

Fork me on GitHub

# Data Science Specialization Course Notes

Notes for all 9 courses in the **Coursera Data Science Specialization** from Johns Hopkins University

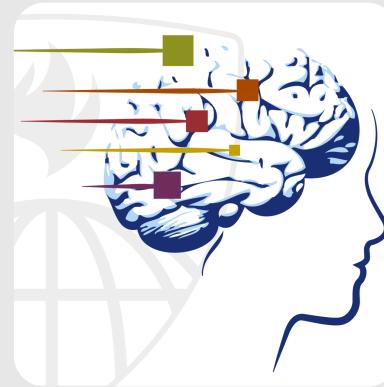
## Happy Learning

SHARE

All notes are written in R Markdown format and encompass all concepts covered in the **Data Science Specialization**, as well as additional examples and materials I compiled from lecture, my own exploration, StackOverflow, and Khan Academy.

They are by no means perfect, but feel free to [follow](#), [fork](#) and/or [contribute](#). Please reach out to [s.xing@me.com](mailto:s.xing@me.com) if you have any questions.

All files are hosted in [this](#) repository.



Course	HTML	PDF	Rmd
The Data Scientist's Toolbox			
R Programming			
Getting and Cleaning Data			
Exploratory Data Analysis			
Reproducible Research			
Statistical Inference			
Regression Models			
Practical Machine Learning			
Developing Data Products			



Xing graduated from Duke University in 2013, worked in consulting in NYC for 16 months, and moved to SF to pursue opportunities in product and data science. You can find him on [LinkedIn](#), [Github](#), or through [s.xing@me.com](mailto:s.xing@me.com).

# Data Scientist's Toolbox Course Notes

*Xing Su*

## Contents

CLI (Command Line Interface) . . . . .	2
GitHub . . . . .	2
Markdown . . . . .	2
R Packages . . . . .	3
Types of Data Science Questions . . . . .	3
Data . . . . .	3
Experimental Design . . . . .	3

## CLI (Command Line Interface)

- `/` = root directory
- `~` = home directory
- `pwd` = print working directory (current directory)
- `clear` = clear screen
- `ls` = list stuff
  - `-a` = see all (hidden)
  - `-l` = details
- `cd` = change directory
- `mkdir` = make directory
- `touch` = creates an empty file
- `cp` = copy
  - `cp <file> <directory>` = copy a file to a directory
  - `cp -r <directory> <newDirectory>` = copy all documents from directory to new Directory \*
  - `-r` = recursive
- `rm` = remove
  - `-r` = remove entire directories (no undo)
- `mv` = move
  - `move <file> <directory>` = move file to directory
  - `move <fileName> <newName>` = rename file
- `echo` = print arguments you give/variables
- `date` = print current date

## GitHub

- **Workflow**
  1. make edits in workspace
  2. update index/add files
  3. commit to local repo
  4. push to remote repository
- `git add .` = add all new files to be tracked
- `git add -u` = updates tracking for files that are renamed or deleted
- `git add -A` = both of the above
  - *Note: add is performed before committing*
- `git commit -m "message"` = commit the changes you want to be saved to the local copy
- `git checkout -b branchname` = create new branch
- `git branch` = tells you what branch you are on
- `git checkout master` = move back to the master branch
- `git pull` = merge your changes into other branch/repo (pull request, sent to owner of the repo)
- `git push` = commit local changes to remote (GitHub)

## Markdown

- `##` = signifies secondary heading (bold big font)
- `###` = signifies tertiary heading (slightly smaller font than secondary, not bold)
- `*` = bullet list item

## R Packages

- Primary location for R packages -> CRAN
- `available.packages()` = all packages available
- `head(rownames(a), 3)` = returns first three names of a
- `install.packages("nameOfPackage")` = install single package
- `install.packages(c("nameOfPackage", "nameOfPackage", "nameOfPackage"))` = install multiple package
- Bioconductor Packages:
  - `source("https://bioconductor.org/biocLite.R")`
  - `biocLite()` = install bioconductor packages
- `library(packagename)` = load package
- `search()` = see all functions in package after loading

## Types of Data Science Questions

- in order of difficulty: **Descriptive** -> **Exploratory** -> **Inferential** -> **Predictive** -> **Causal** -> **Mechanistic**
- **Descriptive analysis** = describe set of data, interpret what you see (census, Google Ngram)
- **Exploratory analysis** = discovering connections (correlation does not = causation)
- **Inferential analysis** = use data conclusions from smaller population for the broader group
- **Predictive analysis** = use data on one object to predict values for another (if X predicts Y, does not = X cause Y)
- **Causal analysis** = how does changing one variable affect another, using randomized studies, Strong assumptions, golden standard for statistical analysis
- **Mechanistic analysis** = understand exact changes in variables in other variables, modeled by empirical equations (engineering/physics)

## Data

- **Data** = values of qualitative or quantitative variables, belonging to a set of items (usually population)
- **Variables** = measurement/characteristic of an item (qualitative vs quantitative)
- **Data** = not always structured, usually raw file, different formats
- Most important thing is question, then it is data
- **Big data** = now possible to collect data cheap, but not necessarily all useful (need the right data)

## Experimental Design

- Formulate your question in advance
- **Statistical inference** = select subset, run experiment, calculate descriptive statistics, use inferential statistics to determine if results can be applied broadly
- **[Inference] Variability** = lower variability + clearer differences = decision
- **[Inference] Confounding** = underlying variable might be causing the correlation (sometimes called Spurious correlation)
  - dealing with confounding: fix variables, stratify (all options), randomize
- **[Prediction]** collection observations for different variable values, build predictive functions
  - similar problems of probability/sampling and confounding variables

- **[Prediction]** Difficult to understand where observation is from from different distributions. (size of effects important)
- **[Prediction]** Positive/negative statuses: True positive, false positive, false negative, true negative
  - **Sensitivity** =  $\Pr(\text{positive test} \mid \text{disease})$
  - **Specificity** =  $\Pr(\text{negative test} \mid \text{no disease})$
  - **Positive Predictive Value** =  $\Pr(\text{disease} \mid \text{positive test})$
  - **Negative Predictive Value** =  $\Pr(\text{no disease} \mid \text{negative test})$
  - **Accuracy** =  $\Pr(\text{correct outcome})$
- **Data dredging** = use data to fit hypothesis
- **Good experiments** = have replication, measure variability, generalize problem, transparent
- Prediction is not inference, and be ware of data dredging

# R Programming Course Notes

*Xing Su*

## Contents

Overview and History of R . . . . .	3
Coding Standards . . . . .	4
Workspace and Files . . . . .	4
R Console and Evaluation . . . . .	4
R Objects and Data Structures . . . . .	5
Vectors and Lists . . . . .	5
Matrices and Data Frames . . . . .	6
Arrays . . . . .	7
Factors . . . . .	8
Missing Values . . . . .	9
Sequence of Numbers . . . . .	10
Subsetting . . . . .	10
Vectors . . . . .	10
Lists . . . . .	11
Matrices . . . . .	11
Partial Matching . . . . .	11
Logic . . . . .	12
Understanding Data . . . . .	12
Split-Apply-Combine Functions . . . . .	13
<code>split()</code> . . . . .	13
<code>apply()</code> . . . . .	13
<code>lapply()</code> . . . . .	13
<code>sapply()</code> . . . . .	14
<code>vapply()</code> . . . . .	14
<code>tapply()</code> . . . . .	14
<code>mapply()</code> . . . . .	14
<code>aggregate()</code> . . . . .	15
Simulation . . . . .	16
Simulation Examples . . . . .	16
Generate Numbers for a Linear Model . . . . .	17
Dates and Times . . . . .	18

Base Graphics	18
Reading Tabular Data	19
Larger Tables	19
Textual Data Formats	19
Interfaces to the Outside World	20
Control Structures	21
<b>if - else</b>	21
<b>for</b>	21
<b>while</b>	22
<b>repeat and break</b>	22
<b>next and return</b>	22
Functions	23
Scoping	24
Scoping Example	24
Lexical vs Dynamic Scoping Example	25
Optimization	26
Debugging	27
R Profiler	27
Miscellaneous	28

## Overview and History of R

- **R** = dialect of the **S** language
  - S was developed by John Chambers @ Bell Labs
  - initiated in 1976 as internal tool, originally FORTRAN libraries
  - 1988 rewritten in C (version 3 of language)
  - 1998 version 4 (what we use today)
- **History of S**
  - Bell labs -> insightful -> Lucent -> Alcatel-Lucent
  - in 1998, S won the Association for computing machinery's software system award
- **History of R**
  - 1991 created in New Zealand by Ross Ihaka & Robert Gentleman
  - 1993 first announcement of R to public
  - 1995 Martin Machler convinces founders to use GNU General Public license to make R free
  - 1996 public mailing list created R-help and R-devel
  - 1997 R Core Group formed
  - 2000 R v1.0.0 released
- **R Features**
  - Syntax similar to S, semantics similar to S, runs on any platforms, frequent releasees
  - lean software, functionalities in modular packages, sophisticated graphics capabilities
  - useful for interactive work, powerful programming language
  - active user community and **FREE** (4 freedoms)
    - \* freedom to run the program
    - \* freedom to study how the program works and adapt it
    - \* freedom to redistribute copies
    - \* freedom to improve the program
- **R Drawbacks**
  - 40 year-old technology
  - little built-in support for dynamic/3D graphics
  - functionality based on consumer demand
  - objects generally stored in physical memory (limited by hardware)
- **Design of the R system**
  - 2 conceptual parts: base R from CRAN vs. everything else
  - functionality divided into different packages
    - \* **base R** contains core functionality and fundamental functions
    - \* other utility packages included in the base install: **util**, **stats**, **datasets**, ...
    - \* Recommended packages: boot class, KernSmooth, etc
  - 5000+ packages available

## Coding Standards

- Always use text files/editor
- Indent code (4 space minimum)
- limit the width of code (80 columns)
- limit the length of individual functions

## Workspace and Files

- `getwd()` = return current working directory
- `setwd()` = set current working directory
- `?function` = brings up help for that function
- `dir.create("path/foldername", recursive = TRUE)` = create directories/subdirectories
- `unlink(directory, recursive = TRUE)` = delete directory and subdirectories“
- `ls()` = list all objects in the local workspace
- `list.files(recursive = TRUE)` = list all, including subdirectories
- `args(function)` = returns arguments for the function
- `file.create("name")` = create file
  - `.exists("name")` = return true/false exists in working directory
  - `.info("name")` = return file info
  - `.info("name")$property` = returns value for the specific attribute
  - `.rename("name1", "name2")` = rename file
  - `.copy("name1", "name2")` = copy file
  - `.path("name1")` = return path of file

## R Console and Evaluation

- `<-` = assignment operator
- `#` = comment
- expression is evaluated after hitting `enter` and result is returned
- autoprinting occurs when you call a variable
  - `print(x)` = explicitly printing
- [1] at the beginning of the output = which element of the vector is being shown

## R Objects and Data Structures

- 5 basic/**atomic classes** of objects:
  1. character
  2. numeric
  3. integer
  4. complex
  5. logical
- **Numbers**
  - numbers generally treated as **numeric** objects (double precision real numbers - decimals)
  - Integer objects can be created by adding L to the end of a number(ex. 1L)
  - Inf = infinity, can be used in calculations
  - NaN = not a number/undefined
  - sqrt(value) = square root of value
- **Variables**
  - variable <- value = assignment of a value to a variable name

## Vectors and Lists

- **atomic vector** = contains one data type, most basic object
  - vector <- c(value1, value2, ...) = creates a vector with specified values
  - vector1\*vector2 = element by element multiplication (rather than matrix multiplication)
    - \* if the vectors are of different lengths, shorter vector will be recycled until the longer runs out
    - \* computation on vectors/between vectors (+, -, ==, /, etc.) are done element by element by default
    - \* %\*% = force matrix multiplication between vectors/matrices
  - vector("class", n) = creates empty vector of length n and specified class
    - \* vector("numeric", 3) = creates 0 0 0
- c() = concatenate
  - T, F = shorthand for TRUE and FALSE
  - 1+0i = complex numbers
- **explicit coercion**
  - as.numeric(x), as.logical(x), as.character(x), as.complex(x) = convert object from one class to another
  - nonsensible coercion will result in NA (ex. as.numeric(c("a", "b")))
  - as.list(data.frame) = converts a **data.frame** object into a **list** object
  - as.characters(list) = converts list into a character vector
- **implicit coercion**
  - matrix/vector can only contain one data type, so when attempting to create matrix/vector with different classes, forced coercion occurs to make every element to same class
    - \* *least common denominator* is the approach used (basically everything is converted to a class that all values can take, numbers → characters) and *no errors generated*
    - \* coercion occurs to make every element to same class (implicit)
    - \* x <- c(NA, 2, "D") will create a vector of character class
  - list() = special vector wit different classes of elements

- `list` = vector of objects of different classes
- elements of list use `[]`, elements of other vectors use `[]`
- **logical vectors** = contain values TRUE, FALSE, and NA, values are generated as result of logical conditions comparing two objects/values
- `paste(characterVector, collapse = " ")` = join together elements of the vector and separating with the `collapse` parameter
- `paste(vec1, vec2, sep = " ")` = join together different vectors and separating with the `sep` parameter
  - **Note:** *vector recycling applies here too*
  - LETTERS, letters= predefined vectors for all 26 upper and lower letters
- `unique(values)` = returns vector with all duplicates removed

## Matrices and Data Frames

- `matrix` can contain **only 1** type of data
- `data.frame` can contain **multiple**
- `matrix(values, nrow = n, ncol = m)` = creates a n by m matrix
  - constructed **COLUMN WISE** –> the elements are placed into the matrix from top to bottom for each column, and by column from left to right
  - matrices can also be created by adding the dimension attribute to vector
    - \* `dim(m) <- c(2, 5)`
  - matrices can also be created by binding columns and rows
    - \* `rbind(x, y), cbind(x, y)` = combine rows/columns; can be used on vectors or matrices
    - \* and / = element by element computation between two matrices
      - \* `%*%` = matrix multiplication
- `dim(obj)` = dimensions of an object (returns NULL if a vector)
  - `dim(obj) <- c(4, 5)` = assign `dim` attribute to an object
    - \* if object is a vector, R converts the vector to a n by m matrix (i.e. 4 rows by 5 column from the example command)
      - **Note:** *if n by m is larger than length of vector, then an error is returned*
    - \* **example**

```
# initiate a vector
x <-c(NA, 1, "cx", NA, 2, "dsa")
class(x)

## [1] "character"

x

## [1] NA     "1"    "cx"   NA     "2"    "dsa"

# convert to matrix
dim(x) <- c(3, 2)
class(x)

## [1] "matrix"
```

x

```
##      [,1] [,2]
## [1,] NA   NA
## [2,] "1"  "2"
## [3,] "cx" "dsa"
```

- `data.frame(var = 1:4, var2 = c(. . .))` = creates a data frame
  - `nrow()`, `ncol()` = returns row and column numbers
  - `data.frame(vector, matrix)` = takes any number of arguments and returns a single object of class “`data.frame`” composed of original objects
  - `as.data.frame(obj)` = converts object to data frame
  - data frames store tabular data
  - special type of list where every list has the same length (can be of different type)
  - data frames are usually created through `read.table()` and `read.csv()`
  - `data.matrix()` = converts a matrix to data frame
- `colMeans(matrix)` or `rowMeans(matrix)` = returns means of the columns/rows of a matrix/dataframe in a vector
- `as.numeric(rownames(df))` = returns row indices for rows of a data frame with unnamed rows
- **attributes**
  - objects can have attributes: `names`, `dimnames`, `row.names`, `dim` (matrices, arrays), `class`, `length`, or any user-defined ones
  - `attributes(obj)`, `class(obj)` = return attributes/class for an R object
  - `attr(object, "attribute") <- "value"` = creates/assigns a value to a new/existing attribute for the object
  - **names attribute**
    - \* all objects can have names
    - \* `names(x)` = returns names (NULL if no name exists)
      - `names(x) <- c("a", . . .)` = can be used to assign names to vectors
    - \* `lists(a = 1, b = 2, . . .)` = a, b are names
    - \* `dimnames(matrix) <- list(c("a", "b"), c("c", "d"))` = assign names to matrices
      - use list of two vectors: row, column in that order
  - `colnames(data.frame)` = return column names (can be used to set column names as well, similar to `dim()`)
  - `row.names` = names of rows in the data frame (attribute)

## Arrays

- multi-dimensional collection of data with  $k$  dimensions
  - `matrix` = 2 dimensional array
- `array(data, dim, dimnames)`
  - `data` = data to be stored in array
  - `dim` = dimensions of the array
    - \* `dim = c(2, 2, 5)` = 3 dimensional array -> creates 5 2x2 array
  - `dimnames` = add names to the dimensions
    - \* input must be a `list`

- \* every element of the `list` must correspond in length to the dimensions of the array
- \* `dimnames(x) <- list(c("a", "b"), c("c", "d"), c("e", "f", "g", "h", "i"))` = set the names for row, column, and third dimension respectively (2 x 2 x 5 in this case)
- `dim()` function can be used to create arrays from vectors or matrices
  - `x <- rnorm(20); dim(x) <- c(2, 2, 5)` = converts a 20 element vector to a 2x2x5 array

## Factors

- Factors are used to represent categorical data (integer vector where each value has a label)
- 2 types: **unordered** vs **ordered**
- treated specially by `lm()`, `glm()`
- Factors easier to understand because they self describe (vs. 1 and 2)
- `factor(c("a", "b"), levels = c("1", "2"))` = creates factor
  - `levels()` argument can be used to specify baseline levels vs other levels
    - \* *Note: without explicit specification, R uses alphabetical order*
  - `table(factorVar)` = how many of each are in the factor

## Missing Values

- `NaN` or `NA` = missing values
  - `NaN` = undefined mathematical operations
  - `NA` = any value not available or missing in the statistical sense
    - \* any operations with `NA` results in `NA`
    - \* `NA` can have different classes potentially (integer, character, etc)
  - **Note:** `NaN` is an `NA` value, but `NA` is not `NaN`
- `is.na()`, `is.nan()` = use to test if each element of the vector is `NA` and `NaN`
  - **Note:** cannot compare `NA` (with `==`) as it is not a value but a **placeholder** for a quantity that is not available
- `sum(my_na)` = sum of a logical vector (`TRUE` = 1 and `FALSE` = 0) is effectively the number of `TRUE`s
- **Removing NA Values**
  - `is.na()` = creates logical vector where `T` is where value exists, `F` is `NA`
    - \* subsetting with the above result can return only the non `NA` elements
  - `complete.cases(obj1, obj2)` = creates logical vector where `TRUE` is where both values exist, and `FALSE` is where any is `NA`
    - \* can be used on data frames as well
    - \* `complete.cases(data.frame)` = creates logical vectors indicating which observation/row is good
    - \* `data.frame[logicalVector, ]` = returns all observations with complete data
- **Imputing Missing Values** = replacing missing values with estimates (can be averages from all other data with the similar conditions)

## Sequence of Numbers

- `1:20` = creates a sequence of numbers from first number to second number
  - works in descending order as well
  - increment = 1
- `?':'` = enclose help for operators
- `seq(1, 20, by=0.5)` = sequence 1 to 20 by increment of .5
  - `length=30` argument can be used to specify number of values generated
- `length(variable)` = length of vector/sequence
- `seq_along(vector)` or `seq(along.with = vector)` = create vector that is same length as another vector
- `rep(0, times = 40)` = creates a vector with 40 zeroes
  - `rep(c(1, 2), times = 10)` = repeats combination of numbers 10 times
  - `rep(c(1, 2), each = 10)` = repeats first value 10 times followed by second value 10 times

## Subsetting

- R uses **one based index** → starts counting at 1
  - `x[0]` returns `numeric(0)`, not error
  - `x[3000]` returns `NA` (not out of bounds/error)
- `[]` = always returns object of same class, can select more than one element of an object `[1:2]`
- `[[]]` = can extract one element from list or data frame, returned object not necessarily list/dataframe
- `$` = can extract elements from list/dataframe that have names associated with it, not necessarily same class

## Vectors

- `x[1:10]` = first 10 elements of vector x
- `x[is.na(x)]` = returns all NA elements
- `x[!is.na(x)]` = returns all non NA elements
  - `x > 0` = would return logical vector comparing all elements to 0 (TRUE/FALSE for all values except for NA and NA for NA elements (NA a placeholder))
- `x[x>"a"]` = selects all elements bigger than a (lexicographical order in place)
- `x[logicalIndex]` = select all elements where logical index = TRUE
- `x[-c(2, 10)]` = returns everything **but** the second and tenth element
- `vect <- c(a = 1, b = 2, c = 3)` = names values of a vector with corresponding names
- `names(vect)` = returns element names for object
  - `names(vet) <- c("a", "b", "c")` = assign/change names of vector
- `identical(obj1, obj2)` = returns TRUE if two objects are exactly equal
- `all.equal(obj1, obj2)` = returns TRUE if two objects are near equal

## Lists

- `x <- list(foo = 1:4, bar = 0.6)`
- `x[1]` or `x["foo"]` = returns the list object `foo`
- `x[[2]]` or `x[["bar"]]` or `x$bar` = returns the content of the second element from the list (in this case vector without name attribute)
- `x[c(1, 3)]` = extract multiple elements of list (`[]`)
  - *Note: \$ can't extract multiple*
- `x[[name]]` = extract using variable, where as `$` must match name of element
- `x[[c(1, 3)]]` or `x[[1]][[3]]` = extracted nested elements of list third element of the first object extracted from the list

## Matrices

- `x[1, 2]` = extract the (row, column) element
  - `x[,2]` or `x[1,]` = extract the entire column/row
- `x[ , 11:17]` = subset the `x data.frame` with all rows, but only 11 to 17 columns
- when an element from the matrix is retrieved, a vector is returned
  - behavior can be turned off (force return a matrix) by adding `drop = FALSE`
    - \* `x[1, 2, drop = F]`

## Partial Matching

- works with `[]` and `$`
- `$` automatically partial matches the name (`x$a`)
- `[]` can partial match by adding `exact = FALSE`
  - `x[["a", exact = false]]`

## Logic

- <, >= → less than, greater or equal to
- == → exact equality
- != → inequality
- A | B = union
- A & B = intersection
- ! = negation
- & or | evaluates every instance/element in vector
- && or || evaluate only first element
  - All AND operators are evaluated before OR operators
- isTRUE(condition) = returns TRUE or FALSE of the condition
- xor(arg1, arg2) = exclusive OR, one argument must equal TRUE one must equal FALSE
- which(condition) = find the indices of elements that satisfy the condition (TRUE)
- any(condition) = TRUE if one or more of the elements in logical vector is TRUE
- all(condition) = TRUE if all of the elements in logical vector is TRUE

## Understanding Data

- use class(), dim(), nrow(), ncol(), names() to understand dataset
  - object.size(data.frame) = returns how much space the dataset is occupying in memory
- head(data.frame, 10), tail(data.frame, 10) = returns first/last 10 rows of data; default = 6
- summary() = provides different output for each variable, depending on class,
  - for numerical variables, displays min max, mean median, etc.
  - for categorical (factor) variables, displays number of times each value occurs
- table(data.frame\$variable) = table of all values of the variable, and how many observations there are for each
  - *Note: mean for variables that only have values 1 and 0 = proportion of success*
- str(data.frame) = structure of data, provides data class, num of observations vs variables, and name of class of each variable and preview of its contents
  - compactly display the internal structure of an R object
  - “What’s in this object”
  - well-suited to compactly display the contents of lists
- View(data.frame) = opens and view the content of the data frame

## Split-Apply-Combine Functions

- loop functions = convenient ways of implementing the Split-Apply-Combine strategy for data analysis

### `split()`

- takes a vector/objects and splits it into group by a factor or list of factors
- `split(x, f, drop = FALSE)`
  - `x` = vector/list/data frame
  - `f` = factor/list of factors
  - `drop` = whether empty factor levels should be dropped
- `interactions(g1(2, 5), g1(5, 2))` = 1.1, 1.2, ... 2.5
  - `g1(n, m)` = group level function
    - \* `n` = number of levels
    - \* `m` = number of repetitions
  - `split` function can do this by passing in `list(f1, f2)` in argument
    - \* `split(data, list(g1(2, 5), g1(5, 2)))` = splits the data into 1.1, 1.2, ... 2.5 levels

### `apply()`

- evaluate a function (often anonymous) over the margins of an array
- often used to apply a function to the row/columns of a matrix
- can be used to average array of matrices (general arrays)
- `apply(x, margin = 2, FUN, ...)`
  - `x` = array
  - `MARGIN` = 2 (column), 1 (row)
  - `FUN` = function
  - ... = other arguments that need to be passed to other functions
- *examples*
  - `apply(x, 1, sum)` or `apply(x, 1, mean)` = find row sums/means
  - `apply(x, 2, sum)` or `apply(x, 2, mean)` = find column sums/means
  - `apply(x, 1, quantile, props = c(0.25, 0.75))` = find 25% 75% percentile of each row
  - `a <- array(rnorm(2*2*10), c(2, 2, 10))` = create 10 2x2 matrix
  - `apply(a, c(1, 2), mean)` = returns the means of 10

### `lapply()`

- loops over a `list` and evaluate a function on each element and always returns a `list`
  - *Note: since input must be a list, it is possible that conversion may be needed*
- `lapply(x, FUN, ...)` = takes list/vector as input, applies a function to each element of the list, returns a list of the same length
  - `x` = list (if not list, will be coerced into list through “`as.list`”, if not possible → error)
    - \* `data.frame` are treated as collections of lists and can be used here
  - `FUN` = function (without parentheses)

- \* anonymous functions are acceptable here as well - (i.e `function(x) x[,1]`)
- ... = other/additional arguments to be passed for FUN (i.e. `min, max` for `rnorm()`)

- *example*

- `lapply(data.frame, class)` = the `data.frame` is a list of vectors, the `class` value for each vector is returned in a list (name of function, `class`, is without parentheses)
- `lapply(values, function(elem), elem[2])` = example of an anonymous function

### `sapply()`

- performs same function as `lapply()` except it simplifies the result
  - if result is of length 1 in every element, `sapply` returns vector
  - if result is vectors of the same length (>1) for each element, `sapply` returns matrix
  - if not possible to simplify, `sapply` returns a `list` (same as `lapply()`)

### `vapply()`

- safer version of `sapply` in that it allows to you specify the format for the result
  - `vapply(flags, class, character(1))` = returns the `class` of values in the `flags` variable in the form of character of length 1 (1 value)

### `tapply()`

- split data into groups, and apply the function to data within each subgroup
- `tapply(data, INDEX, FUN, ..., simplify = FALSE)` = apply a function over subsets of a vector
  - `data` = vector
  - `INDEX` = factor/list of factors
  - `FUN` = function
  - ... = arguments to be passed to function
  - `simplify` = whether to simplify the result
- *example*
  - `x <- c(rnorm(10), runif(10), rnorm(10, 1))`
  - `f <- gl(3, 10); tapply(x, f, mean)` = returns the mean of each group (f level) of x data

### `mapply()`

- multivariate apply, applies a function in parallel over a set of arguments
- `mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE)`
  - `FUN` = function
  - ... = arguments to apply over
  - `MoreArgs` = list of other arguments to `FUN`
  - `SIMPLIFY` = whether the result should be simplified
- *example*

```
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

### aggregate()

- aggregate computes summary statistics of data subsets (similar to multiple tapply at the same time)
- `aggregate(list(name = dataToCompute), list(name = factorVar1, name = factorVar2), function, na.rm = TRUE)`
  - `dataToCompute` = this is what the function will be applied on
  - `factorVar1, factorVar1` = factor variables to split the data by
  - *\*\*Note:* order matters here in terms of how to break down the data\*
  - `function` = what is applied to the subsets of data, can be sum/mean/median/etc
  - `na.rm = TRUE` -> removes NA values

## Simulation

- `sample(values, n, replace = FALSE)` = generate random samples
  - `values` = values to sample from
  - `n` = number of values generated
  - `replace` = with or without replacement
  - `sample(1:6, 4, replace = TRUE, prob=c(.2, .2...))` = choose four values from the range specified with replacing (same numbers can show up twice), with probabilities specified
  - `sample(vector)` = can be used to permute/rearrange elements of a vector
  - `sample(c(y, z), 100)` = select 100 random elements from combination of values y and z
  - `sample(10)` = select positive integer sample of size 10 without repeat
- Each probability distribution functions usually have 4 functions associated with them:
  - `r***` function (for “random”) –> random number generation
  - `d***` function (for “density”) –> calculate density
  - `p***` function (for “probability”) –> cumulative distribution
  - `q***` function (for “quantile”) –> quantile function
- If  $\Phi$  is the cumulative distribution function for a standard Normal distribution, then `pnorm(q) = \Phi(q)` and `qnorm(p) = \Phi^{-1}(q)`.
- `set.seed()` = reproduce same data

## Simulation Examples

- `rbinom(1, size = 100, prob = 0.7)` = returns a binomial random variable that represents the number of successes in a give number of independent trials
  - `1` = corresponds number of observations
  - `size = 100` = corresponds with the number of independent trials that culminate to each resultant observation
  - `prob = 0.7` = probability of success
- `rnorm(n, mean = m, sd = s)` = generate n random samples from the standard normal distribution (mean = 0, std deviation = 1 by default)
  - `rnorm(1000)` = 1000 draws from the standard normal distribution
  - `n` = number of observation generated
  - `mean = m` = specified mean of distribution
  - `sd = s` = specified standard deviation of distribution
- `dnorm(x, mean = 0, sd = 1, log = FALSE)`
  - `log` = evaluate on log scale
- `pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)`
  - `lower.tail` = left side, `FALSE` = right
- `qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)`
  - `lower.tail` = left side, `FALSE` = right
- `rpois(n, lambda)` = generate random samples from the poisson distrbution
  - `n` = number of observations generated
  - `lambda` =  $\lambda$  parameter for the poisson distribution or rate
- `rpois(n, r)` = generating Poisson Data

- `n` = number of values
- `r` = rate
- `ppois(n, r)` = cumulative distribution
  - `ppois(2, 2) = Pr(x <= 2)`
- `replicate(n, rpois())` = repeat operation `n` times

## Generate Numbers for a Linear Model

- Linear model

$$y = \beta_0 + \beta_1 x + \epsilon \text{ where } \epsilon \sim N(0, 2^2), x \sim N(0, 1^2), \beta_0 = 0.5, \beta_1 = 2$$

```
set.seed(20)
x <- rnorm(100)           # normal
x <- rbinom(100, 1, 0.5) # binomial
e <- rnorm(100, 0, 2)
y <- 0.5 + 2* x + e
```

- Poisson model

$$Y \sim Poisson(\mu) \text{ where } \mu = \beta_0 + \beta_1 x \text{ where } \beta_0 = 0.5, \beta_1 = 2$$

```
x <- rnorm(100)
log.mu <- 0.5 + 0.3* x
y <- rpois(100, exp(log.mu))
```

## Dates and Times

- `Date` = date class, stored as number of days since 1970-01-01
- `POSIXct` = time class, stored as number of seconds since 1970-01-01
- `POSIXlt` = time class, stored as list of sec min hours
- `Sys.Date()` = today's date
- `unclass(obj)` = returns what obj looks like internally
- `Sys.time()` = current time in `POSIXct` class
- `t2 <- as.POSIXlt(Sys.time())` = time in `POSIXlt` class
  - `t2$min` = return min of time (only works for `POSIXlt` class)
- `weekdays(date)`, `months(date)`, `quarters(date)` = returns weekdays, months, and quarters of time/date inputed
- `strptime(string, "%B %d, %Y %H:%M")` = convert string into time format using the format specified
- `difftime(time1, time2, units = 'days')` = difference in times by the specified unit

## Base Graphics

- `data(set)` = load data
- `plot(data)` = R plots the data as best as it can
  - `x` = variable, x axis
  - `y` = variable
  - `xlab`, `ylab` = corresponding labels
  - `main`, `sub` = title, subtitle
  - `col = 2` or `col = "red"` = color
  - `pch = 2` = different symbols for points
  - `xlim, ylim(v1, v2)` = restrict range of plot
- `boxplot(x ~ y, data = d)` = creates boxplot for x vs y variables using the `data.frame` provided
- `hist(x, breaks)` = plots histogram of the data
  - `break = 100` = split data into 100 bins

## Reading Tabular Data

- `read.table()`, `read.csv()` = most common, read text files (rows, col) return data frame
- `readLines()` = read lines of text, returns character vector
- `source(file)` = read R code
- `dget()` = read R code files (R objects that have been reparsed)
- `load()`,  `unserialize()` = read binary objects
- writing data
  - `write.table()`, `writeLines()`, `dump()`, `put()`, `save()`, `serialize()`
- `read.table()` arguments:
  - `file` = name of file/connection
  - `header` = indicator if file contains header
  - `sep` = string indicating how columns are separated
  - `colClasses` = character vector indicating what each column is in terms of class
  - `nrows` = number of rows in dataset
  - `comment.char` = char indicating beginning of comment
  - `skip` = number of lines to skip in the beginning
  - `stringsAsFactors` = defaults to TRUE, should characters be coded as Factor
- `read.table` can be used without any other argument to create `data.frame`
  - telling R what type of variables are in each column is helpful for larger datasets (efficiency)
  - `read.csv()` = `read.table` except default `sep` is comma (`read.table` is " ") and `header` = TRUE`

## Larger Tables

- *Note: help page for `read.table` important*
- need to know how much RAM is required → calculating memory requirements
  - `numRow` x `numCol` x 8 bytes/numeric value = size required in bites
  - double the above results and convert into GB = amount of memory recommended
- set `comment.char = ""` to save time if there are no comments in the file
- specifying `colClasses` can make reading data much faster
- `nrow = n`, number of rows to read in (can help with memory usage)
  - `initial <- read.table("file", rows = 100)` = read first 100 lines
  - `classes <- sapply(initial, class)` = determine what classes the columns are
  - `tabAll <- read.table("file", colClasses = classes)` = load in the entire file with determined classes

## Textual Data Formats

- `dump` and `dput` preserve metadata
- text formats are editable, not space efficient, and work better with version control system (they can only track changes in text files)
- `dput(obj, file = "file.R")` = creates R code to store all data and meta data in "file.R" (ex. `data`, `class`, `names`, `row.names`)
- `dget("file.R")` = loads the file/R code and reconstructs the R object
- `dput` can only be used on one object, where as `dump` can be used on multiple objects
- `dump(c("obj1", "obj2"), file= "file2.R")` = stores two objects
- `source("file2.R")` = loads the objects

## Interfaces to the Outside World

- `url()` = function can read from webpages
- `file()` = read uncompressed files
- `gzfile()`, `bzfile()` = read compressed files (gzip, bzip2)
- `file(description = "", open = "")` = file syntax, creates connection
  - `description` = description of file
  - `open` = r -readonly, w - writing, a - appending, rb/wb/ab - reading/writing/appending binary
  - `close()` = closes connection
  - `readLines()` = can be used to read lines after connection has been established
- `download.file(fileURL, destfile = "fileName", method = "curl")`
  - `fileURL` = url of the file that needs to be downloaded
  - `destfile = "fileName"` = specifies where the file is to be saved
    - \* “dir/fileName” = directories can be referenced here
  - `method = "curl"` = necessary for downloading files from “<https://>” links on Macs
    - \* `method = "auto"` = should work on all other machines

## Control Structures

- Common structures are
  - **if**, **else** = testing a condition
  - **for** = execute a loop a fixed number of times
  - **while** = execute a loop while a condition is true
  - **repeat** = execute an infinite loop
  - **break** = break the execution of a loop
  - **next** = skip an iteration of a loop
  - **return** = exit a function
- *Note: Control structures are primarily useful for writing programs; for command-line interactive work, the **apply** functions are more useful*

**if - else**

```
# basic structure
if(<condition>) {
    ## do something
} else {
    ## do something else
}

# if tree
if(<condition1>) {
    ## do something
} else if(<condition2>) {
    ## do something different
} else {
    ## do something different
}
```

- `y <- if(x>3){10} else {0}` = slightly different implementation than normal, focus on assigning value

**for**

```
# basic structure
for(i in 1:10) {
    # print(i)
}

# nested for loops
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
    for(j in seq_len(ncol(x))) {
        # print(x[i, j])
    }
}
```

- `seq_along()` = to length of the vector

- `for(letter in x)` = loop through letter in character vector
- `seq_len()` = 1:length of vector

### **while**

```
count <- 0
while(count < 10) {
  # print(count)
  count <- count + 1
}
```

- conditions can be combined with logical operators

### **repeat and break**

- Repeat initiates an infinite loop
- not commonly used in statistical applications but they do have their uses
- The only way to exit a `repeat` loop is to call `break`

```
x0 <- 1
tol <- 1e-8
repeat {
  x1 <- computeEstimate()
  if(abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1 # requires algorithm to converge
  }
}
```

- *Note:* The above loop is a bit dangerous because there's no guarantee it will stop
  - Better to set a hard limit on the number of iterations (e.g. using a `for` loop) and then report whether convergence was achieved or not.

### **next and return**

- `next` = (no parentheses) skips an element, to continue to the next iteration
- `return` = signals that a function should exit and return a given value

```
for(i in 1:100) {
  if(i <= 20) {
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

## Functions

- `name <- function(arg1, arg2, ...){ }`
  - inputs can be specified with default values by `arg1 = 10`
  - it is possible to define an argument to `NULL`
  - returns **last expression** of function
  - many functions have `na.rm`, can be set to `TRUE` to remove `NA` values from calculation

- structure

```
f <- function(<arguments>) {  
  ## Do something interesting  
}
```

- functions are first class object and can be **treated like other objects** (pass into other functions)
  - functions can be nested, so that you can define a function inside of another function
- functions have named arguments (i.e. `x = mydata`) which can be used to specify **default values**
  - `sd(x = mydata)` (matching by name)
- formal arguments = arguments included in the functional definition
  - `formals()` = returns all formal arguments
  - not all functional call specifies all arguments, some can be missing and may have default values
  - `args()` = return all arguments you can specify
  - multiple arguments inputted in random orders (R performs positional matching) —> not recommended
  - argument matching order: exact —> partial —> positional
    - \* *partial* = instead of typing `data = x`, use `d = x`
- Lazy Evaluation
  - R will evaluate as needed, so everything executes until an error occurs
    - \* `f <- function (a, b) {a^2}`
    - \* if `b` is not used in the function, calling `f(5)` will not produce an error
- “...” argument
  - used to extend other functions by representing the rest of the arguments
  - generic functions use `...` to pass extra arguments (i.e. `mean = 1, sd = 2`)
  - necessary when the number of arguments passed can not be known in advance
    - \* functions that use “...” = `paste()`, `cat()`
  - **Note:** *arguments coming after ... must be explicitly matched and cannot be partially matched*

## Scoping

- scoping rules determine how a value is associated with a free variable in a function
- **free variables** = variables not explicitly defined in the function (not arguments, or local variables - variable defined in the function)
- R uses **lexical/static scoping**
  - common alternative = **dynamic scoping**
  - **lexical scoping** = values of free vars are searched in the environment in which the function is defined
    - \* environment = collection of symbol/value pairs ( $x = 3.14$ )
      - each package has its own environment
      - only environment **without** parent environment is the *empty environment*
    - **closure/function closure** = function + associated environment
  - search order for free variable
    1. environment where the function is defined
    2. parent environment
    3. ... (repeat if multiple parent environments)
    4. top level environment: global environment (worspace) or namespace package
    5. empty environment → produce error
  - when a function/variable is called, R searches through the following list to match the first result
    1. .GlobalEnv
    2. package:stats
    3. package:graphics
    4. package:grDevices
    5. package:utils
    6. package:datasets
    7. package:methods
    8. Autoloads
    9. package:base
  - **order matters**
    - .GlobalEnv = everything defined in the current workspace
    - any package that gets loaded with `library()` gets put in position 2 of the above search list
    - namespaces are separate for functions and non-functions
      - \* possible for object c and function c to coexist

## Scoping Example

```
make.power <- function(n) {  
  pow <- function(x) {  
    x^n  
  }  
  pow  
}  
cube <- make.power(3)  # defines a function with only n defined ( $x^3$ )  
square <- make.power(2) # defines a function with only n defined ( $x^2$ )  
cube(3)                # defines x = 3
```

```

## [1] 27

square(3)           # defines x = 3

## [1] 9

# returns the free variables in the function
ls(environment(cube))

## [1] "n"    "pow"

# retrieves the value of n in the cube function
get("n", environment(cube))

## [1] 3

```

### Lexical vs Dynamic Scoping Example

```

y <- 10
f <- function(x) {
  y <- 2
  y^2 + g(x)
}
g <- function(x) {
  x*y
}

```

- **Lexical Scoping**
  1. f(3) —> calls g(x)
  2. y isn't defined locally in g(x) —> searches in parent environment (working environment/global workspace)
  3. finds y —> y = 10
- **Dynamic Scoping**
  1. f(3) —> calls g(x)
  2. y isn't defined locally in g(x) —> searches in calling environment (f function)
  3. find y —> 2

– **parent frame** = refers to calling environment in R, environment from which the function was called
- **Note:** when the defining environment and calling environment is the same, lexical and dynamic scoping produces the same result
- **Consequences of Lexical Scoping**
  - all objects must be carried in memory
  - all functions carry pointer to their defining environment (memory address)

## Optimization

- Optimization routines in R (`optim`, `nlm`, `optimize`) require you to pass a function whose argument is a vector of parameters
  - *Note: these functions minimize, so use the negative constructs to maximize a normal likelihood*
- **Constructor functions** = functions to be fed into the optimization routines
- **example**

```
# write constructor function
make.NegLogLik <- function(data, fixed=c(FALSE, FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
}
# initialize seed and print function
set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals); nLL

## function(p) {
##   params[!fixed] <- p
##   mu <- params[1]
##   sigma <- params[2]
##   a <- -0.5*length(data)*log(2*pi*sigma^2)
##   b <- -0.5*sum((data-mu)^2) / (sigma^2)
##   -(a + b)
## }
## <environment: 0x7fbf33a2dd80>

# Estimating Parameters
optim(c(mu = 0, sigma = 1), nLL)$par

##       mu     sigma
## 1.218239 1.787343

# Fixing sigma = 2
nLL <- make.NegLogLik(normals, c(FALSE, 2))
optimize(nLL, c(-1, 3))$minimum

## [1] 1.217775

# Fixing mu = 1
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e-6, 10))$minimum

## [1] 1.800596
```

## Debugging

- **message**: generic notification/diagnostic message, execution continues
  - `message()` = generate message
- **warning**: something's wrong but not fatal, execution continues
  - `warning()` = generate warning
- **error**: fatal problem occurred, execution stops
  - `stop()` = generate error
- **condition**: generic concept for indicating something unexpected can occur
- **invisible()** = suppresses auto printing
- **Note**: random number generator must be controlled to reproduce problems (`set.seed` to pinpoint problem)
- **traceback**: prints out function call stack after error occurs
  - must be called right after error
- **debug**: flags function for debug mode, allows to step through function one line at a time
  - `debug(function)` = enter debug mode
- **browser**: suspends the execution of function wherever its placed
  - embedded in code and when the code is run, the browser comes up
- **trace**: allows inserting debugging code into a function at specific places
- **recover**: error handler, freezes at point of error
  - `options(error = recover)` = instead of console, brings up menu (simi)

## R Profiler

- optimizing code cannot be done without performance analysis and profiling

```
# system.time example
system.time({
  n <- 1000
  r <- numeric(n)
  for (i in 1:n) {
    x <- rnorm(n)
    r[i] <- mean(x)
  }
})
```

```
##    user  system elapsed
##  0.147   0.004   0.152
```

- **system.time(expression)**
  - takes R expression, returns amount of time needed to execute (assuming you know where)
  - computes time (in sec) - gives time until error if error occurs
  - can wrap multiple lines of code with {}
  - returns object of class `proc_time`
    - \* **user time** = time computer experience

- \* **elapsed time** = time user experience
- \* usually close for standard computation
  - *elapse > user* = CPU wait around other processes in the background (read webpage)
  - *elapsed < user* = multiple processor/core (use multi-threaded libraries)
- **Note:** R doesn't multi-thread (performing multiple calculations at the same time) with basic package
  - \* Basic Linear Algebra Standard [BLAS] libraries do, prediction, regression routines, matrix
  - \* i.e. vecLib/Accelerate, ATLAS, ACML, MKL
- **Rprof()** - useful for complex code only
  - keeps track of functional call stack at regular intervals and tabulates how much time is spent in each function
  - default sampling interval = 0.02 second
  - calling **Rprof()** generates **Rprof.out** file by default
    - \* **Rprof("output.out")** = specify the output file
- **summaryRprof()** = summarizes **Rprof()** output, 2 methods for normalizing data
  - loads the **Rprof.out** file by default, can specify output file **summaryRprof("output.out")**
  - **by.total** = divide time spent in each function by total run time
  - **by.self** = first subtracts out time spent in functions above in call stack, and calculates ratio to total
  - **\$sample.interval** = 0.02 - interval
  - **\$sampling.time** = 7.41 - seconds, elapsed time
    - \* **Note:** generally user spends all time at top level function (i.e. *lm()*), but the function simply calls helper functions to do work so it is not useful to know about the top level function times
    - **Note:** *by.self* = more useful as it focuses on each individual call/function
- Good to break code into functions so profilers can give useful information about where time is spent
- C/FORTRAN code is not profiled
- **Note:** R must be compiled with profiles support (generally the case)
- **Note:** should NOT be used with *system.time()*

## Miscellaneous

- **unlist(rss)** = converts a list object into data frame/vector
- **ls("package:elasticnet")** = list methods in package

# Getting and Cleaning Data Course Notes

*Xing Su*

## Contents

Overview . . . . .	2
Raw and processed data . . . . .	2
Tidy Data . . . . .	2
Download files . . . . .	3
Reading Excel files . . . . .	3
Reading XML . . . . .	3
Reading JSON . . . . .	4
data.table . . . . .	4
Reading from MySQL [install.packages("RMySQL"); library(RMySQL)] . . . . .	6
HDF5 . . . . .	6
Web (Scraping) ( <a href="#">tutorial</a> ) . . . . .	7
Working with API . . . . .	8
Reading from Other Sources . . . . .	8
dplyr . . . . .	9
tidyverse . . . . .	11
lubridate . . . . .	12
Subsetting and Sorting . . . . .	12
Summarizing Data . . . . .	13
Creating New Variables . . . . .	14
Reshaping Data . . . . .	15
Merging Data . . . . .	16
Editing Text Variables . . . . .	17
Regular Expressions . . . . .	18
Working with Dates . . . . .	19
Data Sources . . . . .	19

## Overview

- finding and extracting raw data
- today any how to make data tiny
- Raw data → processing script → tidy data → data analysis → data communication

## Raw and processed data

- **Data** = values of qualitative/quantitative, variables, belonging to a set of items
  - **variables** = measurement of characteristic of an item
- **Raw data** = original source of data, often hard to use, processing must be done before analysis, may need to be processed only once
- **Processed data** = ready for analysis, processing done (merging, transforming, etc.), all steps should be recorded
- Sequencing DNA: \$1B for Human Genome Project → \$10,000 in a week with Illumina

## Tidy Data

### 1. Raw Data

- no software processing has been done
- did not manipulate, remove, or summarize in anyway

### 2. Tidy data set

- end goal of cleaning data process
- each variable should be in one column
- each observation of that variable should be in a different row
- one table for each kind of variable
  - if there are multiple tables, there should be a column to link them
- include a row at the top of each file with variable names (variable names should make sense)
- in general data should be save in one file per table

### 3. Code book describing each variable and its values in the tidy data set

- information about the variables (w/ units) in dataset *NOT* contained in tidy data
- information about the summary choice that were made (median/mean)
- information about experimental study design (data collection methods)
- common format for this document = markdown/Word/text
  - “*study design*” section = thorough description of how data was collected
  - “*code book*” section = describes each variable and units

### 4. Explicit steps and exact recipe to get through 1 - 3 (instruction list)

- ideally a computer script (no parameters)
- output = processed tidy data
- in addition to script, possibly may need steps to run files, how script is run, and explicit instructions

## Download files

- Set working directory
  - *Relative*: `setwd("./data")`, `setwd("../")` <— move up in directory
  - *Absolute*: `setwd("/User/Name/data")`
- Check if file exists and download file
  - `if(!file.exists("data")){dir.create("data")}`
- Download file
  - `download.file(url, destfile= "directory/filename.extension", method = "curl")`
    - \* `method = "curl"` [mac only for https]
  - `dateDownloaded <- date()` <— record the download date
- Read file and load data
  - `read.table()` —> need to specify file, header, sep, row.names, nrows
    - \* `read.csv()` = automatically set sep = "," and header = TRUE
  - `quote = ""` —> no quotes (extremely helpful, common problem)
  - `na.strings` = set the character that represents missing value
  - `nrows` = how many rows to read
  - `skip` = how many lines to skip
  - `col.names` = specifies column names
  - `check.names = TRUE/FALSE` = If TRUE then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted so that they are

## Reading Excel files

- xlsx package: `read.xlsx(path, sheetIndex = 1, ...)`
  - `colIndex, rowIndex` = can be used to read certain rows and columns
- `write.xlsx()` = write out excel file
- `read.xlsx2()` = faster than `read.xlsx()` but unstable for reading subset of rows
- XLConnect packaage has more options for writing/manipulating Excel files
- generally good to store data in database/csv/tabc separated files (.tab/.txt), easier to distribute

## Reading XML

- XML = extensible markup language
- frequented used to store structured data, widely used in internet apps
- extracting XML = basis for most of web scraping
- components
  - *markup* = labels that give text structure
  - *content* = actual text of document
- `tags = <section> </section> <line-break />`
- `elements = <Greeting> test </Greeting>`
- `attributes = <image src ="a.jpg" alt = "b">`
- reading file into R

- library(XML)
- doc <- xmlTreeParse(fileUrl, useInternal = TRUE) <— loads data
- rootNode <- xmlRoot(doc) = wrapper element for entire document
- xmlName(rootNode) <- returns name of the document
- names(rootNode) <- return names of elements
- rootNode[[1]] = access first elements, similar to list
- rootNode[[1]][[1]] = first sub component in the first element
- xmlSApply(rootNode, xmlValue) = returns every single tagged element in the doc
- **XPath** (new language)
  - get specific elements of document
  - /node = top level node
  - //node = node at any level
  - node[@attr-name = 'bob'] = node with attribute name
    - \* xpathSApply(rootNode, "//name", xmlValue) = get the values of all elements with tag "name"
    - \* xpathSApply(rootNode, "//price", xmlValue) = get the values of all elements with tag "price"
- **extract content by attributes**
  - doc <- htmlTreeParse(url, useInternal = True)
  - scores <- xpathSApply(doc, "//li@class='score'", xmlvalue) = look for li elements with class = "score" and return their value

## Reading JSON

- **JSON** = JavaScript Object Notation
- lightweight data storage, common format for data from application programming interfaces (API)
- similar to XML in structure but different in syntax/format
- data stored as:
  - numbers (double)
  - strings (double quoted)
  - boolean (true/false)
  - array (ordered, comma separated enclosed in [])
  - object (unordered, comma separated collection of key/value pairs enclosed in {})
- **jsonlite** package (json vignette)
  - library(jsonlite) <— loads package
  - data <- fromJSON(url) = strips data
    - \* names(data\$owner) = returns list of names of all columns of owner data frame
    - \* data\$owner\$login = returns login instances
  - data <- toJSON(dataframe, pretty = TRUE) = converts data frame into JSON format
    - \* pretty = TRUE <- formats the code nicely
  - cat(data) = prints out JSON code from the converted data frame
  - fromJSON() = converts from JSON object/code back to data frame

## data.table

- inherits from data.frame (external package) -> all functions that accept data.frame work on data.table

- can be much faster (written in C), ***much much faster*** at subsetting/grouping/updating
- **syntax:** `dt <- data.table(x = rnorm(9), y = rep(c(a, b, c), each = 3), z = rnorm(9))`
- `tables()` = returns all data tables in memory
  - shows name, nrow, MB, cols, key
- some subset works like before  $\rightarrow dt[2, ]$ ,  $dt[dt$y=="a", ]$ 
  - `dt[c(2, 3)]` = subset by rows, rows 2 and 3 in this case
- **column subsetting** (modified for `data.table`)
  - argument after comma is called an ***expression*** (collection of statements enclosed in {})
  - `dt[, list(means(x), sum(z))]` = returns mean of x column and sum of z column (no “” needed to specify column names, x and z in example)
  - `dt[, table(y)]` = get table of y value (perform any functions)
- **add new columns**
  - `dt[, w:=z^2]`
    - \* when this is performed, a new data.table is created and data copied over (not good for large datasets)
  - `dt2 <- dt; dt[, y:= 2]`
    - \* when changes are made to dt, changes get translated to dt2
    - \* *Note: if copy must be made, use the copy() function instead*
- **multiple operations**
  - `dt[, m:= {temp <- (x+z); log2(temp +5)}]`  $\rightarrow$  adds a column that equals  $\log_2(x+z + 5)$
- **plyr like operations**
  - `dt[, a:=x>0]` = creates a new column a that returns TRUE if  $x>0$ , and FALSE other wise
  - `dt[,b:=mean(x+w), by=a]` = creates a new column b that calculates the aggregated mean for  $x+w$  for when a = TRUE/FALSE, meaning every b value is gonna be the same for TRUE, and others are for FALSE
- **special variables**
  - `.N` = returns integer, length 1, containing the number (essentially count)
    - \* `dt <- data.table (x=sample(letters[1:3], 1E5, TRUE))`  $\rightarrow$  generates data table
    - \* `dt[, .N by =x]`  $\rightarrow$  creates a table to count observations by the value of x
- **keys** (quickly filter/subset)
  - *example:* `dt <- data.table(x = rep(c("a", "b", "c"), each 100), y = rnorm(300))`  $\rightarrow$  generates data table
    - \* `setkey(dt, x)`  $\rightarrow$  set the key to the x column
    - \* `dt['a']`  $\rightarrow$  returns a data frame, where x = ‘a’ (effectively filter)
- **joins** (merging tables)
  - *example:* `dt1 <- data.table(x = c('a', 'b', ...), y = 1:4)`  $\rightarrow$  generates data table
    - \* `dt2 <- data.table(x= c('a', 'd', ...), z = 5:7)`  $\rightarrow$  generates data table
    - \* `setkey(dt1, x); setkey(dt2, x)`  $\rightarrow$  sets the keys for both data tables to be column x
    - \* `merge(dt1, dt2)` = returns a table, combine the two tables using column x, filtering to only the values that match up between common elements the two x columns (i.e. ‘a’) and the data is merged together
- **Fast reading of files**
  - *example:* `big_df <- data.frame(norm(1e6), norm(1e6))`  $\rightarrow$  generates data table

```

* file <- tempfile() -> generates empty temp file
* write.table(big.df, file=file, row.names=FALSE, col.names = TRUE, sep =
  "\t". quote = FALSE) -> writes the generated data from big.df to the empty temp file
* fread(file) -> read file and load data = much faster than read.table()

```

## Reading from MySQL [install.packages("RMySQL"); library(RMySQL)]

- free/widely used open sources database software, widely used for internet base applications
- each row = record
- data are structured in databases -> series tables (dataset) -> fields (columns in dataset)
- dbConnect(MySQL(), user = "genome", db = "hg19", host = "genome-mysql.cse.ucsc.edu")  
-> open a connection to the database
  - db = "hg19" -> select specific database
  - MySQL() can be replaced with other arguments to use other data structures
- dbGetQuery(db, "show databases;") -> return the result from the specified SQL query executed through the connection
  - any SQL query can be substituted here
- dbDisconnect(db) -> disconnects the open connection
  - crucial to disconnect as soon as all queries are performed
- dbListFields(db, "name") -> returns the list of fields (columns) from the specified table
- dbReadTable(db, "name") -> returns the the specified table
- query <- dbSendQuery(db, "query") -> send query to MySQL database and store it remotely
- fetch(query, n = 10) -> executes query and returns the result
  - n = 10 -> returns the first 10 rows
- dbClearResult(query) -> clears query from remote database, important
- sqldf package example

## HDF5

- source("http://bioconductor.org/biocLite.R"); biocLite("rhd5"); library(rhdf5) ([tutorial](#))
- used for storing large datasets, supports range of data types, can be used optimize reading/writing from disc to R
- **hierarchical format**
  - groups containing 0 or more datasets and metadata
    - \* each group has group header with group name and list of attributes
    - \* each group has group symbol table with a list of objection in the group
  - datasets -> multidimensional array of data elements with metadata
    - \* each dataset has a header with name, datatype, data space, storage layout
    - \* each dataset has a data array (similar to data frame) with the data
- h5createFile("filename") -> creates HDF5 file and returns TRUE/FALSE
- h5createGroup("filename", "group1/subgroup1/...") -> creates group within the specified file
  - groups are created at the root "/" by default, but subgroups can be created by providing the path AFTER the parent group is created
- h5ls("filename") -> prints out contents of the file by group, name, otype, etc

- `h5write(A, "filename", "groupname")` -> writes A (could be array, matrix, etc.) to the file under the specific group
  - `h5write(A, "filename", "A")` -> writes A directly at the top level
  - `h5write(values, "filename", "group/subgroupname/obj", index = list(1:3, 1))` -> writes values in the specified obj at the specific location
    - \* example: `h5write(c(12, 13, 14), "ex.h5", "foo/A", index = list(1:3, 1))` -> writes values 12, 13, 14 in the object A at the first 3 rows of the first column in the /foo group
- `h5read("filename", "groupname/A")` -> reads A from specified group of the file
  - `h5read("filename", "A")` -> reads A directly from the top level of the file
  - `h5read("filename", "group/subgroupname/obj", index = list(1:3, 1))` -> writes specified values in the specified obj at the group within the file

## Web (Scraping) ([tutorial](#))

- **Webscraping** = programmatically extracting data from the HTML code of websites
- `con = url("website")` -> opens connection from URL
- `htmlCode = readLines(con)` -> reads the html code from the URL
  - always remember to `close(con)` after using it
  - the `htmlCode` return here is a bit unreadable
- **Parsing with XML**
  - `library(XML)`
  - `url <- "http://..."` -> sets the desired url as a character variable
  - `html <- htmlTreeParse(url, useInternalNodes = T)` -> reads and parses the html code
  - `xpathSApply(html, "//title", xmlValue)` -> returns the value of the `//title` node/element
  - `xpathSApply(html, "//td[@id='col-citedBy']", xmlValue)` -> returns the value of the `//td` element where the `id = 'col-citedBy'` in the html code
- **Parsing with httr package** ([tutorial](#))
  - `library(httr)`
  - `html2 <- GET(url)` -> reads the html code from the url
  - `cont = content(html2, as = "text")` -> extracts the html code as a long string
  - `parsedHtml = htmlParse(cont, asText = TRUE)` -> parses the text into HTML (same output as the XML package function `htmlTreeParse`)
  - `xpathSApply(html, "//title", xmlValue)` -> returns the value of the `//title` node/element
  - Accessing websites with passwords
    - \* `pg = GET("url")` -> this would return a status 401 if the website requires log in without authenticating
    - \* `pg2 = GET("url", authenticate("username", "password"))` -> this authenticates before attempting to access the website, and the result would return a status 200 if authentication was successful
    - \* `names2(pg2)` -> returns names of different components
  - Using handles (username/login information)
    - \* using handles allows you to save authentication across multiple parts of the website (only authenticate once for different requests)
    - \* example: `google = handle("http://google.com")`
    - \* `pg1 = GET(handle = google, path = "/")`
    - \* `pg2 = GET(handle = google, path = "search")`

## Working with API

- load `http` package first: `library(httr)`
  - allows GET, POST, PUT, DELETE requests if you are authorized
- `myapp = oauth_app("app", key = "consumerKey", secret = "consumerSecret")` → start authorization process for the app
- `sig = sign_oauth1.0(myapp, token = "tokenGenerated", token_secret = "tokenSecret")` → login using the token information (sets up access so you can use it to get data)
- `homeTL = get("url", sig)` → use the established authentication (instead of username/password) to get the data (usually in JSON format)
  - use the url to specify what data you would like to get
  - use the documentation to get information and parameters for the url and data you have access to
- `json1 = content(homeTL)` → recognizes the data in JSON format and converts it to a structured R object [a bit hard to read]
- `json2 = jsonlite::fromJSON(toJSON(json1))` → converts data back into JSON format and then use the fromJSON function from the jsonlite package to read the data into a data frame
  - each row corresponds to a line of the data you received
- **github example ([tutorial](#)):**
  - `library(httr)`
  - `myapp <- oauth_app("github", key = "clientID", secrete = "clientSecret")`
    - \* an application must be registered with github first to generate the client ID and secrets
  - `github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)`
    - \* `oauth_endpoints()` → returns the the authorize/access url/endpoints for some common web applications (github, Facebook, google, etc)
    - \* `oauth2.0_token(endPoints, app)` → generates an oauth2.0 token with the credentials provided
  - `gtoken <- config(token = github_token)` → sets up the configuration with the token for authentication
  - `req <- with_config(gtoken, GET("https://api.github.com/rate_limit"))` → executes the configuration set to send a get request from the specified URL, and returns a response object
  - `library(jsonlite); json1 <- fromJSON(toJSON(content(req)))` → converts the content of the response object, to JSON format, and converts it again to data frame format
  - `names(json1)` → returns all the column names for the data frame
  - `json1[json1$name == "datasharing",]$created_at` → returns the create date for the data sharing repo

## Reading from Other Sources

- interacting directly with files
  - `file` → open a connection to a text file
  - `url` → opens a connection to a url
  - `gzfile/bzfile` → opens a connection to a .gz/.bz2 file
  - `?connections` → for more information about opening/closing connections in R
- **foreign package**
  - loads data from Minitab/S/SAS/SPSS/Stat/Systat
  - basic functions

```

* read.arff (Weka)
* read.dta (Stata)
* read.mtp (Minitab)
* read.octave (Octave)
* read.spss (SPSS)
* read.xport (SAS)
* read.fwf (fixed width files, [.for])
* example: data <- read.fwf(file = "quiz02q5.for", skip = 4, widths = c(-1,
  9,-5, 4, 4, -5, 4, 4,-5, 4, 4,-5, 4, 4))
* widths = c() -> specifies the width of each variable
* the negative numbers indicate the space to disregard/take out

```

- **Other packages/functions**

- **RPostresSQL** -> provides DBI-compliant database connection from R
- **RODBC** -> provides interfaces to multiple databases including PostgreSQL, MySQL, Microsoft Access, SQLite
- **RMongo/rmongodb** -> provides interfaces to MongoDB
  - \* similar to MySQL, except send queries in the database's syntax
- Reading Images (functions)
  - \* jpeg, readbitmap, png, EBImage (Bioconductor)
- Reading (GIS Geographical Information Systems) data (packages)
  - \* rgdal, rgeos, raster
- Reading music data (packages)
  - \* tuneR, seewave

## dplyr

- external package, load by **library(dplyr)**
  - developed by Hadley Wickham of RStudio
  - optimized/distilled version of the **plyr** package, does not provide new functionality but greatly simplifies existing R functionality
  - very fast, many key operations coded in C++
  - **dplyr** package also works on data.table and SQL interface for relational databases (DBI package)
- load data into **tbl\_df** (data frame table) by **data <- tbl\_df(rawData)**
  - main advantage to using a **tbl\_df** over a regular data frame is printing
  - more compact output/informative -> similar to a combination of head/str
    - \* displays class, dimension, preview of data (10 rows and as many columns as it can fit), undisplayed variables and their class
- **functions**
  - *Note:* for all functions, first argument always the data frame, and result is always a data frame
  - **select()**
    - \* *example:* **select(dataFrameTable, var1, var2, var3)** -> returns a table (similar in format as calling the actual data frame table)
    - \* no need to use \$ as we would normally, since **select()** understands that the variables are from the **dataFrameTable**
    - \* columns are returns in order specified

- \* ":" operator (normally reserved for numbers) can be used to select a range of columns (from this column to that column), works in reverse order as well -> `select(dataFrameTable, var1:var5)`
- \* "-column" can be used to specify columns to throw away -> `select(dataFrameTable, -var1)` -> but this does not modify original dataFrameTable
- \* `-(var1:size)` = eliminate all columns
- \* normally this can be accomplished by finding the indices of names using the `match("value", vector)` function
- `filter()`
  - \* *example:* `filter(cran, package == "swirl")` -> returns a table (similar in format as calling the actual data frame table)
  - \* returns all rows where the condition evaluates to TRUE
  - \* automatically recognized that package is a column without "\$"
  - \* able to specify as many conditions as you want, separated by ",", "|" and "&" work here as well
  - \* multiple conditions specified by "," is equivalent to "&"
  - \* `is.na(var1)` also works here
  - \* **Note:** "?Comparison" brings up relevant documentation for relational comparators
- `arrange()`
  - \* *example:* `arrange(dataFrameTable, var)` -> order the data frame table by specified column/variable
  - \* `desc(var)` -> arrange in descending order by column value
  - \* can specify multiple values to sort by by using ","
  - \* order listed in the call will be the order that the data is sorted by (can use in conjunction with `desc()`)
- `rename()`
  - \* *example:* `rename(dataFrameTable, colName = newColName)` -> renames the specified column with new name
  - \* capable of renaming multiple columns at the same time, no quotes needed
- `mutate()`
  - \* create a new variable based on the value of one or more existing variables in the dataset
  - \* capable of modifying existing columns/variables as well
  - \* *example:* `mutate(dataFrameTable, newSize = size / 2^20)` -> create a new column with specified name and the method of calculating
  - \* multiple columns can be created at the same time by using "," as separator, new variables can even reference each other in terms of calculation
- `summarize()`
  - \* collapses the dataset into a single row
  - \* *example:* `summarize(dataFrameTable, avg = mean(size))` -> returns the mean from the column in a single variable with the specified name
  - \* summarize(can return the requested value for each group in the dataset
- `group_by()`
  - \* *example:* `by_package <- group_by(cran, package)` -> creates a grouped data frame table by specified variable
  - \* `summarize(by_package, mean(size))` -> returns the mean size of each group (instead of 1 value from the `summarize()` example above)
  - \* **Note:** `'n()'` -> counts number of observation in the current group
  - \* **Note:** `n_distinct()` -> Efficiently count the number of unique values in a vector
  - \* **Note:** `quantile(variable, probs = 0.99)` -> returns the 99% percentile from the data

- \* *Note:* by default, dplyr prints the first 10 rows of data if there are more than 100 rows; if there are not, it will print everything
- rbind\_list()
  - \* bind multiple data frames by row and column
  - \* example: rbind\_list(passed, failed)
- Chaining/Piping
  - allows stringing together multiple function calls in a way that is compact and readable, while still accomplishing the desired result
    - \* *Note:* all variable calls refer to the `tbl_df` specified at the same level of the call
  - %>% -> chaining operator
    - \* *Note:* ?chain brings up relevant documentation for the chaining operator
    - \* Code on the right of the operator operates on the result from the code on the left
    - \* `exp1 %>% exp2 %>% exp3 ...`
      - `exp1` is calculated first
      - `exp2` is then applied on `exp1` to achieve a result
      - `exp3` is then applied to the result of that operation, etc.
    - \* *Note:* the chaining aspect is done with the data frame table that is being passed from one call to the next
    - \* *Note:* if the last call has no additional arguments, `print()` for example, then it is possible to leave() off

## tidyr

- gather()
  - gather columns into key value pairs
  - example: `gather(students, sex, count, -grade)` -> gather each key (in this case named `sex`), value (in this case `count`) pair into one row
    - \* effectively translates to (columnName, value) with the new names imposed on both -> all combinations of column name and value
    - \* `-grade` -> signifies that the column does not need to be remapped, so that column is preserved
    - \* `class1:class5` -> can be used instead to specify where to gather the key values
- separate()
  - separate one column into multiple column
  - example: `separate(data = res, col = sex_class, into = c("sex", "class"))` -> split the specified column in the data frame into two columns
    - \* *Note:* the new columns are created in place, and the other columns are pushed to the right
    - \* *Note:* `separate()` is able to automatically split non-alphanumeric values by finding the logical separator; it is also possible to specify the separator by using the “sep” argument
- spread()
  - spread key-value pairs across multiple columns -> turn values of a column into column headers/variables/new columns
  - example: `spread(students3, test, grade)` -> splits “test” column into variables by using it as a key, and “grade” as values
    - \* *Note:* no need to specify what the columns are going to be called, since they are going to be generated using the values in the specified column
    - \* *Note:* the value will be matched and split up according their alignment with the key (“test”) -> `midterm, A`

- `extract_numeric()`
  - extract numeric component of variable
  - *example:* `extract_numeric("class5")` → returns 5
  - *example:* `mutate(class = extract_numeric(class))` → changes the class name to numbers only
- `unique()` → general R function, not specific to `tidyR`
  - returns a vector with the duplicates removed
- **Note:** when there are redundant information, it's better to split up the info into multiple tables; however, each table should also contain primary keys, which identify observations and link data from one table to the next

## **lubridate**

- consistent, memorable syntax for working with dates
- `wday(date, label = TRUE)` → returns number 1 - 7 representing Sunday - Saturday, or returns three letter day of the week if label = TRUE
- `today()`, `now()` → returns the current date and time, with extractable parts (`hour()`, `month()`)
  - `tzone = "America/New_York"` → can use the “tzone” argument to specify time zones (list [here](#))
- `ymd("string")` → converts string in to year month day format to a POSIXct time variable
  - `mdy("string")` → parses date in month day year format
  - `dmy(2508195)` → parses date in day month year format using a number
  - `ymd_hms("string")` → parses the year month day, hour minute second
  - `hms("string")` → parses hour minute second
    - \* `tz = ""` → can use the “tz” argument to specify time zones (list [here](#))
  - **Note:** there are a variety of functions that are available to parse different formats, all of them are capable of converting the correct information if the order of month year day is correct
  - **Note:** when necessary, “//” or “—” should be added to provide clarity in date formatting
- `update(POSIXct, hours = 8, minutes = 34, seconds = 55)` → updates components of a date time
  - **Note:** does not alter the date time passed in unless explicitly assigned
- arithmetic can be performed on date times by using the `days()` `hours()` `minutes()`, etc. functions
  - *example:* `now() + hours(5) + minutes(2)` → returns the date time for 5 hours and 2 minutes from now
- `with_tz(time, tone = "")` → return date-time in a different time zone
- `as.period(new_interval(last_time, arrive))` → return the properly formatted difference between the two date times

## **Subsetting and Sorting**

- **subsetting**
  - `x <- data.frame("var1" = sample(1:5), "var2" = sample(6:10), "var3" = (11:15))` → initiates a data frame with three names columns
  - `x <- x[sample(1:5)]` → this scrambles the rows
  - `x$var2[c(2,3)] = NA` → setting the 2nd and 3rd element of the second column to NA
  - `x[1:2, "var2"]` → subsetting the first two row of the the second column

- `x[(x$var1 <= 3 | x$var3 > 15), ]` –> return all rows of x where the first column is less than or equal to three or where the third column is bigger than 15
- `x[which(x$var2 >8), ]` –> returns the rows where the second column value is larger than 8
  - \* *Note: which(condition) –> useful in dealing with NA values as it returns the indices of the values where the condition holds true (returns FALSE for NA)*

- **sorting/ordering**

- `sort(x$var1)` –> sort the vector in increasing/alphabetical order
  - \* `decreasing = TRUE` –> use decreasing argument to sort vector in decreasing order
  - \* `na.last = TRUE` –> use na.last argument to sort the vector such that all the NA values will be listed last
- `x[order(x$var1, x$var2), ]` –> order the x data frame according to var1 first and var2 second
- `dplyr` package: `arrange(data.frame, var1, desc(var2))` –> see dplyr sections

- **adding row/columns**

- `x$var4 <- rnorm(5)` –> adds a new column to the end called var4
- `cbind(X, rnorm(5))` –> combines data frame with vector (as a column on the right)
  - \* `rbind()` –> combines two objects by putting them on top of each other (as a row on the bottom)
- \* *Note: order specified in the argument is the order in which the operation is performed*

## Summarizing Data

- `head(data.frame, n = 10) / tail(data.frame, n = 10)` –> prints top/bottom 10 rows of data
- `summary(data.frame)` –> displays summary information
  - for factors variables, the summary table will display count of the top 6 values
  - for numeric variables, the summary table will display min, 1st quantile, median, mean, 3rd quantile, max
- `str(data.frame)` –> displays class of the object, dimensions, variables (name, class, preview of data)
- `quantile(variable, na.rm = TRUE, probs = c(0.5, 0.75, 0.9))` –> displays the specified quantile of the variable
  - default returns 0, .25, .5, .75, 1 quantiles
- `table(variable, useNA = "ifany")` –> tabulates the values of the variable by listing all possible values and the number of occurrences
  - `useNA = "ifany"` –> this will add another column if there are any NA values in the variable and displays how many as well
  - `table(var1, var2)` –> tabulate the data against each other to see if theres an relationship between them
- **check for missing values**
  - `sum(is.na(variable))` –> TRUE = 1, FALSE = 0, so if this sum = 0, then there's no missing values
  - `any(is.na(variable))` –> returns TRUE/FALSE of if there is any NA values in the variable
  - `all(variable >0)` –> check all values of a variable against some condition and return TRUE/FALSE
- **row/column sums**
  - `colSums/rowSums(is.na(data.frame))` –> column sum of is.na check for every column; works the exact same way with rowSums

- **values with specific characteristics**

- `table(data.frame$var1 %in% c("str1", "str2"))` –> returns a FALSE/TRUE table that counts how many values from the data frame variable contains the specified values in the subsequent vector
- `x[x$var1 %in% c("str1", "str2"), ]` –> subsets rows from the data frame where the var1 == str1 or str2

- **cross tabs**

- `xt <- xtabs(Freq ~ Gender + Admit, data = data.frame)` –> displays a cross table of Gender and Admit variables, where the values of frequency is displayed

	Admitted	Rejected
Male	1198	1493
Female	557	1278

```
* `xt2 <- xtabs(var1 ~ ., data = data.frame)` --> cross-tabulate variable 1 with all other variables, c
* `ftable(xt2)` --> compacts the different tables and prints out a more compacted version
```

- **size of data set**

- `object.size(obj)` –> returns size of object in bytes
- `print(object.size(obj), units = "Mb")` –> prints size of object in Mb

## Creating New Variables

- **sequences**

- `s <- seq(1, 10, by = 2)` –> creates a sequence from 1 to 10 by intervals of 2
- `length = 3` –> use the length argument to specify how many numbers to generate
- `seq(along = x)` –> create as many elements as vector x

- **subsetting variables**

- `restData$nearMe = restData$neighborhood %in% c("Roland", "Homeland")` –> creates a new variable “nearMe” that returns TRUE if the neighborhood value is Roland or Homeland, and FALSE otherwise

- **binary variables**

- `restData$zipWrong = ifelse(restData$zipCode<0, TRUE, FALSE)` –> creates a new variable “zipWrong” that returns TRUE if the zipcode is less than 0, and FALSE otherwise
- `ifelse(condition, result1, result2)` –> this function is the same as a if-else statement

- **categorical variables**

- `restData$zipGroups = cut(restData$zipCode, breaks = quantile(restData$zipCode))` –> creates new variable “zipGroups” that specify ranges for the zip code data such that the observations are divided into groups created by the quantile function
  - \* `cut(variable, breaks)` –> cuts a variable/vector into groups at the specified breaks
  - \* *Note: class of resultant variable = factor*
  - \* `quantile(variable)` –> returns 0, .25, .5, .75, 1 by default and thus provides for ranges/groups for the data to be divided in

- using **Hmisc** package
  - \* `library(Hmisc)`
  - \* `restData$zipGroups = cut2(restData$zipCode, g = 4)`
  - \* `cut2(variable, g=4)` –> automatically divides the variable values into 4 groups according the quantiles
  - \* *Note: class of resultant variable = factor*
- **factor variables**
  - `restData$zcf <- factor(restData$zipCode)` –> converts an existing vector to factor variable
    - \* `levels = c("yes", "no")` –> use the levels argument to specify the order of the different factors
    - \* *Note: by default, converting variables to the factor class, the levels will be structured alphabetically unless otherwise specified*
  - `as.numeric(factorVariable)` –> converts factor variable values into numeric by assigning the lowest (first) level 1, the second lowest level 2, ..., etc.
- **category + factor split**
  - using **plyr** and **Hmisc** packages
  - `library(plyr); library(Hmisc)`
  - `readData2 <- mutate(restData, zipGroups = cut2(zipCode, g = 4))`
    - \* this creates `zipGroups` and splits the data from `zipCode` all at the same time
- **common transforms**
  - `abs(x)` –> absolute value
  - `sqrt(x)` –> square root
  - `ceiling(x), floor()` –> round up/down to integer
  - `round(x, digits = n)` –> round to the number of digits after the decimal point
  - `signif(x, digits = n)` –> round to the number of significant digits
  - `cos(x), sin(x), tan(x)` ... etc –> trigonometric functions
  - `log(x), log2(x), log10(x)` –> natural log, log 2, log 10
  - `exp(x)` –> exponential of x

## Reshaping Data

- **melting data frames**
  - `library(reshape2)` –> loads the reshape2 package
  - `mtcars$carname <- rownames(mtcars)` –> takes the row names/name of each observation and makes a new variable called “carname”
  - `carMelt <- melt(mtcars, id=c("carname", "gear", "cyl"), measure.vars = c("mpg", "hp"))` –> converts dataframe into a castable melted data frame by reshaping the data
    - \* ID variables and measured variables are defined separately through “id” and “measure.vars” arguments
    - \* ID variables (identifiers) are kept in rows, while all measured variables have been split into variable and value columns
    - \* variable column = “mpg”, “hp” to qualify for the corresponding value column
    - \* value column = contains the numeric value from previous measured variable columns like “mpg” “hp”
    - \* ID variables are repeated when a new measured variable begins such that each row is an unique observation (long/tall table)
- **casting data frames**

- `cylData <- dcast(carMtl, cyl ~ variable)` -> tabulate the data by rows (left hand side variable, cyl in this case) by columns (right hand side variable, variable in this case), so this is a table of cylinder vs mpg and hp
  - \* by default, `dcast()` summarizes the data set by providing the length argument (count)
  - \* can add argument (mean) to specify the value of the table produced
- **calculating factor sums**
  - **method 1:** `tapply(InsectSprays$count, InsectSpray$spray, sum)` -> splits the InsectSpray count values by spray groups and calculates the sum of each group
  - **method 2:** split-apply-combine
    - \* `s <- split(InsectSprays$count, InsectSpray$spray)` -> splits InsectSpray count values into groups by spray
    - \* `sprCount <- lapply(s, sum)` -> apply sum for all of the groups and return a list
    - \* `unlist(sprCount)` -> converts a list into a vector with names
    - \* `sapply(s, sum)` -> apply sum for all of the groups and return a vector
  - **method 3:** plyr package
    - \* `ddply(dataframe, .(variables), method, function)` -> for each subset of a data frame, apply function then combine results into a data frame
    - \* `dataframe` -> data being processed
    - \* `.(variables)` -> variables to group/summarize by
    - \* `method` -> can be a variety of different functions defined within the ply package, mutate, summarize, arrange, filter, select, etc.
    - \* `function` -> how the data is going to be calculated
    - \* `example: ddply(InsectSprays, .(spray), summarize, sum = sum(count))` -> summarize spray groups by providing the same of counts for each group
    - \* creating new variable
    - \* `spraySums<- ddply(InsectSprays, .(spray), summarize, sum = ave(count, FUN = sum))` -> creates a data fram (2 columns) where each row is filled with the corresponding spray and sum (repeated multiple times for each group)
    - \* the result can then be used and added to the dataset for analysis

## Merging Data

- `merge(df1, df2, by/by.x/by.y, all = TRUE)` -> merges two data frames
  - `df1, df2` -> data frames to be merged
  - `by = "col1"/c("col1", "col2")` -> merge the two data frames by columns of the specified names
    - \* *Note: if "by" argument is not specified, the two data frames will be merged with all columns with the same name [default behavior]*
    - \* *Note: columns must have the same names for this to work*
  - `by.x/by.y = "col.x"/"col.y"` -> specifies which columns from x and y should be used to perform the merge operation
  - `all = TRUE` -> if there are values that exist in one but not the other, new rows should be created with NA as values for the missing data
- **plyr package: `join(df1, df2)`** -> merges the columns by columns in common
  - faster but only works with columns that have the same name
  - `dfList = list(df1, df2, df3); join_all(dfList)` -> joins together a list of data frames using the common columns

## Editing Text Variables

- `tolower()` -> make all character values lowercase letters
- `toupper()` -> make all character values uppercase letters
- `strsplit(value, "\\\\.)")` -> splits string into character vector by specified separator
  - *Note:* \\ must be added for the reserved operators in R
- `sapply(list, function)` -> can specify custom functions to return the part of the character desired
  - example: `fElement <- function(x){x[1]}`; `sapply(vector, fElement)`
- `sub(pattern, replacement, object)` -> replaces the first occurrence of pattern and replaces it with the replacement string
  - example: `sub("_", "", nameVector)` -> removes first "\_" from the character vector
- `gsub(pattern, replacement, object)` -> replaces all occurrences of the specified pattern and replaces it with the replacement string
- `grep("text", object, value = FALSE)` -> searches through object to return the indices of where the text is found
  - `value = TRUE` -> returns the values instead of the indices
  - *Note:* `grep()` returns `integer(0)` if no value appears (length of the result = 0)
- `grepl("text", object)` -> searches through object and returns the T/F logical vector for if the text has been found
  - example: `data2 <- data1![grepl("test", data$intersection), ]`
- **string package [library(stringr)]**
  - `nchar(object/string)` -> returns number of characters in each element of object/string, works on matrix/data frames as well
  - `substr("text", 1, 7)` -> returns a substring of the specified beginning and ending characters
    - \* *Note:* R is uses 1 based indexing system, which means the first character of the string = 1
    - \* *Note:* substring returns a string that includes both first AND last letters and everything inbetween
  - `paste("str1", "str2", sep = " ")` -> combines two strings together into one by using the specified separator (default = " ")
  - `paste0(("str1", "str2"))` -> combines series of strings together with no separator
  - `str_trim(" text ")` -> trims off whitespace from start and end of string
- **General Rules**
  - name of variables should be
    - \* all lowercase when possible
    - \* descriptive
    - \* unique
    - \* contains no underscores/dots/space
  - variables with character values
    - \* made into factor variables
    - \* descriptive
    - \* use TRUE/FALSE instead of 1/0
    - \* use Male/Female instead of 1/0 or M/F

## Regular Expressions

- **RegEx** = combination of literals and metacharacters
- used with `grep/grep1/sub/gsub` functions or any other that involve searching for strings in character objects/variables
- `^` = start of the line (metacharacter)
  - *example:* `^text` matches lines such as “text ...”
- `$` = end of the line (metacharacter)
  - *example:* `text$` matches lines such as “... text”
- `[]` = set of characters that will be accepted in the match (character class)
  - *example:* `^[Ii]` matches lines such as “I ...” or “i ...”
- `[0-9]` = searches for a range of characters (character class)
  - *example:* `[a-zA-Z]` will match any letter in upper or lower case
- `[^?.]` = when used at beginning of character class, “`^`” means not (metacharacter)
  - *example:* `[^?.]$` matches any line that does not end in “?” or “.”
- `.` = any character (metacharacter)
  - *example:* `9.11` matches 9/11, 9911, 9-11, etc
- `|` = or, used to combine subexpressions called alternatives (metacharacter)
  - *example:* `^([Gg]ood | [Bb]ad)` matches any lines that start with lower/upper “Good...” and “Bad ...”
  - **Note:** *( ) limits the scope of alternatives divided by “|” here*
- `?` = expression is optional → 0/1 of some character/expression (metacharacter)
  - *example:* `[Gg]eorge( [Ww]\.)?` `[Bb]ush` matches “george bush”, “George W. Bush”
  - **Note:** “`\.` was added before `“.”` because “`.` is a metacharacter, “`.` called escape dot, tells the expression to read it as an actual period instead of an operator”
- `*` = any number of repetition, including none → 0 or more of some character/expression (metacharacter)
  - *example:* `.*` matches anything combination of characters
  - **Note:** *\* is greedy = always matches the longest possible string that satisfies the regular expression*
    - \* greediness of `*` can be turned off with the `?`
    - \* *example:* `s.*?s` matches the shortest “s...s” text
- `+` = 1 or more repetitions → 1 or more of some character/expression (metacharacter)
  - *example:* `[0-9]+` matches many at least digit 1 numbers such as “0”, “90”, or “021442132”
- `{m, n}` = interval quantifier, allows specifying the minimum and maximum number of matches (metacharacter)
  - `m` = at least, `n` = not more than
  - `{m}` = exactly `m` matches
  - `{m, }` = at least `m` matches
  - *example:* `Bush( +[^ ]+ +){1, 5} debates` matches “Bush + (at least one space + any word that doesn’t contain space + at least one space) this pattern repeated between 1 and 5 times + debates”
- `( )` = define group as the the text in parentheses, groups will be remembered and can be referred to by `\1, \2, etc.`
  - *example:* `([a-zA-Z]+) +\1 +` matches “any word + at least one space + the same word repeated + at least one space” → “night night”, “so so”, etc.

## Working with Dates

- `date()` = returns current date in character format
- `Sys.Date()` = returns the current date in Date format
- `format(object, "format")` = formats object in specified format
  - when object = Date, then we can use any combination of the following
    - \* `%d` = day as number (0-31)
    - \* `%a` = abbreviated weekday
    - \* `%A` = unabbreviated weekday
    - \* `%m` = month (00-12)
    - \* `%b` = abbreviated month
    - \* `%B` = unabbreviated month
    - \* `%y` = 2 digit year
    - \* `%Y` = 4 digit year
  - *example:* `format(Sys.Date(), "%a %b %d")` = returns “Sun Jan 18”
- `as.Date("character", "format")` = converts character vector/variable into Date format by using the codes above
  - *example:* `z <- as.Date("1jan1960", "%d%b%Y")` = creates a Date of “1960-01-01”
- `Date1 - Date2` = prints the difference between the dates in this format “Time difference of n days”
  - `as.numeric()` on this result will print/store n, the numeric difference
- `weekdays(Date)`, `months(Date)` = returns the weekday/month of the given Date object
- `julian(Date)` = converts the Date, which is the number of days since the origin
  - `attr(, "origin")` = prints out the origin for the julian date format, which is 1970-01-01
- `lubridate` package [`library(lubridate)`] -> see lubridate section
  - `?Sys.timezone` = documentation on how to determine/set timezones

## Data Sources

- `quantmod` package = get historical stock prices for publicly traded companies on NASDAQ or NYSE

# Exploratory Data Analysis Course Notes

Xing Su

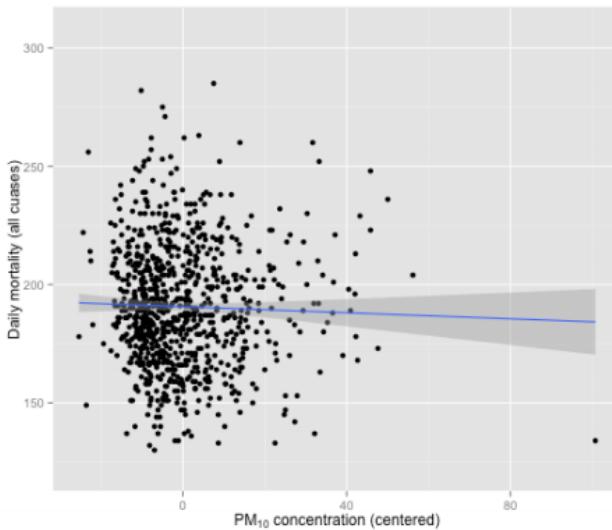
## Contents

Principle of Analytic Graphics . . . . .	3
Exploratory Graphs ( <a href="#">examples</a> ) . . . . .	4
One Dimension Summary of Data . . . . .	4
Two Dimensional Summaries . . . . .	5
Process of Making a Plot/Considerations . . . . .	7
Base Plotting . . . . .	7
Base Graphics Functions and Parameters . . . . .	8
Base Plot Example . . . . .	8
Multiple Plot Example . . . . .	9
Graphics Device . . . . .	11
<a href="#">lattice</a> Plotting System [ <code>library(lattice)</code> ] . . . . .	13
<a href="#">lattice</a> Functions and Parameters . . . . .	13
<a href="#">lattice</a> Example . . . . .	14
<a href="#">ggplot2</a> Plotting System [ <code>library(ggplot2)</code> ] . . . . .	15
<a href="#">ggplot2</a> Functions and Parameters . . . . .	15
<a href="#">ggplot2</a> Comprehensive Example . . . . .	20
Hierarchical Clustering . . . . .	22
Procedure for Constructing Hierarchical Clusters ( <code>hclust</code> function) . . . . .	22
Approaches for Merging Points/Clusters . . . . .	22
Characteristics of Hierarchical Clustering Algorithms . . . . .	23
<code>hclust</code> Function and Example . . . . .	23
<code>myplcclust</code> Function and Example . . . . .	24
<code>heatmap</code> Function and Example . . . . .	25
<code>image</code> Function and Example . . . . .	25
K-means Clustering . . . . .	27
Procedure for Constructing K-means Clusters ( <code>kmeans</code> function) . . . . .	27
Characteristics of K-means Clustering Algorithms . . . . .	28
Dimension Reduction . . . . .	29
Singular Value Decomposition (SVD) . . . . .	30
Principal Components Analysis (PCA) . . . . .	30
SVD and PCA Example . . . . .	30

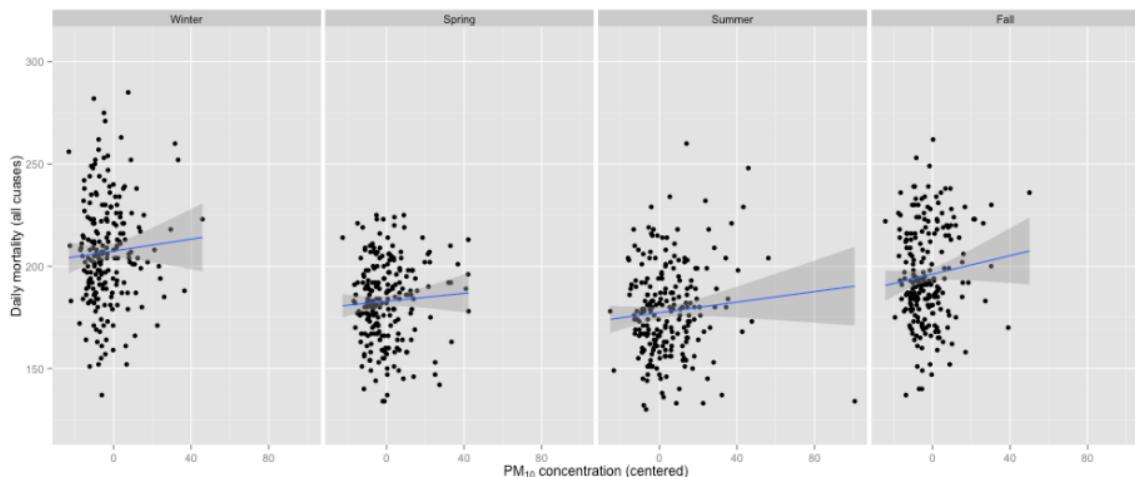
Create Approximations/Data Compression . . . . .	35
Color Packages in R Plots . . . . .	37
<b>grDevices</b> Package . . . . .	37
<b>RColorBrewer</b> Package . . . . .	38
Case Study: Human Activity Tracking with Smart Phones . . . . .	41
Case Study: Fine Particle Pollution in the U.S. from 1999 to 2012 . . . . .	49

## Principle of Analytic Graphics

- **Principle 1: Show Comparisons**
  - always comparative (compared to what)
  - randomized trial - compare control group to test group
  - evidence for a hypothesis is always relative to another competing hypothesis
- **Principle 2: Show causality/mechanism/explanation/systematic structure**
  - form hypothesis to evidence showing a relationship (causal framework, why something happened)
- **Principle 3: Show multivariate data**
  - more than 2 variables because the real world is multivariate
  - show as much data on a plot as you can
  - *example*



- slightly negative relationship between pollution and mortality



- when split up by season, the relationships are all positive  $\rightarrow$  season = confounding variable
- **Principle 4: Integration of evidence**

- use as many modes of evidence/displaying evidence as possible (modes of data presentation)
- integrate words/numbers/images/diagrams (information rich)
- analysis should drive the tool
- **Principle 5: Describe/document evidence with appropriate labels/scales/sources**
  - add credibility to that data graphic
- **Principle 6: Content is the most important**
  - analytical presentations ultimately stand/fall depending on quality/relevance/integrity of content

## Exploratory Graphs ([examples](#))

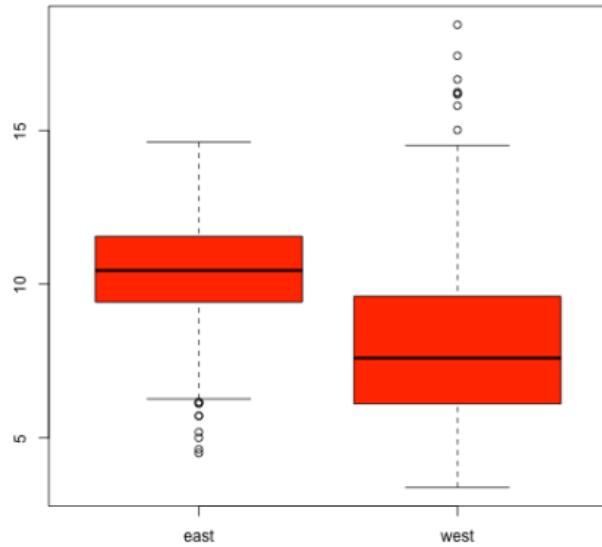
- **Purpose:** understand data properties, find pattern in data, suggest modeling strategies, debug
- **Characteristics:** made quickly, large number produced, gain personal understanding, appearances and presentation are aren't as important

### One Dimension Summary of Data

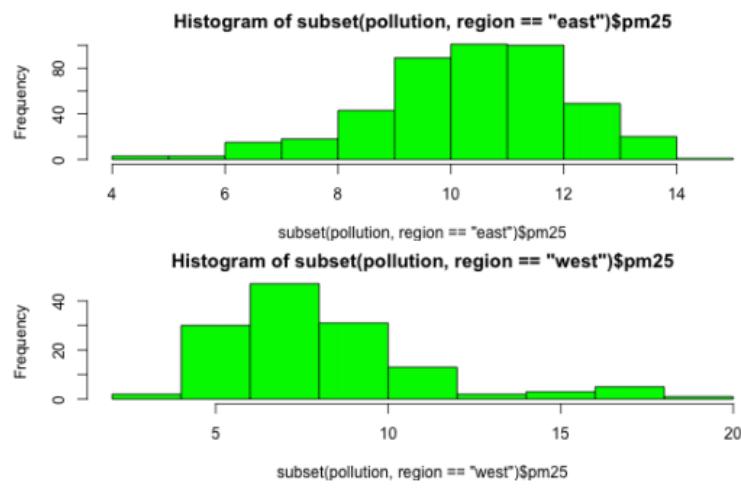
- `summary(data)` = returns min, 1st quartile, median, mean, 3rd quartile, max
- `boxplot(data, col = "blue")` = produces a box with middles 50% highlighted in the specified color
  - `whiskers` =  $\pm 1.58IQR/\sqrt{n}$
  - `IQR` = interquartile range,  $Q_3 - Q_1$
  - `box` = 25%, median, 75%
- `histograms(data, col = "green")` = produces a histogram with specified breaks and color
  - `breaks` = 100  $\rightarrow$  the higher the number is the smaller/narrower the histogram columns are
- `rug(data)` = density plot, add a strip under the histogram indicating location of each data point
- `barplot(data, col = wheat)` = produces a bar graph, usually for categorical data
- **Overlaying Features**
- `abline(h/v = 12)` = overlays horizontal/vertical line at specified location
  - `col = "red"`  $\rightarrow$  specifies color
  - `lwd = 4`  $\rightarrow$  line width
  - `lty = 2`  $\rightarrow$  line type

## Two Dimensional Summaries

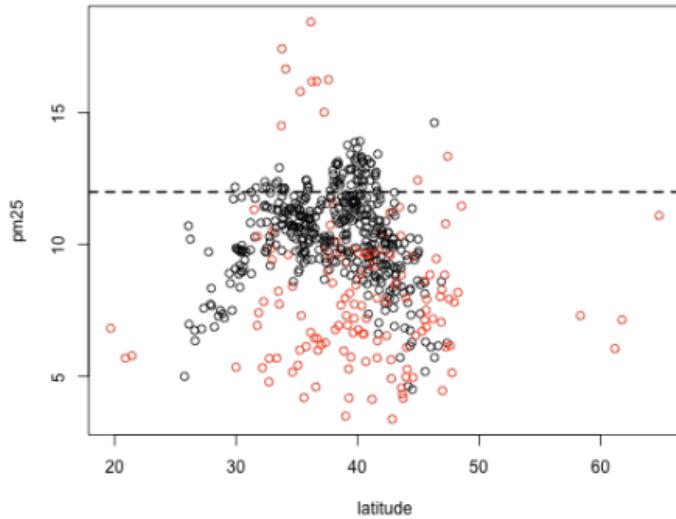
- multiple/overlay 1D plots (using lattice/ggplot2)
- box plots: `boxplot(pm25 ~ region, data = pollution, col = "red")`



- histogram:
  - `par(mfrow = c(2, 1), mar = c(4, 4, 2, 1))` -> set margin
  - `hist(subset(pollution, region == "east")$pm25, col = "green")` -> first histogram
  - `hist(subset(pollution, region == "west")$pm25, col = "green")` -> second histogram



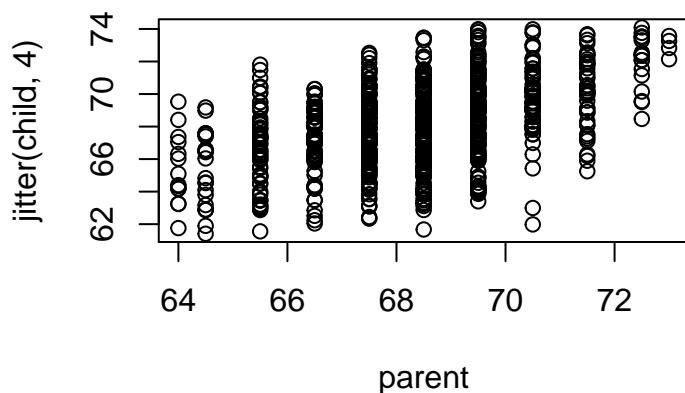
- scatterplot
  - `with(pollution, plot(latitude, pm25, col = region))`
  - `abline(h = 12, lwd = 2, lty = 2)` -> plots horizontal dotted line
  - `plot(jitter(child, 4)~parent, galton)` -> spreads out data points at the same position to simulate measurement error/make high frequency more visible



```

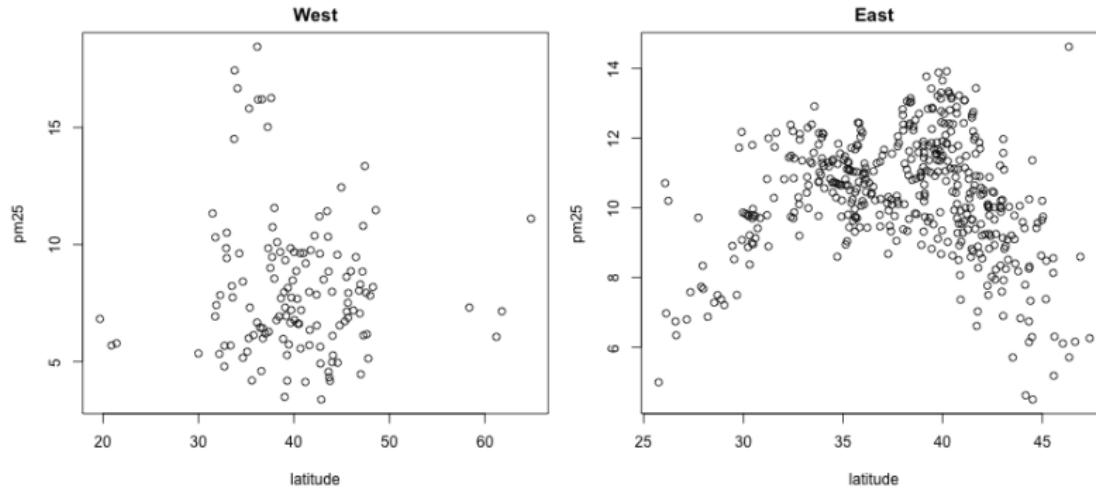
## Loading required package: MASS
## Loading required package: HistData
## Loading required package: Hmisc
## Loading required package: lattice
## Loading required package: survival
## Loading required package: splines
## Loading required package: Formula
##
## Attaching package: 'Hmisc'
##
## The following objects are masked from 'package:base':
## 
##     format.pval, round.POSIXt, trunc.POSIXt, units
## 
## 
## Attaching package: 'UsingR'
##
## The following object is masked from 'package:survival':
## 
##     cancer

```



- multiple scatter plots

- `par(mfrow = c(1, 2), mar = c(5, 4, 2, 1))` –> sets margins
- `with(subset(pollution, region == "west"), plot(latitude, pm25, main = "West"))` –> left scatterplot
- `with(subset(pollution, region == "east"), plot(latitude, pm25, main = "East"))` –> right scatterplot



## Process of Making a Plot/Considerations

- where will plot be made? screen or file?
- how will plot be used? viewing on screen/web browser/print/presentation?
- large amount of data vs few points?
- need to be able to dynamically resize?
- **plotting system:** base, lattice, ggplot2?

