

# R-Course Day 4

Matthew K. Lau

November 29, 2008

**In class last Tuesday we covered:**

1. Review of Analysis in **R**
2. Writing Functions
3. Sorting Data
4. Principal Components Analysis
5. Information Theory Based Multi-Model Inference

## 1 Review of Analysis in R

Before we keep going with progressively complex topics, I would like to bring things back to brass tacks. At the heart of all of these fancy scripts and packages is our desire to conduct efficient inferences about our questions of interest. This inherently includes all aspects of analysis from data management to presentation. It's very easy to lose sight of this in the midst of searching around for the "right" test and presentation format. I constantly remind myself by asking, "what the \$#&% is the point of all this?"

Once you get around the initial difficulties of using **R** it provides an analytical environment that has a greater potential. The goal of making plots and conducting analyses is to generate a coherent, reasoned argument for or against a particular line of inference, not to fulfill some predefined list of rules. Here is a quick run through a hypothetical analysis scenario:

First we create we get data, which in this case are generated by **R**, but we could have also entered them in via any number of data importing functions:

```
> x1 <- rnorm(10, 10, 2)
> x2 <- rnorm(10, 15, 2)
```

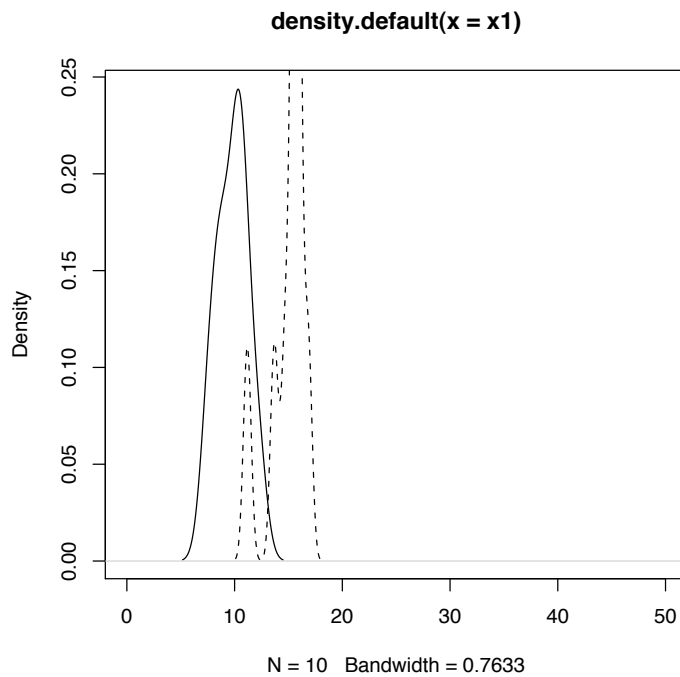
Once we have the data inputted, we would most likely want to visualize and summarize the data:

```

> plot(density(x1), xlim = c(0, 50))
> lines(density(x2), lty = 2)
> summary(cbind(x1, x2))

```

	x1	x2
Min.	: 7.455	Min. :11.18
1st Qu.:	8.838	1st Qu.:14.90
Median :	10.028	Median :15.59
Mean :	9.799	Mean :15.06
3rd Qu.:	10.639	3rd Qu.:15.75
Max.	:12.211	Max. :16.84



Based on our knowledge of statistical inference, we may decide to conduct a test of that the two means are different from each other via a non-directional, two-sample t-test:

```

> tt <- t.test(x1, x2)
> tt

```

Welch Two Sample t-test

```

data: x1 and x2
t = -7.705, df = 17.849, p-value = 4.405e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-6.698512 -3.826834

```

```

sample estimates:
mean of x mean of y
 9.798954 15.061627

```

It is important to keep good records of analyses. One way to do this would be to make a file with both the data and the results of the t-test together. This could stave off a huge headache later-on if a repeat analysis is called for:

```

> file.create("/Users/artemis/Documents/FALL2008/R-Course_Day4_Example/Day4_data.R")
[1] TRUE
> file.create("/Users/artemis/Documents/FALL2008/R-Course_Day4_Example/Day4_results.R")
[1] TRUE
> write.table(cbind(x1, x2), file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_data.R")
> write.table(unlist(tt), file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_results.R")

```

Notice that we used the `cbind` and the `unlist` functions to reformat the data and the results before saving them as ".R" files. To double check that everything was saved correctly, we can read the files back into **R**:

```

> read.table("/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_data.R")

