STATISTICS: AN INTRODUCTION USING R

By M.J. Crawley

Exercises

1. PLOTS: GRAPHICAL METHODS OF DATA EXPLORATION

Producing high quality graphs is one of the main reasons for doing statistical computing. There are two fundamentally different kinds of explanatory variables, continuous and categorical, and each of these leads to a completely different sort of graph. In cases where the explanatory variable is continuous, like length or weight or altitude, the appropriate plot is a **scatterplot**. In cases where the explanatory variable is categorical, like genotype or colour or gender, then the appropriate plot is a **boxplot**.

Plotting with Continuous Explanatory Variables: Scatterplots

The first thing to learn is that **allocation** in SPlus is done using "gets" rather than "equals". "Gets" is a composite symbol <- made up from a 'less than symbol' < and a 'minus symbol' - . Thus, to make a vector called x that contains the numbers 1 through 10, we just type:

```
x<- 1:10
```

which is read as saying "x gets the values 1 to 10 in series". The colon is the series-generating operator. Now if you type **x** you will see the contents of the vector called x

```
x
 [1]  1  2  3  4  5  6  7  8  9 10
```

The [1] at the beginning of the row just means that this is the first (and only) list within the object called x.

Some general points are worth making here:

- variable names are case-sensitive so x is not the same as X
- variable names should not begin with numbers (e.g. 1x) or symbols (e.g. %x)
- variable names should not contain breaks: use back.pay not back_pay or back pay
- for help with commands just type ? followed by the name you want help with

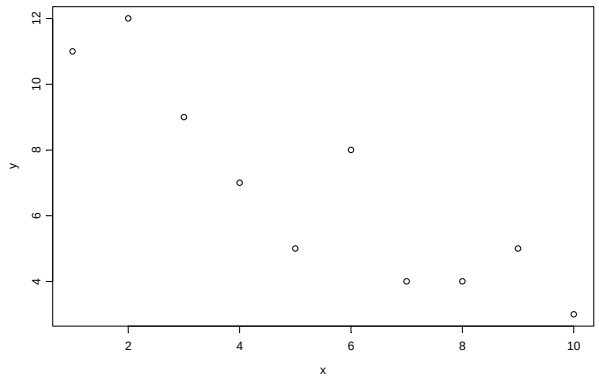
Now we type in the values for the response variable, y. The simplest way to type numbers into a vector is to use the **concatenate** directive "**c**". Concatenation is simply the process of joining lists together, and we shall meet it in many different circumstances.

```
y<-c(11,12,9,7,5,8,4,4,5,3)
```

y is called the **response variable** and is plotted on the vertical axis of a graph; x is called the **explanatory variable** and is plotted on the horizontal axis of a graph. It is extremely simple to obtain a scatter plot of y against x in SPlus.

The **plot** directive needs only 2 arguments: first the name of the explanatory variable (x in this case), and second the name of the response variable (y in this case)

```
plot(x,y)
```



For the purposes of data exploration, this may be all you need. But for publication it is useful to be able to change the appearance of the plot. It is often a good idea to have a longer, more explicit label for the axes than is provided by the variable names that are used as default options (x and y in this case). Suppose we want to change the label "x" into the longer label "Explanatory variable". To do this we use the **xlab** directive with the text of the label enclosed within double quotes. Use the Up Arrow to get back the last command line, and insert a comma after y, then type xlab= then the new label in double quotes, like this:

```
plot(x,y,xlab="Explanatory variable")
```

You might want to alter the label on the y axis as well. The directive you need for that is **ylab**. Use the Up Arrow key again, and insert the y label, like this:

```
plot(x,y,ylab="Response variable",xlab="Explanatory variable")
```

It is easy to change the plotting symbols. At the moment, you are using the default plotting character (**pch**=1) which is an open circle. If you want + (plus signs), use plotting character 3 . Use Up Arrow, then insert a comma after y, then type pch=3

```
plot(x,y,pch=3,ylab="Response variable",xlab="Explanatory variable")
```

Open triangles are plotting character pch=2, and so on.

```
plot(x,y,pch=2,ylab="Response variable",xlab="Explanatory variable")
```

Adding lines to a scatterplot

There are two kinds of lines that you might want to add to a scatterplot: (1) lines that the computer estimates for you (e.g. regression lines); or (2) lines that you specify yourself (e.g. lines indicating some theoretical values for y and x).

Regression lines

The simplest line to fit to some data is the linear regression of y on x.

$$y = a + bx$$

This has 2 variables (y and x) and 2 parameters (a and b, the intercept and the slope respectively). The model is interpreted as saying that y is a function of x, so that changing x might be expected to cause changes in y. The opposite is not true, and changing y by some other means would not be expected to bring about a change in x.

The basic statistical function to derive the two parameters of the linear regression (the intercept, *a*, and the slope, *b*) is the linear model **lm**. This uses the standard model form as its argument

y~x

This is read as "y tilde x" and means "y is to be estimated as a linear function of x". We learn all about this in Practical 4. To draw the regression line through the data, we employ the straight line drawing directive **abline** (intercept and slope, geddit?). This has two arguments: the intercept and the slope, separated by a comma. We can combine the regression analysis and the line drawing into a single directive like this:

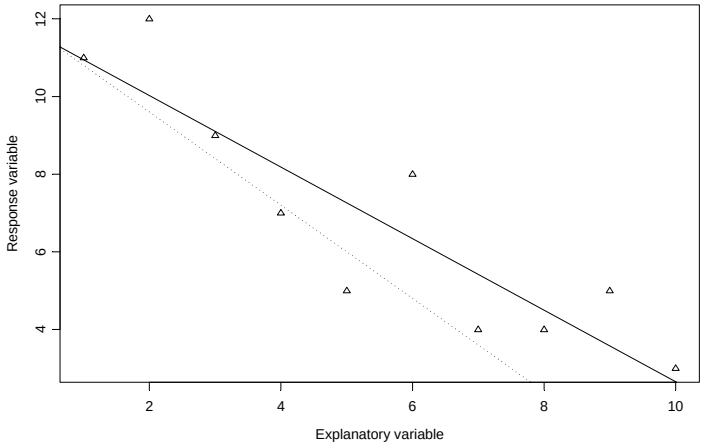
```
abline(lm(y~x))
```

One of the nice features of **abline** is that it automatically draws the line exactly within the limits set by the frame of our graph.

