## Using multivariate data to answer biological questions

## Preamble

This is a manuscript about answering biological questions using multivariate data. It assumes good knowledge of R, such as that would come from reading and working through the book [Getting Started With R, An Introduction for Biologists](http://www.r4all.org), by Beckerman and Petchey. While reading this manuscript you will come across phrases like “All the biological questions we so far addressed…” – here we are referring to what appears in the book. Otherwise you should be able to work through the following material with little trouble (we hope).

All of the code used in the following material, answers to the exercises, and other bits and bobs of code are available on the web page associated with this manuscript (http://www.r4all.org/BR/FreeMulti/index.html). The same goes for the skull dataset used in all the examples.

The material below is drawn from many sources, including:

* Pielou, E.C. (1984) The interpretation of ecological data: a primer on classification and ordination. Wiley, New York, USA.
* Quinn, G.P. & Keough, M.J. (2002) Experimental Design and Data Analysis for Biologists. Cambridge University Press, Cambridge, UK.
* Documentation associated with the excellent vegan() R package, by Jari Oksanen and others.

## Introduction

All of the biological questions we so far addressed were univariate, in the sense that analysis of only one response variable was enough. Many kinds of biological question involve multiple response variables, however, and these are multivariate problems. They require multivariate data, methods for visualizing multivariate data, and methods for analyzing it. This chapter covers the three issues.

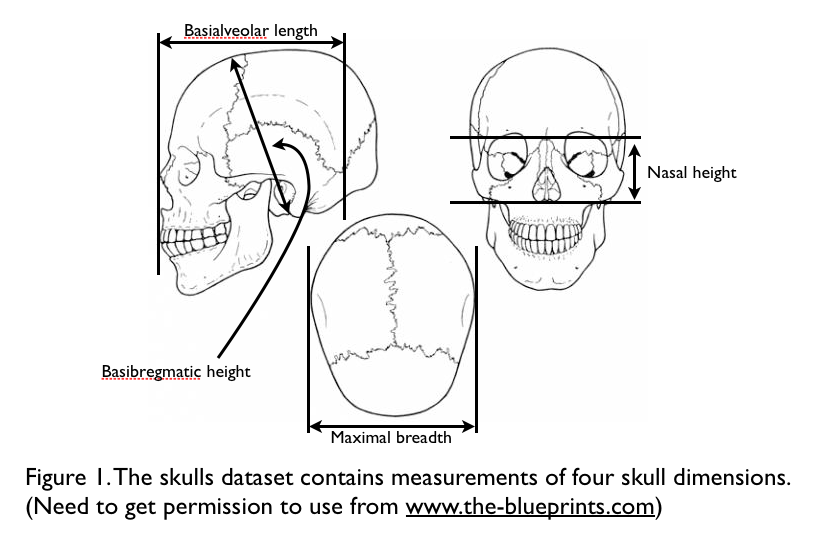
Consider the question of how human skull shape has changed during the last few millennia. If we measured any single dimension of a skull, for example, maximum breadth, this question about shape would be beyond us. We might find that maximum breadth had increased, but this could be associated with increases in every measurement. Such a finding would indicate that skulls had increased in size, but not changed in shape. But with only one measurement, we could not know.

The same applies for the question of how environmental temperature relates to community composition, though we have to be careful about what we mean by community composition. If we define community composition as the abundances of each of the species in a community, then clearly this is a multivariate question. We cannot address such a question if we measure the abundance of only one species. We need to measure the abundances of all the species in the community that interest us, and then explore and analyse this multivariate dataset.

What other types of question are intrinsically multivariate? Questions about environmental effects on life history evolution are multivariate if we choose to interest ourselves simultaneously in multiple life history traits (e.g., behavioural, morphological, and phenological traits). Question about variation in genetic composition among individuals, and variation in variation in gene expression can be multivariate questions, since organisms have many genes. Make a list of other biological questions that require measurement and analysis of multiple response variables.

Consider the example questions above, and note that all questions are about if variability in response variables is associated with variability in explanatory variables. Make sure the questions in your list are of the same nature. Then note that your explanatory and response variables could come from an observational or experimental study. Hence, there seems no need for us to think that univariate methods are better for analyzing experiments, and multivariate for analyzing observational studies. What is important, as always, is careful formulation of specific hypotheses and predictions, such that one does not become involved in a fishing expedition (looking for patterns, with few or no expectations).

Safe in the knowledge that questions exist that require multivariate data and associated methods, let us proceed practically, with the question of whether and how human skull shape has changed over the last few millennia. The dataset we will analyse contains measurements of four skull dimensions (figure 1). We also have, in a separate dataset an estimate of era from when the skulls came.



We will use this dataset to illustrate and discuss the following:

* Importing multivariate data.
* Manipulating multivariate data in R (e.g., switching between “wide” and “long” format).
* Merging two datasets
* Data exploration / visualization.
* Using a loop to graph multiple response variables automatically.
* Rotating data clouds (i.e., ordination).
* Hypothesis testing multivariate data.

Unfortunately, this material covers only a fraction of all the available and possible types of multivariate analysis. Furthermore, with one simple question and example datasets, we give only one illustration of the methods we do cover. This is because our intention is to give you enough information to feel confident that you have the basics in hand, and to then leap into the world of multivariate methods, learning what you need as you need it.

## Human skull shape

First, and as usual, read in to R the dataset skull.dimensions.csv. Examine and check the structure of the data frame (dd in the code snippets below) to which you assigned the dataset. You will probably see this:

> str(dd)

'data.frame': 150 obs. of 5 variables:

$ id : Factor w/ 150 levels "S1","S10","S100",..: 1 63 74 85 ...

$ max.breadth : int 120 113 104 111 116 118 123 128 114 110 ...

$ basi.height : int 66 67 74 73 63 66 69 65 73 77 ...

$ basi.length : int 65 74 74 66 77 72 67 76 80 82 ...

$ nasal.height: int 25 28 30 35 30 32 31 27 29 29 ...

We can see, among other things, that the first column of the data is “id”: a code that uniquely identifies each skull. This isn’t such a good result. Progress will be much smoother if the data frame contains only the dimensions of the skulls. Smoother because many of the functions we will use assume that the row names of the data frame are the unique identifiers. We can efficiently assign to the row names the “id” column by adding an extra argument to read.csv():

dd <-read.csv("~/Dropbox/R4All\_Share/modules/newMV/skull.dimensions.csv",  
 row.names="id")

This, obviously, tells read.csv that the row names should be taken from the column names “id”. After reading in the data with this code the structure of the data is:

> str(dd)

'data.frame': 150 obs. of 4 variables:

$ max.breadth : int 120 113 104 111 116 118 123 128 114 110 ...

$ basi.height : int 66 67 74 73 63 66 69 65 73 77 ...

$ basi.length : int 65 74 74 66 77 72 67 76 80 82 ...

$ nasal.height: int 25 28 30 35 30 32 31 27 29 29 ...

And use head() to see the first few lines of the data we see:

> head(dd)

max.breadth basi.height basi.length nasal.height

S1 120 66 65 25

S2 113 67 74 28

S3 104 74 74 30

S4 111 73 66 35

S5 116 63 77 30

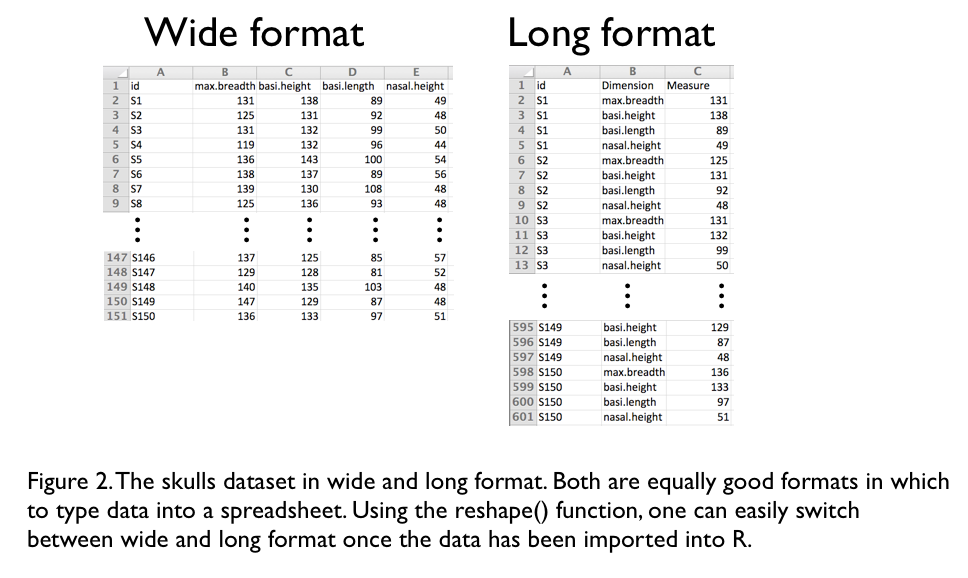
S6 118 66 72 32

There is no id column. Instead, the unique identifier codes (e.g., S1, S2, S3) are the row names of the data frame. You can confirm this to yourself with:

rownames(dd)

Now we have the data read into R in a manner that will make life easier. This is always a bonus. There are 150 skulls (observations) and four variables, all of which are integer (int) variables (they contain only whole numbers, because all measurements are to the nearest millimeter and recorded in millimeters).

