Data Mining with R

Further Data Structures

Hugh Murrell

Reference Books

These slides were created to accompany chapters 4 and 6 of the text:

- SPUR: *Scientific Programming and Simulation using R* Owen Jones, Robert Maillardet, and Andrew Robinson.

A version of this text can be found on the web.

text

Character strings are denoted using either double quotes or single quotes.

Strings can be arranged into vectors and matrices just like numbers.

We can also paste strings together using `paste(..., sep)`.

```
> x <- "Citroen SM"
> y <- "Jaguar XK150"
> z <- "Ford Falcon GT-HO"
> whishList <- paste(x, y, z, sep = ", ")
> whishList
```

```
[1] "Citroen SM, Jaguar XK150, Ford Falcon GT-HO"
```

formating text

The command `cat` displays concatenated character strings. The following example shows how to use formatted output to print a table of numbers.

```
> # input
> x <- 7
> n <- 5
> # display powers
> cat("Powers of", x, "\n")
```

```
Powers of 7
```

```
> cat("exponent result\n\n")
```

```
exponent result
```

formating continued...

```
> result <- 1
> for (i in 1:n) {
+   result <- result * x
+   cat(format(i, width = 8),
+       format(result, width = 10),
+       "\n", sep = "")
+ }
```

```
       1         7
       2        49
       3       343
       4      2401
       5     16807
```

You should try typing this program into an R script and then running it using `source("../scripts/powers.r")`

Input from a file

R provides a number of ways to read data from a file, the most flexible of which is the `scan` function which returns a vector.

`scan(file, what, n, sep, skip, quiet)`

`file` gives the file to read from.

`what` gives an example of the mode of data to be read.

`n` gives the number of elements to read.

`sep` allows you to specify the character that is used to separate values.

`skip` is the number of lines to skip before you start reading, default of 0.

`quiet` controls whether or not scan reports.

Quartiles

The p-th percentage point of a sample is defined to be the smallest sample point x such that at least a fraction p of the sample is less than or equal to x.

The first quartile is the 25% point of a sample, the third quartile the 75% point, and the median is the 50% point.

Computing Quartiles

```
> # Calculate median and quartiles of data in a file.
> file_name = "../data/data1.txt"
> data <- scan(file = file_name)
> # Calculations
> n <- length(data)
> data.sort <- sort(data)
> data.1qrt <- data.sort[ceiling(n/4)]
> data.med <- data.sort[ceiling(n/2)]
> data.3qrt <- data.sort[ceiling(3*n/4)]
> # Output
> cat("Quartiles... 1st:", data.1qrt,
+     ", median:", data.med,
+     ", 3rd:", data.3qrt, "\n")
```

```
Quartiles... 1st: 2 , median: 4 , 3rd: 7
```

Computing Quartiles (one liner)

As for many statistical operations, R has a built-in function for calculating quartiles.

Here is a solution to the same problem using the built-in function `quantile`.

```
> quantile(scan("../data/data1.txt"), (0:4)/4)
```

```
  0%  25%  50%  75% 100%
0.00 2.25 4.50 6.75 9.00
```

Output to a file

R provides a number of commands for writing output to a file. We will generally use `write` or `write.table` for writing numeric values and `cat` for writing text, or a combination of numeric and character values.

The command `write` has the form

```
write(x, file, ncolumns, append)
```

Here x is the vector to be written. If x is a matrix or array then it is converted to a vector (column by column) before being written.

Output to a file, continued...

Because write converts matrices to vectors before writing them, using it to write a matrix to a file can cause unexpected results. Since R stores its matrices by column, you should pass the **transpose** of the matrix to write if you want the output to reflect the matrix structure.

```
x <- matrix(1:24, nrow = 4, ncol = 6))
write(t(x), file = "out.txt", ncolumns = 6)
```

Here is what the file `out.txt` will look like:

```
1 5 9 13 17 21
2 6 10 14 18 22
3 7 11 15 19 23
4 8 12 16 20 24
```

Output to a file, continued...

