Introduction to R for Choice Modelers

Sawtooth Software Conference 2013
Dana Point, CA

Steven Ellis[1], Chris Chapman[2]
elliss@google.com, cchapman@google.com

[1]Google Social, Mountain View
[2]Google AdSense, New York

October 2013

Topics

Why R? Why not R?

R is where almost everything new in statistics happens first.
It is rapidly becoming a must-know for top researchers & analysts.

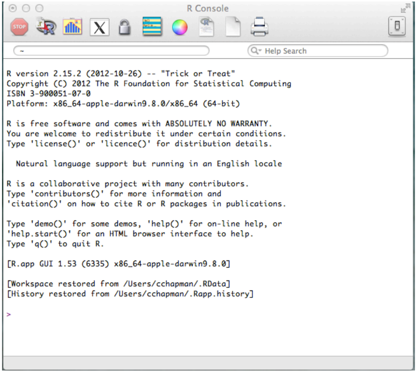
R makes possible much that is impossible in traditional stats software.
R is great for emerging methods, replicable models, iterated analyses.
OTOH it can be less efficient for simple, ad hoc analyses.

R has a steep learning curve.
We are here to force you up the steepest part of the curve.

Even after 16 years (Chris) of S & R, there's **always** *more to learn!*

Running R

R is not really a "statistics program." It is "*a language and environment for statistical computing and graphics.*" (http://www.r-project.org/)



We introduce the **language** and **environment** and what those mean.

This tutorial's .R file

All the code here is provided in a separate .R file.

Inside R: `File | Open Document ...`
Navigate to: `sawtooth-Rtutorial2013-exercises.R`

In R, the convention is that source code files end with "`.R`"

This tutorial is hands-on.
We strongly recommend **typing** *every command we show.*

R as a calculator

The command line is an interactive programming language interpreter.

At the simplest, this works as a handy scientific calculator:

```
123+456
exp(2)
10^6
```

And of course it understands variables:

```
x <- 5   # type: < -
pi*x^2
```

Tip: use cursor up/down to view, edit, and reuse commands.

On Assignment: A Discursion

R understands "=" in three different senses:

- Variable assignment: <−
- Comparison: ==
- Function parameters: =

Always use "<−" for assignment!

Although "=" mostly works for assignment (not comparison), it is regarded as ugly (and as signaling a Fortran programmer!)

Vectors

The R language is highly optimized and designed for working with data.

John Chambers, the designer of its predecessor S (and a contributor to R), won the ACM Software System Award for creating the language.

We start to see glimpses of this in the native support for vectors:

```
1:10
x <- 1:10
x
y <- pi*x^2
y
seq(from=11, to=101, by=10)
```

And math on them:

```
sum(x)
mean(y)
```

Help!

The first place to look for help with R is R:

```
?sum    # just the name
?"+"    # or name with quotes
```

The second place to look is web search.
Google understands "R" in many contexts:

```
R assignment operator
```

There are 1000s of sites and books with useful R information and code.

One especially useful site is CRAN, and the CRAN Task Page:
http://cran.r-project.org/web/views/

Matrix math

In addition to vectors, R can work with matrices

```
xy ← cbind(x, y)
xy
(xy ← cbind(x, y))       # shortcut
(yx ← t(xy))
```

. . . and do matrix math

```
xy - 10    # elements are recycled
xy - 1:10  # watch out for the order!
# between matrices
xy + xy
xy * yx    # doesn't work the same way!
xy %*% yx
xy * xy    # very different
```

. . . plus all the expected matrix functions

```
det(xy %*% yx)  # essentially zero. why? (hint: think volume)
```

Inspecting Objects

What objects do we have?

```
ls()
```

What can we learn about them?

```
x
length(x)
dim(xy)
```

Other functions help when the objects get larger:

```
x <- 1:10000; y <- pi*x^2; xy <- cbind(x, y) # make xy large
ls()
dim(xy)
head(xy)
tail(xy)
str(xy)
```

Indexing Part 1

Vectors can be indexed by position:

```
x[4]
y[4]
```

Higher dimension objects index on multiple dimensions:

```
xy[3141, 2]     # key concept:  [ ROW, COLUMN ]
```

Leaving a dimension blank means *"give me all of that"*.

```
xy[4, ]
xy[, 2]   # oops, too many!
tail(xy[, 2])
```

Indexing by Range

We usually want a subset rather than a single element or dimension. R's vectorization comes to the rescue:

```
x[4:6]
y[c(1,3,7)]
xy[11:20, ]
```

This can use variables and vectors:

```
(z ← seq(from=1, to=100, by=13)^2)      # why extra '( )' ?
xy[z, ]
```

And can even index by dimensional arrays (invaluable on rare occasions)

```
(zz ← cbind(z, c(1,2,2,1,1,1,2,2)))
xy[zz]          # no commas. zz is a 2D "list of coordinates"
```

Pop Quiz

1. How many centimeters is the circumference of the Earth?
(assume circumference to the nearest 1000 miles. No Googling!)

2. What is the **y column value** in `xy`, where `x` column value is same integer as this year ?

3. What is the **row number** of `xy` where `y` is first larger than 10000 ?

Pop Quiz

1. How many centimeters is the circumference of the Earth?
(assume circumference to the nearest 1000 miles. No Googling!)

```
25000 * 5280 * 12 * 2.54

# we also accept Fermi's approximation: ~3000 mi NY-LA =>
# 1000 miles per time zone * 24 time zones * ...
1000 * 24 * 5280 * 12 * 2.54
```

2. What is the **y column value** in `xy`, where `x` column value is same integer as this year ?

```
xy[2013, ]     # not ideal (why not?) but works here
```

3. What is the **row number** of `xy` where `y` is first larger than 10000 ?

```
xy[50:70, ]    # we'll see a better way!
```

Negative Indexing

Negative indexing: everything *except* that:

```
z[-4]
xy[z[-4], ]
```

This is often used especially to remove a bad data point:

```
dim(xy)
xy <- xy[-5280,]
dim(xy)
```

Be careful not to do it twice in a row by mistake!

Data Frames

Data frames are the R equivalent of a data set or table.

Rows = observation units (often "respondents")
Columns = observations (variables) and other data

```
str(xy)

xy.df ← data.frame(xy)
str(xy.df)

head(xy.df)
```

Why not just use matrices?
Matrices comprise items with *one data type*. Data frames can mix them.
And there are various other optimizations and features.

Aside: What's in a name?

R programmers use a variety of naming conventions.

The two most common ones are:

dotted names: `xy.df`, `data.frame()`
camel case: `stringsAsFactors`, `BayesFactor`

A period (".") has no function in an R name; it's just a character.

Find a style that works for you.
We recommend **dotted names**, **type suffixes** to clarify (".df", ".mat"), and **descendent naming** (`xy` → `xy.df` → `xy.df.sub`).

Note: names *starting* with '.' are *hidden* objects by default. More later.

