

A Brief Introduction to R

©2020 Jeremy Balka

1 Introduction

This is a brief introduction to installing and using R, a very good (and free!) statistical computing package. There are many statistics programs that you may run into at other times in your life, such as SPSS, SAS, Minitab, or S-Plus. If you have some experience in one of them, the others are usually not too difficult to pick up.

2 Installing R

R is freely available, and can run on Windows, Mac, and Linux operating systems. (R is also installed on many of the computers around campus at the University of Guelph.) To download R, visit the main R site: <http://www.r-project.org/> (If you forget this URL, googling the letter R is usually helpful.) Click on Download R in the box near the top of the page.

The resulting page should show a list of mirrors (different websites around the world that have the exact same information). Choose a mirror site that is physically close to you.

From the top of the page, under Download and Install R, choose your operating system (Windows, Mac, or Linux).

On a Mac:

Click on the latest version of R to download (the link will look something like `R-4.0.2.pkg`, typically the first link under **Latest Release**). Save this file to your computer, then run it once it has finished downloading.

On a Windows machine:

We want to install the base distribution. Click on either the **base** or **install R for the first time** links. Then click on the link to download the latest version of R, a link at the top of the page that looks something like **Download R 4.0.2 for Windows**. Save this executable file to your computer. When it is finished downloading, run the file. This should install R on your machine.

Find the installed R software, and double click the R icon. When R opens, you should see a window with “R Console” at the top. I call this window the *commands window*.

Many people like **RStudio**, which is an “integrated development environment” for R. RStudio sits on top of R, keeping regular R as one of the windows and providing other options and help. I’ll phrase my help in terms of R, but you’re welcome to try RStudio if you wish. Everything we do in R can be done in RStudio.

3 The Console Window

The R console window is where commands are entered (I sometimes refer to it as the *commands window*).

Basic commands can be entered in the commands window. For example:

`3+2 <enter>` results in an output of 5.

```
> 3+2
[1] 5
>
```

`4+3*4+5^2 <enter>` results in an output of 41.

I usually create the solutions to assignments, quizzes, suggested answers, etc. using R. Students will have to know how to carry out some calculations using a calculator for tests and exams that are written in person, but in most other situations it’s very helpful to offload some of the calculation burden to the computer.

4 Getting Data Into R

4.1 Inputting Small Data Sets into the R Console Window

There are a variety of ways to get data into R. Let's first look at a couple of quick ways for small data sets that are easy to input manually.

First, suppose we have a sample of size $n = 5$: $-2, 4, 9, 8, 108$. The command:
`myfirstvector<-c(-2,4,9,8,108)`

concatenates or combines the numbers $-2, 4, 9, 8, 108$, saving them in an object called *myfirstvector*. (I chose the name *myfirstvector* – choose whatever name you feel is appropriate for your data. In practice, it's usually best to give an object a more meaningful name than *myfirstvector*.) If you type `objects()`, then R will list all the saved objects. *myfirstvector* should appear in the list.

If we now type the command `myfirstvector`, R should output list of our data set:
 $-2, 4, 9, 8, 108$.

```
> myfirstvector<-c(-2,4,9,8,108)
> myfirstvector
[1] -2  4  9  8 108
>
```

N.B. R is case sensitive. R considers *myfirstvector*, *Myfirstvector*, *MYFIRSTVECTOR*, etc., to be completely different names.

```
> Myfirstvector
Error: object 'Myfirstvector' not found
>
```

There are various commands that may come in handy through the course. For now:

`mean(myfirstvector)` gives the mean of *myfirstvector*.

`var(myfirstvector)` gives the variance of *myfirstvector*.

`sd(myfirstvector)` gives the standard deviation of *myfirstvector*. Alternatively, we could use `sqrt(var(myfirstvector))`.

