301107 - Analytics Programming

Nick Tothill

Practical class 1 - Excel; setting up R; R environment; basic data handling

Task 1. Clean up the Iris data

Use Excel to produce basic summaries of the iris data (given in vUWS, under Learning Materials; week02; practical). This famous (Fisher's or Anderson's) iris data set gives the measurements in centimetres of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

The data look like:

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Hover, during the data collection process, something went wrong. As a result, some values are missing, and some values are incorrect (with values ≤ 0).

This is a common problem in data science. So the first thing to do is to

- **clean up the data** by removing any record with **either missing values or incorrect values**
- copy the records with *good* values to another sheet and carry on the next task

Task 2. Summarise the Iris data

Given now you have cleaned the data, try to work out some basic statistics of the flowers summarised according to different species. So you produce a table that looks likes the following (can be in different format):

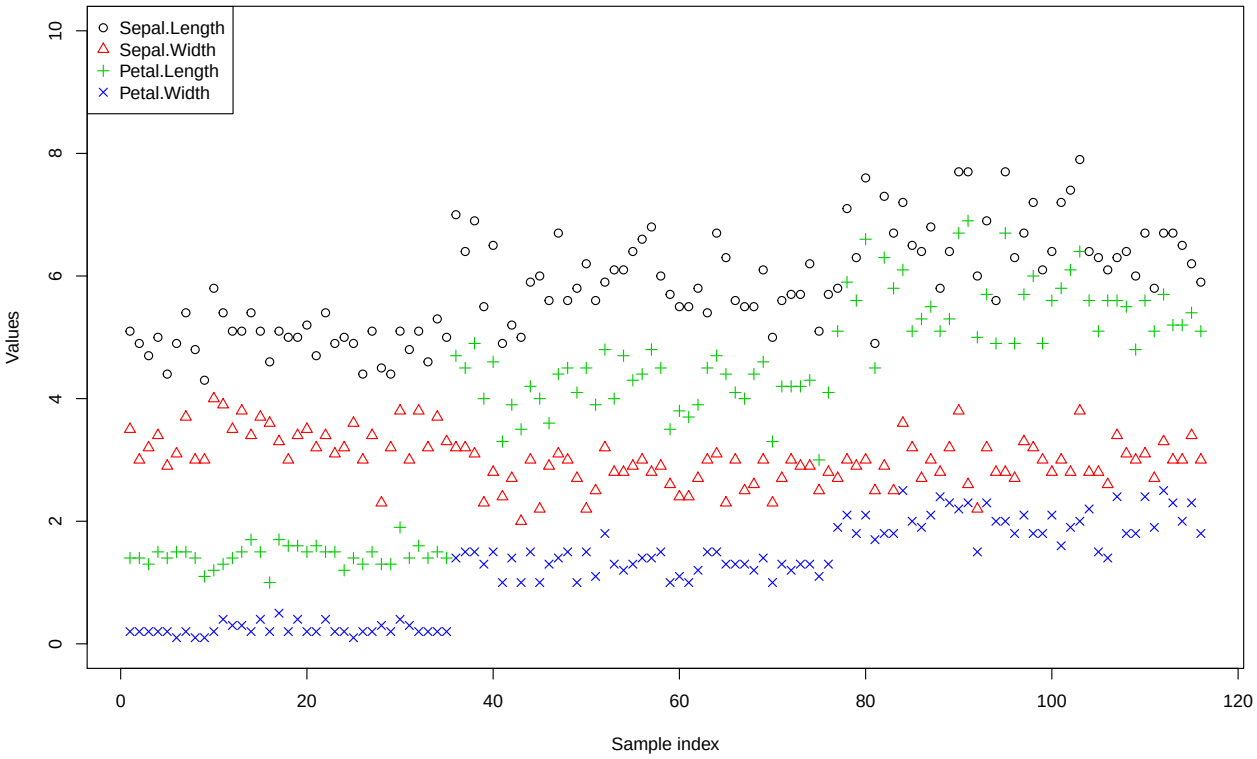
```
[1] "Species: setosa"
  Sepal#Length    Sepal#Width     Petal#Length    Petal#Width
 Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.1000
 1st Qu.:4.750   1st Qu.:3.100   1st Qu.:1.350   1st Qu.:0.2000
 Median :5.000   Median :3.400   Median :1.400   Median :0.2000
 Mean   :4.957   Mean   :3.346   Mean   :1.437   Mean   :0.2371
 3rd Qu.:5.100   3rd Qu.:3.600   3rd Qu.:1.500   3rd Qu.:0.3000
 Max.   :5.800   Max.   :4.000   Max.   :1.900   Max.   :0.5000
[1] "Species: versicolor"
  Sepal#Length    Sepal#Width     Petal#Length    Petal#Width
 Min.   :4.900   Min.   :2.000   Min.   :3.000   Min.   :1.000
 1st Qu.:5.500   1st Qu.:2.500   1st Qu.:3.900   1st Qu.:1.200
 Median :5.800   Median :2.800   Median :4.200   Median :1.300
 Mean   :5.885   Mean   :2.741   Mean   :4.171   Mean   :1.295
 3rd Qu.:6.200   3rd Qu.:3.000   3rd Qu.:4.500   3rd Qu.:1.400
 Max.   :7.000   Max.   :3.200   Max.   :4.900   Max.   :1.800
[1] "Species: virginica"
```

```
 Sepal#Length    Sepal#Width    Petal#Length    Petal#Width
Min.   :4.900   Min.   :2.20   Min.   :4.500   Min.   :1.400
1st Qu.:6.175   1st Qu.:2.80   1st Qu.:5.100   1st Qu.:1.800
Median :6.450   Median :3.00   Median :5.600   Median :2.000
Mean   :6.590   Mean   :2.98   Mean   :5.575   Mean   :2.002
3rd Qu.:7.125   3rd Qu.:3.20   3rd Qu.:5.825   3rd Qu.:2.225
Max.   :7.900   Max.   :3.80   Max.   :6.900   Max.   :2.500
```

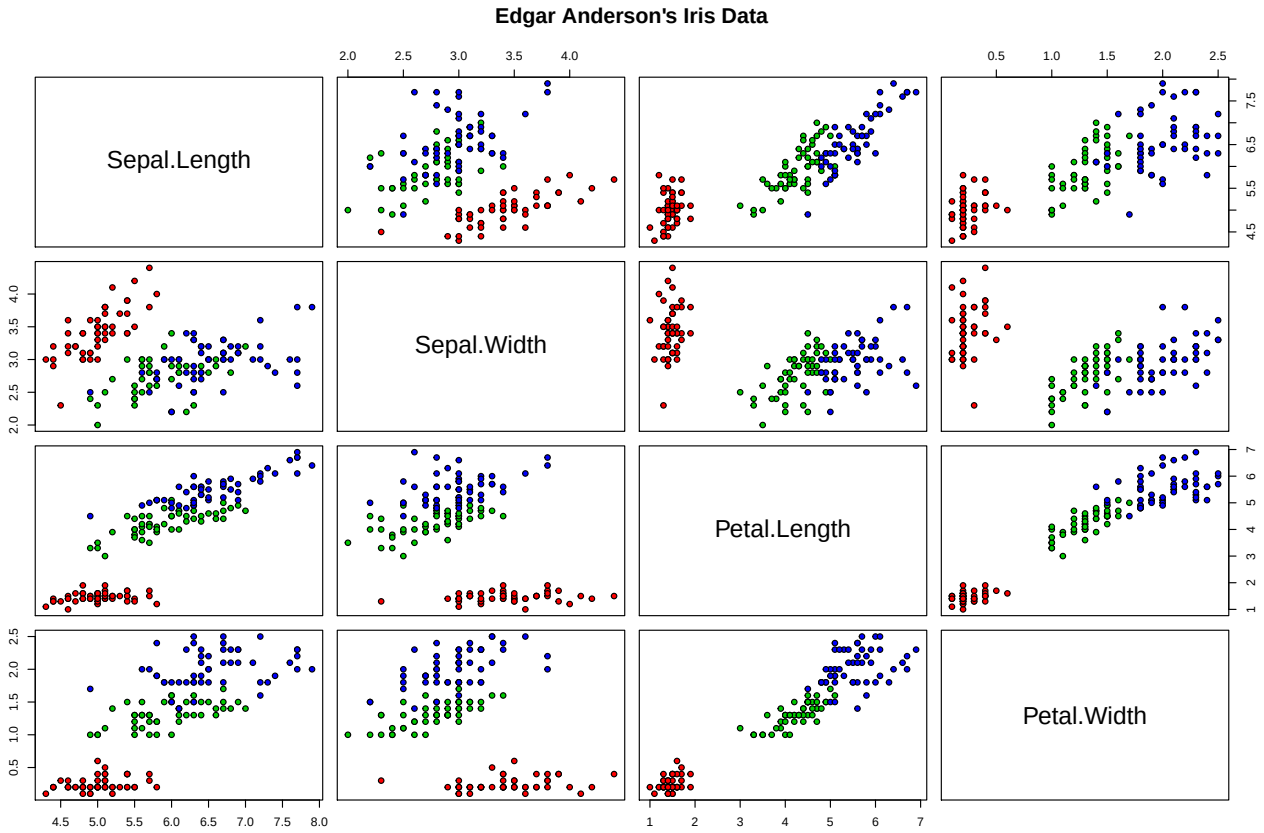
You may notice there are 6 statistics shown above. In this practical class, you *only need to find minimum, maximum and mean* for each variable.

Task 3. Visualise the Data

Plot a figure. Use Excel's X Y (Scatter) chart to produce a plot similar to the following



However, it would be better to separate the species:



Task 4. Starting R

Start Rstudio on your lab computer. If using your own computer, you should have Rstudio installed. If not, download and install while you're working on the practical on a lab computer.

In RStudio, try to run the following code

```
> 1+1
```

```
[1] 2
```

```
> sqrt(2)
```

```
[1] 1.414214
```

```
> print('Hello, world!')
```

```
[1] "Hello, world!"
```

Task 5. Simple R session

Type in the following commands:

```
> x <- 1
> y <- 2
> z <- x+y
> ls()
```

```
> rm(x)
> ls()
> help(ls)
> help.start()
```

Look for *An Introduction to R*.

Go back to RStudio *R Console* panel and continue

```
> ls
> ls <- 1
> ls
> ls()
```

What happens with these different ways of writing `ls`? Notice the changes?

Continue

```
> getwd()
> save.image(file='test.RData')
> history()
```

What is `save.image(file='test.RData')` doing there?

What if I want to save all commands I used so far? (Hint: look up things related to `history`)

```
> rm(list=ls())
> load(file='test.RData')
> ls()
```

We cleared the current workspace by using `rm(list=ls())`. See what the argument `list` of function `rm()` means. What is `load()` doing?

If you have worked out how to save all commands you used in this session, you know how to save your work, which is likely to be important!

Task 6. Practice with vectors

Try out the following code for vectors, and try to figure out what every line is doing

```
x <- seq(0,200,5)
print(paste('x is a vector of length',length(x)))
x
plot(1:length(x),x,main="Plot of vector x", xlab='Index', ylab='Values of elements in x',col=1)

print(paste('The 10th element in x is',x[10]))
cat('The first 5 elements in x are',x[1:5])
cat('The 5th, 7th, and 10th elements in x are',x[c(5,7,10)])
x1 <- x[1:4]
x2 <- x[(length(x)-3):length(x)]
x1
x1[-1]
x2
x1 + x2
-x1
x1 * x2
any(x1>5)
```

```
which(x1>5)
x1[which(x1>5)]
x1[1] <- NA
x1
which(is.na(x1))
x1[-which(is.na(x1))]
all(x2<200)

x1 <- x1[-1]
x1
x1-5
x1*-1
sum(x1)
rep(1,length(x2))
```

There are quite a few indexing and manipulating functions in the above code, as well as a plotting function `plot()`. Find out what they do to a vector (hint: use help.)