Indexing Part 2: Data Frames

Let's make the data set smaller for test purposes:

```
xy.df ← data.frame(xy[1:20, ])   # easier to work with
str(xy.df)
```

Data frames have *names*.

```
names(xy)
names(xy.df)
names(xy.df) ← c("Store", "Sales")   # set new names
```

They can be indexed in several ways:

```
xy.df[, 2]          # by dimension and integer index
xy.df$Store         # by name in object (list) reference style
xy.df[, "Sales"]    # by name in dimensional style
```

Adding data to data frames

You can add data in several ways. Add *rows*:

```
xy.df ← rbind(xy.df, xy.df[1:10, ])
```

Add *columns*:

```
xy.df ← cbind(xy.df, xy.df[ , 2])
head(xy.df)
```

Add *variables*:

```
names(xy.df)[3] ← "Sales2"          # name a column

xy.df$random ← rnorm(nrow(xy.df)) # add & name
head(xy.df)
```

Out of scope here: R can also `merge()` *and use hashes.*

Boolean indexing 1

You can index by logical expression:

```
xy.df[xy.df$Store==10, ]
```

What is going on there?

```
xy.df$Store == 10          # boolean vector

ind ← xy.df$Store == 10
ind
xy.df[ind, ]
```

Logical expressions can be combined:

```
xy.df[xy.df$Sales < 100, ]
xy.df[xy.df$Sales > 100 & xy.df$Sales < 500, ]
xy.df[xy.df$Sales < 50 | xy.df$Sales > 1000, ]
```

Watch out for `&` (vectorized for indices) vs. `&&` (mostly for `if()`).

Subsets

`subset()` is easier than complex vector/data.frame names. Compare:

```
A:  xy.df[xy.df$Sales < 50 | xy.df$Sales > 1000, ]

B:  subset(xy.df, Sales < 50 | Sales > 1000)
```

Use parentheses and spaces liberally to improve readability:

```
subset( xy.df, (Sales<50) | (Sales>1000) )
```

`subset()` and variables are great for dynamic data scoping:

```
low ← 50; high ← 1000
new.xydf ← subset( xy.df, (Sales<low) | (Sales>high) )
new.xydf$Sales
```

One better answer to the pop quiz

3. What is the **row number** of `xy` where `y` is first larger than 10000 ?

Instead of:

```
xy[50:70, ]
```

Try this:

```
head(xy[ xy[,"y"] > 10000,])
```

One better answer to the pop quiz

3. What is the **row number** of `xy` where `y` is first larger than 10000 ?

Instead of:

```
xy[50:70, ]
```

Try this:

```
head(xy[ xy[,"y"] > 10000,])
```

Technically `x` is not a row number here, but you get the point:
Let R do the work to find things for you.

For a more precise answer: use `which()` to find elements:

```
which(xy[,"y"]>10000)
```

And use its results just like any other vector to find the first occurrence:

```
which(xy[,"y"]>10000)[1]
```

Functions: Seeing stars ... no, parentheses

We keep seeing `which()` and `data.frame()`, etc: functions.

Functions are easy to write in R, and just as privileged as built-in ones.

```
my ← function(x) {
  print("Hi, Mom!")
  print(x)
}

pi
my(pi)
```

They generally handle vectors and indexing without problems.

```
my(xy[1:5, "y"])
```

You can see the code of any function written in R by typing its name:

```
my
data.frame
```

Functions are objects

R is similar to Lisp and Scheme in having functions as objects.

```
my
my(my)
my(my(my))
```

The result of a function can be assigned like any other value:

```
my2 ← function(x) {
  my(x)
  return(x^2)
}

my2(2:4)
yy ← my2(2:4)
yy
```

Don't be afraid to write functions when it makes life easier! Just be careful not to overwrite system objects (use "?" first).

R is a complete language

If you've programmed before, the R language will be straightforward:

```
if (BOOLEAN) { STATEMENTS }
if (BOOLEAN) { STATEMENTS } else { STATEMENTS }

while (BOOLEAN) { STATEMENTS }

for (VAR in SEQUENCE) { STATEMENTS }
```

Identifiers are case sensitive. Indentation is aesthetic (non-functional).
Commands may be separated with `;` but end-of-line is preferred.
A function's return value is the final value (or inside "`return()`").

Google has written a complete style guide for R:
http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

Clean up time!

Keep your workspace clean!

```
ls()
rm(x)
```

Iterate as needed.

```
ls()
rm(x, y, z, xy, xy.df, new.xydf, ind, low, high)
# warning about x, but not error

ls()
rm(my, my2, yy, yx, zz)

ls(all.names=TRUE) # show hidden names
```

Q&A

Q&A

Time for a break?

Topics

Reading and writing data

You have a CSV. How do you read it into R?

```
my.object ← read.csv("filename.csv")
```

Three things you will often want to add:

```
my.object ← read.csv("filename.csv",
    stringsAsFactors=FALSE, header=TRUE, row.names=1)
```

Tip: R is picky. Clean up before importing, and check it after loading:

```
head(my.object)
str(my.object)
```

Write it out the same way:

```
write.csv(my.object, "filename.csv")
```

There are options for Tab-separated files as well (see `?read.csv`)

R memory objects

The fundamental commands are `save(x, file="filename")` and `load("filename")`.

```
hi ← 'Hello, world!'
hi
save(hi, file="hello.Rdata")

hi ← 'Make me a sandwich!'
hi
load("hello.Rdata")       # silently overwrites objects!
hi
```

Write complex objects to disk for backup and to share them with others.

R sessions

All objects in the local workspace can be saved with `save.image("filename")`

```
save.image("mywork.Rdata")
ls()
rm(list=ls())   # caution: deletes all objects!
ls()

load("mywork.Rdata")
ls()
```

When you exit R, it can automatically save your session as ".Rdata", and reopen it when you start again.

That is a convenience but it's not a backup plan! And it creates workspace pollution.

Pointers: Other file type and databases

R can work with a variety of files types and databases.
That is out of scope for this tutorial, but you might see:

- Package "foreign" : import data from SPSS, Excel, etc.
- Package "RODBC" : connect to ODBC with SQL query
- Package "RMySQL" : similar for MySQL

Example from RMySQL Help file (not run here):

```
drv <- dbDriver("MySQL")
con <- dbConnect(drv, "usr", "password", "dbname")
res <- dbSendQuery(con, "SELECT * from liv25")
data <- fetch(res, n = -1)
```

Best practices

- Do: Experiment a lot at the command line
- Do: Make all work you "keep" replicable in a .R file
- Do: Start replications at the point of loading a raw data file
- Do: Save a session object if you want a *backup-while-working*
- Do: Clean up your sessions regularly

- Don't: Accumulate 100s or 1000s of environmental variables
- Don't: Count on auto session save

- Maybe: Start every R session from scratch

Clean up regularly and learn to make work reproducible:

```
rm(list=ls())
```

Topics

Get a Data Set We'll Use

We'll use a **fake** data set (reasonable structure but unreasonable data!) It describes sales of "bottles" by store/week with covariates.