User-specified lines

Sometimes, you want to draw your own line on a scatter plot. A common case is where you want to draw a line of slope = 1 that goes through the origin. Or you might what to draw a horizontal line or a vertical line for some purpose. Suppose that in this case, we want to draw a line that has y = 12 for x = 0 and y = 0 when x = 10. To do this, we use concatenate to make 2 lists: a list of x points c(0,10), and a list of y points c(12,0). User-specified lines are then drawn using the **lines** directive. The first 2 arguments contain the x points of the lines and the y points of the lines, and we shall use a third argument to change the line type to number 2 (lty=2) in order to contrast with the solid regression line that we have just added

```
lines(c(0,10),c(12,0),lty=2)
```



Adding more points to a graph

The important point to appreciate is that material is overlaid on your graph until another **plot** directive is issued. You have seen lines added to the scatterplot using **abline** and **lines**. You can add new points to the scatterplot using **points**.

Sometimes we want several data sets on the same graph, and we would want to use different plotting symbols for each data set in order to distinguish between them. Suppose that we want to add the following points to our current graph. We have 5 new values of the explanatory variable: let's call them *v*, and give them the values 2, 4, 6, 8 and 10:

```
v<-c(2,4,6,8,10)
```

And 5 new values of the response variable, w:

```
w<-c(8,5,6,6,2)
```

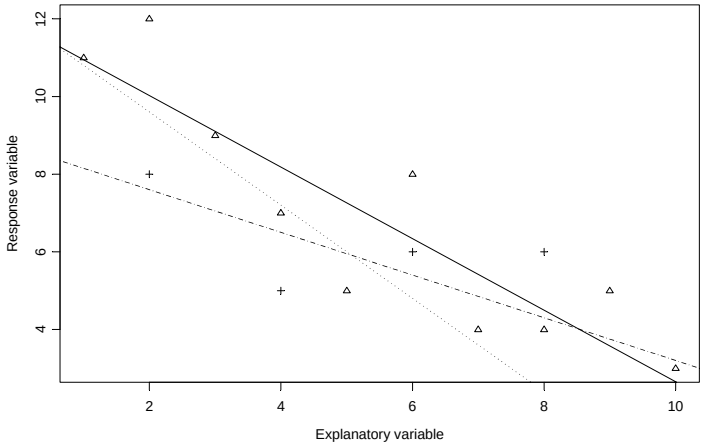
To add these points to the graph, we simply type **points** (not **plot** as this would draw fresh axes and we would lose what we already had done):

```
points(v,w,pch=3)
```

The points are added to the scatterplot using plotting character pch=3 (plus sign +). In SPlus (but **not in R**), a warning message is printed, because one of our points (10,2) is outside the plotting region. It is important to realise that the plotting region is defined when the plot directive is executed at the start of the process, and is not subsequently re-scaled as **lines** or **points** are added. Later on, we shall see how to deal with this in the context of writing general plotting routines that will always accommodate the largest and smallest values of x and y.

If we want to fit a separate regression line for these new data, we just type:

```
abline(lm(w~v),lty=3)
```



This is about as complicated as you would want to make any figure. Adding more information would begin to detract from the message.

Plotting with Categorical Explanatory Variables: Box Plots

Categorical variables are **factors** with 2 or more **levels**. For most kinds of animals, sex is a factor with 2 levels; male and female, and we could make a variable called sex (for example only) like this:

```
sex<-c("male","female")
```

The categorical variable is the factor called sex and the two levels are “male” and “female” respectively. In principle, factor levels can be names or numbers. We shall always use names, because they are so much easier to interpret, but some older software (like GLIM) can only handle numbers as factor levels.

The next example uses the factor called month to investigate weather patterns at Silwood Park. We begin by reading the data from a file like this:

```
weather<-read.table("c:\\temp\\SilwoodWeather.txt",header=T)
```

We set up a data frame called weather, which gets the contents of the file called SilwoodWeather.txt. Note the use of double slash \\ in the file path, and the option “header=T” . This means it is true (“T”) that the first row of the data file contains the names of the variables. We can see what these names are, using **names**

```
names(weather)
```

```
[1] "upper" "lower" "rain"  "month" "yr"
```

The variables in the data frame are upper and lower daily temperatures, rainfall that day, and the names of the month and year to which the data belong. To use these data we need to attach the data frame like this:

```
attach(weather)
```

There is one more bit of housekeeping we need to do before we can plot the data. We need to declare month to be a factor. At the moment, SPlus just thinks it is a number (1 – 12).

```
month<-factor(month)
```

You can always check on the status of a variable by asking “is it a factor ?” using the **is.factor** directive like this

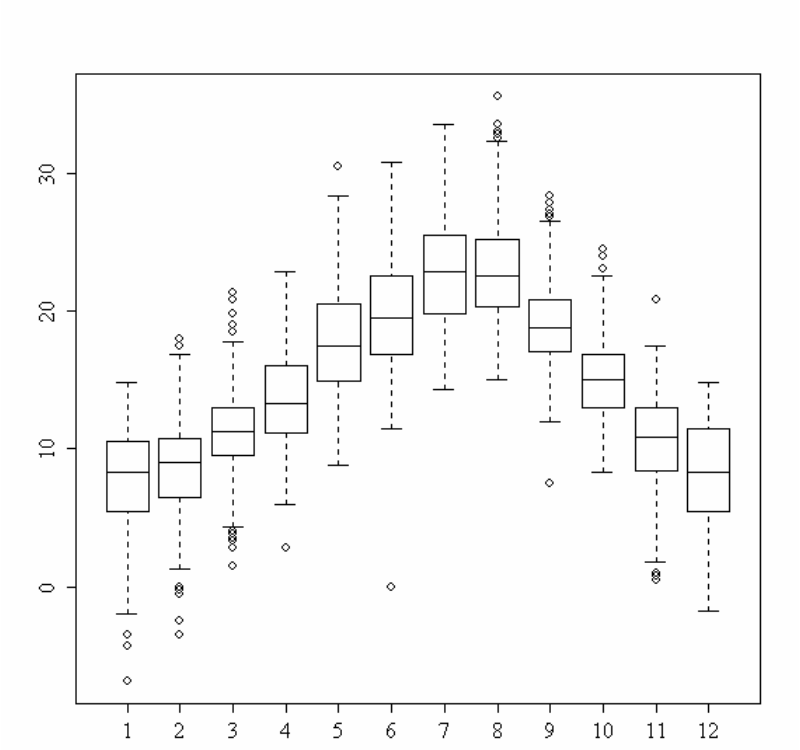
```
is.factor(month)
```

```
[1] TRUE
```

