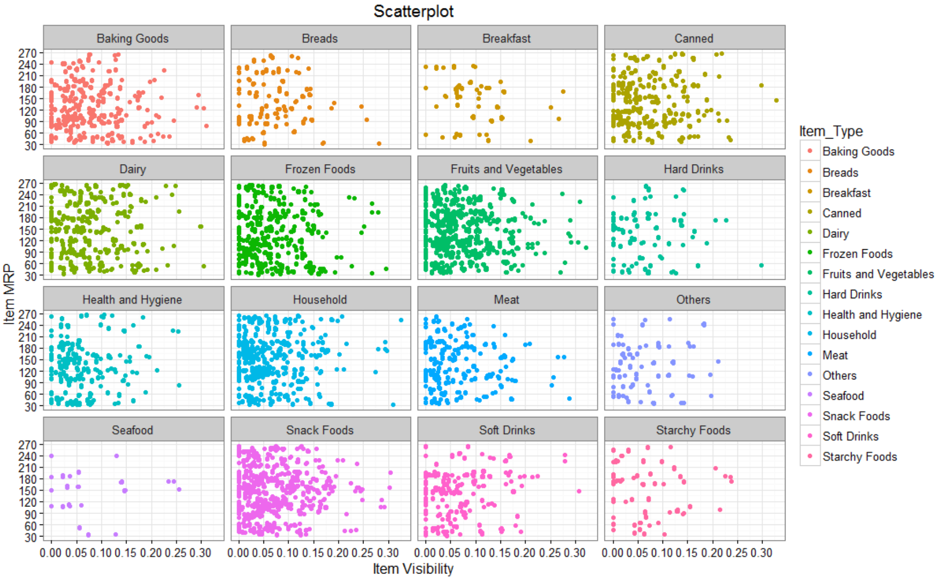
Package name	Description
11. caret	Large-scale model hyperparameter grid search. It is especially useful to combine different models (supervised and unsupervised) using CaretEnsemble!
12. metrics	Get fitting metrics (Caret provides some, but metrics is a lot stronger!)
13. formula	Generate formula objects from code (e.g., using the *paste* function) for use in fitting functions (automatic generation of functions from automatic feature generation)
14. e1071	Original implementation of a SVM in R. Also includes useful things for data analysis, such as fast Fourier transforms, clustering, naïve Bayes, some time series functionality etc.
15. qdap	Sentiment analysis for text mining
16. sentimentr	Sentiment analysis for text mining
17. tidytext	Context (topic) mining for text mining

Evaluation

Package name	Description
18. casher	Cashes results to avoid lengthy computations (particularly useful with large datasets – see the iterator package above)
19. fmsb	Plots radar charts (and business users love radar charts!)
20. wordcloud	Makes nice word clouds (again: looks nice)
21. RColorBrewer	Expands the colour range and automatically generates colour sequences for plots
22. Rserve	Enable access to R functionality from other programs, e.g., Tableau. Very helpful, since Tableau etc. are not able to perform real data analysis, but only visualization.
23. shiny	Produce browser GUIs for code to allow others to use it without needing to understand it (e.g., reporting). See Fig. 2!
24. rmarkdown	Package that produces readable text from within R (is extended in knitr)
25. knitr	Package that compiles "literate code" (a mix of code and human-readable text) to share via html, pdf or doc (see Fig. 1 above). Possible interaction via GUI by integration with package shiny.

Image Processing with R

Data Visualization with R



About R

- R is an open source statistical programming language and environment for statistical computing and graphics
- R supports user defined functions, and is capable of run-time calls to C, C++, FORTRAN, Java
- Available for Windows, Mac or Linux
- Developed by **R**oss Ihaka and **R**obert Gentlemen, University of Auckland, in 1995.
- Capability of R can be extended by packages ($>$1300)
- R feels and looks are the same regardless of the underlying operating system (for the most part)

Concepts of R

Rather than setting up a complete analysis at once, the process is highly interactive. You run a command, take the results and process it through another command, take those results and process it through another command. The cycle may include transforming the data, and looping back through the whole process again. You stop when you feel that you have fully analyzed the data.

How to Download?

- Google it using R or CRAN (Comprehensive R Achive Network) - http://www.r-project.org
- R Studio - https://www.rstudio.com/products/rstudio/download/
- R Commander - http://www.rcommander.com

R Overview

- You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file
- There is a wide variety of data types, including vectors (numerical, character, logical), matrices, dataframes, and lists
- To quit R, use **q()**
- Most functionality is provided through **built-in** and **user-created** functions and all data objects are kept in memory during an interactive session
- Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed

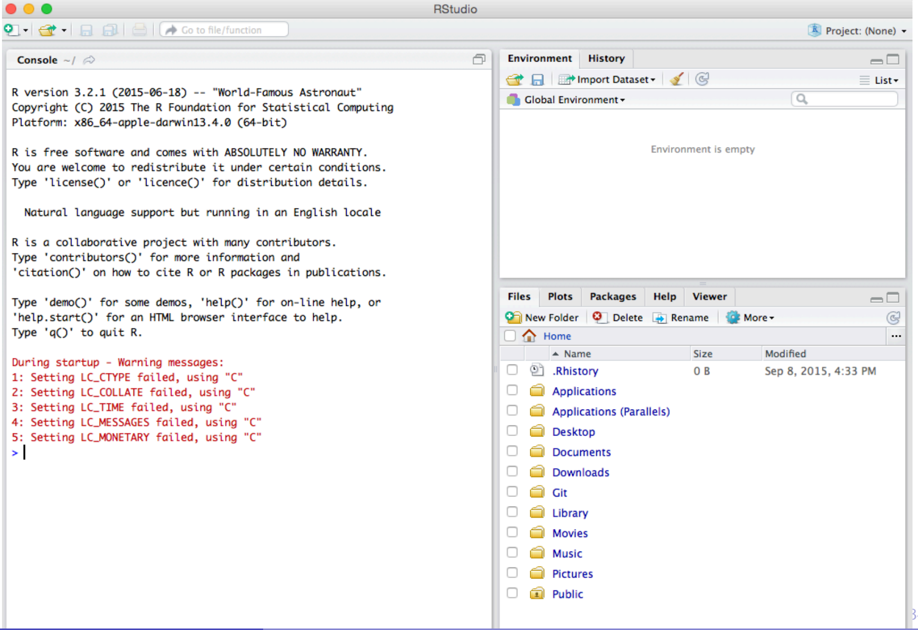
R Overview (cont.)

- A key skill to using **R** effectively is learning how to use the **built-in help system**. Other sections describe the working environment, inputting programs and outputting results, installing new functionality through packages and etc
- A fundamental design feature of **R** is that the output from most functions can be used as input to other functions

R Interface

- Start the R system, the main window (RGui) with a sub window (R Console) will appear
- In the 'Console' window the cursor is waiting for you to type in some R commands

R Session


RStudio
Go to file/function
Project: (None)
Console ~/
R version 3.2.1 (2015-06-18) -- "World-Famous Astronaut"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
During startup - Warning messages:
1: Setting LC_CTYPE failed, using "C"
2: Setting LC_COLLATE failed, using "C"
3: Setting LC_TIME failed, using "C"
4: Setting LC_MESSAGES failed, using "C"
5: Setting LC_MONETARY failed, using "C"
>
Environment
History
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
Home
Name
Size
Modified
.Rhistory
0 B
Sep 8, 2015, 4:33 PM
Applications
Applications (Parallels)
Desktop
Documents
Downloads
Git
Library
Movies
Music
Pictures
Public

R Introduction

- Results of calculations can be stored in objects using the assignment operators:
  - An arrow ($<-$) formed by a smaller than character and a hyphen without a space!
  - The equal character "$=$"
- These objects can then be used in other calculations. There are some restrictions when giving an object a name
  - Object names **cannot** contain '**strange**' symbols like !, +, -, #
  - A dot (.) and an underscore (_) are allowed, also a name starting with a dot
  - Object names can contain a number but **cannot** start with a number
  - R is **case sensitive**, X and x are two different objects, as well as temp and temP.

Examples

```
> #An example
> x <- c(1:10)
> x[(x>8) | (x<5)]
[1]  1  2  3  4  9 10
> x
[1]  1  2  3  4  5  6  7  8  9 10
>
```

R Introduction (cont.)

- To list the objects that you have in your current R session use the function **ls** or the function objects
  $>$ `ls()`
  [1] "x" "y"

- Most functions in R accept certain arguments. For example, one of the arguments of the function ls is pattern. To list all objects starting with the letter x:

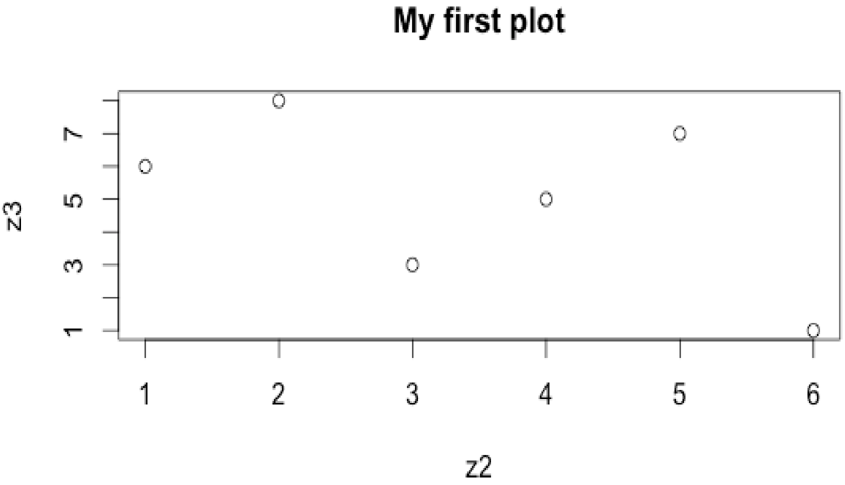
```
 >x2=9
>2 = 10
>ls(pattern="x")
[1] "x" "x2"
```

R Introduction (cont.)

- If you assign a value to an object that already exists then the contents of the object will be **overwritten** with the new value (without a warning!). Use the function rm to remove one or more objects from your session
  $>$ rm(x,x2)
- Lets create two small vectors with data and a scatterplot

```
z2 <- c(1,2,3,4,5,6)
z3  <-  c(6,8,3,5,7,1)
plot(z2,z3)
title("My first plot")
```

My First Plot



R Warning

R is a case sensitive language

FOO, Foo, and foo are three different objects

R Workspace

- Objects that you create during an R session are hold in memory, the collection of objects that you currently have is called the **workspace**
- The workspace is not saved on disk unless you tell R to do so
- Your objects are lost when you close R and not save the objects, or worse when R or your system crashes on you during a session
- If you have saved a workspace image and you start R the next time, it will restore the workspace. So all your previously saved objects are available again
- Commands are entered interactively at the R user prompt. **Up** and **down arrow keys** scroll through your command history

Foundation

R Installation Setup

Download & Install

- https://cran.rstudio.com/
- http://www.rstudio.com

Documentation

- https://cran.rstudio.com/doc/manuals/r-release/R-admin.pdf
- https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf

Arithmetic with R

Addition

```
>1+2
3
```

Subtraction

```
>5-3
2
```

Division

```
>1/2
0.5
```

Exponents

```
>2^3
8
>2**3
8
```

Modulo Exponents

```
>5 %% 2
1
```

Order of Operations Exponents

```
>(100*2) + (50 / 2)
225
>(2+2) * 3
12
```

Getting Help with R

Aside from **Google** search or visiting **StackOverflow**, there are some built-in ways to get help with R!

Most R functions have online documentation.

- help(topic) documentation on topic
- help.search("topic") search the help system
- apropos("topic") the names of all objects in the search list matching the regular expression "topic"
- help.start() start the HTML version of help
- str(a) display the internal structure of an R object
- summary(a) gives a "summary" of a, usually a statistical summary but it is generic meaning it has different operations for different classes of a

Comments

Comments are just everything that follows #. From a # to the end of the line, the R parser just skips the text.

```
# This is a comment.
```

Getting Help with R (cont.)

- ls() show objects in the search path; specify pat="pat" to search on a pattern
- ls.str() str() for each variable in the search path
- dir() show files in the current directory
- methods(a) shows S3 methods of **a**
- methods(class=class(a)) lists all the methods to handle objects of class **a**

Exercises

```r
help(vector)
```

```r
# This will pop up a help window  (need to pass a character string)
help.search('numeric')
```

```r
# Can also use ?? for a search
??vector
```

```r
# Can also do a quick stats summary:
v <- c(1,2,3,4,5,6)
summary(v)
```

Print

We can use the **print()** function to print out variables or strings:

```
print("hello")
[1] "hello"
```

```
x <- 10
print(x)
[1] 10
```

```
print(mtcars)
```

Formatting

We can format strings and variables together for printing in a few different ways:

paste() The paste() function looks like this: **paste (..., sep = " ")**

Where ... are the things you want to paste and **sep** is the separator you want between the pasted items, by default it is a space. For example:

```
print(paste('hello','world'))
[1] "hello world"
```

```
print(paste('hello','world',sep='-|-'))
[1] "hello-|-world"
```

paste0()

paste0(..., collapse) is equivalent to paste(..., sep = "", collapse), slightly more efficiently.

```
paste0('hello','world')
'helloworld'
```

sprintf

srpintf() is a wrapper for the C function sprintf, that returns a character vector containing a formatted combination of text and variable values. Meaning you can use % codes to place in variables by specifying all of them at the end. This is best shown through example:

```
sprintf("%s is %f feet tall\n", "Sven", 7.1)
'Sven is 7.100000 feet tall '
```

Variables

Rules for writing Identifiers in R

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_)
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit
3. Reserved words in R cannot be used as identifiers

Valid identifiers in R

total, Sum, .fine.with.dot, this_is_acceptable, Number5

Invalid identifiers in R

tot@l, 5um, _fine, TRUE, .0ne

Variables (cont.)

You can use the $<-$ character to assign a variable, note how it kind of looks like an arrow pointing from the object to the variable name.

```r
# Use hashtags for comments
variable.name <- 100
```

```r
# Let's see the variable
variable.name
100
```

Working with variables

We can use variables together and work with them, for example:

```
bank.account <- 100
deposit <- 10
bank.account <- bank.account + deposit
bank.account
110
```

You can assign with arrows in both directions, so you could also write the following:

```
2 -> x
```

An assignment won't print anything if you write it into the R terminal, but you can get R to print it just by putting the assignment in parentheses.

```
(y <- "visible")
[1] "visible"
```

R actually allows for five assignment operators:

```
#leftward assignment
x <- 3
x = 3
x <<- 3

#rightward assignment
3 -> x
3 ->> x
```

Note R is a case sensitive programming language.

```
x <- 1
y <- 3
z <- 4
x*y*z
12
x*Y*z
## Error in eval(expr, envir, enclos): object 'Y' not found
```

Built-in Constants

Some of the built-in constants defined in R along with their values.

```
> LETTERS
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
> pi
[1] 3.141593
> month.name
[1] "January"   "February"  "March"     "April"     "May"       "June"
[7] "July"      "August"    "September" "October"   "November"  "December"
> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

R Data Types

Numeric - Decimal (floating point values) are part of the numeric class in R

```
n <- 2.2
```

Integer - Natural (whole) numbers are known as integers and are also part of the numeric class

```
i <- 5
```

When you write numbers like 4 and 3, they are interpreted as floating-point numbers. To explicitly get an integer, you must write 4L and 3L.

```
>class(4)
"numeric"
>class(4L)
"integer"
```

Logical - Boolean values (True and False) are part of the logical class. In R these are written in All Caps.

```
t <- TRUE
f <- FALSE
```

Characters - Text/string values are known as characters in R. You use quotation marks to create a text character string:

```
char <- "Hello World!"
char
'Hello World!'
```

```
# Also single quotes
c <- 'Single Quote Char'
c
'Single Quote Char'
```

Checking Data Type Classes

You can use the class() function to check the data type of a variable:

```
>class(t)
'logical'
>class(f)
'logical'
>class(char)
'character'
>class(c)
'character'
>class(n)
'numeric'
>class(i)
'numeric'
```

Vector Basics

Vectors are one of the key data structures in R which we will be using. A vector is a **1 dimensional array** that can hold character, numeric, or logical data elements.

We can create a vector by using the combine function **c()**. To use the function, we pass in the elements we want in the array, with each individual element separated by a comma.

```
# Using c() to create a vector of numeric elements
>nvec <- c(1,2,3,4,5)
>class(nvec)
'numeric'
```

```
# Vector of characters
>cvec <- c('U','S','A')
>class(cvec)
'character'
```

```
>lvec <- c(TRUE,FALSE)
>lvec
TRUE FALSE
>class(lvec)
'logical'
```

Note we **CANNOT mix** data types of the elements in an array, R will convert the other elements in the array to force everything to be of the same data type.

Here's a quick example of what happens with arrays given **different data types**:

```
>v <- c(FALSE,2)
>v
0 2
>class(v)
'numeric'
```

```
>v <- c('A',1)
>v
'A' '1'
>class(v)
'character'
```

Vector Names

We can use the **names()** function to assign names to each element in our vector. For example, imagine the folowing vector of a week of temperatures:

```
>temps <- c(72,71,68,73,69,75,71)
>temps
72 71 68 73 69 75 71
```

We know we have 7 temperatures for 7 weekdays, but which temperature corresponds to which weekday? Does it start on Monday, Sunday, or another day of the week? This is where **names()** can be assigned in the following manner:

```
>names(temps) <- c('Mon','Tue','Wed','Thu','Fri','Sat','Sun')
```

Now what happens when we display the named vector:

```
>temps
Mon 72
Tue 71
Wed 68
Thu 73
Fri 69
Sat 75
Sun 71
```

We also don't have to rewrite the names vector over and over again, we can use simple use a variable name as a **names()** assignment, for example:

```
>days <- c('Mon','Tue','Wed','Thu','Fri','Sat','Sun')
>temps2 <- c(1,2,3,4,5,6,7)
>names(temps2) <- days
>temp2
Mon 1
Tue 2
Wed 3
Thu 4
Fri 5
Sat 6
Sun 7
```

Vector Indexing and Slicing

You can use bracket notation to index and access individual elements from a vector:

```
>v1 <- c(100,200,300)
>v2 <- c('a','b','c')
>v1
>v2
100 200 300
'a' 'b' 'c'`
```

Indexing works by using brackets and passing the index position of the element as a number. Keep in mind index starts at **1**.

```
# Grab second element
>v1[2]
200
>v2[2]
'b'
```

Multiple Indexing

We can grab multiple items from a vector by passing a vector of index positions inside the square brackets. For example:

```
>v1[c(1,2)]
100 200
>v2[c(2,3)]
'b' 'c'
>v2[c(1,3)]
'a' 'c'
```

Slicing

You can use a colon (:) to indicate a slice of a vector. The format is: **vector[start_index:stop_index]** and you will get that "slice" of the vector returned to you. For example:

```
>v <- c(1,2,3,4,5,6,7,8,9,10)
>v[2:4]
2 3 4
>v[7:10]
7 8 9 10
```

Notice how the element at both the starting index and the stopping index are included.

Indexing with Names

We've previously seen how we can assign names to the elements in a vector, for example:

```
>v <- c(1,2,3,4)
>names(v) <- c('a','b','c','d')
```

We can use those names along with the indexing brackets tograb individual elements from the array.

```
>v['a']
a: 1
```

Or pass in a vector of names we want to grab:

```
# Notice how we can call out of order!
>v[c('a','c','b')]
a 1
c 3
b 2
```

Comparison Operators and Selection

We can use comparison operators to filter out elements from a vector. Sometimes this is referred to as boolean/logical masking, because you are creating a vector of logicals to filter out results you want. Let's see an example of this:

```
>v
a 1
b 2
c 3
d 4
```

```
>v[v>2]
c 3
d 4
```

Let's break this down to see how it works, we first get the vector v>2:

```
>v>2
a FALSE
b FALSE
C TRUE
D TRUE
```

Now we basically pass this vector of logicals through the brackets of the vector and only return true values at the matching index positions:

```
>v[v>2]
c 3
d 4
```

We could also assign names to these logical vectors and pass them as well, for example:

```
>filter <- v>2
>filter
a FALSE
b FALSE
c TRUE
d TRUE
```

```
>v[filter]
c 3
d 4
```

Comparison Operators

In R we can use comparison operators to compare variables and return logical values. Let's see some relatively self-explanatory examples:

Greater Than

```
5 > 6
FALSE
6 > 5
TRUE
```

We can also do element by element comparisons for two vectors:

```
v1 <- c(1,2,3)
v2 <- c(10,20,30)
v1 < v2
TRUE TRUE TRUE
```

Greater Than or Equal to

```
6 >= 6
TRUE
6 >= 5
TRUE
6 >= 7
FALSE
```

Less Than and Less than or Equal To

```
3 < 2
FALSE
2 <= 2
TRUE
```

Be very careful with comparison operators and negative numbers! Use spacing to keep things clear. An example of a dangerous situation:

```
var <- 1
var
1
```

```
# Comparing var less than negative 2
var <  -2
FALSE
```

Not Equal

```
5 != 2
TRUE
5 != 5
FALSE
```

Equal

```
5 == 5
TRUE
2 == 3
FALSE
```

Vector Comparisons

We can apply a comparison of a single number to an entire vector, for example:

```
v <- c(1,2,3,4,5)
v < 2
TRUE FALSE FALSE FALSE FALSE
```

```
v == 3
FALSE FALSE TRUE FALSE FALSE
```

Working with Vectors

We can perform basics arithmetic with vectors and operations will occur on an element by element basis, for example:

```
v1 <- c(1,2,3)
v2 <- c(5,6,7)
```

Adding Vectors

```
>v1+v2
6 8 10
```

Subtracting Vectors

```
>v1-v1
0 0 0
>v1-v2
-4 -4 -4
```

Multiplying Vectors

```
>v1*v2
5 12 21
```

Dividing Vectors

```
>v1/v2
0.2 0.333333333333333 0.428571428571429
```

Functions with Vectors

Some useful functions that we can use with vectors. A function will be in the form: **name_of_function(input)**

For example, if you want to sum all the elements in a numeric vector, you can use the sum() function. For example:

```
>v1
1 2 3
>sum(v1)
6
```

We can also check for things like the standard deviation, variance, maximum element, minimum element, product of elements:

```
v <- c(12,45,100,2)
# Standard Deviation
>sd(v)
44.1691823182937
```

```
#Variance
>var(v)
1950.91666666667
```

```
#Maximum Element
>max(v)
100
```

```
#Minimum Element
>min(v)
2
```

```
#Product of elements
>prod(v1)
6
>prod(v2)
210
```

Check out this

https://cran.r-project.org/doc/contrib/Short-refcard.pdf

Data Structure

Matrix

A matrix will allow us to have a 2-dimensional data structure which contains elements consisting of the same data type.

Tip: *A quick tip for quickly creating sequential numeric vectors, you can use the colon notation from slicing to create sequential vectors:*

```
>1:10
1 2 3 4 5 6 7 8 9 10
> v <- 1:10
1 2 3 4 5 6 7 8 9 10
```

To create a matrix, we use the **matrix()** function. We can pass in a vector into the matrix:

```
matrix(v)
^^I^^I[,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
[7,]    7
[8,]    8
[9,]    9
[10,]   10
```

Here we have a two-dimensional matrix which is 10 rows by 1 column. Now what if we want to specify the number of rows?

We can pass the parameter/argument into the matrix function called **nrow** which stands for number of rows:

```
matrix(v,nrow=2)
^^I^^I^^I[,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

The **byrow** argument allows you to specify whether or not you want to fill out the matrix by rows or by columns. For example:

```
matrix(1:12,byrow = FALSE,nrow=4)
    ^^I^^I[,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Creating Matrices from Vectors

We can combine vectors to later input them into a matrix. For example imagine the following vectors below of stock prices:

```
# not real prices
goog <- c(450,451,452,445,468)
msft <- c(230,231,232,236,228)
stocks <- c(goog,msft)
stock.matrix <- matrix(stocks,byrow=TRUE,nrow=2)
stock.matrix

     ^^I[,1] [,2] [,3] [,4] [,5]
[1,]  450  451  452  445  468
[2,]  230  231  232  236  228
```

Naming Matrices

It would be nice to name the rows and columns for reference. We can do this similarly to the **names()** function for vectors, but in this case we define **colnames()** and **rownames()**. So let's name our stock matrix:

```
days <- c('Mon','Tue','Wed','Thu','Fri')
st.names <- c('GOOG','MSFT')
colnames(stock.matrix) <- days
rownames(stock.matrix) <- st.names
stock.matrix

     ^^IMon Tue Wed Thu Fri
GOOG 450 451 452 445 468
MSFT 230 231 232 236 228
```

Matrix Arithmetic

We can perform element by element mathematical operations on a matrix with a scalar (single number) just like we could with vectors. Let's see some quick examples:

```
> mat <- matrix(1:6,byrow=TRUE,nrow=2)
> mat

^^I^^I^^I[,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
# Multiplication
2*mat

