SPLUS HINTS
by
John S. Hong, MD,MS
Revised 5/27/98

Introduction: I found condensing everything that Dr. Harrell and Dr. Perrugio did in class and in the book has helped me. I don't know if it will help you, but you can see. Just so you understand the format, when I show how to use a function, words that are in *italics* are variables that you will decide on using (exception is when I give a specific example). If the words are written in normal font, then you type those specific words/commands in. Some abbreviations are: fxn = function, x-axis = the vector you put there goes on the x-axis, stratifier = the conditioning variable, # = a number you put in, df = dataframe name.

What helps me is to keep the WORD 97 document open and use the Find command to look up S+ commands. I then copy the command and paste it onto my script file on S+. Also, the Find command is useful because some commands are scattered around this document.

I cannot claim that everything here is right, so take your chances! Enjoy.

VARIABLES

single-valued scalars
vectors – single column of numbers(intergers, single, or double precision) or character strings. They have length (# of elements)
matrices – rectangular table of numbers or characters.
factors - string of variable x (x_1, x_2,...x_k.) Categories are coded as integers (eg M=1, F=2)
data frames – rectangular dataset: a list whre all elements have same # of rows.
lists – "Tree" arbitrary collection of objects including other lists. Elements don't require equal lengths.
Attribute: names and length.

Variables are objects.

1. All have different characteristics: attributes
   a) names - vector of character strings
   b) class
   c) dim
   d) dimnames
   e) label - assign a more full descriptive name to the object
   f) etc.

Objects are usually a vector: eg. age - refers to all ages of the subjects
e.g. age[13] - to refer to the 13th subjects' age
ageg_ifelse(age<16,'Young','old') will assign the vector "ageg" as young if age is <16.
Objects have one or more classes

2. Get results by applying Functions.
   Function - does commands on the vectors. e.g. sum, mean,...

Vectors

1. Create
   a) x_c(1,2,3,4) x is the vector
      c = combination of numbers nonsequentially
      1,2,3,4 are its elements(arguments)
   b) a:b produces a sequence from a to b, one at a time.

c) *vector* - type in the vector name as such and the value set will be printed. Same for any other object

d) Delete: rm(*vector*)

e) Typing in a repeated character: vector_paste('A',1:8,sep='') will make the vector A1, A2,..A8

2. Function commands on vectors

a) *fxn*(*vector*) - put the vector name within parentheses.

b) *fxn(fxn(vector))* - can do functions within functions onto a vector

c) To print elements(observations) by skipping every other value:

vector[seq(*number of position you want to start, number of position you want to end*,by=*number of positions you want to jump by*)]

3. Mode - the type of elements of the vector. All elements must be the same type

a) integers. e.g. c(3,6,9). mode is numeric

b) real numbers. mode is numeric

c) complex numbers . mode is numeric.

d) logical values (T or F). mode is logical

e) character strings e.g. c('x','y','z'). mode is character.

mode*(vector)* - tells us the kind of vector we have.

mode*(vector)_newmodename* will change the mode of the vector

If you use functions on vectors where the modes are inappropriate, SPLUS will try to convert the mode so the function will work.

To test and change modes respectively: is.numeric,as.numeric
is.character,as.character
is.logical,as.logical
is.na – assumes arguments are numeric and tells you about NA's.

The is. will see if the vector is numeric, character, or logical. as. changes it.

4. Missing values for numeric and logical vectors: NA

a) is.na - to test for missing values. is.na(*vectors*)

b) Also to test for missing values, use a table for 2 vectors.

1) eg. table(is.na(x), is.na(y))

2) Under Hmisc, use naclus. Or type: nomiss(*argument*)cd

c) To get rid of missing values(NA): *fxn*(*vector1*[!is.na(*vector1*)])

5. Logical Comparisons

a) Operators are: >,>=,<,<=,==,!=,&,|,!

b) ! means negation

c) & and | for logical and, or

6. Index - select subsets of a vector. you index its elements or other vectors. Use [ ]

a) eg. x[i] x is the vector
i is an element or even another vector
NA is not allowed in the elements.

1) To logical vectors: [1] extracts values of T

2) Character strings: x must have a names attribute

i)print varibles values by certain labels. e.g. print for all females in the vector:
variable[sex= ='female']

Names Attribute - vector of character strings of the same length as x which effectively names each element of x.

GETTING HELP

1. ?*function* or help(*function*). This will tell you what the function does:
    a) Description
    b) Usage
    c) Required arguments
    d) Optional arguments
    e) Value
    f) Details
2. ??*function* - to just remind you what the arguments to the fxn are to use.
3. Under description, you may get symbols like:
    a) na.rm =T. na.rm is to remove missing values. If =T, the function will remove them unless you change them. e.g. mean(x,na.rm=F)
    b) trim - to trim a mean value.
4. To list arguments of a function instead of using ?? or ?
    a) names(*functionname*)
    b) *functionname$argumentnam*e
    c) sort(names(*functionname*)) - lists the function's arguments alphabetically
5. To learn the FULL name of a function
    a) methods(*functionname*)
    b) methods(class=class(*fit object*)) will list all the methods to handle fit objects.

MATRICES, LISTS, AND DATA FRAMES

1. Matrices - collection of vectors of same length and same mode. A matrix is an object.
    a) cbind - puts arguments in columns. eg: cbind(*vectors*)
    b) rbind - puts arguments in rows. eg: rbind(*vectors*)
    c) matrix(*data,nrow,ncol....*) - function is matrix. Reads the data in a stream. By default, the arguments will be put in rows then columns.
    d) apply(*matrix, margin, functionname*)
        1) margin: 1 = rows, 2 = columns
        2) apply will allow the functionname be performed to a row or column of the matrix.
    e) Index – to select a few rows and columns [*rows,columns*]. [,column] and vice-versa will print all the row.
2. Lists – collections of objects of different kinds. Purpose: summerize all info related to a particular task.
    a) Dimnames – to give names to the rows and columns
        Dimnames(*vector*)_list(*rownames,column names*) If you use c(' ',….), you can Individually name each row and column variable.
    b) list – the fxn to create the list. *Listname*_list(*the attributes and vectors components*)
    c) Components of the list can be selected in 2 ways
        1) Refer to position on the list. *Listname*[[*position #*]]
        2) Direct name*: listname*$*component*
    d) tapply(*vector*,list(*variable1, variable2*),*fxn*) Here use list for cross classification of the 2 variables. It collects item together.
        Doesn't work with categorical variables. Use summary(*y~x1+x2*, fun=function(y)c(mean=mean(y),median=median(y)))
3. Data Frames – a list where all components have same length, but unlike matrices, it can have different types.
    a) can contain a matrix if lengths are equal in the data frame

b) Create
   1) read the data into the dataframe from ASCII or SAS: read.table, sas.get
   2) construct a new one: *data.framename*_data.frame(*row variable name,column variable name,*row.names=*vector*)

c) Make table: *vector1*_data.frame(*argument1,argument2,etc*,row.names=*vector2*) Row will begin with the vector 2 arguments that you made previously.

4. Repeat command for rows and tables: rep("*character*",x) x is number of time to repeat.
   a) rep(c(1,2),c(2,3)) → 1,1,2,2,2 because the value of 1 is repeated 2x, and value 2 is repeated 3x.
   b) rep(c(1,2),5) → 1212121212
5. Have extra names of the rows: row.names=*vector*. The vector contains the names of each variable. Eg: df_data.frame(*rows,columns,*row.names=*vector*)

ATTRIBUTES

1. names - vector of character strings
   dim – dimensions of a table (rows and columns)
   dimnames – to names the rows and columns of the table. If no names, NULL
   class – very special attribute related to the concept of methods
   level – in factoring, levels are the values assigned to the factored vectored x.
   row.names – to give extra names to the rows
   label - assign a more full descriptive name to the object
2. Certain characteristics of objects: mode (numeric, complex, logical, character, list) and length (number of elements of a vector or matrix; number of components of a list) and attributes.
3. Each attribute has a function that will extract the attribute out.
   a) attribute(*vector*) – this will tell you that particular attribute for the vector. But sometimes it may only tell you what kind of object you have.
      1) eg. dim(cx) – will tell you the dimensions for cx
      2) attribute(vector) – prints out label, class of the vector.
   b) attribute(*dataframe*) – prints names, row.names' characters, and the class(data.frame). The system know the df is a dataframe and treats it as such.
4. Change attributes or delete them
   a) *attribute(vector)_new properties* eg: dim(rx)_NULL will take away the rows and columns.
      eg. dim(*vector*)_c(5,4) – gives the 5rows and 4 columns.
5. attr – this function will create a new attribute
   attr(df,which="*new attribute name for the vector df*")_*command*
   a) attributes(*vector*) – gives the attributes and how they are composed
      1) if you have thousands of names that will come out, you can limit them by names(attributes(...))
6. Class: fxns will act in different ways depending on the class of the object
   a) Unclass: this will removed the class. eg. If you unclass(*dataframe*), a list will print out instead of a table.
   b) Types of Classes: Factor – object with a discrete set of levels. eg. x_1, x_2,...x_k.
      The level is a vector with the unique set of *values* of x.
      These levels x's have labels to *name* them: l_1,....l_k,;character strings defining the corresponding integer codes. (Internally, the x values will be stored as integers.) BUT, can't do math transformations on it because the categorical variables may or may not be ordinal.