The meanings of the four variables are given in figure 1. At this point it would be a good idea to get a summary of the data, at least to check for typos.



## Wide or long data?

The skulls data is arranged in what is often called *wide format* in the csv spreadsheet, since this is how the researcher decided to type in the data. Wide format is so called because each row contains multiple observations (measurements) of an object (plant, person, stream, skull), each of which has its own column. Lots of columns make for a wide looking dataset. The skulls dataset in wide format has 150 rows and five columns (it is not particularly wide) (figure 2). Many of the R multivariate visualization and analysis functions require data in wide format.

We have previously advised, however, that data be put into spreadsheets in what is often called *long format*. Long format is so called because each row contains only one observation. In long format, there are columns that specify the object (e.g., the skull ID column), that specify the dimensions measured (e.g., basilar height, or maximum width), and that contain the value of that measure (e.g., 135mm). In long format the skulls dataset has 600 rows and 3 columns (figure 2). Data in long format is somewhat easier, in R, to perform calculations on, with the aggregate() or tapply() functions, for example.

Do not worry, however, about whether you should make a spreadsheet with your data in wide or long format. It is easy to switch between wide and long format after your data is imported into R. (Good advice is to enter the data into the spreadsheet in the format it was originally collected. Whether you prefer to collect and record the original data onto a piece of paper [or electronic device] in wide or long format is a matter of personal preference and convenience.)

Switching between wide and long format in R uses the reshape() function. Here’s how reshape() is used to switch the skulls data from wide to long format. First, it is best to read the data without putting the skull ids into the row names, but rather to leave them in their column:

> dd <- read.csv("skull.dimensions.csv")

> dd

id max.breadth basi.height basi.length nasal.height

1 S1 120 66 65 25

2 S2 113 67 74 28

3 S3 104 74 74 30

4 S4 111 73 66 35

5 S5 116 63 77 30

6 S6 118 66 72 32

7 S7 123 69 67 31

8 S8 128 65 76 27

9 S9 114 73 80 29

10 S10 110 77 82 29

Then we use the reshape() function:

dd.long <- reshape(dd,  
 idvar="id",  
 v.names="Measure",  
 timevar="Dimension",  
 times=colnames(dd[,2:5]),  
 varying=colnames(dd[,2:5]),  
 direction="long")

It’s a bit of a beast, and the R help file for reshape() is, one might say, “difficult”. Table 1 explains the meaning of each of arguments used above. The result of using reshape() is the object dd.long. The rownames of dd.long are a bit nasty, but you should by now know how to set them to the numbers 1:600, for example.

Table 1. The arguments used to reshape the skulls dataset from wide to long format.

|  |  |
| --- | --- |
| Argument | Meaning |
| dd | The dataset we want to reshape. |
| idvar="id" | The name of the column in both wide and long format that contains the skull identity. |
| v.names="Measure" | The name of the column in long format that will contain the measurements (e.g., 135mm). |
| timevar="Dimension" | The name of the column in long format that will contain the name of the dimension measured (e.g., basilar height). |
| times=colnames(dd[,2:5]) | The names of the dimensions that were measured. I.e., that values that will go into the timevar (“Dimension”) column. |
| varying=colnames(dd[,2:5]) | The variables in wide format that will be made into a single variable in long format. |
| direction="long" | That we want reshape to make a long format dataset from a wide format one. |

And now, just for laughs, we switch the skulls data set from long to wide format. First read in the long format version of the dataset:

> dd <- read.csv("skull.dimensions.long.csv")

> dd[order(dd$id),]

id Dimension Measure

1 S1 max.breadth 120

151 S1 basi.height 66

301 S1 basi.length 65

451 S1 nasal.height 25

10 S10 max.breadth 110

160 S10 basi.height 77

310 S10 basi.length 82

460 S10 nasal.height 29

100 S100 max.breadth 143

250 S100 basi.height 75

400 S100 basi.length 87

550 S100 nasal.height 40

Then use the reshape function:

reshape(dd,  
 idvar="id",  
 v.names="Measure",  
 timevar="Dimension",  
 direction="wide")

When you run this, you’ll see that the column names of the variables are not so nice (e.g., Measure.max.breadth). You can easily rename the column names, so as to remove the rather redundant “Measure.”, with an instruction to R something like:

colnames(dd.wide)[2:5] <- substr(colnames(dd.wide)[2:5], 9,  
 nchar(colnames(dd.wide)[2:5]))

We realise that our explanation of how to use the reshape() function is quite specific to the skulls dataset. When you come to use rehape(), you will probably need to experiment, read the help file, ask a friend, battle a bit, or some combination of these. But please persevere. Do not, under any circumstances, make the switch in a spreadsheet application (e.g., Excel), or make two versions of your spreadsheet, one in long and one in wide format (we did only for didactic purposes).

Another option for switching in R between long and wide format is a couple of functions in a package called reshape (unfortunately the same word as the reshape() function). In the reshape package two functions, melt() and cast(), do the switch. We’ll let you discover the details of melt() and cast() yourself, but here’s the general idea. The melt() function melts the data down into its constituent parts and the cast() function puts it back together into a specific format. One could say that melt() liquefies the data, and it solidifies in the cast(). The cast function is quite versatile, and can be used in place of aggregate(). We’ve never put the time into learning how to use melt() or cast(), so go no further here.

## Merging two datasets

Recall that we are interested in if and how human skull shape has changed through time. We have the dimensions of 150 uniquely identified skulls in R. Now we need to read in a dataset containing the year that each of the skulls was dated to.

Perhaps you ask yourself why the dates were not put in the same dataset (spreadsheet) as the dimensions data. We are imagining that we have measured the skull’s dimensions, and these data we entered into a spreadsheet. To get the dates, imagine we sent small samples to a radiocarbon dating lab, which returned to us a spreadsheet containing the estimated date of each skull, according to its unique identifier. It is our opinion that as little as possible data manipulation be done outside of R. Therefore, we import into R the dimensions dataset, and then read in the date data. We do not put the two datasets together outside of R. While it is not too difficult to do this in, for example, excel, it is even easier to do in R. Furthermore, if done in R, it is automatically documented in the R script; if do outside of R, there need be no record of what was done to merge the two datasets.

Here is the top of the date data:

> yy <- read.csv("skull.date.csv")

> yy

id thousand.years

1 S1 888

2 S2 112

3 S3 854

4 S4 171

5 S5 544

6 S6 207

7 S7 1488

8 S8 365

9 S9 368

10 S10 405

11 S11 1207

12 S12 511

Also, read in the dimensions data again, just to be sure we’re at the same stage:

mm <- read.csv("skull.dimensions.csv")

Great, now we have two datasets read into R. Notice that they share a variable, “id”: the skull identifier variable. In order to merge, we must have a variable that is shared between the two datasets. Before merging, it is worth checking that all the skulls in the dimensions dataset occur in the dates dataset, and vice versa. A quick way to do this is to sort the variables and see how many times they are then equal:

sum(sort(yy$id)==sort(mm$id))

This should return 150, or however many objects are in your datasets. If this code returns something other than the number of objects in your dataset, you have a problem (though not necessarily a particularly bad one).

If the number is zero, check for a typo in this line of code. If you don’t have one, probably there is some different between the way identity is coded in the two datasets, for example a difference in case:

> "s1"=="S1"

[1] FALSE

If you get a number somewhere between 0 and 150 (or however many objects you have), there are some skulls that appear in the dimensions dataset, but not in the year dataset, or vice versa.

The match() function is very useful to find the objects (identities) that occur in one dataset and not another.

match(mm$id, yy$id)

This asks for the position (index) in yy$id, in which each of the values of mm$id occurs. If a value in mm$id doesn’t occur at all in yy$id, then an NA is returned. So we can ask for a TRUE if a value in mm$id doesn’t occur at all in yy$id with:

is.na(match(mm$id, yy$id))

In our data, this will return FALSE 150 times, since all identities match between the two datasets. One last step for troubleshooting. If there are some TRUES, you can use these to find out which identities appear in mm$id but not yy$id:

yy$id[is.na(match(mm$id, yy$id))]

Swap yy$id and mm$id in order to check for identities in the year dataset that do not appear in the measurement dataset.

With those important preliminaries out of the way, we are ready to merge the datasets:

dd <- merge(mm, yy, by="id")

Running str(dd) reports that dd has 150 rows and six variables. Great, this is what we expected.

Note that merge() will happily proceed regardless of values that appear in one dataset but not the other. We can simulate this situation by removing the first row of one of the datasets, performing the merge, then asking for the structure of the data:

> dd <- merge(mm[-1,], yy, by="id")

> str(dd)

'data.frame': 149 obs. of 6 variables:

$ id : Factor w/ 150 levels "S1","S10","S100",..: 2 3 4 5 ...

