Dr. Wolfgang Rolke
University of Puerto Rico - Mayaguez
Terascale Statistics School 2017
DESY - Hamburg

Table of Content

Type Conversion

Vector Arithmetic

Character Manipulation

Basic Statistics

Model Notation and Regression/Fitting

Random Variates and Probability Functions

R as a Computer Language

Customizing R: .First and .Rprofile

Graphics

Rcpp – Use C++ code in R

ROOT-R Interface

What is R, and why should you care?

- Simply the best computer program for analyzing data. (Think ROOT for Statisticians)
- Used by every professional Statistician in the world today
- If it exists, it exists in R
- Runs on Windows, Mac(OS) and Linux
- It is a full fledged computer language
- And best of all: it’s free!

History of R

- S: language for data analysis developed at Bell Labs circa 1976
- Licensed by *AT&T/Lucent* to *Insightful Corp*. Product name: *S-plus (Cost: $1000+)*.
- R: initially written & released as an open source software by Ross Ihaka and Robert Gentleman at U Auckland during 90s (R plays on name “S”)
- Since 1997: international R-core team ~15 people & 1000s of code writers and statisticians happy to share their libraries!

Installation

Installation is fully automatic. For installation files go the R homepage at CRAN (**C**omprehensive **R** **A**rchive **N**etwork):

https://cran.r-project.org/

When starting R (after some stuff) you get to the R prompt: >
Now it is time to type in the commands.

There a number of GUIs out there (for example Rcdr) but it is much better to learn to use R directly. (You guys are used to this anyway!)

Many people run R via RStudio, also free, which has some nice tools to look at data sets etc.

and Running R

R is project-centric: everything belonging to a project (data and programs) are in one file with the extension .RData

You can download the file for this talk: RDesy.RData

use up and down arrow keys to recall commands. Also history()

use rm() to remove objects

to finish an R session type q() or click on the x in the upper right corner

Be careful: R does NOT save anything automatically.

All my projects have a routine called sv. Running sv() saves everything to the folder and file I want. I run this basically every 10 minutes or so.

```
> sv <- function ()  {
    save.image(paste(getwd(),"/Desy2017.RData",sep=""))
}
```

Help!

R has a help system that is quite good (especially for a freeware program)

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)

> args(mean)                    # doesn't always work so good…
function (x, ...)

> ?mean                         # or help(mean)
```

brings up a web browser with everything you want to know about the mean command:

- ✓ List of arguments
- ✓ List of output values
- ✓ References
- ✓ related functions
- ✓ Examples

R User Groups

Further Help: Tutorials

Each of the following tutorials are in PDF format.

- P. Kuhnert & B. Venables, An Introduction to R: Software for Statistical Modeling & Computing
- J.H. Maindonald, Using R for Data Analysis and Graphics
- B. Muenchen, R for SAS and SPSS Users
- W.J. Owen, The R Guide
- D. Rossiter, Introduction to the R Project for Statistical Computing for Use at the ITC
- W.N. Venebles & D. M. Smith, An Introduction to R

Further Help: Web Links

- Paul Geissler's R tutorial
- Dave Robert's Excellent Labs on Ecological Analysis
- Excellent Tutorials by David Rossitier
- Excellent tutorial an nearly every aspect of R (c/o Rob Kabacoff
- Introduction to R by Vincent Zoonekynd
- R Cookbook
- Data Manipulation Reference

Dozens (Hundreds?) of books on all aspects of R

Packages

Packages (Libraries) are a great way to add functionality to R. They are essentially collections of function and data sets designed for specific tasks.

There are strict rules what a package has to include, for example help files for each function. This makes writing them a bit of work, but makes using them much easier!

Installation:

```
> install.packages("ggplot2")
```

better:

```
> install.packages("ggplot2", lib =
"C:/R/library“,repos="http://cran.rstudio.com/")
```

```
> library(ggplot2)
```

to see what libraries are loaded:

```
> search()
 [1] ".GlobalEnv"        "package:ggplot2"   "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

To see what is in a library use ls(k) where k is the position of the library in the search list:

```
> ls(2)[1:3]                                  #has over 400 objects!
[1] "%+%"        "%+replace%" "aes“
```

In principle, whatever you want to do, there is probably a library that does it.

Actually, there are usually many, and the hard part is trying to figure out which of them is best!

As of today (February 4, 2017) CRAN, the official online repository for R packages, has 10033 (!!!!) of them.

New versions of R come out every few month. Often changes are slight. I update my R version maybe once a year.

If you update R you also MUST update the libraries. It is easy to do:

```
> update.packages()
```

I run R on a number of machines. Trying to make sure all needed packages are installed and loaded on all machines is a bit painful. So I wrote a routine that makes it easy:

```
checkPackages <-
function (pack)
{
    if( !("utils" %in% search()) )
        library(utils)
    for(i in 1:length(pack)) {
#package already installed?
        if( !(pack[i] %in% rownames(installed.packages())))
            install.packages(pack[i],repos=c(CRAN =
                "http://cran.rstudio.com/"))
#package already loaded?
        if( !(pack[i] %in% search()) )
            library(pack[i],character.only=TRUE)
    }
    NULL
}
```

I have about a dozen of these routines to do house keeping chores.

Data Structures: Vectors

> Ages <- c(29,23,19,24,29,31,24)

<- is the assignment character (= works also, but is discouraged for good but somewhat esoteric reasons)

c “concatenate” combines things into one vector

R is case-sensitive (ages is not Ages)

Elements of a vector can be numeric, character, logical or NA:
Vector can **not** be a mix of numeric and character

```
> c(1,4,5,7)
[1] 1 4 5 7

 > c("A","B","C","D")
[1] "A" "B" "C" "D“

> c("A",2,"C",4)
[1] "A" "2" "C" "4"          #automatic type conversion!

> c(1,2,NA,3,TRUE,F)
[1]  1  2 NA  3  1 0

> 1:10>5                      # Boolean Expressions
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> sum(1:10>5)
[1] 5
```

If possible you can force the type conversion

```
> x <- 1:10
> y <- as.character(x)
> y
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10 "

> as.numeric(y)
 [1]  1  2  3  4  5  6  7  8  9 10

> as.numeric(c("1","2","3", "a"))
[1]  1  2  3 NA
Warning message:
NAs introduced by coercion
```

If you need a vector with some structure there are two useful routines – rep and seq

```
> rep(1,10)
 [1] 1 1 1 1 1 1 1 1 1 1

> rep(1:2,5)
 [1] 1 2 1 2 1 2 1 2 1 2

> rep(1:2,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2

> rep(1:3,c(2,3,4))
[1] 1 1 2 2 2 3 3 3 3

> rep(1:3,rep(3,3))
[1] 1 1 1 2 2 2 3 3 3
```

```
> seq(0,1,0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
> seq(0,1,length=10)
 [1] 0.0000000 0.1111111 0.2222222 0.3333333
0.4444444 0.5555556 0.6666667 0.7777778
0.8888889 1.0000000
```

Some more useful routines:

```
> length(x)

> names(x)        # can also be used to assign
                  names
> head(x)         # first six elements, also tail

> ls()             # listing of all objects in file
```

Subsetting a Vector

```
> Age<-c(20,21,27,18,25,23,24)
> names(Age)<-LETTERS[1:7]

> Age[1]
 A
20
> Age[1:3]
 A  B  C
20 21 27
> Age[c(2,5)]
 B  E
21 25
> Age[-c(2,5)]
 A  C  D  F  G
20 27 18 23 24
> Age["A"]
 A
20
```

```
> Age[c("A","B")]
 A  B
20 21
> Age[Age>=21]
 B  C  E  F  G
21 27 25 23 24
> Job<-c(T,T,F,T,T,F,T)
> Age[Job]
 A  B  D  E  G
20 21 18 25 24
> Age[Job & Age>20]
 B  E  G
21 25 24
```

