Fir fecundity example using ADMB via R2admb

Mollie Brooks

February 10, 2014

Introduction

These models are found in chapter 6 of *Ecological Models and Data in R* by Bolker 2008 (hereafter EMD book) and the data originally appeared in Silvertown and Dodd 1999, and Dodd and Silvertown 2000. The data on *Abies balsamea* includes counts of the cones produced, measurements of tree size (diameter at breast height, DBH), and the status (wave / non-wave) indicating wave-like die-offs experienced by some populaitons.

The mean fir fecundity is predicted to follow a power-law and the number of cones should be negative binomially distributed around the mean.

```
\mu = a DBH <sup>b</sup> cones ~NegativeBinomial(\mu, k)
```

In this example, I present three versions of this model: (1) the parameters are constant across wave and non-wave populations; (2) a and b vary between wave and non-wave populations; (3) a, b, and k vary between wave and non-wave populations. I'll describe version 1 in the most detail and then compare and contrast versions 2 and 3.

In many cases, ADMB can converge without the user specifying initial values. Without user-specified starting values, it uses 0 for unbounded numbers and the midpoint of bounds for bounded numbers. However, in this example, better initial values were needed. This was also the case in R. Versions 2 and 3 converge when given the starting conditions from version 1. In general, it is more robust to start complex models using parameter estimates from simplified models.

1 Constant Parameters

1.1 ADMB Code

The ADMB code for model version 1 (in a file called fir0.tpl) looks like this:

1 DATA_SECTION

```
2
            init_int nobs;
3
            init_vector totcones(1, nobs);
4
            init_vector dbh(1, nobs);
   PARAMETER_SECTION
5
6
            init_number a;
 7
            init_number b;
8
            init\_bounded\_number k(0.01,1000);
9
            vector mu(1, nobs); //store predictions in here
10
            objective_function_value nll;
11
   PROCEDURE_SECTION
12
            mu=a*pow(dbh, b);
13
            nll=dnbinom(totcones, mu, k);
```

1.1.1 DATA_SECTION

This is where you define and initialize data objects. It is typical to first define data objects, such as nobs, that control the size of data objects defined further down in the code. This is so that the same .tpl file can be used on different data sets without making changes to the code. Lines 3 and 4 define vectors with indices labeled 1 through nobs. There cannot be any spaces between the numbers in parentheses, although the typesetting of this document might make it look like there are spaces.

1.1.2 PARAMETER_SECTION

This is where you define and initialize parameters to be fit. These will start with <code>init_</code>. Also in this section, you can define objects, such as <code>mu</code> where you'll store calculated values. The last thing in this section is the <code>objective_function_value</code> which will get minimized. I like to call it <code>nll</code> for 'negative log-likelihood'.

1.1.3 PROCEDURE_SECTION

This is where the negative log-likelihood is calculated and stored in the objective_function_value. We calculate the mean mu as a power-law function with parameters a and b on line 12. Then line 13 calculates the negative log-likelihood of the data as being negative binomially distributed. You can find out how the function is parameterized by going to http://admb-project.org/documentation/api/and searching for 'dnbinom'. The function dnbinom takes a mean and overdispersion parameter. The oversdispersion parameter k must be bounded greater than 0 (line 8).

1.1.4 GLOBALS_SECTION

This is where we include any libraries that we want to use in our calculations. In this example, we use the function dnbinom which is contained in statsLib.h.

1.2 R Packages (if you want to use R to organize the data and read the results)

```
library(emdbook)
## Loading required package:
                             MASS
## Loading required package:
                             lattice
## Loading required package: plyr
library(R2admb)
setup_admb()
## [1] "/Users/molliebrooks1/admb-trunk/build/dist/"
sessionInfo()
## R version 3.0.2 (2013-09-25)
## Platform: x86_64-apple-darwin10.8.0 (64-bit)
##
## locale:
## [1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/c/en_GB.UTF-8/en_GB.UTF-8
##
## attached base packages:
## [1] stats graphics grDevices utils datasets methods
                                                                  base
##
## other attached packages:
## [1] R2admb_0.7.10
                                      plyr_1.8
                                                  lattice_0.20-23
                     emdbook_1.3.4
## [5] MASS_7.3-29
                      knitr_1.5
## loaded via a namespace (and not attached):
## [1] evaluate_0.5.1 formatR_0.10 grid_3.0.2
                                                   stringr_0.6.2
## [5] tools_3.0.2
```

The command setup_admb() finds the location of ADMB so that it can be used by R2admb.