`summary(myfirstvector)` yields the mean and the 5-number summary (minimum, first quartile, median, third quartile, maximum).

```
> mean(myfirstvector)
[1] 25.4
> sd(myfirstvector)
[1] 46.37672
>
```

All the basics are covered in R. You can probably guess some of the appropriate commands, like `max`, `min`, `sum`, `mean`, `median` etc.

```
> median(c(8,2,3,-10))
[1] 2.5
>
```

The up arrow (`↑`) comes in very handy in R. If you hit the up arrow, it brings up your previous commands one at a time. If you make a small mistake in a long command, you do not have to type it all out again; hit the up arrow to bring the previous command back into the commands window and then make the appropriate changes.

The basic operations also work on vectors. For example,

```
mysecondvector<- 10+2*myfirstvector
```

will create a second vector, called `mysecondvector`, consisting of the numbers 6, 18, 28, 26, 226.

```
> myfirstvector
[1] -2  4  9  8 108
> mysecondvector<-10+2*myfirstvector
> mysecondvector
[1]  6 18 28 26 226
>
```

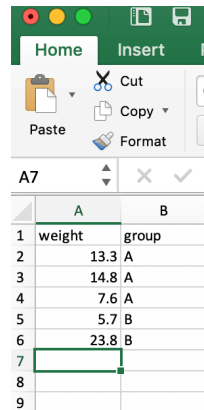
If you run into troubles, help on an R command can be obtained with the command `help(commandname)` or `?commandname` (e.g. `help(mean)` or `?mean`). Either of these commands opens a help window for the command, giving the syntax, possible options, and examples.

Many folks around the world have very helpful information about R on their websites. Internet searches for how to accomplish certain tasks in R are often very fruitful.

4.2 Importing Data into R

Most interesting data sets are a little too big for us to want to punch the data in manually, so we will often want to import data into R from a file.

We can import data from csv (comma separated values) files using the command `read.csv` (we can also use related commands, such as `read.table`). Suppose the following data set is saved in a file called *mydata.csv*:



	A	B
1	weight	group
2	13.3	A
3	14.8	A
4	7.6	A
5	5.7	B
6	23.8	B
7		
8		
9		

The command:

```
mydatainR<-read.csv(file.choose(),sep=",",header=TRUE)
```

will allow you to browse your computer for the file you wish to import (*mydata.csv*). When you find the file and double click it, the data in *mydata.csv* will be imported into R and saved in the R object *mydatainR*. (In practice it's probably best to choose a more meaningful object name than *mydatainR*.)

The `sep=","` option tells R that the data values in the file are separated by commas, as they are in a csv file. (Comma separated values are the default in `read.csv`, so the `sep=","` option can be omitted.) The `header=TRUE` option tells R that the first row represents column names, not data values. This particular data set contains column names, hence the `header=TRUE` option. But not all data sets do (some start with the data in the first row), so we must be careful. If our data set does not contain column names, then we would use `header=FALSE` option (`header=TRUE` is the default for `read.csv`). Make sure you find out whether or not the data set contains contains column names in the first row!

Since `sep=","` and `header=TRUE` are the default options, we could use the simpler command:

```
> mydatainR<-read.csv(file.choose())
>
```

Then browse to find the csv data file that you wish to import.

This is what it should look like if the data has imported properly. No errors, no warnings, seemingly nothing has happened. But something *has* happened behind the scenes — the data from the file you choose has been imported into R and saved in the R object `mydatainR`.

We should always check to see what the data in the object looks like, as it's possible you chose the wrong file, or something else happened that's different from what we were expecting.

```
> mydatainR
  weight group
1   13.3     A
2   14.8     A
3    7.6     A
4    5.7     B
5   23.8     B
>
```

which looks like what we were expecting.

Often, using the `file.choose` option and browsing for the file is the easiest option, but if you know the path to the file, that can be specified instead. For example:

```
mydatainR<-read.csv("C:/Documents and Settings/JB/My Documents/mydata.csv",header=TRUE)
```

would work if I had the file saved in the My Documents folder on my computer. Or, if the file is saved in R's *working directory*, the path can be omitted:

```
mydatainR<-read.csv("mydata.csv",header=TRUE)
```

You can find out what R is using for a working directory with the command `getwd()`.

