Tutorial for the dcglm package

Péter Sólymos
University of Alberta

Abstract

This tutorial package to demonstrate the capabilities of data cloning algorithm via the infrastructure provided by the **dclone** package. Functions reproduce main features of the `glm` base function in R by using data cloning.

Keywords: Bayesian statistics, data cloning, maximum likelihood inference, generalized linear models, R.

1. Introduction

Data cloning is a statistical computing method introduced by Lele *et al.* (2007). It exploits the computational simplicity of the Markov chain Monte Carlo (MCMC) algorithms used in the Bayesian statistical framework, but it provides valid frequentist inferences such as the maximum likelihood estimates and their standard errors for complex hierarchical models. The use of the data cloning algorithm is especially straightforward for complex models, where the number of unknowns increases with sample size (i.e. mixed models), because inference and prediction procedures are often hard to implement in such situations.

The **dclone** R package (Sólymos 2009) aims to provide low level functionality to easily implement more specific higher level procedures based on data cloning for users familiar with the Bayesian methodology. This tutorial, we develop write high level functions to duplicate the some features of the `glm` base function of R by using the data cloning algorithm building on the infrastructure of the **dclone** package.

2. Data generation

We generate random data for Poisson and Binomal GLMs. First we define the number of locations (`n`) and the independent covariate (`x`). `X` represents the design matrix:

```
R> library(dclone)
R> set.seed(1234)
R> n <- 20
R> x <- runif(n, -1, 1)
R> X <- model.matrix(~x)
```

Parameters (`beta1`), linear predictor (`mu1`) and random response (`Y1`) for the Poisson case (log link function):

```
R> beta1 <- c(2, -1)
```

```
R> mu1 <- X %*% beta1
R> Y1 <- rpois(n, exp(mu1))
```

Parameters (`beta2`), linear predictor (`mu2`) and random response (`Y2`) for the Binomial (Bernoulli) case (logistic link function):

```
R> beta2 <- c(0, -1)
R> mu2 <- X %*% beta2
R> Y2 <- rbinom(n, 1, exp(mu2) / (1 + exp(mu2)))
```

3. GLM based on the glm function

Now we fit the Poisson and Binomail GLM by using the `glm` base function and inspect their summaries:

```
R> m1 <- glm(Y1 ~ x, family=poisson)
R> summary(m1)

Call:
glm(formula = Y1 ~ x, family = poisson)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.6336  -0.8138  -0.2134   0.5154   2.0309

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.94162    0.09482  20.478  < 2e-16 ***
x           -1.27648    0.16017  -7.969 1.60e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 90.897  on 19  degrees of freedom
Residual deviance: 19.060  on 18  degrees of freedom
AIC: 99.543

Number of Fisher Scoring iterations: 4

R> m2 <- glm(Y2 ~ x, family=binomial)
R> summary(m2)

Call:
glm(formula = Y2 ~ x, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7850  -0.8787  -0.5794   1.0077   1.5424

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
```

```
(Intercept)  -0.3388     0.5021  -0.675    0.500
x            -1.7377     1.0186  -1.706    0.088 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 27.526  on 19  degrees of freedom
Residual deviance: 24.097  on 18  degrees of freedom
AIC: 28.097

Number of Fisher Scoring iterations: 4
```

4. The full Bayesian model for GLM and data cloning

Here is the JAGS model for the Poisson case, with flat Normal priors for the regression coefficients:

```
R> glm.pois <- function() {
+      for (i in 1:n) {
+          Y[i] ~ dpois(lambda[i])
+          log(lambda[i]) <- inprod(X[i,], beta[1,])
+      }
+      for (j in 1:np) {
+          beta[1,j] ~ dnorm(0, 0.001)
+      }
+   }
```