1) factor(vector *x*,*levels*,*labels*,exclude=NA)
2) label(*dataframe or vector name*) – prints your definitions of the levels.

7. Change names of vectors
   a) names(*vector x*)_c(*character strings*)
   b) BETTER WAY: names(*df1*)_edit(names(*df1*)) and it will put you in notepad to edit.
8. label (not labels): to better document your variables so you remember what they mean.
   label(*variable*)_"*what you want to call it*"
9. GROUPING variables into 2 or more groups:
   a) levels(*x1*)[levels(*x1*) %in% c("no", "verymildly", "mildly")] <- "cases" These 3 answers will be labeled as cases now.
   levels(*x1*)[levels(*x1*) %in% c("moderately", "strongly", "absolutely")] <- "controls"
   b) x1_ifelse(x1=="A",x1,"BC") – This will take anything that is not labeled as A and change them to BC.
   c) x1_ifelse(x1%in%c('B','C'),'BC',x1) – takes all values B and C and changes them to BC.

LIBRARIES

1. Trellis – more later.
   a) library(Matrix) – for advanced matrix fxn
2. Hmisc – library(Hmisc,T)
   a) if need to get it back to it after you close it: help(library='Hmisc')
3. Functions of Hmisc
   a) describe(*df*)
   b) hist.data.frame(*df*) – get histograms for every variable
   c) ecdf(*df*) – step wise graph for each variable. The proportion is from zero to one.
   d) naclus(*df*) – tells you about missing data
      plot(naclus(*df*)) or plot(naclus(df[,c(column numbers)]))
   e) datadensity(*df*,which=continuous) – you get tick marks of frequency on a line for each variable in the dataframe.
4. .First_function(){
   library(Hmisc=T)
   any other libraries or functions you want
   invisible()
   }
   a) to edit .First: .First_edit(.First)

MAKING FUNCTIONS/COMMANDS

1. You can make your own function/commands

```
Lxx_function(x,y,a,b){
x1_sum(x^2)
x2_((sum(x))^2)/length(y)
x1-x2
}
```

   a) name your function, eg: Lxx
   b) After function, you put in the variables needed in the parenthesis. The order counts!
   c) You put one command in per line and it does it in the order you place them. So x1 is calculated, then x2 is calculated, and then x1-x2 is done to give you the answer.
   d) The { braces } go exactly as shown. You must write it just like it is above.

CHAPTER 3: DATA IN S-PLUS

READING RAW DATA

1. Reading ASCII datasets – a table.
    a) scan – more versatile. Input lines do not need to correspond to one complete set of fields and the user decides what mode each field should have.
    b) read.table –easier. But this expects the input data to be in tabular form. Page 49.
    read.table(file,header=<<see below>>,sep,row.name,as.is=F,na.strings="NA",skip=0)
    This tries much harder to interpret the input data auntomatically, figuring out the # of variables & whether fields are numeric. It processes a more structure object as output.
2. Import data through "file" fxn on table menu. Make sure the type of file you are looking for is there.
    a) If has S+ format string: Copy the string in the word document first. Then import data via menu→hit Formatted ASCII file→select the dataframe→go to options panel→under Format String hit control v to obtain the string which will format the columns.
    b) *.asc file: Right click→sent to→notepad. This will open up the word document to read.
    c) Data is imported by rows. If you have data that OVERFLOWS into more than one row, you need to use SCAN: Use matrix function to put things in right columns and rows. Transform the matrix into a dataframe then.
        1) Tell rows (byrow=T) and columns (ncol) . Reads data by row vs. by column. You must tell scan to do this since it is more primitive than read.table.
        2) *df*_data.frame(matrix(scan('*location of file in file manager*'),ncol=#,byrow=T))
        3) Everything is read as numerical, not categorical. Therefore, you convert using factor.
        df$vector1_factor(df$vector1,levels…..)
        4) rename columns: names(df)[*#column*)_'*new name'*
        5) Code missing values. In SAS, 99 is often used. Do this by columns:
        df$column-name1*[*df$column-name1==99]_NA Assigns all 99's to NA.
3. is.*class*: put in the mode/class that you want to see what the vector is.
    eg: is.numeric, is.character. is.na will assume all arguments are numeric and look for NA's.
4. Import/Export another person's S+ dataset
    a) To export, you use 'data.dump' function to make a dataset that is transportable.
    data.dump(c('dataframe'),'a:/whatever folder you want')
    b) To import: data.restore('a:*file*')
    data.restore('c:/windows/whatever folder you want the document in') – to import an S+file

READING SAS DATASETS(FACTOR)

1. sas.get
2. When Labels or levels aren't done in SAS, you create or redefine them in S-PLUS.
    a) factor(*filename from SAS,level,labels*) eg: sex_factor(sex,1:2,c('female','male')). The sex in SAS had level 1 to be female, and level 2 to be male. And level 1 was listed first, so we put 1 first as female. (The order is important to label things)
3. Dates get changed in SPLUS. Use the fxn: chron(*dataframe$datename*) This will print the correct dates
4. Arguments in the library: pages 53-57.
    a) as.is: if as.is=F, then SAS characters are converted to S factor objects.
5. Permanently changing variable attributes from SAS to S+ when adjustments are needed.
    a) Use $ to change individual variable in a list/df.
    b) Temporarily Edit names:
        1)names(*df*)_edit(names(*df*)): changes name of the columns

2) names(*df*)[2]_'*age*': changes individual name in column 2.
3) names(*df*)_changecase(*df*): changes all the names to lower case.

c) Permanently changing names
1) label(*df$age*)_"*Age in years*"
2) for whole factor change: *df$vector*_factor(*df$vector,levels,labels*)
e.g.: *df*$sex_factor(*df*$sex,levels=c(1:2),labels=c('female','male'))
3) for levels: levels(*df$vector*)_c(*character strings*)
eg.: levels(df$treat)_c('treatment A', 'treatment B', 'treatment C') or
df$treat_factor(df$treat, c('a','b','c'),c('treatment A', 'treatment B', 'treatment C'). In both cases, df$treat was originally assinged levels as 'a','b','c'.

NOTE: label and labels are different. Labels is used to name the level. Level is the code. Label is for you to remember what your variables mean.

d) Recode a factor: *vector2*_recode(*vector1*,c('*new1','new2'*),c('*old1','old2'*))

WRITING OUT DATA

1. To share the data with other users: make ASCII file
   a) write.table function. page 61 has the arguments.
2. Customize printing
   a) cat(*character string,object/function,character string 2*)
   eg: cat("the mean of x is",mean(x),"for ill patients")

CHAPTER 4: OPERATING IN S

READING & WRITING DATA FRAMES & VARIABLES

1. _Data is in my ajohn folder for more organized data management.
   a) All objects that you create for a particular project are available because S+ by default goes to _Data.
   b) Also a search list is established for other directories besides _Data.
   c) search() – this function prints out the list of all directories that S+ searches, looking for fxns and data. The directories are printed in order of highest to lowest priority.
   d) find(*dataframe*) – tells you position of that dataframe.
2. Attach function: to reference objects that are not in the default search path.
   a) attach('*directory name'*)
   b) attach(*dataframe*) – df do not have " " around them like directory names do.
   -both above will place the df or directory into position #2. (Position #1 is ajohn/_Data)
   Therefore, you don't have to use (dataframe$vector) anymore. The dataframe is used by default, so you just type in the the vector. Note: when you create a vector for the dataframe, the new vector will not show up in the object browser until AFTER you detach the df!
   c) attach(df,1) will put df into position 1. This uses up a lot of memory. If you don't save the data, it will be erased when you leave. Best used for changing a HUGE dataframe. Then when you detach (see below), you'll have a new df.
   d) Attach subsets: attach(*df*[*vector1='whatever'*,c(*columns desired*)])
3. Objects function: to list the individual objects in the directory you just attached
   a) objects(2) – lists all objects in search position #2.
   b) objects.summary – gives more details.
4. When you attach your dataframe, find out all the variables with names:
   names(*vector*).
   Then to learn more about the vectors, use describe function to get statistics.