If the above method doesn't work for you for some reason, you can try googling “importing data into R” or similar searches for some tips. R is very commonly used, and there is a great deal of online help available.

The `head` command is very useful for larger data sets, as it shows only the column names and the first few rows of data (it doesn't print out the entire data set).

Suppose we wish to find the mean of the weight variable. It might be tempting to try the command `mean(weight)`:

```
> mean(weight)
Error in mean(weight) : object 'weight' not found
>
```

We get an error here, because *R does not look within objects for variable names unless we tell it to*. Here, `weight` is not an object, but a variable within the object `mydatainR`. We must tell R to look in `mydatainR`, either by using the `$` operator, or by *attaching* the `mydatainR` object. R doesn't know of its existence, as far as the above command is concerned.

The command: `mean(mydatainR$weight)`

will tell R to find the variable `weight` in the object `mydatainR` and calculate its mean. R will output 13.04, the mean of 13.3, 14.8, 7.6, 5.7, and 23.8.

```
> mean(mydatainR$weight)
[1] 13.04
>
```

Equivalently in this case, we could have asked R for the mean of the first column in the data set: `mean(mydatainR[,1])`

```
> mean(mydatainR[,1])
[1] 13.04
>
```

Another useful option is the `attach` command. If we *attach* this data set using the command `attach(mydatainR)`, then R will know to look in this data set for variable names (if puts the object in R's search path). After attaching, we can use the simpler command `mean(weight)`, and R will find this variable.

```
> mean(weight)
Error in mean(weight) : object 'weight' not found
> attach(mydatainR)
The following object is masked _by_ '.GlobalEnv':

    group

> mean(weight)
[1] 13.04
>
```

Warning: the `attach` command can make some things a little simpler and more convenient, and I use it frequently, but it can cause potential problems. (For example, if there was also an object called `weight`, then how would we know which of the two variables called `weight` R is using? R has rules about this, but we can get fooled—we need to be a little careful.) R's “masked” warning is related to this, as `mydatainR` contains a variable called “group”, and I have an object with the same name saved

in my R.

Suppose we wanted to find the standard deviation of the just the first 3 weights. The command: `sd(mydatainR$weight[1:3])`

will tell R to look in the object `mydatainR` for the variable `weight`, and calculate the standard deviation of only the first 3 values.

5 Plots

5.1 Basic commands for histograms, boxplots, and scatterplots

Suppose again our data set is called `mydatainR`, and contains the values:

weight	group
13.3	A
14.8	A
7.6	A
5.7	B
23.8	B

`hist(mydatainR$weight)` will plot a histogram of the weights. `?hist` will show all of the options available for the `hist` command. I give a more detailed example in Section 5.2.1.

`boxplot(mydatainR$weight)` will plot a boxplot of all the weights. I give a more detailed example in Section 5.2.3. `?boxplot` will show all of the options available for the `boxplot` command.

`boxplot(mydatainR$weight~mydatainR$group)` will plot side-by-side boxplots of the weight variable for each group individually. (One box plot for the weight values in group A (13.3, 14.8, 7.6), and one boxplot for the weight values in group B (5.7, 23.8)).

Suppose that in addition to `weight`, we also had a variable called `height`, and we wished to plot a scatterplot of weight on height. `plot(height,weight)` will plot a basic scatterplot. I look at a more detailed example in Section 5.2.4. `?plot` will show the options available for the `plot` command.

5.2 Detailed examples

5.2.1 Histograms

Suppose we have a csv file called `autopsy.csv` that contains a variable named `change`, which represents % change in weight before and after autopsy for a sample of patients at a hospital.

First we import the data into R. There are a number of ways to do this, but using `read.csv` with the `file.choose` option is one of the easiest:

```
autopsy<-read.csv(file.choose())
```

then search for the appropriate file.