## Base Plotting

- blank canvas, “artist’s palette”, start with `plot` function
- annotations - text, lines, points, axis
- convenient, but cannot go back when started (need to plan ahead)
- everything need to be manually set carefully to be able to achieve the desired effect (margins)
- core plotting/graphics engine in R encapsulated in the following
  - **graphics:** plotting functions for base graphing system (`plot`, `hist`, `boxplot`, `tex`)
  - **grDevices:** contains all the code implementing the various graphics devices (`x11`, `PDF`, `PostScript`, `PNG`, etc)
- **Two phase:** initialize, annotate
- calling `plot(x, y)` or `hist(x)` will launch a graphics device and draw a plot on device
  - if no argument specified, default called
  - parameters documented in “`?par`”
  - **Note:** it is sometimes necessary to convert column/variable to factor to make plotting easier
    - \* `airquality <- transform(airquality, Month = factor(month))`

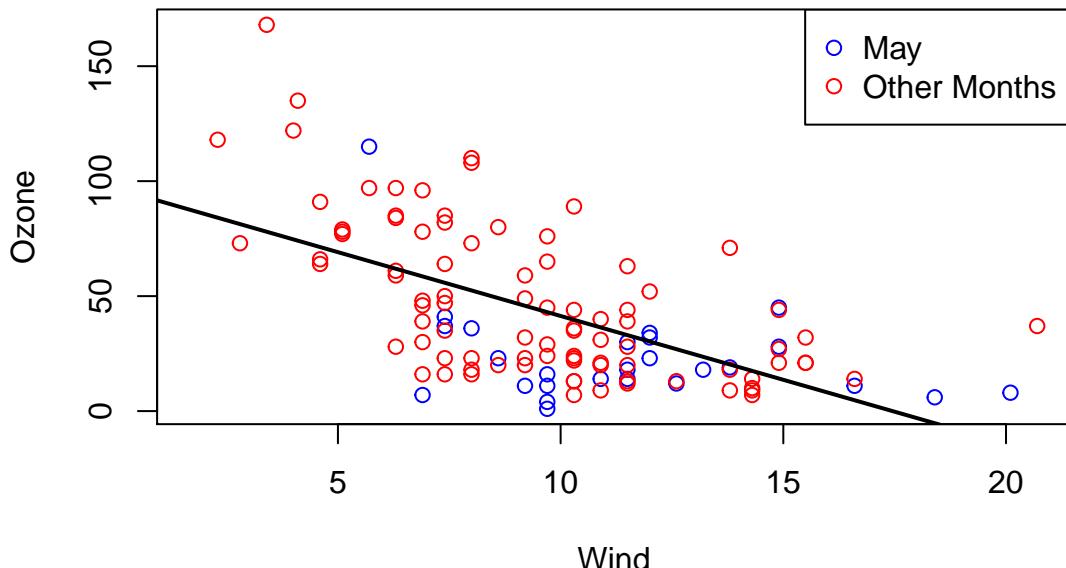
## Base Graphics Functions and Parameters

- **arguments**
  - `pch`: plotting symbol (default = open circle)
  - `lty`: line type (default is solid)
    - \* 0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash
  - `lwd`: line width (integer)
  - `col`: plotting color (number string or hexcode, `colors()` returns vector of colors)
  - `xlab`, `ylab`: x-y label character strings
  - `cex`: numerical value giving the amount by which plotting text/symbols should be magnified relative to the default
    - \* `cex = 0.15 * variable`: plot size as an additional variable
- `par()` function → specifies global graphics parameters, affects all plots in an R session (can be overridden)
  - `las`: orientation of axis labels
  - `bg`: background color
  - `mar`: margin size (order = bottom left top right)
  - `oma`: outer margin size (default = 0 for all sides)
  - `mfrow`: number of plots per row, column (plots are filled row-wise)
  - `mfcoll`: number of plots per row, column (plots are filled column-wise)
  - can verify all above parameters by calling `par("parameter")`
- **plotting functions**
  - `lines`: adds lines to a plot, given a vector of x values and corresponding vector of y values
  - `points`: adds a point to the plot
  - `text`: add text labels to a plot using specified x,y coordinates
  - `title`: add annotations to x,y axis labels, title, subtitles, outer margin
  - `mtext`: add arbitrary text to margins (inner or outer) of plot
  - `axis`: specify axis ticks

## Base Plot Example

```
library(datasets)
# type ="n" sets up the plot and does not fill it with data
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City", type = "n"))
# subsets of data are plotted here using different colors
with(subset(airquality, Month == 5), points(Wind, Ozone, col = "blue"))
with(subset(airquality, Month != 5), points(Wind, Ozone, col = "red"))
legend("topright", pch = 1, col = c("blue", "red"), legend = c("May", "Other Months"))
model <- lm(Ozone ~ Wind, airquality)
# regression line is produced here
abline(model, lwd = 2)
```

## Ozone and Wind in New York City

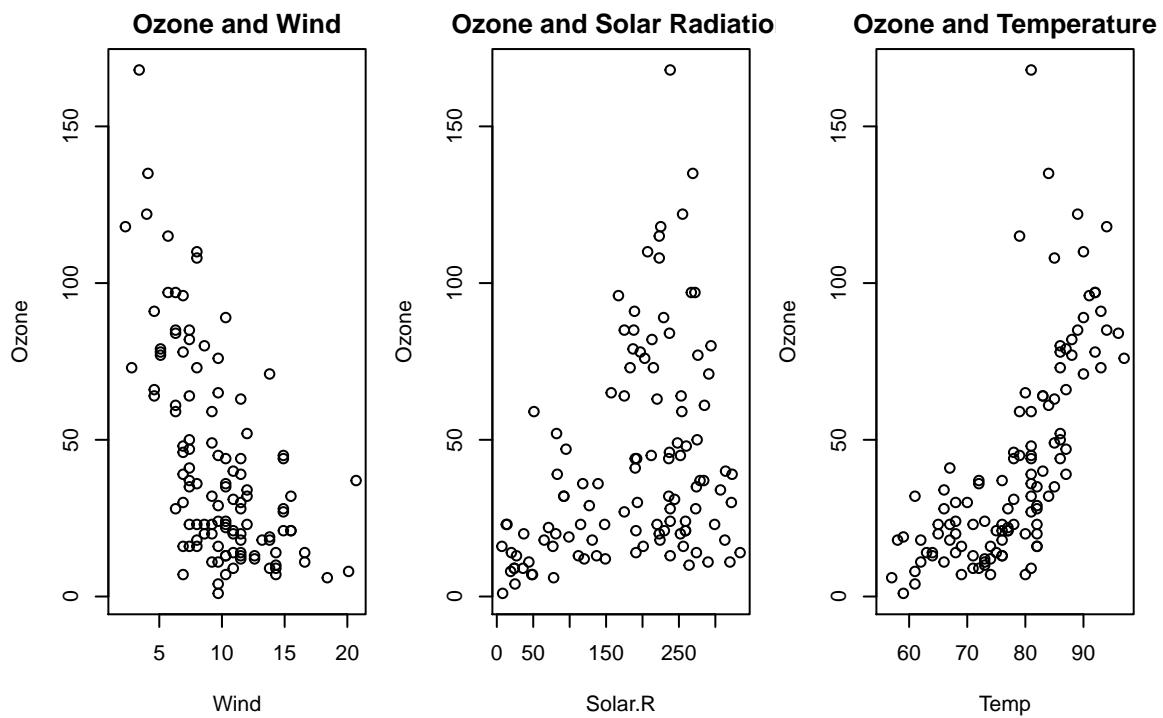


### Multiple Plot Example

- Note: typing `example(points)` in R will launch a demo of base plotting system and may provide some helpful tips on graphing

```
# this expression sets up a plot with 1 row 3 columns, sets the margin and outer margins
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1), oma = c(0, 0, 2, 0))
with(airquality, {
  # here three plots are filled in with their respective titles
  plot(Wind, Ozone, main = "Ozone and Wind")
  plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
  plot(Temp, Ozone, main = "Ozone and Temperature")
  # this adds a line of text in the outer margin*
  mtext("Ozone and Weather in New York City", outer = TRUE)
})
```

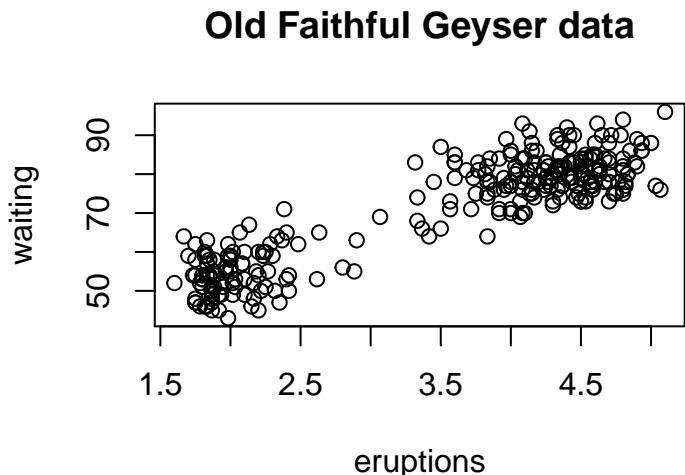
### Ozone and Weather in New York City



## Graphics Device

- A graphics device is something where you can make a plot appear
  - **window on screen** (screen device) <— quick visualizations and exploratory analysis
  - **pdf** (file device) <— plots that may be printed out or incorporated in to document
  - **PNG/JPEG** (file device) <— plots that may be printed out or incorporated in to document
  - **scalable vector graphics** (SVG)
- When a plot is created in R, it has to be sent to a graphics device
- **Most common is screen device**
  - `quartz()` on Mac, `windows()` on Windows, `x11()` on Unix/Linux
  - `?Devices` = lists devices found
- **Plot creation**
  - screen device
    - \* call plot/xplot/qplot → plot appears on screen device → annotate as necessary → use
  - file devices
    - \* explicitly call graphics device → plotting function to make plot (write to file) → annotate as necessary → explicitly close graphics device with `dev.off()`
- **Graphics File Devices**
  - **Vector Formats** (good for line drawings/plots w/ solid colors, a modest number of points)
    - \* **pdf**: useful for line type graphics, resizes well, usually portable, not efficient if too many points
    - \* **svg**: XML based scalable vector graphics, support animation and interactivity, web based
    - \* **win.metafile**: Windows metafile format
    - \* **postscript**: older format, resizes well, usually portable, can create encapsulated postscript file, Windows often don't have postscript viewer (postscript = predecessor of PDF)
  - **Bitmap Formats** (good for plots w/ large number of points, natural scenes/webbased plots)
    - \* **png**: Portable Network Graphics, good for line drawings/image with solid colors, uses lossless compression, most web browsers read this natively, good for plotting a lot of data points, does not resize well
    - \* **JPEG**: good for photographs/natural scenes/gradient colors, size efficient, uses lossy compression, good for plotting many points, does not resize well, can be read by almost any computer/browser, not great for line drawings (aliasing on edges)
    - \* **tiff**: common bitmap format supports lossless compression
    - \* **bmp**: native Windows bitmapped format
- **Multiple Open Graphics Devices**
  - possible to open multiple graphics devices (screen, file, or both)
  - plotting occurs only one device at a time
  - `dev.cur()` = returns the currently active device
  - every open graphics device is assigned an integer  $\geq 2$
  - `dev.set(<integer>)` = change the active graphics device  $<\text{integer}>$  = number associated with the graphics device you want to switch to
- **Copying plots**
  - `dev.copy()` = copy a plot from one device to another
  - `dev.copy2pdf()` = specifically for copying to PDF files
  - **Note:** *Copying a plot is not an exact operation, so the result may not be identical to the original*
  - **example**

```
## Create plot on screen device
with(faithful, plot(eruptions, waiting))
## Add a main title
title(main = "Old Faithful Geyser data")
```



```
## Copy my plot to a PNG file
dev.copy(png, file = "geyserplot.png")
## Don't forget to close the PNG device!
dev.off()
```

## **lattice Plotting System [library(lattice)]**

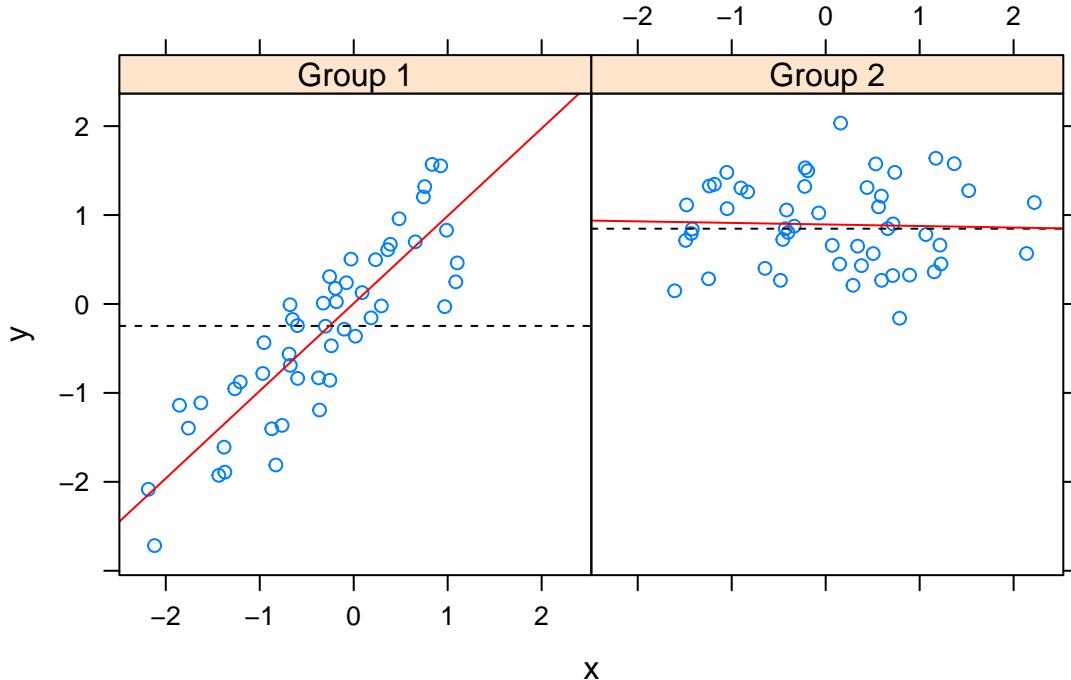
- implemented using the **lattice** and **grid** packages
  - **lattice** package → contains code for producing **Trellis** graphics (independent from base graphics system)
  - **grid** package → implements the graphing system; lattice build on top of grid
- all plotting and annotation is done with **single function call**
  - margins/spacing/labels set automatically for entire plot, good for putting multiple on the screen
  - good for conditioning plots → examining same plots over different conditions how y changes vs x across different levels of z
  - **panel** functions can be specified/customized to modify the subplots
- lattice graphics functions return an object of class “trellis”, whereas base graphics functions plot data directly to graphics device
  - print methods for lattice functions actually plots the data on graphics device
  - trellis objects are auto-printed
  - **trellis.par.set()** → can be used to set global graphic parameters for all trellis objects
- hard to annotate, awkward to specify entire plot in one function call
- cannot add to plot once created, panel/subscript functions hard to prepare

## **lattice Functions and Parameters**

- **Functions**
  - **xyplot()** → main function for creating scatterplots
  - **bwplot()** → box and whiskers plots (box plots)
  - **histogram()** → histograms
  - **stripplot()** → box plot with actual points
  - **dotplot()** → plot dots on “violin strings”
  - **splom()** → scatterplot matrix (like pairs() in base plotting system)
  - **levelplot()/contourplot()** → plotting image data
- **Arguments** for **xyplot(y ~ x | f \* g, data, layout, panel)**
  - default blue open circles for data points
  - formula notation is used here (~) → left hand side is the y-axis variable, and the right hand side is the x-axis variable
  - **f/g** = conditioning/categorical variables (optional)
    - \* basically creates multi-panelled plots (for different factor levels)
    - \* \* indicates interaction between two variables
    - \* intuitively, the xyplot displays a graph between x and y for every level of f and g
  - **data** = the data frame/list from which the variables should be looked up
    - \* if nothing is passed, the parent frame is used (searching for variables in the workspace)
    - \* if no other arguments are passed, defaults will be used
  - **layout** = specifies how the different plots will appear
    - \* **layout = c(5, 1)** → produces 5 subplots in a horizontal fashion
    - \* padding/spacing/margin automatically set
  - [optional] **panel** function can be added to control what is plotted inside each panel of the plot
    - \* **panel** functions receive x/y coordinates of the data points in their panel (along with any additional arguments)
    - \* **?panel.xyplot** → brings up documentation for the panel functions
    - \* **Note:** no base plot functions can be used for lattice plots

## **lattice Example**

```
library(lattice)
set.seed(10)
x <- rnorm(100)
f <- rep(0:1, each = 50)
y <- x + f - f * x + rnorm(100, sd = 0.5)
f <- factor(f, labels = c("Group 1", "Group 2"))
## Plot with 2 panels with custom panel function
xyplot(y ~ x | f, panel = function(x, y, ...) {
  # call the default panel function for xyplot
  panel.xyplot(x, y, ...)
  # adds a horizontal line at the median
  panel.abline(h = median(y), lty = 2)
  # overlays a simple linear regression line
  panel.lmline(x, y, col = 2)
})
```



## ggplot2 Plotting System [library(ggplot2)]

- implementation of Grammar of Graphics by Leland Wilkinson, written by Hadley Wickham (created RStudio)
  - “In brief, the grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars). The plot may also contain statistical transformations of the data and is drawn on a specific coordinate system”
- grammar graphics plot, splits the different between base and lattice systems
- automatically sets spacings/text/tiles but also allows annotations to be added
- default makes a lot of choices, but still customizable

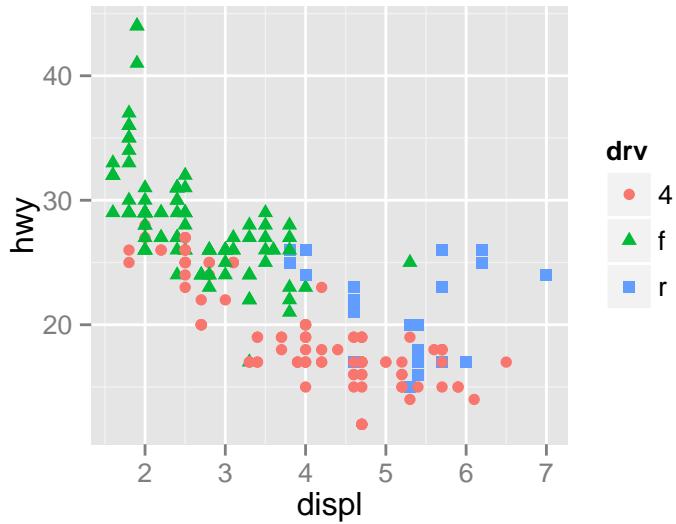
## ggplot2 Functions and Parameters

- **basic components** of a ggplot2 graphic
  - **data frame** -> source of data
  - **aesthetic mappings** -> how data are mapped to color/size (x vs y)
  - **geoms** -> geometric objects like points/lines/shapes to put on page
  - **facets** -> conditional plots using factor variables/multiple panels
  - **stats** -> statistical transformations like binning/quantiles/smoothing
  - **scales** -> scale aesthetic map uses (i.e. male = red, female = blue)
  - **coordinate system** -> system in which data are plotted
- qplot(x, y, data , color, geom) -> quick plot, analogous to base system's plot() function
  - **default style**: gray background, white gridlines, x and y labels automatic, and solid black circles for data points
  - data always comes from data frame (in unspecified, function will look for data in workspace)
  - plots are made up of aesthetics (size, shape, color) and geoms (points, lines)
  - **Note:** capable of producing quick graphics, but difficult to customize in detail
- **factor variables**: important for graphing subsets of data -> they should be labelled with specific information, and not just 1, 2, 3
  - **color = factor1** -> use the factor variable to display subsets of data in different colors on the same plot (legend automatically generated)
  - **shape = factor2** -> use the factor variable to display subsets of data in different shapes on the same plot (legend automatically generated)
  - **example**

```
library(ggplot2)
```

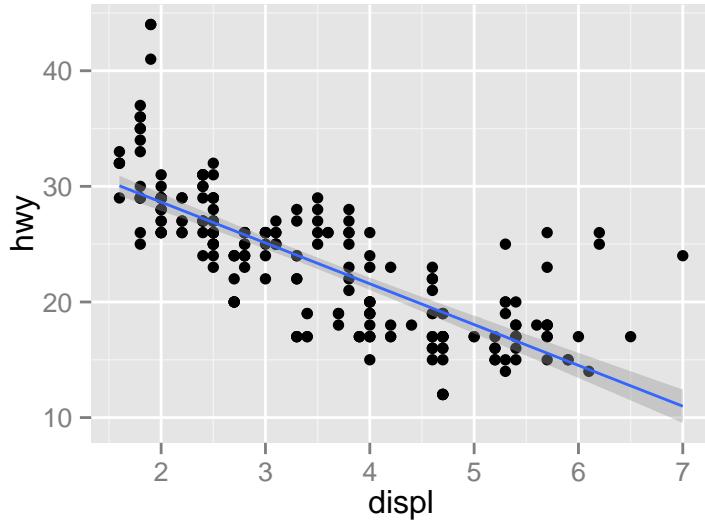
```
##  
## Attaching package: 'ggplot2'  
##  
## The following object is masked from 'package:UsingR':  
##  
##     movies
```

```
qplot(displ, hwy, data = mpg, color = drv, shape = drv)
```



- **adding statistics:** `geom = c("points", "smooth")` -> add a smoother/“low S”
  - “points” plots the data themselves, “smooth” plots a smooth mean line in blue with an area of 95% confidence interval shaded in dark gray
  - `method = "lm"` -> additional argument method can be specified to create different lines/confidence intervals
    - \* `lm` = linear regression
  - **example**

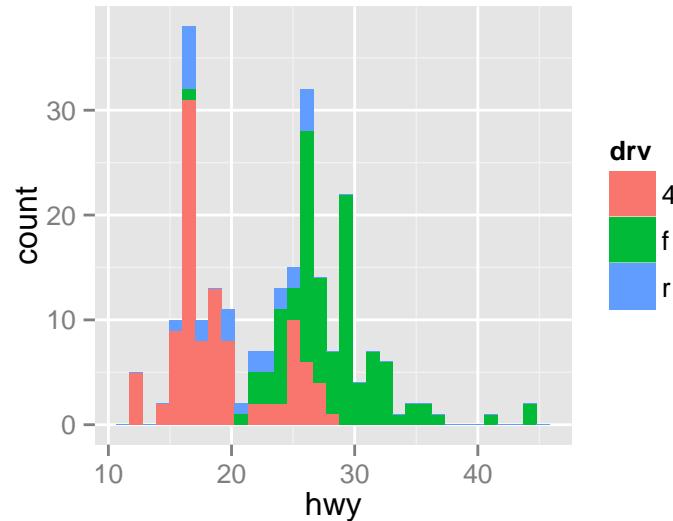
```
qplot(displ, hwy, data = mpg, geom = c("point", "smooth"), method="lm")
```



- **histograms:** if only one value is specified, a histogram is produced
  - `fill = factor1` -> can be used to fill the histogram with different colors for the subsets (legend automatically generated)
  - **example**

```
qplot(hwy, data = mpg, fill = drv)
```

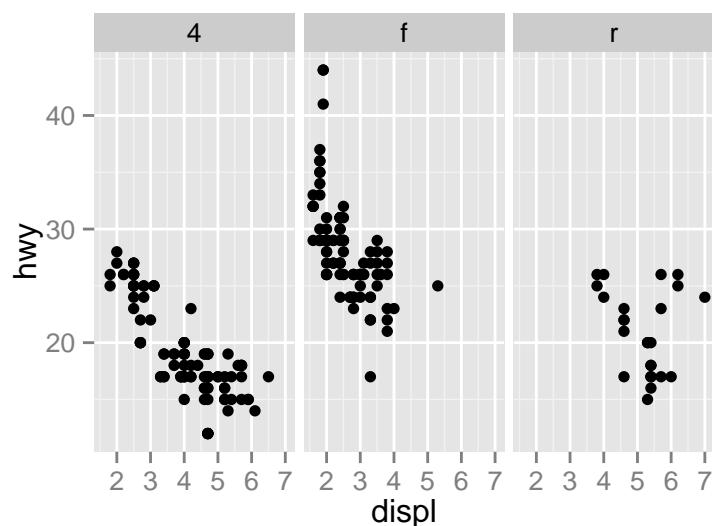
```
## stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```



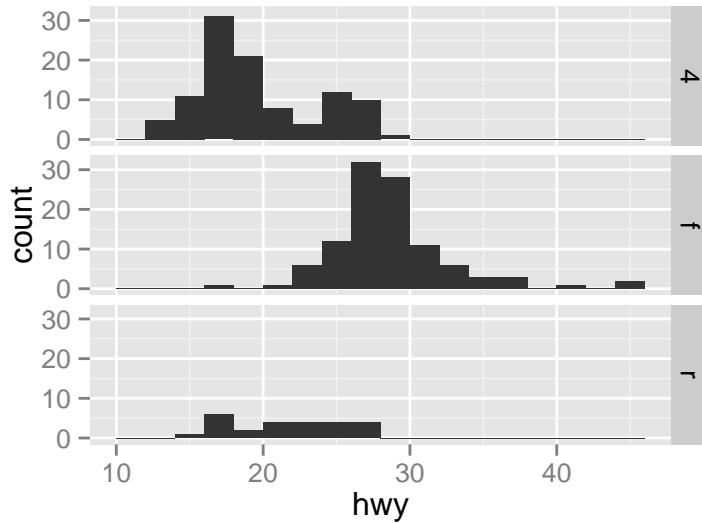
- **facets:** similar to panels in lattice, split data according to factor variables

- `facets = rows ~ columns` –> produce different subplots by factor variables specified (rows/columns)
- `".."` indicates there are no addition row or column
- `facets = . ~ columns` –> creates 1 by col subplots
- `facets = row ~ .` –> creates row by 1 subplots
- labels get generated automatically based on factor variable values
- *example*

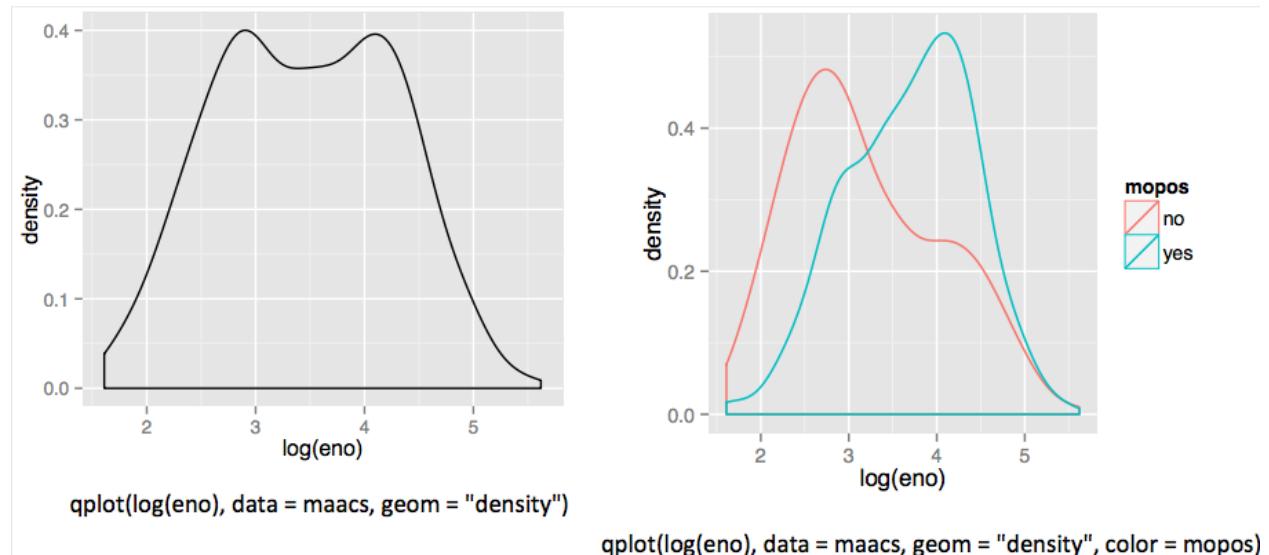
```
qplot(displ, hwy, data = mpg, facets = . ~ drv)
```



```
qplot(hwy, data = mpg, facets = drv ~ ., binwidth = 2)
```

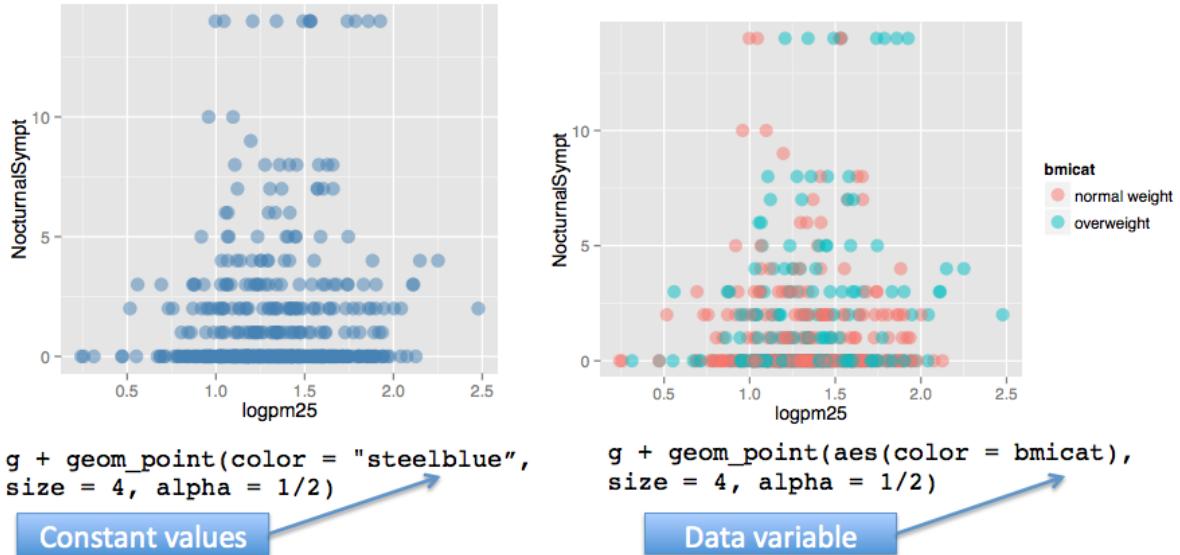


- **density smooth:** smooths the histograms into a line tracing its shape
  - `geom = "density"` –> replaces the default scatterplot with density smooth curve
  - *example*



- **ggplot()**
  - built up in layers/modularly (similar to base plotting system)
    - \* `data` –> overlay summary –> metadata/annotation
  - `g <- ggplot(data, aes(var1, var2))`
    - \* initiates call to `ggplot` and specifies the data frame that will be used
    - \* `aes(var1, var2)` –> specifies aesthetic mapping, or `var1 = x` variable, and `var2 = y` variable
    - \* `summary(g)` –> displays summary of `ggplot` object

- \* `print(g)` -> returns error ("no layer on plot") which means the plot does know how to draw the data yet
- `g + geom_point()` -> takes information from g object and produces scatter plot
- `+ geom_smooth()` -> adds low S mean curve with confidence interval
  - \* `method = "lm"` -> changes the smooth curve to be linear regression
  - \* `size = 4, linetype = 3` -> can be specified to change the size/style of the line
  - \* `se = FALSE` -> turns off confidence interval
- `+ facet_grid(row ~ col)` -> splits data into subplots by factor variables (see facets from qplot())
  - \* conditioning on continuous variables is possible through cutting/making a new categorical variable
  - \* `cutPts <- quantiles(df$cVar, seq(0, 1, length=4), na.rm = TRUE)` -> creates quantiles where the continuous variable will be cut
    - `seq(0, 1, length=4)` -> creates 4 quantile points
    - `na.rm = TRUE` -> removes all NA values
  - \* `df$newFactor <- cut(df$cVar, cutPts)` -> creates new categorical/factor variable by using the cutpoints
    - creates n-1 ranges from n points -> in this case 3
- ***annotations:***
  - \* `xlab(), ylab(), labs(), ggtitle()` -> for labels and titles
    - `labs(x = expression("log " * PM[2.5]), y = "Nocturnal")` -> specifies x and y labels
    - `expression()` -> used to produce mathematical expressions
  - \* `geom` functions -> many options to modify
  - \* `theme()` -> for global changes in presentation
    - *example:* `theme(legend.position = "none")`
  - \* two standard themes defined: `theme_gray()` and `theme_bw()`
  - \* `base_family = "Times"` -> changes font to Times
- ***aesthetics***
  - \* `+ geom_point(color, size, alpha)` -> specifies how the points are supposed to be plotted on the graph (style)
    - *Note:* this translates to `geom_line()`/other forms of plots
    - `color = "steelblue"` -> specifies color of the data points
    - `aes(color = var1)` -> wrapping color argument this way allows a factor variable to be assigned to the data points, thus subsetting it with different colors based on factor variable values
    - `size = 4` -> specifies size of the data points
    - `alpha = 0.5` -> specifies transparency of the data points
  - \* *example*



```
g + geom_point(color = "steelblue",
size = 4, alpha = 1/2)
```

Constant values

```
g + geom_point(aes(color = bmicat),
size = 4, alpha = 1/2)
```

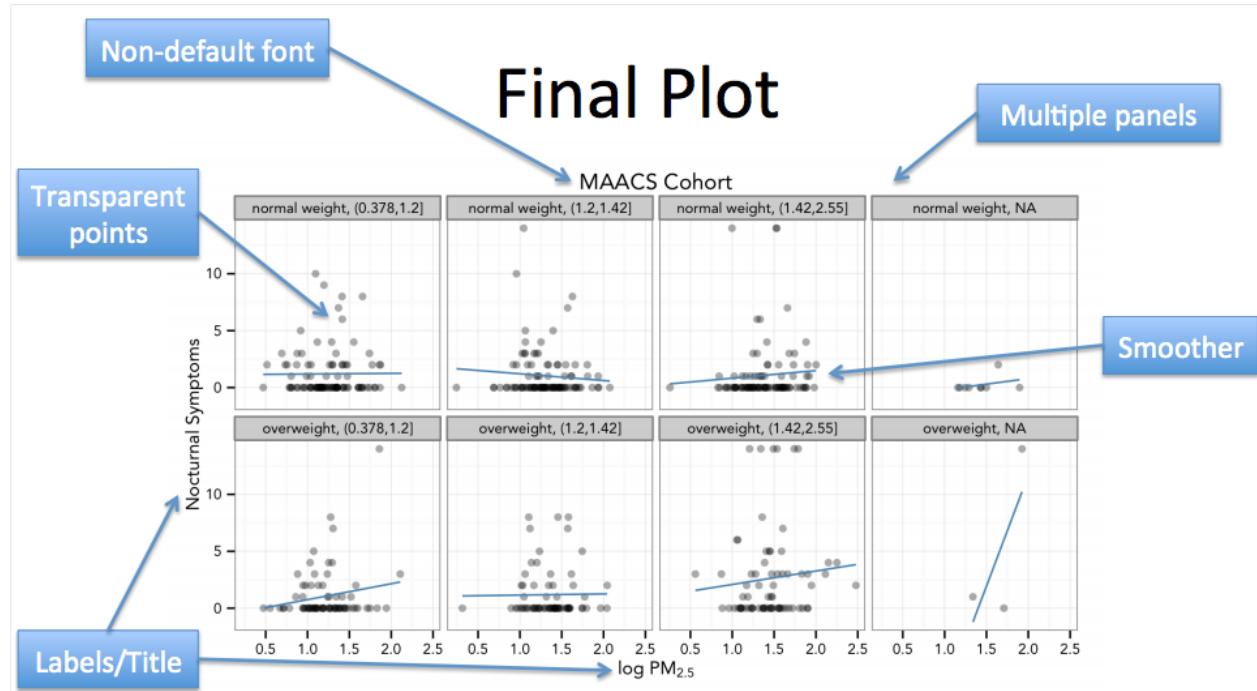
Data variable

#### - axis limits

- \* + ylim(-3, 3) -> limits the range of y variable to a specific range
  - Note: ggplot will exclude (not plot) points that fall outside of this range (outliers), potentially leaving gaps in plot
- \* + coord\_cartesian(ylim(-3, 3)) -> this will limit the visible range but plot all points of the data

## ggplot2 Comprehensive Example

```
# initiates ggplot
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
g + geom_point(alpha = 1/3)                                # adds points
+ facet_wrap(bmicat ~ no2dec, nrow = 2, ncol = 4)        # make panels
+ geom_smooth(method="lm", se=FALSE, col="steelblue")     # adds smoother
+ theme_bw(base_family = "Avenir", base_size = 10)       # change theme
+ labs(x = expression("log " * PM[2.5]))                 # add labels
+ labs(y = "Nocturnal Symptoms")                          # add labels
+ labs(title = "MAACS Cohort")
```

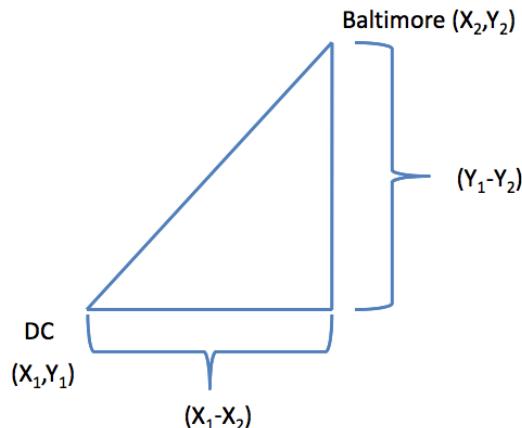


## Hierarchical Clustering

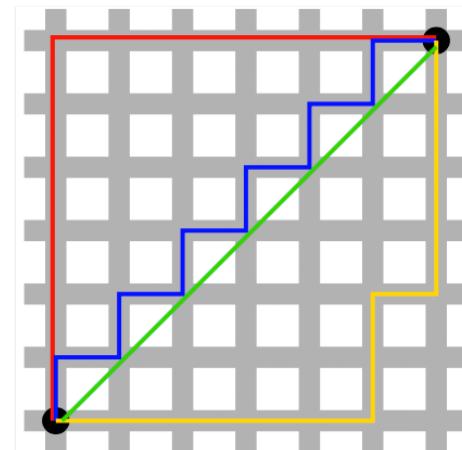
- useful for visualizing high dimensional data, organizes things that are close into groups
- agglomerative approach** (most common) — bottom up
  - start with data
  - find closest pairs, put them together (create “super point” and remove original data)
  - find the next closest
  - repeat = yields a tree showing order of merging (dendrogram)

— requires

  - \* **merging approach:** how to merge two points
  - \* **distance metric:** calculating distance between two points
  - \* **continuous - Euclidean distance**  $\rightarrow \sqrt{(A_1 - A_2)^2 + (B_1 - B_2)^2 + \dots + (Z_1 - Z_2)^2}$
  - \* **continuous - correlation similarity**  $\rightarrow$  how correlated two data points are
  - \* **binary - manhattan distance** (“city block distance”)  $\rightarrow |A_1 - A_2| + |B_1 - B_2| + \dots + |Z_1 - Z_2|$



Euclidean Distance



Binary Distance

### Procedure for Constructing Hierarchical Clusters (`hclust` function)

- calculate all pair wise distances between all points to see which points are closest together
  - `dist(data.frame(x=x, y=y))`  $\rightarrow$  returns pair wise distances for all of the (x,y) coordinates
  - Note:** `dist()` function uses Euclidean distance by default
- group two closest points from the calculated distances and merge them to a single point
- find the next two closest points and merge them, and repeat
- order of clustering is shown in the dendrogram

### Approaches for Merging Points/Clusters

- the approach is specified in the argument `method = "complete"` or `"average"` in `hclust()` function
- average linkage**  $\rightarrow$  taking average of the x and y coordinates for both points/clusters (center of mass effectively)

- *complete linkage* -> to measure distance of two clusters, take the two points in the clusters that are the furthest apart
- **Note:** two approaches may produce different results so it's a good idea to use both approaches to validate results

## Characteristics of Hierarchical Clustering Algorithms

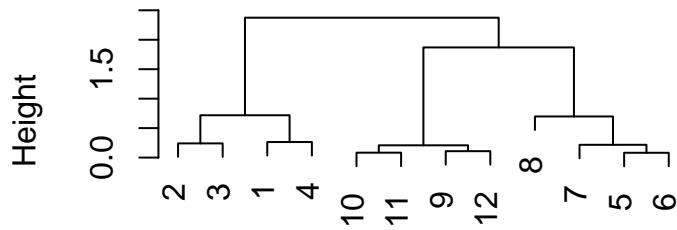
- clustering result/plot maybe *unstable*
  - changing few points/outliers could lead to large changes
  - change different distance metrics to see how sensitive the clustering is
  - change merging strategy
  - scaling of variables could affect the clustering (if one unit/measurement is much larger than another)
- *deterministic* -> running the hclust function with same parameters and the same data will produce the same plot
- determining how many clusters there are (where to cut) may *not always be clear*
- *primarily used for exploratory data analysis*, to see over all pattern in data if there is any at all

## hclust Function and Example

- `hh <- hclust(dist(dataFrame))` function -> produces a hierarchical cluster object based on pairwise distances from a data frame of x and y values
  - `dist()` -> defaults to Euclidean, calculates the distance/similarity between two observations; when applied to a data frame, the function applies the  $\sqrt{(A_1 - A_2)^2 + (B_1 - B_2)^2 + \dots + (Z_1 - Z_2)^2}$  formula to every pair of rows of data to construct a matrix of distances between the rows
    - \* order of the hierarchical cluster is derived from the distance
  - `plot(hh)` -> plots the dendrogram
  - automatically sorts column and row according to cluster
  - `names(hh)` -> returns all parameters of the hclust object
    - \* `hh$order` -> returns the order of the rows/clusters from the dendrogram
    - \* `hh$dist.method` -> returns method for calculating distance/similarity
- **Note:** dendrogram that gets generated **DOES NOT** show how many clusters there are, so cutting (at 2.0 level for example) must be done to determine number of clusters — must be a convenient and sensible point
- **hclust Example**

```
set.seed(1234)
x <- rnorm(12,mean=rep(1:3,each=4),sd=0.2)
y <- rnorm(12,mean=rep(c(1,2,1),each=4),sd=0.2)
dataFrame <- data.frame(x=x,y=y)
distxy <- dist(dataFrame)
hClustering <- hclust(distxy)
plot(hClustering)
```

## Cluster Dendrogram



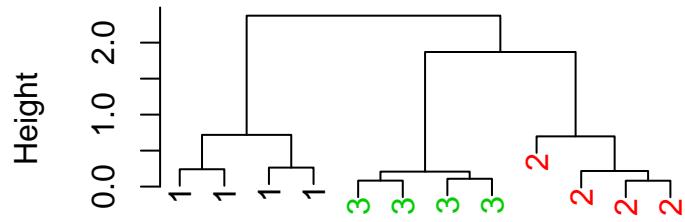
```
distxy
hclust (*, "complete")
```

### myplcclust Function and Example

- Note: `myplcclust` = a function to plot `hclust` objects in color (clusters labelled 1 2 3 etc.), but must know how many clusters there are initially

```
myplcclust <- function(hclust, lab = hclust$labels,
lab.col = rep(1, length(hclust$labels)), hang = 0.1, ...) {
## modifiction of plclust for plotting hclust objects *in colour!* Copyright
## Eva KF Chan 2009 Arguments: hclust: hclust object lab: a character vector
## of labels of the leaves of the tree lab.col: colour for the labels;
## NA=default device foreground colour hang: as in hclust & plclust Side
## effect: A display of hierarchical cluster with coloured leaf labels.
y <- rep(hclust$height, 2)
x <- as.numeric(hclust$merge)
y <- y[which(x < 0)]
x <- x[which(x < 0)]
x <- abs(x)
y <- y[order(x)]
x <- x[order(x)]
plot(hclust, labels = FALSE, hang = hang, ...)
text(x = x, y = y[hclust$order] - (max(hclust$height) * hang), labels = lab[hclust$order],
col = lab.col[hclust$order], srt = 90, adj = c(1, 0.5), xpd = NA, ...)
}
# example
dataFrame <- data.frame(x = x, y = y)
distxy <- dist(dataFrame)
hClustering <- hclust(distxy)
myplcclust(hClustering, lab = rep(1:3, each = 4), lab.col = rep(1:3, each = 4))
```

## Cluster Dendrogram

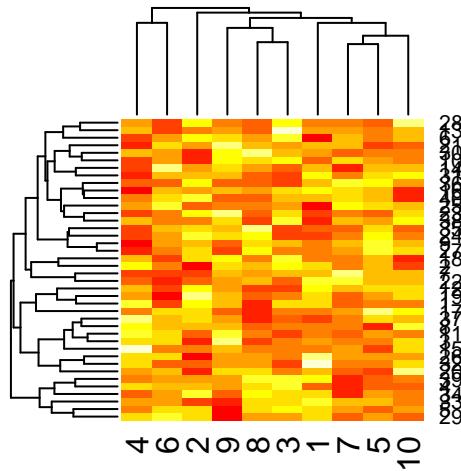


```
distxy  
hclust (*, "complete")
```

### heatmap Function and Example

- `heatmap(data.matrix)` function -> similar to `image(t(x))`
  - good for visualizing high-dimension matrix data, runs hierarchical analysis on rows and columns of table
  - yellow = high value, red = low value
  - **Note:** the input must be a numeric matrix, so `as.matrix(data.frame)` can be used to convert if necessary
- *example*

```
set.seed(12345)  
data <- matrix(rnorm(400), nrow = 40)  
heatmap(data)
```

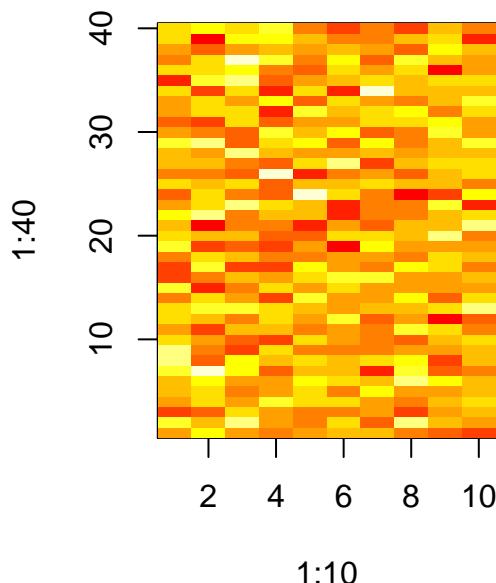


### image Function and Example

- `image(x, y, t(dataMatrix)[, nrow(dataMatrix):1])` -> produces similar color grid plot as the `heatmap()` without the dendograms

- `t(dataMatrix) [, nrow(dataMatrix)]`
- \* `t(dataMatrix)` -> transpose of `dataMatrix`, this is such that the plot will be displayed in the same fashion as the matrix (rows as values on the y axis and columns as values on the x axis)
  - *example* 40 x 10 matrix will have graph the 10 columns as x values and 40 rows as y values
- \* `[, nrow(dataMatrix)]` -> subsets the data frame in reverse column order; when combined with the `t()` function, it reorders the rows of data from 40 to 1, such that the data from the matrix is displayed in order from top to bottom
  - *Note:* without this statement the rows will be displayed in order from bottom to top, as that is in line with the positive y axis
- `x, y` -> used to specify the values displayed on the x and y axis
  - \* *Note:* must be in increasing order
- *example*

```
image(1:10, 1:40, t(data) [, nrow(data):1])
```



## K-means Clustering

- similar to hierarchical clustering, focuses on finding things that are close together
  - define close, groups, visualizing/interpreting grouping
- **partitioning approach**
  1. set number of clusters initially
  2. find centroids for each cluster
  3. assign points to the closest centroid
  4. recalculate centroid
  5. repeat = yields estimate of cluster centroids and which cluster each point belongs to
  - **requires**
    - \* distance metric
    - \* initial number of clusters
    - \* initial guess as to where the cluster centroids are

### Procedure for Constructing K-means Clusters (`kmeans` function)

1. choose three random points as the starting centroids
2. take each of the data points and assign it to the closest centroid (creating a cluster around each starting point)
3. take each cluster and recalculate the centroid (taking the mean) with its enclosed data points
4. repeat step 2 and 3 (reassign points to centroids and update centroid locations) until a stable result is achieved

- *example*

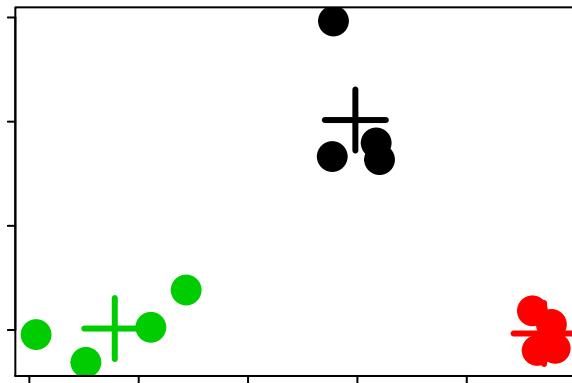
```
set.seed(1234)
x <- rnorm(12,mean=rep(1:3,each=4),sd=0.2)
y <- rnorm(12,mean=rep(c(1,2,1),each=4),sd=0.2)
dataFrame <- data.frame(x=x,y=y)
# specifies initial number of clusters to be 3
kmeansObj <- kmeans(dataFrame,centers=3)
names(kmeansObj)

## [1] "cluster"      "centers"       "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"

# returns cluster assignments
kmeansObj$cluster

## [1] 3 3 3 3 1 1 1 1 2 2 2 2

par(mar=rep(0.2,4))
plot(x,y,col=kmeansObj$cluster,pch=19,cex=2)
points(kmeansObj$centers,col=1:3,pch=3,cex=3,lwd=3)
```



### Characteristics of K-means Clustering Algorithms

- requires number of clusters initially
  - pick by eye/intuition
  - pick by cross validation/information theory, etc. [link]
- not deterministic (starting points chosen at random)
  - useful to run the algorithms a few times with different starting points to validate results

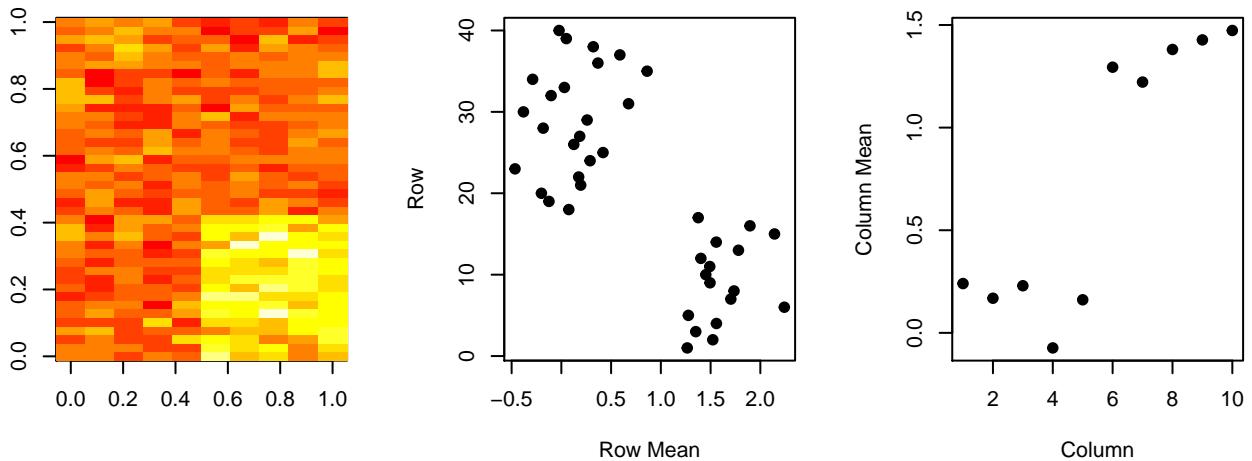
## Dimension Reduction

- two kinds of problems that relate to high-dimension dataset/matrix with many variables
  1. find a new set (smaller) of variables that are uncorrelated and explain as much variance of data as possible
    - normally many variables are not independent (i.e. height vs weight)
    - statistical problem, commonly solved with PCA
  2. find a lower rank matrix (best matrix created with fewer variables) that still explains the data
    - data compression problem, commonly solved SVD
- *example*
  - *Note: we are arbitrarily introduced pattern in data: we flip a coin and if the it is heads, we replace the row with [0, 0, 0, 0, 0, 3, 3, 3, 3, 3]*
  - here we plot the patterns in rows and columns (already sorted)

```

for(i in 1:40){
  # flip a coin
  coinFlip <- rbinom(1,size=1,prob=0.5)
  # if coin is heads add a common pattern to that row
  if(coinFlip){
    data[i,] <- data[i,] + rep(c(0,3),each=5)
  }
}
# hierarchical clustering
hh <- hclust(dist(data))
dataOrdered <- data[hh$order,]
# create 1 x 3 panel plot
par(mfrow=c(1,3))
# heat map (sorted)
image(t(dataOrdered)[,nrow(dataOrdered):1])
# row means (40 rows)
plot(rowMeans(dataOrdered),40:1,,xlab="Row Mean",ylab="Row",pch=19)
# column means (10 columns)
plot(colMeans(dataOrdered),xlab="Column",ylab="Column Mean",pch=19)

```



## Singular Value Decomposition (SVD)

- Let  $X$  = matrix which each variable in column (measurement) and each observation in row (subject)
- SVD in this case is a **matrix decomposition** process, in which  $X$  is divided into *three* separate matrices as follows:

$$X = UDV^T$$

- $U$  = left singular vector, orthogonal matrix (columns independent of each other)
- $D$  = singular values, diagonal matrix
- $V$  = right singular vector, orthogonal matrix (columns independent of each other)
- **Note:** *orthogonal implies that a matrix is always invertible [ $A^{-1} = A^T$ ] and that the product of the matrix and its transpose equals the identity matrix [ $AA^T = I$ ]*
  - \* when a orthogonal matrices,  $A$ , is multiplied by another matrix,  $B$ , it is effectively a linear transformation in that the length and angles of  $B$  are preserved
- **Note:** *diagonal implies that any value outside of the main diagonal ( $\nwarrow$ ) = 0*
  - \* example

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- **Note:** *scale of data matters for SVD/PCA (scaling the data may help), patterns detected maybe mixed together, and computation is intensive for these operations*

## Principal Components Analysis (PCA)

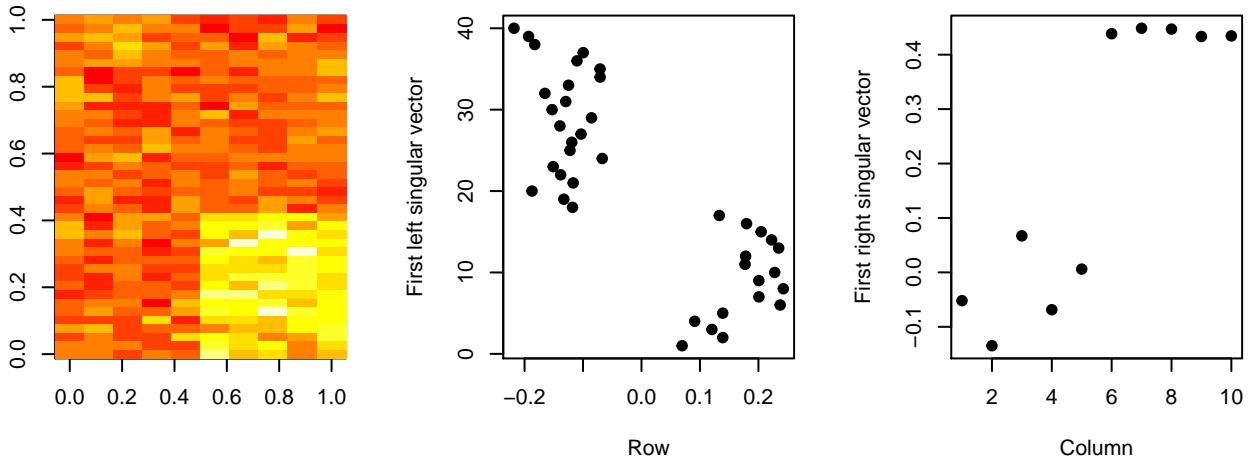
- first scale the variables and run SVD on normalized matrix
  - **scaling** = subtract each column by its mean and divide by its standard deviation
- **principal components** = the right singular values or the  $V$  matrix

## SVD and PCA Example

### • $U$ and $V$ Matrices

- `s <- svd(data)` = performs SVD on data ( $n \times m$  matrix) and splits it into  $u$ ,  $v$ , and  $d$  matrices
  - \* `s$u` =  $n \times m$  matrix  $\rightarrow$  horizontal variation
  - \* `s$d` =  $1 \times m$  vector  $\rightarrow$  vector of the singular/diagonal values
    - `diag(s$d)` =  $m \times m$  diagonal matrix
  - \* `s$v` =  $m \times m$  matrix  $\rightarrow$  vertical variation
  - \* `s$u %*% diag(s$d) %*% t(s$v)` returns the original data  $\rightarrow X = UDV^T$
- `scale(data)` = scales the original data by subtracting each datapoint by its column mean and dividing by its column standard deviation

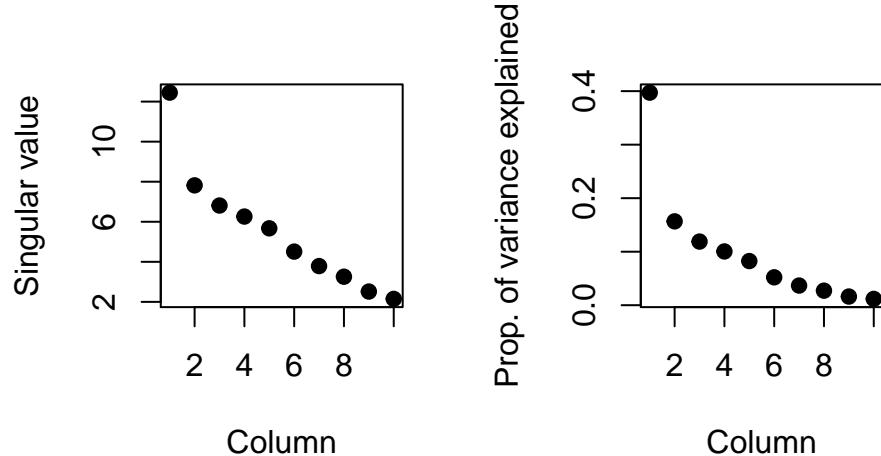
```
# running svd
svd1 <- svd(scale(dataOrdered))
# create 1 by 3 panel plot
par(mfrow=c(1,3))
# data heatmap (sorted)
image(t(dataOrdered)[,nrow(dataOrdered):1])
# U Matrix - first column
plot(svd1$u[,1],40:1,,xlab="Row",ylab="First left singular vector",pch=19)
# V vector - first column
plot(svd1$v[,1],xlab="Column",ylab="First right singular vector",pch=19)
```



- **D Matrix and Variance Explained**

- $d$  matrix ( $\$d$  vector) captures the singular values, or *variation in data that is explained by that particular component* (variable/column/dimension)
- **proportion of variance Explained** = converting the singular values to variance (square the values) and divide by the total variance (sum of the squared singular values)
  - \* effectively the same pattern as the singular values, just converted to percentage
  - \* in this case, the first component/dimension, which captures the shift in means (see previous plot) of SVD captures about 40% of the variation

```
# create 1 x 2 panel plot
par(mfrow=c(1,2))
# plot singular values
plot(svd1$d,xlab="Column",ylab="Singular value",pch=19)
# plot proportion of variance explained
plot(svd1$d^2/sum(svd1$d^2),xlab="Column",ylab="Prop. of variance explained",pch=19)
```

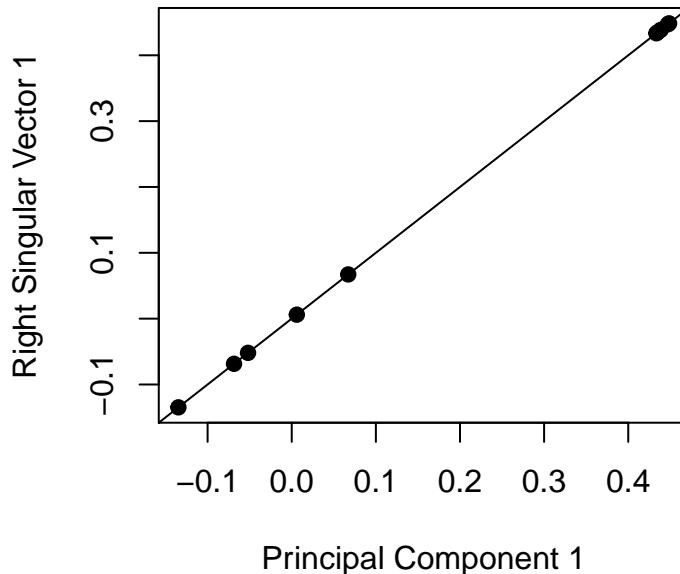


- **Relationship to PCA**

- `p <- prcomp(data, scale = TRUE)` = performs PCA on data specified
  - \* `scale = TRUE` = scales the data before performing PCA
  - \* returns `prcomp` object

- \* `summary(p)` = prints out the principal component's standard deviation, proportion of variance, and cumulative proportion
- PCA's rotation vectors are equivalent to their counterparts in the V matrix from the SVD

```
# SVD
svd1 <- svd(scale(dataOrdered))
# PCA
pca1 <- prcomp(dataOrdered,scale=TRUE)
# Plot the rotation from PCA (Principal Components) vs v vector from SVD
plot(pca1$rotation[,1],svd1$v[,1],pch=19,xlab="Principal Component 1",
      ylab="Right Singular Vector 1")
abline(c(0,1))
```



```
# summarize PCA
summary(pca1)
```

```
## Importance of components:
##          PC1     PC2     PC3     PC4     PC5     PC6     PC7
## Standard deviation 1.9930 1.2518 1.0905 1.0024 0.90836 0.72211 0.60630
## Proportion of Variance 0.3972 0.1567 0.1189 0.1005 0.08251 0.05214 0.03676
## Cumulative Proportion 0.3972 0.5539 0.6728 0.7733 0.85582 0.90797 0.94473
##          PC8     PC9     PC10
## Standard deviation 0.52145 0.40286 0.34423
## Proportion of Variance 0.02719 0.01623 0.01185
## Cumulative Proportion 0.97192 0.98815 1.00000
```

### • More Complex Patterns

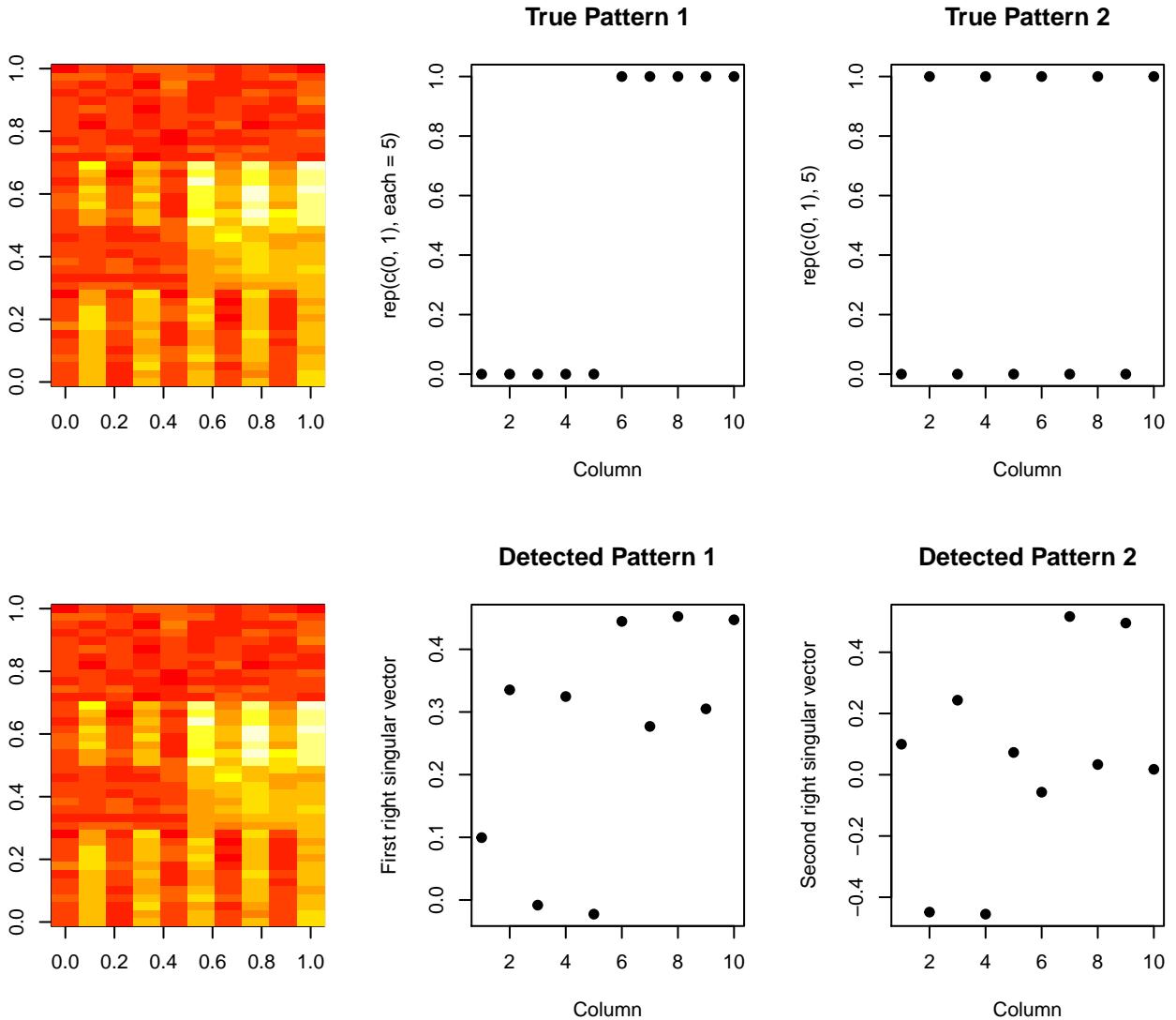
- SVD can be used to ***detect unknown patterns*** within the data (we rarely know the true distribution/pattern about the population we're analyzing)
- however, it may be hard to pinpoint exact patterns as the principal components may confound each other
  - \* in the example below, you can see that the two principal components that capture the most variation have both horizontal shifts and alternating patterns captured in them

```

set.seed(678910)
# setting pattern
data <- matrix(rnorm(400), nrow = 40)
for(i in 1:40){
  # flip a coin
  coinFlip1 <- rbinom(1, size=1, prob=0.5)
  coinFlip2 <- rbinom(1, size=1, prob=0.5)
  # if coin is heads add a common pattern to that row
  if(coinFlip1){
    data[i,] <- data[i,] + rep(c(0,5), each=5)
  }
  if(coinFlip2){
    data[i,] <- data[i,] + rep(c(0,5), 5)
  }
}
hh <- hclust(dist(data)); dataOrdered <- data[hh$order,]

# perform SVD
svd2 <- svd(scale(dataOrdered))
par(mfrow=c(2,3))
image(t(dataOrdered)[,nrow(dataOrdered):1])
plot(rep(c(0,1),each=5), pch=19, xlab="Column", main="True Pattern 1")
plot(rep(c(0,1),5), pch=19, xlab="Column", main="True Pattern 2")
image(t(dataOrdered)[,nrow(dataOrdered):1])
plot(svd2$v[,1], pch=19, xlab="Column", ylab="First right singular vector",
     main="Detected Pattern 1")
plot(svd2$v[,2], pch=19, xlab="Column", ylab="Second right singular vector",
     main="Detected Pattern 2")

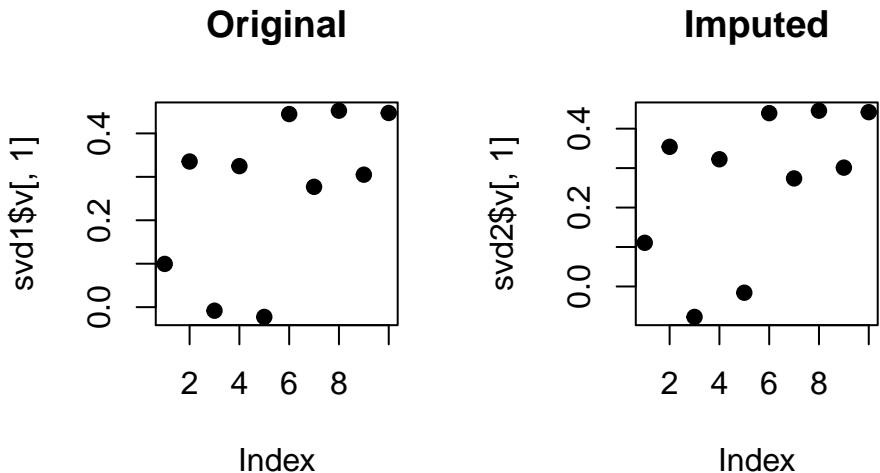
```



- Missing Data

- SVD cannot be performed on dataset with NA values
- `impute` package from [Bioconductor](#) can help approximate missing values from surrounding values
  - \* `impute.knn` function takes the missing row and imputes the data using the  $k$  nearest neighbors to that row
    - $k=10$  = default value (take the nearest 10 rows)

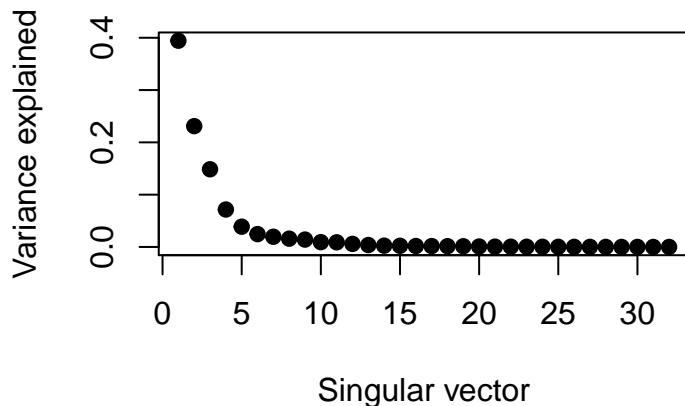
```
library(impute) ## Available from http://bioconductor.org
data2 <- data0rdered
# set random samples = NA
data2[sample(1:100,size=40,replace=FALSE)] <- NA
data2 <- impute.knn(data2)$data
svd1 <- svd(scale(data0rdered)); svd2 <- svd(scale(data2))
par(mfrow=c(1,2))
plot(svd1$v[,1],pch=19, main="Original")
plot(svd2$v[,1],pch=19, main="Imputed")
```



### Create Approximations/Data Compression

- SVD can be used to create lower rank representation, or compressed representation of data
- if we look at the variance explained plot below, *most of the variation* is explained by the *first few principal components*

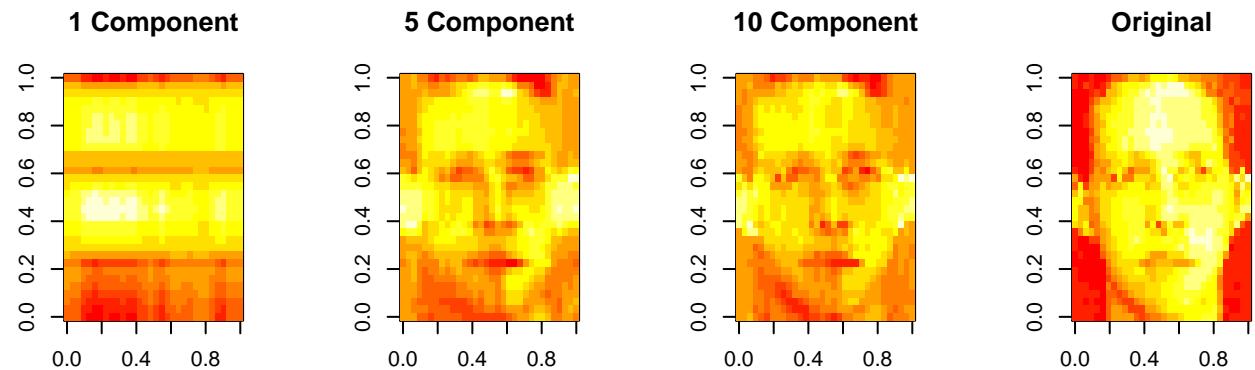
```
# load faceData
load("figures/face.rda")
# perform SVD
svd3 <- svd(scale(faceData))
plot(svd3$d^2/sum(svd3$d^2), pch=19, xlab="Singular vector", ylab="Variance explained")
```



- approximations can thus be created by taking the first few components and using matrix multiplication with the corresponding  $U$ ,  $V$ , and  $D$  components

```
approx1 <- svd3$u[,1] %*% t(svd3$v[,1]) * svd3$d[1]
approx5 <- svd3$u[,1:5] %*% diag(svd3$d[1:5]) %*% t(svd3$v[,1:5])
approx10 <- svd3$u[,1:10] %*% diag(svd3$d[1:10]) %*% t(svd3$v[,1:10])
# create 1 x 4 panel plot
par(mfrow=c(1,4))
# plot original facedata
image(t(approx1)[,nrow(approx1):1], main = "1 Component")
image(t(approx5)[,nrow(approx5):1], main = "5 Component")
```

```
image(t(approx10)[,nrow(approx10):1], main = "10 Component")
image(t(faceData)[,nrow(faceData):1], main = "Original")
```



## Color Packages in R Plots

- proper use of color can help convey the message by improving clarity/contrast of data presented
- default color schemes for most plots in R are fairly terrible, so some external packages are helpful

### grDevices Package

- `colors()` function = lists names of colors available in any plotting function
- **colorRamp function**
  - takes any set of colors and return a function that takes values between 0 and 1, indicating the extremes of the color palette (e.g. see the `gray` function)
  - `pal <- colorRamp(c("red", "blue"))` = defines a `colorRamp` function
  - `pal(0)` returns a 1 x 3 matrix containing values for RED, GREEN, and BLUE values that range from 0 to 255
  - `pal(seq(0, 1, len = 10))` returns a 10 x 3 matrix of 10 colors that range from RED to BLUE (two ends of spectrums defined in the object)
  - *example*

```
# define colorRamp function
pal <- colorRamp(c("red", "blue"))
# create a color
pal(0.67)
```

```
##      [,1] [,2]   [,3]
## [1,] 84.15    0 170.85
```

- **colorRampPalette function**
  - takes any set of colors and return a function that takes integer arguments and returns a vector of colors interpolating the palette (like `heat.colors` or `topo.colors`)
  - `pal <- colorRampPalette(c("red", "yellow"))` defines a `colorRampPalette` function
  - `pal(10)` returns 10 interpolated colors in hexadecimal format that range between the defined ends of spectrums
  - *example*

```
# define colorRampPalette function
pal <- colorRampPalette(c("red", "yellow"))
# create 10 colors
pal(10)
```

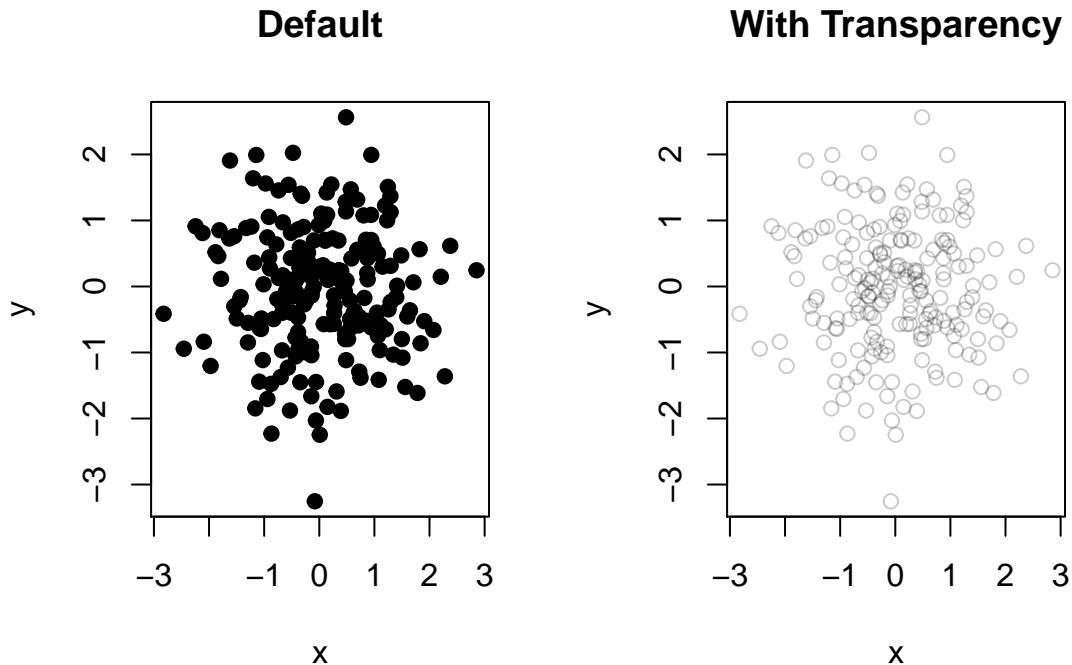
```
## [1] "#FF0000" "#FF1C00" "#FF3800" "#FF5500" "#FF7100" "#FF8D00" "#FFAA00"
## [8] "#FFC600" "#FFE200" "#FFFF00"
```

- **rgb function**
  - `red`, `green`, and `blue` arguments = values between 0 and 1
  - `alpha = 0.5` = transparency control, values between 0 and 1
  - returns hexadecimal string for color that can be used in `plot`/`image` commands
  - `colorspace` package can be used for different control over colors
  - *example*

```

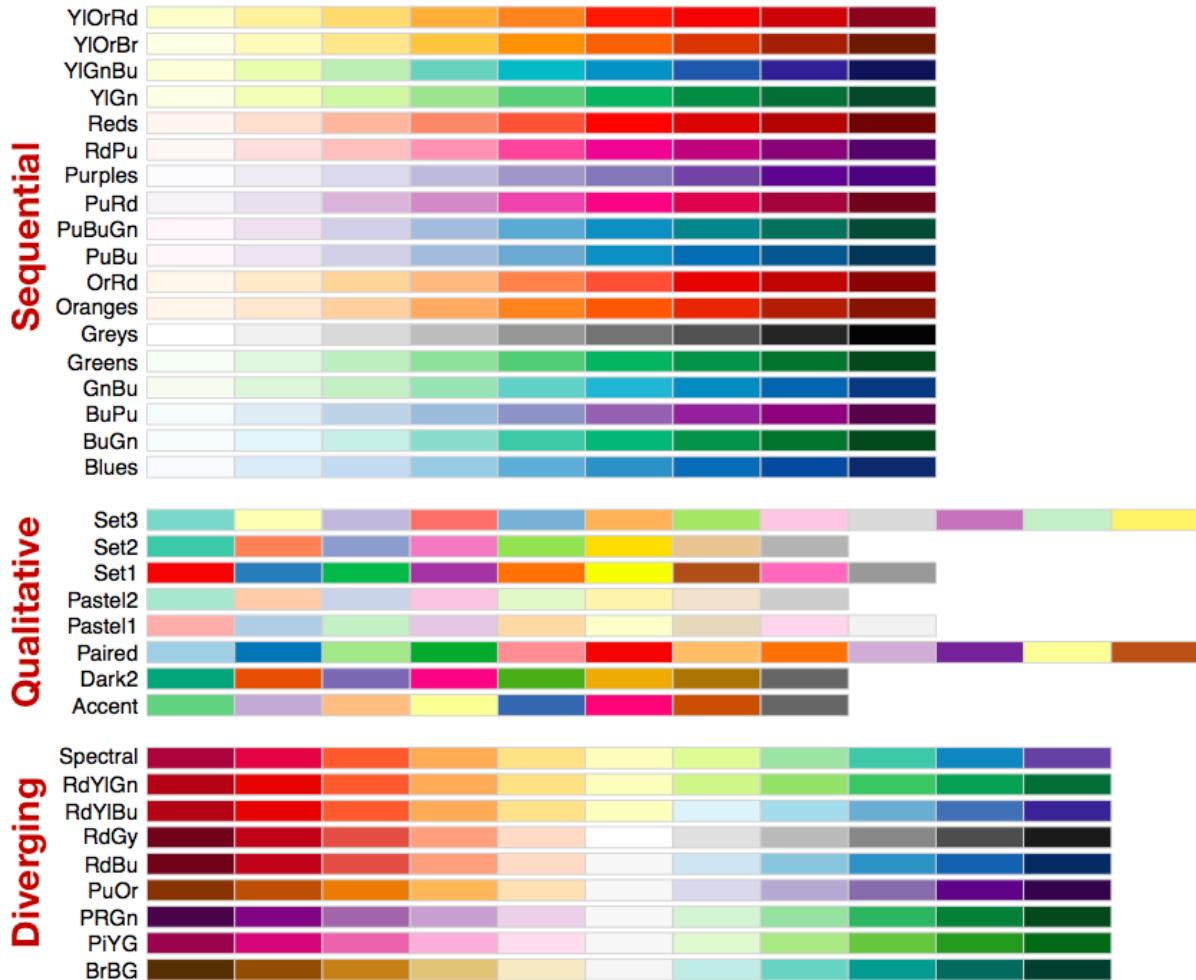
x <- rnorm(200); y <- rnorm(200)
par(mfrow=c(1,2))
# normal scatter plot
plot(x, y, pch = 19, main = "Default")
# using transparency shows data much better
plot(x, y, col = rgb(0, 0, 0, 0.2), main = "With Transparency")

```



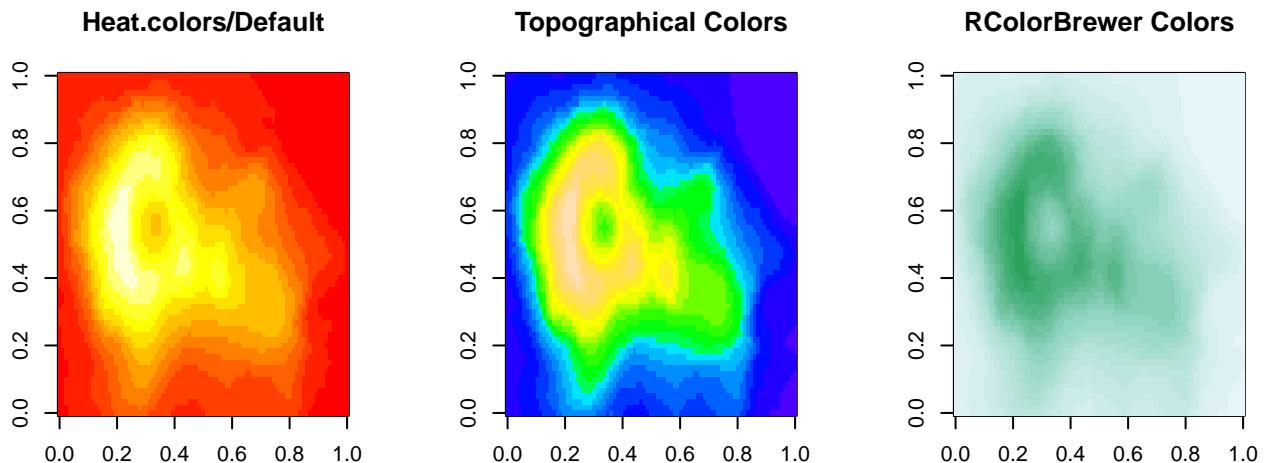
### RColorBrewer Package

- can be found on CRAN that has predefined color palettes
  - `library(RColorBrewer)`
- **types of palettes**
  - *Sequential* = numerical/continuous data that is ordered from low to high
  - *Diverging* = data that deviate from a value, increasing in two directions (i.e. standard deviations from the mean)
  - *Qualitative* = categorical data/factor variables
- palette information from the RColorBrewer package can be used by `colorRamp` and `colorRampPalette` functions
- **available colors palettes**



- `brewer.pal(n, "BuGn")` function
  - n = number of colors to generated
  - "BuGn" = name of palette
    - \* `?brewer.pal` list all available palettes to use
  - returns list of n hexadecimal colors
- *example*

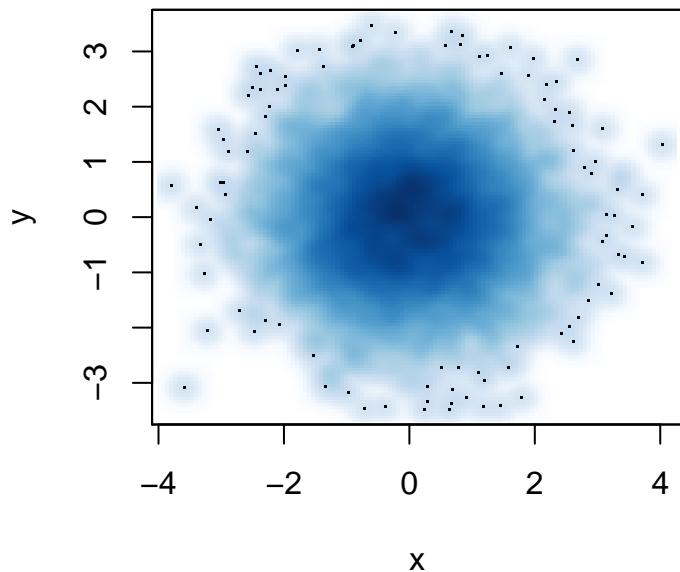
```
library(RColorBrewer)
# generate 3 colors using brewer.pal function
cols <- brewer.pal(3, "BuGn")
pal <- colorRampPalette(cols)
par(mfrow=c(1,3))
# heat.colors/default
image(volcano, main = "Heat.colors/Default")
# topographical colors
image(volcano, col = topo.colors(20), main = "Topographical Colors")
# RColorBrewer colors
image(volcano, col = pal(20), main = "RColorBrewer Colors")
```



- **smoothScatter function**

- used to plot large quantities of data points
- creates 2D histogram of points and plots the histogram
- default color scheme = “Blues” palette from RColorBrewer package
- *example*

```
x <- rnorm(10000); y <- rnorm(10000)
smoothScatter(x, y)
```



## Case Study: Human Activity Tracking with Smart Phones

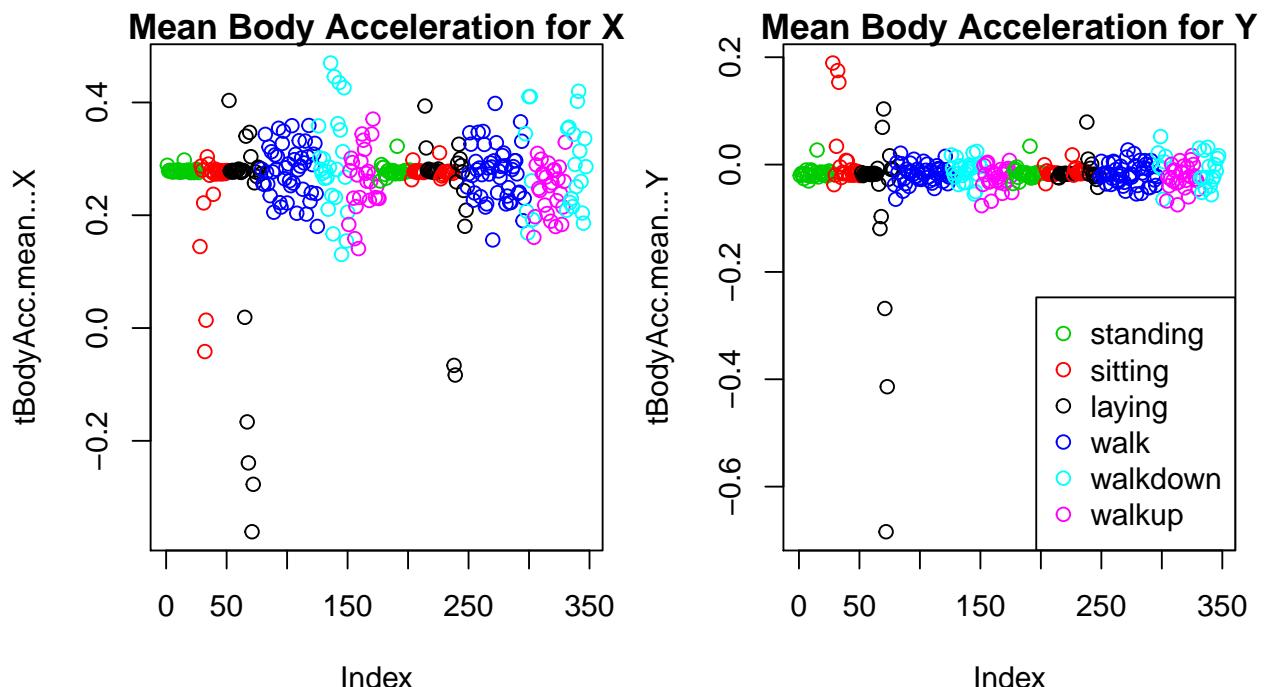
Loading Training Set of Samsung S2 Data from [UCI Repository](#)

```
# load data frame provided
load("samsungData.rda")
# table of 6 types of activities
table(samsungData$activity)

##
##    laying   sitting  standing      walk walkdown   walkup
##    1407      1286     1374      1226      986      1073
```

Plotting Average Acceleration for First Subject

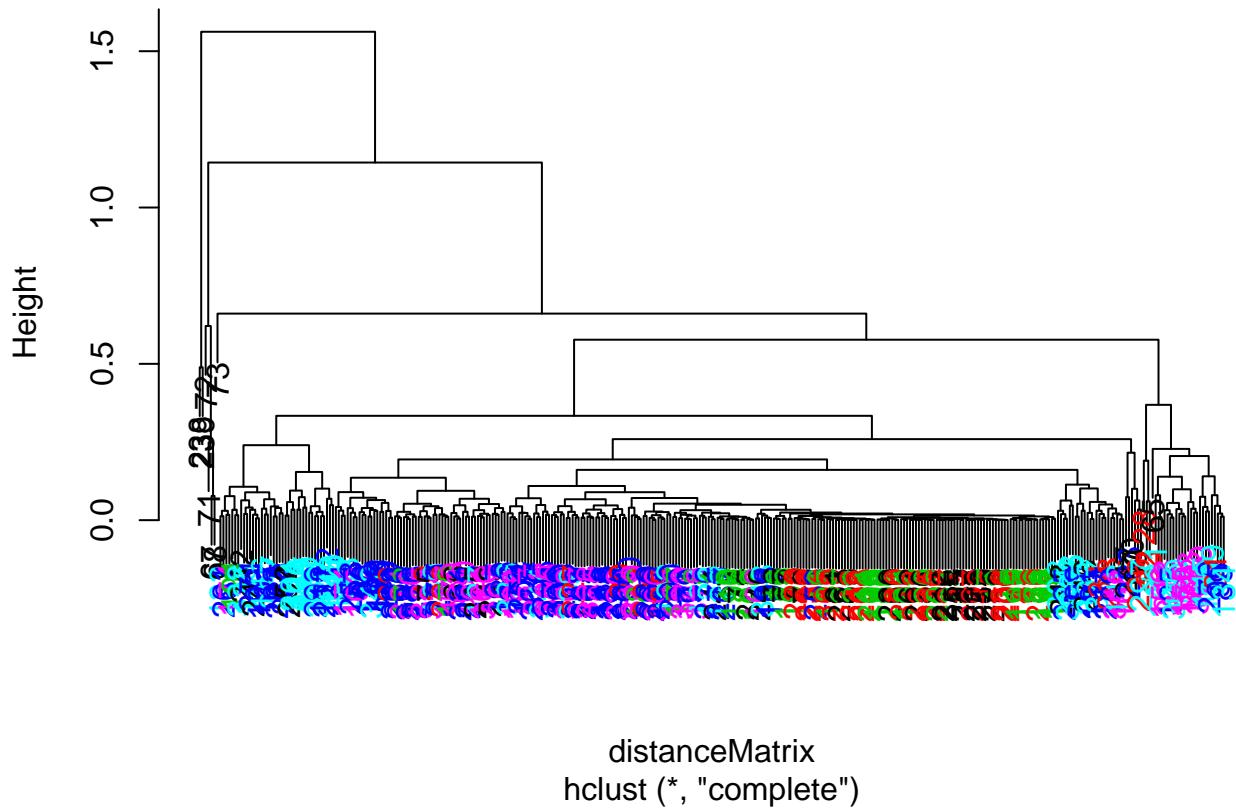
```
# set up 1 x 2 panel plot
par(mfrow=c(1, 2), mar = c(5, 4, 1, 1))
# converts activity to a factor variable
samsungData <- transform(samsungData, activity = factor(activity))
# find only the subject 1 data
sub1 <- subset(samsungData, subject == 1)
# plot mean body acceleration in X direction
plot(sub1[, 1], col = sub1$activity, ylab = names(sub1)[1],
     main = "Mean Body Acceleration for X")
# plot mean body acceleration in Y direction
plot(sub1[, 2], col = sub1$activity, ylab = names(sub1)[2],
     main = "Mean Body Acceleration for Y")
# add legend
legend("bottomright", legend=unique(sub1$activity), col=unique(sub1$activity), pch = 1)
```



## Clustering Based on Only Average Acceleration

```
# load myplclust function
source("myplclust.R")
# calculate distance matrix
distanceMatrix <- dist(sub1[,1:3])
# form hclust object
hclustering <- hclust(distanceMatrix)
# run myplclust on data
myplclust(hclustering, lab.col = unclass(sub1$activity))
```

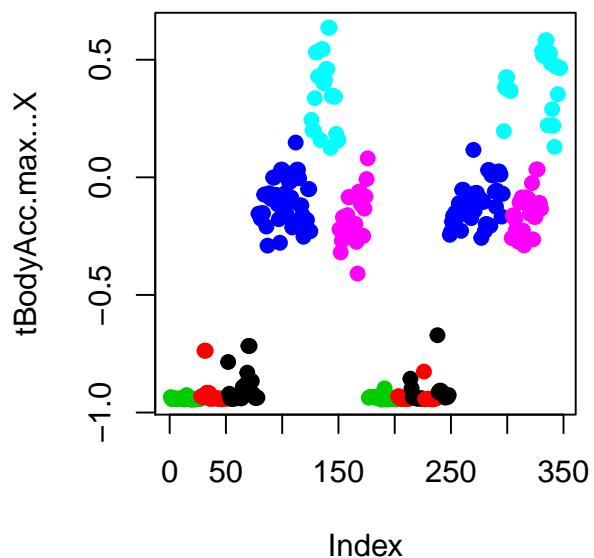
Cluster Dendrogram



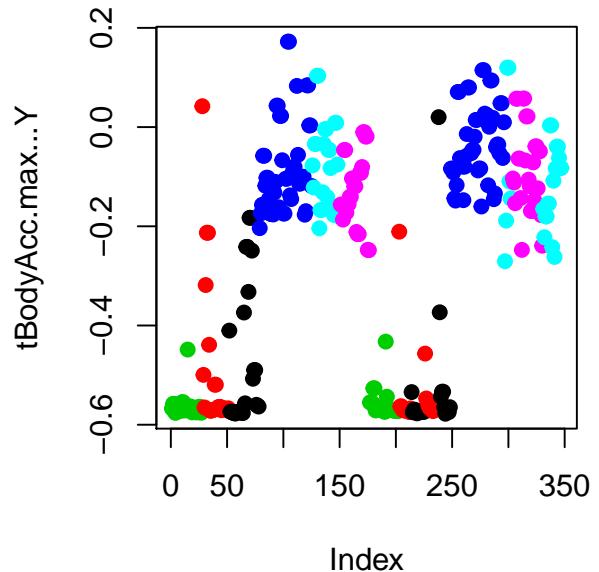
## Plotting Max Acceleration for the First Subject

```
# create 1 x 2 panel
par(mfrow=c(1,2))
# plot max accelerations in x and y direction
plot(sub1[,10], pch=19, col=sub1$activity, ylab=names(sub1)[10],
     main = "Max Body Acceleration for X")
plot(sub1[,11], pch=19, col = sub1$activity, ylab=names(sub1)[11],
     main = "Max Body Acceleration for Y")
```

**Max Body Acceleration for X**



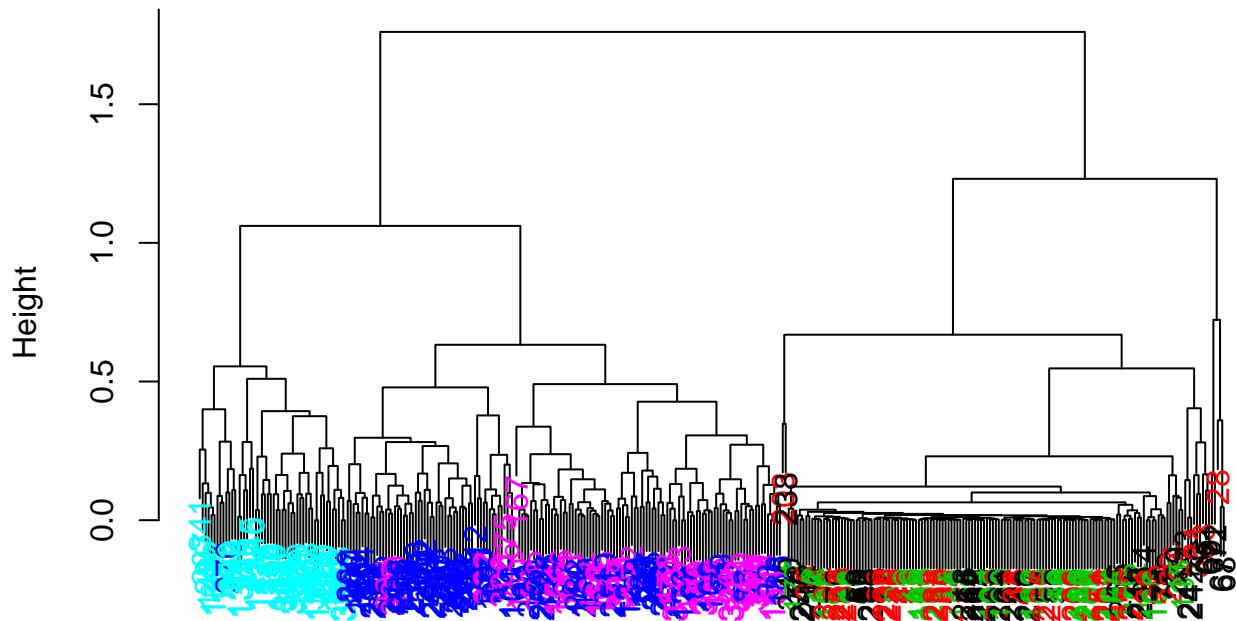
**Max Body Acceleration for Y**



**Clustering Based on Maximum Acceleration**

```
# calculate distance matrix for max distances
distanceMatrix <- dist(sub1[,10:12])
hclustering <- hclust(distanceMatrix)
myplclust(hclustering, lab.col=unclass(sub1$activity))
```

## Cluster Dendrogram

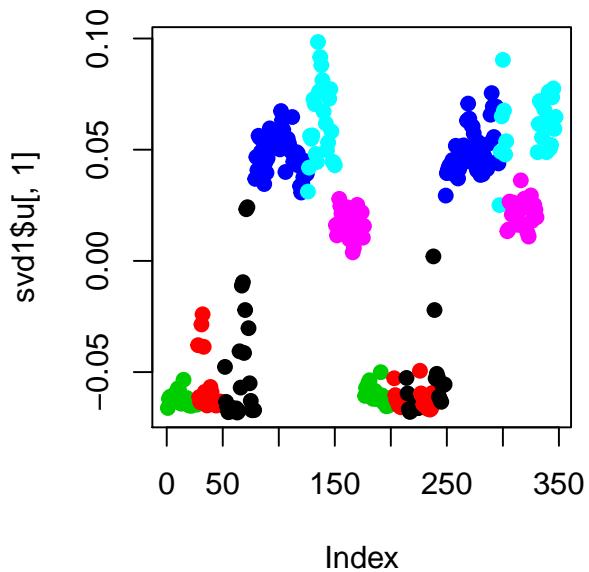


distanceMatrix  
hclust (\*, "complete")

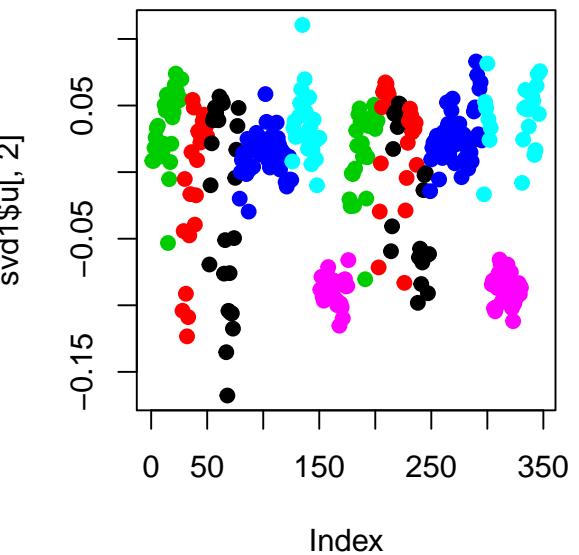
## Singular Value Decomposition

```
# perform SVD minus last two columns (subject and activity)
svd1 = svd(scale(sub1[,-c(562,563)]))
# create 1 x 2 panel plot
par(mfrow=c(1,2))
# plot first two left singular vector
# separate moving from non moving
plot(svd1$u[,1],col=sub1$activity,pch=19, main = "First Left Singular Vector")
plot(svd1$u[,2],col=sub1$activity,pch=19, main = "Second Left Singular Vector")
```

First Left Singular Vector



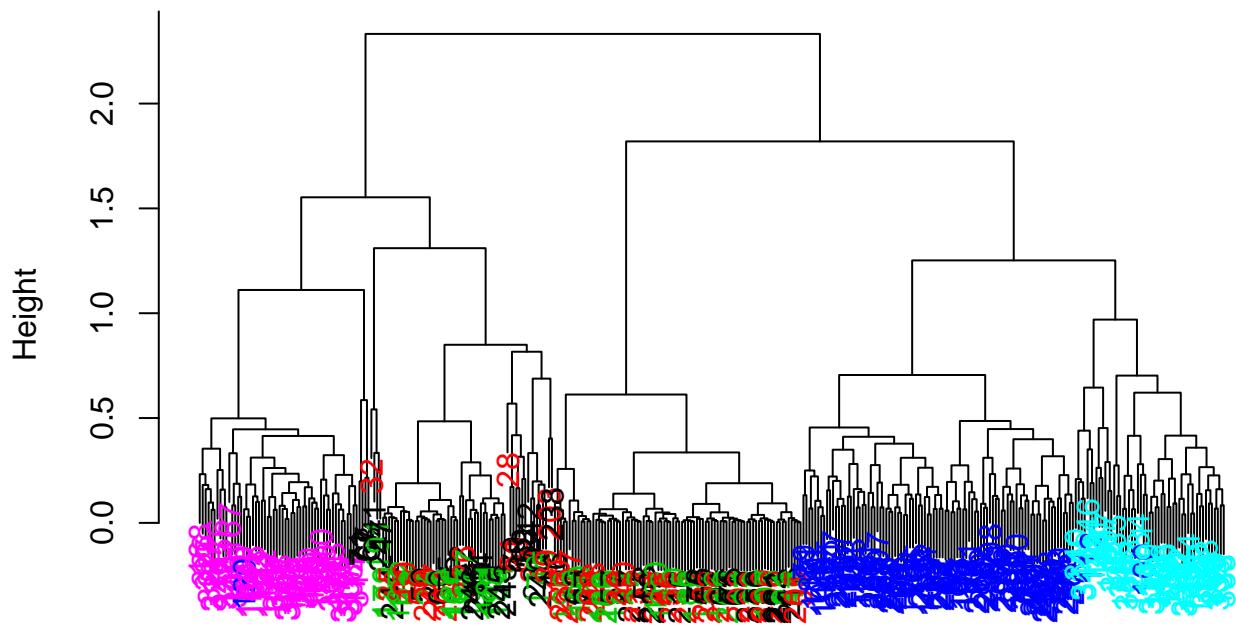
Second Left Singular Vector



#### New Clustering with Maximum Contributors

```
# find the max contributing feature
maxContrib <- which.max(svd1$v[,2])
# recalculate distance matrix
distanceMatrix <- dist(sub1[, c(10:12,maxContrib)])
hclustering <- hclust(distanceMatrix)
myplclust(hclustering, lab.col=unclass(sub1$activity))
```

## Cluster Dendrogram



```
distanceMatrix  
hclust (*, "complete")
```

```
# name of max contributing factor  
names(samsungData)[maxContrib]
```

```
## [1] "fBodyAcc.meanFreq...Z"
```

**K-means Clustering (nstart=1, first try)**

```
# specify 6 centers for data  
kClust <- kmeans(sub1[,-c(562,563)],centers=6)  
# tabulate 6 clusteres against 6 activity but many clusters contain multiple activities  
table(kClust$cluster,sub1$activity)
```

```
##  
##      laying sitting standing walk walkdown walkup  
## 1      0       0       0    0    95      0     0  
## 2     16      12      33    7     0      0     0  
## 3     24      33      46   46     0      0     0  
## 4     10       2       0    0     0      0     0  
## 5      0       0       0    0     0    49      0  
## 6      0       0       0    0     0      0    53
```

**K-means clustering (nstart=100, first try)**

```
# run k-means algorithm 100 times
kClust <- kmeans(sub1[,-c(562,563)],centers=6,nstart=100)
# tabulate results
table(kClust$cluster,sub1$activity)
```

```
##
##      laying sitting standing walk walkdown walkup
## 1     0       37      51   0       0       0
## 2     3       0       0   0       0       53
## 3    18      10      2   0       0       0
## 4     0       0       0   0       0       49
## 5    29      0       0   0       0       0
## 6     0       0       0   0      95       0
```

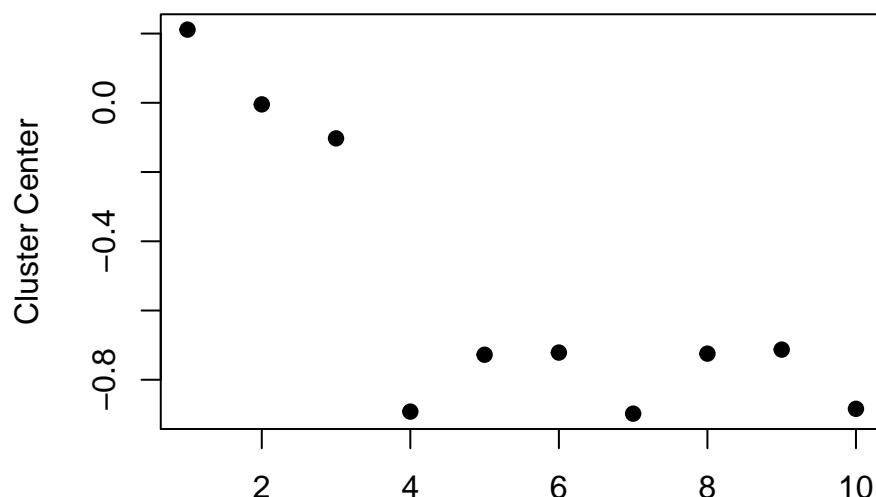
K-means clustering (nstart=100, second try)

```
# run k-means algorithm 100 times
kClust <- kmeans(sub1[,-c(562,563)],centers=6,nstart=100)
# tabulate results
table(kClust$cluster,sub1$activity)
```

```
##
##      laying sitting standing walk walkdown walkup
## 1    18      10      2   0       0       0
## 2     0       0       0   0       0       49
## 3    29      0       0   0       0       0
## 4     0       37      51   0       0       0
## 5     0       0       0   0      95       0
## 6     3       0       0   0       0       53
```

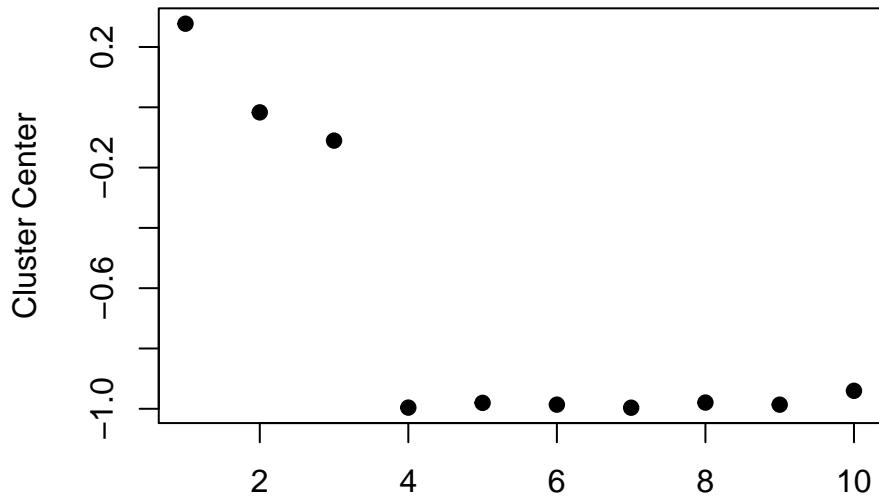
Cluster 1 Variable Centers (Laying)

```
# plot first 10 centers of k-means for laying to understand which features drive the activity
plot(kClust$center[1,1:10],pch=19,ylab="Cluster Center",xlab="")
```



## Cluster 2 Variable Centers (Walking)

```
# plot first 10 centers of k-means for laying to understand which features drive the activity
plot(kClust$center[4,1:10],pch=19,ylab="Cluster Center",xlab="")
```



## Case Study: Fine Particle Pollution in the U.S. from 1999 to 2012

### Read Raw Data from 1999 and 2012

```
# read in raw data from 1999
pm0 <- read.table("pm25_data/RD_501_88101_1999-0.txt", comment.char = "#", header = FALSE, sep = "|", na.strings = "", nrow = 1304290)
# read in headers/column labels
cnames <- readLines("pm25_data/RD_501_88101_1999-0.txt", 1)
# convert string into vector
cnames <- strsplit(substring(cnames, 3), "|", fixed = TRUE)
# make vector the column names
names(pm0) <- make.names(cnames[[1]])
# we are interested in the pm2.5 readings in the "Sample.Value" column
x0 <- pm0$Sample.Value
# read in the data from 2012
pm1 <- read.table("pm25_data/RD_501_88101_2012-0.txt", comment.char = "#", header = FALSE, sep = "|",
na.strings = "", nrow = 1304290)
# make vector the column names
names(pm1) <- make.names(cnames[[1]])
# take the 2012 data for pm2.5 readings
x1 <- pm1$Sample.Value
```

### Summaries for Both Periods

```
# generate 6 number summaries
summary(x1)

##      Min.   1st Qu.    Median     Mean   3rd Qu.     Max.   NA's
## -10.00     4.00     7.63     9.14    12.00   909.00   73133

summary(x0)

##      Min.   1st Qu.    Median     Mean   3rd Qu.     Max.   NA's
##  0.00     7.20    11.50    13.74   17.90   157.10   13217

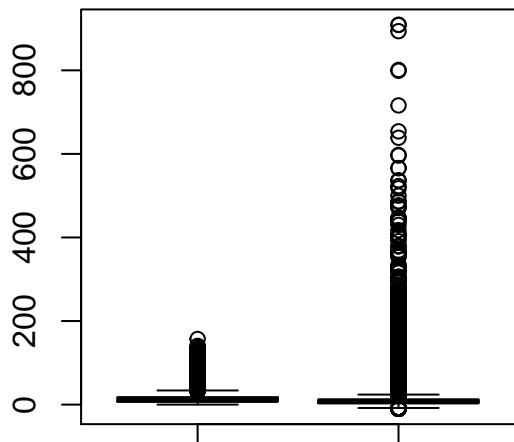
# calculate % of missing values, Are missing values important here?
data.frame(NA.1990 = mean(is.na(x0)), NA.2012 = mean(is.na(x1)))

##      NA.1990    NA.2012
## 1 0.1125608 0.05607125
```

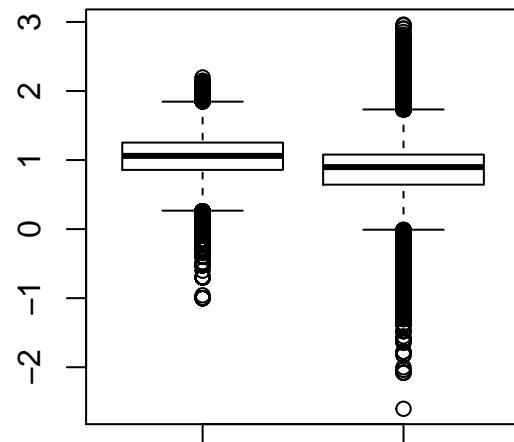
### Make a boxplot of both 1999 and 2012

```
par(mfrow = c(1,2))
# regular boxplot, data too right skewed
boxplot(x0, x1, main = "Regular Boxplot")
# log boxplot, significant difference in means, but more spread
boxplot(log10(x0), log10(x1), main = "log Boxplot")
```

## Regular Boxplot



## log Boxplot



Check for Negative Values in 'x1'

```
# summary again
summary(x1)
```

```
##      Min. 1st Qu. Median     Mean 3rd Qu.    Max.    NA's
## -10.00    4.00   7.63    9.14   12.00  909.00  73133
```

```
# create logical vector for
negative <- x1 < 0
# count number of negatives
sum(negative, na.rm = T)
```

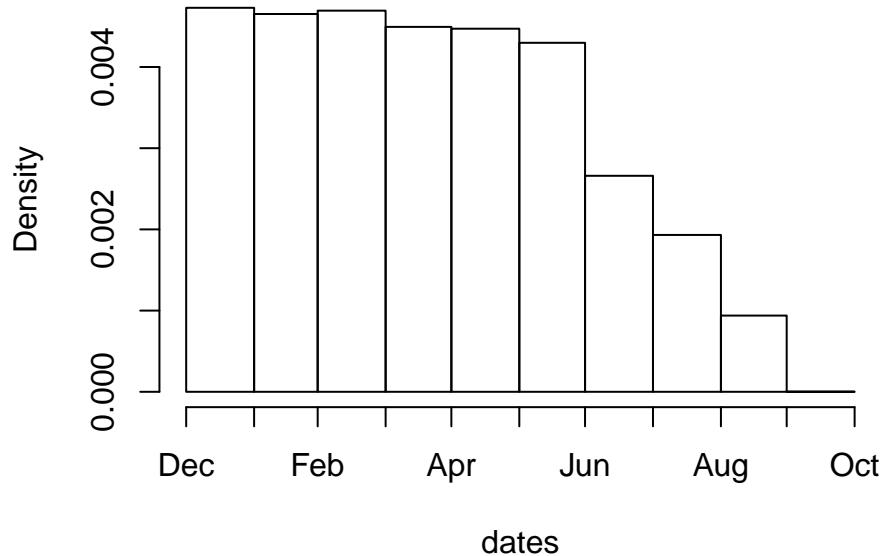
```
## [1] 26474
```

```
# calculate percentage of negatives
mean(negative, na.rm = T)
```

```
## [1] 0.0215034
```

```
# capture the date data
dates <- pm1$Date
dates <- as.Date(as.character(dates), "%Y%m%d")
# plot the histogram
hist(dates, "month") ## Check what's going on in months 1--6
```

## Histogram of dates



Check Same New York Monitors at 1999 and 2012

```
# find unique monitors in New York in 1999
site0 <- unique(subset(pm0, State.Code == 36, c(County.Code, Site.ID)))
# find unique monitors in New York in 2012
site1 <- unique(subset(pm1, State.Code == 36, c(County.Code, Site.ID)))
# combine country codes and siteIDs of the monitors
site0 <- paste(site0[,1], site0[,2], sep = ".")
site1 <- paste(site1[,1], site1[,2], sep = ".")
# find common monitors in both
both <- intersect(site0, site1)
# print common monitors in 1999 and 2012
print(both)
```

```
## [1] "1.5"      "1.12"     "5.80"     "13.11"    "29.5"     "31.3"     "63.2008"
## [8] "67.1015"  "85.55"    "101.3"
```

Find how many observations available at each monitor

```
# add columns for combined county/site for the original data
pm0$county.site <- with(pm0, paste(County.Code, Site.ID, sep = "."))
pm1$county.site <- with(pm1, paste(County.Code, Site.ID, sep = "."))
# find subsets where state = NY and county/site = what we found previously
cnt0 <- subset(pm0, State.Code == 36 & county.site %in% both)
cnt1 <- subset(pm1, State.Code == 36 & county.site %in% both)
# split data by the county/site values and count observations
sapply(split(cnt0, cnt0$county.site), nrow)
```

```
##   1.12    1.5    101.3   13.11    29.5    31.3    5.80  63.2008 67.1015
##     61    122    152     61     61    183     61     122    122
##   85.55
##     7
```

```

sapply(split(cnt1, cnt1$county.site), nrow)

##      1.12      1.5   101.3   13.11    29.5    31.3    5.80 63.2008 67.1015
##      31       64      31      31      33      15      31      30      31
##     85.55
##      31

```

Choose Monitor where County = 63 and Side ID = 2008

```

# filter data by state/county/siteID
pm1sub <- subset(pm1, State.Code == 36 & County.Code == 63 & Site.ID == 2008)
pm0sub <- subset(pm0, State.Code == 36 & County.Code == 63 & Site.ID == 2008)
# there are 30 observations from 2012, and 122 from 1999
dim(pm1sub)

```

```
## [1] 30 29
```

```
dim(pm0sub)
```

```
## [1] 122 29
```

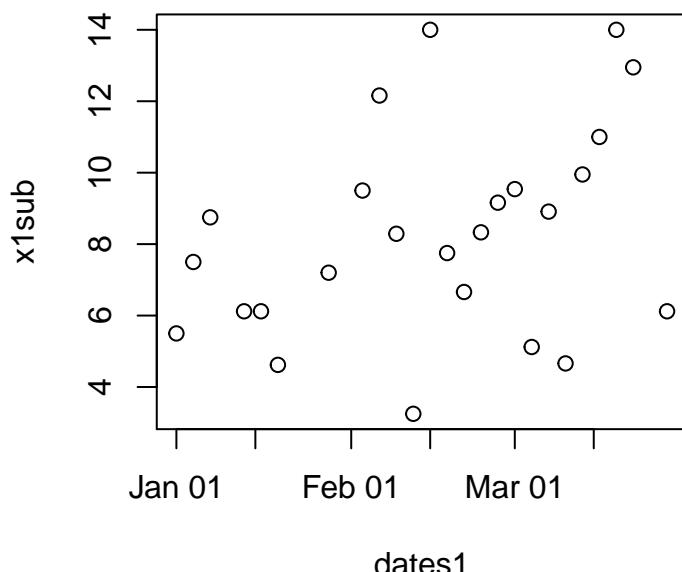
Plot Data for 2012

