Introduction to R for the Geosciences: R Basics & Plotting

Gavin Simpson

30th April — 3rd May 2013

Summary

In this practical class we will introduce you to working with R. You will complete an introductory session with R and then use a data set of Spheroidal Carbonaceous Particle (SCP) surface chemistry and demonstrate some of the Exploratory Data Analysis (EDA) functions in R.

1 Your first R session

R can be used as a glorified calculator; you enter calculations at the prompt, R works out the answer and displays it to you. Try the following sequence of commands. Remember, only type the commands on lines starting with the prompt ">" below. In each case, the output from R is shown.

```
> 3 * 5
```

```
[1] 15
```

```
> 7 + 2
```

```
[1] 9
```

```
> 100 / 5
```

```
[1] 20
```

```
> 99 - 1
```

```
[1] 98
```

After each sum R prints out the result with a [1] at the start of the line. This means the result is a vector of length 1. As you will see later, this string shows you which entry in the vector is displayed at the beginning of each line.

This isn't much use if you can't store results though. To store the result of a calculation, the assignment operator "<-" is used.

```
> a <- 3 * 5
```

Notice that this time R does not print the answer to the console. You have just created an object in R with the name "a". To print out the object to the console you just have to type the name of the object and press Enter.

```
> a
```

```
[1] 15
```

We can now use this to do sums like you would on a calculator using the memory functions.

```
> b <- 7 + 2
> a + b
```

```
[1] 24
```

```
> a * b
```

```
[1] 135
```

```
> c <- a * b
> c
```

```
[1] 135
```

To see a list of the objects in the current workspace use the `ls()` function:

```
> ls()
```

```
[1] "a" "b" "c"
```

2 Exploratory Data Analysis

Use R's EDA functions to examine the SCP data with a view to answering the following questions:

1. Suggest which chemical elements give the best discrimination between coal and oil particles;
2. Suggest which variables are highly correlated and may be dropped from the analysis;
3. Suggest which particles (either coal or oil) have an unusual chemical composition – i. e., are outliers;

SCP's are produced by the high temperature combustion of fossil fuels in oil and coal-fired power stations. Since SCP's and sulphur (as SO_2) are emitted from the same sources and are dispersed in the same way, the record of SCP's in lake sediments may be used to infer the spatial and temporal patterns of sulphur deposition, an acid loading on lake systems. In addition, SCP's produced by the combustion of coal and oil have different chemical signatures, so characterization of particles in a sediment core can be used to partition the pollution loading into distinct sources.

The data set consists of a sample of 100 SCP's from two power stations (50 SCP's from each). One, Pembroke, is an oil-fired station, the other, Drax, is coal-fired. The data form a training set that was used to generate a discriminant function for classifying SCP's derived from lake sediments into fuel type. The samples of fly-ash from power station flues were analysed using Energy Dispersive Spectroscopy (EDS) and the chemical composition was measured.

These data are available in the file `scp.csv`, a standard comma-delimited ASCII text file. You will use R in this practical class, learn how to read the data into R and how to use the software to perform standard EDA and graphics.

3 Reading the data into R

Firstly, start R on the system you are using in the `prac1` directory that contains the files for this practical, instructions on how to do this have been circulated on a separate handout.

The following commands read the data from the `scp.csv` file into a R object called `scp.dat` and perform some rudimentary manipulation of the data:

```
> scp.dat <- read.csv(file = "scp.csv", row.names = 1)
> scp.dat[,18] <- as.factor(scp.dat[,18])
> levels(scp.dat[,18]) <- c("Coal", "Oil")
```

The `read.csv()` function reads the data into an object, `scp.dat`. The argument `row.names` instructs R to use the first column of data as the row names.

By default R assigns names to each observation. As the `scp.csv` file contains a column labelled `"SampleID"`, which contains the sample names, we now convert the first column of the `scp.dat` object to characters (text) and then assign these as the sample names using the `rownames` function. Note that we subset the `scp.dat` object using the following notation:

object[r,c]

Here r and c represent the row and column required, respectively. To select the first column of `scp.dat` we use `scp.dat[,1]`), leaving the row identifier blank.