Data Types: Matrices

a rectangular array of numbers OR characters

```
> cbind(1:3,rep(0,3),c(2,4,6))
     [,1] [,2] [,3]
[1,]    1    0    2
[2,]    2    0    4
[3,]    3    0    6
```

```
> matrix(1:6,ncol=3,byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
> x<-matrix(1:6,ncol=3,byrow=T)
> colnames(x) <- c("A","B","C")
> rownames(x) <- c("X","Y")
> x
  A B C
X 1 2 3
Y 4 5 6
> dim(x)
[1] 2 3
> ncol(x)                    # also nrow
[1] 3
```

```
> x <- matrix(1:10,ncol=5)
> dimnames(x) <- list(LETTERS[1:2],letters[1:5])
> x
  a b c d  e
A 1 3 5 7  9
B 2 4 6 8 10

> x[1,1]
[1] 1
```

```
> x[1,2:3]
b c
3 5

> x[,2:3]
  b c
A 3 5
B 4 6

> x["A",2:3]
b c
3 5
```

```
> dim(x)
[1] 2 5

> ncol(x)
[1] 5

> colnames(x)
[1] "a" "b" "c" "d" "e“

> rownames(x)
[1] "A" "B"
```

```
> dimnames(x)
[[1]]
[1] "A" "B"

[[2]]
[1] "a" "b" "c" "d" "e“

> colnames(x) <- 1:5

> dimnames(x) <-
list(1:2,1:5)
```

Linear Algebra

```
> x<-matrix(1:4,2,2)
> solve(x)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

> y<-c(2,2)
> solve(x,y)
[1] -1  1
```

```
> eigen(x)
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

---

qr, singular value decomposition, sparse matrices…

Data Types: Dataframes

Most common type of data structure in R. It is like a matrix, but different columns can have different data types

Example: Lengths of newborn babies and drug habit of mothers

Cocaine abuse during pregnancy: correlation between prenatal care and perinatal outcome
Authors: SN MacGregor, LG Keith, JA Bachicha, and IJ Chasnoff
Obstetrics & Gynecology 1989;74:882-885

```
> head(mothers,3)
     Status Length
1 Drug Free   44.3
2 Drug Free   45.3
3 Drug Free   46.9
```

subsetting with [,] as well as commands dim, ncol, nrow, dimnames, colnames work the same as with matrices. Also:

```
> table(Status)
Error in table(Status) : object 'Status' not found

> attach(mothers)
> table(Status)
Status
     Drug Free First Trimester      Throughout
            39              19              36
```

```
> search()
 [1] ".GlobalEnv"        "mothers" ….

> detach(mothers)   #when you are done
```

Creating your own data frame is easy:

```
> x<-1:10
> y<-letters(1:10)
> z<-data.frame(x,y)
```

Yet more ways to do subsetting: $ and [[ ]]

```
> head(mothers$Length)
[1] 44.3 45.3 46.9 47.0 47.2 47.8

> head(mothers[[1]])
[1] "Drug Free" "Drug Free" "Drug Free" "Drug Free" "Drug Free" "Drug Free“
```

Data Types: Lists

Most general structure, simply a list of objects of any kind:

```
> x<-1:10
> y<-c("A","B")
> fun<-function(x) x^2
> all.of.it<-list(x=x,y=y,fun=fun)
```

```
> all.of.it
$x
 [1]  1  2  3  4  5  6  7  8  9 10
$y
[1] "A" "B"
$fun
function (x)
x^2
```

On some deep level every data object in R is a list:

```
> x <- 1:10
> x[[1]]
[1] 1
```

```
> all.of.it$fun(all.of.it[[1]])
 [1]   1   4   9  16  25  36  49  64
81 100
```

Many Other Data Types

Example: factor - character vector with explicitly assigned levels

```
> y<-rnorm(90)
> x<-rep(c("Low","Medium","High"),each=30)
> tapply(y,x,mean)
      High        Low     Medium
-0.06274664  0.23318756 -0.12049278

> x<-
factor(x,levels=c("Low","Medium","High"),ordered=T)
> tapply(y,x,mean)
       Low     Medium       High
 0.23318756 -0.12049278 -0.06274664
```

I/O - Data Entry and Export

There are many ways to get electronic data into R. For a simple vector we have

> scan(“folder/file.r”)

> scan(“folder/file.r”, what=“char”)

For spreadsheets and tables use

> read.table(“folder/file.r”)

There are also routines for I/O from many other programs, for example read.csv for entry from Excel spreadsheets. A number of packages are also available, for example *foreign*.

I/O: R to R

the routines *dump* and *source* allow easy transfer from one R project to another. The nice thing is that all formatting is done automatically:

```
> x<-1:10
> y<-list(a=rep(1,3),b=c("A","B"))
> fun<-function(x) x^2+log(x)
> dump(c("x","y","fun"),"c:/tmp/stuff.r")
```

stuff.r

```
x <-
1:10
y <-
structure(list(a = c(1, 1, 1), b = c("A", "B")), .Names =
c("a", "b"))
fun <-
function(x) x^2+log(x)
```

Now in the other RData just run

```
> source("c:/tmp/stuff.r")
```

Type Conversion

Already saw: numeric + character → character

```
> x<-matrix(1:100,ncol=5)     #data frame
> x[1:2, ] #data frame
> x[1, ]    #vector!
```

Basic principle: R automatically converts to most basic data type possible.

Greatly simplifies writing routines, but can get you in trouble!

You can keep it from happening, though:

```
> x[1, ,drop=FALSE]
```

Vector Arithmetic

(Almost) all R functions are written to work on vectors:

```
> x<-1:5;y<-6:10
> x^2
[1]  1  4  9 16 25

> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
1.6094379

> x+y
[1]  7  9 11 13 15
```

This even works with matrices:

```
> x<-matrix(1:10,ncol=5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> x^2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    9   25   49   81
[2,]    4   16   36   64  100
```

but not always!

There are some functions written specifically for vectorizing calculations:

```
> x<-matrix(1:100,ncol=5)
> apply(x,1,sum)
 [1] 205 210 215 220 225 230 235 240 245 250 255 260 265 270 275 280 285 290 295 300

> apply(x,2,mean)
[1] 10.5 30.5 50.5 70.5 90.5
```

This not only yields short code but also runs much faster!

Other data types have their own versions:

```
> x<-list(A=1:10,B=20:30)
> lapply(x,mean)
$A
[1] 5.5

$B
[1] 25
```

Notice an inconsistency here: no type conversion! But:

```
> sapply(x,mean)
   A    B
 5.5 25.0
```

R library: **plyr**

Character Manipulation

There are a number of routines to work on character vectors:

```
> x <- c("try this", "and try that", "but not the other")
> nchar(x)
[1]  8 12 17

> substring(x,1,3)
[1] "try" "and" "but“

> grep("try",x)
[1] 1 2

> sub("try","do",x)
[1] "do this"          "and do that"       "but not the other“
```

```
> paste(c("a","b","c"),"d")
[1] "a d" "b d" "c d "

> paste(c("a","b","c"),"d",sep="+")
[1] "a+d" "b+d" "c+d "

> paste(c("a","b","c"),"d",sep="")
[1] "ad" "bd" "cd "

> paste(c("a","b","c"),"d",collapse=" ")
[1] "a d b d c d "

> paste("a",1:5,sep="")
[1] "a1" "a2" "a3" "a4" "a5 "
```

R Library: **stringr**

Basic Statistics

As one would expect, any basic statistics calculation is easily done in R:

Example: UPR (University of Puerto Rico) student data

