Course Introduction

Mark Dunning

27/07/2015

Welcome!

About us

- Admin at CRUK
  - Victoria, Helen
- Genetics
  - Paul, Gabry, Cathy

Thanks to Cancer Research Uk!

Admin

- Lunch, tea / coffee breaks provided

- Workshop dinner on Thursday @ Downing College
    - no other evening meals
- wifi passwords available
- Course Materials
    - http://bioinformatics-core-shared-training.github.io/cruk-bioinf-sschool/ (http://bioinformatics-core-shared-training.github.io/cruk-bioinf-sschool/)
    - and on the computers at your desk
- Anything else, just ask!

About the Course

- We will tell about '*best practice*' tools that we use in daily work as Bioinformaticians
- You will (probably) not come away being an expert
- We cannot teach you everything about NGS data
    - plus, it is a fast-moving field
- RNA and ChIP only
    - much of the initial processing is the same for other assays
- However, we hope that you will
    - Understand how your data are processed
    - Be able to explore your data - no programming required
    - Increase confidence with R and Bioconductor
    - Be able to explore new technologies, methods, tools as they come out

Further disclaimer

You'll probably see this quote in every stats course you go to

> To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination. He can perhaps say what the experiment died of.”. R.A. Fisher, 1938

If you haven’t designed your experiment properly, then all the Bioinformatics we teach you won’t help: Make friends with your local statistician

Course Outline

Day 1

- Recap of R
- Introduce the Bioconductor project
- Hands-on experience with NGS data
    - IGV
    - FastQC
    - Alignment

Day 2

- Data structures for NGS analysis in R
- Statistical theory behind RNA-seq analysis
- RNA-seq intro
- Aligning and counting for RNA-seq

Day 3

- Differential expression analysis for RNA-seq
- Annotating RNA-seq results
- Using Genome Browsers

Day 4

- Downstream analysis of RNA-seq
- Intro to ChIP-seq
- QA and analysis of ChIP data
- ***Gala Dinner***

Day 5

- Downstream analysis of ChIP-seq
- Reproducible Research
- Invited Talks
    - Jonathan Cairns
    - Peter Van Loo
- End of course :(

Crash-course in R

Support for R

- Online forums
- Local user groups
- Documentation via `?` or `help.start()`

- `browseVignettes()` to see package user guides (‘*vignettes*’)
- ***Get into the habit of using these***
    - or ***google***

RStudio

- Rstudio is a free environment for R
- Convenient menus to access scripts, display plots
- Still need to use *command-line* to get things done
- Developed by some of the leading R programmers

Typical tasks in an R analysis

- Read some data from a `.csv` or `.txt` file
    - R creates some representation of the data
- Explore the data
    - Subset, manipulate to pull out interesting observations
    - Plotting
    - Statistical testing
- Output the results

Variables and functions

- We can save the result of a computation as a *variable* using the *assignment* operator `<-`
- Calculations are done using *functions*

```
x <- sqrt(25)
x + 5
```

```
## [1] 10
```

```
y <- x +5
y
```

```
## [1] 10
```

Vectors

- A vector is often used to combine multiple values. The resulting object is indexed and particular values can be queried using the `[]` operator

```
vec <- c(1,2,3,6)
vec[1]
```

```
## [1] 1
```

- The values can be numeric or text
  - but they must be all the same type

```
vec <- c("A"," B","C","D")
vec[1]
```

```
## [1] "A"
```

Vectors

- Calculations can be performed on vectors

```
vec <- c(1,2,3,6)
vec+2
```

```
## [1] 3 4 5 8
```

```
vec*2
```

```
## [1]  2  4  6 12
```

```
mean(vec)
```

```
## [1] 3
```

```
sum(vec)
```

```
## [1] 12
```

Data frames

- These can be used to represent familiar tabular (row and column) data
- Each column is a vector

```
df <- data.frame(A = c(1,2,3,6), B = c(7,8,10,12))
df
```

```
##   A  B
## 1 1  7
## 2 2  8
## 3 3 10
## 4 6 12
```

- Note that each row is named according to its index
  - we can change this if we wish

Data frames

Don't need the same data *type* in each column

```
df <- data.frame(A = c(1,2,3,6),
                 B = month.name[c(7,8,10,12)])
df
```

```
##   A        B
## 1 1     July
## 2 2   August
## 3 3  October
## 4 6 December
```

Getting data into R

- Various functions can help to read tabular data into R as a data frame
  - `read.csv`, `read.delim`
  - also `read.xls` from `gdata` for Excel data
- Need to know the file path, or read from your ***working directory***
- Usually need to know something about the format of the data
  - comma / tab separated?
  - any header lines?
  - any lines to skip?
- ***Always*** check the data frame before proceeding
  - R may not throw an error, but the format might not be as you expect
  - `head` prints the first few lines
  - `dim` will print the dimensions

Data frames

Once we have imported our data into R, we can start to explore it

- We can subset data frames using the `[]`, but need to specify row and column indices

```
df[1,2]
```

```
## [1] July
## Levels: August December July October
```

```
df[2,1]
```

```
## [1] 2
```

Data frames

Or leave the row or column index blank to get all rows and columns respectively

```
df[1,]
```

```
##   A    B
## 1 1 July
```

```
df[,2]
```

```
## [1] July     August   October  December
## Levels: August December July October
```

Data frames

Can also subset using the column name - the result is a ***vector***

```
df$A
```

```
## [1] 1 2 3 6
```

```
df$B
```

```
## [1] July     August   October  December
## Levels: August December July October
```

Subsetting using vectors

- A vector of indices can be used to subset
- Various shortcuts to define this vector
  - `c`
  - `:` makes a sequence with start and end value
  - `seq` function can also be used
    - `?seq`

```
df[c(1,2,3),]
```

```
##   A       B
## 1 1    July
## 2 2  August
## 3 3 October
```

```
df[1:3,]
```

```
##   A       B
## 1 1    July
## 2 2  August
## 3 3 October
```