The last variable in `scp.dat` is `FuelType`. This variable is currently coded as `"1"`, `"0"`. R has a special way of dealing with "factors", but we need to tell R that `FuelType` is a factor (`FuelType` is currently a numeric variable). The function `as.factor()` is used to *coerce* a vector of data from one type into a factor (`scp.dat[,18] <- as.factor(scp.dat[,18])`. The "levels" of `FuelType` are still coded as `1`, `0`. We can replace the current levels with something more useful using `levels(scp.dat[,18]) <- c("Coal", "Oil")`. Note the use of the concatenate function, `c()`, to create a character vector of length 2.

Simply typing the name of an object followed by return prints the contents of that object to the screen (increase the size of your R console window before doing this otherwise the printed contents will extend over many pages).

```
> scp.dat
> names(scp.dat)
> str(scp.dat)
```

The `names()` function prints out the names describing the contents of the object. The output from names depends on the *class* of the object passed as an argument[1] to `names()`. For a data frame like `scp.dat`, `names()` prints out the labels for the columns, the variables. The `str()` function prints out the **str**ucture of the object. The output from `str()` shows that `scp.dat` contains 17 numeric variables and one (`FuelType`) factor variable with the levels `"Coal"` and `"Oil"`.

4 Summary statistics

Simple summary statistics can be generated using `summary()`:

```
> summary(scp.dat)
```

`summary()` is a generic function, used to summarize an object passed as an argument to it. For data frames, numeric matrices and vectors, `summary()` prints out the minimum and maximum values, the mean, the median and the upper and lower quartiles for each variable in the object. For factor variables, the number of observations of each level is displayed.

Another way of generating specific summary information is to use the corresponding function, e.g. `mean()`, `min()`, `range()` or `median()`:

```
> mean(scp.dat[,1])      # mean for variable 1
```

[1]Arguments are the values entered between the brackets of a call to a function, and indicate a variety of options, such as the name of an object to work on or which method to use. A function's arguments are documented in the help pages for that function

```
[1] -12.0025
```

Doing this for all 17 variables and for each of the descriptive statistical functions could quickly become tedious. R contains many functions that allow us to quickly apply functions across each variable in a data frame. For example, to get the mean for each variable in the `scp.dat` data frame you would type:

```
> apply(scp.dat[, -18], 2, mean)
```

```
      Na       Mg       Al       Si        P        S       Cl
-12.0025  -7.4240   9.8417   6.9593   3.5364  56.2859  -1.0291
       K       Ca       Ti        V       Cr       Mn       Fe
  1.5607  -0.4613   0.5897   0.8021  -0.0081  -0.3536   3.9651
      Ni       Cu       Zn
  0.2438   0.3103   0.1268
```

```
> ## or
> colMeans(scp.dat[, -18])
```

```
      Na       Mg       Al       Si        P        S       Cl
-12.0025  -7.4240   9.8417   6.9593   3.5364  56.2859  -1.0291
       K       Ca       Ti        V       Cr       Mn       Fe
  1.5607  -0.4613   0.5897   0.8021  -0.0081  -0.3536   3.9651
      Ni       Cu       Zn
  0.2438   0.3103   0.1268
```

As its name suggests, `apply()` takes as it argument a vector or array and applies a named function on either the rows, or the columns of that vector or array (the 2 in the above call indicate that we want to work on the columns). Apply can be used to run any appropriate function including a users own custom functions. As an example, we now calculate the trimmed mean:

```
> apply(scp.dat[, -18], 2, mean, trim = 0.1)
```

```
        Na         Mg         Al         Si          P
-11.462000  -6.672250   9.327625   6.749750   3.650125
         S         Cl          K         Ca         Ti
 57.069125  -2.711250   1.527250  -0.712250   0.460250
         V         Cr         Mn         Fe         Ni
  0.853000  -0.168125  -0.377375   2.768500   0.244750
        Cu         Zn
  0.129500   0.117875
```

In the above code snippet, we pass an additional argument to `mean()`, `trim = 0.1`, which calculates the trimmed mean, by trimming 10% of the data off each end of the distribution and then calculating the mean of the remaining data points. How this works is quite simple. `apply()` is told to work on columns 1 through 17 of the `scp.dat` object. Each column in turn is passed to our function and `mean()` calculates the trimmed mean.

Try using `apply()` to calculate the range (`range()`), the variance (`var()`) and the standard deviation (`sd()`) for variables 1 through 17 in `scp.dat`.