    ^^I [,1] [,2] [,3]
[1,]    2    4    6
[2,]    8   10   12
```

```
1/mat

     ^^I[,1] [,2]      [,3]
[1,] 1.00  0.5 0.3333333
[2,] 0.25  0.2 0.1666667
```

```
mat / 2

     ^^I[,1] [,2] [,3]
[1,]  0.5  1.0  1.5
[2,]  2.0  2.5  3.0
```

```
mat ^ 2

    ^^I[,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

Comparison operators with matrices

We can similarly perform comparison operations across an entire matrix to return a matrix of logicals:

```
mat > 3
      ^^I[,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE  TRUE  TRUE
```

Matrix Arithmetic with multiple matrices

We can use multiple matrices with arithmetic as well, for example:

```
mat+mat

     ^^I[,1] [,2] [,3]
[1,]    2    4    6
[2,]    8   10   12
```

```
mat/mat

     ^^I[,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

```
mat ^ mat

^^I^^I^^I[,1] [,2]  [,3]
[1,]    1    4    27
[2,]  256 3125 46656
```

```
mat * mat

     ^^I[,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

Matrix multiplication

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Trick:

$P_1 = A \cdot (F-H)$

$P_2 = (A+B) \cdot H$

$P_3 = (C+D) \cdot E$

$P_4 = D \cdot (G-E)$

$P_5 = (A+D) \cdot (E+H)$

$P_6 = (B-D) \cdot (G+H)$

$P_7 = (A-C) \cdot (E+F)$

$AE+BG = P_5 + P_4 - P_2 + P_6$

$AF+BH = P_1 + P_2$

$CE+DG = P_3 + P_4$

$CF+DH = P_5 + P_1 - P_3 - P_7$

```
mat2 <- matrix(1:9, nrow=3)
mat2 % * % mat2

     ^^I[,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

Matrix Operations

Run the following code to create the stock.matrix from earlier

```
# Prices
goog <- c(450,451,452,445,468)
msft <- c(230,231,232,236,228)

# Put vectors into matrix
stocks <- c(goog,msft)
stock.matrix <- matrix(stocks,byrow=TRUE,nrow=2)

# Name matrix
days <- c('Mon','Tue','Wed','Thu','Fri')
st.names <- c('GOOG','MSFT')
colnames(stock.matrix) <- days
rownames(stock.matrix) <- st.names
```

```
# Display
stock.matrix

^^I^^IMon Tue Wed Thu Fri
GOOG 450 451 452 445 468
MSFT 230 231 232 236 228
```

We can perform functions across the columns and rows, such as **colSum()** and **rowSums()**:

```
colSums(stock.matrix)

Mon Tue Wed Thu Fri
680 682 684 681 696

rowSums(stock.matrix)

GOOG MSFT
2266 1157
```

Binding columns and rows

we can use the cbind() to bind a new column, and **rbind()** to bind a new row. For example, let's bind a new row with Facebook stock:

```
FB <- c(111,112,113,120,145)
tech.stocks <- rbind(stock.matrix,FB)
tech.stocks

     ^^IMon Tue Wed Thu Fri
GOOG 450 451 452 445 468
MSFT 230 231 232 236 228
FB   111 112 113 120 145
```

Now let's add an average column to the matrix:

```
avg <- rowMeans(tech.stocks)
avg

 GOOG  MSFT    FB
453.2 231.4 120.2
```

```
tech.stocks <- cbind(tech.stocks,avg)
tech.stocks

     ^^IMon Tue Wed Thu Fri   avg
GOOG 450 451 452 445 468 453.2
MSFT 230 231 232 236 228 231.4
FB   111 112 113 120 145 120.2
```

Matrix Selection and Indexing

Just like with vectors, we use the square bracket notation to select elements from a matrix. Since we have two dimensions to work with, we'll use a comma to separate our indexing for each dimension.

So the syntax is then: **example.matrix[rows,columns]**

Where the index notation (e.g. 1:5) is put in place of the rows or columns . If either rows or columns is left blank, then we are selecting all the rows and columns.

```
mat <- matrix(1:12,byrow=TRUE,nrow=3)
mat

     ^^I[,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
# Grab first row
mat[1,]

[1] 1 2 3 4

#Grab first column
mat[,1]

[1] 1 5 9

# Grab first 3 rows
mat[1:2,]

^^I     [,1] [,2] [,3] [,4]
[1,]     1    2    3    4
[2,]     5    6    7    8
```

```
mat[1:2,1:2]

    ^^I[,1] [,2]
[1,]     1    2
[2,]     5    6
```

Factor and Categorical Matrices

Imagine we have the following vectors representing data from an animal sanctuary for dogs ('d') and cats ('c') where they each have a corresponding id number in another vector.

```
animal <- c('d','c','d','c','c')
id <- c(1,2,3,4,5)
```

We want to convert the animal vector into information that an algorithm or equation can understand more easily. Meaning we want to begin to check how many categories (factor levels) are in our character vector.

```
factor.ani <- factor(animal)
# Will show levels as well on RStudio or R Console
factor.ani
[1] d c d c c
```

If you wanted to assign an order while using the **factor()** function, you can pass in the arguments **ordered=True** and the pass in the **levels=** and pass in a vector in the order you want the levels to be in. So for example:

cold < med < hot

```
temps <- c('cold','med','cold','med','hot','hot','cold')
fact.temp <- factor(temps,ordered=TRUE,levels=c('cold','med','hot'))
fact.temp
```

This information is useful when used along with the summary() function which is an amazingly convenient function for quickly getting information from a matrix or vector. For example:

```
summary(fact.temp)

cold  med  hot
3    2    2
```

Dataframe Basics

Matrix inputs were limited because all the data inside of the matrix had to be of the same data type (numerics, logicals, etc). With Dataframes we will be able to organize and mix data types to create a very powerful data structure tool.

To get a list of all available built-in dataframes, use **data()**

```
data()
```

We can notice some dataframe are really big, we can use the **head()** and **tail()** functions to view the first and last 6 rows respectively.

```
states <- state.x77
head(states)
```

```
            ^^I^^I^^IPopulation Income Illiteracy Life Exp Murder HS Grad Frost   Area
Alabama          3615   3624        2.1     69.05   15.1    41.3    20  50708
Alaska            365   6315        1.5     69.31   11.3    66.7   152 566432
Arizona          2212   4530        1.8     70.55    7.8    58.1    15 113417
Arkansas         2110   3378        1.9     70.66   10.1    39.9    65  51945
California      21198   5114        1.1     71.71   10.3    62.6    20 156361
Colorado         2541   4884        0.7     72.06    6.8    63.9   166 103766
```

DataFrames - Overview of information

We can use the **str()** to get the structure of a dataframe, which gives information on the structure of the dataframe and the data it contains, such as variable names and data types. We can use **summary()** to get a quick statistical summary of all the columns of a DataFrame.

```
# Statistical summary of data
summary(states)

# Structure of Data
str(states)
```

Creating Data frames

We can create data frames using the **data.frame()** function and pass vectors as arguments, which will then convert the vectors into columns of the data frame. Let's see a simple example:

```
# Some made up weather data
days <- c('mon','tue','wed','thu','fri')
temp <- c(22.2,21,23,24.3,25)
rain <- c(TRUE, TRUE, FALSE, FALSE, TRUE)

# Pass in the vectors:
df <- data.frame(days,temp,rain)
df

  days temp  rain
1  mon 22.2  TRUE
2  tue 21.0  TRUE
3  wed 23.0 FALSE
4  thu 24.3 FALSE
5  fri 25.0  TRUE
```

Overview of Data Frame Operations

```
# Creating Data Frames
empty <- data.frame() # empty data frame
c1 <- 1:10 # vector of integers
c2 <- letters[1:10] # vector of strings
df <- data.frame(col.name.1=c1,col.name.2=c2)
df

   col.name.1 col.name.2
1           1          a
2           2          b
3           3          c
4           4          d
5           5          e
6           6          f
7           7          g
8           8          h
9           9          i
10         10          j
```

```
#Importing and Exporting Data
d2 <- read.csv('some.file.name.csv')

# For Excel Files
# Load the readxl package
library(readxl)
# Call info from the sheets using read.excel
df <- read_excel('Sample-Sales-Data.xlsx',sheet='Sheet1')

# Output to csv
write.csv(df, file='some.file.csv')
```

```r
#Getting Information about Data Frame
nrow(df)
10
ncol(df)
2
```

```r
# Column Names
colnames(df)
"col.name.1" "col.name.2"

# Row names (may just return index)
rownames(df)
"1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```r
#Referencing Cells
vec <- df[[5, 2]] # get cell by [[row,col]] num
newdf <- df[1:5, 1:2] # get multiplt cells in new df
df[[2, 'col.name.1']] <- 99999 # reassign a single cell
```

```r
#Referencing Rows
rowdf <- df[1, ]

# to get a row as a vector, use following
vrow <- as.numeric(as.vector(df[1,]))
```

```r
#Referencing Columns
cars <- mtcars
colv1 <- cars$mpg
colv2 <- cars[, 'mpg']
colv3<- cars[, 1]
colv4 <- cars[['mpg']]
```

```r
#Adding Rows
df2 <- data.frame(col.name.1=2000,col.name.2='new' )
df2

# use rbind to bind a new row!
dfnew <- rbind(df,df2)
```

```r
df$newcol <- rep(NA, nrow(df)) # NA column
df

df[, 'copy.of.col2'] <- df$col.name.2 # copy a col
df

# Can also use equations!
df[['col1.times.2']] <- df$col.name.1 * 2
df
```

```r
# Rename second column
colnames(df)[2] <- 'SECOND COLUMN NEW NAME'
df

# Rename all at once with a vector
colnames(df) <- c('col.name.1', 'col.name.2', 'newcol', 'copy.of.col2' ,'col1.times.2')
df
```

```r
#Selecting Multiple Rows
first.ten.rows <- df[1:10, ] # Same as head(df, 10)
first.ten.rows
```

```r
everything.but.row.two <- df[-2, ]
everything.but.row.two
```

```r
# Conditional Selection
sub1 <- df[ (df$col.name.1 > 8 & df$col1.times.2 > 10), ]
sub1

sub2 <- subset(df, col.name.1 > 8 & col1.times.2 > 10)
sub2
```

```r
#Selecting Multiple Columns
df[, c(1, 2, 3)] #Grab cols 1 2 3

df[, c('col.name.1', 'col1.times.2')] # by name

df[, -1] # keep all but first column

df[, -c(1, 3)] # drop cols 1 and 3
```

Note: we use **[[ ]]** to select a single element by using integer or character indices.

Dealing with Missing Data

```r
any(is.na(df)) # detect anywhere in df

any(is.na(df$col.name.1)) # anywhere in col

# delete selected missing data rows
df <- df[!is.na(df$col), ]

# replace NAs with something else
df[is.na(df)] <- 0 # works on whole df

df$col[is.na(df$col)] <- 999 # For a selected column
```

Data Frame Selection and Indexing

```
^^I# Some made up weather data
^^Idays <- c('mon','tue','wed','thu','fri')
^^Itemp <- c(22.2,21,23,24.3,25)
^^Irain <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
^^I
^^I# Pass in the vectors:
^^Idf <- data.frame(days,temp,rain)
^^Idf
^^I
^^I^^Idays temp  rain
^^I1   mon 22.2  TRUE
^^I2   tue 21.0  TRUE
^^I3   wed 23.0 FALSE
^^I4   thu 24.3 FALSE
^^I5   fri 25.0  TRUE
^^I
```

We can use the same bracket notation we used for matrices:
df[rows,columns]

```
^^I# Everything from first row
^^Idf[1,]
^^I
^^I#Everything from first column
^^Idf[,1]
^^I
^^I# Grab Friday data
^^Idf[5,]
^^I
```

Selecting using column names

we can use column names to select data for the columns instead of having to remember numbers. So for example:

```
# All rain values
df[,'rain']

# First 5 rows for days and temps
df[1:5,c('days','temp')]
```

If you want all the values of a particular column you can use the dollar sign directly after the dataframe as follows: **df.name$column.name**

```
df$rain
df$days
```

You can also use bracket notation to return a data frame format of the same information:

```
df['rain']
df['days']
```

Filtering with a subset condition We can use the subset() function to grab a subset of values from our data frame based off some condition. So for example, imagin we wanted to grab the days where it rained (rain=True), we can use the subset() function as follows:

```
subset(df,subset=rain==TRUE)

^^I  days temp rain
  1  mon 22.2 TRUE
  2  tue 21.0 TRUE
  5  fri 25.0 TRUE
```

Odering a Data Fram

We can sort the **order** of our data frame by using the order function. You pass in the column you want to sort by into the **order()** function, then you use that vector to select from the dataframe. Let's see an example of sorting by the temperature:

```
sorted.temp <- order(df['temp'])
df[sorted.temp,]

^^I  days temp  rain
  2  tue 21.0  TRUE
  1  mon 22.2  TRUE
  3  wed 23.0 FALSE
  4  thu 24.3 FALSE
  5  fri 25.0  TRUE
```

We can pass a negative sign to do descending order.

```
desc.temp <- order(-df['temp'])
df[desc.temp,]

days temp  rain
5  fri 25.0  TRUE
4  thu 24.3 FALSE
3  wed 23.0 FALSE
1  mon 22.2  TRUE
2  tue 21.0  TRUE
```

We could have also used the other column selection methods we learned:

```
sort.temp <- order(df$temp)
df[sort.temp,]
```

R Lists Basics

Lists will allow us to store a variety of data structures under a single variable. This means we could store a vecor,matrix, data frame, etc. under a single list. For example:

```
# Create a vector
v <- c(1,2,3,4,5)

# Create a matrix
m <- matrix(1:10,nrow=2)

# Create a data frame
df <- women
```

Using list()

We can use the list() to combine all the data structures:

```
li <- list(v,m,df)
li

[[1]]
[1] 1 2 3 4 5

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

```
[[3]]
height  weight
1       58     115
2       59     117
3       60     120
4       61     123
5       62     126
6       63     129
7       64     132
8       65     135
9       66     139
10      67     142
11      68     146
12      69     150
13      70     154
14      71     159
15      72     164
```

The **list()** assigned numbers to each of the objects in the list, but we can also assign names in the following manner:

```
li <- list(sample_vec = v,sample_mat = m, sample_df = df)
# Ignore the "error in vapply", this won't occur in RStudio!
li

$sample_vec
[1] 1 2 3 4 5

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

```
$sample_df
^^I^^Iheight  weight
1        58     115
2        59     117
3        60     120
4        61     123
5        62     126
6        63     129
7        64     132
8        65     135
9        66     139
10       67     142
11       68     146
12       69     150
13       70     154
14       71     159
15       72     164
```

Selecting objects from a list

You can use bracket notation to show objects in a list, and double brackets to actually grab the objects form the list, for example:

```
# Single brackets
li[1] # By index

$sample_vec
[1] 1 2 3 4 5

# By name
li['sample_vec']

$sample_vec
[1] 1 2 3 4 5

# Notice the type!
class(li['sample_vec'])

[1] "list"
```

```
# Use double brackets to actually grab the items
li[['sample_vec']]

[1] 1 2 3 4 5

# Can also use $ notation
li$sample_vec

[1] 1 2 3 4 5
```

Combining lists

Lists can hold other lists! You can also combine lists using the combine function **c()**:

```
double_list <- c(li,li)
str(double_list)

List of 6
$ sample_vec: num [1:5] 1 2 3 4 5
$ sample_mat: int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
$ sample_df :'data.frame':^^I15 obs. of  2 variables:
..$ height: num [1:15] 58 59 60 61 62 63 64 65 66 67 ...
..$ weight: num [1:15] 115 117 120 123 126 129 132 135 139 142 ...
$ sample_vec: num [1:5] 1 2 3 4 5
$ sample_mat: int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
$ sample_df :'data.frame':^^I15 obs. of  2 variables:
..$ height: num [1:15] 58 59 60 61 62 63 64 65 66 67 ...
..$ weight: num [1:15] 115 117 120 123 126 129 132 135 139 142 ...
```

Control Structure

Logical Operators

Logical Operators will allow us to combine multiple comparison operators. The logical operators we will learn about are:

- AND - &
- OR - |
- NOT - !

```
# Imagine the variable x
x <- 10
```

Now we want to know if 10 is less than 20 AND greater than 5:

```
x < 20

TRUE
```

```
x > 5
TRUE

x < 20 & x > 5
TRUE
```

We can also add parenthesis for readability and to make sure the order of comparisons is what we expect:

```
(x < 20) & (x>5)
TRUE

(x < 20) & (x>5) & (x == 10)
TRUE
```

NOT!

You can think about NOT as reversing any logical value in front of it, basically asking, “Is this NOT true?” For example:

```
(10==1)
FALSE

!(10==1)
TRUE

# We can stack them (pretty uncommon, but possible)
!!(10==1)
FALSE
```

Use Case Example

```r
df <- mtcars
df[df['mpg'] >= 20,] # Notice the use of indexing with the comma
# subset(df,mpg>=20) # Could also use subset
```

Let's combine filters with logical operators! Let's grab rows with cars of at least 20mpg and over 100 hp.

```r
df[(df['mpg'] >= 20) & (df['hp'] > 100),]
```

Logical Operators with Vectors

We have two options when use logical operators, a comparison of the entire vectors element by element, or just a comparison of the first elements in the vectors, to make sure the output is a single Logical.

```
tf <- c(TRUE,FALSE)
tt <- c(TRUE,TRUE)
ft <- c(FALSE, TRUE)
tt & tf
[1]  TRUE FALSE

tt | tf
[1] TRUE TRUE
```

To compare **first elements** use && or ||

```
ft && tt
[1] FALSE

tt || tf
TRUE

tt || ft
TRUE

tt && tf
TRUE
```

if, else, else if Statements

Our first step in this learning journey for programming will be simple **if, else, and else if** statements.

Here is the syntax for an **if** statement in R:

```
if(condition){
#Execute some code
}
```

We say **if** some condition is **true** then execute the code inside of the curly brackets.

For example, let's say we have two variables, **hot** and **temp**. Imagine that **hot** starts off as FALSE and temp is some number in degrees. If the **temp** is greater than 80 than we want to assign **hot=TRUE**.

Let's see this in action

```
hot <- FALSE
temp <- 60
if (temp > 80){
^^Ihot <- TRUE
}
hot
[1] FALSE
```

```
# Reset temp
temp <- 100

if (temp > 80){
hot <- TRUE

}

hot
[1] TRUE
```

else if

What if we wanted more options to print out, rather than just two, the **if** and the **else**? This is where we can use the **else if** statement to add multiple condition checks, using **else** at the end to execute code if none of our conditions match up with and if or else if.

```
temp <- 30

if (temp > 80){
print("Hot outside!")
} else if(temp<80 & temp>50){
print('Nice outside!')
} else if(temp <50 & temp > 32){
print("Its cooler outside!")
} else{
print("Its really cold outside!")
}

[1] "Its really cold outside!"
```

```r
temp <- 75

if (temp > 80){
print("Hot outside!")
} else if(temp<80 & temp>50){
print('Nice outside!')
} else if(temp <50 & temp > 32){
print("Its cooler outside!")
} else{
print("Its really cold outside!")
}

[1] "Nice outside!"
```

Final Example

Let's see a final more elaborate example of **if,else, and else if** :

```
# Items sold that day
ham <- 10
cheese <- 10

# Report to HQ
report <- 'blank'
if(ham >= 10 & cheese >= 10){
^^Ireport <- "Strong sales of both items"
}else if(ham == 0 & cheese == 0){
^^Ireport <- "Nothing sold!"
}else{
^^Ireport <- 'We had some sales'
}
print(report)

[1] "Strong sales of both items"
```

for loops

A **for loop** allows us to iterate over an object (such as a vector) and we can then perform and execute blocks of codes for every loop we go through. The syntax for a for loop is:

```
for (temporary_variable in object){
# Execute some code at every loop
}
```

For loop over a vector

We can think of looping through a vector in two different ways, the first way would be to create a temporary variable with the use of the in keyword:

```
vec <- c(1,2,3,4,5)
for (temp_var in vec){
^^Iprint(temp_var)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

The other way would be to loop a numbered amount of times and then use indexing to continually grab from the vector:

```
for (i in 1:length(vec)){
print(vec[i])
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

For loop over a list We can do the same thing with a list:

```
li <- list(1,2,3,4,5)
for (temp_var in li){
^^Iprint(temp_var)
}
```

```
for (i in 1:length(li)){
^^Iprint(li[[i]]) # Remember to use double brackets!
}
```

For loop with a matrix

We can similarly loop through each individual element in a matrix:

```
mat <- matrix(1:10,nrow=5)
for (num in mat){
print(num)
}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Nested for loops

We can nest for loops inside one another,however be careful when doing this, as every additional for loop nested inside another may cause a significant amount of additional time for your code to finish executing. For example:

```
for (row in 1:nrow(mat)){
^^Ifor (col in 1:ncol(mat)){
^^I^^Iprint(paste('The element at row:',row,'and col:',col,'is',mat[row,col]))
^^I}
}
[1] "The element at row: 1 and col: 1 is 1"
[1] "The element at row: 1 and col: 2 is 6"
[1] "The element at row: 2 and col: 1 is 2"
[1] "The element at row: 2 and col: 2 is 7"
[1] "The element at row: 3 and col: 1 is 3"
[1] "The element at row: 3 and col: 2 is 8"
......
[1] "The element at row: 5 and col: 2 is 10"
```

while loops

while loops are a while to have your program continuously run some block of code until a condition is met (made TRUE). The syntax is:

```
while (condition){
# Code executed here
# while condition is true
}
```

```
^^Ix <- 0
^^I
^^Iwhile(x < 10){
^^I
^^Icat('x is currently: ',x)
^^Iprint(' x is still less than 10, adding 1 to x')
^^I
^^I# add one to x
^^Ix <- x+1
^^I}
^^Ix is currently:  0[1] " x is still less than 10, adding 1 to x"
^^Ix is currently:  1[1] " x is still less than 10, adding 1 to x"
^^Ix is currently:  2[1] " x is still less than 10, adding 1 to x"
^^I....
^^Ix is currently:  9[1] " x is still less than 10, adding 1 to x"
^^I
```

```r
x <- 0
while(x < 10){
^^Icat('x is currently: ',x)
^^Iprint(' x is still less than 10, adding 1 to x')
^^I# add one to x
^^Ix <- x+1
^^Iif(x==10){
^^I^^Iprint("x is equal to 10! Terminating loop")
^^I}
}
```

break

You can use break to break out of a loop.

```
x <- 0
while(x < 5){
^^Icat('x is :',x,sep="")
^^Iprint(' x is still less than 5, adding 1 to x')
# add one to x
^^Ix <- x+1
^^Iif(x==5){
^^I^^Iprint("x is equal to 5!")
^^I^^Iprint("I will also print, woohoo!")
^^I}
}
x is :0[1] " x is still less than 5, adding 1 to x"
x is :1[1] " x is still less than 5, adding 1 to x"
x is :2[1] " x is still less than 5, adding 1 to x"
x is :3[1] " x is still less than 5, adding 1 to x"
x is :4[1] " x is still less than 5, adding 1 to x"
[1] "x is equal to 5!"
[1] "I will also print, woohoo!"
```

```
x <- 0
^^Iwhile(x < 5){
^^I^^Icat('x is :',x,sep="")
^^I^^Iprint(' x is less than 5, adding 1 to x')
^^I^^I# add one to x
^^I^^Ix <- x+1
^^I^^I^^Iif(x==5){
^^I^^I^^Iprint("x is equal to 5!")
^^I^^I^^Ibreak
^^I^^I^^Iprint("I will also print, woohoo!")
^^I^^I}
^^I}
x is :0[1] " x is less than 5, adding 1 to x"
x is :1[1] " x is less than 5, adding 1 to x"
x is :2[1] " x is less than 5, adding 1 to x"
x is :3[1] " x is less than 5, adding 1 to x"
x is :4[1] " x is less than 5, adding 1 to x"
[1] "x is equal to 5!"
^^I
```

Function

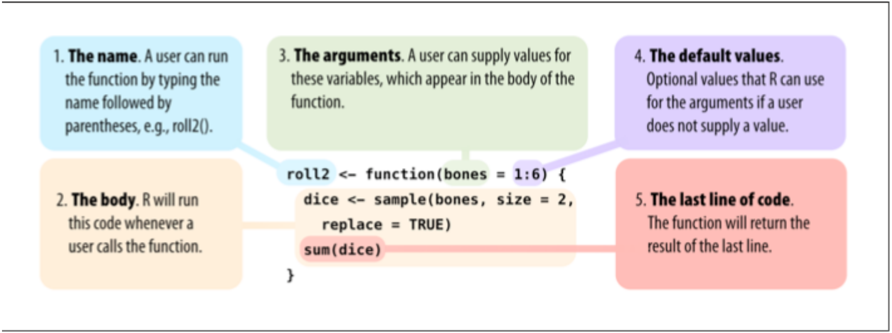
Function Structure

The syntax for writing your own function:

```
name_of_function <- function(arg1, arg2, ...){
^^I#Code that gets executed when function is called
}
```

