

R Seminar

Contents

1	Visual Demos	2
2	R Basics	2
2.1	Statistical Calculator	2
2.2	Some basics	2
3	S Language	2
3.1	Arithmetic	2
3.2	Matrix Arithmetic	4
3.3	Logical expressions	5
3.4	Some useful functions	6
3.5	Loops	7
3.6	Functions	8
4	Graphics	10
4.1	High-level plotting functions	10
4.2	Low-level plotting functions	13
4.3	Interactive functions	14
4.4	Other parameters	14
4.5	Adding Maths Symbols	15
5	Neural Networks	16

1 Visual Demos

- Mount Eden Volcano; `example(volcano)`
- Napoleon's Advance on Moscow; `source('minard.R')`
- Central Limit Theorem,
`plot.rnorm.mean, plot.rexp.mean, plot.norm.pop; add.rmean; add.ci`
- Rainfall Simulator; `source('nsrp.R')`

2 R Basics

2.1 Statistical Calculator

- P-values: `pnorm`
- Quantiles: `qnorm`

2.2 Some basics

- Reading in data `read.table` or `scan`;
- Listing variables `ls`;
- basic syntax.

3 S Language

3.1 Arithmetic

The basic data object in S is the 'vector', which essentially corresponds to univariate data. We can compute with vectors in the same way as numbers on a calculator:

```

# Put the first four values of the speed of light data
# (MM, Table 1.1) into a vector x:
> x <- c(28, 26, 33, 24, 34, -44)
> x * x
[1] 784 676 1089 576 1156 1936
> x ^ 2
[1] 784 676 1089 576 1156 1936
> x + 1
[1] 29 27 34 25 35 -43
> y <- rep(0, 6)
> y
[1] 0 0 0 0 0 0
> x / y
[1] Inf Inf Inf Inf Inf -Inf
> y <- x ^ 3
> y
[1] 21952 17576 35937 13824 39304 -85184
> y <- sqrt(x)
Warning message:
NaNs produced in: sqrt(x)
> y
[1] 5.291503 5.099020 5.744563 4.898979 5.830952      NaN

# N.B. NaN = 'not a number'; NA = not available (missing); Inf = infinity

# 'Complex' numbers can be created using the 'i' option:
> x <- x + 0i
> x
[1] 28+0i 26+0i 33+0i 24+0i 34+0i -44+0i
> y <- sqrt(x) # now taking the square roots of a negative is ok:
> y
[1] 5.291503+0.000000i 5.099020+0.000000i 5.744563+0.000000i 4.898979+0.000000i
[5] 5.830952+0.000000i 0.000000+6.63325i
> z <- 0+1i
> z
[1] 0+1i
> z ^ 2
[1] -1+0i
>

> x <- scan('light.dat') # put speed of light data into x

```

```

Read 66 items
> x <- x[1:6] # use just first 6 values again (as above)
> x
[1] 28 26 33 24 34 -44
> y <- c(rep(1,2), rep(2,3)) # make a vector y of length 5
> x + y # add the two vectors - see how recycling rule works:
[1] 29 27 35 26 36 -43
Warning message:
longer object length
      is not a multiple of shorter object length in: x + y
> # read warning message and then discard

> # integer arithmetic:
> 2 %% 3 # integer division
[1] 0
> 2 %/% 1
[1] 2
> 6 %% 2
[1] 3
> 6 %/% 4
[1] 1
> 6 %% 4 # modulo arithmetic
[1] 2

```

3.2 Matrix Arithmetic

`%%` – matrix multiplication (for element by element use `*` as with vectors)
`t()` – transpose a matrix

```

> x <- matrix ( c(rep(1, 3), rep(2, 3)), nr = 2, byrow = T)
> x
      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2

# Multiply each element by itself - not the same as squaring the matrix!
> x * x

```

```

      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    4    4    4
> x * 4
      [,1] [,2] [,3]
[1,]    4    4    4
[2,]    8    8    8
> x + 1
      [,1] [,2] [,3]
[1,]    2    2    2
[2,]    3    3    3
>