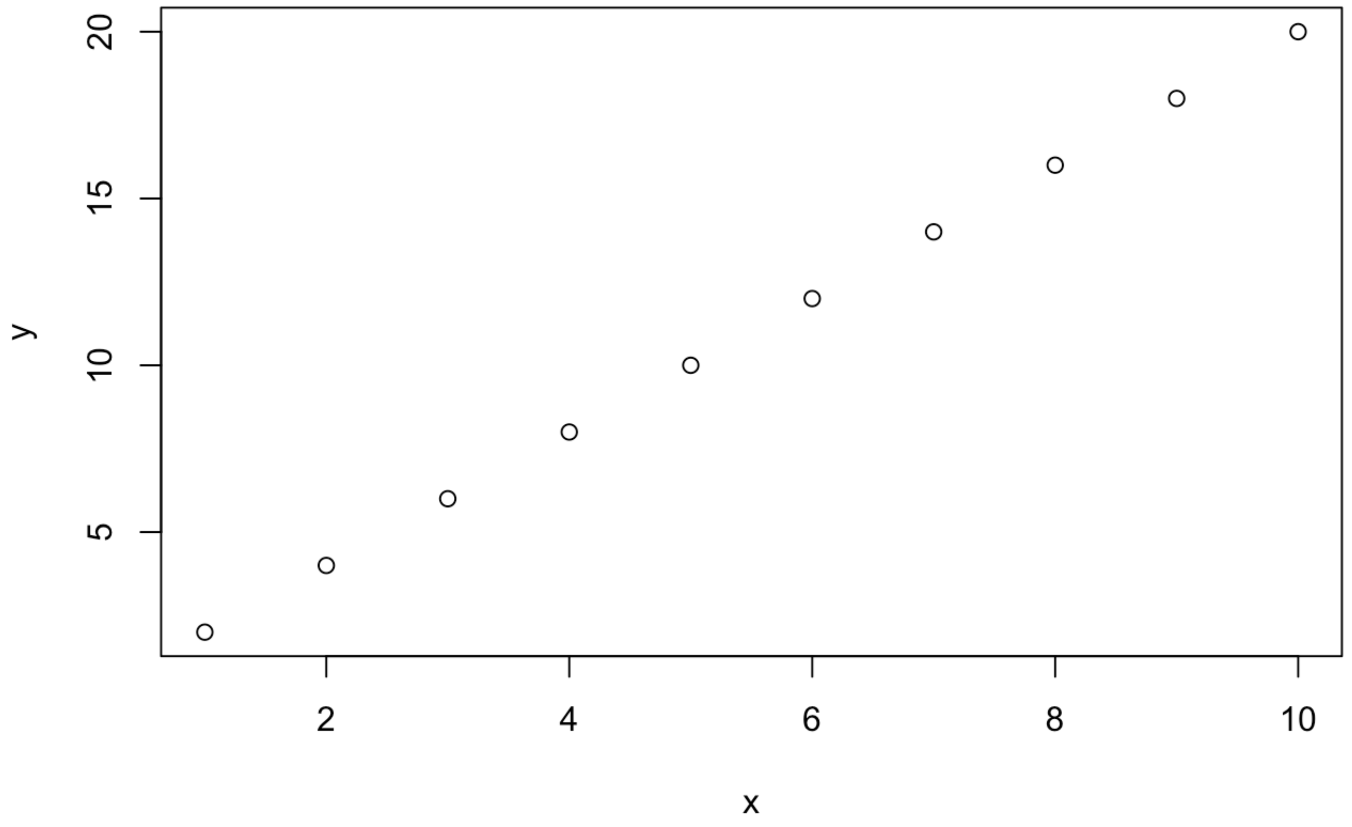
- In both cases, the result is a data frame

Plotting

- R is able to produce all types of graph that we are familiar with
- Example and capabilities can be seen using the `example` function (runs example code that is defined for the function)
    - scatter plot
        - `example(plot)`
    - bar plot
        - `example(barplot)`
    - boxplot
        - `example(boxplot)`
    - histogram
        - `example(hist)`
- A good overview on Quick-R (http://www.statmethods.net/graphs/index.html)
- Plots can be customised
    - colours, labels, adding additional points lines etc
    - layouts
- Plots can be exported in variety of formats

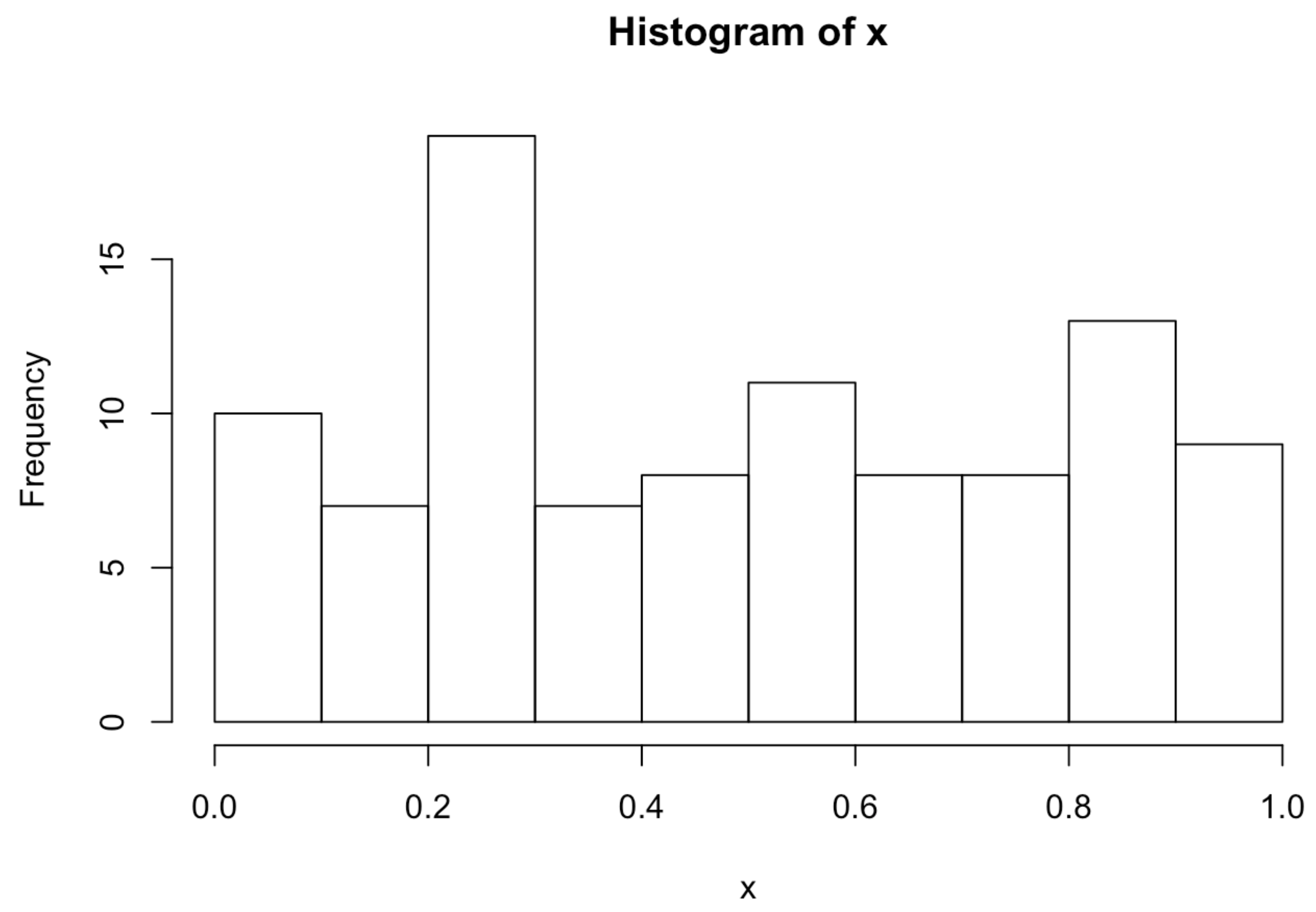
Simple plotting

```
x <- 1:10
y <- 2*x
plot(x,y)
```



