CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 11

Another Use for Classes — Factors

Recall that how R handles an object can be changed by giving it a "class" attribute. That's how lists become data frames. Another example is the "factor" class, which is used to represent a vector of strings as a vector of integers, along with a vector of just the *distinct* string values.

Here's an illustration:

```
> a <- as.factor(c("red","green","yellow","red","green","blue","red"))
> a
[1] red    green  yellow red    green  blue   red
Levels: blue green red yellow
> class(a)        # We can see that this object has the class "factor"
[1] "factor"
> unclass(a)      # Here's what it is without its class attribute
[1] 3 2 4 3 2 1 3
attr(,"levels")
[1] "blue"   "green"  "red"    "yellow"
```

The main reason factors exist is that an integer previously used less memory than a string, though this is less true in recent versions of R. Strings are converted to factors in `read.table`, unless you use the `stringsAsFactors=FALSE` option.

Operations on Factors

Factors look like strings for many purposes:

```
> a <- as.factor(c("red","green","yellow","red","green","blue","red"))
> a == "red"
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Even though factors are represented as integers, mathematical operations on them are not allowed:

```
> sqrt(a)
Error in Math.factor(a) : sqrt not meaningful for factors
```

This is because the integers representing the "levels" of the factor are arbitrary, so treating them like numbers would be misleading. (Unfortunately, R isn't completely consistent in this, and will sometimes use a factor as a number without a warning.)

Another Use of Classes — Dates and Time Differences

R also defines classes for dates, and for differences in dates. Some of what you can do with these is illustrated below:

```
> d1 <- as.Date("2015-03-24")  # d1 will be an object of class "Date"
> d1
[1] "2015-03-24"     # Adding an integer to a date gives a new date
> d1+2
[1] "2015-03-26"
> d1+10              # Addition will automatically change the month
[1] "2015-04-03"
>
> d2 <- as.Date("2015-02-24")
> d1-d2              # The difference has class "difftime"
Time difference of 28 days
> as.numeric(d1-d2)  # We can convert a "difftime" object to a number
[1] 28
```

Defining Your Own Classes

You can attach a class attribute of your choice to any object. If that's all you do, the object gets handled just as before, except the class attribute is carried along:

```
> x <- 9
> class(x) <- "mod17"
> x + 10
[1] 19
attr(,"class")
[1] "mod17"
```

But you can now redefine some operations (ones that are "generic") to operate specially on your class:

```
> `+.mod17` <- function (a,b) {
+     r <- (unclass(a) + unclass(b)) %% 17
+     class(r) <- "mod17"
+     r
+ }
> x + 10
[1] 2
attr(,"class")
[1] "mod17"
```

Defining Your Own Generic Functions

You can also create new generic functions, that you can define "methods" for, that are used when they are called with objects of particular classes. For example:

```
> picture <- function (x) UseMethod("picture")
> picture.default <- function (x) cat(x,"\n")
> picture.mod17 <- function (x) cat(rep("-",x),"X",rep("-",16-x),"\n")
> picture(9)
9
> picture(x)
- - - - - - - - - X - - - - - - -
> picture(x+3)
- - - - - - - - - - - - X - - - -
```

The definition of `picture` just says it's generic. If no special method is defined for a class, `picture.default` is used. By defining `picture.mod17`, we create a special method for class `mod17`. R finds the method to use based on the class of the first argument to the generic function.

The Object-Oriented Approach to Programming

R's classes are designed to support what is called "object-oriented" programming.

This approach to programming has several goals:

- Allow manipulation of "objects" without having to know exactly what kind of object you're manipulating — as long as the object can do the things that you need to do (it has the right "methods").

  Benefit: We can write one just function for all objects, not many functions, that all do the same thing but in somewhat different ways.

- Separate *what* the methods for an object do from *how* they do it (including how the object is represented).

  Benefit: We can change how objects work without having to change all the functions that use them.

- Permit the things that can be done with objects ("methods") and the kinds of objects ("classes") to be extended without changing existing functions.

  Benefit: We can more easily add new facilities, without having to rewrite existing programs.

Generic Functions for Drawing, Rescaling, and Translating

Let's see how we can define a set of generic functions for drawing and transforming objects like circles and boxes.

We start by setting up the generic functions we want:

```
draw <- function (w) UseMethod("draw")
rescale <- function (w,s) UseMethod("rescale")
translate <- function (w,tx,ty) UseMethod("translate")
```

Then we need to define methods for these generic functions for all the classes of objects we want. We also need functions for creating such objects.

Note: We might not have done things in this order. For example, we might have first defined only `draw` and `translate` methods, and then later added the `rescale` method. We would then need to implement a `rescale` method for a class only if we actually will use `rescale` for objects of that class.

Implementing a Circle Object

We'll represent a circle by the x and y coordinates of its centre and its radius.