$ max.breadth : int 110 143 140 135 134 140 143 124 145 125 ...

$ basi.height : int 77 75 73 84 81 85 78 93 91 95 ...

$ basi.length : int 82 87 90 89 89 84 92 89 82 81 ...

$ nasal.height : int 29 40 43 38 41 38 35 44 32 50 ...

$ thousand.years: int 405 1262 988 1264 796 1315 513 1362 1089 1376 ...

No error (or warning) was given, and there are 149 rows in the merge dataset dd. (Don’t be fooled by the 150 levels in the id variable. There are 150 possible identities, but only 149 of are used.)

An additional argument to merge will keep all the records:

dd <- merge(mm[-1,], yy, by="id", all=T)

If you decide to let merge() deal with mismatches, you should definitely know what mismatches are occurring, and why. They could be due to a typo, and therefore be easily fixed. Using merge uncritically would result in the loss of a possibly very hard-won data point. Better to know exactly what merge() is doing. This includes using str() after merge(), in order to make sure merge() is behaving as expected. As always, love R, but don’t trust it.

The merge() function is quite flexible… one can specify to merge by different columns in the two datasets. There are also ways of merging by the values of two columns in each of the datasets, e.g., merge by individual and date (in a study of individuals observed at multiple times).

## Exploring the data – version 1

As usual, one of the best ways to explore the data is to plot some graphs, preferably showing as close to the raw data as possible. And to plot graphs that relate as closely as possible to the biological question. We are interested in how skull shape changes through time, so graphs of skull measurement versus time, with each data point corresponding to a skull are quite appropriate. Make yourself some graphs, both for date as a continuous variable, and with categorized date.

If the data were in long format, we could use xyplot() to make a separate plot of each measurement. However, we only have the merged data in wide format, and rather than change it to long format to plot it, we will plot each of the four graphs separately. First, lets make a list of nice variable names, which we can use on the graphs:

nice.names <- c("Skull ID",  
 "Maximum breadth",  
 "Basilar height",  
 "Basilar length",  
 "Nasal height")

Then we make a 2x2 layout for the graphs to be made in:

layout(matrix(1:4, 2, 2, byrow=T))

Then we make each of the four graphs, using the plot() function for each one:

plot(dd[,2] ~ thousand.years, dd, main=nice.names[2],

xlab="Thousand years ago", ylab=nice.names[2])

plot(dd[,3] ~ thousand.years, dd, main=nice.names[3],

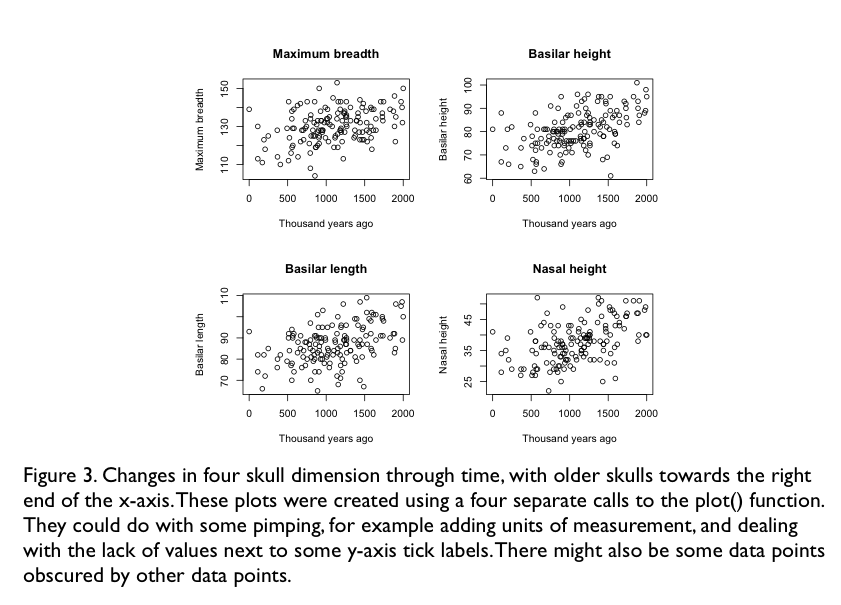
xlab="Thousand years ago", ylab=nice.names[3])

plot(dd[,4] ~ thousand.years, dd, main=nice.names[4],

xlab="Thousand years ago", ylab=nice.names[4])

plot(dd[,5] ~ thousand.years, dd, main=nice.names[5],

xlab="Thousand years ago", ylab=nice.names[5])



The results of this can be seen in figure 3. We have used numbers to refer to the response variable (y variable), and also to the entry in “nice names” that corresponds with the response variable used. Seeing this arrangement of numbers (i.e., 2s in the first call to plot, 3s in the second, 4s in the third, and 5s in the fifth) might prompt you to wonder if there is a way to automate the production of the graphs. Yes, there is, using a loop. Here, we wish to make a loop that goes round four times, and the first time insert a 2 in the plot command, the second time a 3, and so on. Here is the definition of the loop and the call to plot():

for(i in 2:5) {

plot(dd[,i]~dd[,"thousand.years"], main=nice.names[i],

xlab="Year", ylab=nice.names[i])

}

The first line says “please go round a loop once for every value in 2:5, and let *i* be that value”. The first time round the loop *i*=2, the second time *i*=3, the third time *i*=4, and the fourth time *i*=5. The curlies “{ }” embrace all of the code that will be repeated every time round the loop. And you can see that the letter *i* is where before were the numbers 2 thru 5.

What about our question? Looking at figure 3, it is clear that skull shape was larger further into the past (towards the right, i.e., 2000 thousand years ago). Put another way, skulls have shrank from the past to the present.

So far we know that skulls have generally decreased in size through time (figure 3). But what about changes in shape? If one of the dimensions changed at a different rate to another, then the skulls will have changed in shape. Perhaps they got somewhat taller than they got wider. Furthermore, if one variable increases while another decreases, there will be considerable changes in shape (though we can see from figure 3 that this does not happen).

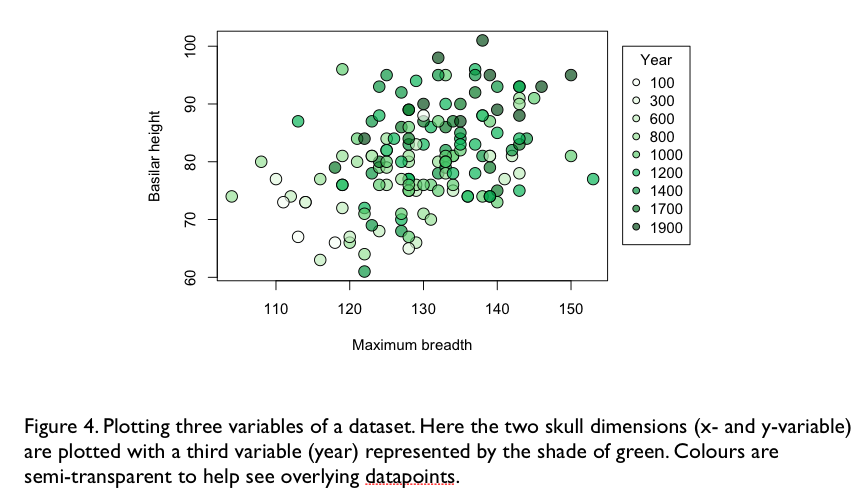
We can imagine other biological situations in which general changes in size might obscure perhaps more interesting changes in shape. And since organisms very often differ greatly in size, the challenge is often to detect changes in shape, after removing the changes in size.

The same goes for variables other than dimensions. The cuticle of ants contains hydrocarbons used in communication among individuals. Chromatography of the mixture of hydrocarbons might find that ants differ in the general level of cuticular hydrocarbons (i.e., some individual ants smell strongly due to high levels of all hydrocarbons), but we might be more interested in variation in the relative amounts of different hydrocarbons. The relative levels of different hydrocarbons would then describe how the ants smell (i.e., their cuticular hydrocarbon signature), rather than how much they smell.

Other multivariate datasets might concern variables about life history characteristics (e.g., age at maturity, size at maturity, number of offspring, number of mates) and levels of expression of genes (e.g., RNA expression levels). A classical example of multivariate data is species composition: the abundances of each of potential many species in an ecological community. Questions concerning variation in community composition are often multivariate questions.

The size and shape example applies equally to all the others mentioned above. We might find some ecological communities in which the abundance of all species is high, and some in which all species are rare. More interesting would probably be changes in the relative abundances of species. Did some species become relatively less common than others? It is likely, however, that high abundances of some species is associated with low abundance of others.

In the skulls dataset there were only four response variables. In the next section, we will attempt to visualize variation in these variables, and we will find that even this problematic. We will then use some new methods that can be used to visualise variation in tens or 100s of variables (for example the abundance of each of the species in a community).