Yes, it is a factor. Now we can **plot** using a categorical explanatory variable (month)

```
plot(month,upper)
```

The syntax is exactly the same, but because the first variable is a factor we get a box plot rather than a scattergraph.



The boxplot summarises a great deal of information very clearly. The horizontal line shows the **median** response for each month. The bottom and top of the box show the 25 and 75 **percentiles** respectively (i.e. the location of the middle 50% of the data). The horizontal line joined to the box by the dashed line (sometimes called the

“whisker”) shows either the maximum or 1.5 times the **interquartile range** of the data (whichever is the smaller). Points beyond the whiskers (outliers) are drawn individually. Boxplots not only show the location and spread of data but also indicate skewness (asymmetry in the sizes of the upper and lower parts of the box). For example, in February the range of lower temperatures was much greater than the range of higher temperatures.

Summary

It is worth re-stating the really important things about plotting:

Plot: plot(xaxis,yaxis) gives a scatterplot if x is continuous, boxplot if x is a factor

Type of plot: options include lines type=”l” or null (axes only) type=”n”

Lines: lines(x,y) plots a smooth function of y against x using the x and y values provided

Line types: useful with multiple line plots, **lty**=2 (an option in plot or lines)

Points: points(x,y) adds another set of data points to a plot

Plotting characters for different data sets: **pch**=2 or **pch**=”*” (an option in points or plot)

Setting limits to x and y axis scales **xlim**=c(0,25), **ylim**=c(0,1) (an option in plot)

Colour in R (SPlus is different)

Plotting in black and white is fine for most purposes (and most printers), but colour is available, and you may want to use it (e.g. for fancy PowerPoint presentations). Try typing this:

```
pie(rep(1, 30), col = rainbow(30), radius = 0.9)
```

It produces a 30-sector colour wheel. The colour function **rainbow** is divided into 30 shades (you can change this) and a pie chart is produced with 30 equally sized sectors (each is width 1, repeated 30 times). The radius directive affects the size of the pie on the screen. See if you can produce a 10-sector colour wheel of radius 0.5.

```
pie(rep(1, 10), col = rainbow(10), radius = 0.5)
```

Colour with graphs

You might want to highlight the points on a graph using colour, draw different lines in different colours, and/or change the colour of the background. Suppose we want to draw lines of these two graphs on the same axes over the range $0 < x < 10$, but using different colours:

$$y_1 = 2 + 3x - 0.25x^2$$

$$y_2 = 3 + 3.3x - 0.3x^2$$

First we calculate values for x

```
x<-seq(0,10,0.1)
```

then the values for the two vectors y1 and y2:

```
y1 <- 2 + 3 * x - 0.25 * x ^ 2
```

```
y2 <- 3 + 3.3 * x - 0.3 * x ^ 2
```

You don't need brackets because of the "hierarchy of calculation": powers first, then times and divide, and finally plus or minus. Now we choose the paper colour to be ghostwhite (**bg** means 'background')

```
par(bg="ghostwhite")
```

The trick with multiple graphs is to draw a set of blank axes to begin with (using type = "n"), making sure that the scale is big enough to accommodate the largest values we want to plot. Since y_2 has larger values, we plot it first:

```
plot(x,y2,type="n",ylab="")
```

Now we draw the two lines, y_2 in red and y_1 in blue

```
lines(x,y2,col="red")
lines(x,y1,col="blue")
```

Coloured scatterplots

Read the following data from a file. They show the daily maximum and minimum temperatures at Silwood in January.

```
jantemps<-read.table("c:\\temp\\jantemp.txt",header=T)
attach(jantemps)
names(jantemps)
```

```
[1] "tmax" "tmin" "day"
```

To scale the axis properly we need to know the maximum and minimum values of y

```
max(tmax)
[1] 10.8


min(tmin)
[1] -11.5
```

Start by plotting the blank axes. We know that the scale on the y-axis needs to go from –12 to +12 degrees, and we want the label on the y-axis to be Temperature. We don't want to plot the data yet, so we use type = "n"

```
plot(day,tmax,ylim=c(-12,12),type="n",ylab="Temperature")
```

Now we can add the **points** to the graph, starting with the minima. We shall plot these as solid symbols (pch = 16, this stands for 'plotting character') in blue:

```
points(day,tmin,col="blue",pch=16)
```

Now for the maxima, in red:

```
points(day,tmax,col="red",pch=16)
```

Finally, we want to join together the maximum and minimum temperatures for the same day with a straight green line. For day 1 this involves plotting from the point (1, tmin[1]) to the point (1,tmax[1]). To automate this procedure for all 31 days in January we put the **lines** directive into a loop, like this

```
for (i in 1:31) lines(c(i ,i ), c( tmin[i], tmax[i] ), col="green")
```

The strong serial correlation in the weather data is clear from this plot. This is what gives rise to the weatherman's dictum when asked what tomorrow will be like: "tomorrow's weather will be like today's".

Colour with histograms

Let's produce a histogram based on 1000 random numbers from a normal distribution with mean 0 and standard deviation 1.

```
x <- rnorm(1000)
```

We shall draw the histogram on cornsilk coloured paper:

```
par(bg = "cornsilk")
```

with the bars of the histogram in a subtle shade of lavender, like this

```
hist(x, col = "lavender",  main = "")
```

The purpose of main = "" is to suppress the graph title. See what happens if you leave this out.

