OEB 153 – Lab 2

Introduction

So far in the Lab part of this course, we've learned much of the basic functionality of the R language, including

- how to store numerical and categorical data,
- how to import and manipulate data,
- how to perform calculations using for loops, and
- how to create scatterplots and histograms.

In this lab we will continue with a focus on practical skills in R. We'll learn additional ways of storing data, additional ways of interacting with data frames, more on how to inspect and explore data, and how to carry out some of the basic hypothesis tests and inference procedures that R offers out of the box.

Matrices

Up until now we have used numerical vectors, factors, and data frames to store our data. Another type of data object in R is the **matrix**. Just as in mathematics, you can think of a matrix as a two-dimensional vector. In R, a matrix *can only store one type of variable.* (That is, a matrix is numerical, character, or logical.) To create a matrix, we can use the `matrix()` function. For example, to create the numerical matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

we would use the following command:

```
A = matrix(c(1,2,3,4), 2, 2, byrow=T)
A
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

The first argument to `matrix()` is a vector containing the data in the matrix. Following that are the number of rows (option name `nrow`) and number of columns (`ncol`). An additional option, `byrow`, indicates whether the data are entered row-wise, rather than columnwise. The default value of `byrow` in is `FALSE`, so if you want to enter the data by row, be sure to specify `byrow=T`. If we had failed to specify `byrow=T` above, we would have gotten the wrong matrix:

```
B = matrix(c(1,2,3,4), 2, 2)
B
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Note The logical value `TRUE` can be abbreviated `T`, and `FALSE` can be abbreviated `F`. `T` and `F` are not reserved words in R: It is possible to define a variable called `T` or `F`. This is a **very bad idea**… don't do it!

Just as $A_{i,j}$ refers to the element in the ith row and jth column in mathematical notation, `A[i,j]` refers to the same element in R syntax. Likewise, just as $A_{i,\bullet}$ refers to the ith row in math, `A[i,]` refers to the ith row in R. The same goes for $A_{\bullet,j}$ and `A[,j]`. To get the second row of our matrix `A`, we would use `A[2,]`:

```
A[2,]
```

```
[1] 3 4
```

Assignments to entire rows or columns can also be made. If the data changes and now we want the second row of A to be $(5, 6)$, we would change A like this:

```
A[2,] = c(5,6)
A
```

```
     [,1] [,2]
[1,]    1    2
[2,]    5    6
```

The area of mathematics that deals with matrices and vectors is linear algebra. R offers the full suite of linear algebra operations and functions. As a commonly used example, the operator `%*%` performs matrix multiplication between two compatible matrices, between a matrix and a compatible vector, or between two compatible vectors. We won't go into detail about linear algebra operations in this course. If you need to use R for matrix algebra, a good place to start is Quick-R.

Lists

In addition to numerical vectors, factors, matrices, and data frames, data can also be stored in **lists**. A list is a very flexible data object that can be used to store any type of data. The objects stored in a list can be vectors, matrices, data frames, factors single numeric values, TRUE/FALSE values, anything you can think of, including other lists. Not all objects have to be of the same type, either:

```
# create a list containing a numerical vector,
# a logical value, and the string "Lisa Simpson"
list1 = list(c(1,2), TRUE, "Lisa Simpson")
list1
```

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

[[2]]
[1] TRUE

[[3]]
[1] "Lisa Simpson"
```

To refer to an element stored in a list (either for reference or assignment), you use *double square brackets*:

```
list1[[1]]
```

```
[1] 1 2
```

```
list1[[3]] = "Homer Simpson"
list1
```

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

[[2]]
[1] TRUE

[[3]]
[1] "Homer Simpson"
```

Elements in a list can also be given names and be referred to by their names. For example, to create a list containing a person's name, year of birth, and email address, you might do something like the following:

```
darwin = list(name = "Charles Darwin", birthYear = 1809, email = "chuckd09@hotmail.com")
darwin
```

```
$name
[1] "Charles Darwin"

$birthYear
[1] 1809

$email
[1] "chuckd09@hotmail.com"
```

There are two ways of referring to a named element in a list.

1. Use double brackets and *put the name in quotation marks* (i.e., use a character object for the name):

```
darwin[["name"]]
```

```
[1] "Charles Darwin"
```

2. Put the name after a dollar sign *without quotation marks*:

```
darwin$email
```

```
[1] "chuckd09@hotmail.com"
```

The first way of referring to named elements in a list is more flexible, since the character object in the double brackets can be represented by a variable. This is not true for the second method.

An object can also be stored in a list after the list is created:

```
darwin[["birthplace"]] = "Shrewsbury"
darwin
```

```
$name
[1] "Charles Darwin"

$birthYear
[1] 1809

$email
[1] "chuckd09@hotmail.com"

$birthplace
[1] "Shrewsbury"
```

Lists are great objects for storing *ragged data*, where the type of data or the number of variables varies from case to case. In general, though, lists are often the least efficient (i.e., slowest) way of storing and working with data in R, and if there is an obvious way to use a vector, matrix, or data frame, these are usually preferable to lists.

Data frames, continued

We now return to data frames, the most common way of storing statistical data in R. We will work again with the dataset from Lab 0, from Gilbert JDJ, Manica A (2010) Parental care trade-offs and life history relationships in insects. American Naturalist 176: 212-226. The dataset was called `insect_stats.csv` in the previous lab (now called `gilbert_and_manica.csv` on the course website). Previously we worked with a version of the dataset not containing all the variables and from which the missing values had been removed. In this lab, we will work with the original data file, available from the course website as `gilbert_and_manica.csv`or from Data Dryad as `tableS3.csv` here.

If you look at the content of the data file, you will see that missing values are denoted with the `*` character, which needs to be specified in the import command:

```
# import insect data, 'ins' for short
ins  = read.csv("gilbert_and_manica.csv", header=T, na.strings="*")

str(ins)
```

```
'data.frame':   286 obs. of  7 variables:
 $ Order             : Factor w/ 16 levels "Coleoptera","Dermaptera",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Family            : Factor w/ 105 levels "Acanthosomatidae",..: 4 14 15 15 16 18 18 20 22 22 ...
 $ Binomial          : Factor w/ 286 levels "Abedus_herberti",..: 16 3 43 70 37 41 154 187 4 7 ...
 $ Parental.care     : Factor w/ 4 levels "Guarding","No care",..: 2 2 2 2 2 2 2 2 1 2 ...
 $ Dry.weight        : num  1.43 5.65 234.52 33.16 5.9 ...
 $ Lifetime.fecundity: num  55 68 279 56 83.5 ...
 $ Egg.volume        : num  NA NA NA NA NA NA NA NA 1.57 1.19 ...
```

Note   You can use the `str()` function to see the *structure* of a data frame (or any other R object), which can be more informative than the `head()` function. In calling `str(ins)`, we are reminded that the values that can be taken in a factor are actually represented as integers, each of which corresponds to a certain level.

Inspecting and changing names of variables The names of the variables in a data frame can be inspected and changed. The function `names()` can be used to do both:

```
names(ins)
```

```
[1] "Order"               "Family"              "Binomial"
[4] "Parental.care"       "Dry.weight"          "Lifetime.fecundity"
[7] "Egg.volume"
```

```
names(ins) = c("order", "family", "binomial", "care", "mass", "fecundity", "egg.volume")

# actually, I don't want a period in 'egg.volume'
names(ins)[7] = "eggvolume"

names(ins)
```

```
[1] "order"     "family"    "binomial"  "care"      "mass"      "fecundity"
[7] "eggvolume"
```

Data frame indexing and subsetting Data frames can be indexed in many ways, not just with the `dat$var[index]` syntax that we have used up until now. Data frames are like both lists and matrices. Like matrices, elements in data frames can be referred to by their row and column:

```
# same as ins$family[1]
ins[1,2]
```

```
[1] Anobiidae
105 Levels: Acanthosomatidae Acrididae Anisolabididae ... Trypetidae
```

```
# same as ins$care[3]
ins[3,4]
```

```
[1] No care
Levels: Guarding No care No care [for female] Provisioning
```

```
# second row of the data table, spilling across two lines of output:
ins[2,]
```

```
       order    family                 binomial    care mass fecundity
2 Coleoptera Bruchidae Acanthoscelides_obtectus No care 5.65        68
  eggvolume
2        NA
```

Like lists, elements of a data frame can be referred to by name. This can be done in multiple ways, not just like `ins$eggvolume`:

```
# what we already know
ins$eggvolume[20]
```

```
[1] 0.6
```

```
# equivalent to above
ins[20, "eggvolume"]
```

```
[1] 0.6
```

```
# get both mass and fecundity of 20th row
ins[20, c("mass", "fecundity")]
```

```
    mass fecundity
20 26.38       700
```