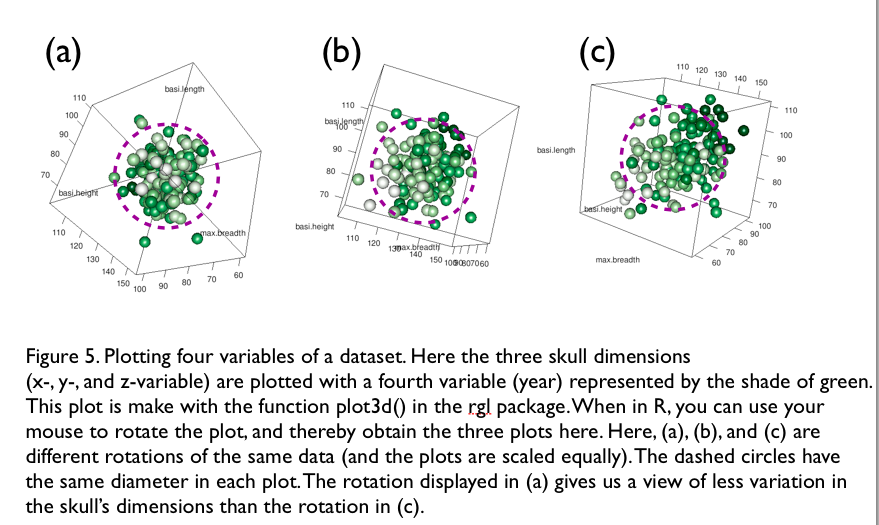
## Visualising three, four, or more variables

Lets say that we’d like to visualize how two of the skull dimensions change through time. One way to do this is to make a bivariate plot of the two dimensions, and to colour the points according to the time (year) they came from. (One could, alternatively, choose to different plotting symbols to indicate different years, or different sizes for the symbols.) Box 1 describes, in some detail, how to plot such a figure (i.e., figure 4).

We can also make such a plot of the skulls in three dimensions, and then colour the points by year. Three dimensional plots can be tricky to make clearly interpretable for readers. One way to increase their value is to allow them to be rotated, so the 3-D cloud of data points can be viewed from different angles. This kind of graph is quite easy to produce in R using the rgl package. If you don’t have it, please now install the package. Then try this (you have to do the preliminaries in Box 1, or just don’t include the cols argument):

plot3d(dd[,2:4], type="s", cex=0.5,

col=col.map$col[match(dd[,7], col.map$year)])



You should get a pretty 3-D plot similar to that in figure 5. Rotating the datacloud should give you the feeling that you’re looking at a cloud of points that are shaped somewhat like a rugby ball / American football. From some viewpoints (i.e., with the data rotated into one particular orientation) we are looking down the length of the rugby ball, and we see relatively little of the variation in the data (i.e., the ball looks small) (figure 5a). From a different viewpoint, we see the rugby ball side on, and it looks bigger (figure 5c), i.e., we see more of the variation among the skulls. (One might say that this 3-D plot is actually 4-D, since we coloured the points by year.)

All this is very nice, but we still have another skull dimension to plot. That is, we have a 5-D dataset. While some recreational pharmaceuticals may allow 5-D visualization, it seems unlikely that many reviewers and readers will want to use such visualization methods. In any case, we may often have many more the 5-D data, for example if we want to understand how some environmental variables affects the composition of communities that contain tens of species, or if we are examining variation in the expression levels of tens or hundreds of genes.

Fortunately, often we may have tens of variables, but due to correlations among the variables, the *effective dimensionality* of the data cloud is quite low. By ‘effective dimensionality’ we mean the smallest number of dimensions required to display a large amount of variation in the data.

Imagine that all four skull dimensions were very strongly correlated. The data cloud would look more like a thin cigar than a fat rugby ball. A thin cigar is quite close to being a line, i.e., a one dimensional object. In this case we need only one dimension to display the majority of the variation in the data.

What shape would be the data cloud if two of the four variables were strongly correlated, as were the other two, but neither of the pairs were correlated with each other. Just to be clear, we are talking about a correlation matrix like this one:

1.0, 0.8, 0.2, 0.2,

0.8, 1.0, 0.2, 0.2,

0.2, 0.2, 1.0, 0.8,

0.2, 0.2, 0.8, 1.0

The data cloud would be shaped like a pancake; two dimensions would be required to display the majority of the variation in the data.

The take home message is that correlations (positive or negative) among our variables can help us, since they reduce the number of dimensions required to see the majority of the variation in the data. And in many biological situations (e.g., morphology, community composition, gene expression, life history traits) correlations among variables are frequent and strong.

Hopefully you now have in you mind that we can rotate data clouds to see more or less variation (figure 5) and that we might need relatively few dimensions to display quite a lot of the variation. Putting these two observations together, we might decide that we’d like to rotate the data so that we can see the greatest amount of variation in the fewest number of dimensions. This is exactly what Principle Components Analysis (PCA) does for us: it rotates the dataset so the greatest amount of variation is displayed in the fewest number of dimensions.

Lets do a PCA of the skulls data, and learn about PCA as we do. The function prcomp() can be used to do a PCA:

pp <- prcomp(dd[,2:5], center=TRUE, scale=TRUE)

The first argument is the data to be rotated. The second and third inform prcomp() that we would like the data to be centered and scaled before the PCA is done. Centering sets the mean to zero, and scaling sets the standard deviation to 1, for each of the variable. More later about when we scale and when we don’t. (Rather than rely on prcomp() to do the centering and scaling for us, we could have used the function scale(), or even done it manually, by first subtracting the minimum and then dividing by the standard deviation.)

Before we look at the output of the PCA, prcomp(), know that PCA rotates the data so we can see the most variation in the fewest dimensions. In doing so PCA gives each of the data points a new position (i.e., set of x, y, z, etc, values). But each of the data points still has four dimensions (the new dimensions created by the PC – often these are called the principle component axes). Therefore, the output of the skulls data will still be *150 skulls each with four values* (i.e., a four dimensional dataset), but, as discussed above, the new dimensions will show us the greatest amount of variation in the fewest number of dimensions.

The result of the PCA is saved into the object we called pp, which we can see the contents of:

> str(pp)

List of 5

$ sdev : num [1:4] 1.492 0.898 0.764 0.62

$ rotation: num [1:4, 1:4] 0.436 0.536 0.44 0.573 -0.689 ...

..- attr(\*, "dimnames")=List of 2

.. ..$ : chr [1:4] "max.breadth" "basi.height" "basi.length" "nasal.height"

.. ..$ : chr [1:4] "PC1" "PC2" "PC3" "PC4"

$ center : Named num [1:4] 130.2 80.8 86.9 37.6

..- attr(\*, "names")= chr [1:4] "max.breadth" "basi.height" "basi.length" "nasal.height"

$ scale : Named num [1:4] 9 8.14 9.12 6.62

..- attr(\*, "names")= chr [1:4] "max.breadth" "basi.height" "basi.length" "nasal.height"

$ x : num [1:150, 1:4] -3.622 -2.216 0.447 0.574 0.574 ...

..- attr(\*, "dimnames")=List of 2

.. ..$ : NULL

.. ..$ : chr [1:4] "PC1" "PC2" "PC3" "PC4"

- attr(\*, "class")= chr "prcomp"

Ugh. That’s rather a nasty mess, but take it line by line. The first line tells us that pp is a “List of 5”, i.e., pp is a list of 5 things (objects). The first thing is called “sdev”, the second “rotation”, the third “center”, the fourth “scale”, and the last “x”. Taking each of these in turn:

**sdev**: is a vector of four standard deviations, the first being the standard deviation of the first principle component axis (the first new dimension of the rotation). There are four standard deviations, one for each of the new variables (i.e., axes, dimensions). And there are four new variables because PCA produces a dataset with the same number of variables as it is given to work on.

**center**: is the number substracted from each variable to center it. To centre the data on zero, we subtract the mean of each variable, from each variable. Check that the values that have been subtracted:

pp$centre

are the same as the means:

apply(dd[,2:5], 2, mean)

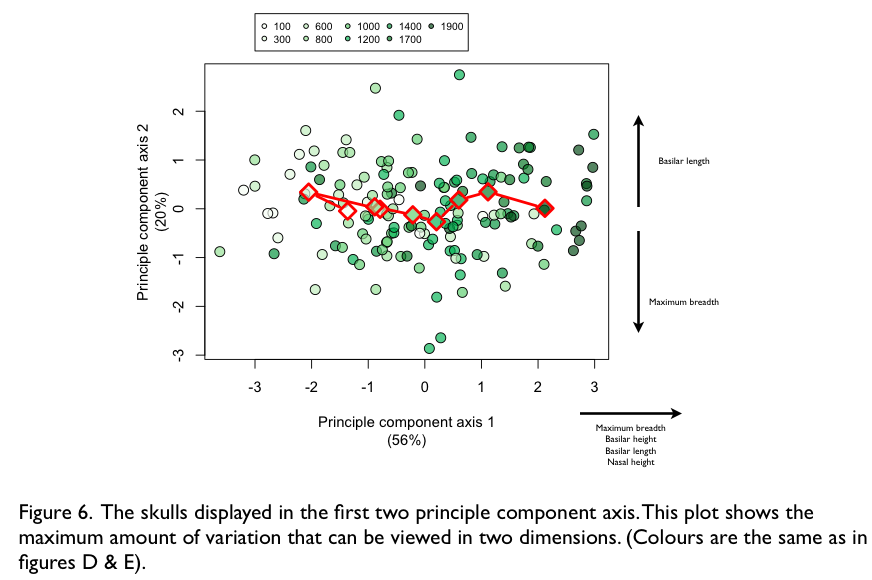
**scale**: is the number that the variables are divided by to set the standard deviations to 1. Check that these values:

pp$scale

are equal to the standard deviation of the variables.

apply(dd[,2:5], 2, sd)

**x**: the last object in the list given to us by prcomp is a matrix of the new variables. These are the new coordinates of the skull, after the rotation has taken place. Now, if we plot the first two of the new variables against each other, we are viewing the maximum amount of variability that can be viewed in two dimensions (figure 6). We can clearly see that skulls get older from left to right.