After you understand the above, finish the following task, using the functions you have learned to make the code as simple as possible.

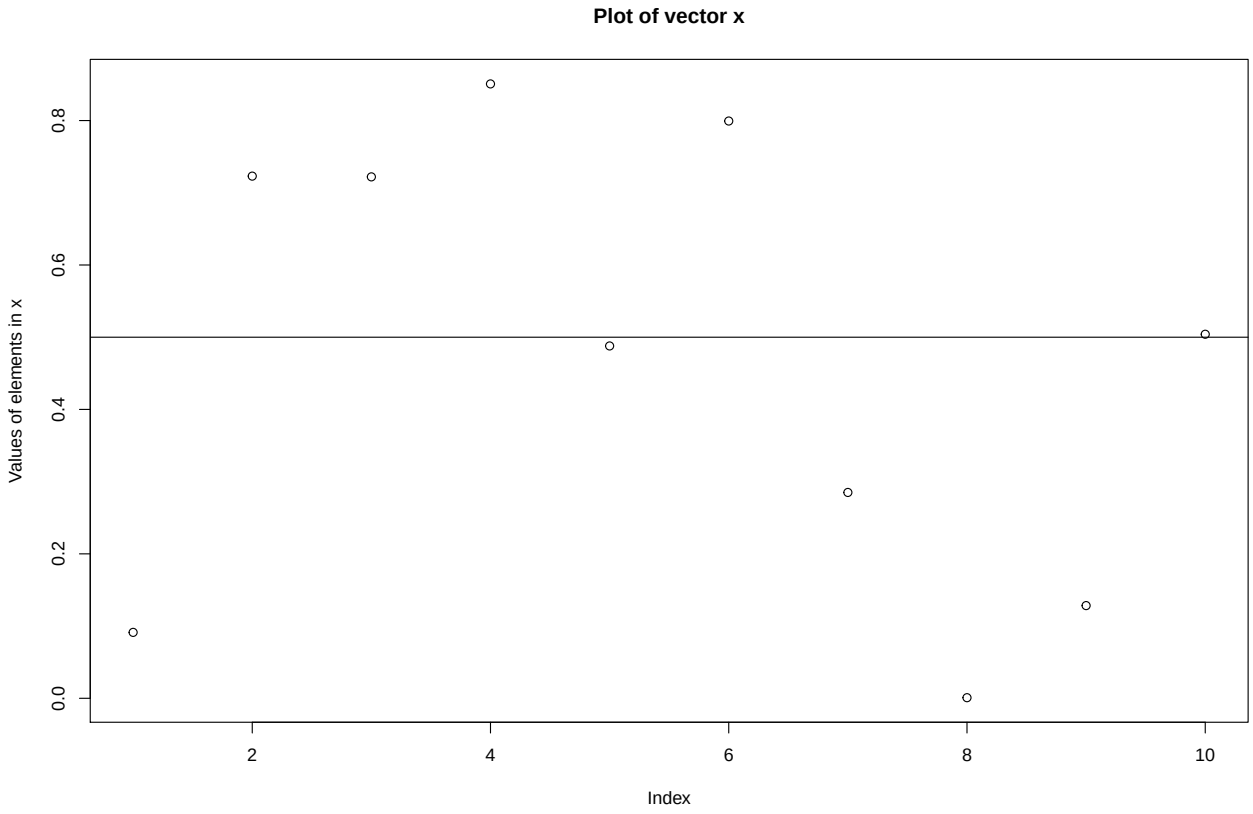
1. generate a random vector of length 10 (the function you need is `runif(10)`);
2. calculate the sum of the samples stored in the vector;
3. calculate the mean of the samples;
4. find all samples that are no less than 0.5 and calculate their mean;
5. find all samples that are less than 0.5 and calculate their mean.

You should have the results similar to the following:

```
My vector is
 0.0912563 0.7230549 0.7220213 0.8507309 0.48788 0.7993466
 0.2850184 0.0008180665 0.128345 0.5040901

Sum of samples: 4.592562

Mean of samples: 0.4592562

Mean of samples no less than 0.5: 0.7198488

Mean of samples less than 0.5: 0.1986636
```

your vector will have different elements in it, as, every time you run `runif` it will generate different values drawn from a uniform distribution between 0 and 1. So your results will be different from the above.

You can plot the vector here to get a visualisation of the vector like the following. The command to use is `plot`. The line in the middle (which is the mean of the data in the vector) is produced by another command `abline`. Use the help pages in R to see if you can figure out how to make a plot like the one shown below.



Task 7. R self learning package swirl

A. install R package `swirl`
1. Start RStudio
2. Type in `install.packages('swirl')`. It will download and install the package.
3. load in the library by using `library(swirl)`
B. Have a `swirl` learning session

N.B. The learning module may take a long time to download.

301107 - Analytics Programming

Nick Tothill

Practical class 02, week 03 - R data structures

In this class, you will familiarise yourselves some basic R data structures and learn a few ways to manipulate them to get the results you need.

1. Generate a vector of length 20 sampled from a normal distribution with mean 0 and standard deviation (sd) 1 (the function you need is `rnorm()`. You can access the help page with `help(rnorm)`. Here is one such vector:

```
 [1]  1.69734022 -1.76946135 -0.32203377 -1.17504365 -0.65729986 -2.24762667  0.48651801
 [8] -0.91165682 -1.52732050  0.91338578  0.13556346 -1.63381026 -0.26077803  0.03595732
[15] -0.63526510  0.21294543  1.66424042 -0.04615740 -0.55399795 -0.68341268
```

Every time you run or re-run `rnorm()`, you'll get different numbers in the vector.

2. Print the number of positive values and negative values and flip the negative values to positive values (the functions you need are `sum()`,`print()`,`paste()` or simply `cat()`)

```
There are 7 positive ones and 13 negative ones in the vector
Now the vector looks like after flipping negative values:

 [1] 1.69734022 1.76946135 0.32203377 1.17504365 0.65729986 2.24762667 0.48651801
 [8] 0.91165682 1.52732050 0.91338578 0.13556346 1.63381026 0.26077803 0.03595732
[15] 0.63526510 0.21294543 1.66424042 0.04615740 0.55399795 0.68341268
```

Hint: There are various ways of doing this correctly. A particular useful one is to use filtering and data type conversion.

3. Divide the values in the vector into 3 equal-sized bins and count how many values fall into each bin (the functions you may need are `seq()`,`max()`,`min()`, `range()` …)

```
Borders of the bins:

[1] 0.03595731 0.77318043 1.51040356 2.24762668

bin 1 bin 2 bin 3
   11     3     6
```

Hint: There is a very simple implementation of this task which is using the *factors* and related function `cut()`.

4. Fill this vector into a matrix with 3 rows and 6 columns. Observe how the matrix is filled and excessive values are discarded.

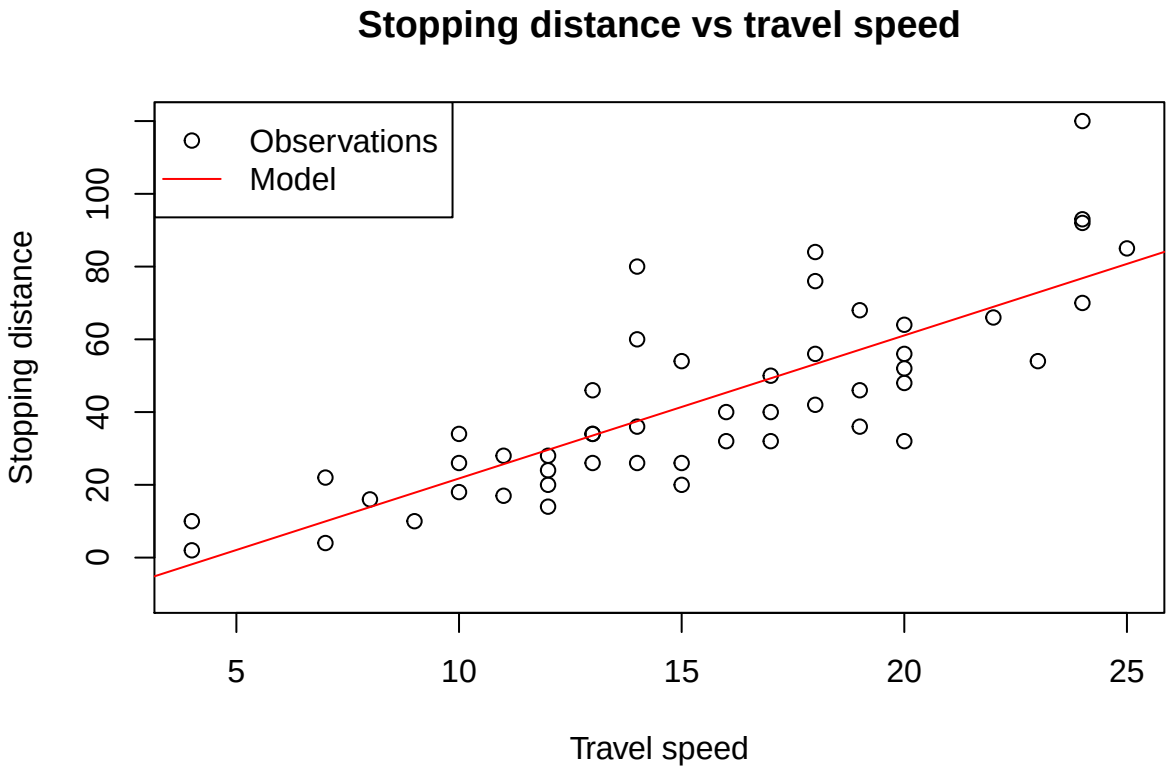
```
          [,1]      [,2]      [,3]      [,4]       [,5]      [,6]
[1,] 1.6973402 1.1750437 0.4865180 0.9133858 0.26077803 0.2129454
[2,] 1.7694614 0.6572999 0.9116568 0.1355635 0.03595732 1.6642404
[3,] 0.3220338 2.2476267 1.5273205 1.6338103 0.63526510 0.0461574
```

Fill in the matrix by rows instead like the following (the function you may need is `t()`):

```
         [,1]       [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 1.697340 1.76946135 0.3220338 1.1750437 0.6572999 2.2476267
[2,] 0.486518 0.91165682 1.5273205 0.9133858 0.1355635 1.6338103
[3,] 0.260778 0.03595732 0.6352651 0.2129454 1.6642404 0.0461574
```

5. Linear regression. Use data in `cars` data set, build a linear regression model. Here is some `cars` data

```
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
```



From the above speed against stopping distance plot, we see that the stopping distance is almost linearly related to travel speed. As such we try to regress distance onto speed to see how this linear model performs.

The model is the following

$$Y = wX + b + E$$

where $Y,\ X,\ E$ are random variables for stopping distance, travel speed, and error, $w,\ b$ are model parameter. In statistics, Y is called *response variable* and X is called *explanatory variable*. Sometimes there may be more than one explanatory variable, in which case the model becomes

$$Y = w_1X_1 + w_2X_2 + \ldots + w_dX_d + b + E$$

where X_i for $i = 1, \ldots d$ is the ith explanatory variable and d is the number of total explanatory variables. We need to obtain the values for w and b using our observations. There are many ways to do this. One widely used is *the least squares (LS)*. The LS solution is given by

$$[w,\ b] = \mathbf{X}^{+}\mathbf{y}$$

where

$$\mathbf{X} = \begin{bmatrix} x_1, 1 \\ x_2, 1 \\ x_3, 1 \\ \vdots, \vdots \\ x_n, 1 \end{bmatrix}$$

$\mathbf{y} = [y_1, y_2, \ldots, y_n]$ and $\mathbf{X}^+ = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top$ is the pseudo inverse of matrix $\mathbf{X}$. $x_1, \ldots, x_n$ are the observaions of speed and likewise $y_1, \ldots, y_n$ are observations of distance in `cars` data set. In this task, you have to estimate w and b by using LS and plot the figure as shown above. You may use `plot()`, `abline()`, `legend()` to produce the figure.