```
> attach(upr)
> nrow(upr)
[1] 23666

> colnames(upr)

 [1] "ID.Code"        "Year"           "Gender"         "Program.Code"
"Highschool.GPA“  "Aptitud.Verbal" "Aptitud.Matem"
"Aprov.Ingles"   "Aprov.Matem"    "Aprov.Espanol"  "IGS"
"Freshmen.GPA"   "Graduated"      "Year.Grad."     "Grad..GPA"
"Class.Facultad"
```

```
> mean(Freshmen.GPA,na.rm=T)
[1] 2.733214

> sd(Freshmen.GPA,na.rm=T)
[1] 0.7791875

> median(Freshmen.GPA,na.rm=T)
[1] 2.83

> quantile(Freshmen.GPA,c(0.2,0.4,0.6,0.8),na.rm=T)
 20%  40%  60%  80%
2.17 2.65 3.00 3.39
```

```
> tapply(Freshmen.GPA,Gender,mean,na.rm=T)
       F        M
2.829027 2.642715

> tapply(Freshmen.GPA,Gender,quantile,
        probs=c(0.25,0.75),na.rm=T)
$F
 25%  75%
2.44 3.34

$M
 25%  75%
2.20 3.22
```

```
> t.test(Freshmen.GPA[Gender=="F"],
         Freshmen.GPA[Gender=="M"])

        Welch Two Sample t-test

data:  Freshmen.GPA[Gender == "F"] and Freshmen.GPA[Gender == "M"]
t = 18.487, df = 23398, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.1665591 0.2060660
sample estimates:
mean of x mean of y
 2.829027  2.642715
```

```
> cor.test(Highschool.GPA, Freshmen.GPA)

        Pearson's product-moment correlation

data:  Highschool.GPA[ok] and Freshmen.GPA[ok]
t = 73.673, df = 23449, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to
0
95 percent confidence interval:
 0.4230958 0.4438828
sample estimates:
      cor
0.433547
```

```
> cor.test(Highschool.GPA,Freshmen.GPA,method="kendall")

        Kendall's rank correlation tau

data:  Highschool.GPA and Freshmen.GPA
z = 73.456, p-value < 2.2e-16
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
0.3263772

> cor.test(Highschool.GPA,Freshmen.GPA,method="spearman")

        Spearman's rank correlation rho

data:  Highschool.GPA and Freshmen.GPA
S = 1.1495e+12, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.4652149
```

Example: Machine Learning

Algorithm	Uses	Package
Decision Trees	Classification, Regression	rpart
Gradient Boosted Machine	Classification, Regression	gbm
Hierarchical Clustering	Clustering	pvclust
k-Means Clustering	Clustering	Base
k-Nearest Neighbor	Classification	class
Lasso	Classification, Regression	glmnet
Linear Regression	Classification, Regression	Base
Logistic Regression	Classification	Base
Model-Based Clustering	Clustering	mclust
Naïve Bayes	Classification	e1071 or klaR
Neural Networks	Classification, Regression	nnet
Random Forest	Classification, Regression	randomForest
Ridge Regression	Classification, Regression	MASS or ridge
Spectral Clustering	Clustering	kernlab
Support Vector Machine	Classification, Regression	e1071 or kernlab
Linear Discriminant Analysis	Classification	MASS

Model Notation and Regression/Fitting

Many routines in R use the ~ notation. This is based on the predictor response paradigm, with what is on the left of the ~ being the response.

Example: data set mtcars has info on car mileage, weight, number of gears, number of cylinders, etc. of 42 cars

```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Say we want to predict mpg by hp:

```
> print(lm(mpg ~ hp,data=mtcars))

Coefficients:
(Intercept)          hp
   30.09886     -0.06823
```

How about a no-intercept model?

```
> print(lm(mpg ~ hp-1,data=mtcars))

Coefficients:
    hp
0.1011
```

Include cyl:

```
> print(lm(mpg ~ hp+cyl,data=mtcars))

Coefficients:
(Intercept)          hp         cyl
   36.90833     -0.01912     -2.26469
```

---

and the product term:

```
> print(lm(mpg ~ hp*cyl,data=mtcars))

Coefficients:
(Intercept)          hp         cyl       hp:cyl
   50.75121     -0.17068     -4.11914      0.01974
```

Everything and the kitchen sink:

```
> print(lm(mpg ~ .^2,data=mtcars))

Coefficients:
(Intercept)          cyl         disp           hp         drat           wt         qsec
 -976.14836   107.63710    -0.71673     5.20820    93.64459    88.58589   -29.84482
         vs           am         gear         carb     cyl:disp      cyl:hp     cyl:drat
   13.95220  -519.42892   293.48749   -56.76076     0.61839    -0.53007    -4.57518
     cyl:wt     cyl:qsec      cyl:vs      cyl:am    cyl:gear    cyl:carb     disp:hp
  -21.69425     7.96779    70.54442   126.80477   -77.85856     3.04024    -0.01392
  disp:drat     disp:wt    disp:qsec     disp:vs     disp:am    disp:gear    disp:carb
    0.28869     0.03281    -0.28755     0.30314    -0.59288     0.62584     0.17424
     hp:drat      hp:wt      hp:qsec      hp:vs      hp:am     hp:gear      hp:carb
   -0.74773     0.39209     0.22681    -3.07399         NA          NA          NA
    drat:wt    drat:qsec     drat:vs     drat:am    drat:gear    drat:carb     wt:qsec
          NA          NA          NA          NA          NA          NA          NA
       wt:vs       wt:am      wt:gear      wt:carb      qsec:vs     qsec:am    qsec:gear
          NA          NA          NA          NA          NA          NA          NA
   qsec:carb       vs:am      vs:gear      vs:carb      am:gear     am:carb    gear:carb
          NA          NA          NA          NA          NA          NA          NA
```

Random Variates and Probability Functions

As you would expect R has a lot of functions built in for probability. They generally have the format

dname – probability density

pname – distribution function

rname – generate events

qname - quantiles

Example: Normal Distribution

```
> round(dnorm(seq(-3,3,0.5)),3)
 [1] 0.004 0.018 0.054 0.130 0.242 0.352 0.399 0.352
0.242 0.130 0.054 0.018 0.004

> round(pnorm(seq(0,30,5),mean=10,sd=5),3)
[1] 0.022 0.158 0.500 0.841 0.977 0.998 1.000

> round(rnorm(5,mean=10,sd=5),3)
[1] 10.400  3.343 11.449 10.076  4.531

> round(qnorm(seq(0.01,0.99,0.2)),3)
[1] -2.326 -0.806 -0.228  0.279  0.878
```

- For the beta distribution see dbeta.
- For the binomial (including Bernoulli) distribution see dbinom.
- For the Cauchy distribution see dcauchy. (Breit-Wigner)
- For the chi-squared distribution see dchisq.
- For the exponential distribution see dexp.
- For the F distribution see df.
- For the gamma distribution see dgamma.
- For the geometric distribution see dgeom. (This is also a special case of the negative binomial.)
- For the hypergeometric distribution see dhyper.
- For the log-normal distribution see dlnorm.
- For the multinomial distribution see dmultinom.
- For the negative binomial distribution see dnbinom.
- For the normal distribution see dnorm.
- For the Poisson distribution see dpois.
- For the Student's t distribution see dt.
- For the uniform distribution see dunif.
- For the Weibull distribution see dweibull.

Many others can be found in packages online, for example **mvtnorm**.