If we don't get an error, then the data in the file `autopsy.csv` should now be saved in the R object `autopsy`. If we use the command `head(autopsy)`, R will show us the first few rows of the data:

```
head(autopsy)
```

```
      change
1      -23
2      -13
3      -11
4      -11
5       -9
6       -8
```

Now attach the R object `autopsy`: `attach(autopsy)`

(Recall that this tells R to put `autopsy` in R's search path, so that R can find variables in `autopsy`.) Now we're ready to plot a histogram. For the most basic histogram of the variable `change` plotted with the R defaults, we can use:

```
hist(change) [Note that this will not work unless autopsy has been attached.]
```

or, equivalently: `hist(autopsy$change)`

(as this plots the variable `change` contained in the R object `autopsy`). The result is contained in Figure 1

Now that's with the *defaults*, but we can tweak the command to suit our needs. There

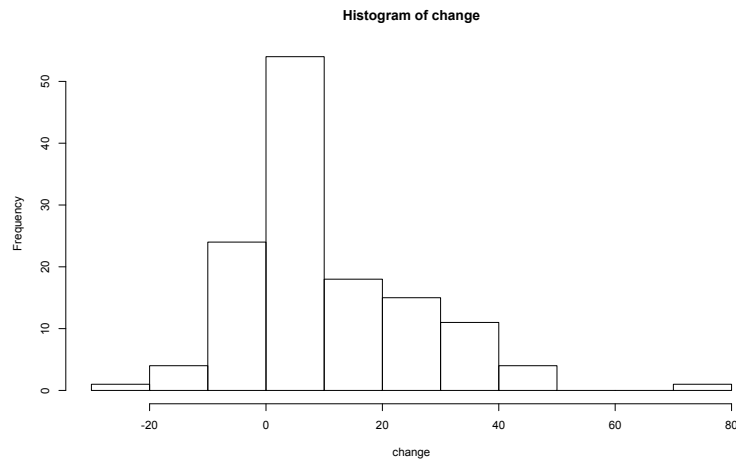


Figure 1: Percent change in weight before and after autopsy for 132 autopsy cases.

are many options available in the `hist` command (see `help(hist)` for full details). Some of the common ones that we'll need:

- `xlab` adds an x -axis label.
- `ylab` adds a y -axis label.
- `main` adds a title.
- `nclass` changes the number of classes (bins) in the histogram. (R generally does a pretty good job of choosing the number of classes automatically. Some fine minds have put thought into this issue over the years.)
- `col` changes the colour.

The size of the title, axis labels, etc. can be changed with the `cex` options.

- `cex.main` controls the size of the title.
- `cex.lab` controls the size of the axis labels.
- `cex.axis` controls the size of the axis numbering.

`cex=1` is the default, with larger values of `cex` increasing the size, and lower values decreasing it. The `cex.main = 2` option would double the size of the title, `cex.main = 0.5` cuts it in half.

The following is simply *an example* of things *we might do*, used to illustrate some of the commands involved. *Don't blindly follow what I do here.*

```
hist(change,xlab="Percent change in body weight",ylab="Frequency",
```

```
main="Histogram of Percent Change in Body Weight at Autopsy",
cex.lab=1.5,nclass=20,col="green")
```

This results are given in Figure 2.

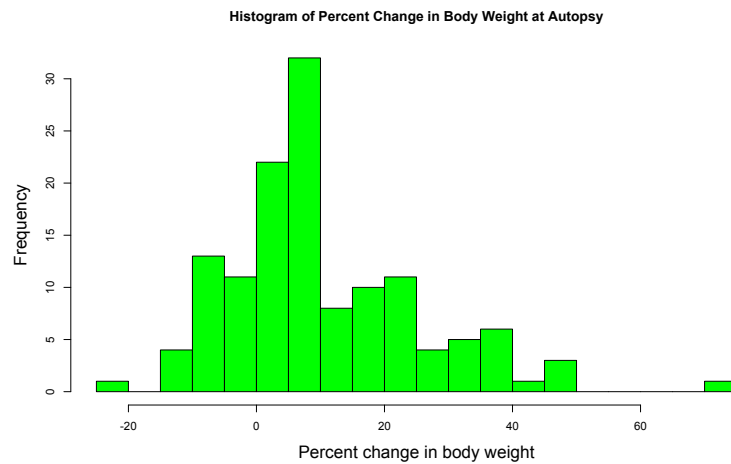


Figure 2: Don't include both a caption and a title. If you include a caption, there's no need for a title and vice-versa.