The data will be represented as a list (if we want to use data cloning consistently, we have to use the list format instead of the global environment):

```
R> dat1 <- list(n = length(Y1), Y = Y1, X = X, np = ncol(X))
R> str(dat1)

List of 4
 $ n : int 20
 $ Y : num [1:20] 14 3 2 4 5 6 25 22 3 9 ...
 $ X : num [1:20, 1:2] 1 1 1 1 1 1 1 1 1 1 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:20] "1" "2" "3" "4" ...
  .. ..$ : chr [1:2] "(Intercept)" "x"
  ..- attr(*, "assign")= int [1:2] 0 1
 $ np: int 2
```

Now let's clone the data set (note that `n` should be multiplies, while `np` must remain unchanged):

```
R> n.clones <- 5
R> dcdat1 <- dclone(dat1, n.clones, multiply = "n", unchanged = "np")
R> str(dcdat1)

List of 4
 $ n : atomic [1:1] 100
```

```
  ..- attr(*, "n.clones")= atomic [1:1] 5
  .. ..- attr(*, "method")= chr "multi"
 $ Y : atomic [1:100] 14 3 2 4 5 6 25 22 3 9 ...
  ..- attr(*, "n.clones")= atomic [1:1] 5
  .. ..- attr(*, "method")= chr "rep"
 $ X : num [1:100, 1:2] 1 1 1 1 1 1 1 1 1 1 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:100] "1_1" "2_1" "3_1" "4_1" ...
  .. ..$ : chr [1:2] "(Intercept)" "x"
  ..- attr(*, "n.clones")= atomic [1:1] 5
  .. ..- attr(*, "method")= chr "rep"
 $ np: int 2
```

To fit the model to the data with data cloning is as easy as this:

```
R> mod1 <- jags.fit(dcdat1, "beta", glm.pois)

R> summary(mod1)

Iterations = 1001:6000
Thinning interval = 1
Number of chains = 3
Sample size per chain = 5000
Number of clones = 5

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

          Mean      SD  DC SD  Naive SE Time-series SE R hat
beta[1]  1.941 0.04200 0.0939 0.0003429      0.0005987 1.001
beta[2] -1.277 0.07142 0.1597 0.0005831      0.0010080 1.001

2. Quantiles for each variable:

          2.5%    25%    50%    75%  97.5%
beta[1]  1.857  1.912  1.941  1.969  2.022
beta[2] -1.415 -1.325 -1.278 -1.228 -1.136
```

Le's compare this `mcmc.list` (more accurately an `mcmc.list.dc`) object with the `glm` results:

```
R> cbind(true.values=beta1,
+       glm.estimates=coef(m1), glm.se=summary(m1)$coefficients[,2],
+       dc.estimates=coef(mod1), dc.se=dcsd(mod1))

            true.values glm.estimates     glm.se dc.estimates     dc.se
(Intercept)           2      1.941624 0.09481704     1.940619 0.0939060
x                    -1     -1.276479 0.16017431    -1.276619 0.1596982
```

Here is the JAGS model for the Binomial (Bernoulli, because k=1) case:

```
R> glm.bin <- function() {
+       for (i in 1:n) {
+           Y[i] ~ dbin(p[i], k)
+           logit(p[i]) <- inprod(X[i,], beta[1,])
+       }
+       for (j in 1:np) {
+           beta[1,j] ~ dnorm(0, 0.001)
+       }
+   }
```

Putting together the data set is similar to the Poisson case:

```
R> dat2 <- list(n = length(Y2), Y = Y2, k = 1, X = X, np = ncol(X))
R> str(dat2)

List of 5
 $ n : int 20
 $ Y : num [1:20] 1 0 0 1 0 1 0 1 0 0 ...
 $ k : num 1
 $ X : num [1:20, 1:2] 1 1 1 1 1 1 1 1 1 1 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:20] "1" "2" "3" "4" ...
  .. ..$ : chr [1:2] "(Intercept)" "x"
  ..- attr(*, "assign")= int [1:2] 0 1
 $ np: int 2
```