Having now looked at the summary statistics for the entire data set, we can now start to look for differences in SCP surface chemistry between the two fuel types. This time we need to subset the data and calculate the summary statistics for each level in `FuelType`. The are many ways of subsetting a data frame, but the easiest way is to use the `aggregate()` function.

Now try the following code snippet:

```
> aggregate(. ~ FuelType, data = scp.dat, FUN = mean)
```

```
  FuelType      Na      Mg      Al      Si      P       S
1     Coal -15.482 -6.6932 20.0824 13.2710 4.0560 40.4590
2      Oil  -8.523 -8.1548 -0.3990  0.6476 3.0168 72.1128
       Cl      K      Ca      Ti       V      Cr      Mn     Fe
1  4.2242 2.6146 -1.2518 1.0392 0.3276  0.1092 -0.2356 0.3748
2 -6.2824 0.5068  0.3292 0.1402 1.2766 -0.1254 -0.4716 7.5554
       Ni     Cu     Zn
1 -0.1174 0.2378 0.2198
2  0.6050 0.3828 0.0338
```

```
> aggregate(. ~ FuelType, data = scp.dat, FUN = mean, trim = 0.1)
```

```
  FuelType        Na     Mg       Al      Si      P        S
1     Coal -14.42000 -5.763 20.07950 13.0840 4.3945 40.88325
2      Oil  -7.99475 -7.488 -0.66575  0.2625 2.8685 73.05375
        Cl       K       Ca       Ti       V       Cr       Mn
1  1.71025 2.77125 -0.74275 0.84725 0.39775 -0.05200 -0.39625
2 -6.36150 0.41350 -0.77250 0.07675 1.23775 -0.28375 -0.32775
        Fe       Ni      Cu     Zn
1 -0.10175 -0.18300 0.13575 0.1970
2  6.82075  0.59875 0.11650 0.0395
```

```
> #aggregate(scp.dat[,-18], list(FuelType), mean, trim = 0.1)
> #apply(scp.dat[,-18], 2, function(v) tapply(v, FuelType, mean))
> #apply(scp.dat[,-18], 2, function(v) tapply(v, FuelType,
> #  mean, trim = 0.1))
```

As before, using the summary functions you have already tried, look for differences in the surface chemistry of SCP's derived from the two different fuel types.

5 Univariate graphical exploratory data analysis

This section of the practical will demonstrate the graphical abilities of R for exploratory data analysis for univariate data.

5.1 Histograms

Histograms can be used to get a rough impression of the distribution. Histograms are drawn by breaking the distribution into a number of sections or *bins*, which are generally of equal width. You will probably have seen frequency histograms used where the number of observations per bin is calculated and graphed. An alternative approach is to plot the relative frequency of observation per bin width (probability density estimates – more on density estimates later). Graphs showing histograms of the same data using the two methods look identical, except for the scaling on the y axis.

In R, the histogram function is called `hist()`. For this practical, however, you will use the `truehist()` function from the `MASS` package, because it offers a wider range of options for drawing histograms. `MASS` should be distributed as a recommended package in your R distribution.

```
> require(MASS)
> op <- par(mfrow = c(3,1))
> with(scp.dat, hist(Na, col = "grey", main = "R default", ylab = "Frequency",
+                    freq = FALSE))
> with(scp.dat, truehist(Na, nbins = "FD", col = "grey",
+                        main = "Freedman-Diaconis rule", ylab = "Frequency"))
> with(scp.dat, truehist(Na, nbins = "scott", col = "grey",
+                        main = "Scott's rule", ylab = "Frequency"))
> par(op)           #reset the plotting parameters
> rm(op)            #tidy up be removing the oldpar object
```

Note the use of the `par()` function above to split the display into three rows and 1 column using the `mfrow` argument. By assigning the current contents of `par()` to `oldpar` and then setting `mfrow = c(3,1)` we can reset the original `par()` settings using `par(oldpar)` when we are finished. The figure produced should look similar to Figure 1.