5.2.2 Bar Charts

Suppose I want to draw a simple bar chart to represent the following data, which represents the number of days out of the last 30 my overall vibe for the day fell into these 3 categories. This one's made up, obv.

Overall vibe	Ugh	Meh	Not too shabby
Count	8	16	6

Table 1: Count of daily vibe over the past 30 days.

One way to do this would be to use R's `barplot` command, but first create an object with the counts:

```
count<-c(8,16,6)
barplot(count)
```

which results in Figure 4.

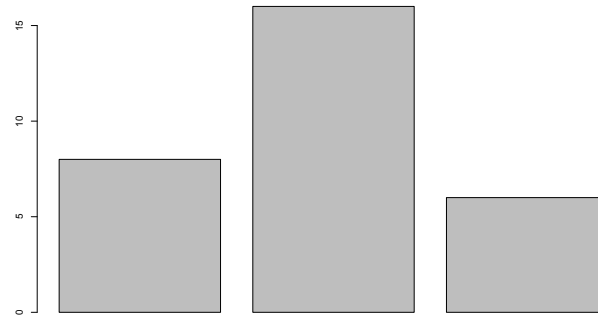


Figure 3: A naked (and not very informative) bar chart.

But this isn't very good, as we don't have appropriate labels. So let's put in appropriate labels and change a few other things (some of these options, like the colour, as just to illustrate what *can* be done, and not something I'm recommending).

```
barplot(count,names=c("Ugh","Meh","Not too shabby"),xlab="Daily Vibe",ylab="Frequency",
cex.names=1.2,cex.lab=1.5,col="aquamarine")
```

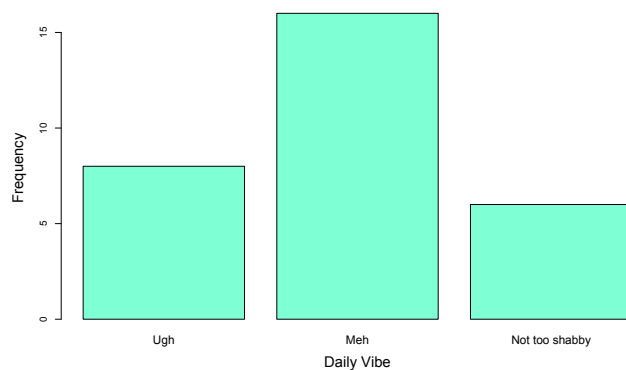


Figure 4: Better labels, at least.

5.2.3 Boxplots

[Hand et al., 1994] report butterfat percentages for milk from 5 different breeds of cow (Original source: [Sokal and Rohlf, 1981]). Suppose we want to plot side-by-side boxplots for this data, contained in a file called `milkfat.csv`. We first need to import the data.

```
cows_butterfat<-read.csv(file.choose())
```

then search for `milkfat.csv`. This will import the data and save it in the R object `cows_butterfat`. R's `head` command shows the first few rows of the data set, and gives us a quick view of what the data looks like.

```
head(cows_butterfat)
  Butterfat  Breed
1    4.44 Ayrshire
2    4.37 Ayrshire
3    4.25 Ayrshire
4    3.71 Ayrshire
5    4.08 Ayrshire
6    3.90 Ayrshire
```

Here the R object `cows_butterfat` contains variables called `Butterfat` and `Breed` (the butterfat percentages for the 5 different breeds) and we want to plot side-by-side boxplots for each breed.