Covariates include store level (neighborhood income) and per store/week (sales of cheese, price, shelf talker yes or no, visitors).

```
# some packages we'll use
install.packages(c("car", "digest", "psych", "MCMCpack",
                   "xtable"))

setwd("/Users/cchapman/Documents/Work/R")    # <== CHANGE
load("Rtutorial-2013-storesales.Rdata")

head(store.sales)
summary(store.sales)

library(car)
some(store.sales)
```

Overview of Plotting

R plotting is as complex and diverse as the rest of R:

- Base plotting commands (`plot()`, `hist()`, `boxplot()`, etc.)
- Specialized plotting packages (*ggplot2*, *lattice*)
- Plotting as sidelines to other packages (e.g., *Hmisc*, *car*)
- Cookbook style examples
- Build your own!

What's missing? *Menus, drag & drop, and clicking!*
Look instead for examples and replicable code in books and online.

Univariate: Histogram

Frequency plot with observation binning: `hist()`

```
hist(store.sales$price)
hist(store.sales$visitors)
hist(store.sales$visitors, breaks=50)
hist(store.sales$visitors, breaks=50, prob=TRUE)
```

Univariate: Density

The **density()** function returns samples of, um ... a density function (continuous pdf, integrable to 1.0).

```
density(store.sales$visitors)
str(density(store.sales$visitors))
```

plot() is the generic plotting function for R. When it takes other objects, data or functions, it tries to do something sensible with them.

```
plot(density(store.sales$visitors))
plot(density(store.sales$visitors, bw=0.5))
plot(density(store.sales$visitors, bw=3))
```

Univariate: Histogram + Density

Plots can be layered. After **plot()**, you can use **lines()** and other functions to add elements.

Combining raw observations with imputed density. Be sure to use "prob=TRUE" in the histogram.

```
hist(store.sales$visitors, breaks=50, prob=TRUE)
lines(density(store.sales$visitors))
```

Aside: Example search for code help

Googling "**R plot density on histogram**"

. . . quickly finds a replicable answer on stackoverflow:

4 Answers

active oldest votes

33

If I understand your question correctly, then you probably want a density estimate along with the histogram:

```
X <- c(rep(65, times=5), rep(25, times=5), rep(35, times=10), rep(45, times=4))
hist(X, prob=TRUE)            # prob=TRUE for probabilities not counts
lines(density(X))             # add a density estimate with defaults
lines(density(X, adjust=2), lty="dotted")   # add another "smoother" density
```

share | improve this answer

answered Sep 30 '09 at 12:02

Dirk Eddelbuettel
98.6k ●9 ●135 ●230

http://stackoverflow.com/questions/1497539/fitting-a-density-curve-to-a-histogram-in-r

Boxplot

A quick look at distribution and outliers

```
boxplot(store.sales[store.sales$store==1, "visitors"])
```

Many plots understand the formula interface.
You can break out boxplots by a factor for easy comparison.

```
boxplot(store.sales[store.sales$store<=20, "visitors"] ~
        store.sales$store[store.sales$store<=20])
```

Plot routines have a variety of useful options (check "?"):

```
boxplot(store.sales[store.sales$store<=20, "visitors"] ~
        store.sales$store[store.sales$store<=20],
        horizontal=TRUE, xlab="Visitors", ylab="Store")
```

Dotchart

A great substitute for box plots, based on Bill Cleveland's visualization work.

```
dotchart(store.sales[store.sales$store==1, "visitors"])
```

Using **with()** can simplify references and syntax.

```
with(store.sales[store.sales$week==52,],
        dotchart(visitors, labels=storeId))
with(store.sales[store.sales$week==52,],
        dotchart(visitors, labels=paste("Store",storeId)))
```

Add titles and labels

Most plotting functions take "**main=**", "**xlab=**", and "**ylab=**" parameters to add titles.

```
with(store.sales[store.sales$week==52,],
  dotchart(visitors,
    labels=paste("Store",storeId),
    main="Visitors per Store in Week 52",
    xlab="Visitors"
  )
)
```

Plot commands become complex quickly. In R, plots are just code, so indent the commands like other code!

Scatterplots: Basics

plot() understands bivariate formulas (and the "**data=**" parameter).

```
plot(ourBottles ~ visitors, data=store.sales)
```

Scatterplots: Basics

plot() understands bivariate formulas (and the "**data=**" parameter).

```
plot(ourBottles ~ visitors, data=store.sales)
```

Discrete values often overlap. **jitter()** can help separate them visually.

```
plot(jitter(ourBottles) ~ jitter(visitors), data=store.sales)
```

Scatterplots: Basics

plot() understands bivariate formulas (and the "**data=**" parameter).

```
plot(ourBottles ~ visitors, data=store.sales)
```

Discrete values often overlap. **jitter()** can help separate them visually.

```
plot(jitter(ourBottles) ~ jitter(visitors), data=store.sales)
```

... but the association is hard to see. Add a simple regression line. **abline()** adds a plot line from the result of **lm(y ∼ x)** (see below).

```
abline(lm(ourBottles ~ visitors, data=store.sales))
```

Formula syntax

So what is this formula thing?

Many R functions, especially for statistics and plotting, understand a special syntax to specify models.

The general model is this:

```
someFunction( dv ~ iv1 + iv2 + ... , data=MyData)
```

Formula syntax

So what is this formula thing?

Many R functions, especially for statistics and plotting, understand a special syntax to specify models.

The general model is this:

```
someFunction( dv ~ iv1 + iv2 + ... , data=MyData)
```

There are many extensions to this, such as:

```
someFunction( ~ iv1 + iv2 + ... , data=MyData)  # no dv

someFunction( dv ~ . , data=MyData)     # all ivs included

someFunction( dv ~ iv1 + iv2 -1, data=MyData) # no intercept

# interactions in addition to or instead of main effects
someFunction( dv ~ iv1 + iv2*iv3 + iv4:iv5, data=MyData)
```

... and others for models such as nested groups.

Scatter matrices

R makes it easy to visualize many associations at once.

pairs() is a simple version using formula syntax.

```
pairs(~ourBottles + theirBottles + lbsCheese + talkerYN +
    visitors, data=store.sales)
```

Scatter matrices

R makes it easy to visualize many associations at once.

pairs() is a simple version using formula syntax.

```
pairs(~ourBottles + theirBottles + lbsCheese + talkerYN +
    visitors, data=store.sales)
```

Use **subset()** if you wish to examine less data for clearer relationships:

```
pairs(~ourBottles + theirBottles + lbsCheese + talkerYN +
    visitors, data=subset(store.sales, storeId==1))
```

Scatter matrices

R makes it easy to visualize many associations at once.

pairs() is a simple version using formula syntax.

```
pairs(~ourBottles + theirBottles + lbsCheese + talkerYN +
    visitors, data=store.sales)
```

Use **subset()** if you wish to examine less data for clearer relationships:

```
pairs(~ourBottles + theirBottles + lbsCheese + talkerYN +
    visitors, data=subset(store.sales, storeId==1))
```

Package "cars" **scatterplotMatrix()** gives additional functionality:

```
require(car)
scatterplotMatrix( ~ourBottles + theirBottles + lbsCheese +
    talkerYN + visitors, data=subset(store.sales, storeId==1)
       )
```

Learning more about plotting

Plotting can be one of the most frustrating things in R.
That is a feature, not a bug: the control is unparalleled.