```

# capture the dates of the subset of data
dates1 <- pm1sub$date
# capture measurements for the subset of data
x1sub <- pm1sub$sample.value
# convert dates to appropriate format
dates1 <- as.Date(as.character(dates1), "%Y-%m-%d")
# plot pm2.5 value vs time
plot(dates1, x1sub, main = "PM2.5 Pollution Level in 2012")

```

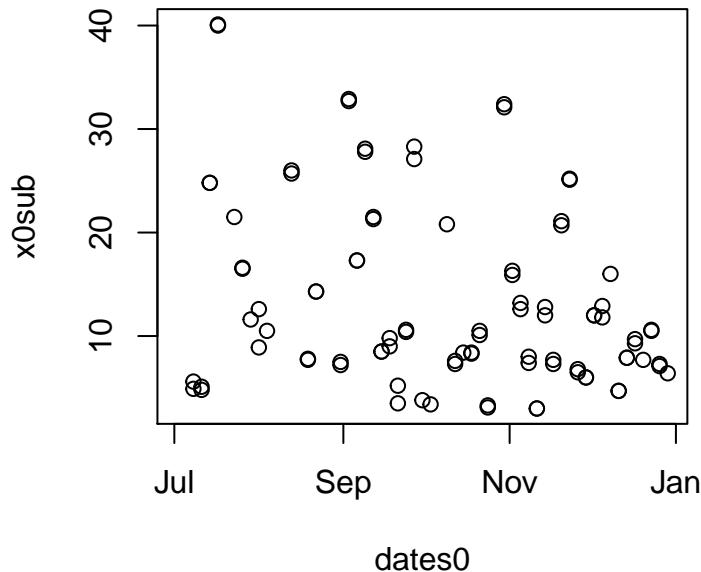
## PM2.5 Pollution Level in 2012



### Plot data for 1999

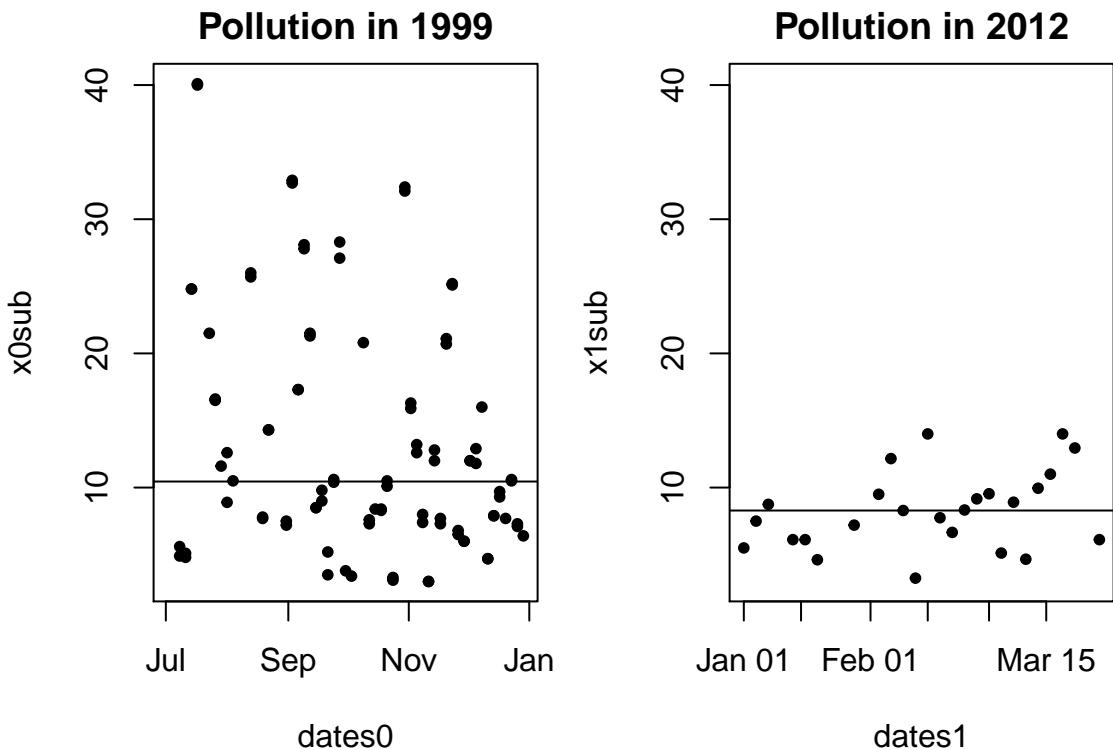
```
# capture the dates of the subset of data
dates0 <- pm0sub$Date
# convert dates to appropriate format
dates0 <- as.Date(as.character(dates0), "%Y%m%d")
# capture measurements for the subset of data
x0sub <- pm0sub$Sample.Value
# plot pm2.5 value vs time
plot(dates0, x0sub, main = "PM2.5 Polution Level in 1999")
```

**PM2.5 Polution Level in 1999**



### Panel Plot for Both Years

```
# find max range for data
rng <- range(x0sub, x1sub, na.rm = T)
# create 1 x 2 panel plot
par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))
# plot time series plot for 1999
plot(dates0, x0sub, pch = 20, ylim = rng, main="Pollution in 1999")
# plot the median
abline(h = median(x0sub, na.rm = T))
# plot time series plot for 2012
plot(dates1, x1sub, pch = 20, ylim = rng, main="Pollution in 2012")
# plot the median
abline(h = median(x1sub, na.rm = T))
```



#### Find State-wide Means and Trend

```
# divide data by state and find the mean of pollution level for 1999
mn0 <- with(pm0, tapply(Sample.Value, State.Code, mean, na.rm = T))
# divide data by state and find the mean of pollution level for 1999
mn1 <- with(pm1, tapply(Sample.Value, State.Code, mean, na.rm = T))
# convert to data frames while preserving state names
d0 <- data.frame(state = names(mn0), mean = mn0)
d1 <- data.frame(state = names(mn1), mean = mn1)
# merge the 1999 and 2012 means by state
mrg <- merge(d0, d1, by = "state")
# dimension of combined data frame
dim(mrg)
```

```
## [1] 52 3
```

```
# first few lines of data
head(mrg)
```

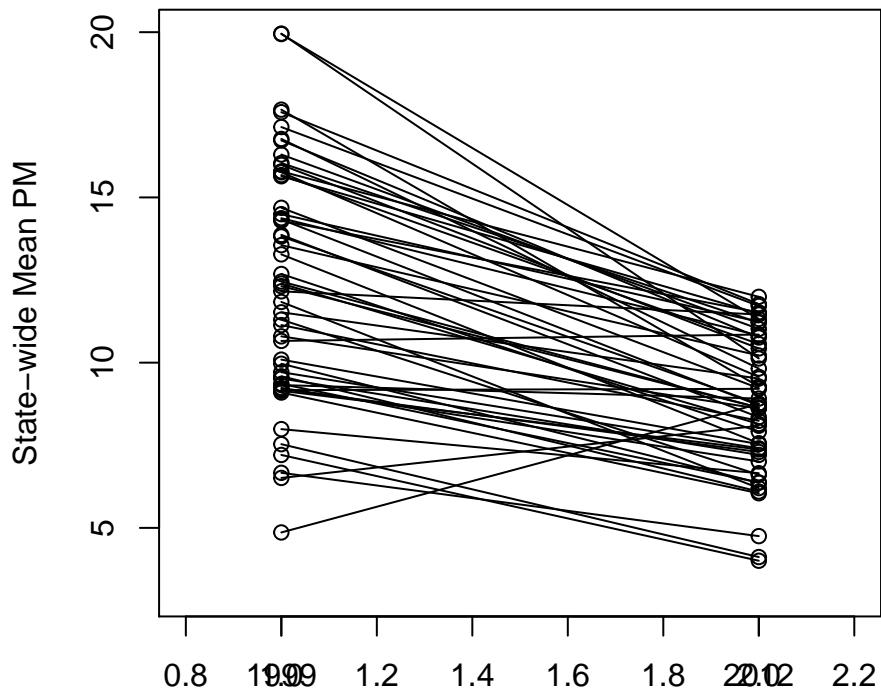
```
##   state   mean.x   mean.y
## 1      1 19.956391 10.126190
## 2     10 14.492895 11.236059
## 3     11 15.786507 11.991697
## 4     12 11.137139  8.239690
## 5     13 19.943240 11.321364
## 6     15  4.861821  8.749336
```

```

# plot the pollution levels data points for 1999
with(mrg, plot(rep(1, 52), mrg[, 2], xlim = c(.8, 2.2), ylim = c(3, 20),
  main = "PM2.5 Pollution Level by State for 1999 & 2012",
  xlab = "", ylab = "State-wide Mean PM"))
# plot the pollution levels data points for 2012
with(mrg, points(rep(2, 52), mrg[, 3]))
# connected the dots
segments(rep(1, 52), mrg[, 2], rep(2, 52), mrg[, 3])
# add 1999 and 2012 labels
axis(1, c(1, 2), c("1999", "2012"))

```

## PM2.5 Pollution Level by State for 1999 & 2012



# Reproducible Research Course Notes

*Xing Su*

## Contents

Replication . . . . .	3
Reproducibility . . . . .	3
Literate/Statistical Programming . . . . .	4
Key Challenge in Data Analysis . . . . .	4
Structure of Data Analysis . . . . .	5
Define The Question . . . . .	5
Determine the Ideal Data Set . . . . .	5
Determine What Data You Can Access . . . . .	5
Obtain The Data . . . . .	5
Clean The Data . . . . .	5
Exploratory Data Analysis . . . . .	6
Statistical Modelling/Prediction . . . . .	7
Interpret Results . . . . .	8
Challenge Results . . . . .	8
Synthesize/Write Up and Results . . . . .	9
Create Reproducible Code . . . . .	9
Organizing Data Analysis . . . . .	10
Coding Standards in R . . . . .	11
Markdown ( <a href="#">documentation</a> ) . . . . .	11
R Markdown . . . . .	11
Communicating Results . . . . .	14
Hierarchy of Information - Research Paper . . . . .	14
Hierarchy of Information - Email Presentation . . . . .	14
RPubs . . . . .	14
Reproducible Research Checklist . . . . .	15
Do's . . . . .	15
Don'ts . . . . .	16
Replication vs Reproducibility . . . . .	17
Background and Underlying Trend for Reproducibility . . . . .	17
Problems with Reproducibility . . . . .	17
Evidence-based Data Analysis . . . . .	18

Caching Computations . . . . .	19
<b>cacher</b> Package . . . . .	19
Case Study: Air Pollution . . . . .	23
Analysis: Does Nickel Make PM Toxic? . . . . .	23
Conclusions and Lessons Learnt . . . . .	25
Case Study: High Throughput Biology . . . . .	26
Conclusions and Lessons Learnt . . . . .	26

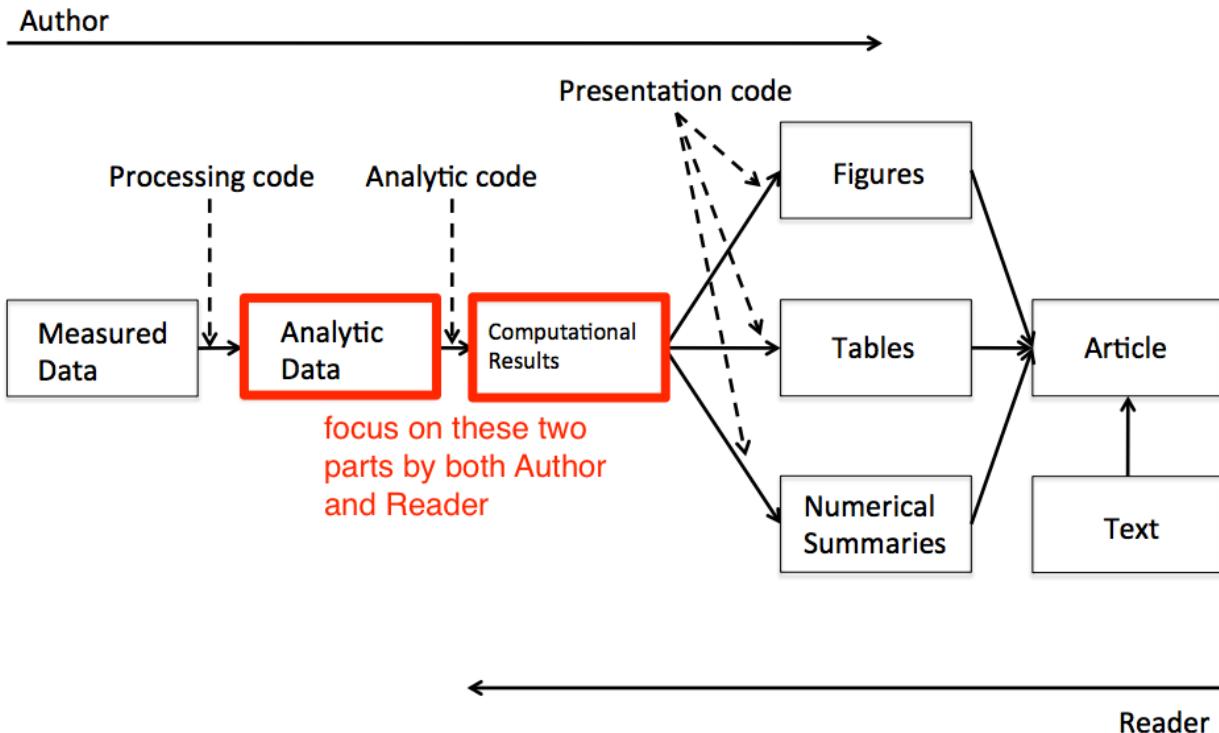
## Replication

- ultimate standard for strengthening of scientific evidence
- reproduce findings and conduct studies with independent investigators/data/analytical methods/laboratories/instruments
- particularly important in studies that affect policy
- HOWEVER, some studies can't be replicated (time, opportunity, money, unique)
  - Make research reproducible (making available data/code so others may reproduce analysis)

## Reproducibility

- bridges gap between replication and letting the study stand alone
- take data/code and replicate findings -> validate their findings
- why we need reproducible research
  - new technologies increase data throughput while adding complexities and dimension to data
  - existing databases merged into bigger collection of data
  - computational power increased

- **Research Pipeline**



- Institute of Medicine report - Evolution of Translational Omics
  - Data/meta data should be made available
  - Computer code/fully specified computational procedures need to be available
  - Data processing and computational analysis steps described
- **Necessities for Reproducible Research**

- analytical data (not raw data)
- analytical code (applied to data, i.e. regression model etc.)
- documentation of data/code
- standard means of distribution

- **Challenges**

- authors make considerable effort to make data/results available
- readers must download data/results individually and piece it all together
- readers may not have the same resources as authors (computing clusters, etc.)
- few tools for readers/authors
- In reality, authors just put their content online (often disorganized, but there are some central databases for various fields) and readers have to find the data and piece it together

## Literate/Statistical Programming

- Notion of an article is a stream of **text** and **code**
- original idea came from *Don Knuth*
- statistical analysis is divided into text and code chunks
  - text explains what's going on
  - code loads data/computes results
  - presentation code formats results (tables, graphics, etc)
- Literate Programming requires
  - **documentation language** - [*weave*] produce human-readable docs (HTML/PDF)
  - **programming language** - [*tangle*] produce machine-readable code
- **Sweave** - original program in R designed to do literate programming
  - LaTeX for documentation, R for programming
  - developed by *Friedrich Leisch* (member of R core)
  - still maintained by R core
  - **limitations**
    - \* focus on LaTeX, difficult to learn markup languages
    - \* lacks features like caching, multiple plots per chunk, mixing programming languages with many other technical terms
    - \* not frequently updated/actively developed
- **knitr** - alternative literate programming package
  - R for programming (can use others as well)
  - LaTeX/HTML/Markdown for documentation
  - developed by Iowa grad student *Yihui Xie*
- Reproducible research = minimum standard for non-replicable/computationally intensive studies
  - infrastructure still need for creating/distributing reproducible documents

## Key Challenge in Data Analysis

- “Ask yourselves, what problem have you solved, ever, that was worth solving, where you knew all of the given information in advance? Where you didn't have a surplus of information and have to filter it out, or you had insufficient information and have to go find some?” - Dan Myer

## Structure of Data Analysis

### Define The Question

- most powerful dimension reduction tool, narrowing down question reduces noise/simplify problem
- Science → Data → Applied Statistics → Theoretical Statistics
- properly using the combination of science/data/applied statistics = proper data analysis
- **example**
  - general question: Can I automatically detect emails that are SPAM from those that are not?
  - concrete question: Can I use quantitative characteristics of the emails to classify them as SPAM/HAM?

### Determine the Ideal Data Set

- after having the concrete question, find the data set that is suitable for your goal:
  - **Descriptive** = whole population
  - **Exploratory** = random sample with multiple variables measured
  - **Inferential** = drawing conclusion from a sample for a larger population, so choosing the right population, and carefully/randomly sampled
  - **Predictive** = need training and test data sets from the same population to build a model and classifier
  - **Causal** = (if this is modified, then that happens) experimental data from randomized study
  - **Mechanistic** = data from all components of system that you want to describe

### Determine What Data You Can Access

- often won't have access to the ideal data set
- free data from the web, paid data from provider (must respect terms of use)
- if data doesn't exist, you may need to generate some of the data

### Obtain The Data

- try to obtain the raw data and reference the source
- if loading data from internet source, record at the minimum url and time of access

### Clean The Data

- raw data need to be processed to be fed into modeling program
- if already processed, need to understand how and understand documentation on how the pre-processing was done
- understanding source of data (i.e. survey - how survey was done?)
- may need to reformat/subsample data → need to record steps
- determine if data is good enough to solve the problem → if not, find other data/quit/change question to avoid misleading conclusions

## Exploratory Data Analysis

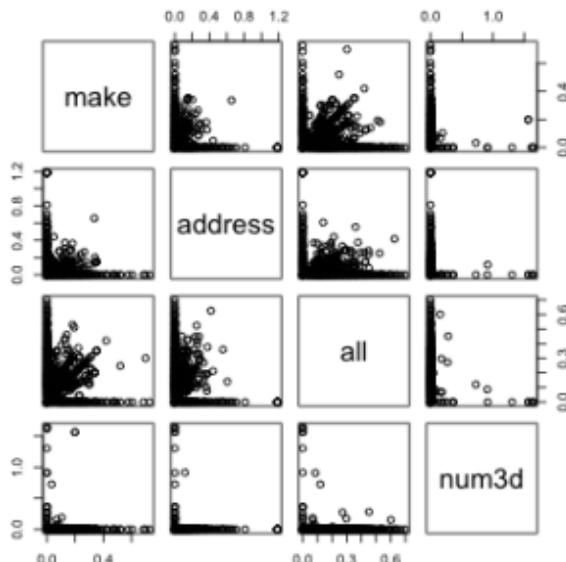
- for the purpose of the SPAM question, data needs to be split into training and test test (Predictive)

```
# If it isn't installed, install the kernlab package with install.packages()
suppressWarnings(library(kernlab))
data(spam)
set.seed(3435)
trainIndicator = rbinom(4601, size = 1, prob = 0.5)
table(trainIndicator)

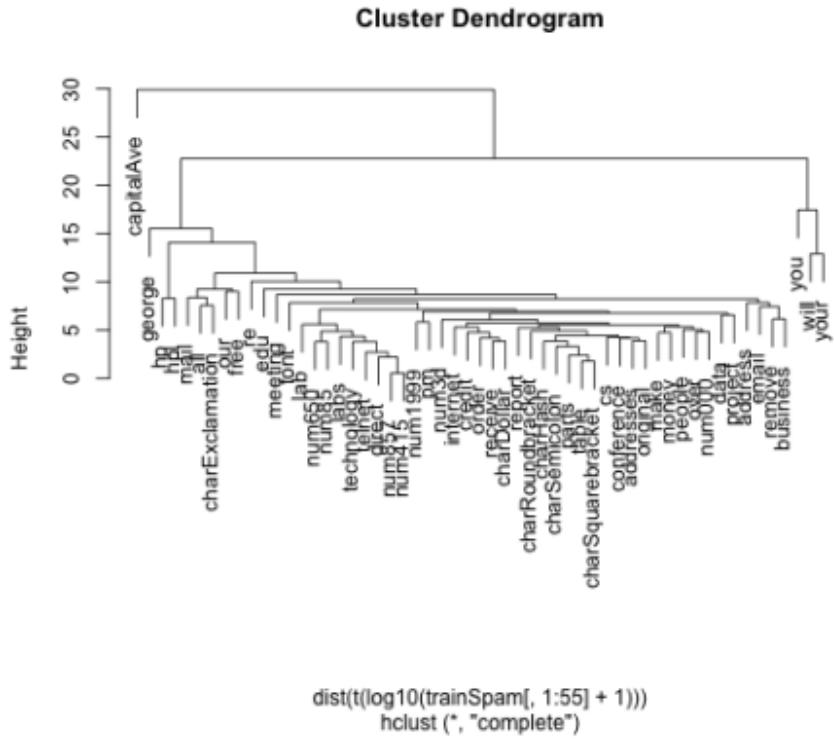
## trainIndicator
##      0      1
## 2314 2287

trainSpam = spam[trainIndicator == 1, ]
testSpam = spam[trainIndicator == 0, ]
```

- look at one/two-dimensional summaries of data, what the distribution looks like, relationships between variables
  - `names()`, `summary()`, `head()`
  - `table(trainSpam$type)`
- check for missing data/why there is missing data
- create exploratory plots
  - `plot(trainSpam$capitalAve ~ trainSpam$type)`
  - `plot(log10(trainSpam$capitalAve + 1) ~ trainSpam$type)`
  - because data is skewed → use log
  - a lot of zeroes → +1
- perform exploratory analysis (clustering)
  - relationships between predictors
  - `plot(log10(trainSpam[, 1:4] + 1))`
    - \* log transformations diagrams between pairs of variables



- hierarchical cluster analysis (**Dendrogram**) = what words/characteristics tend to cluster together
    - `hCluster = hclust(dist(t(trainSpam[, 1:57])))`
      - \* *Note: Dendograms can be sensitive to skewness in distribution of individual variables, so maybe useful to transform predictor (see below)*
    - `hClusterUpdated = hclust(dist(t(log10(trainSpam[, 1:55] + 1))))`



## Statistical Modelling/Prediction

- should be guided by exploratory analysis
  - exact methods depend on the question of interest
  - account for data transformation/processing
    - go through each variable in the dataset and fit logistic regression to see if we can predict if an email is spam or not by using a single variable

```

trainSpam$numType = as.numeric(trainSpam$type) - 1
costFunction = function(x, y) sum(x != (y > 0.5))
cvError = rep(NA, 55)
library(boot)
for (i in 1:55) {
  # creates formula with one variable and the result
  lmFormula = reformulate(names(trainSpam)[i], response = "numType")
  glmFit = glm(lmFormula, family = "binomial", data = trainSpam)

  # cross validated error
  cvError[i] = cv.glm(trainSpam, glmFit, costFunction, 2)$delta[2]
}

```

```

}

# Which predictor has minimum cross-validated error?
names(trainSpam)[which.min(cvError)]


## [1] "charDollar"

• think about measures/sources of uncertainty

# Use the best model from the group
predictionModel = glm(numType ~ charDollar,family="binomial",data=trainSpam)

# Get predictions on the test set
predictionTest = predict(predictionModel,testSpam)
predictedSpam = rep("nonspam",dim(testSpam)[1])

# Classify as 'spam' for those with prob > 0.5
predictedSpam[predictionModel$fitted > 0.5] = "spam"

# Classification table
table(predictedSpam, testSpam$type)

## 
## predictedSpam nonspam spam
##      nonspam    1346  458
##      spam        61   449

# Error rate
(61 + 458)/(1346 + 458 + 61 + 449)

## [1] 0.2242869

```

## Interpret Results

- use appropriate language and not go over what you performed
  - describes, correlates/associated with, lead to/causes, predicts
- give explanation of the results (why certain models predict better than others)
- interpret coefficients
  - The fraction of characters that are dollar signs can be used to predict if an email is Spam
  - Anything with more than 6.6% dollar signs is classified as Spam
  - More dollar signs always means more Spam under our prediction
- interpret measures of uncertainty to calibrate your interpretation of results
  - Our test set error rate was 22.4%

## Challenge Results

- good to challenge the whole process/all results you found (because somebody else will)
  - question -> data source -> processing -> analysis -> conclusion
- challenge measures of uncertainty and choices in what to include in the model
- think about alternative analyses, maybe useful to try in case they produce interesting/better models/predictions

## Synthesize/Write Up and Results

- tell coherent story of the most important part of analysis
- lead with question → provide context to better understand framework of analysis
  - Can I use quantitative characteristics of the emails to classify them as SPAM/HAM?
- summarize the analysis into the story
  - Collected data from UCI -> created training/test sets
  - Explored relationships
  - Choose logistic model on training set by cross validation
  - Applied to test, 78% test set accuracy
- don't need every analysis performed, only include the ones that are needed for the story or address a challenge
  - Number of dollar signs seems to indicate SPAM, "Make money with Viagra \$ \$ \$ \$"!
  - 78% isn't that great
  - I could use more variables
  - Why logistic regression?
- order analysis according to story rather than chronologically
- include well defined figures to help understanding

## Create Reproducible Code

- document your analysis as you do them using markdown or knitr
  - preserve any R code or written summary in a single document using knitr
- ensure all analysis is reproducible (standard that most data analysis are judged on)

## Organizing Data Analysis

- key data analysis files
  - **data:** raw
    - \* should be stored in analysis folder
    - \* if accessed from the web, include in README file the URL, where the data is from, what the data is, brief description of what it is for, date of access
    - \* if data is stored in Git repository, can use the log to talk about the above info
  - **data:** processed
    - \* should be named so that it's easy to see what script generated what data
    - \* In README, important to document what code files were used to transform raw data to processed data
    - \* processed data should be tidy/organized
  - **figures:** exploratory
    - \* made during process of analysis, not necessarily all part of final report
    - \* don't need to be well formatted/annotated
    - \* need to be usable enough so that the author can easily understand what is being presented and how to reproduce it
  - **figures:** final
    - \* more polished, better organized, more readable, possibly multiple panels
    - \* usually a small subset or original exploratory figures (typical journal article contains 5-6 figures)
    - \* clearly labeled and well annotated, axes/color set to make figure clear
  - **R code:** raw/unused scripts
    - \* could be less commented, multiple versions
    - \* may include analyses that are later discarded
    - \* important to name/document appropriately to understand what was performed
    - \* placed in a separate part of the data analysis directory (separate from final)
  - **R code:** final scripts
    - \* clearly commented - small comments for what/when/why/how, big comment blocks for whole sections to explain what is being done
    - \* include processing details for the raw data
    - \* should pertain to only analyses that appear in the final write up
  - **R code:** R markdown files
    - \* not necessary or required but useful to summarize parts of analysis
    - \* can be used to generate reproducible reports, as it can embed code and text into single document and process the document into readable webpage/pdf doc
    - \* easy to create in Rstudio
  - **text:** README file
    - \* explain what is going on in project directory
    - \* not necessary if you have R markdown files (don't separate analysis and code - literate programming principle)
    - \* should contain step-by-step instructions for how analysis was conducted, what code files are called first, what are used to process the data, what are used to fit models, what are used to generate figures, etc.
  - **text:** text of analysis/report
    - \* should include title, introduction/motivation for problem, the method (statistical method), the results (including measures of uncertainty) and conclusions (including pitfalls/problems)

- \* coherent story from all analysis performed, but does not need to include all analysis performed but only the most important and relevant parts
- \* should include references for statistical methods, software packages, implementation that were used (for reproducibility)

- **Resources**

- [Project template](#) -> a external R package
- [Managing a statistical analysis project guidelines and best practices](#)

## Coding Standards in R

- write code in text editor and save as text file
- indenting (4 space minimum)
- limit the width of your code (80 columns)
- limit the length of functions

## Markdown ([documentation](#))

- Markdown = text-to-HTML conversion tool for web writers.
  - simplified version of markup language
  - write using easy-to-read, easy-to-write plain text format
  - any text editor can create markdown document
  - convert the text to structurally valid XHTML/HTML
  - created by John Gruber
- **italics** -> **\*text\***
- **bold** -> **\*\*text\*\***
- **main heading** -> **#Heading**
- **secondary heading** -> **##Heading**
- **tertiary main heading** -> **###Heading**
- **unordered list** -> - first element
- **ordered list** -> 1. first element
  - the number in front actually doesn't matter, as long as there is a number there, markdown will be compiled to an ordered list (no need for renumbering)
- **links** -> [text](url)
  - OR [text][1] -> [1]: url "text"
- **new lines** -> requires a *double space* after the end of a line

## R Markdown

- integration of R code and markdown
  - allows creating documents containing “live” R code
  - R code is evaluated as a part of the processing of the markdown
  - results from R code inserted into markdown document
  - core tool for literate statistical programming
  - **pros**
    - \* text/code all in one place in logical order

- \* data/results automatically updated to reflect external changes
- \* code is live = automatic test when building document
- **cons**
  - \* text/code all in one place, can be difficult to read especially if there's a lot of code
  - \* can substantially slowdown processing of documents
- **knitr package**
  - written by YihuiXie, built into RStudio
  - support Markdown, LaTeX, HTML as documentation languages
  - Exports PDF/HTML
  - good for manuals, short/medium technical documents, tutorials, periodic reports, data preprocessing documents/summaries
  - not good for long research articles, complex time-consuming computations, precisely formatted documents
  - evaluates R markdown documents and return/records the results, and write out a Markdown files
  - Markdown file can then be converted into HTML using markdown package
  - solidify package converts R markdown into presentation slides
- In RStudio, create new R Markdown files by clicking New → R Markdown
  - ======> indicates title of document (large text)
  - \$expression\$ → indicates LaTeX expression/formatting
  - ‘text’ → changes text to code format (typewriter font)
  - ““{r name, echo = FALSE, results = hide}...““ → R code chunk
    - \* name = name of the code chunk
    - \* echo = FALSE → turns off the echo of the R code chunk, which means display only the result
    - \* *Note: by default code in code chunk is echoed → print code AND results*
    - \* results = hide → hides the results from being placed in the markdown document
  - inline text computations
    - \* ‘r variable’→ prints the value of that variable directly inline with the text
  - incorporating graphics
    - \* ““{r scatterplot, fig.height = 4, fig.width = 6} ... plot() ...““ → inserts a plot into markdown document
      - scatterplot = name of this code chunk (can be anything)
      - fig.height = 4 → adjusts height of the figure, specifying this alone will produce a rectangular plot rather than a square one by default
      - fig.width = 6 → adjusts width of the figure
    - \* knitr produces HTML, with the image embedded in HTML using base64 encoding
      - does not depend on external image files
      - not efficient but highly transportable
  - incorporating tables (xtable package: `install.packages("xtable")`)
    - \* `xtable` prints the table in html format, which is better presented than plain text normally

```
library(datasets)
library(xtable)
fit <- lm(Ozone ~ Wind + Temp + Solar.R, data = airquality)
xt <- xtable(summary(fit))
print(xt)
```

% latex table generated in R 3.1.2 by xtable 1.7-4 package % Thu Mar 19 15:03:36 2015

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-64.3421	23.0547	-2.79	0.0062
Wind	-3.3336	0.6544	-5.09	0.0000
Temp	1.6521	0.2535	6.52	0.0000
Solar.R	0.0598	0.0232	2.58	0.0112

- setting global options
  - “`{r setoptions, echo = FALSE} opt\_chunk\$set(echo = FALSE, results = "hide")`” –> sets the default option to not print the code/results unless otherwise specified
- common options
  - **output:** `results = "asis"` OR `"hide"`
    - \* `"asis"` = output to stay in original format and not compiled into HTML
  - **output:** `echo = TRUE` OR `FALSE`
  - **figures:** `fig.height = numeric`
  - **figures:** `fig.width = numeric`
- caching computations
  - add argument to code chunk: `cache = TRUE`
  - computes and stores result of code the first time it is run, and calls the stored result directly from file for each subsequent call
  - useful for complex computations
  - caveats:
    - \* if data/code changes, you will need to re-run cached code chunks
    - \* dependencies not checked explicitly (changes in other parts of the code –> need to re-run the cached code)
    - \* if code does something outside of the document (i.e. produce a png file), the operation cannot be cached
- “Knit HTML” button to process the document
  - alternatively, when not in RStudio, the process can be accomplished through the following
 

```
library(knitr); setwd(<working directory>); knit2html("document.Rmd");
browseURL("document.html")
```
- processing of `knitr` documents
  - author drafts R Markdown (.Rmd) –> `knitr` processes file to Markdown (.md) –> `knitr` converts file to HTML
  - **Note:** *author should NOT edit/save the .md or .mhtml document until you are done with the document*

## Communicating Results

- when presenting information, it is important to breakdown results of an analysis into different levels of granularity/detail
- below are listed in order of *least -> most specific*

### Hierarchy of Information - Research Paper

- *title/author list* -> description of topic covered
- *abstract* -> a few hundred words about the problem, motivation, and solution
- *body/results* -> detailed methods, results, sensitivity analysis, discussion/implication of findings
- *supplementary material* -> granular details about analysis performed and method used
- *code/detail* -> material to reproduce analysis/finding

### Hierarchy of Information - Email Presentation

- *subject line/sender* -> must have, should be concise/descriptive, summarize findings in one sentence if possible
- *email body* -> should be 1-2 paragraphs, brief description of problem/context, summarize findings/results
  - If action needs to be taken, suggest some options and make them as concrete as possible
  - If questions need to be addressed, try to make them yes/no
- *attachment(s)* -> R Markdown/report containing more details, should still stay concise (not pages of code)
- *links to supplementary materials* -> GitHub/web link to code, software, data

## RPubs

- built-in service with RStudio and works seamlessly with `knitr`
- after knitting the R Markdown file in HTML, you can click on **Publish** to publish the HTML document on RPubs
- **Note:** all publications to RPubs are publicly available immediately

## Reproducible Research Checklist

- **Checklist**
  - Are we doing good science?
  - Was any part of this analysis done by hand?
    - \* If so, are those parts *precisely* document?
    - \* Does the documentation match reality?
  - Have we taught a computer to do as much as possible (i.e. coded)?
  - Are we using a version control system?
  - Have we documented our software environment?
  - Have we saved any output that we cannot reconstruct from original data + code?
  - How far back in the analysis pipeline can we go before our results are no longer (automatically) reproducible?

### Do's

- ***start with good science***
  - work on interesting problem (to you & other people)
  - form coherent/focused question to simplify problem
  - collaborate with others to reinforce good practices and habits
- ***teach a computer***
  - worthwhile to automate all tasks through script/programming
  - code = precise instructions to process/analyze data
  - teaching the computer almost guarantee's reproducibility
  - `download.file("url", "filename")` -> convenient way to download file
    - \* full URL specified (instead series of links/clicks)
    - \* name of file specified
    - \* directory specified
    - \* code can be executed in R (as long as link is available)
- ***use version control***
  - GitHub/BitBucket are good tools
  - helps to slow down
    - \* forces the author to think about changes made and commit changes and keep track of analysis performed
  - helps to keep track of history/snapshots
  - allows reverting to old versions
- ***keep track of software environment***
  - some tools/datasets may only work on certain software/environment
    - \* software and computing environment are critical to reproducing analysis
    - \* everything should be documented
  - *computer architecture*: CPU (Intel, AMD, ARM), GPUs, 32 vs 64bit
  - *operating system*: Windows, Mac OS, Linux/Unix
  - *software toolchain*: compilers, interpreters, command shell, programming languages (C, Perl, Python, etc.), database backends, data analysis software
  - *supporting software/infrastructure*: Libraries, R packages, dependencies
  - *external dependencies*: web sites, data repositories (data source), remote databases, software repositories

- *version numbers*: ideally, for everything (if available)
- `sessionInfo()` = prints R version, operating system, local, base/attached/utilized packages
- ***set random number generator seed***
  - random number generators produce pseudo-random numbers based on initial seed (usually number/set of numbers)
    - \* `set.seed()` can be used to specify seed for random generator in R
  - setting seed allows stream of random numbers to be reproducible
  - whenever you need to generate stream of random numbers for non-trivial purpose (i.e. simulations, Markov Chain Monte Carlo analysis), **always** set the seed.
- ***think about entire pipeline***
  - data analysis is length process, important to ensure each piece is reproducible
    - \* final product is important, but the process is just as important
  - raw data → processed data → analysis → report
  - the more of the pipeline that is made reproducible, the more credible the results are

## Don'ts

- ***do things by hand***
  - may lead to unreproducible results
    - \* edit spreadsheets using Excel
    - \* remove outliers (without noting criteria)
    - \* edit tables/figures
    - \* validate/quality control for data
  - downloading data from website (clicking on link)
    - \* need lengthy set of instructions to obtain the same data set
  - moving/split/reformat data (no record of what was done)
  - if necessary, manual tasks must be documented precisely (account for people with different background/context)
- ***point and click***
  - graphical user interfaces (GUI) make it easy to process/analyze data
    - \* GUIs are intuitive to use but actions are **difficult** to track and for others to reproduce
    - \* some GUIs include log files that can be utilized for review
  - any interactive software should be carefully used to ensure all results can be reproduced
  - text editors are usually ok, but when in doubt, document it
- ***save output for convenience***
  - avoid saving data analysis output
    - \* tables, summaries, figures, processed, data
  - output saved stand-alone without code/steps on how it is produced is not reproducible
  - when data changes or error is detected in parts of analysis the output is dependent on, the original graph/table will not be updated
  - intermediate files (processed data) are ok to keep but clear/precise documentation must be created
  - should save the **data/code** instead of the output

## Replication vs Reproducibility

- **Replication**

- focuses on validity of scientific claim
- if a study states “if X is related to Y” then we ask “is the claim true?”
- need to reproduce results with new investigators, data, analytical methods, laboratories, instruments, etc.
- particularly important in studies that can impact broad policy or regulatory decisions
- stands as **ultimate standard** for strengthening scientific evidence

- **Reproducibility**

- focuses on validity of data analysis
- “Can we trust this analysis?”
- reproduce results new investigators, same data, same methods
- important when replication is impossible
- arguably a **minimum standard** for any scientific study

## Background and Underlying Trend for Reproducibility

- some studies cannot be replicated (money/time/unique/opportunistic) -> 25 year studies in epidemiology
- technology increases rate of collection/complexity of data
- existing databases merged to become bigger databases (sometimes used off-label) -> administrative data used in health studies
- computing power allows for more sophisticated analyses
- “computational” fields are ubiquitous
- all of the above leads to the following
  - analysis are difficult to describe
  - inadequate training in statistics/computing and cause errors to be more easily produced through the pipeline
  - large data throughput, complex analysis, and failure to explain the process can inhibit knowledge transfer
  - results are difficult to replicate (knowledge/cost)
  - complicated analyses are not as easily trusted

## Problems with Reproducibility

- aiming for reproducibility provides for
  - transparency of data analysis
    - \* check validity of analysis-> people check each other -> “self-correcting” community
    - \* re-run the analysis
    - \* check the code for bugs/errors/sensitivity
    - \* try alternate approaches
  - data/software/methods availability
  - improved transfer of knowledge
    - \* focuses on “downstream” aspect of research dissemination (does solve problems with erroneous analysis)
- reproducibility does not guarantee

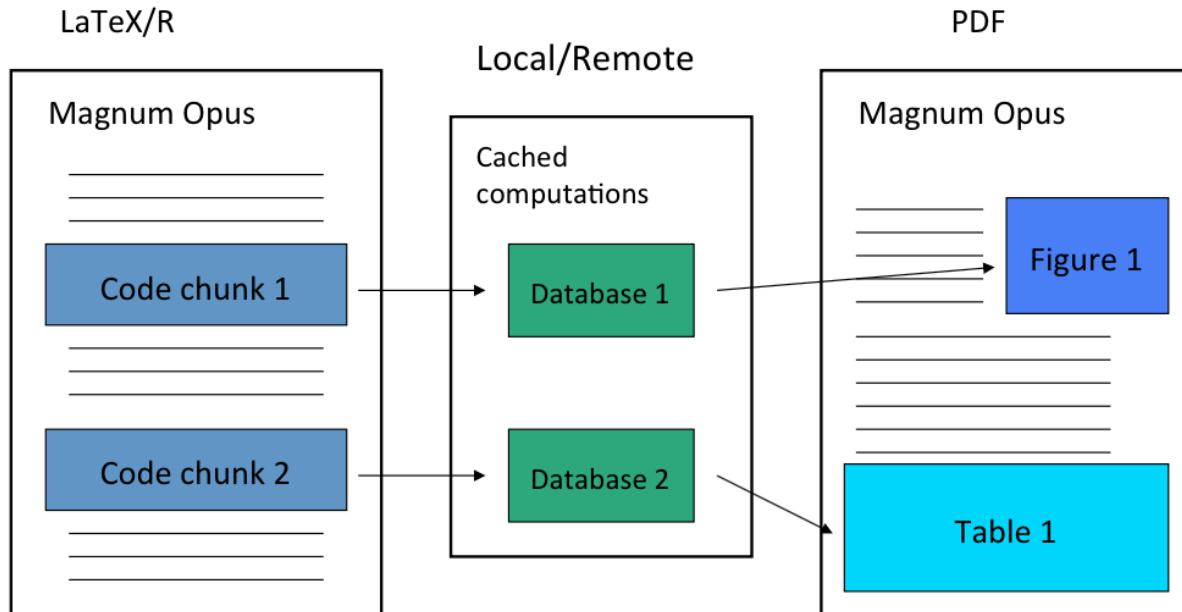
- correctness of analysis (reproducible  $\neq$  true)
- trustworthiness of analysis
- deterrence of bad analysis

## Evidence-based Data Analysis

- create analytic pipelines from evidence-based components – standardize it (“Deterministic Statistical Machine”)
  - apply thoroughly-studied (via statistical research), *mutually-agreed-upon methods* to analyze data whenever possible
  - once pipeline is established, shouldn’t change it (“transparent box”)
  - this reduces the “researcher degrees of freedom” (restricting the ability to alter different parts of analysis pipeline)
    - \* analogous to a pre-specified clinical trial protocol
  - **example:** time series of air pollution/health
    - \* *key question:* “Are short-term changes in pollution associated with short-term changes in a population health outcome?”
    - \* long history of statistical research investigating proper methods of analysis
    - \* pipeline
      1. check for outliers, high leverage, overdispersion  $\rightarrow$  skewedness of data
      2. Fill in missing data?  $\rightarrow$  NO! (systematic missing data, imputing would add noise)
      3. model selection: estimate degrees of freedom to adjust for unmeasured confounding variables
      4. multiple lag analysis
      5. sensitivity analysis with respect to unmeasured confounder adjustment/influential points
- this provides standardized, best practices for given scientific areas and questions
- gives reviewers an important tool without dramatically increasing the burden on them
- allows more effort to be focused on improving the quality of “upstream” aspects of scientific research

## Caching Computations

- Literate (Statistical) Programming
  - paper/code chunks → database/cached computations → published report



### cacher Package

- evaluates code in files and stores intermediate results in a key-value database
- R expressions are given ***SHA-1 hash values*** so changes can be tracked and code can be re-evaluated if necessary
- **cacher packages**
  - used by authors to create data analyses packages for distribution
  - used by readers to clone analysis and inspect/evaluate subset of code/data objects → readers may not have resources/want to see entire analysis
- **conceptual model**
  - dataset/code → source file → result
- **process for authors**
  - **cacher** package parses R source files and creates necessary cache directories/subdirectories
  - if expression = never evaluated
    - \* evaluate it and store any results R objects in cached data (similar to `cache = TRUE` function for `knitr`)
  - else if cached result exists
    - \* lazy load the results from cache database and move on to next expression
  - if expression *does not* create any R objects (nothing to cache)
    - \* add expression to list where evaluation needs to be forced
  - write out metadata for this expression to metadata file

- `cachepackage` function creates `cacher` package storing
  - \* source file
  - \* cached data objects
  - \* metadata
- package file is zipped and can be distributed
- **process for readers**
  - a journal article can say that “the code and data for this analysis can be found in the `cacher` package 092dcc7dda4b93e42f23e038a60e1d44dbec7b3f”
    - \* the code is the SHA-1 hash code and can be loaded/cloned using the `cacher` package
  - when analysis is cloned, local directories are created, source files/metadata are downloaded
    - \* data objects are ***NOT*** downloaded by default
    - \* references to data objects are loaded and corresponding data can be lazy-loaded on demand
      - when user examines it, then it is downloaded/calculated
  - readers are free to examine and evaluate the code
    - \* `clone(id = "#####")` = loads data from cache
      - using the first 4 characters is generally enough to identify
    - \* `showfiles()` = lists R scripts available in cache
    - \* `sourcefile("name.R")` = loads cached R file
    - \* `code()` = prints the content of the R file line by line
    - \* `graphcode()` = plots a graph to demonstrate dependencies/structure of code
    - \* `objectcode("object")` = shows lines of code that were used to generate that specific object (tracing all the way back to reading data)
    - \* `runcode()` = executes code by loading data from cached database (much faster than regular)
      - by default, expressions that result in object being created are ***NOT*** run and are loaded from cached databases
      - expressions not resulting in object ***ARE*** evaluated (i.e. plots)
    - \* `checkcode()` = evaluates all expressions from scratch
      - results of evaluation are checked against stored results to see if they are the same
      - setting random number generators are critical for this to work
    - \* `checkobjects()` = check for integrity of data objects (i.e. see if there are possible data corruption)
    - \* `loadcache()` = loads pointers to data objects in the data base
      - when a specific object is called, cache is then transferred and the data is downloaded

- **example**

```
library(cacher)
clonecache(id = "092dcc7dda4b93e42f23e038a60e1d44dbec7b3f")
clonecache(id = "092d") ## effectively the same as above
# output: created cache directory '.cache'
showfiles() # show files stored in cache
# output: [1] "top20.R"
sourcefile("top20.R") # load R script

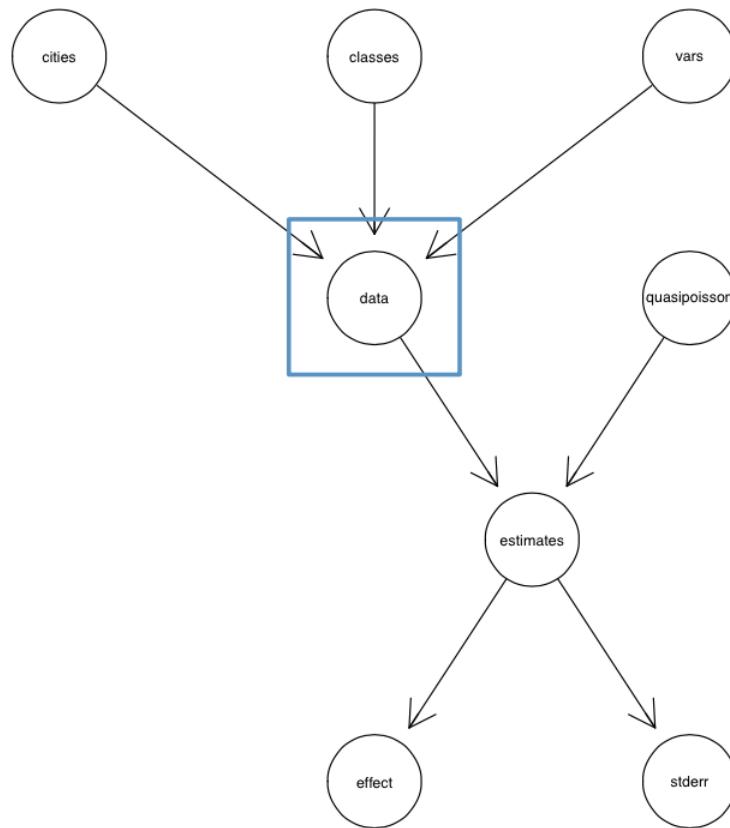
code() # examine the content of the code
# output:
# source file: top20.R
# 1 cities <- readLines("citylist.txt")
# 2 classes <- readLines("colClasses.txt")
```

```

# 3 vars <- c("date", "dow", "death",
# 4 data <- lapply(cities, function(city) {
# 5 names(data) <- cities
# 6 estimates <- sapply(data, function(city) {
# 7 effect <- weighted.mean(estimates[1,
# 8 stderr <- sqrt(1/sum(1/estimates[2,

graphcode() # generate graph showing structure of code

```



```

objectcode("data")
# output:
# source file: top20.R
# 1 cities <- readLines("citylist.txt")
# 2 classes <- readLines("colClasses.txt")
# 3 vars <- c("date", "dow", "death", "tmpd", "rmtmpd", "dptp", "rmdptp", "l1pm10tmean")
# 4 data <- lapply(cities, function(city) {
#         filename <- file.path("data", paste(city, "csv", sep = "."))
#         d0 <- read.csv(filename, colClasses = classes, nrow = 5200)
#         d0[, vars]
#     })
# 5 names(data) <- cities

loadcache()
ls()
# output:

```

```
# [1] "cities"      "classes"       "data"        "effect"
# [5] "estimates"   "stderr"         "vars"

cities
# output:
# / transferring cache db file b8fd490bcf1d48cd06...
# [1] "la"    "ny"    "chic"   "dlft"   "hous"   "phoe"
# [7] "staa"  "sand"  "miam"   "det"    "seat"   "sanb"
# [13] "sanj"  "minn"  "rive"   "phil"   "atla"   "oakl"
# [19] "denv"  "clev"

effect
# output:
# / transferring cache db file 584115c69e5e2a4ae5...
# [1] 0.0002313219

stderr
# output:
# / transferring cache db file 81b6dc23736f3d72c6...
# [1] 0.000052457
```

## Case Study: Air Pollution

- reanalysis of data from MMAPS and link with PM chemical constituent data
  - Lippmann *et al.* found strong evidence that Ni modified the short-term effect of  $PM_{10}$  across 60 US communities
  - National Morbidity, Mortality, and Air Pollution Study (NMMAPS) = national study of the short-term health effects of ambient air pollution
    - \* focused primarily on particulate matter ( $PM_{10}$ ) and ozone ( $O_3$ )
    - \* health outcomes included mortality from all causes and hospitalizations for cardiovascular and respiratory diseases
  - \* Data made available at the Internet-based Health and Air Pollution Surveillance System (iHAPSS)

## Research

---

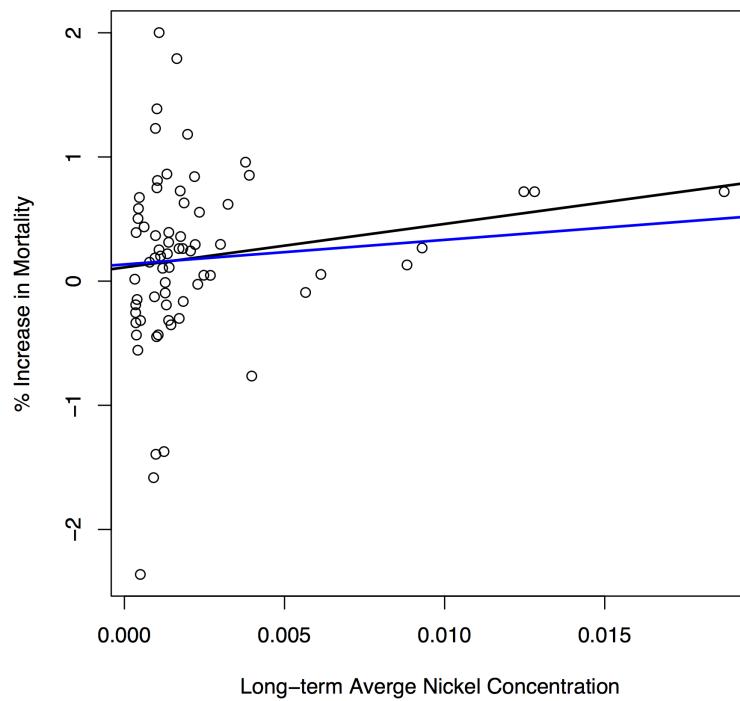
### Does the Effect of $PM_{10}$ on Mortality Depend on PM Nickel and Vanadium Content? A Reanalysis of the NMMAPS Data

**Francesca Dominici,<sup>1</sup> Roger D. Peng,<sup>1</sup> Keita Ebisu,<sup>2</sup> Scott L. Zeger,<sup>1</sup> Jonathan M. Samet,<sup>3</sup> and Michelle L. Bell<sup>2</sup>**

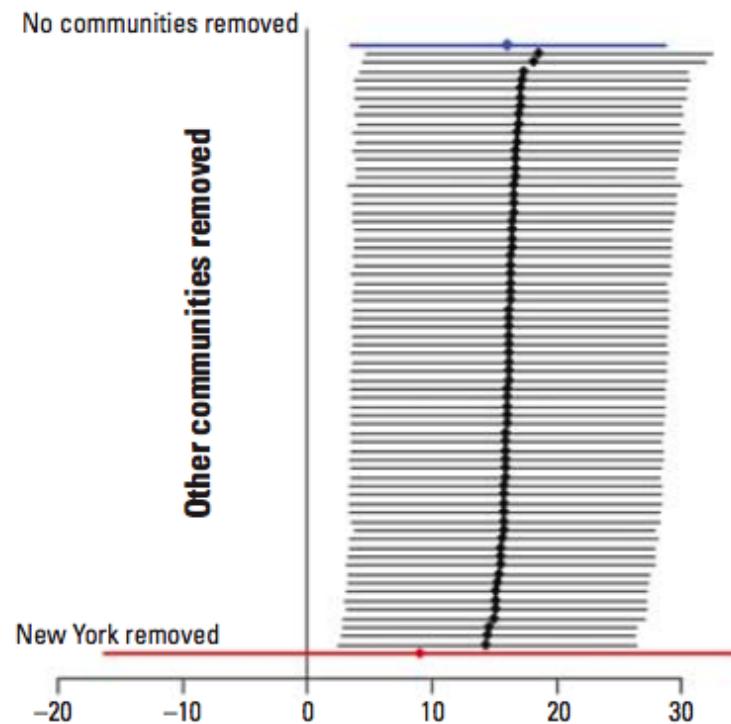
<sup>1</sup>Department of Biostatistics, Johns Hopkins Bloomberg School of Public Health, Baltimore, Maryland, USA; <sup>2</sup>School of Forestry and Environmental Studies, Yale University, New Haven, Connecticut, USA; <sup>3</sup>Department of Epidemiology, Johns Hopkins Bloomberg School of Public Health, Baltimore, Maryland, USA

#### Analysis: Does Nickel Make PM Toxic?

- Long-term average nickel concentrations appear correlated with PM risk ( $p < 0.01 \rightarrow$  statistically significant)
- there appear to be some outliers on the right-hand side (New York City)
  - adjusting the data by removing the New York counties altered the regression line (shown in blue)
  - though the relationship is still positive, regression line no longer statistically significant ( $p < 0.31$ )



- sensitivity analysis shows that the regression line is particularly sensitive to the New York data



## **Conclusions and Lessons Learnt**

- New York has very high levels of nickel and vanadium, much higher than any other US community
- there is evidence of a positive relationship between Ni concentrations and  $PM_{10}$  risk
- strength of this relationship is highly sensitive to the observations from New York City
- most of the information in the data is derived from just 3 observations
- reproducibility of NMMAPS allowed for a secondary analysis (and linking with PM chemical constituent data) investigating a novel hypothesis (Lippmann *et al.*), as well as a critique of that analysis and new findings (Dominici *et al.*)
- original hypothesis not necessarily invalidated, but evidence not as strong as originally suggested (more work should be done)

## Case Study: High Throughput Biology

### Genomic signatures to guide the use of chemotherapeutics

Anil Potti<sup>1,2</sup>, Holly K Dressman<sup>1,3</sup>, Andrea Bild<sup>1,3</sup>, Richard F Riedel<sup>1,2</sup>, Gina Chan<sup>4</sup>, Robyn Sayer<sup>4</sup>, Janiel Cragun<sup>4</sup>, Hope Cottrell<sup>4</sup>, Michael J Kelley<sup>2</sup>, Rebecca Petersen<sup>5</sup>, David Harpole<sup>5</sup>, Jeffrey Marks<sup>5</sup>, Andrew Berchuck<sup>1,6</sup>, Geoffrey S Ginsburg<sup>1,2</sup>, Phillip Febbo<sup>1–3</sup>, Johnathan Lancaster<sup>4</sup> & Joseph R Nevins<sup>1–3</sup>

- Potti *et al.* published in 2006 that microarray data from cell lines (the NCI60) can be used to define drug response “signatures”, which can be used to predict whether patients will respond
- however, analysis performed were fundamentally flawed as there exist much misclassification and mishandling of data and were unreproducible
- the results of their study, even after being corrected twice, still contained many errors and were unfortunately used as guidelines for clinical trials
- the fiasco with this paper and associated research, though spectacular in its own light, was by no means an unique occurrence

#### Conclusions and Lessons Learnt

- most common mistakes are simple
  - confounding in the experimental design
  - mixing up the sample labels
  - mixing up the gene labels
  - mixing up the group labels
- most mixups involve simple switches or offsets
  - this simplicity is often hidden due to incomplete documentation
- research papers should include the following (particularly those that would potentially lead to clinical trials)
  - data (with column names)
  - provenance (who owns what data/which sample is which)
  - code
  - descriptions of non-scriptable steps
  - descriptions of planned design (if used)
- **reproducible research** is the key to minimize these errors
  - literate programming
  - reusing templates
  - report structure
  - executive summaries
  - appendices (sessionInfo, saves, file location)

# Statistical Inference Course Notes

*Xing Su*

## Contents

Overview . . . . .	3
Probability . . . . .	3
General Probability Rules . . . . .	3
Conditional Probability . . . . .	5
Baye's Rule . . . . .	5
Random Variables . . . . .	6
Probability Mass Function (PMF) . . . . .	6
Probability Density Function (PDF) . . . . .	6
Cumulative Distribution Function (CDF) . . . . .	7
Survival Function . . . . .	7
Quantile . . . . .	7
Independence . . . . .	8
IID Random Variables . . . . .	8
Diagnostic Test . . . . .	9
Example . . . . .	9
Likelihood Ratios . . . . .	9
Expected Values/Mean . . . . .	11
Variance . . . . .	14
Sample Variance . . . . .	14
Entire Estimator-Estimation Relationship . . . . .	16
Example - Standard Normal . . . . .	17
Example - Standard Uniform . . . . .	17
Example - Poisson . . . . .	17
Example - Bernoulli . . . . .	18
Example - Father/Son . . . . .	18
Binomial Distribution . . . . .	20
Example . . . . .	20
Normal Distribution . . . . .	21
Example . . . . .	22
Poisson Distribution . . . . .	23
Example . . . . .	23

Example - Approximating Binomial Distribution	23
Asymptotics	25
Law of Large Numbers (LLN)	25
Example - LLN for Normal and Bernoulli Distribution	25
Central Limit Theorem	26
Example - CLT with Bernoulli Trials (Coin Flips)	26
Confidence Intervals - Normal Distribution/Z Intervals	27
Confidence Interval - Bernoulli Distribution/Wald Interval	28
Confidence Interval - Binomial Distribution/Agresti-Coull Interval	29
Confidence Interval - Poisson Interval	31
Confidence Intervals - T Distribution(Small Samples)	33
Confidence Interval - Paired T Tests	34
Independent Group t Intervals - Same Variance	36
Independent Group t Intervals - Different Variance	36
Hypothesis Testing	38
Power	41
Multiple Testing	44
Type of Errors	44
Error Rates	44
Example	45
Resample Inference	47

## Overview

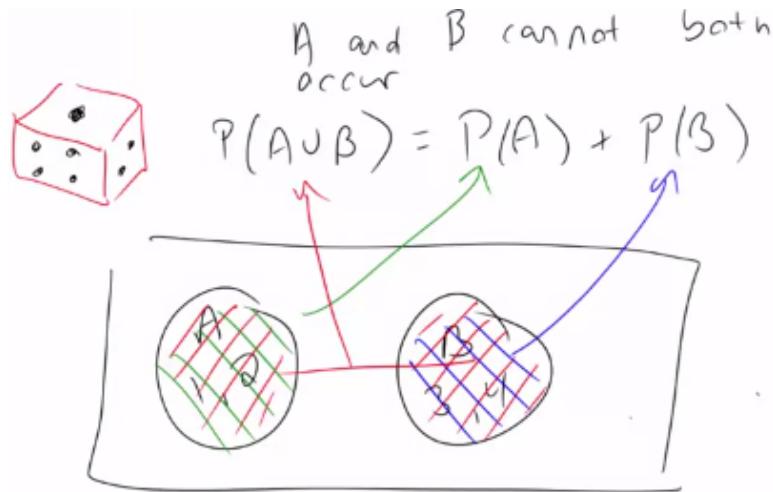
- **Statistical Inference** = generating conclusions about a population from a noisy sample
- Goal = extend beyond data to population
- Statistical Inference = only formal system of inference we have
- many different modes, but **two** broad flavors of inference (inferential paradigms): **Bayesian** vs **Frequentist**
  - **Frequentist** -> uses long run proportion of times an event occurs independent identically distributed repetitions
    - \* frequentist is what this class is focused on
    - \* believes if an experiment is repeated many many times, the resultant percentage of success/something happening defines that population parameter
  - **Bayesian** -> probability estimate for a hypothesis is updated as additional evidence is acquired
- **statistic** = number computed from a sample of data
  - statistics are used to infer information about a population
- **random variable** = outcome from an experiment
  - deterministic processes (variance/means) produce additional random variables when applied to random variables, and they have their own distributions

## Probability

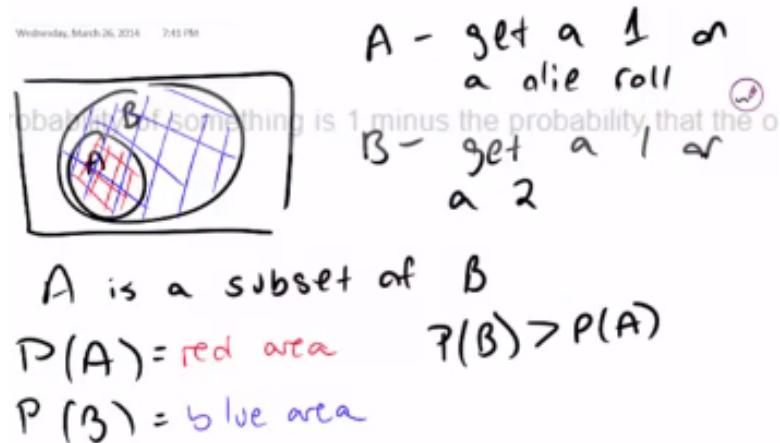
- **Probability** = the study of quantifying the likelihood of particular events occurring
  - given a random experiment, **probability** = population quantity that summarizes the randomness
    - \* not in the data at hand, but a conceptual quantity that exist in the population that we want to estimate

## General Probability Rules

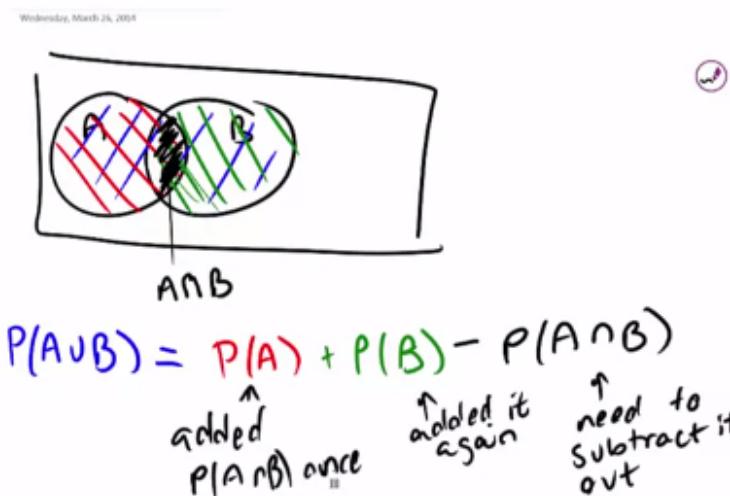
- discovered by Russian mathematician Kolmogorov, also known as “Probability Calculus”
- probability = function of any set of outcomes and assigns it a number between 0 and 1
  - $0 \leq P(E) \leq 1$ , where E = event
- probability that nothing occurs = 0 (impossible, have to roll dice to create outcome), that something occurs is 1 (certain)
- probability of outcome or event E,  $P(E)$  = ratio of ways that E could occur to number of all possible outcomes or events
- probability of something = 1 - probability of the opposite occurring
- probability of the **union** of any two sets of outcomes that have nothing in common (mutually exclusive) = sum of respective probabilities



- if A implies occurrence of B, then  $P(A)$  occurring <  $P(B)$  occurring



- for any two events, probability of at least one occurs = the sum of their probabilities - their intersection (in other words, probabilities can not be added simply if they have non-trivial intersection)



- for independent events A and B,  $P(A \cup B) = P(A) \times P(B)$
- for outcomes that can occur with different combination of events and these combinations are mutually exclusive, the  $P(E_{total}) = \sum P(E_{part})$

## Conditional Probability

- let B = an event so that  $P(B) > 0$
- **conditional probability** of an event A, given B is defined as the probability that BOTH A and B occurring divided by the probability of B occurring

$$P(A | B) = \frac{P(A \cap B)}{P(B)}$$

- if A and B are *independent*, then

$$P(A | B) = \frac{P(A)P(B)}{P(A)} = P(A)$$

- *example*

– for die roll,  $A = \{1\}$ ,  $B = \{1, 3, 5\}$ , then

$$P(1 | Odd) = P(A | B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)}{P(B)} = \frac{1/6}{3/6} = \frac{1}{3}$$

## Baye's Rule

- definition

$$P(B | A) = \frac{P(A | B)P(B)}{P(A | B)P(B) + P(A | B^c)P(B^c)}$$

where  $B^c$  = corresponding probability of event B,  $P(B^c) = 1 - P(B)$

## Random Variables

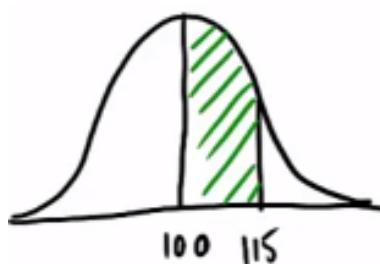
- **random variable** = numeric outcome of experiment
- **discrete** (what you can count/categories) = assign probabilities to every number/value the variable can take
  - coin flip, rolling a die, web traffic in a day
- **continuous** (any number within a continuum) = assign probabilities to the range the variable can take
  - BMI index, intelligence quotients
  - *Note: limitations of precision in taking the measurements may imply that the values are discrete, but we in fact consider them continuous*
- `rbinom()`, `rnorm()`, `rgamma()`, `rpois()`, `runif()` = functions to generate random variables from the binomial, normal, Gamma, Poisson, and uniform distributions
- density and mass functions (population quantities, not what occurs in data) for random variables = best starting point to model/think about probabilities for numeric outcome of experiments (variables)
  - use data to estimate properties of population -> linking sample to population

## Probability Mass Function (PMF)

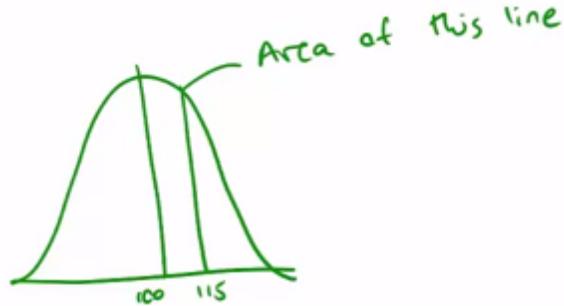
- evaluates the probability that the **discrete random variable** takes on a specific value
  - measures the chance of a particular outcome happening
  - always  $\geq 0$  for every possible outcome
  - $\sum$  possible values that the variable can take = 1
- **Bernoulli distribution example**
  - $X = 0 \rightarrow$  tails,  $X = 1 \rightarrow$  heads
  - \*  $X$  here represents potential outcome
  - $p(X = x) = (\frac{1}{2})^x (\frac{1}{2})^{1-x}$  for  $X = 0, 1$
  - \*  $x$  here represents a value we can plug into the PMF
  - \* general form ->  $p(x) = (\theta)^x (1 - \theta)^{1-x}$
- `dbinom(k, n, p)` = return the probability of getting  $k$  successes out of  $n$  trials, given probability of success is  $p$

## Probability Density Function (PDF)

- evaluates the probability that the **continuous random variable** takes on a specific value
  - always  $\geq$  everywhere
  - total area under the must = 1
- **areas under PDFs** correspond to the probabilities for that random variable taking on that range of values (PMF)



- but the probability of the variable taking a specific value = 0 (area of a line is 0)



- *Note: the above is true because it is modeling random variables as if they have infinite precision, when in reality they do not*
- `dnorm()`, `dgamma()`, `dpois()`, `dunif()` = return probability of a certain value from the normal, Gamma, Poisson, and uniform distributions

### Cumulative Distribution Function (CDF)

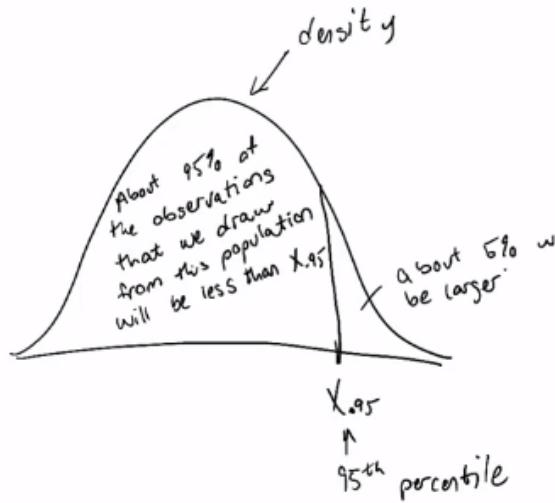
- CDF of a random variable X = probability that the random variable is  $\leq$  value x
  - $F(x) = P(X \leq x)$  <- applies when X is discrete/continuous
- PDF = derivative of CDF
  - integrate PDF  $\rightarrow$  CDF
    - \* `integrate(function, lower=0, upper=1)`  $\rightarrow$  can be used to evaluate integrals for a specified range
- `pbinom()`, `pnorm()`, `pgamma()`, `ppois()`, `punif()` = returns the cumulative probabilities from 0 up to a specified value from the binomial, normal, Gamma, Poisson, and uniform distributions

### Survival Function

- survival function of a random variable X = probability the random variable  $> x$ , complement of CDF
  - $S(x) = P(X > x) = 1 - F(x)$ , where  $F(x) = \text{CDF}$

### Quantile

- the  $\alpha^{th}$  quantile of a distribution with distribution function F = point  $x_\alpha$ 
  - $F(x_\alpha) = \alpha$
  - percentile = quantile with  $\alpha$  expressed as a percent
  - median = 50th percentile
  - $\alpha\%$  of the possible outcomes lie below it



- `qbeta(quantileInDecimals, 2, 1)` = returns quantiles for beta distribution
  - works for `qnorm()`, `qbinom()`, `qgamma()`, `qpois()`, etc.
- median estimated in this fashion = a population median
- probability model connects data to population using assumptions
  - population median = *estimand*, sample median = *estimator*

## Independence

- two events A and B are *independent* if the following is true
  - $P(A \cap B) = P(A)P(B)$
  - $P(A | B) = P(A)$
- two random variables X and Y are *independent*, if for any two sets, **A** and **B**, the following is true
  - $P([X \in A] \cap [Y \in B]) = P(X \in A)P(Y \in B)$
- **independence** = statistically unrelated from one another
- if A is *independent* of B, then the following are true
  - $A^c$  is independent of B
  - A is independent of  $B^c$
  - $A^c$  is independent of  $B^c$

## IID Random Variables

- random variables are said to be **IID** if they are *independent and identically distributed*
  - **independent** = statistically unrelated from each other
  - **identically distributed** = all having been drawn from the same population distribution
- IID random variables = default model for random samples = default starting point of inference

## Diagnostic Test

- Let + and - be the results, positive and negative respectively, of a diagnostic test
- Let  $D$  = subject of the test has the disease,  $D^c$  = subject does not
- sensitivity** =  $P(+ | D)$  = probability that the test is positive given that the subject has the disease (the higher the better)
- specificity** =  $P(- | D^c)$  = probability that the test is negative given that the subject does not have the disease (the higher the better)
- positive predictive value** =  $P(D | +)$  = probability that that subject has the disease given that the test is positive
- negative predictive value** =  $P(D^c | -)$  = probability that the subject does not have the disease given the test is negative
- prevalence of disease** =  $P(D)$  = marginal probability of disease

### Example

- specificity of 98.5%, sensitivity = 99.7%, prevalence of disease = .1%

$$\begin{aligned} P(D | +) &= \frac{P(+ | D)P(D)}{P(+ | D)P(D) + P(+ | D^c)P(D^c)} \\ &= \frac{P(+ | D)P(D)}{P(+ | D)P(D) + \{1 - P(- | D^c)\}\{1 - P(D)\}} \\ &= \frac{.997 \times .001}{.997 \times .001 + .015 \times .999} \\ &= .062 \end{aligned}$$

- low positive predictive value  $\rightarrow$  due to low prevalence of disease and somewhat modest specificity
  - suppose it was known that the subject uses drugs and has regular intercourse with an HIV infected partner (his probability of being + is higher than suspected)
  - evidence implied by a positive test result

## Likelihood Ratios

- diagnostic likelihood ratio** of a **positive** test result is defined as

$$DLR_+ = \frac{\text{sensitivity}}{1 - \text{specificity}} = \frac{P(+ | D)}{P(+ | D^c)}$$

- diagnostic likelihood ratio** of a **negative** test result is defined as

$$DLR_- = \frac{1 - \text{sensitivity}}{\text{specificity}} = \frac{P(- | D)}{P(- | D^c)}$$

- from Baye's Rules, we can derive the *positive predictive value* and *false positive value*

$$P(D | +) = \frac{P(+ | D)P(D)}{P(+ | D)P(D) + P(+ | D^c)P(D^c)} \quad (1)$$

$$P(D^c | +) = \frac{P(+ | D^c)P(D^c)}{P(+ | D)P(D) + P(+ | D^c)P(D^c)} \quad (2)$$

- if we divide equation (1) over (2), the quantities over have the same denominator so we get the following

$$\frac{P(D | +)}{P(D^c | +)} = \frac{P(+ | D)}{P(+ | D^c)} \times \frac{P(D)}{P(D^c)}$$

which can also be written as

$$\text{post-test odds of D} = DLR_+ \times \text{pre-test odds of D}$$

- **odds** =  $p/(1 - p)$
  - $\frac{P(D)}{P(D^c)}$  = **pre-test odds**, or odds of disease in absence of test
  - $\frac{P(D|+)}{P(+|D^c)}$  = **post-test odds**, or odds of disease given a positive test result
  - $DLR_+$  = factor by which the odds in the presence of a positive test can be multiplied to obtain the post-test odds
  - $DLR_-$  = relates the decrease in odds of disease after a negative result
- following the previous example, for sensitivity of 0.997 and specificity of 0.985, so the diagnostic likelihood ratios are as follows

$$DLR_+ = .997/(1 - .985) = 66 \quad DLR_- = (1 - .997)/.985 = 0.003$$

- this indicates that the result of the positive test is the odds of disease is 66 times the pretest odds

## Expected Values/Mean

- useful for characterizing a distribution (properties of distributions)
- **mean** = characterization of the center of the distribution = *expected value*
- expected value operation = *linear*  $\rightarrow E(aX + bY) = aE(X) + bE(Y)$
- **variance/standard deviation** = characterization of how spread out the distribution is
- *sample* expected values for sample mean and variance will estimate the *population* counterparts
- **population mean**
  - expected value/mean of a random variable = center of its distribution (center of mass)
  - **discrete variables**
    - \* for X with PMF  $p(x)$ , the population mean is defined as

$$E[X] = \sum_x xp(x)$$

where the sum is taken over **all** possible values of  $x$

- \*  $E[X]$  = center of mass of a collection of location and weights  $x, p(x)$
- \* *coin flip example:*  $E[X] = 0 \times (1-p) + 1 \times p = p$

### – **continuous variable**

- \* for X with PDF  $f(x)$ , the expected value = the center of mass of the density
- \* instead of summing over discrete values, the expectation **integrates** over a continuous function
  - PDF =  $f(x)$
  - $\int xf(x) =$  area under the PDF curve = mean/expected value of X

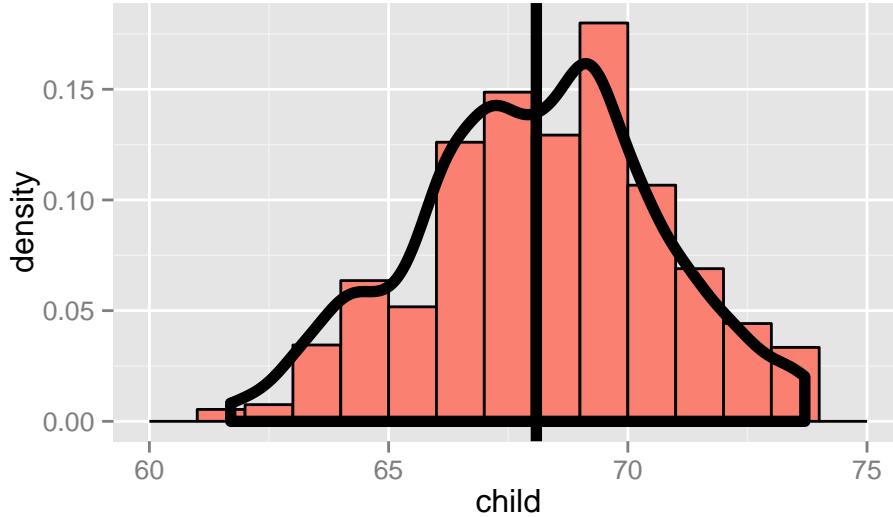
### • **sample mean**

- sample mean estimates the population mean
  - \* sample mean = center of mass of observed data = empirical mean

$$\bar{X} = \sum_x^n x_i p(x_i)$$

where  $p(x_i) = 1/n$

```
# load relevant packages
library(UsingR); data(galton); library(ggplot2)
# plot galton data
g <- ggplot(galton, aes(x = child))
# add histogram for children data
g <- g + geom_histogram(fill = "salmon", binwidth=1, aes(y=..density..), colour="black")
# add density smooth
g <- g + geom_density(size = 2)
# add vertical line
g <- g + geom_vline(xintercept = mean(galton$child), size = 2)
# print graph
g
```



- **average of random variables** = a new random variable where its distribution has an expected value that is the **same** as the original distribution (centers are the same)

- the mean of the averages = average of the original data  $\rightarrow$  estimates average of the population
  - if  $E[\text{sample mean}] = \text{population mean}$ , then estimator for the sample mean is **unbiased**
  - \* [derivation] let  $X_1, X_2, X_3, \dots, X_n$  be a collection of  $n$  samples from the population with mean  $\mu$
  - \* mean of this sample

$$\bar{X} = \frac{X_1 + X_2 + X_3 + \dots + X_n}{n}$$

- \* since  $E(aX) = aE(X)$ , the expected value of the mean is can be written as

$$E\left[\frac{X_1 + X_2 + X_3 + \dots + X_n}{n}\right] = \frac{1}{n} \times [E(X_1) + E(X_2) + E(X_3) + \dots + E(X_n)]$$

- \* since each of the  $E(X_i)$  is drawn from the population with mean  $\mu$ , the expected value of each sample should be

$$E(X_i) = \mu$$

- \* therefore

$$\begin{aligned} E\left[\frac{X_1 + X_2 + X_3 + \dots + X_n}{n}\right] &= \frac{1}{n} \times [E(X_1) + E(X_2) + E(X_3) + \dots + E(X_n)] \\ &= \frac{1}{n} \times [\mu + \mu + \mu + \dots + \mu] \\ &= \frac{1}{n} \times n \times \mu \\ &= \mu \end{aligned}$$

- **Note:** the more data that goes into the sample mean, the more concentrated its density/mass functions are around the population mean

```

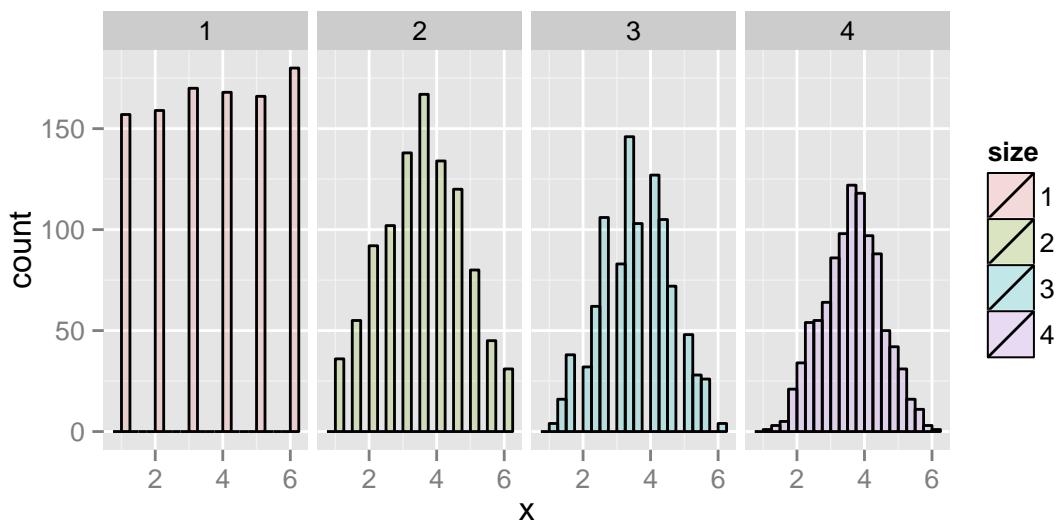
nosim <- 1000
# simulate data for sample size 1 to 4
dat <- data.frame(
  x = c(sample(1 : 6, nosim, replace = TRUE),
        apply(matrix(sample(1 : 6, nosim * 2, replace = TRUE), nosim), 1, mean),

```

```

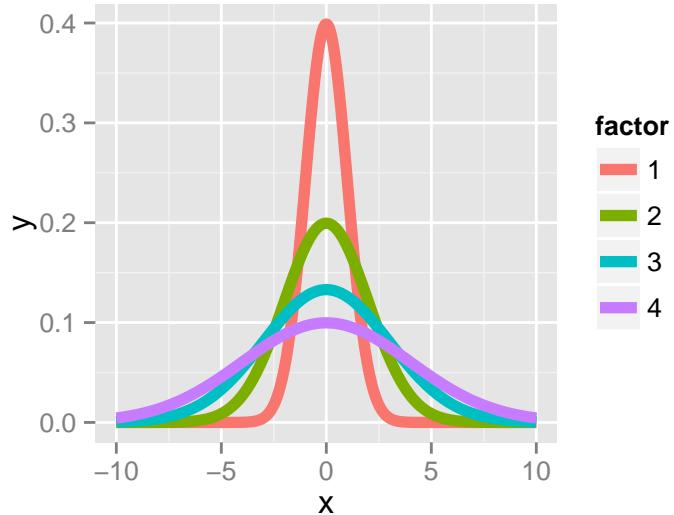
apply(matrix(sample(1 : 6, nosim * 3, replace = TRUE), nosim), 1, mean),
apply(matrix(sample(1 : 6, nosim * 4, replace = TRUE), nosim), 1, mean)),
size = factor(rep(1 : 4, rep(nosim, 4))))
# plot histograms of means by sample size
g <- ggplot(dat, aes(x = x, fill = size)) + geom_histogram(alpha = .20, binwidth=.25, colour = "black")
g + facet_grid(. ~ size)

```



## Variance

```
# generate x value ranges
xvals <- seq(-10, 10, by = .01)
# generate data from normal distribution for sd of 1 to 4
dat <- data.frame(
  y = c(dnorm(xvals, mean = 0, sd = 1),
        dnorm(xvals, mean = 0, sd = 2),
        dnorm(xvals, mean = 0, sd = 3),
        dnorm(xvals, mean = 0, sd = 4)),
  x = rep(xvals, 4),
  factor = factor(rep(1 : 4, rep(length(xvals), 4)))
)
# plot 4 lines for the different standard deviations
ggplot(dat, aes(x = x, y = y, color = factor)) + geom_line(size = 2)
```

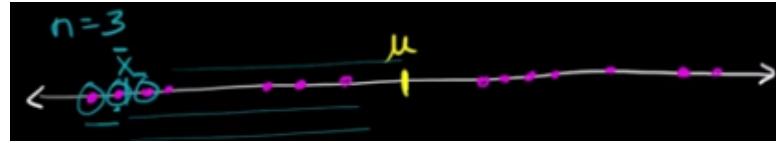


- **variance** = measure of spread, the square of expected distance from the mean (expressed in X's units<sup>2</sup>)
  - as we can see from above, higher variances → more spread, lower → smaller spread
  - $Var(X) = E[(X - \mu)^2] = E[X^2] - E[X]^2$
  - **standard deviation** =  $\sqrt{Var(X)}$  → has same units as X
  - **example**
    - \* for die roll,  $E[X] = 3.5$
    - \*  $E[X^2] = 12 \times 1/6 + 22 \times 1/6 + 32 \times 1/6 + \dots + 62 \times 1/6 = 15.17$
    - \*  $Var(X) = E[X^2] - E[X]^2 \approx 2.92$
  - **example**
    - \* for coin flip,  $E[X] = p$
    - \*  $E[X^2] = 0^2 \times (1 - p) + 1^2 \times p = p$
    - \*  $Var(X) = E[X^2] - E[X]^2 = p - p^2 = p(1 - p)$

## Sample Variance

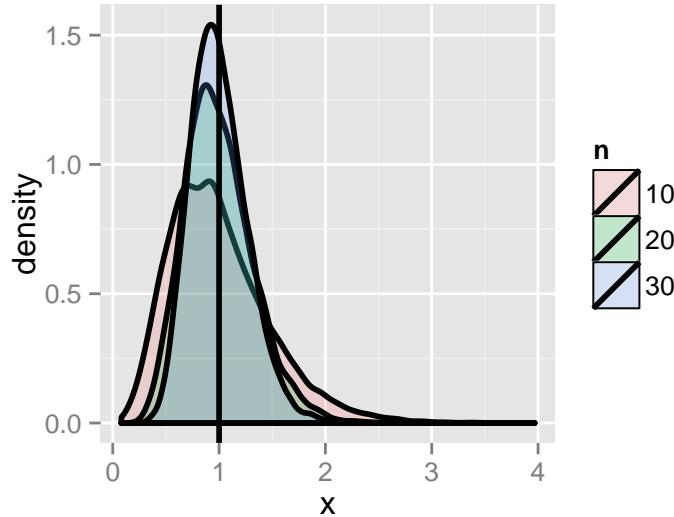
- the **sample variance** is defined as

$$S^2 = \frac{\sum_{i=1} (X_i - \bar{X})^2}{n - 1}$$

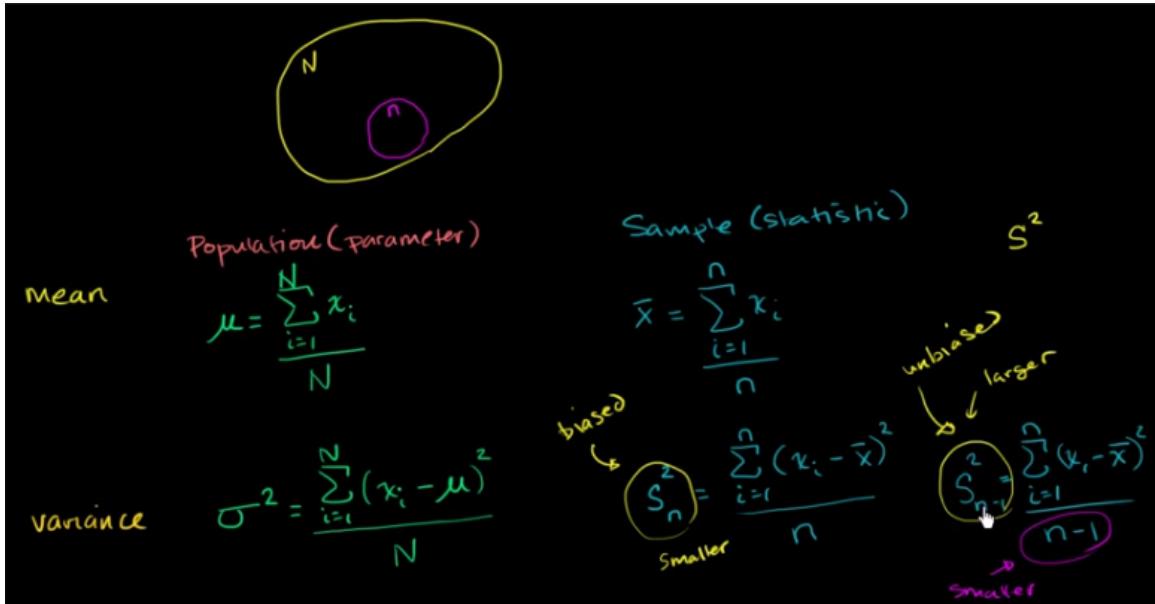


- on the above line representing the population (in magenta), any subset of data (3 of 14 selected, marked in blue) will most likely have a variance that is *lower than* the population variance
- dividing by  $n - 1$  will make the variance estimator *larger* to adjust for this fact  $\rightarrow$  leads to more accurate estimation  $\rightarrow S^2$  = so called ***unbiased estimate of population variance***
  - $S^2$  is a random variable, and therefore has an associated population distribution
  - \*  $E[S^2] = \text{population variance}$ , where  $S = \text{sample standard deviation}$
  - \* as we see from the simulation results below, with more data, the distribution for  $S^2$  gets more concentrated around population variance

```
# specify number of simulations
nosim <- 10000;
# simulate data for various sample sizes
dat <- data.frame(
  x = c(apply(matrix(rnorm(nosim * 10), nosim), 1, var),
        apply(matrix(rnorm(nosim * 20), nosim), 1, var),
        apply(matrix(rnorm(nosim * 30), nosim), 1, var)),
  n = factor(rep(c("10", "20", "30"), c(nosim, nosim, nosim))) )
# plot density function for different sample size data
ggplot(dat, aes(x = x, fill = n)) + geom_density(size = 1, alpha = .2) +
  geom_vline(xintercept = 1, size = 1)
```



- Note:** for any variable, properties of the population = **parameter**, estimates of properties for samples = **statistic**
  - below is a summary for the mean and variance for population and sample



- distribution for mean of random samples

- expected value of the **mean** of distribution of means = expected value of the sample = population mean
  - \*  $E[\bar{X}] = \mu$
- expected value of the variance of distribution of means
  - \*  $Var(\bar{X}) = \sigma^2/n$
  - \* as  $n$  becomes larger, the mean of random sample  $\rightarrow$  more concentrated around the population mean  $\rightarrow$  variance approaches 0
    - this again confirms that sample mean estimates population mean
- **Note:** normally we only have 1 sample mean (from collected sample) and can estimate the variance  $\sigma^2$   $\rightarrow$  so we know a lot about the **distribution of the means** from the data observed

- standard error (SE)

- the standard error of the mean is defined as

$$SE_{mean} = \sigma / \sqrt{n}$$

- this quantity is effectively the standard deviation of the distribution of a statistic (i.e. mean)
- represents variability of means

### Entire Estimator-Estimation Relationship

- Start with a sample
- $S^2$  = sample variance
  - estimates how variable the population is
  - estimates population variance  $\sigma^2$
  - $S^2$  = a random variable and has its own distribution centered around  $\sigma^2$ 
    - \* more concentrated around  $\sigma^2$  as  $n$  increases
- $\bar{X}$  = sample mean
  - estimates population mean  $\mu$

- $\bar{X}$  = a random variable and has its own distribution centered around  $\mu$ 
  - \* more concentrated around  $\mu$  as  $n$  increases
  - \* variance of distribution of  $\bar{X} = \sigma^2/n$
  - \* estimate of variance =  $S^2/n$
  - \* estimate of standard error =  $S/\sqrt{n}$  –> “sample standard error of the mean”
    - estimates how variable sample means ( $n$  size) from the population are

### Example - Standard Normal

- variance = 1
- means of  $n$  standard normals (sample) have standard deviation =  $1/\sqrt{n}$

```
# specify number of simulations with 10 as number of observations per sample
nosim <- 1000; n <- 10
# estimated standard deviation of mean
sd(apply(matrix(rnorm(nosim * n), nosim), 1, mean))
```

```
## [1] 0.3153689
```

```
# actual standard deviation of mean of standard normals
1 / sqrt(n)
```

```
## [1] 0.3162278
```

- `rnorm()` –> generate samples from the standard normal
- `matrix()` –> puts all samples into a `nosim` by  $n$  matrix, so that each row represents a simulation with `nosim` observations
- `apply()` –> calculates the mean of the  $n$  samples
- `sd()` –> returns standard deviation

### Example - Standard Uniform

- standard uniform –> triangle straight line distribution –> mean =  $1/2$  and variance =  $1/12$
- means of random samples of  $n$  uniforms have standard deviation of  $1/\sqrt{12 \times n}$

```
# estimated standard deviation of the sample means
sd(apply(matrix(runif(nosim * n), nosim), 1, mean))
```

```
## [1] 0.09147552
```

```
# actual standard deviation of the means
1/sqrt(12*n)
```

```
## [1] 0.09128709
```

### Example - Poisson

- $Poisson(x^2)$  have variance of  $x^2$
- means of random samples of  $n$   $Poisson(4)$  have standard deviation of  $2/\sqrt{n}$

```
# estimated standard deviation of the sample means  
sd(apply(matrix(rpois(nosim * n, lambda=4), nosim), 1, mean))
```

```
## [1] 0.6449735
```

```
# actual standard deviation of the means  
2/sqrt(n)
```

```
## [1] 0.6324555
```

### Example - Bernoulli

- for  $p = 0.5$ , the Bernoulli distribution has variance of 0.25
- means of random samples of  $n$  coin flips have standard deviations of  $1/(2\sqrt{n})$

```
# estimated standard deviation of the sample means  
sd(apply(matrix(sample(0 : 1, nosim * n, replace = TRUE), nosim), 1, mean))
```

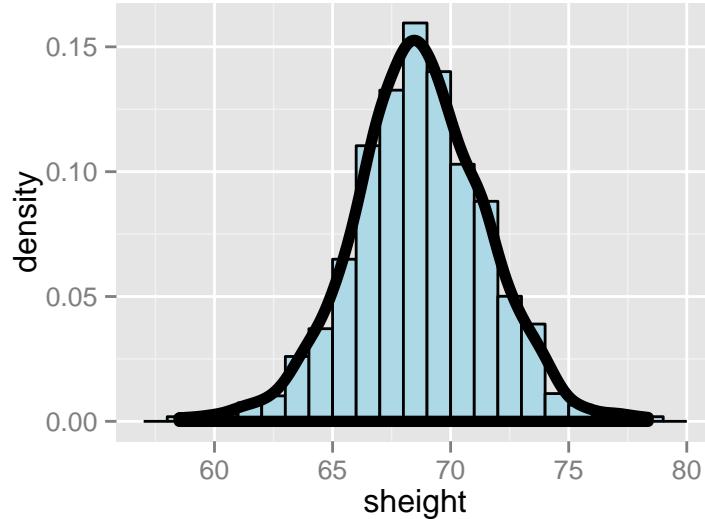
```
## [1] 0.1590323
```

```
# actual standard deviation of the means  
1/(2*sqrt(n))
```

```
## [1] 0.1581139
```

### Example - Father/Son

```
# load data  
library(UsingR); data(father.son);  
# define son height as the x variable  
x <- father.son$sheight  
# n is the length  
n<-length(x)  
# plot histogram for son's heights  
g <- ggplot(data = father.son, aes(x = sheight))  
g <- g + geom_histogram(aes(y = ..density..), fill = "lightblue", binwidth=1, colour = "black")  
g <- g + geom_density(size = 2, colour = "black")  
g
```



```
# we calculate the parameters for variance of distribution and sample mean,
round(c(sampleVar = var(x),
       sampleMeanVar = var(x) / n,
       # as well as standard deviation of distribution and sample mean
       sampleSd = sd(x),
       sampleMeanSd = sd(x) / sqrt(n)),2)
```

	sampleVar	sampleMeanVar	sampleSd	sampleMeanSd
##	7.92	0.01	2.81	0.09

## Binomial Distribution

- **binomial random variable** = sum of **n** Bernoulli variables

$$X = \sum_{i=1}^n X_i$$

where  $X_1, \dots, X_n = Bernoulli(p)$

- PMF is defined as

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x}$$

where  $\binom{n}{x}$  = number of ways selecting  $x$  items out of  $n$  options without replacement or regard to order and for  $x = 0, \dots, n$

- **combination** or “ $n$  choose  $x$ ” is defined as

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

- the base cases are

$$\binom{n}{n} = \binom{n}{0} = 1$$

- **Bernoulli distribution** → binary outcome

- only possible outcomes

- \* 1 = “success” with probability of  $p$
- \* 0 = “failure” with probability of  $1 - p$

- PMF is defined as

$$P(X = x) = p^x (1-p)^{1-x}$$

- mean =  $p$

- variance =  $p(1-p)$

### Example

- of 8 children, what's the probability of 7 or more girls (50/50 chance)?

$$\binom{8}{7} \cdot .5^7 (1 - .5)^1 + \binom{8}{8} \cdot .5^8 (1 - .5)^0 \approx 0.04$$

```
# calculate probability using PMF
```

```
choose(8, 7) * .5 ^ 8 + choose(8, 8) * .5 ^ 8
```

```
## [1] 0.03515625
```

```
# calculate probability using CMF from distribution
```

```
pbinom(6, size = 8, prob = .5, lower.tail = FALSE)
```

```
## [1] 0.03515625
```

- `choose(8, 7)` = R function to calculate  $n$  choose  $x$
- `pbinom(6, size=8, prob =0.5, lower.tail=TRUE)` = probability of 6 or less successes out of 8 samples with probability of 0.5 (CMF)
  - `lower.tail=FALSE` = returns the complement, in this case it's the probability of greater than 6 successes out of 8 samples with probability of 0.5

## Normal Distribution

- normal/Gaussian distribution for random variable X

– notation =  $X \sim N(\mu, \sigma^2)$

– mean =  $E[X] = \mu$

– variance =  $Var(X) = \sigma^2$

– PMF is defined as

$$f(x) = (2\pi\sigma^2)^{-1/2} e^{-(x-\mu)^2/2\sigma^2}$$

- $X \sim N(0, 1)$  = **standard normal distribution** (standard normal random variables often denoted using  $Z_1, Z_2, \dots$ )

– **Note:** see below graph for reference for the following observations

– ~68% of data/normal density  $\rightarrow$  between  $\pm 1$  standard deviation from  $\mu$

– ~95% of data/normal density  $\rightarrow$  between  $\pm 2$  standard deviation from  $\mu$

– ~99% of data/normal density  $\rightarrow$  between  $\pm 3$  standard deviation from  $\mu$

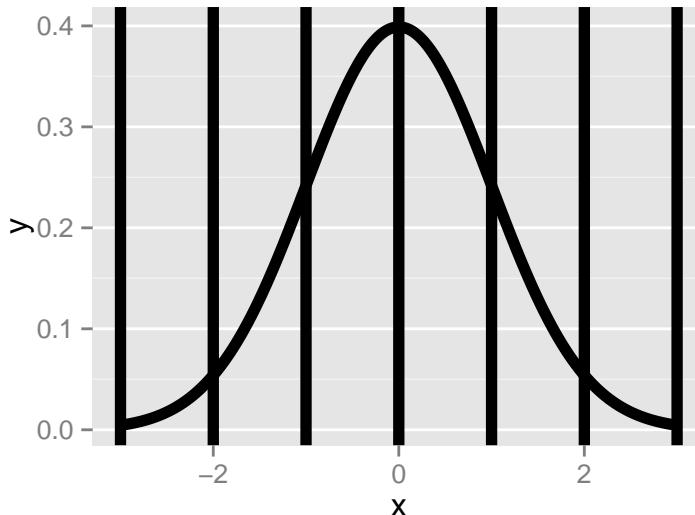
–  $\pm 1.28$  standard deviations from  $\mu \rightarrow 10^{th}$  (-) and  $90^{th}$  (+) percentiles

–  $\pm 1.645$  standard deviations from  $\mu \rightarrow 5^{th}$  (-) and  $95^{th}$  (+) percentiles

–  $\pm 1.96$  standard deviations from  $\mu \rightarrow 2.5^{th}$  (-) and  $97.5^{th}$  (+) percentiles

–  $\pm 2.33$  standard deviations from  $\mu \rightarrow 1^{st}$  (-) and  $99^{th}$  (+) percentiles

```
# plot standard normal
x <- seq(-3, 3, length = 1000)
g <- ggplot(data.frame(x = x, y = dnorm(x)),
             aes(x = x, y = y)) + geom_line(size = 2)
g <- g + geom_vline(xintercept = -3 : 3, size = 2)
g
```



- for any  $X \sim N(\mu, \sigma^2)$ , calculating the number of standard deviations each observation is from the mean **converts** the random variable to a **standard normal** (denoted as  $Z$  below)

$$Z = \frac{X - \mu}{\sigma} \sim N(0, 1)$$

- conversely, a standard normal can then be converted to *any normal distribution* by multiplying by standard deviation and adding the mean

$$X = \mu + \sigma Z \sim N(\mu, \sigma^2)$$

- `qnorm(n, mean=mu, sd=sd)` = returns the  $n^{th}$  percentiles for the given normal distribution
- `pnorm(x, mean=mu, sd=sd, lower.tail=F)` = returns the probability of an observation drawn from the given distribution is larger in value than the specified threshold  $x$

### Example

- the number of daily ad clicks for a company is (approximately) normally distributed with a mean of 1020 and a standard deviation of 50
- What's the probability of getting more than 1,160 clicks in a day?

```
# calculate number of standard deviations from the mean
(1160 - 1020) / 50
```

```
## [1] 2.8
```

```
# calculate probability using given distribution
pnorm(1160, mean = 1020, sd = 50, lower.tail = FALSE)
```

```
## [1] 0.00255513
```

```
# calculate probability using standard normal
pnorm(2.8, lower.tail = FALSE)
```

```
## [1] 0.00255513
```

- therefore, it is not very likely (0.255513% chance), since 1,160 is 2.8 standard deviations from the mean
- What number of daily ad clicks would represent the one where 75% of days have fewer clicks (assuming days are independent and identically distributed)?

```
qnorm(0.75, mean = 1020, sd = 50)
```

```
## [1] 1053.724
```

- therefore, 1053.7244875 would represent the threshold that has more clicks than 75% of days

## Poisson Distribution

- used to model counts
  - mean =  $\lambda$
  - variance =  $\lambda$
  - PMF is defined as
$$P(X = x; \lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$$
where  $X = 0, 1, 2, \dots \infty$
- modeling uses for Poisson distribution
  - count data
  - event-time/survival  $\rightarrow$  cancer trials, some patients never develop and some do, dealing with the data for both (“censoring”)
  - contingency tables  $\rightarrow$  record results for different characteristic measurements
  - approximating binomials  $\rightarrow$  instances where  $n$  is large and  $p$  is small (i.e. pollution on lung disease)
    - \*  $X \sim \text{Binomial}(n, p)$
    - \*  $\lambda = np$
  - rates  $\rightarrow X \sim \text{Poisson}(\lambda t)$ 
    - \*  $\lambda = E[X/t]$   $\rightarrow$  expected count per unit of time
    - \*  $t$  = total monitoring time
- `ppois(n, lambda = lambda*t)` = returns probability of  $n$  or fewer events happening given the rate  $\lambda$  and time  $t$

### Example

- number of people that show up at a bus stop can be modeled with Poisson distribution with a mean of 2.5 per hour
- after watching the bus stop for 4 hours, what is the probability that 3 or fewer people show up for the whole time?

```
# calculate using distribution
ppois(3, lambda = 2.5 * 4)
```

```
## [1] 0.01033605
```

- as we can see from above, there is a 1.0336051% chance for 3 or fewer people show up total at the bus stop during 4 hours of monitoring

### Example - Approximating Binomial Distribution

- flip a coin with success probability of 0.01 a total 500 times (low  $p$ , large  $n$ )
- what's the probability of 2 or fewer successes?