Colour with pie charts

Released insects landed on 6 plant species in the following proportions

```
fate <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
```

We can provide labels for the segments of the pie by naming the plant species like this:

```
names(fate)<-c("Ragwort", "Thistle", "Willowherb", "Rush", "Orchid",
        "Knapweed")
```

The pie chart can now be drawn, specifying different colours for each of the segments in sequence, like this

```
pie(fate, col = c("purple", "violetred1",  "green3", "cornsilk", "cyan",
        "white"))
```

Here **col** is specified as a vector with as many levels as there are sectors in the pie (6 in this case).

Multivariate Plots

Data sets often consist of many continuous variables, and it is important to find out if, and how, the variables are inter-related. Read the data file called pollute.txt

```
pollution<-read.table("c:\\temp\\pollute.txt",header=T)
attach(pollution)
```

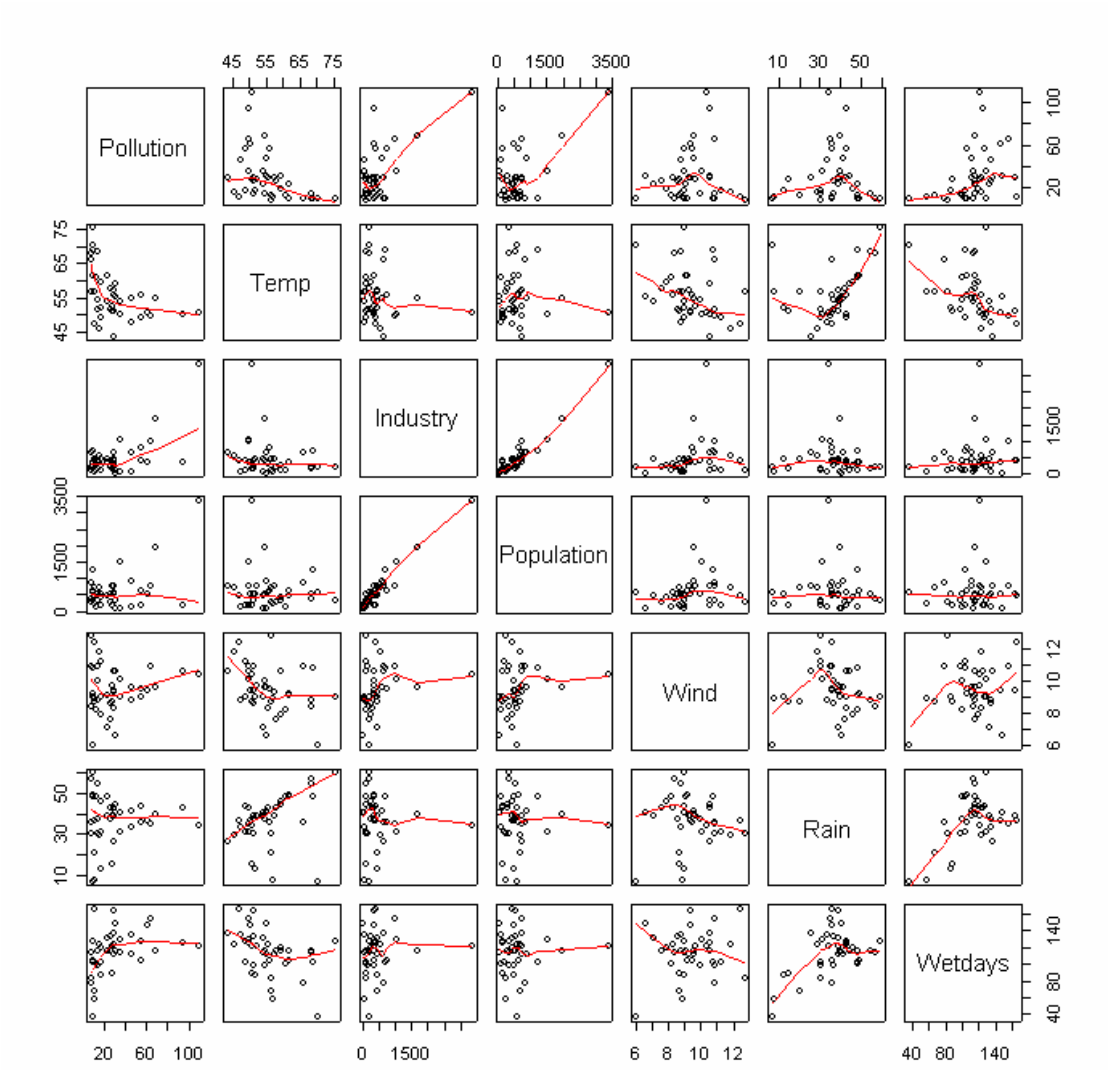
To see which variables are included in the data frame called pollution we use **names**

```
names(pollution)
```

```
[1] "Pollution"  "Temp"       "Industry"   "Population"
"Wind"       "Rain"  "Wetdays"
```

The directive **pairs** produces a matrix of plots of y against x and x against y for all of the variables in the data frame (all 7 in this case).

```
pairs(pollution,panel=panel.smooth)
```



Interpretation of the matrix takes some getting used to. The rows of the matrix have the response variable (the y axis) as labelled by the variable name that appears in that row. For example, all the graphs on the middle row (the 4th row) have Population on the y axis. Likewise, the columns of the matrix all have the same explanatory variable (the x axis) as labelled by the variable name that appears in that column. For example, all the graphs in the right-most column have Wetdays on their x axis.

This kind of multiple scatterplot is good at showing some patterns but poor at showing others. Where the data are well spaced, then relationships can show up clearly. But when the data are clumped at one end of the axis it is much more difficult. Likewise, the plots are poor when variables are categorical (integer). Not unexpectedly, there is a very close correlation between Industry and Population, but the relationship between Pollution and Wind is far from clear. We shall carry out the statistical analysis of these data later on (see Multiple Regression).