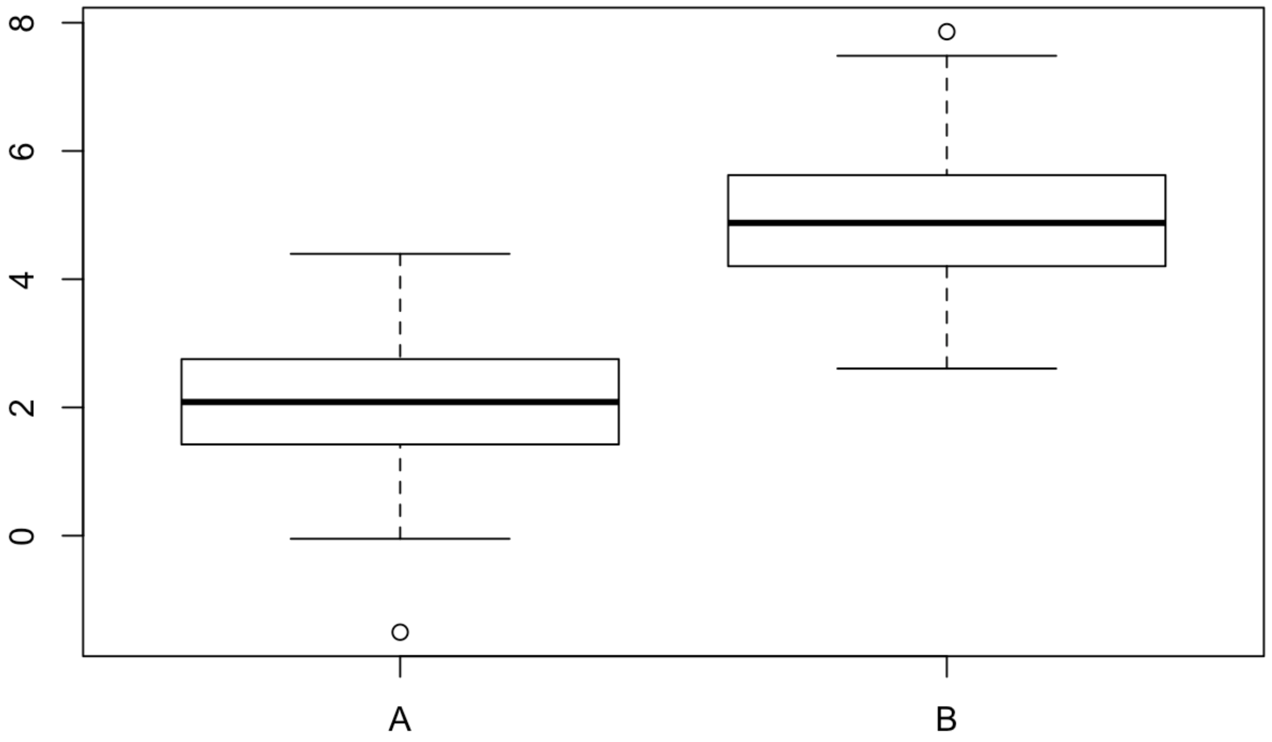
Simple plotting

```
x <- runif(100)
hist(x)
```



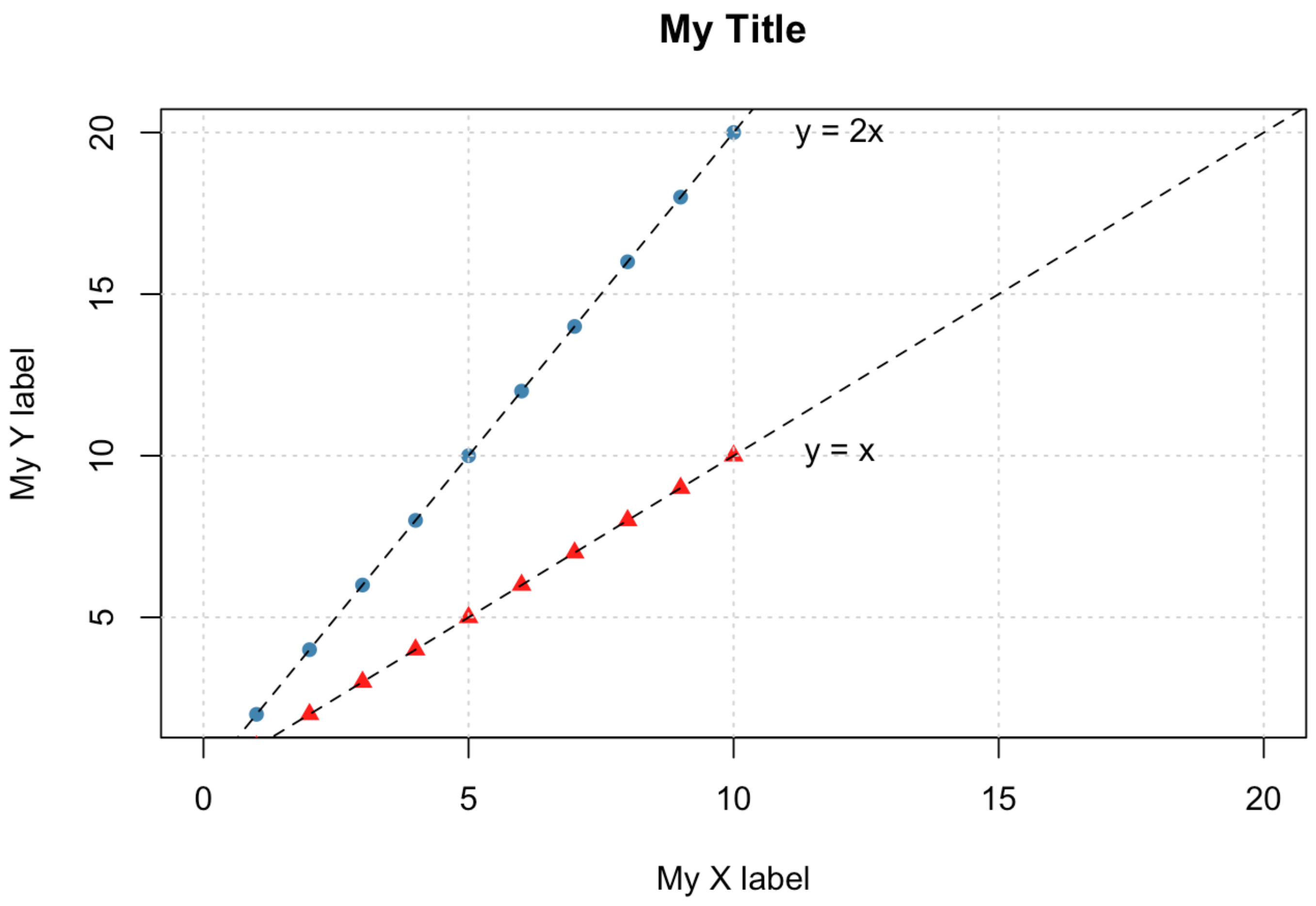
Simple plotting

```
dd <- data.frame(A=rnorm(100, mean = 2),
                 B = rnorm(100, mean=5))
boxplot(dd)
```



Customising a plot

```
plot(x,y,xlab="My X label",ylab="My Y label"
     ,main="My Title",col="steelblue",pch=16,xlim=c(0,20))
points(x,x,col="red",pch=17)
grid()
abline(0,2,lty=2)
abline(0,1,lty=2)
text(12,10, label="y = x")
text(12,20, label="y = 2x")
```



Subsetting / filtering data etc

- R has various comparison operators
  - `<`, `>`, `==`, `!=`
- Each comparison gives a `TRUE` or `FALSE` ***logical*** or ***boolean*** value

```
values <- rnorm(10)
values < 0
```

```
##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
values > 0
```

```
##  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Subsetting / filtering data etc

```
data <- data.frame(Counts = values, Name = rep(c("A","B")))
data
```

```
##        Counts Name
## 1   1.0297018    A
## 2  -0.2027159    B
## 3   1.0352398    A
## 4  -2.1282385    B
## 5  -0.4212584    A
## 6  -0.3770134    B
## 7  -0.7827292    A
## 8  -0.3844247    B
## 9  -1.1625126    A
## 10 -2.0159132    B
```

```
data[values>0,]
```

```
##     Counts Name
## 1 1.029702    A
## 3 1.035240    A
```

Subsetting / filtering data etc

```
data$Name == "A"
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
data[data$Name == "A",]
```