```

	x1	x2
1	7.454862	16.06117
2	10.543494	15.67643
3	10.671207	15.48232
4	8.794536	15.51148
5	12.211224	14.69980
6	11.155776	15.73512
7	8.134445	11.17991
8	9.930993	16.83588
9	8.968685	15.74910
10	10.124315	13.68506

```

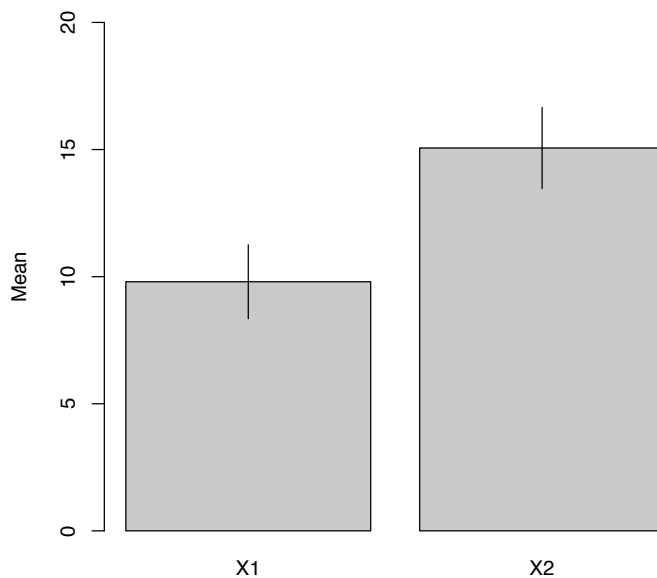
> read.table(file = "/Users/artemis/Documents/FALL2008/
R-Course_Day4_Example/Day4_results.R")

```

	x
statistic.t	-7.7050266505846
parameter.df	17.8489694691014
p.value	4.40486369636702e-07
conf.int1	-6.6985120477549
conf.int2	-3.82683448815842
estimate.mean of x	9.79895376046191
estimate.mean of y	15.0616270284186
null.value.difference in means	0
alternative	two.sided
method	Welch Two Sample t-test
data.name	x1 and x2

Once we have our analyses done, we will most likely want to create a figure to present these results. Previously, we used the `barplot2` function in the `gplots` package, but here we will use the `lines` function to plot our confidence intervals.

```
> barplot(c(mean(x1), mean(x2)), names = c("X1", "X2"), ylab = "Mean",
+         ylim = c(0, 20), col = "grey")
> lines(c(0.7, 0.7), c(mean(x1) - sd(x1), mean(x1) + sd(x1)),
+         lwd = 1)
> lines(c(1.9, 1.9), c(mean(x2) - sd(x2), mean(x2) + sd(x2)))
```



As you can see, the `barplot` function creates an initial plot, which we then "write" on top of with the `lines` functions.

And last, we clean up our mess by removing all of the objects that we created. Had we loaded any packages that we weren't going to use later on, we'd unload them by the `detach` function:

```
> rm(list = ls())
```

## 2 Writing Functions

We have been using functions left and right. All functions do is group together other functions in an applied way. Because **R** is an "open-source" programming language, many of the functions in its employ have been created by users of **R**

(e.g. `barplots2`). We can see all of the code (i.e. the guts) of any function by typing in the name of the function with no parantheses:

```
> sd
function (x, na.rm = FALSE)
{
  if (is.matrix(x))
    apply(x, 2, sd, na.rm = na.rm)
  else if (is.vector(x))
    sqrt(var(x, na.rm = na.rm))
  else if (is.data.frame(x))
    sapply(x, sd, na.rm = na.rm)
  else sqrt(var(as.vector(x), na.rm = na.rm))
}
<environment: namespace:stats>
```

### Function Anatomy

Functions have a structure very similar to loops. In fact, one way of looking at a function is to see it as a function that creates a function from functions. For example, if we want to create a function that will calculate the standard error of a sample for us, we would write:

```
> SE <- function(x = "sample vector") {
+   se <- sd(x)/sqrt(length(x))
+   return(se)
+ }
```

Let's take a look at what's going on. First, there is the `function` function itself. This allows for object oriented arguments to be specified in a general way. When we use our function later on, these arguments will need to be specified. Second, we specify the "action" of the function (i.e. what the function will be doing). This can be anything. Here we calculate the standard error, "se", and then use the `return` function to give us the calculated value. Last, we create an object with the backward arrow, `<-`. This names our function by saving it into as an object. Now, we can use our function:

```
> x <- rnorm(10, 10, 2)
> x
[1]  8.828547 10.691604 10.516017 11.258912 10.312600  8.651738
[7] 11.015661 10.920978 12.779844 11.420712

> SE(x)
[1] 0.3822643
```

Notice here that functions can be composed of any other functions, including loops. For example:

```
> loop.func <- function(x = "number of loops") {
+   for (i in 1:100) {
+     x <- x + 1
+   }
+   return(x)
+ }
> loop.func(10)

[1] 110
```

Functions are the atoms of the nucleus that is **R**. Knowing how to write them will allow you to do many things in **R** that you can not do in your standard point and click stats packages.

