Data Programming with R

Isabella Gollini

Lecture 3 - Lists and Data Frames

Lists and Data Frames

- Creating lists
- Accessing, adding and deleting elements of a list
- Using `lapply()` and `sapply()`
- Creating data frames
- Merging data frames
- Applying functions to data frames

Why are we studying this?

- Lists and data frames allow us to store and therefore analyse more complex forms of data.
- In a list we can store objects of different types, e.g. numbers, text, vectors, matrices, etc and they can have different lengths or dimensions.
- A data frame is more like a matrix, except that each column can have a different type.
- We will learn how to create, manipulate and use these objects when analysing data.

Lists

- Creating lists
- List operations
- Accessing components
- Applying functions to lists

Creating lists

A list can be created via the `list` function:

```
j <- list(name = 'Joe', salary = 55000, union = T)
```

Printing this list to the console gives:

```
j
```

```
## $name
## [1] "Joe"
##
## $salary
## [1] 55000
##
## $union
## [1] TRUE
```

- Here, `name`, `salary` and `union` are optional tags. Without them the components would be named `[[1]]`, `[[2]]`, and `[[3]]`.
- I recommend using tags wherever possible.

List indexing

You can access a list component in (at least) 3 different ways:

```
j$salary
j[['salary']]
j[[2]]
```

- Each of these will return `55000`.
- The `$` notation allows us to access list elements by name.
- Note that not using double square brackets, e.g. `j['salary']` will return a list rather than a number.
- Because lists are also really vectors, commands such as `j[1:2]` are allowed and will give the first two elements of the list.

Adding and deleting list elements

It is easy to add a new component after the list has been created:

```
j$sales <- c(10400, 12300, 13700)
```

We can also do this via a vector index:

```
j[[5]] <- 'Manager'
```

Or delete it by setting it to `NULL`:

```
j[[5]] <- NULL
```

Since a list is a vector, you can get the number of tags in it with length:

```
length(j)
length(j$sales)
```

Note that `length(j)` gives the number of tags in the list, not the number of raw elements.

Accessing list components

To find out what tags are in a list is we can use the functions **`names`** or **`str`**:

```
names(j)
```

```
## [1] "name"   "salary" "union"  "sales"
```

```
str(j)
```

```
## List of 4
##  $ name  : chr "Joe"
##  $ salary: num 55000
##  $ union : logi TRUE
##  $ sales : num [1:3] 10400 12300 13700
```

The function **`unlist`** will convert a list into a vector using the mode of the lowest common denominator (here, that is the character mode).

```
unlist(j)
```

```
##    name  salary   union  sales1  sales2  sales3
##   "Joe" "55000"  "TRUE" "10400" "12300" "13700"
```

By default R will give names to an unlisted object taken from the tags. We can remove them via the **`unname`** command or giving an extra argument:

```
unlist(j, use.names = FALSE)
```

```
## [1] "Joe"   "55000" "TRUE"  "10400" "12300" "13700"
```

Applying functions to lists

The family of `apply` functions (which we first met for matrices) has a version for lists known as `lapply`, e.g.

```
lapply(list(1:3, 25:29), median)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 27
```

This applies the function `median` to each component of the given list and returns the answers in a list `sapply` (for simplified `apply`) produces the exact same output as a vector:

```
sapply(list(1:3, 25:29), median)
```

```
## [1]  2 27
```

Flexible lists

There's nothing to stop you having a list within a list:

```
list(a = 1, b = 2, c = list(d = 3, e = 4))
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## $c$d
## [1] 3
##
## $c$e
## [1] 4
```

. . . or a matrix within a list, and so on:

```
list(a = 1, b = matrix(c(1, 2, 3, 4), ncol = 2, nrow = 2))
```

Extended example – Text concordance

- Suppose we had a block of text like this:

```
"It was a bright cold day in April, and the clocks were striking thirteen.
Winston Smith, his chin nuzzled into his breast in an effort to escape the vile
wind, slipped quickly through the glass doors of Victory Mansions, though not quickly
enough to prevent a swirl of gritty dust from entering along with him."
```

- Suppose we want to create a function which lists the different words used and their position in the text. For example, `his` is used twice in positions 17 and 21.
- To keep this simple we will first remove all the non-letter characters and we save it into a text object:

```
firstpar <- "It was a bright cold day in April and the clocks were striking thirteen
  Winston Smith his chin nuzzled into his breast in an effort to escape the vile wind
  slipped quickly through the glass doors of Victory Mansions though not quickly enough
  to prevent a swirl of gritty dust from entering along with him"
```

`findwords` function

A suitable function would be:

```
findwords <- function(tf) {
  txt <- unlist(strsplit(tf, ' ')) # Read in the words from the text and separate into a vector
  wl <- list() # Create a list to store the words and their positions
  for(i in 1:length(txt)) { # Loop through each word
    wrd <- txt[i]  # Get the current word
    wl[[wrd]] <- c(wl[[wrd]], i) # Add its position to the list with the appropriate tag
    }
  return(wl) # Return the answer as a list
}
```

```
head(findwords(firstpar), 4)
```

```
## $It
## [1] 1
##
## $was
## [1] 2
##
## $a
## [1]  3 52
##
## $bright
## [1] 4
```

What else can we do?