Example

```
hello <- function(){
^^Iprint('hello!')
}
hello()
[1] "hello!"
```


1. The name. A user can run the function by typing the name followed by parentheses, e.g., roll2().
3. The arguments. A user can supply values for these variables, which appear in the body of the function.
4. The default values. Optional values that R can use for the arguments if a user does not supply a value.
2. The body. R will run this code whenever a user calls the function.
5. The last line of code. The function will return the result of the last line.
roll2 <- function(bones = 1:6) {
dice <- sample(bones, size = 2,
replace = TRUE)
sum(dice)
}

```
helloyou <- function(name){
^^Iprint(paste('hello ',name))
}
helloyou('Sammy')
[1] "hello  Sammy"
```

```
add_num <- function(num1,num2){
^^Iprint(num1+num2)
}
add_num(5,10)
[1] 15
```

Default Values

We have had to define every single argument in the function when using it, but we can also have default values by using an equals sign, for example:

```
hello_someone <- function(name='Frankie'){
^^Iprint(paste('Hello ',name))
}
# uses default
hello_someone()
[1] "Hello  Frankie"

# overwrite default
hello_someone('Sammy')
[1] "Hello  Sammy"
```

Returning Values

If we wanted to return the results so that we could assign them to a variable, we can use the return keyword for this task in the following manner:

```
formal <- function(name='Sam',title='Sir'){
^^Ireturn(paste(title,' ',name))
}
var <- formal('Marie Curie','Ms.')
var
[1] "Ms.   Marie Curie"
```

Scope

- Scope is the term we use to describe how objects and variable get defined within R
- if a variable is defined only inside a function than its scope is limited to that function

```
times5 <- function(input) {
^^Iresult <- input ^ 2
^^Ireturn(result)
}
result
Error: object 'result' not found
input
Error: object 'input' not found
```

These error indicate that these variables are only defined inside the **scope** of the function.

```r
v <- "I'm global v"
stuff <- "I'm global stuff"

fun <- function(stuff){
^^Iprint(v)
^^Istuff <- 'Reassign stuff inside func'
^^Iprint(stuff)
}

print(v) #print v
print(stuff) #print stuff
fun(stuff) # pass stuff to function
# reassignment only happens in scope of function
print(stuff)

[1] "I'm global v"
[1] "I'm global stuff"
[1] "I'm global v"
[1] "Reassign stuff inside func"
[1] "I'm global stuff"
```

```r
double <- function(a) {
a <- 2*a
a
}
var <- 5
double(var)
var
[1] 10
[1] 5
```

Statistics

Common Statistics Methods

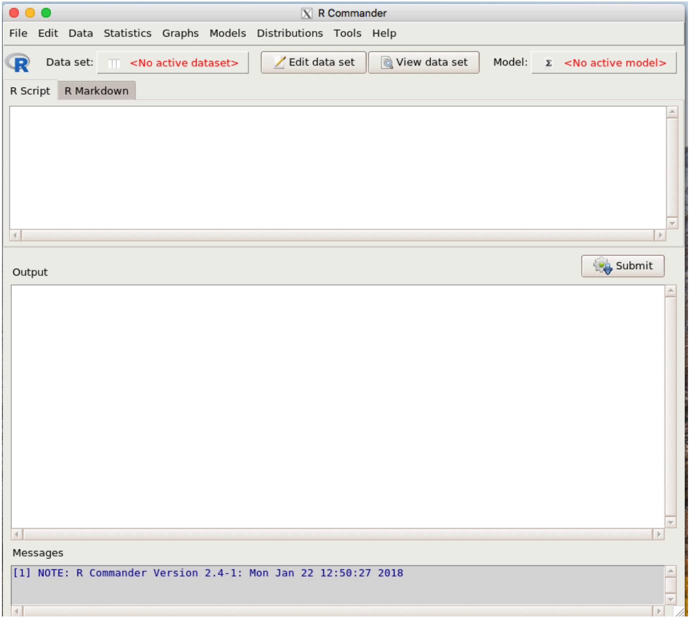
- Correlation
- Linear Regression
- Comparing 2 means
- ANOVA

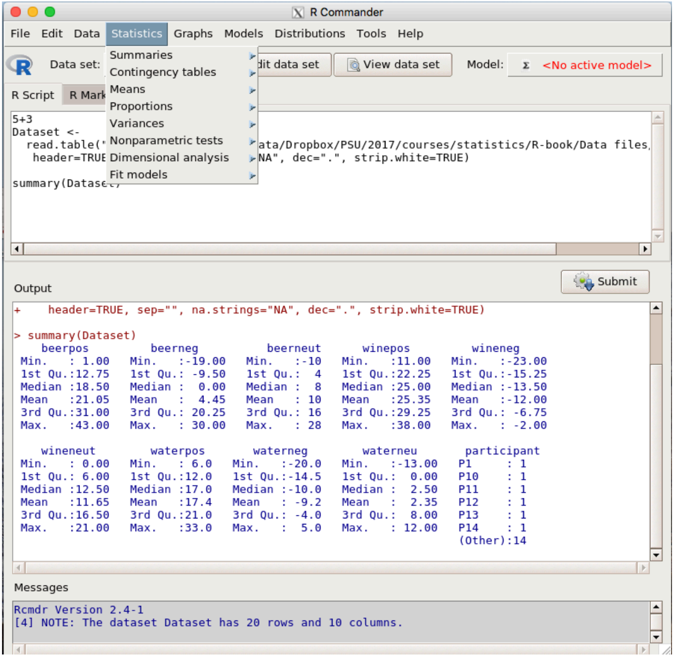
R Commander

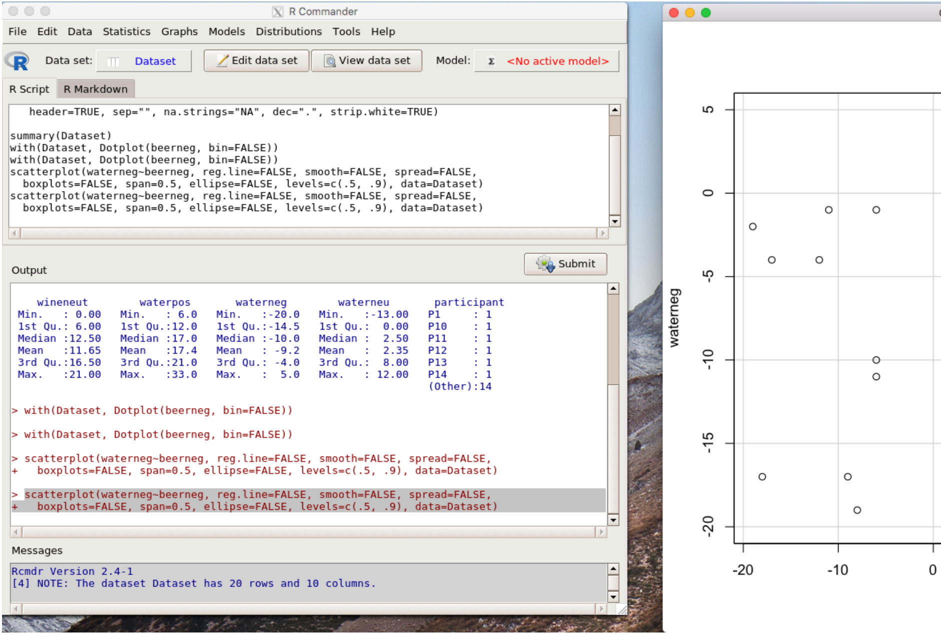
- The R Commander is a graphical user interface (GUI) to the free
- The **Rcmdr** package, which is freely available on CRAN
- Support Statistics via GUI

```
install.packages("Rcmdr")
library(Rcmdr)
^^I
```

Installation notes: `https://socialsciences.mcmaster.ca/jfox/Misc/Rcmdr/installation-notes.html`


R Commander
File Edit Data Statistics Graphs Models Distributions Tools Help
Data set:
<No active dataset>
Edit data set
View data set
Model:
<No active model>
R Script
R Markdown
Submit
Output
Messages
[1] NOTE: R Commander Version 2.4-1: Mon Jan 22 12:50:27 2018


R Commander
File Edit Data Statistics Graphs Models Distributions Tools Help
Data set:
dit data set
View data set
Model:
<No active model>
Summaries
Contingency tables
Means
Proportions
Variances
Nonparametric tests
Dimensional analysis
Fit models
R Script
R Mark
5+3
Dataset <-
read.table("
header=TRUE
ata/Dropbox/PSU/2017/courses/statistics/R-book/Data files,
NA", dec=".", strip.white=TRUE)
summary(Dataset)
Output
Submit
+ header=TRUE, sep="", na.strings="NA", dec=".", strip.white=TRUE)
> summary(Dataset)
beerpos beerneg beerneut winepos wineneg
Min. : 1.00 Min. :-19.00 Min. :-10 Min. :11.00 Min. :-23.00
1st Qu.:12.75 1st Qu.: -9.50 1st Qu.: 4 1st Qu.:22.25 1st Qu.:-15.25
Median :18.50 Median : 0.00 Median : 8 Median :25.00 Median :-13.50
Mean :21.05 Mean : 4.45 Mean : 10 Mean :25.35 Mean :-12.00
3rd Qu.:31.00 3rd Qu.: 20.25 3rd Qu.: 16 3rd Qu.:29.25 3rd Qu.: -6.75
Max. :43.00 Max. : 30.00 Max. : 28 Max. :38.00 Max. : -2.00
wineneut waterpos waterneg waterneu participant
Min. : 0.00 Min. : 6.0 Min. :-20.0 Min. :-13.00 P1 : 1
1st Qu.: 6.00 1st Qu.:12.0 1st Qu.:-14.5 1st Qu.: 0.00 P10 : 1
Median :12.50 Median :17.0 Median :-10.0 Median : 2.50 P11 : 1
Mean :11.65 Mean :17.4 Mean : -9.2 Mean : 2.35 P12 : 1
3rd Qu.:16.50 3rd Qu.:21.0 3rd Qu.: -4.0 3rd Qu.: 8.00 P13 : 1
Max. :21.00 Max. :33.0 Max. : 5.0 Max. : 12.00 P14 : 1
(Other):14
Messages
Rcmdr Version 2.4-1
[4] NOTE: The dataset Dataset has 20 rows and 10 columns.


R Commander
File Edit Data Statistics Graphs Models Distributions Tools Help
Data set: Dataset
Edit data set
View data set
Model: <No active model>
R Script
R Markdown
header=TRUE, sep="", na.strings="NA", dec=".", strip.white=TRUE)
summary(Dataset)
with(Dataset, Dotplot(beerneg, bin=FALSE))
with(Dataset, Dotplot(beerneg, bin=FALSE))
scatterplot(waterneg~beerneg, reg.line=FALSE, smooth=FALSE, spread=FALSE,
boxplots=FALSE, span=0.5, ellipse=FALSE, levels=c(.5, .9), data=Dataset)
scatterplot(waterneg~beerneg, reg.line=FALSE, smooth=FALSE, spread=FALSE,
boxplots=FALSE, span=0.5, ellipse=FALSE, levels=c(.5, .9), data=Dataset)
Submit
Output
wineneut waterpos waterneg waterneu participant
Min. : 0.00 Min. : 6.0 Min. :-20.0 Min. :-13.00 P1 : 1
1st Qu.: 6.00 1st Qu.:12.0 1st Qu.:-14.5 1st Qu.: 0.00 P10 : 1
Median :12.50 Median :17.0 Median :-10.0 Median : 2.50 P11 : 1
Mean :11.65 Mean :17.4 Mean : -9.2 Mean : 2.35 P12 : 1
3rd Qu.:16.50 3rd Qu.:21.0 3rd Qu.: -4.0 3rd Qu.: 8.00 P13 : 1
Max. :21.00 Max. :33.0 Max. : 5.0 Max. : 12.00 P14 : 1
(Other):14
> with(Dataset, Dotplot(beerneg, bin=FALSE))
> with(Dataset, Dotplot(beerneg, bin=FALSE))
> scatterplot(waterneg~beerneg, reg.line=FALSE, smooth=FALSE, spread=FALSE,
+ boxplots=FALSE, span=0.5, ellipse=FALSE, levels=c(.5, .9), data=Dataset)
> scatterplot(waterneg~beerneg, reg.line=FALSE, smooth=FALSE, spread=FALSE,
+ boxplots=FALSE, span=0.5, ellipse=FALSE, levels=c(.5, .9), data=Dataset)
Messages
Rcmdr Version 2.4-1
[4] NOTE: The dataset Dataset has 20 rows and 10 columns.
waterneg
5
0
-5
-10
-15
-20
-20
-10
0

Correlation

- It is a way of measuring the extent to which two variables are relate
- It measures the pattern of responses across variables
- Correlation and Causality
    - In any correlation, causality between two variables cannot be assumed because there may be other measured or unmeasured variables affecting the results
    - Correlation coefficients say nothing about which variable causes the other to change

Correlation (cont.)

To compute basic correlation coefficients there are three main functions that can be used:

- **cor()**
- **cor.test()**
- **rcorr()**

Things to Know about the Correlation

- It varies between -1 and +1 (0 = no relationship)
- It is an effect size
  - $+.1$ or $-.1$ = small effect
  - $+.3$ or $-.3$ = medium effect
  - $+.5$ or $-.5$ = large effect

Pearson correlations

```
cor(examData, use = "complete.obs", method = "pearson")

rcorr(examData, type = "pearson")

cor.test(examData$Exam, examData$Anxiety, method = "pearson")
```

Regression

- A way of predicting the value of one variable from another
  - It is a hypothetical model of the relationship between two variables
  - The model used is a linear one

The Regression Equation

$$\hat{Y} = \mathrm{b}\mathrm{X} + \mathrm{a}$$

$\hat{Y}$ is the predicated value of the Y variable
b is the unstandardized **regression coefficient**, or the **slope**
a is **intercept** (i.e., the point where the regression line intercepts the Y axis)

Regression with R

The basic syntax for a regression analysis in R is **lm**(Y model)

Syntax	Model	Comments
Y ~ A	$Y = \beta_o + \beta_1 A$	Straight-line with an implicit y-intercept
Y ~ -1 + A	$Y = \beta_1 A$	Straight-line with no y-intercept; that is, a fit forced through (0,0)
Y ~ A + I(A^2)	$Y = \beta_o + \beta_1 A + \beta_2 A^2$	Polynomial model; note that the identity function **I( )** allows terms in the model to include normal mathematical symbols.
Y ~ A + B	$Y = \beta_o + \beta_1 A + \beta_2 B$	A first-order model in A and B without interaction terms.
Y ~ A:B	$Y = \beta_o + \beta_1 AB$	A model containing only first-order interactions between A and B.
Y ~ A*B	$Y = \beta_o + \beta_1 A + \beta_2 B + \beta_3 AB$	A full first-order model with a term; an equivalent code is Y ~ A + B + A:B.
Y ~ (A + B + C)^2	$Y = \beta_o + \beta_1 A + \beta_2 B + \beta_3 C + \beta_4 AB + \beta_5 AC + \beta_6 AC$	A model including all first-order effects and interactions up to the n^{th} order, where n is given by **( )^n**. An equivalent code in this case is Y ~ A*B*C – A:B:C.

Regression with R (cont.)

Y = 1,2,3,4,5

X = 2,4,3,5,6

$>$ fit $<-$ lm(Y $\sim$ X)

$>$ summary(fit)

```
Call:
lm(formula = simplelinear$Y ~ simplelinear$X, data = simplelinear)

Residuals:
   1    2    3    4    5
-0.2 -1.0  0.9  0.1  0.2

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)     -0.6000     1.0677  -0.562   0.6134
simplelinear$X   0.9000     0.2517   3.576   0.0374 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7958 on 3 degrees of freedom
Multiple R-squared:   0.81,     Adjusted R-squared:  0.7467
F-statistic: 12.79 on 1 and 3 DF,  p-value: 0.03739
```

Multiple Regression

When we add the second predictor variable to the model, we get the following regression equation:

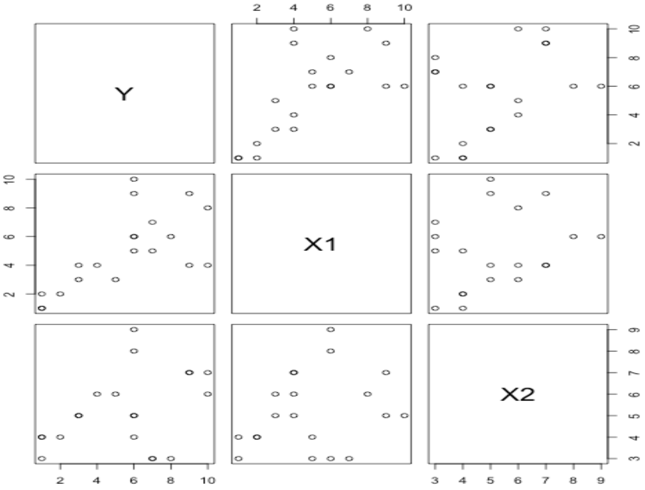
$$\hat{Y} = a + b_1 X_1 + b_2 X_2$$

where
$\hat{Y}$ is the predicted value of the dependent variable,
a is the intercept,
X_1 is the value of the first predictor variable, and
X_2 is the value of the second predictor variable

Example

Y	X1	X2
2	2	4
1	2	4
1	1	4
1	1	3
5	3	6
4	4	6
7	5	3
6	5	4
7	7	3
8	6	3
3	4	5
3	3	5
6	6	9
6	6	8
10	8	6
9	9	7
6	10	5
6	9	5
9	4	7
10	4	7

Example (cont.)



Multiregression with R

```
> results = lm(Y ~ X1 + X2)
> summary(results)
```

```
Call:
lm(formula = multipleRegression$Y ~ multipleRegression$X1 + multipleRegression$X2,
    data = multipleRegression)

Residuals:
    Min      1Q  Median      3Q     Max
-2.8406 -1.4416 -0.9952  1.2632  4.4350

Coefficients:
                      Estimate Std. Error t value Pr(>|t|)
(Intercept)             0.1026     1.6602   0.062  0.95146
multipleRegression$X1   0.6771     0.1953   3.467  0.00295 **
multipleRegression$X2   0.3934     0.2949   1.334  0.19971
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.191 on 17 degrees of freedom
Multiple R-squared:  0.5054,    Adjusted R-squared:  0.4472
F-statistic: 8.686 on 2 and 17 DF,  p-value: 0.002519
```

Multiregression with R (cont.)

From the output, you see that the prediction equation is:

$$\hat{Y} = 0.1026 + 0.6771\mathrm{X}_1 + 0.3934\mathrm{X}_2$$

Multiregression with R (cont.)

From the output, you see that the prediction equation is:

$$\hat{Y} = 0.1026 + 0.6771\mathrm{X}_1 + 0.3934\mathrm{X}_2$$

How?

Comparing 2 means (t-test)

- Two samples of data are collected and the sample means calculated. These means might differ by either a little or a lot.
- We compare the difference between the sample means that we collected to the difference between the sample means that we would expect to obtain if there were no effect (i.e. if the null hypothesis were true)

The Independent t-test

To do a t-test we use the function **t.test()**

If you have the data for different groups stored in a single column:

```
newModel<-t.test(outcome ~ predictor, data = dataFrame, paired = FALSE/TRUE)

ind.t.test<-t.test(Anxiety ~ Group, data = spiderLong)
```

If you have the data for different groups stored in two columns:

```
newModel<-t.test(scores group 1, scores group 2, paired= FALSE/TRUE)

ind.t.test<-t.test(spiderWide$real, spiderWide$picture)
```

The dependent t-test

To do a dependent **t-test** we again use the function t.test() but this time include the option **paired = TRUE**. If we have scores from different groups stored in different columns:

```
dep.t.test<-t.test(spiderWide$real, spiderWide$picture, paired = TRUE)
```

If we had our data stored in long format so that our group scores are in a single column and group membership is expressed in a second column:

```
dep.t.test<-t.test(Anxiety ~ Group, data = spiderLong, paired = TRUE)
```

Analysis of Variance (ANOVA)

- Compares several means
- Can be used when you have manipulated more than one independent variable
- It is an extension of regression (the general linear model)

One-Way ANOVA

Using **lm()**:

```
viagraModel<-lm(libido~dose, data = viagraData)
```

Using **aov()**:

```
viagraModel<-aov(libido ~ dose, data = viagraData)
summary(viagraModel)
```

Post Hoc Tests

- Bonferroni
- BH
- Tukey

```
postHocs<-glht(viagraModel, linfct = mcp(dose = "Tukey"))
summary(postHocs)
confint(postHocs)
```

2-Way ANOVA

- Two-way = 2 Independent variables
- Three-way = 3 Independent variables
- Several independent variables is known as **factorial design**

Factorial ANOVA Model

```r
gogglesModel<-aov(attractiveness ~ gender + alcohol + gender:alcohol, data = gogglesData)
```

```r
gogglesModel<-aov(attractiveness ~ alcohol*gender, data = gogglesData)
```

Data Visualization

Grammar of Graphics and ggplot2

One of the most common and popular libraries for data visualization in R, **ggplot2**

ggplot2 has several advantages:

- Plot specification at a high level of abstraction
- Very flexible
- Theme system for polishing plot appearance
- Mature and complete graphics system
- Many users, active mailing list
- Lot's of online help available (StackOverflow, etc...)

What **ggplot2** not ideal for:

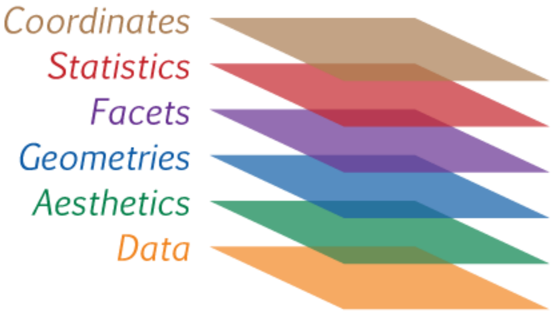
- Interactive graphics
- Graph Theory Plots (Graph Nodes)
- 3-D Graphics

Grammar of Graphics

- ggplot2 is based on the grammar of graphics
- the idea that you can build every graph from the same few components: a data set, a set of geoms—visual marks that represent data points, and a coordinate system.
- To display data values, map variables in the data set to aesthetic properties of the geom like size, color, and x and y locations

Layers for building Visualizations

ggplot2 is based off the grammar of graphics, which sets a paradigm for data visualization in layers:



Geoms in ggplot2

Name	Description
abline	Line, specified by slope and intercept
area	Area plots
bar	Bars, rectangles with bases on y-axis
blank	Blank, draws nothing
boxplot	Box-and-whisker plot
contour	Display contours of a 3d surface in 2d
crossbar	Hollow bar with middle indicated by horizontal line
density	Display a smooth density estimate
density_2d	Contours from a 2d density estimate
errorbar	Error bars
histogram	Histogram
hline	Line, horizontal
interval	Base for all interval (range) geoms
jitter	Points, jittered to reduce overplotting
line	Connect observations, in order of x value
linerange	An interval represented by a vertical line
path	Connect observations, in original order
point	Points, as for a scatterplot
pointrange	An interval represented by a vertical line, with a point in the middle
polygon	Polygon, a filled path
quantile	Add quantile lines from a quantile regression
ribbon	Ribbons, y range with continuous x values
rug	Marginal rug plots
segment	Single line segments
smooth	Add a smoothed condition mean
step	Connect observations by stairs
text	Textual annotations
tile	Tile plot as densely as possible, assuming that every tile is the same size
vline	Line, vertical

Geoms that were created by modifying the defaults of another geom

Aliased geom	Base geom	Changes in default
area	ribbon	`aes(min = 0, max = y), position = "stack"`
density	area	`stat = "density"`
freqpoly	line	`stat = "bin"`
histogram	bar	`stat = "bin"`
jitter	point	`position = "jitter"`
quantile	line	`stat = "quantile"`
smooth	ribbon	`stat = "smooth"`

Data and Set-up

```r
#import ggplot2
library(ggplot2)
```

The general syntax of using ggplot2 will look like this:

```
ggplot(data = <default data set>,
       aes(x = <default x axis variable>,
           y = <default y axis variable>,
           ... <other default aesthetic mappings>),
       ... <other plot defaults>) +

       geom_<geom type>(aes(size = <size variable for this geom>,
                     ... <other aesthetic mappings>),
                  data = <data for this point geom>,
                  stat = <statistic string or function>,
                  position = <position string or function>,
                  color = <"fixed color specification">,
                  <other arguments, possibly passed to the _stat_ function) +

  scale_<aesthetic>_<type>(name = <"scale label">,
                   breaks = <where to put tick marks>,
                   labels = <labels for tick marks>,
                   ... <other options for the scale>) +

  theme(plot.background = element_rect(fill = "gray"),
        ... <other theme elements>)
```

Stat

A statistical transformation, or **stat**, transforms the data, typically by summarizing it in some manner.