but data cloning setup is a bit different. We don't have to repeat the data vectors or columns in the design matrix `n.clones` times, because there is an easier way. We multiply `Y` and `k` with `n.clones` and leave the other elements unchanged:

```
R> dcdat2 <- dclone(dat2, n.clones, multiply = c("Y","k"), unchanged = c("n", "np", "X"))
R> str(dcdat2)

List of 5
 $ n : int 20
 $ Y : atomic [1:20] 5 0 0 5 0 5 0 5 0 0 ...
  ..- attr(*, "n.clones")= atomic [1:1] 5
  .. ..- attr(*, "method")= chr "multi"
 $ k : atomic [1:1] 5
  ..- attr(*, "n.clones")= atomic [1:1] 5
  .. ..- attr(*, "method")= chr "multi"
 $ X : num [1:20, 1:2] 1 1 1 1 1 1 1 1 1 1 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:20] "1" "2" "3" "4" ...
  .. ..$ : chr [1:2] "(Intercept)" "x"
  ..- attr(*, "assign")= int [1:2] 0 1
 $ np: int 2
```

Now fit the model to the data with data cloning:

```
R> mod2 <- jags.fit(dcdat2, "beta", glm.bin)

R> summary(mod2)

Iterations = 1001:6000
Thinning interval = 1
Number of chains = 3
Sample size per chain = 5000
```

```
Number of clones = 5

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

           Mean     SD  DC SD Naive SE Time-series SE R hat
beta[1] -0.3502 0.2299 0.5141 0.001877       0.002585 1.002
beta[2] -1.7937 0.4584 1.0250 0.003743       0.004930 1.001

2. Quantiles for each variable:

           2.5%     25%     50%     75%    97.5%
beta[1] -0.8119 -0.5007 -0.3482 -0.1962  0.09812
beta[2] -2.7214 -2.0912 -1.7828 -1.4752 -0.92590
```

and compare results with `glm` results:

```
R> cbind(true.values=beta2,
+      glm.estimates=coef(m2), glm.se=summary(m2)$coefficients[,2],
+      dc.estimates=coef(mod2), dc.se=dcsd(mod2))

            true.values glm.estimates    glm.se dc.estimates     dc.se
(Intercept)           0    -0.3388286 0.5020926   -0.3501787 0.5140773
x                    -1    -1.7377202 1.0185540   -1.7937179 1.0249985
```

5. The custommodel function

The `custommodel` function enables us to resuse the same JAGS model with minor modifications. For example we combine the above Poisson and Binomial model into one.

```
R> glm.model <- function() {
+      for (i in 1:n) {
+          Y[i] ~ dpois(lambda[i])
+          Y[i] ~ dbin(p[i], k)
+          log(lambda[i]) <- inprod(X[i,], beta[1,])
+          logit(p[i]) <- inprod(X[i,], beta[1,])
+      }
+      for (j in 1:np) {
+          beta[1,j] ~ dnorm(0, 0.001)
+      }
+  }
```

If we want to use this in the `jags.fit` function, it would give the error message about the attempt to define the nodes more than onece. To avoid this, we tell the function which lines ahould be excluded:

```
R> custommodel(glm.model, c(4,6))

[1] "model {"
[2] "    for (i in 1:n) {"
[3] "        Y[i] ~ dpois(lambda[i])"
[4] "        log(lambda[i]) <- inprod(X[i,], beta[1,])"
[5] "    }"
[6] "    for (j in 1:np) {"
[7] "        beta[1,j] ~ dnorm(0, 0.001)"
[8] "    }"
[9] "}"
attr(,"class")
[1] "custommodel"
```

```
R> custommodel(glm.model, c(3,5))
```

```
[1] "model {"
[2] "    for (i in 1:n) {"
[3] "        Y[i] ~ dbin(p[i], k)"
[4] "        logit(p[i]) <- inprod(X[i,], beta[1,])"
[5] "    }"
[6] "    for (j in 1:np) {"
[7] "        beta[1,j] ~ dnorm(0, 0.001)"
[8] "    }"
[9] "}"
attr(,"class")
[1] "custommodel"
```

so eventually we get back our original models. But these are not functions, but character vectors of the class '`custommodel`'. `jags.fit` will recognize this.