Sometimes it is necessary to remove variables from a data frame. This can be done a few different ways. Just as with one-dimensional vectors, providing a negative index for the column creates a copy of the data frame with that column removed:

```
# a look at ins without the first column
head(ins[,-1])
```

```
       family                  binomial    care   mass fecundity eggvolume
1   Anobiidae        Anobium_punctatum No care   1.43      55.0        NA
2   Bruchidae Acanthoscelides_obtectus No care   5.65      68.0        NA
3 Buprestidae     Capnodis_tenebrionis No care 234.52     279.0        NA
4 Buprestidae            Coraebus_rubi No care  33.16      56.0        NA
5   Byturidae      Byturus_tomentosus No care   5.90      83.5        NA
6   Carabidae     Calosoma_sycophanta No care 305.90    1959.0        NA
```

```
# a look at ins without the first and third columns
head(ins[,-c(1,3)])
```

```
       family    care   mass fecundity eggvolume
1   Anobiidae No care   1.43      55.0        NA
2   Bruchidae No care   5.65      68.0        NA
3 Buprestidae No care 234.52     279.0        NA
4 Buprestidae No care  33.16      56.0        NA
5   Byturidae No care   5.90      83.5        NA
6   Carabidae No care 305.90    1959.0        NA
```

It is also possible to remove columns by name, but this is a little more difficult, since negative strings don't exist in R. (That is, R would have no idea what `ins[,-"name"]` means, since `-"name"` is nonsense.) To remove columns by name, we have to use logical vectors.

```
# create a data frame without the parental care variable
# keepVars is a logical vector indicating which variables to keep
# (put right-hand-side in parentheses *just for readability*)
keepVars = (names(ins) != "care")
# inspect keepVars side-by-side with names(ins)
data.frame(names(ins), keepVars)
```

```
  names.ins. keepVars
1      order     TRUE
```

```
2     family     TRUE
3   binomial     TRUE
4       care    FALSE
5       mass     TRUE
6  fecundity     TRUE
7  eggvolume     TRUE
```

```
ins.nocare = ins[, keepVars]

# create a data frame without the parental care and mass variables:
keepVars2 = !(names(ins) %in% c("care", "mass"))
# inspect keepVars2 side-by-side with names(ins)
data.frame(names(ins), keepVars2)
```

```
  names.ins. keepVars2
1      order      TRUE
2     family      TRUE
3   binomial      TRUE
4       care     FALSE
5       mass     FALSE
6  fecundity      TRUE
7  eggvolume      TRUE
```

```
ins.nocarenomass = ins[, keepVars2]
str(ins.nocarenomass)
```

```
'data.frame':   286 obs. of  5 variables:
 $ order    : Factor w/ 16 levels "Coleoptera","Dermaptera",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ family   : Factor w/ 105 levels "Acanthosomatidae",..: 4 14 15 15 16 18 18 20 22 22 ...
 $ binomial : Factor w/ 286 levels "Abedus_herberti",..: 16 3 43 70 37 41 154 187 4 7 ...
 $ fecundity: num  55 68 279 56 83.5 ...
 $ eggvolume: num  NA NA NA NA NA NA NA NA 1.57 1.19 ...
```

Note The `!` operator in the above is the "not" operator. It changes `TRUE` to `FALSE` and vice versa. In the above it's used to find the variables names that are *not* in `c("care", "mass")`. We've seen this before, but it's good to become reaquainted with it.

Of course, logical vectors can also be used to subset a data frame by *rows*. To create a subset of the insect data frame where egg volume is non-missing, we would use the following:

```
# remove rows with egg volume missing
# recall that is.na() returns whether each element is missing
ins.withEggVol = ins[!is.na(ins$eggvolume),]
str(ins.withEggVol)
```

```
'data.frame':   74 obs. of  7 variables:
 $ order    : Factor w/ 16 levels "Coleoptera","Dermaptera",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ family   : Factor w/ 105 levels "Acanthosomatidae",..: 22 22 24 25 25 25 25 30 30 30 ...
 $ binomial : Factor w/ 286 levels "Abedus_herberti",..: 4 7 174 54 64 66 144 136 137 138 ...
 $ care     : Factor w/ 4 levels "Guarding","No care",..: 1 2 2 2 2 2 2 2 2 2 ...
 $ mass     : num  22.96 2.69 8.05 9.68 26.38 ...
 $ fecundity: num  NA NA 89.6 NA 700 ...
 $ eggvolume: num  1.57 1.19 0.13 5.58 0.6 0.6 0.6 0.15 0.15 2.31 ...
```

The `&` ("elementwise and") and `|` ("elementwise or," typed `Shift+\`) operators are used to combine multiple logical vectors. (We saw these in Lab 0, but it's worth re-introducing them now as we talk more about data frame subsetting.) The operator `&` takes two logical vectors and returns whether both values are `TRUE` at each position in the two vectors:

```r
# & applied to two logical vectors
c(TRUE, TRUE, FALSE, FALSE) & c(FALSE, TRUE, TRUE, FALSE)
```

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

The "elementwise or" operator works in a similar way, except it returns a vector indicating at each position whether either the position is `TRUE` in the first vector **or** whether the position is true in the second vector.

```r
# & applied to the same two logical vectors
c(TRUE, TRUE, FALSE, FALSE) | c(FALSE, TRUE, TRUE, FALSE)
```

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

R also has the `&&` and `||` operators, which operate only the first element in each vector they are applied to. These are used in programming control-flow with conditional (if/else) statements, which we will not cover in this lab.

We can use `&` and `|` to subset a data frame by multiple conditions at once. (This also works for subsetting vectors.) To get a data frame representing beetles (order Coleoptera) with "guarding" parental care, we would use the following command:

```r
# Reminder of the levels of the care variable
levels(ins$care)
```

```
[1] "Guarding"             "No care"              "No care [for female]"
[4] "Provisioning"
```

```r
# subset with only guardian beetles
guardingBeetles = ins[ins$order == "Coleoptera" & ins$care == "Guarding",]
head(guardingBeetles)
```

```
        order        family                binomial     care  mass
9  Coleoptera Chrysomelidae          Acromis_sparsa Guarding 22.96
14 Coleoptera Chrysomelidae     Gonioctena_japonica Guarding 21.51
77 Coleoptera    Scolytidae Coccotrypes_carpophagus Guarding  2.01
79 Coleoptera    Scolytidae      Scolytus_rugulosus Guarding  1.92
80 Coleoptera     Silphidae    Ablattaria_laevigata Guarding 87.98
   fecundity eggvolume
9         NA      1.57
14      40.6        NA
77      40.5        NA
79      55.0        NA
80     101.0        NA
```

To create a subset of the data containing species that mass greater than 200 mg or egg volume greater than 40 mm^3 we might try the following:

```r
bigbugs = ins[ins$mass > 200 | ins$eggvolume > 40,]
str(bigbugs)
```

```
'data.frame':   222 obs. of  7 variables:
 $ order    : Factor w/ 16 levels "Coleoptera","Dermaptera",..: NA NA 1 NA NA 1 NA NA NA NA ...
 $ family   : Factor w/ 105 levels "Acanthosomatidae",..: NA NA 15 NA NA 18 NA NA NA NA ...
 $ binomial : Factor w/ 286 levels "Abedus_herberti",..: NA NA 43 NA NA 41 NA NA NA NA ...
 $ care     : Factor w/ 4 levels "Guarding","No care",..: NA NA 2 NA NA 2 NA NA NA NA ...
 $ mass     : num  NA NA 235 NA NA ...
 $ fecundity: num  NA NA 279 NA NA ...
 $ eggvolume: num  NA NA NA NA NA NA NA NA NA NA ...
```

This subset also includes rows with missing values because by default R includes these in indexing. When a value is included because of a NA in an index, R replaces the value of what's indexed with NA. To remove cases where both egg volume and mass are missing, we would use the following:

```r
# overwrite bigbugs, keeping only cases where either egg volume is non-missing
# or mass is non-missing
bigbugs = bigbugs[!is.na(bigbugs$eggvolume) | !is.na(bigbugs$mass),]
head(bigbugs)
```

```
        order        family             binomial         care  mass
3  Coleoptera  Buprestidae Capnodis_tenebrionis      No care 234.5
6  Coleoptera    Carabidae  Calosoma_sycophanta      No care 305.9
57 Coleoptera   Passalidae  Popilius_disjunctus Provisioning 392.9
63 Coleoptera Scarabaeidae          Bubas_bison      No care 223.8
65 Coleoptera Scarabaeidae      Copris_diversus Provisioning 463.0
74 Coleoptera Scarabaeidae   Kheper_nigroaeneus Provisioning 357.9
   fecundity eggvolume
3      279.0        NA
6     1959.0        NA
57        NA     72.38
63      60.0        NA
65       3.5        NA
74       2.0     19.56
```