A stat takes a dataset as input and returns a dataset as output, and so a stat can add new variables to the original dataset.

```
ggplot(diamonds, aes(carat)) + geom_histogram(aes(y = ..density..), binwidth = 0.1)
```

Stats in ggplot2

Name	Description
bin	Bin data
boxplot	Calculate components of box-and-whisker plot
contour	Contours of 3d data
density	Density estimation, 1d
density_2d	Density estimation, 2d
function	Superimpose a function
identity	Don't transform data
qq	Calculation for quantile-quantile plot
quantile	Continuous quantiles
smooth	Add a smoother
spoke	Convert angle and radius to xend and yend
step	Create stair steps
sum	Sum unique values. Useful for overplotting on scatter-plots
summary	Summarise y values at every unique x
unique	Remove duplicates

Default statistics and aesthetics

Name	Default stat	Aesthetics
abline	abline	colour, linetype, size
area	identity	colour, fill, linetype, size, **x**, **y**
bar	bin	colour, fill, linetype, size, weight, **x**
bin2d	bin2d	colour, fill, linetype, size, weight, **xmax**, **xmin**, **ymax**, **ymin**
blank	identity	
boxplot	boxplot	colour, fill, **lower**, **middle**, size, **upper**, weight, **x**, **ymax**, **ymin**
contour	contour	colour, linetype, size, weight, **x**, **y**
crossbar	identity	colour, fill, linetype, size, **x**, **y**, **ymax**, **ymin**
density	density	colour, fill, linetype, size, weight, **x**, **y**
density2d	density2d	colour, linetype, size, weight, **x**, **y**
errorbar	identity	colour, linetype, size, width, **x**, **ymax**, **ymin**
freqpoly	bin	colour, linetype, size
hex	binhex	colour, fill, size, **x**, **y**
histogram	bin	colour, fill, linetype, size, weight, **x**
hline	hline	colour, linetype, size
jitter	identity	colour, fill, shape, size, **x**, **y**
line	identity	colour, linetype, size, **x**, **y**
linerange	identity	colour, linetype, size, **x**, **ymax**, **ymin**
path	identity	colour, linetype, size, **x**, **y**
point	identity	colour, fill, shape, size, **x**, **y**
pointrange	identity	colour, fill, linetype, shape, size, **x**, **y**, **ymax**, **ymin**
polygon	identity	colour, fill, linetype, size, **x**, **y**
quantile	quantile	colour, linetype, size, weight, **x**, **y**
rect	identity	colour, fill, linetype, size, **xmax**, **xmin**, **ymax**, **ymin**
ribbon	identity	colour, fill, linetype, size, **x**, **ymax**, **ymin**
rug	identity	colour, linetype, size
segment	identity	colour, linetype, size, **x**, **xend**, **y**, **yend**
smooth	smooth	alpha, colour, fill, linetype, size, weight, **x**, **y**
step	identity	colour, linetype, size, **x**, **y**
text	identity	angle, colour, hjust, **label**, size, vjust, **x**, **y**
tile	identity	colour, fill, linetype, size, **x**, **y**
vline	vline	colour, linetype, size

Position adjustments

Adjustment	Description
dodge	Adjust position by dodging overlaps to the side
fill	Stack overlapping objects and standardise have equal height
identity	Don't adjust position
jitter	Jitter points to avoid overplotting
stack	Stack overlapping objects on top of one another

The different types of adjustment are best illustrated with a bar chart.

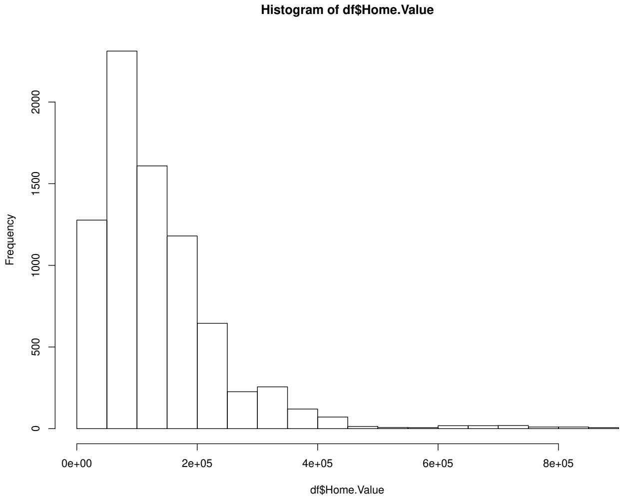
Using ggplot2

Quick Example with Histograms

We have a couple of options for quickly producing histograms off the columns of a data frame.

- hist()
- qplot()
- ggplot()

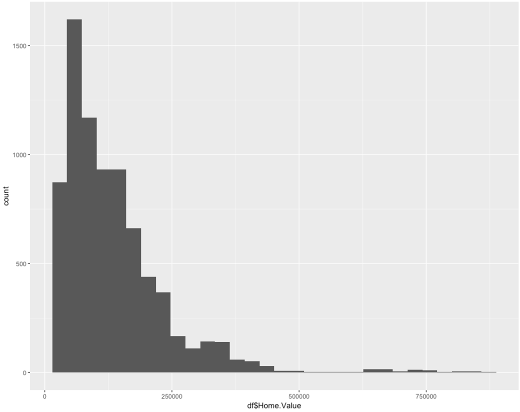
```r
library(data.table)
df <- fread('state_real_estate_data.csv')
hist(df$Home.Value)
```



Using qplot

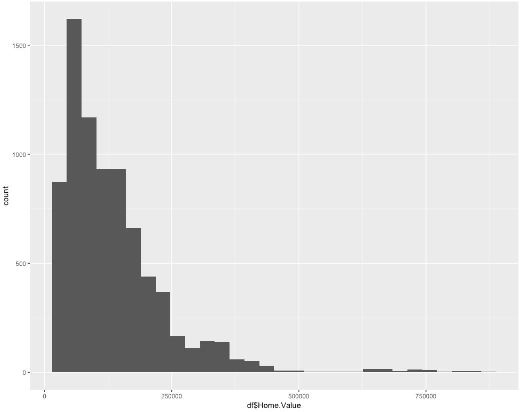
library(ggplot2)

```
library(ggplot2)
qplot(df$Home.Value)
```



Using ggplot

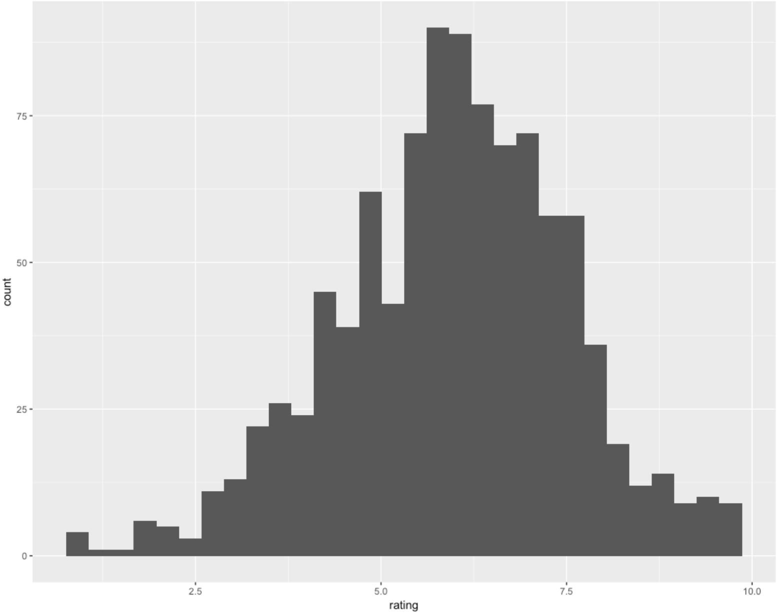
```
# Using ggplot, lots of ability to customize, but bit more complicated!
ggplot(data = df,aes(df$Home.Value))+geom_histogram()
```



Histograms with ggplot2

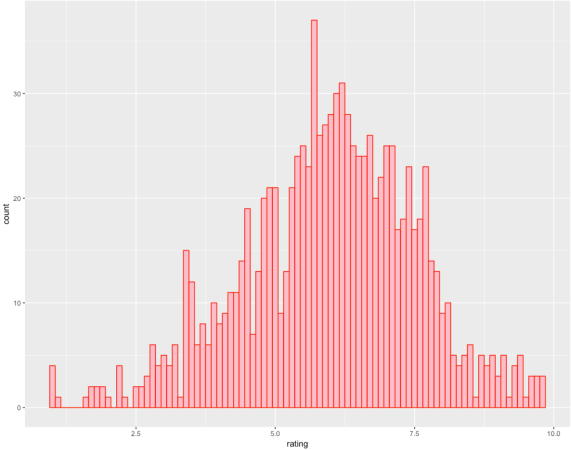
We'll use the movie dataset that comes with ggplot:

```
library(ggplot2movies)
df <- movies <- movies[sample(nrow(movies), 1000), ]

# ggplot(data, aesthetics)
pl <- ggplot(df,aes(x=rating))

# Add Histogram Geometry
pl + geom_histogram()
```


75
50
25
0
count
2.5
5.0
7.5
10.0
rating

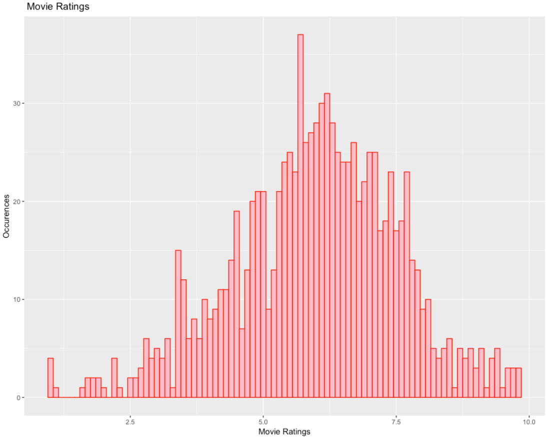
Adding Color

```r
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(binwidth=0.1,color='red',fill='pink')
```



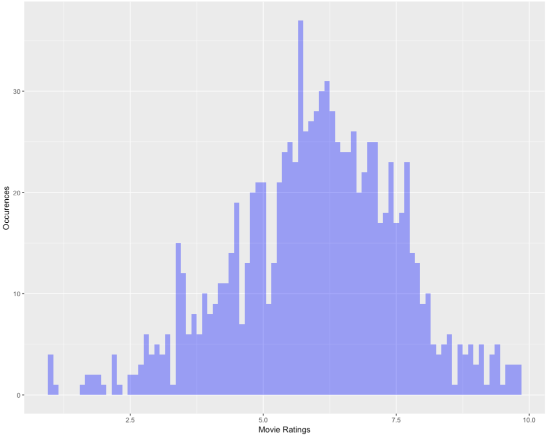
Adding Labels

```
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(binwidth=0.1,color='red',fill='pink') + xlab('Movie Ratings')+ ylab('
     Occurences') + ggtitle(' Movie Ratings')
```



Change Alpha (Transparency)

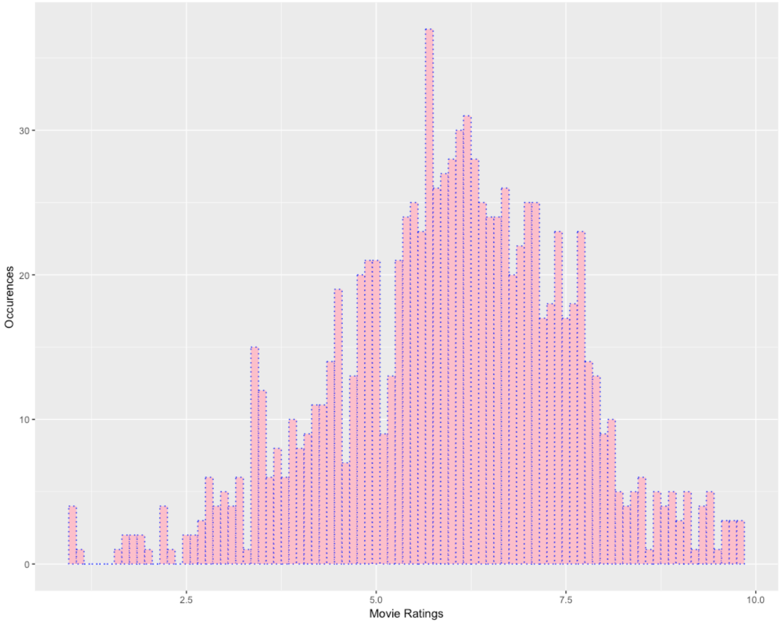
```r
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(binwidth=0.1,fill='blue',alpha=0.4) + xlab('Movie Ratings')+ ylab('
    Occurences')
```



Linetypes

We have the options: "blank", "solid", "dashed", "dotted", "dotdash", "longdash", and "twodash".

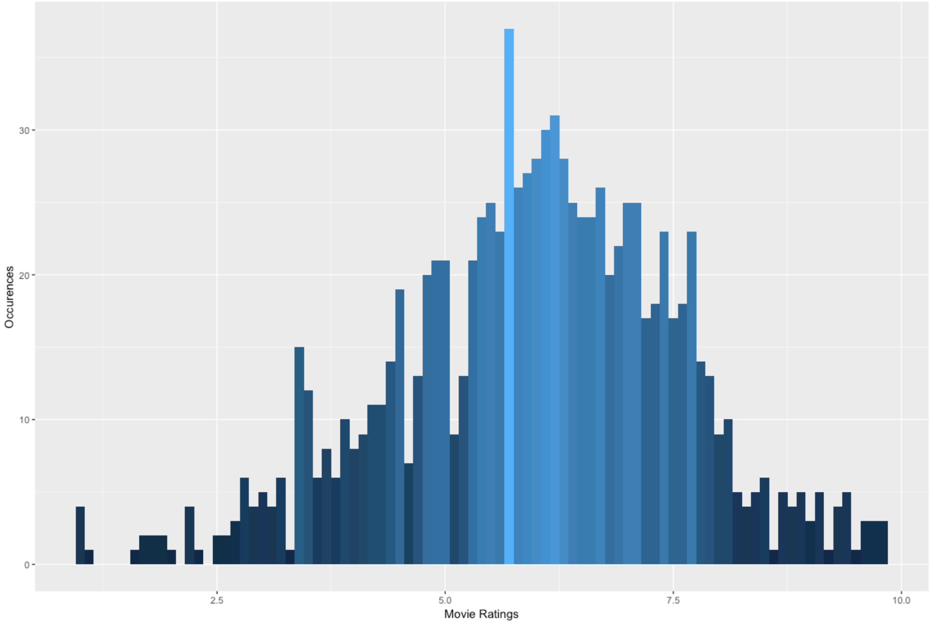
```
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(binwidth=0.1,color='blue',fill='pink',linetype='dotted') + xlab('Movie
      Ratings')+ ylab('Occurences')
```


Occurences
0
10
20
30
2.5
5.0
7.5
10.0
Movie Ratings

Advanced Aesthetics

We can add a **aes()** argument to the geom_histogram for some more advanced features. But, ggplot gives you the ability to edit **color** and **fill** scales.

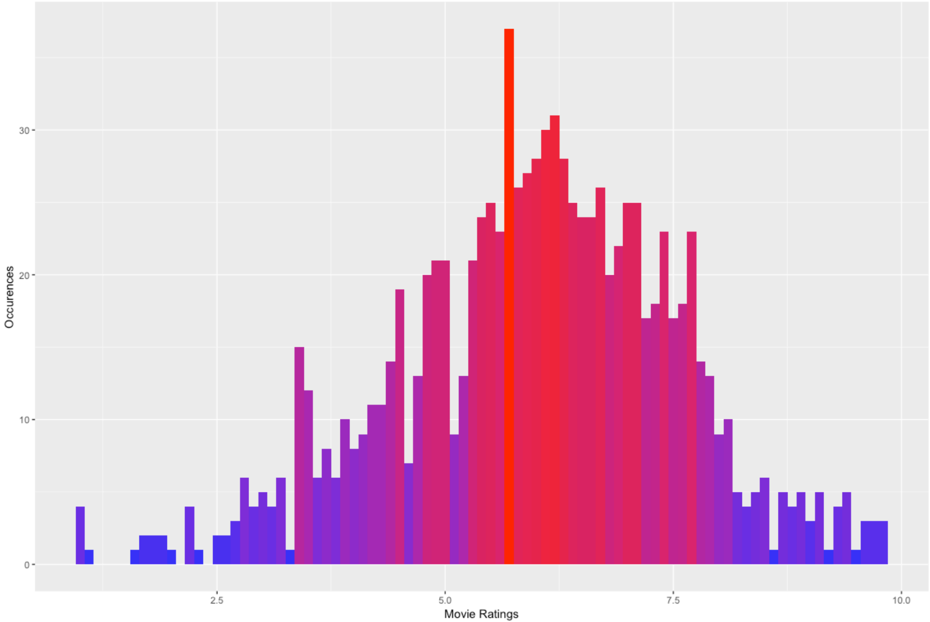
```
# Adding Labels
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(binwidth=0.1,aes(fill=..count..)) + xlab('Movie Ratings')+ ylab('Occurences
     ')
```



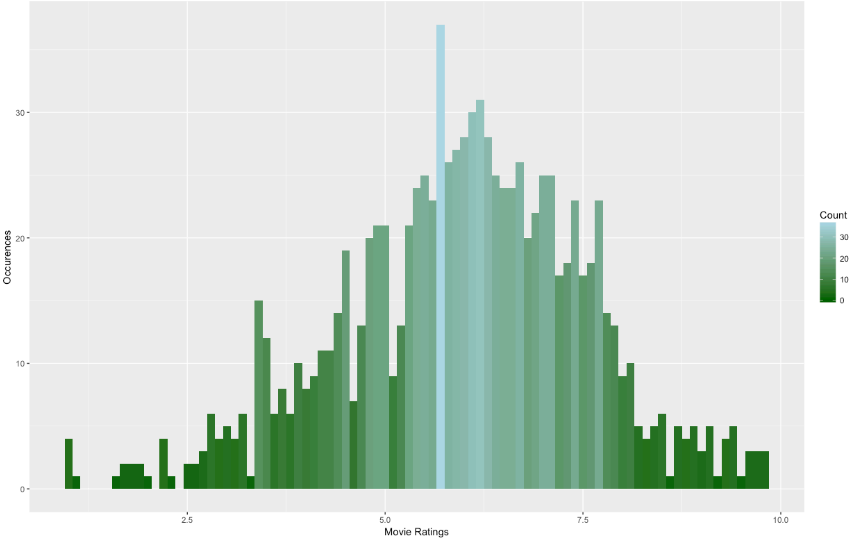
Occurences
30
20
10
0
2.5
5.0
7.5
10.0
Movie Ratings

You can further edit this by adding the **scale_fill_gradient()** function to your ggplot objects:

```
# Adding Labels
pl <- ggplot(df,aes(x=rating))
pl2 <- pl + geom_histogram(binwidth=0.1,aes(fill=..count..)) + xlab('Movie Ratings')+ ylab('
    Occurences')

# scale_fill_gradient('Label',low=color1,high=color2)
pl2 + scale_fill_gradient('Count',low='blue',high='red')
```

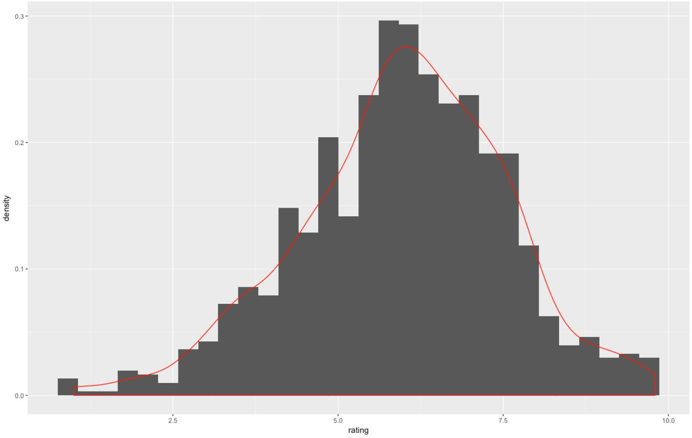

30
20
10
0
Occurences
2.5
5.0
7.5
10.0
Movie Ratings

```r
# scale_fill_gradient('Label',low=color1,high=color2)
pl2 + scale_fill_gradient('Count',low='darkgreen',high='lightblue')
```



Adding Density Plot

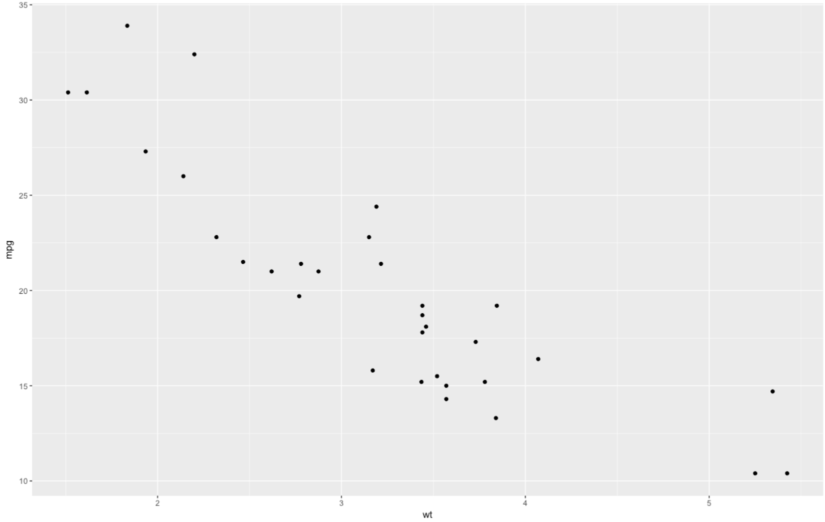
```r
# Adding Labels
pl <- ggplot(df,aes(x=rating))
pl + geom_histogram(aes(y=..density..)) + geom_density(color='red')
```



Scatterplots with ggplot2

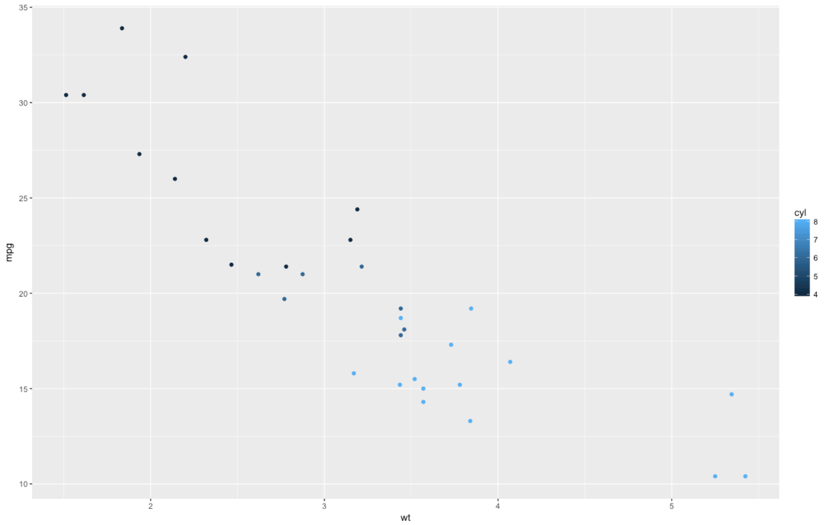
Scatter plots allow us to place points that let us see possible correlations between two features of a data set.

```r
library('ggplot2')
df <- mtcars

pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point()
```

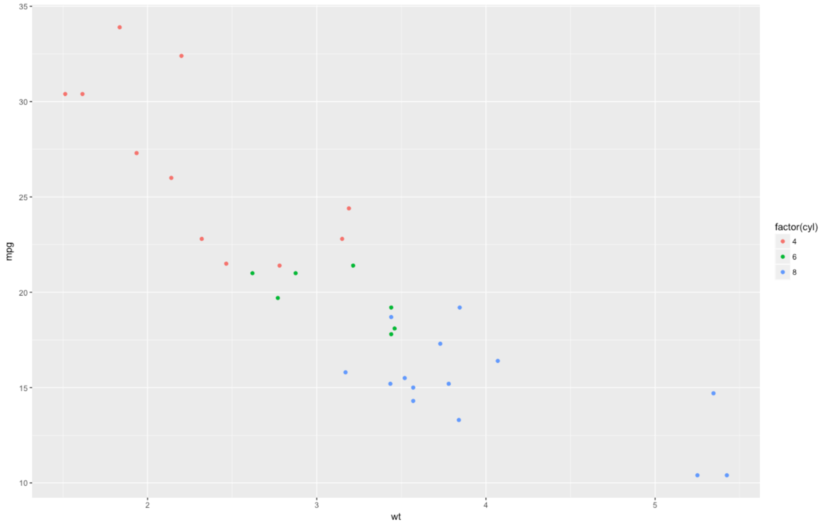

35
30
25
20
15
10
mpg
2
3
4
5
wt

Adding 3rd feature

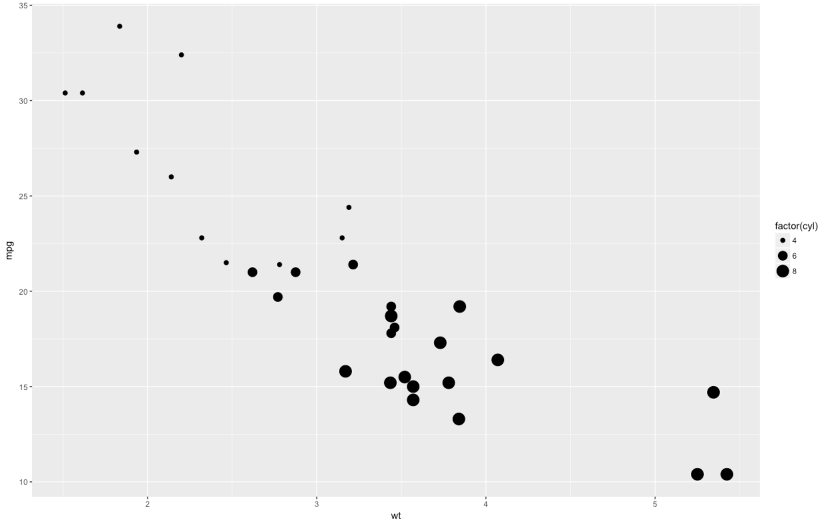
```
pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point(aes(color=cyl))
```