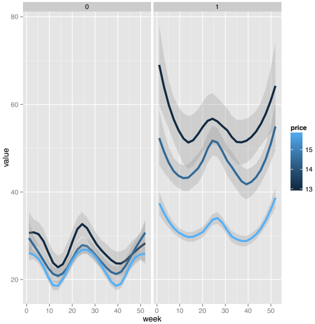
Look for help. Copy examples. Save and reuse working code snippets!

- **Philosophy**: are you `lattice` or `ggplot2`?
- **Books**: *R Graphics Cookbook* + *Lattice* or *ggplot2*
- **Site**: R Graph Gallery: `http://gallery.r-enthusiasts.com/`

A sneak preview of ggplot2

Wine sales: seasonal, with and without shelf talker, by price

```
store.m ← melt(store.sales, id=c("storeId","week","talkerYN",
    "price"))
m.sub ← subset(store.m, variable == "ourBottles")
ggplot(m.sub, aes(week, value)) + facet_grid(. ∼ talkerYN) +
   stat_smooth(aes(group=price, colour=price), alpha=0.3, n
      =25, size=1.5)
```



Topics

CBC in R: Intro

We assume you're already familiar with Choice-Based Conjoint (CBC).

If you were shopping for a new car, which would you prefer?

	Choice A	Choice B	Choice C
Manufacturer	Ford	Toyota	Honda
Styling	140 HP	130 HP	190 HP
Engine type	Hybrid	Gas	Electric
Highway mileage	42 HP	34 mpg	n/a
Horsepower	140 HP	130 HP	190 HP
Sound system	Nakamichi	Factory system	Sound by Bose
Price	$30,200	$29,500	$35,000

This section shows briefly some easy tools for CBC in R.

CBC in R: Why?

Advantages of R for choice models are:

- Complete control over designs, data, estimation
- Advanced estimation tools such as bayesm, mlogit, ChoiceModelR
- Ability to merge data from multiple sources before estimation
- Capabilities to slice, subsample, bootstrap

We can't cover all of that here, but will show you:

- How to create a basic CBC design, collect survey test data, and estimate an aggregate MNL model
- How to do basic market simulation in R
- Pointers to other tools for more advanced models

The key parts of CBC

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- R: MNL estimation
- R: Preference Simulation

Rcbc

We are using our own code called "Rcbc" here, which has pros & cons:

- Pro: Easy to use
- Pro: Does CBC from design to test "fielding" to estimation to market simulation
- Pro: Works easily with Sawtooth Software CBC data
- Pro: Includes a wrapper to do HB easily
- Con: requires rectangular data (same number of tasks and observations for every respondent)
- Con: only does CBC

We'll mention some alternatives later.

Load Rcbc

Locate the Rcbc.R file

$\Rightarrow$ "Source" it in R to make its functions available.

Attribute/Level Structure

- **Design: Attribute/level structure**

```
set.seed(4567)
# five attributes, with 3-5 levels each
attr.list   <- c(3, 3, 5, 5, 4)
```

- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- R: MNL estimation
- R: Preference Simulation

The concept/task design matrix

- Design: Attribute/level structure
- **Design: Attribute/level friendly names**

```
attr.names  ← c("Size", "Performance", "Design", "Memory"
    , "Price")
attr.labels ← c(
    "Nano",  "Thumb",  "Full-length",
    "Low speed",  "Medium speed", "High speed",
    "Tie-dye",  "Silver",  "Black",  "White",  "Prada",
    "1.0GB",  "8.0GB",  "16GB",  "32GB",  "256GB",
    "$9",   "$29",  "$59",   "$89"
)
```

- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- R: MNL estimation

The concept/task design matrix

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- **Design: The concept/task design matrix**

```
tmp.tab ← generateMNLrandomTab(attr.list,
              respondents=3, cards=3, trials=12)
head(tmp.tab)
tmp.des ← convertSSItoDesign(tmp.tab)
head(tmp.des)
```

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- R: MNL estimation
- R: Preference Simulation

Survey presentation

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- **Survey: Presentation**

```
current.wd <- "~/Desktop/"
writeCBCdesignCSV(tmp.tab, attr.list=attr.list,
  lab.attrs=attr.names, lab.levels=attr.labels,
  filename=paste(current.wd,"writeCBCtest.csv",sep=""),
  delimeter=",")
```

- Survey: Responses
- Survey: Data in R

- R: MNL estimation
- R: Preference Simulation

Survey "fielding"

Open that CSV file using your favorite spreadsheet program (we recommend Google Docs!)

$\Rightarrow$ Fill out a couple of choice sets and save back as CSV.

Survey responses

- Survey: Presentation
- **Survey: Responses**

  Using a local spreadsheet:

```
tmp.win ← readCBCchoices(tmp.tab,
  filename=paste(current.wd, "writeCBCtest.csv", sep=""))
```

  Using Google Docs:

```
current.wd ← "~/Downloads/"
tmp.win ← readCBCchoices(tmp.tab,
  filename=paste(current.wd, "writeCBCtest - Sheet 1.csv",
  sep=""))
```

- Survey: Data in R

Survey data in R

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- **Survey: Data in R**

```
(tmp.win.exp <- expandCBCwinners(tmp.win))
```

- R: MNL estimation
- R: Preference Simulation

MNL Estimation

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- **R: MNL estimation**

```
tmp.pws ← estimateMNLfromDesign(tmp.des, tmp.win)
(tmp.res ← data.frame(attr=attr.labels, util=t(tmp.pws)
    [,1]))  # nicer formatting to import to a spreadsheet
```

- R: Preference Simulation

Preference simulation

- Design: Attribute/level structure
- Design: Attribute/level friendly names
- Design: The concept/task design matrix

- Survey: Presentation
- Survey: Responses
- Survey: Data in R

- R: MNL estimation
- **R: Preference Simulation (not run here; next section)**

```
tmp.prod1 <- c(1, 5, 10, 14, 17)    # product 1
tmp.prod2 <- c(2, 6, 11, 15, 20)    # product 2
tmp.pref  <- marketSim(
    tmp.ind.pws,    # individual-level utilities
    list(tmp.prod1, tmp.prod2) ) # products
colMeans(tmp.pref)    # average preference
```

Topics

Market simulation in CBC

In this section, we will load survey responses from a Sawtooth TAB file, estimate individual-level utilities, and do market simulation with those.