Sampling from a finite set:

```
> sample(1:100,5)
[1] 17 80 29  7 53

> sample(1:3,size=10,replace=T)
 [1] 2 1 2 2 2 3 3 3 3 2

> table(sample(1:3,size=1000,replace=T,prob=c(1,5,2)))
  1   2   3
146 622 232
```

set.seed

```
> set.seed(1000);rnorm(5)
[1] -0.44577826 -1.20585657  0.04112631  0.63938841
-0.78655436

> set.seed(1000);rnorm(5)
[1] -0.44577826 -1.20585657  0.04112631  0.63938841
-0.78655436

> rnorm(5)
[1] -0.38548930 -0.47586788  0.71975069 -0.01850562
-1.37311776
```

R as a Computer Language

R is a full-fledged computer language with all the usual parts:

```
> y<-rep(0,1000)
> for(i in 1:1000) {
+ x<-rnorm(1e4)+2*rnorm(1e4,0.5)
+ if(max(x)>10) y[i]<-1
+ }
> table(y)/1000
y
    0     1
0.764 0.236
```

Writing your own Routines

Just open an editor and write your function!

> fix(myfun)

You can use your favorite editor:

> options(editor="myfaveditor")

R will do a basic syntax check. If there is a problem you get an error message and can go back to fix it with edit()

Some people prefer to always write their routines outside of R, save them as ASCII files and then use source to import them into R.

Advantage: you always have an independent version of the source code.

Disadvantage: one extra step (switch to and from R)

Customizing R: .First and .Rprofile

Each of us likes to set things up in a certain way. There are two routines to help with this:

.First is a routine inside an RData. Whatever commands are part of it are executed at startup. I use this mostly to load packages that I need in this project but not necessarily in any others.

.Rprofile is an ASCII file that sits in a folder which is in the search path of R. It has commands that will get executed at startup whenever ANY R session starts.

I use this for a lot of things, for example to set the editor. I also have a number of routines that I wrote for repeating tasks. Rather than having to copy-paste them from an old project when I start a new one, they are always there!

At this writing my .Rprofile has 620 lines of code. You can have a look at it at

http://academic.uprm.edu/wrolke/research/.Rprofile

Graphics

Among the strongest features of R are its graphics capabilities.

Graphs play a huge role in any statistical analysis:

- Exploratory Data Analysis
- Diagnostics
- Presentation

There is a long history of graphs in Statistics:

- The Visual Display of Quantitative Information – Edward R. Tufte
- Visualizing Data – William S. Cleveland
- Exploratory Data Analysis - John W. Tukey
- The Grammar of Graphics – Leland Wilkinson

ggplot2

There are a number of packages available to make graphs in addition to the built-in ones, but the clear current leader is ggplot2.

Advantage: gorgeous graphs!

Disadvantage: very strange syntax, huge learning curve

Example: data set mtcars has info on car mileage, weight, number of gears, number of cylinders, etc. of 42 cars

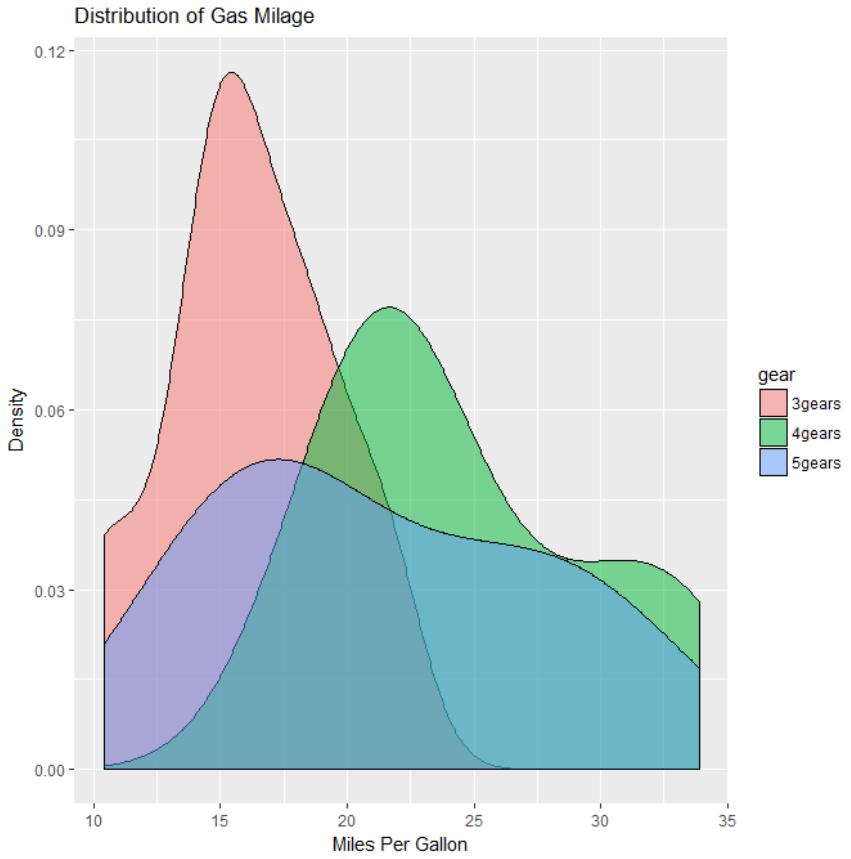
```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

First we make some changes to the data frame. This is just to get better labels:

```
> mtcars$gear <- factor(mtcars$gear,levels=c(3,4,5),
+    labels=c("3gears","4gears","5gears"))
> mtcars$am <- factor(mtcars$am,levels=c(0,1),
+    labels=c("Automatic","Manual"))
> mtcars$cyl <- factor(mtcars$cyl,levels=c(4,6,8),
+    labels=c("4cyl","6cyl","8cyl"))
```

Kernel density plots for mpg grouped by number of gears (indicated by color):