- What if we wanted to find the words in alphabetical order rather than the order they appear in the text? We could sort them using a simple function:

```
alphawl <- function(wrdlst) {
  nms <- names(wrdlst) # Find the tags of the list
  sn <- sort(nms) # Sort them alphabetically
  return(wrdlst[sn]) # Return them
  }
alphawl(findwords(firstpar))
```

Or we could sort by frequency:

```
freqwl <- function(wrdlst) {
  freqs <- sapply(wrdlst, length) # Find the freq of each word
  return(wrdlst[order(freqs)]) # Return them in order
}
freqwl(findwords(firstpar))
```

- See the screencast for detailed explanation of these functions

Data Frames

Creating and accessing data frames Extracting data More on `NA` values - `rbind` and `cbind` - `apply` Merging data frames

Creating data frames

- A data frame is just like a matrix, except that each column can have a different mode.

A simple example:

```
kids <- c('Jack', 'Jill')
ages <- c(12, 10)
d <- data.frame(kids, ages)
d
```

```
##   kids ages
## 1 Jack   12
## 2 Jill   10
```

- By default R will turn strings (here `kids`) into factors which are another data type. We will cover factors later in the module.

Accessing data frames

Though it looks like a matrix, R treats data frames as lists:

```
mode(d)
```

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

As such, we can access it in exactly the same way:

```
d[[2]]
```

```
## [1] 12 10
```

```
d$kids[1]
```

```
## [1] Jack
## Levels: Jack Jill
```

The `str` function will give us slightly more detail:

```
str(d)
```

```
## 'data.frame':    2 obs. of  2 variables:
##  $ kids: Factor w/ 2 levels "Jack","Jill": 1 2
##  $ ages: num  12 10
```

Extracting parts

- Whilst R treats a data frame like a list, it also allows us to extract and filter in the same way as if it were a matrix.

For example, using the `women` data set:

```
data(women)
str(women)
```

```
## 'data.frame':    15 obs. of  2 variables:
##  $ height: num  58 59 60 61 62 63 64 65 66 67 ...
##  $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

```
women[1:2,]
```

```
##   height weight
## 1     58    115
## 2     59    117
```

```
women[women$height > 70,]
```

```
##    height weight
## 14     71    159
## 15     72    164
```

Dealing with NA values

In week 2, we met the `subset` function which allows us to select parts of a vector or matrix. It will similarly work on a data frame:

```
subset(women, weight < 130)
```

- The advantage of this command is that we don't have to include `women$weight < 130` in the second argument
- `subset` also automatically removes `NA` values

A more useful function for removing `NA` values in a data frame is `complete.cases` which will remove any rows which have `NA`s in them:

```
women[1,1] <- women[13,2] <- NA
women[complete.cases(women),]
```

The first line turns some of the values into `NA`s, the second line removes them

rbind and cbind

The `rbind` and `cbind` functions introduced in week 2 for matrices also work for data frames, provided the dimensions match:

```
women2 <- cbind(women, letters[1:15])
women3 <- rbind(women, c(73, 166))
women4 <- cbind(women, women$height * 2.54, women$weight * 0.45)
names(women4) <- c('heightin', 'weightlbs', 'heightcm', 'weightkg')
```

The last command here is useful because the default names given by R to the data frame columns can be a bit untidy.

A neater way may have simply been to write:

```
women$heightcm <- women$height * 2.54
```

Merging data frames

It is very common to come across two data frames that need to be merged together in some way. The R function for this is `merge(x, y)` where `x` and `y` are two data frames with at least one common column name. For example:

```
d1 <- data.frame(kids = c('Jack', 'Jill', 'Jillian', 'John'),
  county = c('Dublin', 'Cork', 'Donegal', 'Kerry'))
d2 <- data.frame(ages = c(10, 7, 12),
  kids = c('Jill', 'Lillian', 'Jack'))
merge(d1, d2)
```

```
##   kids county ages
## 1 Jack Dublin   12
## 2 Jill   Cork   10
```

- The `merge` function has extra arguments `by.x` and `by.y` to tell R which variables over which to merge.
- See the screencast for more details on `merge`.

Applying functions to data frames

Remember that R treats data frames like a list, so calling `apply` on a data frame with a function `f` will evaluate the function on each of the data frame's columns, with the return value given as a list.