Recall that the `attach` command puts the object in R's search path, so that R can find variables within the object.

```
attach(cows_butterfat)
```

Using the default plot: `boxplot(Butterfat~Breed)`, we would end up with Figure 6. *Note that the quantitative variable being plotted is on the left of the tilde, and the grouping variable is on the right.*

That's with the *defaults*, but we can tweak the command to suit our needs. There are many options available in the `boxplot` command (see `help(boxplot)` for full details). Some of the common ones that we'll need:

- `xlab` adds an *x*-axis label.
- `ylab` adds a *y*-axis label.
- `main` adds a title.
- `names` adds group names.

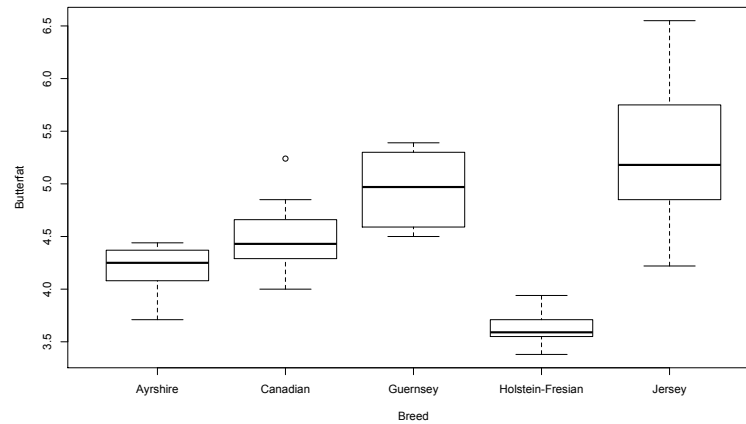


Figure 5: Butterfat percentages for milk from 6 different cow breeds ($n = 9$ for each breed).

- `col` changes the colour.

The size of the title, axis labels, etc. can be changed with the `cex` options.

- `cex.main` controls the size of the title.
- `cex.lab` controls the size of the axis labels.
- `cex.axis` controls the size of the axis numbering.

`cex=1` is the default, with larger values of `cex` increasing the size, and lower values decreasing it. The `cex.main = 2` option would double the size of the title, `cex.main = 0.5` cuts it in half.

The following is simply *an example* of things *we might do*, used to illustrate some of the options involved.

```
boxplot(Butterfat~Breed,ylab="BUTTERFAT, YO",
names=c("Breed 1", "Breed sqrt(4)", "Third one","Last one?","No, really, last one"),
col="pink",cex.axis=1.5,main="Not a great title but it's not bad",cex.main=2)
```

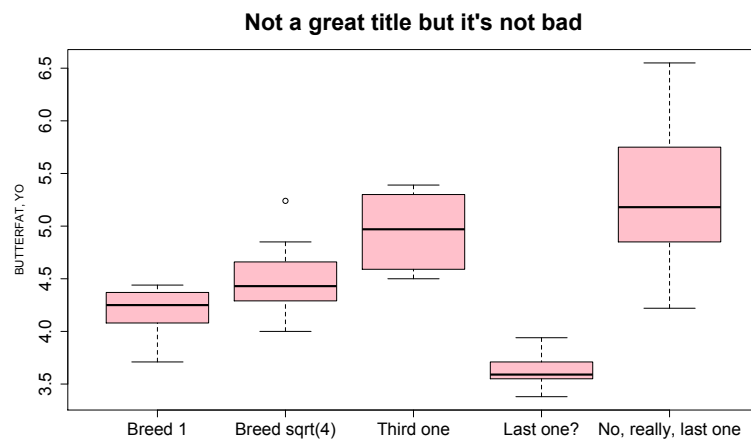


Figure 6: Just showing some of the options available.

5.2.4 Scatterplots

Basic Scatterplot

Appel et al. (2017) investigated nighttime bat activity for several species of bat. In one part of the study, the authors investigated a possible relationship between moonlight intensity and count of the number of passes (recorded by automatic sensors). They looked at several species, but here we'll look only at the results for *Pteronotus parnellii*.