```
> qplot(mpg, data=mtcars, geom="density", fill=gear, alpha=I(.5),
   main="Distribution of Gas Milage", xlab="Miles Per Gallon",
   ylab="Density")
```



qplot – “quick plot” – basic routine to get ggplot2 started

mpg, data=mtcars – use variable mpg of data frame mtcars

geom="density" – geometry(shape) of graph (histogram, scatterplot etc)

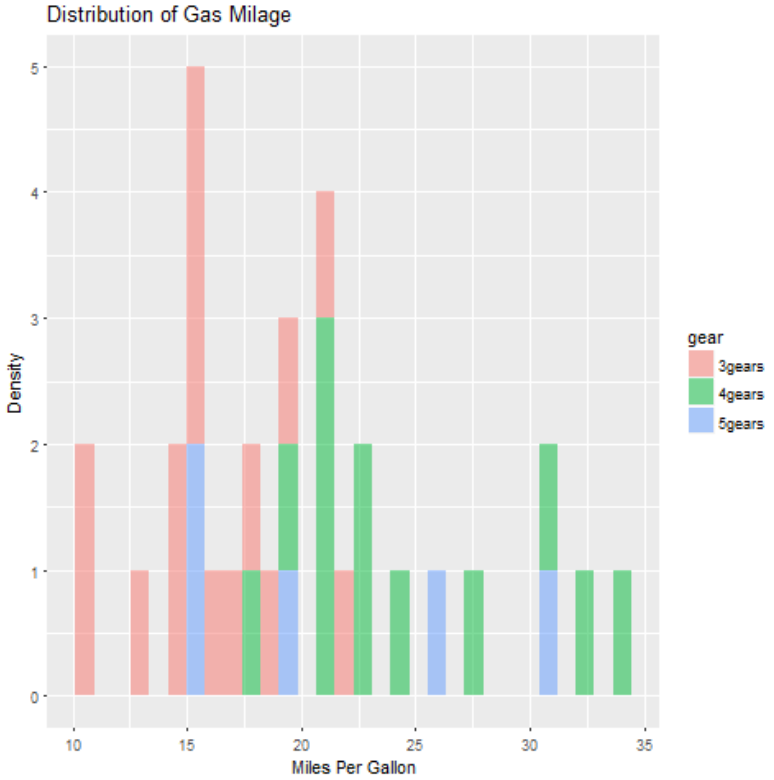
fill=gear – variable to use for grouping

alpha=I(.5) – control amount of shading

main="Distribution of Gas Milage“
xlab="Miles Per Gallon"
ylab="Density"

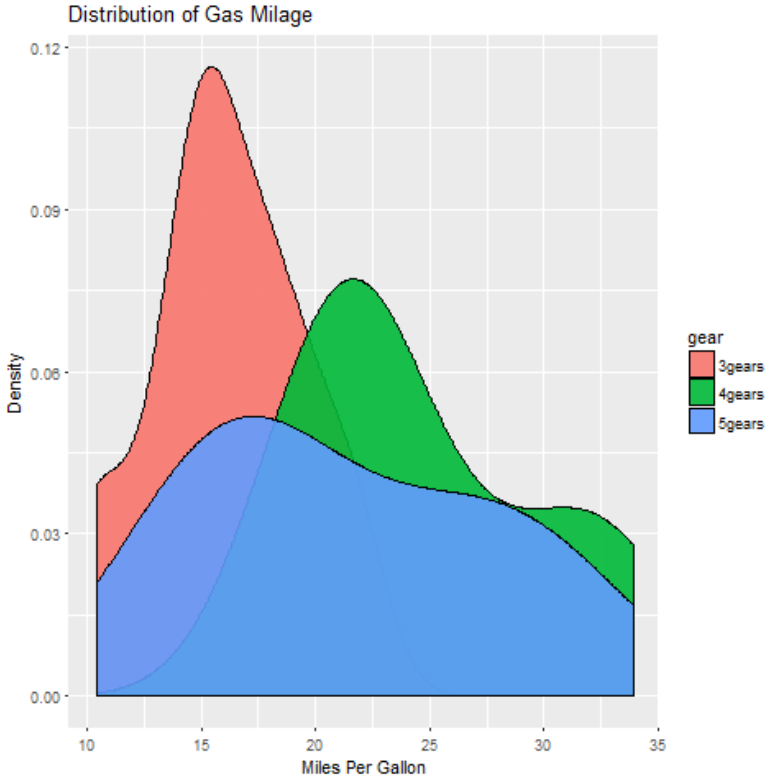
Histogram for mpg grouped by number of gears (indicated by color):

```
> qplot(mpg, data=mtcars, geom="histogram", fill=gear, alpha=I(.5),
   main="Distribution of Gas Milage", xlab="Miles Per Gallon",
   ylab="Density")
```



Deeper shading:

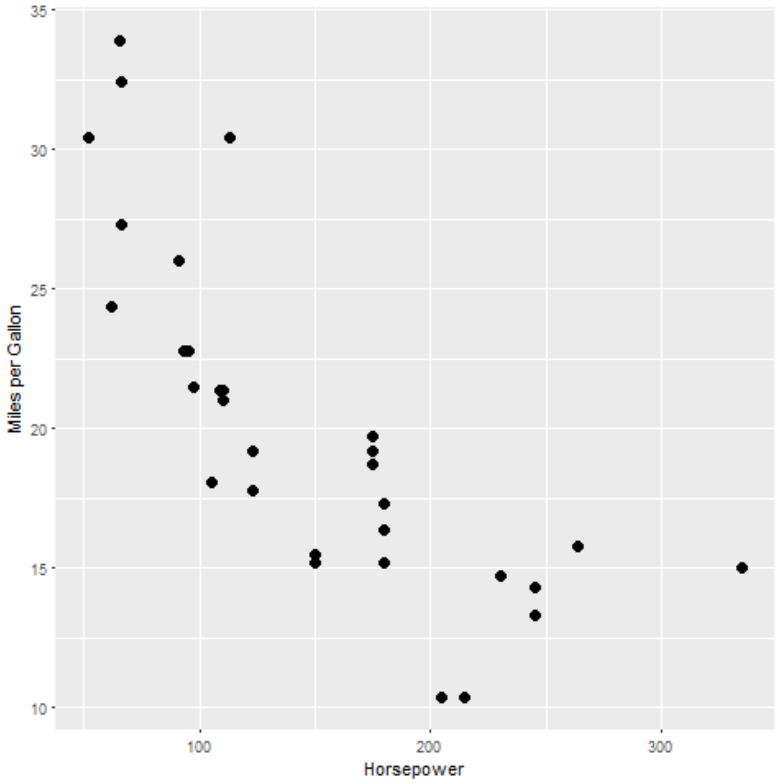
```
> qplot(mpg, data=mtcars,
geom="density", fill=gear,
alpha=I(.9),
   main="Distribution of Gas
Milage", xlab="Miles Per
Gallon",
   ylab="Density")
```



Scatterplot of mpg vs. hp

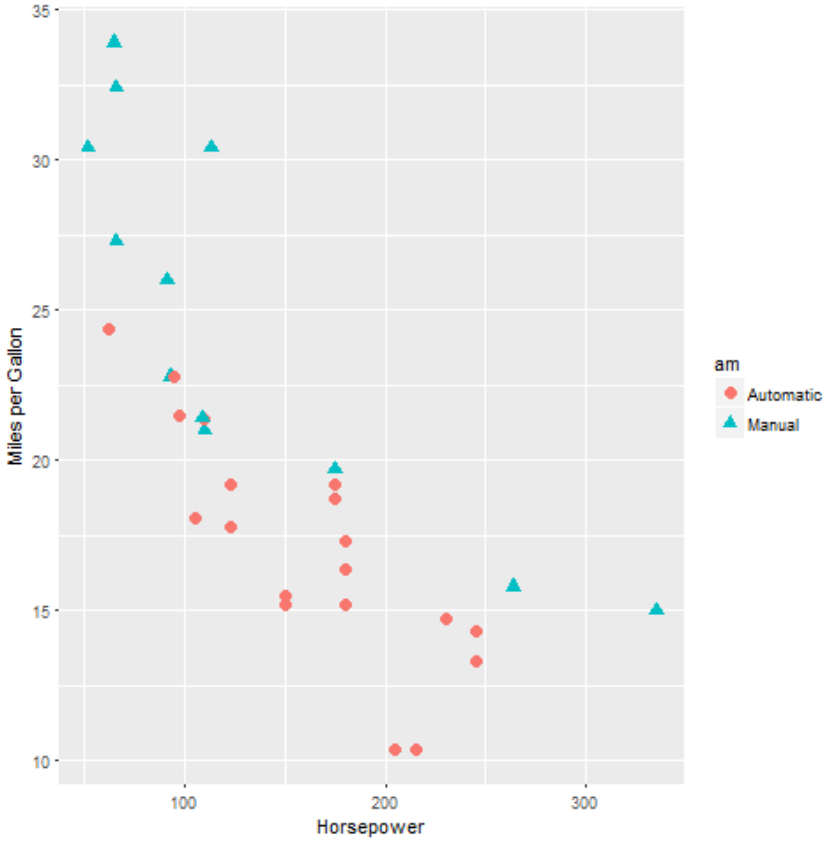
```
qplot(hp, mpg, data=mtcars,
xlab="Horsepower",
ylab="Miles per Gallon")
```

Notice: no geom, scatterplot is the default if two numeric vectors are give.



Scatterplot of mpg vs. hp by gears:

```
qplot(hp, mpg, data=mtcars,
shape=am, color=am,
size=I(3),
xlab="Horsepower",
ylab="Miles per Gallon")
```



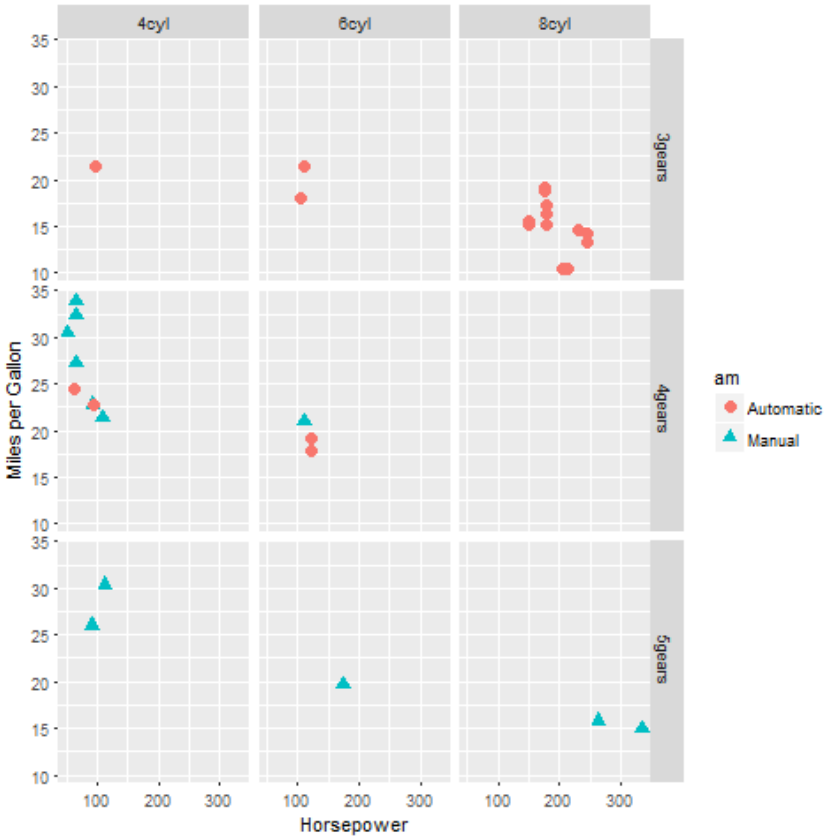
---

shape: different plotting symbols

color: different colors

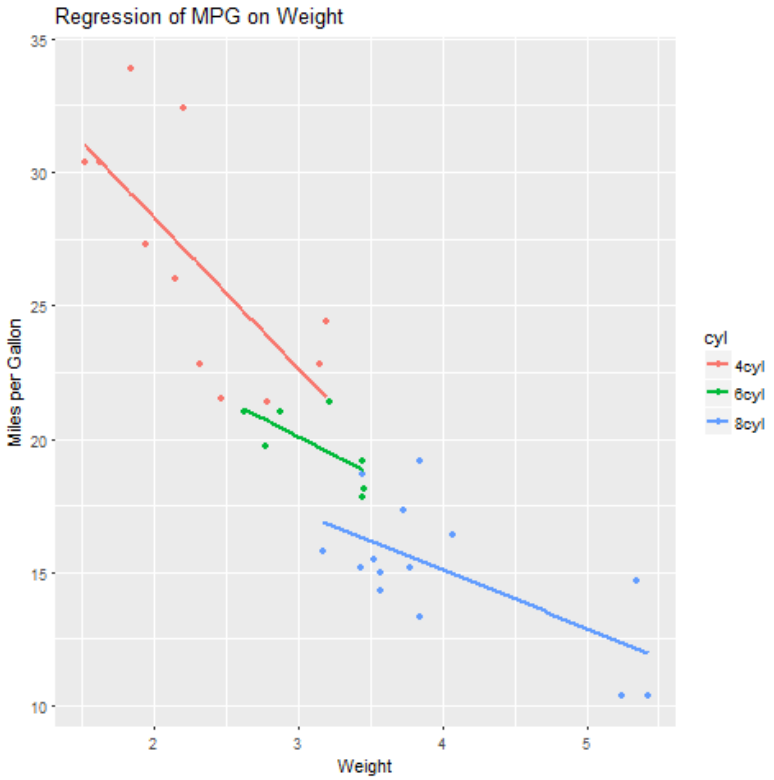
Scatterplot of mpg vs. hp for each combination of gears and cylinders. In each facet, transmission type is represented by shape and color

qplot(hp, mpg, data=mtcars, shape=am, color=am, **facets=gear~cyl**, size=I(3), xlab="Horsepower", ylab="Miles per Gallon")



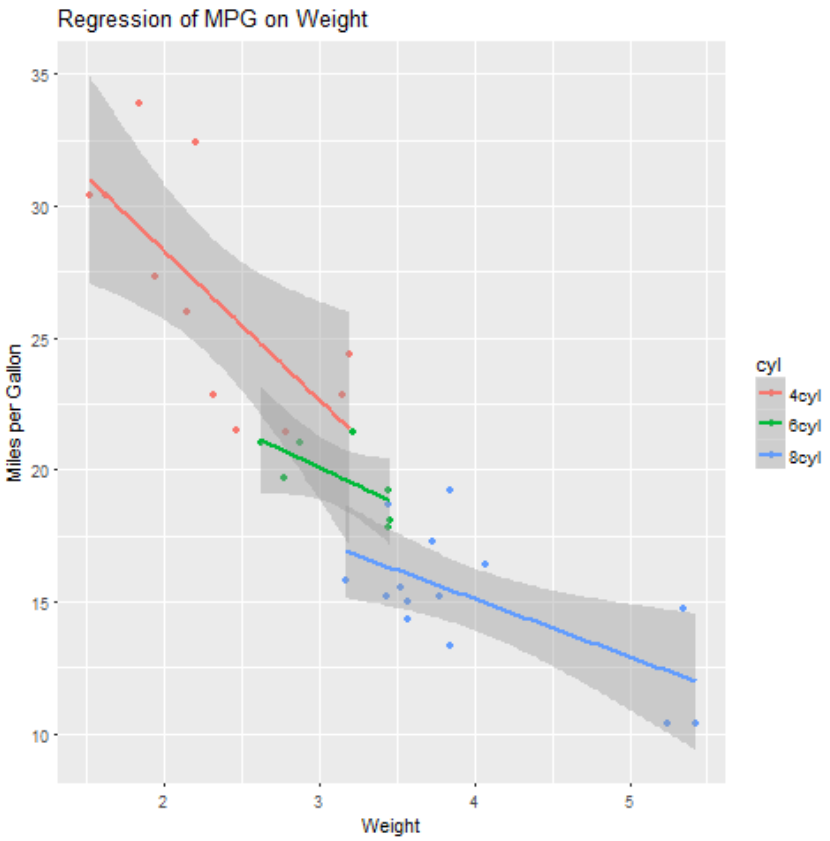
Separate regressions of mpg on weight for each number of cylinders:

```
qplot(wt, mpg, data=mtcars,
geom=c("point",
"smooth"),
method="lm", se=F,
formula=y~x, color=cyl,
main="Regression of MPG
on Weight",
   xlab="Weight", ylab="Miles
per Gallon")
```



With error bands (actually the default, suppress the bands with se=F)