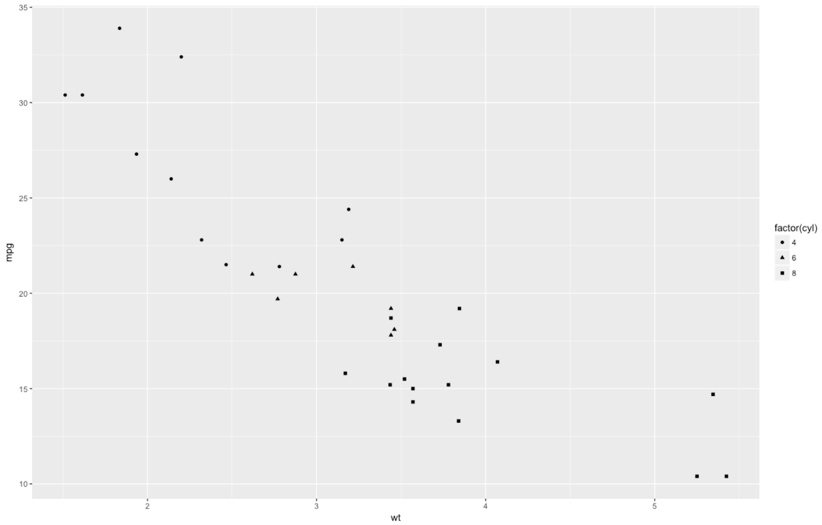
```
pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point(aes(color=factor(cyl)))
```



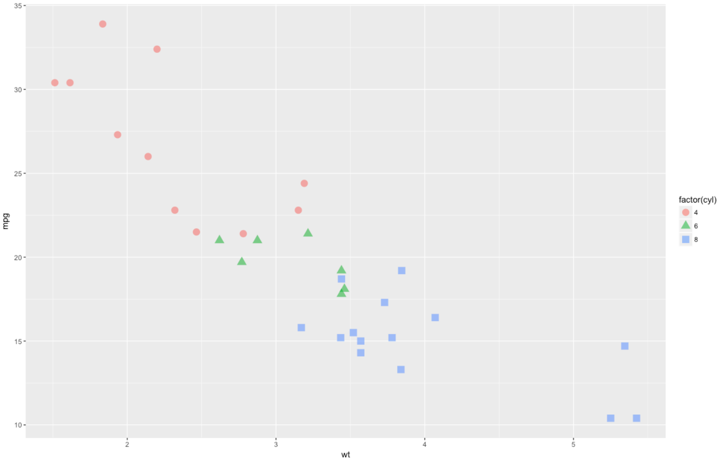
```r
pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point(aes(size=factor(cyl)))
```



```r
# With Shapes
pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point(aes(shape=factor(cyl)))
```

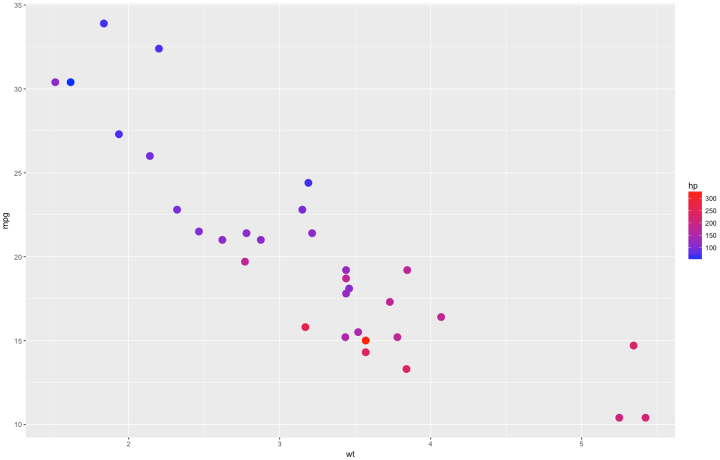


```
# Better version
# With Shapes
pl <- ggplot(data=df,aes(x = wt,y=mpg))
pl + geom_point(aes(shape=factor(cyl),color=factor(cyl)),size=4,alpha=0.6)
```



Gradient Scales

```
pl + geom_point(aes(colour = hp),size=4) + scale_colour_gradient(high='red',low = "blue")
```

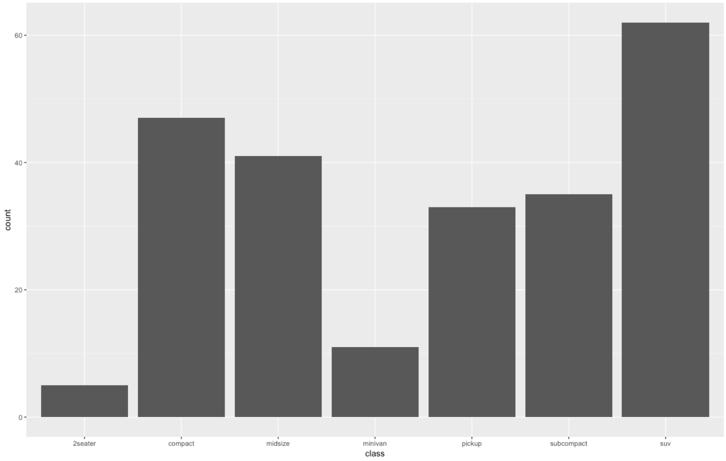


Barplots with ggplot2

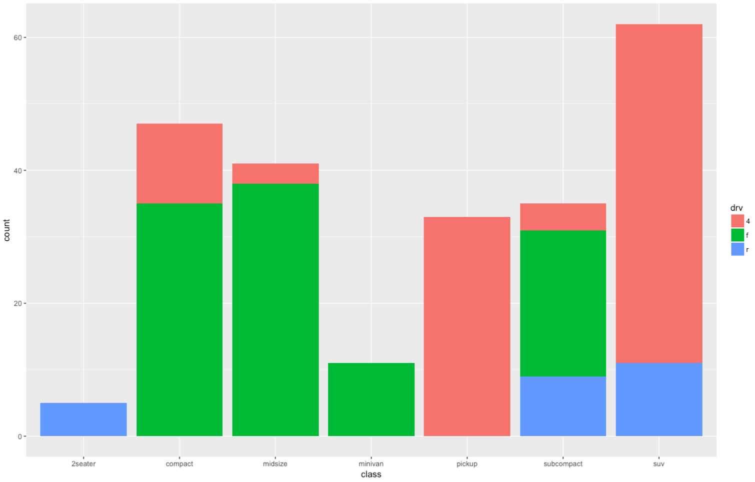
There are two types of bar charts, determined by what is mapped to bar height.

- By **default**, geom_bar uses **stat="count"** which makes the height of the bar proportion to the number of cases in each group
- If you want the heights of the bars to represent values in the data, use **stat="identity"** and map a variable to the **y aesthetic**

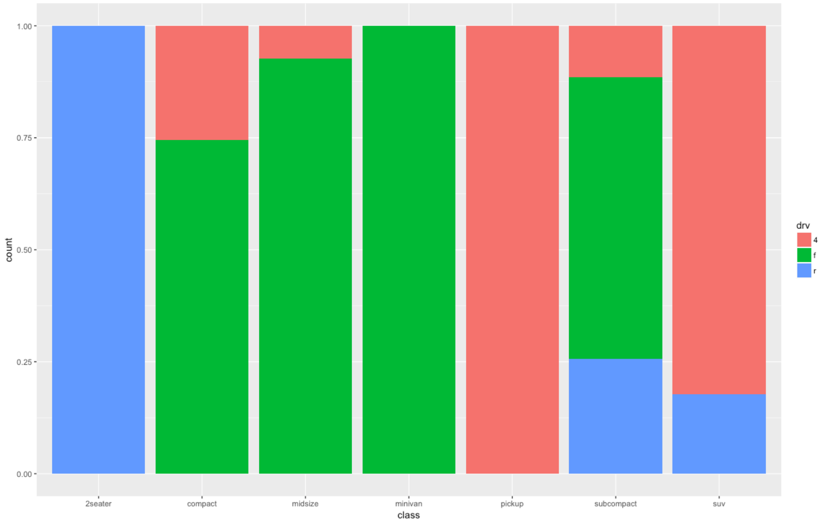
```r
library(ggplot2)
# counts (or sums of weights)
g <- ggplot(mpg, aes(class))
# Number of cars in each class:
g + geom_bar()
```



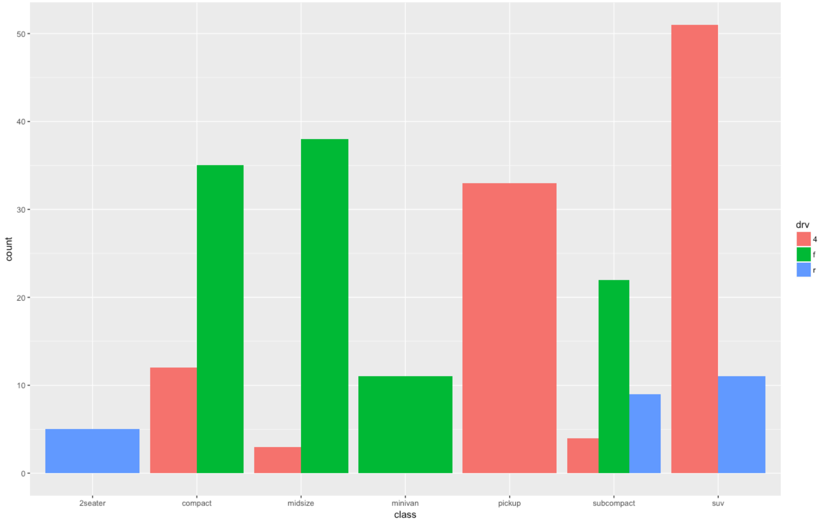
```r
# Bar charts are automatically stacked when multiple bars are placed at the same location
g + geom_bar(aes(fill = drv))
```



```r
g + geom_bar(aes(fill = drv), position = "fill")
```

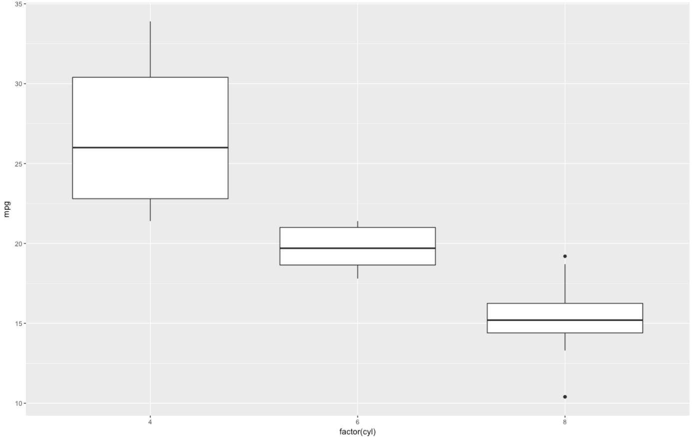


```r
# You can instead dodge, or fill them
g + geom_bar(aes(fill = drv), position = "dodge")
```

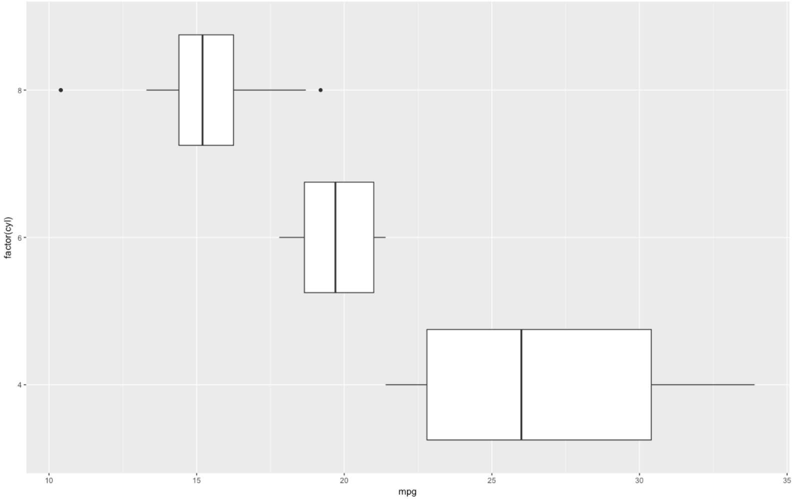


Boxplots with ggplot2

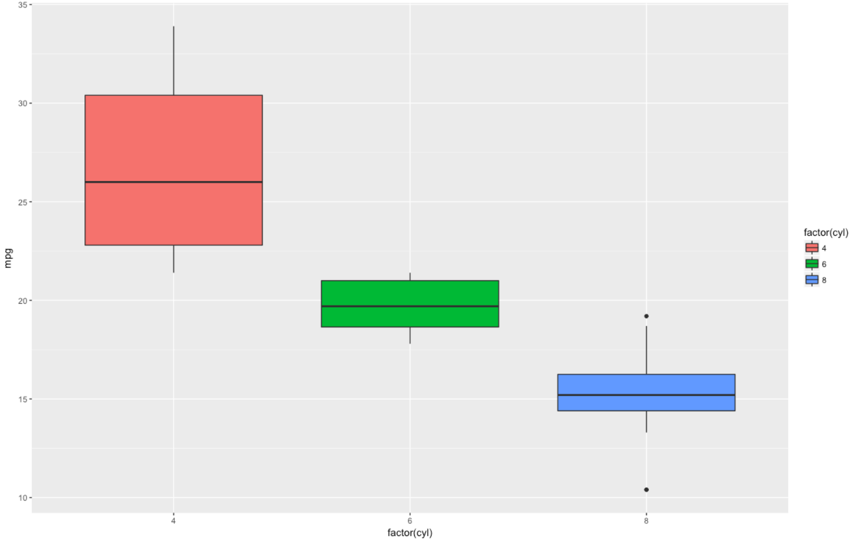
```
df <- mtcars
pl <- ggplot(mtcars, aes(factor(cyl), mpg))
pl + geom_boxplot()
```



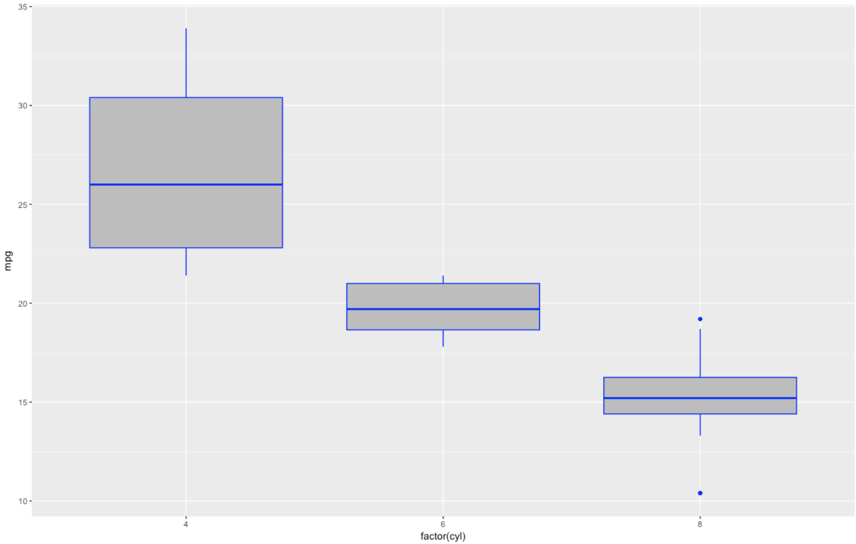
```
pl + geom_boxplot() + coord_flip()
```



```r
pl + geom_boxplot(aes(fill = factor(cyl)))
```

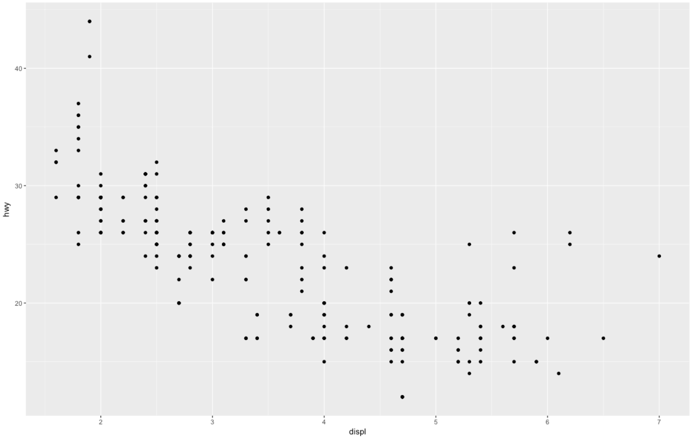


```
pl + geom_boxplot(fill = "grey", color = "blue")
```



Coordinates and Faceting with ggplot2

```r
library(ggplot2)
pl <- ggplot(mpg,aes(x=displ,y=hwy)) + geom_point()
pl
```

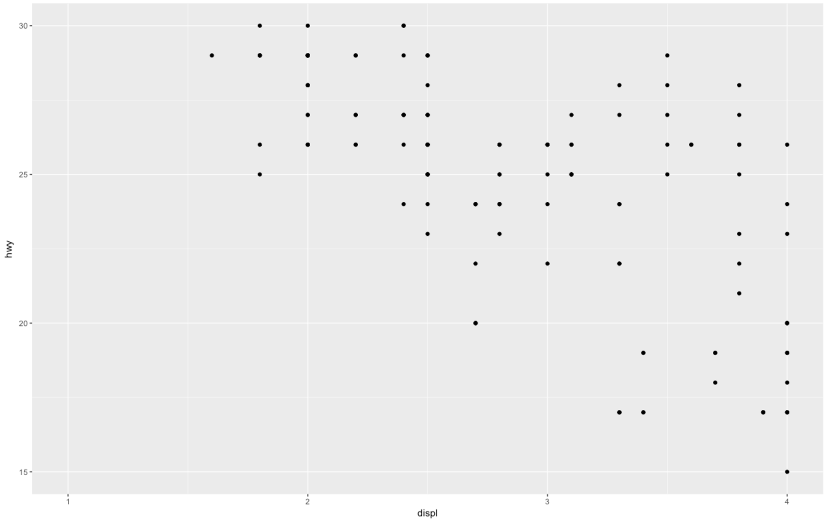


Setting x and y limits

You can use **+ scale_x_continuous** and **scale_y_continuous** with an additional **limits=c(low,high)** argument to set the scale.

A sometimes nicer way to do this is by adding **+ coord_cartesian()** with **xlim** and **ylim** arguments and pass in numeric vectors.

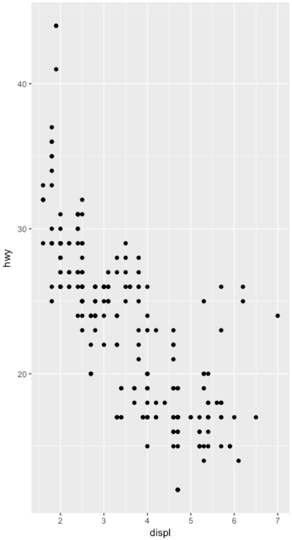
```
pl + coord_cartesian(xlim=c(1,4),ylim=c(15,30))
```


30
25
20
15
hwy
1
2
3
4
displ

Aspect Ratios

You can use the **coord_fixed()** method to change the aspect ratio of a plot (default is 1:1).

```
# aspect ratio, expressed as y / x
pl + coord_fixed(ratio = 1/3)
```

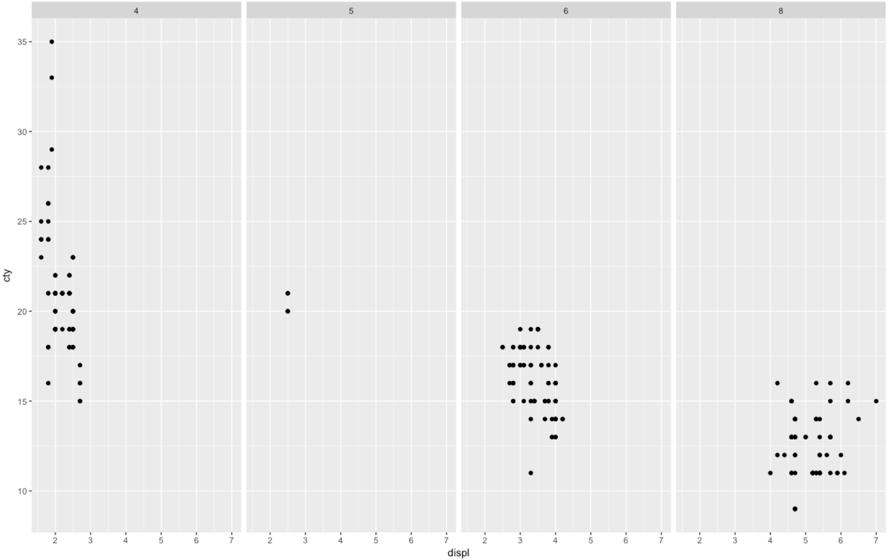

40
30
20
hwy
2
3
4
5
6
7
displ

Facets

The best way to set up a facet grid (multiple plots) is to use **facet_grid()**

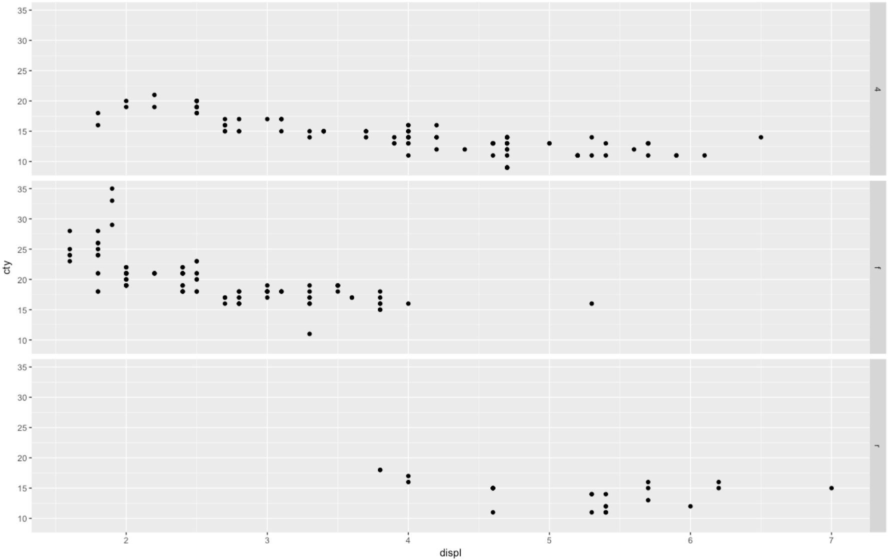
```
help(facet_grid)
```

```
p <- ggplot(mpg, aes(displ, cty)) + geom_point()

p + facet_grid(. ~ cyl)
```

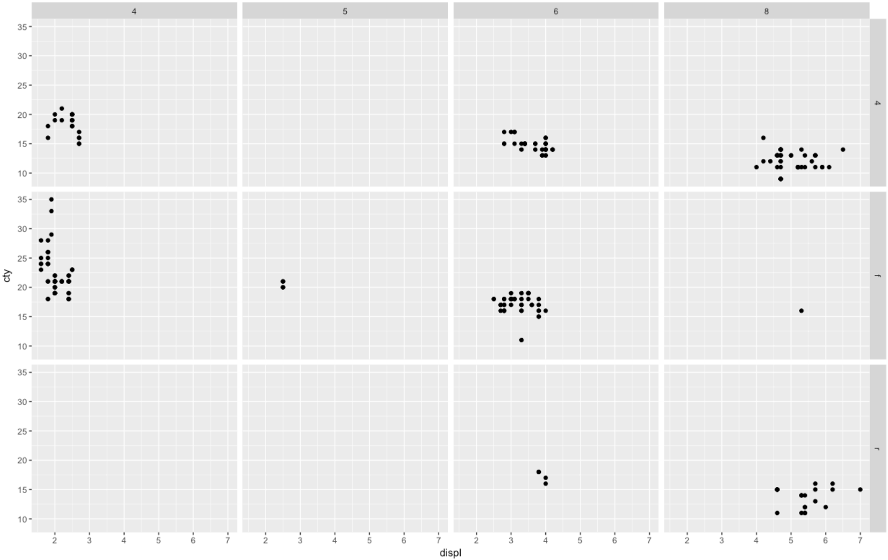


4
5
6
8
cty
35
30
25
20
15
10
2
3
4
5
6
7
displ

```r
p + facet_grid(drv ~ .)
```



```
p + facet_grid(drv ~ cyl)
```