### 3 Sorting Data

Sorting data is an essential part of data management. In **R** sorting is very simply achieved by using the bracket specification. For example, if we wanted to sort our data matrix by the central column from lowest to highest:

```
> data <- cbind(1:10, rep(c(1, 2), 10), rnorm(10))
```

We could go in and create a vector of the position of the all the values in the correct order, and put it in our bracket specification:

```
> sorting.vector <- c(1, 3, 5, 7, 9, 11, 13, 15, 17, 19,
+   2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
> data[sorting.vector, ]

      [,1] [,2]      [,3]
[1,]    1    1 -0.3354490
[2,]    3    1  0.4490912
[3,]    5    1 -0.2910797
[4,]    7    1  2.1438397
[5,]    9    1  0.4063706
[6,]    1    1 -0.3354490
[7,]    3    1  0.4490912
[8,]    5    1 -0.2910797
[9,]    7    1  2.1438397
[10,]   9    1  0.4063706
[11,]   2    2 -1.1793572
[12,]   4    2  1.0220319
[13,]   6    2  1.2972297
[14,]   8    2  0.3452335
[15,]  10    2  0.1251559
[16,]   2    2 -1.1793572
[17,]   4    2  1.0220319
```

```
[18,]    6    2  1.2972297
[19,]    8    2  0.3452335
[20,]   10    2  0.1251559
```

Note that we are placing our "sorting vector" in the row portion of our bracket specification. This is because we are telling **R** to give us the rows of the object *data* as specified by our sorting vector, just like we've use it before to pull out portions of data vectors and matrixes.

This way is a very impractical to sort data on a regular basis. Instead, we can use the `order` function to return the "sorting vector" for us, both separately and within the bracket specification itself:

```
> order(data[, 2])

[1]  1  3  5  7  9 11 13 15 17 19  2  4  6  8 10 12 14 16 18 20

> sorting.vector2 <- order(data[, 2])
> sorting.vector2

[1]  1  3  5  7  9 11 13 15 17 19  2  4  6  8 10 12 14 16 18 20

> data[sorting.vector2, ]

      [,1] [,2]      [,3]
[1,]    1    1 -0.3354490
[2,]    3    1  0.4490912
[3,]    5    1 -0.2910797
[4,]    7    1  2.1438397
[5,]    9    1  0.4063706
[6,]    1    1 -0.3354490
[7,]    3    1  0.4490912
[8,]    5    1 -0.2910797
[9,]    7    1  2.1438397
[10,]   9    1  0.4063706
[11,]   2    2 -1.1793572
[12,]   4    2  1.0220319
[13,]   6    2  1.2972297
[14,]   8    2  0.3452335
[15,]  10    2  0.1251559
[16,]   2    2 -1.1793572
[17,]   4    2  1.0220319
[18,]   6    2  1.2972297
[19,]   8    2  0.3452335
[20,]  10    2  0.1251559

> data[order(data[, 2]), ]

      [,1] [,2]      [,3]
[1,]    1    1 -0.3354490
[2,]    3    1  0.4490912
```

```

[3,] 5 1 -0.2910797
[4,] 7 1 2.1438397
[5,] 9 1 0.4063706
[6,] 1 1 -0.3354490
[7,] 3 1 0.4490912
[8,] 5 1 -0.2910797
[9,] 7 1 2.1438397
[10,] 9 1 0.4063706
[11,] 2 2 -1.1793572
[12,] 4 2 1.0220319
[13,] 6 2 1.2972297
[14,] 8 2 0.3452335
[15,] 10 2 0.1251559
[16,] 2 2 -1.1793572
[17,] 4 2 1.0220319
[18,] 6 2 1.2972297
[19,] 8 2 0.3452335
[20,] 10 2 0.1251559