```
##       Counts Name
## 1  1.0297018    A
## 3  1.0352398    A
## 5 -0.4212584    A
## 7 -0.7827292    A
## 9 -1.1625126    A
```

```
data[data$Name !="A",]
```

```
##        Counts Name
## 2  -0.2027159    B
## 4  -2.1282385    B
## 6  -0.3770134    B
## 8  -0.3844247    B
## 10 -2.0159132    B
```

Subsetting / filtering data etc

- Logical vectors can be combined with `&` and `|` when we want ***all*** tests, or ***any*** test, to be true

```
data$Name == "A" & values > 0
```

```
##  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
data[data$Name == "A" & values > 0,]
```

```
##     Counts Name
## 1 1.029702    A
## 3 1.035240    A
```

```
data$Name == "A" | values > 0
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
data[data$Name == "A" | values > 0,]
```

```
##       Counts Name
## 1  1.0297018    A
## 3  1.0352398    A
## 5 -0.4212584    A
## 7 -0.7827292    A
## 9 -1.1625126    A
```

Conditional behaviour

- A logical test can also dictate the behaviour of our code

    - we use the ***if / else*** syntax

```
if (some.condition.holds){
  do.this...
} else {
  do.this.instead.....
}
```

Example code

- Often used in checking for errors and to give more informative messages to the user
- `file.exists` will return `TRUE` or `FALSE` if a specified file name could be found
- `stop` will stop and give a message to the user

```
if(file.exists(myfile)){
  data <- read.delim(myfile)
  ...rest of your code here...
  ...
} else{
    stop("Could not find input file")
}
```

Automating repetitive tasks

- For an analysis involving many steps, we really want to be writing an R ***script***
- Often we want to repeat the same procedure in the script
    - e.g. Histograms of various columns from a file

```
hist(data[,1])
hist(data[,2])
hist(data[,3])
....
```

- Note that each line of code is the same except for the column index

- Tedious if we have a large number of columns
- Prone to error

Using a `for` loop

- We can simplify this code to

```
hist(data[,i])
```

- Where i can be 1, 2, or 3.

Using a `for` loop

```
i <- 1
hist(data[,i])
i <- 2
hist(data[,i])
i <- 3
hist(data[,i])
```

Using a `for` loop

- A loop can defined as follows. The code inside the `{}` will be run for each value of i in turn

```
for(i in 1:3){
  hist(data[,i])
  }
```

Using a `for` loop

- Multiple lines of code can be included in inside the `{}`

```
for(i in 1:3){
  ...process the data....
  hist(data[,i])
  ...customise the plot...
  ...export...
  ...etc...

  }
```

R packages

- The standard download of R includes the basic functions for importing data, doing stats and plotting
- Anything fancier might require extra packages (of which there are 1000s)
- Most populated repository is CRAN: MetaCRAN (http://www.r-pkg.org/)
    - ***Task Views*** can narrow-down your search
- We will be mostly using Bioconductor (www.bioconductor.org)

Installing a package

- You need to install a package once per R version
- From CRAN

```
install.packages("your.package.name.here")
```

- From Bioconductor
    - will also install any packages that it ***depends*** on

```
source("http://www.bioconductor.org/biocLite.R")
biocLite("my.bioc.package")
```

- Every time you want to use the package, you need to use the `library` function

```
library(your.package.name.here)
library(my.bioc.package)
```

Why use R for High-Throughput Analysis?

The Bioconductor project

- Packages analyse all kinds of Genomic data (>800)
- Compulsory documentation (*vignettes*) for each package
- 6-month release cycle
- Course Materials
- Example data and workflows
- Common, re-usable framework and functionality
- Available Support (https://support.bioconductor.org/)

Example packages

Table 5: Citations for select Bioconductor software packages as captured by Google scholar in July, 2014. 'Citation' may be pubmed id.

Package	Citation	N
limma	Smyth (2005)	2417
vsn	12169536	1463
affy	14960456	1448
xcms	16448051	1027
DESeq2	20979621	1669
edgeR	19910308	1110
DNAcopy	15475419	1104
globaltest	14693814	610
lumi	18467348	685
GOstats	17098774	587
limmaGUI	15297296	419
biomaRt	16082012	351
affycomp	14960458	304
aCGH	16159913	287
eisa	12689096	274
MassSpecWavelet	16820428	249
beadarray	17586828	228
cellHTS2	16869968	183
affylmGUI	16455752	150
made4	15797915	134
tilingArray	16787969	124
GEOquery	17496320	124

Downloading a package

Each package has its own landing page. e.g. http://bioconductor.org/packages/release/bioc/html/beadarray.html (http://bioconductor.org/packages/release/bioc/html/beadarray.html). Here you'll find;

- Installation script (will install all dependancies)
- Vignettes and manuals
- Details of package maintainer
- After downloading, you can load using the `library` function. e.g. `library(beadarray)`

Reading data using Bioconductor

Recall that data can be read into R using `read.csv`, `read.delim`, `read.table` etc. Several packages provided special modifications of these to read raw data from different manufacturers

- `limma` for various two-colour platforms
- `affy` for Affymetrix data
- `beadarray`, `lumi`, `limma` for Illumina BeadArray data

- A common class is used to represent the data

Reading data using Bioconductor

A dataset may be split into different components

- Matrix of expression values
- Sample information
- Annotation for the probes

In Bioconductor we will often put these data the same object for easy referencing. The `Biobase` package has all the code to do this.

Example data

- Biobase is the package that provide the infrastructure to represent microarray data
- Evaluating the name of the object does not print the whole object to screen

```
library(Biobase)
data(sample.ExpressionSet)
sample.ExpressionSet
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 500 features, 26 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: A B ... Z (26 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

Extracting data

- Convenient `accessor` functions are provided
- Each row is a ***gene***
- Each column is a ***sample***

```
evals <- exprs(sample.ExpressionSet)
dim(evals)
```

```
## [1] 500  26
```

```
evals[1:4,1:3]
```

```
##                         A         B        C
## AFFX-MurIL2_at  192.7420  85.75330 176.7570
## AFFX-MurIL10_at  97.1370 126.19600  77.9216
## AFFX-MurIL4_at   45.8192   8.83135  33.0632
## AFFX-MurFAS_at   22.5445   3.60093  14.6883
```

Extracting data

- Note the *rows* in the sample information are in the same order as the *columns* in the expression matrix
- information about the sample in ***column*** 1 of the expression matrix is in ***row*** 1 of the pheno data
- etc

```
sampleMat <- pData(sample.ExpressionSet)
dim(sampleMat)
```

```
## [1] 26  3
```

```
head(sampleMat)
```

```
##      sex    type score
## A Female Control  0.75
## B   Male    Case  0.40
## C   Male Control  0.73
## D   Male    Case  0.42
## E Female    Case  0.93
## F   Male Control  0.22
```

Subsetting rules

`ExpressionSet` objects are designed to behave like data frames. e.g. to subset the first 10 genes

```
sample.ExpressionSet[1:10,]
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 10 features, 26 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: A B ... Z (26 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

Subsetting rules

What does this do?