```
# calculate correct probability from Binomial distribution
pbisom(2, size = 500, prob = .01)
```

```
## [1] 0.1233858
```

```
# estimate probability using Poisson distribution  
ppois(2, lambda=500 * .01)
```

```
## [1] 0.124652
```

- as we can see from above, the two probabilities (12.3385774% vs 12.3385774%) are extremely close

## Asymptotics

- **asymptotics** = behavior of statistics as sample size  $\rightarrow \infty$
- useful for simple statistical inference/approximations
- form basis for frequentist interpretation of probabilities (“Law of Large Numbers”)

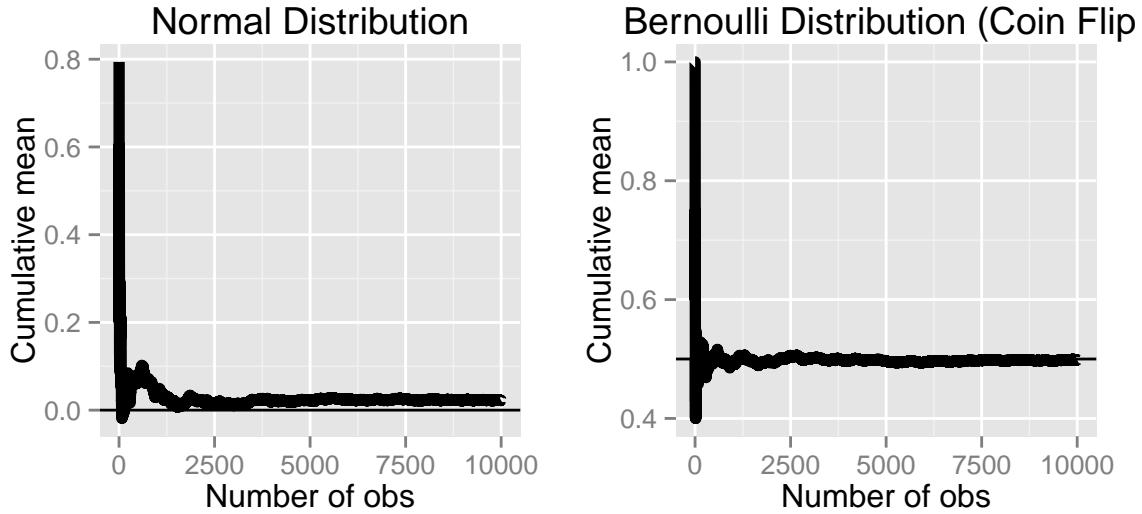
## Law of Large Numbers (LLN)

- IID sample statistic that estimates property of the sample (i.e. mean, variance) *becomes* the population statistic (i.e. population mean, population variance) as  $n$  increases
- *Note: an estimator is consistent if it converges to what it is estimating*
- sample mean/variance/standard deviation are all **consistent estimators** for their population counterparts
  - $\bar{X}_n$  is average of the result of  $n$  coin flips (i.e. the sample proportion of heads)
  - as we flip a fair coin over and over, it *eventually converges* to the true probability of a head

## Example - LLN for Normal and Bernoulli Distribution

- for this example, we will simulate 10000 samples from the normal and Bernoulli distributions respectively
- we will plot the distribution of sample means as  $n$  increases and compare it to the population means

```
# load library
library(gridExtra)
# specify number of trials
n <- 10000
# calculate sample (from normal distribution) means for different size of n
means <- cumsum(rnorm(n)) / (1 : n)
# plot sample size vs sample mean
g <- ggplot(data.frame(x = 1 : n, y = means), aes(x = x, y = y))
g <- g + geom_hline(yintercept = 0) + geom_line(size = 2)
g <- g + labs(x = "Number of obs", y = "Cumulative mean")
g <- g + ggtitle("Normal Distribution")
# calculate sample (coin flips) means for different size of n
means <- cumsum(sample(0 : 1, n, replace = TRUE)) / (1 : n)
# plot sample size vs sample mean
p <- ggplot(data.frame(x = 1 : n, y = means), aes(x = x, y = y))
p <- p + geom_hline(yintercept = 0.5) + geom_line(size = 2)
p <- p + labs(x = "Number of obs", y = "Cumulative mean")
p <- p + ggtitle("Bernoulli Distribution (Coin Flip)")
# combine plots
grid.arrange(g, p, ncol = 2)
```



- as we can see from above, for both distributions the sample means undeniably approach the respective population means as  $n$  increases

### Central Limit Theorem

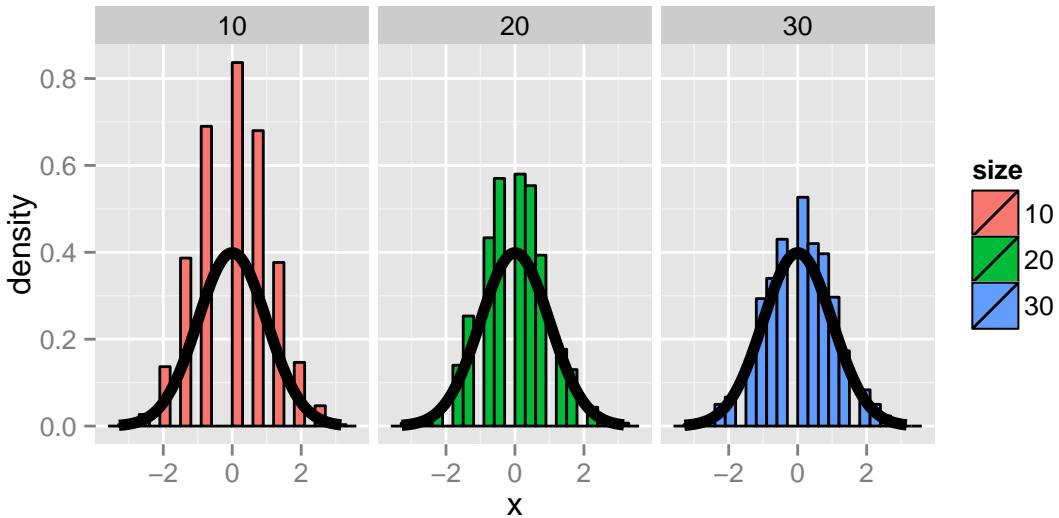
- one of the most important theorems in statistics
- distribution of means of IID variables approaches the standard normal as sample size  $n$  increases
- in other words, for large values of  $n$ ,

$$\frac{\text{Estimate} - \text{Mean of Estimate}}{\text{Std. Err. of Estimate}} = \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} = \frac{\sqrt{n}(\bar{X}_n - \mu)}{\sigma} \rightarrow N(0, 1)$$

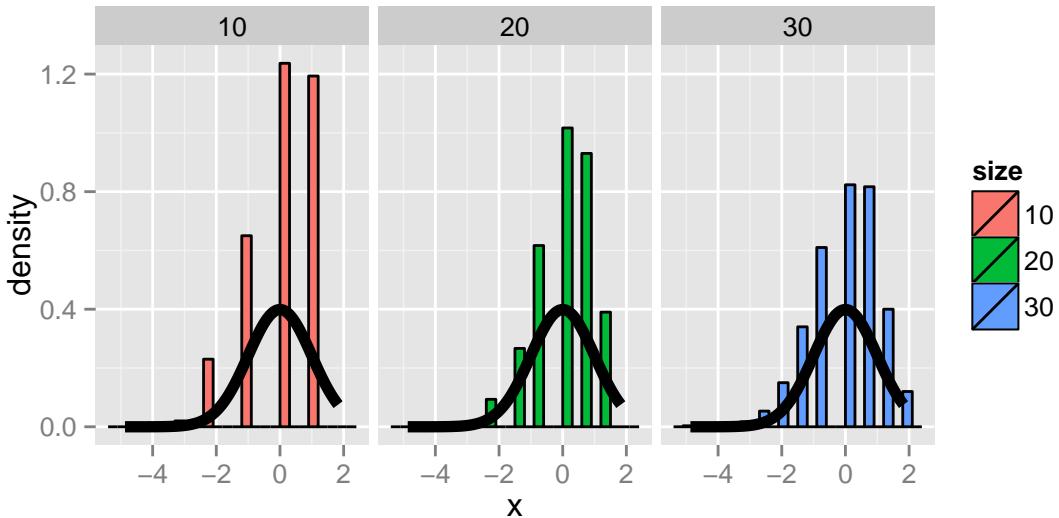
- this translates to the distribution of the sample mean  $\bar{X}_n$  is approximately  $N(\mu, \sigma^2/n)$ 
  - distribution is centered at the population mean
  - with standard deviation = standard error of the mean
- typically the Central Limit Theorem can be applied when  $n \geq 30$

### Example - CLT with Bernoulli Trials (Coin Flips)

- for this example, we will simulate  $n$  flips of a possibly unfair coin
  - $X_i$  be the 0 or 1 result of the  $i^{th}$  flip of a possibly unfair coin
  - sample proportion,  $\hat{p}$ , is the average of the coin flips
  - $E[X_i] = p$  and  $Var(X_i) = p(1-p)$
  - standard error of the mean is  $SE = \sqrt{p(1-p)/n}$
- in principle, normalizing the random variable  $X_i$ , we should get an approximately standard normal distribution
 
$$\frac{\hat{p} - p}{\sqrt{p(1-p)/n}} \sim N(0, 1)$$
- therefore, we will flip a coin  $n$  times, take the sample proportion of heads (successes with probability  $p$ ), subtract off 0.5 (ideal sample proportion) and multiply the result by divide by  $\frac{1}{2\sqrt{n}}$  and compare it to the standard normal



- now, we can run the same simulation trials for an extremely unfair coin with  $p = 0.9$



- as we can see from both simulations, the converted/standardized distribution of the samples convert to the standard normal distribution
- **Note:** speed at which the normalized coin flips converge to normal distribution depends on how biased the coin is (value of  $p$ )
- **Note:** does not guarantee that the normal distribution will be a good approximation, but just that eventually it will be a good approximation as  $n \rightarrow \infty$

### Confidence Intervals - Normal Distribution/Z Intervals

- **Z confidence interval** is defined as

$$Estimate \pm ZQ \times SE_{Estimate}$$

- where  $ZQ$  = quantile from the standard normal distribution
- according to CLT, the sample mean,  $\bar{X}$ , is approximately normal with mean  $\mu$  and sd  $\sigma/\sqrt{n}$

- 95% confidence interval for the population mean  $\mu$  is defined as

$$\bar{X} \pm 2\sigma/\sqrt{n}$$

for the sample mean  $\bar{X} \sim N(\mu, \sigma^2/n)$

- you can choose to use 1.96 to be more accurate for the confidence interval
- $P(\bar{X} > \mu + 2\sigma/\sqrt{n} \text{ or } \bar{X} < \mu - 2\sigma/\sqrt{n}) = 5\%$
- **interpretation:** if we were to repeated samples of size  $n$  from the population and construct this confidence interval for each case, approximately 95% of the intervals will contain  $\mu$
- confidence intervals get **narrower** with less variability or larger sample sizes
- **Note:** Poisson and binomial distributions have exact intervals that don't require CLT
- **example**
  - for this example, we will compute the 95% confidence interval for sons height data in inches

```
# load son height data
data(father.son); x <- father.son$sheight
# calculate confidence interval for sons height in inches
mean(x) + c(-1, 1) * qnorm(0.975) * sd(x)/sqrt(length(x))
```

```
## [1] 68.51605 68.85209
```

### Confidence Interval - Bernoulli Distribution/Wald Interval

- for Bernoulli distributions,  $X_i$  is 0 or 1 with success probability  $p$  and the variance is  $\sigma^2 = p(1-p)$
- the confidence interval takes the form of

$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{p(1-p)}{n}}$$

- since the population proportion  $p$  is unknown, we can use  $\hat{p} = X/n$  as estimate
- $p(1-p)$  is largest when  $p = 1/2$ , so 95% confidence interval can be calculated by

$$\begin{aligned} \hat{p} \pm Z_{0.95} \sqrt{\frac{0.5(1-0.5)}{n}} &= \hat{p} \pm 1.96 \sqrt{\frac{1}{4n}} \\ &= \hat{p} \pm \frac{1.96}{2} \sqrt{\frac{1}{n}} \\ &\approx \hat{p} \pm \frac{1}{\sqrt{n}} \end{aligned}$$

- this is known as the **Wald Confidence Interval** and is useful in *roughly estimating* confidence intervals
- generally need  $n = 100$  for 1 decimal place, 10,000 for 2, and 1,000,000 for 3

- **example**

- suppose a random sample of 100 likely voters, 56 intent to vote for you, can you secure a victory?
- we can use the Wald interval to quickly estimate the 95% confidence interval
- as we can see below, because the interval [0.46, 0.66] contains values below 50%, victory is not guaranteed
- `binom.test(k, n)$conf` = returns confidence interval binomial distribution (collection of Bernoulli trial) with `k` successes in `n` draws

```

# define sample probability and size
p = 0.56; n = 100
# Wald interval
c("WaldInterval" = p + c(-1, 1) * 1/sqrt(n))

## WaldInterval1 WaldInterval2
##          0.46          0.66

# 95% confidence interval
c("95CI" = p + c(-1, 1) * qnorm(.975) * sqrt(p * (1-p)/n))

##      95CI1      95CI2
## 0.4627099 0.6572901

# perform binomial test
binom.test(p*100, n*100)$conf.int

## [1] 0.004232871 0.007265981
## attr(,"conf.level")
## [1] 0.95

```

### Confidence Interval - Binomial Distribution/Agresti-Coull Interval

- for a binomial distribution with smaller values of  $n$  (when  $n < 30$ , thus not large enough for CLT), often time the normal confidence intervals, as defined by

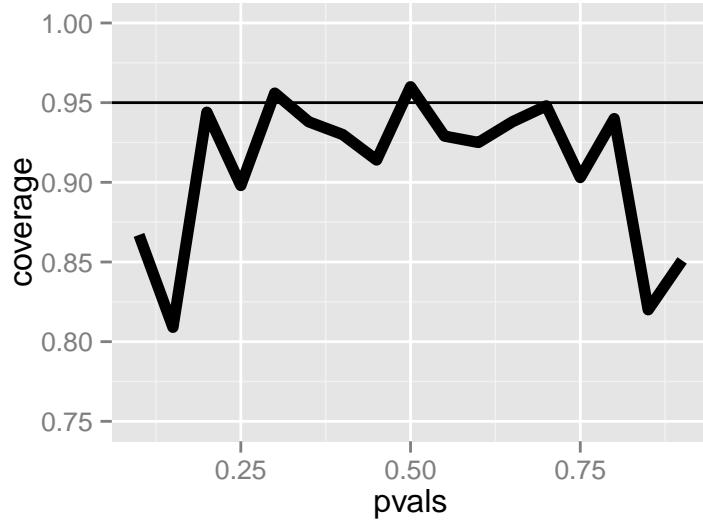
$$\hat{p} \pm z_{1-\alpha/2} \sqrt{\frac{p(1-p)}{n}}$$

do not provide accurate estimates

```

# simulate 1000 samples of size 20 each
n <- 20; nosim <- 1000
# simulate for p values from 0.1 to 0.9
pvals <- seq(.1, .9, by = .05)
# calculate the confidence intervals
coverage <- sapply(pvals, function(p){
  # simulate binomial data
  phats <- rbinom(nosim, prob = p, size = n) / n
  # calculate lower 95% CI bound
  ll <- phats - qnorm(.975) * sqrt(phats * (1 - phats) / n)
  # calculate upper 95% CI bound
  ul <- phats + qnorm(.975) * sqrt(phats * (1 - phats) / n)
  # calculate percent of intervals that contain p
  mean(ll < p & ul > p)
})
# plot CI results vs 95%
ggplot(data.frame(pvals, coverage), aes(x = pvals, y = coverage)) + geom_line(size = 2) + geom_hline(yintercept = 0.95)

```



- as we can see from above, the interval do not provide adequate coverage as 95% confidence intervals (frequently only provide 80 to 90% coverage)
- we can construct the **Agresti-Coull Interval**, which is defined uses the adjustment

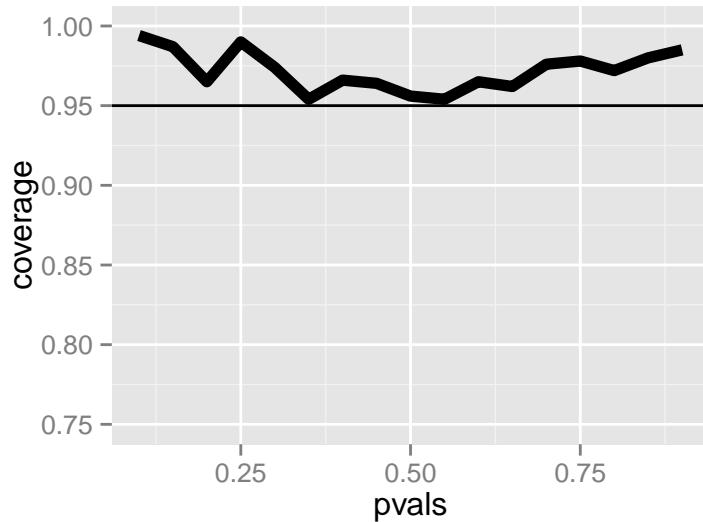
$$\hat{p} = \frac{X + 2}{n + 4}$$

- where we effectively **add 2** to number of successes,  $X$ , and **add 2** to number of failure  
therefore the interval becomes

$$\frac{X + 2}{n + 4} \pm z_{1-\alpha/2} \sqrt{\frac{p(1-p)}{n}}$$

- Note:** interval tend to be *conservative*
- example**

```
# simulate 1000 samples of size 20 each
n <- 20; nosim <- 1000
# simulate for p values from 0.1 to 0.9
pvals <- seq(.1, .9, by = .05)
# calculate the confidence intervals
coverage <- sapply(pvals, function(p){
  # simulate binomial data with Agresti/Coull Interval adjustment
  phats <- (rbinom(nosim, prob = p, size = n) + 2) / (n + 4)
  # calculate lower 95% CI bound
  ll <- phats - qnorm(.975) * sqrt(phats * (1 - phats) / n)
  # calculate upper 95% CI bound
  ul <- phats + qnorm(.975) * sqrt(phats * (1 - phats) / n)
  # calculate percent of intervals that contain p
  mean(ll < p & ul > p)
})
# plot CI results vs 95%
ggplot(data.frame(pvals, coverage), aes(x = pvals, y = coverage)) + geom_line(size = 2) + geom_hline(yintercept = 0.95)
```



- as we can see from above, the coverage is much better for the 95% interval
- in fact, all of the estimates are more conservative as we previously discussed, indicating the Agresti-Coull intervals are **wider** than the regular confidence intervals

### Confidence Interval - Poisson Interval

- for  $X \sim Poisson(\lambda t)$ 
  - estimate rate  $\hat{\lambda} = X/t$
  - $var(\hat{\lambda}) = \lambda/t$
  - variance estimate =  $\hat{\lambda}/t$
- so the confidence interval is defined as
$$\hat{\lambda} \pm z_{1-\alpha/2} \sqrt{\frac{\lambda}{t}}$$
  - however, for small values of  $\lambda$  (few events larger time interval), we **should not** use the asymptotic interval estimated
  - **example**
    - \* for this example, we will go through a specific scenario as well as a simulation exercise to demonstrate the ineffectiveness of asymptotic intervals for small values of  $\lambda$
    - \* nuclear pump failed 5 times out of 94.32 days, give a 95% confidence interval for the failure rate per day?
    - \* `poisson.test(x, T)$conf` = returns Poisson 95% confidence interval for given x occurrence over T time period

```
# define parameters
x <- 5; t <- 94.32; lambda <- x / t
# calculate confidence interval
round(lambda + c(-1, 1) * qnorm(.975) * sqrt(lambda / t), 3)
```

```
## [1] 0.007 0.099
```

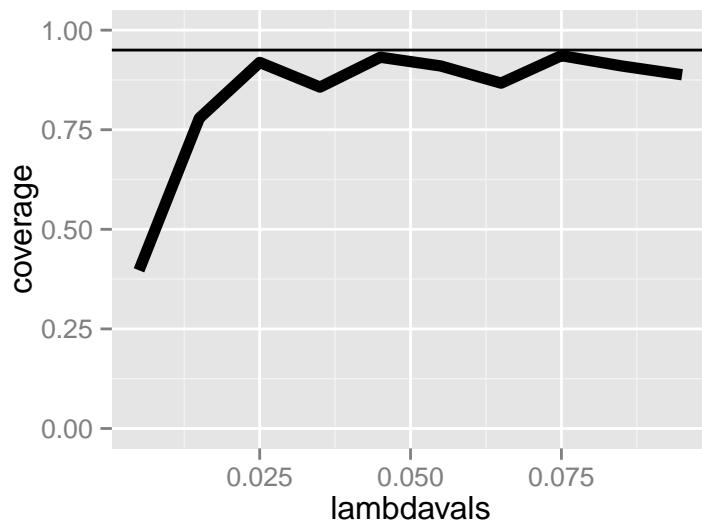
```
# return accurate confidence interval from poisson.test
poisson.test(x, T = 94.32)$conf
```

```

## [1] 0.01721254 0.12371005
## attr(),"conf.level")
## [1] 0.95

# small lambda simulations
lambdavals <- seq(0.005, 0.10, by = .01); nosim <- 1000; t <- 100
# calculate coverage using Poisson intervals
coverage <- sapply(lambdavals, function(lambda){
  # calculate Poisson rates
  lhats <- rpois(nosim, lambda = lambda * t) / t
  # lower bound of 95% CI
  ll <- lhats - qnorm(.975) * sqrt(lhats / t)
  # upper bound of 95% CI
  ul <- lhats + qnorm(.975) * sqrt(lhats / t)
  # calculate percent of intervals that contain lambda
  mean(ll < lambda & ul > lambda)
})
# plot CI results vs 95%
ggplot(data.frame(lambdavals, coverage), aes(x = lambdavals, y = coverage)) + geom_line(size = 2) + geom_vline(xintercept = 0.95)

```



- as we can see above, for small values of  $\lambda = X/t$ , the confidence interval produced by the asymptotic interval is **not** an accurate estimate of the actual 95% interval (not enough coverage)
- however, as  $t \rightarrow \infty$ , the interval becomes the **true 95% interval**

```

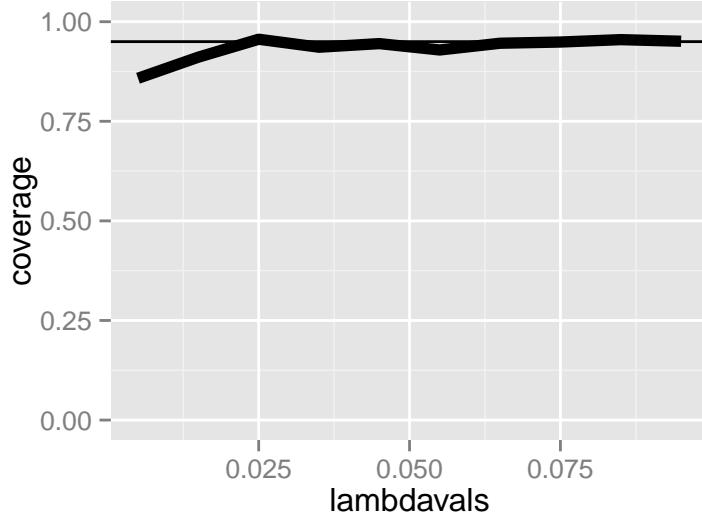
# small lambda simulations
lambdavals <- seq(0.005, 0.10, by = .01); nosim <- 1000; t <- 1000
# calculate coverage using Poisson intervals
coverage <- sapply(lambdavals, function(lambda){
  # calculate Poisson rates
  lhats <- rpois(nosim, lambda = lambda * t) / t
  # lower bound of 95% CI
  ll <- lhats - qnorm(.975) * sqrt(lhats / t)
  # upper bound of 95% CI
  ul <- lhats + qnorm(.975) * sqrt(lhats / t)
  # calculate percent of intervals that contain lambda
  mean(ll < lambda & ul > lambda)
})
# plot CI results vs 95%
ggplot(data.frame(lambdavals, coverage), aes(x = lambdavals, y = coverage)) + geom_line(size = 2) + geom_vline(xintercept = 0.95)

```

```

    mean(l1 < lambda & ul > lambda)
})
# plot CI results vs 95%
ggplot(data.frame(lambdavals, coverage), aes(x = lambdavals, y = coverage)) + geom_line(size = 2) + geom_

```



- as we can see from above, as  $t$  increases, the Poisson intervals become closer to the actual 95% confidence intervals

### Confidence Intervals - T Distribution(Small Samples)

- $t$  confidence interval is defined as

$$Estimate \pm TQ \times SE_{Estimate} = \bar{X} \pm \frac{t_{n-1}S}{\sqrt{n}}$$

- $TQ$  = quantile from T distribution
- $t_{n-1}$  = relevant quantile
- $t$  interval assumes data is IID normal so that

$$\frac{\bar{X} - \mu}{S/\sqrt{n}}$$

follows Gosset's  $t$  distribution with  $n - 1$  degrees of freedom

- works well with data distributions that are roughly symmetric/mound shaped, and **does not** work with skewed distributions
  - skewed distribution  $\rightarrow$  meaningless to center interval around the mean  $\bar{X}$
  - logs/median can be used instead
- paired observations (multiple measurements from same subjects) can be analyzed by t interval of differences
- as more data collected (large degrees of freedom), t interval  $\rightarrow$  z interval
- `qt(0.975, df=n-1)` = calculate the relevant quantile using t distribution

```

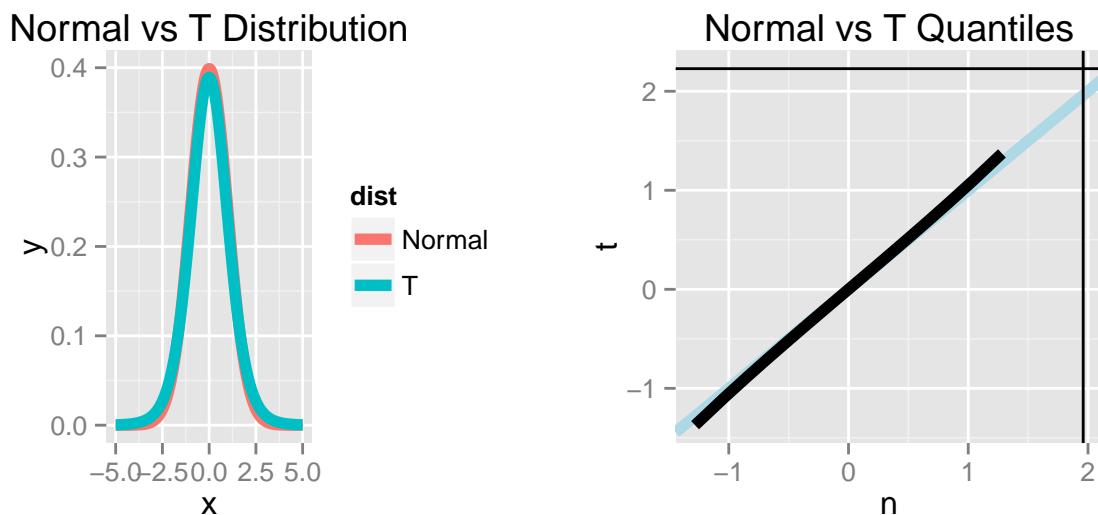
# Plot normal vs t distributions
k <- 1000; xvals <- seq(-5, 5, length = k); df <- 10
d <- data.frame(y = c(dnorm(xvals), dt(xvals, df)), x = xvals,

```

```

    dist = factor(rep(c("Normal", "T"), c(k,k)))
g <- ggplot(d, aes(x = x, y = y))
g <- g + geom_line(size = 2, aes(colour = dist)) + ggtitle("Normal vs T Distribution")
# plot normal vs t quantiles
d <- data.frame(n= qnorm(pvals),t=qt(pvals, df),p = pvals)
h <- ggplot(d, aes(x= n, y = t))
h <- h + geom_abline(size = 2, col = "lightblue")
h <- h + geom_line(size = 2, col = "black")
h <- h + geom_vline(xintercept = qnorm(0.975))
h <- h + geom_hline(yintercept = qt(0.975, df)) + ggtitle("Normal vs T Quantiles")
# plot 2 graphs together
grid.arrange(g, h, ncol = 2)

```



- William Gosset's **t** Distribution ("Student's T distribution")
  - test = Gosset's pseudonym which he published under
  - indexed/defined by **degrees of freedom**, and becomes more like standard normal as degrees of freedom gets larger
  - thicker tails centered around 0, thus confidence interval = **wider** than Z interval (more mass concentrated away from the center)
  - for **small** sample size (value of  $n$ ), normalizing the distribution by  $\frac{\bar{X}-\mu}{S/\sqrt{n}} \rightarrow t$  distribution, **not** the standard normal distribution
    - \*  $S$  = standard deviation may be inaccurate, as the std of the data sample may not be truly representative of the population std
    - \* using the Z interval here thus may produce an interval that is too **narrow**

### Confidence Interval - Paired T Tests

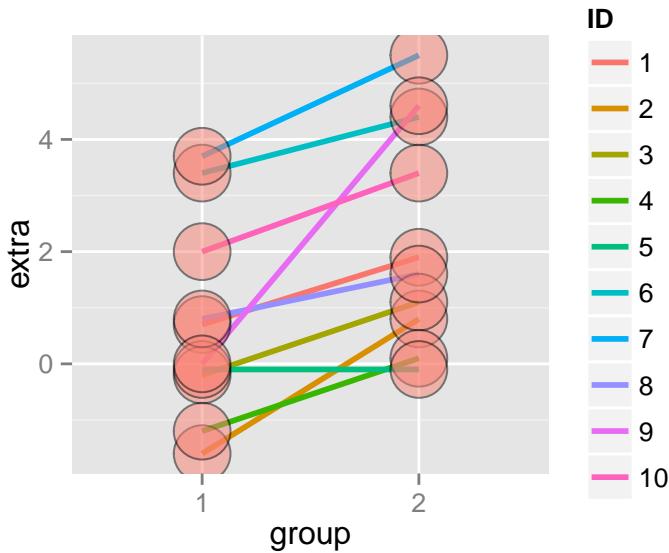
- compare observations for the same subjects over two different sets of data (i.e. different times, different treatments)
- the confidence interval is defined by

$$\bar{X}_1 - \bar{X}_2 \pm \frac{t_{n-1}S}{\sqrt{n}}$$

where  $\bar{X}_1$  represents the first observations and  $\bar{X}_2$  the second set of observations

- `t.test(difference)` = performs group mean t test and returns metrics as results, which includes the confidence intervals
  - `t.test(g2, g1, paired = TRUE)` = performs the same paired t test with data directly
- *example*
  - the data used here is for a study of the effects of two soporific drugs (increase in hours of sleep compared to control) on 10 patients

```
# load data
data(sleep)
# plot the first and second observations
g <- ggplot(sleep, aes(x = group, y = extra, group = factor(ID)))
g <- g + geom_line(size = 1, aes(colour = ID)) + geom_point(size = 10, pch = 21, fill = "salmon", alpha = 0.5)
g
```



```
# define groups
g1 <- sleep$extra[1 : 10]; g2 <- sleep$extra[11 : 20]
# define difference
difference <- g2 - g1
# calculate mean and sd of differences
mn <- mean(difference); s <- sd(difference); n <- 10
# calculate intervals manually
mn + c(-1, 1) * qt(.975, n-1) * s / sqrt(n)
```

```
## [1] 0.7001142 2.4598858
```

```
# perform the same test to get confidence intervals
t.test(difference)
```

```
##
##  One Sample t-test
##
## data: difference
```

```

## t = 4.0621, df = 9, p-value = 0.002833
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
## 0.7001142 2.4598858
## sample estimates:
## mean of x
## 1.58

t.test(g2, g1, paired = TRUE)

##
## Paired t-test
##
## data: g2 and g1
## t = 4.0621, df = 9, p-value = 0.002833
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## 0.7001142 2.4598858
## sample estimates:
## mean of the differences
## 1.58

```

### Independent Group t Intervals - Same Variance

- compare two groups in randomized trial (“A/B Testing”)
- cannot use the paired t test because the groups are independent and may have different sample sizes
- perform randomization to balance unobserved covariance that may otherwise affect the result
- $t$  confidence interval for  $\mu_y - \mu_x$  is defined as

$$\bar{Y} - \bar{X} \pm t_{n_x+n_y-2,1-\alpha/2} S_p \left( \frac{1}{n_x} + \frac{1}{n_y} \right)^{1/2}$$

- $t_{n_x+n_y-2,1-\alpha/2}$  = relevant quantile
- $n_x + n_y - 2$  = degrees of freedom
- $S_p \left( \frac{1}{n_x} + \frac{1}{n_y} \right)^{1/2}$  = standard error
- $S_p^2 = \{(n_x - 1)S_x^2 + (n_y - 1)S_y^2\}/(n_x + n_y - 2)$  = pooled variance estimator
  - \* this is effectively a weighted average between the two variances, such that different sample sizes are taken into account
- **Note:** this interval assumes **constant variance** across two groups; if variance is different, use the next interval

### Independent Group t Intervals - Different Variance

- confidence interval for  $\mu_y - \mu_x$  is defined as

$$\bar{Y} - \bar{X} \pm t_{df} \times \left( \frac{s_x^2}{n_x} + \frac{s_y^2}{n_y} \right)^{1/2}$$

- $t_{df}$  = relevant quantile with df as defined below
- **Note:** normalized statistic does not follow  $t$  distribution but can be approximated through the formula with df defined below

\*

$$df = \frac{(S_x^2/n_x + S_y^2/n_y)^2}{\left(\frac{S_x^2}{n_x}\right)^2/(n_x - 1) + \left(\frac{S_y^2}{n_y}\right)^2/(n_y - 1)}$$

\*  $\left(\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y}\right)^{1/2}$  = standard error

- Comparing other kinds of data

- binomial  $\rightarrow$  relative risk, risk difference, odds ratio
  - binomial  $\rightarrow$  Chi-squared test, normal approximations, exact tests
  - count  $\rightarrow$  Chi-squared test, exact tests

- R commands

- t Confidence Intervals

\* `mean + c(-1, 1) * qt(0.975, n - 1) * std / sqrt(n)`  
· *c(-1, 1)* = plus and minus,  $\pm$

- Difference Intervals (all equivalent)

\* `mean2 - mean1 + c(-1, 1) * qt(0.975, n - 1) * std / sqrt(n)`  
· *n* = number of paired observations  
· *qt(0.975, n - 1)* = relevant quantile for paired  
· *qt(0.975, n<sub>x</sub> + n<sub>y</sub> - 2)* = relevant quantile for independent  
\* `t.test(mean2 - mean1)`  
\* `t.test(data2, data1, paired = TRUE, var.equal = TRUE)`  
· *paired* = whether or not the two sets of data are paired (same subjects different observations for treatment)  $\leftarrow$  TRUE for paired, FALSE for independent  
· *var.equal* = whether or not the variance of the datasets should be treated as equal  $\leftarrow$  TRUE for same variance, FALSE for unequal variances  
\* `t.test(extra ~ I(relevel(group, 2)), paired = TRUE, data = sleep)`  
· *relevel(factor, ref)* = reorders the levels in the factor so that “ref” is changed to the first level  $\rightarrow$  doing this here is so that the second set of measurements come first (1, 2  $\rightarrow$  2, 1) in order to perform `mean2 - mean1`  
· *I(object)* = prepend the class “AsIs” to the object  
· *Note: I(relevel(group, 2)) = explanatory variable, must be factor and have two levels*

## Hypothesis Testing

- Hypothesis testing = making decisions using data
  - **null hypothesis ( $H_0$ )** = status quo
  - assumed to be **true**  $\rightarrow$  statistical evidence required to reject it for **alternative** or “research” hypothesis ( $H_a$ )
    - \* alternative hypothesis typically take form of  $>$ ,  $<$  or  $\neq$
  - Results

Truth	Decide	Result
$H_0$	$H_0$	Correctly accept null
$H_0$	$H_a$	Type I error
$H_a$	$H_a$	Correctly reject null
$H_a$	$H_0$	Type II error

- $\alpha$  = Type I error rate
  - probability of **rejecting** the null hypothesis when the hypothesis is **correct**
  - $\alpha = 0.05 \rightarrow$  standard for hypothesis testing
  - **Note:** as Type I error rate increases, Type II error rate decreases and vice versa
- for large samples (large n), use the **Z Test** for  $H_0 : \mu = \mu_0$ 
  - $H_a$ :
    - \*  $H_1 : \mu < \mu_0$
    - \*  $H_2 : \mu \neq \mu_0$
    - \*  $H_3 : \mu > \mu_0$
  - Test statistic  $TS = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$
  - Reject the null hypothesis  $H_0$  when
    - \*  $H_1 : TS \leq Z_{\alpha}$  OR  $-Z_{1-\alpha}$
    - \*  $H_2 : |TS| \geq Z_{1-\alpha/2}$
    - \*  $H_3 : TS \geq Z_{1-\alpha}$
  - **Note:** In case of  $\alpha = 0.05$  (most common),  $Z_{1-\alpha} = 1.645$  (95 percentile)
  - $\alpha$  = low, so that when  $H_0$  is rejected, original model  $\rightarrow$  wrong or made an error (low probability)
- For small samples (small n), use the **T Test** for  $H_0 : \mu = \mu_0$ 
  - $H_a$ :
    - \*  $H_1 : \mu < \mu_0$
    - \*  $H_2 : \mu \neq \mu_0$
    - \*  $H_3 : \mu > \mu_0$
  - Test statistic  $TS = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$
  - Reject the null hypothesis  $H_0$  when
    - \*  $H_1 : TS \leq T_{\alpha}$  OR  $-T_{1-\alpha}$
    - \*  $H_2 : |TS| \geq T_{1-\alpha/2}$
    - \*  $H_3 : TS \geq T_{1-\alpha}$
  - **Note:** In case of  $\alpha = 0.05$  (most common),  $T_{1-\alpha} = qt(.95, df = n-1)$
  - R commands for T test:

- \* `t.test(vector1 - vector2)`
- \* `t.test(vector1, vector2, paired = TRUE)`
  - alternative argument can be used to specify one-sided tests: `less` or `greater`
  - alternative default = `two-sided`
- \* prints test statistic (`t`), degrees of freedom (`df`), `p-value`, 95% confidence interval, and mean of sample
  - confidence interval in units of data, and can be used to interpret the practical significance of the results
- **rejection region** = region of TS values for which you reject  $H_0$
- **power** = probability of rejecting  $H_0$ 
  - power is used to calculate sample size for experiments
- **two-sided tests**  $\rightarrow H_a : \mu \neq \mu_0$ 
  - reject  $H_0$  only if test statistic is too large/small
  - for  $\alpha = 0.5$ , split equally to 2.5% for upper and 2.5% for lower tails
    - \* equivalent to  $|TS| \geq T_{1-\alpha/2}$
    - \* example: for T test, `qt(.975, df)` and `qt(.025, df)`
  - **Note:** failing to reject one-sided test = fail to reject two-sided
- **tests vs confidence intervals**
  - $(1 - \alpha)\%$  confidence interval for  $\mu$  = set of all possible values that fail to reject  $H_0$
  - if  $(1 - \alpha)\%$  confidence interval contains  $\mu_0$ , fail to reject  $H_0$
- **two-group intervals/test**
  - Rejection rules the same
  - Test  $H_0: \mu_1 = \mu_2 \rightarrow \mu_1 - \mu_2 = 0$
  - Test statistic:
 
$$\frac{\text{Estimate} - H_0 \text{Value}}{SE_{\text{Estimate}}} = \frac{\bar{X}_1 - \bar{X}_2 - 0}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$
  - R Command
    - \* `t.test(values ~ factor, paired = FALSE, var.equal = TRUE, data = data)`
      - `paired = FALSE`  $\rightarrow$  independent values
      - `factor` argument must have only two levels
- **p values**
  - most common measure of statistical significance
  - **p-value** = probability under the null hypothesis of obtaining evidence as extreme or more than that of the obtained
    - \* Given that  $H_0$  is true, how likely is it to obtain the result (test statistic)?
  - **attained significance level** = smallest value for  $\alpha$  for which  $H_0$  is rejected  $\rightarrow$  equivalent to p-value
    - \* if p-value  $< \alpha$ , reject  $H_0$
    - \* for two-sided tests, double the p-values
  - if p-value is small, either  $H_0$  is true AND the observed is a rare event **OR**  $H_0$  is false
  - R Command
    - \* `p-value = pt(statistic, df, lower.tail = FALSE)`
      - `lower.tail = FALSE` = returns the probability of getting a value from the t distribution that is larger than the test statistic

- \* Binomial (coin flips)
  - probability of getting x results out of n trials and event probability of p = `pbinom(x, size = n, prob = p, lower.tail = FALSE)`
  - two-sided interval (testing for  $\neq$ ): find the smaller of two one-sided intervals ( $X < \text{value}$ ,  $X > \text{value}$ ), and double the result
  - *Note: lower.tail = FALSE = strictly greater*
- \* Poisson
  - probability of getting x results given the rate r = `ppois(x - 1, r, lower.tail = FALSE)`
  - $x - 1$  is used here because the upper tail includes the specified number (since we want greater than x, we start at x - 1)
  - r = events that should occur given the rate (multiplied by 100 to yield an integer)
  - *Note: lower.tail = FALSE = strictly greater*

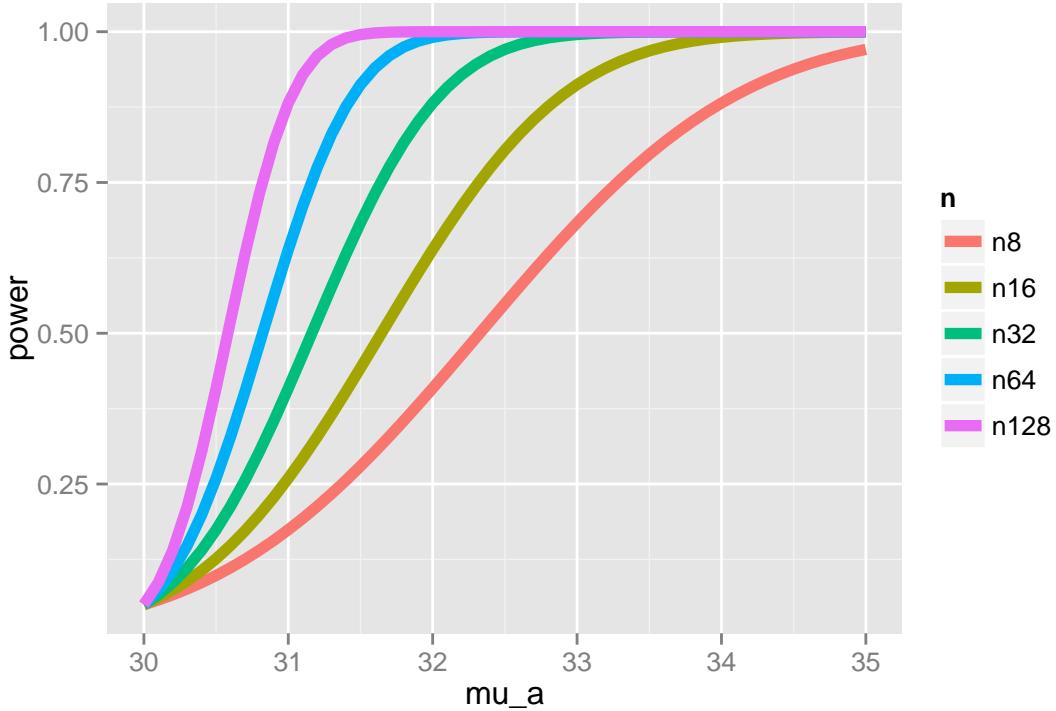
## Power

- **Power** = probability of rejecting the null hypothesis when it is false (the more power the better)
  - most often used in designing studies so that there's a reasonable chance to detect the alternative hypothesis if the alternative hypothesis is true
- $\beta$  = probability of type II error = failing to reject the null hypothesis when it's false
- power =  $1 - \beta$
- **example**
  - $H_0 : \mu = 30 \rightarrow \bar{X} \sim N(\mu_0, \sigma^2/n)$
  - $H_a : \mu > 30 \rightarrow \bar{X} \sim N(\mu_a, \sigma^2/n)$
  - Power:
 
$$Power = P\left(\frac{\bar{X} - 30}{s/\sqrt{n}} > t_{1-\alpha, n-1}; \mu = \mu_a\right)$$
    - \* **Note:** the above function depends on value of  $\mu_a$
    - \* **Note:** as  $\mu_a$  approaches 30, power approaches  $\alpha$
  - assuming the sample mean is normally distributed,  $H_0$  is rejected when  $\frac{\bar{X}-30}{\sigma/\sqrt{n}} > Z_{1-\alpha}$
  - or,  $\bar{X} > 30 + Z_{1-\alpha} \frac{\sigma}{\sqrt{n}}$
- R commands:
  - alpha = 0.05; z = qnorm(1-alpha) –> calculates  $Z_{1-\alpha}$
  - pnorm(mu0 + z \* sigma/sqrt(n), mean = mua, sd = sigma/sqrt(n), lower.tail = FALSE) –> calculates the probability of getting a sample mean that is larger than  $Z_{1-\alpha} \frac{\sigma}{\sqrt{n}}$  given that the population mean is  $\mu_a$
  - \* **Note:** using **mean** = mu0 in the function would = alpha
  - Power curve behavior
    - \* Power increases as  $\mu_a$  increases –> we are more likely to detect the difference in  $\mu_a$  and  $\mu_0$
    - \* Power increases as **n** increases –> with more data, more likely to detect any alternative  $\mu_a$

```

library(ggplot2)
mu0 = 30; mua = 32; sigma = 4; n = 16
alpha = 0.05
z = qnorm(1 - alpha)
nseq = c(8, 16, 32, 64, 128)
mu_a = seq(30, 35, by = 0.1)
power = sapply(nseq, function(n)
  pnorm(mu0 + z * sigma / sqrt(n), mean = mu_a, sd = sigma / sqrt(n),
        lower.tail = FALSE)
)
colnames(power) <- paste("n", nseq, sep = "")
d <- data.frame(mu_a, power)
library(reshape2)
d2 <- melt(d, id.vars = "mu_a")
names(d2) <- c("mu_a", "n", "power")
g <- ggplot(d2,
            aes(x = mu_a, y = power, col = n)) + geom_line(size = 2)
g

```



- Solving for Power

- When testing  $H_a : \mu > \mu_0$  (or  $<$  or  $\neq$ )

$$Power = 1 - \beta = P\left(\bar{X} > \mu_0 + Z_{1-\alpha} \frac{\sigma}{\sqrt{n}}; \mu = \mu_a\right)$$

- where  $\bar{X} \sim N(\mu_a, \sigma^2/n)$
- Unknowns =  $\mu_a, \sigma, n, \beta$
- Knowns =  $\mu_0, \alpha$
- Specify any 3 of the unknowns and you can solve for the remainder; most common are two cases
  1. Given power desired, mean to detect, variance that we can tolerate, find the **n** to produce desired power (designing experiment/trial)
  2. Given the size **n** of the sample, find the power that is achievable (finding the utility of experiment)
- **Note:** for  $H_a : \mu \neq \mu_0$ , calculated one-sided power using  $z_{1-\alpha/2}$ ; however, the power calculation here excludes the probability of getting a large TS in the opposite direction of the truth, but this is only applicable when  $\mu_a$  and  $\mu_0$  are close together

- Power Behavior

- Power increases as  $\alpha$  becomes larger
- Power of one-sided test > power of associated two-sided test
- Power increases as  $\mu_a$  gets further away from  $\mu_0$
- Power increases as **n** increases (sample mean has less variability)
- Power increases as  $\sigma$  decreases (again less variability)
- Power usually depends only  $\frac{\sqrt{n}(\mu_a - \mu_0)}{\sigma}$ , and not  $\mu_a, \sigma$ , and **n**
  - \* **effect size** =  $\frac{\mu_a - \mu_0}{\sigma} \rightarrow$  unit free, can be interpreted across settings

- T-test Power

- for Gossett's T test,

$$Power = P \left( \frac{\bar{X} - \mu_0}{S/\sqrt{n}} > t_{1-\alpha, n-1}; \mu = \mu_a \right)$$

\*  $\frac{\bar{X} - \mu_0}{S/\sqrt{n}}$  does not follow a t distribution if the true mean is  $\mu_a$  and NOT  $\mu_0$   $\rightarrow$  follows a non-central t distribution instead

- `power.t.test`  $\rightarrow$  evaluates the non-central t distribution and solves for a parameter given all others are specified

\* `power.t.test(n = 16, delta = 0.5, sd = 1, type = "one.sample", alt = "one.sided")$power`  
 $\rightarrow$  calculates power with inputs of n, difference in means, and standard deviation

· `delta` = argument for difference in means

· *Note: since effect size = delta/sd, as n, type, and alt are held constant, any distribution with the same effect size will have the same power*

\* `power.t.test(power = 0.8, delta = 0.5, sd = 1, type = "one.sample", alt = "one.sided")$n`  $\rightarrow$  calculates size n with inputs of power, difference in means, and standard deviation

· *Note: n should always be rounded up (ceiling)*

## Multiple Testing

- Hypothesis testing/significant analysis commonly overused
- correct for multiple testing to avoid false positives/conclusions (two key components)
  1. error measure
  2. correction
- multiple testing is needed because of the increase in ubiquitous data collection technology and analysis
  - DNA sequencing machines
  - imaging patients in clinical studies
  - electronic medical records
  - individualized movement data (fitbit)

## Type of Errors

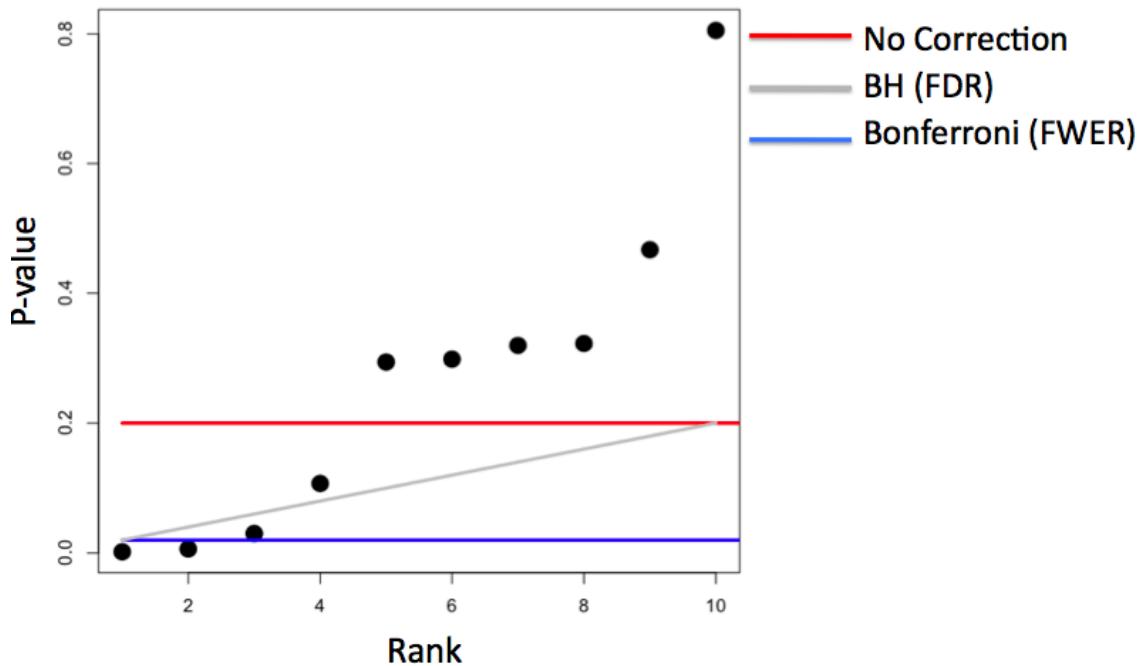
	Actual \$H_0\$ = True   Actual \$H_a\$ = True   Total	
		Conclude \$H_0\$ = True (non-significant)   \$U\$   \$T\$   \$m - R\$ Conclude \$H_a\$ = True (significant)   \$V\$   \$S\$   \$R\$ Total   \$m_0\$   \$m - m_0\$   \$m\$

- $m_0$  = number of true null hypotheses, or cases where  $H_0$  = actually true (unknown)
- $m - m_0$  = number of true alternative hypotheses, or cases where  $H_a$  = actually true (unknown)
- $R$  = number of null hypotheses rejected, or cases where  $H_a$  = concluded to be true (measurable)
- $m - R$  = number of null hypotheses that failed to be rejected, or cases where  $H_0$  = concluded to be true (measurable)
- $V$  = Type I Error / false positives, concludes  $H_a$  = True when  $H_0$  = actually True
- $T$  = Type II Error / false negatives, concludes  $H_0$  = True when  $H_a$  = actually True
- $S$  = true positives, concludes  $H_a$  = True when  $H_a$  = actually True
- $U$  = true negatives, concludes  $H_0$  = True when  $H_0$  = actually True

## Error Rates

- **false positive rate** = rate at which false results are called significant  $E[\frac{V}{m_0}] \rightarrow$  average fraction of times that  $H_a$  is claimed to be true when  $H_0$  is actually true
  - *Note: mathematically equal to type I error rate  $\rightarrow$  false positive rate is associated with a post-prior result, which is the expected number of false positives divided by the total number of hypotheses under the real combination of true and non-true null hypotheses (disregarding the “global null” hypothesis). Since the false positive rate is a parameter that is not controlled by the researcher, it cannot be identified with the significance level, which is what determines the type I error rate.*
- **family wise error rate (FWER)** = probability of at least one false positive  $Pr(V \geq 1)$
- **false discovery rate (FDR)** = rate at which claims of significance are false  $E[\frac{V}{R}]$
- **controlling error rates (adjusting  $\alpha$ )**
  - false positive rate
    - \* if we call all  $P < \alpha$  significant (reject  $H_0$ ), we are expected to get  $\alpha \times m$  false positives, where  $m$  = total number of hypothesis test performed
    - \* with high values of  $m$ , false positive rate is very large as well

- family-wise error rate (FWER)
  - \* controlling FWER = controlling the probability of even one false positive
  - \* *bonferroni* correction (oldest multiple testing correction)
    - for  $m$  tests, we want  $Pr(V \geq 1) < \alpha$
    - calculate P-values normally, and deem them significant if and only if  $P < \alpha_{fewer} = \alpha/m$
  - \* easy to calculate, but tend to be very ***conservative***
- false discovery rate (FDR)
  - \* most popular correction = controlling FDR
  - \* for  $m$  tests, we want  $E[\frac{V}{R}] \leq \alpha$
  - \* calculate P-values normally and sort some from smallest to largest  $\rightarrow P_{(1)}, P_{(2)}, \dots, P_{(m)}$
  - \* deem the P-values significant if  $P_{(i)} \leq \alpha \times \frac{i}{m}$
  - \* easy to calculate, less conservative, but allows for more false positives and may behave strangely under dependence (related hypothesis tests/regression with different variables)
- ***example***
  - \* 10 P-values with  $\alpha = 0.20$



- adjusting for p-values
  - ***Note:*** changing P-values will fundamentally change their properties but they can be used directly without adjusting /alpha
  - *bonferroni* (FWER)
    - \*  $P_i^{fewer} = \max(mP_i, 1)$   $\rightarrow$  since p cannot exceed value of 1
    - \* deem P-values significant if  $P_i^{fewer} < \alpha$
    - \* similar to controlling FWER

## Example

```

set.seed(1010093)
pValues <- rep(NA,1000)
for(i in 1:1000){
  x <- rnorm(20)
  # First 500 beta=0, last 500 beta=2
  if(i <= 500){y <- rnorm(20)}else{ y <- rnorm(20,mean=2*x)}
  # calculating p-values by using linear model; the [2, 4] coeff in result = pvalue
  pValues[i] <- summary(lm(y ~ x))$coeff[2,4]
}
# Controls false positive rate
trueStatus <- rep(c("zero","not zero"),each=500)
table(pValues < 0.05, trueStatus)

##          trueStatus
##          not zero zero
## FALSE      0    476
## TRUE      500     24

# Controls FWER
table(p.adjust(pValues,method="bonferroni") < 0.05,trueStatus)

##          trueStatus
##          not zero zero
## FALSE      23    500
## TRUE      477     0

# Controls FDR (Benjamin Hochberg)
table(p.adjust(pValues,method="BH") < 0.05,trueStatus)

##          trueStatus
##          not zero zero
## FALSE      0    487
## TRUE      500     13

```

## Resample Inference

- **Bootstrap** = useful tool for constructing confidence intervals and calculating standard errors for difficult statistics
  - *principle* = if a statistic's (i.e. median) sampling distribution is unknown, then use distribution defined by the data to approximate it
  - *procedures*
    1. simulate  $n$  observations **with replacement** from the observed data  $\rightarrow$  results in 1 simulated complete data set
    2. calculate desired statistic (i.e. median) for each simulated data set
    3. repeat the above steps  $B$  times, resulting in  $B$  simulated statistics
    4. these statistics are approximately drawn from the sampling distribution of the true statistic of  $n$  observations
    5. perform one of the following
      - \* plot a histogram
      - \* calculate standard deviation of the statistic to estimate its standard error
      - \* take quantiles (2.5<sup>th</sup> and 97.5<sup>th</sup>) as a confidence interval for the statistic ("bootstrap CI")
  - *example*
    - \* Bootstrap procedure for calculating confidence interval for the median from a data set of  $n$  observations  $\rightarrow$  approximate sampling distribution

```
# load data
library(UsingR); data(father.son)
# observed dataset
x <- father.son$sheight
# number of simulated statistic
B <- 1000
# generate samples
resamples <- matrix(
  sample(x, # sample to draw from
         n * B, # draw B datasets with n observations each
         replace = TRUE), # cannot draw n*B elements from x (has n elements) without replacement
  B, n) # arrange results into n x B matrix
# (every row = bootstrap sample with n observations)

# take median for each row/generated sample
medians <- apply(resamples, 1, median)
# estimated standard error of median
sd(medians)

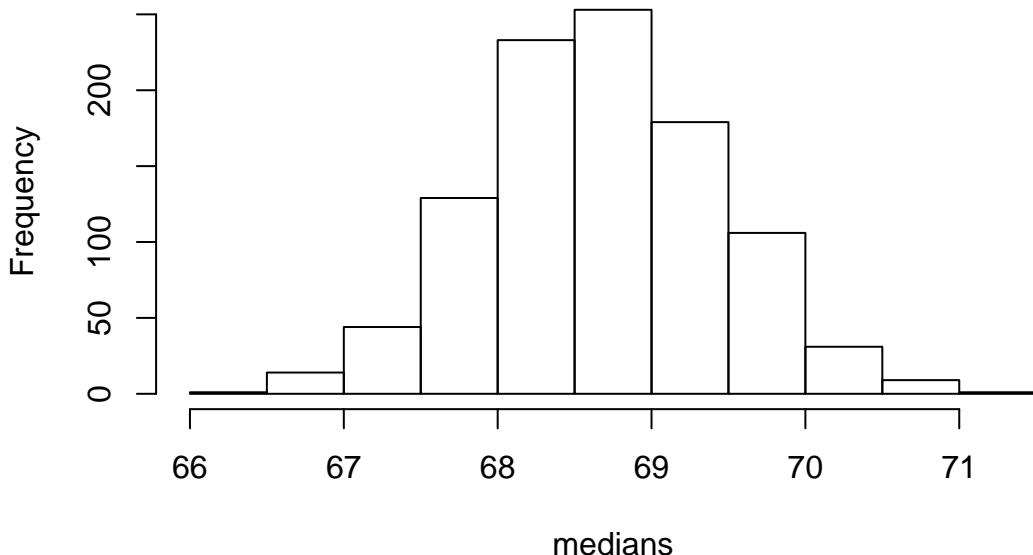
## [1] 0.76595

# confidence interval of median
quantile(medians, c(.025, .975))

##      2.5%    97.5%
## 67.18292 70.16488

# histogram of bootstrapped samples
hist(medians)
```

## Histogram of medians



- *Note:* better percentile bootstrap confidence interval = “bias corrected and accelerated interval” in *bootstrap* package

- Permutation Tests

- *procedures*

- \* compare groups of data and test the null hypothesis that the distribution of the observations from each group = same
      - *Note:* if this is true, then group labels/divisions are irrelevant
    - \* permute the labels for the groups
    - \* recalculate the statistic
      - Mean difference in counts
      - Geometric means
      - T statistic
    - \* Calculate the percentage of simulations where the simulated statistic was more extreme (toward the alternative) than the observed

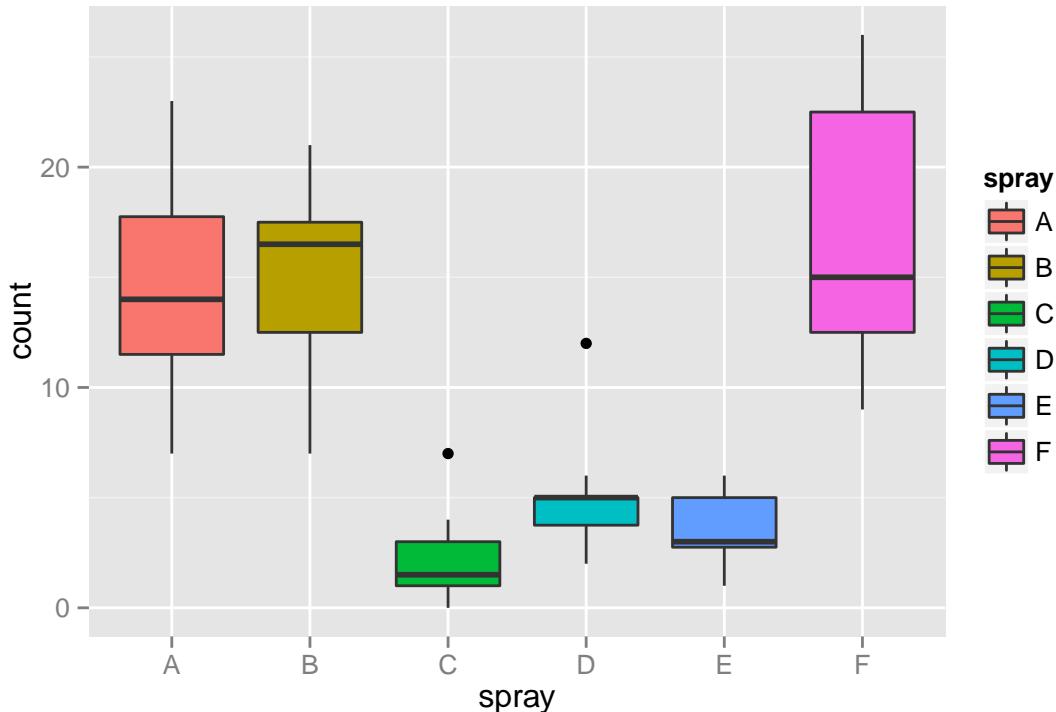
- *variations*

Data type	Statistic	Test name
Ranks	rank sum	rank sum test
Binary	hypergeometric prob	Fisher's exact test
Raw data		ordinary permutation test

- \* *Note:* randomization tests are exactly permutation tests, with a different motivation
- \* For matched data, one can randomize the signs
- \* For ranks, this results in the **signed rank test**
- \* Permutation strategies work for regression by permuting a regressor of interest
- \* Permutation tests work very well in multivariate settings

– *example*

- \* we will compare groups **B** and **C** in this dataset for null hypothesis  $H_0$  : there are no difference between the groups



- we will compare groups **B** and **C** in this dataset for null hypothesis  $H_0$  : there are no difference between the groups

```
# subset to only "B" and "C" groups
subdata <- InsectSprays[InsectSprays$spray %in% c("B", "C"),]
# values
y <- subdata$count
# labels
group <- as.character(subdata$spray)
# find mean difference between the groups
testStat <- function(w, g) mean(w[g == "B"]) - mean(w[g == "C"])
observedStat <- testStat(y, group)
observedStat
```

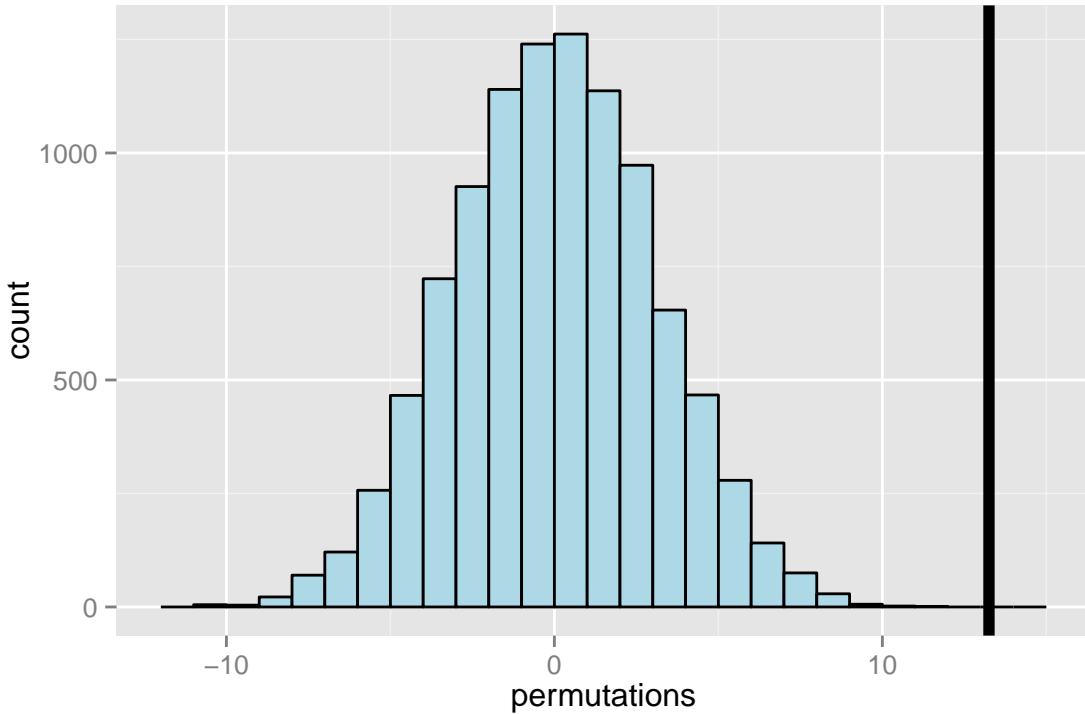
```
## [1] 13.25
```

- the observed difference between the groups is 13.25
- now we changed the resample the lables for groups **B** and **C**

```
# create 10000 permutations of the data with the labels' changed
permutations <- sapply(1 : 10000, function(i) testStat(y, sample(group)))
# find the number of permutations whose difference that is bigger than the observed
mean(permutations > observedStat)
```

```
## [1] 0
```

- we created 1000 permutations from the observed dataset, and found ***no datasets*** with mean differences between groups **B** and **C** larger than the original data
- therefore, p-value is very small and we can ***reject the null*** hypothesis with any reasonable  $\alpha$  levels
- below is the plot for the null distribution/permuations



- as we can see from the black line, the observed difference/statistic is very far from the mean  $\rightarrow$  likely 0 is ***not*** the true difference
  - with this information, formal confidence intervals can be constructed and p-values can be calculated

# Regression Models Course Notes

Xing Su

## Contents

Introduction to Regression . . . . .	4
Notation . . . . .	4
Empirical/Sample Mean . . . . .	4
Empirical/Sample Standard Deviation & Variance . . . . .	4
Normalization . . . . .	5
Empirical Covariance & Correlation . . . . .	5
Dalton's Data and Least Squares . . . . .	6
Derivation for Least Squares = Empirical Mean (Finding the Minimum) . . . . .	8
Regression through the Origin . . . . .	9
Derivation for $\beta$ . . . . .	10
Finding the Best Fit Line (Ordinary Least Squares) . . . . .	11
Least Squares Model Fit . . . . .	11
Derivation for $\beta_0$ and $\beta_1$ . . . . .	12
Examples and R Commands . . . . .	12
Regression to the Mean . . . . .	15
Dalton's Investigation on Regression to the Mean . . . . .	15
Statistical Linear Regression Models . . . . .	17
Interpreting Regression Coefficients . . . . .	17
Use Regression Coefficients for Prediction . . . . .	18
Example and R Commands . . . . .	18
Derivation for Maximum Likelihood Estimator . . . . .	21
Residuals . . . . .	23
Estimating Residual Variation . . . . .	23
Total Variation, $R^2$ , and Derivations . . . . .	24
Example and R Commands . . . . .	26
Inference in Regression . . . . .	29
Intervals/Tests for Coefficients . . . . .	29
Prediction Interval . . . . .	31
Multivariate Regression . . . . .	34
Derivation of Coefficients . . . . .	34
Interpretation of Coefficients . . . . .	37

Example: Linear Model with 2 Variables and Intercept . . . . .	38
Example: Coefficients that Reverse Signs . . . . .	38
Example: Unnecessary Variables . . . . .	41
Dummy Variables . . . . .	43
More Than 2 Levels . . . . .	43
Example: 6 Factor Level Insect Spray Data . . . . .	43
Interactions . . . . .	47
Model: % Hungry ~ Year by Sex . . . . .	47
Model: % Hungry ~ Year + Sex (Binary Variable) . . . . .	48
Model: % Hungry ~ Year + Sex + Year * Sex (Binary Interaction) . . . . .	49
Example: % Hungry ~ Year + Income + Year * Income (Continuous Interaction) . . . . .	51
Multivariable Simulation . . . . .	52
Simulation 1 - Treatment = Adjustment Effect . . . . .	52
Simulation 2 - No Treatment Effect . . . . .	53
Simulation 3 - Treatment Reverses Adjustment Effect . . . . .	55
Simulation 4 - No Adjustment Effect . . . . .	56
Simulation 5 - Binary Interaction . . . . .	58
Simulation 6 - Continuous Adjustment . . . . .	59
Summary and Considerations . . . . .	62
Residuals and Diagnostics . . . . .	63
Outliers and Influential Points . . . . .	64
Influence Measures . . . . .	65
Using Influence Measures . . . . .	66
Example - Outlier Causing Linear Relationship . . . . .	66
Example - Real Linear Relationship . . . . .	68
Example - Stefanski TAS 2007 . . . . .	69
Model Selection . . . . .	71
Rumsfeldian Triplet . . . . .	71
General Rules . . . . .	71
Example - $R^2$ v $n$ . . . . .	72
Adjusted $R^2$ . . . . .	73
Example - Unrelated Regressors . . . . .	73
Example - Highly Correlated Regressors / Variance Inflation . . . . .	74
Example: Variance Inflation Factors . . . . .	76
Residual Variance Estimates . . . . .	77
Covariate Model Selection . . . . .	77

Example: ANOVA . . . . .	78
Example: Step-wise Model Search . . . . .	78
General Linear Models Overview . . . . .	80
Simple Linear Model . . . . .	80
Logistic Regression . . . . .	80
Poisson Regression . . . . .	81
Variances and Quasi-Likelihoods . . . . .	82
Solving for Normal and Quasi-Likelihood Normal Equations . . . . .	83
General Linear Models - Binary Models . . . . .	84
Odds . . . . .	84
Example - Baltimore Ravens Win vs Loss . . . . .	84
Example - Simple Linear Regression . . . . .	85
Example - Logistic Regression . . . . .	86
Example - ANOVA for Logistic Regression . . . . .	89
Further resources . . . . .	90
General Linear Models - Poisson Models . . . . .	91
Properties of Poisson Distribution . . . . .	91
Example - Leek Group Website Traffic . . . . .	92
Example - Linear Regression . . . . .	93
Example - log Outcome . . . . .	93
Example - Poisson Regression . . . . .	94
Example - Robust Standard Errors with Poisson Regression . . . . .	95
Example - Rates . . . . .	96
Further Resources . . . . .	98
Fitting Functions . . . . .	99
Considerations . . . . .	99
Example - Fitting Piecewise Linear Function . . . . .	99
Example - Fitting Piecewise Quadratic Function . . . . .	100
Example - Harmonics using Linear Models . . . . .	101

## Introduction to Regression

- linear regression/linear models -> go to procedure to analyze data
- *Francis Galton* invented the term and concepts of regression and correlation
  - he predicted child's height from parents height
- questions that regression can help answer
  - prediction of one thing from another
  - find simple, interpretable, meaningful model to predict the data
  - quantify and investigate variations that are unexplained or unrelated to the predictor -> **residual variation**
  - quantify the effects of other factors may have on the outcome
  - assumptions to generalize findings beyond data we have -> **statistical inference**
  - **regression to the mean** (see below)

## Notation

- regular letters (i.e.  $X, Y$ ) = generally used to denote **observed** variables
- Greek letters (i.e.  $\mu, \sigma$ ) = generally used to denote **unknown** variables that we are trying to estimate
- $X_1, X_2, \dots, X_n$  describes  $n$  data points
- $\bar{X}, \bar{Y}$  = observed means for random variables  $X$  and  $Y$
- $\hat{\beta}_0, \hat{\beta}_1$  = estimators for true values of  $\beta_0$  and  $\beta_1$

## Empirical/Sample Mean

- **empirical mean** is defined as

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

- **centering** the random variable is defined as

$$\tilde{X}_i = X_i - \bar{X}$$

– mean of  $\tilde{X}_i = 0$

## Empirical/Sample Standard Deviation & Variance

- **empirical variance** is defined as

$$S^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2 = \frac{1}{n-1} \left( \sum_{i=1}^n X_i^2 - n\bar{X}^2 \right) \Leftarrow \text{shortcut for calculation}$$

- **empirical standard deviation** is defined as  $S = \sqrt{S^2}$

– average squared distances between the observations and the mean  
– has same units as the data

- **scaling** the random variables is defined as  $X_i/S$

– standard deviation of  $X_i/S = 1$

## Normalization

- **normalizing** the data/random variable is defined

$$Z_i = \frac{X_i - \bar{X}}{s}$$

- empirical mean = 0, empirical standard deviation = 1
- distribution centered around 0 and data have units = # of standard deviations away from the original mean
  - \* **example:**  $Z_1 = 2$  means that the data point is 2 standard deviations larger than the original mean
- normalization makes non-comparable data **comparable**

## Empirical Covariance & Correlation

- Let  $(X_i, Y_i)$  = pairs of data
- **empirical covariance** is defined as

$$Cov(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y}) = \frac{1}{n-1} \left( \sum_{i=1}^n X_i Y_i - n \bar{X} \bar{Y} \right)$$

- has units of  $X \times$  units of  $Y$
- **correlation** is defined as

$$Cor(X, Y) = \frac{Cov(X, Y)}{S_x S_y}$$

where  $S_x$  and  $S_y$  are the estimates of standard deviations for the  $X$  observations and  $Y$  observations, respectively

- the value is effectively the covariance standardized into a unit-less quantity
- $Cor(X, Y) = Cor(Y, X)$
- $-1 \leq Cor(X, Y) \leq 1$
- $Cor(X, Y) = 1$  and  $Cor(X, Y) = -1$  only when the  $X$  or  $Y$  observations fall perfectly on a positive or negative sloped line, respectively
- $Cor(X, Y)$  measures the strength of the linear relationship between the  $X$  and  $Y$  data, with stronger relationships as  $Cor(X, Y)$  heads towards -1 or 1
- $Cor(X, Y) = 0$  implies no linear relationship

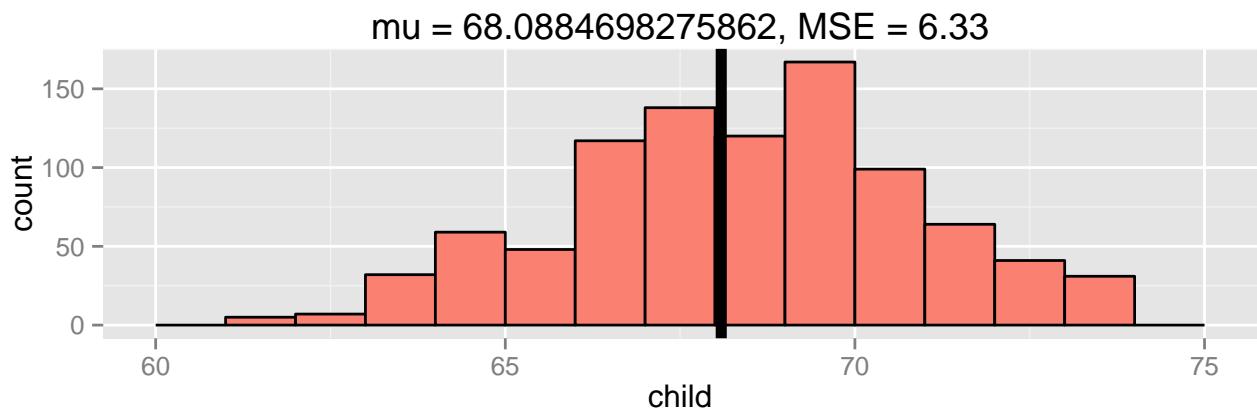
## Dalton's Data and Least Squares

- collected data from 1885 in **UsingR** package
- predicting children's heights from parents' height
- observations from the marginal/individual parent/children distributions
- looking only at the children's dataset to find the best predictor
  - “middle” of children's dataset  $\rightarrow$  best predictor
  - “middle”  $\rightarrow$  center of mass  $\rightarrow$  mean of the dataset
  - \* Let  $Y_i$  = height of child  $i$  for  $i = 1, \dots, n = 928$ , the “middle” =  $\mu$  such that

$$\sum_{i=1}^n (Y_i - \mu)^2$$

- \*  $\mu = \bar{Y}$  for the above sum to be the smallest  $\rightarrow$  **least squares = empirical mean**
- *Note: manipulate function can help to show this*

```
# load necessary packages/install if needed
library(ggplot2); library(UsingR); data(galton)
# function to plot the histograms
myHist <- function(mu){
  # calculate the mean squares
  mse <- mean((galton$child - mu)^2)
  # plot histogram
  g <- ggplot(galton, aes(x = child)) + geom_histogram(fill = "salmon",
    colour = "black", binwidth=1)
  # add vertical line marking the center value mu
  g <- g + geom_vline(xintercept = mu, size = 2)
  g <- g + ggtitle(paste("mu = ", mu, ", MSE = ", round(mse, 2), sep = ""))
  g
}
# manipulate allows the user to change the variable mu to see how the mean squares changes
#   library(manipulate); manipulate(myHist(mu), mu = slider(62, 74, step = 0.5))
# plot the correct graph
myHist(mean(galton$child))
```

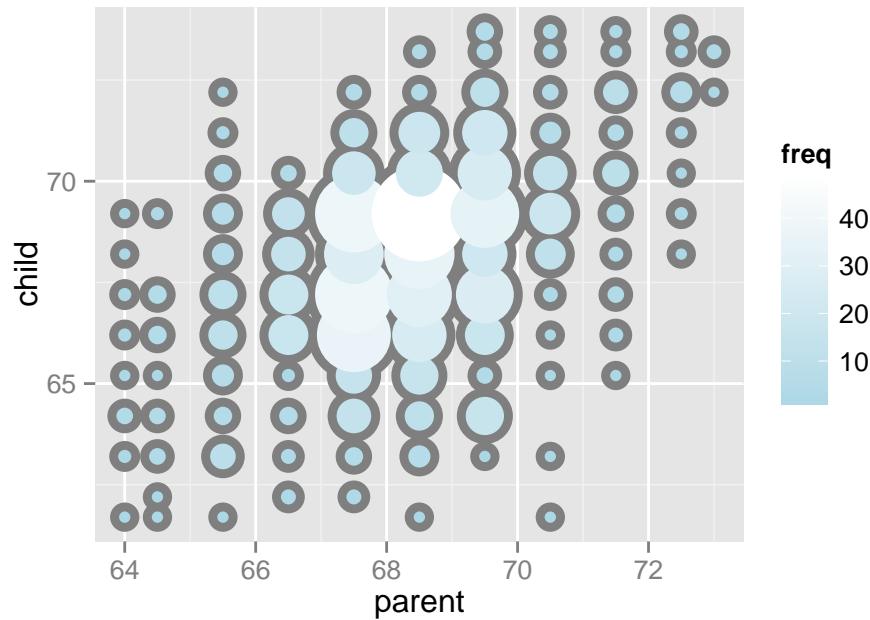


- in order to visualize the parent-child height relationship, a scatter plot can be used
- *Note: because there are multiple data points for the same parent/child combination, a third dimension (size of point) should be used when constructing the scatter plot*

```

library(dplyr)
# constructs table for different combination of parent-child height
freqData <- as.data.frame(table(galton$child, galton$parent))
names(freqData) <- c("child (in)", "parent (in)", "freq")
# convert to numeric values
freqData$child <- as.numeric(as.character(freqData$child))
freqData$parent <- as.numeric(as.character(freqData$parent))
# filter to only meaningful combinations
g <- ggplot(filter(freqData, freq > 0), aes(x = parent, y = child))
g <- g + scale_size(range = c(2, 20), guide = "none" )
# plot grey circles slightly larger than data as base (achieve an outline effect)
g <- g + geom_point(colour="grey50", aes(size = freq+10, show_guide = FALSE))
# plot the accurate data points
g <- g + geom_point(aes(colour=freq, size = freq))
# change the color gradient from default to lightblue-->white
g <- g + scale_colour_gradient(low = "lightblue", high="white")
g

```



## Derivation for Least Squares = Empirical Mean (Finding the Minimum)

- Let  $X_i$  = regressor/predictor, and  $Y_i$  = outcome/result so we want to minimize the the squares:

$$\sum_{i=1}^n (Y_i - \mu)^2$$

- Proof is as follows

$$\begin{aligned}
\sum_{i=1}^n (Y_i - \mu)^2 &= \sum_{i=1}^n (Y_i - \bar{Y} + \bar{Y} - \mu)^2 \Leftarrow \text{added } \pm \bar{Y} \text{ which is adding 0 to the original equation} \\
(\text{expanding the terms}) &= \sum_{i=1}^n (Y_i - \bar{Y})^2 + 2 \sum_{i=1}^n (Y_i - \bar{Y})(\bar{Y} - \mu) + \sum_{i=1}^n (\bar{Y} - \mu)^2 \Leftarrow (Y_i - \bar{Y}), (\bar{Y} - \mu) \text{ are the terms} \\
(\text{simplifying}) &= \sum_{i=1}^n (Y_i - \bar{Y})^2 + 2(\bar{Y} - \mu) \sum_{i=1}^n (Y_i - \bar{Y}) + \sum_{i=1}^n (\bar{Y} - \mu)^2 \Leftarrow (\bar{Y} - \mu) \text{ does not depend on } i \\
(\text{simplifying}) &= \sum_{i=1}^n (Y_i - \bar{Y})^2 + 2(\bar{Y} - \mu)(\sum_{i=1}^n Y_i - n\bar{Y}) + \sum_{i=1}^n (\bar{Y} - \mu)^2 \Leftarrow \sum_{i=1}^n \bar{Y} \text{ is equivalent to } n\bar{Y} \\
(\text{simplifying}) &= \sum_{i=1}^n (Y_i - \bar{Y})^2 + \sum_{i=1}^n (\bar{Y} - \mu)^2 \Leftarrow \sum_{i=1}^n Y_i - n\bar{Y} = 0 \text{ since } \sum_{i=1}^n Y_i = n\bar{Y} \\
\sum_{i=1}^n (Y_i - \mu)^2 &\geq \sum_{i=1}^n (Y_i - \bar{Y})^2 \Leftarrow \sum_{i=1}^n (\bar{Y} - \mu)^2 \text{ is always } \geq 0 \text{ so we can take it out to form the inequality}
\end{aligned}$$

- because of the inequality above, to minimize the sum of the squares  $\sum_{i=1}^n (Y_i - \mu)^2$ ,  $\bar{Y}$  must be equal to  $\mu$
- An alternative approach to finding the minimum is taking the *derivative* with respect to  $\mu$

$$\begin{aligned}
\frac{d(\sum_{i=1}^n (Y_i - \mu)^2)}{d\mu} &= 0 \Leftarrow \text{setting this equal to 0 to find minimum} \\
-2 \sum_{i=1}^n (Y_i - \mu) &= 0 \Leftarrow \text{divide by -2 on both sides and move } \mu \text{ term over to the right} \\
\sum_{i=1}^n Y_i &= \sum_{i=1}^n \mu \Leftarrow \text{for the two sums to be equal, all the terms must be equal} \\
Y_i &= \mu
\end{aligned}$$

## Regression through the Origin

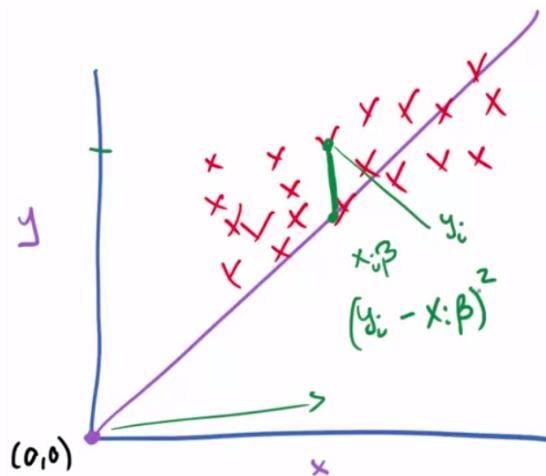
- Let  $X_i$  = parents' heights (**regressor**) and  $Y_i$  = children's heights (**outcome**)
- find a line with slope  $\beta$  that passes through the origin at  $(0,0)$

$$Y_i = X_i\beta$$

such that it minimizes

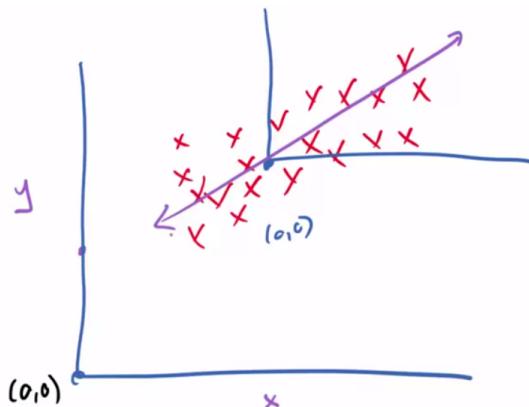
$$\sum_{i=1}^n (Y_i - X_i\beta)^2$$

- Note:** it is generally a bad practice forcing the line through  $(0, 0)$



- Centering Data/Gaussian Elimination**

- Note:** this is **different** from regression through the origin, because it is effectively moving the regression line
- subtracting the means from the  $X_i$ s and  $Y_i$ s moves the origin (reorienting the axes) to the center of the data set so that a regression line can be constructed
- Note:** the line constructed here has an **equivalent slope** as the result from linear regression with intercept



## Derivation for $\beta$

- Let  $Y = \beta X$ , and  $\hat{\beta}$  = estimate of  $\beta$ , the slope of the least square regression line

$$\sum_{i=1}^n (Y_i - X_i\beta)^2 = \sum_{i=1}^n [(Y_i - X_i\hat{\beta}) + (X_i\hat{\beta} - X_i\beta)]^2 \Leftarrow \text{added } \pm X_i\hat{\beta} \text{ is effectively adding zero}$$

(expanding the terms)  $= \sum_{i=1}^n (Y_i - X_i\hat{\beta})^2 + 2 \sum_{i=1}^n (Y_i - X_i\hat{\beta})(X_i\hat{\beta} - X_i\beta) + \sum_{i=1}^n (X_i\hat{\beta} - X_i\beta)^2$

$$\sum_{i=1}^n (Y_i - X_i\beta)^2 \geq \sum_{i=1}^n (Y_i - X_i\hat{\beta})^2 + 2 \sum_{i=1}^n (Y_i - X_i\hat{\beta})(X_i\hat{\beta} - X_i\beta) \Leftarrow \sum_{i=1}^n (X_i\hat{\beta} - X_i\beta)^2 \text{ is always positive}$$

(ignoring the second term for now, for  $\hat{\beta}$  to be the minimizer of the squares, the following must be true)

$$\sum_{i=1}^n (Y_i - X_i\beta)^2 \geq \sum_{i=1}^n (Y_i - X_i\hat{\beta})^2 \Leftarrow \text{every other } \beta \text{ value creates a least square criteria that is } \geq \hat{\beta}$$

(this means)  $\Rightarrow 2 \sum_{i=1}^n (Y_i - X_i\hat{\beta})(X_i\hat{\beta} - X_i\beta) = 0$

(simplifying)  $\Rightarrow \sum_{i=1}^n (Y_i - X_i\hat{\beta})X_i(\hat{\beta} - \beta) = 0 \Leftarrow (\hat{\beta} - \beta) \text{ does not depend on } i$

(simplifying)  $\Rightarrow \sum_{i=1}^n (Y_i - X_i\hat{\beta})X_i = 0$

(solving for  $\hat{\beta}$ )  $\Rightarrow \hat{\beta} = \frac{\sum_{i=1}^n Y_i X_i}{\sum_{i=1}^n X_i^2} = \beta$

- example*

- Let  $X_1, X_2, \dots, X_n = 1$

$$\sum_{i=1}^n (Y_i - X_i\beta)^2 = \sum_{i=1}^n (Y_i - \beta)^2$$

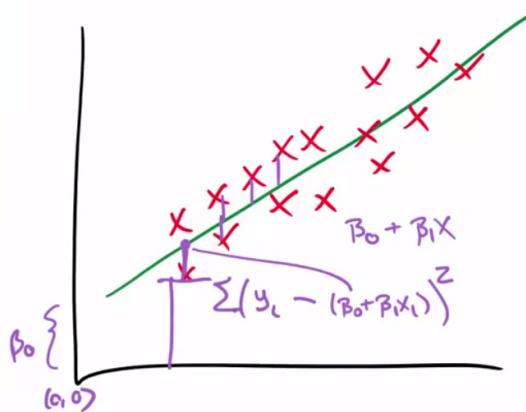
$$\Rightarrow \hat{\beta} = \frac{\sum_{i=1}^n Y_i X_i}{\sum_{i=1}^n X_i^2} = \frac{\sum_{i=1}^n Y_i}{\sum_{i=1}^n 1} = \frac{\sum_{i=1}^n Y_i}{n} = \bar{Y}$$

- Note:* this is the result from our previous derivation for least squares = empirical mean

## Finding the Best Fit Line (Ordinary Least Squares)

- best fitted line for predictor,  $X$ , and outcome,  $Y$  is derived from the **least squares**

$$\sum_{i=1}^n \{Y_i - (\beta_0 + \beta_1 X_i)\}^2$$



- each of the data point contributes equally to the error between their locations and the regression line  $\rightarrow$  goal of regression is to **minimize** this error

### Least Squares Model Fit

- model fit  $\rightarrow Y = \beta_0 + \beta_1 X$  through the data pairs  $(X_i, Y_i)$  where  $Y_i$  as the outcome
  - Note:** this is the model that we use to guide our **estimated** best fit (see below)
- best fit line with estimated slope and intercept ( $X$  as predictor,  $Y$  as outcome)  $\rightarrow$

$$Y = \hat{\beta}_0 + \hat{\beta}_1 X$$

where

$$\hat{\beta}_1 = Cor(Y, X) \frac{Sd(Y)}{Sd(X)} \quad \hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$$

- [slope]  $\hat{\beta}_1$  has the units of  $Y/X$ 
  - \*  $Cor(Y, X)$  = unit-less
  - \*  $Sd(Y)$  = has units of  $Y$
  - \*  $Sd(X)$  = has units of  $X$
- [intercept]  $\hat{\beta}_0$  has the units of  $Y$ 
  - the line passes through the point  $(\bar{X}, \bar{Y})$ 
    - \* this is evident from equation for  $\beta_0$  (rearrange equation)
- best fit line with  $X$  as outcome and  $Y$  as predictor has slope,  $\hat{\beta}_1 = Cor(Y, X) Sd(X) / Sd(Y)$ .
- slope of best fit line = slope of best fit line through the origin for centered data  $(X_i - \bar{X}, Y_i - \bar{Y})$
- slope of best fit line for normalized the data,  $\left\{ \frac{X_i - \bar{X}}{Sd(X)}, \frac{Y_i - \bar{Y}}{Sd(Y)} \right\} = Cor(Y, X)$

## Derivation for $\beta_0$ and $\beta_1$

- Let  $Y = \beta_0 + \beta_1 X$ , and  $\hat{\beta}_0/\hat{\beta}_1$  estimates  $\beta_0/\beta_1$ , the intercept and slope of the least square regression line, respectively

$$\begin{aligned}
\sum_{i=1}^n (Y_i - \beta_0 - \beta_1 X_i)^2 &= \sum_{i=1}^n (Y_i^* - \beta_0)^2 \quad \text{where } Y_i^* = Y_i - \beta_1 X_i \\
\text{solution for } \sum_{i=1}^n (Y_i^* - \beta_0)^2 \Rightarrow \hat{\beta}_0 &= \frac{\sum_{i=1}^n Y_i^*}{n} = \frac{\sum_{i=1}^n Y_i - \beta_1 X_i}{n} \\
&\Rightarrow \hat{\beta}_0 = \frac{\sum_{i=1}^n Y_i}{n} - \beta_1 \frac{\sum_{i=1}^n X_i}{n} \\
&\Rightarrow \hat{\beta}_0 = \bar{Y} - \beta_1 \bar{X} \\
\Rightarrow \sum_{i=1}^n (Y_i - \beta_0 - \beta_1 X_i)^2 &= \sum_{i=1}^n [Y_i - (\bar{Y} - \beta_1 \bar{X}) - \beta_1 X_i]^2 \\
&= \sum_{i=1}^n [(Y_i - \bar{Y}) - (X_i - \bar{X})\beta_1]^2 \\
&= \sum_{i=1}^n [\tilde{Y}_i - \tilde{X}_i \beta_1]^2 \quad \text{where } \tilde{Y}_i = Y_i - \bar{Y}, \tilde{X}_i = X_i - \bar{X} \\
&\Rightarrow \hat{\beta}_1 = \frac{\sum_{i=1}^n \tilde{Y}_i \tilde{X}_i}{\sum_{i=1}^n \tilde{X}_i^2} = \frac{(Y_i - \bar{Y})(X_i - \bar{X})}{\sum_{i=1}^n (X_i - \bar{X})^2} \\
&\Rightarrow \hat{\beta}_1 = \frac{(Y_i - \bar{Y})(X_i - \bar{X})/(n-1)}{\sum_{i=1}^n (X_i - \bar{X})^2/(n-1)} = \frac{Cov(Y, X)}{Var(X)} \\
&\Rightarrow \hat{\beta}_1 = Cor(Y, X) \frac{Sd(Y)}{Sd(X)} \\
&\Rightarrow \hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}
\end{aligned}$$

## Examples and R Commands

- $\hat{\beta}_0$  and  $\hat{\beta}_1$  can be manually calculated through the above formulas
- `coef(lm(y ~ x))` = R command to run the least square regression model on the data with y as the outcome, and x as the regressor
  - `coef()` = returns the slope and intercept coefficients of the `lm` results

```

# outcome
y <- galton$child
# regressor
x <- galton$parent
# slope
beta1 <- cor(y, x) * sd(y) / sd(x)
# intercept
beta0 <- mean(y) - beta1 * mean(x)
# results are the same as using the lm command
results <- rbind("manual" = c(beta0, beta1), "lm(y ~ x)" = coef(lm(y ~ x)))
# set column names
colnames(results) <- c("intercept", "slope")
# print results
results

```

```

##           intercept      slope
## manual      23.94153 0.6462906
## lm(y ~ x)  23.94153 0.6462906

```

- slope of the best fit line = slope of best fit line through the origin for centered data
- $\text{lm}(y \sim x - 1)$  = forces a regression line to go through the origin  $(0, 0)$

```

# centering y
yc <- y - mean(y)
# centering x
xc <- x - mean(x)
# slope
beta1 <- sum(yc * xc) / sum(xc ^ 2)
# results are the same as using the lm command
results <- rbind("centered data (manual)" = beta1, "lm(y ~ x)" = coef(lm(y ~ x))[2],
                 "lm(yc ~ xc - 1)" = coef(lm(yc ~ xc - 1))[1])
# set column names
colnames(results) <- c("slope")
# print results
results

```

```

##           slope
## centered data (manual) 0.6462906
## lm(y ~ x)              0.6462906
## lm(yc ~ xc - 1)        0.6462906

```

- slope of best fit line for normalized the data =  $\text{Cor}(Y, X)$

```

# normalize y
yn <- (y - mean(y))/sd(y)
# normalize x
xn <- (x - mean(x))/sd(x)
# compare correlations
results <- rbind("cor(y, x)" = cor(y, x), "cor(yn, xn)" = cor(yn, xn),
                 "slope" = coef(lm(yn ~ xn))[2])
# print results
results

```

```

##           xn
## cor(y, x) 0.4587624
## cor(yn, xn) 0.4587624
## slope      0.4587624

```

- `geom_smooth(method = "lm", formula = y~x)` function in `ggplot2` = adds regression line and confidence interval to graph
  - `formula = y~x` = default for the line (argument can be eliminated if `y~x` produces the line you want)

```

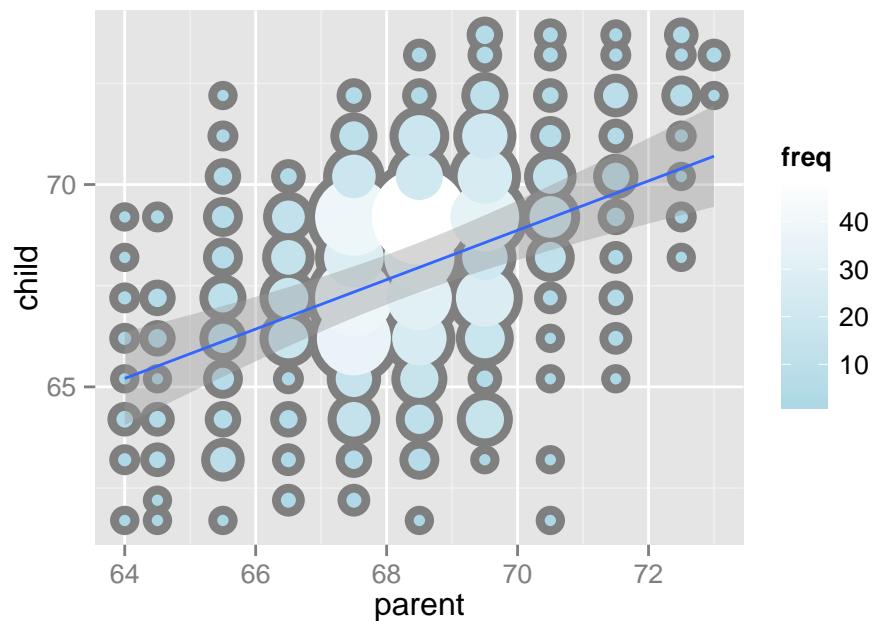
# constructs table for different combination of parent-child height
freqData <- as.data.frame(table(galton$child, galton$parent))
names(freqData) <- c("child (in)", "parent (in)", "freq")

```

```

# convert to numeric values
freqData$child <- as.numeric(as.character(freqData$child))
freqData$parent <- as.numeric(as.character(freqData$parent))
g <- ggplot(filter(freqData, freq > 0), aes(x = parent, y = child))
g <- g + scale_size(range = c(2, 20), guide = "none" )
g <- g + geom_point(colour="grey50", aes(size = freq+10, show_guide = FALSE))
g <- g + geom_point(aes(colour=freq, size = freq))
g <- g + scale_colour_gradient(low = "lightblue", high="white")
g <- g + geom_smooth(method="lm", formula=y~x)
g

```



## Regression to the Mean

- first investigated by Francis Galton in the paper “*Regression towards mediocrity in hereditary stature*”  
*The Journal of the Anthropological Institute of Great Britain and Ireland*, Vol. 15, (1886)
- **regression to the mean** was invented by Francis Galton to capture the following phenomena
  - children of tall parents tend to be tall, but not as tall as their parents
  - children of short parents tend to be short, but not as short as their parents
  - parents of very short children, tend to be short, but not as short as their child
  - parents of very tall children, tend to be tall, but not as tall as their children
- in thinking of the extremes, the following are true
  - $P(Y < x|X = x)$  gets bigger as  $x$  heads to very large values
    - \* in other words, given that the value of  $X$  is already very large (extreme), the chance that the value of  $Y$  is as large or larger than that of  $X$  is small (unlikely)
  - similarly,  $P(Y > x|X = x)$  gets bigger as  $x$  heads to very small values
    - \* in other words, given that the value of  $X$  is already very small (extreme), the chance that the value of  $Y$  is as small or smaller than that of  $X$  is small (unlikely)
- when constructing regression lines between  $X$  and  $Y$ , the line represents the intrinsic relationship (“mean”) between the variables, but does not capture the extremes (“noise”)
  - unless  $\text{Cor}(Y, X) = 1$ , the regression line or the intrinsic part of the relationship between variables won’t capture all of the variation (some noise exists)

## Dalton’s Investigation on Regression to the Mean

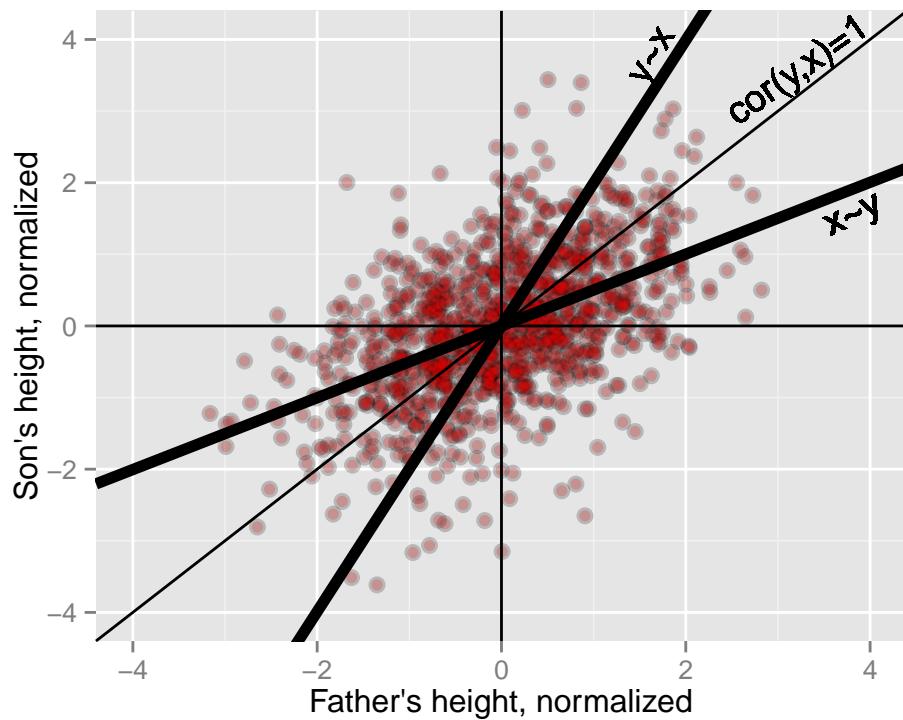
- both  $X$ , child’s heights, and  $Y$ , parent’s heights, are **normalized** so that they mean of 0 and variance of 1
- regression lines must pass  $(\bar{X}, \bar{Y})$  or  $(0, 0)$  in this case
- slope of regression line =  $\text{Cor}(Y, X)$  regardless of which variable is the outcome/regressor (because standard deviations of both variables = 1)
  - **Note:** however, for both regression lines to be plotted on the same graph, the second line’s slope must be  $1/\text{Cor}(Y, X)$  because the two relationships have flipped axes

```
# load father.son data
data(father.son)
# normalize son's height
y <- (father.son$sheight - mean(father.son$sheight)) / sd(father.son$sheight)
# normalize father's height
x <- (father.son$fheight - mean(father.son$fheight)) / sd(father.son$fheight)
# calculate correlation
rho <- cor(x, y)
# plot the relationship between the two
g = ggplot(data.frame(x = x, y = y), aes(x = x, y = y))
g = g + geom_point(size = 3, alpha = .2, colour = "black")
g = g + geom_point(size = 2, alpha = .2, colour = "red")
g = g + xlim(-4, 4) + ylim(-4, 4)
# reference line for perfect correlation between
# variables (data points will lie on diagonal line)
g = g + geom_abline(position = "identity")
# if there is no correlation between the two variables,
# the data points will lie on horizontal/vertical lines
```

```

g = g + geom_vline(xintercept = 0)
g = g + geom_hline(yintercept = 0)
# plot the actual correlation for both
g = g + geom_abline(intercept = 0, slope = rho, size = 2)
g = g + geom_abline(intercept = 0, slope = 1 / rho, size = 2)
# add appropriate labels
g = g + xlab("Father's height, normalized")
g = g + ylab("Son's height, normalized")
g = g + geom_text(x = 3.8, y = 1.6, label="x~y", angle = 25) +
  geom_text(x = 3.2, y = 3.6, label="cor(y,x)=1", angle = 35) +
  geom_text(x = 1.6, y = 3.8, label="y~x", angle = 60)
g

```



## Statistical Linear Regression Models

- goal is use statistics to draw inferences  $\rightarrow$  generalize from data to population through models
- **probabilistic model for linear regression**

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

where  $\epsilon_i$  represents the sampling errors and is assumed to be iid  $N(0, \sigma^2)$

- this model has the following properties
  - $E[Y_i | X_i = x_i] = E[\beta_0] + E[\beta_1 x_i] + E[\epsilon_i] = \mu_i = \beta_0 + \beta_1 x_i$
  - $Var(Y_i | X_i = x_i) = Var(\beta_0 + \beta_1 x_i) + Var(\epsilon_i) = Var(\epsilon_i) = \sigma^2$
  - \*  $\beta_0 + \beta_1 x_i$  = line = constant/no variance
- it can then be said to have  $Y_i$  as independent  $N(\mu, \sigma^2)$ , where  $\mu = \beta_0 + \beta_1 x_i \leftarrow$  likelihood equivalent model
  - **likelihood** = given the outcome, what is the probability?
  - \* in this case, the likelihood is as follows

$$\mathcal{L}(\beta_0, \beta_1, \sigma) = \prod_{i=1}^n \left\{ (2\pi\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}(y_i - \mu_i)^2\right) \right\}$$

where  $\mu_i = \beta_0 + \beta_1 x_i$

- \* above is the **probability density function** of  $n$  samples from the normal distribution  $\rightarrow$  this is because the regression line is normally distributed due to  $\epsilon_i$
- **maximum likelihood estimator (MLE)** = most likely estimate of the population parameter/probability
  - \* in this case, the maximum likelihood = -2 minimum natural log ( $\ln$ , base  $e$ ) likelihood

$$-2 \log \mathcal{L}(\beta_0, \beta_1, \sigma) = n \log(2\pi\sigma^2) + \frac{1}{\sigma^2} \sum_{i=1}^n (y_i - \mu_i)^2$$

- since everything else is constant, minimizing this function would only depend on  $\sum_{i=1}^n (y_i - \mu_i)^2$ , which from our previous derivations yields  $\hat{\mu}_i = \beta_0 + \beta_1 \hat{x}_i$
- \* **maximum likelihood estimate** =  $\mu_i = \beta_0 + \beta_1 x_i$

## Interpreting Regression Coefficients

- for the linear regression line

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

MLE for  $\beta_0$  and  $\beta_1$  are as follows

$$\hat{\beta}_1 = Cor(Y, X) \frac{Sd(Y)}{Sd(X)} \quad \hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$$

- $\beta_0$  = expected value of the outcome/response when the predictor is 0

$$E[Y|X = 0] = \beta_0 + \beta_1 \times 0 = \beta_0$$

- **Note:**  $X = 0$  may not always be of interest as it may be impossible/outside of data range (i.e blood pressure, height etc.)

- it may be useful to move the intercept at times

$$\begin{aligned}
 Y_i &= \beta_0 + \beta_1 X_i + \epsilon_i \\
 &= \beta_0 + a\beta_1 + \beta_1(X_i - a) + \epsilon_i \\
 &= \tilde{\beta}_0 + \beta_1(X_i - a) + \epsilon_i \quad \text{where } \tilde{\beta}_0 = \beta_0 + a\beta_1
 \end{aligned}$$

- **Note:** shifting  $X$  values by value  $a$  changes the intercept, but not the slope
- often,  $a$  is set to  $\bar{X}$  so that the intercept is interpreted as the expected response at the average  $X$  value

- $\beta_1$  = expected change in outcome/response for a 1 unit change in the predictor

$$E[Y | X = x + 1] - E[Y | X = x] = \beta_0 + \beta_1(x + 1) - (\beta_0 + \beta_1x) = \beta_1$$

- sometimes it is useful to change the units of  $X$

$$\begin{aligned}
 Y_i &= \beta_0 + \beta_1 X_i + \epsilon_i \\
 &= \beta_0 + \frac{\beta_1}{a}(X_i a) + \epsilon_i \\
 &= \beta_0 + \tilde{\beta}_1(X_i a) + \epsilon_i
 \end{aligned}$$

- multiplication of  $X$  by a factor  $a$  results in dividing the coefficient by a factor of  $a$
- **example:**

- \*  $X$  = height in  $m$
- \*  $Y$  = weight in  $kg$
- \*  $\beta_1$  has units of  $kg/m$
- \* converting  $X$  to  $cm \implies$  multiplying  $X$  by  $100cm/m$
- \* this mean  $\beta_1$  has to be divided by  $100cm/m$  for the correct units.

$$X \text{ m} \times 100 \frac{cm}{m} = (100 X)cm \quad \text{and} \quad \beta_1 \frac{kg}{m} \times \frac{1}{100} \frac{m}{cm} = \left( \frac{\beta_1}{100} \right) \frac{kg}{cm}$$

- 95% confidence intervals for the coefficients can be constructed from the coefficients themselves and their standard errors (from `summary(lm)`)
  - use the resulting intervals to evaluate the significance of the results

## Use Regression Coefficients for Prediction

- for observed values of the predictor,  $X_1, X_2, \dots, X_n$ , the prediction of the outcome/response is as follows

$$\hat{\mu}_i = \hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X$$

where  $\mu_i$  describes a point on the regression line

## Example and R Commands

- `diamond` dataset from `UsingR` package
  - diamond prices in Singapore Dollars, diamond weight in carats (standard measure of diamond mass,  $0.2g$ )
- `lm(price ~ I(carat - mean(carat)), data=diamond)` = mean centered linear regression
  - **Note:** arithmetic operations must be enclosed in `I()` to work

- `predict(fitModel, newdata=data.frame(carat=c(0, 1, 2)))` = returns predicted outcome from the given model (linear in our case) at the provided points within the `newdata` data frame
  - if `newdata` is unspecified (argument omitted), then `predict` function will return predicted values for all values of the predictor (`x` variable, `carat` in this case)
    - \* *Note: newdata has to be a dataframe, and the values you would like to predict (x variable, carat in this case) has to be specified, or the system won't know what to do with the provided values*
- `summary(fitModel)` = prints detailed summary of linear model
- *example*

```
# standard linear regression for price vs carat
fit <- lm(price ~ carat, data = diamond)
# intercept and slope
coef(fit)
```

```
## (Intercept)      carat
## -259.6259    3721.0249
```

```
# mean-centered regression
fit2 <- lm(price ~ I(carat - mean(carat)), data = diamond)
# intercept and slope
coef(fit2)
```

```
##              (Intercept) I(carat - mean(carat))
##                  500.0833           3721.0249
```

```
# regression with more granular scale (1/10th carat)
fit3 <- lm(price ~ I(carat * 10), data = diamond)
# intercept and slope
coef(fit3)
```

```
##      (Intercept) I(carat * 10)
##      -259.6259     372.1025
```

```
# predictions for 3 values
newx <- c(0.16, 0.27, 0.34)
# manual calculations
coef(fit)[1] + coef(fit)[2] * newx
```

```
## [1] 335.7381 745.0508 1005.5225
```

```
# prediction using the predict function
predict(fit, newdata = data.frame(carat = newx))
```

```
##      1      2      3
## 335.7381 745.0508 1005.5225
```

- **interpretation**

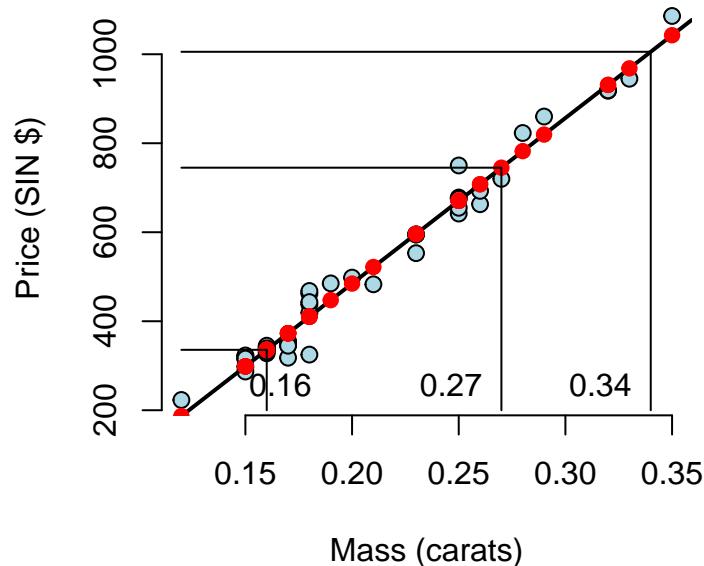
- we expect 3721.02 (SIN) dollar increase in price for *every carat increase* in mass of diamond

- or 372.1 (SIN) dollar increase in price for *every 1/10 carat* increase in mass of diamond

- **prediction**

- for 0.16, 0.27, and 0.34 carats, we predict the prices to be 335.74, 745.05, 1005.52 (SIN) dollars