Tree-based models

The most difficult problems in interpreting multivariate data are that

1. the explanatory variables may be correlated with one another
2. there may be interactions between explanatory variables
3. relationships between the response and explanatory variables may be non-linear

An excellent way of investigating interactions between multiple explanatory variables involves the use of tree-based models. We need to get the code from the library:

```
library(tree)
```

Tree models are constructed on a simple, stepwise principle. The computer works out which of the variables explains most of the variance in the response variable, then determines the threshold value of the explanatory variable that best partitions the variance in the response. Then the process is repeated for the y values associated with *large values* of the first explanatory variable, asking which explanatory variable explains most of this variation. The process is then repeated for the y values associated with *low values* of the first explanatory variable. The process is repeated until there is no residual explanatory power. The procedure is very simple, involving the directive **tree**. For the pollution data set, we type:

```
regtree<-tree(Pollution ~ . , data=pollution)
```

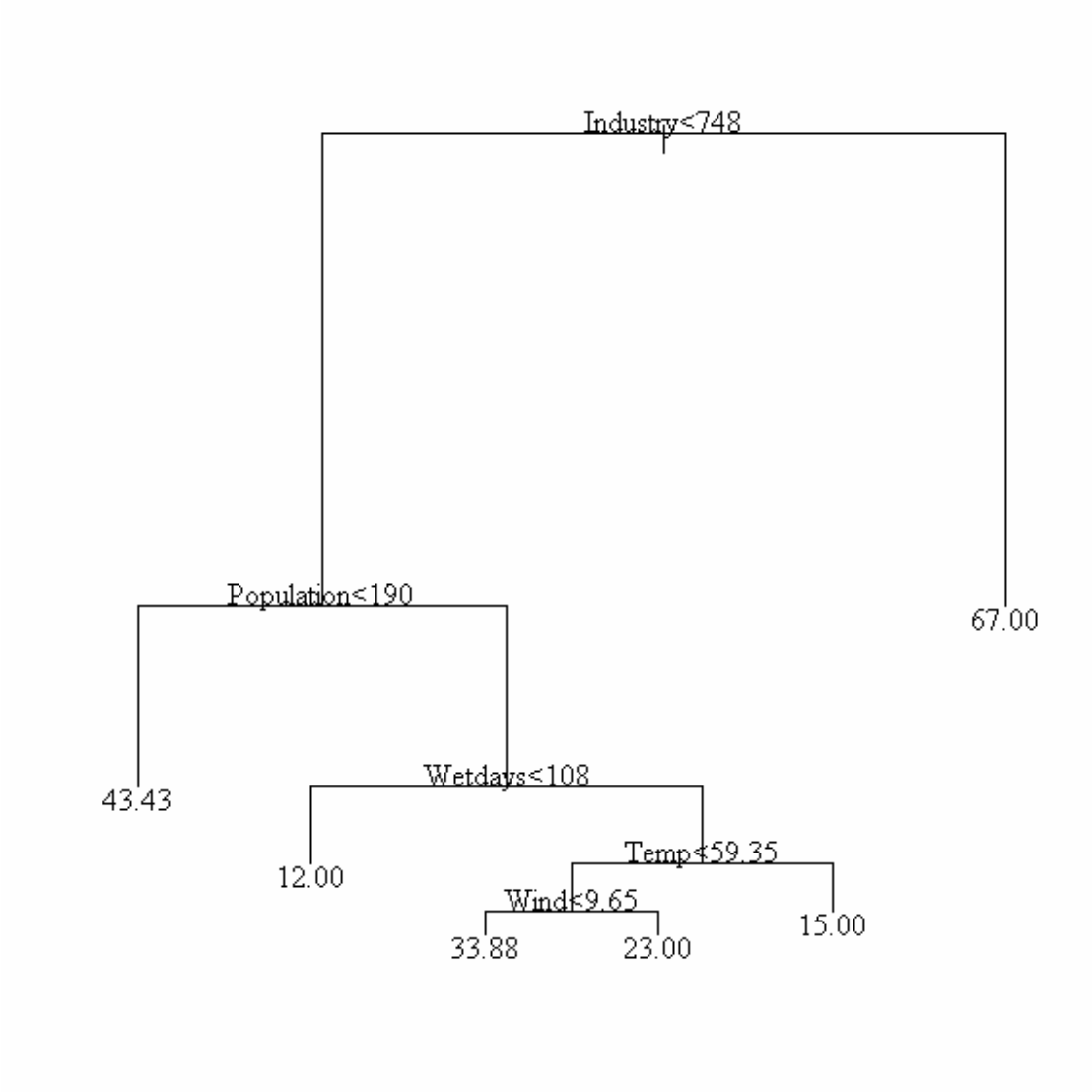
where the model formula reads "Pollution (our response variable) is a function of *all* of the explanatory variables in the data frame called pollution". The power comes from the 'dot option'. The tilde ~ means "is a function of". The dot . means "everything". So make sure you get the syntax right:

the punctuation is "**tilde dot comma**"

Now the object called **regtree** contains the regression tree and we want to see it, and to label it:

```
plot(regtree)
text(regtree)
```

This is interpreted as follows:



The most important explanatory variable is Industry, and the threshold value separating low and high values of Industry is 748. The right hand branch of the tree indicates the mean value of air pollution for high levels of industry (67.00). The fact that this limb is unbranched means that no other variables explain a significant amount of the variation in pollution levels for high values of Industry. The left-hand limb does not show the mean values of pollution for low values of industry, because there are other significant explanatory variables. Mean values of pollution are only shown at the extreme ends of branches. For low values of Industry, the tree shows us that Population has a significant impact on air pollution. At low values of Population (<190) the mean level of air pollution was 43.43. For high values of Population, the number of Wetdays is significant. Low numbers of wet days (< 108) have mean pollution levels of 12.00 while Temperature has a significant impact on pollution for places where the number of wet days is large. At high temperatures (> 59.35 ‘F) the mean pollution level was 15.00 while at lower temperatures the run of Wind is important. For still air (Wind < 9.65) pollution was higher (33.88) than for higher wind speeds (23.00). The statistical analyses of regression trees, including the steps involved in model simplification, are dealt with later (see Practical 10).