5. For large dataframes that use up too much memory:
    a) use.names=F. When you attach, specify this. It will stop the row.names attribute from being copied to each object within the frame.
        eg: attach(titanic,use.names=F)
6. Detach – take df off the search list. 2 arguments: what, save.
    a) what: number denoting a position in the search list
    b) save: character string with the name of the object where we will store the modified df.
        eg: detach(prostate,save='pros'). The new variables we made while in titanic will now be saved (along with the old variables) in a df called pros. It will only save large variables, not small ones.
    Purpose: if you have a huge dataframe you want to alter, attaching a file into position 1 will make it much easier. Then when you are done, you can have a new named df with permanent changes.
    c) Overall do: detach(1,'*df*') to save everything.
        1) save=F will delete all new vectors made.
    TO DELETE: rm(*dataframe*) or you could type in every vector you want instead of the df.
7. Subsetting Dataframes
    -if only a subset of the dataframe needs to be analyzed, it can be easier to temporarily make a subset for easier access. 2 ways
    a) subset function: vector_function(vectors,subset=variable=='subset category')
    e.g.: f_mean(age,subset=sex=='male') – if sex is a vector, and you want only men, you use sex=='male'. By subsetting here, you take the mean ages of only men.
    b) Or create a new dataframe for the subset of interest: eg: df.male_df[df$sex=='male',] and run the functions on df.male.
        1) better to create this after attaching a file: eg attach(df[,c('age','sex')])
        eg. attach(df[1:100,c(1:2,4:7)]) – will get first 100 rows and variables 1,2,4-7 in the columns..
        eg. attach(df[,-4]) – don't get variable #4.
        eg attach(df[,names(df) %nin% c('age','sex')]) – gets all by age & sex using Hmisc.
8. Deleting Variables from DF
    a) df$age_NULL – using NULL permanently deletes variable.
9. Assign & Store save new df's outside of .data subdirectory while in position #1.
    a) assign(*vector,value,frame,where="subdirectory you want*")
        eg: assign("age>50",age>50,where="_Data") to put it in ajohn_Data.
    b) Store – because assign uses memory and slows us down. Store fxn is in Hmisc.
        store(*object,name=as.character(substitute(object))*,where=".Data")
10. Access objects stored in other directory & you don't want to attach it: get and store()
    a) store() – a temporary directory is attached in position one. All new objects reside here temporarily until you quit S+ or decide to store them in a df or subdirectory.
    b) eg: store()
        z_get("model",where='.....') Model is now stored in the temp directory under the name z. We can now access model in our current directory.
11. Documenting df's – for long term projects so you remember which df's are correct, which are still being worked on, which is the original, etc.
    a) comment fxn: attach an attribute to the df. Hmisc is needed.
    eg: comment(df)_'From SAS dataset/myproject/mysas on machine A'
    b) if you type: comment df – it will print out your attribute.
    c) attr function – for an object in the df.
    eg: attr(df,'doc')_ 'From SAS dataset/myproject/mysas on machine A'

If you type attr(df,'doc'), the attribute will be printed.

d) Making new variables to df: *df$vector2_df$vector1$vector*

MISCELLANEOUS FUNCTIONS

1. Functions for sorting
   a) sort(names(*df*)) – arranges in ascending order. Can add na.last=NA,T/F after the vector discard or included missing values.
   b) rev(sort(vector)) – arrange in desceding order
   c) order – returns the order permutation of a vector. The first element is the index corresponding to the smallest element, and so on. You can operate on more than one vector simulatneously.
      eg: order(x,y) – order based on x; ties resolved according to values of y.
2. By Processing
   a) tapply – to do repetitive operations on an object It summerizes the stratification of a vector
      eg: tapply(*main variable,stratifying variable, func*tion) The main variable (which must be a vector) is not labeled in the table. If use 2 stratifiers, use: list(*a,b*)
   b) for combinations of stratifiers
      tapply(*main variable*,interaction(*stratifier#1,stratifier#2*,drop=T),*fxn*) – drop=T drops combinations with no observations in them.
   c) To make table, use list
      tapply(*main variable*,list(*stratifier#1,stratifier#2),fxn,na.rm=T)*
   d) by fxn: process on all variables in a df when summerization fxn operates on df's. For both df and vectors. Stratification is done here as in tapply, but is does grouping better.
      by(*main variable*,list(*name=name*), FUN=*fxn, descript*=label*(variable))*
3. Operating fxn's on a series of variables or all variables in a df: lapply or sapply
   a) lappy – applies a single fxn to every element of a list/df and returns a list as the result.
      lapply(*df,function*)
      eg: lapply(x,quantile,probs=c(.25,.5.,.75)). And x=data.frame(x1=rnorm(100),x2= runif(100)). So the quantiles are done to both x1 and x2.
   b) sapply – it is like lapply, but a list is not printed. Instead, the vector format is printed. So, if the vector is a matrix, the result is a matrix.
      sapply(*vector,function*)
   c) llist fxn: when you perform repetitive operations for several variables & you need to access the labels or names of the variables during processing. Hmisc fxn.
      eg:
   sapply(llist(age,height,pmin(weight,200)),function(x)plot(x,blood.pressure)title(label(x)).
   d) aggregate: for doing separate analyses of multiple variables in a df, with simultaneous stratification on by-variables. It's like 'by' but can use multiple variables. Table is made with the main variable shown unlike in tapply.
      aggregate(*df*[c(*variables, ...*)],*stratifier variable*,FUN=*fxn*)
      1) if you are in position #1: aggregate(data.frame(*variable*),*stratifier*,FUN=*whatever*)
4. Controlling significant digits or rounding
   a) signif(*vector*,dig=*number of sig figures you want*)
   b) round(*vector,number to what decimal point you want*)
   c) to set a default for the significant digits: options(digits=*number you want*)
   d) options(digits=*n*) this sets the default significant digists at n.
5. Creating repeating values:
   a) x_c(rep('*label name',# of times you want it to repeat*)

4.2.4 FXNS FOR DATA MANIPULATIONS & MANAGEMENT

1. seq: specifies a starting(a) to ending point(b) with distance(z) between them for the elements in the vector. seq(*a,b*,by=*z*)
2. unique: find smallest values of a vector
   sort(unique(*x*))[*1:5*] will find the five smallest values
3. match: looks up elements of x in a table. to merge dataframes.
   match(*x,table)*
4. abbreviate: shorten variable names, row.names, or labels.
   names(*df*)_abbreviate(names(*df*)) – abbreviates all data frame names
   row.names(*df*)_abbreviate(row.names(*df*)) – for row names
   label(*x*)_abbreviate(label(*x*)) – for a single label
5. expand.grid: produces df's with a combination of all levels of specified variables.
   z_expand.grid(age=*fxn*(age),rx=levels(rx),bm=c(0,1)) - for example will make a table with all these functions
6. cut2: in Hmisc to categorize variables and return a factor. Good for making labels for levels of the resulting factor variable. It cuts a vector into segments for stratification.
   table(cut2(*df$vector*,cuts(#'s),g=#)) or table(cut2(*df$vector*,cuts(#'s),m=#))
   eg. table(cut2(prostate$age,g=5)) g is to classify it arguments into g intervals with approximately the same # of observations in them. m will classify its arguments in m intervals with minimum number of observations.
   eg: cut2(age,50) – it will print out ages below 50 and above 50 for stratification.
   cut2(age,c(30,50)) will do 3 segments: up to 30, 30-50, and above 50.
      a) an alternative is: x_(age>=30)+(age>=50) will do the same thing.
7. cut: create categorical data out of numerical or continuous data
   eg: can group 50 states into high and low murder. Put the cut into a factor
      factor(cut(state[,"murder"],c(0,8,16),labels=c('low','high'))). What will happen is the levels will be labeled.)