There is also the very useful `dump`, which creates a text representation of almost any R object that can subsequently be read by `source`.

```
> x <- matrix(rep(1:5, 1:5), nrow = 3, ncol = 5)
> dump("x", file = "../results/x.txt")
> rm(x)
> source("../results/x.txt")
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    4    4    5
[2,]    2    3    4    5    5
[3,]    2    3    4    5    5
```

exotic data structures

In this lecture we introduce two new data structure, **factors** and **dataframes** which provide natural extentions to vectors and matrices respectively.

`factors` are vectors that allow categorical data

`dataFrames` are matrices whose columns can be vectors or factors. Unlike matrics, in a dataframe the modes of the columns can be different.

Factors

Statisticians typically recognise three basic types of variable:

- numeric,
- ordinal
- categorical

Both ordinal and categorical variables take values from some finite set, but the set is ordered for ordinal variables.

In an experiment one might grade the level of physical activity as low, medium, or high, giving an **ordinal** measurement.

An example of a **categorical** variable is hair colour.

The data type for ordinal and categorical vectors is `factor`.

The possible values of a `factor` are referred to as its `levels`.

Creating a factor

To create a factor in R we apply the function `factor` to some vector `x`.

By default the distinct values of `x` become the levels, or we can specify them using the optional `levels` argument.

This allows us to have more levels than just those in `x`, which is useful if we wish to change some of the values later.

We check whether or not an object `x` is a factor using `is.factor(x)`, and list its levels using `levels(x)`.

factor example...

```
> hair <- c("blond", "black", "brown", "brown",
+           "black", "gray", "none")
> is.character(hair)
[1] TRUE
> is.factor(hair)
[1] FALSE
> hair <- factor(hair)
> levels(hair)
[1] "black" "blond" "brown" "gray"  "none"
> table(hair)
hair
black blond brown  gray  none
    2     1     2     1     1
```

Note the use of the function `table` to calculate the number of times each level of the factor appears.

Ordered factors

To create an ordered factor we just include the option `ordered = TRUE` in the factor command.

In this case it is usual to specify the levels of the factor yourself, as that determines the ordering.

```
> phys.act <- c("L", "H", "H", "L", "M", "M")
> phys.act <- factor(phys.act,
+                    levels = c("L", "M", "H"),
+                    ordered = TRUE)
> is.ordered(phys.act)
```

```
[1] TRUE
```

```
> phys.act[2] > phys.act[1]
```

```
[1] TRUE
```

Dataframes

The defining characteristic of the vector data structure in R is that all components must be of the same mode.

To work with datasets from real experiments we need a way to group data of differing modes.

For example a forestry experiment in which we randomly selected a number of plots and then from each plot selected a number of trees. For each tree we measured its height and diameter (which are numeric), and also the species of tree (which is a character string).

Dataframes continued...

A **dataframe** is a list that is tailored to meet the practical needs of representing multivariate datasets.

It is a list of vectors restricted to be of equal length.

Each vector, or column, corresponds to a variable in an experiment, and each row corresponds to a single observation or experimental unit.

Each vector can be of any of the basic modes of object.

Dataframes continued...

Large dataframes are usually read into R from a file, using the function `read.table`, which has the form:

```
read.table(file, header, sep)
```

There are many more optional arguments, we have given the most important.

If a header is present, it is used to name the columns of the dataframe.

You can assign your own column names after reading the dataframe using the `names` function.

If there is no header then R uses the names "V1", "V2", etc.

Dataframes continued...

As well as reading in a dataframe from a file, we can construct one from a collection of vectors and/or existing dataframes using the function `data.frame`, which has the form:

```
data.frame(col1 = x1, col2 = x2, df1, df2 )
```

Here col1, col2, are the column names given as character strings without quotes. x1, x2, are vectors of equal length. df1, df2, are dataframes whose columns must be the same length as the vectors x1, x2. Column names may be omitted, in which case R will choose a name for you.