The virtues of tree-based models are numerous:

1) they are easy to appreciate and to describe to other people
2) the most important variables stand out
3) interactions are clearly displayed
4) non-linear effects are captured effectively (e.g. as step functions)
5) the complexity (or lack of it) in the behaviour of the explanatory variables is plain to see

Conditioning plots

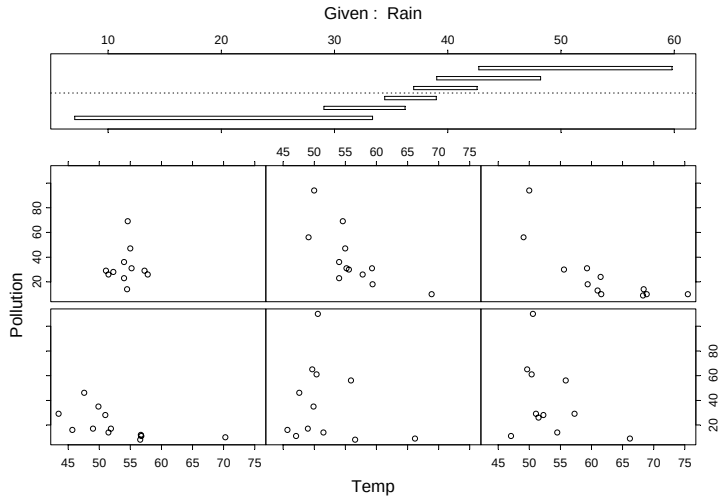
A real difficulty with multivariate data is that the relationship between two variables may be obscured by the effects of other processes.

```
attach(pollution)
```

When you carry out a 2-dimensional plot of y against x, then all of the effects of the other explanatory variables are squashed flat onto the plane of the paper. The function **coplot** produces a conditioning plot automatically in cases like the pollution example here, where the explanatory variables are continuous. It is extremely easy to use. Suppose we want to look at Pollution as a function of temperature but we want to condition this on different levels of rainfall. This is all we do:

```
coplot(Pollution~Temp | Rain)
```

The plot is described by a model formula: Pollution on the y axis, Temp on the x axis, with 6 separate plots conditioned on the value of Rain.



The panels are ordered from lower left, row-wise, to upper right, from lowest rainfall to highest. The upper panel shows the range of values for Rain chosen by **coplot** to form the 6 panels. Note that the range of rainfall values varies from panel to panel. The largest range is in the bottom left plot (range 8 to 33) and the smallest for the

bottom right plot (35 to 39). Note that the ranges of Rain overlap one another at both ends; this is called a **shingle**.

Graphical Parameters (par)

Without doubt, the graphical parameter you will change most often just happens to be the least intuitive to use. This is the number of graphs per screen, called somewhat unhelpfully, **mfrow**. The idea is simple, it is just the syntax that is hard to remember. You need to specify the number of rows of plots you want, and number of plots per row, in a vector of 2 numbers. The first number is the number of rows and the second number is the number of graphs per row. The vector is made using **c** in the normal way (**c** means concatenate). The default single plot screen is

```
par(mfrow=c(1,1))
```

Two plots side by side is

```
par(mfrow=c(1,2))
```

Four plots in a 2 x 2 square is

```
par(mfrow=c(2,2))
```

To move from one plot to the next, you need to execute a new **plot** directive. Control stays within the same plot frame while you execute directives like **points**, **lines** or **text**. Remember to return to the default single plot when you have finished your multiple plot by executing **par(mfrow=c(1,1))**.

If you have more than 2 graphs per row or per column, the character expansion **cex** is set to 0.5 and you get half-size characters and labels.

Logarithmic axes

You will often want to have one or both of your axes on a logarithmic scale.

log="xy", log="x" or log="y" controls logarithmic axes, producing log-log, log-x or log-y axes respectively. Here are the same data plotted 4 ways:

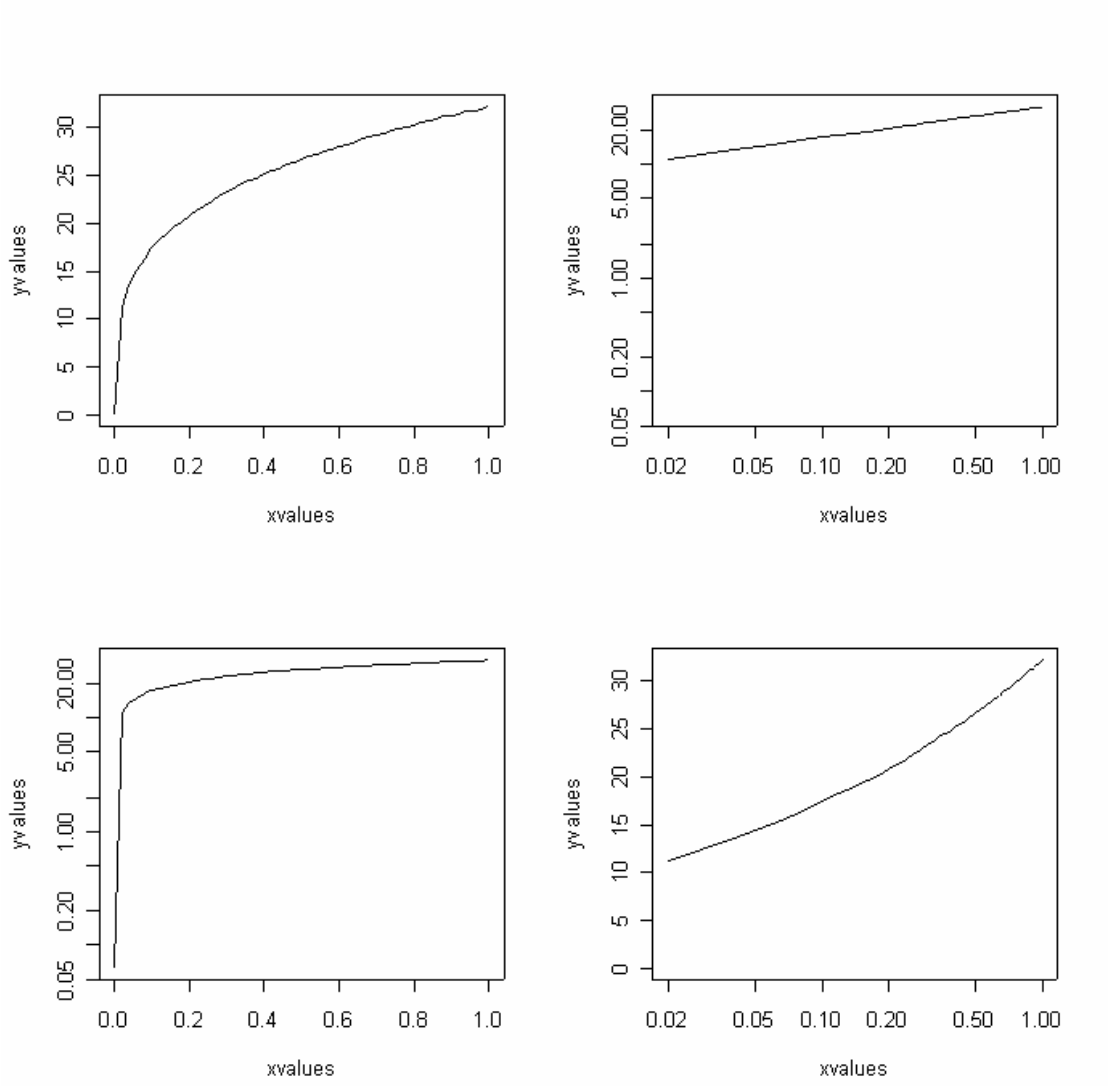
```
plotdata<-read.table("c:\\temp\\plotdata.txt", header =T)

attach(plotdata)
names(plotdata)

[1] "xvalues" "yvalues"
```

