Data Mining with R

Introduction to R and RStudio

Hugh Murrell

Reference Books

These slides were created to accompany chapter two of the text:

- `SPUR`: *Scientific Programming and Simulation using R* Owen Jones, Robert Maillardet, and Andrew Robinson.

A version of this text can be found on the web.

Scientific Programming and Data Mining

- In this course we aim to teach scientific programming and to introduce data mining.
- Scientific programming enables the application of mathematical models to real-world problems.
- Data mining is the computational technique that enables us to find patterns and learn classifiaction rules hidden in data sets.

Scientific programming with R

- We chose the programming language R because of its programming features.
- R is also rich in Statistical functions which are indespensible for data mining.
- We do not only use R as a package, we will also show how to turn algorithms into code.
- Our intended audience is those who want to make tools, not just use them.

The R Programming Language

- This course uses the statistical computing system, R.
- R is based on the computer language S, developed by John Chambers and others at Bell Laboratories in 1976.
- Robert Gentleman and Ross Ihaka developed an implementation, and named it R and made it open source in 1995.
- Hundreds of people around the world have contributed to its development.
- The R Core Team is now responsible for development and maintenance of R.

The R Package

- R is available at `http://cran.r-project.org`.
- This site is referred to as **CRAN**
- Most users download and install a binary version. This is a version that has been translated (by compilers) into machine language for execution on a given operating system.
- R is designed to be very portable: it will run on Microsoft Windows, Linux, Solaris, Mac OSX, and other operating systems, but different binary versions are required for each.

RStudio IDE

- RStudio is a free and open source IDE (integrated development environment) for R.
- You can run it on your desktop (Windows, Mac, or Linux)
- You can run it over the web using RStudio Server.
- RStudio is available at `http://www.rstudio.com`