Themes

There are a lot of built-in themes in ggplot and you can use them in two ways, by stating before your plot to set the theme:

```
theme_set(theme_bw())
```

or by adding them to your plot directly

```
my_plot + theme_bw()
```

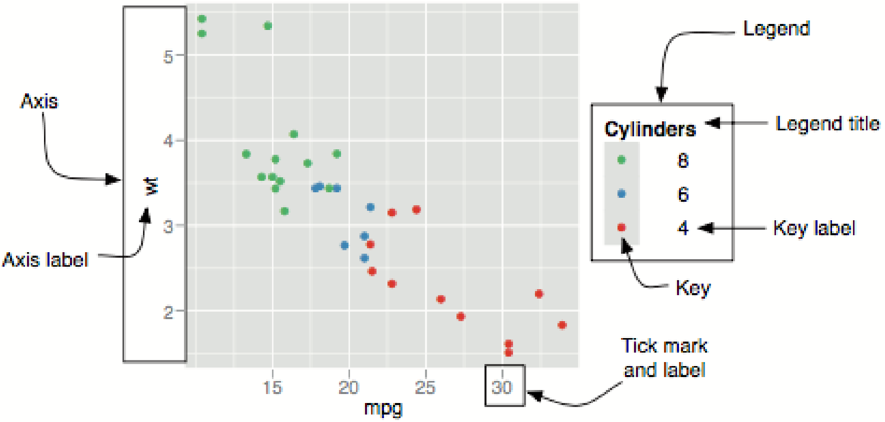
There is also a great library called **ggthemes** which adds even more built-in themes for ggplot. You can also customize your own themes

Themes elements

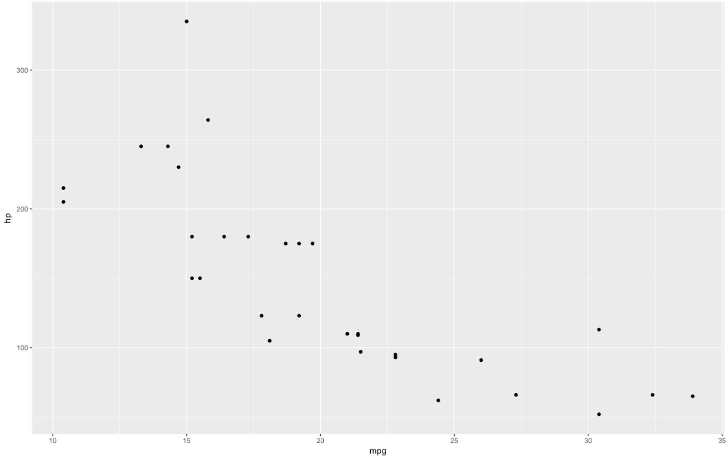
Theme element	Type	Description
`axis.line`	segment	line along axis
`axis.text.x`	text	x axis label
`axis.text.y`	text	y axis label
`axis.ticks`	segment	axis tick marks
`axis.title.x`	text	horizontal tick labels
`axis.title.y`	text	vertical tick labels
`legend.background`	rect	background of legend
`legend.key`	rect	background underneath legend keys
`legend.text`	text	legend labels
`legend.title`	text	legend name
`panel.background`	rect	background of panel
`panel.border`	rect	border around panel
`panel.grid.major`	line	major grid lines
`panel.grid.minor`	line	minor grid lines
`plot.background`	rect	background of the entire plot
`plot.title`	text	plot title
`strip.background`	rect	background of facet labels
`strip.text.x`	text	text for horizontal strips
`strip.text.y`	text	text for vertical strips

Legends and axes

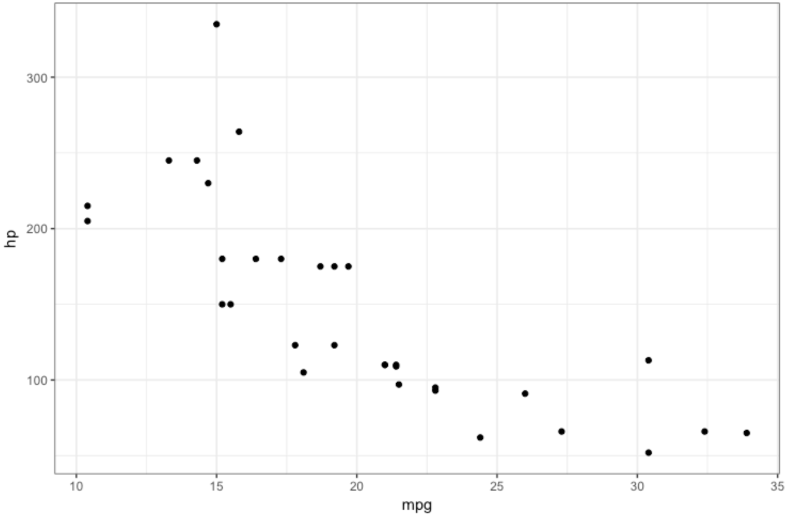
The components of the axes and legend



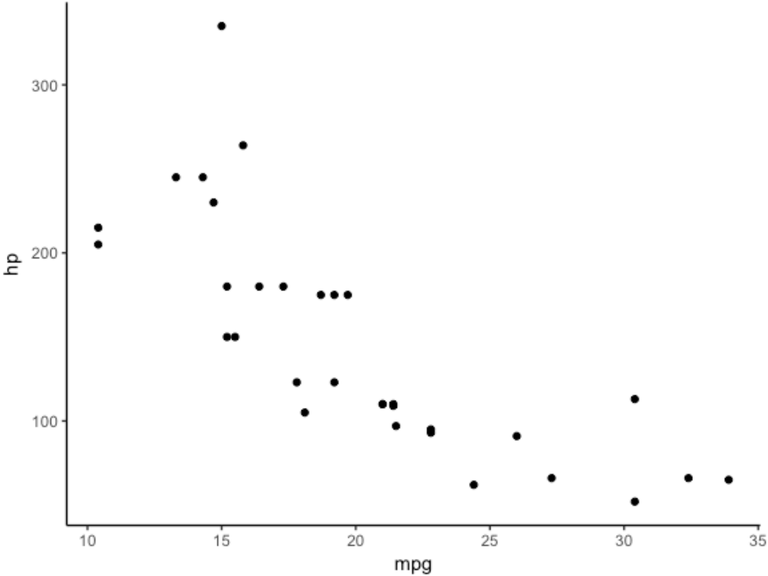
```r
library(ggplot2)
df <- mtcars
pl <- ggplot(df,aes(x=mpg,y=hp)) + geom_point()
print(pl)
```



```r
pl + theme_bw()
```



```
pl + theme_classic()
```



More Built-in Themes...

```
pl + theme_dark()

pl + theme_get()

pl + theme_light()

pl + theme_minimal()

pl + theme_void()
```

Text Mining Application

Natural Language Processing (NLP)

- Imaging you work for Google News and you want to group news articles by topic
- Or you work for a legal firm and you need to sift through thousands of pages of legal documents to find relevant ones

This is where NLP can help!

NLP

We will want to:

- Compile documents
- Featurize them
- Compare their features

NLP

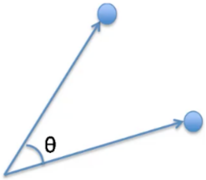
Simple Example:

- You have 2 documents:
  - "Blue House"
  - "Red House"
- Featurize based on word count:
  - "Blue House" $->$ (red,blue,house) $->$ (0,1,1)
  - "Red House" $->$ (red,blue,house) $->$ (1,0,1)

NLP

- A document represented as a vector of word counts is called a “Bag of Words”
  - “Blue House” $->$ (red,blue,house) $->$ (0,1,1)
  - “Red House” $->$ (red,blue,house) $->$ (1,0,1)
- You can use cosine similarity on the vectors made to determine similarity:

$$sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



NLP

- We can improve on Bag of Words by adjusting word counts based on their frequency in corpus (the group of all the documents)
- We can use TF-IDF (Term Frequency - Inverse Document Frequency)
- Term Frequency - Importance of the term within that document
  - TF(d,t) = Number of occurances of term **t** in document **d**
  - Inverse Document Frequency - Importance of the term in the corpus
    - IDF(t) = log(D/t) where D is total number of documents and t is number of documents with the term

NLP

Mathematically, TF-IDF is then expressed:

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

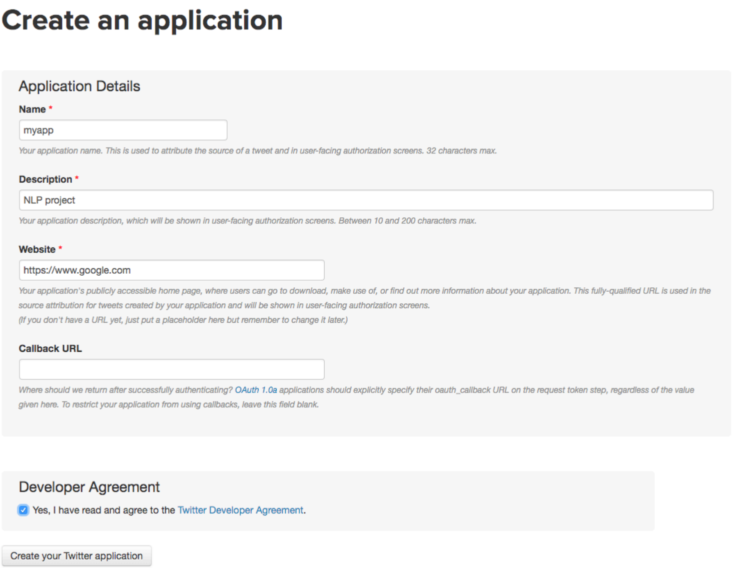
N = total number of documents

Text Mining Application with R

Necessary libraries

- tm
- twitterR
- wordcloud
- RColorBrewer
- e1017
- class

Create a Twitter App



Create an application

Application Details

Name *

myapp

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

NLP project

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

https://www.google.com

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

Yes, I have read and agree to the Twitter Developer Agreement.

Create your Twitter application

Create a Twitter App (cont.)

1. Create an Account on Twitter
2. Create a new app at: `https://apps.twitter.com/`
3. You may need to point it to a personal URL
4. Get Your Keys Under the Keys and Access Tokens tab

Regular Expression Review

grep() - Return the index location of pattern matches

```
grep('A', c('A','B','C','D','A'))

1 5
```

nchar() - Length of a string

```
nchar('helloworld')

10
```

gsub() - perform replacement of the matching patterns

```
gsub('pattern','replacement','hello have you seen the pattern here?')

'hello have you seen the replacement here?'
```

Text Manipulation

paste() - concatenate several strings together

```
print(paste('A','B','C',sep='...'))

[1] "A...B...C"
```

substr() - returns the substring in the given character range start:stop for the given

```
substr('abcdefg',start=2,stop = 5)

'bcde'
```

strsplit() - splits a string into a list of substrings based on another string split in x

```
strsplit('2016-01-23',split='-')

'2016' '01' '23'
```

Twitter Mining

Step 1: Import Libraries

```
library(twitteR)
library(tm)
library(wordcloud)
library(RColorBrewer)
```

Step 2: Search for Topic on Twitter

We'll use the twitteR library to data mine twitter. First you need to connect by setting up your Authorization keys and tokens.

```
setup_twitter_oauth(consumer_key, consumer_secret, access_token=NULL, access_secret=NULL)
```

We will search twitter for the term 'soccer'

```
soccer.tweets <- searchTwitter(``soccer", n=2000, lang=``en'')
soccer.text <- sapply(soccer.tweets, function(x) x$getText())
```

Twitter Mining (cont.)

Step 3: Clean Text Data

We'll remove emoticons and create a corpus

```
soccer.text <- iconv(soccer.text, 'UTF-8', 'ASCII') # remove emoticons
soccer.corpus <- Corpus(VectorSource(soccer.text)) # create a corpus
```

Step 4: Create a Document Term Matrix

We'll apply some transformations using the TermDocumentMatrix Function

```
term.doc.matrix <- TermDocumentMatrix(soccer.corpus, control = list(removePunctuation = TRUE,
     stopwords = c("soccer","http", stopwords("english")), removeNumbers = TRUE,tolower = TRUE
     ))
```

Twitter Mining (cont.)

Step 5: Check out Matrix

```
head(term.doc.matrix)
term.doc.matrix <- as.matrix(term.doc.matrix)
```

Step 6: Get Word Counts

```
word.freqs <- sort(rowSums(term.doc.matrix), decreasing=TRUE)
dm <- data.frame(word=names(word.freqs), freq=word.freqs)
```

Step 7: Create Word Cloud

```
wordcloud(dm$word, dm$freq, random.order=FALSE, colors=brewer.pal(8, "Dark2"))
```


jfcjustinbieber
great
via
new
sfa
will
girls
miss
sport
says
high
xzxsrqrff
music
vista
plays
every
freak
player
lot
get
play
life
last
still
now
today
footbalrlife
can
big
fan
best
justin
wya
fall
season
win
real
fun
think
game
like
mommy
video
world
playing
spend
players
much
four
day
hlkmietr
california
league
don
safa
love
man
know
post
row
team
march
playa
ref
tmz
first
fifa
cup
photo
ball
amp
tweet
duo
sex
red
matter
mind
qjiuqx
free
football
rio
basketball
boys
mls
youtube
let
got
idaaeeuaz
less
coach
final

rtweet package

You need to install an **rtweet** package

```
library(rtweet)

# whatever name you assigned to your created app
appname = "Zim1"
twitter_token = create_token(app = "Zim1", consumer_key = api_key, consumer_secret = api_secret
      )
tw = search_tweets("NorthKorea", n = 1200, token = twitter_token, lang = "en")
head(tw)
```

Example - Who is following whom?

```
library(rtweet)

## get user IDs of accounts followed by BBC
bbc_fds = get_friends("bbc")
## lookup data on those accounts
bbc_fds_data = lookup_users(bbc_fds$user_id)
head(bbc_fds_data)
## get user IDs of accounts following bbc
bbc_flw = get_followers("bbc", n = 1000)
## lookup data on those accounts
bbc_flw_data =lookup_users(bbc_flw$user_id)
head(bbc_flw_data)
## get user IDs of accounts followed by CNN
tmls = get_timelines(c("cnn", "BBCWorld", "foxnews"), n = 3200)
head(tmls)
tmls=as.data.frame(tmls)
head(tmls})
```

Facebook Mining

1. Get token from
   `https://developers.facebook.com/tools/explorer/`

2. Install a **Rfacebook** package

Example - Facebook

Search Group

```r
library(Rfacebook)
token=mytoken
ids <- searchGroup(name="rusers", token=token)
```

Search Page

```r
## search pages relating to Thailand
sp=searchPages("Thailand",token=token,n=15)
View(sp)
head(post)
```

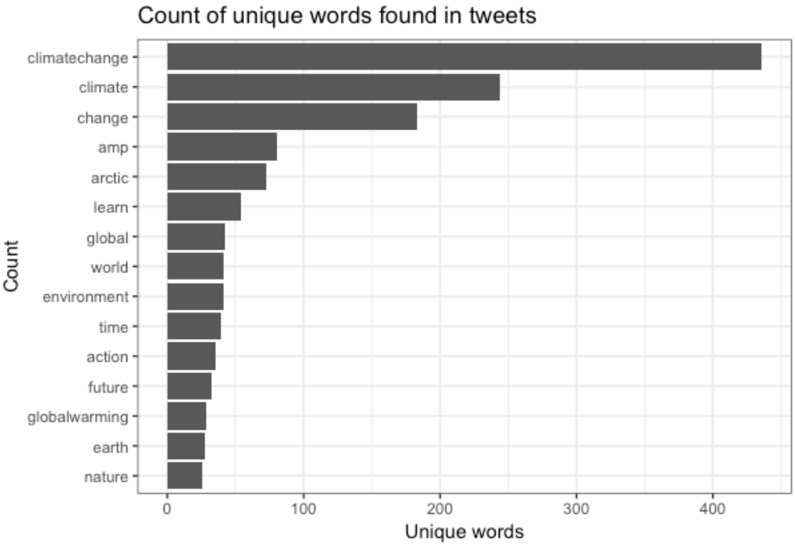
Example - Facebook (cont.)

Get Page and Posts

```
page = getPage(page="rbloggers", token=token, n=1000, since='2018/01/01', until='2018/03/31')
head(page)
post = getPost(post=page$id[1], n=12, token=token)
head(post)
```

Count Unique Tweets

```
library(rtweet)
library(dplyr)
library(tidytext)
library(ggplot2)

climate_tweets <- search_tweets(q = "#climatechange", n = 1000, lang = "en", include_rts =
      FALSE)
head(climate_tweets$text)
# remove http elements manually
climate_tweets$stripped_text <- gsub("http.*","",  climate_tweets$text)
climate_tweets$stripped_text <- gsub("https.*","", climate_tweets$stripped_text)
# remove punctuation, convert to lowercase, add id for each tweet!
climate_tweets_clean <- climate_tweets %>% dplyr::select(stripped_text) %>% unnest_tokens(word,
       stripped_text)
cc=climate_tweets_clean %>% anti_join(stop_words) ##remove stop words
head(cc)
# plot the top 15 words -- notice any issues?
top=cc %>% coun

top%>%
ggplot(aes(x = word, y = n)) +
geom_col() +
xlab(NULL) +
coord_flip() +
labs(x = "Count", y = "Unique words", title = "Count of unique words found in tweets") +
theme_bw()
```


Count of unique words found in tweets
climatechange
climate
change
amp
arctic
learn
global
world
environment
time
action
future
globalwarming
earth
nature
Count
0
100
200
300
400
Unique words

Web Mining

Read in data from HTML tables with XML

```
library(XML)
library(RCurl)

#Read in data from HTML tables with XML
url="https://en.wikipedia.org/wiki/2016_Summer_Olympics_medal_table"
##webpage we are intersted in

urldata <- getURL(url) #get data from this URL
data <- readHTMLTable(urldata, stringsAsFactors = FALSE)
#read the hHTML table

#medal tally
names(data)
head(data)
x=data$`2016 Summer Olympics medal table`
head(x)
```

Web Mining (cont.)

Cleaning Tables Extracted from Webpages

```
library(rvest)
library(stringr)
library(tidyr)

##Access the webpage with the tabular data
url = 'http://espn.go.com/nfl/superbowl/history/winners'
webpage =read_html(url)
sb_table = html_nodes(webpage, 'table')
sb = html_table(sb_table)[[1]] ##acces the first table on the page
head(sb)
## preliminary processing:remove the first two rows, and set the column names
sb = sb[-(1:2), ]#row,column
names(sb) = c("number", "date", "site", "result")
head(sb)
#divide between winner and losers
sb = separate(sb, result, c('winner', 'loser'), sep=', ', remove=TRUE)
head(sb)
## we split off the scores from the winner and loser columns.
##The function str_extract from the stringr package finds a
##substring matching a pattern
pattern =" \\d+$"
sb$winnerScore = as.numeric(str_extract(sb$winner, pattern))
sb$loserScore =as.numeric(str_extract(sb$loser, pattern))
sb$winner = gsub(pattern, "", sb$winner)
sb$loser =gsub(pattern, "", sb$loser)
head(sb)
```

Sentiment Analysis

- Sentiment = feelings (e.g., attitude, emotions, opinions)
- Subjective impressions, not facts
- Generally, a binary opposition in opinions is assumed
- For/against, like/dislike, good/bad, etc.
- Some sentiment analysis jargon: Semantic orientation, Polarity

What is Sentiment Analysis?

- Using NLP, statistics, or machine learning methods to extract, identify, or otherwise characterize the sentiment content of a text unit
- Sometimes referred to as **opinion mining**, although the emphasis in this case is on extraction

Questions SA might ask

- Is this product review positive or negative?
- Is this customer email satisfied or dissatisfied?
- Based on a sample of tweets, how are people responding to this ad campaign/product release/news item?
- How have bloggers' attitudes about the president changed since the election?

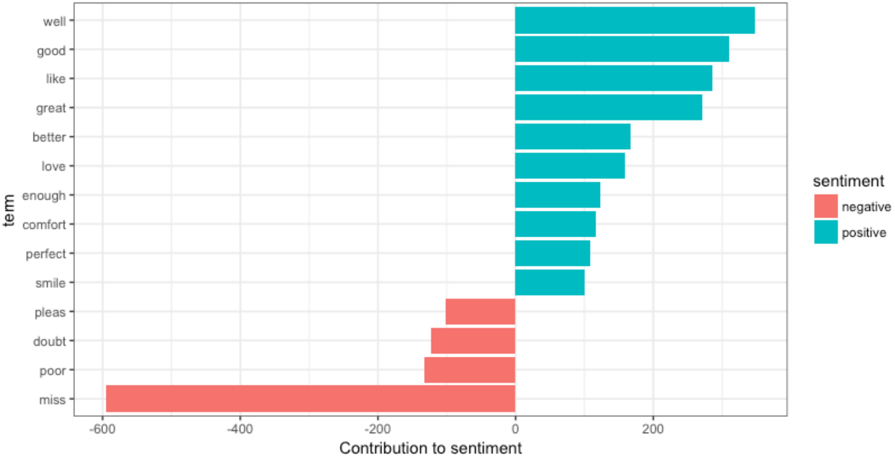
Sentiment Analysis with R

```
library(readr)
library(tm)
library(wordcloud)
s=read.csv("mugabe1.csv")
head(s)
names(s)

text <- as.character(s$text)
## carry out text data cleaning-gsub
some_txt<-gsub("(RT|via)((?:\\b\\w*@\\w+)+)","",s$text)
some_txt<-gsub("http[^[:blank:]]+","",some_txt)
some_txt<-gsub("@\\w+","",some_txt)
some_txt<-gsub("[[:punct:]]"," ",some_txt)
some_txt<-gsub("[^[:alnum:]]"," ",some_txt)
some_txt=as.character(some_txt)
library(syuzhet)
tweetSentiment <- get_nrc_sentiment(text)
#syuzhet pkg
#Calls the NRC sentiment dictionary to calculate the presence of
#eight different emotions and their corresponding valence in a text file.

barplot(sort(colSums(prop.table(tweetSentiment[, 1:8]))),  cex.names = 0.7,  las = 1,  main = "
      Emotions in Tweets text", xlab="Percentage")
```

Tidy Sentiments

```
##Tidy Sentiments
library(janeaustenr)
library(dplyr)
library(tm)
library(tidytext)
library(tidyverse)
library(qdapTools)
library(ggplot2)

austen_books_df=as.data.frame(austen_books(),stringsAsFactors=F)
head(austen_books_df)
head(austen_books_df)
summary(austen_books_df)
## isolate a book
emma=austen_books_df %>% group_by(book) %>% filter(book == "Emma")
head(emma)
```

```r
#########cleaning
corpus <- Corpus(VectorSource(emma$text))
corpus <- tm_map(corpus, removePunctuation)
corpus <- tm_map(corpus, content_transformer(tolower))
corpus <- tm_map(corpus, removeNumbers)
corpus <- tm_map(corpus, stripWhitespace)
corpus <- tm_map(corpus, removeWords, stopwords('english'))
corpus <- tm_map(corpus, stemDocument)

myDtm <- TermDocumentMatrix(corpus) #create a DTM

##require(tidytext)
terms=Terms(myDtm)
head(terms)

ap_td = tidy(myDtm) #convert DTM in a "tidy" form
ap_td
## sentiment analysis using tidy text
ap_sentiments <- ap_td %>%
inner_join(get_sentiments("bing"), by = c(term = "word"))

tail(ap_sentiments)

## which words contribute to positivity
ap_sentiments %>% count(sentiment, term, wt = count) %>% ungroup() %>% filter(n >= 100) %>%
      mutate(n = ifelse(sentiment == "negative", -n, n)) %>% mutate(term = reorder(term, n))
      %>% ggplot(aes(term, n, fill = sentiment)) +
geom_bar(stat = "identity") +
ylab("Contribution to sentiment") +
theme_bw()+
coord_flip() #horizontal barplot
```

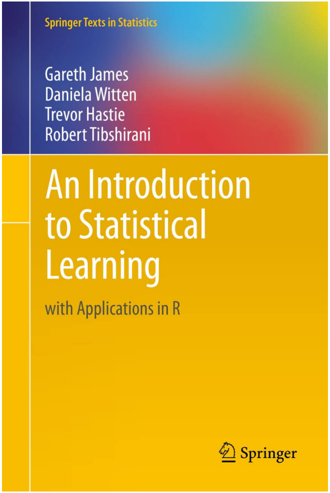


well
good
like
great
better
love
enough
comfort
perfect
smile
pleas
doubt
poor
miss
term
-600
-400
-200
0
200
Contribution to sentiment
sentiment
negative
positive

Machine Learning

Introduction to Machine Learning

We will be using **Introduction to Statistical Learning** by Gareth James as a companion book.