```
# plot the data points
plot(diamond$carat, diamond$price, xlab = "Mass (carats)", ylab = "Price (SIN $)",
      bg = "lightblue", col = "black", cex = 1.1, pch = 21, frame = FALSE)
# plot linear regression line
abline(fit, lwd = 2)
# plot predictions for every value of carat (in red)
points(diamond$carat, predict(fit), pch = 19, col = "red")
# add guidelines for predictions for 0.16, 0.27, and 0.34
lines(c(0.16, 0.16, 0.12), c(200, coef(fit)[1] + coef(fit)[2] * 0.16,
      coef(fit)[1] + coef(fit)[2] * 0.16))
lines(c(0.27, 0.27, 0.12), c(200, coef(fit)[1] + coef(fit)[2] * 0.27,
      coef(fit)[1] + coef(fit)[2] * 0.27))
lines(c(0.34, 0.34, 0.12), c(200, coef(fit)[1] + coef(fit)[2] * 0.34,
      coef(fit)[1] + coef(fit)[2] * 0.34))
# add text labels
text(newx+c(0.03, 0, 0), rep(250, 3), labels = newx, pos = 2)
```



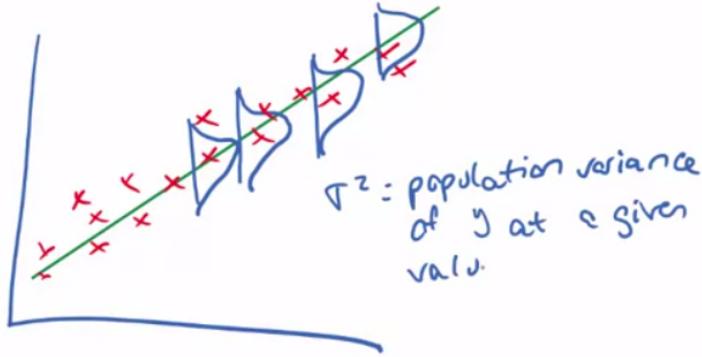
## Derivation for Maximum Likelihood Estimator

- **Note:** this derivation is for the maximum likelihood estimator of the mean,  $\mu$ , of a normal distribution as it is the basis of the linear regression model
- **linear regression model**

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i$$

follows a normal distribution because  $\epsilon_i \sim N(0, \sigma^2)$

- for the above model,  $E[Y_i] = \mu_i = \beta_0 + \beta_1 X_i$  and  $Var(Y_i) = \sigma^2$



- the **probability density function (pdf)** for an outcome  $x$  from the normal distribution is defined as

$$f(x | \mu, \sigma^2) = (2\pi\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}(y_i - \mu_i)^2\right)$$

- the corresponding pdf for  $n$  iid normal random outcomes  $x_1, \dots, x_n$  is defined as

$$f(x_1, \dots, x_n | \mu, \sigma^2) = \prod_{i=1}^n (2\pi\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}(y_i - \mu_i)^2\right)$$

which is also known as the **likelihood function**, denoted in this case as  $\mathcal{L}(\mu, \sigma)$

- to find the **maximum likelihood estimator (MLE)** of the mean,  $\mu$ , we take the derivative of the likelihood  $\mathcal{L}$  with respect to  $\mu \rightarrow \frac{\partial \mathcal{L}}{\partial \mu}$
- since derivatives of products are quite complex to compute, taking the log (base  $e$ ) makes the calculation much simpler

– log properties:

- \*  $\log(AB) = \log(A) + \log(B)$
- \*  $\log(A^B) = B \log(A)$

– because log is always increasing and **monotonic**, or preserves order, finding the maximum MLE = finding the maximum of log transformation of MLE

- -2 log of **likelihood function**

$$\log(\mathcal{L}(\mu, \sigma)) = \sum_{i=1}^n -\frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(y_i - \mu_i)^2 \Leftarrow \text{multiply -2 on both sides}$$

$$-2 \log(\mathcal{L}(\mu, \sigma)) = \sum_{i=1}^n \log(2\pi\sigma^2) + \frac{1}{\sigma^2}(y_i - \mu_i)^2 \Leftarrow \sigma^2 \text{ does not depend on } i$$

$$-2 \log(\mathcal{L}(\mu, \sigma)) = n \log(2\pi\sigma^2) + \frac{1}{\sigma^2} \sum_{i=1}^n (y_i - \mu_i)^2$$

- minimizing -2 log likelihood is *computationally equivalent* as maximizing log likelihood

$$\frac{\partial \log(\mathcal{L}(\mu, \sigma))}{\partial \mu} = \frac{1}{\sigma^2} \frac{\partial \sum_{i=1}^n (y_i - \mu_i)^2}{\partial \mu} = 0 \Leftarrow \text{setting this equal to 0 to find minimum}$$

$$\Rightarrow -\frac{2}{\sigma^2} \sum_{i=1}^n (y_i - \mu_i) = 0 \Leftarrow \text{divide by } -2/\sigma^2 \text{ on both sides}$$

$$\sum_{i=1}^n (y_i - \mu_i) = 0 \Leftarrow \text{move } \mu \text{ term over to the right}$$

$$\sum_{i=1}^n y_i = \sum_{i=1}^n \mu_i \Leftarrow \text{for the two sums to be equal, all the terms must be equal}$$

$$y_i = \mu_i$$

- in the case of our linear regression,  $\mu_i = \beta_0 + \beta_1 X_i$  so

$$\frac{\partial \mathcal{L}(\mu, \sigma)}{\partial \mu} = \frac{\partial \mathcal{L}(\beta_0, \beta_1, \sigma)}{\partial \mu}$$

MLE  $\Rightarrow Y_i = \mu_i$

$$\mu_i = \beta_0 + \beta_1 X_i$$

## Residuals

- Residual,  $e_i$  = difference between the observed and predicted outcome
 
$$e_i = Y_i - \hat{Y}_i$$
  - Or, vertical distance between observed data point and regression line
  - Least squares minimizes  $\sum_{i=1}^n e_i^2$
- $e_i$  can be interpreted as estimates of the regression error,  $\epsilon_i$
- $e_i$  can also be interpreted as the outcome ( $Y$ ) with the linear association of the predictor ( $X$ ) removed
  - or, “ $Y$  adjusted for  $X$ ”
- $E[e_i] = 0 \rightarrow$  this is because the mean of the residuals is expected to be 0 (assumed Gaussian distribution)
  - the Gaussian distribution assumption also implies that the error is **NOT** correlated with any predictors
  - `mean(fitModel$residuals)` = returns mean of residuals  $\rightarrow$  should equal to 0
  - `cov(fit$residuals, predictors)` = returns the covariance (measures correlation) of residuals and predictors  $\rightarrow$  should also equal to 0
- $\sum_{i=1}^n e_i = 0$  (if an intercept is included) and  $\sum_{i=1}^n e_i X_i = 0$  (if a regressor variable  $X_i$  is included)
- for standard linear regression model
  - positive residuals = above the line
  - negative residuals = below
- residuals/residual plots can highlight poor model fit

## Estimating Residual Variation

- residual variation measures how well the regression line fit the data points
- **MLE of variance**,  $\sigma^2$ , of the linear model =  $\frac{1}{n} \sum_{i=1}^n e_i^2$  or the **average squared residual/mean squared error**
  - the square root of the estimate,  $\sigma$ , = **root mean squared error (RMSE)**
- however, a more common approach is to use

$$\hat{\sigma}^2 = \frac{1}{n-2} \sum_{i=1}^n e_i^2$$

- $n-2$  is used instead of  $n$  to make the estimator **unbiased**  $\rightarrow E[\hat{\sigma}^2] = \sigma^2$
- **Note:** the  $-2$  is accounting for the degrees of freedom for intercept and slope, which had to be estimated
- `deviance(fitModel)` = calculates sum of the squared error/residual for the linear model/residual variation
- `summary(fitModel)$sigma` = returns the residual variation of a fit model or the **unbiased RMSE**
  - `summary(fitModel)` = creates a list of different parameters of the fit model

```
# get data
y <- diamond$price; x <- diamond$carat; n <- length(y)
# linear fit
fit <- lm(y ~ x)
# calculate residual variation through summary and manual
rbind("from summary" = summary(fit)$sigma, "manual" =sqrt(sum(resid(fit)^2) / (n - 2)))
```

```
## [,1]
## from summary 31.84052
## manual      31.84052
```

## Total Variation, $R^2$ , and Derivations

- total variation = *residual variation* (variation after removing predictor) + *systematic/regression variation* (variation explained by regression model)

$$\sum_{i=1}^n (Y_i - \bar{Y})^2 = \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + \sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2$$

- $R^2$  = percent of total variability that is explained by the regression model

$$\begin{aligned} R^2 &= \frac{\text{regression variation}}{\text{total variation}} = \frac{\sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \\ &= 1 - \frac{\text{residual variation}}{\text{total variation}} = 1 - \frac{\sum_{i=1}^n (Y_i - \hat{Y}_i)^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \end{aligned}$$

- $0 \leq R^2 \leq 1$
- $R^2$  = sample correlation squared
  - `cor(outcome, predictor)` = calculates the correlation between predictor and outcome  $\rightarrow$  the same as calculating  $R^2$
- $R^2$  can be a *misleading* summary of model fit
  - deleting data  $\rightarrow$  inflate  $R^2$ .
  - adding terms to a regression model  $\rightarrow$  always increases  $R^2$
  - `example(anscombe)` demonstrates the fallacy of  $R^2$  through the following
    - \* basically same mean and variance of X and Y.
    - \* identical correlations (hence same  $R^2$ )
    - \* same linear regression relationship
- relationship between  $R^2$  and  $r$

Correlation between X and Y  $\Rightarrow r = Cor(Y, X)$

$$\begin{aligned} R^2 &= \frac{\sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \\ &\text{recall } \Rightarrow (\hat{Y}_i - \bar{Y}) = \hat{\beta}_1(X_i - \bar{X}) \\ &(\text{substituting } (\hat{Y}_i - \bar{Y})) = \hat{\beta}_1^2 \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \\ &\text{recall } \Rightarrow \hat{\beta}_1 = Cor(Y, X) \frac{Sd(Y)}{Sd(X)} \\ &(\text{substituting } \hat{\beta}_1) = Cor(Y, X)^2 \frac{Var(Y)}{Var(X)} \times \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} \\ &\text{recall } Var(Y) = \sum_{i=1}^n (Y_i - \bar{Y})^2 \text{ and } Var(X) = \sum_{i=1}^n (X_i - \bar{X})^2 \\ &(\text{simplifying}) \Rightarrow R^2 = Cor(Y, X)^2 \\ &\text{Or } R^2 = r^2 \end{aligned}$$

- **total variation derivation**

First, we know that  $\bar{Y} = \hat{\beta}_0 + \hat{\beta}_1 \bar{X}$   
 $(transforming) \Rightarrow \hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$

We also know that  $\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_i$

Next, the residual  $= (Y_i - \hat{Y}_i) = Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_i$   
 $(substituting \hat{\beta}_0) = Y_i - (\bar{Y} - \hat{\beta}_1 \bar{X}) - \hat{\beta}_1 X_i$   
 $(transforming) \Rightarrow (Y_i - \hat{Y}_i) = (Y_i - \bar{Y}) - \hat{\beta}_1 (X_i - \bar{X})$

Next, the regression difference  $= (\hat{Y}_i - \bar{Y}) = \hat{\beta}_0 - \hat{\beta}_1 X_i - \bar{Y}$   
 $(substituting \hat{\beta}_0) = \bar{Y} - \hat{\beta}_1 \bar{X} - \hat{\beta}_1 X_i - \bar{Y}$   
 $(transforming) \Rightarrow (\hat{Y}_i - \bar{Y}) = \hat{\beta}_1 (X_i - \bar{X})$

$$\begin{aligned} \text{Total Variation} &= \sum_{i=1}^n (Y_i - \bar{Y})^2 = \sum_{i=1}^n (Y_i - \hat{Y}_i + \hat{Y}_i - \bar{Y})^2 \Leftarrow (\text{adding } \pm \hat{Y}_i) \\ &\quad (\text{expanding}) = \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + 2 \sum_{i=1}^n (Y_i - \hat{Y}_i)(\hat{Y}_i - \bar{Y}) + \sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2 \end{aligned}$$

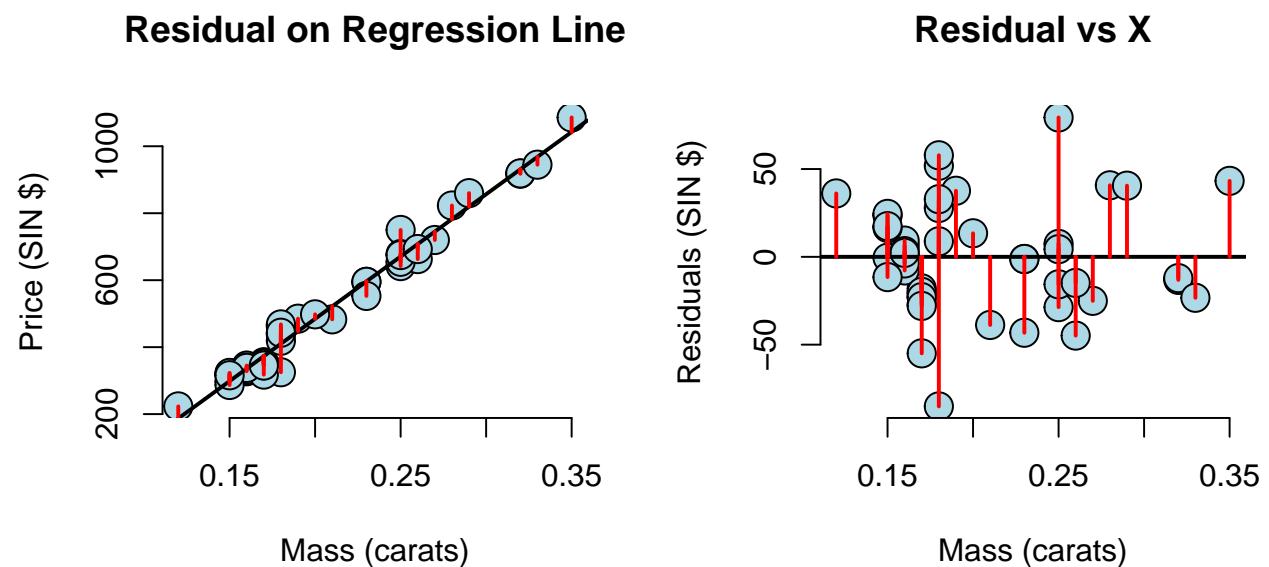
$$\begin{aligned} \text{Looking at } \sum_{i=1}^n (Y_i - \hat{Y}_i)(\hat{Y}_i - \bar{Y}) \\ &\quad (\text{substituting } (Y_i - \hat{Y}_i), (\hat{Y}_i - \bar{Y})) = \sum_{i=1}^n \left[ (Y_i - \bar{Y}) - \hat{\beta}_1 (X_i - \bar{X}) \right] \left[ \hat{\beta}_1 (X_i - \bar{X}) \right] \\ &\quad (\text{expanding}) = \hat{\beta}_1 \sum_{i=1}^n (Y_i - \bar{Y})(X_i - \bar{X}) - \hat{\beta}_1^2 \sum_{i=1}^n (X_i - \bar{X})^2 \\ &\quad (\text{substituting } Y_i, \bar{Y}) \Rightarrow (Y_i - \bar{Y}) = (\hat{\beta}_0 + \hat{\beta}_1 X_i) - (\hat{\beta}_0 + \hat{\beta}_1 \bar{X}) \\ &\quad (\text{simplifying}) \Rightarrow (Y_i - \bar{Y}) = \hat{\beta}_1 (X_i - \bar{X}) \\ &\quad (\text{substituting } (Y_i - \bar{Y})) = \hat{\beta}_1^2 \sum_{i=1}^n (X_i - \bar{X})^2 - \hat{\beta}_1^2 \sum_{i=1}^n (X_i - \bar{X})^2 \\ &\Rightarrow \sum_{i=1}^n (Y_i - \hat{Y}_i)(\hat{Y}_i - \bar{Y}) = 0 \end{aligned}$$

$$\begin{aligned} \text{Going back to } \sum_{i=1}^n (Y_i - \bar{Y})^2 &= \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + 2 \sum_{i=1}^n (Y_i - \hat{Y}_i)(\hat{Y}_i - \bar{Y}) + \sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2 \\ (\text{since second term} = 0) \Rightarrow \sum_{i=1}^n (Y_i - \bar{Y})^2 &= \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 + \sum_{i=1}^n (\hat{Y}_i - \bar{Y})^2 \end{aligned}$$

## Example and R Commands

- `resid(fitModel)` or `fitModel$residuals` = extracts model residuals from the fit model (`lm` in our case)  $\rightarrow$  list of residual values for every value of X
- `summary(fitModel)$r.squared` = return  $R^2$  value of the regression model

```
# load multiplot function
source("multiplot.R")
# get data
y <- diamond$price; x <- diamond$carat; n <- length(y)
# linear regression
fit <- lm(y ~ x)
# calculate residual
e <- resid(fit)
# calculate predicted values
yhat <- predict(fit)
# create 1 x 2 panel plot
par(mfrow=c(1, 2))
# plot residuals on regression line
plot(x, y, xlab = "Mass (carats)", ylab = "Price (SIN $)", bg = "lightblue",
      col = "black", cex = 2, pch = 21, frame = FALSE, main = "Residual on Regression Line")
# draw linear regression line
abline(fit, lwd = 2)
# draw red lines from data points to regression line
for (i in 1 : n){lines(c(x[i], x[i]), c(y[i], yhat[i]), col = "red" , lwd = 2)}
# plot residual vs x
plot(x, e, xlab = "Mass (carats)", ylab = "Residuals (SIN $)", bg = "lightblue",
      col = "black", cex = 2, pch = 21, frame = FALSE, main = "Residual vs X")
# draw horizontal line
abline(h = 0, lwd = 2)
# draw red lines from residual to x axis
for (i in 1 : n){lines(c(x[i], x[i]), c(e[i], 0), col = "red" , lwd = 2)}
```

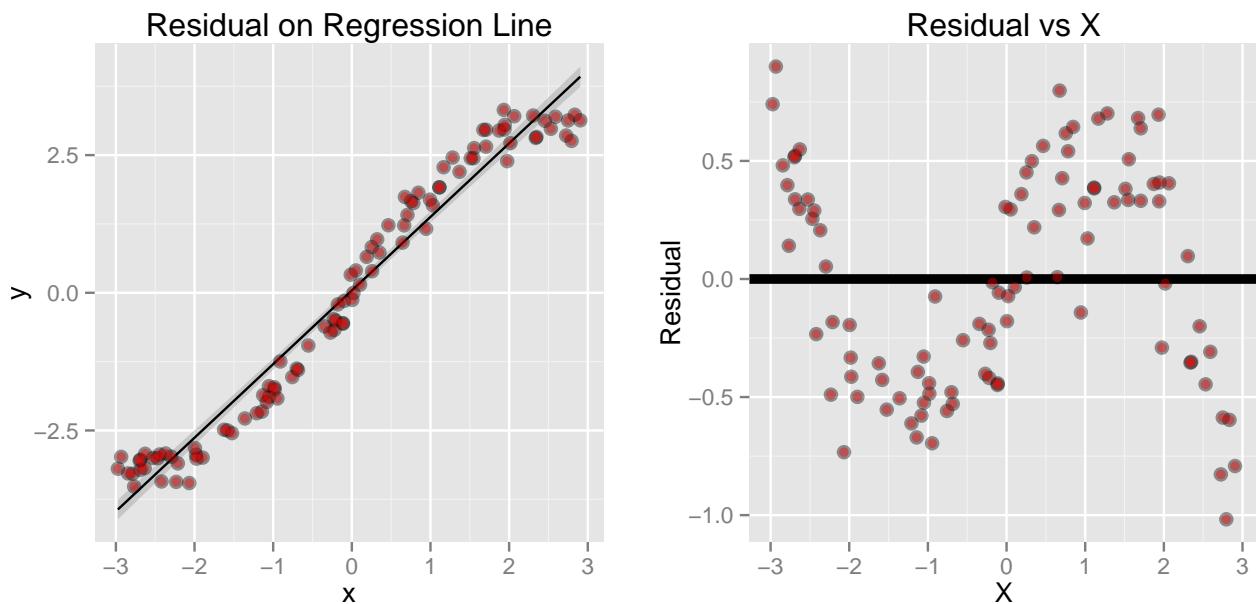


- non-linear data/patterns can be more easily revealed through residual plots

```

# create sin wave pattern
x <- runif(100, -3, 3); y <- x + sin(x) + rnorm(100, sd = .2);
# plot data + regression
g <- ggplot(data.frame(x = x, y = y), aes(x = x, y = y)) +
  geom_smooth(method = "lm", colour = "black") +
  geom_point(size = 3, colour = "black", alpha = 0.4) +
  geom_point(size = 2, colour = "red", alpha = 0.4) +
  ggttitle("Residual on Regression Line")
# plot residuals
f <- ggplot(data.frame(x = x, y = resid(lm(y ~ x))), aes(x = x, y = y)) +
  geom_hline(yintercept = 0, size = 2) +
  geom_point(size = 3, colour = "black", alpha = 0.4) +
  geom_point(size = 2, colour = "red", alpha = 0.4) +
  xlab("X") + ylab("Residual") + ggttitle("Residual vs X")
multiplot(g, f, cols = 2)

```



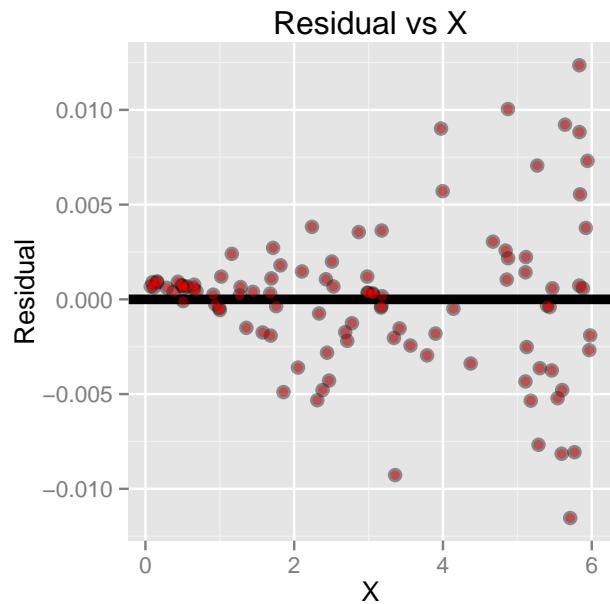
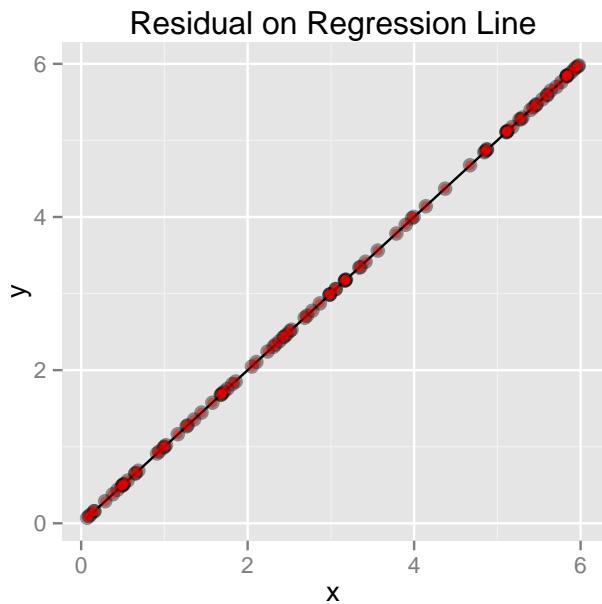
- **heteroskedasticity** → heteroskedastic model's variance is not constant and is a function of x

```

# create heteroskedastic data
x <- runif(100, 0, 6); y <- x + rnorm(100, mean = 0, sd = .001 * x)
# plot data + regression
g <- ggplot(data.frame(x = x, y = y), aes(x = x, y = y)) +
  geom_smooth(method = "lm", colour = "black") +
  geom_point(size = 3, colour = "black", alpha = 0.4) +
  geom_point(size = 2, colour = "red", alpha = 0.4) +
  ggttitle("Residual on Regression Line")
# plot residuals
f <- ggplot(data.frame(x = x, y = resid(lm(y ~ x))), aes(x = x, y = y)) +
  geom_hline(yintercept = 0, size = 2) +
  geom_point(size = 3, colour = "black", alpha = 0.4) +
  geom_point(size = 2, colour = "red", alpha = 0.4) +
  xlab("X") + ylab("Residual") + ggttitle("Residual vs X")

```

```
# combine two plots
multiplot(g, f, cols = 2)
```



## Inference in Regression

- statistics used for hypothesis tests and confidence intervals have the following attributes

$$\frac{\hat{\theta} - \theta}{\hat{\sigma}_{\hat{\theta}}} \sim N(0, 1)$$

- it follows a finite sample Student's **T distribution** and is **normally distributed** if the sample has IID components built in (i.e.  $\epsilon_i$ )
- used to test  $H_0 : \theta = \theta_0$  vs.  $H_a : \theta >, <, \neq \theta_0$ .
- confidence interval for  $\theta = \hat{\theta} \pm Q_{1-\alpha/2} \hat{\sigma}_{\hat{\theta}}$ , where  $Q_{1-\alpha/2}$  = relevant quantile from normal(for large samples)/T distribution(small samples, n-1 degrees of freedom)

## Intervals/Tests for Coefficients

- standard errors for coefficients

$$\begin{aligned} Var(\hat{\beta}_1) &= Var\left(\frac{\sum_{i=1}^n (Y_i - \bar{Y})(X_i - \bar{X})}{((X_i - \bar{X})^2)}\right) \\ (\text{expanding}) &= Var\left(\frac{\sum_{i=1}^n Y_i(X_i - \bar{X}) - \bar{Y} \sum_{i=1}^n (X_i - \bar{X})}{((X_i - \bar{X})^2)}\right) \\ \text{Since } \sum_{i=1}^n X_i - \bar{X} &= 0 \\ (\text{simplifying}) &= \frac{\sum_{i=1}^n Y_i(X_i - \bar{X})}{(\sum_{i=1}^n (X_i - \bar{X})^2)^2} \Leftarrow \text{denominator taken out of } Var \\ (Var(Y_i) = \sigma^2) &= \frac{\sigma^2 \sum_{i=1}^n (X_i - \bar{X})^2}{(\sum_{i=1}^n (X_i - \bar{X})^2)^2} \\ \sigma_{\hat{\beta}_1}^2 &= Var(\hat{\beta}_1) = \frac{\sigma^2}{\sum_{i=1}^n (X_i - \bar{X})^2} \\ \Rightarrow \sigma_{\hat{\beta}_1} &= \frac{\sigma}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2}} \end{aligned}$$

by the same derivation  $\Rightarrow$

$$\begin{aligned} \sigma_{\hat{\beta}_0}^2 &= Var(\hat{\beta}_0) = \left(\frac{1}{n} + \frac{\bar{X}^2}{\sum_{i=1}^n (X_i - \bar{X})^2}\right) \sigma^2 \\ \Rightarrow \sigma_{\hat{\beta}_0} &= \sigma \sqrt{\frac{1}{n} + \frac{\bar{X}^2}{\sum_{i=1}^n (X_i - \bar{X})^2}} \end{aligned}$$

- $\sigma$  is unknown but it's estimate is as follows

$$\hat{\sigma}^2 = \frac{1}{n-2} \sum_{i=1}^n e_i^2$$

- under IID Gaussian errors (assumed in linear regression with term  $\epsilon_0$ ), statistics for  $\hat{\beta}_0$  and  $\hat{\beta}_1$  are as follows

$$\frac{\hat{\beta}_j - \beta_j}{\hat{\sigma}_{\hat{\beta}_j}} \quad \text{where } j = 0, 1$$

- these statistics follow **t distribution** with  $n - 2$  degrees of freedom for small  $n$  and normal distribution for large  $n$

- `summary(fitModel)$coefficients` = returns table summarizing the estimate, standard error, t value and p value for the coefficients  $\beta_0$  and  $\beta_1$ 
  - the below example reproduces the table `summary(fitModel)$coefficients` produces
- **Note:** the variability in the slope, or  $Var(\hat{\beta}_1)$ , is the largest when the predictor values are spread into two clusters that are far apart from each other
  - when modeling linear relationships, it is generally good practice to have many data points that evenly cover the entire range of data, increasing the denominator  $\sum_{i=1}^n (X_i - \bar{X})^2$
  - this is so that variance of slope is minimized and we can be more confident about the relationship
- *example*

```
# getting data
y <- diamond$price; x <- diamond$carat; n <- length(y)
# calculate beta1
beta1 <- cor(y, x) * sd(y) / sd(x)
# calculate beta0
beta0 <- mean(y) - beta1 * mean(x)
# Gaussian regression error
e <- y - beta0 - beta1 * x
# unbiased estimate for variance
sigma <- sqrt(sum(e^2) / (n-2))
# (X_i - X Bar)
ssx <- sum((x - mean(x))^2)
# calculate standard errors
seBeta0 <- (1 / n + mean(x)^2 / ssx)^.5 * sigma
seBeta1 <- sigma / sqrt(ssx)
# testing for H0: beta0 = 0 and beta1 = 0
tBeta0 <- beta0 / seBeta0; tBeta1 <- beta1 / seBeta1
# calculating p-values for Ha: beta0 != 0 and beta1 != 0 (two sided)
pBeta0 <- 2 * pt(abs(tBeta0), df = n - 2, lower.tail = FALSE)
pBeta1 <- 2 * pt(abs(tBeta1), df = n - 2, lower.tail = FALSE)
# store results into table
coefTable <- rbind(c(beta0, seBeta0, tBeta0, pBeta0), c(beta1, seBeta1, tBeta1, pBeta1))
colnames(coefTable) <- c("Estimate", "Std. Error", "t value", "P(>|t|)")
rownames(coefTable) <- c("(Intercept)", "x")
# print table
coefTable
```

```
##             Estimate Std. Error     t value    P(>|t|)
## (Intercept) -259.6259   17.31886 -14.99094 2.523271e-19
## x           3721.0249   81.78588  45.49715 6.751260e-40
```

```
# regression model and the generated table from lm (identical to above)
fit <- lm(y ~ x); summary(fit)$coefficients
```

```
##             Estimate Std. Error     t value    Pr(>|t|)
## (Intercept) -259.6259   17.31886 -14.99094 2.523271e-19
## x           3721.0249   81.78588  45.49715 6.751260e-40
```

```

# store results in matrix
sumCoef <- summary(fit)$coefficients
# print out confidence interval for beta0
sumCoef[1,1] + c(-1, 1) * qt(.975, df = fit$df) * sumCoef[1, 2]

## [1] -294.4870 -224.7649

# print out confidence interval for beta1 in 1/10 units
(sumCoef[2,1] + c(-1, 1) * qt(.975, df = fit$df) * sumCoef[2, 2]) / 10

## [1] 355.6398 388.5651

```

- **interpretation:** With 95% confidence, we estimate that a 0.1 carat increase in diamond size results in a 355.6 to 388.6 increase in price in (Singapore) dollars.

## Prediction Interval

- estimated prediction,  $\hat{y}_0$ , at point  $x_0$  is

$$\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$$

- we can construct two prediction intervals

1. interval for **the line** at  $x_0$

$$\text{Interval : } \hat{y}_0 \pm t_{n-2,1-\alpha/2} \times SE_{line}$$

$$\text{where } \hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$$

$$\text{and } SE_{line} = \hat{\sigma} \sqrt{\frac{1}{n} + \frac{(x_0 - \bar{X})^2}{\sum_{i=1}^n (X_i - \bar{X})^2}}$$

- interval has **varying width**
- interval is narrow as we are quite confident in the regression line
- as  $n$  increases, the interval becomes **narrower**, which makes sense because as more data is collected, we are able to get a better regression line

\* **Note:** if we knew  $\beta_0$  and  $\beta_1$ , this interval would have zero width \*\*

2. interval for the **predicted value**,  $\hat{y}_0$ , at  $x_0$

$$\text{Interval : } \hat{y}_0 \pm t_{n-2,1-\alpha/2} \times SE_{\hat{y}_0}$$

$$\text{where } \hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$$

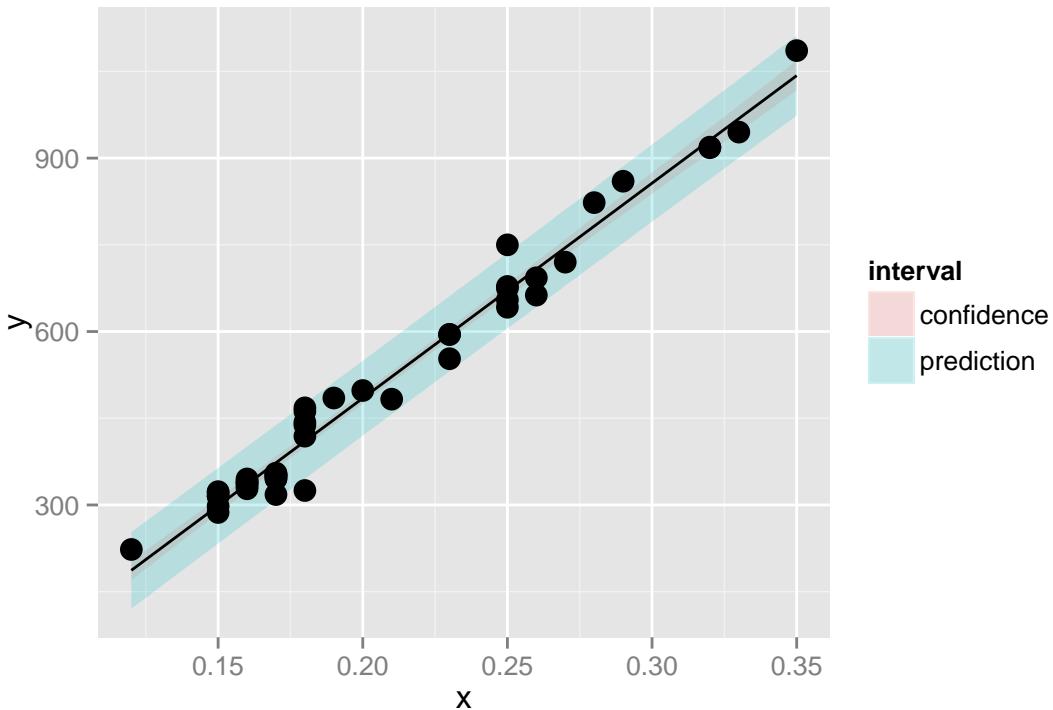
$$\text{and } SE_{\hat{y}_0} = \hat{\sigma} \sqrt{1 + \frac{1}{n} + \frac{(x_0 - \bar{X})^2}{\sum_{i=1}^n (X_i - \bar{X})^2}}$$

- interval has **varying width**
- the 1 part in the  $SE_{\hat{y}_0}$  formula represents the inherent variability in the data
  - \* no matter how good of a regression line we get, we still can not get rid of the variability in the data
- \* **Note:** even if we know  $\beta_0$  and  $\beta_1$ , the interval would still have width due to data variance

- `predict(fitModel, data, interval = ("confidence"))` = returns a 3-column matrix with data for `fit` (regression line), `lwr` (lower bound of interval), and `upr` (upper bound of interval)

- `interval = ("confidence")` = returns interval for the line
- `interval = ("prediction")` = returns interval for the prediction
- `data` = must be a new data frame with the values you would like to predict
- *example (ggplot2)*

```
# create a sequence of values that we want to predict at
newx = data.frame(x = seq(min(x), max(x), length = 100))
# calculate values for both intervals
p1 = data.frame(predict(fit, newdata= newx,interval = ("confidence")))
p2 = data.frame(predict(fit, newdata = newx,interval = ("prediction")))
# add column for interval labels
p1$interval = "confidence"; p2$interval = "prediction"
# add column for the x values we want to predict
p1$x = newx$x; p2$x = newx$x
# combine the two dataframes
dat = rbind(p1, p2)
# change the name of the first column to y
names(dat)[1] = "y"
# plot the data
g <- ggplot(dat, aes(x = x, y = y))
g <- g + geom_ribbon(aes(ymin = lwr, ymax = upr, fill = interval), alpha = 0.2)
g <- g + geom_line()
g + geom_point(data = data.frame(x = x, y=y), aes(x = x, y = y), size = 4)
```



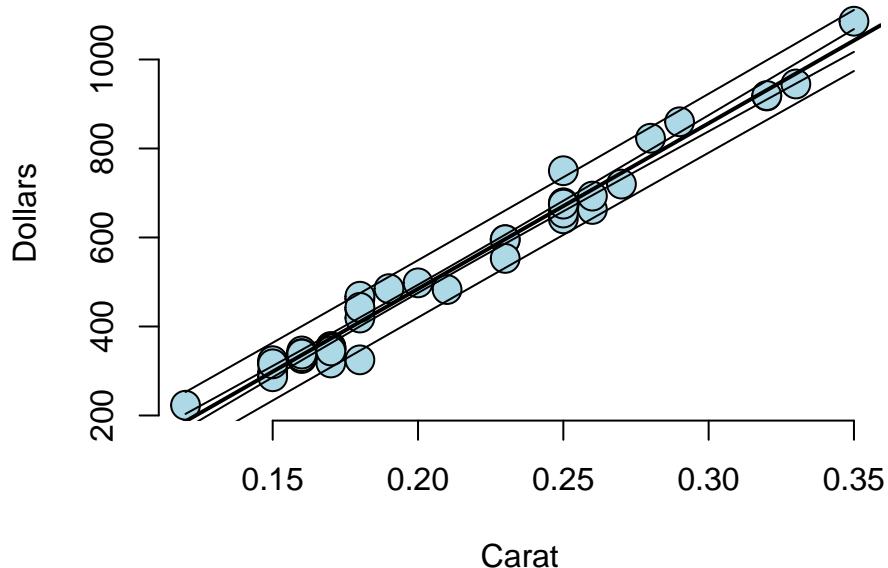
- *example (base)*

```
# plot the x and y values
plot(x,y,frame=FALSE,xlab="Carat",ylab="Dollars",pch=21,col="black",bg="lightblue",cex=2)
# add the fit line
```

```

abline(fit,lwd=2)
# create sequence of x values that we want to predict at
xVals<-seq(min(x),max(x),by=.01)
# calculate the predicted y values
yVals<-beta0+beta1*xVals
# calculate the standard errors for the interval for the line
se1<-sigma*sqrt(1/n+(xVals-mean(x))^2/ssx)
# calculate the standard errors for the interval for the predicted values
se2<-sigma*sqrt(1+1/n+(xVals-mean(x))^2/ssx)
# plot the upper and lower bounds of both intervals
lines(xVals,yVals+2*se1); lines(xVals,yVals-2*se1)
lines(xVals,yVals+2*se2); lines(xVals,yVals-2*se2)

```



## Multivariate Regression

- linear models = most important applied statistical/machine learning technique
- **generalized linear model** extends simple linear regression (SLR) model

$$Y_i = \beta_1 X_{1i} + \beta_2 X_{2i} + \dots + \beta_p X_{pi} + \epsilon_i = \sum_{k=1}^p X_{ki} \beta_j + \epsilon_i$$

where  $X_{1i} = 1$  typically, so that an intercept is included

- least squares/MLE for the model (under IID Gaussian errors) minimizes

$$\sum_{i=1}^n \left( Y_i - \sum_{k=1}^p X_{ki} \beta_j \right)^2$$

- **linearity of coefficients** is what defines a linear model as transformations of the variables (i.e. squaring) still yields a linear model

$$Y_i = \beta_1 X_{1i}^2 + \beta_2 X_{2i}^2 + \dots + \beta_p X_{pi}^2 + \epsilon_i$$

- performing multivariate regression = pick any regressor and replace the outcome and all other regressors by their residuals against the chosen one

## Derivation of Coefficients

- we know for simple univariate regression through the origin

$$E[Y_i] = X_{1i} \beta_1 \hat{\beta}_1 = \frac{\sum_{i=1}^n X_i Y_i}{\sum_{i=1}^n X_i^2}$$

- we want to minimize

$$\sum_{i=1}^n (Y_i - X_{1i} \beta_1 - X_{2i} \beta_2 - \dots - X_{pi} \beta_p)^2$$

- we begin by looking at the two variable model where

$$E[Y_i] = \mu_i = X_{1i} \beta_1 + X_{2i} \beta_2 \hat{\mu}_i = X_{1i} \hat{\beta}_1 + X_{2i} \hat{\beta}_2$$

- from our previous derivations, to minimize the sum of squares, the following has to true

$$\sum_{i=1}^n (Y_i - \hat{\mu}_i)(\hat{\mu}_i - \mu_i) = 0$$

- plugging in  $\hat{\mu}_i$  and  $\mu_i$

$$\begin{aligned} \sum_{i=1}^n (Y_i - \hat{\mu}_i)(\hat{\mu}_i - \mu_i) &= \sum_{i=1}^n (Y_i - X_{1i} \hat{\beta}_1 - X_{2i} \hat{\beta}_2)(X_{1i} \hat{\beta}_1 + X_{2i} \hat{\beta}_2 - X_{1i} \beta_1 - X_{2i} \beta_2) \\ (\text{simplifying}) &= \sum_{i=1}^n (Y_i - X_{1i} \hat{\beta}_1 - X_{2i} \hat{\beta}_2) \left[ X_{1i}(\hat{\beta}_1 - \beta_1) + X_{2i}(\hat{\beta}_2 - \beta_2) \right] \\ (\text{expanding}) &= \sum_{i=1}^n (Y_i - X_{1i} \hat{\beta}_1 - X_{2i} \hat{\beta}_2) X_{1i}(\hat{\beta}_1 - \beta_1) + \sum_{i=1}^n (Y_i - X_{1i} \hat{\beta}_1 - X_{2i} \hat{\beta}_2) X_{2i}(\hat{\beta}_2 - \beta_2) \end{aligned}$$

- for the entire expression to equal to zero,

$$\sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - X_{2i}\hat{\beta}_2)X_{1i}(\hat{\beta}_1 - \beta_1) = 0$$

(since  $\hat{\beta}_1, \beta_1$  don't depend on  $i$ )  $\Rightarrow \sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - X_{2i}\hat{\beta}_2)X_{1i} = 0 \quad (1)$

$$\sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - X_{2i}\hat{\beta}_2)X_{2i}(\hat{\beta}_2 - \beta_2) = 0$$

(since  $\hat{\beta}_2, \beta_2$  don't depend on  $i$ )  $\Rightarrow \sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - X_{2i}\hat{\beta}_2)X_{2i} = 0 \quad (2)$

- we can hold  $\hat{\beta}_1$  fixed and solve (2) for  $\hat{\beta}_2$

$$\begin{aligned} \sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1)X_{2i} - \sum_{i=1}^n X_{2i}^2\hat{\beta}_2 &= 0 \\ \sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1)X_{2i} &= \sum_{i=1}^n X_{2i}^2\hat{\beta}_2 \\ \hat{\beta}_2 &= \frac{\sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1)X_{2i}}{\sum_{i=1}^n X_{2i}^2} \end{aligned}$$

- plugging  $\hat{\beta}_2$  back into (1), we get

$$\begin{aligned} \sum_{i=1}^n \left[ Y_i - X_{1i}\hat{\beta}_1 - \frac{\sum_{j=1}^n (Y_j - X_{1j}\hat{\beta}_1)X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right] X_{1i} &= 0 \\ \sum_{i=1}^n \left[ Y_i - X_{1i}\hat{\beta}_1 - \frac{\sum_{j=1}^n Y_j X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} + \frac{\sum_{j=1}^n X_{1j}X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i}\hat{\beta}_1 \right] X_{1i} &= 0 \\ \sum_{i=1}^n \left[ \left( Y_i - \frac{\sum_{j=1}^n Y_j X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) - \hat{\beta}_1 \left( X_{1i} - \frac{\sum_{j=1}^n X_{1j}X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) \right] X_{1i} &= 0 \\ &\Rightarrow \left( Y_i - \frac{\sum_{j=1}^n Y_j X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) = \text{residual for } Y_i = X_{i2}\hat{\beta}_2 + \epsilon_i \\ &\Rightarrow \left( X_{1i} - \frac{\sum_{j=1}^n X_{1j}X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) = \text{residual for } X_{i1} = X_{i2}\hat{\gamma} + \epsilon_i \end{aligned}$$

- we can rewrite

$$\sum_{i=1}^n \left[ \left( Y_i - \frac{\sum_{j=1}^n Y_j X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) - \hat{\beta}_1 \left( X_{1i} - \frac{\sum_{j=1}^n X_{1j}X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) \right] X_{1i} = 0 \quad (3)$$

as

$$\sum_{i=1}^n \left[ e_{i,Y|X_1} - \hat{\beta}_1 e_{i,X_1|X_2} \right] X_{1i} = 0$$

where

$$e_{i,a|b} = a_i - \frac{\sum_{j=1}^n a_j b_j}{\sum_{j=1}^n b_j^2} b_i$$

which is interpreted as the residual when regressing b from a without an intercept

- solving (3), we get

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n e_{i,Y|X_2} e_{i,X_1|X_2}}{\sum_{i=1}^n e_{i,X_1|X_2} X_{1i}} \quad (4)$$

- to simplify the denominator, we will look at

$$\begin{aligned} \sum_{i=1}^n e_{i,X_1|X_2}^2 &= \sum_{i=1}^n e_{i,X_1|X_2} \left( X_{1i} - \frac{\sum_{j=1}^n X_{1j} X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} \right) \\ &= \sum_{i=1}^n e_{i,X_1|X_2} X_{1i} - \frac{\sum_{j=1}^n X_{1j} X_{2j}}{\sum_{j=1}^n X_{2j}^2} \sum_{i=1}^n e_{i,X_1|X_2} X_{2i} \\ &\quad (\text{recall that } \sum_{i=1}^n e_i X_i = 0, \text{ so the 2nd term is 0}) \\ \Rightarrow \sum_{i=1}^n e_{i,X_1|X_2}^2 &= \sum_{i=1}^n e_{i,X_1|X_2} X_{1i} \end{aligned}$$

- plugging the above equation back in to (4), we get

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n e_{i,Y|X_2} e_{i,X_1|X_2}}{\sum_{i=1}^n e_{i,X_1|X_2}^2}$$

- general case

- pick one regressor and to replace all other variables by the residuals of their regressions against that one

$$\sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - \dots - X_{pi}\hat{\beta}_p) X_k = 0$$

for  $k = 1, \dots, p$  yields  $p$  equations with  $p$  unknowns

- holding  $\hat{\beta}_1, \dots, \hat{\beta}_{p-1}$  constant, we get

$$\hat{\beta}_p = \frac{\sum_{i=1}^n (Y_i - X_{1i}\hat{\beta}_1 - \dots - X_{p-1,i}\hat{\beta}_{p-1}) X_{pi}}{\sum_{i=1}^n X_{pi}^2}$$

- plugging  $\hat{\beta}_p$  back into the equation

$$\sum_{i=1}^n (e_{i,Y|X_p} - e_{i,X_1|X_p}\hat{\beta}_1 - \dots - e_{i,X_{p-1}|X_p}\hat{\beta}_{p-1}) X_k = 0$$

- since we know that

$$X_k = e_{i,X_i|X_p} + \frac{\sum_{i=1}^n X_{ki} X_{pi}}{\sum_{i=1}^n X_{pi}^2} X_p$$

and that

$$\sum_{i=1}^n e_{i,X_i|X_p} X_{pi} = 0$$

the equation becomes

$$\sum_{i=1}^n (e_{i,Y|X_p} - e_{i,X_1|X_p}\hat{\beta}_1 - \dots - e_{i,X_{p-1}|X_p}\hat{\beta}_{p-1}) e_{i,X_k|X_p} = 0$$

- this procedure reduces  $p$  LS equations and  $p$  unknowns to  $p-1$  LS equations and  $p-1$  unknowns
  - \* every variable is replaced by its residual with  $X_p$

- \* process iterates until **only**  $Y$  and **one variable remains**
- \* or more intuitively, we take residuals over the confounding variables and do regression through the origin

- **example**

- for simple linear regression,  $Y_i = \beta_1 X_{1i} + \beta_2 X_{2i}$  where  $X_{2i} = 1$  is an intercept term
- the residuals

$$e_{i,Y|X_2} = Y_i - \frac{\sum_{j=1}^n Y_j X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} = Y_i - \bar{Y}$$

- \* **Note:** this is according to previous derivation of the slope of a regression line through the origin

- the residuals  $e_{i,X_1|X_2} = X_{1i} - \frac{\sum_{j=1}^n X_{1j} X_{2j}}{\sum_{j=1}^n X_{2j}^2} X_{2i} = X_{1i} - \bar{X}_1$

- \* **Note:** this is according to previous derivation of the slope of a regression line through the origin

- Thus

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n e_{i,Y|X_2} e_{i,X_1|X_2}}{\sum_{i=1}^n e_{i,X_1|X_2}^2} = \frac{\sum_{i=1}^n (X_{1i} - \bar{X}_1)(Y_i - \bar{Y})}{\sum_{i=1}^n (X_{1i} - \bar{X}_1)^2} = Cor(X, Y) \frac{Sd(Y)}{Sd(X)}$$

## Interpretation of Coefficients

- from the derivation in the previous section, we have

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n e_{i,Y|X_2} e_{i,X_1|X_2}}{\sum_{i=1}^n e_{i,X_1|X_2}^2}$$

- this is interpreted as the effect of variable  $X_1$  when the effects of all other variables have been removed from  $X_1$  and the predicted result  $Y$  (holding everything else constant/adjusting for all other variables)
- the expected response is as follows

$$E[Y|X_1 = x_1, \dots, X_p = x_p] = \sum_{k=1}^p x_k \beta_k$$

so the expected change in the response through change in one variable is

$$\begin{aligned} & E[Y|X_1 = x_1 + 1, \dots, X_p = x_p] - E[Y|X_1 = x_1, \dots, X_p = x_p] \\ &= (x_1 + 1)\beta_1 + \sum_{k=2}^p x_k \beta_k - \sum_{k=1}^p x_k \beta_k \\ &= (x_1 + 1)\beta_1 + \sum_{k=2}^p x_k \beta_k - (x_1 \beta_1 + \sum_{k=2}^p x_k \beta_k) \\ &= \beta_1 \end{aligned}$$

- therefore, interpretation of a multivariate regression coefficient  $\rightarrow$  expected change in the response per unit change in the regressor, holding all of the other regressors fixed
- all of the SLR properties/calculations extend to generalized linear model

- **model** =  $Y_i = \sum_{k=1}^p X_{ik} \beta_k + \epsilon_i$  where  $\epsilon_i \sim N(0, \sigma^2)$
- **fitted response** =  $\hat{Y}_i = \sum_{k=1}^p X_{ik} \hat{\beta}_k$

- **residual** =  $e_i = Y_i - \hat{Y}_i$
- **variance estimate** =  $\hat{\sigma}^2 = \frac{1}{n-p} \sum_{i=1}^n e_i^2$
- **predicted responses at new values**,  $x_1, \dots, x_p$  = plug  $x$  values into  $\sum_{k=1}^p x_k \hat{\beta}_k$
- **standard errors of coefficients** =  $\hat{\sigma}_{\hat{\beta}_k}$
- **test/CI statistic** =  $\frac{\hat{\beta}_k - \beta_k}{\hat{\sigma}_{\hat{\beta}_k}}$  follows a  $T$  distribution with  $n - p$  degrees of freedom
- **predicted/expected response intervals** = calculated using standard errors of predicted responses of  $\hat{Y}_i$

### Example: Linear Model with 2 Variables and Intercept

```
# simulate the data
n = 100; x = rnorm(n); x2 = rnorm(n); x3 = rnorm(n)
# equation = intercept + var1 + var2 + var3 + error
y = 1 + x + x2 + x3 + rnorm(n, sd = .1)
# residual of y regressed on var2 and var3
ey = resid(lm(y ~ x2 + x3))
# residual of y regressed on var2 and var3
ex = resid(lm(x ~ x2 + x3))
# estimate beta1 for var1
sum(ey * ex) / sum(ex ^ 2)
```

## [1] 1.004808

```
# regression through the origin with xval with var2 var3 effect removed
coef(lm(ey ~ ex - 1))
```

```
##      ex
## 1.004808
```

```
# regression for all three variables
coef(lm(y ~ x + x2 + x3))
```

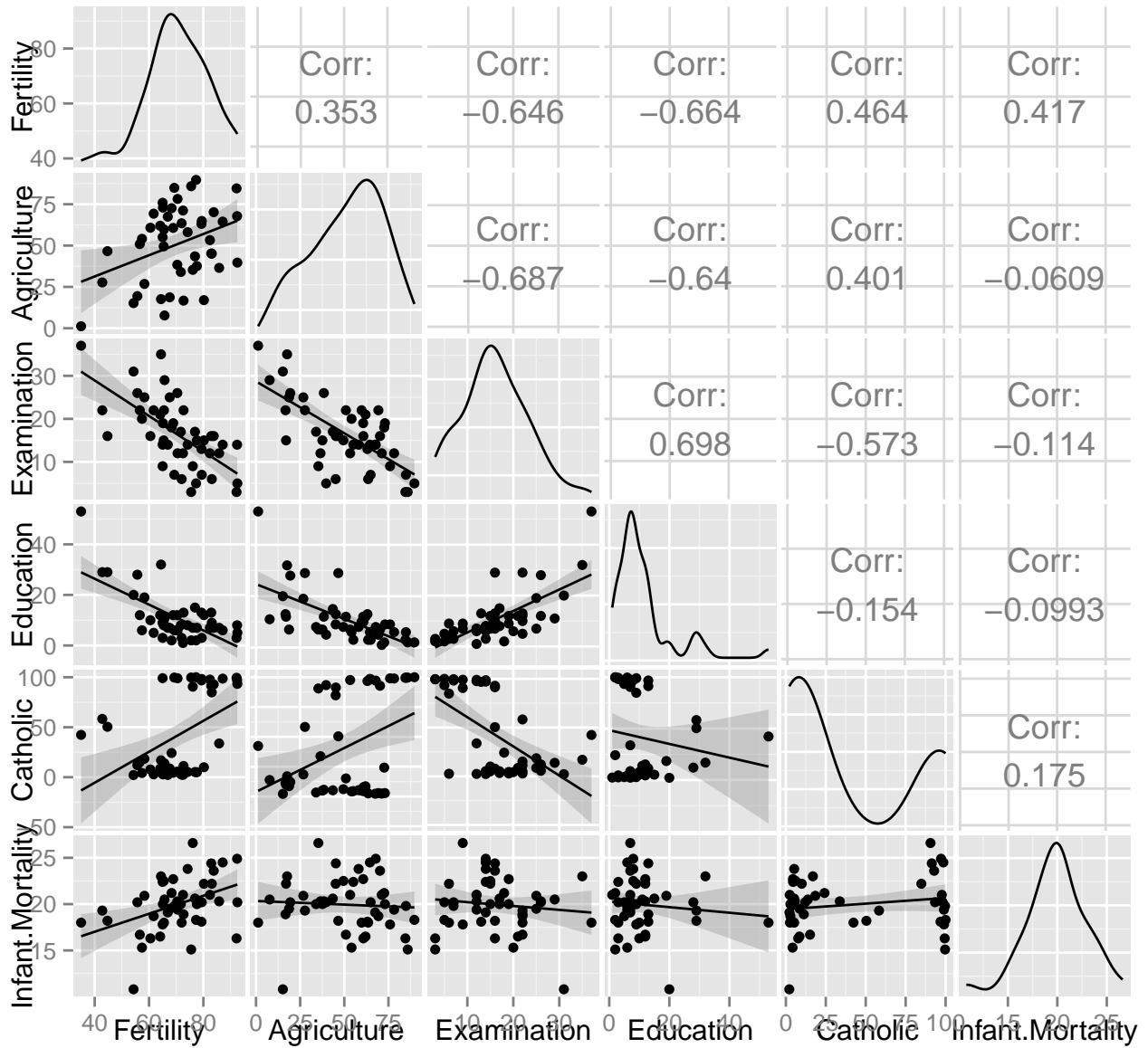
```
## (Intercept)          x          x2          x3
##  0.9985873   1.0048083   0.9930447   1.0014737
```

### Example: Coefficients that Reverse Signs

- **Note:** more information can be found at ‘?swiss’
- data set is composed of standardized fertility measure and socio-economic indicators for each of 47 French-speaking provinces of Switzerland at about 1888
- data frame has 47 observations on 6 variables, each of which is in percent [0, 100]
  - **Fertility** = common standardized fertility measure → outcome
  - **Agriculture** = % of males involved in agriculture as occupation
  - **Examination** = % draftees receiving highest mark on army examination
  - **Education** = % education beyond primary school for draftees
  - **Catholic** = % catholic vs protestant
  - **Infant.Mortality** = live births who live less than 1 year

- [GGally package] `ggpairs(data)` = produces pair wise plot for the predictors similar to `pairs` in base package

```
# load dataset
require(datasets); data(swiss); require(GGally)
# produce pairwise plot using ggplot2
ggpairs(swiss, lower = list(continuous = "smooth"), params = c(method = "loess"))
```



```
# print coefficients of regression of fertility on all predictors
summary(lm(Fertility ~ . , data = swiss))$coefficients
```

	Estimate	Std. Error	t value	Pr(> t )
## (Intercept)	66.9151817	10.70603759	6.250229	1.906051e-07
## Agriculture	-0.1721140	0.07030392	-2.448142	1.872715e-02
## Examination	-0.2580082	0.25387820	-1.016268	3.154617e-01

```

## Education      -0.8709401  0.18302860 -4.758492 2.430605e-05
## Catholic       0.1041153  0.03525785  2.952969 5.190079e-03
## Infant.Mortality 1.0770481  0.38171965  2.821568 7.335715e-03

```

- interpretation for Agriculture coefficient
  - we expect an -0.17 **decrease** in standardized **fertility** for every 1% increase in percentage of **males involved in agriculture** in holding the remaining variables constant
  - since the p-value is 0.0187272, the t-test for  $H_0 : \beta_{Agri} = 0$  versus  $H_a : \beta_{Agri} \neq 0$  is **significant**
- however, if we look at the unadjusted estimate (marginal regression) for the coefficient for Agriculture

```

# run marginal regression on Agriculture
summary(lm(Fertility ~ Agriculture, data = swiss))$coefficients

```

```

##             Estimate Std. Error   t value   Pr(>|t|)    
## (Intercept) 60.3043752 4.25125562 14.185074 3.216304e-18
## Agriculture  0.1942017 0.07671176  2.531577 1.491720e-02

```

- interpretation for Agriculture coefficient
  - we expect an 0.19 **increase** in standardized **fertility** for every 1% increase in percentage of **males involved in agriculture** in holding the remaining variables constant
    - Note: the coefficient flipped signs*
  - since the p-value is 0.0149172, the t-test for  $H_0 : \beta_{Agri} = 0$  versus  $H_a : \beta_{Agri} \neq 0$  is **significant**
- to see intuitively how a **sign change** is possible, we can look at the following simulated example

```

# simulate data
n <- 100; x2 <- 1 : n; x1 <- .01 * x2 + runif(n, -.1, .1); y = -x1 + x2 + rnorm(n, sd = .01)
# print coefficients
c("with x1" = summary(lm(y ~ x1))$coef[2,1],
  "with x1 and x2" = summary(lm(y ~ x1 + x2))$coef[2,1])

##           with x1 with x1 and x2
## 93.709372      -1.019861

# print p-values
c("with x1" = summary(lm(y ~ x1))$coef[2,4],
  "with x1 and x2" = summary(lm(y ~ x1 + x2))$coef[2,4])

##           with x1 with x1 and x2
## 1.893991e-66  5.505244e-84

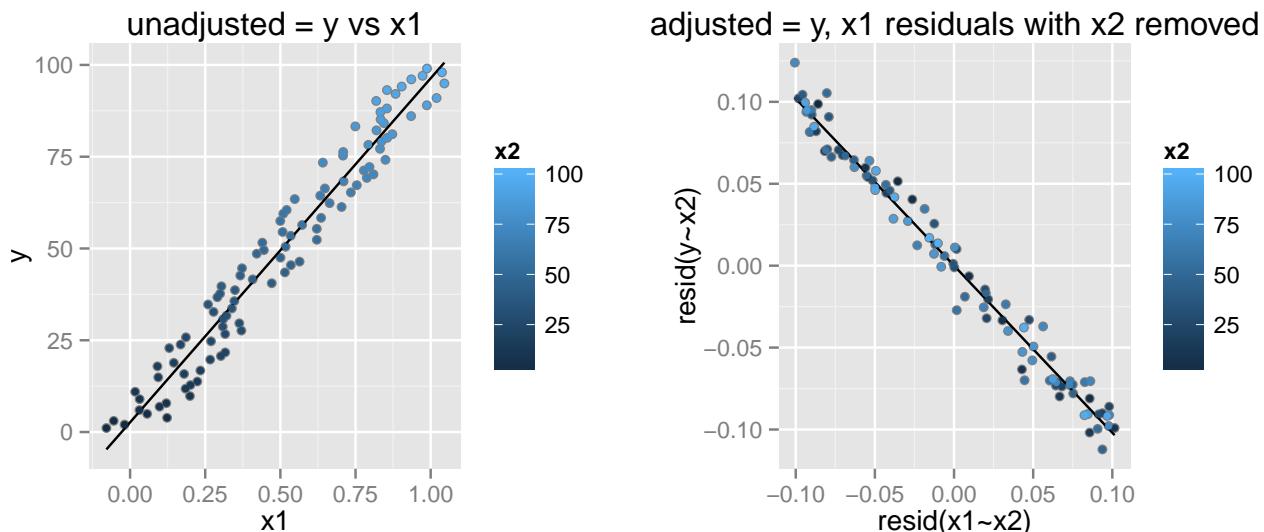
# store all data in one data frame (ey and ex1 are residuals with respect to x2)
dat <- data.frame(y = y, x1 = x1, x2 = x2, ey = resid(lm(y ~ x2)), ex1 = resid(lm(x1 ~ x2)))
# plot y vs x1
g <- ggplot(dat, aes(y = y, x = x1, colour = x2)) +
  geom_point(colour="grey50", size = 2) +
  geom_smooth(method = lm, se = FALSE, colour = "black") + geom_point(size = 1.5) +
  ggtitle("unadjusted = y vs x1")
# plot residual of y adjusted for x2 vs residual of x1 adjusted for x2

```

```

g2 <- ggplot(dat, aes(y = ey, x = ex1, colour = x2)) +
  geom_point(colour="grey50", size = 2) +
  geom_smooth(method = lm, se = FALSE, colour = "black") + geom_point(size = 1.5) +
  ggttitle("adjusted = y, x1 residuals with x2 removed") + labs(x = "resid(x1~x2)",
  y = "resid(y~x2)")
# combine plots
multiplot(g, g2, cols = 2)

```



- as we can see from above, the correlation between  $y$  and  $x_1$  flips signs when adjusting for  $x_2$
- this effectively means that within each consecutive group/subset of points (each color gradient) on the left hand plot (unadjusted), there exists a negative relationship between the points while the overall trend is going up
- ***going back to the swiss data set***, the sign of the coefficient for Agriculture *reverses* itself with the inclusion of Examination and Education (both are ***negatively correlated*** with Agriculture)
  - correlation between Agriculture and Education: -0.64
  - correlation between Agriculture and Examination: -0.69
  - correlation between Education and Examination: 0.7
  - \* this means that the two variables are likely to be measuring the same things
- **Note:** it is **difficult to interpret** and determine which one is the correct model -> one **should not** claim positive correlation between Agriculture and Fertility simply based on marginal regression  $lm(Fertility \sim Agriculture, data=swiss)$

### Example: Unnecessary Variables

- **unnecessary predictors** = variables that don't provide any new linear information, meaning that the variable are simply ***linear combinations*** (multiples, sums) of other predictors/variables
- when running a linear regression with unnecessary variables, R automatically drops the linear combinations and returns **NA** as their coefficients

```
# add a linear combination of agriculture and education variables
z <- swiss$Agriculture + swiss$Education
# run linear regression with unnecessary variables
lm(Fertility ~ . + z, data = swiss)$coef
```

```
##          (Intercept)      Agriculture   Examination     Education
##       66.9151817     -0.1721140    -0.2580082    -0.8709401
##      Catholic Infant.Mortality           z
##      0.1041153      1.0770481        NA
```

- as we can see above, the R dropped the unnecessary variable *z* by excluding it from the linear regression  
-> *z*'s coefficient is NA

## Dummy Variables

- **dummy variables** = binary variables that take on value of **1** when the measurement is in a particular group, and **0** when the measurement is not (i.e. in clinical trials, treated = 1, untreated = 1)
- in linear model form,

$$Y_i = \beta_0 + X_{i1}\beta_1 + \epsilon_i$$

where  $X_{i1}$  is a binary/dummy variable so that it is a 1 if measurement  $i$  is in a group and 0 otherwise

- for people in the group, the mean or  $\mu_{X_{i1}=1} = E[Y_i] = \beta_0 + \beta_1$
- for people **not** in the group, the mean or  $\mu_{X_{i1}=0} = E[Y_i] = \beta_0$
- predicted mean for group =  $\hat{\beta}_0 + \hat{\beta}_1$
- predicted mean for not in group =  $\hat{\beta}_0$
- coefficient  $\beta_1$  for  $X_{i1}$  is interpreted as the **increase or decrease** in the **mean** when comparing two groups (in vs not)
- **Note:** including a dummy variable that is 1 for not in the group would be **redundant** as it would simply be a linear combination  $1 - X_{i1}$

## More Than 2 Levels

- for 3 factor levels, we would need 2 dummy variables and the model would be

$$Y_i = \beta_0 + X_{i1}\beta_1 + X_{i2}\beta_2 + \epsilon_i$$

- for this example, we will use the above model to analyze US political party affiliations (Democrats vs Republicans vs independents) and denote the variables as follows:

- $X_{i1} = 1$  for Republicans and 0 otherwise
- $X_{i2} = 1$  for Democrats and 0 otherwise
- If  $i$  is Republican,  $X_{i1} = 1, X_{i2} = 0, E[Y_i] = \beta_0 + \beta_1$
- If  $i$  is Democrat,  $X_{i1} = 0, X_{i2} = 1, E[Y_i] = \beta_0 + \beta_2$
- If  $i$  is independent,  $X_{i1} = 0, X_{i2} = 0, E[Y_i] = \beta_0$
- $\beta_1$  compares Republicans to independents
- $\beta_2$  compares Democrats to independents
- $\beta_1 - \beta_2$  compares Republicans to Democrats

- **Note:** choice of reference category (independent in this case) changes the interpretation

- **reference category** = the group whose binary variable has been eliminated

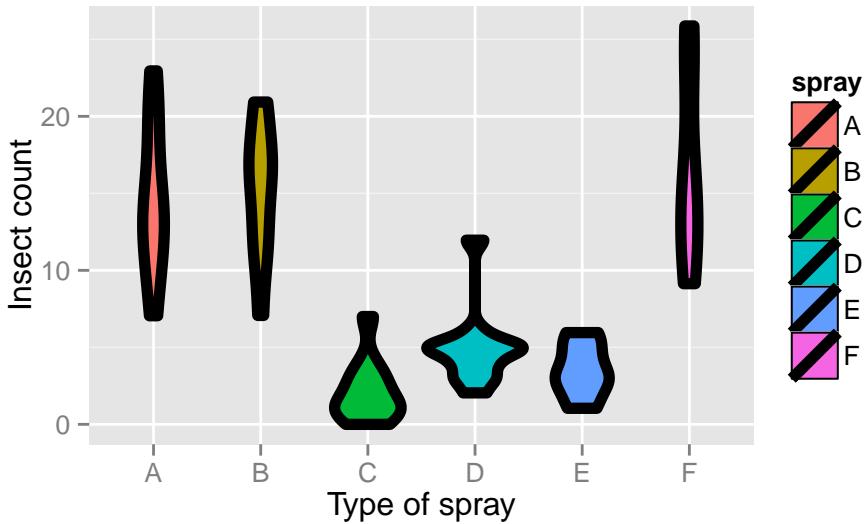
- the same principles explained above can be expanded to  $p$  level model

$$Y_i = \beta_0 + X_{i1}\beta_1 + X_{i2}\beta_2 + \dots + X_{ip}\beta_p + \epsilon_i$$

## Example: 6 Factor Level Insect Spray Data

- below is a violin plot of the 6 different types (A, B, C, D, E, and F) of insect sprays and their potency (kill count) from **InsectSprays** data set
  - **Note:** the varying width of each bar indicates the density of measurement at each value

```
# load insect spray data
data(InsectSprays)
ggplot(data = InsectSprays, aes(y = count, x = spray, fill = spray)) +
  geom_violin(colour = "black", size = 2) + xlab("Type of spray") +
  ylab("Insect count")
```



Linear model fit with group A as reference category

```
# linear fit with 5 dummy variables
summary(lm(count ~ spray, data = InsectSprays))$coefficients
```

```
##             Estimate Std. Error    t value   Pr(>|t|)
## (Intercept) 14.500000  1.132156 12.8074279 1.470512e-19
## sprayB      0.8333333  1.601110  0.5204724 6.044761e-01
## sprayC     -12.4166667  1.601110 -7.7550382 7.266893e-11
## sprayD     -9.5833333  1.601110 -5.9854322 9.816910e-08
## sprayE     -11.0000000  1.601110 -6.8702352 2.753922e-09
## sprayF      2.1666667  1.601110  1.3532281 1.805998e-01
```

- *Note:* R automatically converts factor variables into  $n - 1$  dummy variables and uses the first category as reference
  - mean of group A is therefore the default intercept
- the above coefficients can be interpreted as the difference in means between each group (B, C, D, E, and F) and group A (the intercept)
  - example: the mean of group B is 0.83 higher than the mean of group A, which is 14.5
  - means for group B/C/D/E/F = the intercept + their respective coefficient
- all t-tests are for comparisons of Sprays versus Spray A

### Hard-coding the dummy variables

- this produces the exact same result as the command `lm(count ~ spray, data = InsectSprays)`

```
# hard coding dummy variables
lm(count ~ I(1 * (spray == 'B')) + I(1 * (spray == 'C')) +
  I(1 * (spray == 'D')) + I(1 * (spray == 'E')) +
  I(1 * (spray == 'F')), data = InsectSprays)$coefficients
```

```

##           (Intercept) I(1 * (spray == "B")) I(1 * (spray == "C"))
##             14.5000000          0.8333333         -12.4166667
## I(1 * (spray == "D")) I(1 * (spray == "E")) I(1 * (spray == "F"))
##            -9.5833333        -11.0000000         2.1666667

```

### Linear model fit with all 6 categories

```

# linear fit with 6 dummy variables
lm(count ~ I(1 * (spray == 'B')) + I(1 * (spray == 'C')) +
   I(1 * (spray == 'D')) + I(1 * (spray == 'E')) +
   I(1 * (spray == 'F')) + I(1 * (spray == 'A')) ,
  data = InsectSprays)$coefficients

##           (Intercept) I(1 * (spray == "B")) I(1 * (spray == "C"))
##             14.5000000          0.8333333         -12.4166667
## I(1 * (spray == "D")) I(1 * (spray == "E")) I(1 * (spray == "F"))
##            -9.5833333        -11.0000000         2.1666667
## I(1 * (spray == "A"))
##             NA

```

- as we can see from above, the coefficient for group A is NA
- this is because  $X_{iA} = 1 - X_{iB} - X_{iC} - X_{iD} - X_{iE} - X_{iF}$ , or the dummy variable for A is a linear combination of the rest of the dummy variables

### Linear model fit with omitted intercept

- eliminating the intercept would mean that each group is compared to the value 0, which would yield 6 variables since A is no longer the reference category
- this means that the coefficients for the 6 variables are simply the **mean** of each group
  - when  $X_{iA} = 1$ , all the other dummy variables become 0, which means the linear model becomes

$$Y_i = \beta_A + \epsilon_i$$

- then  $E[Y_i] = \beta_A = \mu_A$
- this makes sense because the **best prediction** for the kill count of spray of type A is the **mean** of recorded kill counts of A spray

```

# linear model with omitted intercept
summary(lm(count ~ spray - 1, data = InsectSprays))$coefficients

```

```

##           Estimate Std. Error    t value Pr(>|t|)
## sprayA  14.500000  1.132156 12.807428 1.470512e-19
## sprayB  15.333333  1.132156 13.543487 1.001994e-20
## sprayC   2.083333  1.132156  1.840148 7.024334e-02
## sprayD   4.916667  1.132156  4.342749 4.953047e-05
## sprayE   3.500000  1.132156  3.091448 2.916794e-03
## sprayF  16.666667  1.132156 14.721181 1.573471e-22

```

```

# actual means of count by each variable
round(tapply(InsectSprays$count, InsectSprays$spray, mean), 2)

```

```
##      A      B      C      D      E      F
## 14.50 15.33  2.08  4.92  3.50 16.67
```

- all t-tests are for whether the groups are different than zero (i.e. are the expected counts 0 for that spray?)
- to compare between different categories, say B vs C, we can simply subtract the coefficients
- to reorient the model with other groups as reference categories, we can simply **reorder the levels** for the factor variable
  - `relevel(var, "l")` = reorders the factor levels within the factor variable `var` such that the specified level “l” is the reference/base/lowest level
    - \* *Note: R automatically uses the first/base level as the reference category during a linear regression*

```
# reorder the levels of spray variable such that C is the lowest level
spray2 <- relevel(InsectSprays$spray, "C")
# rerun linear regression with relevelled factor
summary(lm(count ~ spray2, data = InsectSprays))$coef
```

```
##              Estimate Std. Error   t value    Pr(>|t|) 
## (Intercept) 2.083333  1.132156 1.840148 7.024334e-02
## spray2A     12.416667  1.601110 7.755038 7.266893e-11
## spray2B     13.250000  1.601110 8.275511 8.509776e-12
## spray2D     2.833333  1.601110 1.769606 8.141205e-02
## spray2E     1.416667  1.601110 0.884803 3.794750e-01
## spray2F     14.583333  1.601110 9.108266 2.794343e-13
```

- it is important to note in this example that
  - counts are bounded from below by 0 → **violates** the assumption of normality of the errors
  - there are counts near zero → **violates** intent of the assumption → not acceptable in assuming normal distribution
  - variance does not appear to be constant across different type of groups → violates assumption
    - \* taking  $\log(\text{counts}) + 1$  may help (+1 since there are the zero values)
  - **Poisson GLMs** are better (don’t have to worry about the assumptions) for fitting count data

## Interactions

- **interactions** between variables can be added to a regression model to test how the outcomes change under different conditions
- we will use the data set from the Millennium Development Goal from the UN which can be found [here](#)
  - **Numeric** = values for children aged <5 years underweight (%)
  - **Sex** = records whether
  - **Year** = year when data was recorded
  - **Income** = income for the child's parents

```
# load in hunger data
hunger <- read.csv("hunger.csv")
# exclude the data with "Both Sexes" as values (only want Male vs Female)
hunger <- hunger[hunger$Sex!="Both sexes", ]
# structure of data
str(hunger)
```

```
## 'data.frame': 948 obs. of 12 variables:
##   $ Indicator      : Factor w/ 1 level "Children aged <5 years underweight (%)": 1 1 1 1 1 1 1 1 1 1 ...
##   $ Data.Source    : Factor w/ 670 levels "NLIS_310005",...: 7 52 519 380 548 551 396 503 643 632 ...
##   $ PUBLISH.STATE: Factor w/ 1 level "Published": 1 1 1 1 1 1 1 1 1 1 ...
##   $ Year          : int 1986 1990 2005 2002 2008 2008 2003 2006 2012 1999 ...
##   $ WHO.region    : Factor w/ 6 levels "Africa","Americas",...: 1 2 2 3 1 1 1 2 1 2 ...
##   $ Country        : Factor w/ 151 levels "Afghanistan",...: 115 104 97 69 57 54 71 13 115 144 ...
##   $ Sex            : Factor w/ 3 levels "Both sexes","Female",...: 3 3 3 2 2 3 3 3 3 2 ...
##   $ Display.Value : num 19.3 2.2 5.3 3.2 17 15.7 19.3 4 15.5 4.2 ...
##   $ Numeric        : num 19.3 2.2 5.3 3.2 17 15.7 19.3 4 15.5 4.2 ...
##   $ Low             : logi NA NA NA NA NA NA ...
##   $ High            : logi NA NA NA NA NA NA ...
##   $ Comments       : logi NA NA NA NA NA NA ...
```

### Model: % Hungry ~ Year by Sex

- this will include 2 models with 2 separate lines
- model for % hungry ( $H_F$ ) vs year ( $Y_F$ ) for females is

$$H_{Fi} = \beta_{F0} + \beta_{F1}Y_{Fi} + \epsilon_{Fi}$$

- $\beta_{F0}$  = % of females hungry at year 0
- $\beta_{F1}$  = decrease in % females hungry per year
- $\epsilon_{Fi}$  = standard error (or everything we didn't measure)
- model for % hungry ( $H_M$ ) vs year ( $Y_M$ ) for males is

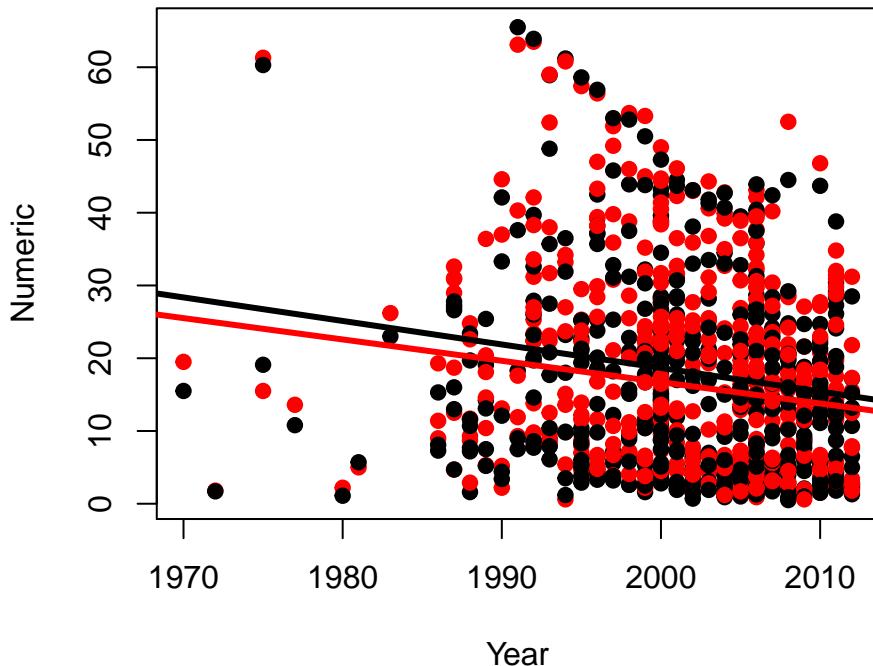
$$H_{Mi} = \beta_{M0} + \beta_{M1}Y_{Mi} + \epsilon_{Mi}$$

- $\beta_{M0}$  = % of males hungry at year 0
- $\beta_{M1}$  = decrease in % males hungry per year
- $\epsilon_{Mi}$  = standard error (or everything we didn't measure)
- each line has **different** residuals, standard errors, and variances
- **Note:**  $\beta_{F0}$  and  $\beta_{M0}$  are the interpolated intercept at year 0, which does hold any interpretable value for the model

- it's possible to subtract the model by a meaningful value (% hungry at 1970, or average), which moves the intercept of the lines to something interpretable

- **Note:** we are also assuming the error terms  $\epsilon_{Fi}$  and  $\epsilon_{Mi}$  are Gaussian distributions  $\rightarrow$  mean = 0

```
# run linear model with Numeric vs Year for male and females
male.fit <- lm(Numeric ~ Year, data = hunger[hunger$Sex == "Male", ])
female.fit <- lm(Numeric ~ Year, data = hunger[hunger$Sex == "Female", ])
# plot % hungry vs the year
plot(Numeric ~ Year, data = hunger, pch = 19, col=(Sex=="Male")*1+1)
# plot regression lines for both
abline(male.fit, lwd = 3, col = "black")
abline(female.fit, lwd = 3, col = "red")
```



#### Model: % Hungry ~ Year + Sex (Binary Variable)

- this will include 1 model with 2 separate lines with the *same* slope
- model for % hungry ( $H$ ) vs year ( $Y$ ) and dummy variable for sex ( $X$ ) is

$$H_i = \beta_0 + \beta_1 X_i + \beta_2 Y_i + \epsilon_i^*$$

- $\beta_0$  = % of females hungry at year 0
- $\beta_0 + \beta_1$  = % of males hungry at year 0

- \* **Note:** the term  $\beta_1 X_i$  is effectively an **adjustment** for the intercept for males and DOES NOT alter the slope in anyway

- \*  $\beta_1$  = difference in means of males vs females

- $\beta_2$  = decrease in % hungry (males and females) per year

- \* this means that the slope is **constant** for both females and males

- $\epsilon_i^*$  = standard error (or everything we didn't measure)

- \* we are still assuming Gaussian error term

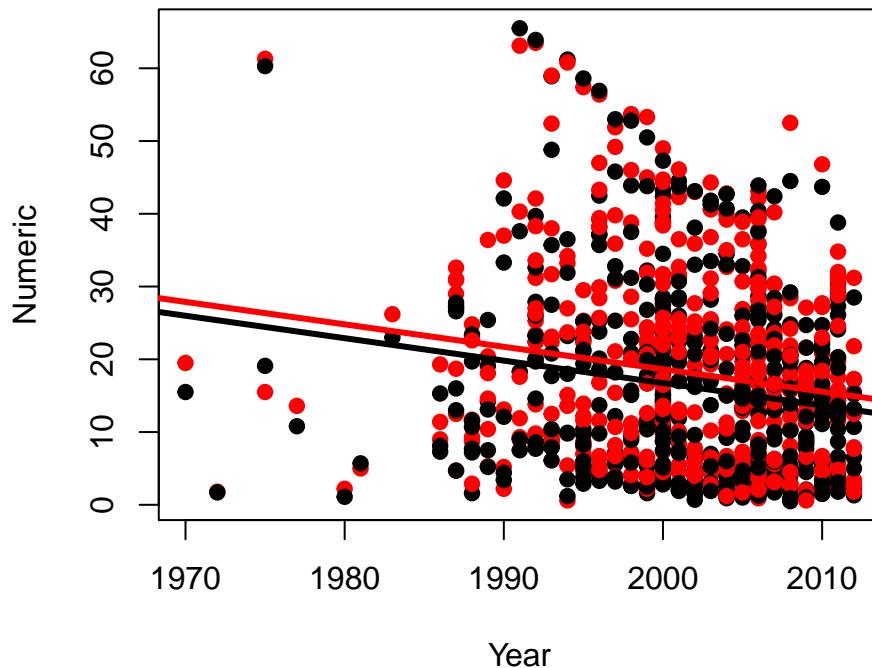
- `abline(intercept, slope)` = adds a line to the existing plot based on the intercept and slope provided
  - `abline(lm)` = plots the linear regression line on the plot

```
# run linear model with Numeric vs Year and Sex
both.fit <- lm(Numeric ~ Year+Sex, data = hunger)
# print fit
```

```
both.fit$coef
```

```
## (Intercept)      Year      SexMale
##  633.528289   -0.308397    1.902743
```

```
# plot % hungry vs the year
plot(Numeric ~ Year, data = hunger, pch = 19, col=(Sex=="Male")*1+1)
# plot regression lines for both (same slope)
abline(both.fit$coef[1], both.fit$coef[2], lwd = 3, col = "black")
abline(both.fit$coef[1]+both.fit$coef[3], both.fit$coef[2], lwd = 3, col = "red")
```



**Model: % Hungry ~ Year + Sex + Year \* Sex (Binary Interaction)**

- this will include 1 model with an interaction term with binary variable, which produces 2 lines with *different* slopes
- we can introduce an interaction term to the previous model to capture the different slopes between males and females
- model for % hungry ( $H$ ) vs year ( $Y$ ), sex ( $X$ ), and interaction between year and sex ( $Y \times X$ ) is

$$H_i = \beta_0 + \beta_1 X_i + \beta_2 Y_i + \beta_3 X_i Y_i + \epsilon_i^+$$

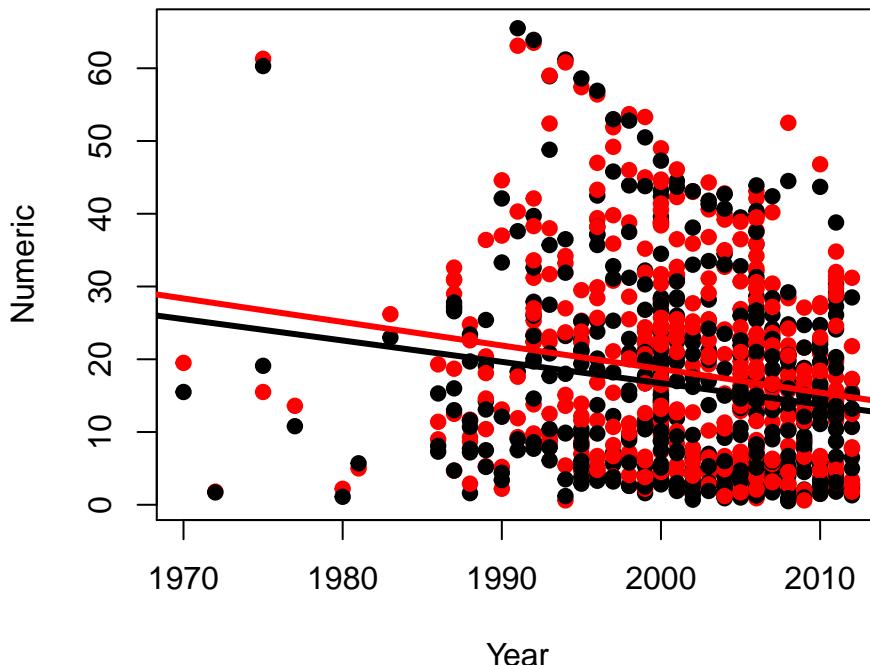
- $\beta_0$  = % of females hungry at year 0
- $\beta_0 + \beta_1$  = % of males hungry at year 0
- \*  $\beta_1$  = change in *intercept* for males

- $\beta_2$  = decrease in % hungry (females) per year
- $\beta_2 + \beta_3$  = decrease in % hungry (males) per year
  - \*  $\beta_3$  = change in *slope* for males
- $\epsilon_i^+$  = standard error (or everything we didn't measure)
- expected value for males is  $E[H_i]_M = (\beta_0 + \beta_1) + (\beta_2 + \beta_3)Y_i$
- expected value for females is  $E[H_i]_F = \beta_0 + \beta_2 Y_i$ 
  - $\beta_1$  and  $\beta_3$  are effectively adjusting the intercept and slope for males
- `lm(outcome ~ var1*var2)` = whenever an interaction is specified in `lm` function using the `*` operator, the individual terms are added automatically
  - `lm(outcome ~ var1+var2+var1*var2)` = builds the exact same model
  - `lm(outcome ~ var1:var2)` = builds linear model with *only* the interaction term (specified by `:` operator)

```
# run linear model with Numeric vs Year and Sex and interaction term
interaction.fit <- lm(Numeric ~ Year*Sex, data = hunger)
# print fit
interaction.fit$coef
```

```
## (Intercept)          Year       SexMale Year:SexMale
## 603.50579986 -0.29339638  61.94771998 -0.03000132
```

```
# plot % hungry vs the year
plot(Numeric ~ Year, data = hunger, pch = 19, col=(Sex=="Male")*1+1)
# plot regression lines for both (different slope)
abline(interaction.fit$coef[1], interaction.fit$coef[2], lwd = 3, col = "black")
abline(interaction.fit$coef[1]+interaction.fit$coef[3],
      interaction.fit$coef[2]+interaction.fit$coef[4], lwd = 3, col = "red")
```



### Example: % Hungry ~ Year + Income + Year \* Income (Continuous Interaction)

- this will include 1 model with an interaction term with continuous variable, which produces a curve through the plot
- for **continuous interactions** (two continuous variables) with model

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{1i}X_{2i} + \epsilon_i$$

the expected value for a given set of values  $x_1$  and  $x_2$  is defined as

$$E[Y_i|X_{1i} = x_1, X_{2i} = x_2] = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$$

- holding  $X_2$  constant and varying  $X_1$  by 1, we have

$$\begin{aligned} \frac{\partial Y_i}{\partial X_{1i}} &= E[Y_i|X_{1i} = x_1 + 1, X_{2i} = x_2] - E[Y_i|X_{1i} = x_1, X_{2i} = x_2] \\ &= \beta_0 + \beta_1(x_1 + 1) + \beta_2 x_2 + \beta_3(x_1 + 1)x_2 - [\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2] \\ &= \beta_1 + \beta_3 x_2 \end{aligned}$$

- **Note:** this means that slope for  $X_{1i}$  **not** a constant and is **dependent** on  $X_{2i}$
- $\beta_1$  is the slope for  $X_{1i}$  when  $X_{2i} = 0$

- by the same logic, if we vary  $X_1$  by 1 and find the change, and vary  $X_2$  by 1 and find the change, we get

$$\begin{aligned} \frac{\partial}{\partial X_{2i}} \left( \frac{\partial Y_i}{\partial X_{1i}} \right) &= E[Y_i|X_{1i} = x_1 + 1, X_{2i} = x_2 + 1] - E[Y_i|X_{1i} = x_1, X_{2i} = x_2 + 1] \\ &\quad - (E[Y_i|X_{1i} = x_1 + 1, X_{2i} = x_2] - E[Y_i|X_{1i} = x_1, X_{2i} = x_2]) \\ &= \beta_0 + \beta_1(x_1 + 1) + \beta_2(x_2 + 1) + \beta_3(x_1 + 1)(x_2 + 1) - [\beta_0 + \beta_1 x_1 + \beta_2(x_2 + 1) + \beta_3 x_1(x_2 + 1)] \\ &\quad - (\beta_0 + \beta_1(x_1 + 1) + \beta_2 x_2 + \beta_3(x_1 + 1)x_2 - [\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2]) \\ &= \beta_3 \end{aligned}$$

- this can be interpreted as  $\beta_3$  = the **expected change in  $Y$  per unit change in  $X_1$  per unit change of  $X_2$**
- in other words,  $\beta_3$  = the change in slope of  $X_1$  per unit change of  $X_2$

- coming back to the hunger data, model for % hungry ( $H$ ) vs year ( $Y$ ), income ( $I$ ), and interaction between year and income ( $Y \times I$ ) is

$$H_i = \beta_0 + \beta_1 I_i + \beta_2 Y_i + \beta_3 I_i Y_i + \epsilon_i^+$$

- $\beta_0$  = % hungry children (whose parents have no income) at year 0
- $\beta_1$  = change in % hungry children for **each dollar in income** in year 0
- $\beta_2$  = change in % hungry children (whose parents have no income) **per year**
- $\beta_3$  = change in % hungry children **per year** and for **each dollar in income**
  - \* if income is \$10,000, then the change in % hungry children **per year** will be  $\beta_1 - 10000 \times \beta_3$
- $\epsilon_i^+$  = standard error (or everything we didn't measure)

- **Note:** much care needs to be taken when interpreting these coefficients

```
# generate some income data
hunger$Income <- 1:nrow(hunger)*10 + 500*runif(nrow(hunger), 0, 10) +
  runif(nrow(hunger), 0, 500)^1.5
# run linear model with Numeric vs Year and Income and interaction term
lm(Numeric ~ Year*Income, data = hunger)$coef
```

```
##   (Intercept)      Year      Income  Year:Income
##  8.308745e+02 -4.072087e-01 -1.669858e-02  8.401616e-06
```

## Multivariable Simulation

- we will generate a series of simulated data so that we know the true relationships, and then run linear regressions to interpret and compare the results to truth
- **treatment effect** = effect of adding the treatment variable  $t$  to the regression model (i.e. how adding  $t$  changes the regression lines)
  - effectively measures how much the regression lines for the two groups separate with regression  $\text{lm}(y \sim x + t)$
- **adjustment effect** = adjusting the regression for effects of  $x$  such that we just look at how  $t$  is *marginally related* to  $Y$ 
  - ignore all variation of  $x$  and simply look at the group means of  $t = 1$  vs  $t = 0$

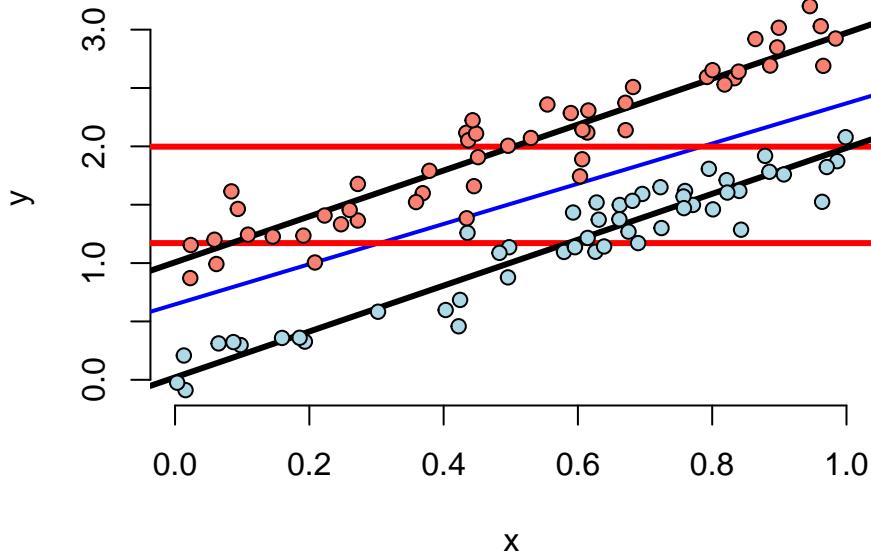
### Simulation 1 - Treatment = Adjustment Effect

- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 t_i + \epsilon_i$$

where  $t = 0, 1 \rightarrow$  binary variable

```
# simulate data
n <- 100; t <- rep(c(0, 1), c(n/2, n/2)); x <- c(runif(n/2), runif(n/2));
# define parameters/coefficients
beta0 <- 0; beta1 <- 2; beta2 <- 1; sigma <- .2
# generate outcome using linear model
y <- beta0 + x * beta1 + t * beta2 + rnorm(n, sd = sigma)
# set up axes
plot(x, y, type = "n", frame = FALSE)
# plot linear fit of y vs x
abline(lm(y ~ x), lwd = 2, col = "blue")
# plot means of the two groups (t = 0 vs t = 1)
abline(h = mean(y[1 : (n/2)]), lwd = 3, col = "red")
abline(h = mean(y[(n/2 + 1) : n]), lwd = 3, col = "red")
# plot linear fit of y vs x and t
fit <- lm(y ~ x + t)
# plot the two lines corresponding to (t = 0 vs t = 1)
abline(coef(fit)[1], coef(fit)[2], lwd = 3)
abline(coef(fit)[1] + coef(fit)[3], coef(fit)[2], lwd = 3)
# add in the actual data points
points(x[1 : (n/2)], y[1 : (n/2)], pch = 21, col = "black", bg = "lightblue", cex = 1)
points(x[(n/2 + 1) : n], y[(n/2 + 1) : n], pch = 21, col = "black", bg = "salmon", cex = 1)
```



```
# print treatment and adjustment effects
rbind("Treatment Effect" = lm(y~t+x)$coef[2], "Adjustment Effect" = lm(y~t)$coef[2])
```

```
##                                     t
## Treatment Effect  0.9860302
## Adjustment Effect 0.8261934
```

- in the above graph, the elements are as follows:
  - red* lines = means for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t)$
  - black* lines = regression lines for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t + x)$
  - blue* line = overall regression of  $y$  vs  $x$   $\rightarrow$  line representing  $\text{lm}(y \sim x)$
- from the graph, we can see that
  - $x$  variable is **unrelated** to group status  $t$ 
    - \* distribution of each group (salmon vs light blue) of  $Y$  vs  $X$  is effectively the same
  - $x$  variable is **related** to  $Y$ , but the intercept **depends** on group status  $t$
  - group variable  $t$  is **related** to  $Y$ 
    - \* relationship between  $t$  and  $Y$  disregarding  $x \approx$  the same as holding  $x$  constant
    - \* difference in group means  $\approx$  difference in regression lines
    - \* **treatment effect** (difference in regression lines)  $\approx$  **adjustment effect** (difference in group means)

### Simulation 2 - No Treatment Effect

- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 t_i + \epsilon_i$$

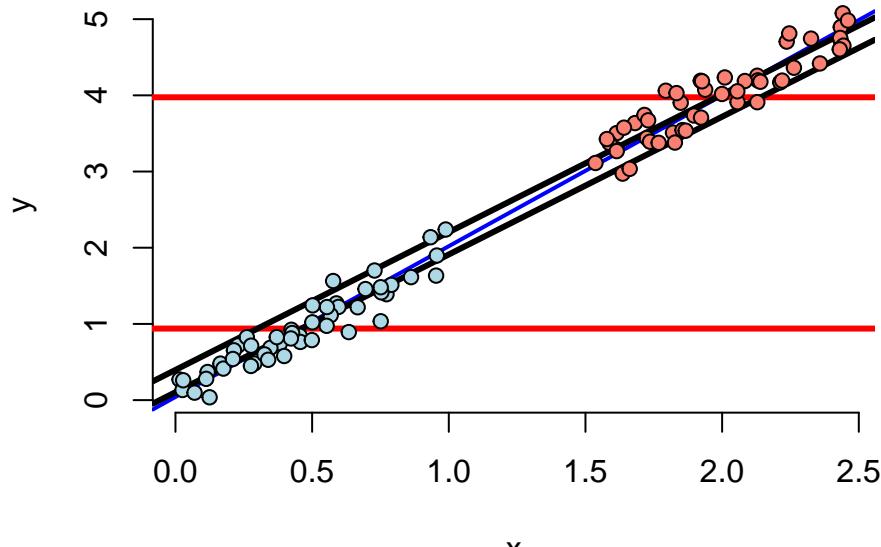
where  $t = \{0, 1\} \rightarrow$  binary variable

- in this case,  $\beta_2$  is set to 0

```

# simulate data
n <- 100; t <- rep(c(0, 1), c(n/2, n/2)); x <- c(runif(n/2), 1.5 + runif(n/2));
# define parameters/coefficients
beta0 <- 0; beta1 <- 2; beta2 <- 0; sigma <- .2
# generate outcome using linear model
y <- beta0 + x * beta1 + t * beta2 + rnorm(n, sd = sigma)
# set up axes
plot(x, y, type = "n", frame = FALSE)
# plot linear fit of y vs x
abline(lm(y ~ x), lwd = 2, col = "blue")
# plot means of the two groups (t = 0 vs t = 1)
abline(h = mean(y[1 : (n/2)]), lwd = 3, col = "red")
abline(h = mean(y[(n/2 + 1) : n]), lwd = 3, col = "red")
# plot linear fit of y vs x and t
fit <- lm(y ~ x + t)
# plot the two lines corresponding to (t = 0 vs t = 1)
abline(coef(fit)[1], coef(fit)[2], lwd = 3)
abline(coef(fit)[1] + coef(fit)[3], coef(fit)[2], lwd = 3)
# add in the actual data points
points(x[1 : (n/2)], y[1 : (n/2)], pch = 21, col = "black", bg = "lightblue", cex = 1)
points(x[(n/2 + 1) : n], y[(n/2 + 1) : n], pch = 21, col = "black", bg = "salmon", cex = 1)

```



```

# print treatment and adjustment effects
rbind("Treatment Effect" = lm(y~t+x)$coef[2], "Adjustment Effect" = lm(y~t)$coef[2])

```

```

##          t
## Treatment Effect  0.2927632
## Adjustment Effect 3.0353627

```

- in the above graph, the elements are as follows:

- *red* lines = means for two groups ( $t = 0$  vs  $t = 1$ ) -> two lines representing  $\text{lm}(y \sim t)$
- *black* lines = regression lines for two groups ( $t = 0$  vs  $t = 1$ ) -> two lines representing  $\text{lm}(y \sim t + x)$

- \* in this case, both lines correspond to  $\text{lm}(y \sim x)$  since coefficient of  $t$  or  $\beta_2 = 0$
- *blue* line = overall regression of  $y$  vs  $x \rightarrow$  line representing  $\text{lm}(y \sim x)$ 
  - \* this is overwritten by the *black* lines
- from the graph, we can see that
  - $x$  variable is ***highly related*** to group status  $t$ 
    - \* clear shift in  $x$  with salmon vs light blue groups
  - $x$  variable is ***related*** to  $Y$ , but the intercept ***does not depend*** on group status  $t$ 
    - \* intercepts for both lines are the same
  - $x$  variable shows ***similar relationships*** to  $Y$  for both groups ( $t = 0$  vs  $t = 1$ , or salmon vs lightblue)
    - \* the  $x$  values of the two groups of points both seem to be linearly correlated with  $Y$
  - group variable  $t$  is ***marginally related*** to  $Y$  when disregarding  $X$ 
    - \*  $x$  values capture most of the variation
    - \* ***adjustment effect*** (difference in group means) is very large
  - group variable  $t$  is ***unrelated or has very little*** effect on  $Y$ 
    - \* ***treatment effect*** is very small or non-existent
    - \* ***Note:*** the groups ( $t = 0$  vs  $t = 1$ ) are ***incomparable*** since there is no data to inform the relationship between  $t$  and  $Y$
    - \* the groups (salmon vs lightblue) don't have any overlaps so we have no idea how they behave
    - \* this conclusion is based on the constructed alone

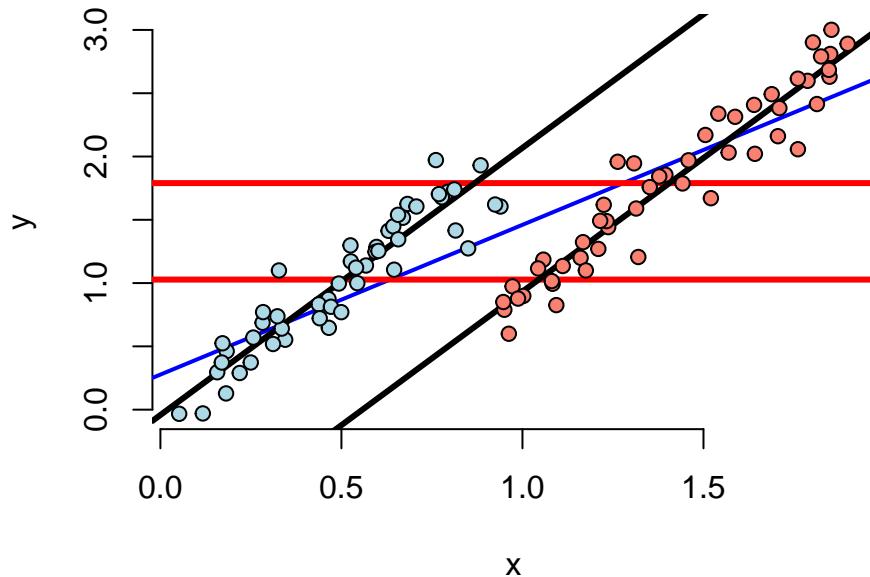
### Simulation 3 - Treatment Reverses Adjustment Effect

- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 t_i + \epsilon_i$$

- where  $t = \{0, 1\} \rightarrow$  binary variable
- in this case,  $\beta_0$  is set to 0  $\rightarrow$  no intercept

```
# simulate data
n <- 100; t <- rep(c(0, 1), c(n/2, n/2)); x <- c(runif(n/2), .9 + runif(n/2));
# define parameters/coefficients
beta0 <- 0; beta1 <- 2; beta2 <- -1; sigma <- .2
# generate outcome using linear model
y <- beta0 + x * beta1 + t * beta2 + rnorm(n, sd = sigma)
# set up axes
plot(x, y, type = "n", frame = FALSE)
# plot linear fit of y vs x
abline(lm(y ~ x), lwd = 2, col = "blue")
# plot means of the two groups (t = 0 vs t = 1)
abline(h = mean(y[1 : (n/2)]), lwd = 3, col = "red")
abline(h = mean(y[(n/2 + 1) : n]), lwd = 3, col = "green")
# plot linear fit of y vs x and t
fit <- lm(y ~ x + t)
# plot the two lines corresponding to (t = 0 vs t = 1)
abline(coef(fit)[1], coef(fit)[2], lwd = 3)
abline(coef(fit)[1] + coef(fit)[3], coef(fit)[2], lwd = 3)
# add in the actual data points
points(x[1 : (n/2)], y[1 : (n/2)], pch = 21, col = "black", bg = "lightblue", cex = 1)
points(x[(n/2 + 1) : n], y[(n/2 + 1) : n], pch = 21, col = "black", bg = "salmon", cex = 1)
```



```
# print treatment and adjustment effects
rbind("Treatment Effect" = lm(y~t+x)$coef[2], "Adjustment Effect" = lm(y~t)$coef[2])
```

```
##                                     t
## Treatment Effect   -1.1354773
## Adjustment Effect  0.7621201
```

- in the above graph, the elements are as follows:
  - *red* lines = means for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t)$
  - *black* lines = regression lines for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t + x)$
  - *blue* line = overall regression of  $y$  vs  $x$   $\rightarrow$  line representing  $\text{lm}(y \sim x)$
- from the graph, we can see that
  - disregarding/adjusting for  $x$ , the mean for salmon group is ***higher*** than the mean of the blue group (***adjustment effect*** is positive)
  - when adding  $t$  into the linear model, the treatment actually reverses the orders of the group  $\rightarrow$  the mean for salmon group is ***lower*** than the mean of the blue group (***treatment effect*** is negative)
  - group variable  $t$  is ***related*** to  $x$
  - some points overlap so it is possible to compare the subsets two groups holding  $x$  fixed

#### Simulation 4 - No Adjustment Effect

- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 t_i + \epsilon_i$$

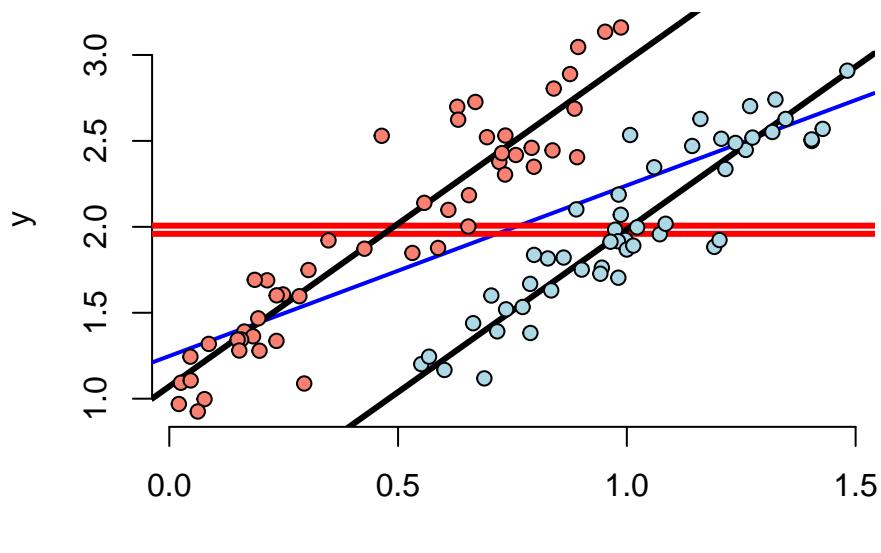
where  $\$t = \{0, 1\} \rightarrow$  binary variable

- in this case,  $\beta_0$  is set to 0  $\rightarrow$  no intercept

```

# simulate data
n <- 100; t <- rep(c(0, 1), n/2, n/2); x <- c(.5 + runif(n/2), runif(n/2));
# define parameters/coefficients
beta0 <- 0; beta1 <- 2; beta2 <- 1; sigma <- .2
# generate outcome using linear model
y <- beta0 + x * beta1 + t * beta2 + rnorm(n, sd = sigma)
# set up axes
plot(x, y, type = "n", frame = FALSE)
# plot linear fit of y vs x
abline(lm(y ~ x), lwd = 2, col = "blue")
# plot means of the two groups (t = 0 vs t = 1)
abline(h = mean(y[1 : (n/2)]), lwd = 3, col = "red")
abline(h = mean(y[(n/2 + 1) : n]), lwd = 3, col = "red")
# plot linear fit of y vs x and t
fit <- lm(y ~ x + t)
# plot the two lines corresponding to (t = 0 vs t = 1)
abline(coef(fit)[1], coef(fit)[2], lwd = 3)
abline(coef(fit)[1] + coef(fit)[3], coef(fit)[2], lwd = 3)
# add in the actual data points
points(x[1 : (n/2)], y[1 : (n/2)], pch = 21, col = "black", bg = "lightblue", cex = 1)
points(x[(n/2 + 1) : n], y[(n/2 + 1) : n], pch = 21, col = "black", bg = "salmon", cex = 1)

```



```

# print treatment and adjustment effects
rbind("Treatment Effect" = lm(y~t+x)$coef[2], "Adjustment Effect" = lm(y~t)$coef[2])

```

```

##                                     t
## Treatment Effect    0.97962763
## Adjustment Effect -0.04747772

```

- in the above graph, the elements are as follows:

- *red* lines = means for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t)$
- *black* lines = regression lines for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t + x)$

- *blue* line = overall regression of  $y$  vs  $x \rightarrow$  line representing  $\text{lm}(y \sim x)$
- from the graph, we can see that
  - no *clear relationship* between group variable  $t$  and  $Y$ 
    - \* two groups have similar distributions with respect to  $Y$
  - *treatment effect* is substantial
    - \* separation of regression lines is large
  - *adjustment effect* is effectively 0 as there are no large differences between the means of the groups
  - group variable  $t$  is *not related* to  $x$ 
    - \* distribution of each group (salmon vs light blue) of  $Y$  vs  $X$  is effectively the same
  - lots of direct evidence for comparing two groups holding  $X$  fixed

## Simulation 5 - Binary Interaction

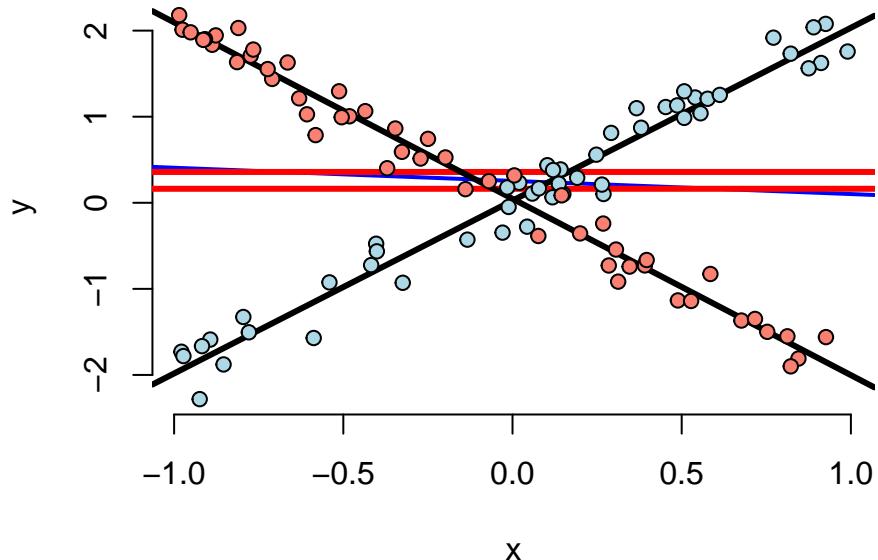
- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 t_i + \beta_3 x_i t_i + \epsilon_i$$

where  $t = \{0, 1\} \rightarrow$  binary variable

- in this case,  $\beta_0$  and  $\beta_2$  are set to 0