```
sample.ExpressionSet[,1:10]
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 500 features, 10 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: A B ... J (10 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

Subsetting rules

```
males <- sampleMat[,1] == "Male"
sample.ExpressionSet[,males]
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 500 features, 15 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: B C ... X (15 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

```
maleData <- sample.ExpressionSet[,males]
```

Subsetting rules

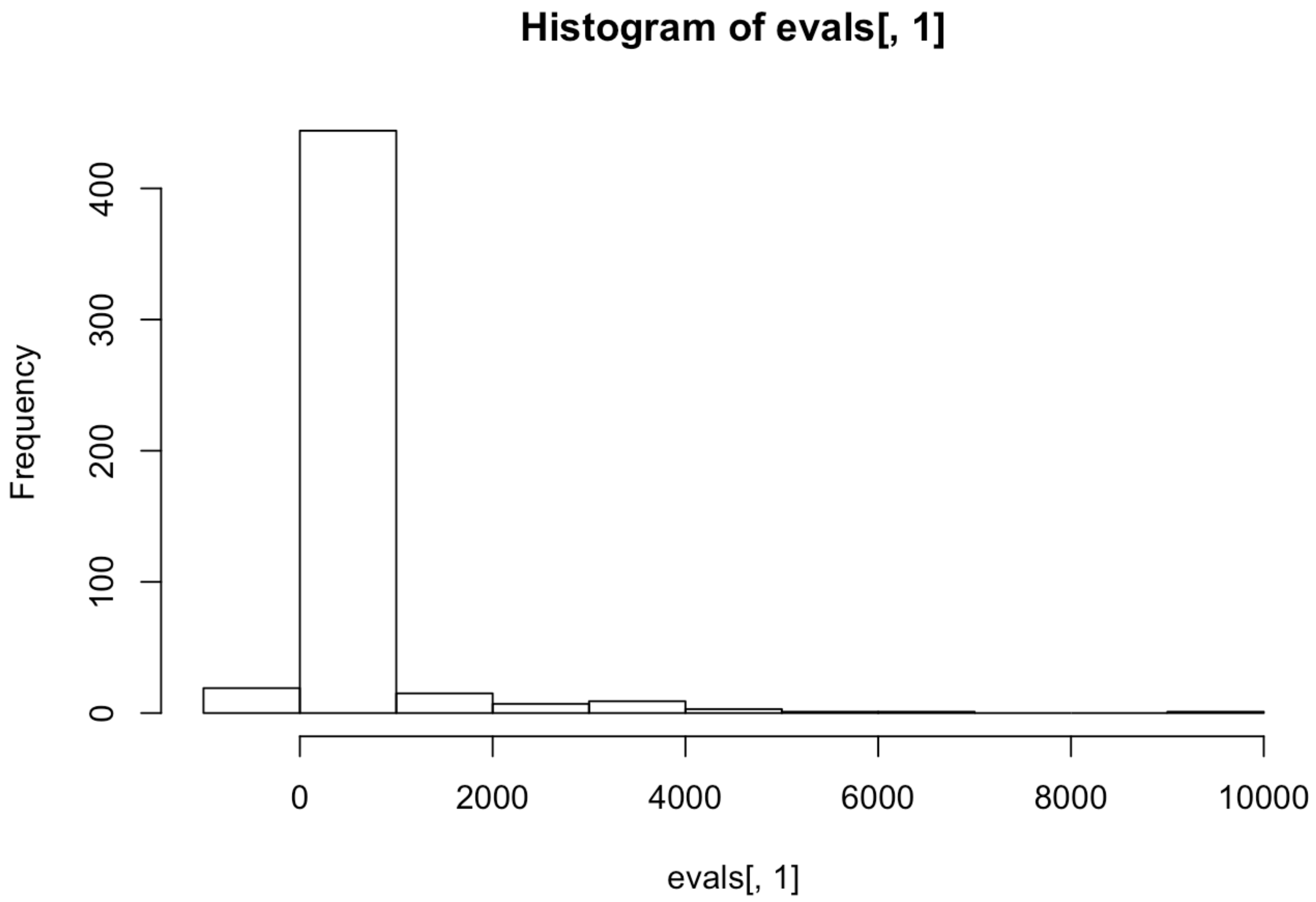
```
sample.ExpressionSet[,
        sampleMat$score < 0.5
        ]
```