RStudio Screen Shot

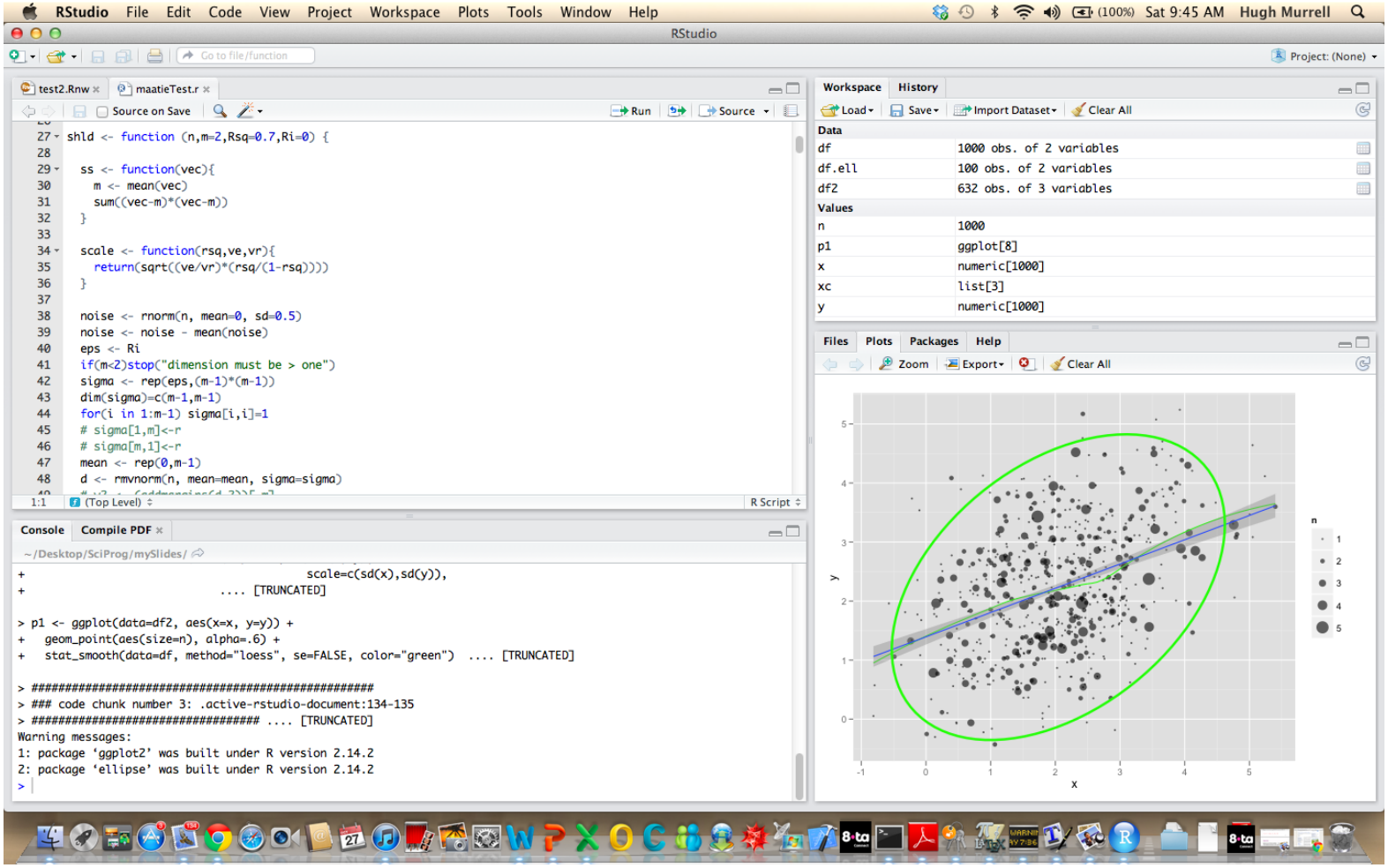

Figure: RStudio Frames.

RStudio Pannels

top left

- edit and execute code segments
- code folding
- projects and packages

bottom left

- R Console
- command history

top right

- data sets
- workspace
- debugging

bottom right

- package tracking
- help browsing
- plotting

Using R as a Calulator

After starting RStudio you can interact with the R consol (bottom left pane) and use R in calculator mode.

The $>$ sign tells you that R is ready for you to type in a command.

For example, you can do addition:

```
> 123+456
```

```
[1] 579
```

In these slides we adopt the convention that your input is shown in midnightblue and R's output is shown in darkred. The `[1]` that prefixes the output indicates that this is item `1` in a vector of output.

R Scripts

In order to save your R work it is recomended that you load your R script into the edit panel of RStudio (top left)

- You can make changes to your script and save your work
- you can execute code segments of your script.
- The results will appear in the R consol pannel or the plotting pannel.

Arithmetic Operators

R uses the usual symbols for addition, subtraction, multiplication, division, and exponentiation.

Parentheses can be used to specify the order of operations.

```
> 2 * (1 + 1/100)^100
```

```
[1] 5.409628
```

R also provides operators for modulus and integer division.

```
> 17%%5
```

```
[1] 2
```

```
> 17%/%5
```

```
[1] 3
```

Mathematical Functions

R has a number of built-in functions, for example sin(x), cos(x), tan(x), (with argument in radians), exp(x), log(x), and sqrt(x). The functions floor(x) and ceiling(x) round down and up respectively, to the nearest integer. Some special constants such as pi are also predefined.

```
> exp(1)
[1] 2.718282
> floor(exp(1))
[1] 2
> sin(pi/6)
[1] 0.5
```

R calculates to a high precision, but by default only displays 7 significant digits. You can change the display to x digits using `options(digits = x)`.

Variables

To assign a value to a variable we use the assignment command `<-`.

Variables are created the first time you assign a value to them.

Variable names are case sensitive.

To display the value of variable `x` on the screen we just type `x`.

This is shorthand for `print(x)` which we will have to use when writing scripts or printing results inside a loop.

```
> x <- log(exp(1))
> x
[1] 1
```

Variables continued...

When assigning a value to a variable, the expression on the right-hand side is evaluated first, then that value is placed in the variable on the left-hand side. It is thus possible (and quite common) to have the same variable appearing on the right- and left-hand sides.

```
> n <- 1
> n <- n + 1
> n
```

```
[1] 2
```

In common with most programming languages, R allows the use of = for variable assignment, as well as <-. The latter is preferred, because then there is no possibility of confusion with mathematical equality.