4.3 RECODING VARIABLES & CREATING DERIVED VARIABLES

1. ageg_ifelse(age<16,'Young','old') will assign the vector "ageg" as young if age is <16, and "old" if not less than 16. ("if this- then that, else give the other value.")
2. To recode a string of values, create a new vector with the new codes first
   a) newcodes_c(cat='feline',dog='canine',pig='porker')
   Then take the old vector and subscript it into the newcode
   if old vector was animals_c('cat',''cat','pig','dog')
   Then animals_newcodes[animals] will label all the character strings.
3. score.binary: create a new categorical, ordinal, or numeric variable from a series of binary or logical values. In Hmisc. It acts like the ifelse function.
   x_score.binary(cx=='done',abx=='yes',cx=='done'&abx=='yes') Then table(x) will give you the frequency of these criteria you just put in.

None	Cx=='done'	abx=='yes'	cx=='done'&abx=='yes'
14	4	5	2

   In this scenario, the criteria are all printed, but so is the None part by default.
4. table – will report the frequency of the variables.
   eg: x_(abx=='yes')+(cx=='yes')
   table(x)

0	1	2
14	9	2

   (0 means 2 no's, 1 means one is yes and one is no,2 means both yes)
5. Recode a factor: *vector2*_recode(*vector1*,c('*new1*','*new2*'),c('*old1*','*old2*'))

4.4 MISSING VALUE IMPUTATION USING HMISC

1. Replaces missing values with the median (unless you type in fun=mean)
   x_impute(*variable*)
   summary(x) – will tell you what has been imputed and the value assinged. Quartiles too.
   describe(x) – tells how many values were imputed and the mean and quartiles.
   is.imputed(x) – T means that spot was imputed.
   mean(*x*[subset=!is.imputed(*x*)) – this will give the mean of x without using the imputed value.
   Print NA in a factor that contains " " (blank): x_factor(variable,exclude="")
2. Reassign a value
   a) 99 with NA's.
      df[,c(columns you want)][df[,c(same columns)]==99]_NA
   b) df$x[x==99]_NA – but I had some trouble with this method.

CHAPTER 5
5.1 BASIC FUNCTIONS FOR STATISTICAL SUMMARIES

1. Problems with NA: cor, mean, median, min, max, quantile.
   a) use na.rm=T to delete NA
   b) for mean and median, use APPLY fxn to calculate columns of matrices.
2. Min and max values for matrices: pmin & pmax
   a) for vectors in a dataframe, use fxn max or min:
      x_table(sex)
      x[x==max(x)] and it will print out in table form the sex category that is more frequent and what the frequency itself is. ie Female 14.
3. rcorr: computes Pearson and Spearman rank correlation matrices and P-values. Does pairwise deletions of NA's.
   rcorr(cbind(x1,x2,x3….),type="spearman")
4. hoeffd: pairwise deletion of NA's in Hoeffding's general measure of dependence between 2 variables.
5. bystats & bystats2 (Hmisc): obtain stats on a variable by levels of several classifications. But it is better to use **summary.formula** – gives many statistics for more than one variable.
   bystats(*main variable, stratifier 1, stratifier 2*, fun=*fxn*) default fun is mean.
6. Page 102: table of functions
7. d: density
   p: cumulative distribution
   q: quantiles
   r: random samples

Put any of these 4 letters to get the name of the desired function. So you add it as a prefix to any of the fucntions: beta, binom, cauchy, chisq, exp, f, gamma, geom, lnorm, logis, nbinom, norm, pois, t, unif, weibull, ecdf, bpplot.

   pnorm(1.96) – gives you the Z score (left of the phi value)
   rnorm(10) – gives you 10 random normal distribution values
   qnorm(.975) – gives you Z_p value (percentile Z)
8. Generate just random numbers alone: x_runif(#)
   a) runif(1000) – will peform your function on 1000 r.v.'s
   b) Generate random letters: x_letters[1:7] gives you a-g.
9. Pseudorandom number generator: set.seed(*number of random numbers you want*)
10. Quantiles: Uses F quantile distributions. f=.5 is the median, .25 is lower quartile, .75 upper quartile.
    a) quantile(x,probs=seq(0,1,.25),na.rm=F) is the default. X is vector of data. Probs is the vector desired probability.
    b) Make your own quantiles: quantile(*x*,c*(.1,.2,etc*))
    c) qqplot(x,y,plot=T) X and Y are 2 different vectors and don't have to be same length. Gives

a linear looking plot starting from zero. Graphs the quartile values for x and y. 45 degree angle means the distributions are the same.

1) qqplot(x,0.5*y) will multiply the values of y
2) qqplot(abs(x),sqrt(abs(yy))) does absolute values and square root.

5.3 HMISC FXNS FOR POWER AND SAMPLE SIZE CALCULATIONS

1. bpower: to estimate power (page 105)
   bpower(p1(known for pop),p2(your estimate),OR,percent.reduction,n,n1,n2,alpha=.05)
   a)ballocation
   b)bpower.sim(p1,p2,OR,percent.reduction,n,n1,n2,alpha=.05) n must be whole integer of factor 10.
   c)bsamsize(p1,p2,fraction=.5,alpha=.05,power=.8)
2. popower: power for 2sample test for ordinal responses
   popower(p,OR,n,n1,n2,alpha=.05) – p may be written as c(p1,p2)
   posamsize – sample size for 2sample test ordinals
3. cpower: cox/log-rank 2sample test. Page 108 for format

5.4 STATISTICAL TESTS

1. 2SAMPLE T-TEST, 2sample Wilcoxon test, Spearman rank correlation test, Chi-square
   See table 5.4 page 113.
2. Spearman Correlation Test: get P-value and degrees of freedom
   a) spearman.test (in HMISC): spearman.test(variable1, variable2)
   b) rcorr(variable1, variable2,'spearman') – this is an alternative way to get spearman test.
   c) non-montonic relationships between 2 continuous variables
      spearman.test(variable1, variable2,2) – the 2 at the end gets a 2 d.f. test allowing for one turn in the non-monotonic fxn.

3.somers2(Hmisc) – correlation measure when one variable is Binary
   somers2(variable,sex='male')

4. rcorr.cens: get correlation when censoring is absent
   rcorr.cens(variable1, variable2)
   a) calculate 2-tailed P-value: use results from rcorr.cens. Take Dxy and S.D. values as such:
      (1-pnorm(Dxy/S.D.))*2
5. Wilcoxon Tests
   a) 2 sample: wilcox.test(variable[sex=='female'],variable2[sex=='male'])
      Give Z and P-values
   b) 1 sample: wilcox.test(variable[sex=='whatever'])
   c) 2sample as a special case of proportional odds model. Give very accurate P-value but is VERY slow to do.
      library(Design,T)
      lrm(variable~value)
   d) Using MENU: Click under statistics → 2 sample tests → Wilcoxon. Under the dataframe box, type in your dataframe. If you want to subset it: *df*[*df*$*vector*=='*whatever*',] Leave the space blank after the comma to indicate you want all the columns to be included. Then under x variable, put in your independent variable. For y, you can change it to a grouping variable by clicking the little white box up in the left corner. This allows for factor with 2 or more parts.

5.4.2 PARAMETRIC TESTS

1. T-TEST
   t.test(variable[*index variable*],variable2[*index*]) – give P-value, df

CHAPTER 6: MAKING TABLES

1. print.char.matrix: format tables into boxed cells
2. crosstabs: frequency tables, computes Pearson chi-squared stats
    a) you can first open a data set inside Splus called solder. crosstabs(~*row variable*+*column var*) crosstabs(~Solder+Opening,data=solder,subset=skips>10) – this prints out a boxed table for the data set solder, chisquare value, and p-value
3. Tables that categorize: *vector3_vector1*==('*whatever1'*)+(*vector2*=='*whatever2*') The + makes it divide the table into both, one or the other, and none.