There is a very easy and simple way to do this by using `lm()` which uses *data frame* and *formula* as input. Try to use `lm()` to get the same results.

Practical 3 - Analytics programming (301107)

Graeme Wong & Nick Tothill

Recap from last week

Generating a list (i.e. a vector) of random numbers

Generating a vector of length of 20 sampled from a normal distribution with mean 0 and standard deviation (sd) 1. By default the mean and standard deviation is already set to 0 and 1, respectively. Refer to the R documentation.

```
rnorm(20, mean = 0, sd = 1)
rnorm(20)
```

To use the same combination of values for other tasks, assign the vector to 'x'. Otherwise a different combination of numbers will be generated.

```
x <- rnorm(20)
x # Show the output of x
```

```
##  [1] -1.12786124  2.08492842  0.71975695  1.94424690 -0.35668623
##  [6] -0.31958903 -0.48476670 -0.35884384 -0.37293078  0.36360010
## [11] -0.03396173  0.53624519 -0.10067629 -0.15556646 -1.30880607
## [16] -2.43711358  0.67105053 -0.88106509  0.42455008  0.66538626
```

Data processing with these numbers

The following shows how you can view the positive and negative numbers

```
x[x<0]  # negative numbers
```

```
##  [1] -1.12786124 -0.35668623 -0.31958903 -0.48476670 -0.35884384
##  [6] -0.37293078 -0.03396173 -0.10067629 -0.15556646 -1.30880607
## [11] -2.43711358 -0.88106509
```

```
x[x>=0]  # Positive numbers
```

```
## [1] 2.0849284 0.7197569 1.9442469 0.3636001 0.5362452 0.6710505 0.4245501
## [8] 0.6653863
```

N.B. we treat zero as a positive number, so use >=, instead of just >

Using the 'sum()' function, we can count the values that are positive and negative.

```
cat('There are',sum(x>0),'positive ones and',sum(x<0),'negative ones in the vector')
```

```
## There are 8 positive ones and 12 negative ones in the vector
```

There are three ways to invert the values that are below 0, using different mathematical operations:

1. subtract the negative numbers from zero: $-(-1.5) = 0 - -1.5 = 1.5$
2. multiply the negative numbers by -1: $-1 \times -3.45 = 3.45$
3. Use the absolute function abs(), which returns the absolute size of the number without the sign

If you put a single value next to a vector, R will 'recycle' the single value and apply it across the entire vector.

```
-x[x<0]  # minus the negative numbers
```

```
##  [1] 1.12786124 0.35668623 0.31958903 0.48476670 0.35884384 0.37293078
##  [7] 0.03396173 0.10067629 0.15556646 1.30880607 2.43711358 0.88106509
```

```
-1*x[x<0]  # -1 times the negative numbers
```

```
##  [1] 1.12786124 0.35668623 0.31958903 0.48476670 0.35884384 0.37293078
##  [7] 0.03396173 0.10067629 0.15556646 1.30880607 2.43711358 0.88106509
```

```
abs(x)
```

```
##  [1] 1.12786124 2.08492842 0.71975695 1.94424690 0.35668623 0.31958903
##  [7] 0.48476670 0.35884384 0.37293078 0.36360010 0.03396173 0.53624519
## [13] 0.10067629 0.15556646 1.30880607 2.43711358 0.67105053 0.88106509
## [19] 0.42455008 0.66538626
```

Options 1 and 2 extract and convert the values which are below 0, so you need to add them back into the your original vector. Option 3 applies the mathematical function on the entire vector.

```
cat('\nTaking option 1 and subsituting the converted values back into x')
```

```
##
## Taking option 1 and subsituting the converted values back into x
```

```
x[x<0] <- -x[x<0]
```

Bin your data!

To do this you first need to work out the bins - these N are equally-spaced intervals which together contain all your data. You can do this manually and then use 'seq' to create your ranges, alternatively we can use the minimum and maximum values, and create N bins that cover this range.

```
# First create your data ranges (in this case I am using 5 bins)
offset=1e-8
bins <- seq(min(x)-offset,max(x)+offset,length.out=5+1)
cat('The min and max ranges for the different bins:\n',bins)
```

```
## The min and max ranges for the different bins:
##  0.03396172 0.5145921 0.9952225 1.475853 1.956483 2.437114
```

NB: 1. the offset is introduced so that the top and bottom ends of the bin range lie *just outside* the minimum and maximum data values; so the maximum and minimum values are counted in the bins. 2. If you are creating 5 bins you need 5 data ranges which means you need 6 numbers (the minimum and maximum range per bin).

Next step is to divide your data based on the data ranges and put it into a table. The commands table() and cut() will be your friends here.

```
table(cut(x,bins,labels = c('Bin 1', 'Bin 2','Bin 3','Bin 4','Bin 5')))
```

```
##
## Bin 1 Bin 2 Bin 3 Bin 4 Bin 5
##    10     5     2     1     2
```

```
# where x is the data and bins is the min and max ranges for each bin.
```

Continuing on - Data Manipulation, Linear Regression and Visualisation

Reformat the vector into a matrix

Use the vector of random numbers to fill up a matrix with 3 rows and 6 columns. Observe how the matrix is filled and extra values are discarded.

```
NX <- 3
NY <- 6
X <- matrix(x,NX,NY)
```

```
## Warning in matrix(x, NX, NY): data length [20] is not a sub-multiple or
## multiple of the number of rows [3]
```

```
X
```

```
##           [,1]      [,2]      [,3]       [,4]      [,5]      [,6]
## [1,] 1.1278612 1.9442469 0.4847667 0.36360010 0.1006763 2.4371136
## [2,] 2.0849284 0.3566862 0.3588438 0.03396173 0.1555665 0.6710505
## [3,] 0.7197569 0.3195890 0.3729308 0.53624519 1.3088061 0.8810651
```

Fill the matrix by rows instead like the following (the function you may need is `t()`):

```
X <- t(matrix(x,NY,NX))
```

```
## Warning in matrix(x, NY, NX): data length [20] is not a sub-multiple or
## multiple of the number of rows [6]
```

```
X
```

```
##           [,1]      [,2]      [,3]      [,4]       [,5]      [,6]
## [1,] 1.1278612 2.0849284 0.7197569 1.9442469 0.35668623 0.3195890
## [2,] 0.4847667 0.3588438 0.3729308 0.3636001 0.03396173 0.5362452
## [3,] 0.1006763 0.1555665 1.3088061 2.4371136 0.67105053 0.8810651
```

Linear Regression

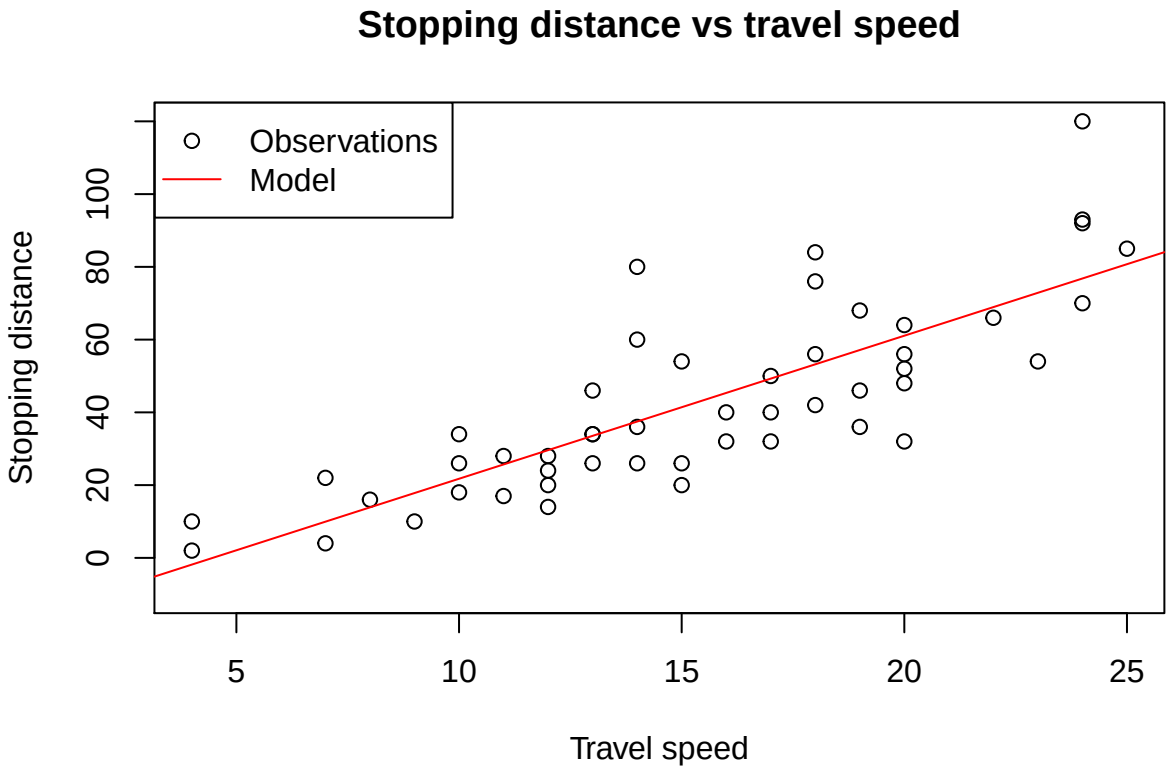
Using data in `cars` data set, build a linear regression model. Here are some `cars` data.

```
?cars
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

'?cars' will extract the dataset into a *data frame* with columns for speed and stopping difference (see details in your RStudio Environment window).

To can carry out a linear regression use the function 'lm' (linear model); 'abline' will help with plotting the line.



Data manipulation within R

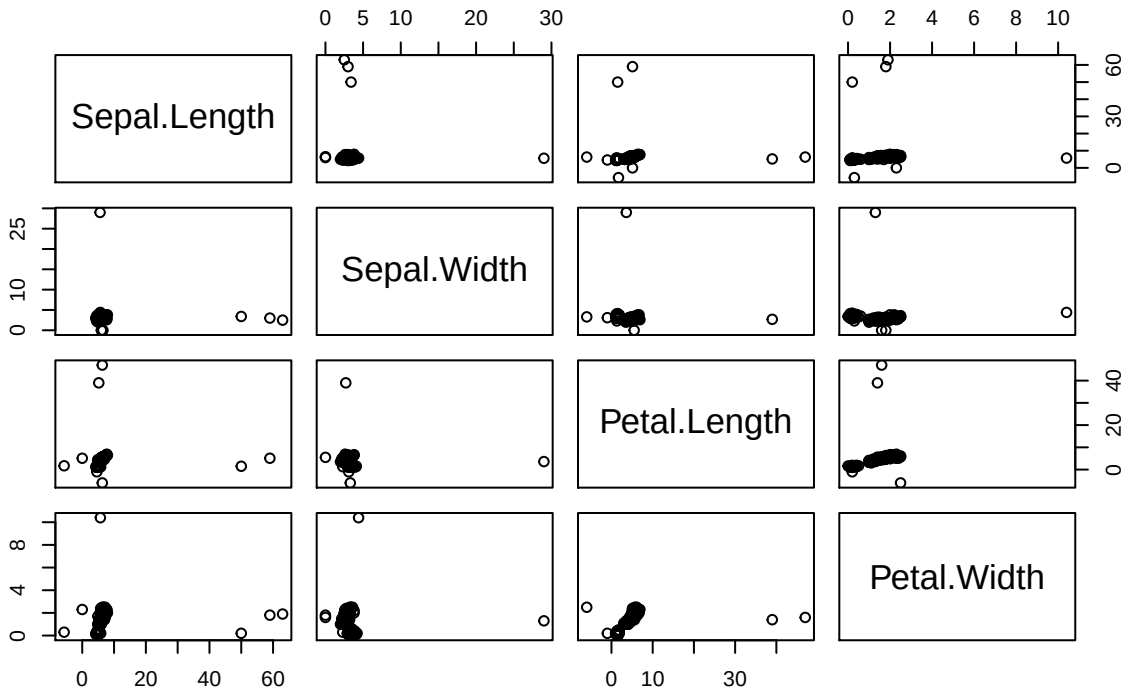
Back in Practical 1, you were asked to use Excel to produce summaries of the iris data. As a part of the task you needed to clean up the data (it was advised you use the filter or sort commands to identifiy and then remove the bad data).

Task 1. Reading in iris data

You will need to download the "iris raw text file" and examine the contents. Each record (one observation of a plant) has 5 fields, sepal length, sepal width, petal length, petal width and species. You will need to be careful here as the data is stored as one continuous line.

5.6,2.7,4.2,1.3,versicolor,7.6,3,6.6,2.1,virginica,

If you just plot the data you get the following.



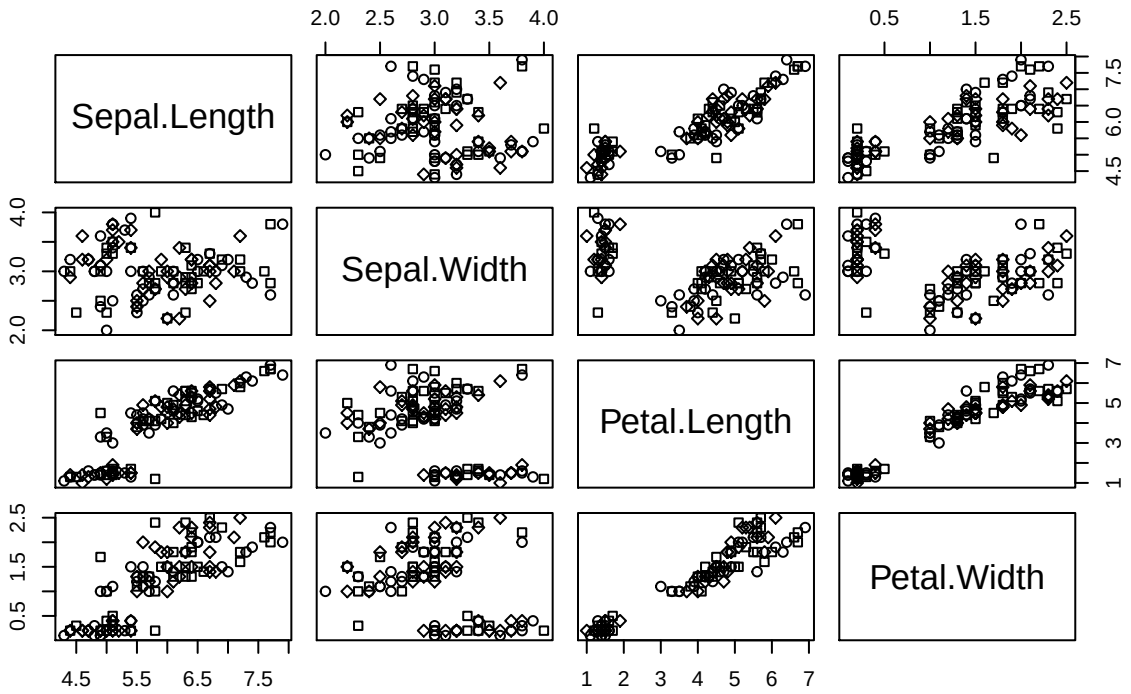
Your task here is to read in this text file in R, clean the data (remove all invalid ones that cannot be recovered), reformat it into a data frame in the format the same as iris in R. In other words, all records are concatenated together and the fields are separated by comma. Moreover, you will need to correct the following

1. Missing values
2. negative values
3. too large values
4. zeros values
5. mis-spelling
6. duplicate records

Examine the 'read' command and the parameters that will be needed to import the data (col.names will be your friend here). The information above will help you get the values below 0 and NA values. Commands 'subset' will be needed to extract the useful data, while 'duplicated' will help you identify the duplicates (see R documentation for more info). Swirl lessons 2 and 12 will be useful here.

After you clean the data you will have 111 records (the mis-spelling can be corrected). The code to create the pairwise plot is shown below as well as the end result of the data clean:

```
colors <-c("red2","green3","blue","black")
plot(clear_data[,1:4],main='Cleaned data',pch=c(21,22,23),bg=colors[clear_data$Species])
```



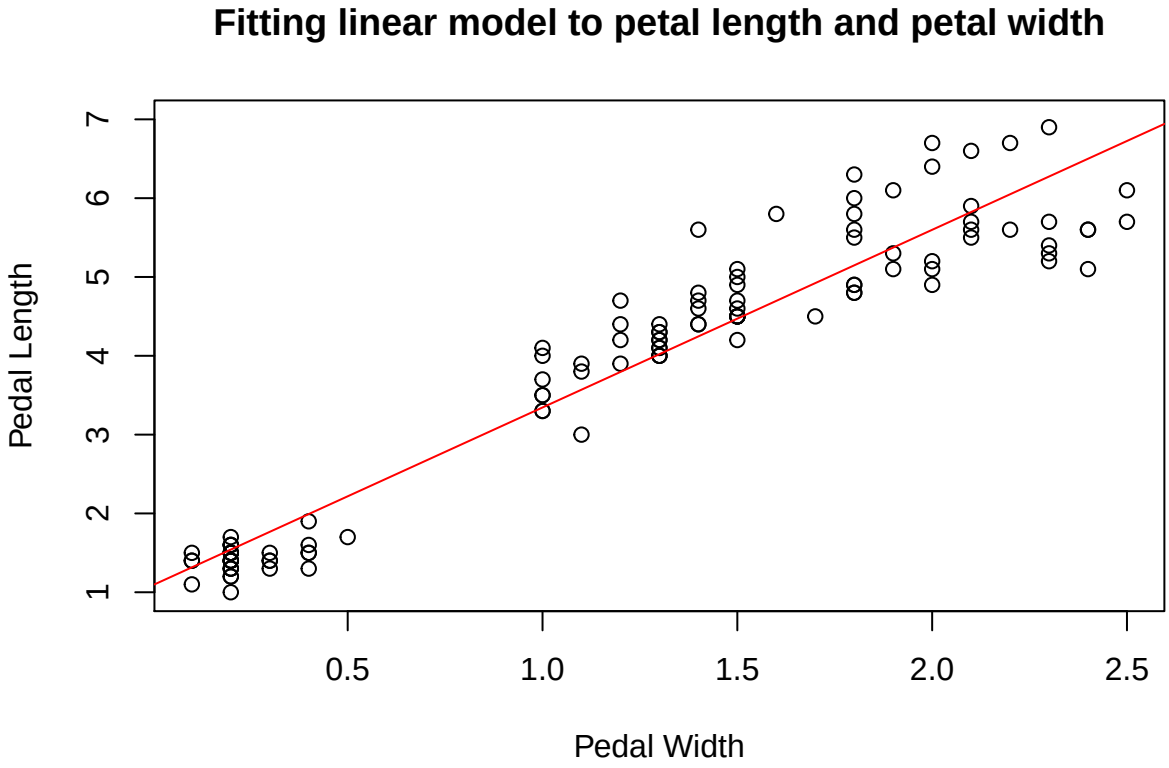

```
str(clear_data)
```

```
## 'data.frame':    111 obs. of  5 variables:
##  $ Sepal.Length: num  5.6 7.6 5.1 6.7 5.1 6 5.7 5.5 6.4 7.7 ...
##  $ Sepal.Width : num  2.7 3 3.5 3.3 3.3 2.2 2.6 2.4 3.2 2.6 ...
##  $ Petal.Length: num  4.2 6.6 1.4 5.7 1.7 4 3.5 3.8 4.5 6.9 ...
##  $ Petal.Width : num  1.3 2.1 0.2 2.1 0.5 1 1 1.1 1.5 2.3 ...
##  $ Species     : chr  "versicolor" "virginica" "setosa" "virginica" ...
```

Task 2. Linear Regression

From the pairwise plot given in task 1, we observe that petal length and petal width have almost linear structure, i.e. if petal length is large, then petal width is large proportionally. So fit this pair using a linear model we have learnt previously to see if linear model fits well. You can fit 3 separate linear models for each species and produce the plot.

NB: This is similar to carrying out a linear regression for the cars exercise.


Fitting linear model to petal length and petal width
Pedal Length
1
2
3
4
5
6
7
0.5
1.0
1.5
2.0
2.5
Pedal Width

Practical 4 & 5 - Analytics programming (301107)

Project: Conway's Game of Life

Graeme Wong & Nick Tothill

Some tips on how to handle the problem

This document will give you an idea of the steps that will be required to complete the tasks in practicals 4 and 5. In this exercise you will be implementing the Conway's Game of Life. To implement this game you will need an understanding of the following:

1. Generate a random binomial distribution
2. Viewing elements in a matrix
3. Loops and conditional statements
4. Visualising results

Binomial distribution and Viewing elements in a matrix

In previous practicals there has been various ways to view a matrix/data frame. The following is an example on how to generate a binomial distribution and then viewing a column, row, and element.

```
# Generate a matrix with a Binomial Distribution
starting_point <- matrix(rbinom(10 * 10,20,0.45),10)
starting_point   # just for viewing purposes
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]   13    8   10   13   10   11   10    9   10     8
##  [2,]    3   10   14    5    8   10    9    9   12    10
##  [3,]    9    9    8    7    8   11   11   10    6    11
##  [4,]    7    5    8    9    8    9    7    6   11    11
##  [5,]    7    8   10    7    7    9    9    6   12    10
##  [6,]   12    7    9   11   12    9    8    7    4     6
##  [7,]    9    9   10    8   11    8   10    7   11     8
##  [8,]    8   10    9   11    7    7   12    5   12    11
##  [9,]    7   10   14    7    6   11    8    9   11    10
## [10,]    9   11    8    8   11    9    9    9    8    11
```

```
cat(" Column 6: ",starting_point[,6],
    "\n Row 3: ",starting_point[3,],
    "\n Element 3,6: ",starting_point[3,6]
    )
```

```
##  Column 6:  11 10 11 9 9 9 8 7 11 9
##  Row 3:  9 9 8 7 8 11 11 10 6 11
##  Element 3,6:  11
```

As a part of Practical 4 you will need to inspect individual elements and then make the appropriate assessment based on the game of life. To achieve this you will need to loop through the rows and columns.

```
for (x in 1:10){c
  for (y in 1:10){                # loop through the row
    cat(starting_point[x,y]," ")  # view the element
```

```
  }
  cat("\n")   # this is just creating a new line nothing to worry about
}
```

```
## 13  8  10  13  10  11  10  9  10  8
## 3  10  14  5  8  10  9  9  12  10
## 9  9  8  7  8  11  11  10  6  11
## 7  5  8  9  8  9  7  6  11  11
## 7  8  10  7  7  9  9  6  12  10
## 12  7  9  11  12  9  8  7  4  6
## 9  9  10  8  11  8  10  7  11  8
## 8  10  9  11  7  7  12  5  12  11
## 7  10  14  7  6  11  8  9  11  10
## 9  11  8  8  11  9  9  9  8  11
```

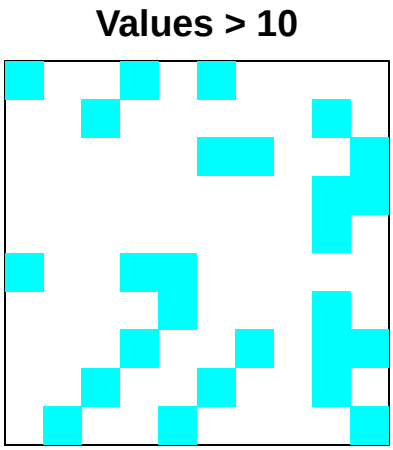
Visualise the data based on a condition

You have a matrix and you want to visualise numbers which are above a threshold (in this case above 10) as a 2D map. To make an assessment of the element, you will need to use an "if" statement. In addition, you want to create another matrix which is blank (values are all zero). This will be used as part of visualising the results. As you loop through the entire matrix you check the elements to see if the value is above 10, if it is then you want to indicate the true value in the blank matrix.

