Lecture 6-10: Data Manipulation(Data analysis using R)

Outline

- Creating New Variable
- Operators
- Built-in functions
- Control Structures
- User Defined Functions
- Sorting Data
- Merging Data
- Aggregating Data
- Reshaping Data
- Sub-setting Data
- Data Type Conversions

Introduction

Once you have access to your data, you will want to massage it into useful form. This includes creating new variables (including recoding and renaming existing variables), sorting and merging datasets, aggregating data, reshaping data, and subsetting datasets (including selecting observations that meet criteria, randomly sampling observation, and dropping or keeping variables).

Introduction

Each of these activities usually involve the use of **R**'s built-in operators (arithmetic and logical) and functions (numeric, character, and statistical). Additionally, you may need to use control structures (if-then, for, while, switch) in your programs and/or create your own functions. Finally you may need to convert variables or datasets from one type to another (e.g. numeric to character or matrix to dataframe).

Creating new variables

- Use the assignment operator **<-** to create new variables. A wide array of operators and functions are available here.
-
  ```
  # Three examples for doing the same computations

  mydata$sum <- mydata$x1 + mydata$x2
  mydata$mean <- (mydata$x1 + mydata$x2)/2

  attach(mydata)
  mydata$sum <- x1 + x2
  mydata$mean <- (x1 + x2)/2
  detach(mydata)
  ```
-
  ```
  mydata <- transform( mydata,
  sum = x1 + x2,
  mean = (x1 + x2)/2
  )
  ```

Creating new variables

Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.

-
  ```
  # create 2 age categories
  mydata$agecat <- ifelse(mydata$age > 70, c("older"), c("younger"))
  # another example: create 3 age categories
  attach(mydata)
  mydata$agecat[age > 75] <- "Elder"
  mydata$agecat[age > 45 & age <= 75] <- "Middle Aged"
  mydata$agecat[age <= 45] <- "Young"
  detach(mydata)
  ```

Creating new variables

Recoding variables

- In order to recode data, you will probably use one or more of R's control structures.

-
  ```
  # create 2 age categories
  mydata$agecat <- ifelse(mydata$age > 70,
  c("older"), c("younger"))

  # another example: create 3 age categories
  attach(mydata)
  mydata$agecat[age > 75] <- "Elder"
  mydata$agecat[age > 45 & age <= 75] <- "Middle Aged"
  mydata$agecat[age <= 45] <- "Young"
  detach(mydata)
  ```

Creating new variables

Renaming variables

- You can rename variables programmatically or interactively.

-
  ```
  # rename interactively
  fix(mydata) # results are saved on close

  # rename programmatically
  library(reshape)
  mydata <- rename(mydata, c(oldname="newname"))

  # you can re-enter all the variable names in order
  # changing the ones you need to change.the limitation
  # is that you need to enter all of them!
  names(mydata) <- c("x1","age","y", "ses")
  ```

Arithmetic Operators

Operator	**Description**
+	addition
-	subtraction
*****	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

Logical Operators

Operator	**Description**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if x is TRUE

Control Structures

- **R** has the standard control structures you would expect. **expr** can be multiple (compound) statements by enclosing them in braces { }. It is more efficient to use built-in functions rather than control structures whenever possible.

Control Structures

- **if-else**
- if (*cond*) *expr*
  if (*cond*) *expr1* else *expr2*
- **for**
- for (*var* in *seq*) *expr*
- **while**
- while (*cond*) *expr*
- **switch**
- switch(*expr*, ...)
- **ifelse**
- ifelse(*test*,*yes*,*no*)

Control Structures

```
# transpose of a matrix
# a poor alternative to built-in t() function

mytrans <- function(x) {
  if (!is.matrix(x)) {
    warning("argument is not a matrix: returning NA")
    return(NA_real_)
  }
  y <- matrix(1, nrow=ncol(x), ncol=nrow(x))
  for (i in 1:nrow(x)) {
    for (j in 1:ncol(x)) {
      y[j,i] <- x[i,j]
    }
  }
return(y)
}
```

Control Structures

- # try it
  z <- matrix(1:10, nrow=5, ncol=2)
  tz <- mytrans(z)

R built-in functions

Almost everything in **R** is done through functions. Here I'm only referring to numeric and character functions that are commonly used in creating or recoding variables.