```

3.3 Logical expressions

The operators `<=`, `>=`, `<`, `>`, `==`, `!=` act on logical vectors or scalars. We also have `&` for ‘and’ and `|` for ‘or’ when comparing two logical vectors element by element (the result of the operation will also be a vector of equal length). For comparing scalars we have `&&` and `||`.

```

x <- scan('light.dat')
Read 66 items
> x
 [1]  28  26  33  24  34 -44  27  16  40  -2 ... etc
[20]  19  24  20  36  32  36  28  25  21  28 ...
[39]  30  22  36  23  27  27  28  27  31  27 ...
[58]  25  32  25  29  27  28  29  16  23
> x > 0
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE ... etc
[13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE ... etc
[25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> sum(x < 0)
[1] 2
> y <- x[x < 0]
> y
[1] -44 -2

```

3.4 Some useful functions

- `round()`, `signif()` – round data to a number of decimal/significant figures;
- `abs`, `log`, `sqrt`, `exp`, `sin`, `cos`, `tan` – some well known functions;
- `sum()`, `prod()` – finds the sum and product of elements in a vector;
- `max()`, `min()` – find the max and min of a vector;

```
> z <- 12345.67
> z
[1] 12345.67
> round(z,1)
[1] 12345.7
> round(z,2)
[1] 12345.67
> round(z)
[1] 12346
> signif(z, 4)
[1] 12350
> signif(z, 1)
[1] 10000

> y <- c(rep(1,2), rep(2,3), rep(-1,3))
> y
[1] 1 1 2 2 2 -1 -1 -1
> z <- c(rep(1,3), rep(2,3), rep(3,2))
> z
[1] 1 1 1 2 2 2 3 3
> y > z
[1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE

> x <- ifelse(sum(z) > sum(y), "z bigger", "z smaller")
> x
[1] "z bigger"
>
```

3.5 Loops

In S these should be avoided, when possible, as calculations using matrix and vector arithmetic are much faster. The loops available in S are:

- `for (variable in sequence) statement`
- `while (condition) statement`
- `repeat statement`

A ‘statement’ can consist of one or more commands in brackets `{}`. Repeat continues until a ‘break’ command is met (‘break’ can also be used to exit the other loops).

```
> for (i in 1:3) write(file='', 'hello')
hello
hello
hello
```

```
> for (i in 1:5) write(file='', i)
1
2
3
4
5
```

```
> x <- 0
> for (i in 1:10) x <- c(x,i)
> x
[1] 0 1 2 3 4 5 6 7 8 9 10
>
```

```
> num <- c('0')
> for (i in 1:5) num <- paste(num, i, sep='')
> num
[1] "012345"
```

```
> num <- c('0')
> j <- c(2, 99, 1)
```

```

> num <- c('0')
> for (i in j) num <- paste(num, i, sep='')
> num
[1] "02991"

>

```

3.6 Functions

S is a 'functional language', so functions are 'first class' objects, which means they can be manipulated and returned like any other S object. Functions are created with the 'function' statement. Here are some examples:

```

> f <- function(x) x ^ 2
> f(3)
[1] 9

```

use brackets for longer functions:

```

> f <- function (x)
  { if (x < 0)
    { write(file='', "Warning: Neg value entered, changed to pos");
      y <- -x
    }
    else y <- x
    log(y)
  }
> f(2)
[1] 0.6931472
> f(-2)
Warning: Neg value entered, changed to pos
[1] 0.6931472
>

```

```

> rm(list=ls()) # clear the workspace and look at how R handles functions:
> ls()
character(0)

```

```

> g <- function (f, x) f(x)
> ls() # only the function 'g' should be in the workspace:

```

```

[1] "g"
> g(function(x) x^2, 3)
[1] 9
> ls() # again only 'g' should be in the workspace:
[1] "g"
> # note: the function f never had to be named

# function composition is also straightforward:
> rm(list=ls()) # start by clearing workspace
> ls()
character(0)

> g <- function (x) x^2
> f <- function (x) x+1
> h <- function (x) g(f(x)) # h is a composite function
> h(2)
[1] 9

# a function can return a function:
> g <- function(f) function(x) f(x) * f(x)
> h <- function (x) x + 1
> g(h)(2)
[1] 9

> # It wasn't necessary to assign h:
> rm(list=ls()) # clear workspace then:

> g <- function(f) function(x) f(x) * f(x)
> g(function (x) x + 1)(2)
[1] 9

# Note also: functions don't have to be named at all:
> rm(list=ls())
> (function (x) x + 1)(2)
[1] 3

> rm(list=ls())
> (function(f) function(x) f(x) * f(x))(function (x) x + 1)(2)
[1] 9

```

4 Graphics

4.1 High-level plotting functions

These create a new plot on the graphics device, and include the following:

- `plot()` – the generic all-purpose plotting function;
- `boxplot()` – ‘box and whiskers’ plot;
- `barplot()` – ‘bar’ plots of one or more variables;
- `hist()` – histograms;
- `pairs()` – ‘matrix’ plot for multivariate data;
- `coplot()` – conditional plots for multivariate data;
- `qqplot()` – quantile plots;
- `contour()` – 2D contour plot;
- `persp()` – 3D surface plot.

```
# Location of Earthquakes in Tonga
> quakes <- read.table('quakes.dat', head=T)
> quakes[1:4,]
      lat  long depth mag stations
1 -20.42 181.62  562 4.8      41
2 -20.62 181.03  650 4.2      15
3 -26.00 184.10   42 5.4      43
4 -17.97 181.66  626 4.1      19
> attach(quakes)
> plot(long, lat)

# Boxplot of exchange rates by month
> rate <- read.table('exrates.dat', head=T)
> rate[1:4,]
  year mth  rate
1 1991   1 3.2515
2 1991   2 3.2641
```

```

3 1991    3 3.0746
4 1991    4 2.9697
> rate.split <- split(rate$rate, factor(rate$mth))
> boxplot(rate.split)

# barplot for mean rainfall data
> rain <- read.table('nzrain.dat', head=T)
> rain.mean <- apply( matrix(rain$rain, nr=12), 1, mean )
> barplot(rain.mean, names.arg=month.abb)

# coplot
> quakes <- read.table('quakes.dat', head=T)
> quakes[1:4,]
      lat   long depth mag stations
1 -20.42 181.62  562 4.8      41
2 -20.62 181.03  650 4.2      15
3 -26.00 184.10   42 5.4      43
4 -17.97 181.66  626 4.1      19
> coplot(lat ~ long | depth, data = quakes)
> coplot(lat ~ long | mag, data = quakes)

# to illustrate how this plot works:
> x <- seq(1,20) # create a sequence
> x
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> y <- x + rnorm(20) # y = x + random N(0,1) residual
> plot(x,y)
> z <- c(rep(1,5), 3, rep(2, 5), rep(3, 4), rep(5,5))
> xyz.data <- data.frame(x = x, y = y, z = z)
> xyz.data[1:4,]
      x      y z
1  1  1.604621 1
2  2  2.840789 1
3  3  1.802540 1
4  4  3.496996 1

> coplot(y ~ x | z) # plot y against x conditional on z
> coplot(y ~ x | factor(z)) # plot y against x conditional on z as a 'factor'
>

# Quantile plot and histogram

```

```

> rain <- read.table('nzrain.dat', head=T)
> rain[1:4,]
  year mth  rain
1 1949   1  58.3
2 1949   2  82.9
3 1949   3  50.2
4 1949   4 143.9
> rain.mat <- matrix ( rain$rain, nr = 12 )
> hist(rain.mat[1,], main = "Histogram of Rainfall for January")
> hist(rain.mat[1,], main = "Histogram of Rainfall for January",
+ xlab = "rain / mm")
> qqplot(rain.mat[1,], rain.mat[7,])
> abline(0,1) # put straight line on plot
> qqplot(rain.mat[1,], rain.mat[7,], xlab='Jan rain / mm', ylab='July rain /mm')
> abline(0,1)
>
> qqnorm (rain.mat[7,]) # Normal quantile plot
> qqline(rain.mat[7,]) # Add line
> qqnorm (rain.mat[1,])
> qqline(rain.mat[1,])
>
# contour plot of Mt Eden volcano

> volcano <- read.table('volcano.dat', head=T)
> volcano <- as.matrix (volcano)
> x <- 10 * (1:nrow(volcano))
> y <- 10 * (1:ncol(volcano))
> contour(x,y,volcano)

> persp(x,y,volcano, xlab='easting / 10m', ylab = 'northing / 10m',
        zlab='height / m', theta=45)
> persp(x,y,volcano, xlab='easting / 10m', ylab = 'northing / 10m',
        zlab='height / m', theta=45, phi = 45)
> persp(x,y,volcano, xlab='easting / 10m', ylab = 'northing / 10m',
        zlab='height / m', theta=45, phi = 45, col="red")

```

4.2 Low-level plotting functions

These are used to add extra information to an existing plot, and include:

- `points()` – add points;
- `lines()` – add a line defined by two points;
- `abline()` – add a line defined by a slope and intercept;
- `text()` – add text;
- `legend()` – add a legend;

```
> weight.dat <- read.table ('weight.dat', head = T)
> attach(weight.dat)
> weight.dat[1:4,]
  sex weight rate
1  M   62.0 1792
2  M   62.9 1666
3  F   36.1  995
4  F   54.6 1425
> w <- split (weight, factor(sex))
> r <- split (rate, factor(sex))
> w
$F
[1] 36.1 54.6 48.5 42.0 50.6 42.0 40.3 33.1 42.4 34.5 51.1 41.2

$M
[1] 62.0 62.9 47.4 48.7 51.9 51.9 46.9

> w.max <- max(weight)
> w.min <- min(weight)
> r.max <- max(rate)
> r.min <- min(rate)

> plot (w$F, r$F, xlim = c(w.min, w.max), ylim = c(r.min, r.max),
+ xlab = 'weight', ylab = 'rate', pch = "F")
> points(w$M, r$M, pch = "M")
```

4.3 Interactive functions

These allow the user to add or retrieve information using the mouse, and include:

- `locator()` – returns the position that the mouse is ‘clicked’ on;
- `identify()` – returns the position of the nearest point when the mouse is clicked;

```
> identify(w$M, r$M)
> legend(locator(), c("Female", "Male"), pch=c("F", "M"))
> text(locator(), "Outlier ?") # to add text to a plot
```

4.4 Other parameters

The function `par()` can be used to change more detailed information. It is usual to assign a variable ‘oldpar’ to the existing ‘par’, so that the previous settings can be restored; for example,

```
> oldpar <- par() > par(col="red") # change the output colour to red
...produce some plots; then reset to old values – > par(oldpar)
```

Another useful function is `layout()`, which takes a matrix as a parameter and produces an equivalent matrix of plots on the *same* graphics device. For example, the command:

```
> layout(matrix(c(1,2)))
```

will produce two plots, one above the other – useful if you want to compare data on the same plot. The following would produce the plots side-by-side:

```
> layout(matrix(c(1,2), nc=2))
```

For example, to plot the ‘Chocolate, Beer, and Electricity’ data on one graphics image:

```
> cbe <- read.table('cbe.dat', head=T)
> cbe[1:4,]
  choc beer elec
```

```

1 1451 96.3 1497
2 2037 84.4 1463
3 2477 91.2 1648
4 2785 81.9 1595
> layout(matrix( c(1,2,3), nr = 3 ))
> attach(cbe)
> plot(choc)
> plot(beer)
> plot(elec)

> layout(matrix( c(1,1,2,3), nr = 2 ))
> plot(choc)
> plot(beer)
> plot(elec)

> layout(matrix( c(1,1,2,3), nr = 2, byrow = T ))
> plot(choc)
> plot(beer)
> plot(elec)

> layout( matrix( c(1,2,3), nr = 3) )
> par(col="red")
> plot(elec)
> par(col="chocolate")
> plot(choc)
> par(col="sandybrown")
> plot(beer)

```

4.5 Adding Maths Symbols

Use `expression` to add maths symbols to plots.

```

# function to show the sample mean is approximately Normal
# when the population is not Normal
# (the exponential distribution is used for the underlying population)
plot.rexp.mean <- function(popmean = 1, n = 20, repl=1000)
{

```

```

# generate a random sample of size n*repl
# from an exp(popmean, popsd) distribution
# and put result in an 'n' by 'repl' matrix x
x <- matrix( rexp(n*repl, rate=1/popmean), nr = n )
popsd <- popmean
# calculate the sample mean of the columns:
x.mean <- apply(x, 2, mean)
xvalues <- seq(min(x.mean), max(x.mean), len=1000)
# plot 'prob' histogram
hist.values <- hist(x.mean, fr=F, plot=F)
y.max <- max(c(hist.values$density, dnorm(xvalues, mean=popmean, sd=popsd/
hist(x.mean, fr=F, col='lightblue', main="", ylim = c(0, y.max),
      xlab='Sample Means')
title(main=expression(paste("Central Limit Theorem: Sample Mean ",
                             bar(X) %-%>% "N(", mu, " , ",
                             frac(sigma, sqrt(n)), ")",
                             " as n " %-%>% infinity))), cex.main=1.5)
title( sub=paste('(Random sample of size', as.character(n),
                 'taken from a Non-Normal population with mean',
                 as.character(popmean),')'))
# add Normal density curve
points (xvalues,
        dnorm(xvalues, mean=popmean,
              sd=popsd/sqrt(n)),
        type='l', lty=2, col='red')
list(mean = mean(x.mean), sd=sd(x.mean) )
}

```

5 Neural Networks

Notes:

- 'black box' function;
- similar to a regression model with a very large number of parameters;
- fitted by minimizing errors;

- potential problem of *over-fitting* \Rightarrow to terminate the fitting algorithm before ‘convergence’;
- usual to test the fitted model against data not used in the fitting procedure;
- R function for fitting a neural network is: `nnet()` in library ‘nnet’.