```
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 500 features, 14 samples
##   element names: exprs, se.exprs
## protocolData: none
## phenoData
##   sampleNames: B D ... Z (14 total)
##   varLabels: sex type score
##   varMetadata: labelDescription
## featureData: none
## experimentData: use 'experimentData(object)'
## Annotation: hgu95av2
```

Starting to visualise the data

Recall that several plots can be created from a *vector* of numerical values

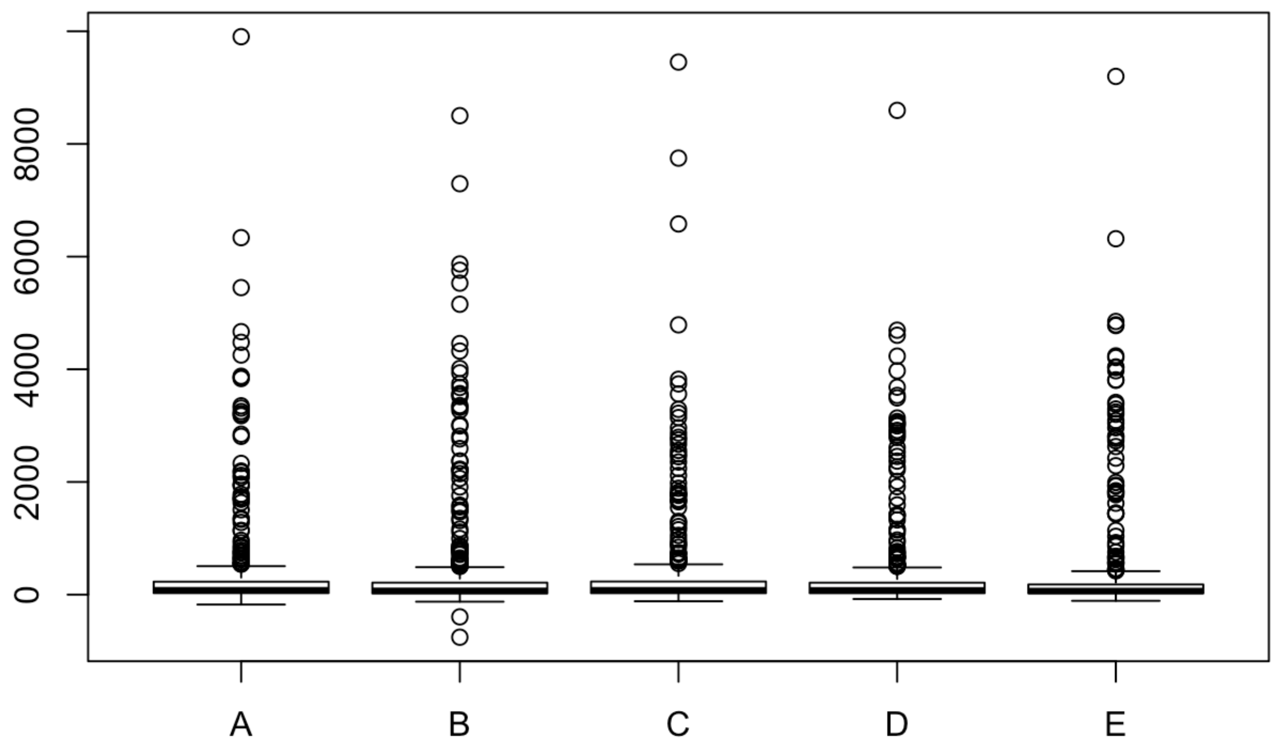
```
hist(evals[,1])
