Notes on R

Created by Brent Wagenseller, a GT Student

[Comments](#)
[Running R](#)
[Installing Packages](#)
[Saving screen output to a file](#)
[Vectors](#)
[Lists](#)
[Arrays](#)
[Saving / loading your current session](#)
[Importing data into data frames from CSVs](#)
[Importing data from SPSS files](#)
[Importing data from MySQL](#)
[Importing data from generic ODBC SQL](#)
[Exploring and understanding data](#)
[Boxplots / box-and-whisker plots](#)
[Histograms](#)
[Variance / Standard Deviation](#)
[Examining categorical variables](#)

Comments

"
multiline comments aren't natively supported in R - instead, wrap the entire
line in double quotes like this. R will actually read the comment BUT it won't do anything with it
"
#A single-line comment is written like this - with a # at the beginning of the comment

##Running R
"

Running R

Ubuntu
To run R in Ubuntu (after installation), simply type this at a prompt
(capitalization is important): R

To quit, type: q()

Windows
For Windows, when R is installed a program 'RGui' is also installed - simply open that program.
I assume you can type 'R' in at a command line on windows as well but I am not sure.

To quit R, type this at the R command prompt: q()
"
###Running R

Scripts
#to run a script in R, note its path and on a command line run
source("my/pat/to/the/R_file.R")

Installing Packages

###Installing packages
#Sometimes you need to load functions from a remote mirror / repository (R will list mirrors for you if need be)
#Note that you either have to be root for this (in UNIX) OR have to launch the current window as 'administrator' (in Windows)
#OR be OK with R storing the package localls (so all users will have to download the package if they wish to use it); R give you the choice when you load a package / library

#this installs the “foreign” package, which allows for the importation of SPSS data in its native format
#only has to be done once (as root; if installed as a user it must be installed once per user)
#Note the double quotes may not be necessary (sometimes they are, sometimes they arent) but Google recommends them in their R standards
install.packages("foreign")

#this is the way to load a package / library from your local machine (once installed)
#this loads the example “foreign” library
library("foreign")

#This is used for when you actually create a function or package – this is used instead of library(“foreign”)
#This can be used in place of library(“foreign”) for the command line too, but most people just use library(“foreign”)
require("foreign")

Unload Package
#This unloads – but does not delete – the package 'foreign'
#By default, all installed packages are removed when R quits
detach("package:foreign", unload=TRUE)

#Checks for updates; this is unfortunately a manual thing. Do this periodically!
update.packages()
######Timing Code
#create a start time variable
startTime <- proc.time()
proc.time() - startTime#displays to screen; elapsed time is the time it took to run

#conversely, you can run code inside system.time() and the output is the time
system.time(a <- c(1,2,3))

Saving screen output to a file

#get working directory
getwd()

#set working directory
setwd('/home/myHomeDirectory')

#open a file for writing
sink("sink.txt")

```
#modify a variable
c <- c(1,2,3,4,5)

#print to file (remember, output goes to file and not the screen)
c

#return the output to the screen
sink()

#delete the file
unlink('sink.txt')
```

Vectors

```
"
The fundamental R data structure is the vector, which stores an ordered set of values called
elements.
Vectors can only contain the same type - numbers OR character strings but not both.
Vector types:
 integer
 numeric(decimal)
 character
 logical (true or false)
Additionally any vector can have:
 NULL (absence of any value)
 NA (missing value)
"
#Vector examples:
#note: the c() function is read 'combine' and combines elements to make a vector
#here, the vector 'subject_name' is created and three values are stored to it
#note the use of '<-' - this is used in place of '=' for assignment in R for the most part
#also note that simply writing a variable name on a line in R will print it to the screen
subjectName <- c("John", "Jane", "Steve")

#to reference specific values in the vector you can reference them by a range of numbers
corresponding to their position like so
subjectName[2:3]

#you can exclude certain positions with a minus sign:
subjectName[-2]
```

#you can also construct a logical vector on the fly to tell R which elements to include
subjectName[c(FALSE, TRUE, TRUE)]