1.3 Data

Data organization is the same as in the EMD book.

```
data(FirDBHFec)
X = na.omit(FirDBHFec[, c("TOTCONES", "DBH", "WAVE_NON")])
X$TOTCONES = round(X$TOTCONES)
```

Then we use write_dat() to put the values for the DATA_SECTION of our .tpl file into a .dat file. The order of the objects in the list must match lines 2 through 4 of the .tpl file. write_pin() puts the initial values for our parameters into a .pin file. The order of these objects must match lines 6 through 8 of our .tpl file. mu and nll are the only objects in our PARAMETER_SECTION that do not need to be initialized because they are calculated in the PROCEDURE_SECTION. Also, note that mu and nll are the only objects not beginning with init_

```
write_dat("fir0", L = list(nobs = nrow(X), totcones = X$TOTCONES, dbh = X$DBH))
write_pin("fir0", L = list(a = 1, b = 1, k = 10))
```

1.4 Running the model via R2admb

First we must compile the code. Then run it.

```
compile_admb("fir0", verbose = TRUE)
## compiling with args: ' ' ...
## compile output:
     *** Parse tpl: fir0.tpl tpl2cpp fir0 *** Compile: fir0.cpp c++ -c -03 -DSAFE_ALL -D__GNU
## compile log:
run_admb("fir0", verbose = TRUE)
## running compiled executable with args: ' '...
## Run output:
##
##
##
##
##
## Initial statistics: 3 variables; iteration 0; function evaluation 0; phase 1
                    5.4476447e+03; maximum gradient component mag
## Function value
## Var
         Value
                  Gradient
                             |Var
                                    Value
                                             Gradient
                                                        |Var
                                                               Value
                                                                        Gradient
       1.00000 -5.10792e+03 | 2 1.00000 -1.17080e+04 | 3 -0.87253
##
                                                                       4.49545e+04
## Intermediate statistics: 3 variables; iteration 10; function evaluation 22; phase 1
                    1.1390151e+03; maximum gradient component mag
## Function value
                  Gradient
                             |Var
                                    Value
                                             Gradient
                                                        Var
                                                               Value
    1 0.27202 -2.64374e+01 | 2 2.35965 -1.42706e+01 | 3 -1.04377 1.10051e+03
```

```
## 3 variables; iteration 20; function evaluation 33; phase 1
## Function value
                   1.1360298e+03; maximum gradient component mag -2.0628e+01
        Value
                 Gradient
                             Var
                                   Value
                                            Gradient
                                                       Var
                                                              Value
       0.28720 -6.74336e+00 |
                                  2.34340 -3.42434e+00 |
                                                          3 -1.04932 -2.06283e+01
    1
    ic > imax in fminim is answer attained ?
                 Gradient
                            |Var
                                   Value
                                                              Value
## Var
        Value
                                            Gradient
                                                       Var
    1 0.30359 2.32540e-05 | 2 2.31984 1.19350e-05 | 3 -1.04921 -4.05320e-04
## Function minimizer not making progress ... is minimum attained?
## Minimprove criterion =
                           0.0000e+00
##
  - final statistics:
## 3 variables; iteration 28; function evaluation 70
## Function value
                  1.1360e+03; maximum gradient component mag -4.0532e-04
                                   1.0000e-04
## Exit code = 1; converg criter
                            |Var
## Var
        Value
                 Gradient
                                            Gradient
                                                              Value
                                   Value
                                                       |Var
##
    1 0.30359 2.32540e-05
                               2 2.31984
                                           1.19350e-05 | 3 -1.04921 -4.05320e-04
## Estimating row 1 out of 3 for hessian
## Estimating row 2 out of 3 for hessian
## Estimating row 3 out of 3 for hessian
```

I always check the final statistics 'Function value' and 'maximum gradient component'; the former should be a real number and the latter should be less than 10^{-4} .

1.5 Reading the results via R2admb

```
m0_admb = read_admb("fir0")
summary(m0_admb)
## Model file: fir0
## Negative log-likelihood: 1136.0
                                       AIC:
                                              2278.0
## Coefficients:
     Estimate Std. Error z value Pr(>|z|)
##
## a
        0.304
                    0.121
                             2.51
                                     0.012 *
## b
        2.320
                   0.186
                            12.49
                                    <2e-16 ***
        1.503
                   0.143
                            10.54
## k
                                    <2e-16 ***
## Signif. codes:
                   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The function summary performs Wald tests on the parameter estimates.

2 Allow a and b to vary between wave and non-wave populations

For the model with a and b varying across wave and non-wave populaitons the GLOBALS_SECTION is the same as above. The rest of the code (in a file called fir_ab.tpl) looks like this:

```
DATA_SECTION
 1
2
            init_int nobs;
3
            init_vector totcones(1, nobs);
4
            init_vector dbh(1, nobs);
            init_matrix wave_non(1, nobs, 1, 2);
 5
6
   PARAMETER SECTION
7
            init\_vector a\_coeffs(1,2);
            init\_vector b\_coeffs(1,2);
8
9
            init\_bounded\_number k(0.01,1000);
            vector a(1, nobs);
10
11
            vector b(1, nobs);
12
            vector mu(1, nobs); //store predictions in here
13
            objective_function_value nll;
   PROCEDURE SECTION
14
15
            a=wave_non * a _ coeffs;
16
            b=wave_non*b_coeffs;
17
            mu=elem_prod(a, pow(dbh, b));
            nll=dnbinom(totcones, mu, k);
18
```

2.1 Differences in the code compared to version 1

2.1.1 PARAMETER_SECTION

Now a and b are vectors of coefficients. The first element is the coefficient for the non-wave populations and the second element is the offset for the wave populations from that baseline of the non-wave populations. Lines 10 and 11 allocate vectors where we will store the value of a or b associated with each observation.

2.1.2 PROCEDURE_SECTION

Lines 15 and 16 use matrix algebra to put the coefficient associated with each observation into vectors \mathbf{a} and \mathbf{b} . Line 17 does element wise multiplication ($\mathbf{elem_prod}$) so that the correct value of \mathbf{a} for each observation is used to predict the mean of that observation μ . In ADMB the default multiplication is matrix multiplication, unlike in R. We didn't need to use $\mathbf{elem_prod}$ in version 1 where \mathbf{a} is a scalar. The function \mathbf{pow} always does element wise operations.

2.2 Data

For this version, we also need to know which observations were from wave and non-wave populations, so the data includes a model matrix. This model matrix has the non-wave populations as the baseline and then models the wave populations with an offset. Using model matrices makes code run more efficiently.

2.3 Initial Values

We'll use the coefficients from version 1 as starting values. We initialize the offsets to be 0 (the null hypothesis).

```
coef0 = coef(m0_admb)
write_pin("fir_ab", L = list(c(coef0["a"], 0, coef0["b"], 0, coef0["k"])))
```

2.4 Compile and Run

Let's use the default verbose=FALSE this time.

```
compile_admb("fir_ab")
run_admb("fir_ab")
```

2.5 View the results

```
mab_admb = read_admb("fir_ab")
summary(mab_admb)
## Model file: fir_ab
## Negative log-likelihood:
                                              2281.4
                              1135.7
                                        AIC:
## Coefficients:
##
              Estimate Std. Error z value Pr(>|z|)
## a_coeffs.1
                  0.288
                             0.181
                                       1.59
                                                 0.11
                                                 0.69
## a_coeffs.2
                  0.120
                             0.305
                                       0.40
## b_coeffs.1
                  2.355
                             0.281
                                       8.38
                                               <2e-16 ***
```

This tells us that neither a nor b differ between wave or non-wave populations.

3 Allow a, b, and k to vary between wave and non-wave populations

Again, the GLOBALS_SECTION is the same as above. Also, the DATA_SECTION is the same as in version 2. The ADMB code for the model with a, b, and k varying across wave and non-wave populaitons (in a file called fir_abk.tpl) looks like this:

```
PARAMETER SECTION
 1
 2
            init_vector a_coeffs(1,2);
3
            init_vector b_coeffs(1,2);
 4
            init\_bounded\_number k\_non(0.01,1000);
            init_number k_diff;
5
6
            vector k_coeffs(1,2);
7
            vector a(1, nobs);
            vector b(1, nobs);
8
9
            vector k(1, nobs);
10
            vector mu(1, nobs); //store predictions in here
11
            objective_function_value nll;
   PROCEDURE SECTION
12
13
            k_c coeffs(1)=k_non;
14
            k_c coeffs(2)=k_d iff;
15
            a=wave_non * a _ coeffs;
            b=wave_non*b_coeffs;
16
17
            k=wave_non*k_coeffs;
18
            mu=elem_prod(a, pow(dbh, b));
19
            nll=dnbinom(totcones, mu, k);
```

3.1 Differences in the code compared to version 2

3.1.1 PARAMETER_SECTION

Here, we use parameter k_non for the baseline parameter of non-wave populations and k_diff for the difference between wave and non-wave populations. Unlike with a and b, we have to make baseline k and the offset of k be separate objects because the baseline of k must be greater than 0, but the offset can be any value. Then, to be efficient using matrix multiplication, we create a

vector k_coeffs where we'll store the bounded baseline k_non and the unbounded offset k_diff. Alternatively, we could have created a bounded vector containing a separate parameter for each population, but then it would be more difficult to test if the difference is non-zero. Also, that parameterization would require a different model matrix.

As with, a and b, we create a vector to store the different value of k associated with each observation (line 9).

3.1.2 PROCEDURE_SECTION

On lines 13 and 14, we store the baseline and offset values of k for the non-wave and wave populations into first and second elements of the vector k_coeffs. Then we can use k_coeffs in the same way we use a_coeffs and b_coeffs to pull out the correct parameter value associated with each observation. On line 19, dnbinom is able to accept a vector of k values. If you were to look at the documentation for dnbinom at http://admb-project.org/documentation/api/ you would see that there are different (overloaded) versions written for different data and parameter types; C++ automatically uses the appropriate version if one exists.

3.2 Initial Values and Data

Now there are 2 values for k. We initialize the offset to be 0 (the null hypothesis).

3.3 Compile and Run