Companion Book

- Students who want the mathematical theory should do the reading
- Students who just want light theory and more interested in R application
- Read Chapter 1 and 2 to gain a background understanding the machine learning

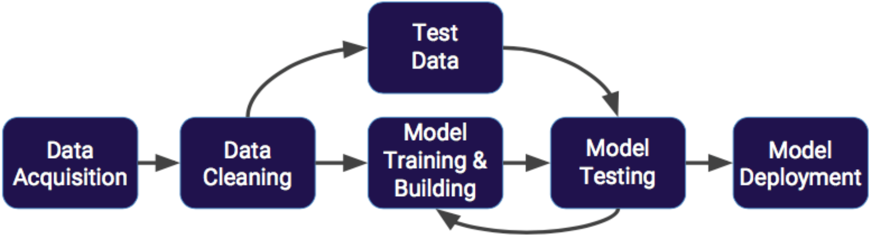
What is Machine Learning?

- Machine learning is a method of data analysis that automates analytical model building
- Using algorithms that iteratively learn from data, machine learning allows computers to find hidden insights without being explicitly programmed where to look

What is it used for?

- Fraud detection
- Web search results
- Real-time ads on web pages
- Credit scoring and next-best offers
- Prediction of equipment failures
- New pricing models
- Network intrusion detection
- Recommendation Engines
- Customer Segmentation
- Text Sentiment Analysis
- Predicting Customer Churn
- Pattern and image recognition
- Email spam filtering
- Financial Modeling

Machine Learning Process



Supervised Learning

- **Supervised learning** algorithms are trained using **labeled** examples, such as an input where the desired output is known
- For example, a piece of equipment could have data points labeled “F” (failed) or “R” (runs)
- The learning algorithm receives a set of inputs along with the corresponding correct outputs, and the algorithm learns by comparing its actual output with correct outputs to find errors
- It then modifies the model accordingly

Supervised Learning (cont.)

- Through methods like classification, regression, prediction and gradient boosting, supervise learning uses **patterns** to predict the values of the label on additional unlabeled data
- Supervised learning is commonly used in applications where **historical data predicts likely future events**
- For example, it can anticipate where credit card transactions are likely to be fraudulent or which insurance customer is likely to file a claim
- Or it can attempt to predict the price of a house based on different features for houses for which we have historical price data

Unsupervised Learning

- **Unsupervised learning** is used against data that has no historical labels
- The system is not told the "right answer." The algorithm must figure out what is being shown
- The goal is to explore the data and find some structure within
- Or it can find the main attributes that separate customer segments from each other
- Popular techniques include self-organizing maps, **nearest-neighbor mapping**, **k-means clustering** and singular value decomposition
- These algorithms are also used to segment text topics, recommend items and identify data outliers

Reinforcement Learning

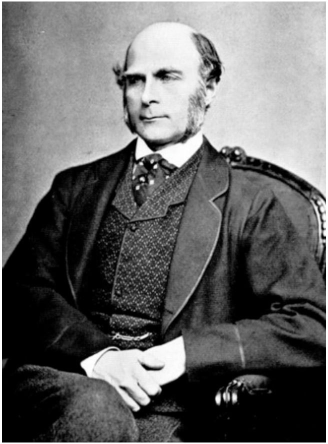
- **Reinforcement learning** is often used for robotics, gaming and navigation
- With reinforcement learning, the algorithm discovers through trial and error which actions yield the greatest rewards
- This type of learning has three primary components: the agent (the learner or decision maker), the environment (everything the agent interacts with) and actions (what the agent can do)
- The objective is for the agent to choose actions that maximize the expected reward over a given amount of time
- The agent will reach the goal much faster by following a good policy
- So the goal in reinforcement learning is to learn the best policy

Linear Regression

History

This all started in the 1800s with a guy named **Francis Galton. Galton** was studying the relationship between parents and their children. In particular, he investigated the relationship between the heights of fathers and their sons. However Galton's breakthrough was that the son's height tended to be closer to the overall average height of all people

What he discovered was that a man's son tended to be roughly as tall as his father.



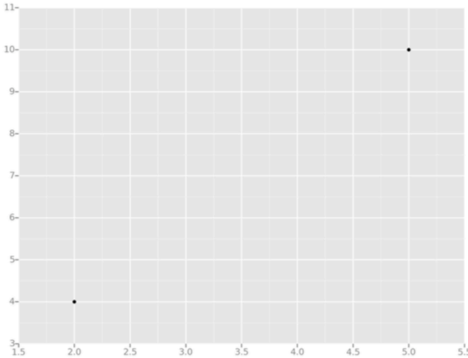
Example

Let's take **Shaquille O'Neal** as an example. Shaq is really tall:7ft 1in (2.2 meters). If Shaq has a son, chances are he'll be pretty tall too. However, Shaq is such an anomaly that there is also a very good chance that his son will be **not be as tall as Shaq**.

Turns out this is the case: Shaq's son is pretty tall (6 ft 7 in), but not nearly as tall as his dad. Galton called this phenomenon **regression**, as in "A father's son's height tends to regress (or drift towards) the mean (average) height."

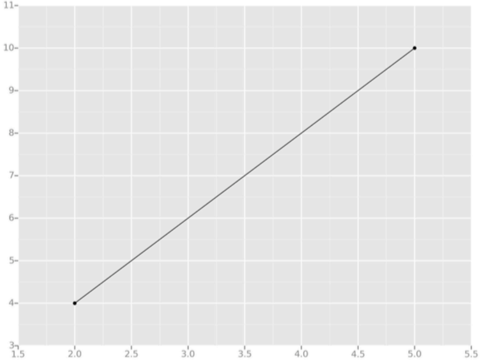
Example

Let's take the simplest possible example: calculating a regression with only 2 data points.Let's take the simplest possible example: calculating a regression with only 2 data points.



All we're trying to do when we calculate our regression line is draw a line that's as close to every dot as possible.

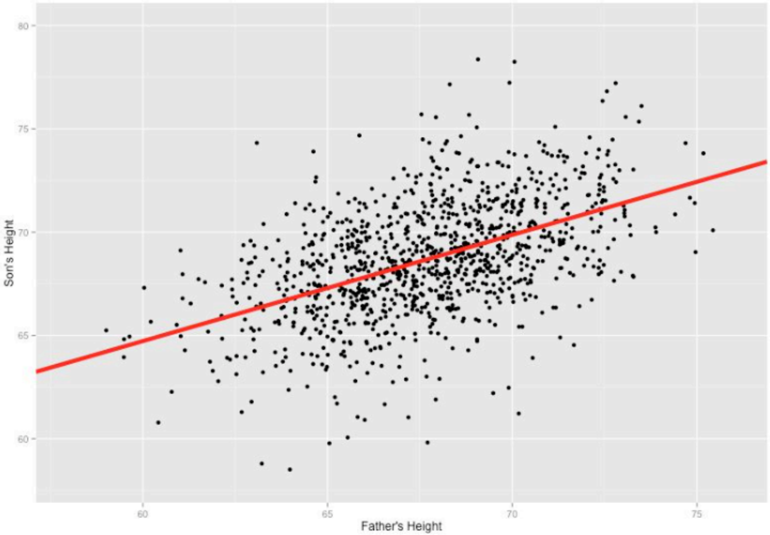
For classic linear regression, or "Least Squares Method", you only measure the closeness in the "up and down" direction



Now wouldn't it be great if we could apply this same concept to a graph with more than just two data points?

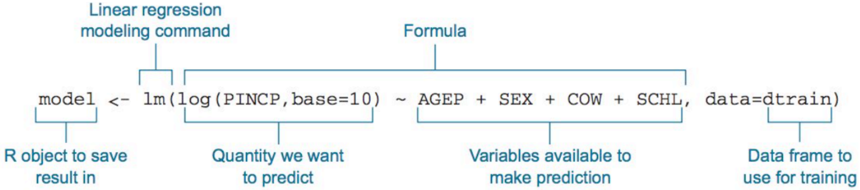
Our goal with linear regression is to **minimize the vertical distance** between all the data points and our line. So in determining the **best line**, we are attempting to minimize the distance between **all** the points and their distance to our line.

There are lots of different ways to minimize this, (sum of squared errors, sum of absolute errors, etc), but all these methods have a general goal of minimizing this distance.


80
75
70
65
60
Son's Height
60
65
70
75
Father's Height

Using R for Linear Regression

Formulas in R take the form (y $\sim$ x). To add more predictor vairables, just use + sign. i.e. (y $\sim$ x+z)



Linear
regression model

Prediction
command

Data to use
in prediction

```
dtest$predLogPINCP <- predict(model,newdata=dtest)
```

Store the prediction as a new
column named "predLogPINCP"

```
dtrain$predLogPINCP <- predict(model,newdata=dtrain)
```

Same operation on training data

Example - Linear Regression with R

Remember that Linear Regression is a supervised learning algorithm, meaning we'll have labeled data and try to predict new labels on unlabeled data. We'll explore some of the following concepts:

- Get our Data
- Exploratory Data Analysis (EDA)
- Clean our Data
- Review of Model Form
- Train and Test Groups
- Linear Regression Model

Get our Data

We will use the Student Performance Data Set from UC Irvine's Machine Learning Repository (student-mat.csv).

```
# Read CSV, note the delimiter (sep)
df <- read.csv('student-mat.csv',sep=';')
head(df)
```

Clean the Data

Next we have to clean this data. This data is actually already cleaned for you, But here are some things you may want to consider doing for other data sets:

Check for NA values

```
any(is.na(df))
FALSE
```

Categorical Features

Moving on, let's make sure that categorical variables have a factor set to them. For example, the MJob column refers to categories of Job Types, not some numeric value from 1 to 5. R is actually really good at detecting these sort of values and will take of this work for you a lot of the time, but always keep in mind the use of **factor()** as a possible. Luckily this is basically already, we can check this using the str() function:

```
str(df)
```

Building a Model

The general model of building a linear regression model in R look like this:

```
model <- lm(y $\sim$ x1 + x2, data)
```

or to use all the features in your data

```
model <- lm(y $\sim$ ., data) \#Uses all features
```

Train and Test Data

We'll need to split our data into a training set and a testing set in order to test our accuracy. We can do this easily using the **caTools** library:

```
# Import Library
library(caTools)
set.seed(101)

# Split up the sample, basically randomly assigns a booleans to a new column "sample"
sample <- sample.split(df$age, SplitRatio = 0.70) # SplitRatio = percent of sample==TRUE

# Training Data
train = subset(df, sample == TRUE)

# Testing Data
test = subset(df, sample == FALSE)
```

Training our Model

Let's train out model on our training data, then ask for a summary of that model:

```
model <- lm(G3 ~ ., train)
summary(model)
```

Model Interpretation

#	Name	Description
1	Residuals	The residuals are the difference between the actual values of the variable you're predicting and predicted values from your regression--$y - \hat{y}$. For most regressions you want your residuals to look like a normal distribution when plotted. If our residuals are normally distributed, this indicates the mean of the difference between our predictions and the actual values is close to 0 (good) and that when we miss, we're missing both short and long of the actual value, and the likelihood of a miss being far from the actual value gets smaller as the distance from the actual value gets larger. Think of it like a dartboard. A good model is going to hit the bullseye some of the time (but not everytime). When it doesn't hit the bullseye, it's missing in all of the other buckets evenly (i.e. not just missing in the 16 bin) and it also misses closer to the bullseye as opposed to on the outer edges of the dartboard.
2	Significance Stars	The stars are shorthand for significance levels, with the number of asterisks displayed according to the p-value computed. *** for high significance and * for low significance. In this case, *** indicates that it's unlikely that no relationship exists b/w absences and G3 scores.
3	Estimated Coeffecient	The estimated coefficient is the value of slope calculated by the regression. It might seem a little confusing that the Intercept also has a value, but just think of it as a slope that is always multiplied by 1. This number will obviously vary based on the magnitude of the variable you're inputting into the regression, but it's always good to spot check this number to make sure it seems reasonable.
4	Standard Error of the Coefficient Estimate	Measure of the variability in the estimate for the coefficient. Lower means better but this number is relative to the value of the coefficient. As a rule of thumb, you'd like this value to be at least an order of magnitude less than the coefficient estimate.
5	t-value of the Coefficient Estimate	Score that measures whether or not the coefficient for this variable is meaningful for the model. You probably won't use this value itself, but know that it is used to calculate the p-value and the significance levels.

Model Interpretation

6	Variable p-value	Probability the variable is *NOT* relevant. You want this number to be as small as possible. If the number is *really* small, R will display it in scientific notation.
7	Significance Legend	The more punctuation there is next to your variables, the better. Blank=bad, Dots=pretty good, Stars=good, More Stars=very good
8	Residual Std Error / Degrees of Freedom	The Residual Std Error is just the standard deviation of your residuals. You'd like this number to be proportional to the quantiles of the residuals in #1. For a normal distribution, the 1st and 3rd quantiles should be 1.5 +/- the std error. The Degrees of Freedom is the difference between the number of observations included in your training sample and the number of variables used in your model (intercept counts as a variable).
9	R-squared	Metric for evaluating the goodness of fit of your model. Higher is better with 1 being the best. Corresponds with the amount of variability in what you're predicting that is explained by the model. WARNING: While a high R-squared indicates good correlation, correlation does *not* always imply causation.
10	F-statistic & resulting p-value	Performs an F-test on the model. This takes the parameters of our model (in our case we only have 1) and compares it to a model that has fewer parameters. In theory the model with more parameters should fit better. If the model with more parameters (your model) doesn't perform better than the model with fewer parameters, the F-test will have a high p-value (probability *NOT* significant boost). If the model with more parameters is better than the model with fewer parameters, you will have a lower p-value. The DF, or degrees of freedom, pertains to how many variables are in the model. In our case there is one variable so there is one degree of freedom.

Looks like Absences, Farmrel, G1, and G2 scores are good predictors. With age and activities aslo possiblby contributing to a good model.

Predictions

Let's test our model by predicting on our testing set:

```
G3.predictions <- predict(model,test)
```

Now we can get the root mean squared error, a standardized measure of how off we were with our predicted values:

```
results <- cbind(G3.predictions,test$G3)
colnames(results) <- c('pred','real')
results <- as.data.frame(results)
```

Now let's take care of negative predictions! Lot's of ways to this, here's a more complicated way, but its a good example of creating a custom function for a custom problem:

```
to_zero <- function(x){
^^Iif  (x < 0){
^^I^^Ireturn(0)
^^I}else{
^^I^^Ireturn(x)
^^I}
}
```

```
results$pred <- sapply(results$pred,to_zero)
```

There's lots of ways to evaluate the prediction values, for example the MSE (mean squared error):

```
mse <- mean((results$real-results$pred)^2)
print(mse)
[1] 4.411405
^^I
```

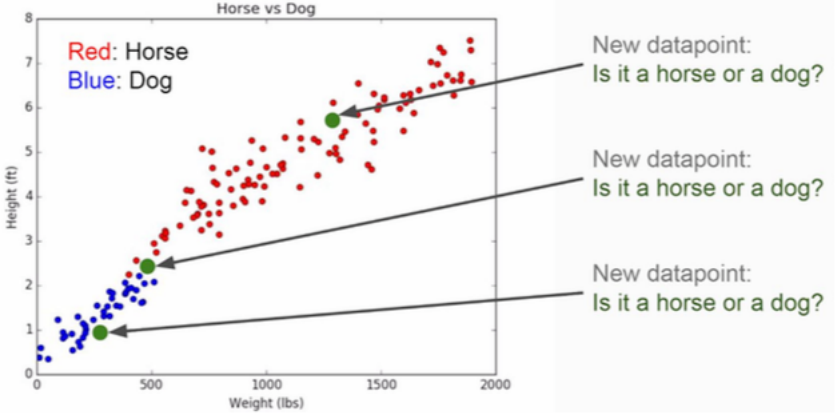
Or the root mean squared error:

```
mse^0.5
2.10033451255583
```

K Nearest Neighbors (KNN)

KNN

- K Nearest Neighbors is a **classification** algorithm that operates on a very simple principle
- Imagine we had some imaginary data on Dogs and Horses, with heights and weights



KNN (cont.)

Training Algorithm:

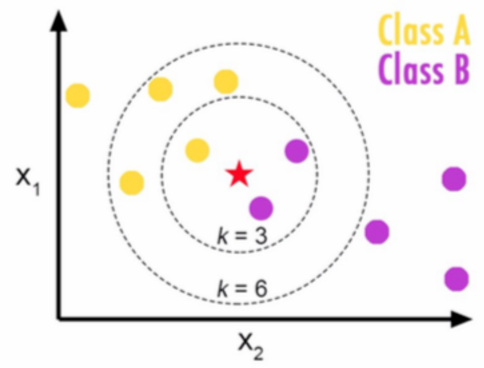
1. Store all the Data

Prediction Algorithm:

1. Calculate the distance from x to all points in your data
2. Sort the points in your data by increasing distance from x
3. Predict the majority label of the "k" closest points

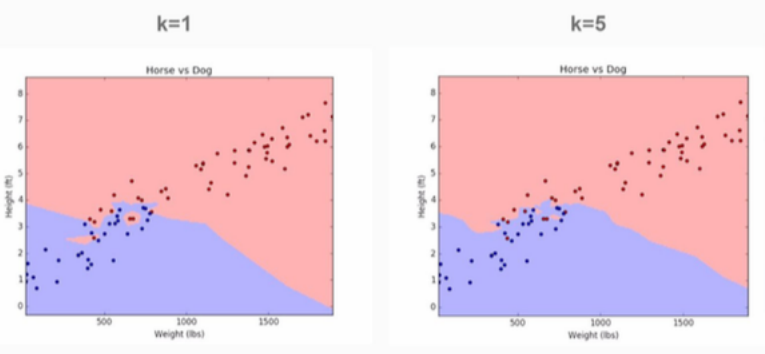
KNN (cont.)

Choosing a K will affect what class a new point is assigned to:



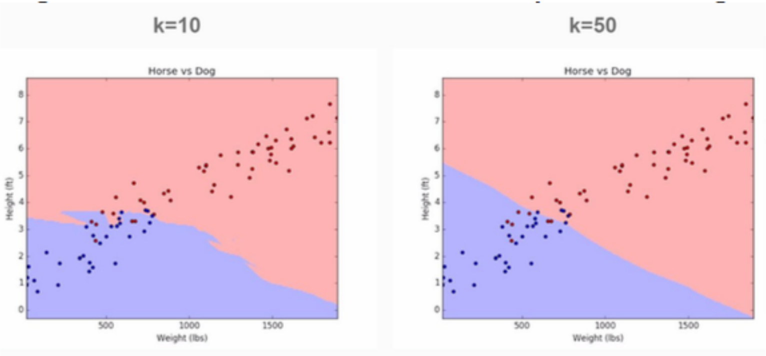
KNN (cont.)

Choosing a K will affect what class a new point is assigned to:



KNN (cont.)

Choosing a K will affect what class a new point is assigned to:



Pros

- Very simple
- Training is trivial
- Works with any number of classes
- Easy to add more data
- Few parameters
  - K
  - Distance Metric

Cons

- High Prediction Cost (worse for large data sets)
- Not good with high dimensional data
- Categorical Features don't work well

Example - K Nearest Neighbors

We'll use the **ISLR package** to get the data, you can download it with the code below. Remember to call the library as well.

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

We will apply the KNN approach to the Caravan data set, which is part of the ISLR library. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is Purchase, which indicates whether or not a given individual purchases a Caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

Let's look at the structure:

```
str(Caravan)
summary(Caravan$Purchase)
```

Cleaning Data

Let's just remove any NA values by dropping the rows with them.

```
any(is.na(Caravan))
```

Standardize Variables

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the distance between the observations, and hence on the KNN classifier, than variables that are on a small scale.

For example, let's check out the variance of two features:

```
var(Caravan[,1])
165.037847395189
var(Caravan[,2])
0.164707781931954
```

Clearly the scales are different! We are now going to standarize all the X variables except Y (Purchase). The Purchase variable is in column 86 of our dataset, so let's save it in a separate variable because the **knn()** function needs it as a separate argument.

```
# save the Purchase column in a separate variable
purchase <- Caravan[,86]

# Standarize the dataset using "scale()" R function
standardized.Caravan <- scale(Caravan[,-86])
```

Let's check the variance again:

```
var(standardized.Caravan[,1])
1
var(standardized.Caravan[,2])
1
```

We can see that now that all independent variables (X's) have a mean of 1 and standard deviation of 0. Great, then let's divide our dataset into testing and training data. We'll just do a simple split of the first 1000 rows as a test set:

```
# First 100 rows for test set
test.index <- 1:1000
test.data <- standardized.Caravan[test.index,]
test.purchase <- purchase[test.index]
```

```
# Rest of data for training
train.data <- standardized.Caravan[-test.index,]
train.purchase <- purchase[-test.index]
```

Using KNN

Rememeber that we are trying to come up with a model to predict whether someone will purchase or not. We will use the knn() function to do so, and we will focus on 4 of its arguments that we need to specify. The first argument is a data frame that contains the training data set(remember that we don't have the Y here), the second argument is a data frame that contains the testing data set (again no Y variable), the third argument is the train.purchase column (Y) that we save earlier, and the fourth argument is the k (how many neighbors). Let's start with k = 1. knn() function returns a vector of predicted Y's.

```
library(class)
set.seed(101)
predicted.purchase <- knn(train.data,test.data,train.purchase,k=1)
head(predicted.purchase)

No No No No No No
```

Now let's evaluate the model we trained and see our misclassification error rate.

```
mean(test.purchase != predicted.purchase)
0.116
```

Choosing K Value

Let's see what happens when we choose a different K value:

```
predicted.purchase <- knn(train.data,test.data,train.purchase,k=3)
mean(test.purchase != predicted.purchase)
0.073
```

Interesting! Our Misclassification rate went down! What about k=5?

```
predicted.purchase <- knn(train.data,test.data,train.purchase,k=5)
mean(test.purchase != predicted.purchase)
0.066
```

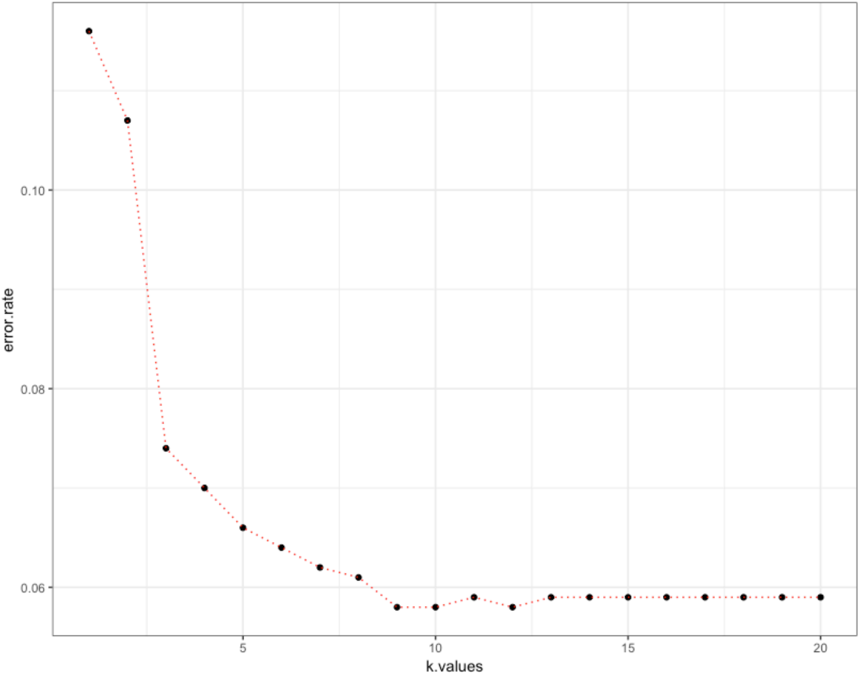
Should we manually change k and see which k gives us the minimal misclassification rate? NO! we have computers, so let's automate the process with a for() loop. A loop in R repeats the same command as much as you specify. For example, if we want to check for k =1 up to 100, then we have to write 3 x 100 lines of code, but with a for loop, you just need 4 lines of code, and you can repeat those 3 lines up to as many as you want. (Note this may take awhile because you're running the model 20 times!)

```
predicted.purchase = NULL
error.rate = NULL

for(i in 1:20){
set.seed(101)
predicted.purchase = knn(train.data,test.data,train.purchase,k=i)
error.rate[i] = mean(test.purchase != predicted.purchase)
}
print(error.rate)
```

Elbow Method

We can plot out the various error rates for the K values. We should see an **"elbow"** indicating that we don't get a decrease in error rate for using a higher K. This is a good cut-off point:

```
library(ggplot2)
k.values <- 1:20
error.df <- data.frame(error.rate,k.values)
ggplot(error.df,aes(x=k.values,y=error.rate)) + geom_point()+ geom_line(lty="dotted",color='red
    ') + theme_bw()
```


error.rate
0.10
0.08
0.06
5
10
15
20
k.values

Tree Methods

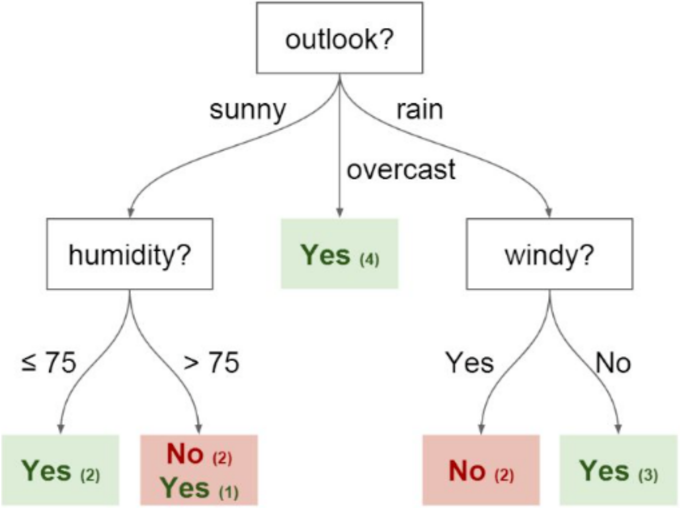
Tree Methods

Imagine that I play Tennis every Saturday and I always invite a friend to come with me. Sometimes my friend shows up, sometimes not. For him it depends on a variety of factors, such as: weather, temperature, humidity, wind etc..