```

To sort from highest to lowest, we merely specify the *decreasing* argument in the *order* function:

```
> data[order(data[, 2], decreasing = TRUE), ]
```

```

      [,1] [,2]      [,3]
[1,]    2    2 -1.1793572
[2,]    4    2  1.0220319
[3,]    6    2  1.2972297
[4,]    8    2  0.3452335
[5,]   10    2  0.1251559
[6,]    2    2 -1.1793572
[7,]    4    2  1.0220319
[8,]    6    2  1.2972297
[9,]    8    2  0.3452335
[10,]  10    2  0.1251559
[11,]    1    1 -0.3354490
[12,]    3    1  0.4490912
[13,]    5    1 -0.2910797
[14,]    7    1  2.1438397
[15,]    9    1  0.4063706
[16,]    1    1 -0.3354490
[17,]    3    1  0.4490912
[18,]    5    1 -0.2910797
[19,]    7    1  2.1438397
[20,]    9    1  0.4063706

```

## 4 Principal Components Analysis (PCA)

The goal of any ordination technique is to reduce data complexity. This either the number of factors or response variables. Typically, ecologists are interested

in the latter.

PCA is one ordination method that uses linear combinations of a subset of variables to represent the full set of variables. This method, although more informative than Non-Metric multidimensional Scaling (NMS), is not typically compatible with community data, which often has strong non-linear relationships. This is typically done through a subjective process of assessing the proportion of variance explained by each "principal component" (i.e. one linear combination of variables). This is actually a very simple process. Although there are multiple PCA functions available, because of its simplicity, it is not much finger-work to do it from scratch by writing loops:

Load *vegan* package and the "dune" data, which comes with it.

```
> library(vegan)
> data(dune)
> dune[1:3, ]
```

```
      Belper Empnig Junbuf Junart Airpra Elepal Rumace Viclat Brarut
2       3      0      0      0      0      0      0      0      0      0
13      0      0      3      0      0      0      0      0      0      0
4       2      0      0      0      0      0      0      0      0      2
      Ranfla Cirarv Hyprad Leoaut Potpal Poapra Calcus Tripra Trirep
2       0      0      0      5      0      4      0      0      0      5
13      2      0      0      2      0      2      0      0      0      2
4       0      2      0      2      0      4      0      0      0      1
      Antodo Salrep Achmil Poatri Chealb Elyrep Sagpro Plalan Agrsto
2       0      0      3      7      0      4      0      0      0      0
13      0      0      0      9      1      0      2      0      0      5
4       0      0      0      5      0      4      5      0      0      8
      Lolper Alogen Brohor
2       5      2      4
13      0      5      0
4       5      2      3
```

Calculate the covariance matrix from the data matrix:

```
> c <- cov(dune)
```

Calculate eigenvalues from the covariance matrix:

```
> dune.e <- eigen(c)
> e.values <- dune.e$values
> is.vector(e.values)
```

```
[1] TRUE
```

```
> e.values
```

```
[1] 2.479532e+01 1.814662e+01 7.629135e+00 7.152772e+00
[5] 5.695027e+00 4.333307e+00 3.199365e+00 2.781865e+00
```

```

[9] 2.481984e+00 1.853767e+00 1.747117e+00 1.313583e+00
[13] 9.905115e-01 6.377937e-01 5.508266e-01 3.505841e-01
[17] 1.995562e-01 1.487978e-01 1.157526e-01 2.477742e-16
[21] 1.265238e-16 9.915497e-17 7.827192e-17 7.767934e-17
[25] -1.615479e-17 -1.670766e-16 -1.790788e-16 -1.994938e-16
[29] -7.571117e-16 -7.735208e-15

```

Calculate the Percent Variance Explained (PVE) by each principal component (NOTE: the eigenvalues are in order from highest to lowest and each principal component is defined by the rank of its associated eigenvalue):

```
> e.values[1]/sum(e.values)
```

```
[1] 0.2947484
```

This is more easily done by using a loop:

```

> pve = numeric()
> for (i in 1:length(e.values)) {
+   pve[i] <- e.values[i]/sum(e.values)
+ }
> pve

[1] 2.947484e-01 2.157136e-01 9.068950e-02 8.502685e-02
[5] 6.769826e-02 5.151114e-02 3.803168e-02 3.306874e-02
[9] 2.950399e-02 2.203621e-02 2.076844e-02 1.561490e-02
[13] 1.177447e-02 7.581619e-03 6.547818e-03 4.167483e-03
[17] 2.372176e-03 1.768798e-03 1.375981e-03 2.945356e-18
[21] 1.504021e-18 1.178681e-18 9.304386e-19 9.233944e-19
[25] -1.920362e-19 -1.986082e-18 -2.128757e-18 -2.371435e-18
[29] -8.999983e-18 -9.195042e-17