Make a frame for 4 adjacent plots is a 2 x 2 array:

```
par(mfrow=c(2,2))
```

From top left to bottom right plot the data with untransformed axes, as a log-log plot, as log(y) against x and as y against log(x)

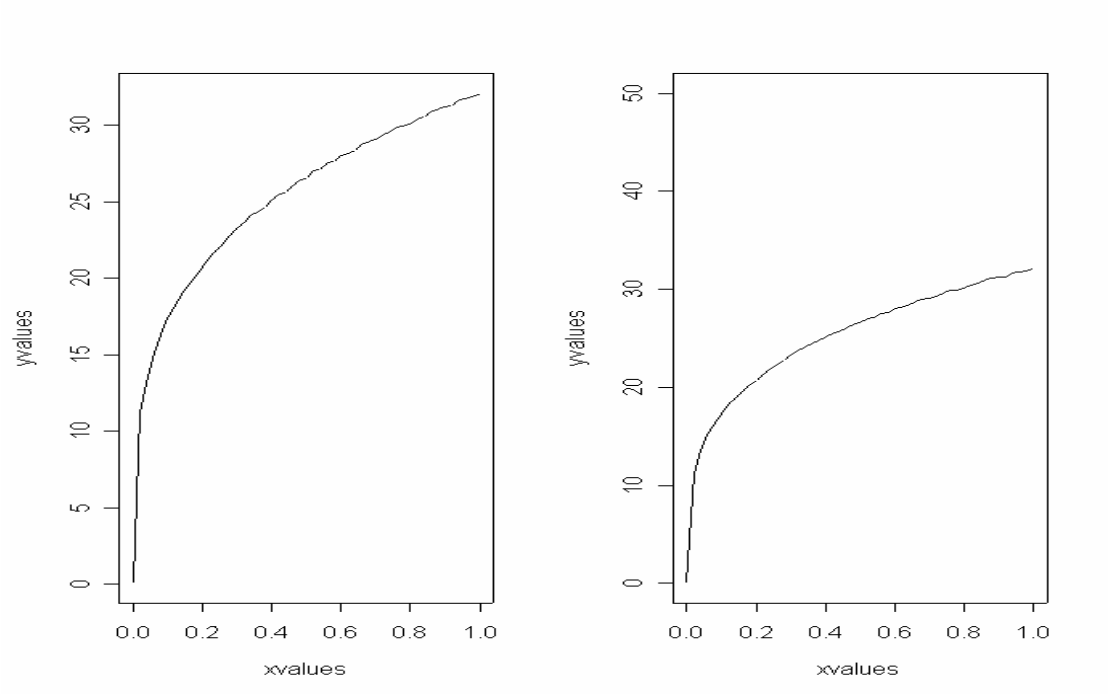
```
plot(xvalues,yvalues,type=”l”)
plot(xvalues,yvalues,log="xy",type=”l”)
plot(xvalues,yvalues,log="y", type=”l”)
plot(xvalues,yvalues,log="x", type=”l”)
```



Scaling the axes

To change the upper and lower values on the axes use **xlim** and **ylim** like this

```
par(mfrow=c(1,2))
plot(xvalues,yvalues,type=”l”)
plot(xvalues,yvalues,ylim=c(0,50),type=”l”)
```



You concatenate **c** a list of length 2, containing the approximate minimum and maximum values to be put on the axis. These values are automatically rounded to make them "pretty" for axis labelling. You will want to control the scaling of the axis

- when you want 2 comparable graphs side by side
- when you want to overlay several lines or sets of points on the same axes
- remember that the initial **plot** directive sets the axes scales
- this can be a problem if subsequent **lines** or **points** are off-scale

Text in graphs

It is very easy to add text to graphics. Just work out the x and y coordinates where you want the centre of the text to appear on an existing plot, using locator(1) to move the cursor to the appropriate position then left click the mouse to see the coordinates x and y. Suppose we want to add “(b)” at the point (0.8,45), then we just type:

```
text(0.8,45,”(b)”)
```