Functions

Functions in R take one or more arguments and produce one or more return values. To call or invoke a function in R you write the name of the function followed by its argument values enclosed in parentheses and separated by commas. We illustrate with a function that produces arithmetic sequences:

```
> seq(from = 1, to = 9, by = 2)
```

```
[1] 1 3 5 7 9
```

To find out about default values and alternative usages of the built-in functions, you can access the built-in help by typing `help(fname)` or `?fname`.

Functions continued...

Every function has a default order for the arguments. If you provide arguments in this order, then they do not need to be named, but you can choose to give the arguments out of order provided you name them:

```
> seq(1, 9, 2)
[1] 1 3 5 7 9
> seq(to = 9, from = 1)
[1] 1 2 3 4 5 6 7 8 9
> seq(by = -2, 9, 1)
[1] 9 7 5 3 1
```

Each argument value can be an expression:

```
> x <- 9
> seq(1, x, x/3)
[1] 1 4 7
```

Vectors

A vector is an indexed list of variables. A simple variable is just a vector with length 1 (also called atomic). To create vectors of length greater than 1 we use functions that produce vector valued output. The three basic functions for constructing vectors are:

- `c(...)  # combine`
- `seq(from,to,by) # sequence`
- `rep(x, times) # repeat`

Lets try them out:

```
> x <- seq(1, 20, by = 2)
> y <- rep(3, 4)
> (z <- c(y, x))
```

```
 [1]  3  3  3  3  1  3  5  7  9 11 13 15 17 19
```

Vectors continued...

To refer to element i of vector x, we use x[i]. If i is a vector of positive integers, then x[i] is the corresponding subvector of x.

```
> (x <- 100:110)
```

```
 [1] 100 101 102 103 104 105 106 107 108 109 110
```

```
> i <- c(1, 3, 2)
> x[i]
```

```
[1] 100 102 101
```

If the elements of i are negative, then the corresponding values are omitted.

```
> j <- c(-1, -2, -3)
> x[j]
```

```
[1] 103 104 105 106 107 108 109 110
```

Vectors continued...

Algebraic operations on vectors act on each element separately, that is elementwise.

```
> x <- c(1, 2, 3); y <- c(4, 5, 6)
> x*y
```

```
[1]  4 10 18
```

When you apply an algebraic expression to two vectors of unequal length, R automatically repeats the shorter vector until it has something the same length as the longer vector.

```
> c(1, 2, 3, 4) + c(1, 2)
```

```
[1] 2 4 4 6
```

This happens even when the shorter vector is of length 1:

```
> 2 + c(1, 2, 3)
```

```
[1] 3 4 5
```

Vectors continued...

A useful set of functions taking vector arguments are:

- sum(...), prod(...), max(...), min(...),
- sqrt(...), sort(x), mean(x), var(x).

Note that functions applied to a vector may be defined to act elementwise or may act on the whole vector input to return a result:

```
> sqrt(1:5)  # 1:5 is short for seq(from=1,to=5,by=1)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
> mean(1:5)
[1] 3
```

Plotting vectors

To plot one vector against another, we use the function

- plot(x, y, type).

When using plot, x and y must be vectors of the same length. The optional argument type is a graphical parameter used to control the appearance of the plot:

"p" for points (the default);

"l" for lines;

"o" for points over lines;

Plotting vectors, an example...

As an example lets use R to compute the limit of a well known function from first year:

$$\lim_{x \to \infty} (1 + 1/x)^x = e$$

First we generate a sequence of x values and then we evaluate the function at each point in the sequence

```
> x <- seq(10, 200, by = 10)
> y <- (1 + 1/x)^x
```

On the next slide we use plot to reveal the trend....