```

Now, we can calculate the Cumulative Percent Variance explained by each successive principal component using yet another loop:

```

> cpve = numeric()
> for (i in 1:length(pve)) {
+   if (i == 1) {
+     cpve[i] <- pve[i]
+   }
+   else {
+     cpve[i] <- cpve[i - 1] + pve[i]
+   }
+ }
> cpve

[1] 0.2947484 0.5104620 0.6011515 0.6861783 0.7538766 0.8053877
[7] 0.8434194 0.8764881 0.9059921 0.9280283 0.9487968 0.9644117
[13] 0.9761861 0.9837677 0.9903156 0.9944830 0.9968552 0.9986240
[19] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[25] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000

```

A scree plot is a nice visual summary of this information:

```
■fig=true■ plot(cpve) abline(a=0.95,b=0)
```

To get the principal components, we simply multiply the original data matrix by the eigenvector matrix that we can get from the `eigen` function that we used to get our eigenvalues. To do this, we need to first convert the dune data into a matrix (it's currently a data frame), and then we need to use the matrix multiplier `"%%"` to multiply the two together:

```
> X <- as.matrix(dune)
> V <- eigen(cov(dune))$vector
> PC <- X %% V
> PC[1:5, ]

      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
2  7.8632333 -6.9156455 -1.802930 -0.7592717  1.0038289  4.880190
13 -0.1837002 -9.7166345  4.098879 -1.7969386 -0.5913193  3.223097
4   2.0525466 -9.3132762 -2.539806 -0.7808169 -3.3998013  1.247622
16 -7.5610307 -3.9151554 -1.455246 -6.4509398 -1.8612388  2.165339
6   8.9943182 -0.1316022  2.075440 -8.7597034 -4.8445081  2.326789

      [,7]      [,8]      [,9]      [,10]     [,11]     [,12]
2 -5.013353 -2.2455974 -2.6476661 -2.8427649 -0.4593202  1.2118617
13 -2.296845  0.8166063 -2.1470136  0.3124062 -2.8458103  2.1506161
4  -7.624091  3.8758407  1.2243159 -1.9621888 -1.0915741  0.6602083
16 -2.827212  1.5025837 -2.2036058 -3.7778249  0.3165852  0.5928802
6  -3.365366  0.2061268  0.2744863 -3.2406152 -0.9433286  2.7334334

      [,13]     [,14]     [,15]     [,16]     [,17]     [,18]
2 -0.40358193  0.1596677  1.22251362  1.0052880  0.4765954 -0.6637914
13 -0.73115235  1.6597452 -0.03941401  1.9043549 -0.3642813 -0.7397268
4   0.12584658  1.6981585  0.38371399  0.9805425  0.3957807 -0.8773133
16  0.08986328  1.6755371  1.84847618  2.3480531  0.1527719 -0.4932394
6   0.67422064  0.9965782 -0.47402683  1.3346653  0.3012177 -0.5218971

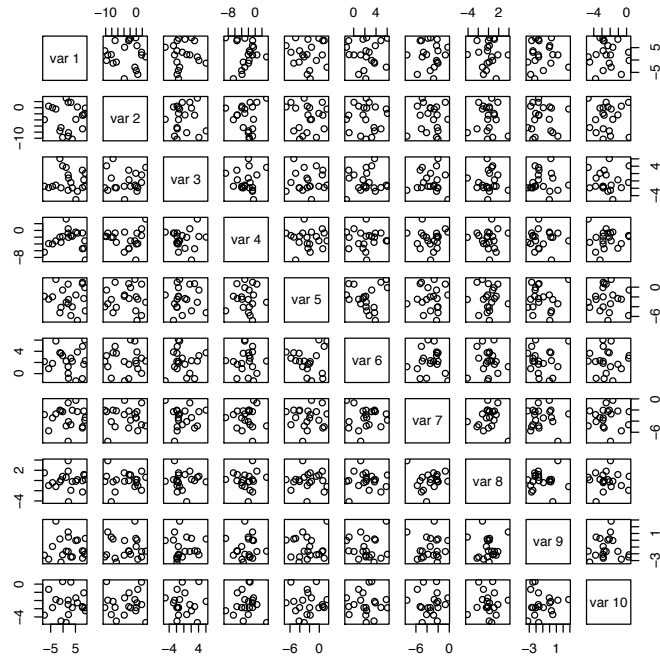
      [,19]     [,20]     [,21]     [,22]     [,23]     [,24]     [,25]
2  0.8675266  0.4202482  0.6969981  0.349236 -1.300027  0.2059339  0.6115661
13 0.4284945  0.4202482  0.6969981  0.349236 -1.300027  0.2059339  0.6115661
4   0.4281225  0.4202482  0.6969981  0.349236 -1.300027  0.2059339  0.6115661
16 0.3784169  0.4202482  0.6969981  0.349236 -1.300027  0.2059339  0.6115661
6   0.6381589  0.4202482  0.6969981  0.349236 -1.300027  0.2059339  0.6115661

      [,26]     [,27]     [,28]     [,29]     [,30]
2 -0.8477374 -0.1990721 -0.8456182  0.1189631  0.5027105
13 -0.8477374 -0.1990721 -0.8456182  0.1189631  0.5027105
4  -0.8477374 -0.1990721 -0.8456182  0.1189631  0.5027105
16 -0.8477374 -0.1990721 -0.8456182  0.1189631  0.5027105
6  -0.8477374 -0.1990721 -0.8456182  0.1189631  0.5027105
```