```



Starting to visualise the data

Or from a data frame

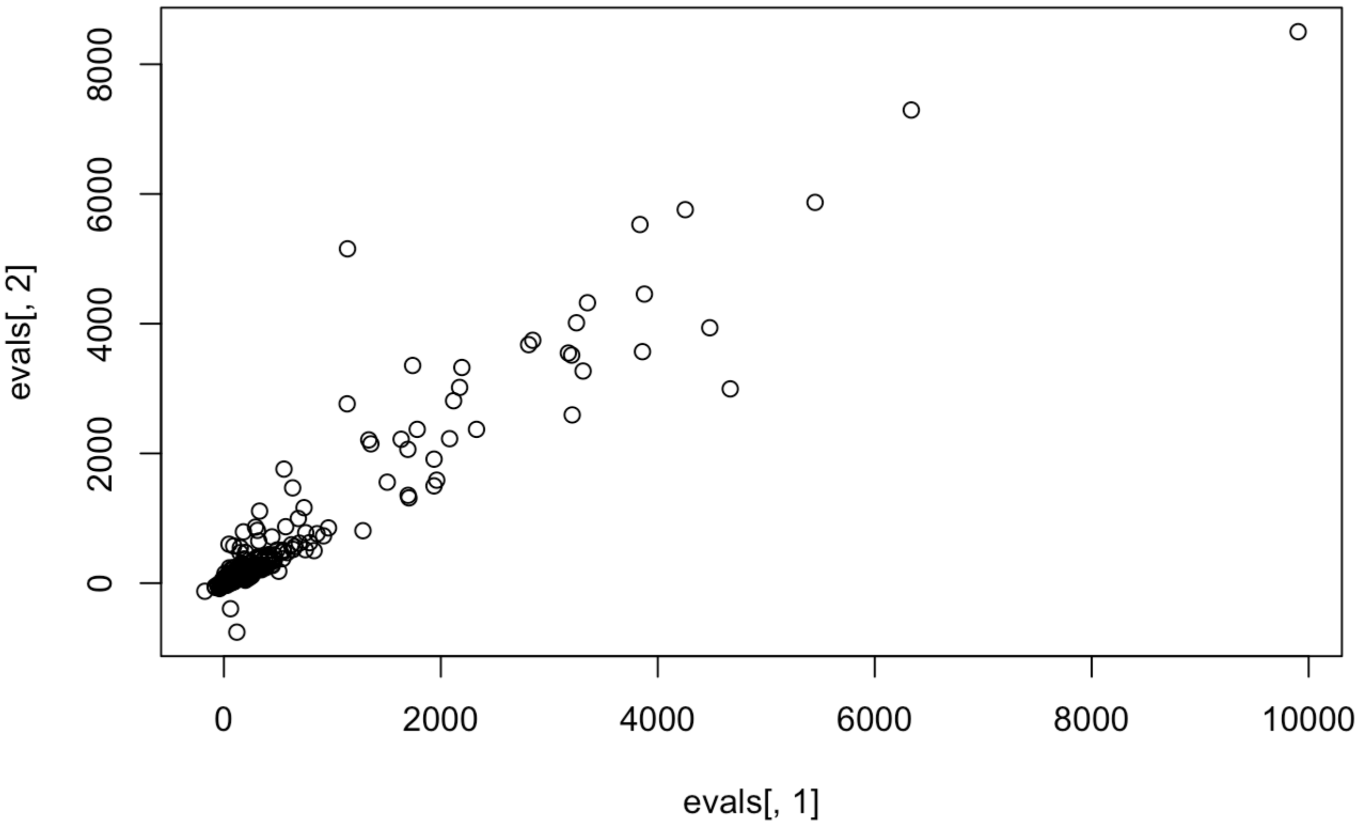
```
boxplot(evals[,1:5])
```



Starting to visualise the data

One sample against another

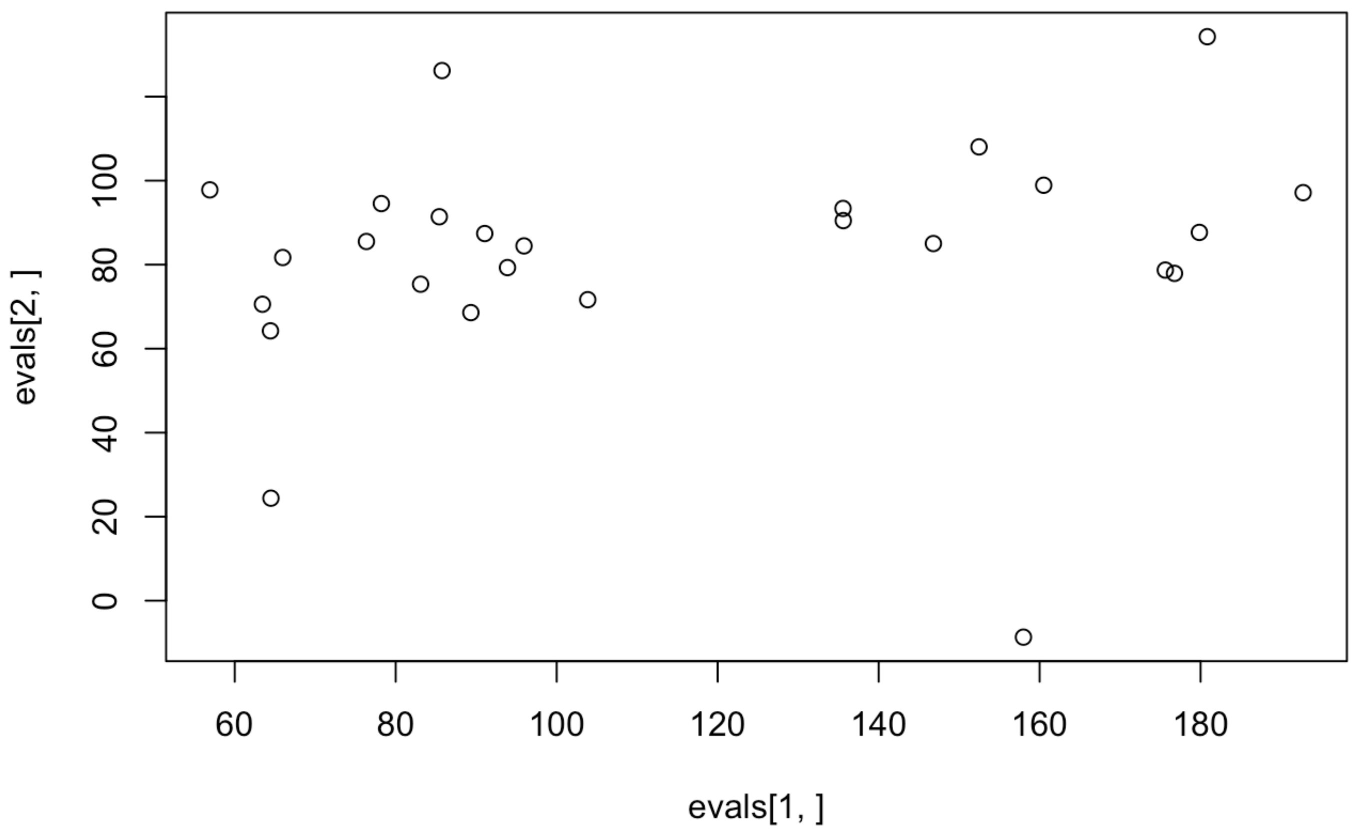
```
plot(evals[,1],evals[,2])
```



Starting to visualise the data

One gene against another

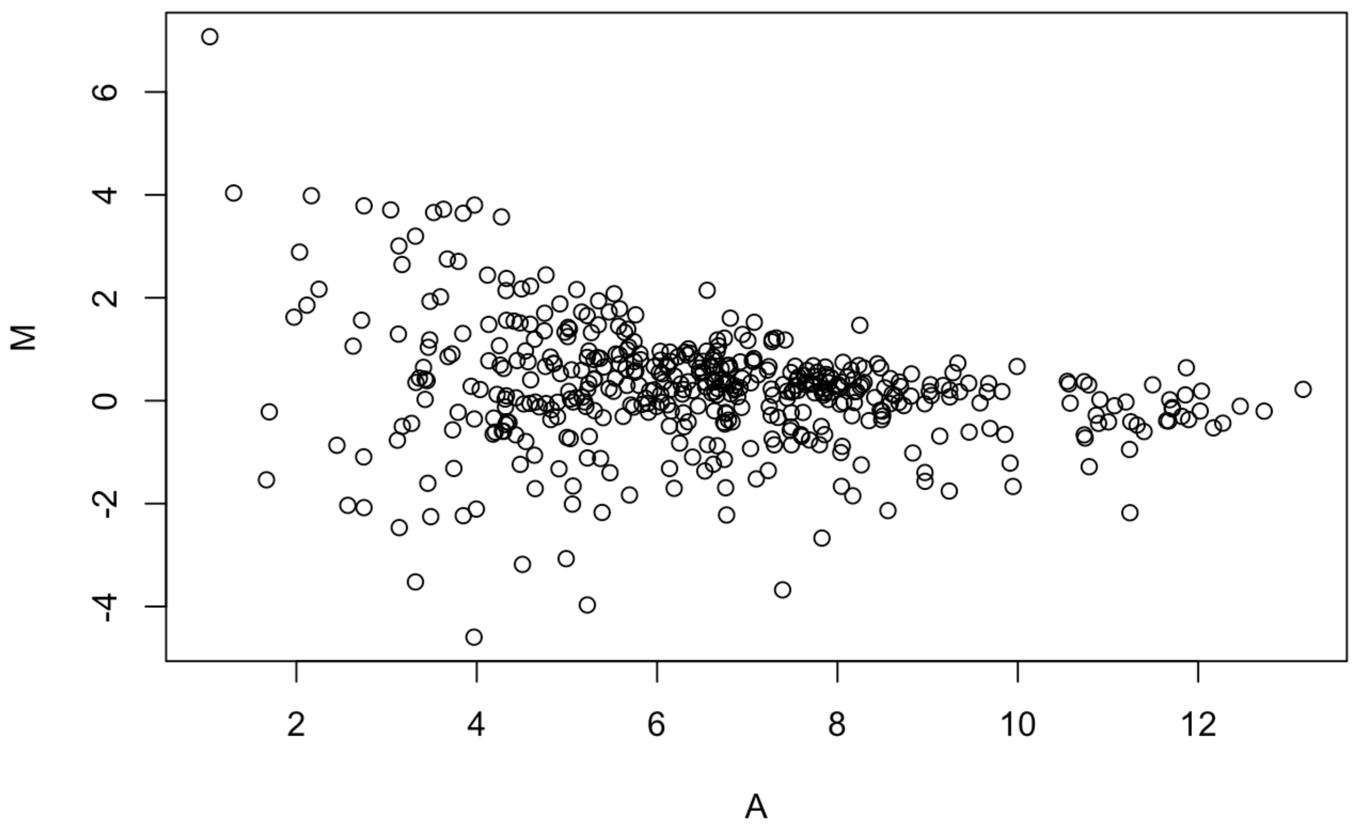
```
plot(evals[1,],evals[2,])
```



The MA plot

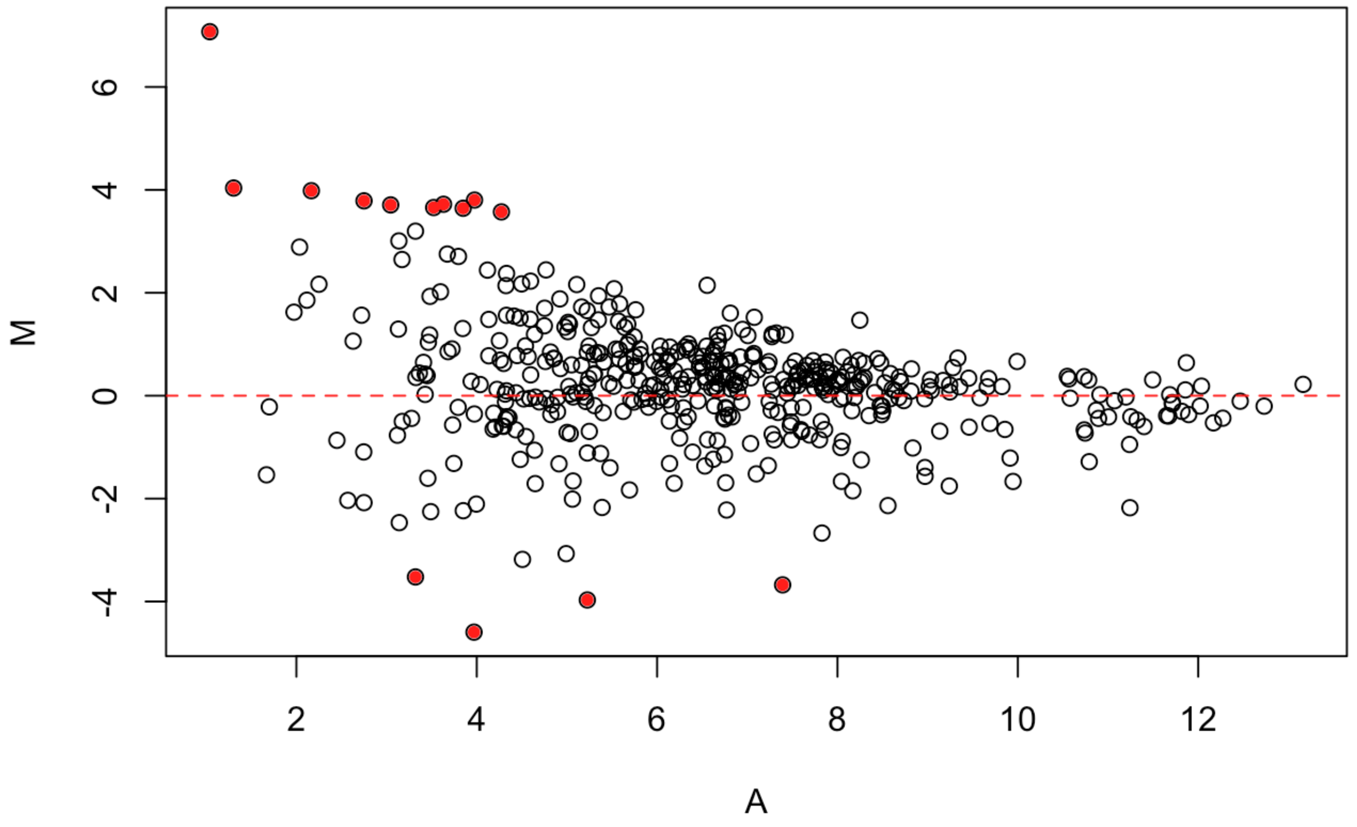
We often work with **M** and **A** values as defined