Import the data into R: `bat_activity <- read.csv(file.choose())`. Check to see what the data looks like after the import: `head(bat_activity)`

	Moonlight	Count
[1,]	2	7
[2,]	4	11
[3,]	9	36
[4,]	14	4
[5,]	17	8
[6,]	18	19

After attaching `bat_activity`, we can create a basic scatterplot with `Count` on the y axis and `Moonlight` on the x axis with the command `plot(Moonlight, Count)`. The results are illustrated in Figure 7.

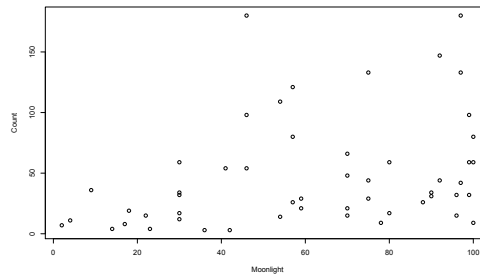


Figure 7: Number of passes vs moonlight % for *P. parnellii*.

That's with the *default* options, but we can tweak them to suit our needs. There are many options available in the `plot` command (see `help(plot)` for full details). Some of the common ones that we'll need:

- `xlab` adds an x -axis label.
- `ylab` adds a y -axis label.

- `main` adds a title.
- `pch` controls the plotting symbol. (For symbol options in R, try `help(pch)`. `pch=1` is an open circle, `pch=15` is a filled-in square, `pch=16` is a filled-in circle.)
- `col` changes the colour of the plotting symbols.

The following is simply *an example* of things *we might do*, used to illustrate some of the options involved.

```
plot(Moonlight,Count,xlab="Moonlight %",
     ,ylab="Number of Passes",cex.lab=1.6,pch=16,cex=2,col="blue",cex.axis=1.5)
```

The results are given in Figure 8.

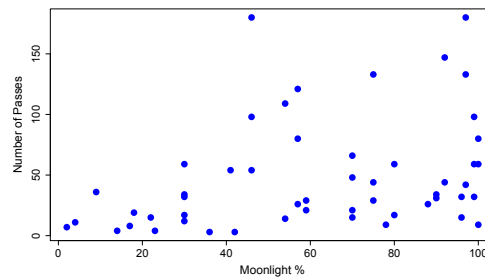


Figure 8: Number of passes vs moonlight % for *P. parnellii*.

6 Calculating Binomial and Poisson Probabilities in R

It is often convenient to have R calculate probabilities for different probability distributions. Here are some common commands for the binomial and Poisson distributions:

`dbinom(x,n,p)` yields $P(X = x)$ for a binomial distribution. For example, `dbinom(3,10,.5)` yields 0.1171875.

`pbinom(x,n,p)` yields $P(X \leq x)$ for a binomial distribution. For example, `pbinom(3,10,.5)` yields 0.171875.

`dbinom` gives the probability that the random variable is exactly equal to x , `pbinom` gives the probability it is less than or equal to x . That is, `pbinom` yields the *cumulative distribution function*.

`dpois(x,lambda)` yields $P(X = x)$ for a Poisson distribution. For example, `dpois(3,5)` yields 0.1403739.

`ppois(x,lambda)` yields $P(X \leq x)$ for a Poisson distribution. For example, `ppois(3,5)` yields 0.2650259.

Similar commands exist for other discrete probability distributions such as the geometric and hypergeometric.

7 Useful R Commands for the Normal Distribution and t Distribution

A few helpful commands in R:

- `pnorm(z)` yields the area to the left of z under the standard normal curve. For example, `pnorm(-1.96)` yields the value 0.02499790, the area to the left of -1.96 under the standard normal curve.
- `pnorm(x,mu,sigma)` yields the area to the left of x under a normal curve with $\mu = \text{mu}$ and $\sigma = \text{sigma}$. For example, `pnorm(2.5,2,1.8)` yields 0.6094085, which is the area to the left of 2.5 under a normal curve with a mean of 2 and standard deviation of 1.8.
- `qnorm(p)` gives the value of z such that the area to the left of z under a standard normal curve is p . For example, `qnorm(0.02499790)` yields -1.96 . Note that `qnorm`

and `pnorm` are inverse functions, so `qnorm(pnorm(1.96))` yields 1.96.