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Note: How to save a TAB file from SSI Web 1

In version 8.x, Sawtooth Software changed the TAB file export.
To get the format now in SSI/Web:

File, Data Management ...
Add job ...
(delete the export action)
Add ... CBC
Set "File format" to "**Single Format CSV**"

Single format because the design and responses are in a single file.

Note: How to save a TAB file from SSI Web 2

The export should look like this in the resulting file:

```
"sys_RespNum",Task,Concept,"Brand:","Performance:","Price:",
    Response
2,0,1,1,3,4,0
2,0,2,4,2,2,0
2,0,3,2,1,3,0
2,0,4,0,0,0,1
2,1,1,2,1,3,1
...
```

Each row is 1 concept with ID, attribute levels, and chosen-or-not in last column.
In this format, it is ready for Rcbc.

The single CSV (tab) format

```
"sys_RespNum",Task,Concept,"Brand:","Performance:","Price:",
   Response
2,0,1,1,3,4,0
2,0,2,4,2,2,0
2,0,3,2,1,3,0
2,0,4,0,0,0,1
2,1,1,2,1,3,1
...
```

sys_RespNum	ID variable for respondent
Task	Choice set (e.g., 0 of 8 or whatever)
Concept	Card within the choice set (e.g., 1-3)
(other columns)	Nominal attribute levels dummy-coded
Response	1=concept chosen, 0=not chosen

Get the TAB file data 1

- **R: Import Sawtooth survey data**
  We generated fake data for the Sawtooth Software "Golf" example (SSI Web 8.2.0) for N=100 respondents, and saved it to a TAB file.

```
current.wd <- "~/Documents/Chris Documents/papers/Sawtooth/
    SSC 2013/R tutorial/handouts/"

tmp.raw.all <- read.csv(paste(current.wd,"Sawtooth R
    tutorial 2013 handouts/CBCgolfexercise.csv",sep=""))
head(tmp.raw.all, 20)
table(tmp.raw.all$sys_RespNum)

summary(as.numeric(table(tmp.raw.all$sys_RespNum)))
```

- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Get the TAB file data 2

- **R: Import Sawtooth survey data**
  A bit of clean up – remove "none" rows (out of scope for this intro – see code and ?choicemodelr)

```
# remove every 4th row (none), out of scope today
tmp.cutrows ← seq(from=4, to=nrow(tmp.raw.all), by=4)

tmp.raw.all ← tmp.raw.all[-tmp.cutrows, ]
str(tmp.raw.all)
```

- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Prepare ChoiceModelR data

- R: Import Sawtooth survey data
- **R: Prepare parameters**

```
tmp.tab ← tmp.raw.all[, 4:6]   # design columns
head(tmp.tab)
str(tmp.tab)

tmp.win ← tmp.raw.all[, 7] # response column
summary(tmp.win)
```

- R: Estimate individual-level utilities
- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Estimate individual-level utilities

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- **R: Estimate individual-level utilities**

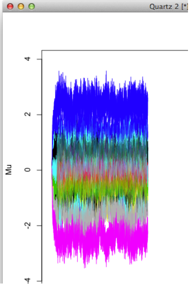
```
tmp.logitHB <- estimateMNLfromDesignHB(tmp.tab, tmp.win,
    kCards=3, kTrials=17, kResp=100)
```

- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Estimate individual-level utilities

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- **R: Estimate individual-level utilities**

```
                    Logit Data
==================================================
Attribute       Type         Levels
-----------------------------------
Attribute 1     Part Worth      6
Attribute 2     Part Worth      6
Attribute 3     Part Worth      5
Attribute 4     Part Worth      2
Attribute 5     Part Worth      5
Attribute 6     Part Worth      3
Attribute 7     Part Worth      3

23 parameters to be estimated.

146 total units.
Average of 3 alternatives in each of 12 sets per unit.
1752 tasks in total.

Table of choice data pooled across units:
Choice  Count   Pct.
--------------------
   1     623    35.56%
   2     684    39.04%
   3     445    25.4%

      MCMC Inference for Hierarchical Logit
==================================================
Total Iterations:          1e+05
Draws used in estimation:  5000
Units:                     146
Parameters per unit:       23
Constraints not in effect.
Draws are to be saved.
Prior degrees of freedom:  5
Prior variance:            2

MCMC Iteration Beginning...
Iteration  Acceptance   RLH     Pct. Cert.   Avg. Var.   RMS     Time to End
      100  0.338        0.435   0.222        0.48        0.37    13:55
      200  0.306        0.544   0.427        0.85        0.73    12:45
      300  0.304        0.607   0.535        1.23        1.02    12:08
      400  0.302        0.645   0.597        1.54        1.20    11:44
      500  0.297        0.665   0.626        1.84        1.34    12:18
      600  0.298        0.677   0.645        2.11        1.46    12:56
      700  0.301        0.686   0.656        2.32        1.55    12:41
```



- R: Model individual preferences
- R: Assign choices and sum
- R: Bootstrap for confidence

Model individual preferences

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- **R: Model individual preferences**
  1. Use the individual-level average draws:

```
(tmp.attrs ← findSSIattrs(tmp.tab)) # infer CBC structure
tmp.HBindbetas ← extractHBbetas(tmp.logitHB, tmp.attrs)
head(tmp.HBindbetas)
```

  – OR –

  2. Use the full individual-level draws (ask us afterwards :)
- R: Assign choices and sum
- R: Bootstrap for confidence

Create some product definitions to model

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- **R: Model individual preferences**

```
# create products for market simulation
golf1 <- c(1, 5, 8)  # "1", "1", "1"
golf2 <- c(2, 7, 11) # "2", "3", "4"
```

- R: Assign choices and sum
- R: Bootstrap for confidence

Assign choices and sum

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- R: Model individual preferences
- **R: Assign choices and sum**

```
tmp.sim1 <- marketSim(tmp.HBindbetas, list(golf1, golf2))
head(tmp.sim1)
colMeans(tmp.sim1)
```

- R: Bootstrap for confidence

Bootstrap for confidence

- R: Import Sawtooth survey data
- R: Prepare ChoiceModelR parameters
- R: Estimate individual-level utilities
- R: Model individual preferences
- R: Assign choices and sum
- **R: Bootstrap for confidence** *use.error & draws*

```
# if "use.error" then we want to repeat
tmp.sim2 <- marketSim(tmp.HBindbetas, list(golf1, golf2),
                style="first", draws=1000, use.error=TRUE)
head(tmp.sim2)
colMeans(tmp.sim2)
```

Topics

- Basics of R
- Loading and saving data
- Plotting
- CBC in R
- Individual-Level Market Simulation
- **Final Rcbc notes**
- General Data Analysis and Statistics
- Specialized Topics & Pointers
- Overview: GUIs for R
- Conclusion