Why do we want to complicate our lives with the `custommodel`? Because `dpois`, `dbin`, and `inprod` are not recognised as valid R objects or functions. So if our aim is to make an R package that passes the rather strict `R CMD check`, so won't be published at the Comprehensive R Archive Network (CRAN). A way to overcome this situation is to define fake objects as e.g. `inprod <- function() NULL`, but this option should be regaded as ugly and inefficient (unnecessary) as compared to a clean `custommodel` approach that will be presented in the next section.

6. The main function dcglm

Here our main function for the data cloning based estimating procedure for the Poisson and Binomial GLMs:

```
R> dcglm <- function(formula, data = parent.frame(), family=c("poisson", "binomial"), n.clones=5,
+       glm.model <- c("model {",
+                      "    for (i in 1:n) {",
+                      "        Y[i] ~ dpois(lambda[i])",
+                      "        Y[i] ~ dbin(p[i], k)",
+                      "        log(lambda[i]) <- inprod(X[i,], beta[1,])",
+                      "        logit(p[i]) <- inprod(X[i,], beta[1,])",
+                      "    }",
+                      "    for (j in 1:np) {",
+                      "        beta[1,j] ~ dnorm(0, 0.001)",
+                      "    }",
+                      "}")
+       family <- match.arg(family)
+       lhs <- formula[[2]]
+       Y <- eval(lhs, data)
+       formula[[2]] <- NULL
+       rhs <- model.frame(formula, data)
+       X <- model.matrix(attr(rhs, "terms"), rhs)
+       dat <- list(n = length(Y), Y = Y, X = X, np = ncol(X), k = 1)
+       if (family == "poisson") {
+           model <- model <- custommodel(glm.model, c(4,6))
+           dcdat <- dclone(dat, n.clones, multiply = "n", unchanged = "np")
+       } else {
+           model <- custommodel(glm.model, c(3,5))
+           dcdat <- dclone(dat, n.clones, multiply = c("Y","k"), unchanged = c("n", "np", "X"))
+       }
+       mod <- jags.fit(dcdat, "beta", model, ...)
+       COEF <- coef(mod)
+       SE <- dcsd(mod)
+       names(COEF) <- names(SE) <- colnames(X)
+       mu <- X %*% COEF
+       if (family == "poisson") {
+           fitval <- drop(exp(mu))
+           ll <- sum(log(fitval^Y * exp(-fitval)) - log(factorial(Y)))
+       } else {
+           fitval <- drop(exp(mu) / (1 + exp(mu)))
+           ll <- sum(log(choose(1, Y) * fitval^Y * (1-fitval)^(1-Y)))
+       }
+       rval <- list(call=match.call(),
+           mcmc = mod,
+           y = Y,
+           x = rhs,
+           model = X,
+           fitted.values = fitval,
+           linear.predictors = mu,
+           formula = formula,
+           coefficients = COEF,
+           std.error = SE,
+           loglik = ll,
+           family = family,
+           df.residual = length(Y) - length(COEF),
+           df.null = length(Y) - 1)
+       class(rval) <- c("dcglm")
+       rval
+   }
```

Let'g go through it step-by-step as pseudo-code:

1. `glm.model` is the `custommodel` version of the BUGS model, unifying the Poisson and Binomial cases, as we have seen before.
2. The `family` argument is recognized, and as a result, it can be given not only in full (e.g. `family = "p"` is equivalent of `family = "poisson"`).
3. `lhs` is the left-hand-side of the formula, `Y` is the value as a result of evaluating `lhs` in `data` (that is the parent frame, which is usually the global environment if not called from inside of a function).

4. `formula[[2]] <- NULL` removes the left-hand-side from the formula.

5. `rhs` is the right-hand-side, that is a model frame with variables defined in `data`.