```
lapply(d,sort)
```

```
## $kids
## [1] Jack Jill
## Levels: Jack Jill
##
## $ages
## [1] 10 12
```

To get it back into a data frame, we could use:

```
data.frame(lapply(d, sort))
```

```
##   kids ages
## 1 Jack   10
## 2 Jill   12
```

Note that this has broken the row structure of the original data frame! (Jack was 12, now he is 10).

Extended example

Risk factors associated with low infant birth weight via logistic regression

Logistic regression

- Logistic regression is a statistical method used to predict whether a binary variable is 0 or 1 based on other explanatory variables.
- It is used very widely in classification problems.
- If the binary variable is called Y and the explanatory variable X, then the model can be written as follows:

$$P(Y = 1|X = x) = \frac{1}{1 + \exp[-(\beta_0 + \beta_1 x)]}$$

- Here the unknown parameters are β_0 and β_1.
- We will estimate them using the R function `glm` with the argument `family = 'binomial'`.

The data set `birthwt`

The `birthwt` data set comes with the R package `MASS`. We can load it in with:

```
library(MASS)
```

and explore it with:

```
help(birthwt)
str(birthwt)
```

- From the above commands we can see that there are 10 columns, the first of which (`low`) represents the binary response variable Y which is the presence of low birth weight in a child.
- The other columns represent potential explanatory variables. We will drop the variables `race` and `bwt` as the former is a factor (which we cover later in the module) and the latter is the true birth weight so would give us perfect predictions.

```
birthwt2 <- birthwt[,-c(4, 10)]
```

Running a logistic regression

Let's now write a function that fits a logistic regression model using each of the remaining explanatory variables in turn:

```
logfun <- function(x) {
    glm(birthwt2$low ~ x, family = binomial)$coef
}
sapply(birthwt2[,-1], logfun)
```

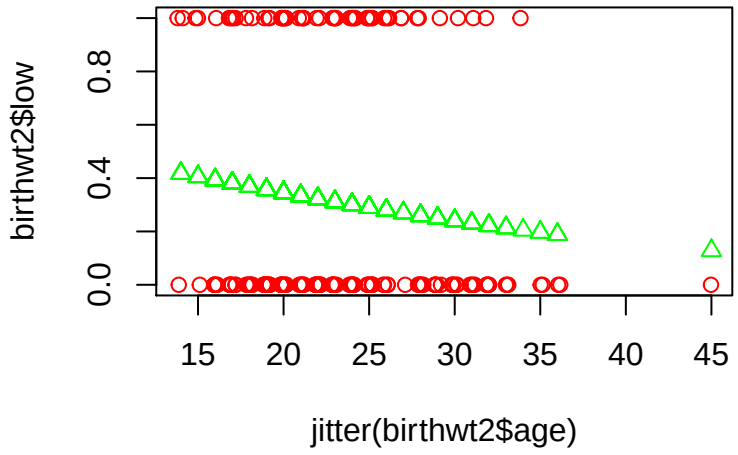
```
##                      age         lwt      smoke        ptl        ht
## (Intercept)  0.38458192  0.99831432 -1.0870515 -0.9641890 -0.877070
## x           -0.05115294 -0.01405826  0.7040592  0.8018058  1.213542
##                      ui        ftv
## (Intercept) -0.9469277 -0.6867585
## x            0.9469277 -0.1351199
```

- The top row here contains values of `beta0`, whilst the second row contains values of `beta1`.
- Notice that some of the coefficients of `x` are negative, indicating that, for example, as age increases the probability of a low birth weight goes down.

Plotting the output

Let's see the relationship between age and low birth weight in a plot:

```
mod <- glm(birthwt2$low ~ birthwt2$age, family = binomial)$coef
plot(jitter(birthwt2$age), birthwt2$low, col = "red")
points(birthwt2$age, 1 / (1 + exp(-(mod[1] + mod[2] * birthwt2$age))),
  pch = 2, col = "green")
```



Lessons from this week

- Creating and editing lists and data frames. Many of the same ideas as vectors, matrices and arrays. Useful for storing mixed-mode data.
- A few ways to deal with `NA` values, and the difference between `NA` and `NULL`.
- More useful functions (`apply`, `merge`,...)
- Logistic regression very useful for classification - more on this later.