```
# simulate data
n <- 100; t <- rep(c(0, 1), c(n/2, n/2)); x <- c(runif(n/2, -1, 1), runif(n/2, -1, 1));
# define parameters/coefficients
beta0 <- 0; beta1 <- 2; beta2 <- 0; beta3 <- -4; sigma <- .2
# generate outcome using linear model
y <- beta0 + x * beta1 + t * beta2 + t * x * beta3 + rnorm(n, sd = sigma)
# set up axes
plot(x, y, type = "n", frame = FALSE)
# plot linear fit of y vs x
abline(lm(y ~ x), lwd = 2, col = "blue")
# plot means of the two groups (t = 0 vs t = 1)
abline(h = mean(y[1 : (n/2)]), lwd = 3, col = "red")
abline(h = mean(y[(n/2 + 1) : n]), lwd = 3, col = "red")
# plot linear fit of y vs x and t and interaction term
fit <- lm(y ~ x + t + I(x * t))
# plot the two lines corresponding to (t = 0 vs t = 1)
abline(coef(fit)[1], coef(fit)[2], lwd = 3)
abline(coef(fit)[1] + coef(fit)[3], coef(fit)[2] + coef(fit)[4], lwd = 3)
# add in the actual data points
points(x[1 : (n/2)], y[1 : (n/2)], pch = 21, col = "black", bg = "lightblue", cex = 1)
points(x[(n/2 + 1) : n], y[(n/2 + 1) : n], pch = 21, col = "black", bg = "salmon", cex = 1)
```



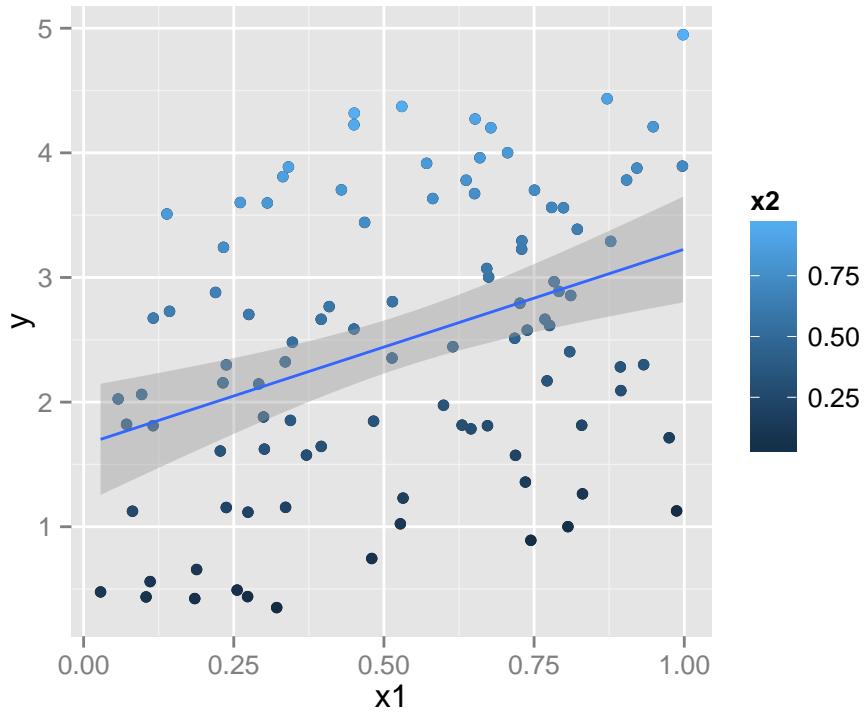
- in the above graph, the elements are as follows:
  - red* lines = means for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t)$
  - black* lines = regression lines for two groups ( $t = 0$  vs  $t = 1$ )  $\rightarrow$  two lines representing  $\text{lm}(y \sim t + x + t*x)$
  - blue* line = overall regression of  $y$  vs  $x$   $\rightarrow$  line representing  $\text{lm}(y \sim x)$ 
    - \* this is completely meaningless in this case
- from the graph, we can see that
  - treatment effect* does not apply since it varies with  $x$ 
    - \* impact of treatment/group variable **reverses itself** depending on  $x$
  - adjustment effect* is effectively zero as the means of the two groups are very similar
  - both intercept and slope of the two lines depend on the group variable  $t$
  - group variable and  $x$  are **unrelated**
  - lots of information for comparing group effects holding  $x$  fixed

### Simulation 6 - Continuous Adjustment

- the following code simulates the linear model,

$$Y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$$

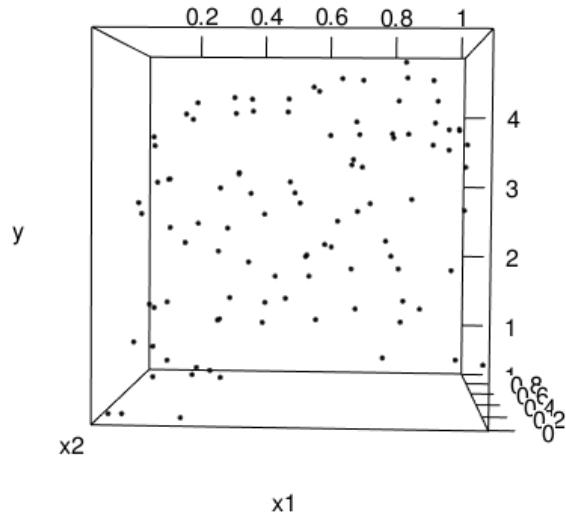
```
# simulate data
p <- 1; n <- 100; x2 <- runif(n); x1 <- p * runif(n) - (1 - p) * x2
# define parameters/coefficients
beta0 <- 0; beta1 <- 1; beta2 <- 4 ; sigma <- .01
# generate outcome using linear model
y <- beta0 + x1 * beta1 + beta2 * x2 + rnorm(n, sd = sigma)
# plot y vs x1 and x2
qplot(x1, y) + geom_point(aes(colour=x2)) + geom_smooth(method = lm)
```



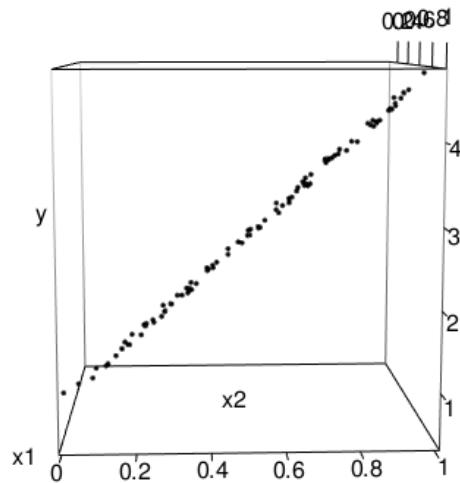
- in the above graph, we plotted  $y$  vs  $x_1$  with  $x_2$  denoted as the gradient of color from blue to white
- to investigate the bivariate relationship more clearly, we can use the following command from the `rgl` package to generate *3D plots*

```
rgl::plot3d(x1, x2, y)
```

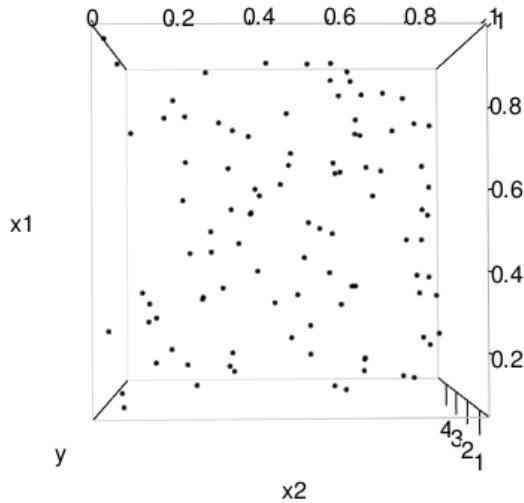
- $y$  vs  $x_1$



- $y$  vs  $x_2$

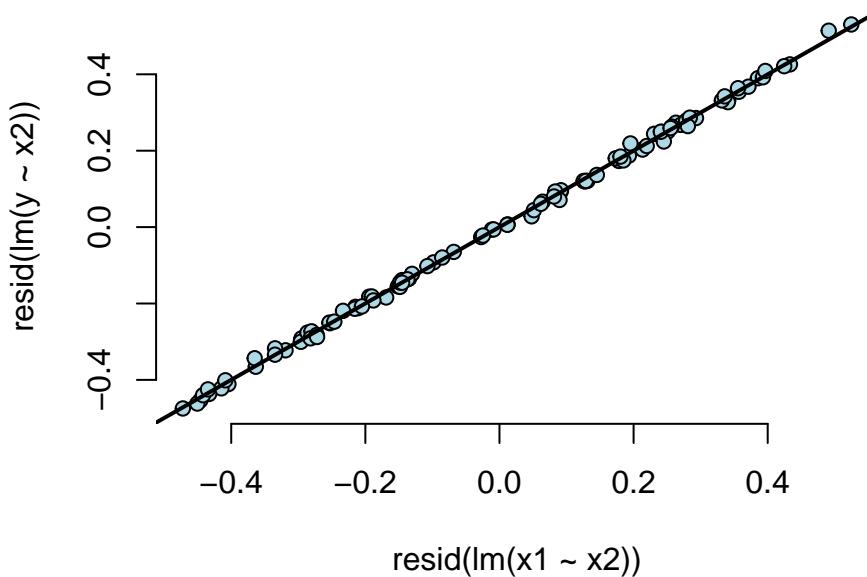


- $x_1$  vs  $x_2$



- residual plot with effect of  $x_2$  removed from both  $y$  and  $x_1$

```
# plot the residuals for y and x1 with x2 removed
plot(resid(lm(x1 ~ x2)), resid(lm(y ~ x2)), frame = FALSE,
      col = "black", bg = "lightblue", pch = 21, cex = 1)
# add linear fit line
abline(lm(I(resid(lm(y ~ x2))) ~ I(resid(lm(x1 ~ x2)))), lwd = 2)
```



- from the generated plots above, we can see that
  - $x_1$  is ***unrelated*** to  $x_2$
  - $x_2$  ***strongly correlated*** to  $y$
  - relationship between  $x_1$  and  $y$  (loosely correlated  $\rightarrow R^2 = 0.14$ ) ***largely unchanged*** by when  $x_2$  is considered
    - \*  $x_2$  captures the vast majority of variation in data
    - \* there exists almost no residual variability after removing  $x_2$

## Summary and Considerations

- modeling multivariate relationships is ***difficult***
  - modeling for prediction is fairly straight forward
  - interpreting the regression lines is much harder, as adjusting for variables can have profound effect on variables of interest
- it is often recommended to explore with simulations to see how inclusion or exclusion of another variable affects the relationship of variable of interest and the outcome
- variable selection simply affects associations between outcome and predictors, using the model to formulate causal relationship are even more difficult (entire field dedicated to this  $\rightarrow$  ***causal inference***)

## Residuals and Diagnostics

- recall that the **generalized linear model** is defined as

$$Y_i = \sum_{k=1}^p X_{ik}\beta_j + \epsilon_i$$

where  $\epsilon_i \stackrel{iid}{\sim} N(0, \sigma^2)$

- the **predicted outcome**,  $\hat{Y}_i$ , is defined as

$$\hat{Y}_i = \sum_{k=1}^p X_{ik}\hat{\beta}_j$$

- the **residuals**,  $e_i$ , is defined as

$$e_i = Y_i - \hat{Y}_i = Y_i - \sum_{k=1}^p X_{ik}\hat{\beta}_j$$

- the unbiased estimate for **residual variation** is defined as

$$\hat{\sigma}_{resid}^2 = \frac{\sum_{i=1}^n e_i^2}{n - p}$$

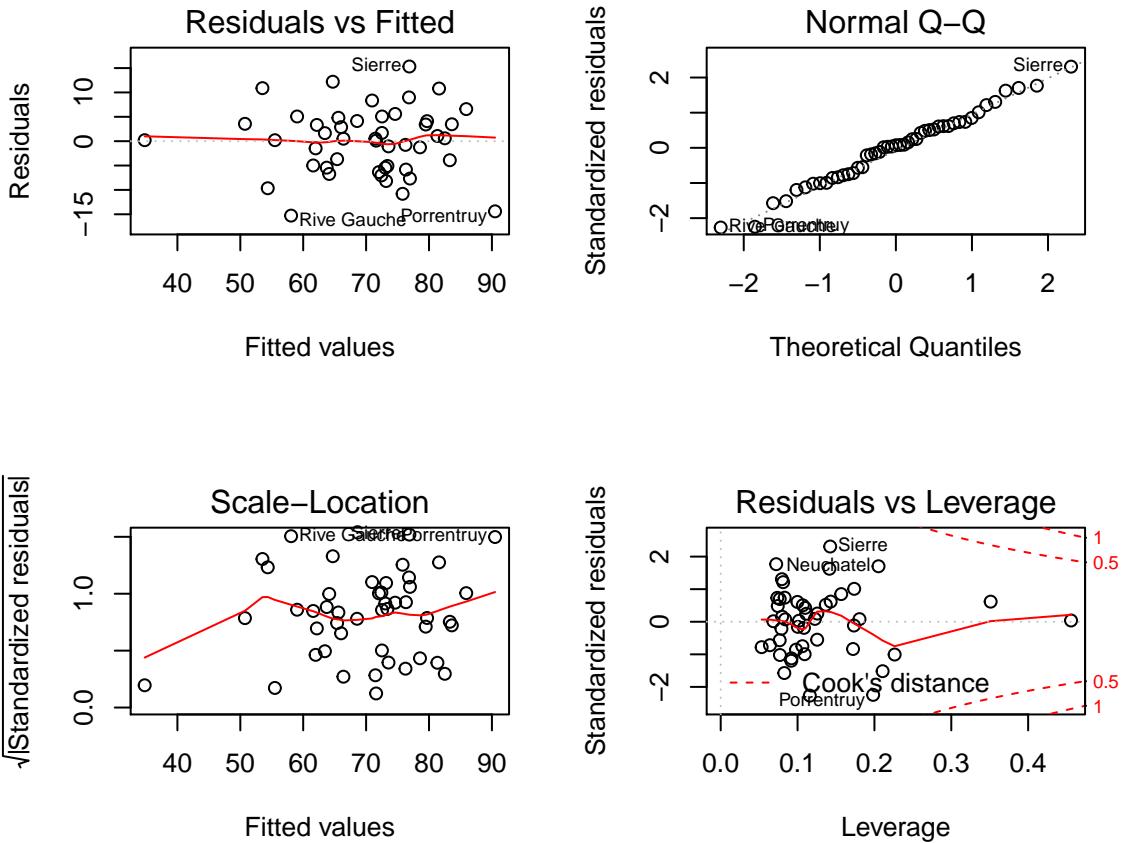
where the denominator is  $n - p$  so that  $E[\hat{\sigma}^2] = \sigma^2$

- to evaluate the fit and residuals of a linear model generated by R (i.e. `fit <- lm(y~x)`), we can use the `plot(fit)` to produce a series of **4 diagnostic plots**

- **Residuals vs Fitted** = plots ordinary residuals vs fitted values → used to detect patterns for missing variables, heteroskedasticity, etc
- **Scale-Location** = plots standardized residuals vs fitted values → similar residual plot, used to detect patterns in residuals
- **Normal Q-Q** = plots theoretical quantiles for standard normal vs actual quantiles of standardized residuals → used to evaluate normality of the errors
- **Residuals vs Leverage** = plots cooks distances comparison of fit at that point vs potential for influence of that point → used to detect any points that have substantial influence on the regression model

- example**

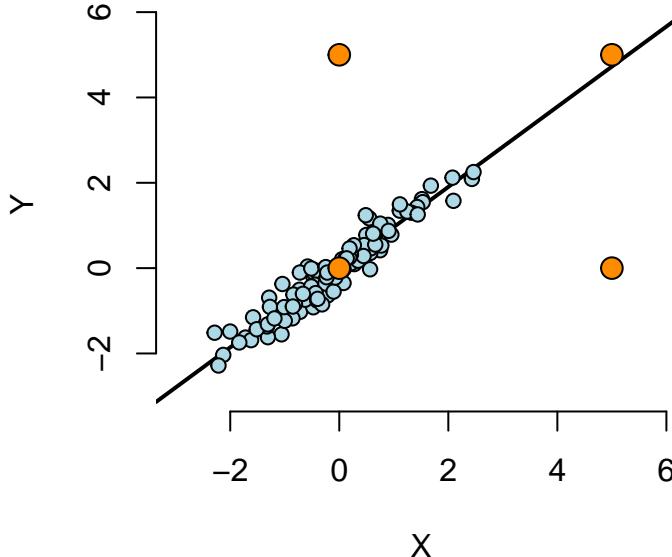
```
# load swiss data and
data(swiss)
# run linear regression on Fertility vs all other predictors
fit <- lm(Fertility ~ . , data = swiss)
# generate diagnostic plots in 2 x 2 panels
par(mfrow = c(2, 2)); plot(fit)
```



## Outliers and Influential Points

- **outlier** = an observation that is distant from the other observations of the data set
  - can be results of **spurious** or real processes
  - can conform to the regression relationship (i.e. being marginally outlying in X or Y, but not outlying given the regression relationship)

```
# generate data
n <- 100; x <- rnorm(n); y <- x + rnorm(n, sd = .3)
# set up axes
plot(c(-3, 6), c(-3, 6), type = "n", frame = FALSE, xlab = "X", ylab = "Y")
# plot regression line for y vs x
abline(lm(y ~ x), lwd = 2)
# plot actual (x, y) pairs
points(x, y, cex = 1, bg = "lightblue", col = "black", pch = 21)
# plot 4 points of interest
points(0, 0, cex = 1.5, bg = "darkorange", col = "black", pch = 21)
points(0, 5, cex = 1.5, bg = "darkorange", col = "black", pch = 21)
points(5, 5, cex = 1.5, bg = "darkorange", col = "black", pch = 21)
points(5, 0, cex = 1.5, bg = "darkorange", col = "black", pch = 21)
```



- different outliers can have *varying* degrees of *influence*
  - influence = actual effect on model fit
  - leverage = potential for influence
- in the plot above, we examine 4 different points of interest (in orange)
  - **lower left:** low leverage, low influence, *not* an outlier in any sense
  - **upper left:** low leverage, low influence, classified as outlier because it does not conform to the regression relationship
    - \* *Note: this point, though far away from the rest, does not impact the regression line since it lies in the middle of the data range because the regression line must always pass through the mean/center of observations*
  - **upper right:** high leverage, low influence, classified as outlier because it lies far away from the rest of the data
    - \* *Note: this point has low influence on regression line because it conforms to the overall regression relationship*
  - **lower right:** high leverage, high influence, classified as outlier because it lies far away from the rest of the data AND it does not conform to the regression relationship

## Influence Measures

- there exists many pre-written functions to measure influence of observations already in the `stats` package in R
  - *Note: typing in ?influence.measures in R will display the detailed documentation on all available functions to measure influence*
  - *Note: the model argument referenced in the following functions is simply the linear fit model generated by the lm function (i.e. model <- lm(y~x))*
  - `rstandard(model)` = standardized residuals → residuals divided by their standard deviations
  - `rstudent(model)` = standardized residuals → residuals divided by their standard deviations, where the  $i^{th}$  data point was deleted in the calculation of the standard deviation for the residual to follow a t distribution
  - `hatvalues(model)` = measures of leverage
  - `dfbetas(model)` = change in the predicted response when the  $i^{th}$  point is deleted in fitting the model

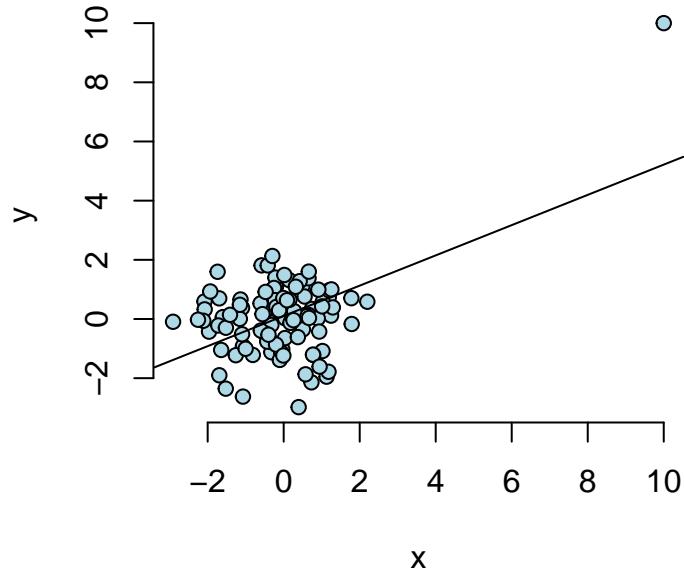
- \* effectively measures influence of a point on prediction
- `dfbetas(model)` = change in individual coefficients when the  $i^{th}$  point is deleted in fitting the model
  - \* effectively measures influence of the individual coefficients
- `cooks(model).distance` = overall change in coefficients when the  $i^{th}$  point is deleted
- `resid(model)` = returns ordinary residuals
- `resid(model)/(1-hatvalues(model))` = returns *PRESS* residuals (i.e. the leave one out cross validation residuals)
  - \* *PRESS* residuals measure the differences in the response and the predicted response at data point  $i$ , where it was not included in the model fitting
  - \* effectively measures the prediction error based on model constructed with every other point but the one of interest

## Using Influence Measures

- the purpose of these functions are to probe the given data in different ways to *diagnose* different problems
  - patterns in **residual plots** (most important tool) -> generally indicate some poor aspect of model fit
    - \* heteroskedasticity -> non-constant variance
    - \* missing model terms
    - \* temporal patterns -> residuals versus collection order/index exhibit pattern
  - **residual Q-Q plots** plots theoretical quantile vs actual quantiles of residuals
    - \* investigates whether the errors follow the standard normal distribution
  - **leverage measures** (hat values) measures the potential to influence the regression model
    - \* only depend on  $x$  or predictor variables
    - \* can be useful for diagnosing data entry errors
  - **influence measures** (i.e. `dfbetas`) measures actual influence of points on the regression model
    - \* evaluates how deleting or including this point impact a particular aspect of the model
- it is important to understand these functions/tools and use carefully in the *appropriate context*
- not all measures have *meaningful absolute scales*, so it may be useful to apply these measures to different values in the same data set but *almost never* to different datasets

## Example - Outlier Causing Linear Relationship

```
# generate random data and point (10, 10)
x <- c(10, rnorm(n)); y <- c(10, c(rnorm(n)))
# plot y vs x
plot(x, y, frame = FALSE, cex = 1, pch = 21, bg = "lightblue", col = "black")
# perform linear regression
fit <- lm(y ~ x)
# add regression line to plot
abline(fit)
```



- data generated
  - 100 points are randomly generated from the standard normal distribution
  - point (10, 10) added to the data set
- there is no regression relationship between X and Y as the points are simply random noise
- the regression relationship was able to be constructed precisely because of the presence of the point (10, 10)
  - $R^2 = 0.2623504$
  - a single point has created a strong regression relationship where there shouldn't be one
    - \* point (10, 10) has high leverage and high influence
  - we can use diagnostics to detect this kind of behavior
- `dfbetas(fit)` = difference in coefficients for including vs excluding each data point
  - the function will return a  $n \times m$  dataframe, where  $n$  = number of values in the original dataset, and  $m$  = number of coefficients
  - for this example, the coefficients are  $\beta_0$  (intercept), and  $\beta_1$  (slope), and we are interested in the slope

```
# calculate the dfbetas for the slope the first 10 points
round(dfbetas(fit)[1 : 10, 2], 3)
```

```
##       1      2      3      4      5      6      7      8      9      10
##  6.390 -0.047 -0.008 -0.002  0.024  0.000  0.027  0.013 -0.001  0.013
```

- as we can see from above, the slope coefficient would *change dramatically* if the first point (10, 10) is left out
- `hatvalues(fit)` = measures the potential for influence for each point

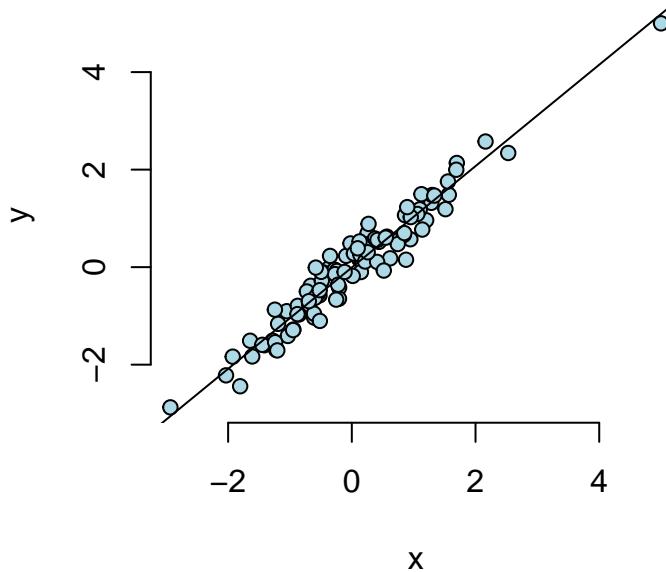
```
# calculate the hat values for the first 10 points
round(hatvalues(fit)[1 : 10], 3)
```

```
##       1      2      3      4      5      6      7      8      9      10
##  0.499  0.018  0.010  0.010  0.011  0.010  0.015  0.012  0.010  0.010
```

- again, as we can see from above, the *potential for influence is very large* for the first point (10, 10)

### Example - Real Linear Relationship

```
# generate data
x <- rnorm(n); y <- x + rnorm(n, sd = .3)
# add an outlier that fits the relationship
x <- c(5, x); y <- c(5, y)
# plot the (x, y) pairs
plot(x, y, frame = FALSE, cex = 1, pch = 21, bg = "lightblue", col = "black")
# perform the linear regression
fit2 <- lm(y ~ x)
# add the regression line to the plot
abline(fit2)
```



- data generated
  - 100 directly correlated points are generated
  - point (5, 5) added to the data set
- there is a linear relationship between X and Y
  - $R^2 = 0.9397932$
  - point (5, 5) has high leverage and low influence

```
# calculate the dfbetas for the slope the first 10 points
round(dfbetas(fit2)[1 : 10, 2], 3)
```

```
##      1      2      3      4      5      6      7      8      9     10
## -0.372 -0.021 -0.042 -0.004 -0.120  0.025  0.086 -0.069  0.156  0.013
```

```
# calculate the hat values for the first 10 points
round(hatvalues(fit2)[1 : 10], 3)
```

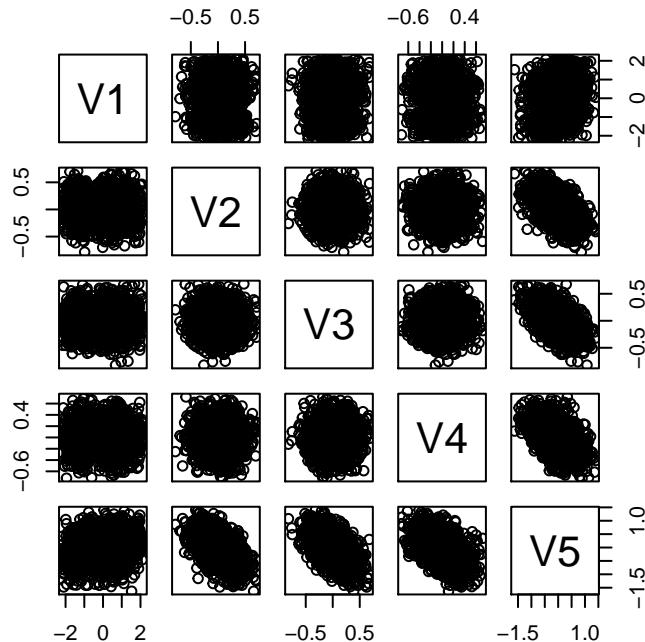
```
##      1     2     3     4     5     6     7     8     9     10
## 0.213 0.012 0.011 0.023 0.017 0.011 0.017 0.030 0.022 0.010
```

- as we can see from above, the point (5, 5) no longer has a large dfbetas value (indication of low influence) but still has a substantial hatvalue (indication of high leverage)
  - this is in line with our expectations

### Example - Stefanski TAS 2007

- taken from Leonard A. Stefanski's paper [Residual \(Sur\)Realism](#)
- data set can be found [here](#)
- the data itself exhibit no sign of correlation between the variables

```
# read data
data <- read.table('http://www4.stat.ncsu.edu/~stefanski/NSF_Supported/Hidden_Images/orly_owl_files/orly'
  header = FALSE)
# construct pairwise plot
pairs(data)
```



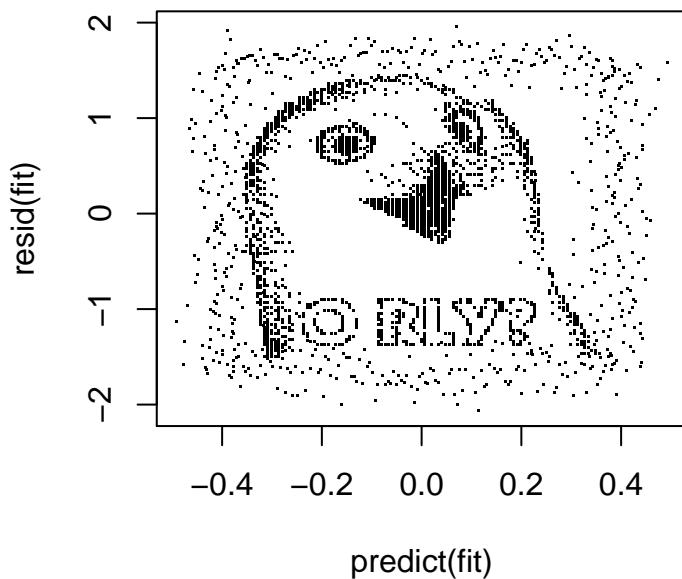
```
# perform regression on V1 with all other predictors (omitting the intercept)
fit <- lm(V1 ~ . - 1, data = data)
# print the coefficient for linear model
summary(fit)$coef
```

```
##      Estimate Std. Error    t value    Pr(>|t|)
## V2  0.9856157  0.12798121  7.701253 1.989126e-14
```

```
## V3 0.9714707 0.12663829 7.671225 2.500259e-14  
## V4 0.8606368 0.11958267 7.197003 8.301184e-13  
## V5 0.9266981 0.08328434 11.126919 4.778110e-28
```

- as we can see from above, the p-values for the coefficients indicate that they are significant
- if we take a look at the residual plot, an interesting pattern appears

```
# plot the residuals vs fitted values  
plot(predict(fit), resid(fit), pch = '.')
```



## Model Selection

- **goal for modeling** = find *parsimonious, interpretable representations* of data that enhance our understanding

“A model is a lense through which to look at your data” – Scott Zeger

“There’s no such thing as a correct model” – Brian Caffo

- whichever model connects data to a true, parsimonious statement would be **best** model
- like nearly all aspects of statistics, good modeling decisions are context dependent
- good model for prediction  $\neq$  model to establish causal effects
  - prediction model may tolerate more variables and variability

## Rumsfeldian Triplet

“There are known knowns. These are things we know that we know. There are known unknowns. That is to say, there are things that we know we don’t know. But there are also unknown unknowns. There are things we don’t know we don’t know.” – Donald Rumsfeld

- **known knowns** = regressors that we know and have, which will be evaluated to be included in the model
- **known unknowns** = regressors that we but don’t have but would like to include in the model
  - didn’t or couldn’t collect the data
- **unknown unknowns** = regressors that we don’t know about that we should have included in the model

## General Rules

- **omitting variables**  $\rightarrow$  generally results in *increased bias* in coefficients of interest
  - exceptions are when the omitted variables are uncorrelated with the regressors (variables of interest/included in model)
    - \* *Note: this is why randomize treatments/trials/experiments are the norm; it's the best strategy to balance confounders, or maximizing the probability that the treatment variable is uncorrelated with variables not in the model*
    - \* often times, due to experiment conditions or data availability, we cannot randomize
    - \* however, if there are too many unobserved confounding variables, even randomization won’t help
- **including irrelevant/unnecessary variables**  $\rightarrow$  generally *increases standard errors* (estimated standard deviation) of the coefficients
  - *Note: including any new variables increases true standard errors of other regressors, so it is not wise to idly add variables into model*
- whenever highly correlated variables are included in the same model  $\rightarrow$  the standard error and therefore the *variance* of the model *increases*  $\rightarrow$  this is known as **variance inflation**

- actual increase in standard error of coefficients for adding a regressor = estimated by the ratio of the estimated standard errors minus 1, or in other words

$$\Delta_{\sigma \mid \text{adding } x_2} = \frac{\hat{\sigma}_{y \sim x_1 + x_2}}{\hat{\sigma}_{y \sim x_1}} - 1$$

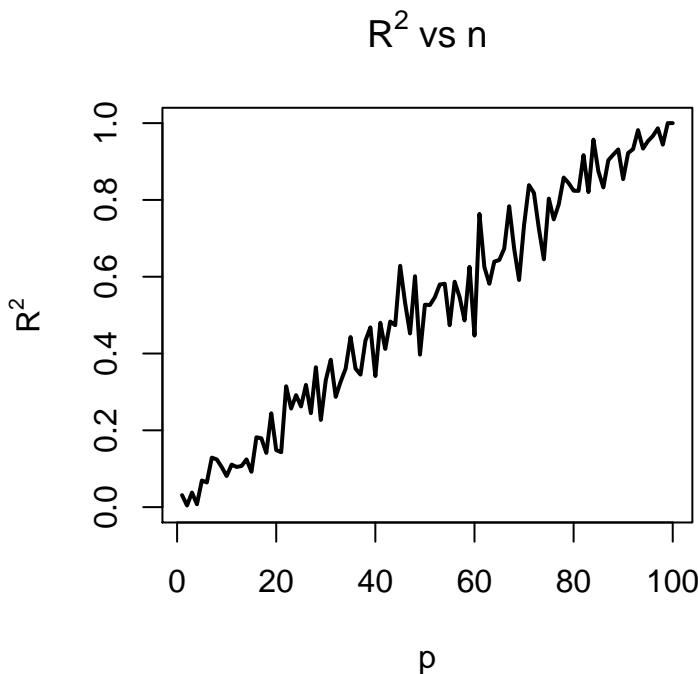
for all coefficients for the regression model

- \* **example:** if standard error of the  $\beta_1$  of  $y \sim x_1 + x_2 = 1.5$  and standard error for the  $\beta_1$  of  $y \sim x_1 = 0.5$ , then the actual increase in standard error of the  $\beta_1 = 1.5/0.5 - 1 = 200\%$
- **Note:** when the regressors added are orthogonal (statistically independent) to the regressor of interest, then there is no variance inflation
  - \* **variance inflation factor (VIF)** = the increase in the variance for the  $i^{th}$  regressor compared to the ideal setting where it is orthogonal to the other regressors
  - \*  $\sqrt{VIF}$  = increase in standard error
- **Note:** variance inflation is only part of the picture in that sometimes we will include variables even though they dramatically inflate the variation because it is an empirical part of the relationship we are attempting to model
- as the number of non-redundant variables increases or approaches  $n$ , the model **approaches a perfect fit** for the data
  - $R^2$  monotonically increases as more regressors are included
  - Sum of Squared Errors (SSE) monotonically decreases as more regressors are included

### Example - $R^2$ v n

- for the simulation below, no actual regression relationship exist as the data generated are simply standard normal noise
- it is clear, however, that as  $p$ , the number of regressors included in the model, approaches  $n$ , the  $R^2$  value approaches 1.0, which signifies perfect fit

```
# set number of measurements
n <- 100
# set up the axes of the plot
plot(c(1, n), 0 : 1, type = "n", xlab = "p", ylab = expression(R^2),
     main = expression(paste(R^2, " vs n")))
# for each value of p from 1 to n
r <- sapply(1 : n, function(p){
  # create outcome and p regressors
  y <- rnorm(n); x <- matrix(rnorm(n * p), n, p)
  # calculate the R^2
  summary(lm(y ~ x))$r.squared
})
# plot the R^2 values and connect them with a line
lines(1 : n, r, lwd = 2)
```



### Adjusted $R^2$

- recall that  $R^2$  is defined as the percent of total variability that is explained by the regression model, or

$$R^2 = \frac{\text{regression variation}}{\text{total variation}} = 1 - \frac{\sum_{i=1}^n (Y_i - \hat{Y}_i)^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2} = 1 - \frac{Var(e_i)}{Var(Y_i)}$$

- Estimating  $R^2$  with the above definition is **acceptable** when there is a *single* variable, but it becomes less and **less helpful** as the *number of variables increases*
  - as we have shown previously,  $R^2$  always increases as more variables are introduced and is thus **biased**
- adjusted  $R^2$**  is a better estimate of variability explained by the model and is defined as

$$R_{adj}^2 = 1 - \frac{Var(e_i)}{Var(Y_i)} \times \frac{n - 1}{n - k - 1}$$

where  $n$  = number of observations, and  $k$  = number of predictors in the model

- since  $k$  is always greater than zero, the adjusted  $R^2$  is **always smaller** than the unadjusted  $R^2$
- adjusted  $R^2$  also penalizes adding large numbers of regressors, which would have inflated the unadjusted  $R^2$

### Example - Unrelated Regressors

- in the simulation below, outcome  $y$  is only related to  $x_1$ 
  - $x_2$  and  $x_3$  are random noise
- we will run 1000 simulations of 3 linear regression models, and calculate the **standard error of the slope**
  - $y$  vs  $x_1$

- $y$  vs  $x_1 + x_2$
- $y$  vs  $x_1 + x_2 + x_3$

```
# simulate data
n <- 100; nosim <- 1000
# generate 3 random noise, unrelated variables
x1 <- rnorm(n); x2 <- rnorm(n); x3 <- rnorm(n);
# calculate betas of three different regression
betas <- sapply(1 : nosim, function(i){
  # generate outcome as only related to x1
  y <- x1 + rnorm(n, sd = .3)
  # store beta1 of linear regression on y vs x1
  c(coef(lm(y ~ x1))[2],
    # store beta1 of linear regression on y vs x1 and x2
    coef(lm(y ~ x1 + x2))[2],
    # store beta1 of linear regression on y vs x1 x2 and x3
    coef(lm(y ~ x1 + x2 + x3))[2])
})
# calculate the standard error of the betas for the three regressions
beta1.se <- round(apply(betas, 1, sd), 5)
# print results
rbind("y ~ x1" = c("beta1SE" = beta1.se[1]),
      "y ~ x1 + x2" = beta1.se[2],
      "y ~ x1 + x2 + x3" = beta1.se[3])

##                                beta1SE.x1
## y ~ x1                  0.02623
## y ~ x1 + x2              0.02623
## y ~ x1 + x2 + x3        0.02639
```

- as we can see from the above result, if we include unrelated regressors  $x_2$  and  $x_3$ , the *standard error increases*

### Example - Highly Correlated Regressors / Variance Inflation

- in the simulation below, outcome  $y$  is related to  $x_1$ 
  - $x_2$  and  $x_3$  are highly correlated with  $x_1$
  - $x_3$  is more correlated with  $x_1$  than  $x_2$
- we will run 1000 simulations of 3 linear regression models, and calculate the *standard error of beta<sub>1</sub>*, the coefficient of  $x_1$ 
  - $y$  vs  $x_1$
  - $y$  vs  $x_1 + x_2$
  - $y$  vs  $x_1 + x_2 + x_3$

```
# generate number of measurements and trials
n <- 100; nosim <- 1000
# generate random variables that are correlated with each other
x1 <- rnorm(n); x2 <- x1/sqrt(2) + rnorm(n) /sqrt(2)
x3 <- x1 * 0.95 + rnorm(n) * sqrt(1 - 0.95^2);
# calculate the betas for 1000 trials
```

```

betas <- sapply(1 : nosim, function(i){
  # generate outcome as only related to x1
  y <- x1 + rnorm(n, sd = .3)
  # store beta1 of linear regression on y vs x1
  c(coef(lm(y ~ x1))[2],
    # store beta1 of linear regression on y vs x1 and x2
    coef(lm(y ~ x1 + x2))[2],
    # store beta1 of linear regression on y vs x1 x2 and x3
    coef(lm(y ~ x1 + x2 + x3))[2])
})

# calculate the standard error of the beta1 for the three regressions
beta1.se <- round(apply(betas, 1, sd), 5)
# print results
rbind("y ~ x1" = c("beta1SE" = beta1.se[1]),
      "y ~ x1 + x2" = beta1.se[2],
      "y ~ x1 + x2 + x3" = beta1.se[3])

```

```

##                                beta1SE.x1
## y ~ x1                  0.02819
## y ~ x1 + x2              0.04420
## y ~ x1 + x2 + x3        0.09793

```

- as we can see from above, adding highly correlated regressors *drastically increases* the standard errors of the coefficients
- to estimate the actual change in variance, we can use the ratio of estimated variances for the  $\beta_1$  coefficient for the different models
  - `summary(fit)$cov.unscaled` = returns p x p covariance matrix for p coefficients, with the diagonal values as the true variances of coefficients
  - \* `summary(fit)$cov.unscaled[2,2]` = true variance for the  $\beta_1$

```

# generate outcome that is correlated with x1
y <- x1 + rnorm(n, sd = .3)
# store the variance of beta1 for the 1st model
a <- summary(lm(y ~ x1))$cov.unscaled[2,2]
# calculate the ratio of variances of beta1 for 2nd and 3rd models with respect to 1st model
c(summary(lm(y ~ x1 + x2))$cov.unscaled[2,2],
  summary(lm(y~ x1 + x2 + x3))$cov.unscaled[2,2]) / a - 1

```

```

## [1] 1.435784 10.284551

```

```

# alternatively, the change in variance can be estimated by calculating ratio of trials variance
temp <- apply(betas, 1, var); temp[2 : 3] / temp[1] - 1

```

```

##          x1          x1
## 1.457405 11.065503

```

- as we can see from the above results
  - adding  $x_2$  increases the variance by approximately 1 fold
  - adding  $x_2$  and  $x_3$  increases the variance by approximately 11 folds
- the estimated values from the 1000 trials are *different but close* to the true increases, and they will approach the true values as the number of trials increases

## Example: Variance Inflation Factors

- we will use the `swiss` data set for this example, and compare the following models
  - Fertility vs Agriculture
  - Fertility vs Agriculture + Examination
  - Fertility vs Agriculture + Examination + Education

```
# load swiss data
data(swiss)

# run linear regression for Fertility vs Agriculture
fit <- lm(Fertility ~ Agriculture, data = swiss)

# variance for coefficient of Agriculture
a <- summary(fit)$cov.unscaled[2,2]

# run linear regression for Fertility vs Agriculture + Examination
fit2 <- update(fit, Fertility ~ Agriculture + Examination)

# run linear regression for Fertility vs Agriculture + Examination + Education
fit3 <- update(fit, Fertility ~ Agriculture + Examination + Education)

# calculate ratios of variances for Agriculture coef for fit2 and fit3 w.r.t fit1
c(summary(fit2)$cov.unscaled[2,2], summary(fit3)$cov.unscaled[2,2]) / a - 1
```

```
## [1] 0.8915757 1.0891588
```

- as we can see from above
  - adding Examination variable to the model increases the variance by 89%
  - adding Examination and Education variables to the model increases the variance by 109%
- we can also calculate the **variance inflation factors** for all the predictors and see how variance will change by adding each predictor (assuming all predictor are orthogonal/independent of each other)
  - [car library] `vif(fit)` = returns the variance inflation factors for the predictors of the given linear model

```
# load car library
library(car)

# run linear regression on Fertility vs all other predictors
fit <- lm(Fertility ~ . , data = swiss)

# calculate the variance inflation factors
vif(fit)
```

```
##          Agriculture      Examination       Education      Catholic
##            2.284129        3.675420        2.774943        1.937160
## Infant.Mortality
##            1.107542
```

```
# calculate the standard error inflation factors
sqrt(vif(fit))
```

```
##          Agriculture      Examination       Education      Catholic
##            1.511334        1.917138        1.665816        1.391819
## Infant.Mortality
##            1.052398
```

- as we can see from the above results, Education and Examination both have relatively higher inflation factors, which makes sense as the two variables are likely to be correlated with each other

## Residual Variance Estimates

- assuming that the model is linear with additive iid errors (with finite variance), we can mathematically describe the impact of omitting necessary variables or including unnecessary ones
  - **underfitting** the model  $\rightarrow$  variance estimate is **biased**  $\rightarrow E[\hat{\sigma}^2] \neq \sigma^2$
  - **correctly fitting** or **overfitting** the model  $\rightarrow$  variance estimate is **unbiased**  $\rightarrow E[\hat{\sigma}^2] = \sigma^2$ 
    - \* however, if unnecessary variables are included, the variance estimate is **larger** than that of the correctly fitted variables  $\rightarrow Var(\hat{\sigma}_{\text{overfitted}}) \geq Var(\hat{\sigma}_{\text{correct}})$
    - \* in other words, adding unnecessary variables increases the variability of estimate for the true model

## Covariate Model Selection

- automated covariate/predictor selection is difficult
  - the space of models explodes quickly with interactions and polynomial terms
  - **Note:** in the **Practical Machine Learning** class, many modern methods for traversing large model spaces for the purposes of prediction will be covered
- principal components analysis (PCA) or factor analytic models on covariates are often useful for reducing complex covariate spaces
  - find linear combinations of variables that captures the most variation
- good experiment design can often eliminate the need for complex model searches during analyses
  - randomization, stratification can help simply the end models
  - unfortunately, control over the design is **often limited**
- it is also viable to manually explore the covariate space based on understanding of the data
  - use covariate adjustment and multiple models to probe that effect of adding a particular predictor on the model
  - **Note:** this isn't a terribly systematic or efficient approach, but it tends to teach you a lot about the data through the process
- if the models of interest are nested (i.e. one model is a special case of another with one or more coefficients set to zero) and without lots of parameters differentiating them, it's fairly possible to use nested likelihood ratio tests (ANOVA) to help find the best model
  - **Analysis of Variance** (ANOVA) works well when adding one or two regressors at a time
    - \* `anova(fit1, fit2, fit3)` = performs ANOVA or analysis of variance (or deviance) tables for a series of nested linear regression models
  - **Note:** it is extremely important to get the order of the models correct to ensure the results are sensible
  - an example can be found [here](#)
- another alternative to search through different models is the **step-wise search** algorithm that repeatedly adds/removes regressors one at a time to find the best model with the least **Akaike Information Criterion (AIC)**
  - `step(lm, k=df)` = performs step wise regression on a given linear model to find and return best linear model
    - \* `k=log(n)` = specifying the value of `k` as the log of the number of observations will force the step-wise regression model to use Bayesian Information Criterion (BIC) instead of the AIC
    - \* **Note:** both BIC and AIC penalizes adding parameters to the regression model with an additional penalty term; the penalty is **larger** for BIC than AIC
  - `MASS:::stepAIC(lm, k = df)` = more versatile, rigorous implementation of the step wise regression
  - an example can be found [here](#)

## Example: ANOVA

- we will use the `swiss` data set for this example, and compare the following nested models
  - Fertility vs Agriculture
  - Fertility vs Agriculture + Examination + Education
  - Fertility vs Agriculture + Examination + Education + Catholic + Infant.Mortality

```
# three different regressions that are nested
fit1 <- lm(Fertility ~ Agriculture, data = swiss)
fit3 <- update(fit1, Fertility ~ Agriculture + Examination + Education)
fit5 <- update(fit1, Fertility ~ Agriculture + Examination + Education + Catholic + Infant.Mortality)
# perform ANOVA
anova(fit1, fit3, fit5)

## Analysis of Variance Table
##
## Model 1: Fertility ~ Agriculture
## Model 2: Fertility ~ Agriculture + Examination + Education
## Model 3: Fertility ~ Agriculture + Examination + Education + Catholic +
##           Infant.Mortality
##   Res.Df   RSS Df Sum of Sq    F    Pr(>F)
## 1     45 6283.1
## 2     43 3180.9  2    3102.2 30.211 8.638e-09 ***
## 3     41 2105.0  2    1075.9 10.477 0.0002111 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- the ANOVA function returns a formatted table with the follow information
  - `Res.Df` = residual degrees of freedom for the models
  - `RSS` = residual sum of squares for the models, measure of fit
  - `Df` = change in degrees of freedom from one model to the next
  - `Sum of Sq` = difference/change in residual sum of squares from one model to the next
  - `F` = F statistic, measures the ratio of two scaled sums of squares reflecting different sources of variability
$$F = \frac{\frac{RSS_1 - RSS_2}{p_2 - p_1}}{\frac{RSS_2}{n - p_2}}$$

where  $p_1$  and  $p_2$  = number of parameters in the two models for comparison, and  $n$  = number of observations

  - `Pr(>F)` = p-value for the F statistic to indicate whether the change in model is significant or not
  - from the above result, we can see that both going from first to second, and second to third models result in significant reductions in RSS and **better model fits**

## Example: Step-wise Model Search

- we will use the `mtcars` data set for this example, and perform step-wise regression/model selection algorithm on the following initial model
  - Miles Per Gallon vs Number of Cylinder + Displacement + Gross Horse Power + Rear Axle Ratio + Weight

```

# load the mtcars data starting regression model
data(mtcars); fit <- lm(mpg ~ cyl + disp + hp + drat + wt, data = mtcars)
# step-wise search using BIC
step(fit, k = log(nrow(mtcars)))

## Start: AIC=73.75
## mpg ~ cyl + disp + hp + drat + wt
##
##          Df Sum of Sq    RSS    AIC
## - drat   1     3.018 170.44 70.854
## - disp   1     6.949 174.38 71.584
## - cyl    1    15.411 182.84 73.100
## <none>          167.43 73.748
## - hp    1    21.066 188.49 74.075
## - wt    1    77.476 244.90 82.453
##
## Step: AIC=70.85
## mpg ~ cyl + disp + hp + wt
##
##          Df Sum of Sq    RSS    AIC
## - disp   1     6.176 176.62 68.528
## - hp    1    18.048 188.49 70.609
## <none>          170.44 70.854
## - cyl   1    24.546 194.99 71.694
## - wt    1    90.925 261.37 81.069
##
## Step: AIC=68.53
## mpg ~ cyl + hp + wt
##
##          Df Sum of Sq    RSS    AIC
## - hp    1    14.551 191.17 67.595
## - cyl   1    18.427 195.05 68.237
## <none>          176.62 68.528
## - wt    1   115.354 291.98 81.147
##
## Step: AIC=67.6
## mpg ~ cyl + wt
##
##          Df Sum of Sq    RSS    AIC
## <none>          191.17 67.595
## - cyl   1     87.15 278.32 76.149
## - wt    1    117.16 308.33 79.426

##
## Call:
## lm(formula = mpg ~ cyl + wt, data = mtcars)
##
## Coefficients:
## (Intercept)      cyl          wt
## 39.686        -1.508       -3.191

```

- as we can see from above, the best model that captures most of the variability in the data is simply  $\text{mpg} \sim \text{cyl} + \text{wt}$

## General Linear Models Overview

- limitations of linear models:
  - response can be discrete (i.e. 0, 1, etc.) or strictly positive  $\rightarrow$  linear response models don't make much sense
  - if outcome must be positive, Gaussian errors ( $\pm$  errors) don't make sense as negative outcomes are possible
  - transformations on predictors ( $\log + 1$ ) are often hard to interpret
    - \* modeling the data on the scale that it was collected is most ideal
    - \* even for interpretable transformations, *natural logarithms* specifically, aren't applicable for negative/zero values
- **general linear models** = introduced in 1972 RSSB paper by Nelder and Wedderburn and has **3** parts
  1. exponential family model for response/outcome (i.e. Gaussian, Bernoulli distribution)
  2. systematic component for linear predictor  $\rightarrow$  incorporates the information about the independent variables into the model
    - denoted by  $\eta = X\beta$  where  $X$  is a matrix of independent variables/predictors and  $\beta$  is the coefficients
  3. link function that connects means of the outcome/distribution to linear predictor
    - the relationship is defined as  $\eta = g(\mu)$ , or the linear predictor  $\eta$  is a function of the mean of the distribution  $\mu$

## Simple Linear Model

- *exponential family distribution*: Gaussian distribution, assumed  $Y_i \sim N(\mu_i, \sigma^2)$
- *linear predictor*:  $\eta_i = \sum_{k=1}^p X_{ik}\beta_k$
- *link function* :  $g(\mu) = \eta = \mu$ 
  - for linear models,  $g(\mu) = \mu$ , so  $\eta_i = \mu_i$
- **result**: the same likelihood model (see [derivation](#)) as the additive Gaussian error linear model

$$Y_i = \sum_{k=1}^p X_{ik}\beta_k + \epsilon_i$$

where  $\epsilon_i \stackrel{iid}{\sim} N(0, \sigma^2)$

## Logistic Regression

- *exponential family distribution*: binomial/Bernoulli distribution, assumed  $Y_i \sim Bernoulli(\mu_i)$  where the probability of success is  $\mu_i$ 
  - due to the properties of the binomial/Bernoulli distribution,  $E[Y_i] = \mu_i$  where  $0 \leq \mu_i \leq 1$
- *linear predictor*:  $\eta_i = \sum_{k=1}^p X_{ik}\beta_k$
- *link function* :  $g(\mu) = \eta = \log\left(\frac{\mu}{1-\mu}\right)$ 
  - **odds** for success for a binomial/Bernoulli distribution is defined as

$$\text{odds} = \frac{p}{1-p}$$

- **logit** is defined as

$$\log(\text{odds}) = \log \frac{\mu}{1-\mu}$$

- \* **Note:** the log here is the **natural** log
- **inverse logit** is defined as
$$\mu_i = \frac{\exp(\eta_i)}{1 + \exp(\eta_i)}$$
- complement of inverse logit is
$$1 - \mu_i = \frac{1}{1 + \exp(\eta_i)}$$
- **result:** the likelihood model
$$\begin{aligned}
L(\beta) &= \prod_{i=1}^n \mu_i^{y_i} (1 - \mu_i)^{1-y_i} \\
(\text{plug in } \mu_i \text{ and } 1 - \mu_i \text{ from above}) &= \prod_{i=1}^n \left( \frac{\exp(\eta_i)}{1 + \exp(\eta_i)} \right)^{y_i} \left( \frac{1}{1 + \exp(\eta_i)} \right)^{1-y_i} \\
(\text{multiply 2nd term by } \frac{\exp(\eta_i)}{\exp(\eta_i)}) &= \prod_{i=1}^n \left( \frac{\exp(\eta_i)}{1 + \exp(\eta_i)} \right)^{y_i} \left( \frac{\exp(\eta_i)}{1 + \exp(\eta_i)} \right)^{1-y_i} \left( \frac{1}{\exp(\eta_i)} \right)^{1-y_i} \\
(\text{simplify}) &= \prod_{i=1}^n \left( \frac{\exp(\eta_i)}{1 + \exp(\eta_i)} \right) \left( \frac{1}{\exp(\eta_i)} \right)^{1-y_i} \\
(\text{simplify}) &= \prod_{i=1}^n \left( \frac{\exp(\eta_i)}{1 + \exp(\eta_i)} \right) \exp(\eta_i)^{y_i-1} \\
(\text{simplify}) &= \prod_{i=1}^n \frac{\exp(\eta_i)^{y_i}}{1 + \exp(\eta_i)} \\
(\text{change form of numerator}) &= \exp \left( \sum_{i=1}^n y_i \eta_i \right) \prod_{i=1}^n \frac{1}{1 + \exp(\eta_i)} \\
(\text{substitute } \eta_i) \Rightarrow L(\beta) &= \exp \left( \sum_{i=1}^n y_i \left( \sum_{k=1}^p X_{ik} \beta_k \right) \right) \prod_{i=1}^n \frac{1}{1 + \exp \left( \sum_{k=1}^p X_{ik} \beta_k \right)}
\end{aligned}$$

- maximizing the likelihood  $L(\beta)$  (solving for  $\frac{\partial L}{\partial \beta} = 0$ ) would return a set of optimized coefficients  $\beta$  that will fit the data

## Poisson Regression

- *exponential family distribution:* Poisson distribution, assumed  $Y_i \sim \text{Poisson}(\mu_i)$  where  $E[Y_i] = \mu_i$
- *linear predictor:*  $\eta_i = \sum_{k=1}^p X_{ik} \beta_k$
- *link function :*  $g(\mu) = \eta = \log(\mu)$ 
  - **Note:** the log here is the **natural** log
  - since  $e^x$  is the inverse of  $\log(x)$ , then  $\eta_i = \log(\mu_i)$  can be transformed into  $\mu_i = e^{\eta_i}$

- **result:** the likelihood model

$$\begin{aligned}
L(\beta) &= \prod_{i=1}^n (y_i!)^{-1} \mu_i^{y_i} e^{-\mu_i} \\
(\text{substitute } \mu_i = e^{\eta_i}) &= \prod_{i=1}^n \frac{(e^{\eta_i})^{y_i}}{y_i! e^{\eta_i}} \\
(\text{transform}) &= \prod_{i=1}^n \frac{\exp(\eta_i y_i)}{y_i! \exp(e^{\eta_i})} \\
(\text{taking log of both sides}) \quad \mathcal{L}(\beta) &= \sum_{i=1}^n \eta_i y_i - \sum_{i=1}^n e^{\eta_i} - \sum_{i=1}^n \log(y_i!) \\
(\text{since } y_i \text{ is given, we can ignore } \log y_i!) \quad \mathcal{L}(\beta) &\propto \sum_{i=1}^n \eta_i y_i - \sum_{i=1}^n e^{\eta_i} \\
(\text{substitute } \eta_i = \sum_{k=1}^p X_{ik} \beta_k) \Rightarrow \mathcal{L}(\beta) &\propto \sum_{i=1}^n y_i \left( \sum_{k=1}^p X_{ik} \beta_k \right) - \sum_{i=1}^n \exp \left( \sum_{k=1}^p X_{ik} \beta_k \right)
\end{aligned}$$

- maximizing the log likelihood  $\mathcal{L}(\beta)$  (solving for  $\frac{\partial \mathcal{L}}{\partial \beta} = 0$ ) would return a set of optimized coefficients  $\beta$  that will fit the data

## Variances and Quasi-Likelihoods

- in each of the linear/Bernoulli/Poisson cases, the **only** term in the likelihood functions that depend on the **data** is

$$\sum_{i=1}^n y_i \eta_i = \sum_{i=1}^n y_i \sum_{k=1}^p X_{ik} \beta_k = \sum_{k=1}^p \beta_k \sum_{i=1}^n X_{ik} y_i$$

- this means that we don't need all of the data collected to maximize the likelihoods/find the coefficients  $\beta$ , but **only** need  $\sum_{i=1}^n X_{ik} y_i$ 
  - **Note:** this simplification is a consequence of choosing “**canonical**” link functions,  $g(\mu)$ , to be in specific forms
- [Derivation needed] all models achieve their **maximum** at the root of the **normal equations**

$$\sum_{i=1}^n \frac{(Y_i - \mu_i)}{\text{Var}(Y_i)} W_i = 0$$

where  $W_i = \frac{\partial g^{-1}(\mu_i)}{\partial \mu_i}$  or the derivative of the inverse of the link function

- **Note:** this is similar to deriving the least square equation where the middle term must be set to 0 to find the solution (see Derivation for  $\beta$ )
- **Note:**  $\mu_i = g^{-1}(\eta_i) = g^{-1}(\sum_{k=1}^p X_{ik} \beta_k)$ , the normal functions are really functions of  $\beta$
- the variance,  $\text{Var}(Y_i)$ , is defined as
  - **linear model:**  $\text{Var}(Y_i) = \sigma^2$ , where  $\sigma$  is constant
  - **binomial model:**  $\text{Var}(Y_i) = \mu_i(1 - \mu_i)$
  - **Poisson model:**  $\text{Var}(Y_i) = \mu_i$
- for binomial and Poisson models, there are **strict relationships** between the mean and variance that can be easily tested from the data:
  - binomial: mean =  $\mu_i$ , variance =  $\mu_i(1 - \mu_i)$

- Poisson: mean =  $\mu_i$ , variance =  $\mu_i$
- it is often relevant to have a ***more flexible*** variance model (i.e. data doesn't follow binomial/Poisson distributions exactly but are approximated), even if it doesn't correspond to an actual likelihood, so we can add an extra parameter,  $\phi$ , to the normal equations to form **quasi-likelihood normal equations**

$$\text{binomial : } \sum_{i=1}^n \frac{(Y_i - \mu_i)}{\phi\mu_i(1 - \mu_i)} W_i = 0 \quad \text{Poisson : } \sum_{i=1}^n \frac{(Y_i - \mu_i)}{\phi\mu_i} W_i = 0$$

where  $W_i = \frac{\partial g^{-1}(\mu_i)}{\partial \mu_i}$  or the derivative of the inverse of the link function

- for R function `glm()`, its possible to specify for the model to solve using quasi-likelihood normal equations instead of normal equations through the parameter `family = quasi-binomial` and `family = quasi-poisson` respectively
- **Note:** the quasi-likelihood models generally share properties as normal GLM

### Solving for Normal and Quasi-Likelihood Normal Equations

- normal equations have to be solved ***iteratively***
  - the results are  $\hat{\beta}_k$ , estimated coefficients for the predictors
  - for quasi-likelihood normal equations,  $\hat{\phi}$  will be part of the results as well
  - in R, [Newton/Raphson's algorithm](#) is used to solve the equations
  - **asymptotics** are used for inference of results to broader population (see **Statistical Inference** course)
  - **Note:** many of the ideas, interpretation, and conclusions derived from simple linear models are applicable to GLMs
- **predicted linear predictor responses** are defined as

$$\hat{\eta} = \sum_{k=1}^p X_k \hat{\beta}_k$$

- **predicted mean responses** can be solved from

$$\hat{\mu} = g^{*1}(\hat{\eta})$$

- **coefficients** are interpreted as the ***expected change in the link function*** of the expected response ***per unit change*** in  $X_k$  holding other regressors constant, or

$$\beta_k = g(E[Y|X_k = x_k + 1, X_{\sim k} = x_{\sim k}]) - g(E[Y|X_k = x_k, X_{\sim k} = x_{\sim k}])$$

## General Linear Models - Binary Models

- **Bernoulli/binary** models are frequently used to model outcomes that have two values
  - alive vs dead
  - win vs loss
  - success vs failure
  - disease vs healthy
- **binomial outcomes** = collection of exchangeable binary outcomes (i.e. flipping coins repeatedly) for the same covariate data
  - in other words, we are interested in the count of predicted 1s vs 0s rather than individual outcomes of 1 or 0

### Odds

- **odds** are useful in constructing logistic regression models and fairly easy to interpret
  - imagine flipping a coin with success probability  $p$ 
    - \* if heads, you win  $X$
    - \* if tails, you lose  $Y$
  - how should  $X$  and  $Y$  be set so that the game is *fair*?
  - $$E[earnings] = Xp - Y(1-p) = 0 \Rightarrow \frac{Y}{X} = \frac{p}{1-p}$$

payout ratio      odds
  - odds can be interpreted as “How much should you be willing to pay for a  $p$  probability of winning a dollar?”
    - \* if  $p > 0.5$ , you have to pay more if you lose than you get if you win
    - \* if  $p < 0.5$ , you have to pay less if you lose than you get if you win
- odds are **NOT** probabilities
- odds ratio of 1 = no difference in odds or 50% - 50%
  - $p = 0.5 \Rightarrow odds = \frac{0.5}{1-0.5} = 1$
  - log odds ratio of 0 = no difference in odds
    - \*  $p = 0.5 \Rightarrow odds = \log\left(\frac{0.5}{1-0.5}\right) = \log(1) = 0$
- odds ratio  $< 0.5$  or  $> 2$  commonly a “moderate effect”
- **relative risk** = ratios of probabilities instead of odds, and are often easier to interpret but harder to estimate
 
$$\frac{Pr(W_i|S_i = 1)}{Pr(W_i|S_i = 0)}$$
  - *Note: relative risks often have boundary problems as the range of log(p) is  $(-\infty, 0]$  where as the range of logit  $\frac{p}{1-p}$  is  $(-\infty, \infty)$*
  - for small probabilities Relative Risk  $\approx$  Odds Ratio but **they are not the same!**

### Example - Baltimore Ravens Win vs Loss

- the data for this example can be found [here](#)
  - the data contains the records 20 games for Baltimore Ravens, a professional American Football team

- there are 4 columns
  - \* `ravenWinNum` = 1 for Raven win, 0 for Raven loss
  - \* `ravenWin` = W for Raven win, L for Raven loss
  - \* `ravenScore` = score of the Raven team during the match
  - \* `opponentScore` = score of the Raven team during the match

```
# load the data
load("ravensData.rda")
head(ravensData)
```

```
##   ravenWinNum ravenWin ravenScore opponentScore
## 1           1      W       24          9
## 2           1      W       38         35
## 3           1      W       28         13
## 4           1      W       34         31
## 5           1      W       44         13
## 6           0      L       23        24
```

### Example - Simple Linear Regression

- **simple linear regression** can be used model win vs loss for the Ravens

$$W_i = \beta_0 + \beta_1 S_i + \epsilon_i$$

- $W_i$  = binary outcome, 1 if a Ravens win, 0 if not
- $S_i$  = number of points Ravens scored
- $\beta_0$  = probability of a Ravens win if they score 0 points
- $\beta_1$  = increase in probability of a Ravens win for each additional point
- $\epsilon_i$  = residual variation, error

- the expected value for the model is defined as

$$E[W_i|S_i, \beta_0, \beta_1] = \beta_0 + \beta_1 S_i$$

- however, the model wouldn't work well as the predicted results *won't* be 0 vs 1
  - the error term,  $\epsilon_i$ , is assumed to be continuous and normally distributed, meaning that the prediction will likely be a decimal
  - therefore, this is *not* a good assumption for the model

```
# perform linear regression
summary(lm(ravenWinNum ~ ravenScore, data = ravensData))
```

```
##
## Call:
## lm(formula = ravenWinNum ~ ravenScore, data = ravensData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -0.7302 -0.5076  0.1824  0.3215  0.5719 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 0.5719    0.3215   1.78    0.082    
## ravenScore  0.3215    0.1824   1.78    0.082    
```

```

## (Intercept) 0.285032   0.256643   1.111   0.2814
## ravenScore  0.015899   0.009059   1.755   0.0963 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.4464 on 18 degrees of freedom
## Multiple R-squared:  0.1461, Adjusted R-squared:  0.09868
## F-statistic:  3.08 on 1 and 18 DF,  p-value: 0.09625

```

- as expected, the model produces a poor fit for the data ( $R^2_{adj} = 0.0987$ )
- adding a threshold to the predicted outcome (i.e. if  $\hat{W}_i < 0.5$ ,  $\hat{W}_i = 0$ ) and using the model to predict the results would be *viable*
  - however, the coefficients for the model are *not very interpretable*

### Example - Logistic Regression

- **probability** of Ravens win is defined as

$$Pr(W_i|S_i, \beta_0, \beta_1)$$

- **odds** is defined as

$$\frac{Pr(W_i|S_i, \beta_0, \beta_1)}{1 - Pr(W_i|S_i, \beta_0, \beta_1)}$$

which ranges from 0 to  $\infty$

- log odds or **logit** is defined as

$$\log\left(\frac{Pr(W_i|S_i, \beta_0, \beta_1)}{1 - Pr(W_i|S_i, \beta_0, \beta_1)}\right)$$

which ranges from  $-\infty$  to  $\infty$

- we can use the link function and linear predictors to construct the **logistic regression** model

$$g(\mu_i) = \log\left(\frac{\mu_i}{1 - \mu_i}\right) = \eta_i$$

$$(substitute \mu_i = Pr(W_i|S_i, \beta_0, \beta_1)) \quad g(\mu_i) = \log\left(\frac{Pr(W_i|S_i, \beta_0, \beta_1)}{1 - Pr(W_i|S_i, \beta_0, \beta_1)}\right) = \eta_i$$

$$(substitute \eta_i = \beta_0 + \beta_1 S_i) \Rightarrow g(\mu_i) = \log\left(\frac{Pr(W_i|S_i, \beta_0, \beta_1)}{1 - Pr(W_i|S_i, \beta_0, \beta_1)}\right) = \beta_0 + \beta_1 S_i$$

which can also be written as

$$Pr(W_i|S_i, \beta_0, \beta_1) = \frac{\exp(\beta_0 + \beta_1 S_i)}{1 + \exp(\beta_0 + \beta_1 S_i)}$$

- for the model

$$\log\left(\frac{Pr(W_i|S_i, \beta_0, \beta_1)}{1 - Pr(W_i|S_i, \beta_0, \beta_1)}\right) = \beta_0 + \beta_1 S_i$$

–  $\beta_0$  = log odds of a Ravens win if they score zero points

–  $\beta_1$  = log odds ratio of win probability for each point scored (compared to zero points)

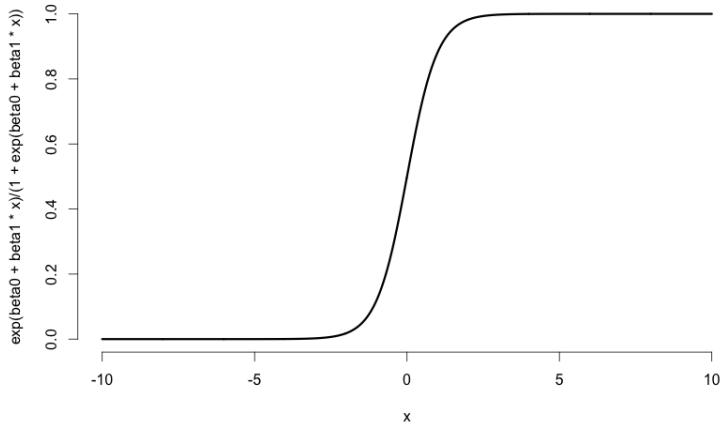
$$\beta_1 = \log(ods(S_i = S_i + 1)) - \log(ods(S_i = S_i)) = \log\left(\frac{ods(S_i = S_i + 1)}{ods(S_i = S_i)}\right)$$

–  $\exp(\beta_1)$  = odds ratio of win probability for each point scored (compared to zero points)

$$\exp(\beta_1) = \frac{ods(S_i = S_i + 1)}{ods(S_i = S_i)}$$

- we can leverage the `manupulate` function vary  $\beta_0$  and  $\beta_1$  to fit logistic regression curves for simulated data

```
# set x values for the points to be plotted
x <- seq(-10, 10, length = 1000)
# "library(manipulate)" is needed to use the manipulate function
manipulate(
  # plot the logistic regression curve
  plot(x, exp(beta0 + beta1 * x) / (1 + exp(beta0 + beta1 * x)),
       type = "l", lwd = 3, frame = FALSE),
  # slider for beta1
  beta1 = slider(-2, 2, step = .1, initial = 2),
  # slider for beta0
  beta0 = slider(-2, 2, step = .1, initial = 0)
)
```



- we can use the `glm(outcome ~ predictor, family = "binomial")` to fit a logistic regression to the data

```
# run logistic regression on data
logRegRavens <- glm(ravenWinNum ~ ravenScore, data = ravensData, family = "binomial")
# print summary
summary(logRegRavens)
```

```
##
## Call:
## glm(formula = ravenWinNum ~ ravenScore, family = "binomial",
##      data = ravensData)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.7575  -1.0999   0.5305   0.8060   1.4947
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
```

```

## (Intercept) -1.68001   1.55412  -1.081     0.28
## ravenScore    0.10658    0.06674   1.597     0.11
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 24.435  on 19  degrees of freedom
## Residual deviance: 20.895  on 18  degrees of freedom
## AIC: 24.895
##
## Number of Fisher Scoring iterations: 5

```

- as we can see above, the coefficients  $\beta_0$  and  $\beta_1$  are -1.68, 0.107, which are interpreted to be the log odds ratios
- we can convert the log ratios as well as the log confidence intervals to ratios and confidence intervals (in the same units as the data)

```

# take e^coefs to find the log ratios
exp(logRegRavens$coeff)

```

```

## (Intercept) ravenScore
## 0.1863724 1.1124694

```

```

# take e^log confidence interval to find the confidence intervals
exp(confint(logRegRavens))

```

```

## Waiting for profiling to be done...

```

```

##           2.5 % 97.5 %
## (Intercept) 0.005674966 3.106384
## ravenScore  0.996229662 1.303304

```

- **Note:**  $\exp(x) \approx 1 + x$  for small values (close to 0) of  $x$ , this can be a quick way to estimate the coefficients
- we can interpret the slope,  $\beta_1$  as 11.247 % increase in probability of winning for every point scored
- we can interpret the intercept,  $\beta_0$  as 0.186 is the odds for Ravens winning if they scored 0 points
  - **Note:** similar to the intercept of a simple linear regression model, the intercept should be interpreted carefully as it is an extrapolated value from the model and may not hold practical meaning
- to calculate specific probability of winning for a given number of points

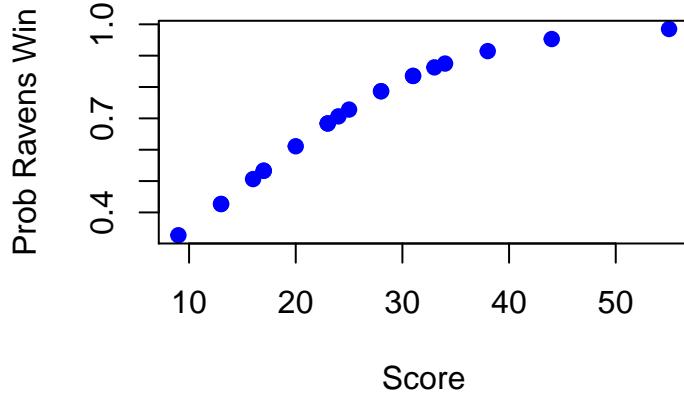
$$Pr(W_i|S_i, \hat{\beta}_0, \hat{\beta}_1) = \frac{\exp(\hat{\beta}_0 + \hat{\beta}_1 S_i)}{1 + \exp(\hat{\beta}_0 + \hat{\beta}_1 S_i)}$$

- the resulting logistic regression curve can be seen below

```

# plot the logistic regression
plot(ravensData$ravenScore, logRegRavens$fitted, pch=19, col="blue", xlab="Score", ylab="Prob Ravens Win")

```



### Example - ANOVA for Logistic Regression

- ANOVA can be performed on a single logistic regression, in which it will analyze the change in variances with addition of parameters in the model, or multiple nested logistic regression (similar to linear models)

```
# perform analysis of variance
anova(logRegRavens,test="Chisq")
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: ravenWinNum
##
## Terms added sequentially (first to last)
##
##
##          Df Deviance Resid. Df Resid. Dev Pr(>Chi)
## NULL              19      24.435
## ravenScore  1    3.5398     18      20.895  0.05991 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- ANOVA returns information about the model, link function, response, as well as analysis of variance for adding terms
  - Df = change in degrees of freedom
    - the value 1 refers to adding the ravenScore parameter (slope)
  - Deviance = measure of goodness of model fit compare to the previous model
  - Resid. Dev = residual deviance for current model
  - Pr(>Chi) = used to evaluate the significance of the added parameter
    - in this case, the Deviance value of 3.54 is used to find the corresponding p-value from the Chi Squared distribution, which is 0.06
      - Note: Chi Squared distribution with 1 degree of freedom is simply the squared of normal distribution, so z statistic of 2 corresponds to 95% for normal distribution indicates that deviance of 4 corresponds to approximately 5% in the Chi Squared distribution (which is what our result shows)*

## Further resources

- [Wikipedia on Logistic Regression](#)
- [Logistic regression and GLMs in R](#)
- Brian Caffo's lecture notes on: [Simpson's Paradox, Retrospective Case-control Studies](#)
- [Open Intro Chapter on Logistic Regression](#)

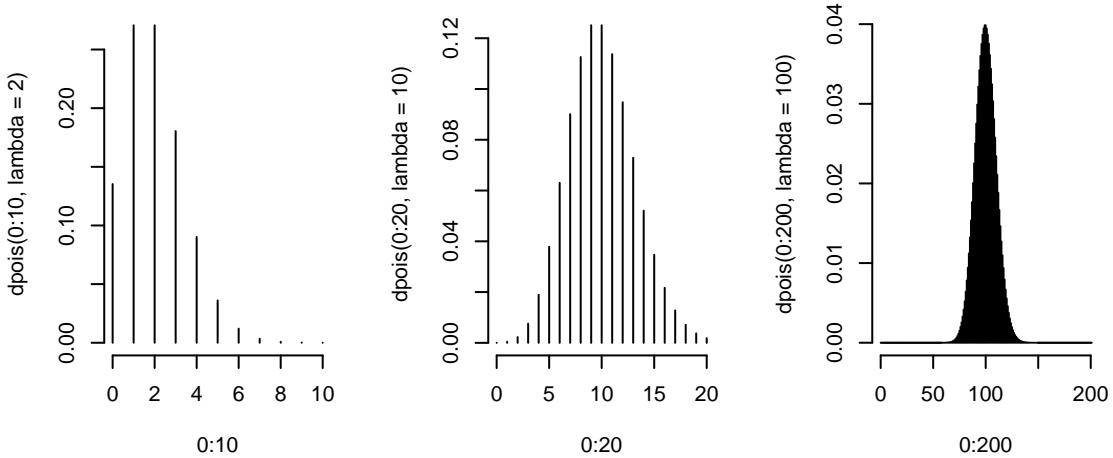
## General Linear Models - Poisson Models

- **Poisson distribution** is a useful model for counts and rates
  - **rate** = count per unit of time
  - linear regression with transformation is an alternative
- count data examples
  - calls to a call center
  - number of flu cases in an area
  - number of cars that cross a bridge
- rate data examples
  - percent of children passing a test
  - percent of hits to a website from a country
  - radioactive decay
- Poisson model examples
  - modeling web traffic hits incidence rates
  - approximating binomial probabilities with small  $p$  and large  $n$
  - analyzing contingency table data (tabulated counts for categorical variables)

### Properties of Poisson Distribution

- a set of data  $X$  is said to follow the Poisson distribution, or  $X \sim Poisson(t\lambda)$ , if
$$P(X = x) = \frac{(t\lambda)^x e^{-t\lambda}}{x!}$$
where  $x = 0, 1, \dots$ 
  - $\lambda$  = rate or expected count per unit time
  - $t$  = monitoring time
- **mean** of Poisson distribution is  $E[X] = t\lambda$ , thus  $E[X/t] = \lambda$
- **variance** of the Poisson is  $Var(X) = t\lambda = \mu(\text{mean})$ 
  - *Note: Poisson approaches a Gaussian/normal distribution as  $t\lambda$  gets large*
- below are the Poisson distributions for various values of  $\lambda$

```
# set up 1x3 panel plot
par(mfrow = c(1, 3))
# Poisson distribution for t = 1, and lambda = 2
plot(0 : 10, dpois(0 : 10, lambda = 2), type = "h", frame = FALSE)
# Poisson distribution for t = 1, and lambda = 10
plot(0 : 20, dpois(0 : 20, lambda = 10), type = "h", frame = FALSE)
# Poisson distribution for t = 1, and lambda = 100
plot(0 : 200, dpois(0 : 200, lambda = 100), type = "h", frame = FALSE)
```

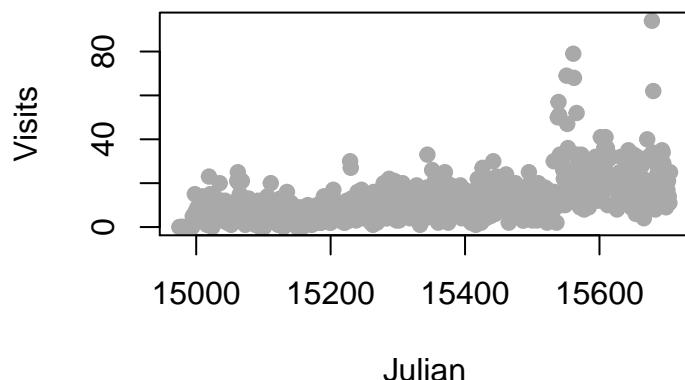


- as we can see from above, for large values of  $\lambda$ , the distribution looks like the Gaussian

### Example - Leek Group Website Traffic

- for this example, we will be modeling the daily traffic to Jeff Leek's web site: <http://biostat.jhsph.edu/~jleek/>
  - the data comes from Google Analytics and was extracted by the `rga` package that can be found at <http://skardhamar.github.com/rga/>
- for the purpose of the example, the time is *always* one day, so  $t = 1$ , Poisson mean is interpreted as web hits per day
  - if  $t = 24$ , we would be modeling web hits per hour
- the data can be found [here](#)

```
# load data
load("gaData.rda")
# convert the dates to proper formats
gaData$julian <- julian(gaData$date)
# plot visits vs dates
plot(gaData$julian,gaData$visits,pch=19,col="darkgrey",xlab="Julian",ylab="Visits")
```



## Example - Linear Regression

- the traffic can be modeled using linear model as follows

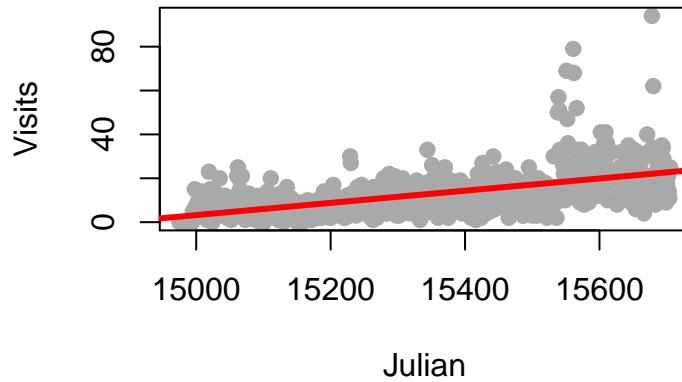
$$NH_i = \beta_0 + \beta_1 JD_i + \epsilon_i$$

- $NH_i$  = number of hits to the website
- $JD_i$  = day of the year (Julian day)
- $\beta_0$  = number of hits on Julian day 0 (1970-01-01)
- $\beta_1$  = increase in number of hits per unit day
- $\epsilon_i$  = variation due to everything we didn't measure

- the expected outcome is defined as

$$E[NH_i | JD_i, \beta_0, \beta_1] = \beta_0 + \beta_1 JD_i$$

```
# plot the visits vs dates
plot(gaData$julian, gaData$visits, pch=19, col="darkgrey", xlab="Julian", ylab="Visits")
# perform linear regression
lm1 <- lm(gaData$visits ~ gaData$julian)
# plot regression line
abline(lm1, col="red", lwd=3)
```



## Example - log Outcome

- if we are interested in relative increases in web traffic, we can take the natural log of the outcome, so the linear model becomes

$$\log(NH_i) = \beta_0 + \beta_1 JD_i + \epsilon_i$$

- $\log(NH_i)$  = number of hits to the website
- $JD_i$  = day of the year (Julian day)
- $\beta_0$  = log number of hits on Julian day 0 (1970-01-01)
- $\beta_1$  = increase in log number of hits per unit day
- $\epsilon_i$  = variation due to everything we didn't measure

- when we take the natural log of outcomes and fit a regression model, the exponentiated coefficients estimate quantities based on the geometric means rather than the measured values

- $e^{E[\log(Y)]}$  = geometric mean of  $Y$

\* geometric means are defined as

$$e^{\frac{1}{n} \sum_{i=1}^n \log(y_i)} = \left( \prod_{i=1}^n y_i \right)^{1/n}$$

which is the estimate for the **population geometric mean**

- \* as we collect infinite amount of data,  $\prod_{i=1}^n y_i)^{1/n} \rightarrow E[\log(Y)]$
- $e^{\beta_0}$  = estimated geometric mean hits on day 0
- $e^{\beta_1}$  = estimated relative increase or decrease in geometric mean hits per day
- **Note:** not we can not take the natural log of zero counts, so often we need to adding a constant (i.e. 1) to construct the log model
  - \* adding the constant changes the interpretation of coefficient slightly
  - \*  $e^{\beta_1}$  is now the relative increase or decrease in geometric mean hits + 1 per day
- the expected outcome is

$$E[\log(NH_i|JD_i, \beta_0, \beta_1)] = \beta_0 + \beta_1 JD_i$$

```
round(exp(coef(lm(I(log(gaData$visits + 1)) ~ gaData$julian))), 5)
```

```
##   (Intercept) gaData$julian
##       0.00000     1.00231
```

- as we can see from above, the daily increase in hits is 0.2%

### Example - Poisson Regression

- the Poisson model can be constructed as log of the mean

$$\log(E[NH_i|JD_i, \beta_0, \beta_1]) = \beta_0 + \beta_1 JD_i$$

or in other form

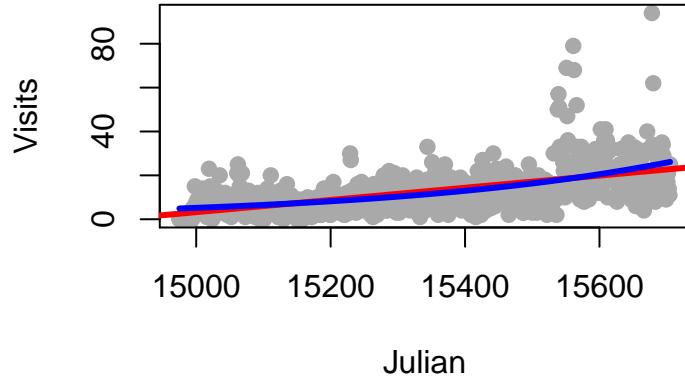
$$E[NH_i|JD_i, \beta_0, \beta_1] = \exp(\beta_0 + \beta_1 JD_i)$$

- $NH_i$  = number of hits to the website
- $JD_i$  = day of the year (Julian day)
- $\beta_0$  = expected number of hits on Julian day 0 (1970-01-01)
- $\beta_1$  = expected increase in number of hits per unit day
- **Note:** Poisson model differs from the log outcome model in that the coefficients are interpreted naturally as expected value of outcome where as the log model is interpreted on the log scale of outcome
- we can transform the Poisson model to

$$E[NH_i|JD_i, \beta_0, \beta_1] = \exp(\beta_0 + \beta_1 JD_i) = \exp(\beta_0) \exp(\beta_1 JD_i)$$

- $\beta_1 = E[NH_i|JD_i + 1, \beta_0, \beta_1] - E[NH_i|JD_i, \beta_0, \beta_1]$
- $\beta_1$  can therefore be interpreted as the **relative** increase/decrease in web traffic hits per one day increase
- `glm(outcome~predictor, family = "poisson")` = performs Poisson regression

```
# plot visits vs dates
plot(gaData$julian,gaData$visits,pch=19,col="darkgrey",xlab="Julian",ylab="Visits")
# construct Poisson regression model
glm1 <- glm(gaData$visits ~ gaData$julian,family="poisson")
# plot linear regression line in red
abline(lm1,col="red",lwd=3)
# plot Poisson regression line in blue
lines(gaData$julian,glm1$fitted,col="blue",lwd=3)
```

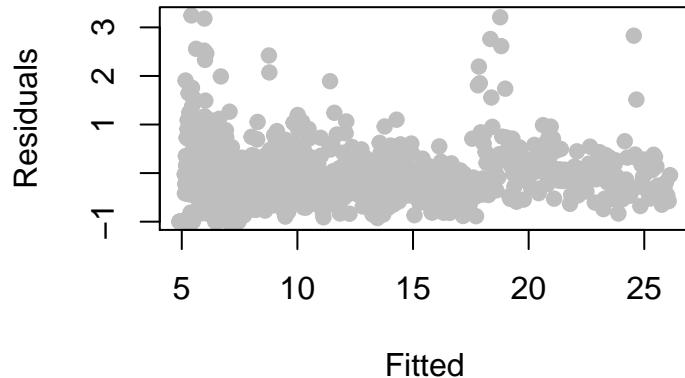


- Note: the Poisson fit is non-linear since it is linear only on the log of the mean scale

### Example - Robust Standard Errors with Poisson Regression

- variance of the Poisson distribution is defined to be the mean of the distribution, so we would expect the variance to increase with higher values of  $X$
- below is the residuals vs fitted value plot for the Poisson regression model

```
# plot residuals vs fitted values
plot(glm1$fitted,glm1$residuals,pch=19,col="grey",ylab="Residuals",xlab="Fitted")
```



- as we can see from above, the residuals don't appear to be increasing with higher fitted values
- even if the mean model is correct in principle, there could always be a certain degree of **model mis-specification**
- to account for mis-specifications for the model, we can use
  1. `glm(outcome~predictor, family = "quasi-poisson")` = introduces an additional multiplicative factor  $\phi$  to denominator of model so that the variance is  $\phi\mu$  rather than just  $\mu$  (see [Variances and Quasi-Likelihoods](#))
  2. more generally, *robust standard errors* (effectively constructing wider confidence intervals) can be used
- **model agnostic standard errors**, implemented through the `sandwich` package, is one way to calculate the robust standard errors
  - algorithm assumes the mean relationship is specified correctly and attempts to get a general estimate of the variance that isn't highly dependent on the model

- it uses assumption of large sample sizes and asymptotics to estimate the confidence intervals that is robust to model mis-specification
- **Note:** more information can be found at <http://stackoverflow.com/questions/3817182/vcovhc-and-confidence-interval>

```
# load sandwich package
library(sandwich)
# compute
confint.agnostic <- function (object, parm, level = 0.95, ...)
{
  cf <- coef(object); pnames <- names(cf)
  if (missing(parm))
    parm <- pnames
  else if (is.numeric(parm))
    parm <- pnames[parm]
  a <- (1 - level)/2; a <- c(a, 1 - a)
  pct <- stats:::format.perc(a, 3)
  fac <- qnorm(a)
  ci <- array(NA, dim = c(length(parm), 2L), dimnames = list(parm,
    pct))
  ses <- sqrt(diag(sandwich::vcovHC(object)))[parm]
  ci[] <- cf[parm] + ses %o% fac
  ci
}
# regular confidence interval from Poisson Model
confint(glm1)

##                2.5 %      97.5 %
## (Intercept) -34.346577587 -31.159715656
## gaData$julian  0.002190043  0.002396461

# model agnostic standard errors
confint.agnostic(glm1)

##                2.5 %      97.5 %
## (Intercept) -36.362674594 -29.136997254
## gaData$julian  0.002058147  0.002527955
```

- as we can see from above, the robust standard error produced slightly wider confidence intervals

### Example - Rates

- if we were to model the percentage of total web hits that are coming from the *Simply Statistics* blog, we could construct the following model

$$\begin{aligned} E[NHSS_i|JD_i, \beta_0, \beta_1]/NH_i &= \exp(\beta_0 + \beta_1 JD_i) \\ (\text{take log of both sides}) \log(E[NHSS_i|JD_i, \beta_0, \beta_1]) - \log(NH_i) &= \beta_0 + \beta_1 JD_i \\ (\text{move } \log(NH_i) \text{ to right side}) \log(E[NHSS_i|JD_i, \beta_0, \beta_1]) &= \underbrace{\log(NH_i)}_{\text{offset}} + \beta_0 + \beta_1 JD_i \end{aligned}$$

- when **offset** term is present in the Poisson model, the interpretation of the coefficients will be relative to the offset quantity

- it's important to recognize that the fitted response doesn't change
- **example:** to convert the outcome from daily data to hourly, we can add a factor 24 so that the model becomes

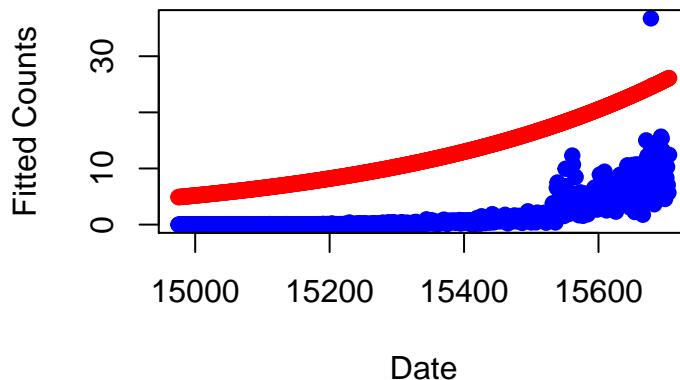
$$E[NHSS_i|JD_i, \beta_0, \beta_1]/24 = \exp(\beta_0 + \beta_1 JD_i)$$

(take log of both sides)  $\log(E[NHSS_i|JD_i, \beta_0, \beta_1]) - \log(24) = \beta_0 + \beta_1 JD_i$

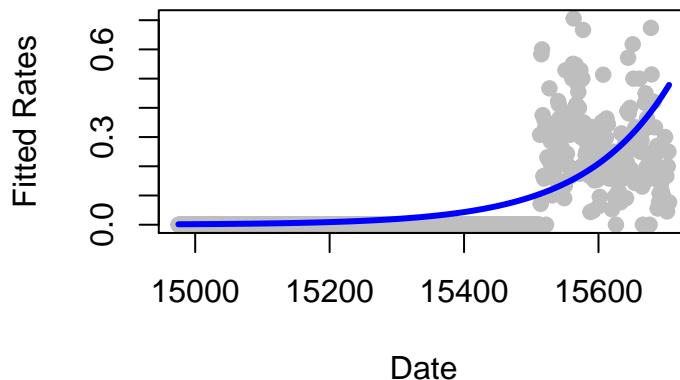
(move  $\log(24)$  to right side)  $\log(E[NHSS_i|JD_i, \beta_0, \beta_1]) = \log(24) + \log(NH_i) + \beta_0 + \beta_1 JD_i$

- back to the rates model, we fit the Poisson model now with an offset so that the model is interpreted with respect to the number of visits
  - `glm(outcome ~ predictor, offset = log(offset), family = "poisson")` = perform Poisson regression with offset
  - `glm(outcome ~ predictor + log(offset))` = produces the same result

```
# perform Poisson regression with offset for number of visits
glm2 <- glm(gaData$simplystats ~ julian(gaData$date), offset=log(visits+1),
             family="poisson", data=gaData)
# plot the fitted means (from simply statistics)
plot(julian(gaData$date), glm2$fitted, col="blue", pch=19, xlab="Date", ylab="Fitted Counts")
# plot the fitted means (total visit)
points(julian(gaData$date), glm1$fitted, col="red", pch=19)
```



```
# plot the rates for simply stats
plot(julian(gaData$date), gaData$simplystats/(gaData$visits+1), col="grey", xlab="Date",
      ylab="Fitted Rates", pch=19)
# plot the fitted rates for simply stats (visit/day)
lines(julian(gaData$date), glm2$fitted/(gaData$visits+1), col="blue", lwd=3)
```



- **Note:** we added 1 to the  $\log(\text{visits})$  to address 0 values

## Further Resources

- [Log-linear Models and Multi-way Tables](#)
- [Wikipedia on Poisson Regression](#)
- [Wikipedia on Overdispersion](#)
- [Regression Models for count data in R](#)
- **pscl package** -
  - often times in modeling counts, there may be more zero counts in the data than anticipated, which the regular Poisson model doesn't account for
  - the function `zeroinfl` fits zero inflated Poisson (ziP) models to such data

## Fitting Functions

- **scatterplot smoothing** = fitting functions (multiple linear models, piece-wise zig-zag lines) to data in the form  $Y_i = f(X_i) + \epsilon_i$
- consider the model

$$Y_i = \beta_0 + \beta_1 X_i + \sum_{k=1}^d (x_i - \xi_k)_+ \gamma_k + \epsilon_i$$

where  $(a)_+ = a$  if  $a > 0$  and 0 otherwise and  $\xi_1 \leq \dots \leq \xi_d$  are known **knot points**

- the mean function

$$E[Y_i] = \beta_0 + \beta_1 X_i + \sum_{k=1}^d (x_i - \xi_k)_+ \gamma_k$$

is continuous at the knot points

- for  $\xi_k = 5$ , the expected value for  $Y_i$  as  $x_i$  approaches 5 from the left is

$$\begin{aligned} E[Y_i]_{\xi=5|left} &= \beta_0 + \beta_1 x_i + (x_i - 5)_+ \gamma_k \\ (\text{since } x_i < 5) \quad E[Y_i]_{\xi=5|left} &= \beta_0 + \beta_1 x_i \end{aligned}$$

- the expected value for  $Y_i$  as  $x_i$  approaches 5 from the right is

$$\begin{aligned} E[Y_i]_{\xi=5|right} &= \beta_0 + \beta_1 x_i + (x_i - 5)_+ \gamma_k \\ (\text{since } x_i > 5) \quad E[Y_i]_{\xi=5|right} &= \beta_0 + \beta_1 x_i + (x_i - 5) \gamma_k \\ (\text{simplify}) \quad E[Y_i]_{\xi=5|right} &= \underbrace{\beta_0 - 5 \gamma_k}_{\text{intercept}} + \underbrace{(\beta_1 + \gamma_k)}_{\text{slope}} x_i \end{aligned}$$

- as we can see from above, the right side is just another line with different intercept and slope so as  $x$  approaches 5, both sides converge

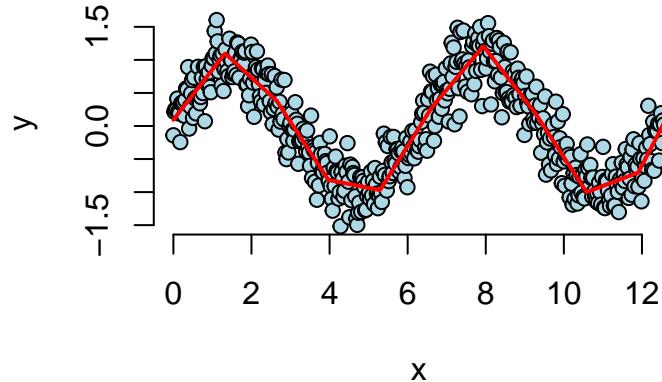
## Considerations

- **basis** = the collection of regressors
- single knot point terms can fit *hockey-stick-like* processes
- these bases can be used in GLMs (as an additional term/predictor) as well
- issue with these approaches is the **large** number of parameters introduced
  - requires some method of **regularization**, or penalize for large number of parameters (see **Practical Machine Learning** course)
  - introducing large number of knots have significant consequences

## Example - Fitting Piecewise Linear Function

```
# simulate data
n <- 500; x <- seq(0, 4 * pi, length = n); y <- sin(x) + rnorm(n, sd = .3)
# define 20 knot points
knots <- seq(0, 8 * pi, length = 20);
# define the ()+ function to only take the values that are positive after the knot pt
splineTerms <- sapply(knots, function(knot) (x > knot) * (x - knot))
# define the predictors as X and spline term
xMat <- cbind(x, splineTerms)
# fit linear models for y vs predictors
yhat <- predict(lm(y ~ xMat))
```

```
# plot data points (x, y)
plot(x, y, frame = FALSE, pch = 21, bg = "lightblue")
# plot fitted values
lines(x, yhat, col = "red", lwd = 2)
```



### Example - Fitting Piecewise Quadratic Function

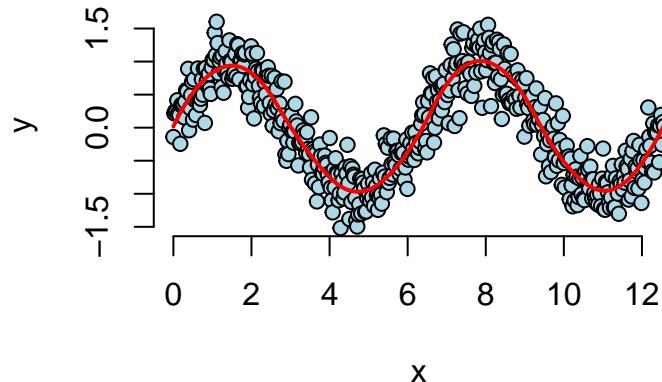
- adding squared terms makes it *continuous* AND *differentiable* at the knot points, and the model becomes

$$Y_i = \beta_0 + \beta_1 X_i + \beta_2 X_i^2 + \sum_{k=1}^d (x_i - \xi_k)_+^2 \gamma_k + \epsilon_i$$

where  $(a)_+^2 = a^2$  if  $a > 0$  and 0 otherwise

- adding cubic terms makes it twice continuously differentiable at the knot points, etcetera

```
# define the knot terms in the model
splineTerms <- sapply(knots, function(knot) (x > knot) * (x - knot)^2)
# define the predictors as x, x^2 and knot terms
xMat <- cbind(x, x^2, splineTerms)
# fit linear models for y vs predictors
yhat <- predict(lm(y ~ xMat))
# plot data points (x, y)
plot(x, y, frame = FALSE, pch = 21, bg = "lightblue")
# plot fitted values
lines(x, yhat, col = "red", lwd = 2)
```



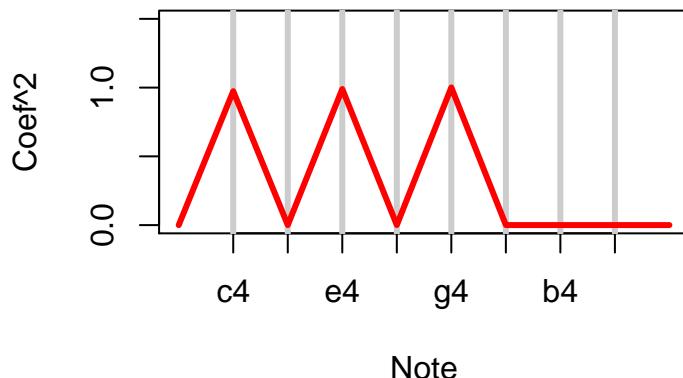
## Example - Harmonics using Linear Models

- **discrete Fourier transforms** = instance of linear regression model, use sin and cosine functions as basis to fit data
- to demonstrate this, we will generate 2 seconds of sound data using sin waves, simulate a chord, and apply linear regression to find out which notes are playing

```
# frequencies for white keys from c4 to c5
notes4 <- c(261.63, 293.66, 329.63, 349.23, 392.00, 440.00, 493.88, 523.25)
# generate sequence for 2 seconds
t <- seq(0, 2, by = .001); n <- length(t)
# define data for c4 e4 g4 using sine waves with their frequencies
c4 <- sin(2 * pi * notes4[1] * t); e4 <- sin(2 * pi * notes4[3] * t);
g4 <- sin(2 * pi * notes4[5] * t)
# define data for a chord and add a bit of noise
chord <- c4 + e4 + g4 + rnorm(n, 0, 0.3)
# generate profile data for all notes
x <- sapply(notes4, function(freq) sin(2 * pi * freq * t))
# fit the chord using the profiles for all notes
fit <- lm(chord ~ x - 1)
```

- after generating the data and running the linear regression, we can plot the results to see if the notes are correctly identified

```
# set up plot
plot(c(0, 9), c(0, 1.5), xlab = "Note", ylab = "Coef^2", axes = FALSE, frame = TRUE, type = "n")
# set up axes
axis(2)
axis(1, at = 1 : 8, labels = c("c4", "d4", "e4", "f4", "g4", "a4", "b4", "c5"))
# add vertical lines for each note
for (i in 1 : 8) abline(v = i, lwd = 3, col = grey(.8))
# plot the linear regression fits
lines(c(0, 1 : 8, 9), c(0, coef(fit)^2, 0), type = "l", lwd = 3, col = "red")
```

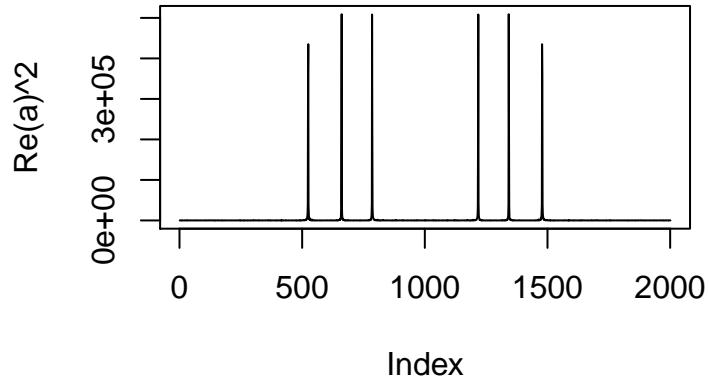


- as we can see from above, the correct notes were identified
- we can also use the **Fast Fourier Transforms** to identify the notes
  - `fft(data)` = performs fast Fourier transforms on provided data
  - `Re(data)` = subset to only the real components of the complex data

```

# perform fast fourier transforms on the chord matrix
a <- fft(chord)
# plot only the real components of the fft
plot(Re(a)^2, type = "l")

```



- **Note:** the algorithm checks for all possible notes at all frequencies it can detect, which is why the peaks are very high in magnitude
- **Note:** the symmetric display of the notes are due to periodic symmetries of the sine functions

# Practical Machine Learning Course Notes

Xing Su

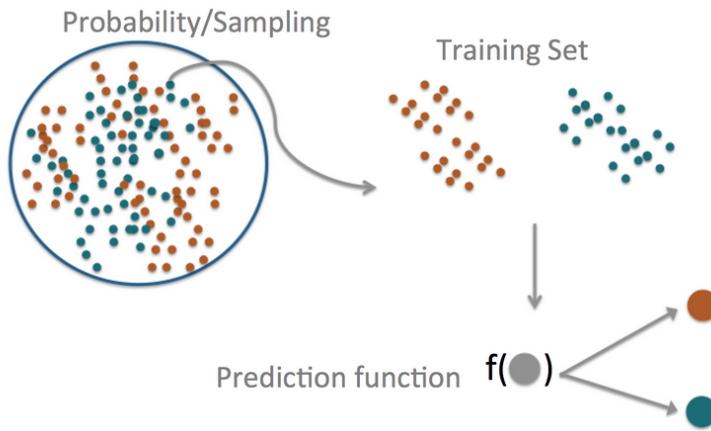
## Contents

Prediction . . . . .	3
In Sample vs Out of Sample Errors . . . . .	4
Prediction Study Design . . . . .	6
Sample Division Guidelines for Prediction Study Design . . . . .	7
Picking the Right Data . . . . .	8
Types of Errors . . . . .	9
Notable Measurements for Error – Binary Variables . . . . .	9
Notable Measurements for Error – Continuous Variables . . . . .	11
Receiver Operating Characteristic Curves . . . . .	12
Cross Validation . . . . .	14
Random Subsampling . . . . .	14
K-Fold . . . . .	15
Leave One Out . . . . .	15
<b>caret</b> Package ( <a href="#">tutorial</a> ) . . . . .	16
Data Slicing . . . . .	16
Training Options ( <a href="#">tutorial</a> ) . . . . .	19
Plotting Predictors ( <a href="#">tutorial</a> ) . . . . .	21
Preprocessing ( <a href="#">tutorial</a> ) . . . . .	25
Covariate Creation/Feature Extraction . . . . .	28
Creating Dummy Variables . . . . .	28
Removing Zero Covariates . . . . .	29
Creating Splines (Polynomial Functions) . . . . .	29
Multicore Parallel Processing . . . . .	30
Preprocessing with Principal Component Analysis (PCA) . . . . .	31
<b>prcomp</b> Function . . . . .	31
<b>caret</b> Package . . . . .	32
Predicting with Regression . . . . .	35
R Commands and Examples . . . . .	35
Prediction with Trees . . . . .	40
Process . . . . .	40
Measures of Impurity ( <a href="#">Reference</a> ) . . . . .	40

Constructing Trees with <code>caret</code> Package . . . . .	42
Bagging . . . . .	44
Bagging Algorithms . . . . .	45
Random Forest . . . . .	47
R Commands and Examples . . . . .	47
Boosting . . . . .	50
R Commands and Examples . . . . .	51
Model Based Prediction . . . . .	53
Linear Discriminant Analysis . . . . .	54
Naive Bayes . . . . .	56
Compare Results for LDA and Naive Bayes . . . . .	57
Model Selection . . . . .	58
Example: Training vs Test Error for Combination of Predictors . . . . .	58
Split Samples . . . . .	60
Decompose Expected Prediction Error . . . . .	60
Hard Thresholding . . . . .	61
Regularized Regression Concept ( <a href="#">Resource</a> ) . . . . .	62
Regularized Regression - Ridge Regression . . . . .	62
Regularized Regression - LASSO Regression . . . . .	64
Combining Predictors . . . . .	68
Example - Majority Vote . . . . .	68
Example - Model Ensembling . . . . .	69
Forecasting . . . . .	70
R Commands and Examples . . . . .	71
Unsupervised Prediction . . . . .	75
R Commands and Examples . . . . .	75

## Prediction

- **process for prediction** = population  $\rightarrow$  probability and sampling to pick set of data  $\rightarrow$  split into training and test set  $\rightarrow$  build prediction function  $\rightarrow$  predict for new data  $\rightarrow$  evaluate
  - *Note: choosing the right dataset and knowing what the specific question is are paramount to the success of the prediction algorithm (GoogleFlu failed to predict accurately when people's search habits changed)*

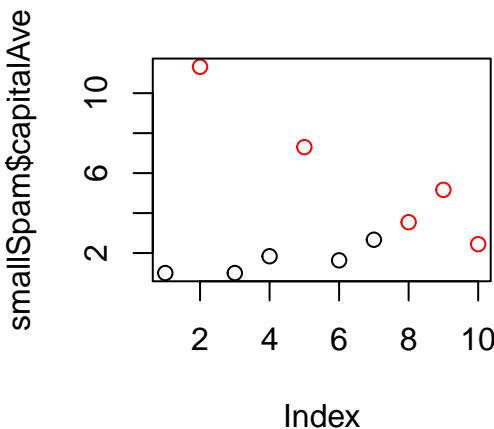


- **components of predictor** = question  $\rightarrow$  input data  $\rightarrow$  features (*extracting variables/characteristics*)  $\rightarrow$  algorithm  $\rightarrow$  parameters (*estimate*)  $\rightarrow$  evaluation
- **relative order of importance** = question (concrete/specific)  $>$  data (relevant)  $>$  features (properly extract)  $>$  algorithms
- **data selection**
  - *Note: "garbage in = garbage out"  $\rightarrow$  having the correct/relevant data will decide whether the model is successful*
  - data for what you are trying to predict is most helpful
  - more data  $\rightarrow$  better models (usually)
- **feature selection**
  - good features  $\rightarrow$  lead to data compression, retain relevant information, created based on expert domain knowledge
  - common mistakes  $\rightarrow$  automated feature selection (can yield good results but likely to behave inconsistently with slightly different data), not understanding/dealing with skewed data/outliers, throwing away information unnecessarily
- **algorithm selection**
  - matter less than one would expect
  - getting a sensible approach/algorithm will be the basis for a successful prediction
  - more complex algorithms can yield incremental improvements
  - ideally *interpretable* (simple to explain), accurate, scalable/fast (may leverage parallel computation)
- prediction is effectively about ***trade-offs***
  - find the correct balance between interpretability vs accuracy vs speed vs simplicity vs scalability
  - *interpretability* is especially important in conveying how features are used to predict outcome
  - scalability is important because for an algorithm to be of practical use, it needs to be implementable on large datasets without incurring large costs (computational complexity/time)

## In Sample vs Out of Sample Errors

- **in sample error** = error resulted from applying your prediction algorithm to the dataset you built it with
  - also known as *resubstitution error*
  - often optimistic (less than on a new sample) as the model may be tuned to error of the sample
- **out of sample error** = error resulted from applying your prediction algorithm to a new data set
  - also known as *generalization error*
  - out of sample error most important as it better evaluates how the model should perform
- in sample error < out of sample error
  - reason is ***over-fitting***: model too adapted/optimized for the initial dataset
    - \* data have two parts: *signal* vs *noise*
    - \* goal of predictor (should be simple/robust) = find signal
    - \* it is possible to design an accurate in-sample predictor, but it captures both signal and noise
    - \* predictor won't perform as well on new sample
  - often times it is better to give up a little accuracy for more robustness when predicting on new data
- **example**