6. The design matrix `X` is a result of using the `"terms"` attribute of `rhs` and evaluated in `rhs`.

7. `dat` is the Bayesian data representation.

8. The `model` and the data cloned data representation (`dcdat`) depends on the `family` argument.

9. `mod` is the fitted `mcmc.list` object. Dots (`...`) represents all the additional arguments that can be passed, including `n.update`, `n.iter`, and `n.chains`.

10. `COEF` is the `coef` method evaluated on the `mcmc.list` object `mod`. `SD` is the data cloned standard error (scaled by $\sqrt{k}$). Names of `COEF` and `SD` follow column names of `X`.

11. `mu` is the linear predictor (on log/logit scale), while `fitval` is the fitted value (response scale after using the appropriate inverse link function) and `ll` is the log-likelihood calculated from the probability mass function.

12. `rval` is the return value, that is a list with elements commonly applied in objects representing model fit (cf. for example element names with `names(m1)`):

    call the function call,

    mcmc the fitted `mcmc.list` object,

    y the response,

    x the model frame (right-hand-side),

    model the design matrix,

    fitted.values fitted values,

    linear.predictors linear predictors,

    formula the formula argument of the call,

    coefficients means of the joint posterior distribution (maximum likelihood estimates),

    std.error standard errors of the MLE,

    loglik log-likelihood,

    family family argument of the call,

    df.residual residual degrees of freedom,

    df.null degrees of freedom in the null model.

13. Finally, we attach the class attribute and return `rval`.

Fun, isn't it? See if it is actually working:

```
R> dcm1 <- dcglm(Y1 ~ x)
R> dcm2 <- dcglm(Y2 ~ x, family = "binomial")
```

If we are about to inspect these objects, well, it is a mess without some additional helper functions. The most basic such functions (called methods in R jargon) are `print` and `summary`. For our convenience, we also define some other methods, too. These are based on the so called S3 method dispatch system. That is, if a generic function is defined, we can add class specific methods to it.

In our case, the most simple methods are the `coef` and `fitted`, because these only extract an element from the objects:

```
R> coef.dcglm <- function(object, ...) object$coefficients
R> fitted.dcglm <- function(object, ...) object$fitted.values
```

Compare with the `glm` results:

```
R> rbind(glm=coef(m1), dcglm=coef(dcm1))

      (Intercept)         x
glm      1.941624 -1.276479
dcglm    1.939917 -1.278057

R> rbind(glm=coef(m2), dcglm=coef(dcm2))

      (Intercept)         x
glm    -0.3388286 -1.737720
dcglm  -0.3515468 -1.792584

R> rbind(glm=fitted(m1), dcglm=fitted(dcm1))

             1        2        3        4        5        6        7
glm   18.68691 5.100808 5.273269 5.086763 2.773811 4.871574 24.38239
dcglm 18.67779 5.090142 5.262459 5.076110 2.765928 4.861112 24.37850
             8        9       10       11       12       13       14
glm   13.79645 4.561353 6.721032 4.252018 6.214003 12.13746 2.364617
dcglm 13.78455 4.551187 6.709265 4.242173 6.202523 12.12506 2.357432
            15       16       17       18       19       20
glm   11.84414 2.946219 12.02980 12.64069 15.50880 13.80789
dcglm 11.83169 2.938065 12.01739 12.62842 15.49766 13.79599

R> rbind(glm=fitted(m2), dcglm=fitted(dcm2))

              1         2         3         4         5         6         7
glm   0.7317898 0.3178060 0.3276999 0.3169928 0.1689386 0.3043920 0.7967153
dcglm 0.7375729 0.3121664 0.3222797 0.3113356 0.1617195 0.2984722 0.8032913
              8         9        10        11        12        13        14
glm   0.6435203 0.2857649 0.4041127 0.2666583 0.3786844 0.6025899 0.1405842
dcglm 0.6473257 0.2794916 0.4006783 0.2600701 0.3745399 0.6052502 0.1335849
             15        16        17        18        19        20
glm   0.5945881 0.1807789 0.5996819 0.6157557 0.6791673 0.6437790
dcglm 0.5970133 0.1735307 0.6022571 0.6187958 0.6838679 0.6475913
```