Adding columns to data frames Just as it is possible to add entries to a list, it is possible to add columns to a data frame. For example, if we wanted to have a variable in our data frame ins representing fecundity per mass, we could define the following:

```r
# add a fecundity per mass variable to the ins data frame
ins$fecunditypermass = ins$fecundity / ins$mass
str(ins)
```

```
'data.frame':   286 obs. of  8 variables:
 $ order           : Factor w/ 16 levels "Coleoptera","Dermaptera",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ family          : Factor w/ 105 levels "Acanthosomatidae",..: 4 14 15 15 16 18 18 20 22 22 ...
 $ binomial        : Factor w/ 286 levels "Abedus_herberti",..: 16 3 43 70 37 41 154 187 4 7 ...
 $ care            : Factor w/ 4 levels "Guarding","No care",..: 2 2 2 2 2 2 2 2 1 2 ...
 $ mass            : num  1.43 5.65 234.52 33.16 5.9 ...
 $ fecundity       : num  55 68 279 56 83.5 ...
 $ eggvolume       : num  NA NA NA NA NA NA NA NA 1.57 1.19 ...
 $ fecunditypermass: num  38.46 12.04 1.19 1.69 14.15 ...
```

Attaching and detaching data frames Sometimes it is tedious to type the name of a data frame each time you want to refer to a variable within it. This is where the functions `attach()` and `detach()` are helpful. Attaching a data frame makes it possible to refer to the columns within it as variables. For example, if we attach the data frame `ins` with the command `attach(ins)`, we can calculate the mean egg volume per mass with the following code:

```r
attach(ins)
# notice there is no reference to ins
mean(eggvolume/mass, na.rm = T)
```

```
[1] 0.3272
```

```r
# always detach after attaching!!!
detach()
# after detaching, eggvolume will not be found
eggvolume
```

```
Error: object 'eggvolume' not found
```

Generally, attaching a data frame is a bad idea. It is easy to forget that you are actually working within a data frame, and sometimes you will find that you have column names that conflict with variable or function names outside of the data frame. Also, you can only read from attached variable names; to write to them, you have to refer to the data frame. If you are going to use `attach()` be sure to write the `detach()` call into your script at the same time you write the call to `attach()`, so that you don't forgot to call `detach()` later.

A good alternative is the `with()` function, which allows you to perform a calculation within the context of a particular data frame:

```r
# same calculation as above, but using with()
with(ins, mean(eggvolume/mass, na.rm = T))
```

```
[1] 0.3272
```

Built-in hypothesis tests and inference procedures

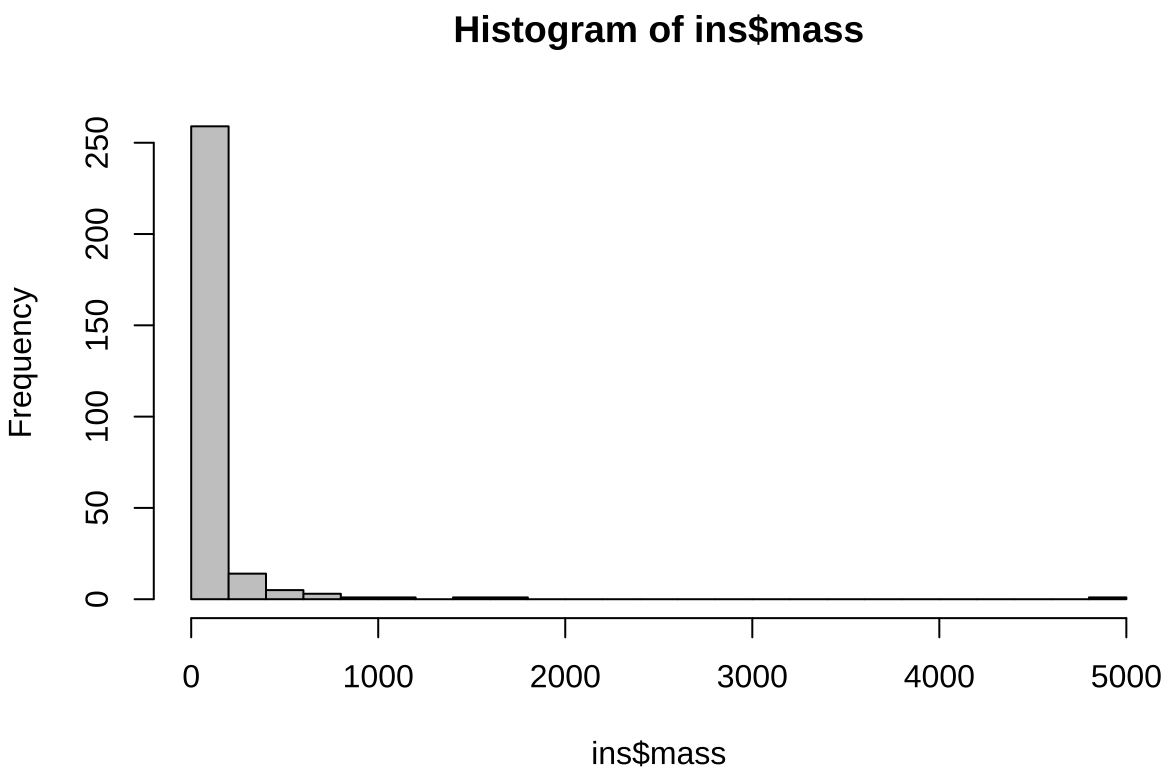
So far we have hardly done any statistical tests in R! We've done maximum-likelihood estimation of proportions, and we've calculated some confidence intervals for those proportions, but we have yet to perform any hypothesis tests.

R provides functions to perform all of the standard hypothesis tests, plus many more. Each of these carry with them a set of assumptions about the data, which need to be checked whenever these tests are applied. **In this lab we will not go into these assumptions. Instead, we will focus on how to carry out the tests. It is up to you to check the assumptions of a test and make sure it is appropriate for the data and questions you have!** You will learn about the assumptions of these tests as you learn about the tests in lecture in the coming weeks.

Transformations and assessing for Normal distribution

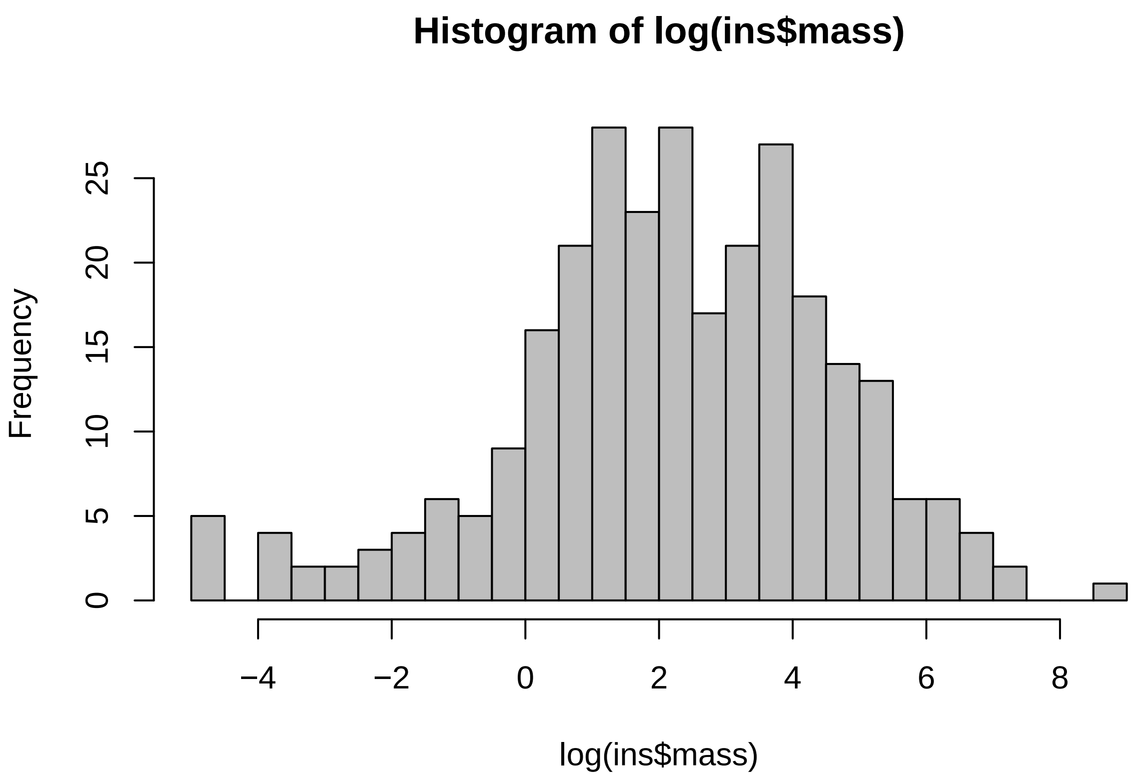
Before we get started with hypothesis tests, we will spend some time learning how to check one of the most common assumptions of a hypothesis: that the data are Normally distributed. The most intuitive way to determine whether data are Normally distributed is by looking directly at their distribution. For example, let's use a histogram to examine the distribution of insect masses in our data set:

```
hist(ins$mass, col = "gray",breaks=20)
```