Gradient pricing

Can be easily done in R, when given a set pricing attribute:

```
price.14 <- 0.5*util\$price.9 + 0.5*util\$price.19
```

as well as when given conditional pricing, requires a bit more work:

```
costLevels <- function(attributeList) {
    costs <- c(0,0,...,29,62)
    sums <- c(51,80,...,106,116)
    lows <- c(25,53,...,66,76)
    mids <- c(51,109,...,135,155)
    highs <- c(77,165,...,204,234)

    attrSum <- sum(costs[attributeList])+22
    sumIndex <- which(sums == attrSum)
    matches <- list("low" = lows[sumIndex],"mid" = mids[
        sumIndex], "high" = highs[sumIndex])

    return(matches)
}
```

MarketSim supports

A None option:

```
marketSim(...,use.none=TRUE,...)
```

as well as introducing random error:

```
marketSim(...,use.error=TRUE,...)
```

and using a tuning factor:

```
marketSim(...,tuning=1.0,...)
```

Some other conjoint options in R

`bayesm` : Bayesian models from Rossi, Allenby, McCulloch
Versatile, but advanced code can be tricky

`ChoiceModelR` : works well with Sawtooth Software data
Rcbc provides a wrapper that is good starting point

`mlogit` : good regression approach; only aggregate models

`clogit` & `conjoint` packages not best starting points for Sawtooth Software users (concepts vary too much).

Topics

The Common Statistics Model

Data analysis in R typically consists of these elements:

- Explore your data
- Find the right statistics function for a model
- Define your model with *formula syntax*
- Run the function and save its results to an object
- Inspect the `summary()` of the result object
- Drill into the details of the model result
- Compare models if needed

Example (based on `?swiss`) :

```
summary(swiss)
fert.mod1 <- lm(Fertility ~ Agriculture + Education + Catholic
     , data = swiss)
summary(fert.mod1)
```

Example result

```
> summary(fert.mod1)

Call:
lm(formula = Fertility ~ Agriculture + Education + Catholic,
    data = swiss)

Residuals:
    Min      1Q  Median      3Q     Max
-15.178  -6.548   1.379   5.822  14.840

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 86.22502    4.73472  18.211  < 2e-16 ***
Agriculture -0.20304    0.07115  -2.854  0.00662 **
Education   -1.07215    0.15580  -6.881 1.91e-08 ***
Catholic     0.14520    0.03015   4.817 1.84e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
```

Univariate Statistics

```
table(store.sales$ourBottles)
mean(store.sales$ourBottles)
median(store.sales$ourBottles)

summary(store.sales)
summary(store.sales$income)

require(psych)
describe(store.sales)

quantile(store.sales$income)
quantile(store.sales$income, pr=0.80)
quantile(store.sales$income, pr=c(0, 0.20, 0.5, 0.80, 0.98))
```

Inverse quantiles: distribution function

Remember that functions are just objects like any other?

The return value of a function can be a function.

ecdf() constructs a function that gives the inverse quantile of a value. In other words, it maps raw values to their percentiles (as observed empirically in some data).

```
inc.cdf ← ecdf(store.sales$income)
inc.cdf
plot(inc.cdf)

inc.cdf(70000)
inc.cdf(seq(from=30000, to=80000, by=5000))
```

Bivariate statistics

Correlation

```
cor(store.sales$income, store.sales$ourBottles)    #
    coefficient

cor.test(store.sales$income, store.sales$ourBottles)

cor(store.sales)
```

`t.test()` compares the mean of two vectors:

```
t.test(1:20, 5:18)

t.test(1:20, jitter(3:22))
t.test(1:20, jitter(3:22), paired=TRUE)

with(store.sales,
  t.test(ourBottles[storeId==1], ourBottles[storeId==36])
)
```

Data Cleanup Before Proceeding

Our data includes some variables that are not continuous measures.

```
summary(store.sales)
```

Let's set those to be `factor` or `Date` types:

```
store.sales$storeId ← as.factor(store.sales$storeId)

(tmp.date ← (store.sales$week-1) * 7)
(tmp.date ← as.Date(tmp.date, origin="2012-01-02"))

store.sales$date ← tmp.date
table(store.sales$date)

summary(store.sales)
head(store.sales)
some(store.sales)
```

Aggregating Data

R provides superb (and many) ways to aggregate data.
Simple functions are `colMeans()` and `rowMeans()`:

```
xy ← matrix(1:100, ncol=4, byrow=TRUE)
head(xy)

colMeans(xy)
rowMeans(xy)
```

A generalized version is `apply(`*`data, margin, FUN`*`)`.
margin = 1 for rows, 2 for columns.
FUN can be any appropriate function.

```
apply(xy, 1, mean)
apply(xy, 2, mean)
```

More on `apply()`

The function can more complex than simple math, and can return complex objects:

```
apply(xy, 1, summary)
apply(xy, 2, summary)
```

You can write your own *anonymous function* and it will be `apply()`'ed:

```
apply(xy, 2, function(x) { log(summary(x)) }) # anonymous fn
```

The result is an object just like any other object:

```
xy.log ← apply(xy, 2, function(x) { log(summary(x)) }) #
    results are a matrix
xy.log

xy.log[5,] - xy.log[2,]     # interquartile range
```

aggregate()

`aggregate()` lets you aggregate using formula syntax:

```
aggregate(ourBottles ~ price + talkerYN, data=store.sales,
    mean)
```

The result is – of course – an object you can work with:

```
bot.ag <- aggregate(ourBottles ~ price + talkerYN,
                    data=store.sales, mean)

head(bot.ag)
bot.ag[bot.ag$price==12.99 & bot.ag$talkerYN==1, 3]
```

Crosstabs

`xtabs()` can turn the result into a cross-tab table:

```
xtabs(ourBottles ~ ., data=bot.ag)
bot.xtab ← xtabs(ourBottles ~ ., data=bot.ag)
```

And the *xtable* package will format that for LaTeX:

```
library(xtable)
xtable(bot.xtab)
```

The output from `xtable()`:

```
\begin{table}[ht]
\centering
\begin{tabular}{rrr}
  \hline
  & 0 & 1 \\
  \hline
  12.99 & 27.24 & 55.72 \\
  14.49 & 24.57 & 46.29 \\
  ...
```

Looks great, and is easy to edit for publication or slides:

	0	1
12.99	27.24	55.72
14.49	24.57	46.29
15.99	23.11	31.39

Q&A

Let's clean up the environment!

```
rm(xy, xy.log, bot.ag, bot.xtab)
```

Questions?

Time for a break before linear models?