```
qplot(wt, mpg, data=mtcars,
geom=c("point", "smooth"),
method="lm",
formula=y~x, color=cyl,
main="Regression of MPG
on Weight",
   xlab="Weight", ylab="Miles
per Gallon")
```

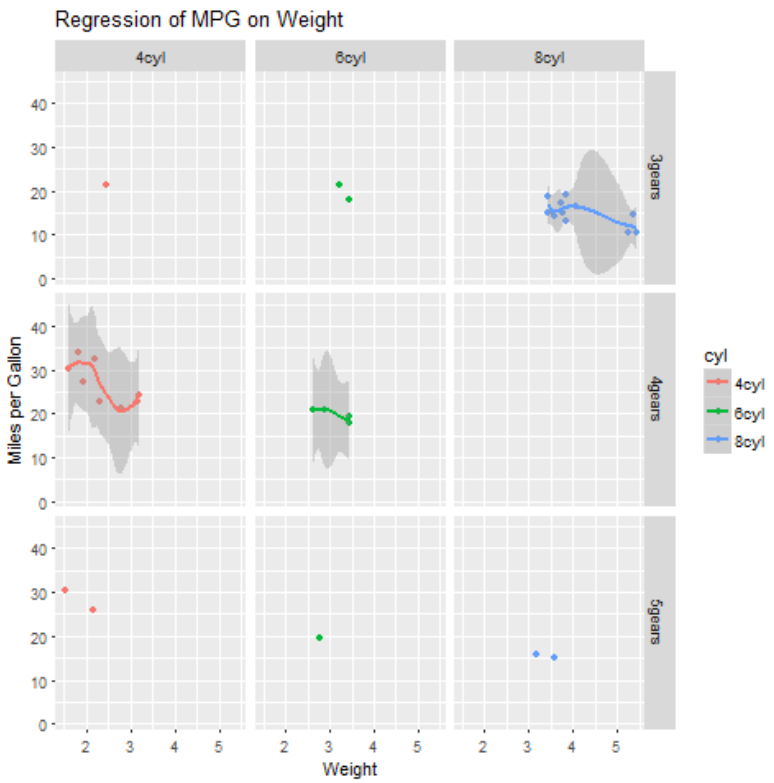


Non-parametric regression with LOESS

```
qplot(wt, mpg, data=mtcars,
geom=c("point", "smooth"),
method="loess",
formula=y~x, color=cyl,
main="Regression of MPG
on Weight",
   xlab="Weight", ylab="Miles
per Gallon")
```

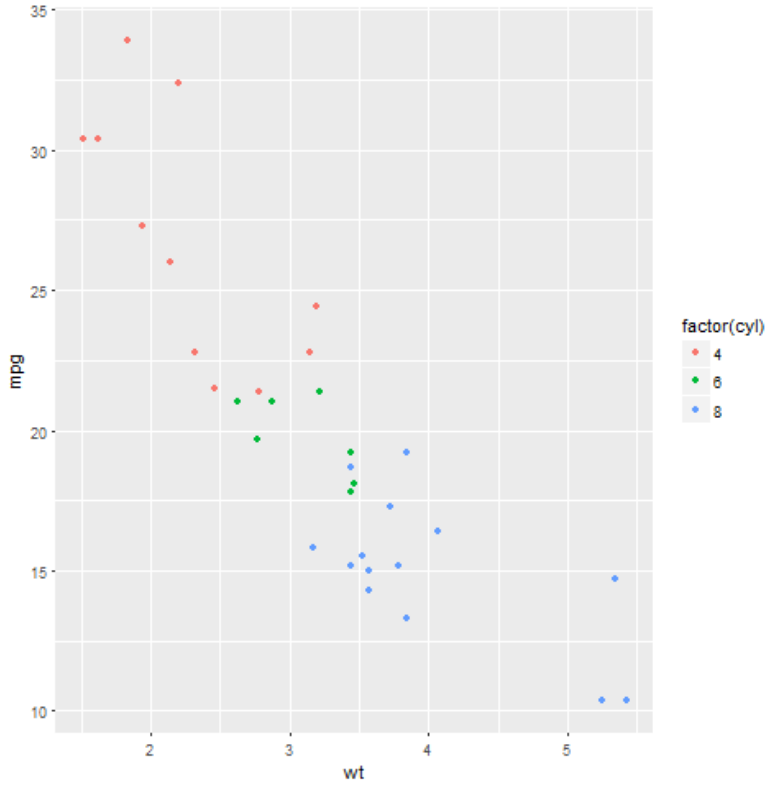
and by gear:

```
qplot(wt, mpg, data=mtcars,
geom=c("point", "smooth"),
method="loess",
formula=y~x, color=cyl,
facet=gear~cyl,
main="Regression of MPG
on Weight",
xlab="Weight",
ylab="Miles per Gallon")
```



Use ggplot for more Control

```
> plt <-
ggplot(data=mtcars,
aes(x = wt, y = mpg,
color=factor(cyl))) +
geom_point()
> print(plt)
```

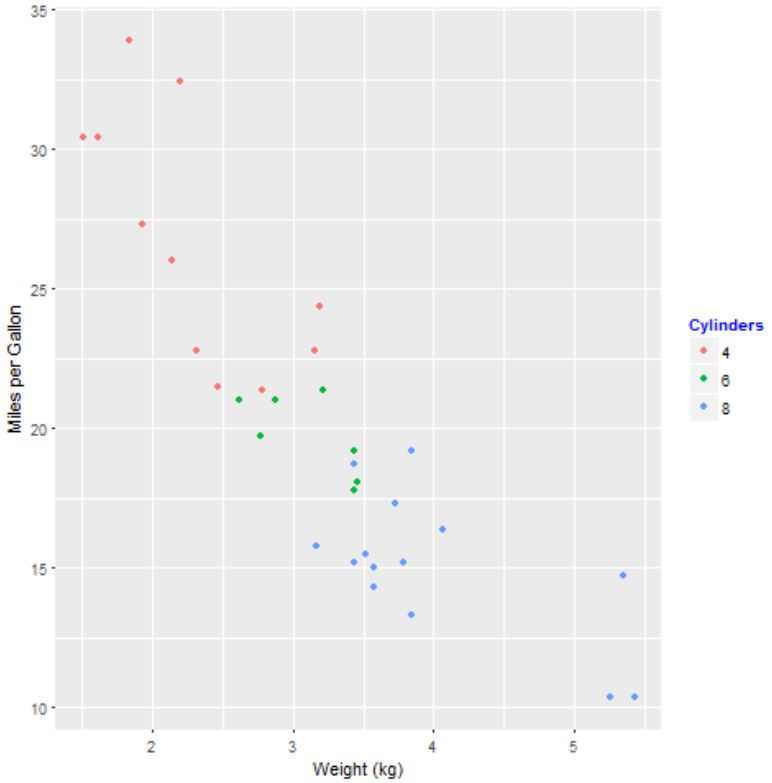


---

aes – (aesthetics) which variable goes where in the graph

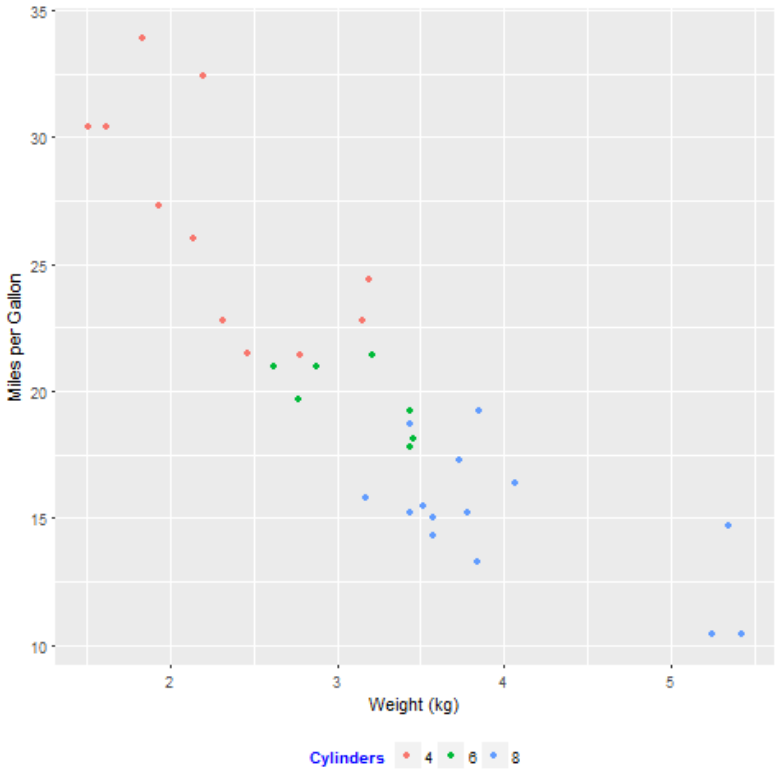
fix axis labels:

```
>  plt<-plt+
labs(x="Weight (kg)",
y="Miles per
Gallon",color="Cylinders")+
theme(legend.title =
element_text(colour="blue",
size=10, face="bold"))

> print(plt)
```



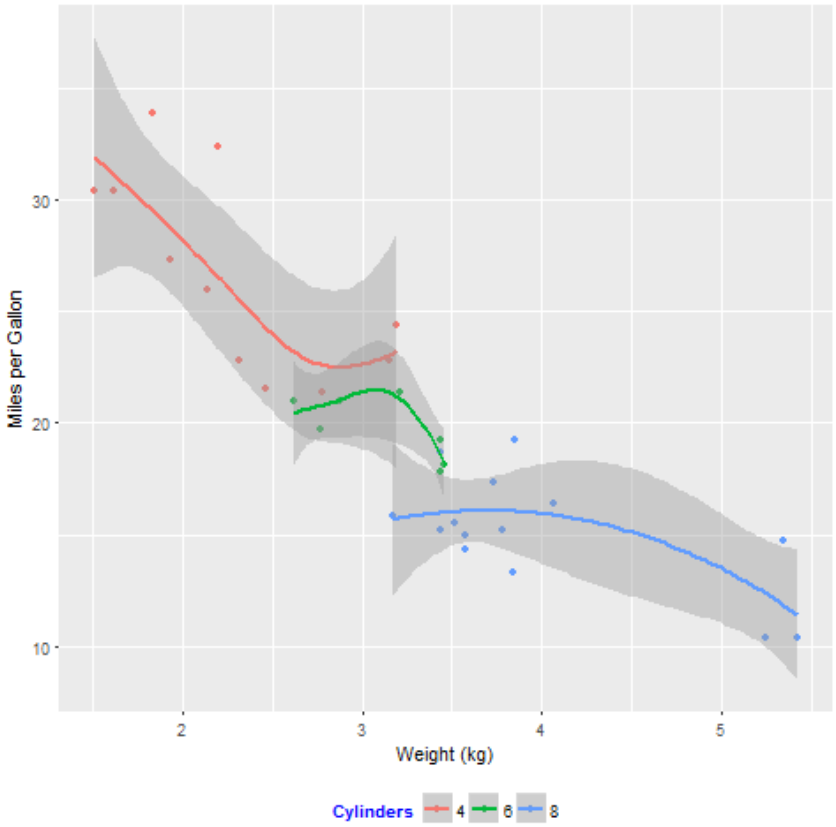
We can move the legend around:

```
> plt<-plt+
theme(legend.position="bottom")


> print(plt)
```



Let’s add a non-parametric regression fit with a larger smoothing parameter than the default:

```
> plt<-plt+
geom_smooth(span=1.5)

> print(plt)
```



Graphs are easily saved in a number of formats:

```
> png("c:/folder/mygraph.png")
>print(plt)
>dev.off()       #return to R console
```

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Got interested?

for an online tutorial of ggplot2 graphs go to

https://www.datacamp.com/courses/data-visualization-with-ggplot2-1

Rcpp – Use C++ code in R

R is great for writing programs – vectorization and other features make code very short

This has a price: R code can be

VEEERY SLOOOOW

One way to speed things up – write some parts in C++

Also useful if you already have a program written in C++ and want to execute it inside R

Example: say we want to do a coverage study of the classic confidence interval for the mean of a normal distribution with unknown standard deviation.

William S. Gosset (aka Student) 1908: A 100(1-α)% confidence interval for the population mean is given by

$$\overline{X} \pm t_{n-1,\alpha/2} \frac{s}{\sqrt{n}}$$

and we are concerned about the sample size n. So we write a simulation in R:

```
RcppExample1 <-
function (B=1000,npoints=50,alpha=0.95,mu=0,sigma=1)
{
    n<-1:npoints+4
    Coverage<-rep(0,npoints)
    for(j in 1:npoints) {
        crit <- qt(1-(1-alpha)/2, n[j]-1)
        counter=0;
        for(i in 1:B) {
            x<-rnorm(n[j],mu,sigma)
            if(mean(x)-crit*sd(x)/sqrt(n[j])>mu) counter=counter+1
            if(mean(x)+crit*sd(x)/sqrt(n[j])<mu) counter=counter+1
        }
        Coverage[j]<-100-counter/B*100
    }
    plot(n,Coverage,ylim=c(90,100),pch="0",col="red")
    abline(h=alpha*100,lwd=2)
}
```

Running this on my desktop takes just about 5 seconds.

Now let's redo the inner loop in Rcpp. First we write a C++ program as follows:

```
#include <Rcpp.h>
// [[Rcpp::export]]
double ci(int n, double mu, double sigma, int B,  double crit) {
    double Coverage;
    Rcpp::NumericVector x(n);
    int counter=0;
    for(int i=0;i<B;++i) {
        x=Rcpp::rnorm(n,mu,sigma);
        if(mean(x)-crit*sd(x)/sqrt(n)>mu) ++counter;
        if(mean(x)+crit*sd(x)/sqrt(n)<mu) ++counter;
    }
    Coverage=100-counter*(100.0/B);
    return Coverage;
}
```

Notice: almost exactly the same code as in R, including the use of vector arithmetic!

in R we first have to compile the code:
> sourceCpp("C:/Desy R/ci.cpp")

and write the rest of the R routine:

```
RcppExample2 <-
function (B=1000,npoints=50,alpha=0.95,mu=0,sigma=1)
{
    n<-1:npoints+4
    Coverage<-rep(0,npoints)
    for(j in 1:npoints) {
        crit <- qt(1-(1-alpha)/2, n[j]-1)
        Coverage[j]<-ci(n[j],mu,sigma,B,crit)
    }
    plot(n,Coverage,ylim=c(90,100),pch="0",col="red")
    abline(h=alpha*100,lwd=2)
}
```

Time to run this: 0.2 seconds, OR ALMOST 25 TIMES FASTER

ROOT-R Interface

- ROOT-R package to use R in the ROOT environment (in C++)
  - not to access ROOT tools inside R
- Simple way to call R functions from ROOT prompt or C++ code
- Conversion between ROOT/C++ objects and R objects
- Class TRDataFrame to support also r data-frame objects
  - Plug-ins can be developed to hide detail of ROOT-R interface
    - ROOT Minimizer plug-in using optimisation packages from R (RMinimizer)
    - Interface to use Machine Learning tools from R in the ROOT TMVA (RMVA library)
- See the ROOT-R Users Guide and the tutorials (in tutorials/r)

```
auto r = ROOT::R::TRInterface::Instance();
//executing simple r commands with the operator <<
r << "print('hello ROOTR')";
r << "vec=c(1,2,3)" << "print(vec)";
```

Thanks!