This data are clearly not Normally distributed. When data aren't Normally distributed, it's often possible to perform a *transformation* on the data such that the transformed data are Normally distributed. When data are skewed to the right (i.e., when they have a long tail), the logarithm of the data often looks Normally distributed:

```
hist(log(ins$mass), col = "gray", breaks = 20)
```



This looks much more Normally distributed. (Another sometimes-found transformation to make data more Normal is to apply the arcsin function to data, `asin()` in R. This would only work for proportions. The square root transformation is sometimes also encountered.) To further check for Normality, we can fit a Normal distribution to the mean and standard deviation (i.e., we can use the method of moments to fit a Normal distribution to the data), and then draw the density of that curve on the histogram. The histogram will need to be redrawn with the `prob=TRUE` option specified, which Normalizes the area in the histogram's rectangles to 1. Then we can add the density of the Normal distribution to the histogram using the `lines()` function, which is called after `hist()` or `plot()`, or whatever other function we used to create the plot.

```
# estimate the mean
ybar = mean(log(ins$mass), na.rm=T)
# estimate the standard deviation
s = sd(log(ins$mass), na.rm=T)
# examine the mean and standard deviation
ybar; s
```

```
[1] -Inf
```

```
[1] NaN
```

The fitted mean and standard deviation reflect a zero value amongst the insect masses. (Recall that $\log(0) = -\infty$. The sample mean is thus also negative infinity, and the standard deviation is undefined, or `NaN` ["not a number"].) We can either remove the zero values or add a small value to all of the masses, something like 0.001 mg.

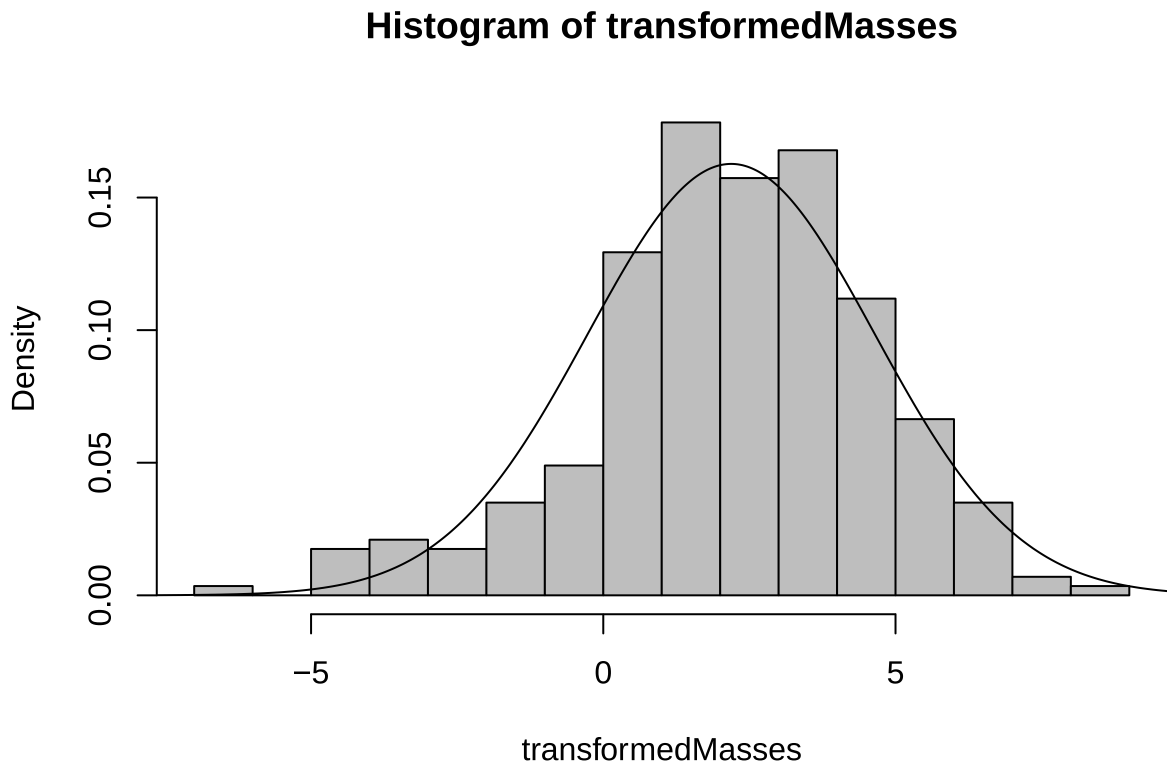
```
transformedMasses = log(ins$mass + 0.001)
ybar = mean(transformedMasses)
s = sd(transformedMasses)
```

```r
# check the new ybar and s
ybar; s
```

```
[1] 2.189
```

```
[1] 2.452
```

```r
# create a histogram of the newly transformed data
# note prob = T, for adding a density line. See ?hist.
hist(transformedMasses, col="gray", breaks=20, prob=T)

# plotting the density
# x-axis values
xval = seq(-10, 10, length=500)
# values to plot for each x-value
yval = dnorm(xval, mean = ybar, sd = s)
lines(xval, yval)
```

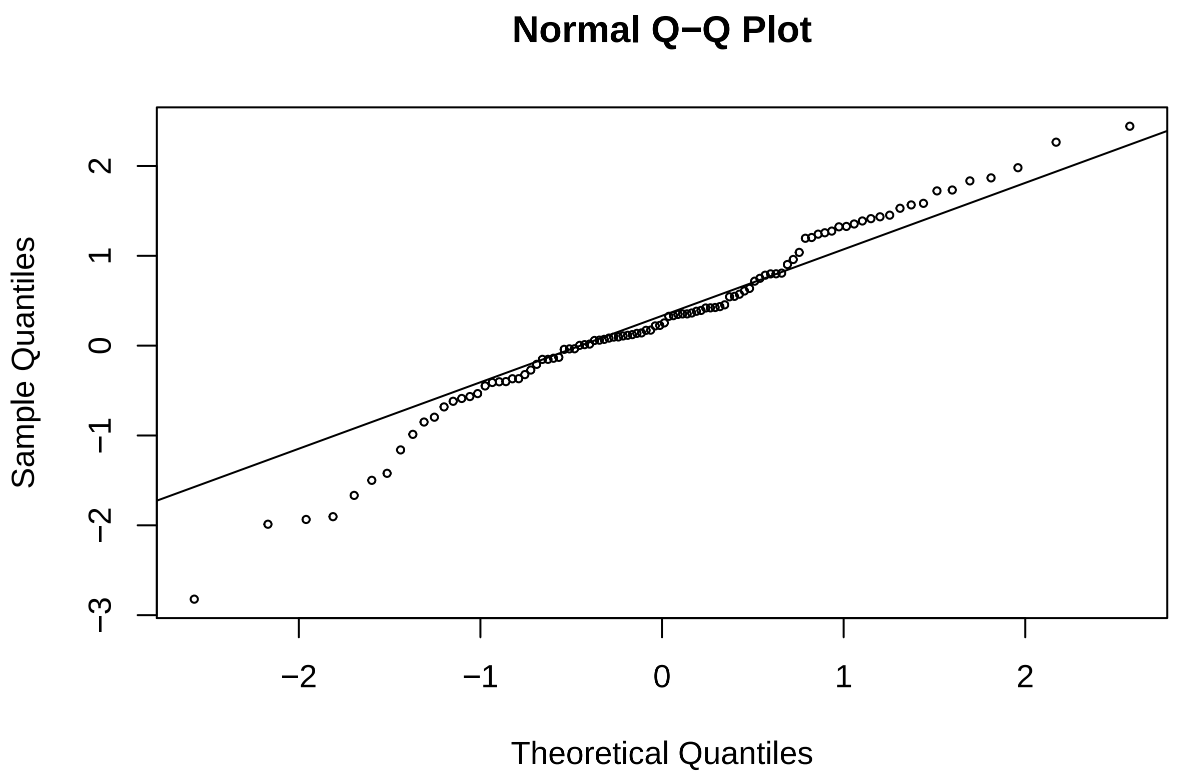


Now we can see that the fit is not terrible but not amazing, either.