Note that while the examples on this page apply functions to individual variables, many can be applied to vectors and matrices as well.

Numeric Functions

Function	**Description**
abs(*x*)	absolute value
sqrt(*x*)	square root
ceiling(*x*)	ceiling(3.475) is 4
floor(*x*)	floor(3.475) is 3
trunc(*x*)	trunc(5.99) is 5
round(*x*, digits=*n*)	round(3.475, digits=2) is 3.48
signif(*x*, digits=*n*)	signif(3.475, digits=2) is 3.5
cos(*x*), sin(*x*), tan(*x*)	also acos(*x*), cosh(*x*), acosh(*x*), etc.
log(*x*)	natural logarithm
log10(*x*)	common logarithm
exp(*x*)	e^*x*

Character Functions

Function	**Description**
substr(*x***, start=***n1***, stop=***n2***)**	Extract or replace substrings in a character vector. x <- "abcdef" substr(x, 2, 4) is "bcd" substr(x, 2, 4) <- "22222" is "a222ef"
grep(*pattern***,** *x* **, ignore.case=**FALSE**, fixed=**FALSE**)**	Search for *pattern* in *x*. If fixed =FALSE then *pattern* is a regular expression. If fixed=TRUE then *pattern* is a text string. Returns matching indices. grep("A", c("b","A","c"), fixed=TRUE) returns 2
sub(*pattern***,** *replacement***,** *x***, ignore.case =**FALSE**, fixed=**FALSE**)**	Find *pattern* in *x* and replace with *replacement* text. If fixed=FALSE then *pattern* is a regular expression._x005F_x000b_If fixed = T then *pattern* is a text string. sub("\\s",".","Hello There") returns "Hello.There"
strsplit(*x***,** *split***)**	Split the elements of character vector *x* at *split*. strsplit("abc", "") returns 3 element vector "a","b","c"

paste(..., sep="")	Concatenate strings after using *sep* string to seperate them.
	paste("x",1:3,sep="") returns c("x1","x2" "x3")
	paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3")
	paste("Today is", date())

toupper(*x***)**	Uppercase
tolower(*x***)**	Lowercase

Stat/Prob Functions

- The following table describes functions related to probaility distributions. For random number generators below, you can use set.seed(1234) or some other integer to create reproducible pseudo-random numbers.

Function	**Description**

dnorm(*x*)	normal density function (by default m=0 sd=1)
	# plot standard normal curve
	x <- pretty(c(-3,3), 30)
	y <- dnorm(x)
	plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i")

pnorm(*q*)	cumulative normal probability for q (area under the normal curve to the right of q) pnorm(1.96) is 0.975
qnorm(*p*)	normal quantile. value at the p percentile of normal distribution qnorm(.9) is 1.28 # 90th percentile
rnorm(*n*, m=0**,sd=**1**)**	n random normal deviates with mean m and standard deviation sd. #50 random normal variates with mean=50, sd=10 x <- rnorm(50, m=50, sd=10)

dbinom(*x*, *size*, *prob*)
pbinom(*q*, *size*, *prob*)
qbinom(*p*, *size*, *prob*)
rbinom(*n*, *size*, *prob*)
	binomial distribution where size is the sample size
	and prob is the probability of a heads (pi)
	# prob of 0 to 5 heads of fair coin out of 10 flips
	dbinom(0:5, 10, .5)
	# prob of 5 or less heads of fair coin out of 10 flips
	pbinom(5, 10, .5)

dpois(*x*, *lamda*)
ppois(*q*, *lamda*)
qpois(*p*, *lamda*)
rpois(*n*, *lamda*)
	poisson distribution with m=std=lamda
	#probability of 0,1, or 2 events with lamda=4
	dpois(0:2, 4)
	# probability of at least 3 events with lamda=4
	1- ppois(2,4)

dunif(*x*, min=0**, max=**1**)**
punif(*q*, min=0, max=1**)**
qunif(*p*, min=0**, max=**1**)**
runif(*n*, min=0**, max=**1**)**
	uniform distribution, follows the same pattern
	as the normal distribution above.
	#10 uniform random variates
	x <- runif(10)

