

Introduction to Ecological Analysis in R (Day 3)

Matthew Lau

November 14, 2008

1. Community Analysis Continued
 - (a) Multivariate Tests for Differences in Community Composition
 - i. AnoSim
 - ii. MRPP
 - iii. PerMANOVA
 - (b) Indicator Species Analysis
2. Looping: Getting R to Do Many Individual Tests

1 Community Analysis Continued

The data file "CommData.csv" has both our species abundance data and our grouping data combined in one matrix. First we need to separate the two:

```
> com.data <- read.csv("CommData.csv")
> com <- com.data[, -1]
> env <- factor(com.data[, 1])
```

1.1 Multivariate Tests for Differences in Community Composition

We now have a vector "env", which is formatted as a factor using factor, containing the grouping data and a matrix "com" composed of our species abundance data. Here are three ways to test for differences in composition between our two groups using commands available in the *vegan* package, which we installed last time:

```
> library(vegan)
```

1.1.1 Analysis of Similarity (AnoSim)

For this test, we first need to convert our species abundance data into dissimilarities (i.e. distances) using the `vegdist` function. We then use the `anosim` function on our distances specifying our groups with our "env" data vector:

```
> dist.com <- vegdist(com, method = "bray")
> anosim(dis = dist.com, grouping = env)
```

```
Call:
anosim(dis = dist.com, grouping = env)
Dissimilarity: bray
```

```
ANOSIM statistic R:      1
Significance: < 0.001
```

Based on 1000 permutations

1.1.2 Multiple Response Permutation Procedure (MRPP)

Here, we basically do the exact same thing, except that we don't have to convert our abundance matrix to a distance matrix. The function will do the conversion for us. We just need to specify the distance metric to use:

```
> mrpp(com, env, distance = "bray")
```

```
Call:
mrpp(dat = com, grouping = env, distance = "bray")
```

```
Dissimilarity index: bray
Weights for groups:  n
```

```
Chance corrected within-group agreement A: 0.4165
Based on observed delta 0.2852 and expected delta 0.4887
```

```
Significance of delta: < 0.001
Based on 1000 permutations
```

1.1.3 Last, there is a little used permutation multivariate analysis of variance test (PerMANOVA) that is useful when you are interested in testing for the effect of more than one factor. The specification looks very much that same as a regular ANOVA of regression except that we use the function, adonis, and we need to specify the number of permutations:

```
> adonis(com ~ env, permutations = 1000)
```

```
Call:
adonis(formula = com ~ env, permutations = 1000)
```

	Df	SumsOfSqs	MeanSqs	F.Model	R2	Pr(>F)
env	1.000000	1.903327	1.903327	45.775944	0.7178	< 0.001 ***
Residuals	18.000000	0.748426	0.041579		0.2822	
Total	19.000000	2.651753			1.0000	

NOTE: the number of permutations directly effects the resulting p-value. This is because the p-value is calculated as the number of permutations that result in a F-statistic greater than or equal to our observed F-statistic divided by the total number of permutations. For instance, if you only ran ten permutations the smallest, non-zero, p-value that you could possibly get is 0.10 (i.e. one divided by ten). The rule of thumb is to run at least 200 permutations, but try and run as many as you can. Usually 1000 is sufficient.

1.2 Indicator Species Analysis

The above test will only tell us that there is a difference in composition. They don't tell us what this difference is. The most common method used for this is an Indicator Species Analysis. This will tell us the species that tend to be found in one of our groups versus another. Luckily, the *labdsv* package provides useful functions to do this:

```
> library(labdsv)
> IS <- duleg(com, env)
> summary(IS)
```