Lists

We have seen that a vector is an indexed set of objects. All the elements of a vector have to be of the same type numeric, character, or logical, which is called the mode of the vector.

Like a vector, a list is an indexed set of objects, but unlike a vector the elements of a list can be of different types, including other lists!

The mode of a list is list.

A list might contain an individual measurement, a vector of observations on a single response variable, a dataframe, or even a list of dataframes containing the results of several experiments. Dataframes are special kinds of lists.

In R lists are often used for collecting and storing complicated function output.

Lists continued...

A list is created using the `list(...)` command.

```
> my.list <- list("one", TRUE, 3, c("f","o","u","r"))
```

Single square brackets are used to select a sublist;

```
> my.list[2:3]
```

```
[[1]]
[1] TRUE
[[2]]
[1] 3
```

Double square brackets are used to extract a single element.

```
> my.list[[4]]
```

```
[1] "f" "o" "u" "r"
```

Lists continued...

The elements of a list can be named when the list is created, using arguments of the form name1 = x1, name2 = x2, etc.

Or they can be named later by assigning a value to the names attribute.

Unlike a dataframe, the elements of a list do not have to be named.

Names can be used when indexing with square brackets, or they can be used after a dollar sign to extract a list element.

To flatten a list x, that is convert it to a vector, we use unlist(x).

```
> x <- list(1, c(2, 3), c(4, 5, 6))
> unlist(x)
[1] 1 2 3 4 5 6
```

Lists continued...

Many functions produce list objects as their output.
For example, when we fit a least squares regression, the regression object itself is a list, and can be manipulated using list operations.

Least squares regression fits a straight line
`y = ax + b`
to a set of observations
`(xi, yi)`.

In R this can be achieved using the `lm` function,

```
> lm.xy <- lm(y ~ x, data=data.frame(x=1:5, y=1:5))
> mode(lm.xy)
```

```
[1] "list"
```

Lists continued...

```
> names(lm.xy)
 [1] "coefficients"  "residuals"     "effects"       "r
 [5] "fitted.values" "assign"        "qr"            "q
 [9] "xlevels"       "call"          "terms"         "m
```

we observe that `lm` returns a list:
the first element (called coefficients) is a vector giving
`a and b`;

the second element (called residuals) is a vector giving the
`yi - (xi + b)`

the third element (called fitted.values) is a vector giving the
`xi + b`

For a further insight into working with lists have a look at the *Australian Football League* example in chapter 6 of SPUR.

The apply family

R provides many techniques for manipulating lists and dataframes. In particular R has several functions that allow you to easily apply a function to all or selected elements of a list or dataframe.

`tapply` is a useful function that allows us to vectorise the application of a function to subsets of data. It has the form:

```
tapply(X, INDEX, FUN, ...)
```

`X` is the target vector to which the function will be applied;

`INDEX` is a `factor`, the same length as X, which is used to group the elements of `X`.

`FUN` is the function to be applied. It is applied to subvectors of `X` corresponding to a single `level` of `INDEX`.

The apply family continued...

To apply a function to a list we use either `sapply` or `lapply`.

`lapply(X, FUN, ...)`

applies the function `FUN` to each element of the list `X` and returns a list.

`sapply(X, FUN, ...)`

applies the function `FUN` to each element of `X`, which can be a list or a vector, and by default will try to return the results in a vector or a matrix, if this makes sense, otherwise in a list.

Extra parameters can be passed to `FUN` by way of the `...`.

The apply family continued...