6.2 Hmisc summary.formula Function (page 118)

1. summary.formula – do a summary command on a formula object: constructs a large variety of tables of descriptive stats.
    a) typesetting: Use LATEX from a display package
    b) plot(summary(y~x1+x2…)): converts tables into **dot charts**. GREAT WAY TO SEE CHART OF HOW EACH VARIABLE RELATES TO THE OUTCOME.
    c) summary(*vector1~stratifier*,data=*dataframe*,fun=*table/mean/etc*): will print a table (if you use fun=table)for vector1 in the columns (dependent variable), and the stratifiers (independent) in the rows. Mean is the default. The vector is the formula, and the ~ assigns the stratifiers onto this formula. ~ means, "is modeled as". Leave out data=_____ if the dataframe is attached to position #1. na.rm=T is default.
        The stratifiers are independent variables.
    d) can use more than one stratifier
    summary(vector1~stratifier1+stratifier2,data=dataframe)
    e) can index as well
    summary(vector1=='yes'~stratifier,data=dataframe,fun=table)
    f) To get multiple function results, name a new fxn "y" to is a vector with both mean and median.
    x_summary(duration~sex+age,data=hospital,fun=function(y)c(Mean=mean(y),Median=median(y))
)
2. Cross-tabulate
    a) summary(*vector*=='yes'~*stratifier1*+*stratifier2*,data=*dataframe*,method='cross')
3. Method
    a)method='response' is the default. Makes the fxn summarize 1 or more response variables separately by levels of any number of right-hand-side variables. Quartiles are reported for stratifiers that can be divided as such.
        1) If you want to not report it in quartiles, you can use the g command. eg: for ages 4-82, the quartiles will be reported as [4,25),[25,41),[41,56),[56,82]. If you want age reported in 2 group (above and below the median), use g=2.
        summary(cx=='yes'~sex+service+age,data=hospital,method='response',g=2)
    If you also added, duration, that would be broken into 2 groups as well.
        2) Suppose you want age to be segmented into 2 groups, but you want duration broken into 3 groups? Use cut2
        summary(cx=='yes'~sex+service+cut2(age,50)+duration,data=hospital,method= 'response',g=3) This will give age as [4,50),[50,82], but duration as [3,5),[5,9),[9,30]
        or summary(cx…..+cut2(age,g=4)+cut2(duration,g=3)….) will give age in 4 quartiles and duration in 3.
    b)method='cross' results in a multiway breakdown – categorical right-hand variables are broken down into all their levels and tabulated against one another with the main variable in mind.
    c) method='reverse' reverses the meaning of the left-hand & right-hand-side variables. It switches what ib meant by dependent and independent variables.
4. LATEX –using print.display library to get typesetted tables.

latex(*variable*,npct='both',npct.size='normalsize',here=T) npct='both' will print both numerators and denominators.. Normal size font will be used.

5. summary.formula
   a) summary(*dependent variable~independent variable*,data=*df name*,fun=table/mean/median)
      Don't use data=df if the df is attached to position 1. This will print out a table with the independent variable in the rows. The columns will be the dependent variable.
      You can index the variables if you want. eg: dependent variable=="yes'.
   b) add on more independent variables using +
      summary(cx~sex+service) fun by default is mean.
   c) Cross Tabulate – use method='cross'
      summary(cx=='yes'~sex+service,method='cross') This will put sex on the rows and service on columns as a 2x2 table with culture as the main variable in question.
      Can make multivariant boxes using cbind:
      summary(cbind(*vector1*,*vector2*)~*vector3*,method='cross')
   d) If you put in a dependent variable that is continuous, you will by default get quartiles
      summary(cx~sex+service+age,g=#). g=# (# is 1-4) will break the quartiles down into the number of subsets you want.
      1) but if you have several different continuous variables, you may want to break them down individually using cut2
      summary(cx~sex+service+cut2(age,50)+duration,g=3) The g=3 will do it on duration.
      2) Or could put the g=# inside the cut2
      summary(cx~sex+cut2(age,g=4)+cut2(duration,g=3))
   e) Do more than one statistic
      summary(cx~age+sex,fun=function(y)c(Mean=mean(y),Median=median(y))) –typed exactly like this will print out both the mean and median columns. "Mean" will be written in the column as you typed it, as well as median.
   e) x_summary……. Then you can plot(x) to get a plot of it.

CHAPTER 9: PLOTS

1. plot(*fun1(var-x),fun2(var-y),…*) – plots a transformation fun1 of var on the x-axis vs. transformation fun2 of var2 on the y-axis. ie. plot(x axis, y axis). Fun can be things like log. If you don't do a fxn, just type in the variable alone.
   a) plot(x) – plots only only column 1 results – the mean.
   b) plot(x,which=2) – plots only second results out of function that normally gives 2 results.
   c) plot(x,which=1:2,pch=c(1,2)) –plots 2 results with different characters. which=1 does mean, which=2 does median. pch will do the mean on character 1 and median on character 2. If pch=c(7,9), the mean would be done on character 7 and median on 9.
2. Adding to plot
   a) plot(vectorx,vectory,xlab="*whatever1*",ylab='*whatever2*',main="*title of graph*") – the xlab will label the x-axis, ylab for the y-axis, and main for the header. To print the labels rotated 90 degrees, put in at the end: rotate=T
   b) To print it out: go to file menu and hit print option.
   c) To add lines into the dotted plot graph
      lines(supsmu(*vector1*,*vector2*)) – supsmu makes a non-parametric smooth fit line. Could use loess instead. loess does not accept missing values though like supsmu.
      Vector 1 is the x axis, vector 2 is the y axis.
   d) Factor variables – prints out as boxplot. If you want the mean to be shown within the boxes:
      boxmeans=T

eg. plot(Type,Mileage,boxmeans=T,rotate=T). The factor Type will be done in boxtype, and the boxmeans will mark the means of them. The labels will be printed 90degrees.

1) boxplot(split(Mileage,Type),varwidth=T,notch=T) is a better way to print the boxplot. Splint function is to classify Mileage by Type. varwidth keeps box widths proportional to the square root of #observations/box. notch – to make whiskers.

2) See Trellis as well.

e) title(sub='*text to decribe the symbols*',adj=#) will cause a subtitle to print. #: 0-left justification .5 center, 1 right.

f) title("*main title text*") does main title top

g) text(locator(#),'*some title*') prints out your title wherever your click the mouse. # refers to the order you click. So the first one is 1, the second one you make up is 2, etc.

text(locator(3),c('title1','title2','title3'))

h) point(locator(#),pch=#) puts a pch symbol wherever you click once.

i) show.pch() – the menu of pch values.

3. Editing plots

1) Don't plot out points: plot(*x,y*,type='n')

2) plot(x,y,lab=c(0,0,0)): lab is labels for tick marks. Default is c(5,5,7) for x,y,length.
To delete tick marks, put at end of plot command: xaxt='n' for x-axis, and yaxt='n' for y-axis.

3) Put 4 points onto corners:
corners_rbind(ne,nw,sw,se,ne)
points(corners[,1],corners[,2],pch=2)
lines(corners,lty=2) will connect the 4 points with dotted line. lty=1 does solid line.

4) plot(*x,y*,xlab='*xaxis name*',ylab='*yaxis name*',xlim=c(#,#),ylim=c(#,#)) # are the values along the x and y scalelines. ylim and xlim limits your scale lines so that you can make the ranges of the axis how you like them to be.

5) points(temp[sex=='female'],wbc[sex=='female'],pch='F'] this will make all point "F"

6) text(*xvariable* + 0.05, *yvariable*,adj=0) Moves text over 0.05 lines

7) abline(v=mean(x),lty=1) puts (v)ertical line of the mean. h for horizontal

8) portrait graph: win.printer(width=8, height=10.5,file='clipboard')

9) Put 2 graphs on same page
a) par(mfrow=c(*#rows,#columns*),oma=*c(bottom # of lines,left # of lines, top # of lines, right # lines*)) oma means margins. Often use oma=c(0,0,7,0) This sets up the page.
b) par(mar=c(*bottom#,left#,top#,right#*)+.1) for outer margins of graph #1. Often use c(5,5,7,0)
c) plot(*first graph*)
d) par(mar=c(bottom,left,top,right#)+.1) for graph #2.
e) plot
f) mtext(side=(#),line=2,cex=*,outer=T,"*Main title*")
is 1-4: 1 is bottom, 4 is right side. Cex is size of font. 2 is good.

10) Get both graphs on same scale limits →make new vectors to do this.
x.lim_c(min(x)-0.1,max(x)+0.1)
y.lim_c(min(y)-0.1,max(y)+0.5)
Then plot 1 and 2 using the xlim=x.lim and ylim=y.lim

11) lty = #, col = 3. These give you different colors and different types of lines.

12) for(*sx* in levels(sex){
plot(f,age=NA,pclass=NA,sex=sx,col=1:3,fun=plogis,ylab='Probability of survival)
title(sx)
}

Makes a loop so you will set variables to each sex when you make predictions. You run the commands on each level separately. Hence, you get 2 graphs on 1 page: one for male, the other for female.

```
for(x in seq(80,180,by=20)){
    plot(f,age=x,sex=NA)
    }
```

This will break down age into 80,100,120,140,160, and 180.