A few questions might spring into your head:

* How much variability is explained in the first two axes?
* What is the biological interpretation of each of these axes?

R happily tells us the variation explained in each axes:

> summary(pp)

Importance of components:

PC1 PC2 PC3 PC4

Standard deviation 1.4916 0.8981 0.7643 0.62007

Proportion of Variance 0.5562 0.2016 0.1460 0.09612

Cumulative Proportion 0.5562 0.7578 0.9039 1.00000

The first two axes together explain 75.8% of the variation in the data. If there raw data were entirely random, we would expect each axes to explain 25% of the variability, and the first two together to explain 50%. So the PCA has helped us visualize the variability. And it has been able to help because the raw variables we gave to the PCA were quite strongly correlated (recall the rugby ball shaped data cloud).

Now we need to know what is the biological meaning (interpretation of the new axes). We can find out by asking R to give us the correlations between the raw variables and the new principle component variables:

> round(cor(dd[,2:5], pp$x),2)

PC1 PC2 PC3 PC4

max.breadth 0.65 -0.62 -0.43 0.07

basi.height 0.80 -0.10 0.49 0.33

basi.length 0.66 0.64 -0.36 0.17

nasal.height 0.86 0.07 0.15 -0.49

The first new axis (PC1) is strongly and positively correlated with all four original variables (look at the first column of correlations). The second new axis (PC2) is negatively correlated with max.breadth, and positively correlated with basi.length. This means that large values of PC2 are related to large values of basi.length, and small values of max.breadth.

All of this information, i.e., the amounts of variance explained and the meaning of the axes, can be efficiently displayed on one graph; figure 6 gives an example of how. This figure also plots the means of the year classes that we created for the colouring of the points (Box 1). If you prefer, you can use the function biplot() to make the same plot but with the information displayed in a different way. Have a go.

That gives you enough information about PCA to be dangerous. There is lots more to know. Some is explained in Box 2, including how to decide whether to scale and or center data, and how to do a PCA by hand (well, at least using the R function eigen() ).

## Ordination – NMDS

There are many situations in which PCA will not suffice, however. For example if your data contains any non-numeric variables (like binary, ordered categorical, or categorical variables). Another situation in which PCA is less useful is analysis of variation along a gradient, along which variables increase in value and then decrease (for example species abundances along an elevational or fertility gradient). With such data, PCA tends to reveal an “arch” shaped data cloud, where the two ends can be quite close together, despite the datapoints at the ends coming from the two ends of the gradient. One reason for this result is that PCA performs a linear transformation of the data: PCA rotates the data cloud, but does not change its shape.

So, if you can’t use PCA, or just don’t want to, what are your options? Unfortunately there are many, with lots of complicated names that can be rather confusing. Feel free to immerse yourself in the diversity and wonder of the many types of ordination. Alternatively, see if NMDS (nonmetric multidimensional scaling) can provide you with the answer to your biological question (You do remember your question, right? ☺ Don’t let the statistics take over your brain.)

There are good reasons to go with NMDS. First, it is quite flexible, in that it can work with variables of any type, or even mix of types. Second, it has been quite thoroughly tested and found to be quite robust. To find out exactly what this means, you could look at the original paper *An evaluation of the relative robustness of techniques for ecological ordination*, by Peter R. Minchin, published in Vegatio, 1987, volume 69, page 89-107. It is a lovely read, really. Third, and finally, although NMDS has a rather scary name, it is in fact quite simple to understand. Here is how it works.

The first step should be to look at the distribution of your data. If any of the variables have very skewed distributions, think about transforming them. If there any univariate or multivariate outliers, think about what might be causing them (a typo during data entry, perhaps), and whether to exclude them. (See Box 5 for some information about identifying univariate and multivariate outliers.)

Second, we calculate all of the pairwise distances among our datapoints. In the skulls dataset we have 150 data points (skulls). Calculate for yourself how many unique pairwise distances there are among 150 data points. There are many different distance measures that one could use, such as Euclidean distance, Bray-Curtis dissimilarity, Gower distance (Box 3 contains more information about distance measures and how to choose one).

Third, we pass the distances to the NMDS, along with the number of dimensions (variables) that the new data will occupy. This is different from PCA, where the same number of variables is produced by the PCA as are given to it. With NMDS, we tell it the number of variables it can use. Often people ask for the new data to occupy two dimensions, and therefore two variables are returned. We will continue by assuming we have asked for two dimensions, so picture in your head, a blank piece of paper on which the datapoints (skulls) will be placed.

Fourth, NMDS places the datapoints (150 skulls) on the piece of paper, and assesses how closely their new pairwise distances on the piece of paper match the pairwise distances of the original dataset. NMDS then moves the datapoints a little, and checks again how closely the new distances match those of the original data. NMDS continues to move the datapoints on the paper until it cannot make the match between the new distances and those of the original data any better (this is known as *convergence*, since the match has converged on a particular value).

That is NMDS done (though we obviously need to look at the result). Lets follow this workflow for the skulls dataset, going back to the point just after we have merged the date and dimensions datasets, to create the data frame called dd:

> str(dd)

'data.frame': 150 obs. of 6 variables:

$ id : Factor w/ 150 levels "S1","S10","S100",..: 2 3 4 5 6...

$ max.breadth : int 110 143 140 135 134 140 143 124 145 125 ...

$ basi.height : int 77 75 73 84 81 85 78 93 91 95 ...

$ basi.length : int 82 87 90 89 89 84 92 89 82 81 ...

$ nasal.height : int 29 40 43 38 41 38 35 44 32 50 ...

$ thousand.years: int 405 1262 988 1264 796 1315 513 1362 1089 1376 ...

First we check the distributions of the variables. Figure 3 shows reasonably well that there aren’t any outliers, or otherwise strange features in the distributions of the four dimensions variables. Good. At this point it is often worth standardizing the data in one way or another.

If we had performed a standardization, we would have a new dataset. Lets call this dataset yy1, and progress through the numbers yy2, yy3, etc. as the steps of the NMDS unfold.

yy1 <- scale(dd[,2:5])

Next we calculate the distances. Since all the variables are numeric, we can use Euclidean distance, and the most convenient function to calculate Euclidean distances is dist() with the method argument specifying that we want Euclidean distances:

yy2 <- dist(yy1, method="euclidean")

The structure of the object returned by the dist() function appears a little complex, but you can ignore most of it. Interesting, however, should be the number of pairwise distances. Did you already calculate the number of pairwise distances you expected among 150 data points? We hope so – love R, but never trust it. If there are n data points, there will be n\*(n-1)/2 pairwise distances. If n = 150, this give 11175. Good, dist() is doing what we expect.

> str(yy22)

Class 'dist' atomic [1:11175] 22.9 36.3 37.4 36 35.4 ...

..- attr(\*, "Size")= int 150

..- attr(\*, "Diag")= logi FALSE

..- attr(\*, "Upper")= logi FALSE

..- attr(\*, "method")= chr "euclidean"

..- attr(\*, "call")= language dist(x = yy1, method = "euclidean")

Next we do the NMDS, using the function monoMDS in the vegan package. Other NMDS functions are available, such as isoMDS in the MASS package. However, we like to use the functions of the vegan package (including the distance calculating function, vegdist()) because of the absolutely super documentation that the author of the vegan package, Jari Oksanen, provides. Not only are the standard help files extensive and accessible, but there is the document *Multivariate Analysis of Ecological Communities in R: vegan tutorial*, provided online by Jari Oksanen. Enough praise, lets get on with the NMDS:

dd3 <- monoMDS(dd2, k=2, model="local")

Done. We asked for the new data to have to dimensions (k=2) and for local NMDS (for more information about other options here, see Box 3).