I start keeping track of these features and whether or not he showed up to play with me

Temperature	Outlook	Humidity	Windy	Played?
Mild	Sunny	80	No	Yes
Hot	Sunny	75	Yes	**No**
Hot	Overcast	77	No	Yes
Cool	Rain	70	No	Yes
Cool	Overcast	72	Yes	Yes
Mild	Sunny	77	No	**No**
Cool	Sunny	70	No	Yes
Mild	Rain	69	No	Yes
Mild	Sunny	65	Yes	Yes
Mild	Overcast	77	Yes	Yes
Hot	Overcast	74	No	Yes
Mild	Rain	77	Yes	**No**
Cool	Rain	73	Yes	**No**
Mild	Rain	78	No	Yes

I want to use this data to predict whether or not he will show up to play. An intuitive way to do this is through a Decision Tree.



In this tree we have:

- Nodes - Split for the value of a certain attribute
- Edges - Outcome of a split to next node
- Root - The node that performs the first split
- Leaves - Terminal nodes that predict the outcome

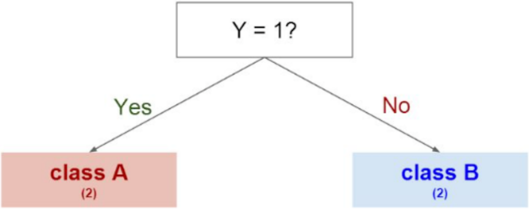
Intuition Behind Splits

Imaginary Data with 3 features (X,Y, and Z) with two possible classes.

X	Y	Z	Class
1	1	1	A
1	1	0	A
0	0	1	B
1	0	0	B

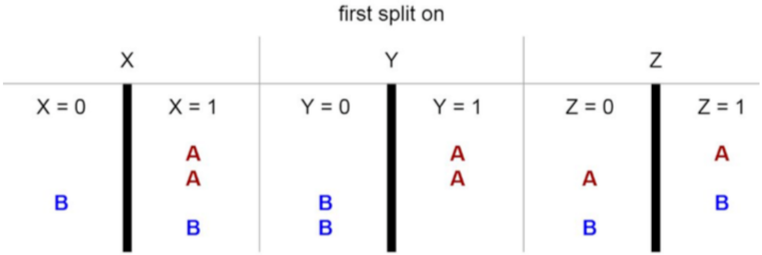
Intuition Behind Splits (cont.)

Splitting on Y gives us a clear separation between classes



Intuition Behind Splits (cont.)

We could have also tried splitting on other features first:



Random Forests

To improve performance, we can use many trees with a random sample of features chosen as the split.

- A new random sample of features is chosen for **every single tree at every single split**
- For **classification**, m is typically chosen to be the square root of p.

Random Forests

What's the point?

- Suppose there is **one very strong feature** in the data set. When using "bagged" trees, most of the trees will use that feature as the top split, resulting in an ensemble of similar trees that are **highly correlated**
- Averaging highly correlated quantities does not significantly reduce variance
- By randomly leaving out candidate features from each split, **Random Forests "decorrelates" the tree,** such that the averaging process can reduce the variance of the resulting model

Decision Trees and Random Forests

You may need to install the **rpart** library.

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

We can then use the **rpart()** function to build decision tree model:
rpart(formula, data=, method=,control=) where

- the formula is in the format: outcome $\sim$ predictor1+predictor2+predictor3+ect.
- data= specifies the data frame
- method= "class" for a classification tree
- "anova" for a regression tree
- control= optional parameters for controlling tree growth

Sample Data

We'll use the **kyphosis** data frame which has 81 rows and 4 columns. representing data on children who have had corrective spinal surgery. It has the following columns:

- Kyphosis-a factor with levels absent present indicating if a kyphosis (a type of deformation) was present after the operation.
- Age-in months
- Number-the number of vertebrae involved
- Start-the number of the first (topmost) vertebra operated on.

```
tree <-  rpart(Kyphosis ~ . , method='class', data= kyphosis)
```

Examining Results of the Tree Model

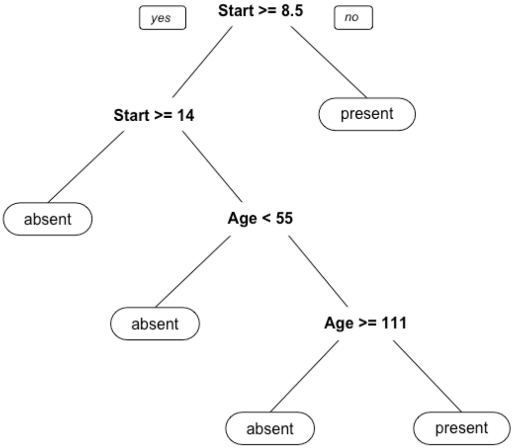
printcp(*fit*)	display cp table
plotcp(*fit*)	plot cross-validation results
rsq.rpart(*fit*)	plot approximate R-squared and relative error for different splits (2 plots). labels are only appropriate for the "anova" method.
print(*fit*)	print results
summary(*fit*)	detailed results including surrogate splits
plot(*fit*)	plot decision tree
text(*fit*)	label the decision tree plot
post(*fit*, file=)	create postscript plot of decision tree

```r
printcp(tree)
```

```
Classification tree:
rpart(formula = Kyphosis ~ ., data = kyphosis, method = "class")

Variables actually used in tree construction:
[1] Age   Start

Root node error: 17/81 = 0.20988

n= 81

        CP nsplit rel error xerror    xstd
1 0.176471      0   1.00000      1 0.21559
2 0.019608      1   0.82353      1 0.21559
3 0.010000      4   0.76471      1 0.21559
```

Tree Visualization

```
#install.packages('rpart.plot')
library(rpart.plot)
prp(tree)
```



Random Forests

Random forests improve predictive accuracy by generating a large number of bootstrapped trees (based on random samples of variables), classifying a case using each tree in this new "forest", and deciding a final predicted outcome by combining the results across all of the trees (an average in regression, a majority vote in classification).

```
install.packages("randomForest")
library(randomForest)
model <- randomForest(Kyphosis ~ .,   data=kyphosis)
print(model) # view results
```

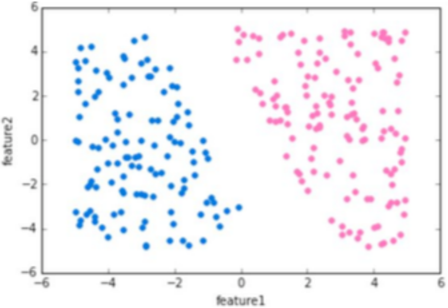
Support Vector Machines

Support Vector Machines

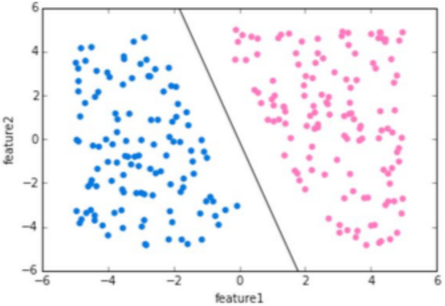
- Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis
- An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible
- New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on

Support Vector Machines

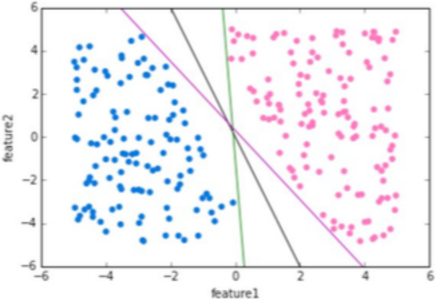
Let's show the basic intuition behind SVMs. Imagine the labeled training data below:



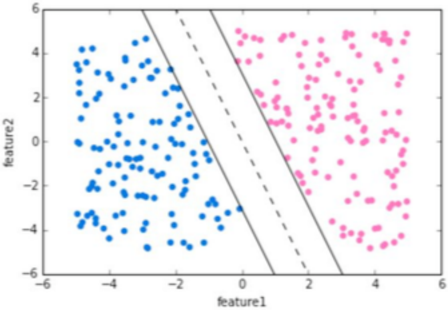
We can draw a separating “hyperplane” between the classes



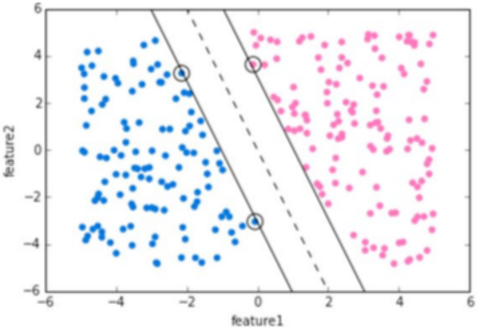
But we have many options of hyperplanes that separate perfectly...



We would like to choose a hyperplane that maximizes the margin between classes



The vector points that the margin lines touch Support Vectors



Example - Support Vector Machine

Building the Model

We'll need the **e1071** library

```r
install.packages("e1071")
library(e1071)

model <- svm(Species ~ ., data=iris)
summary(model)
```

```
Call:
svm(formula = Species ~ ., data = iris)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.25

Number of Support Vectors:  51

 ( 8 22 21 )


Number of Classes:  3

Levels:
 setosa versicolor virginica
```

Example Predictions

We have a small data set, so instead of splitting it into training and testing sets (which you should always try to do!) we'll just score out model against the same data it was tested against:

```
predicted.values <- predict(model,iris[1:4])
table(predicted.values,iris[,5])

predicted.values setosa versicolor virginica
setosa          50          0         0
versicolor       0         48         2
virginica        0          2        48
```

Advanced - Tuning

We can try to tune parameters to attempt to improve our model, you can refer to the help() documentation to understand what each of these parameters stands for. We use the tune function:

```
# Tune for combos of gamma 0.5,1,2
# and costs 1/10 , 10 , 100
tune.results <- tune(svm,train.x=iris[1:4],train.y=iris[,5],kernel='radial', ranges=list(cost
     =10^(-1:2), gamma=c(.5,1,2)))
summary(tune.results)
```

```
Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 cost gamma
    1   0.5

- best performance: 0.03333333

- Detailed performance results:
    cost gamma      error dispersion
1    0.1   0.5 0.07333333 0.07981460
2    1.0   0.5 0.03333333 0.06478835
3   10.0   0.5 0.04000000 0.04661373
4  100.0   0.5 0.05333333 0.06885304
5    0.1   1.0 0.04666667 0.06324555
6    1.0   1.0 0.05333333 0.06126244
7   10.0   1.0 0.06000000 0.05837300
8  100.0   1.0 0.06000000 0.05837300
9    0.1   2.0 0.07333333 0.08577893
10   1.0   2.0 0.05333333 0.06126244
11  10.0   2.0 0.04666667 0.04499657
12 100.0   2.0 0.04000000 0.04661373
```

We can now see that the best performance occurs with cost=1 and gamma=0.5. You could try to train the model again with these specific parameters in hopes of having a better model:

```
tuned.svm <- svm(Species ~ ., data=iris, kernel="radial", cost=1, gamma=0.5)
summary(tuned.svm)
tuned.predicted.values <- predict(tuned.svm,iris[1:4])
table(tuned.predicted.values,iris[,5])

tuned.predicted.values setosa versicolor virginica
setosa         50          0         0
versicolor      0         48         2
virginica       0          2        48
```

Looks like we weren't able to improve on our model! The concept of trying to tune for parameters by just trying many combinations in generally known as a grid search. In this case, we likely have too little data to actually improve our model through careful parameter selection.

K Means Clustering

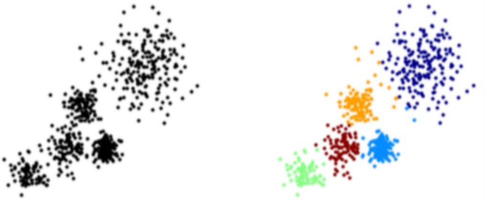
K Means Clustering

K Means Clustering is an unsupervised learning algorithm that will attempt to group similar clusters together in your data.

So what does a typical clustering problem look like?

- Cluster similar documents
- Cluster customers based on features
- Market segmentation
- Identify similar physical groups

The overall goal is to divide data into distinct groups such that observations within each group are similar



K Means Clustering Algorithm

- Choose a number of Cluster "K"
- Randomly assign each point to a cluster
- Until clusters stop changing, repeat the following:
  - For each cluster, compute the cluster centroid by taking the mean vector of points in the cluster
  - Assign each data point to the cluster for which the centroid is the closest

Choosing a K value

- There is no easy answer for choosing a “best” K value
- One way is the elbow method

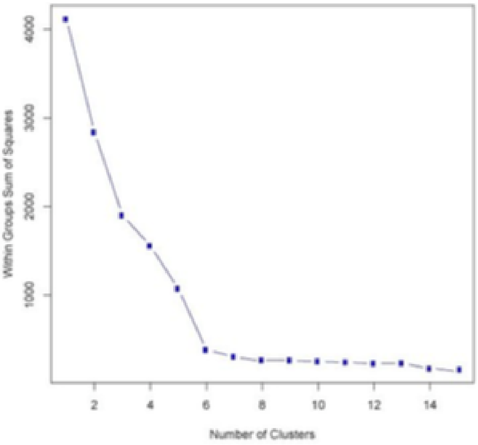
First of all, compute the sum of squared error (SSE) for some values of k (for example 2,4,6,8, etc.)

The SSE is defined as the sum of the squared distance between each member of the cluster and its centroid

Choosing a K value (cont.)

- If you plot k against the SSE, you will see that **the error decreases as k gets larger;** this is because when the number of cluster increases, they should be smaller, so distortion is also smaller
- The idea of the elbow method is to choose the k at which the SSE decreases abruptly
- This produces an "elbow effect"in the graph, as you can see in the following picture:

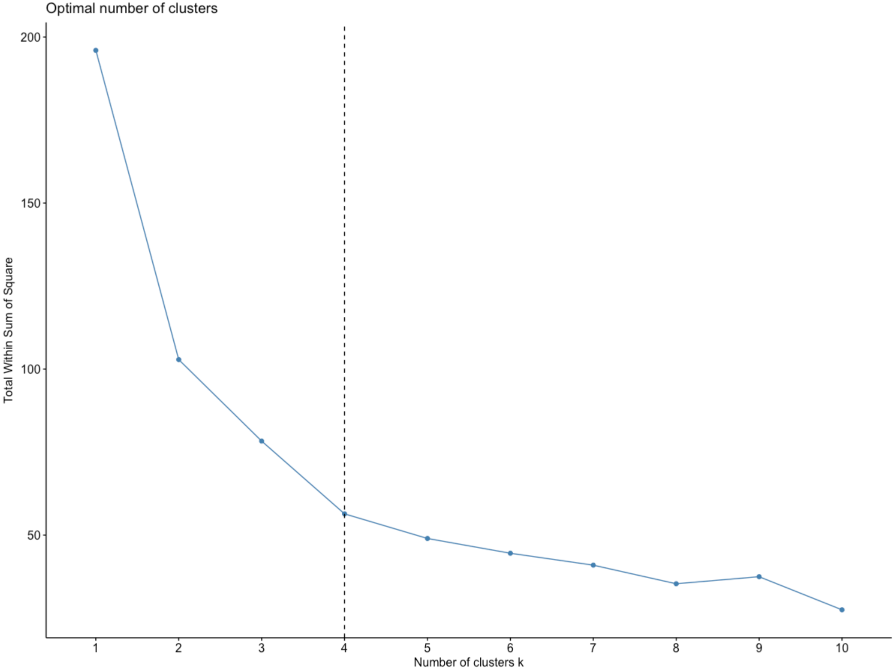
Note: Generally, K must be provided by the user.


Within Groups Sum of Squares
1000
2000
3000
4000
2
4
6
8
10
12
14
Number of Clusters

Choosing a K value (cont.)

The R function **fviz_nbclust()** [in factoextra package] provides a convenient solution to estimate the optimal number of clusters.

```
library(factoextra)
data("USArrests")      # Loading the data set
df <- scale(USArrests) # Scaling the data
fviz_nbclust(df, kmeans, method = "wss") + geom_vline(xintercept = 4, linetype = 2)
```



Optimal number of clusters
Total Within Sum of Square
200
150
100
50
1
2
3
4
5
6
7
8
9
10
Number of clusters k

Example - K Means Clustering

Usually when dealing with an unsupervised learning problem, its difficult to get a good measure of how well the model performed. For this example, we will use data from the UCI archive based off of red and white wines (this is a very commonly used data set in ML).

We will then add a label to the a combined data set, we'll bring this label back later to see how well we can cluster the wine into groups.

Data: `http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/`

```r
df1 <- read.csv('winequality-red.csv',sep=';')
df2 <- read.csv('winequality-white.csv',sep=';')
```

Now add a label column to both df1 and df2 indicating a label 'red' or 'white'.

```r
# Using sapply with anon functions
df1$label <- sapply(df1$pH,function(x){'red'})
df2$label <- sapply(df2$pH,function(x){'white'})
```

Combine df1 and df2 into a single data frame called wine

```r
wine <- rbind(df1,df2)
```

Building the Clusters

```
wine.cluster <- kmeans(wine[1:12], 2)
print(wine.cluster$centers)
print(wine.cluster$cluster)
```

Evaluating the Clusters

You usually won't have the luxury of labeled data with KMeans, but let's go ahead and see how we did! Use the table() function to compare your cluster results to the real results. Which is easier to correctly group, red or white wines?

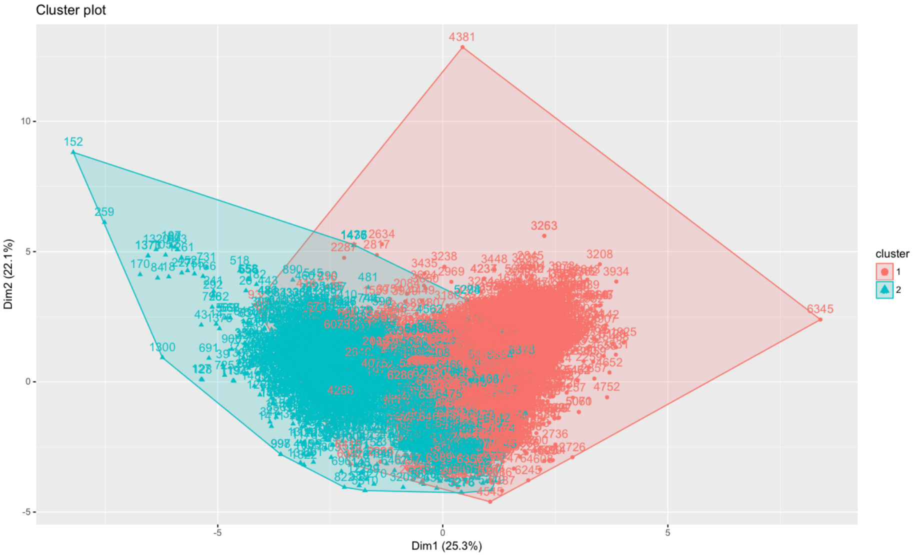
```
table(wine$label,wine.cluster$cluster)

          1    2
red    1515   84
white 1310 3588
```

We can see that red is easier to cluster together. There seems to be a lot of noise with white wines, this could also be due to "Rose" wines being categorized as white wine, while still retaining the qualities of a red wine.

It's important to note here, that **K-Means can only give you the clusters, it can't directly tell you what the labels should be, or even how many clusters you should have**, we are just lucky to know we expected two types of wine. This is where domain knowledge really comes into play.

We can also view our results by using **fviz_cluster**. This provides a nice illustration of the clusters. If there are more than two dimensions (variables) fviz_cluster will perform principal component analysis (PCA) and plot the data points according to the first two principal components that explain the majority of the variance.

```r
library(factoextra)
fviz_cluster(wine.cluster, data = wine[1:12])
```


Cluster plot
Dim1 (25.3%)
Dim2 (22.1%)
cluster
1
2

Exercise

Linear Regression Project

For this project you will be doing the **Bike Sharing Demand Kaggle challenge** (`https://www.kaggle.com/c/bike-sharing-demand/data`). You must predict the total count of bikes rented during each hour.

The data has the following features:

- datetime - hourly date + timestamp
- season - 1 = spring, 2 = summer, 3 = fall, 4 = winter
- holiday - whether the day is considered a holiday
- workingday - whether the day is neither a weekend nor holiday

- weather :
    1. Clear, Few clouds, Partly cloudy, Partly cloudy
    2. Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
    3. Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
    4. Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp - temperature in Celsius
- atemp - ”feels like” temperature in Celsius
- humidity - relative humidity
- windspeed - wind speed
- casual - number of non-registered user rentals initiated
- registered - number of registered user rentals initiated
- count - number of total rentals

Support Vector Machines Project

For this project we will be exploring publicly available data from LendingClub.com. Lending Club connects people who need money (borrowers) with people who have money (investors). Hopefully, as an investor you would want to invest in people who showed a profile of having a high probability of paying you back. We will try to create a model that will help predict this.

Lending club had a very interesting year in 2016, so let's check out some of their data and keep the context in mind. This data is from before they even went public.

We will use lending data from 2007-2010 and be trying to classify and predict whether or not the borrower paid back their loan in full. You can download the data from here
`https://www.lendingclub.com/info/download-data.action`

Here are what the columns represent:

- credit.policy: 1 if the customer meets the credit underwriting criteria of LendingClub.com, and 0 otherwise.
- purpose: The purpose of the loan (takes values "credit_card", "debt_consolidation", "educational", "major_purchase", "small_business", and "all_other").
- int.rate: The interest rate of the loan, as a proportion (a rate of 11% would be stored as 0.11). Borrowers judged by LendingClub.com to be more risky are assigned higher interest rates.
- installment: The monthly installments owed by the borrower if the loan is funded.
- log.annual.inc: The natural log of the self-reported annual income of the borrower.
- dti: The debt-to-income ratio of the borrower (amount of debt divided by annual income).
- fico: The FICO credit score of the borrower.

- days.with.cr.line: The number of days the borrower has had a credit line.
- revol.bal: The borrower's revolving balance (amount unpaid at the end of the credit card billing cycle).
- revol.util: The borrower's revolving line utilization rate (the amount of the credit line used relative to total credit available).
- inq.last.6mths: The borrower's number of inquiries by creditors in the last 6 months.
- delinq.2yrs: The number of times the borrower had been 30+ days past due on a payment in the past 2 years.
- pub.rec: The borrower's number of derogatory public records (bankruptcy filings, tax liens, or judgments).

Tree Methods Project

For this project we will be exploring the use of tree methods to classify schools as Private or Public based off their features.

Let's start by getting the data which is included in the **ISLR** library, the **College** data frame.

A data frame with 777 observations on the following 18 variables.

- Private A factor with levels No and Yes indicating private or public university
- Apps Number of applications received
- Accept Number of applications accepted
- Enroll Number of new students enrolled
- Top10perc Pct. new students from top 10
- Top25perc Pct. new students from top 25
- F.Undergrad Number of fulltime undergraduates
- P.Undergrad Number of parttime undergraduates
- Outstate Out-of-state tuition

- Room.Board Room and board costs
- Books Estimated book costs
- Personal Estimated personal spending
- PhD Pct. of faculty with Ph.D.'s
- Terminal Pct. of faculty with terminal degree
- S.F.Ratio Student/faculty ratio
- perc.alumni Pct. alumni who donate
- Expend Instructional expenditure per student
- Grad.Rate Graduation rate