The best way to judge whether a distribution follows a particular distribution is with a **quantile-quantile** (or **Q-Q**) **plot**. A Q-Q plot displays predicted/theoretical quantiles along the x-axis and observed/sample quantiles along the y-axis. If the observed distribution follows the predicted distribution, the observed quantiles will fall on a straight line in the plot.

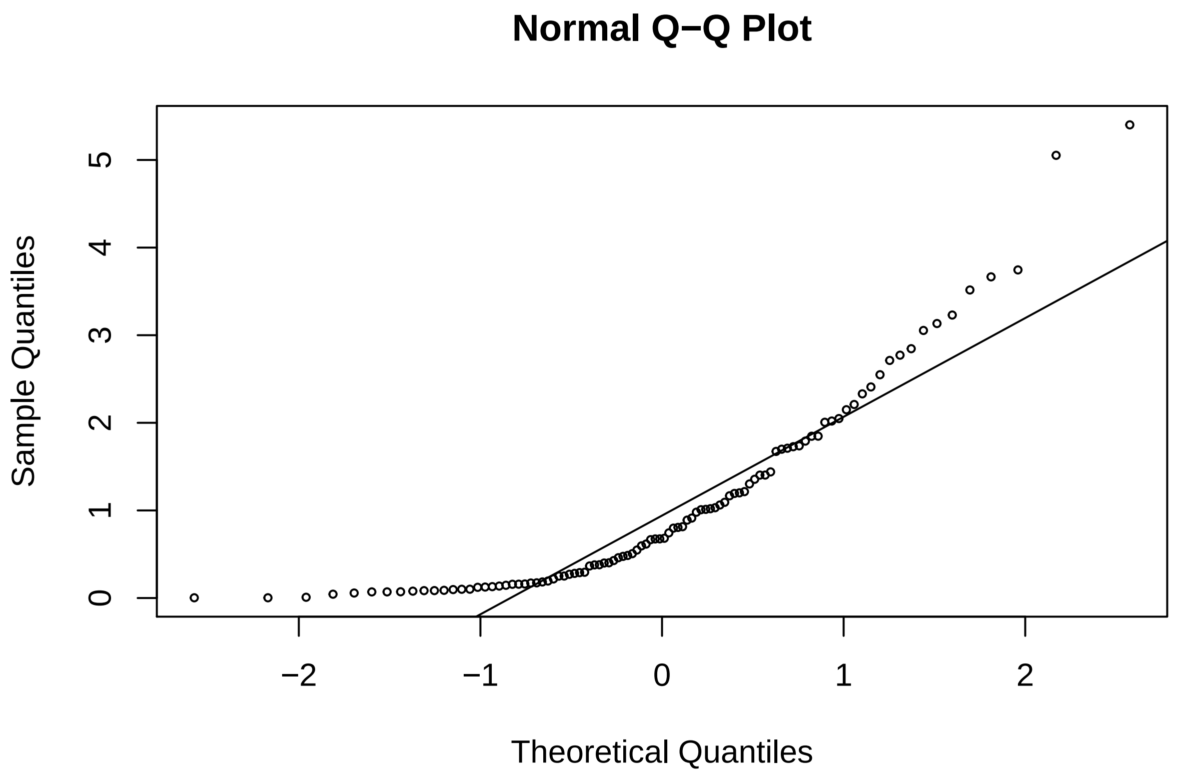
In R, the function for plotting a Normal Q-Q plot is `qqnorm()`. A straight line can be added with the `qqline()` function.

```r
# simulate N(0,1) data
normData = rnorm(100, mean = 0, sd = 1)

# generate Q-Q plot
qqnorm(normData, cex = 0.5)
# add straight line for reference
qqline(normData)
```



The simulated Normal data follow the straight line (predicted quantiles) because the underlying data is Normal. Simulating Exponential data shows how data that aren't Normally distributed don't follow a straight line in the Normal Q-Q plot:

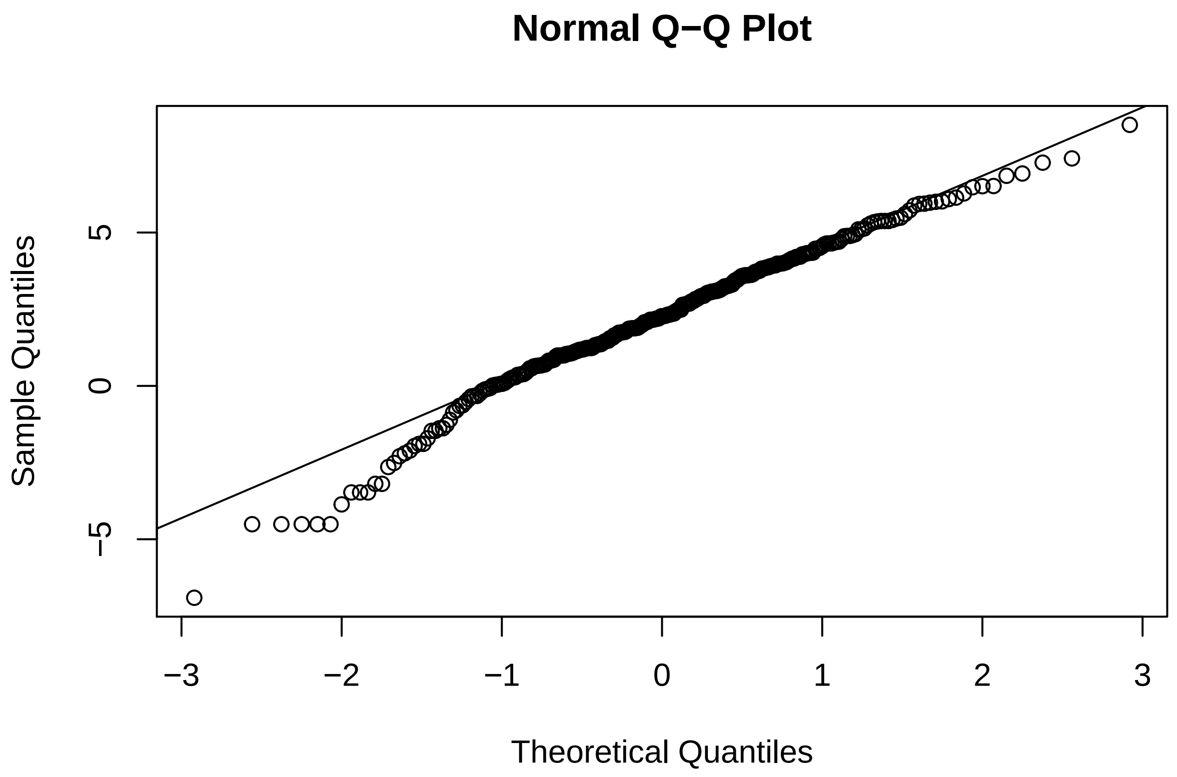
```r
# simulate Exponential data
expoData = rexp(100, rate = 1)
# generate Q-Q plot for Exponential data
qqnorm(expoData, cex = 0.5)
# add straight line for reference
qqline(expoData)
```



The Exponential quantiles "curve up," when plotted against the Normal quantiles, which indicates that the observed distribution (Exponential) is a right-skewed distribution. One way to think of this is that the lower quantiles are too close to the median (0 on the x-axis), and the upper quantiles are too far away. Thus the data are right-skewed. For more on interpretation of Q-Q plots, see here. The Whitlock and Schluter textbook also has a short section on Normal quantile-quantile plots (pp. 371-374).

Now let's produce a Q-Q plot of the transformed mass data:

```r
qqnorm(transformedMasses)
qqline(transformedMasses)
```



The data look approximately Normally distributed, but the first quantiles are too small. That is, the distribution has a heavy left tail, which we can also discern in the histogram above.

Hypothesis tests

Chi-square tests: `chisq.test()` The function to carry out a Chi-square test is `chisq.test()`. Just as there are different types of hypotheses to test with a chi-square test, there are multiple ways to call the function `chisq.test()`.

To perform a *chi-square test of independence* (a.k.a. chi-square test of association), we provide a matrix as the first argument to `chisq.test()`. The matrix represents a *contingency table* (a.k.a. cross-tabulation table), which counts the number of individuals or sampling units that fall into different categories, where each category represents either a level of a categorical variable or a binning of a continuous variable.