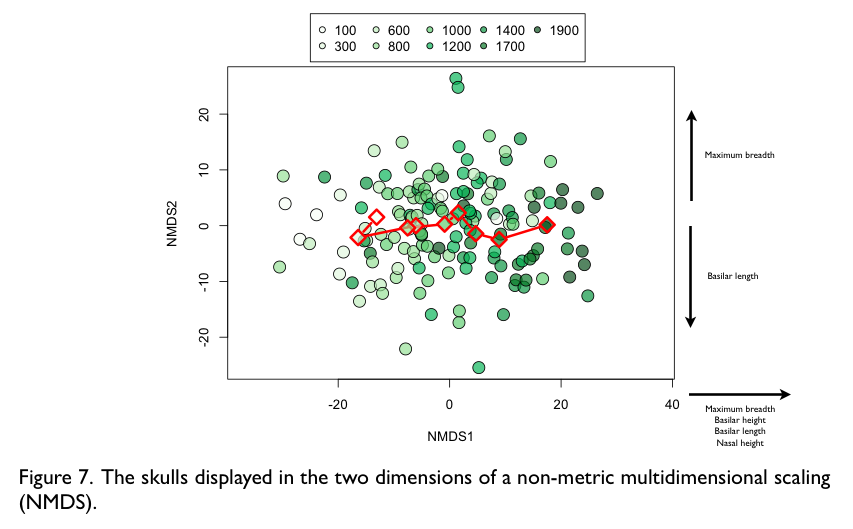
For once we won’t print here the structure of the object returned by monoMDS… it is too long… a list of 29 objects. The most important two objects are *points* and *stress*. The object dd3$points contains the new coordinates of the datapoints, so that plot(dd3$points) will plot the data points (skulls) in the new. We’re not going to do this yet, and when we do, we’ll use what we already learned in this chapter to colour code the points by date (similar to figures 4 and 6).

Stress is a measure of the goodness of fit of the NMDS, with lower values indicating better fit. Here we have a stress of 0.176; not bad. One can also plot the relationship between new distances and those of the original data, using the function stressplot(). Give it a go.

Now, lets go back a bit, to remember that in NMDS, there has to be some initial positioning of the datapoints on the piece of paper (i.e., in the new dimensions). Given this, we need to be careful that the choice of initial positions isn’t affecting the results of our NMDS. So we need to run the NMDS a few times with different initial positions. How to do this easily is covered in Box 3.

Lets now assume that we have done our NMDS, that we’re happy to have a low stress, and the solution is robust to initial positioning. Figure 7 shows the results of the NMDS (i.e., the skulls arrange on two axis at coordinates that are their NMDS scores), along with the mean position for each of the date classes we created. Again we can find the biological meaning of the new axes by correlating the raw data against the NMDS scores (this may not be possible if the raw data involved non-numeric variables). Comparing the ordination by PCA and the NMDS shows that the axes have the same meaning (except the direction of the y-axis is reversed – but this is arbitrary so is not a problem), and also that the positioning of individual points bears some similarities. This is reassuring, though note that when using NMDS in anger, we probably won’t be able to (nor would want to) compare its performance to that of PCA.

The vegan package contains some nice functions for drawing ellipses around classes of points and various other methods for visualising differences among datapoints. Please use the excellent vegan documentation for further learning.



## Hypothesis testing the wrong way

Recall our question: how has skull shape changed over time? We know from our previous data exploration that it has. Skulls have reduced in size, but show little change in shape. Now, on the graphs we made, these patterns are quite clear. There is little doubt that the reduction in size is not a result of chance. However, we will probably have to report the results of a hypothesis test to stand any chance of getting this work published. More relevant is that there might be cases when patterns are less clear, and we really do need the confirmation provided by a hypothesis test of some kind.

We might decide to do a hypothesis test on each of the four dimensions separately, perhaps using linear regression to test for a significant relationship between each variable and time. A challenge for you is to do this, putting the resulting regression line onto each of the plots in figure 3, and also adding to the figure the test statistic, degrees of freedom, and p-value. The code file associated with this chapter / document contains an example of how one can do this.

Probably many of you know that doing multiple tests like this is not such a good idea. The more tests we do, the greater the chance of finding a significant result, even for completely random data. If we stare long enough at the stars, we see patterns. Box 4, as well as providing some learning about programming in R, is a demonstration of the folly of multiple testing. We can easily get significant patterns just by chance.

One solution is to use a more conservative threshold p-value to indicate statistical significance. Perhaps 0.01, or 0.005. One can even calculate a threshold p-value such that the overall (type 1) error rate in a set of analyses will be 0.05 (e.g,. Bonferroni correction). There is another solution: to use a multivariate hypothesis test.

## Hypothesis testing the right way

At this point, we can’t strongly enough emphasise our frequent advice to fully explore and visualize your data before performing any hypothesis test. This is perhaps even more important for multivariate compared to univariate tests. The results (i.e., coefficients) of univariate tests are relatively easy to interpret compared to the results of multivariate tests.

Multivariate hypothesis tests allow us to not perform multiple tests. Such tests result in one significance (p) value, and hence suffer no multiple testing problems. There are just as many multivariate tests as there are univariate, ranging from multivariate versions of the t-test, through to complex multivariate generalized linear mixed models. This is neither the place nor the time to go into them all in detail. So all we do is two things. First, we confirm what we see in the skull dimensions dataset: skulls get significantly smaller. Second, we give a list of some common multivariate tests of significance with a sentence or two of description (plus directions to further reading).

First we do the significance tests, and we will do a few, namely multivariate ANOVA, multivariate regression, and a couple of non-parametric methods. Multivariate ANOVA implies that we have a categorical explanatory variable, here it is the year classes that we previously created. We also have the continuous variable ‘year’ and with this can do multivariate regression.

The first step is to place all the response variables into a data.frame, using cbind:

m.dd <- cbind(dd[,2], dd[,3], dd[,4], dd[,5])

Then we run the MANOVA using the function manova():

manova.model <- manova(m.dd ~ as.factor(dd$cut.year))

The explanatory variable dd$cut.year is the categorical year classes that we previously made. However, they are stored in R as a numeric variable, so to ensure MANOVA (rather than multivariate regression) we must make the variable a factor, using the function as.factor().

Next, we know that we must check if the model assumptions are met, before we even think about looking at the p-value. This is very important, just as it is for univariate tests. If we don’t test the model assumptions, we have no idea if we can trust the p-value. Unfortunately, testing the assumptions of MANOVA isn’t quite as easy as testing the assumptions of ANOVA (there are no easily available diagnostic plots, as there are with linear and generalized linear models using plot(model). However, there are functions available for testing most of the assumptions of MANOVA, and these are listed in Box 5.

When researchers report the outcome of a MANOVA, they choose a test statistic (e.g., Pillai trace), report it, and report the associated p-value. For example,

> summary(manova.model, test="Pillai")

Df Pillai approx F num Df den Df Pr(>F)

as.factor(dd$cut.year) 8 0.55942 2.8657 32 564 5.224e-07 \*\*\*

Residuals 141

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Here we see the value of Pillai trace (0.56) and the p-value (5.2 x 10-7), and the numerator and denominator degrees of freedom. Clearly the multivariate test is showing a very significant results. But what does this mean biologically? Simply put, it means that variation in the multiple response variables is significantly associated with variation in the explanatory variables. And just as with ANOVA (or any univariate test) the p-value tells us nothing about the *nature* of the differences. Fortunately, we already know the nature of the differences from all of the previous data exploration (including PCA and NMDS) and graphing we performed.

Similar to ANOVA, we can look at the MANOVA coefficients in an attempt to interpret its results.

> round(manova.model$coefficients, 4)

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

(Intercept) 122.3333 76.0000 78.1667 34.8333

as.factor(dd$cut.year)300 -3.0833 -1.7500 2.5833 -6.3333

as.factor(dd$cut.year)600 6.2667 -1.2667 6.3000 0.5667

as.factor(dd$cut.year)800 3.6267 1.6800 5.0333 -0.5933

as.factor(dd$cut.year)1000 8.9359 3.8462 8.2949 0.7436

as.factor(dd$cut.year)1200 10.2529 6.0000 7.7299 3.5115

as.factor(dd$cut.year)1400 8.4667 7.4500 11.5833 5.8167

as.factor(dd$cut.year)1700 10.8667 8.2667 15.9667 7.3667

as.factor(dd$cut.year)1900 15.1667 16.0000 16.7333 10.2667

Ugh. What a mess! Or perhaps not… there are nine coefficients for each of the four response variables, and as we expect, the coefficients become larger as we go further into the past (i.e., skulls become smaller as time passes).