#'nominal' data is features that represent a characteristic with categories of values (for example, cars:ford/honda, meats:chicken/veal/fish/beef, etc - basically anything not numbers)
#R provides a data structure known as a 'factor' to store nominal data
#its important to use factors for string-based categories as R can efficiently store them with an internal ID; if a factor isnt used it has to save the actual string for each row, also some algorithms REQUIRE nominals to be used in factors
#Example
gender <- factor(c("MALE", "FEMALE", "MALE"))

#factors are also useful if there is another data type that is not represented in our data - yet - but we want to acknowlede exists. For example:
bloodType <- factor(c("O", "AB", "A"), levels = c("A", "B", "AB", "O"))
#note that 'B' is not used but still noted as a possible type

Lists

#Lists are similar to vectors but do NOT require each element to be of the same type (so character strings and numbers can coexist in a list)
#Lists are used often because of this
#as an example, lets create a list - subject1 - and populate it with some of the variables we already created
subject1 <- list(fullname = subjectName[1], gender = gender[1], bloodType = bloodType[1])
#subject1 is now a list! be warned though: this is an actual list; in this instance we just use one record but a list can hold several records

#note that if we wanted to get a specific column in the list we just created, its the variable and column separated by a $ like so
subject1$fullname

######Data Frames
#this is by far the most important data structure in R
#similar to a spreadsheet or database since it has rows and columns
#a data frame is like a list of vectors or factors, each having exactly the sname number of values

#lets create a data frame
#note: patient.data is a variable name below - its NOT a function or an object. the '.' is not treated as an object in R variables but simply another character

#note the use of 'stringAsFactors = FALSE' - this tells R to NOT convert strings to factors; mainly because the name is definitely not a 'category' and shold not be treated as such, you CAN go back later and specify c
patient.data <- data.frame(subjectName, gender, bloodType, stringAsFactors = FALSE)

#you can eliminate a specific row by using a construct and listing only the columns/vectors/frames you want to list
#whats cool about this is information is extracted by column name
patient.data[c("subjectName", "bloodType")]

#you can treate the data frame much like a multi-domensional array in other languages; use the row number then the column number
#here is how you can extract the gender of the third person
patient.data[3,2]

######Matrixes
#A matrix is a data structure that represents a two dimensional table
#most often used for mathematical operations so they usually only store numbers, but they can store characters as well

#creates a matrix of a/b/c/d with two rows; how this will work is the first row will be a and b and the second row will be c and d
#note that R loads the data into a COLUMN first, NOT across rows
myMatrix <- matrix(c('a', 'b','c','d'),nrow=2)

Arrays

#apparently very similar to a matrix yet while a matrix just has rows and columns, an array has rows, columns, and several other layers
#arrays are not used often and are apparently somewhat advanced - so they are mentioned but not used in my training documents

Saving / loading your current session

#since R ususally isnt run by scripts sequentially - and is really command-line driven - there
#is a method for saving variables for a later time; to save specific data frame / matrices / arrays / lists / frames / etc:
save(patient.data, myMatrix, file = 'learnR.RData')
#note the variables were listed first, then a file was assigned

```
#to laod this data later back from that file:
load("learnR.RData")

#also note the save.image() command saves the ENTIRE session to a file called .RData
#By default, R looks for this file when loading initially to restore the previous session
```

Importing data into data frames from CSVs

```
#While possible to import excel files, they are special packages and they do not always work; instead, use CSV
#R converts missing data to “NA”
#For windows, the normal backslashes for directories will not work; instead you either have to use double backslashes OR use front slashes a la UNIX

#read in a csv file into the dataframe 'sn.csv' and signify that the first row contains column headers (otherwise R names them V1 V2 V3 etc etc)
#note the 'sn.csv' is NOT a function call to csv; instead, its a fancy way to name a variable; the entire variable name IS 'sn.csv' – dots can be used in variable names and Google suggests it to separate words in variable names
#On the other hand, 'read.csv' is the function that reads the CSV file into the variable
sn.csv <- read.csv("/home/me/social_network.csv", header = T)

#this is the 'structure' function, and it transposes the columns and rows and then prints them to the screen
str(sn.csv)

#write a data frame to a csv file
write.csv(patient.data, file = 'patientData.csv')
```