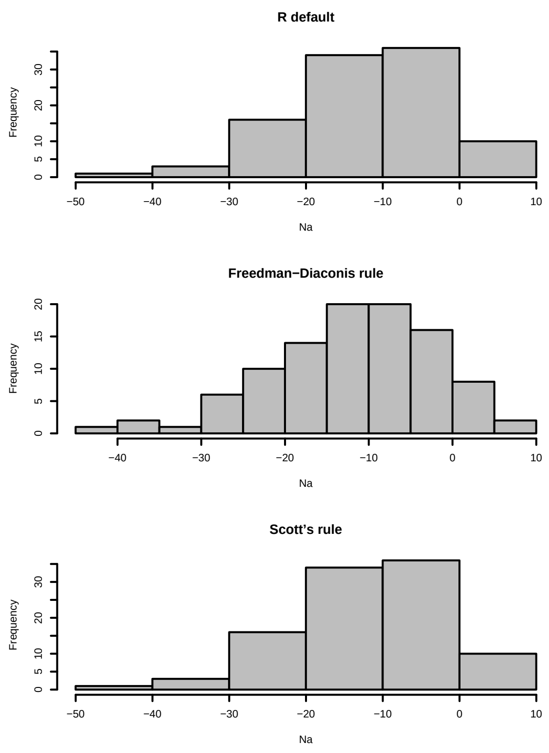
For sodium (Na), Scott's rule and the default setting for calculating the number of bins produce the same result. The Freedman-Diaconis rule, however, produces a histogram with a greater number of bins. Note also that we have restricted this example to frequency histograms. In the next section probability density histograms will be required.

The recommended number of bins using the Freedman-Diaconis rule is generated using:

$$\left\lceil \frac{n^{1/3}(\max - \min)}{2(Q_3 - Q_1} \right\rceil,$$

where n is the number of observations, $\max - \min$ is the range of the data, $Q_3 - Q_1$ is the interquartile range. The brackets represent the *ceiling*, which indicates that you round up to the next integer (so you don't end up with 5.7 bins for example!)

Figure 1: Histograms illustrating the use of different methods for calculating the number of bins.



5.2 Density estimation

A useful alternative to histograms is nonparametric density estimation, which results in a smoothing of the histogram. The *kernel-density estimate* at the value x of a variable X is given by:

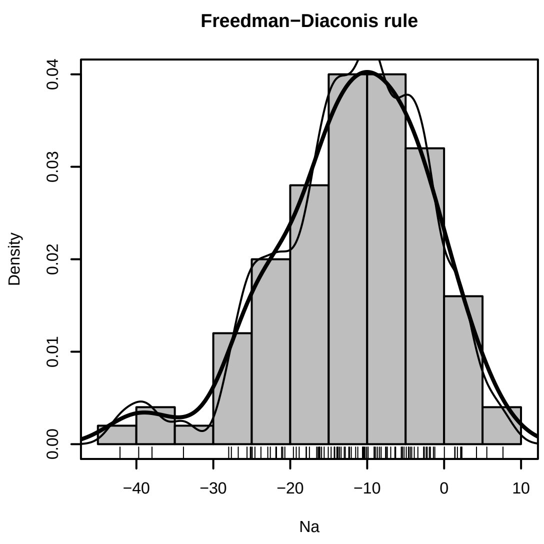
$$\hat{f}(x) = \frac{1}{b} \sum_{j=1}^{n} K\left(\frac{x - x_j}{b}\right),$$

where x_j are the n observations of X, K is a kernel function (such as the normal density) and b is a bandwidth parameter controlling the degree of smoothing. Small bandwidths produce rough density estimates whilst large bandwidths produce smoother estimates.

It is easier to illustrate the features of density estimation than to explain them mathematically. The code snippet below draws a histogram and then overlays two density estimates using different bandwidths. The code also illustrates the use of the `lines()` functions to build up successive layers of graphics on a single plot.

```
> with(scp.dat, truehist(Na, nbins = "FD", col = "grey", prob = TRUE,
+                        ylab = "Density",
+                        main = "Freedman-Diaconis rule"))
> with(scp.dat, lines(density(Na), lwd = 2))  # default bandwidth
```

Figure 2: Comparison between histogram and density estimation techniques.



```
> with(scp.dat, lines(density(Na, adjust = 0.5),
+                        lwd = 1))   # half the default bandwidth
> with(scp.dat, rug(Na))             # adds a rug plot
> box()                              # draws a box round the plot
```

Note the use of the argument `prob = TRUE` in the `truehist()` call above, which draws a histogram so that it is scaled similarly to the density estimates. You should see something similar to Figure 2.