- `qnorm(p, mu, sigma)` yields the value of x such that the area to the left of x under a normal curve with a mean of `mu` and standard deviation of `sigma`. For example, `qnorm(.43,12,24)` would yield the 43rd percentile of a normal distribution with a mean of 12 and a standard deviation of 24.
- `dnorm(x, mu, sigma)` yields the probability density function (height of the curve at point x) for a normal distribution with a mean of `mu` and standard deviation of `sigma`. This command is often used when plotting the normal distribution pdf.
- `rnorm(n,mu,sigma)` randomly generates n values from a normal distribution with a mean of `mu` and a standard deviation of `sigma`. This type of command is often used in my simulation functions.
- `pt(x,df)` yields the area to the left of x under a t distribution with `df` degrees of freedom. For example, `pt(-1.96,5)` yields the value 0.05364398, the area to the left of -1.96 under a t distribution with 5 degrees of freedom.
- `qt(p,df)` yields the value x such that the area to the left of x under a t distribution with `df` degrees of freedom is p . For example, `qt(0.975,23)` yields 2.068658. Note that `qt` and `pt` are inverse functions, so `qt(pt(1.96,5),5)` yields 1.96.

Similar commands exist for other continuous probability distributions, e.g. `pf` and `qf` for the F distribution, `pchisq` and `qchisq` for the χ^2 distribution.

8 One-sample t Procedures

The command:

```
t.test(mydatainR$weight,mu=10)
```

tests the null hypothesis that the true mean of the weight variable is 10. It also outputs a 95% confidence interval for the true mean of the weight variable. The default is a two-sided alternative hypothesis, and the default confidence level is 95%. These options can be changed (try `?t.test` to see the options available).

9 Two-sample t Procedures

Suppose again our data set is called `mydatainR`, and contains the values:

```
weight group
13.3      A
14.8      A
 7.6      A
 5.7      B
23.8      B
```

One of the easiest ways to carry out these procedures is to first use the `attach(mydatainR)` command, to put this data set into R's search path. If this `attach` command is used first, then:

`t.test(weight~group)` will test the null hypothesis that $\mu_A = \mu_B$ and output a 95% confidence interval for $\mu_A - \mu_B$. (For the grouping variable, R chooses the name that comes first alphabetically to be Group 1. Be careful! Make sure you know which group R is using as Group 1 in the calculations!)

The default is the Welch (unpooled variance) method. This can be changed with the `var.equal` option:

`t.test(weight~group,var.equal=TRUE)` uses the pooled-variance procedure (which assumes the population variances are equal).

`t.test(weight~group,var.equal=FALSE)` uses the Welch procedure (which does not assume the population variances are equal).

There are other options, such as a one-sided alternative, and different confidence levels (use `?t.test` or `help(t.test)` for details).

If we did not use the `attach` command, then we would need to tell R to look in the data set `mydatainR`:

`t.test(mydatainR$weight~mydatainR$group)` will test the null hypothesis that $\mu_A = \mu_B$ and output a 95% confidence interval for $\mu_A - \mu_B$.

Another option that achieves the same results:

```
weight_A<-mydatainR$weight[1:3] assigns the weights in group A to an object called
weight_A
weight_B<-mydatainR$weight[4:5] assigns the weights in group B to an object called
```

`weight_B`

`t.test(weight_A,weight_B)` would yield the same output as `t.test(weight~group)`

References

- [Hand et al., 1994] Hand, D., Daly, F., Lunn, A., McConway, K., and Ostrowsky, E. (1994). *A Handbook of Small Data Sets*. Chapman and Hall.
- [Sokal and Rohlf, 1981] Sokal, R. and Rohlf, F. (1981). *Biometry*. W.H. Freeman, San Francisco.