As an example, we will use the data from a survey given to Statistics I students at the University of Adelaide. This data frame is called `survey`, and it is in the `MASS` library, which comes bundled with R. There are many datasets available in R; to see the datasets that are available from the packages currently loaded, call `data()`.

```r
# load the library MASS
library(MASS)
# look at the structure of the survey data frame
str(survey)
```

```
'data.frame':   237 obs. of  12 variables:
 $ Sex   : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
 $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
 $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
 $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
 $ Fold  : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ...
 $ Pulse : int  92 104 87 NA 35 64 83 74 72 90 ...
 $ Clap  : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3 3 ...
 $ Exer  : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ...
 $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ...
 $ Height: num  173 178 NA 160 165 ...
 $ M.I   : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
 $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

We will test whether a person's gender is associated (or "independent") with smoking. The two variables of interest in `survey` are `survey$Sex` and `survey$Smoke`. To learn more about the `survey` dataset, call `?survey` after loading the `MASS` library. For categorical variables (factors), we can produce a contingency table using the function `table()`:

```r
# view the contingency table:
sex.vs.smoke = table(survey$Sex, survey$Smoke)
sex.vs.smoke
```

```
         Heavy Never Occas Regul
  Female     5    99     9     5
  Male       6    89    10    12
```

```r
# perform the chi-square test, store result in sex.smoke.chisq
sex.smoke.chisq = chisq.test(sex.vs.smoke)
sex.smoke.chisq
```

```

    Pearson's Chi-squared test

data:  sex.vs.smoke
X-squared = 3.554, df = 3, p-value = 0.3139
```

Here, the null hypothesis is that a person's gender is independent with how much they smoke. The alternative hypothesis is that the two variables are not independent. Here, we find that the p-value from the test is 0.3139, so we do not reject the null hypothesis.

Chi-square tests of homogeneity are calculated in the same way as *chi-square tests of independence*. (These two tests differ only in study design.)

To perform a *chi-square goodness-of-fit test*, we would specify to `chisq.test()` a numerical vector `x` containing counts (integers) of the number of observations in each of n bins, and a numerical vector `p` that contains the probabilities of observing an outcome in each of the n bins.

Note For a goodness-of-fit test, `chisq.test()` automatically assumes that the number of parameters estimated to calculate the expected frequencies under the null distribution is zero. This is not always the case, and one additional degree of freedom is lost for every parameter estimated to calculate the expected frequencies. `chisq.test()` can still be used to calculate the test statistic X^2, which can be accessed from the return value of the function. (The function `chisq.test()` actually returns a list, even though it doesn't look like it. To see what a particular function returns, see "Value" under the help page. In this case, it returns a list, and the X^2 is stored as `statistic` in the list. To get the X^2 statistic for the `sex.smoke.chisq` test result above, we would reference `sex.smoke.chisq$statistic`. See `?chisq.test`.) With the X^2 statistic, you can calculate the p-value for the χ^2 distribution with the correct degrees of freedom using `1-pchisq(xsquared, df = k)`, where you replace `xsquared` and `k` with your test statistic and degrees of freedom, respectively.

Tests of proportions: `prop.test()` A second test to be familiar with in R is a test of proportions, performed with `prop.test()`. This function tests one of two different null hypotheses:

- the proportions (probabilities of success) in several Bernoulli samples are the same, or
- the probability of success in a single sample of i.i.d. Bernoulli random variables equals a certain value.

To test whether the probabilities of success are the same in a number of samples, provide a vector `x` containing the counts of success in each sample, and a vector `n` of sample sizes. For example:

```
# simulate two binomial counts of success, both of them Binomial(n = 30, p = 0.4)
x = rbinom(2, size = 30, p = 0.4)
x
```

```
[1] 11 10
```

```
prop.test(x, c(30, 30))
```

```

    2-sample test for equality of proportions with continuity
    correction

data:  x out of c(30, 30)
X-squared = 0, df = 1, p-value = 1
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.2412  0.3079
sample estimates:
prop 1 prop 2
0.3667 0.3333
```

To test the null hypothesis that the probability of success is equal to some p, you specify a number of successes, a number of trials, and a probability:

```
x = rbinom(1, size = 20, p = 0.3)
x
```

```
[1] 6
```

```
prop.test(x, 20, 0.4)
```

```

    1-sample proportions test with continuity correction

data:  x out of 20, null probability 0.4
X-squared = 0.4688, df = 1, p-value = 0.4936
alternative hypothesis: true p is not equal to 0.4
95 percent confidence interval:
 0.1284 0.5433
sample estimates:
  p
0.3
```

See `?prop.test` to learn more.

t-tests: `t.test()` A t-test is a statistical hypothesis test to test one of three null hypotheses:

- the true mean of some Normally distributed data is μ
- two Normally distributed samples have the same mean
- the average difference between paired Normal samples is zero

The function to perform t-tests is `t.test()`. To test whether the means of two normally distributed samples `x` and `y` are the same, you would call `t.test(x,y)`. This is called a **two-sample t-test**.

```
x = rnorm(20, mean = 10, sd = 4)
y = rnorm(20, mean = 12, sd = 3)
difftest.xy = t.test(x, y)
difftest.xy
```

```

	Welch Two Sample t-test

data:  x and y
t = -2.932, df = 37.54, p-value = 0.005708
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -5.729 -1.048
sample estimates:
mean of x mean of y
    9.466    12.855
```

To test the null hypothesis that the mean of some Normally distributed data is equal to some true mean μ, you specify the data vector `x = datavec` and the mean `mu`. This is a **one-sample t-test**.

```
# simulate normal data with mean 0.2, sd = 0.4
datavec = rnorm(25, mean = 0.2, sd = 0.4)
# test null hypothesis that mean of datavec is 0
t.test(x = datavec, mu = 0)
```

```

	One Sample t-test

data:  datavec
t = 2.969, df = 24, p-value = 0.006686
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.06235 0.34683
sample estimates:
mean of x
   0.2046
```

A **paired t-test**, which tests whether the average difference between paired observations is zero, can be carried out by specifying `paired = TRUE` in the `t.test()` function. See pp. 333-335 in Whitlock and Schluter.

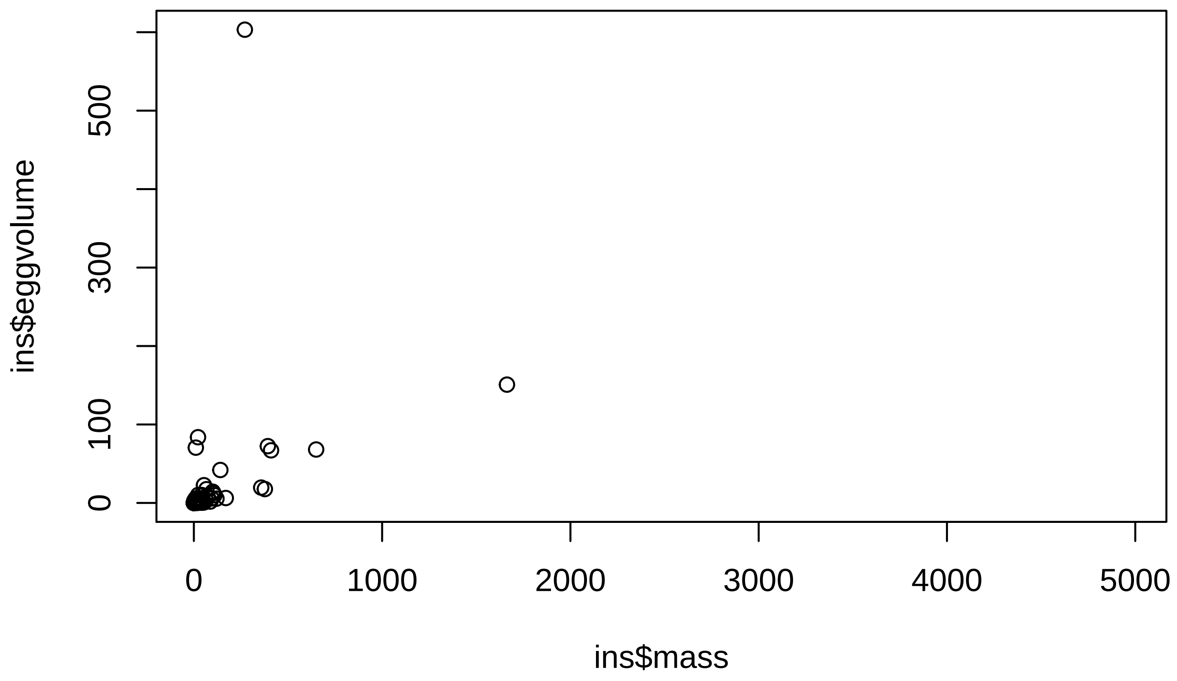
Test of association: cor.test() An **association test** (or **correlation test**) tests the null hypothesis that there is no association between two random variables. When the test statistic is a correlation coefficient, the test is called a **correlation test**.

The function to test association between two variables is `cor.test()` To take an example from the `ins` dataset, we can test the correlation between the insect mass and egg volume:

```
cor.test(ins$mass, ins$eggvolume, na.action = na.omit, method = "spearman")
```