Neural Net example:

```
> library(nnet)

> x <- sort(rnorm(1000))
> x2 <- x^2
> y <- 2 * x + x2 + rnorm(1000)
> xy <- data.frame(x = x, y = y)
> plot(xy, col='yellow')

# start with linear model
> xy.lm <- lm(y ~ x + x2)
# add fitted values to plot:
> points(x, xy.lm$fit, type='l', col='blue')

# Now fit neural net:
> xy.nn <- nnet(y ~ x, size=10, linout = T, maxit=200)
# add fitted values to plot:
> points(x, xy.nn$fit, type='l', col='blue')

# there are usually curves in the predicted values
# indicating the model is over-parameterised

# Therefore terminate the algorithm earlier:
> xy.nn <- nnet(y ~ x, size=10, linout = T, maxit=100)
> plot(xy)
> points(x, xy.nn$fit, type='l', col='blue')
# should look better.

# Example of fitting nnet to a non-linear function:
> rm(list=ls())
> x <- 1:200
# define a non-linear function:
```

```

> f <- function(x) 10 + 100 * exp(-x/100)
# plot f against x:
> plot(x, f(x))
# use a lighter colour and a line:
> plot(x, f(x), type='l', col='lightgreen')

# make up a response variable based on f and random error:
> y <- f(x) + rnorm(200)

# plot x, y, and underlying function:
> plot(x,y, pch=4, col='lightgreen')
> points(x, f(x), col='blue', typ='l')

> xy <- data.frame( x = x, y = y )
# fit a polynomial:
> xy.lm <- lm(y ~ x + I(x^2) + I(x^3), data=xy)
# fit a neural net:
> xy.nn <- nnet(y ~ x, data=xy, size=5, linout = T)

# create some data for prediction:
> x.new <- 150:300
> new.dat <- data.frame( x = x.new )
# get predicted values:
> xy.lm.pred <- data.frame( x = x.new, y = predict(xy.lm, new.dat))
> xy.nn.pred <- data.frame( x = x.new, y = predict(xy.nn, new.dat))
# plot predicted and true values:
> plot(x.new, f(x.new), typ='l', col='lightgreen', ylim=c(0,50))
> points(xy.lm.pred, type='l', col='red')
> points(xy.nn.pred, type='l', col='blue')
# you should find the neural network does better than
# the regression model.

# Example of lm, rp, nn for air quality data:
> rm(list=ls())
> data(airquality)
> airquality[1:3,]
  Ozone Solar.R Wind Temp Month Day
1    41    190  7.4  67     5   1
2    36    118  8.0  72     5   2
3    12    149 12.6  74     5   3
> air.lm <- lm(Ozone ~ Solar.R + Wind + Temp, data=airquality)

```

```

> air.lm.pred <- predict(air.lm, data=airquality)
> air.rp <- rpart(Ozone ~ Solar.R + Wind + Temp, data=airquality)
> air.rp.pred <- predict(air.rp, data=airquality)
> air.nn <- nnet(Ozone ~ Solar.R + Wind + Temp, data=airquality, linout=T, size=
> air.nn.pred <- predict(air.nn, data=airquality)

# preparation for plot:
> layout(matrix(c(1,2,3), nc=1))
> x.max <- max(air.nn.pred, air.lm.pred, air.rp.pred)
> x.min <- min(air.nn.pred, air.lm.pred, air.rp.pred)
> st <- function(x) (x - mean(x))/sd(x) # func for standardising data

# plot standardised residuals against predicted values:
> plot(air.nn.pred, st(residuals(air.nn)), xlim=c(x.min, x.max),
      main='Res Plot for NN')
> plot(air.rp.pred, st(residuals(air.rp)), xlim=c(x.min, x.max),
      main='Res Plot for RP')
> plot(air.lm.pred, st(residuals(air.lm)), xlim=c(x.min, x.max),
      main='Res Plot for LM')

```