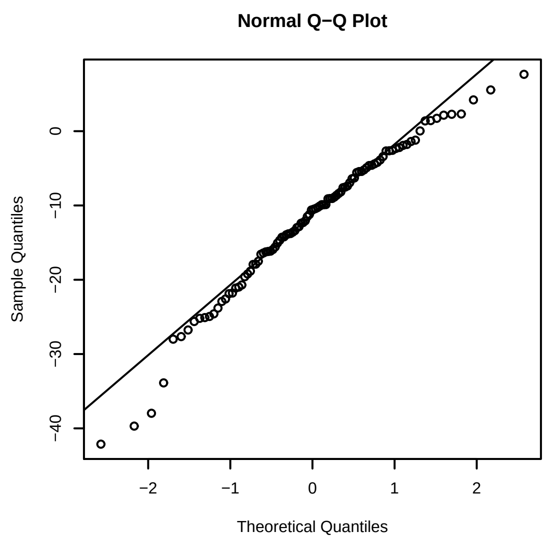
Examine the probability density estimates for some of the other SCP surface chemistry variables.

5.3 Quantile quantile plots

Quantile quantile or Q-Q plots are a useful tool for determining whether your data are normally distributed or not. Q-Q plots are produced using the function `qqnorm()` and its counterpart `qqline()`. Q-Q plots illustrate the relationship between the distribution of a variable and a reference or theoretical distribution. We will confine ourselves to using the normal distribution as our reference distribution, though other R functions can be used to compare data that conform to other distributions. A Q-Q plot graphs the relationship between our ordered data and the corresponding quantiles of the reference distribution. To illustrate this, type in the following code snippet:

```
> with(scp.dat, qqnorm(Na))  # draws the Q-Q plot
> with(scp.dat, qqline(Na))  # plots a line through the 1st and 3rd quartiles
```

Figure 3: A quantile quantile plot of Sodium from the SCP surface chemistry data set.



If the data are normally distributed they should plot on a straight line passing through the 1st and 3rd quartiles. The line added using `qqline` aids the interpretation of this plot. Where there is a break in slope of the plotted points, the data deviate from the reference distribution. From the example above, we can see that the data are reasonably normally distributed but have a slightly longer left tail (Figure 3).

Plot Q-Q plots for some of the chemical variables. Suggest which of the variables are normally distributed and which variables are right- or left-skewed.

5.4 Boxplots

Boxplots are another useful tool for displaying the properties of numerical data, and as we will see in a minute, are useful for comparing the distribution of a variable across two or more groups. Boxplots are also known as box-whisker plots on account of their appearance. Boxplots are produced using the `boxplot()` function. Here, many of the enhanced features of boxplots are illustrated:

```
> with(scp.dat, boxplot(Na, notch = TRUE, col = "grey", ylab = "Na",
+                        main = "Boxplot of Sodium", boxwex = 0.5))
```

Figure 4 shows the resulting boxplot. The box is drawn between the 1st and 3rd quartiles, with the median being represented by the horizontal line running through the box. The notches either side of the median are used to compare boxplots between groups. If the notches of any two boxplots do not overlap then the median values of a variable for those two groups are significantly different at the 95% level.

When plotting only a few boxplots per diagram, the `boxwex` argument can be used to control the width of the box, which often results in a better plot. Here we scale the box width by 50%.

Figure 4: A boxplot of Sodium from the SCP surface chemistry data set.

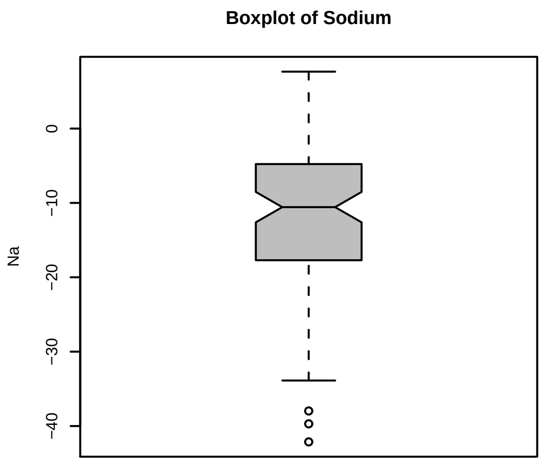
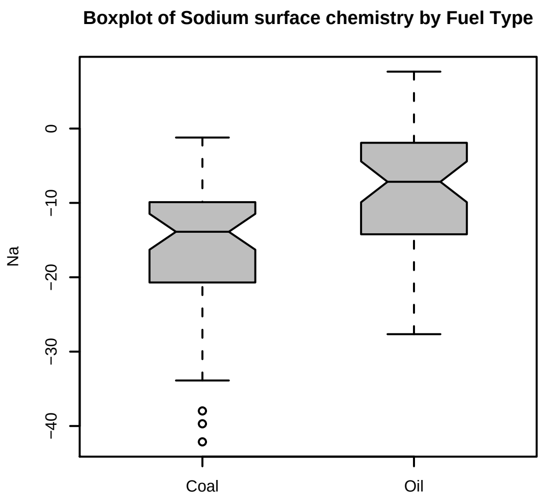