Importing data from SPSS files

```
#Very common statistical package used by researchers
```

#SPSS is very much like an Excel spreadsheet
#In SPSS, its common to see a string column immediately followed by a number column that represents the string column (redundant)
#for example, there may be a column that is strictly limited to male/female and then the next column would be a 1(for male) or 0 (for female).
#For the column that is a number, you can additionally place a 'label' for each numeric value (separate from the associated string column)
#A program that can edit SPSS files is 'IBM SPSS Statistics Data Editor'
#Lynda suggests saving it as a CSV file and then importing it using the command 'read.csv'
sn.spss.f <- read.spss("/home/me/social_network.sav", to.data.frame=T, use.value.labels=T)
#to.data.frame=T means we want to convert this to a data.frame (not sure why you wouldnt do this)
#use.value.labels=T means use the associated labels with all numeric columns that have labels

Importing data from MySQL

#Its possible to interact with a MySQL Database in R
#more info at: www.r-bloggers.com/mysql-and-r/
#Installation in Ubuntu - Before you run the install command in R, you need to install a package in Ubuntu itself (as root) via: apt-get install libmysqlclient-dev
#Installation in Windows - when I did this in windows I simply loaded the package (no extra steps required)

#In R itself: After following the steps above for your platform, run the following in R:
install.packages("RMySQL")

#load the MySQL library
library("RMySQL")

#this is the connection string
#user/password – self-explanatory
#host = the IP (or hostname) of the MySQL database
#dbname – the database name / schema name
con <- dbConnect(MySQL(), user = 'userName', password = 'userPassword', host = '127.0.0.1', dbname='mySchemaName')

#This writes a data frame to a table
#conn – the connection string
#name – the name of the table. Note if the schema is different than the one supplied on the connection string you can reference it here as you would in MySQL a la Schema.TableName
#value – the data frame variable

dbWriteTable(conn=con, name = 'Test2', value = patient.data)

#This loads a tables contents into a data frame ('fuzzy' is the data frame that will hold the new data)
#conn – the connection string
#name – the name of the table. Note if the schema is different than the one supplied on the connection string you can reference it here as you would in MySQL a la Schema.TableName.
#Note: Views can be used – so if you want to massage data in a table, simply create a view and use that in place of the table in R!
fuzzy = dbReadTable(conn=con,name='Test2')

#if you wish to write a query, this is how it's done
#preps the 'sql' variable for a SQL query; note varId and varName are not set in this example but they should be set prior to this point
sql <- sprintf("SELECT * FROM MySchema.MyTable WHERE ID = %d AND myName = '%s'", varId, varName)
#sets the data frame 'rs' to the query results
rs <- dbGetQuery(con, sql)

#disconnects the database
dbDisconnect(con)

Importing data from generic ODBC SQL

#This will work for Oracle, MySQL, Microsfot SQL, SQLite, or PostgreSQL (possibly HP Vertica)
#The instructional book didnt get into connection strings - which are needed. however the connection string may be
#able to be pulled from a perl/python/php script (I havent tried it).
#the package 'RODBC' is needed
install.packages("RODBC")
library("RODBC")

#save your connection string here
my_dsn <- "my awesome connection string"

#create a connection
myGenericConnection <- odbcConnect("my_dsn")

#IF the connection requires a separate username and password, use this method instead
myGenericConnection <- odbcConnect("my_dsn", uid = "myUserName", pwd = "myPAssword")

#create the query and pull into data frame 'newPatientDataFrame'
patientQuery <- "SELECT * FROM MyTable WHERE alive = 1"
newPatientDataFrame <- sqlQuery(channel = myGenericConnection, query = patientQuery, stringAsFactors = FALSE)

#close connection
odbcClose(myGenericConnection)

Exploring and understanding data

#This section coincides with the lesson in 'Machine Learning in R' by Brett Lantz, section 'Exploring the structure of data'
#on p43. The whole point of R is data analysis; the code is important, but knowing how to USE R is also important

#this section explores data surrounding used cars
#pull in the data to a data frame
usedcars <- read.csv("usedcars.csv", stringsAsFactors = FALSE)

#A good first start is looking at the summary statistics of the data - the str() (structure) function is good at getting a first look
str(usedcars)

