Basic Statistical Methods

Lecture 9

Nicholas Christian

BIOST 2094 Spring 2011

Outline

Summary Statistics - Vectors

- Functions for calculating summary statistics of vector elements

`mean(x)`	Mean of `x`
`median(x)`	Median of `x`
`var(x)`	Variance of `x`
`sd(x)`	Standard deviation of `x`
`cov(x,y)`	Covariance of `x` and `y`
`cor(x,y)`	Correlation of `x` and `y`
`min(x)`	Minimum of `x`
`max(x)`	Maximum of `x`
`range(x)`	Range of `x`
`quantile(x)`	Quantiles of `x` for the given probabilities

```
> x <- rnorm(10)
> y <- runif(10)
> mean(x)
[1] 0.2210303
> cor(x,y)
[1] 0.3335438
```

Summary Statistics - Dataframes

- Functions for calculating summary statistics of the columns of a dataframe

`summary()`	Summary statistics of each column; type of statistics depends on data type
`apply()`	Apply a function to each column, works best if all columns are the same data type
`tapply()`	Divide the data into subsets and apply a function to each subset, returns an array
`by()`	Similar to `tapply()`, return an object of class by
`ave()`	Similar to `tapply()`, returns a vector the same length as the argument vector
`aggregate()`	Similar to `tapply()`, returns a dataframe
`sweep()`	"Sweep out" a summary statistic from a dataframe, matrix or array

- The major difference between `tapply()`, `ave()`, `by()`, and `aggregate()` is the format of the returned object

tapply()

- tapply() applies a function to each subset of a vector, where the subsets are determined by the unique combination of levels of factors

```
tapply(X, INDEX, FUN, ..., simplify=TRUE)
```

X	Vector applying FUN to
INDEX	List of factors for determining subsets. Each factor is the same length as X
FUN	The function to apply to each subset
...	Additional arguments to FUN
simplify	If FALSE returns a list, otherwise returns an array

`sweep()`

- The function `sweep()`, "sweeps out" or subtracts a summary statistic from the specified margins of an array or dataframe.

```
sweep(x, MARGIN, STATS, FUN="-", ...)
```

`x`	Array or dataframe
`MARGIN`	Margin to carry out the sweep on
`STATS`	Array of summary statistics to sweep out
`FUN`	Function used to carry out the sweep (default is subtraction). Should be a function of two arguments: `x` and an array with the same dimensions as `x` generated from `STATS`
`...`	Additional arguments to `FUN`

- The dimension of `STATS` depends on the value of `MARGIN`. So if `MARGIN=2` (columns) and `x` has three columns then `STATS` is a vector of length 3
- `sweep()` appropriately expands `STATS` to match the dimension of `x` and then applies `FUN`.

Example - Dataframe Summary Statistics

```
# Create a dataset of state data from the built-in matrix state.x77
# with two continuous variables (population and income) and
# two factor variables (region and area)
area.factor <- cut(state.x77[,"Area"],
                 breaks=quantile(state.x77[,"Area"], c(0, .25, .75, 1)),
                 labels=c("small", "medium", "large"),
                 include.lowest=TRUE)

state <- data.frame(pop=state.x77[,"Population"],
                    inc=state.x77[,"Income"],
                    area=area.factor,
                    region=state.region)

head(state)
dim(state)
```

Example - Dataframe Summary Statistics

```
# Basic summary statistics for each column
summary(state)

# Variance of first two columns
apply(state[,1:2], 2, var)

# Mean income by region
tapply(state$inc, state$region, FUN=mean)                   # Return a vector
by(state$inc, state$region, FUN=mean)                       # Return a by object,
                                                            # extract elements using [ ]
ave(state$inc, state$region, FUN=mean)                      # Return a vector, same
                                                            # length as first argument
aggregate(state$inc, list(state$region), FUN=mean) # Return a dataframe, subset
                                                            # variable needs to be a list

# Mean population by region and area
with(state, tapply(inc, list(region, area), FUN=mean))
with(state, by(inc, list(region, area), FUN=mean))
with(state, ave(inc, list(region, area), FUN=mean))
with(state, aggregate(inc, list(region, area), FUN=mean))
```