Function	**Description**
mean(*x***, trim=**0**, na.rm=**FALSE**)**	mean of object x # trimmed mean, removing any missing values and # 5 percent of highest and lowest scores mx <- mean(x,trim=.05,na.rm=TRUE)
sd(*x***)**	standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation.
median(*x***)**	median
quantile(*x***,** *probs***)**	quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1]. # 30th and 84th percentiles of x y <- quantile(x, c(.3,.84))
range(*x***)**	range
sum(*x***)**	sum
diff(*x***, lag=***1***)**	lagged differences, with lag indicating which lag to use
min(*x***)**	minimum
max(*x***)**	maximum
scale(*x***, center=**TRUE**, scale=**TRUE**)**	column center or standardize a matrix.

Other Useful Functions

Function	**Description**
seq(*from* **,** *to***,** *by***)**	generate a sequence indices <- seq(1,10,2) #indices is c(1, 3, 5, 7, 9)

rep(*x***,** *ntimes***)**	repeat *x n* times
	y <- rep(1:3, 2)
	# y is c(1, 2, 3, 1, 2, 3)

cut(*x***,** *n***)**	divide continuous variable in factor with *n* levels y <- cut(x, 5)

Sorting

- To sort a dataframe in R, use the **order( )** function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.
-
  ```
  # sorting examples using the mtcars dataset
  data(mtcars)
  # sort by mpg
  newdata = mtcars[order(mtcars$mpg),]
  # sort by mpg and cyl
  newdata <- mtcars[order(mtcars$mpg, mtcars$cyl),]
  #sort by mpg (ascending) and cyl (descending)
  newdata <- mtcars[order(mtcars$mpg, -mtcars$cyl),]
  ```

Merging

To merge two dataframes (datasets) horizontally, use the **merge** function. In most cases, you join two dataframes by one or more common key variables (i.e., an inner join).

```
# merge two dataframes by ID
total <- merge(dataframeA,dataframeB,by="ID")

# merge two dataframes by ID and Country
total <- merge(dataframeA,dataframeB,by=c("ID","Country"))
```

Merging

ADDING ROWS

To join two dataframes (datasets) vertically, use the **rbind** function. The two dataframes **must** have the same variables, but they do not have to be in the same order.

```
total <- rbind(dataframeA, dataframeB)
```

If dataframeA has variables that dataframeB does not, then either:

Delete the extra variables in dataframeA or

Create the additional variables in dataframeB and set them to NA (missing)

before joining them with rbind.

Aggregating

- **It is relatively easy to collapse data in R using one or more BY variables and a defined function.**

-
  ```
  # aggregate dataframe mtcars by cyl and vs,
  returning means
  # for numeric variables
  attach(mtcars)
  aggdata <-aggregate(mtcars, by=list(cyl),
    FUN=mean, na.rm=TRUE)
  print(aggdata)
  ```
- OR use apply

Aggregating

- **When using the aggregate() function, the by variables must be in a list (even if there is only one). The function can be built-in or user provided.**
- **See also:**
- **summarize() in the** Hmisc **package**
- **summaryBy() in the** doBy **package**

Data Type Conversion

- **Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to character.**
- **Use is.*foo* to test for data type *foo*. Returns TRUE or FALSE**
  Use as.*foo* to explicitly convert it.
- **is.numeric(), is.character(), is.vector(), is.matrix(), is.data.frame()**
  as.numeric(), as.character(), as.vector(), as.matrix(), as.data.frame)