If you have factor level names with spaces in them (e.g. multiple words), then the best format for reading files is comma delimited (“.csv” rather than the standard tab delimited, ”.txt” files). You read them into a dataframe in R using read.csv in place of read.table. The next example is a bit more complicated. The task is to produce a map of place names, where the names required for plotting are in one file called map.places.csv, but their coordinates are in another, much longer file called bowens.csv, containing all place names.

```
map.places <-read.csv("c:\\temp\\map.places.csv",header=T)
attach(map.places)
names(map.places)
```

```
[1] "wanted"
```

```
map.data<-read.csv("c:\\temp\\bowens.csv",header=T)
attach(map.data)
names(map.data)

[1] "place" "east"  "north"
```

There is a slight complication to do with the coordinates. The northernmost places are in a different 100 km square so, for instance, a northing of 3 needs to be altered to 103. It is convenient that all of the values that need to be changed have northings < 60 in the dataframe:

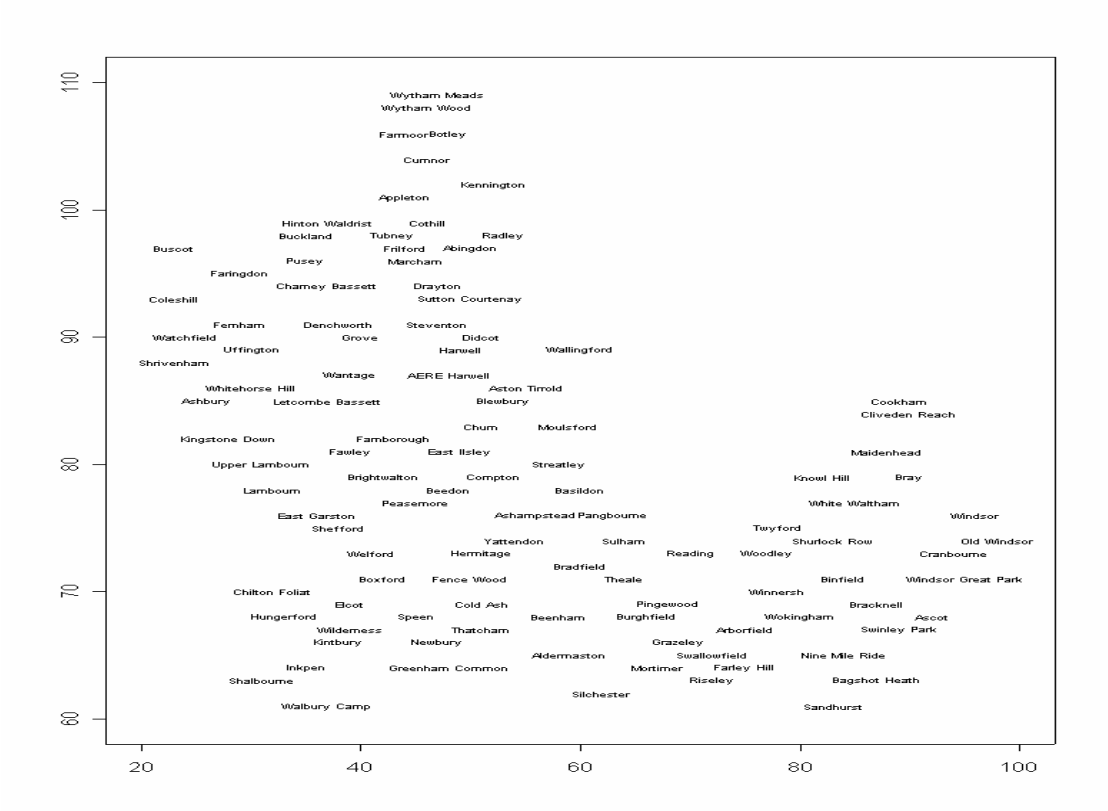
```
nn<-ifelse(north<60,north+100,north)
```

This means change all of the northings for which north < 60 is TRUE to nn<-north+100, and leave unaltered all the others (FALSE) as nn<-north. We begin by plotting a blank space (type=”n”) of the right size (eastings from 20 to 100 and northings from 60 to 110) with blank axis labels “”.

```
plot(c(20,100),c(60,110),type="n",xlab="",ylab="")
```

The trick is to select the appropriate places in the vector called place and use text to plot each name in the correct position (east[i],nn[i]). For each place name in wanted we find the correct subscript for that name within place using the which function

```
for (i in 1:length(wanted)){
ii <- which(place == as.character(wanted[i]))
text(east[ii], nn[ii], as.character(place[ii]), cex = 0.6) }
```



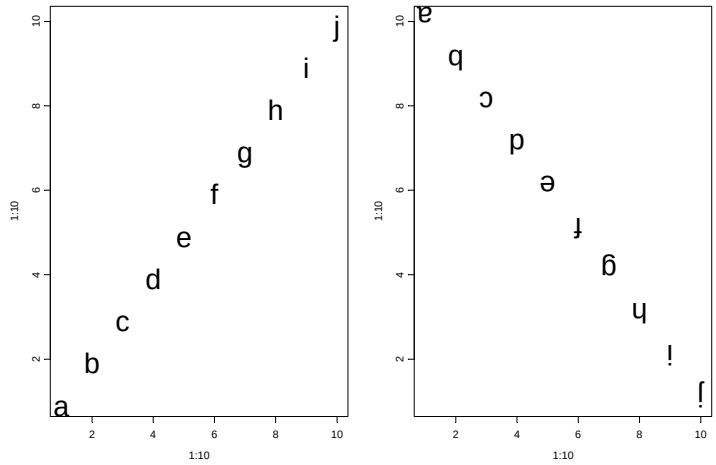
Character alignment

```
labels<-letters[1:10]
labels
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Plot type = “n” is useful for scaling the axes but suppressing the creation of any filler for the space within the axes:

```
plot(1:10,1:10,type="n")
text(1:10,1:10,labels,cex=2)
plot(1:10,1:10,type="n")
text(1:10,10:1,labels,cex=2,srt=180)
```



Note the use of string (character) rotation **srt** (degrees counter clockwise from the horizontal) to turn the letters upside down (**srt** = 180) in the right hand plot. Note also the reversal of the sequence argument, so that the y values of the character locations in the right hand plot go down from 10 to 1 as x goes up from 1 to 10. These letters are twice normal size (**cex** = 2 stands for “character expansion 2-fold”).