```
zero <- matrix(0,10,10)              # "blank" matrix used for visualising the data
for (x in 1:10){                     # loop through the columns
  for (y in 1:10){                   # loop through the rows
    if (starting_point[x,y] > 10){   # conditional statement to assess element
      zero[x,y] <- 1                 # if true, make an indication in the "blank" matrix
    }
  }
}
```

To visualise the matrix as a map.

```
par(pty = "s",mai=c(0.1,0.1,0.4,0.1))                    # your plotting region is in a square
# visualise the result
rotate <- function(x) t(apply(x, 2, rev))
image(rotate(zero),col=c(0,5),axes=FALSE, frame.plot=TRUE,main="Values > 10")
```



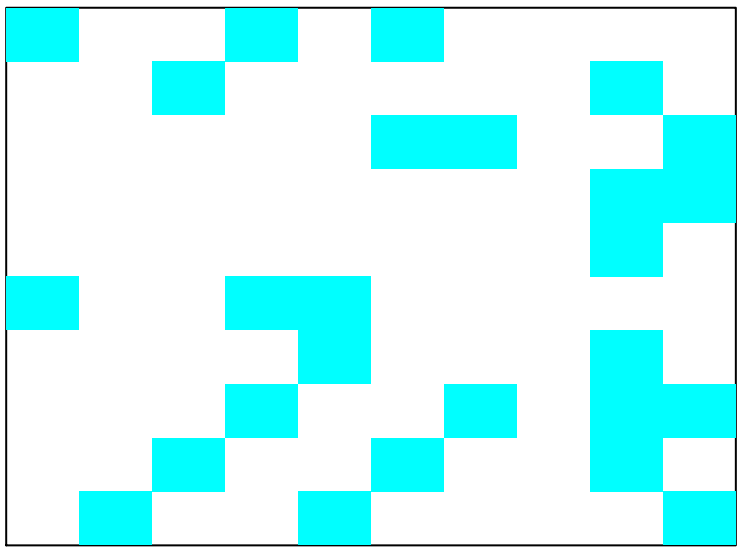
Summary of the results: ## Initial matrix

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]   13    8   10   13   10   11   10    9   10     8
##  [2,]    3   10   14    5    8   10    9    9   12    10
##  [3,]    9    9    8    7    8   11   11   10    6    11
##  [4,]    7    5    8    9    8    9    7    6   11    11
##  [5,]    7    8   10    7    7    9    9    6   12    10
##  [6,]   12    7    9   11   12    9    8    7    4     6
##  [7,]    9    9   10    8   11    8   10    7   11     8
##  [8,]    8   10    9   11    7    7   12    5   12    11
##  [9,]    7   10   14    7    6   11    8    9   11    10
## [10,]    9   11    8    8   11    9    9    9    8    11
```

Values above 10 (boolean)

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    0    0    1    0    1    0    0    0     0
##  [2,]    0    0    1    0    0    0    0    0    1     0
##  [3,]    0    0    0    0    0    1    1    0    0     1
##  [4,]    0    0    0    0    0    0    0    0    1     1
##  [5,]    0    0    0    0    0    0    0    0    1     0
##  [6,]    1    0    0    1    1    0    0    0    0     0
##  [7,]    0    0    0    0    1    0    0    0    1     0
##  [8,]    0    0    0    1    0    0    1    0    1     1
##  [9,]    0    0    1    0    0    1    0    0    1     0
## [10,]    0    1    0    0    1    0    0    0    0     1
```

The visual representation



301107 - Analytics Programming

Project: Conway's Game of Life

Nick Tothill, Graeme Wong and Yi Guo

Practical Class 5 & 6 - R Programming

In this practical class we are going to build an R project to implement the famous Conway's game of life and visualise the whole thing in a sequence of images. Descriptions about Conway's Game of Life can be found here: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life. However, two paragraphs (quoted below) suffice to explain it.

> The universe of the Game of Life is an infinite two-dimensional orthogonal cell of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
>
> 1.Any live cell with fewer than two live neighbours dies, as if caused by under-population.
>
> 2.Any live cell with two or three live neighbours lives on to the next generation.
>
> 3.Any live cell with more than three live neighbours dies, as if by over-population.
>
> 4.Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Here we implement this game in a finite square plane, of size say 10 by 10. You could imagine this as a square plate containing bacterial colonies. So we start with a plate like this:

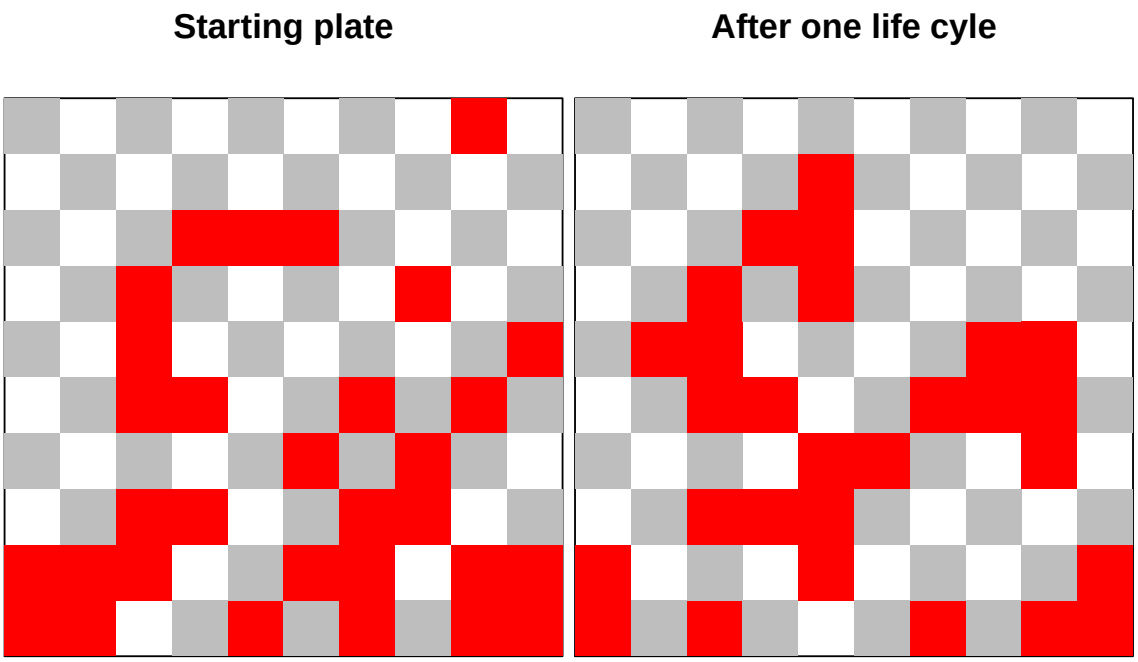


Figure 1: Starting plate and the plate after one life cycle

In Fig. 1, each cell (i.e. square) on the plate is a bacterium. The red ones are alive and white and grey ones are dead. There is no difference between white and grey dead ones (the white and grey colours are used just to show the cells clearly). We can also represent live bacteria by 1s and dead by 0s. Now we can use a matrix as a plate, and the elements of the matrix (1s and 0s) are then the status of bacteria.

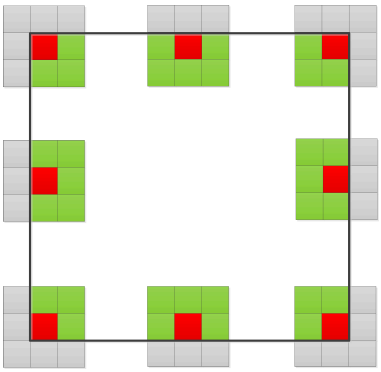


Figure 2: Illustration of neighbours on edges. Thick black rectangle shows the edges of the plate. Red dot is the central cell. They are on edges at different locations. Green ones are neighbours actually inside the plate. Gray ones are imaginary neighbours that are always treated as dead bacteria.

The left panel shows the starting plate. The colonies of bacteria are generated randomly using

```
matrix(rbinom(n*n,1,prob=0.3),n)
```

which is sampling from a *binomial* distribution with probability 0.3 in n^2 trials each of which is size 1. A Binomial distribution with size 1 is essentially a Bernoulli distribution, which deals with the probability of observations with *two* values. The Bernoulli distribution can be simply thought as the probability of flipping a coin, but the coin can be unfair, i.e. one outcome can be more probable than the other. This is reflected in `rbinom()` call with `prob=0.3`.[1] Because each cell in the matrix can only have one of two values (1 or 0, alive or dead), the process is like flipping a coin for each cell, and the Bernoulli distribution is applicable.

Then we apply the above set of rules 1–4. So some bacteria die and some are reborn. Note that not every cell has 8 neighbours, for example, the ones on the border of the plate. In that case, we still imagine those cells have 8 neighbours stretching out of the plate, but if the neighbours are **not in** the plate, we simply treat them as dead ones (illustrated in Fig. 2).

To sum up,

- we use a matrix as the data structure to represent the plate;
- the matrix has only 1s and 0s representing live and dead bacteria, respectively;
- the current matrix stands for current state of the bacteria;
- we apply the rules (1–4 at the beginning of this document) to determine the state of colonies of bacteria, i.e. change elements of the matrix according to the rules. This is called a life cycle;
- we observe many life cycles to see how the colonies are evolving.

What you need to do:

1. implement this game in R with random starting plate;
2. visualise the life cycles as images;
3. tidy up your code to create function(s) taking input arguments including (at least) the size of plate and how many life cycles;

4*. change the starting plate to some other patterns instead of random;

5*. edit the rules of the game, for example, add some rules to see their effect;

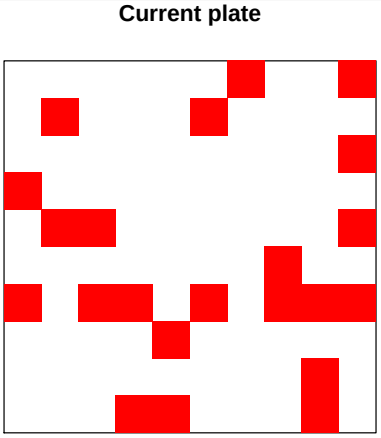
6**. save the images of life cycles into a .gif file so you can play it outside of R.

[1] You can refer to `https://en.wikipedia.org/wiki/Bernoulli_distribution` for details about this probability distribution.

The ones with * and ** are not essential, but are valuable exercises.

There are only a couple of other functions you may need for this project. Use `image(M,axes=FALSE, frame.plot=TRUE,col=c(0,2),main='Current plate')` to visualise the matrix and `par(pty='s')` before you call `image` to set the axes to be square. For example

```
> M <- matrix(rbinom(n * n,1,0.3),n)
> par(pty = "s")
> image(M,axes=FALSE, frame.plot=TRUE,col=c(0,2),main='Current plate')
```



So you don't have to produce a check board as in Fig. 1. The other function you may use is `Sys.sleep(t)` which will wait for `t` seconds and then move on. This is useful in generating an animation as demonstrated below:

```
for(i in 1:4)
{
  M <- matrix(rbinom(n * n,1,0.3),n)
  par(pty = "s")
  image(M,axes=FALSE, frame.plot=TRUE,col=c(0,2),main=i)
  Sys.sleep(0.3)
}
```

If you run the above code in RStudio, you will see the "animation" is played in the `Plots` panel with 4 frames played one after another for 0.3 seconds.

301107 - Analytics Programming

Graeme Wong & Nick Tothill

Practical Class 7 - Simulation

Week 07 Task: Clara's random walk on a grid

If you did the Programming Fundamentals unit, you worked with a simulation of a bug — Clara — wandering around an artificial world.

We can do something similar in R.

In this task, you are going to simulate Clara's random wandering on a grid with given size, for example, a grid consisting of 10 by 10 cells. Clara lands in one cell at random on this grid and begins to explore her surroundings. Assume the grid has hard outside boundaries (perhaps Clara is surrounded by rocks). Depending on which cell Clara is in at time t, she has a few choices for the next step, i.e. time $t + 1$.

If she is in the middle of the grid, she has four possible choices, North, South, East and West. Each of these directions has the same probability, that is $1/4 = 0.25$.

However, things are different on the edges and corners. There are still four directions to go with the same probability, but some of them would take her past the boundary, and should therefore be rejected. For example, if the Clara is in the North-West corner at time t, then the random process selects North or West, this selection will be rejected as she hits the boundary, bounces back, and stays in the same place at time $t + 1$. (Then the cell can be counted as visited twice.)

We can simulate Clara's path crawling around the grid and calculate the probability that a cell is crawled (visited) overall, that is, to count how many times each cell has been crawled or visited divided by the total number of walks. You can use the techniques used in Conway's Game of Life to visualise the crawling history by plotting the matrix using `image()` function.

This task can be subdivided:

1. Make a function to carry out this random walk simulation. The input arguments of this function are
   - the size of the grid (default 10 by 10)
   - number of 'steps' in the random walk (default 10000)
   - starting cell (default to starting at random)
   - probabilities of each direction (assume only four directions, default as all equal, i.e. 0.25)

   The output is the probabilities of each cell being crawled or visited. The crawling motion is visualised on a grid.
2. Output a GIF file for the random walk with all default settings in the function you programmed, i.e. 10 by 10 grid, starting from a random cell and moving in four directions, each of which has a 0.25 probability to be chosen with rejections on the boundaries.

3*. Change probabilities for each direction in the 'four directions' case to some uneven values and observe what happens. Always start from the same corner (say, South-West).

4**. Add another move 'direction' called stay besides N, S, E, W, with some probability. This denotes staying on the same cell at time $t + 1$. N.B. probabilities of moving in all directions at a given time should add up to 1.

5***. Extend the random walk on a grid to a more general graph consisting of vertices and edges. Vertices are like cells in the grid and edges connecting vertices define where you can move. Think about how to

do random walk on this graph. This random walk on connected graph is the basis of Google's early page ranking system.

Sub-tasks with *, **, *** are not essential — but highly educational!

Task 1.1 Hints

Break down the problem, look at the individual steps (requirements of the task) and create the code for each step, e.g. How do you create a random location for Clara to land? Using `sample.int` you can create a random number for the `x` and `y` values.

```
# Size of the grid in X and Y
gridSize_X <- 10; gridSize_Y <- 10

# Generate a starting point for Clara to walk around
Claras_landing <- c(sample.int(gridSize_X,1),sample.int(gridSize_Y,1))

# Output of the starting position for Clara
cat('Clara\'s starting point is x =',Claras_landing[1],' y =',Claras_landing[2])
```

```
## Clara's starting point is x = 1  y = 8
```

The function `sample` will also help you with next position Clara will move to, but you need code or a function that will handle boundaries!

301107 - Analytics Programming

Nick Tothill

Practical 8 - Using R Markdown

R Markdown

In this practical, you will create R Markdown documents. See the pre-lecture 8 slides on vuws for an introduction (in Learning Materials, week10). See also http://rmarkdown.rstudio.com.

R Markdown documents are not only useful to present the results of your work in R (like a word document), they also have your R code embedded in them, so that the reader can see your code. Your reader can re-run your code (in fact, they must, when they 'knit' the document).

R Markdown documents are therefore a great resource for *transparent*, *reproducible* research.

Make your first document

From the 'File' menu in RStdio, select 'New File'and then 'R Markdown'. You will need to select an output format. HTML is generally safe. The new document will contain some example content.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

embed a code chunk to do the following:

```
print(1:10)
```

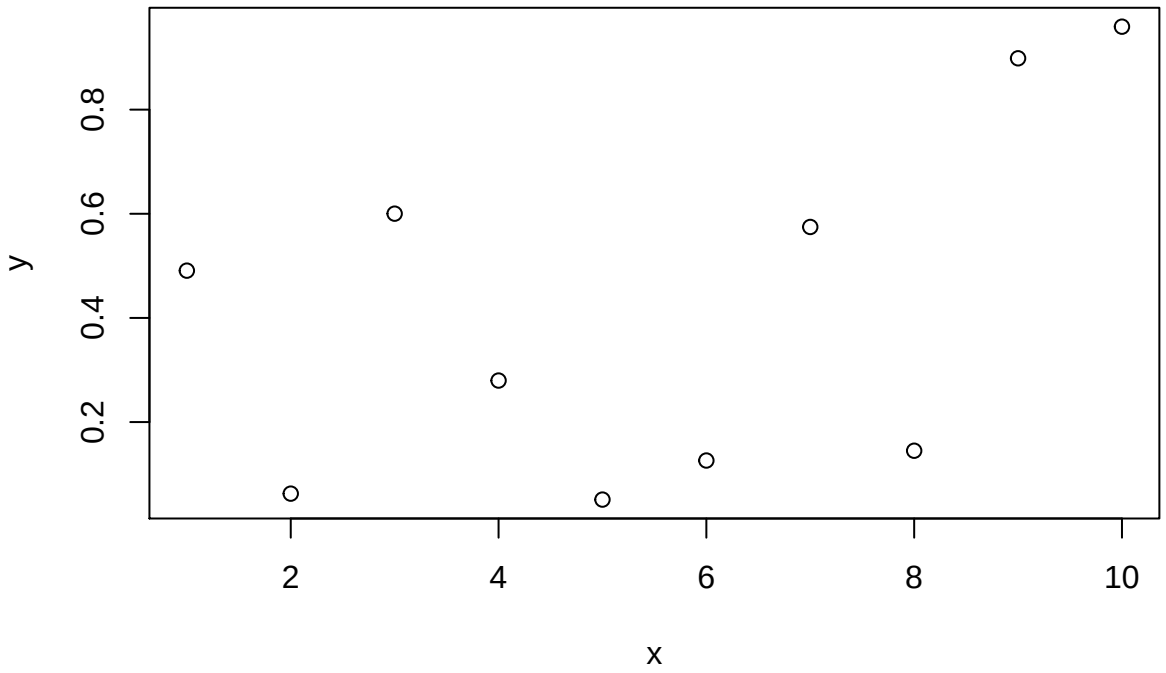
```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

and one to remove the code, leaving only the result:

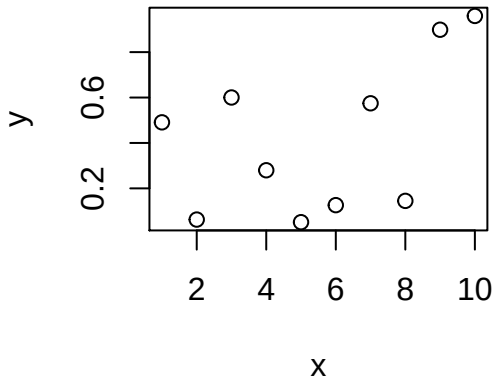
```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Now add a plot

```
x <- 1:10
y <- runif(10)
plot(x,y)
```



Now remove the R code, so you just have the plot, and make it 3 inches high and 3 inches wide:



Behaviour of Dynamic Documents

Knit your document into output and rename the file to a different name. Then knit the document again. Compare the plots in the two output files. Why are they different?

Alter the code to give the same output when knitted again and again.

Why might you want each of the two behaviours?

Build a larger document

The material shown in blue below is based on the lectures. Build a markdown document that produces this output.

Data frames

A data frame is a matrix with columns which may have different modes.

```
kids <- c("Jack","Jill")
ages <- c(12,10)
d <- data.frame(kids,ages,stringsAsFactors=FALSE)
d
```

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

The first column of `d` is character, while the second one is numeric. Data frame is sort of a generalised list.

As a result, data frame can be accessed as a list, or as a matrix. Many vector and matrix applicable functions can be used on data frame as well, e.g. `cbind`, `rbind`, `apply`, `subset`, etc. There are many functions only for data frames such as `merge`, `by`, etc. including many extremely useful statistical analysis functions.

```
d
```

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

```
d$kids # Show kids component
```

```
## [1] "Jack" "Jill"
```

```
d[[1]] # Show the first component
```

```
## [1] "Jack" "Jill"
```

```
d[,1] # Show the first column
```

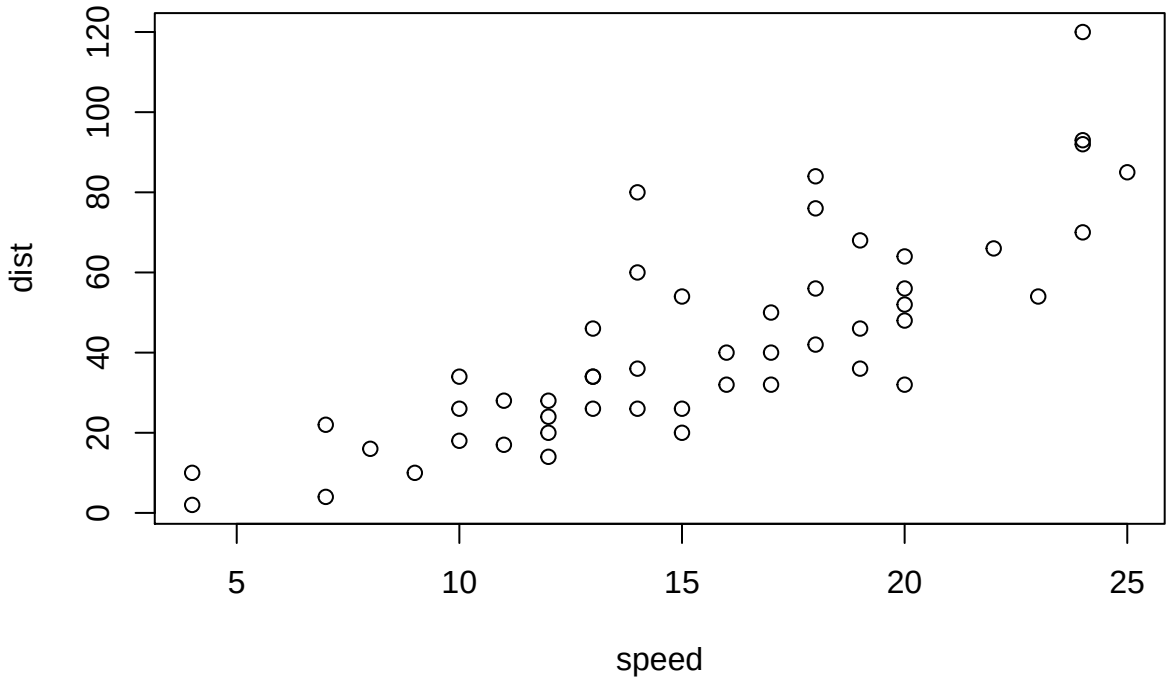
```
## [1] "Jack" "Jill"
```

Data frame is the main container for storing data

```
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

```
plot(cars)
```



Factors

Factors are categorical variables in statistics such as the categories of iris

Data Conversion

- convert data from one type to another (rule of thumb: don't sacrifice precision!)
  - numeric – character
  - ...
- convert data from one structure to another
  - matrix – data frame
  - ...
- Very useful functions:
  - `as.type()`
  - `is.type()`

`type` is meant to be replaced by actual names such as `matrix`,`vector`,`numeric`,...