13) FOR TRELLIS – can't do par with bwplot.

w_bwplot

print(w,split=c(x,y,nx,ny)) – this divides your page into rows and columns. nx is # columns you want, ny is # rows. x & y are column and row you want your plot in.

eg: split=c(1,2,2,3)

bwplot put here	

4. Plot – applied to fitted models to display how the response function behaves as the predictors in the model vary. Therefor must adjust the predictors that are NOT being plotted. 2 ways:
   a) Passing them explicitly to plot as arguments
   b) datadist fxn. – takes a df/list and returns an object of class "datadist" (pg 172)
5. Plot looks at its argument class to make the appropriate plot.
   a) eg: plot.z (z is a class attribute of vector x). Plot will look from left to right until it finds a fxn of the form plot.z. (Print, summary, and anova do the same.)
6. 2 other plotting fxn
   a) qqnorm – nl probability plot
      qqnorm(resid(f)) – resid extracts the residuals. This fill see if in vector f, if the residuals are normally distributed.
   b) coplot – to see how variable1 depends on variable2 acrross different types. "co" –conditional.
      coplot(y~x|z,given.values=z,panel=panel.smooth) – gives scatterplot of y vs x conditioning on the values of z.
7. New HMISC plots and panel
   a) bwplot(*x-var~y-variable*,panel=panel.bpplot,data=*df*) will give box percentiles with more info.
   bwplot(*x-var~y-variable*,panel=panel.bpplot,data=*df*,datadensity=T,probs=seq(*you put in your percentiles*) will give datadensity on the lines and will give your own chosen percentiles.
   b) panel: panel=function means "Hey, I'm in charge so let me do the following." There are many panel.fxns available.
   c) plsmo = Plot smoothes estimates for x and y.
   plsmo(*x,y*,method=c('*lowess' or 'supsmu' or 'raw' though lowess is default*), datadensity=T, group=*stratifier*,col=*a:b*) col=a:b means how many different colors you want for each line.

9.2 ADDING TEXT OR LEGENDS AND IDENTIFYING OBSERVATIONS

1. identify(*xname,yname*,label=( )) Then use mouse to click on scatter plot points to label the point.
2. legend(locator(1),c('any text1,'anytext2'),lty=1:2)
   locator(n) connect n points with lines as you draw them on the screen by clicking the mouse
   The legend will then print your text into a box on the graph where you click

3.legends without mouse: legend(*x-axis point, y-axis point*,legend=c('*name1*','*name2*'),marks=c(1,2))

TRELLIS

1. dotplot(*y-axis~x-axis variable| conditioner1*conditioner2*,data=*df*,aspect=0.4,xlab='*name*')

2. xyplot(*y-axis variable~x-axis|conditioner,*,data=*df*,panel=function(x,y,…){panel.*function*(x,y,…)})
gives a scatterplot. 3 dots are used for everything else if you have a conditioner. *Function* would be: lmline – regression line. eg: xyplot(temp~wbc|sex,data=hospital,panel=function(…) {panel.lmline(…);panel.loess(…)}) Using (…) is better. And put the semicolon between panels.
a) For 2 colored dots for say, Male and Female:
xyplot(temp~wbc,group=sex,panel=panel.superimpose,data=hospital)
b) Using plsmo for better editing that loess. Need to put commands on separate lines as such:
xyplot(*yvar~xvar*,groups=*conditioner*,panel=function(x,y,subscripts,groups,...){
panel.superpose(x,y,subscripts,groups,...)
plsmo(x,y,group=groups[subscripts],add=T,trim=0,iter=3,prefix='')
})
3. densityplot(*y-axis~x-axis|conditioner*,data=*df*,layout=c(*column#,row#*),aspect='xy',xlab="*x name*", width=5) width determines width of window used to smooth histogram. Higher the #, smoother the curve. Aspect is ratio between x and y (banking technique).
4. Black and white: trellis.device(color=F)
5. Menu: FIRST TURN OFF EDITABLE OPTIONS UNDER GRAPH OPTIONS! object browser→click on dataframe→insert graph menu→your option (ie 2D density plot)→ under dataset choose your df→x column choice. You can now drag conditioners from the object browser into the graph AFTER YOU TURN THE EDITABLE OPTIONS BACK ON.
6. trellis.args – this shows what arguments are accepted by trellis.
7. bwplot – boxplots horizontally. Whiskers show highest value allowed that would not be an outlier.
bwplot(*y-axis variable~x-axis variable| conditioner1*conditioner2*, data=*df*,xlab='*name*')
8. histogram(*~x-axis | conditoner*,data=*df*)
9. splom (scatter plot matrix). Matrix where you have a scatter plot responding to 2 variables. At the intersection of any row and column, you get a scatter plot of those 2 variables.
a) Doesn't handle missing data
b)
splom(*~x-axis | conditioner*, layout=c(*column#, row#*),main='t*itle*')
10. Print darker graphs
a) fxn(*~vector1|conditioner*,col=1) col=color and 1 is black. Only can do if you have a one color plot.
b) Do this order: win.printer() → fxn(~vector……) → dev.off(). It will now print your graph.
11. Missing data removal: *fxn*(~nomiss(*vector*))
To examine how much data is missing, you can make a tree:
*f*_tree(is.na(*vector1*)*~independent variables*,data=*df*)
plot(f): At the ends of the branches, you see numbers for the categories. It reads left to right (eg: if male=1, female=2, then the first number will be for men.)
12. plsmo: stratifies and gives many graphs
xyplot(*y variable ~ x variable|stratifier1*stratifier2*,panel=panel.plsmo)

MODELING

ANOVA TABLES

1. F-test p values: 1-pf(*ratio variable, df.numerator, df.denominator*)
F ratio = MSR/MSE with F k,n-k-1 degress of freedom. (k is # predictors)
2. T-test for p value: 2 * (1 – pt(abs(*t ratio*), df))
b = Lxy/Lxx t ratio = b/(MSE/Lxx)^1/2

3. anova(f): Sequential Sums of Squares with F-test and p-values. Last reported value is a partial SS with the partial p-value for that variable.
4. Partial F-test: anova(*submodel f, f*) The submodel contains all the predictors except the one of interest. The one of interest will have a p-value give to see if there is an association with y. Degrees of freedom in the numerator is #variables in full model - #variables in submodel. Denominator d.f. = n-1 (n is #parameters.)
5. Interactions: to test if 2 or more variables have the same slopes for one common variable. eg: test if slopes for height and age are the same for the 2 sexes. If Y=FEV:

a) f_lm(fev~age+sex+height+smoker)
f.ia_lm(fev~age*sex + height*sex + sex + smoker)
summary(f.ia) will show individually if interactions with age and with height are significant.
anova(f,f.ia) – shows you if age*sex and height*sex together affect fev.

MISSING DATA

1. Work with NO NA's
   a) attach(*df*[!is.na(*df$variable1* + *df$var2* + *etc*,]) - you delete the rows containing NA's in columns var1, var2, etc. The blank space after the column is to include all columns.
   b) subsetting NA's:
      s_!is.na(*y*)|is.na(*x*))
      kruskal.test(*y*[s],*x*[s])
2. f_lm(y~xvariables,data=df,na.action=na.omit) The na.omit deletes rows with NA's.
3. plot(naclus(*df*)) - display variables that tend to be missing on the same subjects.
4. f_tree(is.na(*y*)~*x1+x2+etc*,na.action=na.tree.replace.all) – gives tree with nodes that show the percent of people with variable x that are missing data for y. .all is added at the end for continuous data that has NA's. The na.action is to allow the x's to have NA's. na.tree.replace categorizes continuous variables I into quartiles.
   To limit #nodes of tree, add to end: control=tree.control(nobs=#,mincut=#). nobs is # of observations. NA is default. mincut is stop splitting if fewer than # patients left.
   a) plot(f)
   text(f,cex=#) To label your tree. cex is to size the font.
5. Impute
   a) x1_impute(x1) – replaces NA with median x1. x1 is now permanently changed.
   b) x1_impute(x1,mean) – uses mean x1
   c) x1_impute(x1,#) – imputes with the # you specify
   d) f_lrm(y~x1+x2,subset=!is.imputed(x1)) – excludes subjects with imputed x1
   e) describe(f) – tells you the number of imputed values.
6. transcan: automatically develops customized imputation rules (multiple regression models using spline functions) to allow each predictor to be predicted from all the other predictors. Apply these models to impute variables before fitting the response model.
   a) *imp.models*_transcan(~*x1+x2+x3*,imputed=T)
   *x1*_impute(*imp.models,x1*)
   f_ols(y~*x1+x2+x3*+rcs(x4,4)+rcs(x5,4)) –good for non-monotonic transformations
7. rpart: Recursive partitioning. Splits data like a tree to see how variables as associated with one another. If you type in is.na, it sees how missing variables (when you type in is.na) are distributed so that you can eventually impute. If the variable for a split is itself missing, rpart will split on a related variable.
   a) f_rpart(is.na(y)~x1+x2….) The NA's in y will now be analyzed to see how the x variables are associated with the NA's.
   b) to impute a <u>categorical</u> variable y:
      f_rpart(y~x1+x2….,data=*df*) – will set up missing values to impute for y

p_predict(f,*dataframe*) – this imputes everything for y.
plot(f)
text(f) – for classification tree
phi_(t(apply(-p,1,order)))[,1]
y_impute(y,phi[is.na(y)]) - Now I'm not 100% sure this works.