graphics output

```
> plot(x,y,type="o")
```

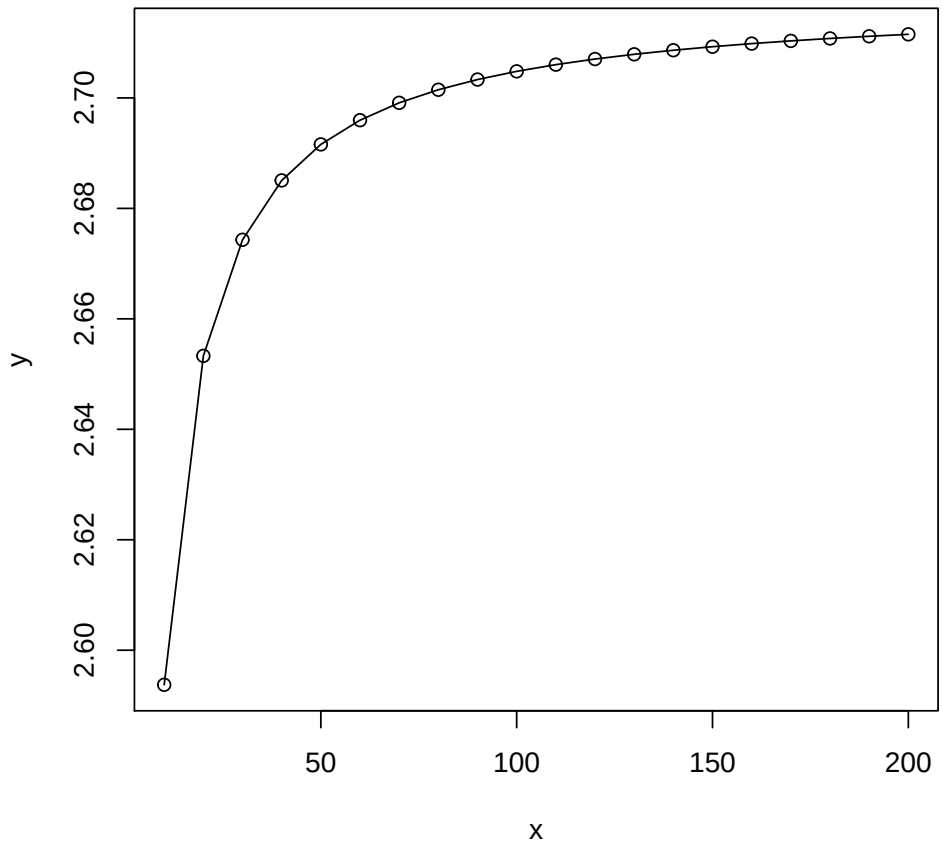

Figure: $\lim_{x\to\infty}(1+1/x)^x = e$.

Missing data

In real experiments it is often the case that observations may be missing. Missing data can be propagated or ignored. R represents missing observations through the data value `NA`. They can be mixed in with all other kinds of data. We can detect missing values using `is.na`.

```
> a <- c(11,NA,13)          # assign some values
> is.na(a)                  # are some missing missing?

[1] FALSE  TRUE FALSE

> mean(a)                   # NAs can propagate

[1] NA

> mean(a, na.rm = TRUE) # NAs can be removed

[1] 12
```

Expressions

In R, the term expression is used to denote a phrase of code that can be executed. If the evaluation of an expression is saved, using the <- operator, then the combination is called an assignment. The following are examples of expressions and assignments:

```
> seq(10, 20, by = 3)                # generate a sequence

[1] 10 13 16 19

> x1 <- mean(c(1, 2, 3))             # compute and save the
> x1

[1] 2
```

Logical expressions

A logical expression is formed using the comparison operators
`<, >, <=, >=, ==, and !=` ;
and the logical operators
`& (and), | (or), and !  (not)` .
The value of a logical expression is either TRUE or FALSE.
The integers 1 and 0 can also be used to represent TRUE and FALSE, respectively.
Note if you want exclusive disjunction then use xor(A,B):

```
> c(0, 0, 1, 1) | c(0, 1, 0, 1)     # logical or
[1] FALSE  TRUE  TRUE  TRUE
> xor(c(0, 0, 1, 1), c(0, 1, 0, 1)) # exclusive or
[1] FALSE  TRUE  TRUE FALSE
```

Subvectors

One way of extracting a subvector is to provide an subset as a vector of TRUE or FALSE values, the same length as x. The result of the x[subset] command is that subvector of x for which the corresponding elements of subset are TRUE.
For example, suppose we wished to find all those integers between 1 and 20 that are divisible by 4.

```
> x <- 1:20
> x%%4 == 0
```

```
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FA
[13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
> (y <- x[x%%4 == 0])
```

```
[1]  4  8 12 16 20
```

Matrices

A matrix is created from a vector using the function matrix, which has the form:

```
matrix(data, nrow = 1, ncol = 1, byrow = FALSE)
```

Here data is a vector of length at most nrow*ncol, nrow and ncol are the number of rows and columns respectively (with default values of 1), and byrow can be either TRUE or FALSE (defaults to FALSE) and indicates whether you would like to fill the matrix up row-by-row or column-by-column.