As an example we consider tree growth data from the spuRs package

```
> treeg <- read.csv("treeg.csv",row.names=1)
> head(treeg)
```

```
  tree.ID forest habitat dbh.in height.ft age
1       1      4       5   14.6      71.4  55
2       1      4       5   12.4      61.4  45
3       1      4       5    8.8      40.1  35
4       1      4       5    7.0      28.6  25
5       1      4       5    4.0      19.6  15
6       2      4       5   20.0     103.4 107
```

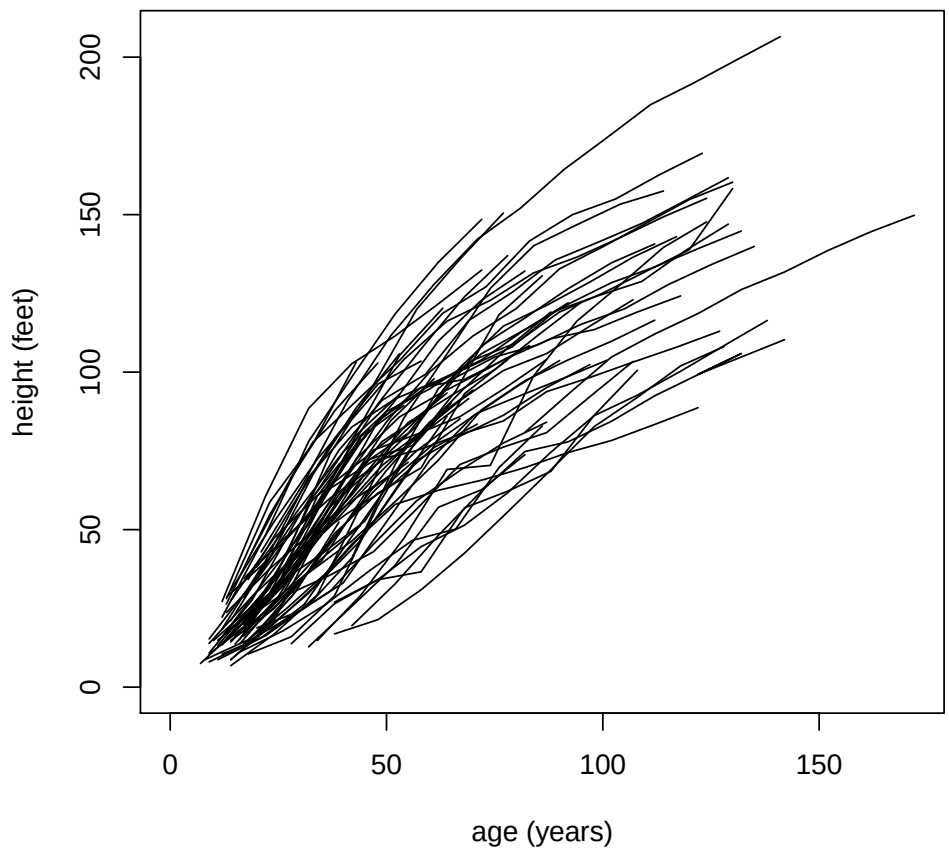
The apply family continued...

now we use tapply to map lines to data

```
> my.max <- function(x, i) max(x[[i]])
> max.age <- my.max(treeg,"age")
> max.height <- my.max(treeg,"height.ft")
> ages <- tapply(treeg$age,treeg$tree.ID,
+                identity)
> heights <- tapply(treeg$height.ft,treeg$tree.ID,
+                   identity)
> nt <- length(ages)
```

Graphics output with R's `plot` function.

```
> plot(c(0, max.age), c(0, max.height), type = "n",
+   xlab = "age (years)", ylab = "height (feet)")
> for (i in 1:nt) lines(ages[[i]], heights[[i]])
```



exercise

Write an R script to compute the average tree growth rate for each forest in the `treeg` dataset described in these slides and downloadable from the datasets section of my website.

You have to come up with a plan to compute growth rate for any particular tree and then you must average growth rates across all trees in a forest.

Your R script must present a table of average growth rates for each forest in the dataset.

exercise

e-mail your solution to me as a single R script by:

6:00 am, Monday the 7th March.

Make sure that the first line of your R script is a comment statement containing your name, student number and the week number for the exercise. (in this case week 03).

Use: `dm03-STUDENTNUMBER.R` as the filename for the script.

Use: `dm03-STUDENTNUMBER` as the subject line of your email.

There will be no extensions. No submission implies no mark.