We can do easily do a multivariate regression using the accurate year as the explanatory variable:

manova.model <- manova(m.dd ~ dd$thousand.years)

If you care to look at the significance of Pillai trace, you will find it more significant (i.e., lower p-value) due to the increased power of regression. Looking at the model coefficients, we find there are two for each response variable:

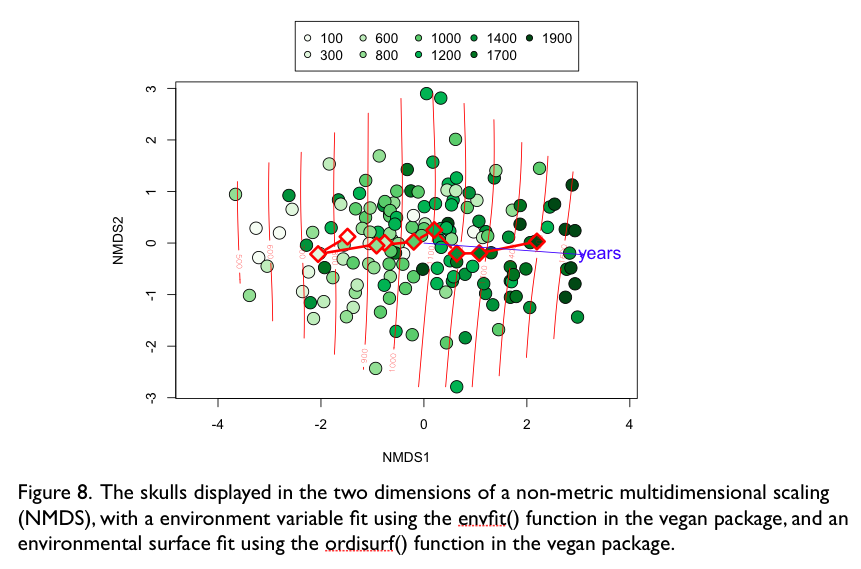
> round(manova.model$coefficients,4)

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

(Intercept) 122.0678 70.8967 77.2178 29.5855

dd$thousand.years 0.0075 0.0091 0.0089 0.0074

Just as in univariate regression, there is an intercept and a slope. Note that it might be worth reporting the slopes as change per thousand years, so that the numbers are not small fractions. In any case, the answer is the same as expected: skulls increase in size into the past.



So, we have shown with two parameteric multivariate tests that the decrease in skull size through time is significant. What if some of the assumptions of these parametric tests are not met, and in any case, how can we show what the model fit is, when we have multivariate data. Thankfully, the vegan package has a function to help us: envfit(). This is short for “please fit an environmental variable” to my ordination. Although vegan suggests that an environmental variable is fit, in fact, any variable that might be associated with the ordination can be used. (Recall that the ordination [say NMD] contains the multiple response variables, and the explanatory variables are what vegan is calling “environmental variables”). Use of envfit() is relatively simple, and it can handle continuous or categorical explanatory variables.

years <- dd$thousand.years  
model1 <- envfit(skulls.NMDS ~ years, perm=999)

The first line is just a bit of housekeeping. When we put the model onto the ordination plot, we would rather just have the word “years” than “dd$thousand.years”. The second line fits the model, and also asks that a permutation test (with 999 permutations) of the significance of the model is performed. We plot the fit of the model onto the ordination we made before:

plot(model1, cex=1.3)

The line tells us the direction (i.e., vector) of change in skull shape that the environment variable explains the greatest variability in. There is nothing surprising with the results we see (figure 8). We can find the significance of this model fit:

> model1

\*\*\*VECTORS

NMDS1 NMDS2 r2 Pr(>r)

years 0.997348 -0.072787 0.362 0.000999 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

P values based on 1000 permutations.

Very significant, confirming the findings of the MANOVA and multivariate regression.

The function envfit() assumes a linear relationship between the ordination and the environmental variable. If one wishes to fit a non-linear relationship (a generalized additive model in this case), the function ordisurf() is available:

model2 <- ordisurf(skulls.NMDS ~ years, add=T)  
summary(model2)

Which again gives us a nice visualization of how year (the explanatory variable) relates with the ordination (figure 8). In fact, since the contours are parallel and quite evenly spaced, we can see that the relationship is indeed quite linear. The call to summary() provides the significance value of the surface fit.

If we have a categorical variable, we still use the envfit() function:

year <- as.factor(dd$cut.year)  
model2 <- envfit(skulls.NMDS ~ year, perm=999)

And if you plot this model, or look at its summary() you will see that it has fit to the data each of the year categories, and that these lie on top of the means (red triangles in the figures) that we already have. As expected, the permutation test indicates that categorical year is significantly related to the ordination of skull dimensions.

Great – that all works very well. Of course, the skulls dataset is not too complicated, and has a strong pattern. This may not be the case with your own data, and in that case some perseverance will be required, but if you follow our advice, you should not have too much trouble. First, have a well specified question. Second, explore the data, preferably producing a graph that directly and clearly gives you a good idea of the answer to your question. Third, make a significance test to confirm that the pattern is unlikely to be a result of chance.

Maybe mention tests of homogeneity here (betadisper() and PERMDISP), since these concern a new type of question.

## Wrapping up

To be written, if required.

## Box 1. Colouring plotting symbols by a variable.

Often we would like to make a scatter plot in which the colour of the plotting symbols, and perhaps the type of symbols, correspond with the values in a variable. Previously (in GSWR) we gave a few methods for achieving this. Here we give a more complete and general method, and also show how to colour by a continuous variable.

In the skulls data, we could make a bivariate scatter plot of one dimension against another, and colour the symbols according to the year they came from. Or we might want to plot age and size at maturity of many individual Daphnia, and to colour the individuals according to their strain. Or we might have plot the abundance of one species against that of another, and colour the points according to the temperature of the location they occurred in.

The first step in colouring points (or anything in graphs) is to decide on the colours to use (i.e., decide on a colour palette). We recommend use of the colour palettes in a package called RColorBrewer. You can easily explore what the package with a visit to the website colorbrewer2.org. A great deal of thought went into the design of the palettes in RColorBrewer. We recommend you take advantage of it.

There are three types of palette in RColorBrewer, and they correspond with the three example in the first paragraph of this box. Sequential palettes are good for illustrating variables in which the values have an order, for example the date on which a skull was found (e.g., figure 4). Diverging palettes are also good for variables that have order, but are special because they can show diverge up and down from some central value. In the example above, zero degree Celcius would a natural point at which to centre the diverging palette. Finally, qualitative palettes are good for variables that are factors, i.e., their entries have no intrinsic order.

The next step is to decide how many colours you will have in the palette. The maximum number of colours available in a qualitative palette in RColourBrewer is 12. If the data is quantitative, or otherwise ordered, one can have a maximum of nine colours in a sequential palette, or 11 in a diverging palette. The reason for these maximum number of colours is that have more can be confusing and also lead to colours that are too similar too reliably distinguish. One might decide to use many many colours to illustrate continuous gradients, but this is not our concern right now. Here’s how, using RColorBrewer, we make a palette of nine shades of green:

library(RColorBrewer) ## download the package first?  
cc <- brewer.pal(9, "Greens")

Now that we have our palette, we must define an association between the colours in the palette, and the levels of the variable we are going to colour by. If this variable is continuous, first we must discretize it (i.e., put it into classes). To do this we first make a vector of the break points, or boundaries between the classes:

breaks <- seq(min(yy$thousand.years),  
 max(yy$thousand.years),  
 length=10)

The length argument tells seq() that we would like to have ten boundaries, and we would like ten because this will result in nine classes, and we want nine classes because our sequential palette has nine colours. We would also like to know the midpoints of the classes:

mids <- round((breaks[-1] + breaks[-10]) / 2, -2)

If you don’t know, figure out how this code works, and what the -2 does.

Next we add a new variable (below we name it cut.year) to the original data, that contains the midpoint of the year class that the skull belongs to:

dd <- transform(dd, cut.year=mids[as.numeric(cut(dd$thousand.years,  
 breaks=breaks,  
 include.lowest=T))])

The hard work is done by the cut() function, which we give three arguments. The first is the variable to be cut up into classes. The second (breaks) is the breaks (boundaries) separating the classes. The third (include.lowest=T) tells cut() to include in the classes values that fall on the lower boundary. Then we do a bit of jiggery-pokery (i.e., mids[as.numeric(…)]) to get a variable of the midpoints, rather than the rather obscure class names provided by cut().

Now we are ready to make a dataframe that contains the association between the colours in the palette, and the levels of the variable we are going to colour by:

col.map <- data.frame(year.classes= unique(dd$cut.year),  
 colz=I(cc))

> col.map

year.classes colz

1 100 #F7FCF5

2 300 #E5F5E0

3 600 #C7E9C0

4 800 #A1D99B

5 1000 #74C476

6 1200 #41AB5D

7 1400 #238B45

8 1700 #006D2C

9 1900 #00441B

The column colz contains the colours stored as hexadecimal codes. It would be nicer to have the shades of green given in words, but computers will be computers!

Phew, nearly there. All that remains is to plot the data, and to make the colour of the points correspond to the class in dd$year.class:

plot(dd[,2], dd[,3],  
 xlab=nice.names[2],  
 ylab=nice.names[3],  
 pch=21,  
 bg=col.map$col[match(dd$cut.year, col.map$year.classes)])

The hard work here is done by the match() function. According to match()’s help file, it “returns a vector of the positions of matches of its first argument in its second. So, if the second argument were the letters A, B, C, D, E, and the first argument were C, D, A, what do you think match would return. See the bottom of the next page for the answer. Try to remember that the first argument is what you would like to look up the position of in the second. Hence, match() gives you the same number of answers (positions in the second argument) as there are values in the first argument. The match() function can also be used for merging datasets, and for checking if entries in one vector occur in another, without having to do any sorting (as we did above on page XX).