If length(data) is less than nrow*ncol (for example, the length is 1), then data is reused as many times as is needed. This provides a compact way of making a matrix of zeros or ones.

Matrices continued...

We refer to the elements of a matrix using two indices.

```
> A <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
> A[1, 3] <- 0
> A
     [,1] [,2] [,3]
[1,]    1    2    0
[2,]    4    5    6
> A[, 2:3]
     [,1] [,2]
[1,]    2    0
[2,]    5    6
```

Matrices continued...

To create a diagonal matrix we use diag(x).
To join matrices with rows of the same length (stacking vertically) use rbind(...).
To join matrices with columns of the same length (stacking horizontally) use cbind(...).

The usual algebraic operations act elementwise on matrices.
To perform matrix multiplication we use the operator %*%.

We also have a number of functions for using with matrices,
nrow(x), ncol(x),
det(x) (the determinant),
t(x) (the transpose),
solve(A, B), which returns x such that A %*% x == B.
If A is invertible then solve(A) returns the matrix inverse of A.

Matrices examples...

```
> (A <- matrix(c(3, 5, 2, 3), nrow = 2, ncol = 2))
```

```
     [,1] [,2]
[1,]    3    2
[2,]    5    3
```

```
> (B <- matrix(c(1, 1, 0, 1), nrow = 2, ncol = 2))
```

```
     [,1] [,2]
[1,]    1    0
[2,]    1    1
```

```
> A %*% B
```

```
     [,1] [,2]
[1,]    5    2
[2,]    8    3
```

Matrices examples...

```
> A*B
```

```
     [,1] [,2]
[1,]    3    0
[2,]    5    3
```

```
> (A.inv <- solve(A))
```

```
     [,1] [,2]
[1,]   -3    2
[2,]    5   -3
```

```
> A %*% A.inv
```

```
     [,1]          [,2]
[1,]    1 -8.881784e-16
[2,]    0  1.000000e+00
```

what's going on here?

```
> A
     [,1] [,2]
[1,]    3    2
[2,]    5    3
> A %*% A^(-1)
          [,1]     [,2]
[1,] 1.400000 2.166667
[2,] 2.266667 3.500000
```

Coercion

If you wish to find out if an object is a matrix or vector, then you use `is.matrix(x)` and `is.vector(x)`.

Mathematically speaking, a vector is equivalent to a matrix with one row or column, but they are treated as different types of object in R.

To create a matrix A with one column from a vector x, we use `A <- as.matrix(x)`.

To create a vector from the columns of a matrix A we use `x <- as.vector(A)`;

This process of changing the object type is called **coercion**. In many instances R will implicitly coerce the type of an object in order to apply the operations or functions you ask it to.

Arrays

Occasionally it is convenient to arrange objects in arrays of more than two dimensions. In R this is done with the command:

```
array(data, dim)
```

where `data` is a vector containing the elements of the array and `dim` is a vector whose length is the number of dimensions and whose elements give the size of the array along each dimensional axis.

To fill the array you need `length(data)` equal to `prod(dim)`; see the online help for details of how the elements of data are indexed within the array.

Generate some data

```
> n <- 1000 # number of samples
> x <- rnorm(n, mean=2)
> y <- 1.5 + 0.4*x + rnorm(n)
> df <- data.frame(x=x, y=y)
> mean(x)
[1] 1.965138
> mean(y)
[1] 2.270543
>
```