```
new_circle <- function (x, y, r) {
    w <- list (centre_x=x, centre_y=y, radius=r)
    class(w) <- "circle"
    w
}
draw.circle <- function (w) {
    angles <- seq (0, 2*pi, length=100)
    lines (w$centre_x + w$radius*cos(angles),
           w$centre_y + w$radius*sin(angles))
}
rescale.circle <- function (w,s) {
    w$radius <- w$radius * s;
    w
}
translate.circle <- function (w,tx,ty) {
    w$centre_x <- w$centre_x + tx; w$centre_y <- w$centre_y + ty
    w
}
```

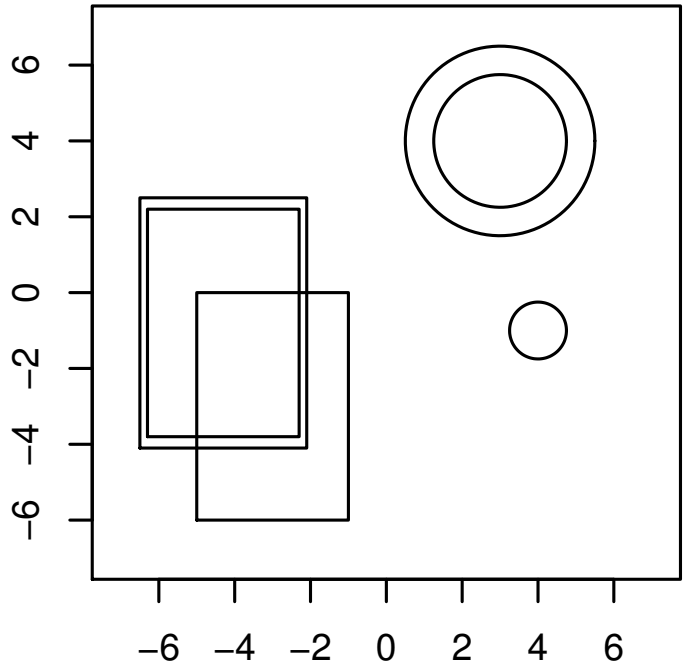
Implementing a Box Object

We'll represent a box by the x and y coordinates at its left/right top/bottom.
But to create a box we'll give coordinates for its centre and offsets to the corners.

```
new_box <- function (x, y, sx, sy) {
    w <- list (x1=x-sx, x2=x+sx, y1=y-sy, y2=y+sy)
    class(w) <- "box"
    w
}
draw.box <- function (w) {
    lines (c(w$x1,w$x1,w$x2,w$x2,w$x1), c(w$y1,w$y2,w$y2,w$y1,w$y1))
}
rescale.box <- function (w,s) {
    xm <- (w$x1+w$x2) / 2
    w$x1 <- xm + s*(w$x1-xm); w$x2 <- xm + s*(w$x2-xm)
    ym <- (w$y1+w$y2) / 2
    w$y1 <- ym + s*(w$y1-ym); w$y2 <- ym + s*(w$y2-ym)
    w
}
translate.box <- function (w,tx,ty) {
    w$x1 <- w$x1 + tx; w$x2 <- w$x2 + tx
    w$y1 <- w$y1 + ty; w$y2 <- w$y2 + ty
    w
}
```

An Example of Drawing Objects This Way

```
> plot(NULL,xlim=c(-7,7),ylim=c(-7,7),xlab="",ylab="",asp=1)
> c <- new_circle(3,4,2.5)
> draw(c); draw(rescale(c,0.7)); draw(translate(rescale(c,0.3),1,-5))
> b <- new_box(-3,-3,2,3)
> b2 <- translate(b,-1.3,2.2)
> draw(b); draw(b2); draw(rescale(b2,1.1))
```



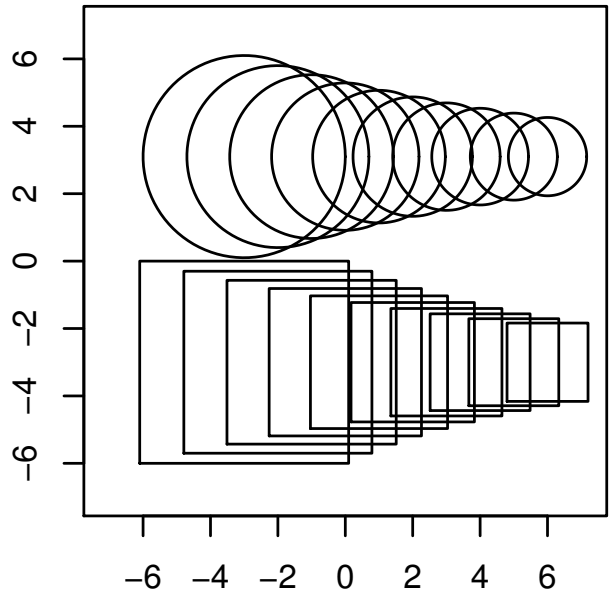
Defining a Function That Works On Both Circles and Boxes

Here is a function that should work for circles, boxes, or any other class of object that has `draw`, `rescale`, and `translate` methods:

```
smaller <- function (w, n)
    for (i in 1:n) { draw (w); w <- rescale(translate(w,1,0),0.9) }
```

Here are two uses of it:

```
> plot(NULL,xlim=c(-7,7),ylim=c(-7,7), xlab="",ylab="",asp=1)
> smaller (new_circle(-3,3.1,3),10)
> smaller (new_box(-3,-3,3.1,3),10)
```



Statistical Facilities in R

In this course, we've mostly looked at R as a programming language, and at general programming concepts.

But R is most popular as a language for statistical applications. So it has many special facilities for doing statistics. I'll talk about some now.

Don't worry if you don't understand some of the statistical concepts — that's OK for this course. Though learning about R's statistical facilities is one good way to learn statistics in a hands-on way!

Creating Tables of Counts

R can count how many times a value or combination of values occurs in a data set, with the `table` function. It returns an object of class `table`, which looks like a vector or matrix of integer counts.

For a vector, `table` counts how many times each unique value occurs:

```
> colours <- c("red","blue","red","red","green","blue")
> print (tcol <- table(colours))
colours
 blue green   red
    2     1     3
> names(tcol)
[1] "blue"  "green" "red"
> ages <- c(4,9,12,2,4,9,10)
> print (tage <- table(ages))
ages
 2  4  9 10 12
 1  2  2  1  1
> names(tage)
[1] "2"  "4"  "9"  "10" "12"
```

Tables of Joint Counts

When used with two vectors, or a data frame with two columns, `table` creates a two-dimensional table of how often each combination of values occurs. Examples:

```
> colours <- c("red","blue","red","red","green","blue")
> shapes <- c("round","round","square","square","square","round")
> table(colours,shapes)
        shapes
colours round square
  blue      2      0
  green     0      1
  red       1      2
> df <- data.frame(col=colours,shape=shapes)
> table(df)
        shape
col      round square
  blue      2      0
  green     0      1
  red       1      2
```

Statistical Modeling in R

One big part of statistics is fitting a *model* to data. R has many functions for doing this, but I'll mention only `lm`, which fits a linear model.

Models in R are often specified using *formulas*, that say how one thing is modelled in terms of other things.

For `lm`, we want to specify that some *response* variable is modelled as a linear combination (plus noise) of some *explanatory* variables. This is done using a formula such as

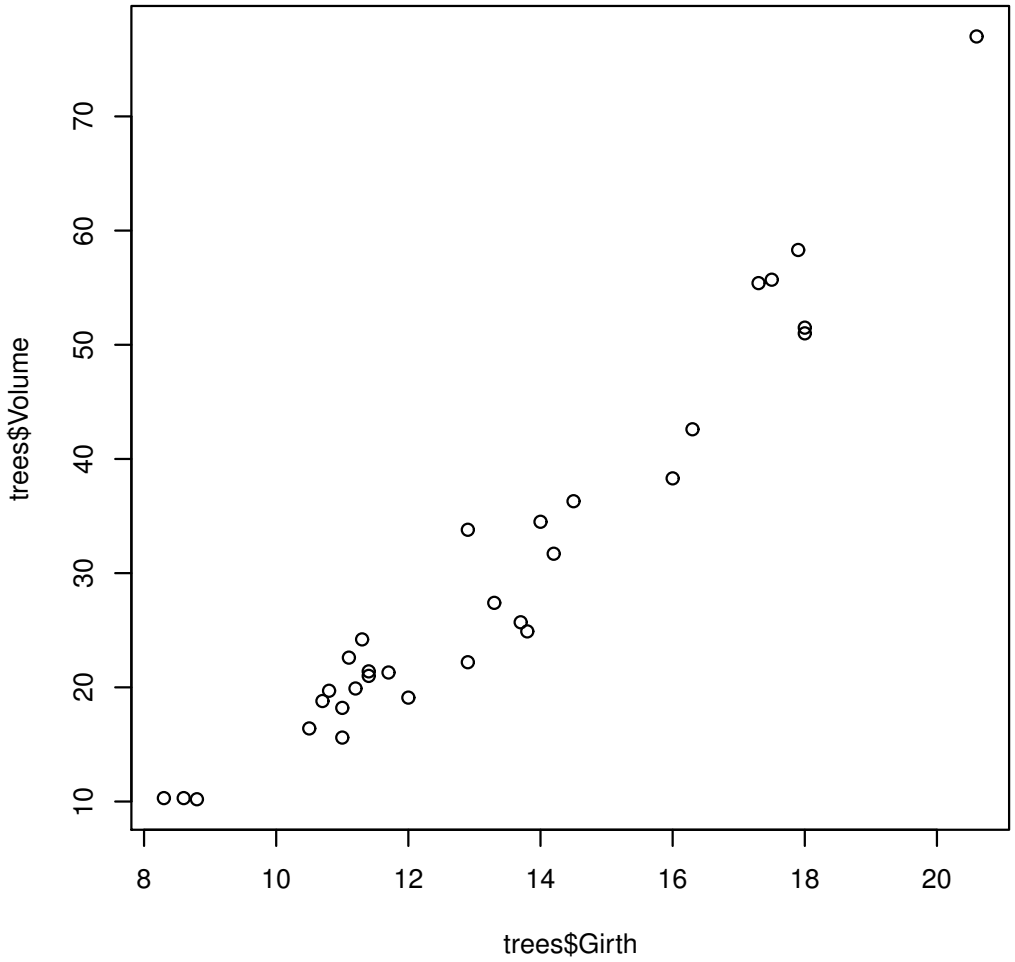
```
growth ~ ave_temp + fertilizer + variety
```

This might express that the amount by which some plant grows is linearly related to the average temperature, the amount of fertilizer used, and a set of indicator variables indicating the variety of the plant.

A Simple Example of a Linear Model

Here, I'll show the results of a very simple linear model, relating the volume of wood in a cherry tree to its girth (diameter of trunk). The data is in the data frame `trees` that comes with R.

Here's a plot of the data:



Fitting the Model with `lm`

We can fit a linear model for volume given girth as follows:

```
> lm (trees$Volume ~ trees$Girth)


Call:
lm(formula = trees$Volume ~ trees$Girth)


Coefficients:
(Intercept)  trees$Girth
    -36.943        5.066
```

The result says that best fit model for the volume is

$$\text{Volume} \;=\; -36.943 \;+\; 5.066\,\text{Girth} \;+\; \text{noise}$$

We can get the same result with an abbreviated formula by saying the data comes from the data frame `trees`:

```
lm (Volume ~ Girth, data=trees)
```

Using the Result of `lm`

The value returned by `lm` is an object of class `"lm"`, which has special methods for printing and other operations.

We can save the result, and then get the regression coefficients with `coef`.

```
> m <- lm (Volume ~ Girth, data=trees)
> coef(m)
(Intercept)       Girth
 -36.943459    5.065856
```

We could use these coefficients to predict the volume for a new tree, with girth of 11.6:

```
> coef(m) %*% c(1,11.6)   # %*% will compute the dot product
         [,1]
[1,] 21.82048
```

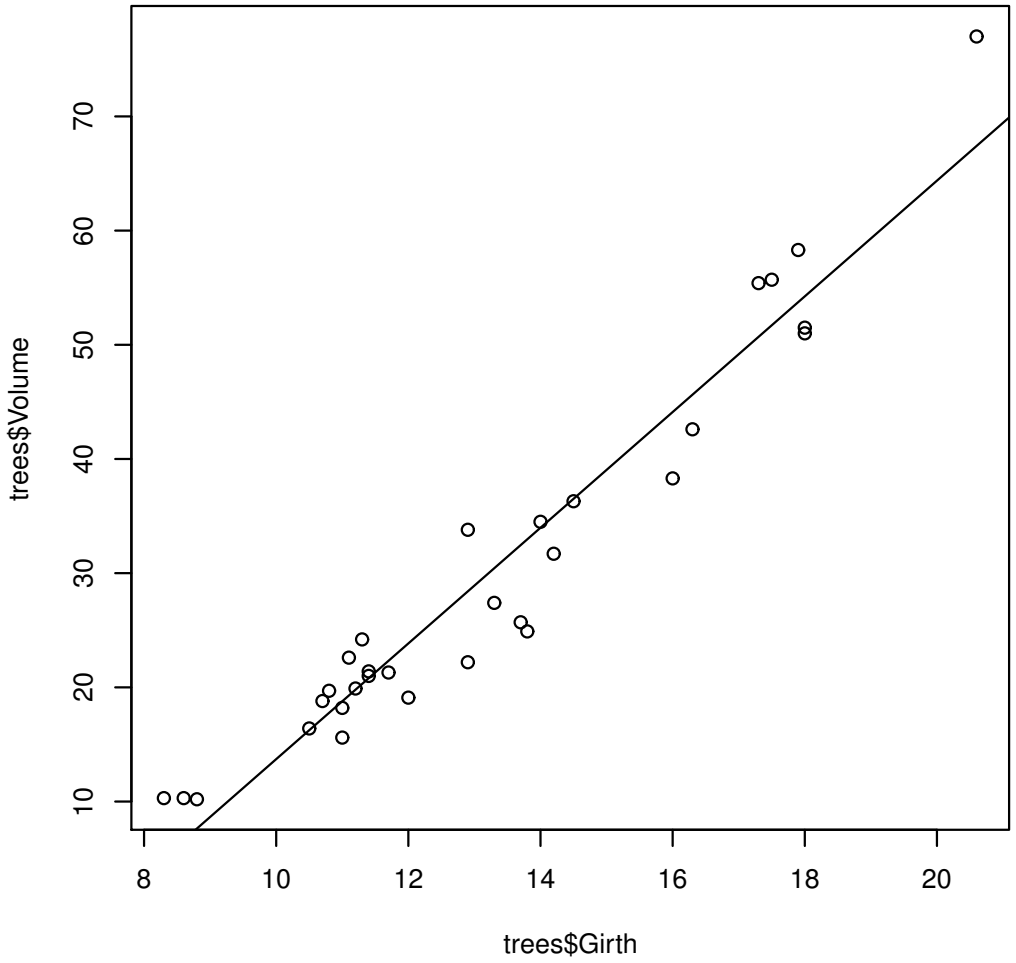
Getting More Details on the Model Fitted

We can also ask for more statistical details with `summary`:

```
> summary(m)

Call:
lm(formula = Volume ~ Girth, data = trees)

Residuals:
   Min     1Q Median     3Q    Max
-8.065 -3.107  0.152  3.495  9.587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
Girth         5.0659     0.2474   20.48  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.252 on 29 degrees of freedom
Multiple R-squared: 0.9353,    Adjusted R-squared: 0.9331
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

Plotting the Regression Line

We can also plot the regression line from the fitted model on top of a scatterplot of the data, using `abline(m)`:



The plot shows some indication that the relationship is actually curved.

Trying a Quadratic Model

Let's try fitting volume to both girth and the square of girth:

```
> Girth_squared <- trees$Girth^2
> summary (lm (trees$Volume ~ trees$Girth + Girth_squared))