Linear Models

Let's start with a basic model: *ourBottles* $\sim$ *storeid* $+$ *visitors*:

```
mod1 ← lm(ourBottles ~ storeId + visitors, data=store.sales)
summary(mod1)
```

R understands factors, so we can get a typical ANOVA output (e.g., testing *storeId* as a significant nominal grouping factor):

```
anova(mod1)
```

Comparing Models

How good is the *mod1* model?
Let's add `lbsCheese` and see if it improves:

```
mod2 ← update(mod1, . ~ . + lbsCheese)
summary(mod2)

BIC(mod1, mod2)
```

If you like stepwise functions, those are available too:

```
mod3 ← step(lm(ourBottles ~ ., data=store.sales))
```

More appropriate for count data might be something like a poisson model, but we'll leave that aside . . . except to point out `glm()` that handles `poisson` (and many other) models.

```
summary(glm(ourBottles ~., data=store.sales, family=poisson))
```

Pop Quiz

1. Add a new variable to the *store.sales* data set for `log()` of *price*.

2. Is *price* or *log of price* a better predictor of sales in these data?

Pop Quiz

1. Add a new variable to the *store.sales* data set for `log()` of *price*.

```
store.sales$logprice <- log(store.sales$price)
```

2. Is *price* or *log of price* a better predictor of sales in these data?

```
mod.p1 <- lm(ourBottles ~ price + lbsCheese + talkerYN,
             data=store.sales)
mod.p2 <- lm(ourBottles ~ logprice + lbsCheese + talkerYN,
             data=store.sales)

summary(mod.p1)
summary(mod.p2)

BIC(mod.p1, mod.p2)
anova(mod.p1, mod.p2)
```

Overview of Model Diagnostics

What about OLS vs. Poisson: which model is better?
That's mostly an a priori choice about the structure you expect.

However, in addition to model comparison, you should examine model diagnostics:

```
plot(mod3)
```

We see in the residuals for model 3 that something looks bad.

Starting to debug a model

Let's look at some of the variables that `step()` omitted:

```
with (store.sales, plot(visitors, ourBottles))
with (store.sales, plot(week, ourBottles))
```

This suggests a time element, so let's look at data lag:

```
acf(residuals(mod3))

durbinWatsonTest(mod3, max.lag=12)
```

We have correlated errors, violating `lm()` assumptions.

Fitting a model with correlated error

The residuals have a sine-wave-like pattern, so we suspect second order ARMA (autoregressive moving average) model.

Fit a `gls()` with correlated error (using 5 stores only for speed):

```
library(nlme)

mod4 <- gls(ourBottles ~ storeId + lbsCheese + talkerYN +
        price + theirBottles,
        correlation=corARMA(p=2) ,
        data=subset(store.sales,
                    as.numeric(as.character(storeId)) < 6))

plot(mod4)
summary(mod4)
summary(mod3)
```

(We cast *storeId* to “numeric” in `subset()` to select data easily.)

Q&A

Even better would be a **time-series model** . . . but that goes beyond our scope: `cran.r-project.org/web/views/TimeSeries.html`

Q&A

Time for a break?

Topics

Bayesian core methods: MCMCpack

MCMCpack implements MCMC in C++ for fast Bayesian estimation.

```
library(MCMCpack)

mod.b1 ← MCMCregress(ourBottles ~ visitors,
    data=store.sales, burnin=10000, mcmc=10000, verbose=5000,
    b0=0, B0=0.1, marginal.likelihood="Chib95")

plot(mod.b1)
summary(mod.b1)
```

Bayesian core methods: MCMCpack

MCMCpack implements MCMC in C++ for fast Bayesian estimation.

```
library(MCMCpack)

mod.b1 <- MCMCregress(ourBottles ~ visitors,
    data=store.sales, burnin=10000, mcmc=10000, verbose=5000,
    b0=0, B0=0.1, marginal.likelihood="Chib95")

plot(mod.b1)
summary(mod.b1)
```

Now let's add *lbsCheese*:

```
mod.b2 <- MCMCregress(ourBottles ~ visitors + lbsCheese,
    data=store.sales, burnin=10000, mcmc=10000, verbose=5000,
    b0=0, B0=0.1, marginal.likelihood="Chib95")

plot(mod.b2)
summary(mod.b2)
```

Simple Bayesian model comparison

`BayesFactor()`: does model 2 fit the data better than model 1?

```
BF.12 ← BayesFactor(mod.b1, mod.b2)
BF.12

PostProbMod(BF.12)
```

Simple Bayesian model comparison

`BayesFactor()`: does model 2 fit the data better than model 1?

```
BF.12 <- BayesFactor(mod.b1, mod.b2)
BF.12

PostProbMod(BF.12)
```

Is *talkerYN* + *price* a better model than *visitors* + *lbsCheese*?

```
mod.b3 <- MCMCregress(ourBottles ~ talkerYN + price,
    data=store.sales, burnin=10000, mcmc=10000, verbose=5000,
    b0=0, B0=0.1, marginal.likelihood="Chib95")

BF.123 <- BayesFactor(mod.b1, mod.b2, mod.b3)
summary(BF.123)
PostProbMod(BF.123)
```

Be careful with model comparison; but it's better than NHST!

That's all! Well, almost . . .

That concludes the **language** part of the tutorial!

We'll discuss a few other **environment** topics now.

Some packages for marketers to know about

There are >4500 R packages on CRAN. Here are some of our favorites:

cluster	Divisive and agglomerative clustering
party, partykit	Classification trees
mclust, flexmix	Model-based clustering
randomForest	Random forests
ggplot2, lattice	Powerful, publication-quality graphics
Sweave	Integrated code, documents & reporting with LaTeX
survey	Stratified estimation, imputation, database support
parallel, foreach	Multicore processing (`foreach` & `mcapply()`)
bigmemory, biglm	Support for very large and sparse data
psych, sem	Psychometrics; Structural equation models
bayesm	Bayesian methods for marketing

Book Recommendations: Publishers' Lines

(Caveat: Chris's opinions!)

R is hot in the publishing industry, but the books have a wide range of target audiences. In general:

O'Reilly	Broadly accessible, easy cookbooks; coding not stats
Springer intros	Great intros, sometimes not sufficiently applied
Springer general	High-quality although many are only for specialists
Chapman & Hall	Some are very specialized; review before buying
Other publishers	Mixed quality, review case-by-case

Book Recommendations: For Researchers New to R

Muenchen (2008). R for SAS and SPSS Users. Nice how-to on the R environment and data handling (not stats).

Zuur et al (2009). A Beginner's Guide to R. Basics of R for applied researchers. Better for them than typical intros for undergraduates.

Teetor (2011). R Cookbook. 1-volume handbook of many topics in problem+solution format. Broad coverage; only simple stats models.

Everitt & Hothorn (2009). A Handbook of Statistical Analyses Using R. Excellent examples of common models.

Fox (2010). An R Companion to Applied Regression. Great text on regression and regression diagnostics in R, with associated package.

Matloff (2011). The Art of R Programming. Single best book on *programming* in R (not statistics).

Using R in Commercial Settings

Some IT and Legal departments are skeptical of R. There is a lot of misinformation, perhaps especially in regulated industries.

Suggestions:

- Have legal review the GNU GPL and make sure it is OK
- If you write code, strongly consider making it GPL
- Check the licenses separately for each package; not all are GPL
- Show that R is where most new stats methods appear first!
- Show that R is used by many cutting-edge companies

Commercially-supported alternatives exist:

Spotfire S+	Modest compatibility, IDE, data-mining enhancements
Revolution R	Highly compatible, support, IDE; slower releases

Topics

GUIs

Graphical User Interfaces for R can:

- Highlight syntax, do advanced editing, and basic error-checking
- Manage multiple files and complex projects
- Integrate help, debugging, and plotting
- Allow inspection of variables, code structure, etc.