For the `logLik` method, it is necessary to follow the standard rules, because AIC calculations depend on this method (this means, that we don't have to define a method for AIC if the `logLik` method exists for a class):

```
R> logLik.dcglm <- function (object, ...)
+       structure(object$loglik,
+           df = object$df.null + 1 - object$df.residual,
+           nobs = object$df.null + 1,
+           class = "logLik")
```

Compare with the `glm` results:

```
R> logLik(m1)

'log Lik.' -47.77174 (df=2)

R> logLik(dcm1)

'log Lik.' -47.7719 (df=2)

R> logLik(m2)

'log Lik.' -12.04875 (df=2)

R> logLik(dcm2)

'log Lik.' -12.05029 (df=2)

R> AIC(m1, dcm1, m2, dcm2)

     df      AIC
m1    2 99.54347
dcm1  2 99.54380
m2    2 28.09749
dcm2  2 28.10058
```

Now it is possible to write the `print` method:

```
R> print.dcglm <- function(x, digits = max(3, getOption("digits") - 3), ...) {
+       cat("\nCall: ", deparse(x$call), "\n\n")
+       cat("Coefficients:\n")
+       print.default(format(x$coefficients, digits = digits), print.gap = 2, quote = FALSE)
+       cat("\nDegrees of Freedom:", x$df.null, "Total (i.e. Null); ", x$df.residual, "Residual\n"
+       cat("Log Likelihood:\t   ", format(signif(x$loglik, digits)), "\n")
+       invisible(x)
+   }
```

Let's have a look at the resulting objects of our `dcglm` function:

```
R> dcm1

Call:  dcglm(formula = Y1 ~ x)

Coefficients:
(Intercept)            x
      1.940       -1.278

Degrees of Freedom: 19 Total (i.e. Null);  18 Residual
Log Likelihood:            -47.77
```

```
R> dcm2

Call:  dcglm(formula = Y2 ~ x, family = "binomial")

Coefficients:
(Intercept)            x
    -0.3515      -1.7926

Degrees of Freedom: 19 Total (i.e. Null);  18 Residual
Log Likelihood:            -12.05
```

Well done so far!

7. Methods for inference

The `summary` method returns the ML estimates, data cloning standard errors, and Wald-type z statistics and p-values:

```
R> summary.dcglm <- function(object, ...){
+       COEF <- coef(object)
+       SE <- object$std.error
+       z <- COEF / SE
+       p <-  2 * pnorm(-abs(z))
+       stab <- cbind("Estimate" = COEF, "Std. Error" = SE,
+           "z value" = z, "Pr(>|z|)" = p)
+       rval <- list(call = object$call,
+           coefficients = stab,
+           loglik = object$loglik,
+           df.residual = object$df.residual,
+           df.null = object$df.null)
+       class(rval) <- "summary.dcglm"
+       rval
+   }
```

The return value here is also a list, repeating some of the elements of the fitted `object`. To appropriately format the summary, we use the `print` method for the object class '`summary.dcglm`':

```
R> print.summary.dcglm <-
+   function (x, digits = max(3, getOption("digits") - 3),
+       signif.stars = getOption("show.signif.stars"), ...)
+   {
+       cat("\nCall:\n")
+       cat(paste(deparse(x$call), sep = "\n", collapse = "\n"), "\n", sep = "")
+       cat("\nCoefficients:\n")
+       printCoefmat(x$coefficients, digits = digits, signif.stars = signif.stars, na.print = "NA"
+       cat("\nDegrees of Freedom:", x$df.null, "Total (i.e. Null); ", x$df.residual, "Residual\n"
+       cat("Log Likelihood:\t   ", format(signif(x$loglik, digits)), "\n")
+       invisible(x)
+   }
```