```
Warning: Cannot compute exact p-value with ties


    Spearman's rank correlation rho

data:  ins$mass and ins$eggvolume
S = 11019, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
   rho
0.8368
```

```
plot(ins$mass, ins$eggvolume)
```



`cor.test()` allows you to specify one of three different methods, each of which has its own test statistic and null distribution. The default method of `pearson` (for Pearson's correlation coefficient) has the highest power but requires both variables to come from a Normal distribution. The other two methods (`spearman` and `kendall`) have lower power but do not require the data to be Normally distributed. (They are "non-parametric" tests, which we will learn about soon in lecture.) Here, since mass and egg volume are not Normally distributed, we used the Spearman test. See `?cor.test` for more details.

Linear regression: `lm()` Association/correlation tests test for an association between random variables. A linear regression model is a way of creating a model for predicting one variable (thought of as the dependent variable, typically y) from the values of another variable (the independent variable, x). The linear model is

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i,$$

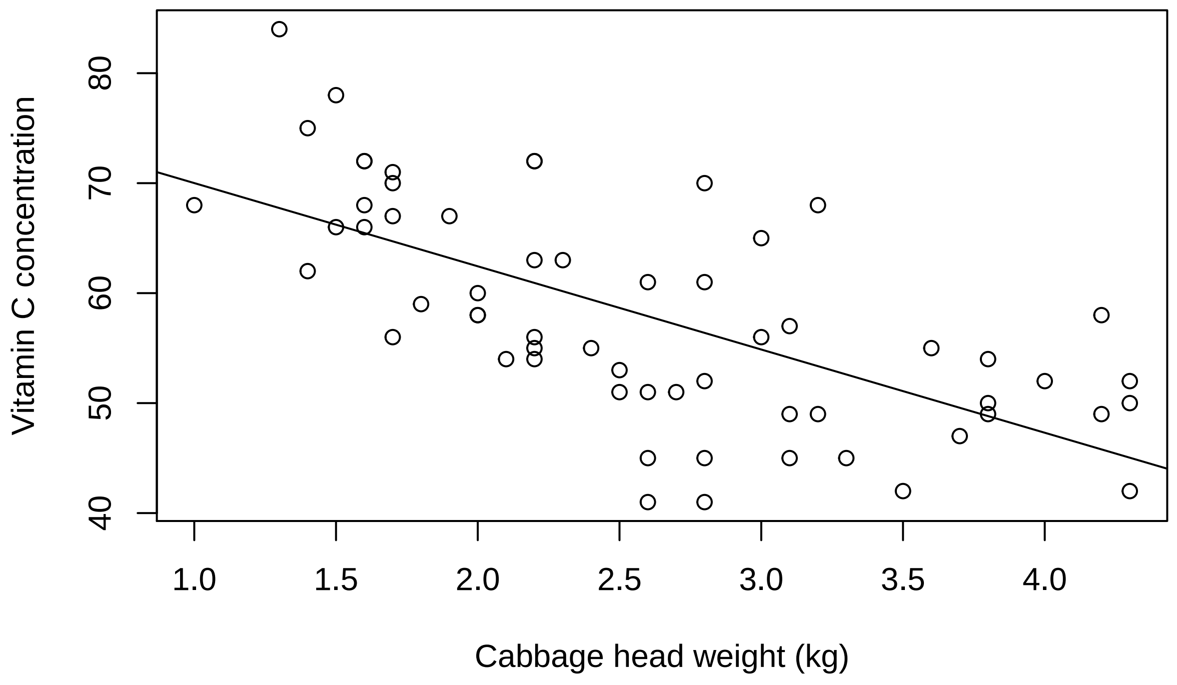
where y_i is the response/dependent variable, β_0 is a "y-intercept", β_1 is the per-unit linear effect on x (i.e., the slope of the line), and ϵ_i is a Normally distributed i.i.d. error term. The basic null hypothesis is that $\beta_1 = 0$, and the alternative hypothesis is that $\beta_1 \neq 0$.

To perform a linear regression in R, we use the `lm()` function. For an example, we will use data from the `cabbages` dataset from the `MASS` library. We will test whether a cabbage head's weight (`HeadWt`) influences its Vitamin C concentration (`VitC`).

```r
# load MASS again, just in case we didn't earlier
# (MASS contains the cabbages dataset)
library(MASS)

# create a linear model object
# (i.e., fit a regression and store the results in cabbage.reg)
cabbage.reg = lm(cabbages$VitC ~ cabbages$HeadWt)

# view a summary of the regression, including estimates, statistics, and p-values
summary(cabbage.reg)
```

```

Call:
lm(formula = cabbages$VitC ~ cabbages$HeadWt)

Residuals:
    Min      1Q  Median      3Q     Max 
-16.900  -5.251   0.357   5.019  16.263 

Coefficients:
                Estimate Std. Error t value Pr(>|t|)    
(Intercept)        77.57       3.10   25.05  < 2e-16 ***
cabbages$HeadWt    -7.57       1.13   -6.69  9.8e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.67 on 58 degrees of freedom
Multiple R-squared:  0.435, Adjusted R-squared:  0.426 
F-statistic: 44.7 on 1 and 58 DF,  p-value: 9.75e-09
```

```r
# plot the relationship to get a sense of whether or not there is a linear relationship.
plot(cabbages$HeadWt, cabbages$VitC,
     xlab = "Cabbage head weight (kg)", ylab = "Vitamin C concentration")
# add a regression line to the previous plot
abline(cabbage.reg)
```



There is a lot going on here! First, notice that we called `lm(cabbages$VitC ~ cabbages$HeadWt)`. The `~` signifies that what was provided in the call to `lm()` was a *formula object*. The formula object `y ~ x` says that `y` is the response variable and that it depends linearly on `x` (a numerical variable). This is how we specify to R that we want to fit the linear relationship $y = \beta_0 + \beta_1 x + \epsilon_i$.

This formula notation is used in all linear models in R. If `x` was a categorical variable, we would have performed an ANOVA (see below).

We stored the model object in the variable `cabbage.reg`. Since there are many things to do with a regression model *after* the model calculations have been made, it's always best to store a test result in a variable, especially for linear models.

If we had just called `cabbage.reg` after creating it, we would have seen the estimates of the coefficients β_0 and β_1. This is fine but not as informative as calling `summary(cabbage.reg)`, which provides much more information, including the p-values for the coefficients under the null hypothesis that they are zero. Other quantities, like the the t statistics used to calculate the p-values, are also provided.

Finally, we can add our fitted line to a plot of the two variables with `abline(cabbage.reg)`. Usually, you provide two parameters to `abline()`: a slope and an intercept. But if you provide a linear model, it knows to add the line given by the model.

There are many other things we can do with the `cabbage.reg` object. We can call `plot(cabbage.reg)`, which takes you through an interactive sequence of plots used to check the assumptions of linear regression, which will be discussed in lecture. Other functions allow you to use the line to predict values and calculate residuals, amongst other things. See the "See Also" section in `?lm`.

ANOVA An ANOVA (or ANalysis Of VAriance) is a linear model to test the null hypothesis that the mean is the same in each of multiple groups. The alternative hypothesis is that the mean is not the same in all groups.

The function for fitting ANOVA models in R is `aov()`. As an example, we will use data from the `InsectSprays` dataset, which is automatically loaded in R. This data represents counts of insects in plots that have been sprayed with different insecticides. The data are from `Beall, G., (1942) The Transformation of data from entomological field experiments, Biometrika, 29, 243-262.` There are six different insecticide sprays. To test the null hypothesis that each insecticide produces insect counts with the same mean, we would use the following:

```r
# perform the ANOVA
# (notice that aov has a data parameter that allows one
# to specify a data frame containing the variables)
insect.anova = aov(count ~ spray, data = InsectSprays)
summary(insect.anova)
```

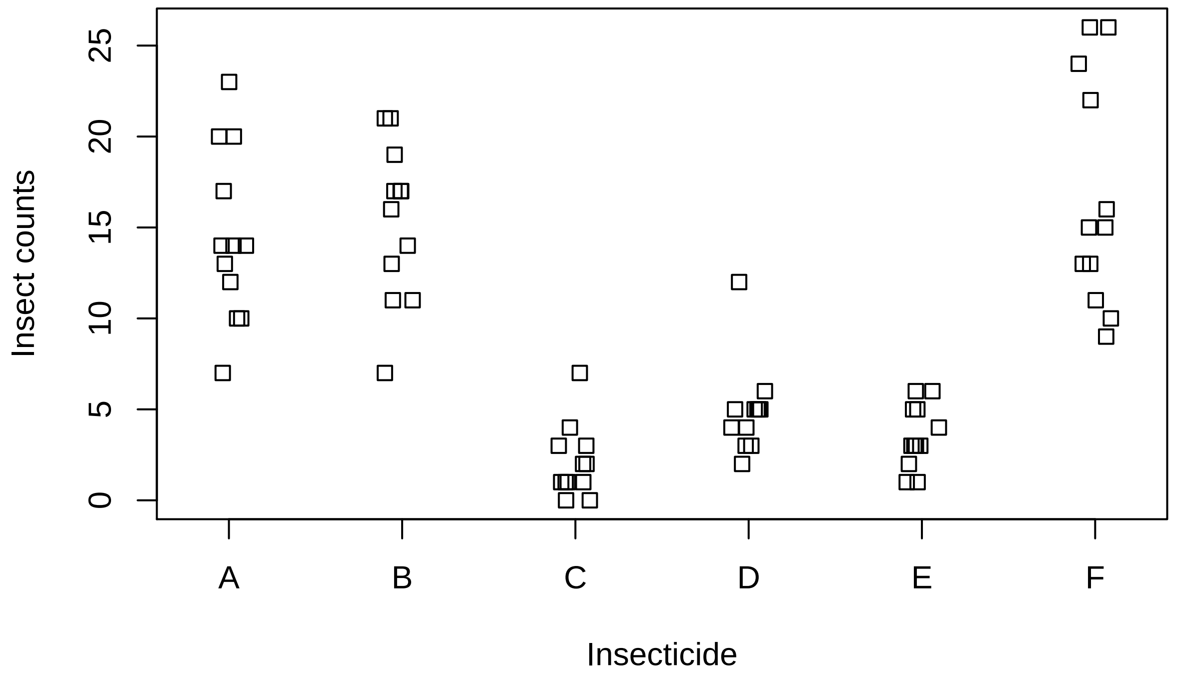
```
            Df Sum Sq Mean Sq F value Pr(>F)    
spray        5   2669     534    34.7 <2e-16 ***
Residuals   66   1015      15                   
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We call `summary()` on the fitted model object `insect.anova`, just as we did for our linear regression model. We are provided with the test statistic ($F = 34.7$), the number of degrees of freedom in the F distribution ($df = 5$), and the p-value for the null hypothesis that the means are the same for every insecticide. In this case, the p-value is very small (reported here $< 2 \cdot 10^{-16}$).

Sometimes you want to test the differences between the means of the different groups. If these tests are not planned ahead of time, one test that allows you to do this is with `TukeyHSD()`, for Tukey's Honest Significant Differences. The Whitlock and Schluter textbook has a nice section on planned and unplanned comparisons in the context of ANOVAs.

To plot the data, we can use the function `stripchart()`, which will plot the raw data.

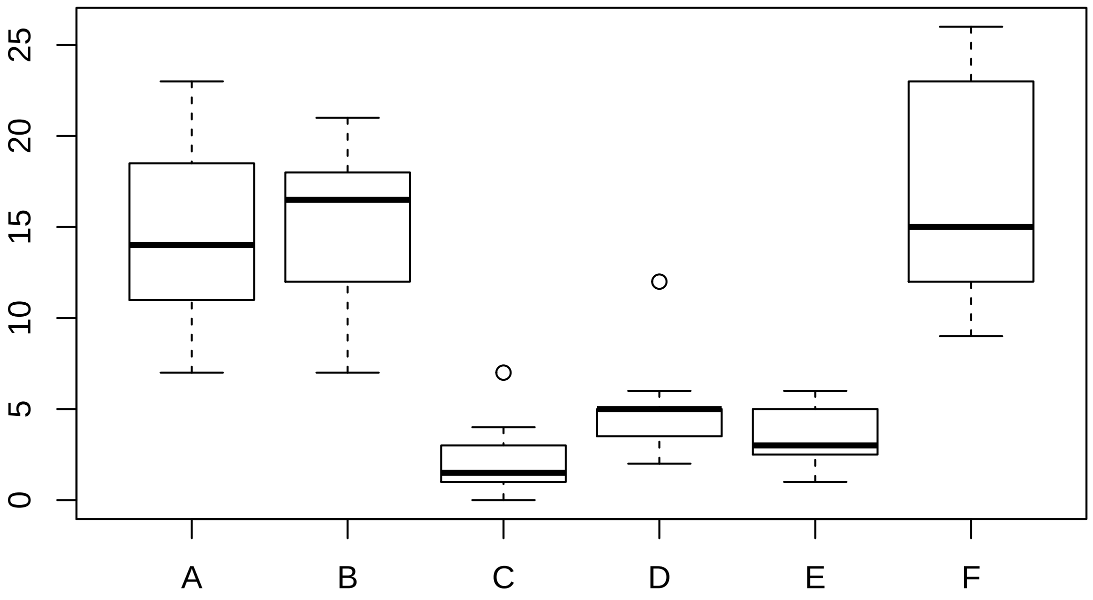
```r
stripchart(InsectSprays$count ~ InsectSprays$spray, vertical = T,
           xlab = "Insecticide", ylab = "Insect counts", method = "jitter")
```



Again, we use a formula object to tell R that we want to plot `count` as it depends on the factor `spray`. By specifying `method = jitter`, we have R "jitter" the individual points by adding random (Uniform) noise to each point so that the points aren't on top of each other. We could add means or medians to the plot with `lines()` or `segments()`.

The raw data is nice to see in this case because there's not too much of it. You can easily discern that the mean is not the same for each group. If there was a lot more data, it might be better to plot a boxplot, using `boxplot()`.

```
boxplot(count ~ spray, data = InsectSprays)
```



The boxplot plots various summary statistics of the data for each group. The thick bar is the median, the box is the "inter-quartile range," and the whiskers display an approximately 95% confidence interval for the *median*. Not every statistical computation environment produces the same boxplot, so if you use a boxplot, you always need to specify what your boxes represent.

Two-way ANOVA A two-way ANOVA allows you to test whether either or both of two (categorical) explanatory variables has an effect on a single response variable. As an example, consider the dataset `ToothGrowth`, which comes automatically loaded in R. These data represent a full-factorial experiment measuring how vitamic C dosage (`dose`, three levels: 0.5, 1, and 2 mg) and vitamin C supplementation method (`supp`, orange juice or absorbic acid) affect tooth growth in guinea pigs (represented by the variable `len`). For more info, see `?ToothGrowth`.

```
tooth.aov = aov(len ~ dose + supp + dose:supp, data = ToothGrowth)
summary(tooth.aov)
```

```
            Df Sum Sq Mean Sq F value  Pr(>F)    
dose         1   2224    2224  133.42 < 2e-16 ***
supp         1    205     205   12.32 0.00089 ***
dose:supp    1     89      89    5.33 0.02463 *  
Residuals   56    934      17                    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

With two explanatory variables, we fit a linear effect of `dose`, a linear effect of `supp`, and an interaction effect of the two variables, which is written `dose:supp` in the formula object above. The results indicate that all three terms are significant at the $\alpha = 0.05$ level. Another way to write `len ~ dose + supp + dose:supp` is `len ~ dose*supp`.

ANOVAs can be extended to more than two variables and are calulcated differently depending on how the experiment was set up. The experimental design dictates the setup of the ANOVA test in a way that will be discussed in lecture.

Note One of the aspects of an ANOVA test that varies from application to application is how the *sums of squares* are calculated. There are three different types of sums of squares calculations, (I, II, and III), and which you use will depend on how you set up your experiment. By default, R performs Type I sums of squares calculations. To get a summary of an ANOVA with type II sums of squares, you could use the `Anova()` function from the `car` package:

```r
install.packages("car")
library(car)
# apply Anova() function to aov model fit object
Anova(tooth.aov, type = 2)
```

You will learn more about sums of squares and ANOVAs in general in lecture.

Conclusion and assignment

In this lab we have briefly covered matrices and lists, two new ways of storing data. We went through some new techniques for handling data frames: creating subsets, adding and removing columns, attaching and detaching data frames. We learned several ways of assessing the Normality of a distribution, including with Q-Q plots. We also went through how to carry out several of the most commonly used statistical hypothesis tests.

Hopefully, after this lab you will more confident handling and exploring data in R. Now that we have learned all of the most common ways of storing data in R, you should feel more confident reading a function's documentation, figuring out what sort of data the function expects as arguments and understanding what sort of data the function returns.

R offers many more tests and procedures than what we have covered in lab, either as built-in ("base") functions or from third-party libraries. At this point, you should feel comfortable finding a particular test, reading its documentation, carrying out the test, and understanding its output and return values.

Assignment

For this Lab's assignment, download `lab2_assignment.R` from the course website. This R script contains commented-out questions, to which you will respond by writing *well-commented* code within the relevant sections of the script. Email me (`pwilton@fas.harvard.edu`) your script by 10am on Friday November 7.