Example - Dataframe Summary Statistics

```
# Sweep out the columns means for population and income
# i.e., substract the overall mean population and overall mean income
# from the population and income for each state, respectively.

# First calculate the statistic we want to sweep out,
# columns means of population and income
col.means <- apply(state[1:5,1:2], 2, mean)
sweep(state[1:5,1:2], 2, col.means, FUN="-")

matrix(rep(col.means, each=5), nrow=5) - state[1:5, 1:2]        # Without sweep()

sweep(state[1:5, 1:2], 2, col.means, FUN=function(x, s)(x-s)/s) # Custom function

# Recall problem 1 part (e) from Homework 1, find P(X | Y, Z)
distr <- array(rep(c(.04,.01,.24,.06,.12,.03), 2), dim=c(2,3,2),
           dimnames=list(Y=c(1,-1), X=c(10,20,30), Z=c(0,1)))
marg.YZ <- apply(distr, c(1,3), sum)

# Without using sweep()
expand.marg.YZ <- array(rep(c(marg.YZ), 3), dim=c(2,3,2),
                    dimnames=list(Y=c(1,-1), X=c(10,20,30), Z=c(0,1)))
distr/expand.marg.YZ

# sweep() appropriately expands marg.YZ to match the dimension
# of distr then applies the division function
sweep(distr, c(1,3), marg.YZ, FUN="/")
```

Apply family and related functions

`apply()`	Apply a function to the margins of an array
`lapply()`	Apply a function to each element of a list or vector, returns a list
`sapply()`	Same as `lapply()`, but returns a vector or matrix by default
`vapply()`	Similar to `sapply()`, but has a pre-specified type of return value
`mapply()`	Multivariate version of sapply(), apply a function to the corresponding elements from multiple lists or vectors
`tapply()`	Apply a function to each subset of a vector, where the subsets are determined by the unique combination of factor levels
`outer()`	General outer product of two vectors
`sweep()`	Sweep out a summary statistic from the margins of an array
`replicate()`	Repeated evaluation of an expression

- These functions allow code to be more compact and clearer

Contingency Tables

- The following functions are used for creating data tables

`table()`	Create a contingency table of the counts at each combination of factor levels
`ftable()`	Similar to `table()`, useful for multidimensional tables
`xtabs()`	Create a contingency table using a formula interface, useful if the data have already been tabulated also includes an argument for specifying a dataframe
`prop.table()`	Table of proportions, relative to the given margin
`addmargins()`	Add margins to a table (default is to sum over all margins in the table)

- Although it appears that these functions return matrices or arrays, these functions actually return objects of class `"table"` for `table()`, `"ftable"` for `ftable()`, and `"xtabs"` for `xtabs()`. Both `prop.table()` and `addmargins()` return an object of class `"table"`.

Example - Contingency Tables

```
# Create a table of population and area
# Note: prop.table() and addmargins() require a table object
# argument and do not accept factor arguments
pop.area <- with(state.tab, table(pop, area))
addmargins(pop.area)
prop.table(pop.area)            # Overall frequency
prop.table(pop.area, margin=2) # Column frequency

# Create a multidimension table of population, area, and region
with(state.tab, ftable(pop, area, region))

# To include margins/proportions it is best to add the margins/proportions
# to a table object and then pass the new table object to ftable
pop.area.region <- with(state.tab, table(pop, area, region))
ftable(addmargins(pop.area.region))
ftable(prop.table(pop.area.region))
ftable(prop.table(pop.area.region, margin=3))

# Change row and column variables
with(state.tab, ftable(pop, area, region, row.vars="region"))
with(state.tab, ftable(pop, area, region, row.vars=c("region", "pop")))
with(state.tab, ftable(pop, area, region, row.vars="pop", col.vars="area")
```