HMISC REGRESSION MODELS

1. lm – the main linear regression command. f will symbolize the fitted formula
   a) f_lm(*y variable ~ x variable 1 + var2 + etc*)
   b) summary(f) – gives all the partial t-tests, s.e.'s, and t-values. Coefficients given too. R^2 and overall F-test with p-value.
   c) plot(f) – get 6 graphs to show the fit
2. options(contrast=c('contr.treatment','contr.poly')) : this allows dummy variables to be made
   a) f$contrasts - this shows how the dummies are coded.
3. Checking the fit
   a) plot(f)
   b) coef(f) = get the coefficients
   c) fitted(f) or predict(f) = computes yhat
   d) resid(f) – computer residuals. plot(x,resid(f)) – plots residuals vs. x alone.
   e) predict(f,se.fit=T) Original yhats with se for E(y|x)
      1) predict(f,data.frame(x1=1,x2=2,x3=3)): yhat for one person given specific values for each predictor variable.
      2) predict(f,expand,grid(age=c(1,2,3),sex=factor('male',c('female','male'))),se.fit=T)
      This is for FACTOR variables so that you can see yhats for all the men with ages 1,2,3.
      3) x1s_seq(min(age),max(age),length=100)
      pred_predict(f,expand.grid(age=x1s,sex=factor('female',levels=c('male','female'))), se.fit=T)
      pred$fit
      pred$se.fit
      ci_pointwise(pred,coverage=.99)
      ci$upper
      ci$lower
      plot(x1s,pred$fit,type='1',ylab='Yhat')
      lines(x1s,ci$upper,lty=2)
      lines(x1s,ci$lower,lty=2)
      ci_pointwise(pred,coverage=.95)
      plot(x1s,pred$fit,type='1',ylab='maxfwt')
      The x1s is to make an x-axis evenly spaced.
4. Interpreting beta coefficients:
   a) If y = a + b1x1 + b2x2 + b3x3:
   Ho: b1=b2=b3=0; Ha: One of the b's does not equal 0.
   a = y intercept when x1, x2, and x3 are at reference level. So if x1=age, x2=sex('male','female') and x3=low, medium, high; then a is when age=0, sex=male, and x3=low.
   b1 – the value is when the other b values are held constant or at zero. The b values are weighted values to predict y.

   b) y=b0 + b1x1 +b2x2 + b3x1x2 (interaction)
   E(Y|x1=x2=0) = b0
   Ho: no association with x1 = Ho: b1=b3=0 (2 d.f.); Ha: b1 does not =0 or b3 doesn't = 0.
   Ho: no association with x2 = Ho: b2=b3=0 (2df); Ha:b2 does not = 0 or b3 does not = 0.

Ho: no interaction with x1,x2 = Ho: b3 = 0; Ha: b3 does not = 0.
b1 is slope for x1 with x2 = 0. b2 is vice versa.

c) Interactions: C(Y|age,sex) = b0 +b1age + b2(sex=f) + b3age(sex=f)
b0: when age=0 and sex=m (because male is the reference group)
b1: slope for male
b2: y intercepts for female – y intercept for male
b3: difference in slopes for female and male
b1+b3: slope for female
Ho: b1=0. Age is not associated with Y for males
Ho: b2=0. sex is not associated with Y for zero year olds.
Ho: b3=0. Age and sex are additive. Effect of age is independent of sex, or effect of sex is independent of age. Ha: age interacts with sex and vice versa. They are synergistic.
Ho: b1=b3=0. Age is not associated with Y. Ha: b1 or b3 does not =0. Age is associated with Y. Or age is associated with Y for either females or males.
Ho: b2=b3=0. Sex is not associated with Y. Ha: b2 or b3 does not=0. Sex is associated with Y. Or sex is associated with Y for some value of age.
Ho: b1=b2=b3=0. Neither age nor sex is associated with Y. Ha: b1 or b2 or b3 does not=0. Either age or sex is associated with Y.
2 df because: 1. interaction effect. 2. If lines are parallel between sex and age, there can still be an effect.

5. summary(f): all beta coefficients with p-values are when the other values are held constant

DESIGN

1. Make model: f_ols(*y~x1*) – this is like lm
   *dd*_datadist(f) – this starts memory of characteristics of all your variables. Must do this! Or use dataframe instead of f so you don't have to keep doing this step for every fit.
   options(datadist='*dd*') – Holds the fit so that you can plot.
   plot(f,*x1*=NA) – x1=NA allows the x-axis to go the full length(x1).
   points(*x1*,*y*) – to print out data dots.
   The adjusted R^2 takes change and bias into account.
2. Fit a simple linear spline at one point:
   f_ols(y~lsp(x1,#)) – lsp is linear spline. You put it at the x1 value of #.
   x and x' are made. If 4 knots, has 5 parameters (knots+1 = #parameters besides b0)
   In b2x', if b2 doesn't =0, and p-value <.05, you cannot fit a linear model.
3. More than one point in a spline
   a) f_ols(*y*~rcs(*x1*,c(#,#,etc))) - restricted cubic spline
   b) f_ols(y~rcs(x1,#)) - # evenly spaced knots. #=3, there are 3 knots.
   4 knots has 3 parameters (knots-1=#parameters besides b0)
4. Interpretation
   a) f_ols(y~x1)
   b1 is a test of association. Ho: b1=0.
   b) f_ols(y~lsp(x1,#)) Spline
   y = b0 + b1(x1) + b2(x1')
   Ho: x1 is not associated with y = Ho: b1=0.
   Ho: the linear model <u>may</u> fit the data = HO; b2 = 0. But you get a phantom d.f.
   Ha: linear model would not fit the data = Ha: b2 does not = 0. It does NOT mean your nonlinear model fits.

Cannot interpret the b coefficients for knots like in a linear model. The only p-value you can interpret are for variables that don't have knots nor are categorized into levels. The slope to the right of the knot is b1+b2.

c) Using logs: beta coefficients that follow b0 are all relative to b0.
If y= b0 – b1(x1) – b2(x2)
e^{b0-b1} is to get the b1 coefficient.