Operators

Relational Operators	Logical Operators
`x <= y`	`!x`
`x >= y`	`x & y`
`x == y`	`x && y`
`x != y`	`x | y`
`x || y`	
`xor(x, y)`	
`isTRUE(x)`	

Using other output formats

Experiment with other output formats. The defaults are HTML, PDF and Word. Try them all if you can. (the PDF option requires you to have the LaTeX text processing engine available on the machine - the lab machines should be OK, but you might not have it on your own machine.)

In particular, mathematical typesetting is generally best done with the LaTeX/PDF combination. Try adding the equation $y = ax^2 + bx + c$ to your document in various formats.

301107 - Analytics Programming

Nick Tothill

Practical Class 9 - Input/output

Task 1. Read and write iris data

The iris data are available in UCI machine learning data repository. You can go to http://archive.ics.uci.edu/ml/machine-learning-databases/iris/ to see the list of files. http://archive.ics.uci.edu/ml/datasets/Iris contains descriptions about this data set.

Read iris data from the website and reformat it to a data frame just like `iris` data set in R. The file you will read is called `iris.data` created on 08 March 1993. Assume `myiris` is the data frame you read the data into,

```
> head(myiris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width     Species
1          5.1         3.5          1.4         0.2 Iris-setosa
2          4.9         3.0          1.4         0.2 Iris-setosa
3          4.7         3.2          1.3         0.2 Iris-setosa
4          4.6         3.1          1.5         0.2 Iris-setosa
5          5.0         3.6          1.4         0.2 Iris-setosa
6          5.4         3.9          1.7         0.4 Iris-setosa
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "Iris-setosa",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Notice that the `Species` column is different from the `iris` data set in R. You have to modify it so it is the same as in `iris`. The function you can use is `gsub()`.

```
> str(myiris)
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Save `myiris` data to 4 files, i.e. R data object file, plain text file, CSV file and binary file. Compare sizes of these 3 files using `file.info()` or `file.size()`.

```
> file.size('myiris.RData')
```

```
[1] 1102
```

```
> file.size('myiris.txt')
```

```
[1] 4818
```

```
> file.size('myiris.csv')
```

```
[1] 4026
```

```
> file.size('myiris.bin')
```

```
[1] 6000
```

Note that the binary data format requries all values to be the same type, so you need to convert the `Species` *component to make sure it is compatible with the others.* The binary file has the largest size because everything is stored with the highest accuracy.

Read your iris data in from all 4 files and make sure they are all correctly read in, i.e. what you read in R is what you are expecting.

N.B. Reading from binary file needs some special care as the data is written sequentially.

Task 2. Fetching Twitter Tweets

It is not too difficult to access Twitter data (text messages) by using Twitter's REST API, which is a set of rules for getting data from web service. REST stands for REpresentational State Transfer. REST is a web standards based architecture and uses HTTP Protocol for data communication, and API is short for Application Program Interface. The details of REST API can be found here: https://www.tutorialspoint.com/restful/index.htm. However, understanding of REST API is not required in this practice.

To access tweets from Twitter, you need to create your access token for authentication purposes by following the steps below:

1. Create a Twitter account if you do not have one yet.
2. Go to the Twitter Apps website (https://dev.twitter.com/apps) and sign in with your Twitter account.
3. Click "Create New App" button on the next page.
4. Complete the application form and click "Create your Twitter application" on the next page. You can enter whatever you like for most fields. You can provide a temporary homepage such as the University homepage (http://www.westernsydney.edu.au).
5. Switch to the "Keys and Access Tokens" tab on the App main page. You will find your Consumer Key (API Key) and Consumer Secret (API Secret) at the top. You will also need access tokens (Access Key and Access secret) to use Twitter APIs for your application. These can be created by scrolling to the bottom of the page and clicking "Create my access token".
6. Keep these details and they will be used in the follow example.

Next you need to use `twitteR` package. So install it if you don't have it yet. Type in

```
install.packages("twitteR")
```

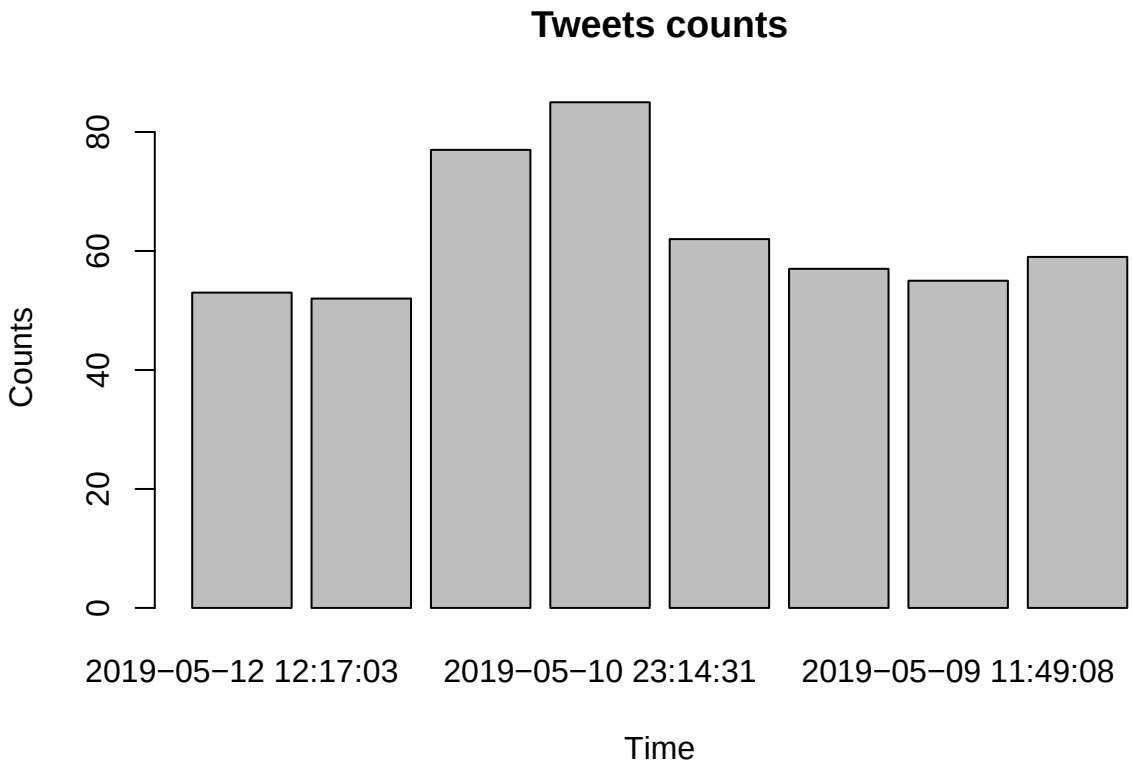
Then use the following code to get some tweets:

```
library(twitteR)
api_key <- your_api_key_as_a_string
api_secret <- your_api_secret_as_a_string
token <- your_token_as_a_string
token_secret <- your_token_secret_as_a_string
setup_twitter_oauth(api_key, api_secret, token, token_secret)
tweets <- searchTwitter("R programming OR machine learning", n = 500, lang = "en")
df <- twListToDF(tweets)
```

In above code, the 2nd to 6th line is to setup the authentication so that you can fetch data. `searchTwitter` is to search tweets on the topics you are interested in. In this case, it is about R programming or machine learning so that tweets about either topics mentioned will be pulled back to your computer in R and saved as a list. `n=500` is asking for 500 tweets and `lang=en` is getting tweets in English. `twListToDF(tweets)` is to convert list `tweets` as data frame, which is nicer to work with.

There is a component in `df` called `created`, which shows when the tweet was created in the format called `POSIXct`. `POSIXct` is a class for dates and times. Use ?POSIXct to see its description. Next bin the tweets into 8 equal time intervals and produce a bar plot like the following

```
[1] "Using direct authentication"
```



Hint: POSIXct can be treated just as numeric. We have done binning (histogram) before, and the plotting function to use is `barplot()`.

Task 3 — Optional. Customised write/read data frame to and from binary files

Do not attempt this task unless you have completed the previous two

Write two functions. The first is to save a data frame to binary file and the second is to read from the binary file and reformat it into a data frame. We assume that there are only 3 type of data in the data frame — numeric, integers and factors. Use the following template for these functions.

```
writedftobin <- function(df,file,strfile=paste(file,'.str',sep=''))
{
  # df is the input data frame you want to save
  # file is the name of the file that the data will be written to
  # strfile: the text file containing data frame structure. The default
  # name of strfile is file.str. So if the binary data file is called
  # mydata, then strfile is mydata.str as default.
  # Your code goes from here
  ...


  # Save df data structure to a text file
  # such as number of observations, names and data types of components
```

```
  # Save df into a binary file

}

readdffrombin <- function(file,strfile=paste(file,'.str',sep=''))
{
  # file: the actual binary file containing all the data
  # strfile: the text file containing data frame structure.
  # Your code goes from here
  ...


  # Return data frame read from the specified data file here assuming it
  # is called df
  df
}
```

We are mimicking R `save()` function without a complicated header structure in the file. The functions you may need to use are `is.data.frame()`, `names()`, `class()`, etc. There are many ways of doing this. We use the simplest one: use a text file to store the structure of the data frame and a binary file to store its data. That is why we have `strfile` as an input argument in the function templates. The structure of a data frame includes the names of its components, data type (numeric, integer and factor in this case) and levels if any component is a factor. It is easier to export this structure into a text file. Of course you are free to do anything you like to implement the same functionality, for example, encode the structure into binary format so you need only a single file. But it will increase the complexity of coding.

301107 - Analytics Programming

Nick Tothill

Practical Class 10 - Graphics

Task 1: A `swirl` session on graphics

Find "Exploratory Data Analysis" and finish lesson 1 to lesson 7. You are free to do more if you wish.

Task 2: Plot histogram by using `barplot()`

Use `barplot()` to plot histogram. Take the first column of `iris` data to do this. You are suppose to plot a figure as in Fig. 1. The right panel is generated by `hist(x)`.

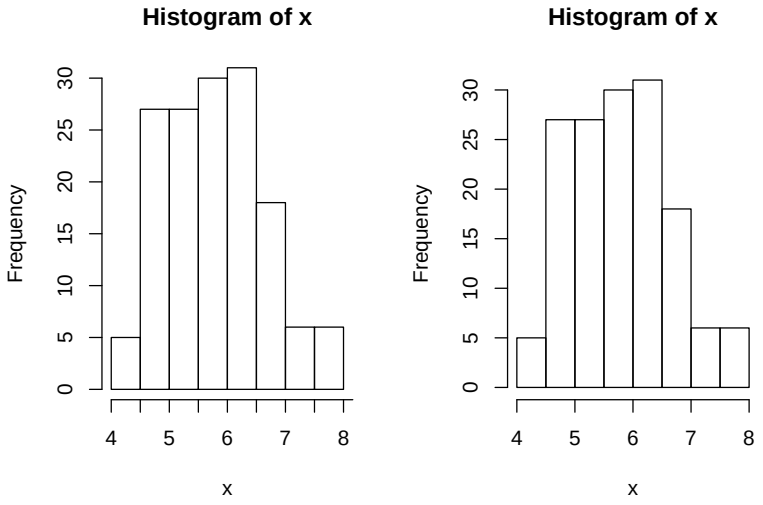


Figure 1: Two verions of histogram plotting. Left: by using barplot(), right: by using hist().