Call:
lm(formula = trees$Volume ~ trees$Girth + Girth_squared)

Residuals:
    Min      1Q  Median      3Q     Max
-5.4889 -2.4293 -0.3718  2.0764  7.6447

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   10.78627   11.22282   0.961 0.344728
trees$Girth   -2.09214    1.64734  -1.270 0.214534
Girth_squared  0.25454    0.05817   4.376 0.000152 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.335 on 28 degrees of freedom
Multiple R-squared: 0.9616,    Adjusted R-squared: 0.9588
F-statistic: 350.5 on 2 and 28 DF,  p-value: < 2.2e-16
```

Some Useful Functions of Vectors

The `unique` function returns a vector of unique values:

```
> colours <- c("red","blue","red","red","green","blue")
> unique(colours)
[1] "red"   "blue"  "green"
```

The `sort` function sorts a vector in increasing order (or decreasing order if you use `decreasing=TRUE`):

```
> ages <- c(4,9,12,2,4,9,10)
> sort(ages)
[1]  2  4  4  9  9 10 12
> sort(unique(ages),decreasing=TRUE)
[1] 12 10  9  4  2
```

The `which.min` and `which.max` functions give the index of the smallest and largest elements in a vector (first occurrence if they occur more than once):

```
> which.min(ages)
[1] 4
> which.max(ages)
[1] 3
```

Checking if Things are in a Set

The `%in%` operator checks whether values are in some set of values (represented by a vector of values in the set):

```
> colours <- c("red","blue","red","red","green","blue")
> "black" %in% colours
[1] FALSE
> colours %in% c("red","green")
[1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE
```

You can use the results to find the elements of a vector that are in some set:

```
> colours [ colours %in% c("red","green") ]
[1] "red"   "red"   "red"   "green"
```

With `which`, which returns indexes of `TRUE` in a logical vector, you can also find the indexes of the elements that are in the set:

```
> which (colours %in% c("red","green"))
[1] 1 3 4 5
```