Figure 5: A boxplot of Sodium from the SCP surface chemistry data set by fuel type.



The boxplot for Sodium (Na) in Figure 4 again illustrates that these data are almost normally distributed, the whiskers are about the same length with only a few observations at the lower end extended past the whiskers.

As mentioned earlier, boxplots are particularly useful for comparing the properties of a variable among two or more groups. The `boxplot()` function allows us to specify the grouping variable and the variable to be plotted using a standard formula notation. We will see more of this in the Regression practical. Try the following code snippet:

```
> boxplot(Na ~ FuelType, data = scp.dat, notch = TRUE, col = "grey",
+         ylab = "Na",
+         main = "Boxplot of Sodium surface chemistry by fuel type",
+         boxwex = 0.5, varwidth = TRUE)
```

Figure 5 shows the resulting plot. The formula notation takes the form:

$$\textit{variable} \sim \textit{grouping variable}$$

The only new argument used above is `varwidth = TRUE`, which plots the widths of the boxes proportionally to the variance of the data. This can visually illustrate differences in a variance among groups.

Are the median values for Na split by `FuelType` significantly different at the 95 % level?

Plot boxplots for some of the other variables by `FuelType`. Suggest which variables show different distributions between fuel types.

6 Bivariate and multivariate graphical data analysis

This section of the practical will demonstrate the graphical abilities of R for exploratory data analysis of bivariate and multivariate data. The standard R plotting function is `plot()`. `plot` is a generic function and works in different ways depending upon what type of data is passed as an argument to it.

6.1 Scatter plots

The most simple bivariate plot is a scatter plot:

```
> plot(S ~ Na, data = scp.dat, main = "Scatter plot of sodium against sulphur")
```

The x and y variables are passed as arguments to `plot()` and we define a main title for the plot. There are lots of embellishments one can use to alter the appearance of the plot, but one of the most useful is a scatter plot smoother, which should highlight the relationship between the two plotted variables.

```
> plot(S ~ Na, data = scp.dat,  main = "Scatter plot of sodium against sulphur")
> with(scp.dat, lines(lowess(x = Na, y = S), col = "red", lwd = 1))
```

Here we just renew the original plot for clarity, but technically you only need to enter the `lines()` command if the scatter plot is still visible in the plotting device and was the last plot made. The resulting plot is shown in Figure 6. To understand how this works type the following at the R prompt:

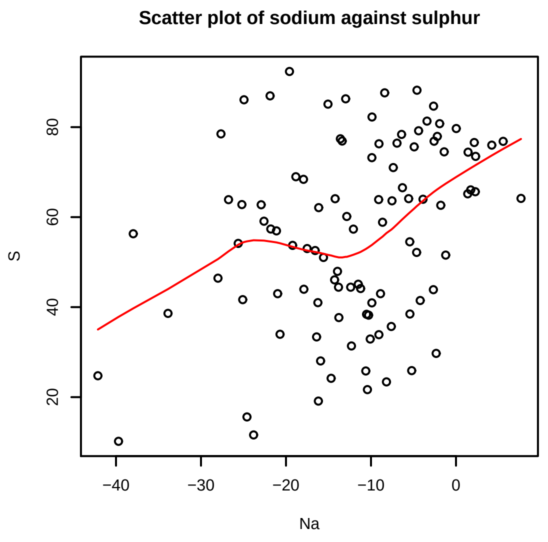
```
> with(scp.dat, lowess(x = Na, y = S))              #hit return
```

The `lowess()` function takes as its arguments (in its most simplest form) two vectors, `x` and `y` corresponding to the two variables or interest, and returns an object with two vectors of points corresponding to the smoothed value fitted at each pair of x and y. The `lines()` function expects to be passed pairs of coordinates (`x` and `y`), and in our example draws a line of width 1 (`lwd = 1`, which isn't strictly necessary as the default is to use whatever `lwd` is set to in `par()`) in red (`col = "red"`). So you can see that we do not need to save the results of `lowess()` in order to plot them. This is a very useful feature of R, being able to nest function calls inside one another, and will be used more frequently throughout the remaining practical classes.