Task 3. Manipulate a hyperspectral image

Background

Hyperspectral images can be seen as a generalisation of normal colour images such as RGB images. In a normal RGB colour image (e.g. a JPEG), there are 3 channels, i.e. channels for the image as seen in red light, green light, and blue light. (There may be a fourth channel for transparency, or "alpha"). Such an image is normally organised in a 3D array, e.g. an array `X` of size $100 \times 120 \times 3$, for a raster image with 100 pixels in its vertical direction and 120 pixels in its horizontal direction. Then `X[ , , 1]`, the first "slice" of the array — which is itself a matrix — is the matrix containing all red light information. This can be encoded in a number of ways — one option is to have the red pixels range from 0 to 1 with 0 meaning totally dark and 1 being the brightest red light. Similarly `X[ , , 2]` is for green light and `X[ , , 3]` for blue light.

A Hyperspectral image (HSI) has similar structure as RGB images. Besides RGB channels, a typical HSI may have hundreds of other channels (sometimes called bands or wavelengths). R can render an RGB image, so in the same way, HSI can also be displayed as an image but using its RGB channels.

A hyperspectral image is also known as a datacube, in that the array structure is a 3-dimensional cuboid. (It should probably be called "datacuboid", but everyone says datacube…)

Another way to look at a hyperspectral image is that each pixel in the HSI is a spectrum. For example, if `X` of size $145 \times 145 \times 200$ is a 3D array storing a HSI, then there are 200 channels in this HSI. Then `X[2,3,]` is a vector of length 200 which is the spectrum at the second row and the third column (imagine an HSI as a cuboid). Due to the rich information contained in the bands, HSI may be used to characterise the objects in the image (e.g. the detailed colour information may be used to the nature of the ground at that pixel). Each pixel may be either a pure material or a mixture of several materials.

What to do

Here we will look at a HSI of a location called Indian Pine — this was remotely sensed (essentially photographed from space) by the AVIRIS sensor `http://aviris.jpl.nasa.gov/`, with 224 channels. The data can be accessed from `http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes` where some more information about the data set can be found. Each pixel has been classified manually as one of the 16 agriculture classes in the scene plus a background class. The whole data set and ground truth are packed in a matlab file called `indianpine.mat` which is in `vuws`. This is the file you need to work on. You can use `readMat()` in the R package called `R.matlab` to read in the matlab file

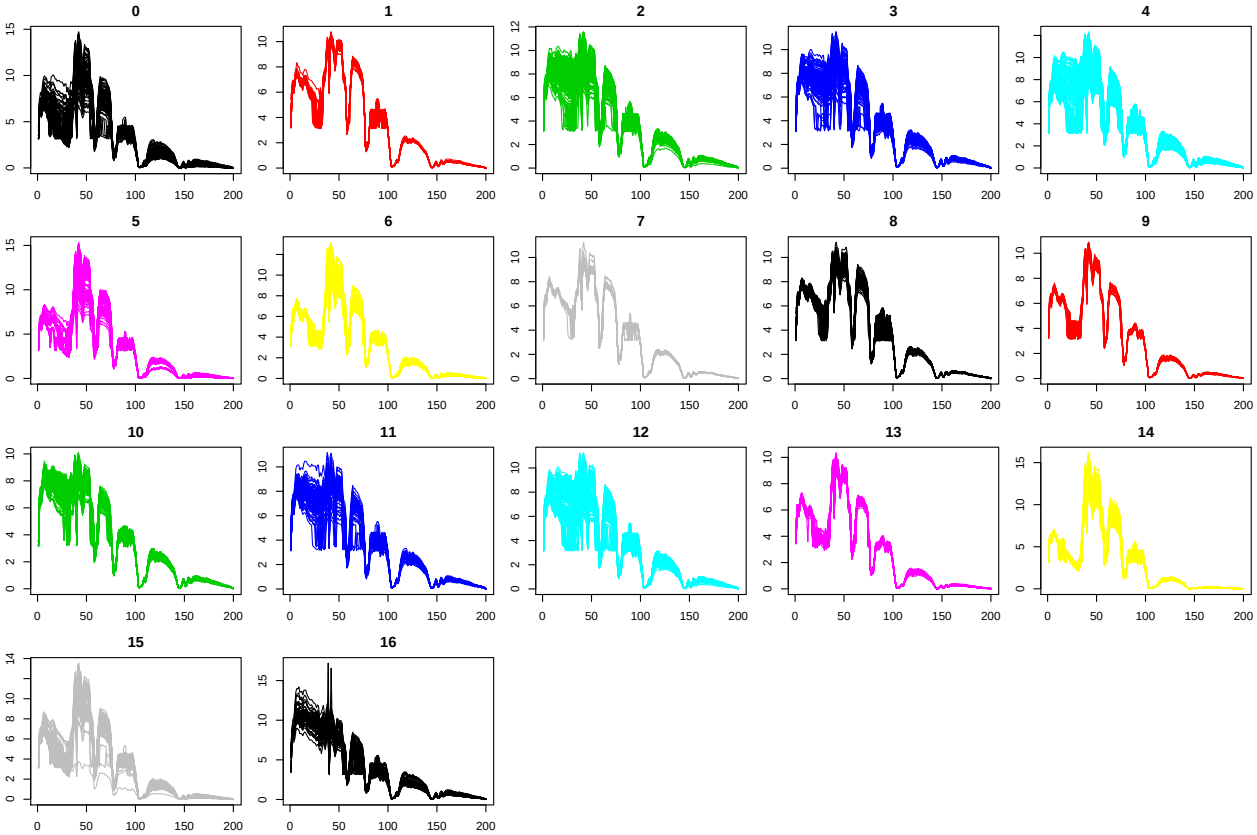
```
> library(R.matlab)
> A <- readMat('indianpine.mat')
> names(A)
```

```
[1] "X"           "groundtruth" "orgX"
```

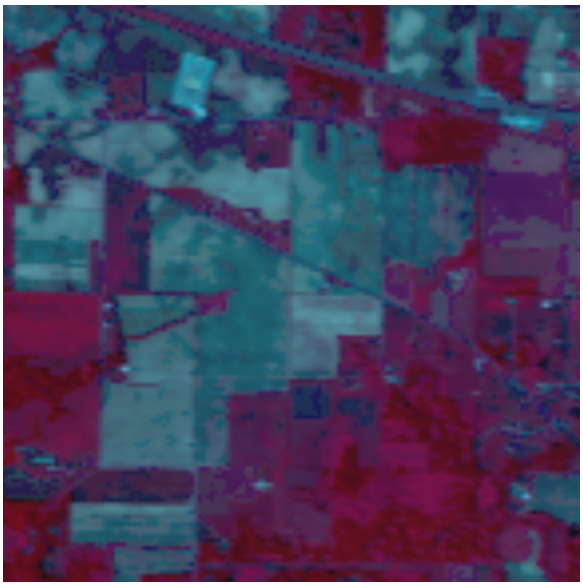
`groundtruth` is the classification information of each pixel and `X` and `orgX` are the actual Indian pine HSI. You can work on either `X` or `orgX`. `X` is a version of `orgX` after removing some noisy bands.

Write R scripts for the following tasks.

1. Produce a plot of 100 spectra randomly chosen from each class and plot classes in separate subfigures as in the following. If there are less than 100 spectra in one class, take all the spectra without sampling.



2. Display the Indian pine data set as a colour image using channels 57, 27 and 17 as the RGB channels. The most useful function to use is `grid.raster()` in the package called `grid`. Note that the values in the image has to be normalised to the range from 0 to 1, i.e. make sure that the highest R, G, and B values are scaled to 1, the lowest ones are scaled to zero, and everything in between is scaled sensibly.



301107 Analytics Programming — Practical 11

Nick Tothill

Accessing data with SQL

Task 1. Using SQL queries on R Data — `sqldf`

Install `sqldf` package with `install.packages("sqldf")`. This may take a while on the lab computers (maybe 10 minutes!), especially if you are not at your home campus. Be patient. It shouldn't take nearly as long on your own machine.

You can then load it with `library(sqldf)`.

You can run an SQL query directly on an R data frame using the `sqldf()` function. For example, using the `iris` built-in data set, you can select some of the data and dump the results into another dataframe:

```
> tmpsql <- sqldf('select * from iris where Species == "setosa"')
```

The R code to do the same thing is

```
> tmpR <- iris[iris[,5]=='setosa',]
```

You can show that the two statements are equivalent by comparing the outputs:

```
> all.equal(tmpsql,tmpR)
```

```
[1] TRUE
```

Get the `iris` data into a holding data structure (`a`), and add column names as follows:

```
library(sqldf);a <- iris;
names(a) <- c("SepalLength","SepalWidth","PetalLength","PetalWidth","Species")
```

Try:

```
sqldf('select * from iris')
sqldf('select count (*) from iris')
```

what is the difference between `select *` and `select count (*)`?

Now use SQL commands to answer the following:

a. How many irises in the sample belong to either of the *virginica* or *versicolor* species?
b. How many irises of species *setosa* have sepal length greater than 5.0?
c. Select out all the Sepal lengths of species *setosa*; now plot these as a histogram. (*Hint:* the selected data will come back as a data frame — you'll have to break the data out of this to easily plot it.)
d. How many irises in the sample have a NULL species, i.e. are unclassified?
e. add a sample with a NULL species
f. replace all of the measured lengths of samples with NULL species with lengths of -1.0
g. delete all samples with NULL species

Translating SQL commands into R

Run these SQL queries on the `iris` data set using `sqldf`, see how they work, and work out how to produce the same result using R.

```
sqldf('select PetalWidth , SepalWidth from a')
sqldf('select * from iris')
sqldf('select count (*) from iris')
sqldf('select * from iris where Species == "setosa"')
sqldf('select count (*) from a where Species == "setosa" and SepalLength>5')
sqldf('select count (*) from iris where Species like "%a"')
sqldf('select count (*) from a where SepalLength between 6.5 and 7 AND Species== "versicolor"')
sqldf('select count (*) from a where SepalLength in (6.5,6.6) OR SepalWidth in (2.3, 2.4)')
sqldf('select count (DISTINCT SepalLength) from a where Species like "%a"')
sqldf('select * from a where Species== "versicolor" ORDER BY PetalLength ASC, PetalWidth DESC')
```

Assume the above SQL queries are executed sequentially.

Task 2. Connecting to real MySQL database using `RMySQL`

We have a MySQL database for you to practice on, hosted in one of the SCEM servers:

database host: 137.154.179.253
database: rstudio
Username: rstudio
Password: 1.2.RStudio!

You can install `RMySQL` and `DBI`packages in R and use the code below to connect to the above database.

(This connection has been tested in the labs, and should work. Another option to try in the event of problems is to go to https://r.scem.uws.edu.au, which is a web based RStudio running on the SCEM server. You need to login there using your SCEM account. If you don't have one, go to PHPMyAdmin: https://students.scem.uws.edu.au/phpmyadmin/ to create one.)

The following code can be used to connect to the above database, create a table and send queries. Once you get the hang of it, use the `iris` data to create a table in the database and perform the queries in Task 1 on this "real" database. (Many student may be accessing at a time — so it may be wise to label your `iris` table with your student number or similar.)

```
con <- dbConnect(RMySQL::MySQL(),dbname="rstudio",host="137.154.179.253",
                 username='rstudio',password='1.2.RStudio!')

# List all tables in this database
dbListTables(con)

# Create a table named iris using iris data set.
dbWriteTable(con,'iris',iris,row.names=F,overwrite=T)
dbListFields(con, 'iris')

# Send SQL query
res <- dbSendQuery(con, "SELECT * FROM iris")
data <- dbFetch(res, n = -1) # Get all data from the query

dbWriteTable(con, "arrests", datasets::USArrests, overwrite = TRUE)

# Another way of sending query
dbGetQuery(con, "SELECT * FROM arrests limit 3")

# Remove the table named arrests
dbRemoveTable(con, "arrests")
```

```r
# Terminate the connection
dbDisconnect(con)
```

There are many other functions available in RMySQL. See its help pages for details.