```
# load data
library(kernlab); data(spam); set.seed(333)
# picking a small subset (10 values) from spam data set
smallSpam <- spam[sample(dim(spam)[1], size=10),]
# label spam = 2 and ham = 1
spamLabel <- (smallSpam$type=="spam")*1 + 1
# plot the capitalAve values for the dataset with colors differentiated by spam/ham (2 vs 1)
plot(smallSpam$capitalAve, col=spamLabel)
```



```
# first rule (over-fitting to capture all variation)
rule1 <- function(x){
  prediction <- rep(NA,length(x))
  prediction[x > 2.7] <- "spam"
  prediction[x < 2.40] <- "nonspam"
  prediction[(x >= 2.40 & x <= 2.45)] <- "spam"
  prediction[(x > 2.45 & x <= 2.70)] <- "nonspam"
```

```

        return(prediction)
    }
# tabulate results of prediction algorithm 1 (in sample error --> no error in this case)
table(rule1(smallSpam$capitalAve),smallSpam$type)

##  

##           nonspam  spam  

##   nonspam      5     0  

##   spam         0     5

# second rule (simple, setting a threshold)
rule2 <- function(x){
  prediction <- rep(NA,length(x))
  prediction[x > 2.8] <- "spam"
  prediction[x <= 2.8] <- "nonspam"
  return(prediction)
}
# tabulate results of prediction algorithm 2(in sample error --> 10% in this case)
table(rule2(smallSpam$capitalAve),smallSpam$type)

##  

##           nonspam  spam  

##   nonspam      5     1  

##   spam         0     4

# tabulate out of sample error for algorithm 1
table(rule1(spam$capitalAve),spam$type)

##  

##           nonspam  spam  

##   nonspam    2141  588  

##   spam       647 1225

# tabulate out of sample error for algorithm 2
table(rule2(spam$capitalAve),spam$type)

##  

##           nonspam  spam  

##   nonspam    2224  642  

##   spam       564 1171

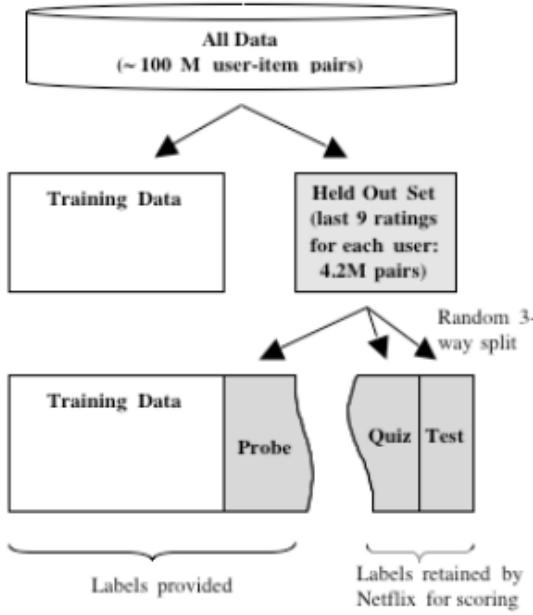
# accuracy and total correct for algorithm 1 and 2
rbind("Rule 1" = c(Accuracy = mean(rule1(spam$capitalAve)==spam$type),
  "Total Correct" = sum(rule1(spam$capitalAve)==spam$type)),
  "Rule 2" = c(Accuracy = mean(rule2(spam$capitalAve)==spam$type),
  "Total Correct" = sum(rule2(spam$capitalAve)==spam$type)))

##           Accuracy Total Correct
## Rule 1  0.7315801          3366
## Rule 2  0.7378831          3395

```

## Prediction Study Design

- **procedures**
  1. define error rate (type I/type II)
  2. split data into:
    - training, testing, validation (optional)
  3. pick features from the training set
    - use cross-validation
  4. pick prediction function (model) on the training set
    - use cross-validation
  5. if no validation set
    - apply **1 time** to test set
  6. if there is a validation set
    - apply to test set and refine
    - apply **1 time** to validation
  - **Note:** *it's important to hold out an untouched sample to accurately estimate the out of sample error rate*
- benchmarks (i.e. set all variables = 0) can help pinpoint/test the model to see what is wrong with the model
- avoid small sample sizes
  - consider binary outcomes (i.e. coin flip)
  - for  $n = 1$ , the probability of perfect classification (100% accuracy) is 50%
  - for  $n = 10$ , the probability of perfect classification (100% accuracy) is 0.1%
  - so it's important to have bigger samples so that when you do get a high accuracy, it may actually be a significant result and not just by chance
- **example: Netflix rating prediction competition**
  - split data between training and “held-out”
    - \* held-out included probe, quiz and test sets
    - \* probe is used to test the predictor built from the training dataset
    - \* quiz is used to realistically evaluate out of sample error rates
    - \* test is used to finally evaluate the validity of algorithm
  - important to not tune model to quiz set specifically



### Sample Division Guidelines for Prediction Study Design

- for large sample sizes
  - 60% training
  - 20% test
  - 20% validation
- for medium sample sizes
  - 60% training
  - 40% test
  - no validation set to refine model (to ensure test set is of sufficient size)
- for small sample sizes
  - carefully consider if there are enough sample to build a prediction algorithm
  - no test/validation sets
  - perform cross validation
  - report caveat of small sample size and highlight the fact that the prediction algorithm has never been tested for out of sample error
- there should always be a test/validation set that is held away and should ***NOT*** be looked at when building model
  - when complete, apply the model to the held-out set only one time
- ***randomly sample*** training and test sets
  - for data collected over time, build training set in chunks of times
- datasets must reflect structure of problem
  - if prediction evolves with time, split train/test sets in time chunks (known as *backtesting* in finance)
- subsets of data should reflect as much diversity as possible

## Picking the Right Data

- use like data to predict like
- to predict a variable/process X, use the data that's as closely related to X as possible
- weighting the data/variables by understanding and intuition can help to improve accuracy of prediction
- data properties matter -> knowing how the data connects to what you are trying to measure
- predicting on unrelated data is the most common mistake
  - if unrelated data must be used, be careful about interpreting the model as to why it works/doesn't work

## Types of Errors

- when discussing the outcome decided on by the algorithm, **Positive** = identified and **negative** = rejected
  - True positive** = correctly identified (predicted true when true)
  - False positive** = incorrectly identified (predicted true when false)
  - True negative** = correctly rejected (predicted false when false)
  - False negative** = incorrectly rejected (predicted false when true)
- example: medical testing*
  - True positive* = Sick people correctly diagnosed as sick
  - False positive* = Healthy people incorrectly identified as sick
  - True negative* = Healthy people correctly identified as healthy
  - False negative* = Sick people incorrectly identified as healthy

## Notable Measurements for Error – Binary Variables

- accuracy** = weights false positives/negatives equally
- concordance** = for multi-class cases,

$$\kappa = \frac{\text{accuracy} - P(e)}{1 - P(e)}$$

where

$$P(e) = \frac{TP + FP}{\text{total}} \times \frac{TP + FN}{\text{total}} + \frac{TN + FN}{\text{total}} \times \frac{FP + TN}{\text{total}}$$

		DISEASE	
		+	-
TEST	+	TP	FP
	-	FN	TN

Sensitivity	→ $\Pr(\text{positive test}   \text{disease})$
Specificity	→ $\Pr(\text{negative test}   \text{no disease})$
Positive Predictive Value	→ $\Pr(\text{disease}   \text{positive test})$
Negative Predictive Value	→ $\Pr(\text{no disease}   \text{negative test})$
Accuracy	→ $\Pr(\text{correct outcome})$

		DISEASE	
		+	-
TEST	+	TP	FP
	-	FN	TN

Sensitivity

$$\rightarrow TP / (TP+FN)$$

Specificity

$$\rightarrow TN / (FP+TN)$$

Positive Predictive Value

$$\rightarrow TP / (TP+FP)$$

Negative Predictive Value

$$\rightarrow TN / (FN+TN)$$

Accuracy

$$\rightarrow (TP+TN) / (TP+FP+FN+TN)$$

- *example*

- given that a disease has 0.1% prevalence in the population, we want to know what's probability of a person having the disease given the test result is positive? the test kit for the disease is 99% sensitive (most positives = disease) and 99% specific (most negatives = no disease)
- what about 10% prevalence?

		DISEASE	
		+	-
TEST	+	99	999
	-	1	98901

Sensitivity

$$\rightarrow 99 / (99+1) = 99\%$$

Specificity

$$\rightarrow 98901 / (999+98901) = 99\%$$

Positive Predictive Value

$$\rightarrow 99 / (99+999) \approx 9\%$$

Negative Predictive Value

$$\rightarrow 98901 / (1+98901) > 99.9\%$$

Accuracy

$$\rightarrow (99+98901) / 100000 = 99\%$$

		DISEASE	
		+	-
TEST	+	9900	900
	-	100	89100

<b>Sensitivity</b>	$\rightarrow 9900 / (9900+100) = 99\%$
<b>Specificity</b>	$\rightarrow 89100 / (900+89100) = 99\%$
<b>Positive Predictive Value</b>	$\rightarrow 9900 / (9900+900) \approx 92\%$
<b>Negative Predictive Value</b>	$\rightarrow 89100 / (100+89100) \approx 99.9\%$
<b>Accuracy</b>	$\rightarrow (9900+89100) / 100000 = 99\%$

### Notable Measurements for Error – Continuous Variables

- Mean squared error (MSE) =

$$\frac{1}{n} \sum_{i=1}^n (Prediction_i - Truth_i)^2$$

- Root mean squared error (RMSE) -

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (Prediction_i - Truth_i)^2}$$

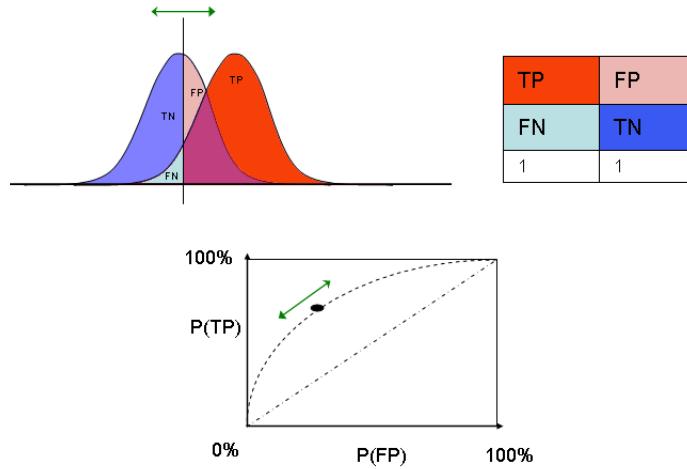
- in the same units as variable
- most commonly used error measure for continuous data
- is not an effective measure when there are outliers
  - \* one large value may significantly raise the RMSE

- median absolute deviation =

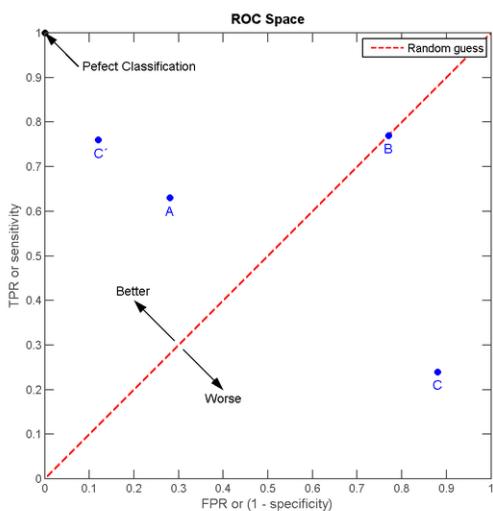
$$\text{median}(|Prediction_i - Truth_i|)$$

## Receiver Operating Characteristic Curves

- predictions for binary classification often are quantitative (i.e. probability, scale of 1 to 10)
  - different cutoffs/threshold of classification ( $> 0.8 \rightarrow$  one outcome) yield different results/predictions
  - **Receiver Operating Characteristic** curves are generated to compare the different outcomes

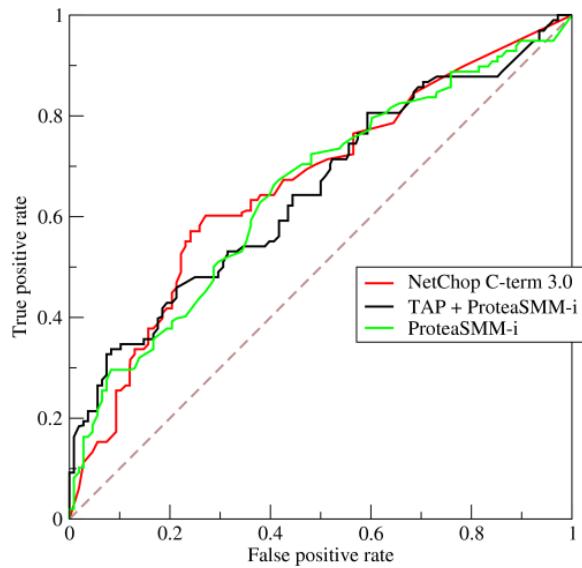


- ROC Curves
  - **x-axis** =  $1 - \text{specificity}$  (or, probability of false positive)
  - **y-axis** = sensitivity (or, probability of true positive)
  - **points plotted** = cutoff/combinations
  - **areas under curve** = quantifies whether the prediction model is viable or not
    - \* higher area  $\rightarrow$  better predictor
    - \* area = 0.5  $\rightarrow$  effectively random guessing (diagonal line in the ROC curve)
    - \* area = 1  $\rightarrow$  perfect classifier
    - \* area = 0.8  $\rightarrow$  considered good for a prediction algorithm



- *example*

- each point on the graph corresponds with a specificity and sensitivity



## Cross Validation

- **procedures**

1. split training set into sub-training/test sets
2. build model on sub-training set
3. evaluate on sub-test set
4. repeat and average estimated errors

- **result**

- we are able to fit/test various different models with different variables included to find the best one on the cross-validated test sets
- we are able to test out different types of prediction algorithms to use and pick the best performing one
- we are able to choose the parameters in prediction function and estimate their values
- *Note: original test set completely untouched, so when final prediction algorithm is applied, the result will be an unbiased measurement of the **out of sample accuracy** of the model*

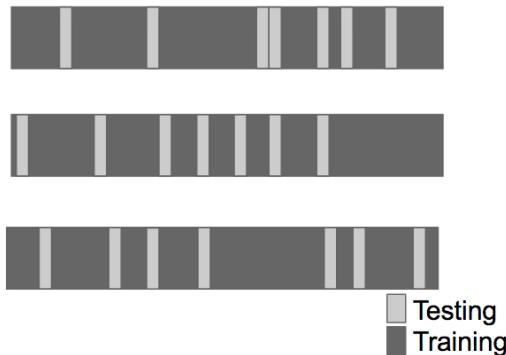
- **approaches**

- random subsampling
- K-fold
- leave one out

- **considerations**

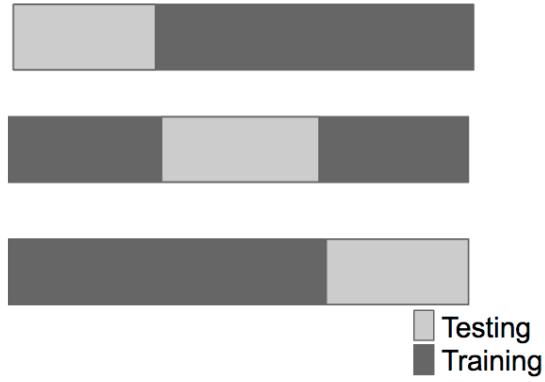
- for time series data data must be used in “chunks”
  - \* one time period might depend on all time periods previously (should not take random samples)
- if you cross-validate to pick predictors, the out of sample error rate may not be the most accurate and thus the errors should still be measured on independent data

## Random Subsampling



- a randomly sampled test set is subsetted out from the original training set
- the predictor is built on the remaining training data and applied to the test set
- the above are **three** random subsamplings from the same training set
- **considerations**
  - must be done *without replacement*
  - random sampling with replacement = *bootstrap*
    - \* underestimates of the error
    - \* can be corrected, but it is complicated ([0.632 Bootstrap](#))

## K-Fold



- break training set into  $K$  subsets (above is a 3-fold cross validation)
- build the model/predictor on the remaining training data in each subset and applied to the test subset
- rebuild the data  $K$  times with the training and test subsets and average the findings
- *considerations*
  - larger  $k$  = less bias, more variance
  - smaller  $k$  = more bias, less variance

## Leave One Out



- leave out exactly one sample and build predictor on the rest of training data
- predict value for the left out sample
- repeat for each sample

## **caret** Package ([tutorial](#))

- core functionality
  - preprocessing/cleaning data -> `preProcess()`
  - cross validation/data splitting -> `createDataPartition()`, `createResample()`, `createTimeSlices()`
  - train algorithms on training data and apply to test sets -> `train()`, `predict()`
  - model comparison (evaluate the accuracy of model on new data) -> `confusionMatrix()`
- machine learning algorithms in **caret** package
  - linear discriminant analysis
  - regression
  - naive Bayes
  - support vector machines
  - classification and regression trees
  - random forests
  - boosting
  - many others
- **caret** provides uniform framework to build/predict using different models
  - create objects of different classes for different algorithms, and **caret** package allows algorithms to be run the same way through `predict()` function

<b>obj</b>	<b>Class</b>	<b>Package</b>	<b>predict Function Syntax</b>
lda	MASS		<code>predict(obj)</code> (no options needed)
glm	stats		<code>predict(obj, type = "response")</code>
gbm	gbm		<code>predict(obj, type = "response", n.trees)</code>
mda	mda		<code>predict(obj, type = "posterior")</code>
rpart	rpart		<code>predict(obj, type = "prob")</code>
Weka	RWeka		<code>predict(obj, type = "probability")</code>
LogitBoost	caTools		<code>predict(obj, type = "raw", nIter)</code>

## Data Slicing

- `createDataPartition(y=data$var, times=1, p=0.75, list=FALSE)` -> creates data partitions using given variable
  - `y=data$var` = specifies what outcome/variable to split the data on
  - `times=1` = specifies number of partitions to create (number of data splitting performed)
  - `p=0.75` = percent of data that will be for training the model
  - `list=FALSE` = returns a matrix of indices corresponding to p% of the data (training set)
    - \* *Note: matrix is easier to subset the data with, so list = FALSE is generally what is used*
    - \* `list=TRUE` = returns a list of indices corresponding to p% of the data (training set)
  - the function effectively returns a list of indexes of the training set which can then be leveraged to subset the data
    - \* `training<-data[inTrain, ]` = subsets the data to training set only
    - \* `testing<-data[-inTrain, ]` = the rest of the data set can then be stored as the test set
  - *example*

```

# load packages and data
library(caret)

# create training set indexes with 75% of data
inTrain <- createDataPartition(y=spam$type, p=0.75, list=FALSE)

# subset spam data to training
training <- spam[inTrain,]

# subset spam data (the rest) to test
testing <- spam[-inTrain,]

# dimension of original and training dataset
rbind("original dataset" = dim(spam), "training set" = dim(training))

```

```

##          [,1] [,2]
## original dataset 4601   58
## training set     3451   58

```

- `createFolds(y=data$var, k=10, list=TRUE, returnTrain=TRUE)` = slices the data in to  $k$  folds for cross validation and returns  $k$  lists of indices
  - `y=data$var` = specifies what outcome/variable to split the data on
  - `k=10` = specifies number of folds to create (See  **$K$  Fold Cross Validation**)
    - \* each training set has approximately has  $\frac{k-1}{k}$ % of the data (in this case 90%)
    - \* each training set has approximately has  $\frac{1}{k}$ % of the data (in this case 10%)
  - `list=TRUE` = returns  $k$  list of indices that corresponds to the cross-validated sets
    - \* *Note:* the returned list conveniently splits the data into  $k$  datasets/vectors of indices, so `list=TRUE` is generally what is used
    - \* when the returned object is a list (called `folds` in the case), you can use `folds[[1]][1:10]` to access different elements from that list
    - \* `list=FALSE` = returns a vector indicating which of the  $k$  folds each data point belongs to (i.e. 1 - 10 is assigned for each of the data points in this case)
      - *Note:* these group values corresponds to test sets for each cross validation, which means everything else besides the marked points should be used for training
  - [only works when `list=T`] `returnTrain=TRUE` = returns the indices of the training sets
    - \* [default value when unspecified] `returnTrain=FALSE` = returns indices of the test sets
  - *example*

```

# create 10 folds for cross validation and return the training set indices
folds <- createFolds(y=spam$type, k=10, list=TRUE, returnTrain=TRUE)

# structure of the training set indices
str(folds)

```

```

## List of 10
## $ Fold01: int [1:4141] 1 2 3 4 6 7 8 9 10 12 ...
## $ Fold02: int [1:4140] 1 2 3 4 5 7 8 9 10 11 ...
## $ Fold03: int [1:4141] 1 3 4 5 6 7 8 9 10 11 ...
## $ Fold04: int [1:4141] 2 3 4 5 6 7 8 9 10 11 ...
## $ Fold05: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
## $ Fold06: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
## $ Fold07: int [1:4141] 1 2 3 4 5 6 8 9 11 12 ...
## $ Fold08: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
## $ Fold09: int [1:4141] 1 2 4 5 6 7 10 11 12 13 ...
## $ Fold10: int [1:4141] 1 2 3 5 6 7 8 9 10 11 ...

```

```

# return the test set indices instead
# note: returnTrain = FALSE is unnecessary as it is the default behavior
folds.test <- createFolds(y=spam$type, k=10, list=TRUE, returnTrain=FALSE)
str(folds.test)

## List of 10
## $ Fold01: int [1:460] 15 16 18 40 45 62 68 81 82 102 ...
## $ Fold02: int [1:459] 1 41 55 58 67 75 117 123 151 175 ...
## $ Fold03: int [1:461] 3 14 66 69 70 80 90 112 115 135 ...
## $ Fold04: int [1:460] 5 19 25 65 71 83 85 88 91 93 ...
## $ Fold05: int [1:460] 6 10 17 21 26 56 57 104 107 116 ...
## $ Fold06: int [1:459] 7 8 13 39 52 54 76 89 99 106 ...
## $ Fold07: int [1:461] 4 23 27 29 32 33 34 38 49 51 ...
## $ Fold08: int [1:460] 2 9 30 31 36 37 43 46 47 48 ...
## $ Fold09: int [1:461] 12 20 24 44 53 59 60 64 84 98 ...
## $ Fold10: int [1:460] 11 22 28 35 42 61 72 86 92 118 ...

# return first 10 elements of the first training set
folds[[1]][1:10]

```

```
## [1] 1 2 3 4 6 7 8 9 10 12
```

- `createResample(y=data$var, times=10, list=TRUE)` = create 10 resamplings from the given data with replacement
  - `list=TRUE` = returns list of n vectors that contain indices of the sample
    - \* *Note: each of the vectors is of length of the data, and contains indices*
  - `times=10` = number of samples to create

```

# create 10 resamples
resamples <- createResample(y=spam$type, times=10, list=TRUE)
# structure of the resamples (note some samples are repeated)
str(resamples)

```

```

## List of 10
## $ Resample01: int [1:4601] 1 4 4 4 7 8 12 13 13 14 ...
## $ Resample02: int [1:4601] 3 3 5 7 10 12 12 13 13 14 ...
## $ Resample03: int [1:4601] 1 2 2 3 4 5 8 10 11 12 ...
## $ Resample04: int [1:4601] 1 3 3 4 7 8 8 9 10 14 ...
## $ Resample05: int [1:4601] 2 4 5 6 7 7 8 8 9 12 ...
## $ Resample06: int [1:4601] 3 6 6 7 8 9 12 13 13 14 ...
## $ Resample07: int [1:4601] 1 2 2 5 5 6 7 8 9 10 ...
## $ Resample08: int [1:4601] 2 2 3 4 4 7 7 8 8 9 ...
## $ Resample09: int [1:4601] 1 4 7 8 8 9 12 13 15 15 ...
## $ Resample10: int [1:4601] 1 3 4 4 7 7 9 9 10 11 ...

```

- `createTimeSlices(y=data, initialWindow=20, horizon=10)` = creates training sets with specified window length and the corresponding test sets
  - `initialWindow=20` = number of consecutive values in each time slice/training set (i.e. values 1 - 20)
  - `horizon=10` = number of consecutive values in each predict/test set (i.e. values 21 - 30)

- `fixedWindow=FALSE` = training sets always start at the first observation
  - \* this means that the first training set would be 1 - 20, the second will be 1 - 21, third 1 - 22, etc.
  - \* but the test sets are still like before (21 - 30, 22 - 31, etc.)

```
# create time series data
tme <- 1:1000
# create time slices
folds <- createTimeSlices(y=tme, initialWindow=20, horizon=10)
# name of lists
names(folds)

## [1] "train" "test"

# first training set
folds$train[[1]]

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# first test set
folds$test[[1]]

## [1] 21 22 23 24 25 26 27 28 29 30
```

## Training Options ([tutorial](#))

- `train(y ~ x, data=df, method="glm")` = function to apply the machine learning algorithm to construct model from training data

```
# returns the arguments of the default train function
args(train.default)
```

```
## function (x, y, method = "rf", preProcess = NULL, ..., weights = NULL,
## metric = ifelse(is.factor(y), "Accuracy", "RMSE"), maximize = ifelse(metric ==
## "RMSE", FALSE, TRUE), trControl = trainControl(), tuneGrid = NULL,
## tuneLength = 3)
## NULL
```

- `train` function has a large set of parameters, below are the default options
  - `method="rf"` = default algorithm is random forest for training a given data set; `caret` contains a large number of algorithms
    - \* `names(getModelInfo())` = returns all the options for `method` argument
    - \* list of models and their information can be found [here](#)
  - `preProcess=NULL` = set preprocess options (see [\*Preprocessing\*](#))
  - `weights=NULL` = can be used to add weights to observations, useful for unbalanced distribution (a lot more of one type than another)
  - `metric=ifelse(is.factor(y), "Accuracy", "RMSE")` = default metric for algorithm is *Accuracy* for factor variables, and *RMSE*, or root mean squared error, for continuous variables

- \*  $Kappa$  = measure of concordance (see *Notable Measurements for Error – Binary Variables*)
- \*  $RSquared$  can also be used here as a metric, which represents  $R^2$  from regression models (only useful for linear models)
- `maximize=ifelse(metric=="RMSE", FALSE, TRUE)` = the algorithm should maximize *accuracy* and minimize *RMSE*
- `trControl=trainControl()` = training controls for the model, more details below
- `tuneGrid=NULL`
- `tuneLength=3`

```
# returns the default arguments for the trainControl object
args(trainControl)
```

```
## function (method = "boot", number = ifelse(grepl("cv", method),
##     10, 25), repeats = ifelse(grepl("cv", method), 1, number),
##     p = 0.75, initialWindow = NULL, horizon = 1, fixedWindow = TRUE,
##     verboseIter = FALSE, returnData = TRUE, returnResamp = "final",
##     savePredictions = FALSE, classProbs = FALSE, summaryFunction = defaultSummary,
##     selectionFunction = "best", preProcOptions = list(thresh = 0.95,
##         ICAcomp = 3, k = 5), index = NULL, indexOut = NULL, timingSamps = 0,
##     predictionBounds = rep(FALSE, 2), seeds = NA, adaptive = list(min = 5,
##         alpha = 0.05, method = "gls", complete = TRUE), allowParallel = TRUE)
## NULL
```

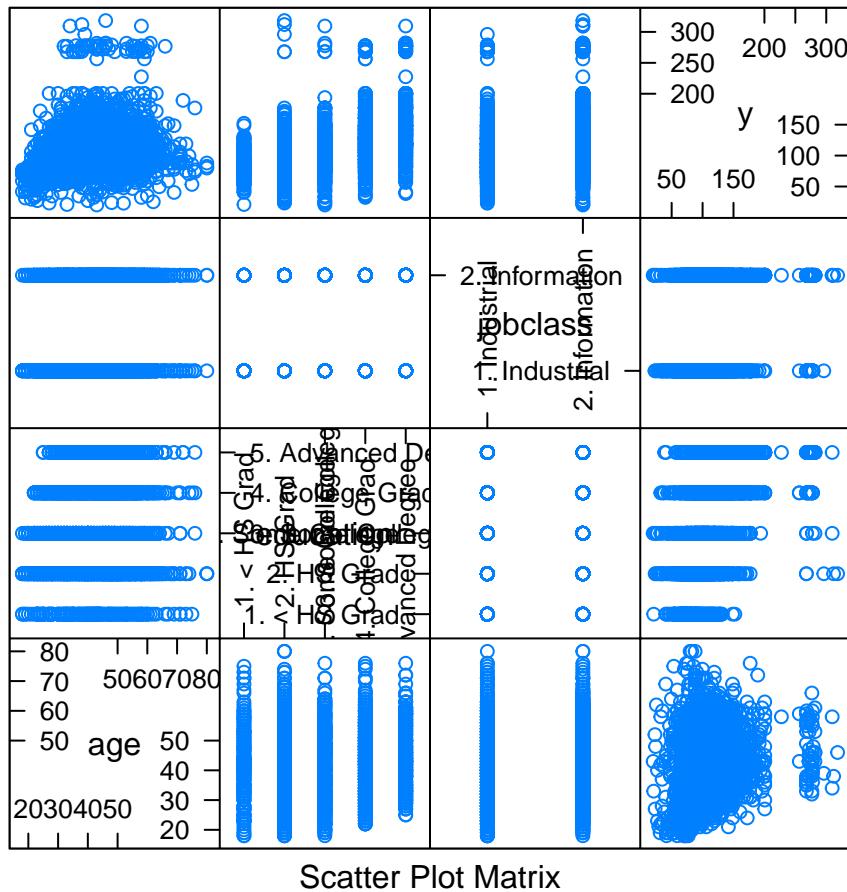
- `trainControl` creates an object that sets many options for how the model will be applied to the training data
  - *Note: the default values are listed below but you can use them to set the parameters to your discretion*
  - `method="boot"`
    - \* `"boot"` = bootstrapping (drawing with replacement)
    - \* `"boot632"` = bootstrapping with adjustment
    - \* `"cv"` = cross validation
    - \* `"repeatedcv"` = repeated cross validation
    - \* `"LOOCV"` = leave one out cross validation
  - `number=ifelse(grepl("cv", method), 10, 25)` = number of subsamples to take
    - \* `number=10` = default for any kind of cross validation
    - \* `number=25` = default for bootstrapping
    - \* *Note: number should be increased when fine-tuning model with large number of parameter*
  - `repeats=ifelse(grepl("cv", method), 1, number)` = numbers of times to repeat the subsampling
    - \* `repeats=1` = default for any cross validation method
    - \* `repeats=25` = default for bootstrapping
  - `p=0.75` = default percentage of data to create training sets
  - `initialWindow=NULL, horizon=1, fixedWindow=TRUE` = parameters for time series data
  - `verboseIter=FALSE` = print the training logs
  - `returnData=TRUE, returnResamp = "final",`
  - `savePredictions=FALSE` = save the predictions for each resample
  - `classProbs=FALSE` = return classification probabilities along with the predictions
  - `summaryFunction=defaultSummary` = default summary of the model,

- `preProcOptions=list(thresh = 0.95, ICAcomp = 3, k = 5)` = specifies preprocessing options for the model
- `predictionBounds=rep(FALSE, 2)` = specify the range of the predicted value
  - \* for numeric predictions, `predictionBounds=c(10, NA)` would mean that any value lower than 10 would be treated as 10 and no upper bounds
- `seeds=NA` = set the seed for the operation
  - \* *Note: setting this is important when you want to reproduce the same results when the `train` function is run*
- `allowParallel=TRUE` = sets for parallel processing/computations

## Plotting Predictors ([tutorial](#))

- it is important to only plot the data in the training set
  - using the test data may lead to over-fitting (model should not be adjusted to test set)
- goal of producing these exploratory plots = look for potential outliers, skewness, imbalances in outcome/predictors, and explainable groups of points/patterns
- `featurePlot(x=predictors, y=outcome, plot="pairs")` = short cut to plot the relationships between the predictors and outcomes

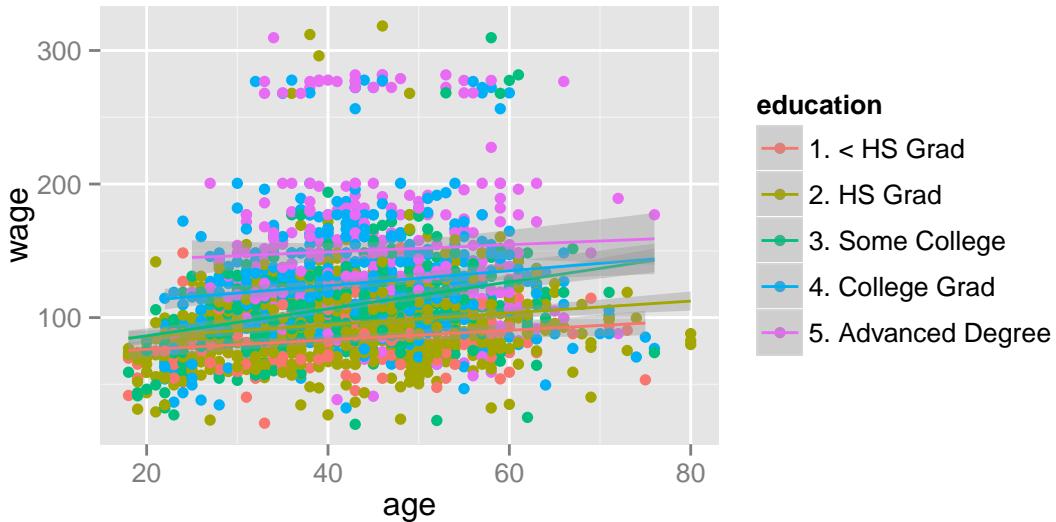
```
# load relevant libraries
library(ISLR); library(ggplot2);
# load wage data
data(Wage)
# create training and test sets
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]
testing <- Wage[-inTrain,]
# plot relationships between the predictors and outcome
featurePlot(x=training[,c("age","education","jobclass")], y = training$wage,plot="pairs")
```



Scatter Plot Matrix

- `qplot(age, wage, color=education, data=training)` = can be used to separate data by a factor variable (by coloring the points differently)
  - `geom_smooth()` = adds a regression line to the plots
  - `geom=c("boxplot", "jitter")` = specifies what kind of plot to produce, in this case both the boxplot and the point cloud

```
# qplot plus linear regression lines
qplot(age,wage,colour=education,data=training)+geom_smooth(method='lm',formula=y~x)
```



- `cut2(variable, g=3)` = creates a new factor variable by cutting the specified variable into n groups (3 in this case) based on percentiles
  - *Note: cut2 function is part of the Hmisc package, so library(Hmisc) must be run first*
  - this variable can then be used to tabulate/plot the data
- `grid.arrange(p1, p2, ncol=2)` = ggplot2 function the print multiple graphs on the same plot
  - *Note: grid.arrange function is part of the gridExtra package, so library(gridExtra) must be run first*

```
# load Hmisc and gridExtra packages
library(Hmisc);library(gridExtra);
# cute the wage variable
cutWage <- cut2(training$wage,g=3)
# plot the boxplot
p1 <- qplot(cutWage,age, data=training,fill=cutWage,
            geom=c("boxplot"))
# plot boxplot and point clusters
p2 <- qplot(cutWage,age, data=training,fill=cutWage,
            geom=c("boxplot","jitter"))
# plot the two graphs side by side
grid.arrange(p1,p2,ncol=2)
```



- `table(cutVariable, data$var2)` = tabulates the cut factor variable vs another variable in the dataset
- `prop.table(table, margin=1)` = converts a table to a proportion table
  - `margin=1` = calculate the proportions based on the rows
  - `margin=2` = calculate the proportions based on the columns

```
# tabulate the cutWage and jobclass variables
```

```
t <- table(cutWage,training$jobclass)
```

```
# print table
```

```
t
```

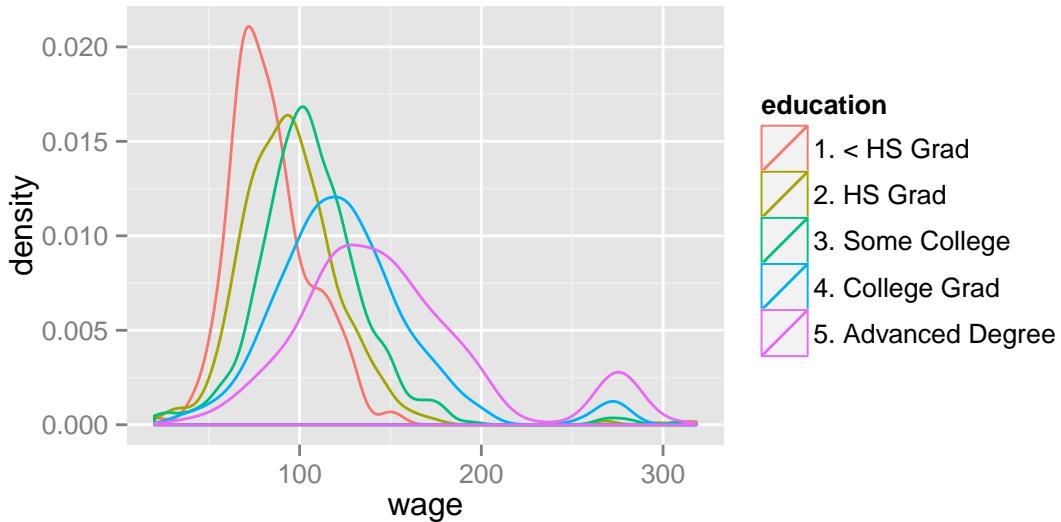
```
##  
## cutWage          1. Industrial 2. Information  
##   [ 20.1, 92.7)      445      256  
##   [ 92.7,119.1)      376      347  
##   [119.1,318.3]      269      409
```

```
# convert to proportion table based on the rows  
prop.table(t,1)
```

```
##  
## cutWage          1. Industrial 2. Information  
##   [ 20.1, 92.7)      0.6348074      0.3651926  
##   [ 92.7,119.1)      0.5200553      0.4799447  
##   [119.1,318.3]      0.3967552      0.6032448
```

- `qplot(var1, color=var2, data=training, geom="density")` = produces density plot for the given numeric and factor variables
  - effectively smoothed out histograms
  - provides for easy overlaying of groups of data
    - \* break different variables up by group and see how outcomes change between groups

```
# produce density plot
qplot(wage, colour=education, data=training, geom="density")
```



## Preprocessing ([tutorial](#))

- some predictors may have strange distributions (i.e. skewed) and may need to be transformed to be more useful for prediction algorithm
  - particularly true for model based algorithms → naive Bayes, linear discriminate analysis, linear regression
- **centering** = subtracting the observations of a particular variable by its mean
- **scaling** = dividing the observations of a particular variable by its standard deviation
- **normalizing** = centering and scaling the variable → effectively converting each observation to the number of standard deviations away from the mean
  - the distribution of the normalized variable will have a mean of 0 and standard deviation of 1
  - *Note: normalizing data can help remove bias and high variability, but may not be applicable in all cases*
- *Note: if a predictor/variable is standardized when training the model, the same transformations must be performed on the **test** set with the mean and standard deviation of the **train** variables*
  - this means that the mean and standard deviation of the normalized test variable will **NOT** be 0 and 1, respectively, but will be close
  - transformations must likely be imperfect but test/train sets must be processed the same way
- `train(y~x, data=training, preProcess=c("center", "scale"))` = preprocessing can be directly specified in the **train** function
  - `preProcess=c("center", "scale")` = normalize all predictors before constructing model
- `preProcess(trainingData, method=c("center", "scale"))` = function in the **caret** to standardize data
  - you can store the result of the **preProcess** function as an object and apply it to the **train** and **test** sets using the **predict** function

```

# load spam data
data(spam)

# create train and test sets
inTrain <- createDataPartition(y=spam$type, p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]

# create preProcess object for all predictors (" -58" because 58th = outcome)
preObj <- preProcess(training[,-58], method=c("center", "scale"))

# normalize training set
trainCapAveS <- predict(preObj, training[,-58])$capitalAve

# normalize test set using training parameters
testCapAveS <- predict(preObj, testing[,-58])$capitalAve

# compare results for capitalAve variable
rbind(train = c(mean = mean(trainCapAveS), std = sd(trainCapAveS)),
      test = c(mean(testCapAveS), sd(testCapAveS)))

```

```

##           mean      std
## train 6.362510e-18 1.000000
## test   7.548133e-02 1.633866

```

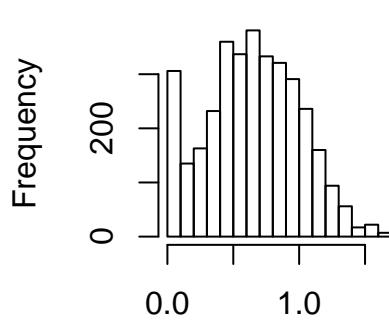
- `preprocess(data, method="BoxCox")` = applies BoxCox transformations to continuous data to help normalize the variables through maximum likelihood
  - *Note: note it assumes continuous values and DOES NOT deal with repeated values*
  - `qqnorm(processedVar)` = can be used to produce the Q-Q plot which compares the theoretical quantiles with the sample quantiles to see the normality of the data

```

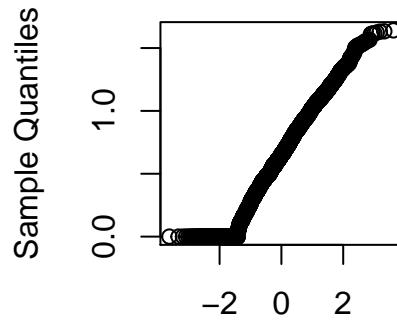
# set up BoxCox transforms
preObj <- preProcess(training[,-58], method=c("BoxCox"))
# perform preprocessing on training data
trainCapAveS <- predict(preObj, training[,-58])$capitalAve
# plot histogram and QQ Plot
# Note: the transformation definitely helped to
# normalize the data but it does not produce perfect result
par(mfrow=c(1,2)); hist(trainCapAveS); qqnorm(trainCapAveS)

```

**Histogram of trainCapAveS**



**Normal Q-Q Plot**



**trainCapAveS**

**Theoretical Quantiles**

- `preProcess(data, method="knnImpute")` = impute/estimate the missing data using **k nearest neighbors (knn)** imputation

- `knnImpute` = takes the k nearest neighbors from the missing value and averages the value to impute the missing observations
- *Note: most prediction algorithms are not build to handle missing data*

```
# Make some values NA
training$capAve <- training$capitalAve
selectNA <- rbinom(dim(training)[1], size=1, prob=0.05) == 1
training$capAve[selectNA] <- NA
# Impute and standardize
preObj <- preProcess(training[,-58], method="knnImpute")
capAve <- predict(preObj, training[,-58])$capAve
# Standardize true values
capAveTruth <- training$capitalAve
capAveTruth <- (capAveTruth - mean(capAveTruth)) / sd(capAveTruth)
# compute differences between imputed values and true values
quantile(capAve - capAveTruth)
```

```
##          0%         25%         50%         75%        100%
## -1.8242434686 -0.0005069839  0.0007532392  0.0013598891  0.4062295931
```

## Covariate Creation/Feature Extraction

- [level 1]: construct covariate (usable metric, feature) from raw data depends heavily on application
  - ideally we want to summarize data without too much information loss
  - examples
    - \* *text files*: frequency of words, frequency of phrases ([Google ngrams](#)), frequency of capital letters
    - \* *images*: edges, corners, blobs, ridges ([computer vision feature detection](#))
    - \* *webpages*: number and type of images, position of elements, colors, videos ([A/B Testing](#))
    - \* *people*: height, weight, hair color, sex, country of origin
  - generally, more knowledge and understanding you have of the system/data, the easier it will be to extract the summarizing features
    - \* when in doubt, more features is always safer → lose less information and the features can be filtered during model construction
  - this process can be automated (i.e. PCA) but generally have to be very careful, as one very useful feature in the training data set may not have as much effect on the test data set
  - **Note:** *science is the key here, Google “feature extraction for [data type]” for more guidance*
    - \* the goal is always to find the salient characteristics that are likely to be different from observation to observation
- [level 2]: construct new covariates from extracted covariate
  - generally transformations of features you extract from raw data
  - used more for methods like regression and support vector machines (SVM), whose accuracy depend more on the distribution of input variables
  - models like classification trees don't require as many complex covariates
  - best approach is through exploratory analysis (tables/plots)
  - should only be performed on the train dataset
  - new covariates should be added to data frames under recognizable names so they can be used later
  - `preProcess()` can be leveraged to handle creating new covariates
  - **Note:** *always be careful about over-fitting*

## Creating Dummy Variables

- convert factor variables to indicator/dummy variable → qualitative become quantitative
- `dummyVars(outcome~var, data=training)` = creates a dummy variable object that can be used through `predict` function to create dummy variables
  - `predict(dummyObj, newdata=training)` = creates appropriate columns to represent the factor variable with appropriate 0s and 1s
    - \* 2 factor variable → two columns which have 0 or 1 depending on the outcome
    - \* 3 factor variable → three columns which have 0, 0, and 1 representing the outcome
  - \* **Note:** *only one of the columns can have values of 1 for each observation*

```
# setting up data
inTrain <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# create a dummy variable object
dummies <- dummyVars(wage ~ jobclass, data=training)
# create the dummy variable columns
head(predict(dummies, newdata=training))
```

```

##      jobclass.1. Industrial jobclass.2. Information
## 231655          1          0
## 86582           0          1
## 11443           0          1
## 305706          1          0
## 160191          1          0
## 448410          1          0

```

## Removing Zero Covariates

- some variables have no variability at all (i.e. variable indicating if an email contained letters)
- these variables are not useful when we want to construct a prediction model
- `nearZeroVar(training, saveMetrics=TRUE)` = returns list of variables in training data set with information on frequency ratios, percent uniques, whether or not it has zero variance
  - `freqRatio` = ratio of frequencies for the most common value over second most common value
  - `percentUnique` = percentage of unique data points out of total number of data points
  - `zeroVar` = TRUE/FALSE indicating whether the predictor has only one distinct value
  - `nzv` = TRUE/FALSE indicating whether the predictor is a near zero variance predictor
  - *Note: when nzv = TRUE, those variables should be thrown out*

```

# print nearZeroVar table
nearZeroVar(training,saveMetrics=TRUE)

```

	<code>freqRatio</code>	<code>percentUnique</code>	<code>zeroVar</code>	<code>nzv</code>
## year	1.017647	0.33301618	FALSE	FALSE
## age	1.231884	2.85442436	FALSE	FALSE
## sex	0.000000	0.04757374	TRUE	TRUE
## maritl	3.329571	0.23786870	FALSE	FALSE
## race	8.480583	0.19029496	FALSE	FALSE
## education	1.393750	0.23786870	FALSE	FALSE
## region	0.000000	0.04757374	TRUE	TRUE
## jobclass	1.070936	0.09514748	FALSE	FALSE
## health	2.526846	0.09514748	FALSE	FALSE
## health_ins	2.209160	0.09514748	FALSE	FALSE
## logwage	1.011765	18.83920076	FALSE	FALSE
## wage	1.011765	18.83920076	FALSE	FALSE

## Creating Splines (Polynomial Functions)

- when you want to fit curves through the data, basis functions can be leveraged
- [splines package] `bs(data$var, df=3)` = creates 3 new columns corresponding to the var,  $\text{var}^2$ , and  $\text{var}^3$  terms
- `ns()` and `poly()` can also be used to generate polynomials
- `gam()` function can also be used and it allows for smoothing of multiple variables with different values for each variable
- *Note: the same polynomial operations must be performed on the test sets using the predict function*

```

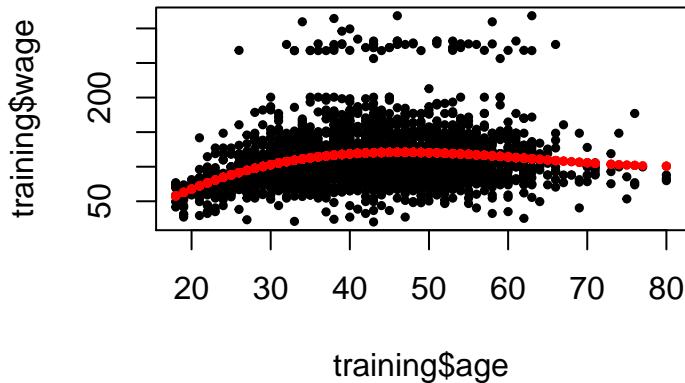
# load splines package
library(splines)

```

```

# create polynomial function
bsBasis <- bs(training$age,df=3)
# fit the outcome on the three polynomial terms
lm1 <- lm(wage ~ bsBasis,data=training)
# plot all age vs wage data
plot(training$age,training$wage,pch=19,cex=0.5)
# plot the fitted polynomial function
points(training$age,predict(lm1,newdata=training),col="red",pch=19,cex=0.5)

```



```

# predict on test values
head(predict(bsBasis,age=testing$age))

```

```

##          1         2         3
## [1,] 0.00000000 0.00000000 0.000000000
## [2,] 0.23685006 0.02537679 0.000906314
## [3,] 0.36252559 0.38669397 0.137491189
## [4,] 0.42616898 0.14823269 0.017186399
## [5,] 0.44221829 0.19539878 0.028779665
## [6,] 0.01793746 0.20448709 0.777050955

```

## Multicore Parallel Processing

- many of the algorithms in the **caret** package are computationally intensive
- since most of the modern machines have multiple cores on their CPUs, it is often wise to enable **multicore parallel processing** to expedite the computations
- **doMC** package is recommended to be used for **caret** computations ([reference](#))
  - `doMC::registerDoMC(cores=4)` = registers 4 cores for R to utilize
  - the number of cores you should specify depends on the CPU on your computer (system information usually contains the number of cores)
    - \* it's also possible to find the number of cores by directly searching for your CPU model number on the Internet
  - **Note:** once registered, you should see in your task manager/activity monitor that 4 "R Session" appear when you run your code

## Preprocessing with Principal Component Analysis (PCA)

- constructing a prediction model may not require every predictor
- ideally we want to capture the ***most variation*** with the ***least*** amount of variables
  - weighted combination of predictors may improve fit
  - combination needs to capture the ***most information***
- PCA is suited to do this and will help reduce number of predictors as well as reduce noise (due to averaging)
  - statistical goal = find new set of multivariate variables that are *uncorrelated* and explain as much variance as possible
  - data compression goal = find the best matrix created with fewer variables that explains the original data
  - PCA is most useful for linear-type models (GLM, LDA)
  - generally more difficult to interpret the predictors (complex weighted sums of variables)
  - **Note:** outliers are can be detrimental to PCA as they may represent a lot of variation in data
    - \* exploratory analysis (plots/tables) should be used to identify problems with the predictors
    - \* transformations with log/BoxCox may be helpful

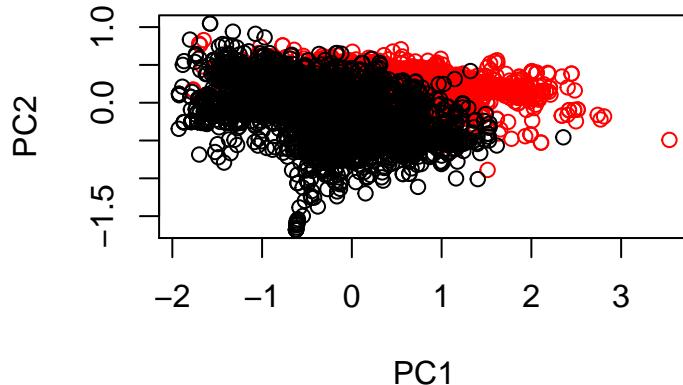
### prcomp Function

- `pr<-prcomp(data)` = performs PCA on all variables and returns a `prcomp` object that contains information about standard deviations and rotations
  - `pr$rotations` = returns eigenvectors for the linear combinations of all variables (coefficients that variables are multiplied by to come up with the principal components) -> how the principal components are created
  - often times, it is useful to take the `log` transformation of the variables and adding 1 before performing PCA
    - \* helps to reduce skewness or strange distribution in data
    - \*  $\log(0) = -\infty$ , so we add 1 to account for zero values
    - \* makes data more Gaussian
  - `plot(pr)` = plots the percent variation explained by the first 10 principal components (PC)
    - \* can be used to find the PCs that represent the most variation

```
# load spam data
data(spam)
# perform PCA on dataset
prComp <- prcomp(log10(spam[,-58]+1))
# print out the eigenvector/rotations first 5 rows and PCs
head(prComp$rotation[, 1:5], 5)
```

```
##          PC1         PC2         PC3         PC4         PC5
## make    0.019370409  0.0427855959 -0.01631961  0.02798232 -0.014903314
## address 0.010827343  0.0408943785  0.07074906 -0.01407049  0.037237531
## all     0.040923168  0.0825569578 -0.03603222  0.04563653  0.001222215
## num3d   0.006486834 -0.0001333549  0.01234374 -0.01005991 -0.001282330
## our     0.036963221  0.0941456085 -0.01871090  0.05098463 -0.010582039
```

```
# create new variable that marks spam as 2 and nospam as 1
typeColor <- ((spam$type=="spam")*1 + 1)
# plot the first two principal components
plot(prComp$x[,1],prComp$x[,2],col=typeColor,xlab="PC1",ylab="PC2")
```



### **caret Package**

- `pp<-preProcess(log10(training[,-58]+1),method="pca",pcaComp=2,thresh=0.8)` = perform PCA with `preProcess` function and returns the number of principal components that can capture the majority of the variation
  - creates a `preProcess` object that can be applied using `predict` function
  - `pcaComp=2` = specifies the number of principal components to compute (2 in this case)
  - `thresh=0.8` = threshold for variation captured by principal components
    - \* `thresh=0.95` = default value, which returns the number of principal components that are needed to capture 95% of the variation in data
- `predict(pp, training)` = computes new variables for the PCs (2 in this case) for the training data set
  - the results from `predict` can then be used as data for the prediction model
  - **Note:** the same PCA must be performed on the test set

```
# create train and test sets
inTrain <- createDataPartition(y=spam$type,p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]
# create preprocess object
preProc <- preProcess(log10(training[,-58]+1),method="pca",pcaComp=2)
# calculate PCs for training data
trainPC <- predict(preProc,log10(training[,-58]+1))
# run model on outcome and principle components
modelFit <- train(training$type ~ .,method="glm",data=trainPC)
# calculate PCs for test data
testPC <- predict(preProc,log10(testing[,-58]+1))
# compare results
confusionMatrix(testing$type,predict(modelFit,testPC))
```

```
## Confusion Matrix and Statistics
##
```

```

##           Reference
## Prediction nonspam spam
##   nonspam      656    41
##     spam        82   371
##
##           Accuracy : 0.893
##             95% CI : (0.8737, 0.9103)
##   No Information Rate : 0.6417
##   P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.7724
##   Mcnemar's Test P-Value : 0.0003101
##
##           Sensitivity : 0.8889
##           Specificity : 0.9005
##   Pos Pred Value : 0.9412
##   Neg Pred Value : 0.8190
##           Prevalence : 0.6417
##   Detection Rate : 0.5704
##   Detection Prevalence : 0.6061
##   Balanced Accuracy : 0.8947
##
##   'Positive' Class : nonspam
##

```

- alternatively, PCA can be directly performed with the `train` method
  - `train(outcome ~ ., method="glm", preProcess="pca", data=training)` = performs PCA first on the training set and then runs the specified model
    - \* effectively the same procedures as above (`preProcess -> predict`)
- *Note: in both cases, the PCs were able to achieve 90+% accuracy*

```

# construct model
modelFit <- train(training$type ~ ., method="glm", preProcess="pca", data=training)
# print results of model
confusionMatrix(testing$type, predict(modelFit, testing))

```

```

## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##   nonspam      668    29
##     spam        59   394
##
##           Accuracy : 0.9235
##             95% CI : (0.9066, 0.9382)
##   No Information Rate : 0.6322
##   P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.8379
##   Mcnemar's Test P-Value : 0.001992
##
##           Sensitivity : 0.9188

```

```
##          Specificity : 0.9314
##      Pos Pred Value : 0.9584
##      Neg Pred Value : 0.8698
##          Prevalence : 0.6322
##      Detection Rate : 0.5809
## Detection Prevalence : 0.6061
##      Balanced Accuracy : 0.9251
##
##      'Positive' Class : nonspam
##
```

## Predicting with Regression

- **prediction with regression** = fitting regression model (line) to data -> multiplying each variable by coefficients to predict outcome
- useful when the relationship between the variables can be modeled as linear
- the model is easy to implement and the coefficients are easy to interpret
- if the relationships are non-linear, the regression model may produce poor results/accuracy
  - *Note: linear regressions are generally used in combination with other models*

- **model**

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \dots + \beta_p X_{pi} + e_i$$

- where  $\beta_0$  is the intercept (when all variables are 0)
- $\beta_1, \beta_2, \dots, \beta_p$  are the coefficients
- $X_{1i}, X_{2i}, \dots, X_{pi}$  are the variables/covariates
- $e_i$  is the error
- $Y_i$  is the outcome

- **prediction**

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1i} + \hat{\beta}_2 X_{2i} + \dots + \hat{\beta}_p X_{pi}$$

- where  $\hat{\beta}_0$  is the estimated intercept (when all variables are 0)
- $\hat{\beta}_1, \hat{\beta}_2, \dots, \hat{\beta}_p$  are the estimated coefficients
- $X_{1i}, X_{2i}, \dots, X_{pi}$  are the variables/covariates
- $\hat{Y}_i$  is the **predicted outcome**

## R Commands and Examples

- `lm<-lm(y ~ x, data=train)` = runs a linear model of outcome y on predictor x -> univariate regression
  - `summary(lm)` = returns summary of the linear regression model, which will include coefficients, standard errors, t statistics, and p values
  - `lm(y ~ x1+x2+x3, data=train)` = run linear model of outcome y on predictors x1, x2, and x3
  - `lm(y ~ ., data=train)` = run linear model of outcome y on all predictors
- `predict(lm, newdata=df)` = use the constructed linear model to predict outcomes ( $\hat{Y}_i$ ) for the new values
  - `newdata` data frame must have the same variables (factors must have the same levels) as the training data
  - `newdata=test` = predict outcomes for the test set based on linear regression model from the training
  - *Note: the regression line will not be a perfect fit on the test set since it was constructed on the training set*
- RSME can be calculated to measure the accuracy of the linear model
  - *Note: RSME<sub>test</sub>, which estimates the out-of-sample error, is almost always **GREATER** than RSME<sub>train</sub>*

```
# load data
data(faithful)
# create train and test sets
inTrain <- createDataPartition(y=faithful$waiting, p=0.5, list=FALSE)
trainFaith <- faithful[inTrain,]; testFaith <- faithful[-inTrain,]
# build linear model
```

```

lm1 <- lm(eruptions ~ waiting,data=trainFaith)
# print summary of linear model
summary(lm1)

## 
## Call:
## lm(formula = eruptions ~ waiting, data = trainFaith)
##
## Residuals:
##       Min     1Q   Median     3Q    Max
## -1.30541 -0.35970  0.04711  0.37624  1.07683
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -1.870699  0.225663  -8.29 1.01e-13 ***
## waiting      0.075674  0.003137   24.12 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5024 on 135 degrees of freedom
## Multiple R-squared:  0.8117, Adjusted R-squared:  0.8103 
## F-statistic: 581.8 on 1 and 135 DF,  p-value: < 2.2e-16

```

```

# predict eruptions for new waiting time
newdata <- data.frame(waiting=80)
predict(lm1,newdata)

```

```

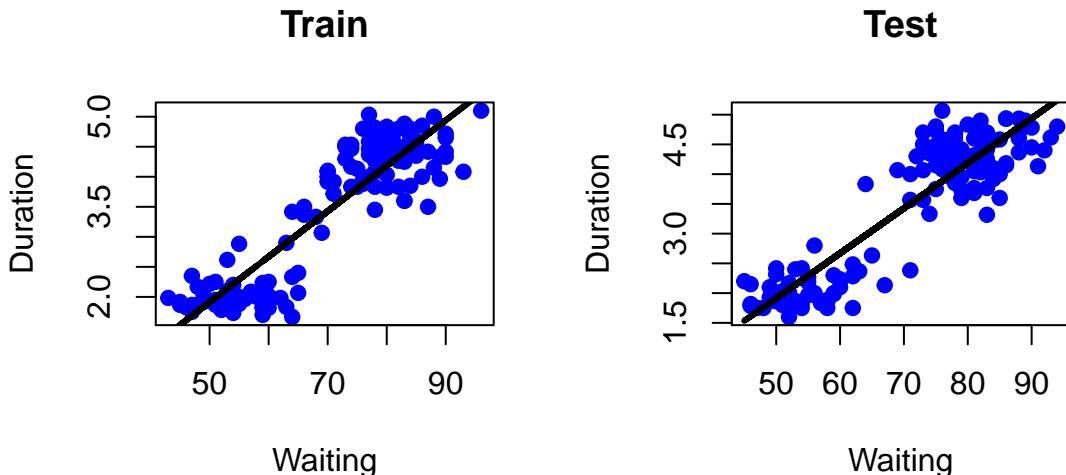
##          1
## 4.183192

```

```

# create 1 x 2 panel plot
par(mfrow=c(1,2))
# plot train data with the regression line
plot(trainFaith$waiting,trainFaith$eruptions,pch=19,col="blue",xlab="Waiting",
      ylab="Duration", main = "Train")
lines(trainFaith$waiting,predict(lm1),lwd=3)
# plot test data with the regression line
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue",xlab="Waiting",
      ylab="Duration", main = "Test")
lines(testFaith$waiting,predict(lm1,newdata=testFaith),lwd=3)

```

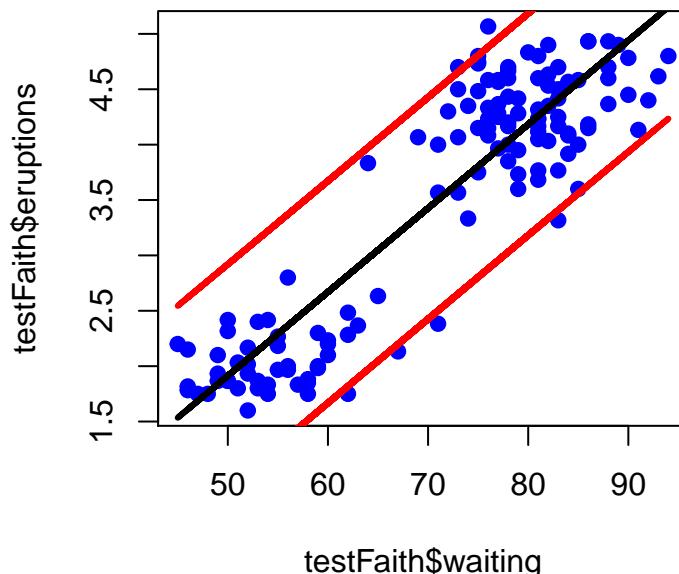


```
# Calculate RMSE on training and test sets
c(trainRMSE = sqrt(sum((lm1$fitted-trainFaith$eruptions)^2)),
  testRMSE = sqrt(sum((predict(lm1,newdata=testFaith)-testFaith$eruptions)^2)))
```

```
## trainRMSE  testRMSE
## 5.836980  5.701161
```

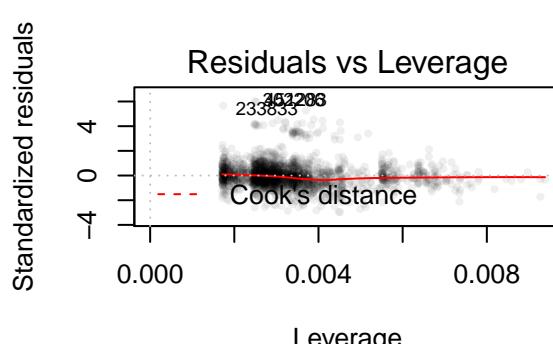
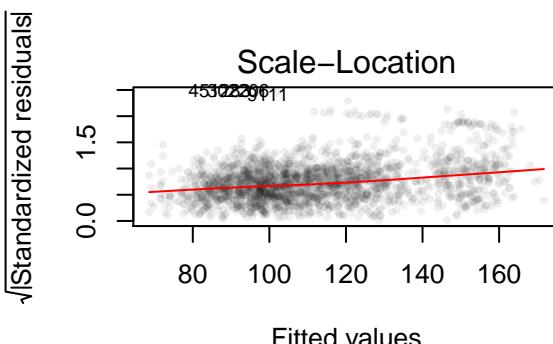
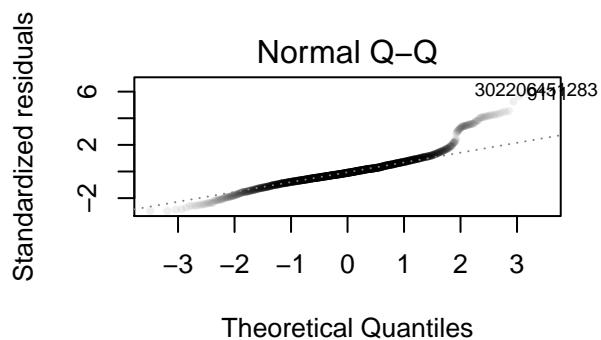
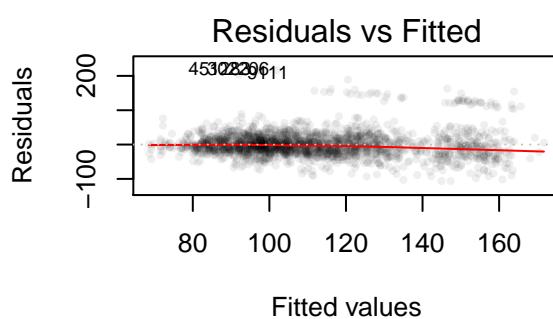
- `pi<-predict(lm, newdata=test, interval="prediction")` = returns 3 columns for `fit` (predicted value, same as before), `lwr` (lower bound of prediction interval), and `upr` (upper bound of prediction interval)
  - `matlines(x, pi, type="l")` = plots three lines, one for the linear fit and two for upper/lower prediction interval bounds

```
# calculate prediction interval
pred1 <- predict(lm1,newdata=testFaith,interval="prediction")
# plot data points (eruptions, waiting)
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue")
# plot fit line and prediction interval
matlines(testFaith$waiting,pred1,type="l",,col=c(1,2,2),lty = c(1,1,1), lwd=3)
```



- `lm <- train(y ~ x, method="lm", data=train)` = run linear model on the training data → identical to `lm` function
  - `summary(lm$finalModel)` = returns summary of the linear regression model, which will include coefficients, standard errors, *t* statistics, and p values → identical to `summary(lm)` for a `lm` object
  - `train(y ~ ., method="lm", data=train)` = run linear model on all predictors in training data
    - \* multiple predictors (dummy/indicator variables) are created for factor variables
  - `plot(lm$finalModel)` = construct 4 diagnostic plots for evaluating the model
    - \* *Note: more information on these plots can be found at ?plot.lm*
    - \* **Residual vs Fitted**
    - \* **Normal Q-Q**
    - \* **Scale-Location**
    - \* **Residual vs Leverage**

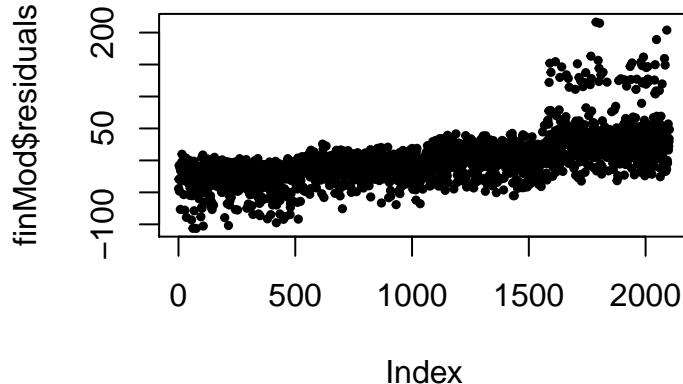
```
# create train and test sets
inTrain <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# fit linear model for age jobclass and education
modFit<- train(wage ~ age + jobclass + education, method = "lm", data=training)
# store final model
finMod <- modFit$finalModel
# set up 2 x 2 panel plot
par(mfrow = c(2, 2))
# construct diagnostic plots for model
plot(finMod, pch=19, cex=0.5, col="#00000010")
```



- plotting residuals by index can be helpful in showing missing variables

- `plot(finMod$residuals)` = plot the residuals against index (row number)
- if there's a trend/pattern in the residuals, it is highly likely that another variable (such as age/time) should be included
  - \* residuals should not have relationship to index

```
# plot residual by index
plot(finMod$residuals,pch=19,cex=0.5)
```



- here the residuals increase linearly with the index, and the highest residuals are concentrated in the higher indexes, so there must be a missing variable

## Prediction with Trees

- **prediction with trees** = iteratively split variables into groups (effectively constructing decision trees)  
-> produces nonlinear model
  - the classification tree uses interactions between variables -> the ultimate groups/leafs may depend on many variables
- the result (tree) is easy to interpret, and generally performs better predictions than regression models when the relationships are ***non-linear***
- transformations less important -> monotone transformations (order unchanged, such as log) will produce same splits
- trees can be used for regression problems as well and use RMSE as *measure of impurity*
- however, without proper cross-validation, the model can be ***over-fitted*** (especially with large number of variables) and results may be variable from one run to the next
  - it is also harder to estimate the uncertainty of the model
- **party**, **rpart**, **tree** packages can all build trees

## Process

1. start with all variables in one group
2. find the variable that best splits the outcomes into two groups
3. divide data into two groups (*leaves*) based on the split performed (*node*)
4. within each split, find variables to split the groups again
5. continue this process until all groups are sufficiently small/homogeneous/“pure”

## Measures of Impurity (Reference)

$$\hat{p}_{mk} = \frac{\sum_i^m \mathbb{1}(y_i = k)}{N_m}$$

- $\hat{p}_{mk}$  is the probability of the objects in group  $m$  to take on the classification  $k$
- $N_m$  is the size of the group
- **Misclassification Error**

$$1 - \hat{p}_{m \ k(m)}$$

where  $k(m)$  is the most common classification/group

- 0 = perfect purity
- 0.5 = no purity

\* *Note: it is not 1 here because when  $\hat{p}_{m \ k(m)} < 0.5$  or there's not predominant classification for the objects, it means the group should be further subdivided until there's a majority*

- **Gini Index**

$$\sum_{k \neq k'} \hat{p}_{mk} \times \hat{p}_{mk'} = \sum_{k=1}^K \hat{p}_{mk} (1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^K p_{mk}^2$$

- 0 = perfect purity
- 0.5 = no purity

- Deviance

$$-\sum_{k=1}^K \hat{p}_{mk} \log_e \hat{p}_{mk}$$

- 0 = perfect purity
- 1 = no purity

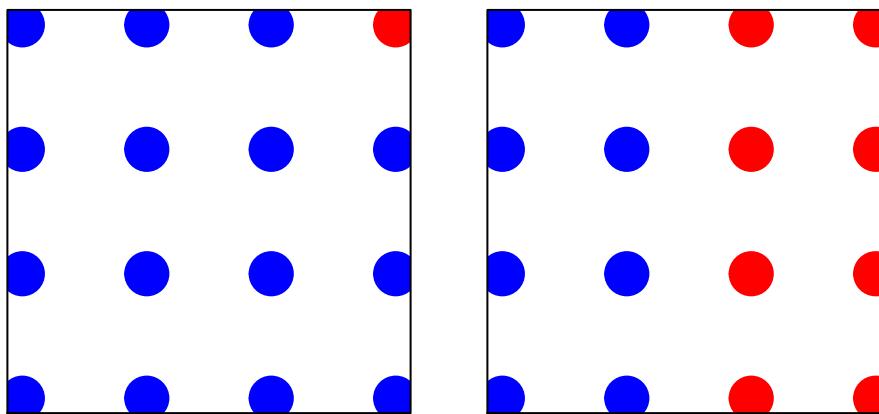
- Information Gain

$$-\sum_{k=1}^K \hat{p}_{mk} \log_2 \hat{p}_{mk}$$

- 0 = perfect purity
- 1 = no purity

- example

```
# set margin and seed
par(mar=c(1,1,1,1), mfrow = c(1, 2)); set.seed(1234);
# simulate data
x = rep(1:4,each=4); y = rep(1:4,4)
# plot first scenario
plot(x,y,xaxt="n",yaxt="n",cex=3,col=c(rep("blue",15),rep("red",1)),pch=19)
# plot second scenario
plot(x,y,xaxt="n",yaxt="n",cex=3,col=c(rep("blue",8),rep("red",8)),pch=19)
```



- left graph

- Misclassification:  $\frac{1}{16} = 0.06$
- Gini:  $1 - [(\frac{1}{16})^2 + (\frac{15}{16})^2] = 0.12$
- Information:  $-[\frac{1}{16} \times \log_2(\frac{1}{16}) + \frac{15}{16} \times \log_2(\frac{1}{16})] = 0.34$

- right graph

- Misclassification:  $\frac{8}{16} = 0.5$
- Gini:  $1 - [(\frac{8}{16})^2 + (\frac{8}{16})^2] = 0.5$
- Information:  $-[\frac{8}{16} \times \log_2(\frac{8}{16}) + \frac{8}{16} \times \log_2(\frac{8}{16})] = 1$

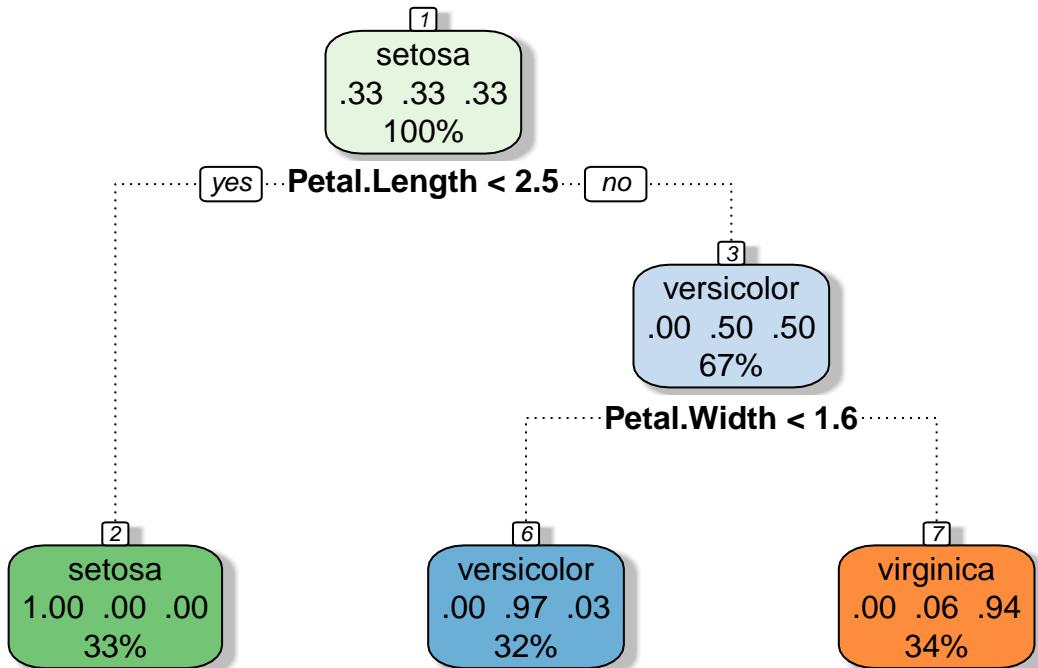
## Constructing Trees with caret Package

- `tree<-train(y ~ ., data=train, method="rpart")` = constructs trees based on the outcome and predictors
  - produces an `rpart` object, which can be used to `predict` new/test values
  - `print(tree$finalModel)` = returns text summary of all nodes/splits in the tree constructed
- `plot(tree$finalModel, uniform=TRUE)` = plots the classification tree with all nodes/splits
  - [rattle package] `fancyRpartPlot(tree$finalModel)` = produces more readable, better formatted classification tree diagrams
  - each split will have the condition/node in bold and the splits/leafs on the left and right sides following the “yes” or “no” indicators
    - \* “yes” -> go left
    - \* “no” -> go right

```
# load iris data set
data(iris)
# create test/train data sets
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
# fit classification tree as a model
modFit <- train(Species ~ ., method="rpart", data=training)
# print the classification tree
print(modFit$finalModel)

## n= 105
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 105 70 setosa (0.33333333 0.33333333 0.33333333)
##    2) Petal.Length< 2.45 35 0 setosa (1.00000000 0.00000000 0.00000000) *
##    3) Petal.Length>=2.45 70 35 versicolor (0.00000000 0.50000000 0.50000000)
##      6) Petal.Width< 1.65 34 1 versicolor (0.00000000 0.97058824 0.02941176) *
##      7) Petal.Width>=1.65 36 2 virginica (0.00000000 0.05555556 0.94444444) *

# plot the classification tree
rattle::fancyRpartPlot(modFit$finalModel)
```



Rattle 2015-Mar-04 22:13:01 Xing

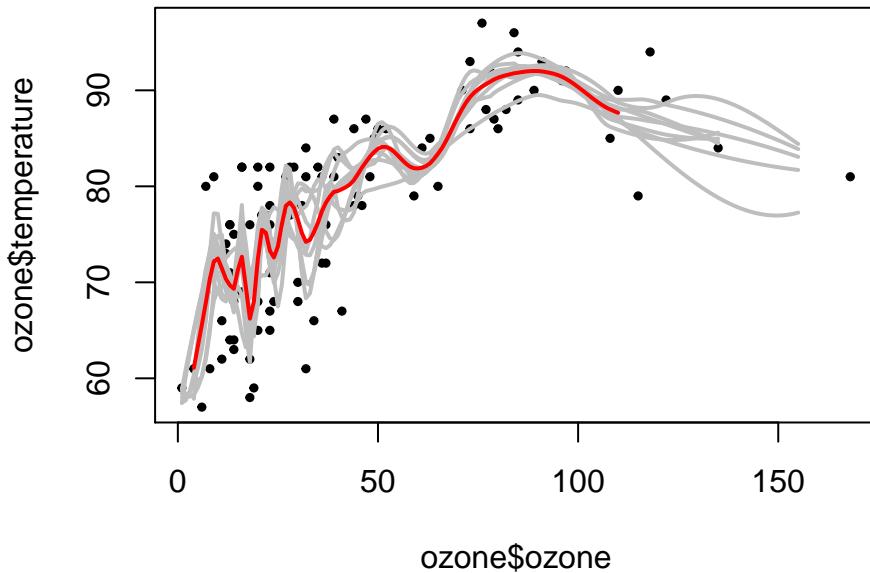
```
# predict on test values
predict(modFit,newdata=testing)
```

```
## [1] setosa      setosa      setosa      setosa      setosa      setosa
## [7] setosa      setosa      setosa      setosa      setosa      setosa
## [13] setosa     setosa      setosa      versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica   virginica   virginica   virginica   virginica   virginica
## [37] versicolor   virginica   virginica   versicolor versicolor virginica
## [43] virginica   virginica   virginica
## Levels: setosa versicolor virginica
```

## Bagging

- **bagging** = bootstrap aggregating
  - resample training data set (with replacement) and recalculate predictions
  - average the predictions together or majority vote
  - more information can be found [here](#)
- averaging multiple complex models have *similar bias* as each of the models on its own, and *reduced variance* because of the average
- most useful for non-linear models
- *example*
  - `loess(y ~ x, data=train, span=0.2)` = fits a smooth curve to data
    - \* `span=0.2` = controls how smooth the curve should be

```
# load data
library(ElemStatLearn); data(ozone, package="ElemStatLearn")
# reorder rows based on ozone variable
ozone <- ozone[order(ozone$ozone),]
# create empty matrix
ll <- matrix(NA, nrow=10, ncol=155)
# iterate 10 times
for(i in 1:10){
  # create sample from data with replacement
  ss <- sample(1:dim(ozone)[1], replace=T)
  # draw sample from the data and reorder rows based on ozone
  ozone0 <- ozone[ss,]; ozone0 <- ozone0[order(ozone0$ozone),]
  # fit loess function through data (similar to spline)
  loess0 <- loess(temperature ~ ozone, data=ozone0, span=0.2)
  # prediction from loess curve for the same values each time
  ll[i,] <- predict(loess0, newdata=data.frame(ozone=1:155))
}
# plot the data points
plot(ozone$ozone, ozone$temperature, pch=19, cex=0.5)
# plot each prediction model
for(i in 1:10){lines(1:155, ll[i,], col="grey", lwd=2)}
# plot the average in red
lines(1:155, apply(ll, 2, mean), col="red", lwd=2)
```

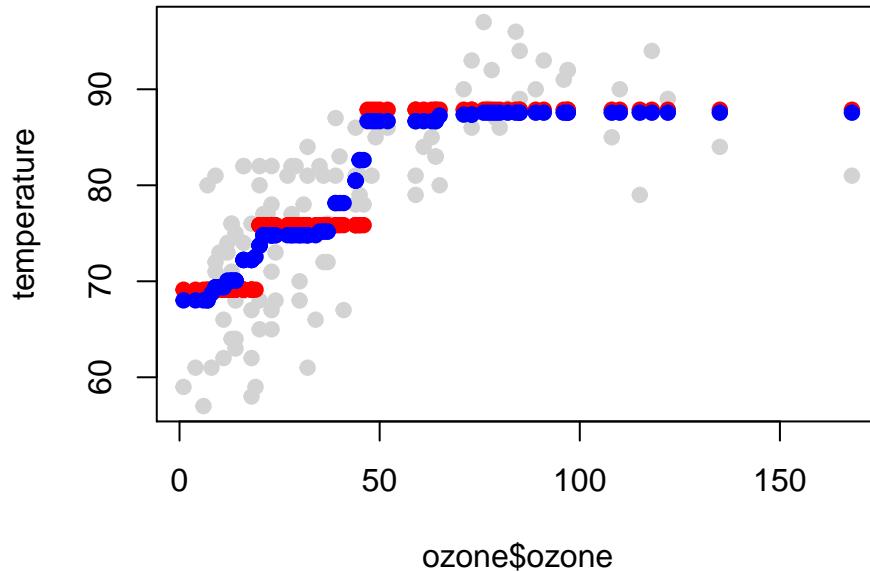


## Bagging Algorithms

- in the `caret` package, there are three options for the `train` function to perform bagging
  - `bagEarth` - Bagged MARS ([documentation](#))
  - `treebag` - Bagged CART ([documentation](#))
  - `bagFDA` - Bagged Flexible Discriminant Analysis ([documentation](#))
- alternatively, custom `bag` functions can be constructed ([documentation](#))
  - `bag(predictors, outcome, B=10, bagControl(fit, predict, aggregate))` = define and execute custom bagging algorithm
    - \* `B=10` = iterations/resampling to perform
    - \* `bagControl()` = controls for how the bagging should be executed
      - `fit=ctreeBag$fit` = the model ran on each resampling of data
      - `predict=ctreeBag$predict` = how predictions should be calculated from each model
      - `aggregate=ctreeBag$aggregate` = how the prediction models should be combined/averaged
  - *example*

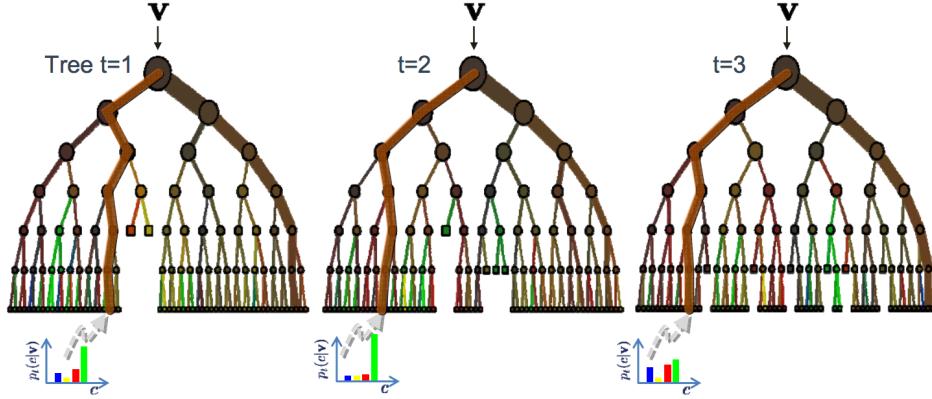
```
# load relevant package and data
library(party); data(ozone, package="ElemStatLearn")
# reorder rows based on ozone variable
ozone <- ozone[order(ozone$ozone),]
# extract predictors
predictors <- data.frame(ozone=ozone$ozone)
# extract outcome
temperature <- ozone$temperature
# run bagging algorithm
treebag <- bag(predictors, temperature, B = 10,
                 # custom bagging function
                 bagControl = bagControl(fit = ctreeBag$fit,
                                         predict = ctreeBag$pred,
                                         aggregate = ctreeBag$aggregate))
```

```
# plot data points
plot(ozone$ozone,temperature,col='lightgrey',pch=19)
# plot the first fit
points(ozone$ozone,predict(treebag$fits[[1]]$fit,predictors),pch=19,col="red")
# plot the aggregated predictions
points(ozone$ozone,predict(treebag,predictors),pch=19,col="blue")
```



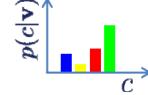
## Random Forest

- **random forest** = extension of bagging on classification/regression trees
  - one of the most used/accurate algorithms along with boosting



### The ensemble model

$$\text{Forest output probability } p(c|v) = \frac{1}{T} \sum_t^T p_t(c|v)$$



- **process**
  - bootstrap samples from training data (with replacement)
  - split and bootstrap variables
  - grow trees (repeat split/bootstrap) and vote/average final trees
- **drawbacks**
  - algorithm can be slow (process large number of trees)
  - hard to interpret (large numbers of splits and nodes)
  - over-fitting (difficult to know which tree is causing over-fitting)
  - **Note:** it is extremely important to use cross validation when running random forest algorithms

## R Commands and Examples

- `rf<-train(outcome ~ ., data=train, method="rf", prox=TRUE, ntree=500)` = runs random forest algorithm on the training data against all predictors
  - **Note:** random forest algorithm automatically bootstrap by default, but it is still important to have train/test/validation split to verify the accuracy of the model
  - `prox=TRUE` = the proximity measures between observations should be calculated (used in functions such as `classCenter()` to find center of groups)
    - \* `rf$finalModel$prox` = returns matrix of proximities
  - `ntree=500` = specify number of trees that should be constructed
  - `do.trace=TRUE` = prints logs as the trees are being built -> useful by indicating progress to user
  - **Note:** `randomForest()` function can be used to perform random forest algorithm (syntax is the same as `train`) and is much faster
- `getTree(rf$finalModel, k=2)` = return specific tree from random forest model

- `classCenters(predictors, outcome, proximity, nNbr)` = return computes the cluster centers using the `nNbr` nearest neighbors of the observations
  - `prox = rf$finalModel$prox` = proximity matrix from the random forest model
  - `nNbr` = number of nearest neighbors that should be used to compute cluster centers
- `predict(rf, test)` = apply the random forest model to test data set
  - `confusionMatrix(predictions, actualOutcome)` = tabulates the predictions of the model against the truths
    - \* *Note: this is generally done for the validation data set using the model built from training*
- *example*

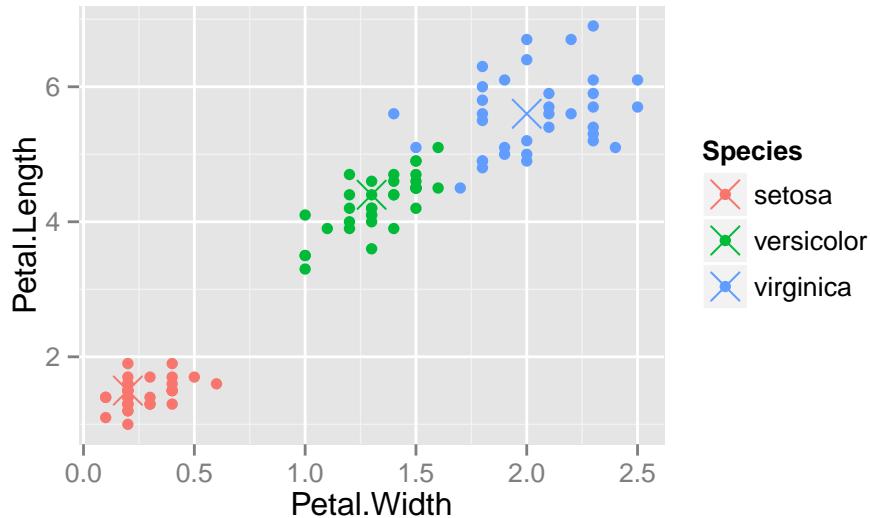
```
# load data
data(iris)

# create train/test data sets
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]

# apply random forest
modFit <- train(Species ~ ., data=training, method="rf", prox=TRUE)
# return the second tree (first 6 rows)
head(getTree(modFit$finalModel, k=2))
```

##	left	daughter	right	daughter	split	var	split	point	status	prediction
## 1		2		3	4		0.80	1		0
## 2		0		0	0		0.00	-1		1
## 3		4		5	1		6.05	1		0
## 4		6		7	4		1.75	1		0
## 5		8		9	4		1.65	1		0
## 6		0		0	0		0.00	-1		2

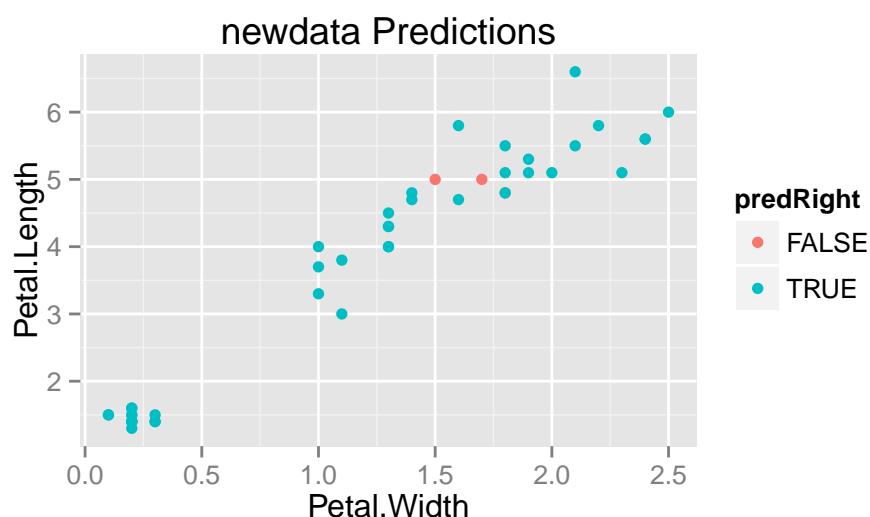
```
# compute cluster centers
irisP <- classCenter(training[,c(3,4)], training$Species, modFit$finalModel$prox)
# convert irisP to data frame and add Species column
irisP <- as.data.frame(irisP); irisP$Species <- rownames(irisP)
# plot data points
p <- qplot(Petal.Width, Petal.Length, col=Species, data=training)
# add the cluster centers
p + geom_point(aes(x=Petal.Width, y=Petal.Length, col=Species), size=5, shape=4, data=irisP)
```



```
# predict outcome for test data set using the random forest model
pred <- predict(modFit, testing)
# logic value for whether or not the rf algorithm predicted correctly
testing$predRight <- pred==testing$Species
# tabulate results
table(pred, testing$Species)
```

```
## 
##   pred      setosa versicolor virginica
##   setosa       15        0        0
##   versicolor    0       13        1
##   virginica     0        2       14
```

```
# plot data points with the incorrect classification highlighted
qplot(Petal.Width, Petal.Length, colour=predRight, data=testing, main="newdata Predictions")
```



## Boosting

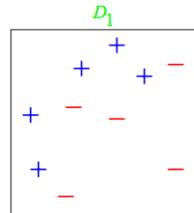
- **boosting** = one of the most widely used and accurate prediction models, along with random forest
- boosting can be done with any set of classifiers, and a well-known approach is [gradient boosting](#)
- more detail tutorial can be found [here](#)
- **process:** take a group of weak predictors  $\rightarrow$  weight them and add them up  $\rightarrow$  result in a stronger predictor
  - start with a set of classifiers  $h_1, \dots, h_k$ 
    - \* examples: all possible trees, all possible regression models, all possible cutoffs (divide data into different parts)
  - calculate a weighted sum of classifiers as the prediction value

$$f(x) = \sum_i \alpha_i h_i(x)$$

where  $\alpha_i$  = coefficient/weight and  $h_i(x)$  = value of classifier

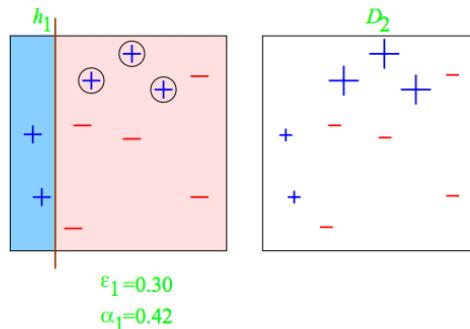
- \* goal = minimize error (on training set)
- \* select one  $h$  at each step (iterative)
- \* calculate weights based on errors
- \* up-weight missed classifications and select next  $h$

- *example*

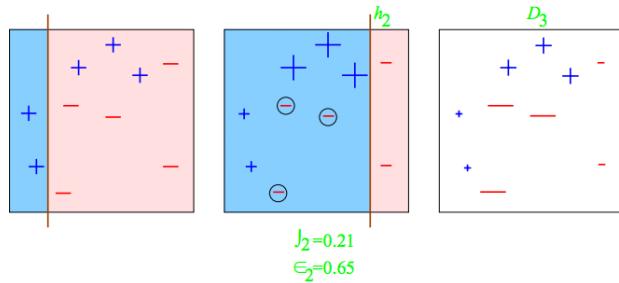


- we start with space with **blue +** and **red -** and the goal is to classify all the object correctly
- only straight lines will be used for classification

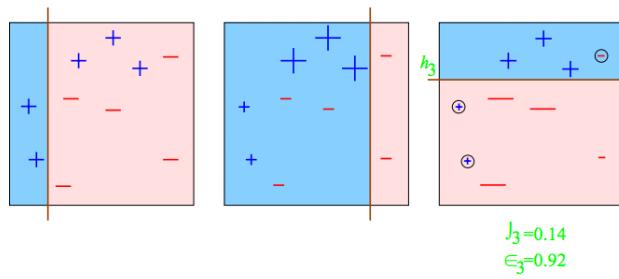
### Round 1



### Round 2



### Round 3



### Final Hypothesis

$$H_{\text{final}} = \text{sign} \left( 0.42 \begin{array}{|c|c|} \hline \text{blue} & \text{red} \\ \hline \end{array} + 0.65 \begin{array}{|c|c|} \hline \text{blue} & \text{red} \\ \hline \end{array} + 0.92 \begin{array}{|c|c|} \hline \text{white} & \text{red} \\ \hline \end{array} \right)$$

$$= \begin{array}{|c|c|} \hline \text{blue} & \text{red} \\ \hline \end{array}$$

The final hypothesis combines the three weak predictors ( $h_1, h_2, h_3$ ) with weights 0.42, 0.65, and 0.92 respectively, using a sign function to produce a single strong predictor. The resulting decision region is a white square with red '+' signs in the top-right and bottom-right quadrants, and red '-' signs in the top-left and bottom-left quadrants.

- from the above, we can see that a group of weak predictors (lines in this case), can be combined and weighed to become a much stronger predictor

### R Commands and Examples

- `gbm <- train(outcome ~ variables, method="gbm", data=train, verbose=F)` = run boosting model on the given data
  - options for `method` for boosting

- \* **gbm** - boosting with trees
- \* **mboost** - model based boosting
- \* **ada** - statistical boosting based on **additive logistic regression**
- \* **gamBoost** for boosting generalized additive models
- \* **Note:** differences between packages include the choice of basic classification functions and combination rules

- predict function can be used to apply the model to test data, similar to the rest of the algorithms in caret package
- *example*

```
# load data
data(Wage)
# remove log wage variable (we are trying to predict wage)
Wage <- subset(Wage,select=-c(logwage))
# create train/test data sets
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# run the gbm model
modFit <- train(wage ~ ., method="gbm",data=training,verbose=FALSE)
# print model summary
print(modFit)
```

```
## Stochastic Gradient Boosting
##
## 2102 samples
##    10 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
##
## Resampling results across tuning parameters:
##
##     interaction.depth  n.trees   RMSE    Rsquared   RMSE SD   Rsquared SD
##     1                  50        34.24090  0.3326536  1.470968  0.02004537
##     1                  100       33.72966  0.3415924  1.416432  0.02047354
##     1                  150       33.66723  0.3435784  1.374203  0.02054864
##     2                  50        33.70800  0.3434378  1.477201  0.02229064
##     2                  100       33.61315  0.3454457  1.458452  0.02307921
##     2                  150       33.69484  0.3433553  1.434715  0.02274775
##     3                  50        33.63670  0.3449055  1.447481  0.02194369
##     3                  100       33.73993  0.3416540  1.449907  0.02384793
##     3                  150       33.93379  0.3355735  1.425671  0.02281075
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 100,
## interaction.depth = 2 and shrinkage = 0.1.
```

## Model Based Prediction

- **model based prediction** = assumes the data follow a probabilistic model/distribution and use *Bayes' theorem* to identify optimal classifiers/variables
  - can potentially take advantage of structure of the data
  - could help reduce computational complexity (reduce variables)
  - can be reasonably accurate on real problems
- this approach does make ***additional assumptions*** about the data, which can lead to model failure/reduced accuracy if they are too far off
- **goal** = build parameter-based model (based on probabilities) for conditional distribution  $P(Y = k | X = x)$ , or the probability of the outcome  $Y$  is equal to a particular value  $k$  given a specific set of predictor variables  $x$ 
  - **Note:**  $X$  is the data for the model (observations for all predictor variables), which is also known as the **design matrix**
- **typical approach/process**
  1. start with the quantity  $P(Y = k | X = x)$
  2. apply *Bayes' Theorem* such that

$$P(Y = k | X = x) = \frac{P(X = x | Y = k)P(Y = k)}{\sum_{\ell=1}^K P(X = x | Y = \ell)P(Y = \ell)}$$

where the denominator is simply the sum of probabilities for the predictor variables are the set specified in  $x$  for all outcomes of  $Y$

3. assume the term  $P(X = x | Y = k)$  in the numerator follows a parameter-based probability distribution, or  $f_k(x)$ 
  - common choice = **Gaussian distribution**

$$f_k(x) = \frac{1}{\sigma_k \sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

4. assume the probability for the outcome  $Y$  to take on value of  $k$ , or  $P(Y = k)$ , is determined from the data to be some known quantity  $\pi_k$ 
  - **Note:**  $P(Y = k)$  is known as the **prior probability**
5. so the quantity  $P(Y = k | X = x)$  can be rewritten as

$$P(Y = k | X = x) = \frac{f_k(x)\pi_k}{\sum_{\ell=1}^K f_\ell(x)\pi_\ell}$$

6. estimate the parameters  $(\mu_k, \sigma_k^2)$  for the function  $f_k(x)$  from the data
  7. calculate  $P(Y = k | X = x)$  using the parameters
  8. the outcome  $Y$  is where the value of  $P(Y = k | X = x)$  is the highest
- prediction models that leverage this approach
    - **linear discriminant analysis** = assumes  $f_k(x)$  is multivariate Gaussian distribution with **same** covariance for each predictor variables
      - \* effectively drawing lines through “covariate space”
    - **quadratic discriminant analysis** = assumes  $f_k(x)$  is multivariate Gaussian distribution with **different** covariance for predictor variables
      - \* effectively drawing curves through “covariate space”
    - **normal mixture modeling** = assumes more complicated covariance matrix for the predictor variables

- **naive Bayes** = assumes independence between predictor variables/features for model building (covariance = 0)
  - \* *Note: this may be an incorrect assumption but it helps to reduce computational complexity and may still produce a useful result*

## Linear Discriminant Analysis

- to compare the probability for outcome  $Y = k$  versus probability for outcome  $Y = j$ , we can look at the ratio of

$$\frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)}$$

- take the **log** of the ratio and apply Bayes' Theorem, we get

$$\log \frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)} = \log \frac{f_k(x)}{f_j(x)} + \log \frac{\pi_k}{\pi_j}$$

which is effectively the log ratio of probability density functions plus the log ratio of prior probabilities

- *Note: log = monotone transformation, which means taking the log of a quantity does not affect implication of the ratio since the log(ratio) is directly correlated with ratio*

- if we substitute  $f_k(x)$  and  $f_j(x)$  with Gaussian probability density functions

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

so the ratio can be simplified to

$$\log \frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)} = \underbrace{\log \frac{\pi_k}{\pi_j}}_{\text{constant}} - \underbrace{\frac{1}{2}(\mu_k + \mu_j)^T \Sigma^{-1} (\mu_k + \mu_j)}_{\text{constant}} + \underbrace{x^T \Sigma^{-1} (\mu_k - \mu_j)}_{\text{slope}}$$

where  $\Sigma^{-1}$  = covariance matrix for the predictor variables,  $x^T$  = set of predictor variables, and  $\mu_k / \mu_j$  = mean of  $k, j$  respectively

- as annotated above, the log-ratio is effectively an equation of a line for a set of predictor variables  $x$

- *Note: the lines are also known as decision boundaries*

- therefore, we can classify values based on **which side of the line** the value is located ( $k$  vs  $j$ )

- **discriminant functions** are used to determine value of  $k$ , the functions are in the form of

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log(\pi_k)$$

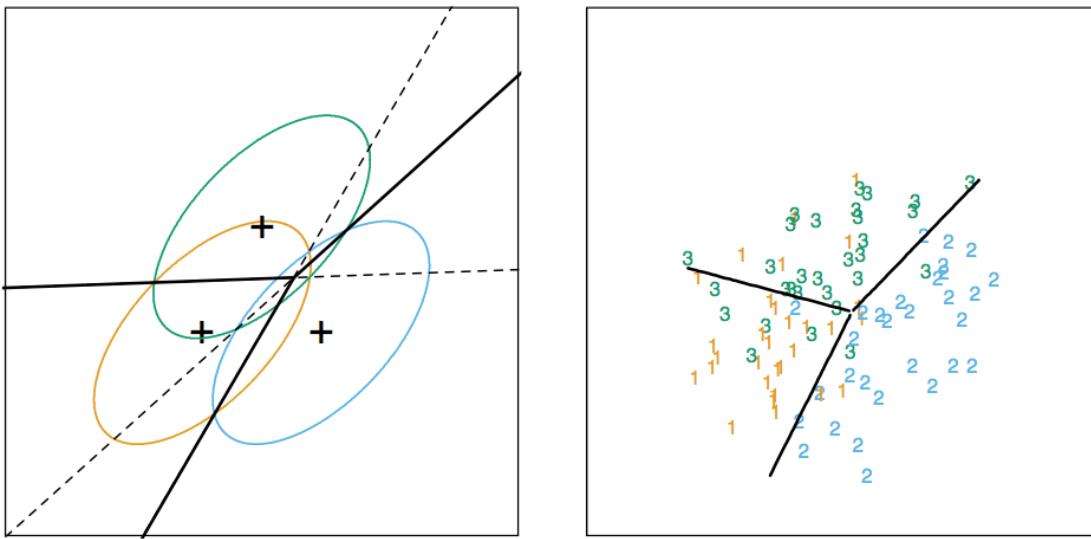
- plugging in the set of predictor variables,  $x^T$ , into the discriminant function, we can find the value of  $k$  that **maximizes** the function  $\delta_k(x)$
- the terms of the discriminant function can be estimated using maximum likelihood

- the predicted value for the outcome is therefore  $\hat{Y}(x) = \operatorname{argmax}_k \delta_k(x)$

- **example**

- classify a group of values into 3 groups using 2 variables ( $x, y$  coordinates)
- 3 lines are draw to split the data into 3 Gaussian distributions
  - \* each line splits the data into two groups  $\rightarrow 1$  vs  $2$ ,  $2$  vs  $3$ ,  $1$  vs  $3$

- \* each side of the line represents a region where the probability of one group (1, 2, or 3) is the highest
- the result is represented in the the following graph



- **R Commands**

- `lda<-train(outcome ~ predictors, data=training, method="lda")` = constructs a linear discriminant analysis model on the predictors with the provided training data
- `predict(lda, test)` = applies the LDA model to test data and return the prediction results in data frame
- *example: caret package*

```
# load data
data(iris)

# create training and test sets
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]

# run the linear discriminant analysis on training data
lda <- train(Species ~ ., data=training, method="lda")

# predict test outcomes using LDA model
pred.lda <- predict(lda, testing)

# print results
pred.lda
```

```
## [1] setosa    setosa    setosa    setosa    setosa    setosa
## [7] setosa    setosa    setosa    setosa    setosa    setosa
## [13] setosa    setosa    setosa    versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor
## [31] virginica virginica virginica virginica virginica virginica
## [37] virginica virginica virginica virginica virginica virginica
## [43] virginica virginica virginica
## Levels: setosa versicolor virginica
```

## Naive Bayes

- for predictors  $X_1, \dots, X_m$ , we want to model  $P(Y = k | X_1, \dots, X_m)$
- by applying *Bayes' Theorem*, we get

$$P(Y = k | X_1, \dots, X_m) = \frac{\pi_k P(X_1, \dots, X_m | Y = k)}{\sum_{\ell=1}^K P(X_1, \dots, X_m | Y = k) \pi_\ell}$$

- since the denominator is just a sum (constant), we can rewrite the quantity as

$$P(Y = k | X_1, \dots, X_m) \propto \pi_k P(X_1, \dots, X_m | Y = k)$$

or the probability is **proportional to** the numerator

– *Note: maximizing the numerator is the same as maximizing the ratio*

- $\pi_k P(X_1, \dots, X_m | Y = k)$  can be rewritten as

$$\begin{aligned} \pi_k P(X_1, \dots, X_m | Y = k) &= \pi_k P(X_1 | Y = k) P(X_2, \dots, X_m | X_1, Y = k) \\ &= \pi_k P(X_1 | Y = k) P(X_2 | X_1, Y = k) P(X_3, \dots, X_m | X_1, X_2, Y = k) \\ &= \pi_k P(X_1 | Y = k) P(X_2 | X_1, Y = k) \dots P(X_m | X_1 \dots, X_{m-1}, Y = k) \end{aligned}$$

where each variable has its own probability term that depends on all the terms before it

- this is effectively indicating that each of the predictors may be dependent on other predictors
- however, if we make the assumption that all predictor variables are **independent** to each other, the quantity can be simplified to

$$\pi_k P(X_1, \dots, X_m | Y = k) \approx \pi_k P(X_1 | Y = k) P(X_2 | Y = k) \dots P(X_m | , Y = k)$$

which is effectively the product of the prior probability for  $k$  and the probability of variables  $X_1, \dots, X_m$  given that  $Y = k$

– *Note: the assumption is naive in that it is unlikely the predictors are completely independent of each other, but this model still produces useful results particularly with large number of binary/categorical variables*

- \* text and document classification usually require large quantities of binary and categorical features

- **R Commands**

- `nb <- train(outcome ~ predictors, data=training, method="nb")` = constructs a naive Bayes model on the predictors with the provided training data
- `predict(nb, test)` = applies the naive Bayes model to test data and return the prediction results in data frame
- *example: caret package*

```
# using the same data from iris, run naive Bayes on training data
nb <- train(Species ~ ., data=training, method="nb")
# predict test outcomes using naive Bayes model
pred.nb <- predict(nb, testing)
# print results
pred.nb
```

```
## [1] setosa    setosa    setosa    setosa    setosa    setosa
## [7] setosa    setosa    setosa    setosa    setosa    setosa
## [13] setosa   setosa    setosa    versicolor versicolor versicolor
```

```

## [19] versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor
## [31] virginica virginica virginica virginica virginica virginica
## [37] virginica virginica virginica virginica versicolor virginica
## [43] virginica virginica virginica
## Levels: setosa versicolor virginica

```

### Compare Results for LDA and Naive Bayes

- linear discriminant analysis and naive Bayes generally produce similar results for small data sets
- for our example data from `iris` data set, we can compare the prediction the results from the two models

```

# tabulate the prediction results from LDA and naive Bayes
table(pred.lda,pred.nb)

```

```

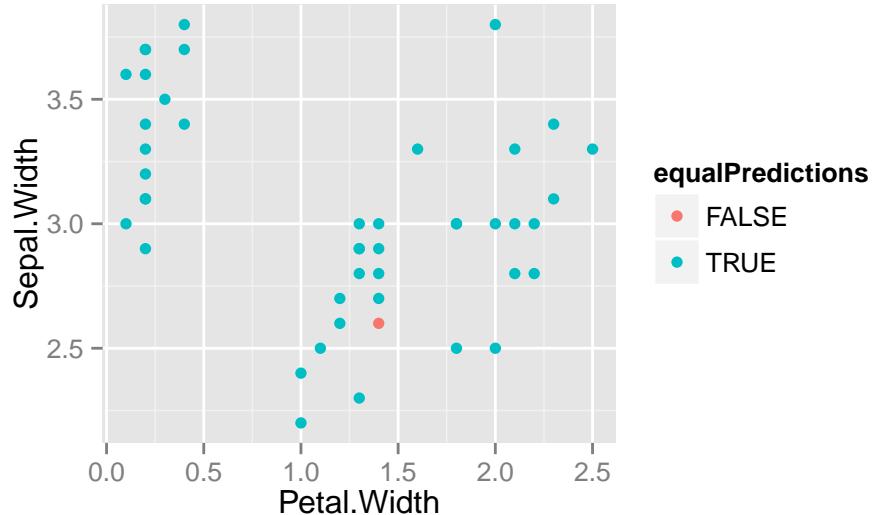
##          pred.nb
## pred.lda      setosa versicolor virginica
##   setosa        15       0       0
##   versicolor     0      15       0
##   virginica      0       1      14

```

```

# create logical variable that returns TRUE for when predictions from the two models match
equalPredictions <- (pred.lda==pred.nb)
# plot the comparison
qplot(Petal.Width,Sepal.Width,colour=equalPredictions,data=testing)