Example - Contingency Tables

```
# Create a dataset where state.tab is already tabulated
# The outputted inc column is actually the frequency of each
# combination since the function we used was length
state.tab.freq <- aggregate(inc~pop+area+region,data=state.tab,FUN=length)
colnames(state.tab.freq)[4] = "freq"

# We also could have used tapply(), but in this case aggregate()
# returns the most useful object
with(state.tab, tapply(inc, list(pop, area, region), FUN=length))

# Since the data is already tabulated xtabs() is the most appropriate
# function. xtabs() is the only table function that has a dataframe
# argument
xtabs(freq~pop+area, data=state.tab.freq)

# If the data was not tabulated omit the left hand side
xtabs(~pop+area, data=state.tab)
```

Expand a Dataframe

- In some cases we may want to expand a dataframe of explanatory variables that has a frequency or count variable into a dataframe with as many repeated rows as specified by a frequency variable. That is, create a dataframe with a separate row for each case.

```
# Create a new dataframe with a separate row for each case
# For example, we want 4 copies of the first row (small, small, Northeast)
state.tab.freq

# Remember a dataframe is like a list where each component of the list
# has the same number of elements
temp <- lapply(state.tab.freq, function(x) rep(x, state.tab.freq$freq))

# Convert to a dataframe and remove the fourth column, freq
as.data.frame(temp)[-4]
```

Collapse a Dataframe

- On the other hand we may want to collapse a dataframe with repeated rows and create a new dataframe with one row for each unique combination of the explanatory variables and a frequency variable giving the number of occurrences.
- There are a couple of ways to do this, use `table()` to include all unique combinations of the variables including those with zero counts and use `aggregate()` to only include the unique variable combinations with non-zero counts.

```
# Collapse state.tab (ignoring inc)
head(state.tab)

# Include all unique combinations, including those with zero counts
with(state.tab, as.data.frame(table(pop, area, region)))

# Only include unique combinations with non-zero counts
state.tab.freq <- aggregate(inc~pop+area+region,data=state.tab,FUN=length)
colnames(state.tab.freq)[4] = "freq"
```

Comparing Means

- One sample tests, two sample tests and paired tests of the population mean(s) can be performed using either the *t*-test or the Wilcoxon rank-sum test, the non-parametric alternative to the *t*-test.
  - For the *t*-test, `t.test()`
  - For the Wilcoxon rank-sum test, `wilcox.test()`,
- The function `power.t.test()` will compute the power for a *t*-test or determine the parameters needed to obtain the desired power

Example - *t*-test

```
# The ToothGrowth dataset looks at the effect of vitamin C on
# tooth growth in guinea pigs. Three dose levels of vitamin C
# (0.5, 1, and 2 mg) were given by two different methods
# (orange juice and ascorbic acid) to 10 guinea pigs.
head(ToothGrowth)

# Only focus on the 1mg dose
Tooth.1mg <- subset(ToothGrowth, dose==1)

# From the boxplot orange juice leads to longer teeth
with(Tooth.1mg,boxplot(len~supp, names=c("Orange Juice","Ascorbic Acid"),
                        xlab="Supplment Method", ylab="Tooth Length"))

# Two-sample t-test
tt <- t.test(len~supp, data=Tooth.1mg, alternative="two.sided",
                var.equal=FALSE, conf.level=.95)

# Extract results
# See the documentation for a description of the values returned
names(tt)
tt$p.value
tt$conf.int
```

Example - *t*-test

```
# Suppose the dataset does not have an indicator column
uns <- unstack(Tooth.1mg, len~supp)
t.test(uns[,1], uns[,2], alternative="greater", var.equal=TRUE,
         conf.level=.90)

# One-sample t-test just for Orange Juice
# Ho: mu=10 vs Ha: mu!=10
t.test(Tooth.1mg[Tooth.1mg$supp=="OJ","len"], mu=10)

# Paired t-test
# The sleep dataset is a match paired study with 10 subjects. Compared
# the effect of two drugs on increasing hours of sleep.
t.test(extra~group, data=sleep, paired=TRUE)
with(sleep, t.test(extra[group==1]-extra[group==2]))

# Power
# Exactly one of the parameters n, delta, power, sd, and sig.level must
# be passed as NULL, and that parameter is determined from the others
pw <- power.t.test(delta=1, sd=1, sig.level=.05, power=.8,
                    type="two.sample", alternative="two.sided")
names(pw)
pw$n
```