```
M <- log2(evals[,1]) - log2(evals[,2])
A <- 0.5*(log2(evals[,1]) + log2(evals[,2]))
plot(A,M)
```



The MA plot

- log transformation is used to put values on scale 0 to 16
- Line *M=0* indicates equivalent expression in two arrays
  - where we would expect most genes to be
- Outliers on y axis are *candidates* to be *differentially expressed*



Statistical Testing

- `R` started as a language *for* statisticians, *made by* statisticians
- naturally, it has a whole range of statistical tests available as functions
  - `t.test`
  - `wilcox.test`
  - `var.test`
  - `anova`
  - etc.....

Statistical Testing

```
mygene
```

```
##         A          B          C          D          E          F 
##  11.069500 -26.100600  14.165500   8.759730   4.473810   9.857120 
##         G          H          I          J          K          L 
##   2.129690  -3.160350  23.917000   8.841620 -13.306100   6.991630 
##         M          N          O          P          Q          R 
##   4.625380  -7.571780  23.905600  11.327000  10.738700  12.639800 
##         S          T          U          V          W          X 
##   9.737230  12.834800  -0.203757  22.000500  12.463800   2.936350 
##         Y          Z 
##  10.891500  12.021200
```

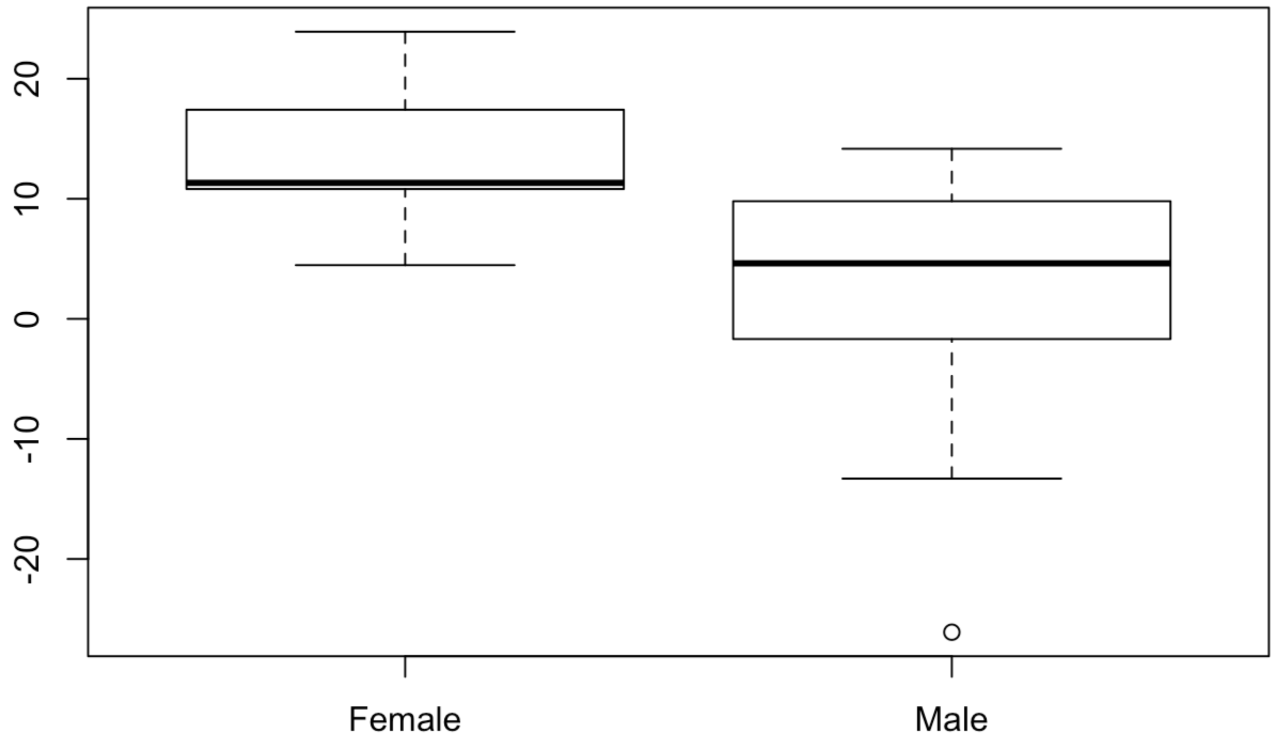
```
myfactor
```

```
##  [1] Female Male   Male   Male   Female Male   Male   Male   Female Male  
## [11] Male   Female Male   Male   Female Female Female Male   Male   Female
## [21] Male   Female Male   Male   Female Female
## Levels: Female Male
```

Statistical Testing

The ***tilde*** (`~`) is R’s way of creating a ***formula***

```
boxplot(mygene~myfactor)
```



Statistical Testing

```
t.test(mygene~myfactor)
```

```
##
##  Welch Two Sample t-test
##
## data:  mygene by myfactor
## t = 3.215, df = 23.216, p-value = 0.003808
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   4.020114 18.508597
## sample estimates:
## mean in group Female   mean in group Male
##            13.651931             2.387576
```

- we need to be wary of multiple-testing issues

Biological Interpretation of Results

- Bioconductor provide a number of annotation packages
  - e.g. `hgu95av2.db` which can be installed in the same manner as other Bioconductor packages
  - can map between manufacturer ID more-familiar IDs
  - can map to pathways ontologies
    - using the latest database versions etc

```
library(hgu95av2.db)
mget("31553_at",hgu95av2SYMBOL)
```

```
## $`31553_at`
## [1] "ZNF460"
```

```
mget("31553_at",hgu95av2ENTREZID)
```

```
## $`31553_at`
## [1] "10794"
```

```
mget("31553_at",hgu95av2GO)
```

```
## $`31553_at`
## $`31553_at`$`GO:0006351`
## $`31553_at`$`GO:0006351`$GOID
## [1] "GO:0006351"
##
## $`31553_at`$`GO:0006351`$Evidence
## [1] "IEA"
##
## $`31553_at`$`GO:0006351`$Ontology
## [1] "BP"
##
##
## $`31553_at`$`GO:0006355`
## $`31553_at`$`GO:0006355`$GOID
## [1] "GO:0006355"
##
## $`31553_at`$`GO:0006355`$Evidence
## [1] "IEA"
##
## $`31553_at`$`GO:0006355`$Ontology
## [1] "BP"
##
##
## $`31553_at`$`GO:0005634`
## $`31553_at`$`GO:0005634`$GOID
## [1] "GO:0005634"
##
## $`31553_at`$`GO:0005634`$Evidence
## [1] "IEA"
##
## $`31553_at`$`GO:0005634`$Ontology
## [1] "CC"
```

```
##
##
## $`31553_at`$`GO:0003677`
## $`31553_at`$`GO:0003677`$GOID
## [1] "GO:0003677"
##
## $`31553_at`$`GO:0003677`$Evidence
## [1] "IEA"
##
## $`31553_at`$`GO:0003677`$Ontology
## [1] "MF"
##
##
## $`31553_at`$`GO:0046872`
## $`31553_at`$`GO:0046872`$GOID
## [1] "GO:0046872"
##
## $`31553_at`$`GO:0046872`$Evidence
## [1] "IEA"
##
## $`31553_at`$`GO:0046872`$Ontology
## [1] "MF"
##
##
## $`31553_at`$`GO:0005515`
## $`31553_at`$`GO:0005515`$GOID
## [1] "GO:0005515"
##
## $`31553_at`$`GO:0005515`$Evidence
## [1] "IPI"
##
## $`31553_at`$`GO:0005515`$Ontology
## [1] "MF"
```

Introducing the practical

- Refresh your memory of R skills
  - reading data
  - subsetting data
  - plotting
- Introduce some Bioconductor classes