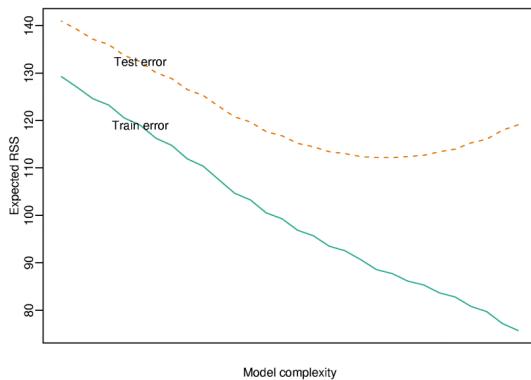
```



- as we can see from above, only one data point, which is located inbetween the two classes is predicted differently by the two models

## Model Selection

- the general behavior of the errors of the training and test sets are as follows
  - as the number of predictors used increases (or model complexity), the error for the prediction model on *training set* **always decreases**
  - the error for the prediction model on *test set* **decreases first and then increases** as number of predictors used approaches the total number of predictors available



- this is expected since as more predictors used, the model is more likely to *overfit* the training data
- **goal** in selecting models = *avoid overfitting* on training data and *minimize error* on test data
- **approaches**
  - split samples
  - decompose expected prediction error
  - hard thresholding for high-dimensional data
  - regularization for regression
    - \* ridge regression
    - \* lasso regression
- **problems**
  - time/computational complexity limitations
  - high dimensional

### Example: Training vs Test Error for Combination of Predictors

- *Note:* the code for this example comes from [Hector Corrada Bravo's Practical Machine Learning Course](#)
- to demonstrate the behavior of training and test errors, the prostate dataset from *Elements of Statistical Learning* is used
- all combinations of predictors are used to produce prediction models, and Residual Squared Error (RSS) is calculated for all models on both the training and test sets

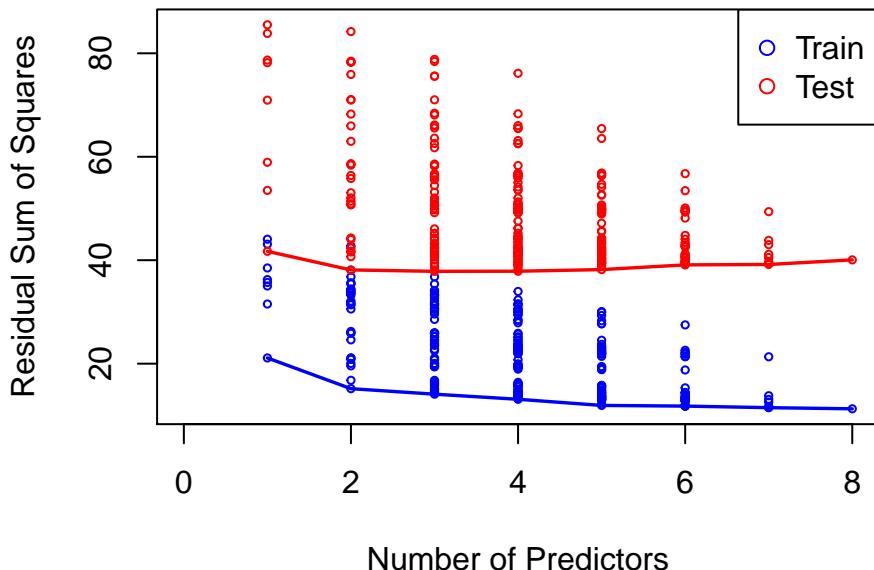
```
# load data and set seed
data(prostate); set.seed(1)
# define outcome y and predictors x
covnames <- names(prostate[-(9:10)])
y <- prostate$lpsa; x <- prostate[,covnames]
# create test set predictors and outcomes
```

```

train.ind <- sample(nrow(prostate), ceiling(nrow(prostate))/2)
y.test <- prostate$lpsa[-train.ind]; x.test <- x[-train.ind,]
# create training set predictors and outcomes
y <- prostate$lpsa[train.ind]; x <- x[train.ind,]
# p = number of predictors
p <- length(covnames)
# initialize the list of residual sum squares
rss <- list()
# loop through each combination of predictors and build models
for (i in 1:p) {
  # compute matrix for p choose i predictors for i = 1...p (creates i x p matrix)
  Index <- combn(p,i)
  # calculate residual sum squares of each combination of predictors
  rss[[i]] <- apply(Index, 2, function(is) {
    # take each combination (or column of Index matrix) and create formula for regression
    form <- as.formula(paste("y~", paste(covnames[is], collapse="+"), sep=""))
    # run linear regression with combination of predictors on training data
    isfit <- lm(form, data=x)
    # predict outcome for all training data points
    yhat <- predict(isfit)
    # calculate residual sum squares for predictions on training data
    train.rss <- sum((y - yhat)^2)
    # predict outcome for all test data points
    yhat <- predict(isfit, newdata=x.test)
    # calculate residual sum squares for predictions on test data
    test.rss <- sum((y.test - yhat)^2)
    # store each pair of training and test residual sum squares as a list
    c(train.rss, test.rss)
  })
}
# set up plot with labels, title, and proper x and y limits
plot(1:p, 1:p, type="n", ylim=range(unlist(rss)), xlim=c(0,p),
      xlab="Number of Predictors", ylab="Residual Sum of Squares",
      main="Prostate Cancer Data - Training vs Test RSS")
# add data points for training and test residual sum squares
for (i in 1:p) {
  # plot training residual sum squares in blue
  points(rep(i, ncol(rss[[i]])), rss[[i]][1, ], col="blue", cex = 0.5)
  # plot test residual sum squares in red
  points(rep(i, ncol(rss[[i]])), rss[[i]][2, ], col="red", cex = 0.5)
}
# find the minimum training RSS for each combination of predictors
minrss <- sapply(rss, function(x) min(x[1]))
# plot line through the minimum training RSS data points in blue
lines((1:p), minrss, col="blue", lwd=1.7)
# find the minimum test RSS for each combination of predictors
minrss <- sapply(rss, function(x) min(x[2]))
# plot line through the minimum test RSS data points in blue
lines((1:p), minrss, col="red", lwd=1.7)
# add legend
legend("topright", c("Train", "Test"), col=c("blue", "red"), pch=1)

```

## Prostate Cancer Data – Training vs Test RSS



- from the above, we can clearly see that test RSS error approaches the minimum at around 3 predictors and increases slightly as more predictors are used

### Split Samples

- the best method to pick predictors/model is to split the given data into different test sets
- **process**
  1. divide data into training/test/validation sets (60 - 20 - 20 split)
  2. train all competing models on the training data
  3. apply the models on validation data and choose the best performing model
  4. re-split data into training/test/validation sets and repeat steps 1 to 3
  5. apply the overall best performing model on test set to appropriately assess performance on new data
- **common problems**
  - limited data = if not enough data is available, it may not be possible to produce a good model fit after splitting the data into 3 sets
  - computational complexity = modeling with all subsets of models can be extremely taxing in terms of computations, especially when a large number of predictors are available

### Decompose Expected Prediction Error

- the outcome  $Y_i$  can be modeled by

$$Y_i = f(X_i) + \epsilon_i$$

where  $\epsilon_i$  = error term

- the **expected prediction error** is defined as

$$EPE(\lambda) = E \left[ \left( Y - \hat{f}_\lambda(X) \right)^2 \right]$$

where  $\lambda$  = specific set of tuning parameters

- estimates from the model constructed with training data can be denoted as  $\hat{f}_\lambda(x^*)$  where  $X = x^*$  is the new data point that we would like to predict at
- the expected prediction error is as follows

$$\begin{aligned} E \left[ (Y - \hat{f}_\lambda(x^*))^2 \right] &= \sigma^2 + \left( E[\hat{f}_\lambda(x^*)] - f(x^*) \right)^2 + E \left[ \hat{f}_\lambda(x^*) - E[\hat{f}_\lambda(x^*)] \right]^2 \\ &= \text{Irreducible Error} + \text{Bias}^2 + \text{Variance} \end{aligned}$$

- goal of prediction model** = minimize overall expected prediction error
- irreducible error = noise inherent to the data collection process  $\rightarrow$  cannot be reduced
- bias/variance = can be traded in order to find optimal model (least error)

## Hard Thresholding

- if there are more predictors than observations (high-dimensional data), linear regressions will only return coefficients for some of the variables because there's not enough data to estimate the rest of the parameters
  - conceptually, this occurs because the design matrix that the model is based on cannot be inverted
    - Note:** ridge regression can help address this problem
- hard thresholding** can help estimate the coefficients/model by taking subsets of predictors and building models
- process**
  - model the outcome as
$$Y_i = f(X_i) + \epsilon_i$$
  - where  $\epsilon_i$  = error term
  - assume the prediction estimate has a linear form
$$\hat{f}_\lambda(x) = x' \beta$$
  - where only  $\lambda$  coefficients for the set of predictors  $x$  are **nonzero**
  - after setting the value of  $\lambda$ , compute models using all combinations of  $\lambda$  variables to find which variables' coefficients should be set to be zero
- problem**
  - computationally intensive
- example**
  - as we can see from the results below, some of the coefficients have values of NA

```
# load prostate data
data(prostate)
# create subset of observations with 10 variables
small = prostate[1:5,]
# print linear regression
lm(lpsa ~ ., data = small)
```

```
##
## Call:
## lm(formula = lpsa ~ ., data = small)
##
## Coefficients:
## (Intercept)    lcavol     lweight       age       lbph
## 9.60615      0.13901    -0.79142     0.09516      NA
## svi          lcp        gleason     pgg45   trainTRUE
## NA           NA         -2.08710      NA         NA
```

## Regularized Regression Concept ([Resource](#))

- **regularized regression** = fit a regression model and adjust for the large coefficients in attempt to help with bias/variance trade-off or model selection
  - when running regressions unconstrained (without specifying any criteria for coefficients), the model may be susceptible to high variance (coefficients explode  $\rightarrow$  very large values) if there are variables that are highly correlated
  - controlling/regularizing coefficients may slightly *increase bias* (lose a bit of prediction capability) but will *reduce variance* and improve the prediction error
  - however, this approach may be very demanding computationally and generally does not perform as well as random forest/boosting
- **Penalized Residual Sum of Squares (PRSS)** is calculated by adding a penalty term to the prediction squared error

$$PRSS(\beta) = \sum_{j=1}^n (Y_j - \sum_{i=1}^m \beta_{1i} X_{ij})^2 + P(\lambda; \beta)$$

- penalty shrinks coefficients if their values become too large
- penalty term is generally used to reduce complexity and variance for the model, while respecting the structure of the data/relationship
- *example: co-linear variables*

- given a linear model,

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

- where  $X_1$  and  $X_2$  are nearly perfectly correlated (co-linear)
- the model can then be approximated by this model by omitting  $X_2$ , so the model becomes

$$Y = \beta_0 + (\beta_1 + \beta_2) X_1 + \epsilon$$

- with the above model, we can get a good estimate of  $Y$ 
  - \* the estimate of  $Y$  will be biased
  - \* but the variance of the prediction may be reduced

## Regularized Regression - Ridge Regression

- the penalized residual sum of squares (PRSS) takes the form of

$$PRSS(\beta_j) = \sum_{i=1}^N \left( y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

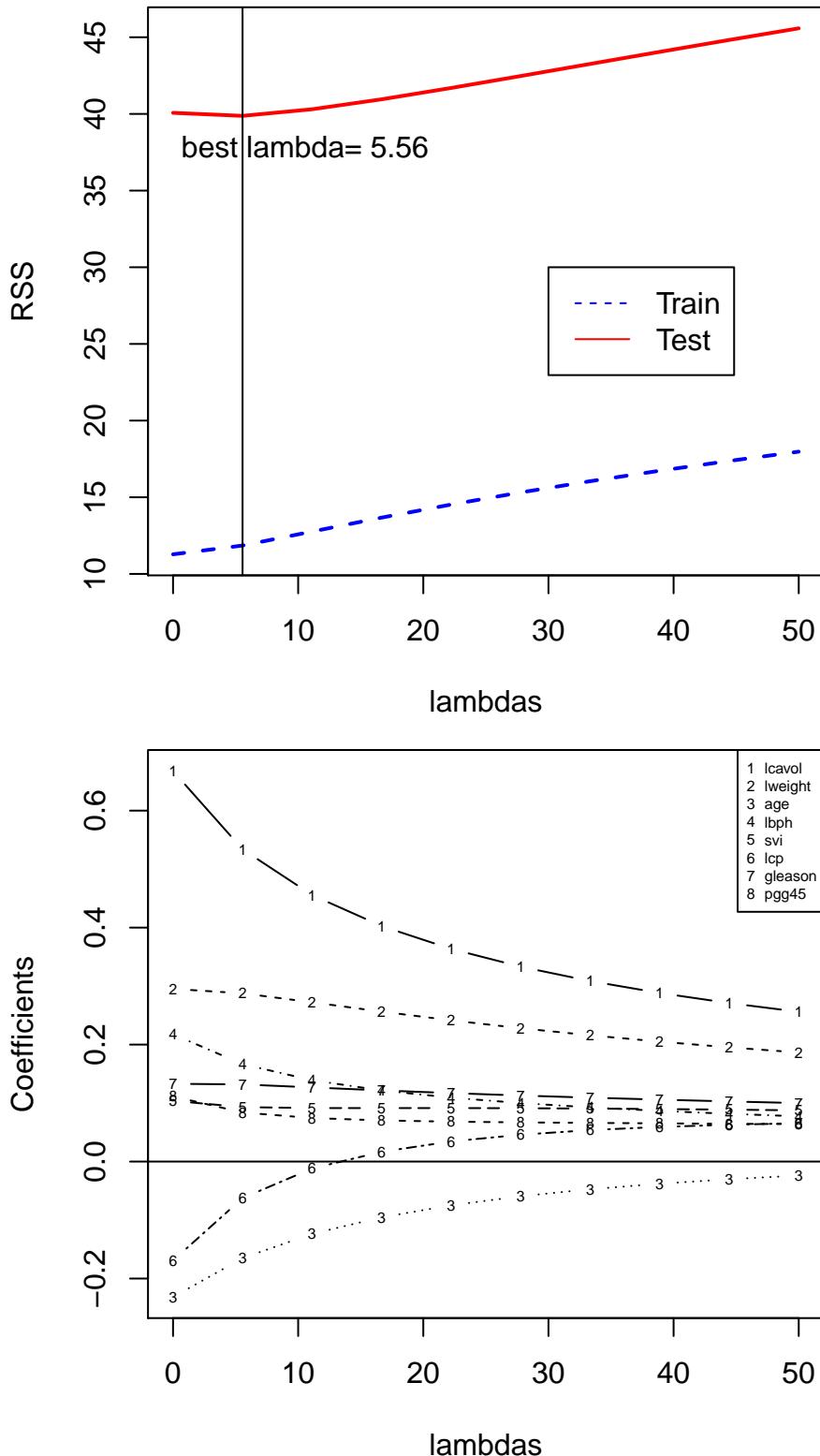
- this is equivalent to solving the equation

$$PRSS(\beta_j) = \sum_{i=1}^N \left( y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j \right)^2$$

subject to constraint  $\sum_{j=1}^p \beta_j^2 \leq s$  where  $s$  is inversely proportional to  $\lambda$

- if the coefficients  $\beta_j$  are large in value, the term  $\sum_{j=1}^p \beta_j^2$  will cause the overall PRSS value to increase, leading to worse models
- the presence of the term thus requires some of the coefficients to be small

- inclusion of  $\lambda$  makes the problem *non-singular* even if  $X^T X$  is not invertible
  - this means that even in cases where there are more predictors than observations, the coefficients of the predictors can still be estimated
- $\lambda$  = tuning parameter
  - controls size of coefficients or the amount of regularization
  - as  $\lambda \rightarrow 0$ , the result approaches the least square solution
  - as  $\lambda \rightarrow \infty$ , all of the coefficients receive large penalties and the conditional coefficients  $\hat{\beta}_{\lambda=\infty}^{ridge}$  approaches zero collectively
  - $\lambda$  should be carefully chosen through cross-validation/other techniques to find the optimal tradeoff of bias for variance
  - *Note: it is important realize that all coefficients (though they may be shrunk to very small values) will still be included in the model when applying ridge regression*
- **R Commands**
  - [MASS package] `ridge<-lm.ridge(outcome ~ predictors, data=training, lambda=5)` = perform ridge regression with given outcome and predictors using the provided  $\lambda$  value
    - \* *Note: the predictors are centered and scaled first before the regression is run*
    - \* `lambda=5` = tuning parameter
    - \* `ridge$xm` = returns column/predictor mean from the data
    - \* `ridge$scale` = returns the scaling performed on the predictors for the ridge regression
      - *Note: all the variables are divided by the biased standard deviation  $\sum(X_i - \bar{X}_i)/n$*
    - \* `ridge$coef` = returns the conditional coefficients,  $\beta_j$  from the ridge regression
    - \* `ridge$ym` = return mean of outcome
  - [caret package] `train(outcome ~ predictors, data=training, method="ridge", lambda=5)` = perform ridge regression with given outcome and predictors
    - \* `preProcess=c("center", "scale")` = centers and scales the predictors before the model is built
      - *Note: this is generally a good idea for building ridge regressions*
    - \* `lambda=5` = tuning parameter
  - [caret package] `train(outcome ~ predictors, data=training, method="foba", lambda=5, k=4)` = perform ridge regression with variable selection
    - \* `lambda=5` = tuning parameter
    - \* `k=4` = number of variables that should be retained
      - this means that `length(predictors)-k` variables will be eliminated
  - [caret package] `predict(model,test)` = use the model to predict on test set  $\rightarrow$  similar to all other caret algorithms
- *example: ridge coefficient paths vs  $\lambda$* 
  - using the same `prostate` dataset, we will run ridge regressions with different values of  $\lambda$  and find the optimum  $\lambda$  value that minimizes test RSS



### Regularized Regression - LASSO Regression

- LASSO (least absolute shrinkage and selection operator) was introduced by Tibshirani (Journal of the Royal Statistical Society 1996)

- similar to **ridge**, with slightly different penalty term
- the penalized residual sum of squares (PRSS) takes the form of

$$PRSS(\beta_j) = \sum_{i=1}^N \left( y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

- this is equivalent to solving the equation

$$PRSS(\beta_j) = \sum_{i=1}^N \left( y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j \right)^2$$

subject to constraint  $\sum_{j=1}^p |\beta_j| \leq s$  where  $s$  is inversely proportional to  $\lambda$

- $\lambda$  = tuning parameter
  - controls size of coefficients or the amount of regularization
  - large values of  $\lambda$  will set some coefficient equal to zero
- \* *Note: LASSO effectively performs model selection (choose subset of predictors) while shrinking other coefficients, whereas Ridge only shrinks the coefficients*

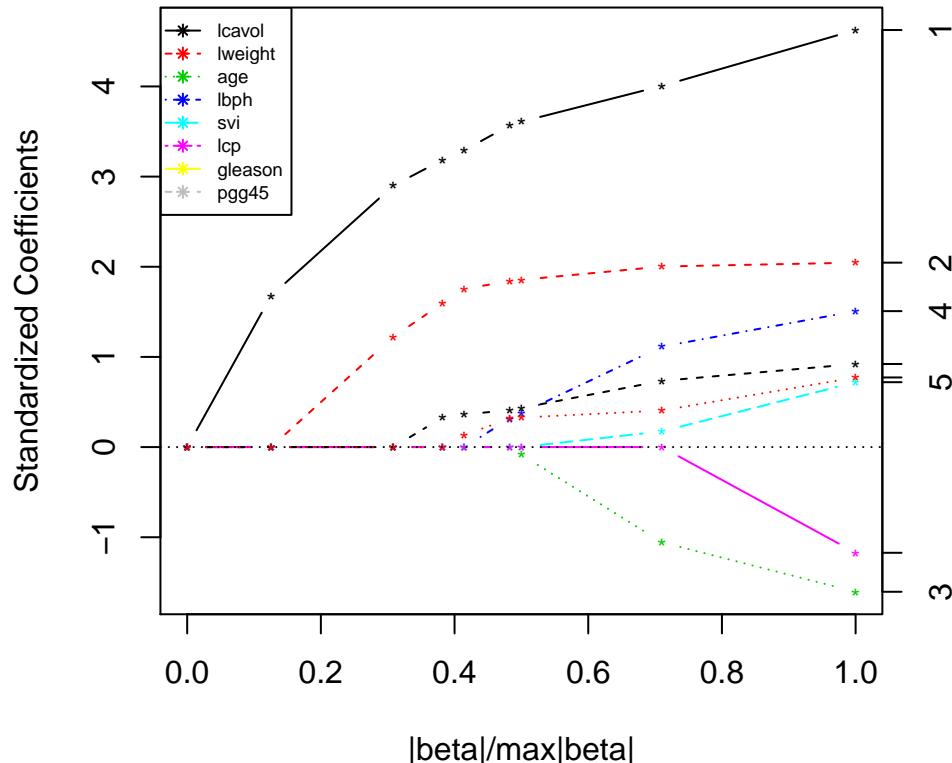
## • R Commands

- [lars package] `lasso<-lars(as.matrix(x), y, type="lasso", trace=TRUE)` = perform lasso regression by adding predictors one at a time (or setting some variables to 0)
  - \* *Note: the predictors are centered and scaled first before the regression is run*
  - \* `as.matrix(x)` = the predictors must be in matrix/dataframe format
  - \* `trace=TRUE` = prints progress of the lasso regression
  - \* `lasso$lambda` = return the  $\lambda$ s used for each step of the lasso regression
  - \* `plot(lasso)` = prints plot that shows the progression of the coefficients as they are set to zero one by one
  - \* `predict.lars(lasso, test)` = use the lasso model to predict on test data
    - \* *Note: more information/documentation can be found in ?predict.lars*
- [lars package] `cv.lars(as.matrix(x), y, K=10, type="lasso", trace=TRUE)` = computes K-fold cross-validated mean squared prediction error for lasso regression
  - \* effectively the `lars` function is run  $K$  times with each of the folds to estimate the
  - \* `K=10` = create 10-fold cross validation
  - \* `trace=TRUE` = prints progress of the lasso regression
- [enet package] `lasso<-enet(predictors, outcome, lambda = 0)` = perform elastic net regression on given predictors and outcome
  - \* `lambda=0` = default value for  $\lambda$ 
    - \* *Note: lasso regression is a special case of elastic net regression, and forcing lambda=0 tells the function to fit a lasso regression*
  - \* `plot(lasso)` = prints plot that shows the progression of the coefficients as they are set to zero one by one
  - \* `predict.enet(lasso, test)` = use the lasso model to predict on test data
- [caret package] `train(outcome ~ predictors, data=training, method="lasso")` = perform lasso regression with given outcome and predictors
  - \* *Note: outcome and predictors must be in the same dataframe*
  - \* `preProcess=c("center", "scale")` = centers and scales the predictors before the model is built
    - \* *Note: this is generally a good idea for building lasso regressions*

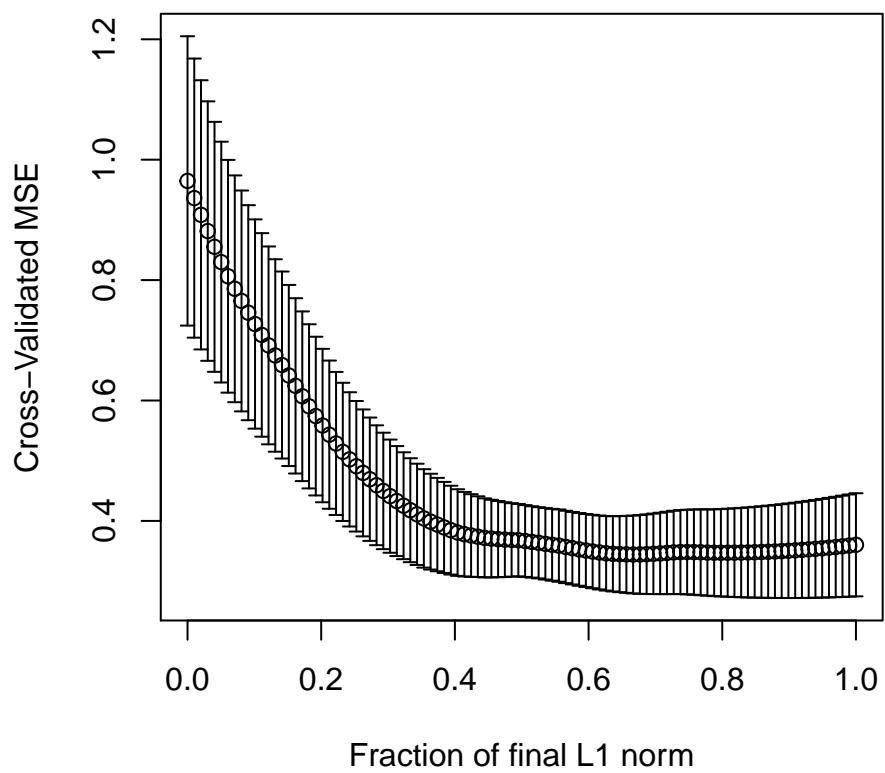
- [caret package] `train(outcome~predictors, data=train, method="relaxo", lambda=5, phi=0.3)`  
 = perform relaxed lasso regression on given predictors and outcome
  - \* `lambda=5` = tuning parameter
  - \* `phi=0.3` = relaxation parameter
    - `phi=1` corresponds to the regular Lasso solutions
    - `phi=0` computes the OLS estimates on the set of variables selected by the Lasso
- [caret package] `predict(model, test)` = use the model to predict on test set → similar to all other caret algorithms
- example: *lars package*

```
# load lars package
library(lars)
# perform lasso regression
lasso.fit <- lars(as.matrix(x), y, type="lasso", trace=TRUE)
# plot lasso regression model
plot(lasso.fit, breaks=FALSE, cex = 0.75)
# add legend
legend("topleft", covnames, pch=8, lty=1:length(covnames),
       col=1:length(covnames), cex = 0.6)
```

## LASSO



```
# plots the cross validation curve
lasso.cv <- cv.lars(as.matrix(x), y, K=10, type="lasso", trace=TRUE)
```



## Combining Predictors

- **combining predictors** = also known as *ensembling methods in learning*, combine classifiers by averaging/voting to improve accuracy (generally)
  - this reduces interpretability and increases computational complexity
  - boosting/bagging/random forest algorithms all use this idea, except all classifiers averaged are of the same type
- Netflix Competition was won by a team that blended together 107 machine learning algorithms, Heritage Health Prize was also won by a combination of algorithms
  - *Note: the winning algorithm for Netflix was not used because it was too computationally complex/intensive, so the trade-off between accuracy and scalability is very important*
- **approach**
  - combine similar classifiers using bagging/boosting/random forest
  - combine different classifiers using model stacking/ensembling
    - \* build an odd number of models (we need an odd number for the majority vote to avoid ties)
    - \* predict with each model
    - \* combine the predictions and predict the final outcome by majority vote
- **process: simple model ensembling**
  1. **build multiple models** on the training data set
  2. use the models to **predict** on the training/test set
    - *predict on training* if the data was divided to only training/test sets
    - *predict on test* if the data was divided into training/test/validation sets
  3. combine the prediction results from each model and the true results for the training/test set into a **new data frame**
    - one column for each model
    - add the true outcome from the training/test set as a separate column
  4. train the new data frame with a **new model** the true outcome as the outcome, and the predictions from various models as predictors
  5. use the combined model fit to predict results on the training/test data
    - calculate the RMSE for all models, including combined fit, to evaluate the accuracy of the different models
    - *Note: the RMSE for the combined fit should generally be lower than the the rest of the models*
  6. to predict on the final test/validation set, use all the initial models to predict on the data set first to **recreate a prediction data frame** for the test/validation set like in **step 3**
    - one column for each model, no truth column this time
  7. **apply the combined fit** on the combined prediction data frame to get the final resultant predictions

### Example - Majority Vote

- suppose we have 5 independent classifiers/models
- each has 70% accuracy
- **majority vote accuracy (mva)** = probability of the majority of the models achieving 70% at the same time

$$\begin{aligned}
\text{majority vote accuracy} &= \text{probability}(3 \text{ correct}, 2 \text{ wrong}) + \text{probability}(4 \text{ correct}, 1 \text{ wrong}) \\
&\quad + \text{probability}(5 \text{ correct}) \\
&= \binom{5}{3} \times (0.7)^3(0.3)^2 + \binom{5}{4} \times (0.7)^4(0.3)^1 - \binom{5}{5} (0.7)^5 \\
&= 10 \times (0.7)^3(0.3)^2 + 5 \times (0.7)^4(0.3)^1 - 1 \times (0.7)^5 \\
&= 83.7
\end{aligned}$$

- with 101 classifiers, the majority vote accuracy becomes 99.9%

### Example - Model Ensembling

```

# set up data
inBuild <- createDataPartition(y=Wage$wage, p=0.7, list=FALSE)
validation <- Wage[-inBuild,]; buildData <- Wage[inBuild,]
inTrain <- createDataPartition(y=buildData$wage, p=0.7, list=FALSE)
training <- buildData[inTrain,]; testing <- buildData[-inTrain,]
# train the data using both glm and random forest models
glm.fit <- train(wage ~ ., method="glm", data=training)
rf.fit <- train(wage ~ ., method="rf", data=training,
                 trControl = trainControl(method="cv"), number=3)
# use the models to predict the results on the testing set
glm.pred.test <- predict(glm.fit, testing)
rf.pred.test <- predict(rf.fit, testing)
# combine the prediction results and the true results into new data frame
combinedTestData <- data.frame(glm.pred=glm.pred.test,
                                 rf.pred = rf.pred.test, wage=testing$wage)
# run a Generalized Additive Model (gam) model on the combined test data
comb.fit <- train(wage ~ ., method="gam", data=combinedTestData)
# use the resultant model to predict on the test set
comb.pred.test <- predict(comb.fit, combinedTestData)
# use the glm and rf models to predict results on the validation data set
glm.pred.val <- predict(glm.fit, validation)
rf.pred.val <- predict(rf.fit, validation)
# combine the results into data frame for the comb.fit
combinedValData <- data.frame(glm.pred=glm.pred.val, rf.pred=rf.pred.val)
# run the comb.fit on the combined validation data
comb.pred.val <- predict(comb.fit, combinedValData)
# tabulate the results - test data set RMSE Errors
rbind(test = c(glm = sqrt(sum((glm.pred.test-testing$wage)^2)),
              rf = sqrt(sum((rf.pred.test-testing$wage)^2)),
              combined = sqrt(sum((comb.pred.test-testing$wage)^2))),
      # validation data set RMSE Errors
      validation = c(sqrt(sum((glm.pred.val-validation$wage)^2)),
                    sqrt(sum((rf.pred.val-validation$wage)^2)),
                    sqrt(sum((comb.pred.val-validation$wage)^2))))
##          glm      rf   combined
## test      858.7074 889.3876 845.1588
## validation 1061.0891 1085.2706 1057.7058

```

## Forecasting

- **forecasting** = typically used with time series, predict one or more observations into the future
  - data are dependent over time so subsampling/splitting data into training/test is more complicated and must be done very carefully
- specific patterns need to be considered for time series data (*time series decomposition*)
  - **trend** = long term increase/decrease in data
  - **seasonal** = patterns related to time of week/month/year/etc
  - **cyclic** = patterns that rise/fall periodically
- **Note:** issues that arise from time series are similar to those from spatial data
  - dependency between nearby observations
  - location specific effects
- all standard predictions models can be used but requires more consideration
- **Note:** more detailed tutorial can be found in *Rob Hyndman's Forecasting: principles and practice*
- **considerations for interpreting results**
  - unrelated time series can often seem to be correlated with each other (*spurious correlations*)
  - geographic analysis may exhibit similar patterns due to population distribution/concentrations
  - extrapolations too far into future can be dangerous as they can produce in insensible results
  - dependencies over time (seasonal effects) should be examined and isolated from the trends
- **process**
  - ensure the data is a time series data type
  - split data into training and test sets
    - \* both must have consecutive time points
  - choose forecast approach (SMA - `ma` vs EMA - `ets`, see below) and apply corresponding functions to training data
  - apply constructed forecast model to test data using `forecast` function
  - evaluate accuracy of forecast using `accuracy` function
- **approaches**
  - **simple moving averages** = prediction will be made for a time point by averaging together values from a number of prior periods
$$Y_t = \frac{1}{2 * k + 1} \sum_{j=-k}^k y_{t+j}$$
  - **exponential smoothing/exponential moving average** = weight time points that are closer to point of prediction than those that are further away
$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha)\hat{y}_{t-1}$$

\* **Note:** many different methods of exponential smoothing are available, more information can be found [here](#)

## R Commands and Examples

- `quantmod` package can be used to pull trading/price information for publicly traded stocks
  - `getSymbols("TICKER", src="google", from=date, to=date)` = gets the **daily** high/low/open/close price and volume information for the specified stock ticker
    - \* returns the data in a data frame under the stock ticker's name
    - \* "TICKER" = ticker of the stock you are attempting to pull information for
    - \* `src="google"` = get price/volume information from Google finance
      - default source of information is Yahoo Finance
    - \* `from` and `to` = from and to dates for the price/volume information
      - both arguments must be specified with `date` objects
    - \* *Note: more information about how to use `getSymbols` can be found in the documentation `?getSymbols`*
  - `to.monthly(GOOG)` = converts stock data to monthly time series from daily data
    - \* the function aggregates the open/close/high/low/volume information for each day into monthly data
    - \* `GOOG` = data frame returned from `getSymbols` function
    - \* *Note: `?to.period` contains documentation for converting time series to OHLC (open high low close) series*
  - `googOpen<-Op(GOOG)` = returns the opening price from the stock data frame
    - \* `C1()`, `Hi()`, `Lo()` = returns the close, high and low price from the stock data frame
  - `ts(googOpen, frequency=12)` = convert data to a time series with `frequency` observations per time unit
    - \* `frequency=12` = number of observations per unit time (12 in this case because there are 12 months in each year → converts data into **yearly** time series)
- `decompose(ts)` = decomposes time series into trend, seasonal, and irregular components by using moving averages
  - `ts` = time series object
- `window(ts, start=1, end=6)` = subsets the time series at the specified starting and ending points
  - `start` and `end` arguments must correspond to the **time unit** rather than the **index**
    - \* for instance, if the `ts` is a yearly series (`frequency = 12`), `start/end` should correspond to the row numbers or year (each year has 12 observations corresponding to the months)
    - \* `c(1, 7)` can be used to specify the element of a particular year (in this case, July of the first year/row)
    - \* *Note: you can use 9.5 or any decimal as values for `start/end`, and the closest element (June of the 9th year in this case) will be used*
    - \* *Note: `end=9-0.01` can be used a short cut to specify "up to 9", since `end = 9` will include the first element of the 9th row*
- `forecast` package can be used for forecasting time series data
  - `ma(ts, order=3)` = calculates the simple moving average for the order specified
    - \* `order=3` = order of moving average smoother, effectively the number of values that should be used to calculate the moving average
  - `ets(train, model="MMM")` = runs exponential smoothing model on training data
    - \* `model = "MMM"` = method used to create exponential smoothing

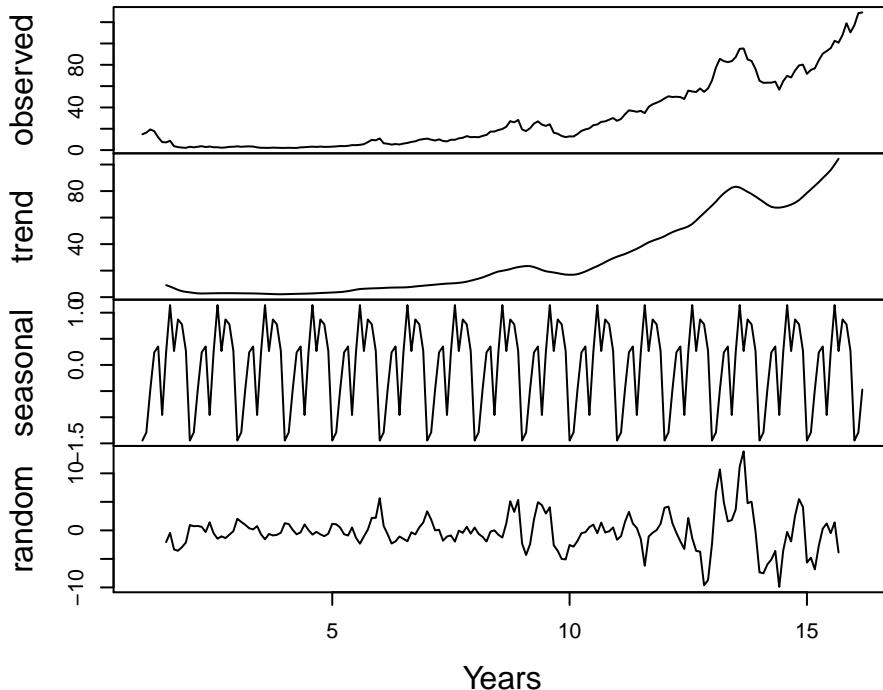
- Note: more information can be found at `?ets` and the corresponding model chart is [here](#)
- `forecast(ts)` = performs forecast on specified time series and returns 5 columns: forecast values, high/low 80 confidence intervals bounds, high/low 95 percent interval bounds
  - \* `plot(forecast)` = plots the forecast object, which includes the training data, forecast values for test periods, as well as the 80 and 95 percent confidence interval regions
- `accuracy(forecast, testData)` = returns the accuracy metrics (RMSE, etc.) for the forecast model
- `quandl` package is also used for finance-related predictions
- example: decomposed time series

```
# load quantmod package
library(quantmod);
# specify to and from dates
from.dat <- as.Date("01/01/00", format="%m/%d/%y")
to.dat <- as.Date("3/2/15", format="%m/%d/%y")
# get data for AAPL from Google Finance for the specified dates
getSymbols("AAPL", src="google", from = from.dat, to = to.dat)

## [1] "AAPL"

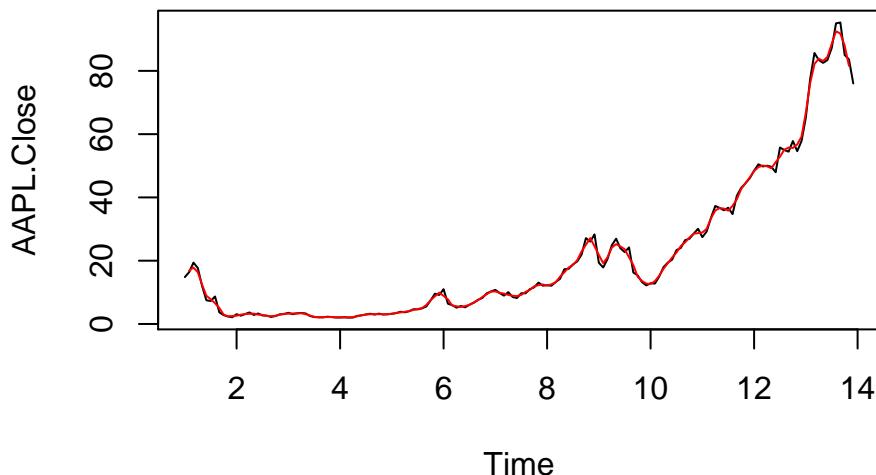
# convert the retrieved daily data to monthly data
mAAPL <- to.monthly(AAPL)
# extract the closing price and convert it to yearly time series (12 observations per year)
ts <- ts(Cl(mAAPL), frequency = 12)
# plot the decomposed parts of the time series
plot(decompose(ts), xlab="Years")
```

## Decomposition of additive time series



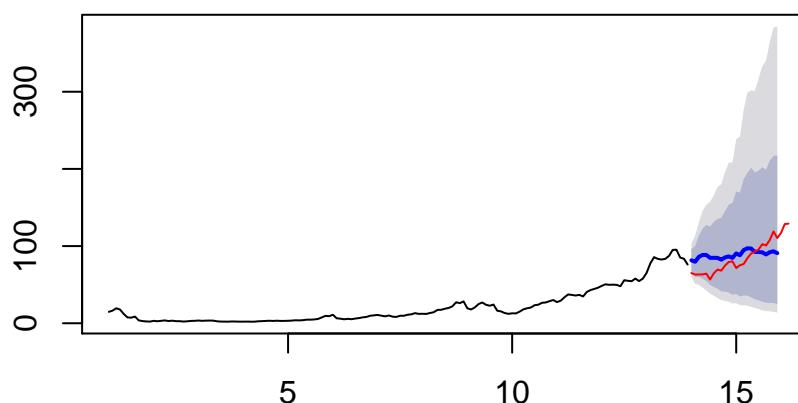
- example: *forecast*

```
# load forecast library
library(forecast)
# find the number of rows (years)
rows <- ceiling(length(ts)/12)
# use 90% of the data to create training set
ts.train <- window(ts, start = 1, end = floor(rows*.9)-0.01)
# use the rest of data to create test set
ts.test <- window(ts, start = floor(rows*.9))
# plot the training set
plot(ts.train)
# add the moving average in red
lines(ma(ts.train,order=3),col="red")
```



```
# compute the exponential smoothing average
ets <- ets(ts.train,model="MMM")
# construct a forecasting model using the exponential smoothing function
fcast <- forecast(ets)
# plot forecast and add actual data in red
plot(fcast); lines(ts.test,col="red")
```

### Forecasts from ETS(M,Md,M)



```
# print the accuracy of the forecast model  
accuracy(fcast,ts.test)
```

```
##               ME      RMSE      MAE      MPE      MAPE      MASE  
## Training set 0.1188298 2.825883 1.646959 -0.61217 10.68901 0.1924329  
## Test set     -7.8132889 16.736910 15.079222 -13.64900 20.31005 1.7618772  
##             ACF1 Theil's U  
## Training set 0.09773823       NA  
## Test set     0.84664431  3.360515
```

## Unsupervised Prediction

- **supervised classification** = predicting outcome when we know what the different classifications are
  - *example:* predicting the type of flower (setosa, versicolor, or virginica) based on sepal width/length
- **unsupervised classification** = predicting outcome when we don't know what the different classifications are
  - *example:* splitting all data for sepal width/length into different groups (cluster similar data together)
- **process**
  - provided that the labels for prediction/outcome are unknown, we first build clusters from observed data
    - \* creating clusters are not noiseless process, and thus may introduce higher variance/error for data
    - \* **K-means** is an example of a clustering approach
  - label the clusters
    - \* interpreting the clusters well (sensible vs non-sensible clusters) is incredibly challenging
  - build prediction model with the clusters as the outcome
    - \* all algorithms can be applied here
  - in new data set, we will predict the clusters labels
- unsupervised prediction is effectively a **exploratory technique**, so the resulting clusters should be carefully interpreted
  - clusters may be highly variable depending on the method through which the data is sample
- generally a good idea to create custom clustering algorithms for given data as it is **crucial** to define the process to identify clusters for interpretability and utility of the model
- unsupervised prediction = basic approach to **recommendation engines**, in which the tastes of the existing users are clustered and applied to new users

## R Commands and Examples

- `kmeans(data, centers=3)` = can be used to perform clustering from the provided data
  - `centers=3` = controls the number of clusters the algorithm should aim to divide the data into
- `cl_predict` function in `clue` package provides similar functionality

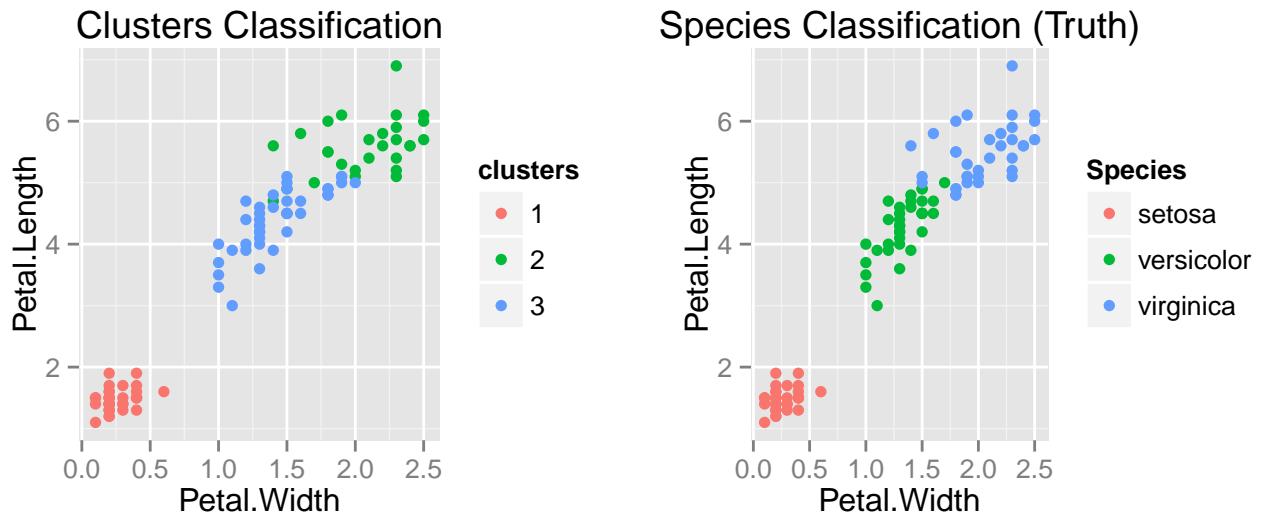
```
# load iris data
data(iris)

# create training and test sets
inTrain <- createDataPartition(y=iris$Species, p=0.7, list=FALSE)
training <- iris[inTrain,]; testing <- iris[-inTrain,]

# perform k-means clustering for the data without the Species information
# Species = what the true clusters are
kMeans1 <- kmeans(subset(training, select=-c(Species)), centers=3)

# add clusters as new variable to training set
training$clusters <- as.factor(kMeans1$cluster)

# plot clusters vs Species classification
p1 <- qplot(Petal.Width, Petal.Length, colour=clusters, data=training) +
  ggtitle("Clusters Classification")
p2 <- qplot(Petal.Width, Petal.Length, colour=Species, data=training) +
  ggtitle("Species Classification (Truth)")
grid.arrange(p1, p2, ncol = 2)
```



- as we can see, there are three clear groups that emerge from the data
  - this is fairly close to the actual results from Species
  - we can compare the results from the clustering and Species classification by tabulating the values

```
# tabulate the results from clustering and actual species
table(kMeans1$cluster, training$Species)
```

```
##
##      setosa versicolor virginica
## 1     35          0         0
## 2      0          3        25
## 3      0         32        10
```

- with the clusters determined, the training data can be trained on all predictors with the clusters from k-means as outcome

```
# build classification trees using the k-means cluster
clustering <- train(clusters ~ ., data=subset(training, select=-c(Species)), method="rpart")
```

- we can compare the prediction results on training set vs truth

```
# tabulate the prediction results on training set vs truth
table(predict(clustering, training), training$Species)
```

```
##
##      setosa versicolor virginica
## 1     35          0         0
## 2      0          0        23
## 3      0         35        12
```

- similarly, we can compare the prediction results on test set vs truth

```
# tabulate the prediction results on test set vs truth
table(predict(clustering,testing),testing$Species)
```

```
##          setosa versicolor virginica
## 1         15          0         0
## 2         0          0        11
## 3         0         15         4
```

# Developing Data Products Course Notes

*Xing Su*

## Contents

Shiny ( <a href="#">tutorial</a> ) . . . . .	3
Structure of Shiny App . . . . .	3
ui.R . . . . .	3
ui.R Example . . . . .	5
server.R . . . . .	7
server.R Example . . . . .	8
Distributing Shiny Application . . . . .	8
Debugging . . . . .	8
manipulate Package . . . . .	9
Example . . . . .	9
rCharts . . . . .	10
Example . . . . .	10
GoogleVis API . . . . .	12
Example (line chart) . . . . .	13
Example (merging graphs) . . . . .	14
ShinyApps.io ( <a href="#">link</a> ) . . . . .	15
plot.ly ( <a href="#">link</a> ) . . . . .	15
Example . . . . .	15
Structure of a Data Analysis Report . . . . .	17
Slidify . . . . .	18
YAML (YAML Ain't Markup Language/Yet Another Markup Language) . . . . .	18
Slides . . . . .	19
Publishing . . . . .	20
RStudio Presentation . . . . .	21
Creating Presentation . . . . .	21
Slidify vs RStudio Presenter . . . . .	22
R Package . . . . .	23
R Package Components . . . . .	23
DESCRIPTION file . . . . .	24
R Code . . . . .	24
NAMESPACE file . . . . .	24

Documentation . . . . .	25
Building/Checking Package . . . . .	26
Checklist for Creating Package . . . . .	27
Example: <code>topen</code> function . . . . .	27
R Classes and Methods . . . . .	29
Objected Oriented Programming in R . . . . .	29
Creating a New Class/Methods . . . . .	32
Yhat ( <a href="#">link</a> ) . . . . .	34
Deploying the Model . . . . .	35
Accessing the Model . . . . .	35

## Shiny ([tutorial](#))

- Shiny = platform for creating interactive R program embedded on web pages (made by RStudio)
- knowledge of these helpful: HTML (web page structure), css (style), JavaScript (interactivity)
- **OpenCPU** = project created by Jerom Ooms providing API for creating more complex R/web apps
- `install.packages("shiny"); library(shiny)` = install/load shiny package
- capabilities
  - upload or download files
  - tabbed main panels
  - editable data tables
  - dynamic UI
  - user defined inputs/outputs
  - submit button to control when calculations/tasks are processed

### Structure of Shiny App

- **two** scripts (in one directory) make up a Shiny project
  - `ui.R` - controls appearance/all style elements
    - \* alternatively, a `www` directory with an `index.html` file enclosed can be used instead of `ui.R`
    - *Note: output is rendered into HTML elements based on matching their id attribute to an output slot and by specifying the requisite css class for the element (in this case either shiny-text-output, shiny-plot-output, or shiny-html-output)*
    - it is possible to create highly customized user-interfaces using user-defined HTML/CSS/JavaScript
  - `server.R` - controls functions
- `runApp()` executes the Shiny application
  - `runApp(display.mode = 'showcase')` = displays the code from `ui.R` and `server.R` and highlights what is being executed depending on the inputs
- *Note: ", " must be included ONLY INBETWEEN objects/functions on the same level*

### `ui.R`

- `library(shiny)` = first line, loads the shiny package
- `shinyUI()` = shiny UI wrapper, contains sub-methods to create panels/part/viewable object
- `pageWithSideBar()` = creates page with main/side bar division
- `headerPanel("title")` = specifies header of the page
- `sideBarPanel()` = specifies parameters/objects in the side bar (on the **left**)
- `mainPanel()` = specifies parameters/objects in the main panel (on the **right**)
- for better control over style, use `shinyUI(fluidpage())` ([tutorial](#)) <- produces responsive web pages
  - `fluidRow()` = creates row of content with width 12 that can be subdivided into columns
    - \* `column(4, ...)` = creates a column of width 4 within the fluid row
    - \* `style = "CSS"` = can be used as the last element of the column to specify additional style
- `absolutePanel(top=0, left=0, right=0)` = used to produce floating panels on top of the page ([documentation](#))
  - `fixed = TRUE` = panel will not scroll with page, which means the panel will always stay in the same position as you scroll through the page

- `draggable = TRUE` = make panel movable by the user
- `top = 40 / bottom = 50` = position from the top/bottom edge of the browser window
  - \* `top = 0, bottom = 0` = creates panel that spans the entire vertical length of window
- `left = 40 / right = 50` = position from the left/right edge of the browser window
  - \* `top = 0, bottom = 0` = creates panel that spans the entire horizontal length of window
- `height = 30 / width = 40` = specifies the height/width of the panel
- `style = "opacity:0.92; z-index = 100"` = makes panel transparent and ensures the panel is always the top-most element

- **content objects/functions**

- *Note:* more HTML tags can be found [here](#)
- *Note:* most of the content objects (`h1`, `p`, `code`, etc) can use **both** double and single quotes to specify values, just be careful to be consistent
- `h1/2/3/4('heading')` = creates heading for the panel
- `p('paragraph')` = creates regular text/paragraph
- `code('code')` = renders code format on the page
- `br()` = inserts line break
- `tags$hr()` = inserts horizontal line
- `tags$ol() / tags$ul()` = initiates ordered/unordered list
- `div( ... , style = "CSS Code") / span( ... , style = "CSS Code")` = used to add additional style to particular parts of the app
  - \* `div` should be used for a section/block, `span` should be used for a specific part/inline
- `withMathJax()` = add this element to allow Shiny to process LaTeX
  - \* inline LaTeX must be wrapped like this: `\\"(LaTeX\\\")`
  - \* block equations are still wrapped by: `$$LaTeX$$`

- **inputs**

- `textInput(inputId = "id", label = "textLabel")` = creates a plain text input field
  - \* `inputId` = field identifier
  - \* `label` = text that appear above/before a field
- `numericInput('HTMLlabel', 'printedLabel', value = 0, min = 0, max = 10, step = 1)` = create a number input field with incrementer (up/down arrows)
  - \* `'HTMLlabel'` = name given to the field, not printed, and can be called
  - \* `'printedLabel'` = text that shows up above the input box explaining the field
  - \* `value` = default numeric value that the field should take; 0 is an example
  - \* `min` = minimum value that can be set in the field (if a smaller value is manually entered, then the value becomes the minimum specified once user clicks away from the field)
  - \* `max` = max value that can be set in the field
  - \* `step` = increments for the up/down arrows
  - \* more arguments can be found in `?numericInput`
- `checkboxGroupInput("id2", "Checkbox", choices = c("Value 1" = "1", ...), selected = "1", inline = TRUE)` = creates a series of checkboxes
  - \* `"id2", "Checkbox"` = field identifier/label
  - \* `choices` = list of checkboxes and their labels
    - `format = "checkboxName" = "fieldIdentifier"`
    - *Note:* `fieldIdentifier` should generally be different from `checkbox` to `checkbox`, so we can properly identify the responses
  - \* `selected` = specifies the checkboxes that should be selected by default; uses `fieldIdentifier` values

- \* `inline` = whether the options should be displayed inline
- `dateInput("fieldID", "fieldLabel")` = creates a selectable date field (dropdown calendar/date picker automatically generated)
  - \* `"fieldID"` = field identifier
  - \* `"fieldLabel"` = text/name displayed above fields
  - \* more arguments can be found in `?dateInput`
- `submitButton("Submit")` = creates a submit button that updates the output/calculations only when the user submits the new inputs (default behavior = all changes update reactively/in real time)
- `actionButton(inputId = "goButton", label = "test")` = creates a button with the specified label and id
  - \* output can be specified for when the button is clicked
- `sliderInput("id", "label", value = 70, min = 62, max = 74, 0.05)` = creates a slider for input
  - \* arguments similar to `numericInput` and more information can be found `?sliderInput`

- **outputs**

- *Note: every variable called here must have a corresponding method corresponding method from the `output` element in `server.R` to render their value*
- `textOutput("fieldId", inline = FALSE)` = prints the value of the variable/field in text format
  - \* `inline = TRUE` = inserts the result inline with the HTML element
  - \* `inline = FALSE` = inserts the result in block code format
- `verbatimTextOutput("fieldId")` = prints out the value of the specified field defined in `server.R`
- `plotOutput('fieldId')` = plots the output ('sampleHist' for example) created from `server.R` script
- `output$test <- renderText({input$goButton}); isolate(paste(input$t1, input$t2))}`
  - = `isolate` action executes when the button is pressed
  - \* `if (input$goButton == 1){ Conditional statements }` = create different behavior depending on the number of times the button is pressed

## ui.R Example

- below is part of the ui.R code for a project on Shiny

```
# load shiny package
library(shiny)
# begin shiny UI
shinyUI(navbarPage("Shiny Project",
  # create first tab
  tabPanel("Documentation",
    # load MathJax library so LaTeX can be used for math equations
    withMathJax(),
    h3("Why is the Variance Estimator  $\sqrt{S^2}$  divided by  $n-1$ "),
    # paragraph and bold text
    p("The ", strong("sample variance"), " can be calculated in ", strong(em("two")),
      " different ways:",
      "$$S^2 = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1}$$",
      "The unbiased calculation is most often used, as it provides a ",
      strong(em("more accurate")), " estimate of population variance"),
    # break used to space sections
    br(), p("To show this empirically, we simulated the following in the ",
```

```

        strong("Simulation Experiment"), " tab: "), br(),
# ordered list
tags$ol(
  tags$li("Create population by drawing observations from values 1 to 20."),
  tags$li("Draw a number of samples of specified size from the population"),
  tags$li("Plot difference between sample and true population variance"),
  tags$li("Show the effects of sample size vs accuracy of variance estimated")
),
# second tab
tabPanel("Simulation Experiment",
  # fluid row for space holders
  fluidRow(
    # fluid columns
    column(4, div(style = "height: 150px")),
    column(4, div(style = "height: 150px")),
    column(4, div(style = "height: 150px"))),
  # main content
  fluidRow(
    column(12,h4("We start by generating a population of ",
      span(textOutput("population", inline = TRUE),
        style = "color: red; font-size: 20px"),
      " observations from values 1 to 20:"),
      tags$hr(),htmlOutput("popHist"),
      # additional style
      style = "padding-left: 20px"
    )
  ),
  # absolute panel
  absolutePanel(
    # position attributes
    top = 50, left = 0, right =0,
    fixed = TRUE,
    # panel with predefined background
    wellPanel(
      fluidRow(
        # sliders
        column(4, sliderInput("population", "Size of Population:",
          min = 100, max = 500, value = 250),
          p(strong("Population Variance: "),
            textOutput("popVar", inline = TRUE))),
        column(4, sliderInput("numSample", "Number of Samples:",
          min = 100, max = 500, value = 300),
          p(strong("Sample Variance (biased): "),
            textOutput("biaVar", inline = TRUE))),
        column(4, sliderInput("sampleSize", "Size of Samples:",
          min = 2, max = 15, value = 10),
          p(strong("Sample Variance (unbiased): "),
            textOutput("unbiaVar", inline = TRUE))),
        style = "opacity: 0.92; z-index: 100;"
      )
    )
  )))

```

## server.R

- preamble/code to set up environment (executed only *once*)
  - start with `library()` calls to load packages/data
  - define/initiate variables and relevant default values
    - \* `<->` operator should be used to assign values to variables in the parent environment
    - \* `x <- x + 1` will define `x` to be the sum of 1 and the value of `x` (defined in the parent environment/working environment)
  - any other code that you would like to only run once
- `shinyServer()` = initiates the server function
  - `function(input, output){}` = defines a function that performs actions on the inputs user makes and produces an output object
  - **non-reactive** statements/code will be executed *once for each page refresh/submit*
  - **reactive** functions/code are **run repeatedly** as values are updated (i.e. render)
    - \* *Note: Shiny only runs what is needed for reactive statements, in other words, the rest of the code is left alone*
    - \* `reactive(function)` = can be used to wrap functions/expressions to create reactive expressions
      - `renderText({x()})` = returns value of `x`, “`()`” must be included (syntax)

- reactive function example

```
# start shinyServer
shinyServer(
# specify input/output function
function(input, output) {
  # set x as a reactive function that adds 100 to input1
  x <- reactive({as.numeric(input$text1)+100})
  # set value of x to output object text1
  output$text1 <- renderText({x()})
  # set value of x plus value of input object text2 to output object text1
  output$text2 <- renderText({x() + as.numeric(input$text2)})
})
```

- functions/output objects in `shinyServer()`

- `output$oid1 <- renderPrint({input$id1})` = stores the user input value in field `id1` and stores the rendered, printed text in the `oid1` variable of the `output` object
  - \* `renderPrint({expression})` = reactive function to render the specified expression
  - \* `{}` is used to ensure the value is an expression
  - \* `oid1` = variable in the output object that stores the result from the subsequent command
- `output$sampleHist <- renderPlot({code})` = stores plot generated by code into `sampleHist` variable
  - \* `renderPlot({code})` = renders a plot generated by the enclosed R code
- `output$sampleGVisPlot <- renderGvis({code})` = renders Google Visualization object

## server.R Example

- below is part of the server.R code for a project on Shiny that uses googleVis

```
# load libraries
library(shiny)
require(googleVis)
# begin shiny server
shinyServer(function(input, output) {
  # define reactive parameters
  pop<- reactive({sample(1:20, input$population, replace = TRUE)})
  bootstrapSample<-reactive({sample(pop(),input$sampleSize*input$numSample,
    replace = TRUE)})
  popVar<- reactive({round(var(pop()),2)})
  # print text through reactive function
  output$biasVar <- renderText({
    sample<- as.data.frame(matrix(bootstrapSample(), nrow = input$numSample,
      ncol =input$sampleSize))
    return(round(mean(rowSums((sample-rowMeans(sample))^2)/input$sampleSize), 2))
  })
  # google visualization histogram
  output$popHist <- renderGvis({
    popHist <- gvisHistogram(data.frame(pop()), options = list(
      height = "300px",
      legend = "{position: 'none'}", title = "Population Distribution",
      subtitle = "samples randomly drawn (with replacement) from values 1 to 20",
      histogram = "{ hideBucketItems: true, bucketSize: 2 }",
      hAxis = "{ title: 'Values', maxAlternation: 1, showTextEvery: 1}",
      vAxis = "{ title: 'Frequency'}"
    ))
    return(popHist)
  })
})
```

## Distributing Shiny Application

- running code locally = running local server and browser routing through local host
- quickest way = send application directory
- possible to create R package and create a wrapper that calls `runApp` (requires R knowledge)
- another option = run shiny server ([link](#))

## Debugging

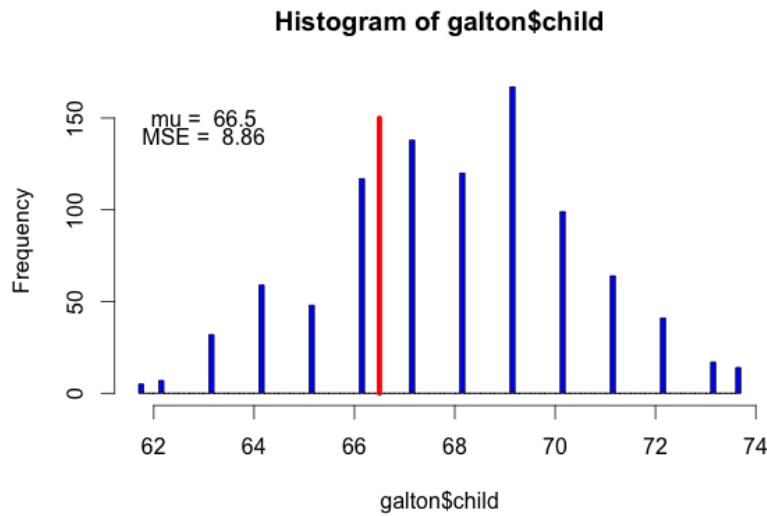
- `runApp(display.mode = 'showcase')` = highlights execution while running a shiny application
- `cat` = can be used to display output to stdout/R console
- `browser()` = interrupts execution ([tutorial](#))

## manipulate Package

- **manipulate** = package/function can be leveraged to create quick interactive graphics by allowing the user to vary the different variables to a model/calculation
- creates sliders/checkbox/picker for the user ([documentation](#))

### Example

```
# load data and manipulate package
library(UsingR)
library(manipulate)
# plotting function
myHist <- function(mu){
  # histogram
  hist(galton$child,col="blue",breaks=100)
  # vertical line to highlight the mean
  lines(c(mu, mu), c(0, 150),col="red",lwd=5)
  # calculate mean squared error
  mse <- mean((galton$child - mu)^2)
  # updates the mean value as the mean is changed by the user
  text(63, 150, paste("mu = ", mu))
  # updates the mean squared error value as the mean is changed by the user
  text(63, 140, paste("MSE = ", round(mse, 2)))
}
# creates a slider to vary the mean for the histogram
manipulate(myHist(mu), mu = slider(62, 74, step = 0.5))
```

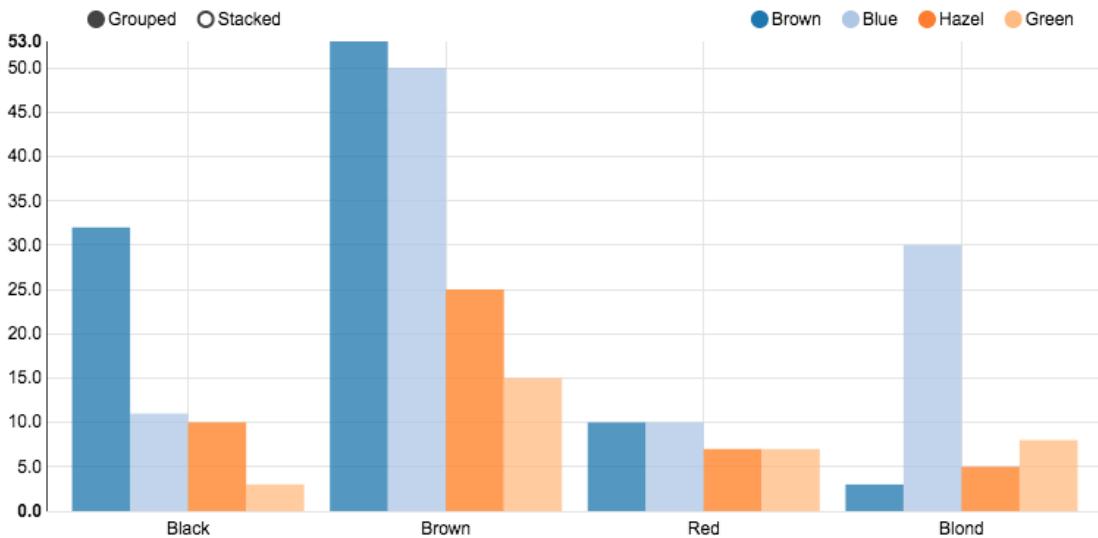


## rCharts

- rCharts = simple way of creating interactive JavaScript visualization using R
  - more in-depth knowledge of D3 will be needed to create more complex tools
  - written by Ramnath Vaidyanathan
  - uses formula interface to specify plots (like the `lattice` plotting system)
  - displays interactive tool tips when hovering over data points on the plots
- installation
  - `devtools` must be installed first (`install.packages("devtools")`)
  - `require(devtools); install_github('rCharts', 'ramnathv')` installs the rCharts package from GitHub
- plot types
  - *Note: each of the following JS library has different default styles as well as a multitude of capabilities; the following list is just what was demonstrated and more documentation can be found in the corresponding links*
  - [Polychart js library] `rPlot` –> paneled scatter plots
  - [Morris js library] `mPlot` –> time series plot (similar to stock price charts)
  - [NVD3 js library] `nPlot` –> stacked/grouped bar charts
  - [`xCharts` js library] –> shaded line graphs
  - [`HighChart` js library] –> stacked (different styles) scatter/line charts
  - [`LeafLet` js library] –> interactive maps
  - [`Rickshaw` js library] –> stacked area plots/time series
- rChart objects have various attributes/functions that you can use when saved `n1 <- nplot(...)`
  - `n1$ + TAB` in R Console brings up list of all functions contained in the object
  - `n1$html()` = prints out the HTML for the plot
  - `n1$save(filename)` = embeds code into slidify document
  - `n1$print()` = print out the JavaScript
- `n1$show("inline", include_assets = TRUE, cdn = F)` = embed HTML/JS code directly with in Rmd file (for HTML output)
  - `n1$publish('plotname', host = 'gist'/'rpubs')` = publishes the plot under the specified plotname as a gist or to rpubs
- to use with `slidify`, the following YAML (Yet Another Markup Language/YAML Ain't Markup Language) must be added
  - `yaml ext_widgets : {rCharts: ["libraries/nvd3"]}`

## Example

```
# load rCharts package
require(rCharts); library(datasets); library(knitr)
# create dataframe with HairEyeColor data
haireye = as.data.frame(HairEyeColor)
# create a nPlot object
n1 <- nPlot(Freq ~ Hair, group = 'Eye', type = 'multiBarChart',
            data = subset(haireye, Sex == 'Male'))
# save the nPlot object to a html page
n1$show("inline", include_assets = TRUE, cdn = F)
```



## GoogleVis API

- GoogleVis allows R to create interactive HTML graphics (Google Charts)
- **chart types** (format = gvis+ChartType)
  - Motion charts: `gvisMotionChart`
  - Interactive maps: `gvisGeoChart`
  - Interactive tables: `gvisTable`
  - Line charts: `gvisLineChart`
  - Bar charts: `gvisColumnChart`
  - Tree maps: `gvisTreeMap`
  - more charts can be found [here](#)
- configuration options and default values for arguments for each of the plot types can be found [here](#)
- `print(chart, "chart")` = prints the JavaScript for creating the interactive plot so it can be embedded in slidify/HTML document
  - `print(chart)` = prints HTML + JavaScript directly
- alternatively, to print the charts on a HTML page, you can use `op <- options(gvis.plot.tag='chart')`
- this sets the googleVis options first to change the behaviour of `plot.gvis`, so that **only the chart component** of the HTML file is written into the output file
- `plot(chart)` can then be called to print the plots to HTML
- `gvisMerge(chart1, chart2, horizontal = TRUE, tableOptions = "bgcolor = \"#CCCCCC\"", cellspacing = 10)` = combines the two plots into one horizontally (1 x 2 panel)
  - *Note: gvisMerge() can only combine TWO plots at a time*
  - `horizontal = FALSE` = combines plots vertically (TRUE for horizontal combination)
  - `tableOptions = ...` = used to specify attributes of the combined plot
- `demo(googleVis)` = demos how each of the plot works
- **resources**
  - [vignette](#)
  - [documentation](#)
  - [plot gallery](#)
  - [FAQ](#)

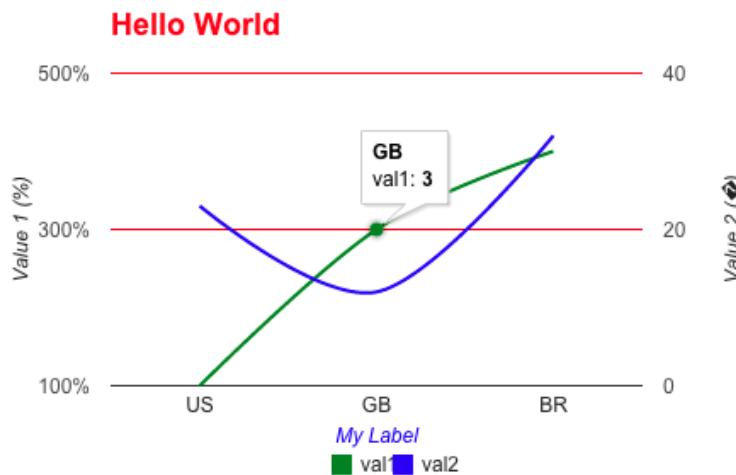
### Example (line chart)

```
# load googleVis package
library(googleVis)

# set gvis.plot options to only return the chart
op <- options(gvis.plot.tag='chart')

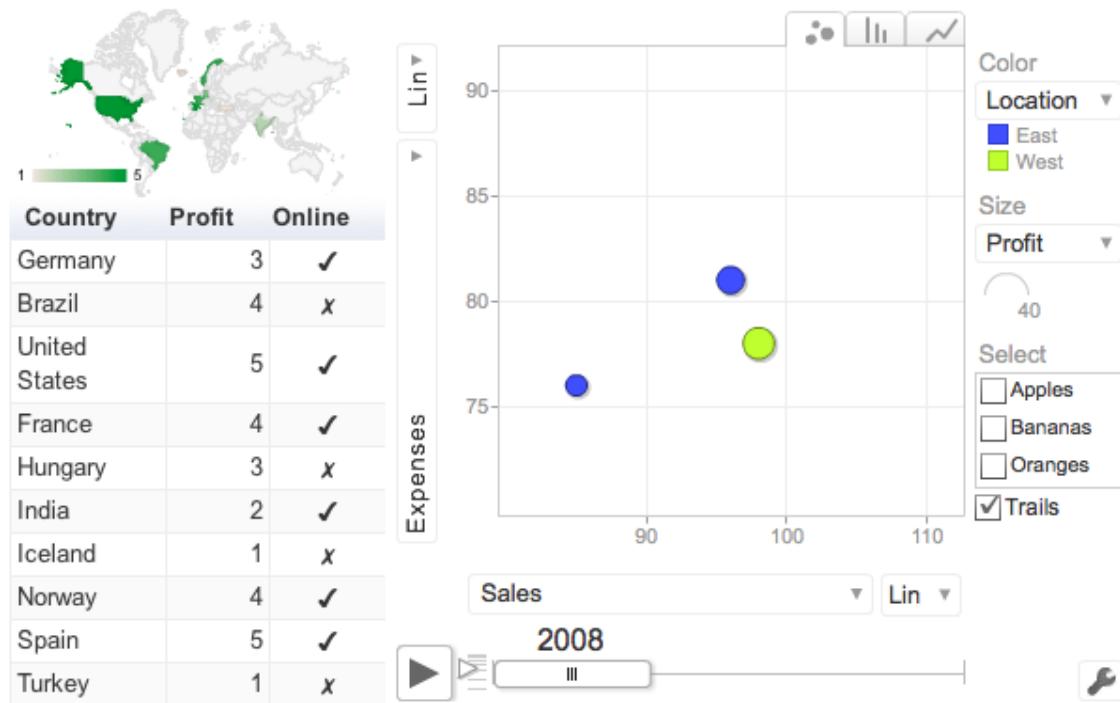
# create initial data with x variable as "label" and y variable as "var1/var2"
df <- data.frame(label=c("US", "GB", "BR"), val1=c(1,3,4), val2=c(23,12,32))

# set up a gvisLineChart with x and y
Line <- gvisLineChart(df, xvar="label", yvar=c("val1","val2"),
                      # set options for the graph (list) - title and location of legend
                      options=list(title="Hello World", legend="bottom",
                                   # set title text style
                                   titleTextStyle="{color:'red', fontSize:18}",
                                   # set vertical gridlines
                                   vAxis="{gridlines:{color:'red', count:3}}",
                                   # set horizontal axis title and style
                                   hAxis="{title:'My Label', titleTextStyle:{color:'blue'}}",
                                   # set plotting style of the data
                                   series="[{color:'green', targetAxisIndex: 0},
                                             {color: 'blue',targetAxisIndex:1}]",
                                   # set vertical axis labels and formats
                                   vAxes="[{title:'Value 1 (%)', format:'##,#####%'},
                                            {title:'Value 2 (\u20ac)'}]",
                                   # set line plot to be smoothed and set width and height of the plot
                                   curveType="function", width=500, height=300
                      ))
# print the chart in JavaScript
plot(Line)
```



## Example (merging graphs)

```
G <- gvisGeoChart(Exports, "Country", "Profit", options=list(width=200, height=100))
T1 <- gvisTable(Exports, options=list(width=200, height=270))
M <- gvisMotionChart(Fruits, "Fruit", "Year", options=list(width=400, height=370))
GT <- gvisMerge(G, T1, horizontal=FALSE)
GTM <- gvisMerge(GT, M, horizontal=TRUE, tableOptions="bgcolor=\"#CCCCCC\" cellspacing=10")
plot(GTM)
```



- *Note:* the motion chart only displays when it is hosted on a server or a trusted Macromedia source, see [googlVis vignette](#) for more details

## ShinyApps.io ([link](#))

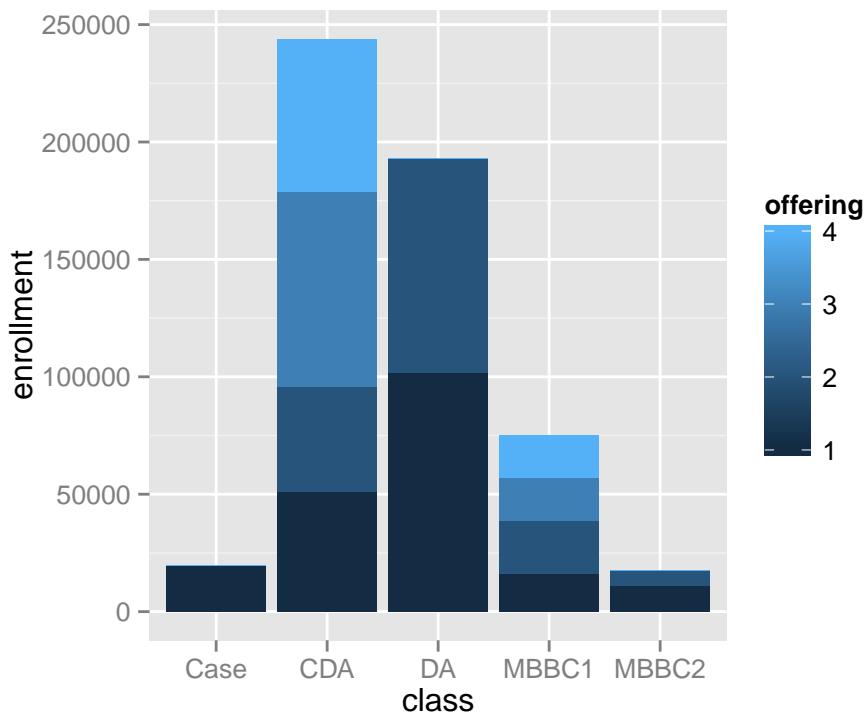
- platform created by RStudio to share Shiny apps on the web
- log in through GitHub/Google, and set up access in R
  1. Make sure you have `devtools` installed in R (`install.packages("devtools")`)
  2. enter `devtools::install_github('rstudio/shinyapps')`, which installs the `shinyapps` package from GitHub
  3. follow the instructions to authenticate your shiny apps account in R through the generated token
  4. publish your app through `deployApp()` command
- the apps you deploy will be hosted on ShinyApps.io under your account

## plot.ly ([link](#))

- platform share and edit plots modularly on the web ([examples](#))
  - every part of the plot can be customized and modified
  - graphs can be converted from one language to another
- you can choose to log in through Facebook/Twitter/Google/GitHub
  1. make sure you have `devtools` installed in R
  2. enter `devtools::install_github("ropensci/plotly")`, which installs `plotly` package from GitHub
  3. go to <https://plot.ly/r/getting-started/> and follow the instructions
  4. enter `library(plotly); set_credentials_file("<username>", "<token>")` with the appropriate username and token filled in
  5. use `plotly()` methods to upload plots to your account
  6. modify any part of the plot as you like once uploaded
  7. share the plot

## Example

```
# load packages
library(plotly); library(ggplot2)
# make sure your plot.ly credentials are set correctly using the following command
#   set_credentials_file(username= FILL IN, api_key= FILL IN)
# load data
load("courseraData.rda")
# bar plot using ggplot2
g <- ggplot(myData, aes(y = enrollment, x = class, fill = offering))
g <- g + geom_bar(stat = "identity")
g
```



```
# initiate plotly object
py <- plotly()
# interface with plot.ly and ggplot2 to upload the plot to plot.ly under your credentials
out <- py$ggplotly(g)
# typing this in R console will return the url of the generated plot
out$response$url
```

```
## [1] "https://plot.ly/~sxing/54"
```

- the above is the response URL for the plot created on plot.ly
- *Note: this particular URL corresponds to the plot on my personal account*

## Structure of a Data Analysis Report

- can be blog post or formal report for analysis of given dataset
- components
  - ***prompt*** file = analysis being performed (this file is not mandatory/available in all cases)
  - ***data folder*** = data to be analyzed + report files
    - \* *Note: always keep track of data provenance (for reproducibility)*
  - ***code folder***= rawcode vs finalcode
    - \* *rawcode* = analysis performed (.Rmd file)
      - all exploratory data analysis
      - not for sharing with others
    - \* *finalcode* = only analysis to be shared with other people/summarizations (.Rmd file)
      - not necessarily final but pertinent to analysis discussed in final report
    - \* ***figures folder*** = final plots that have all appropriate formatting for distribution/presentation
    - \* ***writing folder*** = final analysis report and final figure caption
      - report should have the following sections: title, introduction, methods, results, conclusions, references
      - final figure captions should correspond with the figures created

## Slidify

- create data-centric presentations created by Ramnath Vaidyanathan
- amalgamation of knitr, Markdown, JavaScript libraries for HTML5 presentations
- easily extendable/customizable
- allows embedded code chunks and mathematical formulas (MathJax JS library) to be rendered correctly
- final products are HTML files, which can be viewed with any web browser and shared easily
- **installation**
  1. make sure you have `devtools` package installed in R
  2. enter `install_github('slidify', 'ramnathv');` `install_github('slidifyLibraries', 'ramnathv')` to install the slidify packages
  3. load slidify package with `library(slidify)`
  4. set the working directory to the project you are working on with `setwd("~/project")`
- `author("title")` = sets up initial files for a new slidify project (performs the following things)
  1. `title` (or any name you typed) directory is created inside the current working directory
  2. `assets` subdirectory and a file named `index.Rmd` are created inside `title` directory
  3. `assets` subdirectory is populated with the following empty folders:
    - `css`
    - `img`
    - `js`
    - `layouts`

– *Note: any custom CSS/images/JavaScript you want to use should be put into the above folders correspondingly*
  4. `index.Rmd` R Markdown file will open up in RStudio
- `slidify("index.Rmd")` = processes the R Markdown file into a HTML page and imports all necessary libraries
- `library(knitr); browseURL("index.html")` = opens up the built-in web browser in R Studio and displays the slidify presentation
  - *Note: this is only necessary the first time; you can refresh the page to reflect any changes after saving the HTML file*

## YAML (YAML Ain't Markup Language/Yet Another Markup Language)

- used to specify options for the R Markdown/slidify at the beginning of the file
- format: `field : value # comment`
  - `title` = title of document
  - `subtitle` = subtitle of document
  - `author` = author of document
  - `job` = occupation of author (can be left blank)
  - `framework` = controls formatting, usually the name of a library is used (i.e. `io2012`)
    - \* `io2012`
    - \* `html5slides`
    - \* `deck.js`

- \* dzslides
- \* landslide
- \* Slidy
- **highlighter** = controls effects for presentation (i.e `highlight.js`)
- **hitheme** = specifies theme of code (i.e. `tomorrow`)
- **widgets** = loads additional libraries to display LaTeX math equations(`mathjax`), quiz-styles components (`quiz`), and additional style (`bootstrap` -> Twitter-created style)
  - \* for math expressions, the code should be enclosed in `$expresion$` for inline expressions, and `$$expression$$` for block equations
- **mode** = `selfcontained/standalone/draft` -> depending whether the presentation will be given with Internet access or not
  - \* `standalone` = all the JavaScript libraries will be save locally so that the presentation can be executed without Internet access
  - \* `selfcontained` = load all JavaScript library at time of presentation
- **logo** = displays a logo in title slide
- **url** = specify path to assets/other folders that are used in the presentation
  - \* *Note:* `../` signifies the parent directory

- **example**

```
---
title      : Slidify
subtitle   : Data meets presentation
author     : Jeffrey Leek, Assistant Professor of Biostatistics
job        : Johns Hopkins Bloomberg School of Public Health
logo       : bloomberg_shield.png
framework  : io2012      # {io2012, html5slides, shower, dzslides, ...}
highlighter: highlight.js # {highlight.js, prettify, highlight}
hitheme    : tomorrow      #
url:
  lib: ../../libraries
  assets: ../../assets
widgets   : [mathjax]      # {mathjax, quiz, bootstrap}
mode      : selfcontained # {standalone, draft}
---
```

## Slides

- `##` = signifies the title of the slide -> equivalent of `h1` element in HTML
- `---` = marks the end of a slide
- `.class #id` = assigns `class` and `id` attributes (CSS) to the slide and can be used to customize the style of the page
- *Note:* make sure to leave space between each component of the slidify document (title, code, text, etc) to avoid errors
- advanced HTML can be added directly to the `index.Rmd` file and most of the time it should function correctly
- interactive element (quiz questions, rCharts, shiny apps) can be embedded into slidify documents (`demos`)
  - quiz elements
    - \* `##` = signifies title of questions
    - \* the question can be type in plain text format

- \* the multiple choice options are listed by number (1. a, 2. b, etc.)
  - wrap the correct answer in underscores (2. b)
- \* \*\*\* .hint = denotes the hint that will be displayed when the user clicks on **Show Hint** button
- \* \*\*\* .explanation = denotes the explanation that will be displayed when the user clicks on **Show Answer** button
- \* a page like the one below will be generated when processed with slidify

```
## Question 1
```

What is  $1 + 1$ ?

1. 1
2. 2
3. 3
4. 4

\*\*\* .hint

This is a hint

\*\*\* .explanation

This is an explanation

## Question 1

What is  $1 + 1$ ?

- 1
- 2
- 3
- 4

**Submit**

**Show Hint**

**Show Answer**

**Clear**

- **knit HTML** button can be used to generate previews for the presentation as well

## Publishing

- first, you will need to create a new repository on GitHub
- `publish_github("user", "repo")` can be used to publish the slidify document on to your on-line repo

## RStudio Presentation

- presentation authoring tool within the RStudio IDE ([tutorial](#))
- output = html5 presentation
- `.Rpres` file → converted to `.md` file → `.html` file
- uses R Markdown format as slidify/knitr
  - `mathjax` JS library is loaded by default
- RStudio format/runs the code when the document is saved

## Creating Presentation

- file → New File → R Presentation (`alt + f + p`)
- `class:` `classname` = specify slide-specific control from CSS
- `css:` `file.css` = can be used to import an external CSS file
  - alternatively, a css file that has the same name as the presentation will be automatically loaded
- knowledge of CSS/HTML/JavaScript useful to customize presentation more granularly
  - *Note: though the end HTML file can be edited directly, it should be used as a last resort as it defeats the purpose of reproducible presentations*
- clicking on **Preview** button brings up **Presentation** viewer in RStudio
  - *navigation controls* (left and right arrows) are located in right bottom corner
  - the *Notepad* icon on the menu bar above displays the section of code that corresponds with the current slide in the main window
  - the *More* button has four options
    - \* “Clear Knitr Cache” = clears cache for the generated presentation previews
    - \* “View in Browser” = creates temporary HTML file and opens in default web browser (does not create a local file)
    - \* “Save as Web Page” = creates a copy of the presentation as a web page
    - \* “Publish to RPubs” = publishes presentation on RPubs
  - the *Refresh* button refreshes the page
  - the *Zoom* button opens a new window to display the presentation
- **transitions between slides**
  - just after the beginning of each slide, the `transition` property (similar to YAML) can be specified to control the transition between the previous and current slides
  - `transition: linear` = creates 2D linear transition (html5) between slides
  - `transition: rotate` = creates 3D rotating transition (html5) between slides
  - more transition options are found [here](#)
- **hierarchical organization**
  - attribute `type` can be added to specify the appearance of the slide (“slide type”)
  - `type: section` and `type: sub-section` = distinct background and font colors, slightly larger heading text, appear at a different indent level within the slide navigation menu
  - `type: prompt` and `type: alert` = distinct background color to communicate to viewers that the slide has different intent
- **columns**
  - simply place `***` in between two sections of content on a slide to separate it into two columns
  - `left: 70%` can be used to specify the proportions of each column

- right: 30% works similarly
- change slide font ([guide](#))
  - font-family: fontname = changes the font of slide (specified in the same way as HTML)
  - font-import: <http://fonts.googleapis.com/css?family=Risque> = imports font
    - \* *Note: fonts must be present on the system for presentation (or have Internet), or default fonts will be used*
  - *Note: CSS selectors for class and IDs must be preceded by .reveal to work (.reveal section del applies to any text enclosed by ~~text~~)*

## Slidify vs RStudio Presenter

- Slidify
  - flexible control from the `.Rmd` file
  - under constant development
  - large user base, more likely to get answer on *StackOverflow*
  - lots of styles and options by default
  - steeper learning curve
  - more command-line oriented
- R Studio Presenter
  - embedded in R Studio
  - more GUI oriented
  - very easy to get started
  - smaller set of easy styles and options
  - default styles look nice
  - as flexible as Slidify with CSS/HTML knowledge

## R Package

- R packages = collection of functions/data objects, extends base functionality of R
  - organized to provide consistency
  - written by people all over the world
- primarily available from CRAN and Bioconductor
  - installed with `install.packages()`
- also available on GitHub/BitBucket/etc
  - installed using `devtools::install_github()`
- **documentation/vignettes** = forces author to provide detailed explanation for the arguments/results of their functions and objects
  - allows for well defined Application Programming Interface (API) to tell users what and how to use the functions
  - much of the implementation details can be hidden from the user so that updates can be made without interfering with use cases
- if package is available on CRAN then it must hold standards of reliability and robustness
- fairly easily maintained with proper documentation
- **package development process**
  - write code in R script
  - desire to make code available to others
  - incorporate R script file into R package structure
  - write documentation for user functions
  - include examples/demos/datasets/tutorials
  - package the contents
  - submit to CRAN/Bioconductor
  - push source code repository to GitHub/other code sharing site
    - \* people find problems with code and expect the author to fix it
    - \* alternatively, people might fix the problem and show the author the changes
  - incorporate changes and release a new version

## R Package Components

- **directory** with name of R package = created as first step
- **DESCRIPTION file** = metadata/information about package
- **R code** = code should be in `R/` sub-directory
- **Documentation** = file should be in `man/` sub-directory
- **NAMESPACE** = optional but common and best practice
- full requirements documented in [Writing R Extensions](#)

## DESCRIPTION file

- **Package** = name of package (e.g. `library(name)` to load the package)
- **Title** = full name of package
- **description** = longer description of package in one or two sentences
- **Version** = version number (usually `M.m-p` format, “`majorNumber.minorNumber-patchLevel`”)
- **Author** = Name of the original author(s)
- **Maintainer** = name + email of person (maintainer) who fixes problems
- **License** = license for the source code, describes the term that the source code is released under
  - common licenses include GNU/BSD/MIT
  - typically a standard open source license is used
- optional fields
  - **Depends** = R packages that your package depends on
  - **Suggests** = optional R packages that users may want to have installed
  - **Date** = release date in `YYYY-MM-DD` format
  - **URL** = package home page/link to repository
  - **Other** = fields can be added (generally ignored by R)
- **example:** `gpclib`
  - **Package:** gpclib
  - **Title:** General Polygon Clipping Library for R
  - **Description:** General polygon clipping routines for R based on Alan Murta’s C library
  - **Version:** 1.5-5
  - **Author:** Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu) with contributions from Duncan Murdoch and Barry Rowlingson; GPC library by Alan Murta
  - **Maintainer:** Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu)
  - **License:** file LICENSE
  - **Depends:** R ( $\geq 2.14.0$ ), methods
  - **Imports:** graphics
  - **Date:** 2013-04-01
  - **URL:** <http://www.cs.man.ac.uk/~toby/gpc/>, <http://github.com/rdpeng/gpclib>

## R Code

- copy R code to R/ directory
- can be any number of files
- separate out files to logical groups (read/fit models)
- all code should be included here and not anywhere else in the package

## NAMESPACE file

- effectively an API for the package
- indicates which functions are *exported*  $\rightarrow$  public functions that users have access to and can use
  - functions not exported cannot be called directly by the user
  - hides the implementation details from the users (clean package interface)
- lists all dependencies on other packages/indicate what functions you *imported* from other packages
  - allows for your package to use other packages without making them visible to the user
  - importing a function loads the package but *does not* attach it to the search list

- **key directives**

- `export("\<function>")` = export a function
- `import("\<package>")` = import a package
- `importFrom("\<package>", "\<function>")` = import specific function from a package
- `exportClasses("\<class>")` = indicate the new types of S4 (4<sup>th</sup> version of *S*) classes created with the package (objects of the specified class can be created)
- `exportMethods("\<generic>")` = methods that can operate on the new class objects
- *Note:* though they look like R functions, the above directives are not functions that users can use freely

- **example**

```
# read.polyfile/write.polyfile are functions available to user
export("read.polyfile", "write.polyfile")
# import plot function from graphics package
importFrom(graphics, plot)
# gpc.poly/gpc.poly.nohole classes can be created by the user
exportClasses("gpc.poly", "gpc.poly.nohole")
# the listed methods can be applied to the gpc.poly/gpc.poly.nohole classes
exportMethods("show", "get.bbox", "plot", "intersect", "union", "setdiff",
             "[", "append.poly", "scale.poly", "area.poly", "get.pts",
             "coerce", "tristrip", "triangulate")
```

## Documentation

- documentation files (.Rd) should be placed in the `man/` sub-directory
- written in specific markup language
- required for every exported function available to the user (serves to limit the number of exported functions)
- concepts/package/datasets overview can also be documented

- **components**

- `\name{}` = name of function
- `\alias{}` = anything listed as alias will bring up the help file (`?line` is the same as `?residuals.tukeyline`)
  - \* multiple aliases possible
- `\title{}` = full title of the function
- `\description{}` = full description of the purpose of function
- `\usage{}` = format/syntax of function
- `\arguments{}` = explanation of the arguments in the syntax of function
- `\details{}` = notes/details about limitation/features of the function
- `\value{}` = specifies what object is returned
- `\reference{}` = references for the function (paper/book from which the method is created)

- **example: line function**

```

\name{line}
\alias{line}
\alias{residuals.tukeyline}
\title{Robust Line Fitting}
\description{
  Fit a line robustly as recommended in \emph{Exploratory Data Analysis}.
}
\usage{
  line(x, y)
}
\arguments{
  \item{x, y}{the arguments can be any way of specifying x-y pairs. See
    \code{\link{xy.coords}}.}
}
\details{
  Cases with missing values are omitted.

  Long vectors are not supported.
}
\value{
  An object of class \code{"tukeyline"}.

  Methods are available for the generic functions \code{coef},
  \code{residuals}, \code{fitted}, and \code{print}.
}
\references{
  Tukey, J. W. (1977).
  \emph{Exploratory Data Analysis},
  Reading Massachusetts: Addison-Wesley.
}

```

## Building/Checking Package

- **R CMD (Command) Build** = command-line program that creates a package archive file (format = `.tar.gz`)
- **R CMD Check** = command-line program that runs a battery of tests on the package to ensure structure is consistent/all components are present/export and import are appropriately specified
- R CMD Build/R CMD check can be run from the command-line using terminal/command-shell applications
- alternatively, they can be run from R using the `system()` function
  - `system("R CMD build newpackage")`
  - `system("R CMD check newpackage")`
- the package must pass *all* tests to be put on CRAN
  - documentation exists for all exported function
  - code can be loaded without any major coding problems/errors
  - contains license
  - ensure examples in documentation can be executed
  - check documentation of arguments matches argument in the code
- `package.skeleton()` function in the `utils` package = creates a “skeleton” R package

- automatically creates directory structure (`R/`, `man/`), DESCRIPTION file, NAMESPACE file, documentation files
- if there are any visible/stored functions in the current workspace, their code will be written as R code files in the `R/` directory
- documentation stubs are created in `man/` directory
- the rest of the content can then be modified and added
- alternatively, you can click on the menu on the top right hand corner of RStudio: *Project* → *New Project* → *New Directory* → *R Package* → fill in package names and details → automatically generate structure/skeleton of a new R package

### Checklist for Creating Package

- create a new directory with `R/` and `man/` sub-directories (or just use `package.skeleton()`)
- write a DESCRIPTION file
- copy R code into the `R/` sub-directory
- write documentation files in `man/` sub-directory
- write a NAMESPACE file with exports/imports
- build and check

### Example: `topen` function

- when creating a package, generate skeleton by clicking on *Project* → *New Project* → *New Directory* → *R Package* → fill in package names and details
- write the code first in a `.R` script and add documentation directly to the script
  - **Roxygen2** package will be leveraged to extract and format the documentation from R script automatically
- Roxygen2 syntax
  - `#'` = denotes the beginning of documentation
    - \* R will automatically add `#'` on the subsequent lines as you type or complete sections
    - title should be on the first line (relatively concise, a few words)
      - \* press **ENTER** after you are finished and R will automatically insert an empty line and move the cursor to the next section
    - description/summary should begin on the third line (one/two sentences)
      - \* press **ENTER** after you are finished and R will automatically insert an empty line and move the cursor to the next section
  - `@param x definition` = format of the documentation for the arguments
    - \* `x` = argument name (formatted in code format when processed to differentiate from definition)
    - \* `definition` = explanation of what `x` represents
  - `@author` = author of the function
  - `@details` = detailed description of the function and its purpose
  - `@seealso` = links to relevant functions used in creating the current function that may be of interest to the user
  - `@import package function` = imports specific function from specified package
  - `@export` = denotes that this function is exported for public use
  - `@return` = specifies what is returned by the method

```

#' Building a Model with Top Ten Features
#'
#' This function develops a prediction algorithm based on the top 10 features
#' in 'x' that are most predictive of 'y'.
#'
#' @param x a n x p matrix of n observations and p predictors
#' @param y a vector of length n representing the response
#' @return a 'lm' object representing the linear model with the top 10 predictors
#' @author Roger Peng
#' @details
#' This function runs a univariate regression of y on each predictor in x and
#' calculates the p-value indicating the significance of the association. The
#' final set of 10 predictors is the taken from the 10 smallest p-values.
#' @seealso \code{lm}
#' @import stats
#' @export

topten <- function(x, y) {
  p <- ncol(x)
  if(p < 10)
    stop("there are less than 10 predictors")
  pvalues <- numeric(p)
  for(i in seq_len(p)) {
    fit <- lm(y ~ x[, i])
    summ <- summary(fit)
    pvalues[i] <- summ$coefficients[2, 4]
  }
  ord <- order(pvalues)
  x10 <- x[, ord]
  fit <- lm(y ~ x10)
  coef(fit)
}

#' Prediction with Top Ten Features
#'
#' This function takes a set coefficients produced by the \code{topten}
#' function and makes a prediction for each of the values provided in the
#' input 'X' matrix.
#'
#' @param X a n x 10 matrix containing n observations
#' @param b a vector of coefficients obtained from the \code{topten} function
#' @return a numeric vector containing the predicted values

predict10 <- function(X, b) {
  X <- cbind(1, X)
  drop(X %*% b)
}

```

## R Classes and Methods

- represents new data types (i.e. list) and its own structure/functions that R doesn't support yet
- R classes and method → system for object oriented programming (OOP)
- R is interactive and supports OOP where as a lot of the other common OOP languages (C++, Java, Python, Perl) are generally not interactive
- John Chambers wrote most of the code support classes/methods (documented in *Programming with Data*)
- goal is to allow *user* (leverage system capabilities) to become *programmers* (extend system capabilities)
- OOB structure in R is structured differently than most of the other languages
- **two style of classes and methods**
  - S3 (version three of the *S* language)
    - \* informal, “old-style”, kludgey (functional but not elegant)
  - S4 (version four of the *S* language)
    - \* rigorous standard, more formal, “new-style”
    - \* code for implementing S4 classes and methods found in `methods` package
  - two systems living side by side
    - \* each is independent but S4 are encouraged for new projects

## Objected Oriented Programming in R

- **class** = description of some thing/object (new data type/idea)
  - can be defined using `setClass()` function in `methods` package
  - all objects in R have a class, which can be determined through the `class()` function
    - \* `numeric` = number data, can be vectors as well (series of numbers)
    - \* `logical` = TRUE, FALSE, NA
      - *Note:* NA is by default of logical class, however you can have numeric/character NA's as results of operations
    - \* `character` = string of characters
    - \* `lm` = linear model class, output from a linear model
- **object** = instances of a class
  - can be created using `new()`
- **method** = function that operates on certain class of objects
  - also can be thought of as an implementation of a *generic function* for an object of particular class
  - can write new method for existing generic functions or create new generic function and associated methods
  - `getS3method(<genericFunction>, <class>)` = returns code for S3 method for a given class
    - \* some S3 methods can be called directly (i.e. `mean.default`)
    - \* but you should **never** call them, always use the generic
  - `getMethod(<genericFunction>, <signature/class>)` = returns code for S4 method for a given class
    - \* *signature* = character vector indicating class of objects accepted by the method

- \* S4 methods can not be called at all
- **generic function** = R function that dispatches methods to perform a certain task (i.e. `plot`, `mean`, `predict`)
  - performs different calculations depending on context
  - *Note: generic functions themselves don't perform any computation; typing the function name by itself (i.e. `plot`) will return the content of the function*
  - S3 and S4 functions look different but are similar conceptually
  - `methods("mean")` = returns methods associated with S3 generic function
  - `showMethods("show")` = returns methods associated with S4 generic function
    - \* *Note: show is equivalent of print, but generally not called directly as objects are auto-printed*
  - first argument = object of particular class
  - **process:**
    1. generic function checks class of object
    2. if appropriate method for class exists, call that method on object (process complete)
    3. if no method exists for class, search for default method
    4. if default method exists, default method is called on object (process complete)
    5. if no default method exists, error thrown (process complete)
  - for classes like `data.frame` where each column can be of different class, the function uses the methods correspondingly
    - \* plotting `as.ts(x)` and `x` are completed different
      - `as.ts()` = converts object to time series
- *Note: ?Classes, ?Methods, ?setClass, ?setMethod, and ?setGeneric contains very helpful documentation*
- **example**

```
# S3 method: mean
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fbde4a7a1c8>
## <environment: namespace:base>

# associated methods
methods("mean")

## [1] mean.Date      mean.default   mean.difftime mean.POSIXct  mean.POSIXlt

# code for mean (first 10 lines)
# note: no specific function got numeric class, so default is used
head(getS3method("mean", "default"), 10)

## 
## 1  function (x, trim = 0, na.rm = FALSE, ...)
## 2  {
## 3    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
## 4      warning("argument is not numeric or logical: returning NA")
```

```

## 5      return(NA_real_)
## 6  }
## 7  if (na.rm)
## 8    x <- x[!is.na(x)]
## 9  if (!is.numeric(trim) || length(trim) != 1L)
## 10    stop("'trim' must be numeric of length one")

```

```

# S4 method: show
show

```

```

## standardGeneric for "show" defined from package "methods"
##
## function (object)
## standardGeneric("show")
## <bytecode: 0x7fbde3cd5c78>
## <environment: 0x7fbde2f6af50>
## Methods may be defined for arguments: object
## Use showMethods("show") for currently available ones.
## (This generic function excludes non-simple inheritance; see ?setIs)

```

```

# associated methods
showMethods("show")

```

```

## Function: show (package methods)
## object="ANY"
## object="C++Class"
## object="C++Function"
## object="C++Object"
## object="classGeneratorFunction"
## object="classRepresentation"
## object="color"
## object="Enum"
## object="EnumDef"
## object="envRefClass"
## object="function"
##   (inherited from: object="ANY")
## object="genericFunction"
## object="genericFunctionWithTrace"
## object="MethodDefinition"
## object="MethodDefinitionWithTrace"
## object="MethodSelectionReport"
## object="MethodWithNext"
## object="MethodWithNextWithTrace"
## object="Module"
## object="namedList"
## object="ObjectsWithPackage"
## object="oldClass"
## object="refClassRepresentation"
## object="refMethodDef"
## object="refObjectGenerator"
## object="signature"
## object="sourceEnvironment"
## object="standardGeneric"

```

```

##      (inherited from: object="genericFunction")
## object="SymbolicConstant"
## object="traceable"

```

## Creating a New Class/Methods

- reason for creating new classes/data type (not necessarily unknown to the world, but just unknown to R)
  - powerful way to extend the functionality of R
  - represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
  - new concepts/ideas that haven't been thought of yet (e.g. a fitted point process model, mixed-effects model, a sparse matrix)
  - abstract/hide implementation details from the user
- **classes** = define new data types
- **methods** = extend *generic functions* to specify the behavior of generic functions on new classes
- **setClass()** = function to create new class
  - at minimum, name of class needs to be specified
  - *slots* or attributes can also be specified
    - \* a class is effectively a list, so slots are elements of that list
- **setMethod()** = define methods for class
  - @ is used to access the slots/attributes of the class
- **showClass()** = displays definition/information about class
- when drafting new class, new methods for **print**, **show**, **summary**, and **plot** should be written
- **Note:** creating classes are not something to be done on the console and are much better suited for a script
- **example**
  - create **polygon** class with set of (x, y) coordinates with **setClass()**
  - define a new plot function by extending existing **plot** function with **setMethod()**

```

# load methods library
library(methods)

# create polygon class with x and y coordinates as slots
setClass("polygon", representation(x = "numeric", y = "numeric"))
# create plot method for polygon class (polygon = signature in this case)
setMethod("plot", "polygon",
  # create function
  function(x, y, ...) {
    # plot the x and y coordinates
    plot(x@x, x@y, type = "n", ...)
    # plots lines between all (x, y) pairs
    # x@x[1] is added at the end because we need
    # to connect the last point of polygon to the first
    xp <- c(x@x, x@x[1])
    yp <- c(x@y, x@y[1])
    lines(xp, yp)
  })

```

```
## Creating a generic function for 'plot' from package 'graphics' in the global environment
## [1] "plot"

# print polygon method
showMethods("plot")

## Function: plot (package graphics)
## x="ANY"
## x="color"
## x="polygon"
```

## Yhat ([link](#))

- develop back-ends to algorithms to be hosted on-line for other people to access
- others can create APIs (front-ends) to leverage the algorithm/model
- before uploading, create an account and set up API key

```
## Create dataset of PM and O3 for all US taking year 2013 (annual
## data from EPA)

## This uses data from
## http://aqsdr1.epa.gov/aqsweb/aqstmp/airdata/download_files.html

## Read in the 2013 Annual Data
d <- read.csv("annual_all_2013.csv", nrow = 68210)
# subset data to just variables we are interested in
sub <- subset(d, Parameter.Name %in% c("PM2.5 - Local Conditions", "Ozone")
               & Pollutant.Standard %in% c("Ozone 8-Hour 2008", "PM25 Annual 2006"),
               c(Longitude, Latitude, Parameter.Name, Arithmetic.Mean))
# calculate the average pollution for each location
pollavg <- aggregate(sub[, "Arithmetic.Mean"],
                      sub[, c("Longitude", "Latitude", "Parameter.Name")],
                      mean, na.rm = TRUE)
# refactors the Name parameter to drop all other levels
pollavg$Parameter.Name <- factor(pollavg$Parameter.Name, labels = c("ozone", "pm25"))
# renaming the last column from "x" (automatically generated) to "level"
names(pollavg)[4] <- "level"

# Remove unneeded objects
rm(d, sub)
# extract out just the location information for convenience
monitors <- data.matrix(pollavg[, c("Longitude", "Latitude")])
# load fields package which allows us to calculate distances on earth
library(fields)
# build function to calculate the distances for the given set of coordinates
# input = lon (longitude), lat (latitude), radius (radius in miles for finding monitors)
pollutant <- function(df) {
    # extract longitude/latitude
    x <- data.matrix(df[, c("lon", "lat")])
    # extract radius
    r <- df$radius
    # calculate distances between all monitors and input coordinates
    d <- rdist.earth(monitors, x)
    # locations for find which distance is less than the input radius
    use <- lapply(seq_len(ncol(d)), function(i) {
        which(d[, i] < r[i])
    })
    # calculate levels of ozone and pm2.5 at each selected locations
    levels <- sapply(use, function(idx) {
        with(pollavg[idx, ], tapply(level, Parameter.Name, mean))
    })
    # convert to data.frame and transpose
    dlevel <- as.data.frame(t(levels))
    # return the input data frame and the calculated levels
    data.frame(df, dlevel)
```

```
}
```

## Deploying the Model

- once the functions are complete, three more functions should be written in order to upload to yhat,
  - `model.require()`{} = defines dependencies on other packages
    - \* if there are no dependencies, this does not need to be defined
  - `model.transform()`{} = needed if the data needs to be transformed in anyway before feeding into the model
  - `model.predict()`{} = performs the prediction
- store the following information as a vector named `yhat.config`
  - `username = "<user@email.com>"` = user name for yhat website
  - `apikey = "<generatedKey>"` = unique API key generated when you open an account with yhat
  - `env="http://sandbox.yhathq.com/"` = software environment (always going to be this link)
- `yhat.deploy("name")` = uploads the model to yhat servers with provided credentials under the specified name
  - returns a data frame with status/model information

```
## Send to yhat
library(yhattr)

model.require <- function() {
    library(fields)
}

model.transform <- function(df) {
    df
}

model.predict <- function(df) {
    pollutant(df)
}

yhat.config  <- c(
    username="email@gmail.com",
    apikey="90d2a80bb532cabb2387aa51ac4553cc",
    env="http://sandbox.yhathq.com/"
)

yhat.deploy("pollutant")
```

## Accessing the Model

- once uploaded, the model can be accessed directly on the yhat website
  - click on name of the model after logging in or go to "<http://cloud.yhathq.com/model/name>" where name is the name you uploaded the model under
  - enter the inputs in JSON format (associative arrays): { "variable" : "value"}
  - \* example: { "lon" : -76.61, "lat": 39.28, "radius": 50 }

- click on **Send Data to Model**
- results return in the *Model Response* section
- the model can also be accessed from R directly through the `yhat.predict` function
  - store the data you want to predict on in a data frame with the correct variable names
  - set up the configuration information through `yhat.config` (see above section)
  - `yhat.predict("name", df)` = returns the result by feeding the input data to the model hosted on yhat under your credentials
  - can be applied to multiple rows of data at the same time

```
library(yhattr)
yhat.config <- c(
  username="email@gmail.com",
  apikey="90d2a80bb532cabb2387aa51ac4553cc",
  env="http://sandbox.yhathq.com/"
)
df <- data.frame(lon = c(-76.6167, -118.25), lat = c(39.2833, 34.05),
                 radius = 20)
yhat.predict("pollutant", df)
```

- the model can also be accessed from command line interfaces (CLI) such as cmd on Windows and terminal on Mac

```
curl -X POST -H "Content-Type: application/json" \
--user email@gmail.com:90d2a80bb532cabb2387aa51ac4553cc \
--data '{ "lon" : -76.61, "lat": 39.28, "radius": 50 }' \
http://cloud.yhathq.com/rdpeng@gmail.com/models/pollutant/
```

- *additional example*

```
# load library
library(yhattr)
# yhat functions
model.require <- function() {}
model.transform <- function(df) {
  transform(df, Wind = as.numeric(as.character(Wind)),
            Temp = as.integer(as.character(Temp)))
}
model.predict <- function(df) {
  result <- data.frame(Ozone = predict(fit, newdata = df))
  cl <- data.frame(clWind = class(df$Wind), clTemp = class(df$Temp))
  data.frame(result, Temp = as.character(df$Temp),
             Wind = as.character(df$Wind), cl)
}
# model
fit <- lm(Ozone ~ Wind + Temp, data = airquality)
# configuration
yhat.config <- c(
  username="email@gmail.com",
  apikey="90d2a80bb532cabb2387aa51ac4553cc",
  env="http://sandbox.yhathq.com/"
)
```

```
# deploy to yhat
yhat.deploy("ozone")
# predict using uploaded model
yhat.predict("ozone", data.frame(Wind = 9.7, Temp = 67))
```