Tinn-R

A good code editor for Windows, with basic R console integration.

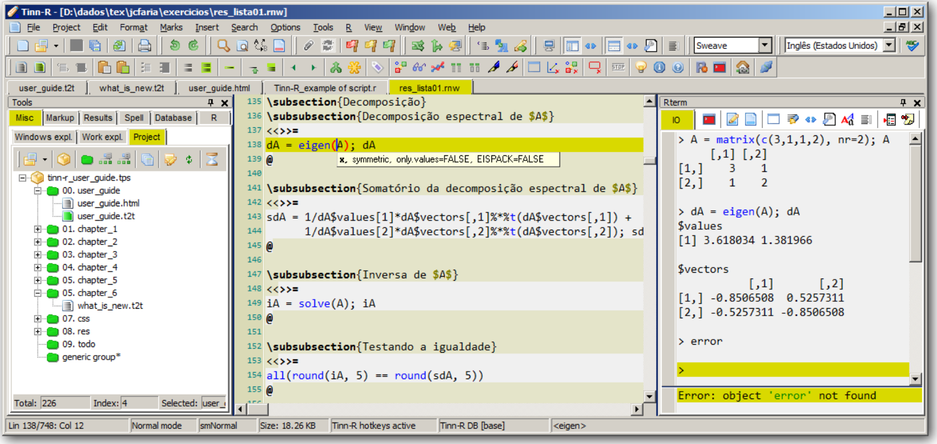


Figure : from http://nbcgib.uesc.br/lec/software/des/editores/tinn-r/en

RStudio

Clean, multi-platform, integrated, focused on R. Great plot support. Both local and client-server models; controversial AGPL license.

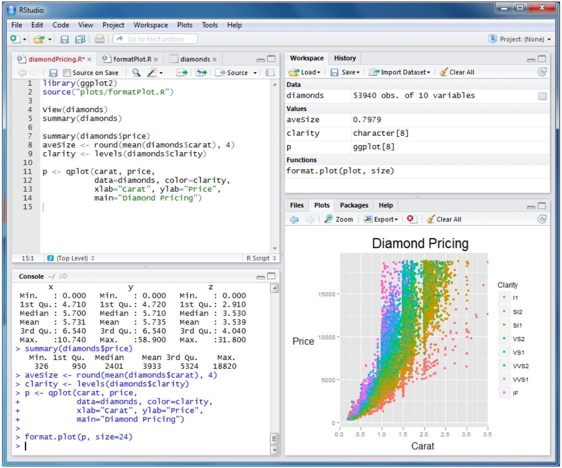
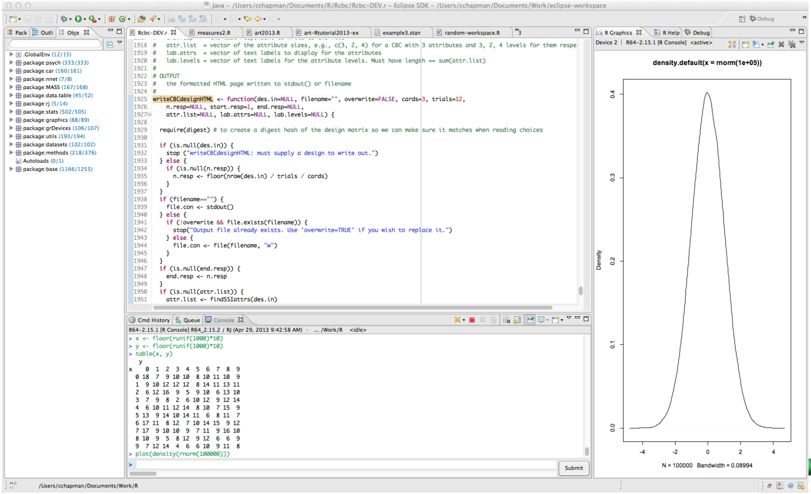


Figure : from http://www.rstudio.com/ide/

Eclipse + StatEt

Appeals to developers who use Eclipse IDE for other purposes (e.g., Java); pro level coding support, somewhat rough edges on R plug-in.



Emacs Speaks Statistics (ESS)

Integrating R into Emacs. For Emacs users only IMHO, who swear by it.

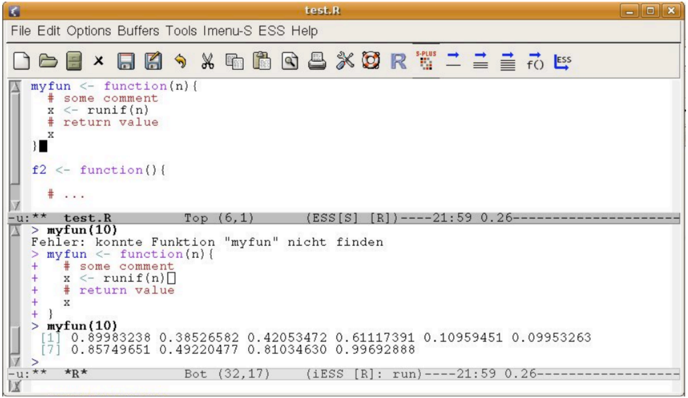


Figure : from http://www.rali.boku.ac.at/fileadmin/_/H85/H851/r-workshop/Gebhardt_ess3.pdf

R Commander

Interactive menus for R, somewhat similar to SPSS.

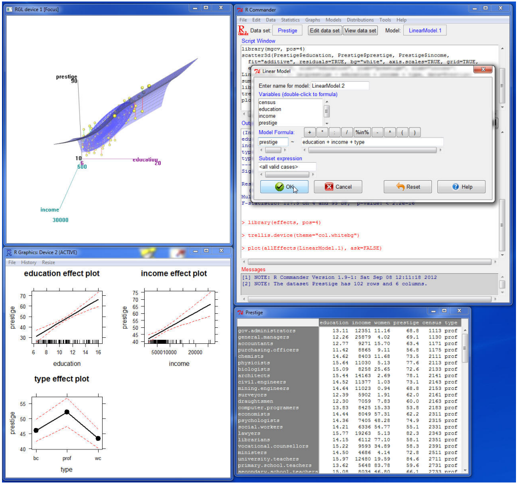


Figure : from http://www.sciviews.org/_rgui/

GUI Recommendation

You'll want a code editor.
Your favorite may integrate with R, or use copy & paste.

On Windows, Tinn-R and Notepad++ are highly regarded.

For more functionality, RStudio has many adherents.

If you do any proprietary code development, RStudio has a lesser-known license (AGPL). Get your legal team to review.

Topics

Conclusion

Three final tips / reminders:

- Clean up your workspace
- Make important analyses replicable start-to-finish in a `.R`
- Write readable code and document it

Thank you!

{elliss, cchapman}@google.com