	cluster	indicator_value	probability
V1	1	0.9001	0.001
V10	1	0.8668	0.001
V4	1	0.8388	0.001
V8	1	0.8285	0.001
V6	1	0.8180	0.001
V7	1	0.7754	0.001
V2	1	0.7705	0.001
V9	1	0.7311	0.010
V3	1	0.7305	0.022
V21	2	0.9479	0.001
V11	2	0.9273	0.001
V27	2	0.9210	0.001
V23	2	0.8954	0.001
V25	2	0.8902	0.001
V28	2	0.8712	0.001
V20	2	0.8688	0.001
V19	2	0.8661	0.001
V17	2	0.8569	0.001
V15	2	0.8418	0.001
V18	2	0.8329	0.001
V12	2	0.8285	0.001
V30	2	0.8231	0.001
V26	2	0.8073	0.001
V16	2	0.8052	0.001
V14	2	0.7955	0.001
V22	2	0.7873	0.001
V24	2	0.7796	0.002
V13	2	0.7664	0.001

```
V29      2      0.7588      0.001
```

```
Sum of probabilities = 0.213
```

```
> detach(package:labdsv)
```

This removes all of the *labdsv* functions from the R library.

2 Looping: Getting R to Do Many Individual Tests

There are some situations in which we would like to perform a task repeatedly. We can get R to do a repeat task or set of tasks by doing what is called "looping" in programming lingo. To create a "loop", all we need to know is how to use the `for` function. Essentially, the structure of a "for-loop" says, "for this many steps, do this task." For instance, if we wanted to have R add up the numbers one to ten, we could code it with a for-loop:

```
> a = 0
> for (i in 1:10) {
+   a = a + 1
+ }
> a

[1] 10
```

This is a very simple task, and we most likely would never do this, but it illustrates all you need know to construct almost any for-loop, which is a very powerful thing. Note that inside of the curly-brackets, we are telling R to add one to the object, "a", for ten steps. That is R adds one to "a" then does it again to the "new" value of "a" generated by the last step, and so on, until it has done it ten times. Also note, we created "a" *outside* of the loop. This is important because it needs to be created before we can do anything with it, and if we have it inside the loop, R will make a new "a" at each step.

Now, say for instance our advisor asks us to run tests for the effect of our environmental factor on all of our species. This could take a long time or a lot of code lines to this. However, for-loops can save the day. We simply write the test into a for-loop properly and, *voilà*, R will do all of tests for us:

```
> results <- list()
> for (i in 1:ncol(com)) {
+   results[i] <- summary(aov(com[, i] ~ env))
+ }
> results[1:10]
```

```

[[1]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1 146890  146890  24.496 0.0001036 ***
Residuals 18 107938    5997
---

```

```

[[2]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1  69502   69502  24.208 0.0001104 ***
Residuals 18  51679    2871
---

```

```

[[3]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1  36722   36722   8.0937 0.01075 *
Residuals 18  81668    4537
---

```

```

[[4]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1 112650  112650  47.359 1.95e-06 ***
Residuals 18  42816    2379
---

```

```

[[5]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1  16416   16416   5.3427 0.03286 *
Residuals 18  55309    3073
---

```

```

[[6]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1  78250   78250  76.257 6.885e-08 ***
Residuals 18  18470    1026
---

```

```

[[7]]
      Df Sum Sq Mean Sq F value    Pr(>F)
env      1  65322   65322  34.603 1.435e-05 ***
Residuals 18  33979    1888
---

```

```

[[8]]

```

```

          Df Sum Sq Mean Sq F value    Pr(>F)
env          1 135137  135137  126.88 1.388e-09 ***
Residuals   18  19171    1065
---

```

```
[[9]]
```

```

          Df Sum Sq Mean Sq F value    Pr(>F)
env          1  57459   57459  14.535 0.001275 **
Residuals   18  71156    3953
---

```

```
[[10]]
```

```

          Df Sum Sq Mean Sq F value    Pr(>F)
env          1 193258  193258  50.243 1.313e-06 ***
Residuals   18  69236    3846
---

```

You'll notice that we first made a new list object called, "results". We then wrote the for-loop, which consists of two lines:

1. The number of loops
2. The function to be repeated, which say run this test of the column of com equal to "i" (i.e. the loop or step number) and save it as the "ith" component of the list "results"

For-loops will do anything you tell them to and can be extremely useful. Play around and try and make it part of your working R vocabulary.