And, if we were so inclined, we could plot all pairs of the first 10 principal components using the `pairs` function:

```
> detach(package:vegan)
> rm(list = ls())
```

```
> pairs(PC[, 1:10])
```



## 5 Information Theory Based Multi-Model Inference

It is far too often in ecology that the model or hypothesis set cannot be narrowed to a dichotomous null-hypothesis test. Without getting to much into the nitty-gritty details, Information Theory provides a framework with which to simultaneously compare multiple competing models by weighting the performance of a model by its complexity (i.e. the number of model parameters). Say for instance that we are comparing five regression models of the effect of various factors on species richness from the *dune* and *dune.env* data-sets in the *vegan* package:

### MODELS

1. Interaction - Richness = Moisture + Manure + Moisture x Manure
2. No Interaction - Richness = Moisture + Manure
3. Moisture Only - Richness = Moisture
4. Manure Only - Richness = Manure
5. Null (intercept Only) - Richness = Intercept

We can get our richness estimate from the *dune* data by using the `specnumber` function:

```
> library(vegan)
> data(dune)
> data(dune.env)
> R = specnumber(dune)
```

Once we have our data and our models of interest. The steps for conducting the analysis are:

1. Fit the models (here we use the `aov` function)
2. Generate likelihood based information criteria (in this case Akaike's Information Criterion (AIC))
3. Calculate Akaike Weights (W) (here we right a function to do this for use)

```
> model1 <- aov(R ~ Moisture + Manure + Moisture * Manure,
+ data = dune.env)
> model2 <- aov(R ~ Moisture + Manure, data = dune.env)
> model3 <- aov(R ~ Moisture, data = dune.env)
> model4 <- aov(R ~ Manure, data = dune.env)
> model5 <- lm(R ~ 1, data = dune.env)

> aic.values <- c(AIC(model1), AIC(model2), AIC(model3),
+ AIC(model4), AIC(model5))

> weights <- function(ic = "vector of information criterion values") {
+ d <- numeric()
+ for (i in 1:length(ic)) {
+ d[i] <- ic[i] - min(ic)
+ }
+ exp.d <- numeric()
+ for (i in 1:length(d)) {
+ exp.d[i] <- exp(-d[i]/2)
+ }
+ w <- numeric()
+ for (i in 1:length(exp.d)) {
+ w[i] <- exp.d[i]/sum(exp.d)
+ }
+ model <- seq(1, length(ic))
+ out <- data.frame(model, ic, d, exp.d, w)
+ colnames(out) <- c("Model #", "Info. Crit.", "D", "Exp(-D/2)",
+ "Weights")
+ out <- out[order(out$Weights, decreasing = TRUE), ]
+ rownames(out) = seq(1, nrow(out))
+ return(out)
+ }
> w.out <- weights(aic.values)
> w.out
```

Model #	Info. Crit.	D	Exp(-D/2)	Weights
1	1	85.69719	0.000000	1.000000000 0.977879763
2	4	94.94671	9.249522	0.009805996 0.009589085
3	5	94.95225	9.255069	0.009778840 0.009562530
4	2	98.66867	12.971484	0.001525029 0.001491295
5	3	98.68749	12.990304	0.001510745 0.001477327

From this we can see that the clear winner is Model 1 or the Interaction Model, which has the highest Akaike weight.