6.2 Coded scatter plots and other enhancements

In the previous section the use of the `plot()` function to draw simple scatter plots was demonstrated. Now we illustrate a few additional R functions that can be used to make better-looking and more informative plots. Try the following:

Figure 6: A simple scatter plot of sodium against sulphur with a LOWESS smoother drawn through the data.



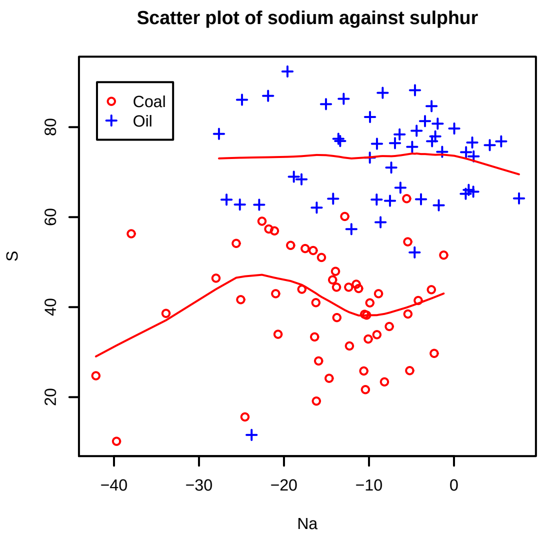
```
> plot(S ~ Na, data = scp.dat, pch = c(21,3)[FuelType],
+      col = c("red", "blue")[FuelType],
+      main = "Scatter plot of sodium against sulphur")
> with(scp.dat, lines(lowess(x = Na[FuelType == "Coal"],
+                            y = S[FuelType == "Coal"]),
+                     col = "red", lwd = 1))
> with(scp.dat, lines(lowess(x = Na[FuelType == "Oil"],
+                            y = S[FuelType == "Oil"]),
+                     col = "blue", lwd = 1))
> legend("topleft", legend = c("Coal", "Oil"), col = c("red","blue"),
+        pch = c(21,3))
```

The resulting plot is shown in Figure 7.

The first command should be mostly familiar to you by now. We have specified the plotting characters (using `pch = c(21,3)`). What is special about this though is that we use the `FuelType` variable to determine which character is plotted. This works because the `c()` function creates a vector from its arguments. It is possible to subset a vector in much the same way as you did earlier for a data frame, but because a vector has only a single column, we only need to specify which *entry* in the vector we want. `FuelType` is a factor and evaluates to 1 or 2 when used in this way, depending on the value of each corresponding `FuelType`. Which character is plotted is determined by whether `FuelType` is “coal” or “oil”. We use the same notation to change the plotting colour from `"red"` to `"blue"`.

A similar method is used to plot the LOWESS scatter plot smoothers for each fuel type. This time however, we only want to fit a LOWESS a single fuel type at a time. So we select values from `Na` and `S` where the corresponding `FuelType` is either “coal” or “oil”. We have to repeat this

Figure 7: A coded scatter plot of sodium against sulphur by fuel type. A LOWESS smoother is drawn through the data points for each fuel type.



twice, once of each level in `FuelType`[2].

The final command make use of the `legend()` function to create a legend for the plot. The first argument to `legend()` should be a vector of coordinates for the upper left corner of the box that will contain the legend. These coordinates need to be given in terms of the *scale* of the current plot region. Instead we use the `locator()` function to set the location of the legend interactively.

Draw coded scatter plots in the same way for other variables in the SCP data set. Which variables seem to best discriminate between the two fuel types?

6.3 Scatter plot matrices

When there are more than two variables of interest it is useful to plot a scatter plot matrix, which plots each variable against every other one. The `pairs()` function is the standard R graphics function for plotting scatter plot matrices[3].