5. Splines
   a) More knots makes tails not fit well. Therefore, non-robust.
   b) Put knots where data are dense: quantiles.
   c) spearman.test(x1,y,2) – 2 is degrees of freedom for continuous y variable. If y is binary, you use 1. Look at the R^2 to see if there is a big association. We are not sure exactly what is considered a big association, but consider this when making your rcs in a spline model:
   3 knots for .02-.08, 4 knots for .08-.15, and 5 knots >.15.
6. lrm – for binary responses
7. Made quadratic: pol(x,2). Makes x and x^2 into a matrix. pol(x,3) makes cubic.
   f_ols(y~pol(x,2))

	lm	ols
dummy variables	need to state "options" command	traditional coding
NA	must specify na.action=na.omit But can't use resid, fitted, or predict commands now.	Automatically deletes NA's and holds places for them. Can us resid, fitted, and predict.
^ß	no differences	no differences
print	abbreviated summary of model	summary statistics with chisquare, all coefficients, s.e., t stats, p-values (all partial). #NA's, adjusted R^2.
summary	like print is to ols	
anova	F-tests (sequential). No "main effect" really.	Wald ChiSquare, **partial tests**. Test for linearity with splines. With interactions: "total effect" for main+interaction. Pooled test with multi-predictors for splines.
		nomogram, validate, calibrate

8. Data Reduction:
   a) *vc*_varclus(~x1+x2+…,similarity=c('spearman')) -can use spearman to cluster variables.
   plot(*vc*,ylab='Squared Spearman",legend=T,maxlen=16)
   b) How many independent variables is acceptable?
      1) n = sample size. m = effective sample size. p = # predictors
      For continuous data, m=n. For rare outcomes, treat it like a case-control with a 2-3:1 matching of controls to cases. eg: if 3 CA's in 100,000 people, the ratio is 3:1 = 9-12. m=9-12, not 100,000.
      2) If m/p < 15, you need data reduction.
      If m/p > 15, you can probably use all variables you have.
9. Interactions
   a) f_lrm(survived~rcs(age,4)*pclass*sex) – from titanic dataset. This has a 3 way interaction: see how age interacts with pclass & sex, how pclass interacts with sex & age, and how sex

interacts with age&pclass. "Third ordered interaction"

b) page(anova(f)) – gives Wald stats for total association, total nonlinear + interaction, total interaction of 2nd and 3rd order, and total 1st 2nd 3rd order interaction.

c) If only need to look for 2nd order: f_lrm(survived~rcs(age,4)+pclass+sex)^2)

d) plot(f) – look at the conditions at bottom of graphs to see how to interpret.

1) plot(f,sex=NA,pclass='1st') – how sex affects 30yo in 1st class. (30 is age default here)

2) plot(f,sex=NA,pclass='1st',age=40) how sex affects 40yo 1st class.

e) plot(anova(f)) – see how varibes interact and affect Y. Chisquare on x-axis.

10. plogis: converts log odds to Probability

plogis(#) – eg. predict(f,dataframe(age=21,pclass='1st',sex='female')) – you get an odds ratio value, then plug it into plogis(#).

plot(f,fun=plogis)

11. specs(f) – specification of the design. Give the assumptions – linear, nonlinear, splines, etc. df's, cuts and dummy variables. If you add, long=T to the end, you get even more info.

12. page(summary(f)) – get estimates of effect on predictive values changing x from 1st to 3rd quartiles.

plot(summary(f)) gives a very nice graph.

13. Make a function our of your model for convenience sake.

*g*_Function(f)

This will have a default/reference it looks at. Type g to see it.

plogis(g( )) – gives probability of your model.

plogis(g(sex='female')) – can specify what levels to look at.

14. Singularity

ols(……..tol=1e-14) will cure this problem.

15. Test for normality

*r*_resid(f)

Go to menu → 2Dplot →Qqnormal with line. Click on r.

BOOTSTRAP

1. summary(bootstrap(*x*,median)) – nonparametric 95%CI for the population median of x.
2. bootstrap(x,mean) Estimate standard deviation of a sample mean for x.
3. Validation: how well the model will fit with future data.

a) Split sample validation: training sample for model development; test sample for validation. Get apparent R^2, then freeze the beta of the test sample to get the Validated R^2, which is unbiased.

b) 5 fold cross-validation is a good way to do split sample validation.

c) Bootstrap is best.

f_ols(y~x1+x2+x3,x=T,y=T). You need the x=T and y=T to keep original data retained in order for bootstrap.

OR f_update(f,x=T,y=T) does it for you if f was already made.

validate(f,B=150) – B=150 means to bootstrap 150x on model f.

In the results, you got your R^2 boot apparent – R^2 boot original = optimism. Do 150x.

R^2 corrected = R^2 apparent – mean optimism.

C = Concordance Probability (ROC) which is used for Discrimination: ability to predict one outcome vs. another for individual patients. C > .80 is good.

Dxy = 2(C-.5). Dxy=1 means predictors are perfectly discriminating.

Brier (B) = mean ((phat-y)^2). Want a LOW score and is used to compare 2 models to see which is better.

4. Calibrate: see accuracy of predictions. You compare your bootstrap to the 45 degree line

*cal*_calibrate(f,B=150)
plot(cal) – doesn't show how well it fits but really what it is fitting (shape of effects).
Mean absolute error – want it to be less than .05 or .1. Mean of differences between points on bootstrap and apparent.

5. Nomogram
nomogram(f)
nomogram(f,fun=plogis,funlabel='Probability') – plogis takes the antilog of the odds ratio to give probability.

Logistic Models

1. plogis(#) – takes a log odd and converts it into a probability
2. qlogis: takes proportions and converts them to log odds
plsmo(*x*,*y*,group=*stratifier*,fun=qlogis) – graph of log odds
3. To analyze binary output data:
a) options(drop.factor.levels=F) – do before you perform cross-tabulation on binary data
summary(b*inary y ~ binary x + categorical x* ,method='cross') – stratifies the 2 x variables into a table with number of occurrences and the proportions.
b) Make odds and log odds
1) summary(*binary y ~ binary x*) – to make 2 x 2 table of proportions (no #occurrences)
qlogis(proportion #) = log odds
exp(log odds) = odds
or you can take *proportion # / (*1-*proportion #*) = odds.
2) table(binary x, binary y) – get table with 0 & 1 columns (1 means event occurred).
odds = # 1 events/# 0 events
OR = ad/bc = odds level 1/ odds level 2
3) table(categorical x, binary y) – k>2. Can make odds for each level.
4) Make table of all x variables: summary(binary y~ x1 + x2)
4. lrm(binary y~binary x)
a) In this case with binary x, the beta 1 is = log odds ratio.
lrm(response~sex):
If beta1 reads sex=male 1.69. The log odds ratio of male:female is 1.69.
With no other variables, the beta0 is log odds (not OR) for female (the reference).
beta0 + beta1 = log odds for males.
exp(# * logOR) - # is change in unit. Multiply this by the log OR. eg: The OR is on age. You want to know the change in odds when you increase age from 30 to 35. exp(5*logOR) = 2.5, then you have a 2.5x increase risk (150%).
b) f_lrm(y~x1+….)
Top line: LR chi-square is the overall test of association. If there is only one x variable in model, use this p-value as well for the p-value of x.
c) f_lrm(y~x1+rcs(x2,4)….)
Cannot interpret the partial tests. Only LR chi-square on top line is interpreted for overall test of association. What you need to do is:
anova(f): this will give the partial tests of association and partial tests of nonlinearity.
You also get overall tests of nonlinearity and interaction.
plot(anova(f)) – gives strength (ranks of importance) of each predictor.

Proportional Odds Logistic Regression Models

1. Used in ordinal outcomes. (0,1,2,3….k)

Prob(Y>=j|X) = 1/(1+exp –[alpha j + Xbeta]). You get as many alphas as levels of your outcome variable (except for reference group). j is each level of the oucome (0,1,2,3….k)

2. Test for ordinality of Y with each predictor, x.
   plot.xmean.ordinaly(y~x1+…….)
      Categorical variables must be specified: sex='male', etc.
      You want the plots to be consistently increasing or decreasing in order to say it is ordinal. (Y variable is actually on the x-axis!)
      The dotted line should match somewhat closely with solid line to say the PO holds.
3. f_lrm(y~x1+x2….)
   a) Output is in log odds.
   b) OR's are constant over j (level 0 to 1 to 2…) in assumptions for PO's.

Survival Analysis

1. Proportion of patients having censored times: table(*y*) – y is outcome variable like death
   0 column is # alive, and 1 column is # dead. Take # in 0 column and divide by total n.
2.Graph of censored f/u's
   ecdf(*time to death variable*, group = *death variable*, q=.5)
   ecdf(*time to death variable*, group = *death variable* == 0) shows both censored and dead
3.Survival data
   S_Surv(*time to death variable, death variable*)
   summary(S) – gives your mean and quartiles for time until event.
4. Survival Curve Figure (Kaplan-Meier)
   S_Surv(*time to death variable, death variable*)
   f_survfit(S)
   survplot(f, conf='none',n.risk=T) – number at risk is added on and CI's are removed.
   abline(h=.5,lty=1) – to draw a line for 50% survival on figure.

   b) Stratified by class
      f1_survfit(S~*x1*) – x1 is a categorical variable
   c) Stratify by continuous variable
      f2_survfit(S~cut2(*x2*,g=#)) – can make x2 (like age) into tertiles, quartiles, etc.

Simple Cox Model

1. S_Surv(*d.time, death*)
2. f_cph(S~*x1*) stratifies for variable x1, like sex or age.
   Score and Score p-values are given when you type f. No y-intercept.