```
compile_admb("fir_abk")
run_admb("fir_abk")
```

3.4 View the results

```
mabk_admb = read_admb("fir_abk")
summary(mabk_admb)

## Model file: fir_abk
## Negative log-likelihood: 1135.0 AIC: 2282.0
```

```
## Coefficients:
##
             Estimate Std. Error z value Pr(>|z|)
## a_coeffs.1 0.288
                           0.173
                                    1.67
                                           0.096 .
## a_coeffs.2 0.122
                           0.313
                                    0.39
                                           0.697
             2.355
                           0.269
## b_coeffs.1
                                    8.77
                                         < 2e-16 ***
## b_coeffs.2 -0.208
                           0.418
                                   -0.50
                                           0.619
                           0.203
                                         4.2e-16 ***
## k_non
               1.654
                                   8.13
## k_diff
               -0.330
                           0.283
                                   -1.16
                                           0.244
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.05 '.' 0.1 ' ' 1
```

A Wald test might not be appropriate for testing if k_diff differs from 0 because the Wald test assumes the parameter is symmetric.

4 log-likelihoods

```
logLik(m0_admb)

## 'log Lik.' -1136 (df=3)

logLik(mab_admb)

## 'log Lik.' -1136 (df=5)

logLik(mabk_admb)

## 'log Lik.' -1135 (df=6)
```

A comparison of these log-likelihood values shows that adding parameters doesn't significantly improve the model fit, agreeing with the Wald test.

5 Comparison with R

```
library(bbmle)
## Loading required package: stats4
##
```

```
## Attaching package: 'bbmle'
##
## The following object is masked from 'package:R2admb':
##
      stdEr
nbfit.0 = mle2(TOTCONES ~ dnbinom(mu = a * DBH^b, size = k), start = list(a = 1,
   b = 1, k = 1), data = X)
## Warning: NaNs produced
## Warning:
            NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning:
            NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning:
            NaNs produced
## Warning:
            NaNs produced
## Warning: NaNs produced
## Warning: NaNs produced
## Warning:
            NaNs produced
start.ab = as.list(coef(nbfit.0))
nbfit.ab = mle2(TOTCONES ~ dnbinom(mu = a * DBH^b, size = k), start = start.ab,
    data = X, parameters = list(a ~ WAVE_NON, b ~ WAVE_NON))
## Warning: NaNs produced
## Warning:
            NaNs produced
## Warning:
            NaNs produced
## Warning: NaNs produced
all.equal(coef(m0_admb), coef(nbfit.0))
```

```
all.equal(coef(mab_admb), coef(nbfit.ab))
## [1] "Names: 4 string mismatches"
## [2] "Mean relative difference: 0.0008365"
nbfit.abk = mle2(TOTCONES ~ dnbinom(mu = a * DBH^b, size = k), start = start.ab,
   data = X, parameters = list(a ~ WAVE_NON, b ~ WAVE_NON, k ~ WAVE_NON))
## Warning: NaNs produced
## Warning: NaNs produced
## Warning:
            NaNs produced
            NaNs produced
## Warning:
            NaNs produced
## Warning:
## Warning:
            NaNs produced
## Warning:
            NaNs produced
all.equal(coef(mabk_admb), coef(nbfit.abk))
## [1] "Names: 6 string mismatches"
## [2] "Mean relative difference: 0.0002164"
```

6 Conclusion

We got the same parameter estimates and log-likelihood as the example in the EMD book. It might seem like this isn't worth the extra effort, but if you wanted to increase the complexity of the model, include any random effects, or do MCMC sampling, then the flexibility of ADMB would be a selling point.