Produce a Plot

```
> # take a bootstrap sample
> df <- df[sample(nrow(df), nrow(df), rep=TRUE),]
> xc <- with(df, xyTable(x, y))
> df2 <- cbind.data.frame(x=xc$x, y=xc$y,
+                         n=xc$number)
> df.ell <- as.data.frame(with(df,
+                 ellipse(cor(x, y),
+                   scale=c(sd(x),sd(y)),
+                   centre=c(mean(x),mean(y)))))
> p2 <- ggplot(data=df2, aes(x=x, y=y)) +
+   geom_point(aes(size=n), alpha=.6) +
+   stat_smooth(data=df, method="loess",
+               se=FALSE, color="green") +
+   stat_smooth(data=df, method="lm") +
+   geom_path(data=df.ell, colour="green",
+             size=1.2)
```

graphics output

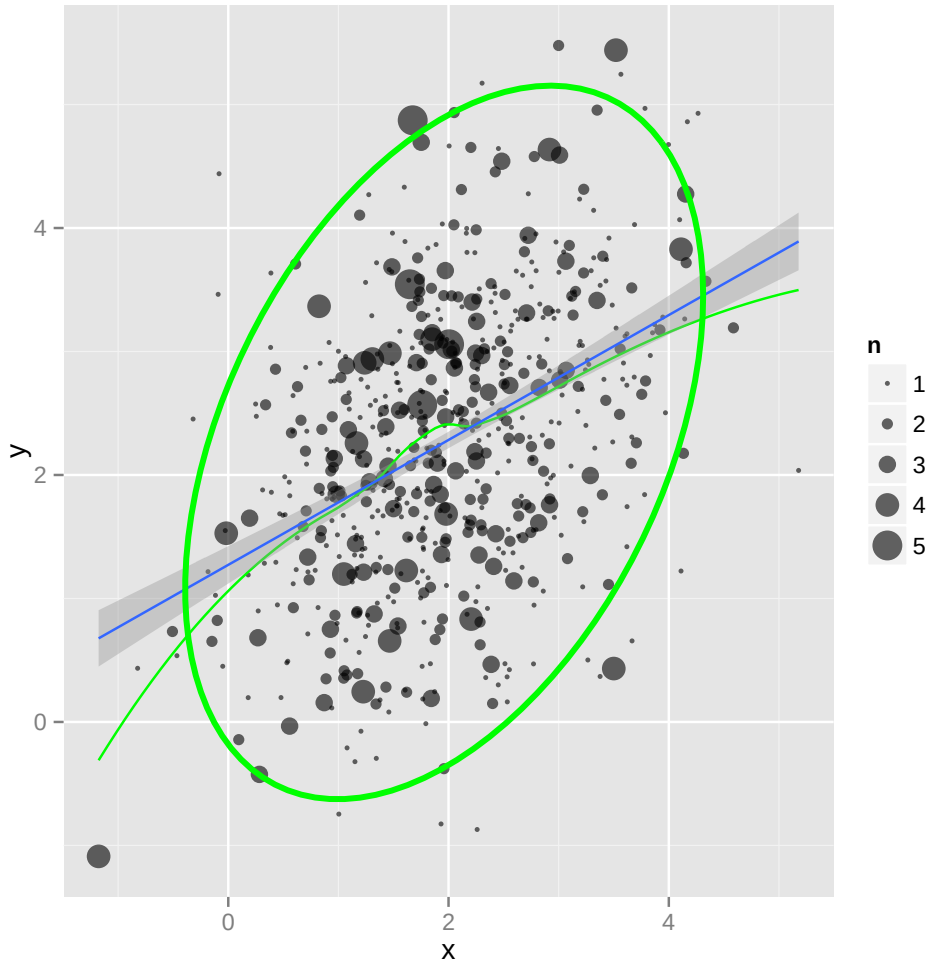

Figure: Here goes the caption.

exercise

Generate data of the form:

$$y = 1 + 2x + \textit{errors}$$

and then try to estimate the intercept and slope from the data and see how well you do.

Your *errors* should be made to fan out as you go along the line. Such *heteroscedasticity* is a common feature of many real data sets.

exercise (submission instructions)

e-mail your solutions to me as a single R script by:

6:00 am, Monday the 22nd February.

Make sure that the first line of your R script is a comment statement containing your name, student number and the week number for the exercise. (in this case week 01).

Use: `dm01-STUDENTNUMBER.R` as the filename for the script.

Use: `dm01-STUDENTNUMBER` as the subject line of your email.

There will be no extensions. No submission implies no mark.