#We note that this seems to be data on used cars - but we need to know more
#We pick a variable - year - and explore it using the 'summary;' command
summary(usedcars$year)
#Year could be deceiving - is it the year the car was made or the year the car was sold to the dealership?
#the summary data shows the minimum is 2000, so its a safe bet its the year the car was made

#now take a look at the summary of two more columns - price and mileage
summary(usedcars[c("price", "mileage")])

#now lets look at some other numbers. Again, read chapter 2 of 'Machine Learning with R' by Brett Lantz to get a
#better understanding of how to use this data

# the min/max of used car prices
range(usedcars$price)

# the difference of the range
diff(range(usedcars$price))

# IQR for used car prices; IQR is the 'interquartile range', which is Q3 - Q1. This is helpful to determine the spread of data
IQR(usedcars$price)

# use quantile to calculate five-number summary; without parameters set, this returns the 5 basic quanties (that is, 0/25/50/75/100)
quantile(usedcars$price)

# get the 1st and the 99th percentile
quantile(usedcars$price, probs = c(0.01, 0.99))

If we wanted to find the quintiles of the data (cutting it into 5 groups with an equal number of data points per group)
#we can use the seq() (sequence function) which generates vectors of evenly-spaced values
quantile(usedcars$price, seq(from = 0, to = 1, by = 0.20))

Boxplots / box-and-whisker plots

#This is how a boxplot is created in R
#Boxplots are useful in determining how clustered the data is
#Boxplots can be highly customized - type ?boxplot for more info
# boxplot of used car prices and mileage
boxplot(usedcars$price, main="Boxplot of Used Car Prices", ylab="Price ($)")
boxplot(usedcars$mileage, main="Boxplot of Used Car Mileage", ylab="Odometer (mi.)")

Histograms

#This is how a boxplot is created in R
#Histograms are useful in determining how clustered the data is
#unlike boxplots, histograms do not divide the data into 4 (for quartile) equal bins; instead, they use a constant bin size and report how many variables / data points fall in that bin
#Histograms can be highly customized - type ?histograms for more info
histograms of used car prices and mileage

hist(usedcars$price, main = "Histogram of Used Car Prices", xlab = "Price ($)")
hist(usedcars$mileage, main = "Histogram of Used Car Mileage", xlab = "Odometer (mi.)")

Variance / Standard Deviation

variance and standard deviation of the used car data
#see the notes on variance and standard deviation in chapter 2 of 'Machine Learning with R'

#note that R uses the sample variance (n-1) and NOT the typical population variance (n) in the variance and standard deviation equations.
#This shouldnt effect the outcome too much provided the dataset isnt too small (5-1 has much more of an impact than 500-1)

var(usedcars$price)
sd(usedcars$price)
var(usedcars$mileage)
sd(usedcars$mileage)

Examining categorical variables

one-way tables for the used car data - tells us the count of each category
#a 'table' is not like a table in a database - rather, its a way to count the different number of occurances in a categorical variable data set
table(usedcars$year)
table(usedcars$model)
table(usedcars$color)

# compute table proportions - what % of the table represents Toyota?
model_table <- table(usedcars$model)
prop.table(model_table)

make the above data more recognizeable as a percentage by multiplying it by 100, then round the data to 1 decimal

```
color_table <- table(usedcars$color)
color_pct <- prop.table(color_table) * 100
round(color_pct, digits = 1)

# scatterplot of price vs. mileage
plot(x = usedcars$mileage, y = usedcars$price,
    main = "Scatterplot of Price vs. Mileage",
    xlab = "Used Car Odometer (mi.)",
    ylab = "Used Car Price ($)")

# new variable indicating conservative colors
usedcars$conservative <-
 usedcars$color %in% c("Black", "Gray", "Silver", "White")

# checking our variable
table(usedcars$conservative)

library("gmodels")
# Crosstab of conservative by model
#chisq = TRUE means the chi-squared test for independence is included; its in the last line of
output, p = number.
#in this case p = .92591; since its high it means there probably isnt a correlation between model
and conservative colors
CrossTable(x = usedcars$model, y = usedcars$conservative, chisq = TRUE)
```