Comparing Variances

- Use `var.test()` to performs an *F*-test to compare the variances of two samples from normal populations.

```
# ToothGrowth dataset where dose=1mg
Tooth.1mg <- subset(ToothGrowth, dose==1)

# Sample Variances
with(Tooth.1mg, tapply(len, supp, var))

# Formula interface
var.test(len~supp, data=Tooth.1mg, alternative="two.sided")

# From histograms the normality assumption of the F-test is not reasonable
par(mfrow=c(1,2))
with(Tooth.1mg, hist(len[supp=="OJ"],
   main="Histogram of length for Orange Juice", xlab="Length"))
with(Tooth.1mg, hist(len[supp=="VC"],
   main="Histogram of length for Ascorbic Acid", xlab="Length"))

# Vector interface, Ho:theta1/theta2 = 2 vs Ha: theta1/theta2 > 2
uns <- unstack(Tooth.1mg, len~supp)
vt <- var.test(uns[,1], uns[,2], alternative="greater", ratio=2)

names(vt)      # Extract results
vt$p.value
```

Comparing Proportions

- To compare proportions using a normal approximation use `prop.test()`
- To perform an exact binomial test use `binom.test()`
- The function `power.prop.test()` will compute the power for a two-sample test for proportions or determine the parameters needed to obtain the desired power

```
# One-sample test, 17 successes out of 30 trials
prop.test(x=17, n=30)         # Ho: p.hat=.5 vs Ha: p.hat!=.5 (default)
prop.test(x=17, n=30, p=.25)  # Ho: p.hat=.25 vs Ha: p.hat!=.25

# Two-sample test, Ho: p1=p2 vs Ha: p1!=p2 (default)
prop.test(x=c(14,19), n=c(20,30))

# Can also use a matrix to specify the successes and failures
obs <- matrix(c(14,6,19,11), ncol=2, byrow=TRUE,
         dimnames=list(c("Type 1", "Type 2"), c("Success", "Failure")))

# Ho: p1=p2 vs Ha: p1>p2
# Do not use the continuity correction, default does use the correction
pt <- prop.test(obs, alternative="greater", correct=FALSE)

names(pt)
pt$conf.int
```

Normality

- To perform the Shapiro-Wilk test of normality use `shapiro.test()`, Ho: Data is normally distributed vs Ha: Data is not normally distributed
- To produce normal quantile-quantile plots use `qqnorm()` along with `qqline()`

```
# Normal Data
x <- rnorm(100)
shapiro.test(x)
qqnorm(x) # Draw Q-Q plot
qqline(x) # Add a line to the plot that passes through the
          # first and third quantiles

# Non-Normal Data
y <- rgamma(100, shape=2, scale=.5)
shapiro.test(y)
qqnorm(y)
qqline(y)
```

Independence

- To test for correlation/association between paired samples using either Pearson's correlation, Kendall's tau or Spearman's rho, use `cor.test()`
- Kendall's tau or Spearman's rho statistic is used to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.
- All three methods test,
  Ho: No Correlation/association vs Ha: Correlation/association
- To perform a chi-square test of independence use `chisq.test()`
- To perform Fisher's exact test use `fisher.test()`

Example - Independence

```
# Pearson Correlation Test
# Ho: rho=0 vs Ha: rho!=0
x <- rnorm(20)
y <- 3*x + rnorm(20)
plot(x,y)
ct <- cor.test(x, y)
names(ct)
ct$p.value

# Hair and eye color for male and female statistics students
HairEyeColor

# Chi-Square Test for males
# Ho: Indpendent Ha: Not independent
ct <- chisq.test(HairEyeColor[,,"Male"])

# Warning message since expected count is less than 5
# Note this information is not in the documentation, need to look
# at the body of the chisq.test() function
names(ct)
ct$expected
```