Finally, we might want to make the colours transparent, so that we have a chance of seeing overlying data points. One way to do this requires we know something about how the hexadecimal numbers (e.g., #11223344) are used to encode colours. The hash (#) you can ignore. The next two numbers (11 in our example) give the amount of red (00 is no red, ff is maximum red), then the same for green (22 in our example), and then the same for blue (33 in our example). The last two numbers (44) specify the transparency of the colour (00 is completely transparent, ff is completely solid) (if one misses off the last two digits, the colour is solid). To stick on two additional numbers to the colours in the object col.map, we used the paste() function:

col.map$col <- paste(col.map$col, "aa", sep="")

This replaces the col variable in the col.map data.frame with a new variable that is **“**aa**”** pasted onto the end of the previous colours (i.e., the ones that came from RColorBrewer), without any separator. The result is in Figure D.

The method given above is quite long winded, but very flexible and safe. For example, it is easily modified to allow plotting symbols to correspond to categorical treatments (i.e., factors). It is safe because one can use col.map to make a legend in which the correspondence between colours and values cannot be wrong. However, there are easier ways, not least of which is to use lattice (or ggplot2), though it may be harder to customize these. (Don’t forget that lattice [and ggplot2] are excellent for exploratory graphics.)

The answer was 3, 4, 1.

## Box 2. Some additional information about PCA

PCA transforms the data into a set of orthogonal (independent) variables. Check what cor(pp$scores) gives.

PCA is most useful if the first two axes explain a large proportion of the variance in the original data, since then we can use 2-D plots. What is large is a matter of judgement.

PCA is performed on raw or scaled data. If you variables are measured in different units (e.g., some time, some length, some mass) you have no choice but to scale the data. If they are measured with the same units, you have a choice. Scale the data if you want to look at relative differences; don’t scale if you want to see absolute differences.

PCA performs linear transformations of the original data. It rotates the data. It does not squeeze, stretch, or bend the data cloud. Other methods can be used to reduce the dimensionality of non-linear data; the one we will look at is NMDS (non-metric multidimensional scaling). This method is also very useful for datasets where not all of the variables are continuous (e.g., some continuous, some categorical).

One mathematical method for PCA is eigen analysis, so you may hear people talk about eigen values in the context of PCA. To do a PCA by hand in R, you calculate the eigen values of the covariance (or correlation) matrix of the data, and then premultiply the original data by the result:

t(t(eigen(cor(dd[,2:5]))$vectors) %\*% t(scale(dd[,2:5])))

Ugly huh?

## Box 3. NMDS, some of the gory details

What does the name mean? *Non-metric* refers to the fact that rank correlation is used to measure how closely the distances of the original data match the distances of the transformed data. *Multidimensional* refers to the fact that multiple dimensions can be handled, and *scaling* just refers to the scaling of the data that occurs. One might think about whether one could replace *scaling* with *ordination*.

The second step, after data transformation/standardisation is to choose a distance measure. Euclidean distance can be a good choice when all your variables contain numeric data. Bray-Curtis dissimilarity is good for binary data. Gower distance is good for mixed variables (some binary, some numeric, some categorical). Calculating the distances can be done with several functions, including:

* dist() in the stats package
* vegdist() in the vegan package
* daisy() in the cluster package

There is, fact, not a single type of NMDS, and the monoMDS() function in the vegan package can perform four types of NMDS: local, global, linear, or hybrid NMDS.

NMDS is quite a difficult computational problem, that does not necessarily result in a robust answer. The answer may, for example, be sensitive to the initial position in the NMDS that data points are given in. Comparing the output of two NMDSs (or in fact any ordinations) can be easily performed using the procrustes() function in the vegan package. It returns information about the match between the two ordinations, and this can be used to measure how similar they are.

For the specific purpose of detecting gradients in species composition data, Minchin (1987) identified “*LNMDS, using the Bray-Curtis dissimilarity coefficient, as a robust technique for the analysis of community data when the aim is to recover the compositional dimensions associated with underlying environmental gradients*” and that “*The results suggest a preference for 'local' over 'global' NMDS and the Bray-Curtis measure was among the most robust of the coefficients compared, in terms of its rank correlation with simulated environmental distance.”*

Minchin, P. (1987). An evaluation of the relative robustness of techniques for ecological ordination. Plant Ecology, 69, 89–107.

## Box 4. A short program illustrating the folly of multiple testing.

Here we demonstrate how to make a dataset of random numbers, perform a T-test on each of the variables in this dataset, record the p-value of each T-test, and then show the minimum p-value. As well as showing you that multiple tests are dangerous, you can learn some simple programming skills.

First we make a matrix of 20 variables, each with 60 values (rows). The values are taken from a standard normal distribution:

dd <- matrix(rnorm(60\*20), 60, 20)

Then we make an empty vector of 20 numbers, ready to receive the p-value of the T-test applied to each of the 20 variables in the dataset.

p.value <- numeric(20)

Now we make a loop to repeat the T-test on each of the 20 variables:

for(i in 1:20)  
 p.value[i] <- t.test(dd[1:30,i], dd[31:60,i])$p.value

Finally we ask R to report the minimum p-value of the 20 tests.

min(p.value)

Finally we ask R to report the minimum p-value of the 20 tests. If you run this code, you will find that often (how often?) the p-value is less than 0.05, indicating that the T-test has found a significant difference, even though the data is entirely random.

By the way, some say that loops are not required in R, and any (at least most) loops can be replaced by a function. For example, instead of the loop above we could have used apply():

min(apply(dd, 2, function(x) t.test(x[1:30], x[31:60])$p.value))

The function apply() applies a function on each the rows (if the second argument is 1) or columns (if the second argument is 2). We have asked apply() to apply the t.test() function, with the $p.value on the end, to extract the p-value. There is no column dimension in the data passed to t.test(), because apply() passes to t.test() each of the columns in turn.

As is often the case, there are multiple ways to get to the same goal. People growing up with R should try to minimize loops by maximizing use of functions like apply() (look at the “See Also” section of the apply() help document). People grown up with C, fortran, etc., would probably benefit from attempting the same, though reverting back to loops is often quite comforting.

## Box 5. Testing the assumptions of multivariate significance tests.

First we must make sure each of the variables meets the assumptions of ANOVA: normality, homogeneity of variances, and lack of outliers. As usual, this involves some combination of graphical exploration and statistical tests (see below). Below are listed the assumptions alongside advice on the graphs and functions that can be used to test them. In fact, it’s a good idea to check all of these, even if one is going to perform a hypothesis test other than MANOVA. Functions that apply to some univariate tests are marked with an asterisk (\*).

Check against Gustavo’s ms.

*Univariate outliers*

Univariate: Calculate z-scores and check for values much greater than three or four.

*Multivariate outliers*

aq.plot() in mvoutlier package

*Univariate normality*

qqnorm() and qqline()

shapiro.test(), and functions in the nortest package.

*Multivariate normality*

mshapiro.test() in the mvnormtest package

*Homogeneity of variances*\*

Visual inspection of the distributions of the data.

bartlett.test()

Fit a model with heterogeneous variance and test the importance of this property.

*Homogeneity of covariances*

Good luck!… check Gustavo’s ms.

## Box 6. Other multivariate methods and significance tests

There are quite a lot of multivariate methods, and, unfortunately, their names can be rather confusing. Below is a list of some of the common type

**Unconstrained ordination**

Principle components analysis (via prcomp(), princomp() or rda() in vegan). Assumes linear responses of species to an environmental gradient, or a very short part of a gradient).

Principle coordinates analysis (metric scaling, MDS). Assumes unimodal relationships between species and an environmental gradient.

Correspondence analysis (via cca() in vegan) (an analysis of non-independence of rows and columns).

Detrended correspondence analysis (decorana() in vegan)

Factor analysis (not often used by biologists, Quinn & Keough)

**Constrained ordinations**

Constrained analysis of proximities (capscale() in vegan)

Redundancy analysis (rda() in vegan)

Canonical correspondence analysis (cca() in vegan). Unimodal context.

Canonical correlation analysis (CCA). This is a generalization of multiple regression, in which there are both multiple explanatory variables and multiple response variables. A typical question would be how are the frequencies of multiple genes (the response variables) related to a group of environmental variables. Canonical *correlation* analyses assumes linear relationships between variables, whereas canonical *correspondence* analysis assumes unimodal relationships.

**Hypopthesis testing**

MANOVA based on dissimilarities (adonis() in vegan) (examines differences in group means)

Discriminant function analysis (same as MANOVA, but with more focus on classification)

Homogeneity of groups (betadisper() in vegan)

(Robust MANOVA using randomization).

PERMANOVA

PERMDISP

MANTEL

Procrustes test. Procrustes analysis can, in fact, be used to compare two ordinations that have the same datapoints, but that were derived from different data. One ordination could be about the species that occur at particular sites (with sites being the datapoints) while the other ordination could be about the environmental conditions at the same sites.

**Classification**

hclust(), reorder()

labdsv package

Discriminant function analysis (same as MANOVA, but with more focus on classification)

Random trees

ANN