Enter the following code snippet and look at the resulting plot:

```
> pairs(~ Na + Ni + S + Fe, data = scp.dat, gap = 0, panel = panel.smooth)
```

The plot generated is shown in Figure 8.

[2]There are ways of automating this in R code, but this involves writing a custom function using a loop and so is beyond the scope of this practical

[3]I mentioned earlier that `plot()` was a generic function. If you were to pass `plot()` a matrix or data frame containing three or more variables, `plot()` would call pairs to perform the plot instead of say `plot.default()` say, which was used behind the scenes to produce the plots in Figure 7

Figure 8: A scatter plot matrix showing the relationship between Sodium, Nickel, Sulphur and Iron. A LOWESS smoother is plotted though each individual plot.

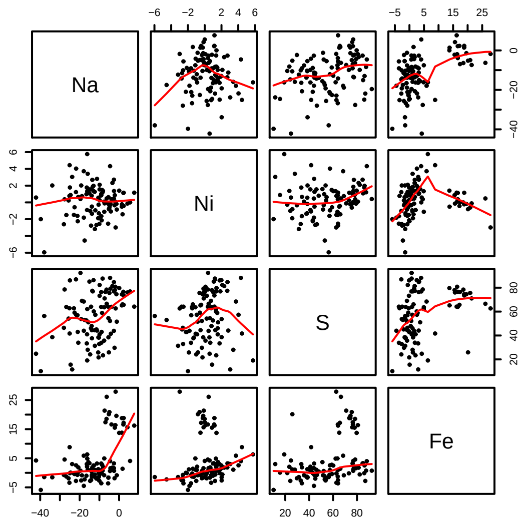
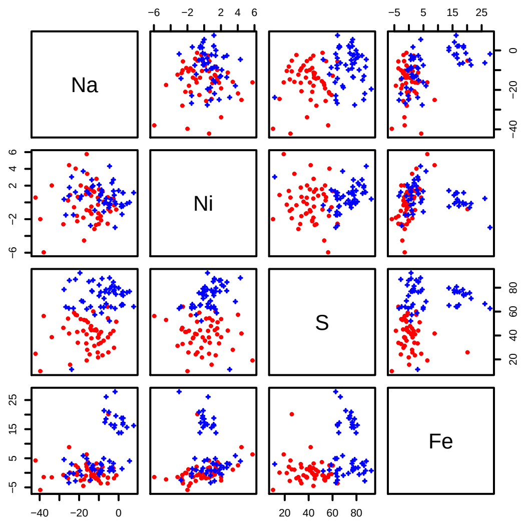


Figure 9: A scatter plot matrix showing the relationship between Sodium, Nickel, Sulphur and Iron. Red circles denote Coal-fired power stations and blue crosses, Oil-fired ones.



Lets take a moment to explain the above functions. Firstly, the `cbind()` function (*c*olumn *bind*) has been used to stitch together the four named vectors (variables) into a matrix (or data frame). `cbind()` works in a similar way to `c()` but instead of creating a vector from its arguments, `cbind()` takes vectors (and other R objects) and creates a matrix or data frame from the arguments. We have done this, because `pairs()` needs to work on a matrix or data frame of variables.

The final argument in the call to `pairs()` is the `panel` argument. The details of `panel` are beyond the scope of this practical, but suffice it to say that `panel` is used to plot any appropriate function on the upper and lower panels of the scatter plot matrix (the individual scatter plots). Later in the practical you will use a custom panel function to add features to the diagonal panel where currently only the variable label is displayed. In the above code snippet we make use of an existing panel function, `panel.smooth` which plots a LOWESS smoother through each of the panels using the default span (bandwidth[4]).

As an alternative to Figure 8, we can plot coded scatter plot matrices like so:

```
> with(scp.dat, pairs(~ Na + Ni + S + Fe, data = scp.dat,
+                     pch = c(21,3)[FuelType],
+                     col = c("red", "blue")[FuelType], gap = 0))
```

This example should be much more familiar to you than the previous one, as the structure of the commands are similar to the single coded scatter plot earlier. The same syntax is used to

[4]You will learn more about the bandwidth parameter in `lowess()` in the practical on Advanced Regression techniques.

subset data into each fuel type and are plotted in different characters and colours. The resulting plot is shown in Figure 9.

Plot different combinations of the variables in the SCP data set in scatter plot matrices. Which variables seem to best discriminate between the two fuel types?