Summaries of the `glm()` results and our models:

```
R> summary(m1)
```

```
Call:
glm(formula = Y1 ~ x, family = poisson)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.6336  -0.8138  -0.2134   0.5154   2.0309

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.94162    0.09482  20.478  < 2e-16 ***
x           -1.27648    0.16017  -7.969 1.60e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 90.897  on 19  degrees of freedom
Residual deviance: 19.060  on 18  degrees of freedom
AIC: 99.543

Number of Fisher Scoring iterations: 4

R> summary(dcm1)

Call:
dcglm(formula = Y1 ~ x)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.93992    0.09343  20.763  < 2e-16 ***
x           -1.27806    0.15840  -8.069 7.11e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Degrees of Freedom: 19 Total (i.e. Null);  18 Residual
Log Likelihood:            -47.77

R> summary(m2)

Call:
glm(formula = Y2 ~ x, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7850  -0.8787  -0.5794   1.0077   1.5424

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.3388     0.5021  -0.675    0.500
x            -1.7377     1.0186  -1.706    0.088 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 27.526  on 19  degrees of freedom
Residual deviance: 24.097  on 18  degrees of freedom
AIC: 28.097

Number of Fisher Scoring iterations: 4
```

```
R> summary(dcm2)

Call:
dcglm(formula = Y2 ~ x, family = "binomial")

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.3515     0.5083  -0.692   0.4892
x            -1.7926     1.0388  -1.726   0.0844 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Degrees of Freedom: 19 Total (i.e. Null);  18 Residual
Log Likelihood:            -12.05
```

Piece of cake!

For the `confint`, we use the asymptotic normality result of the data cloning theory (Lele *et al.* 2007), and the `confint` method defined for the data cloned `mcmc.list` part of the fitted model object:

```
R> confint.dcglm <- function(object, parm, level = 0.95, ...) {
+      rval <- confint(object$mcmc, parm, level, ...)
+      rownames(rval) <- names(coef(object))
+      rval
+  }
```

The 95% confidence intervals for the model estimates are:

```
R> confint(m1)

                2.5 %    97.5 %
(Intercept)  1.748410  2.120651
x           -1.596540 -0.968007
```

```
R> confint(dcm1)

                2.5 %     97.5 %
(Intercept)  1.756798  2.1230355
x           -1.588514 -0.9675994
```

```
R> confint(m2)

                2.5 %     97.5 %
(Intercept) -1.404579 0.62478977
x           -4.014429 0.09589234
```

```
R> confint(dcm2)

                2.5 %    97.5 %
(Intercept) -1.347883 0.6447896
x           -3.828686 0.2435175
```

Differences are due to the fact, that `confint` for `glm` uses profile likelihood, while `dcglm` confidence intervals are based on the asymptotic normality assumption. Profile likelihood can be computed based on data cloning (Ponciano *et al.* 2009) but that procedure is not covered here.

8. Prediction based on the joint posterior distribution

In the prediction, we use MCMC. The likelihood part of the BUGS model for the prediction is the same as for the estimation. The only difference is in the prior specification:

```
R> glm.pred <- function() {
+       for (i in 1:n) {
+           Y[i] ~ dpois(z[i])
+           Y[i] ~ dbin(z[i], k)
+           log(z[i]) <- mu[i]
+           logit(z[i]) <- mu[i]
+           mu[i] <- inprod(X[i,], beta[1,])
+       }
+       beta[1,1:np] <- mvn[1:np]
+       mvn[1:np] ~ dmnorm(coefs[], prec[,])
+   }
```

Note that we denote `lambda` or `p` as `z`, this will make life easier later. We use again the `custommodel` approach to differentiate between the Poisson and Binomial cases:

```
R> custommodel(glm.pred, c(4,6))

[1] "model {"
[2] "    for (i in 1:n) {"
[3] "        Y[i] ~ dpois(z[i])"
[4] "        log(z[i]) <- mu[i]"
[5] "        mu[i] <- inprod(X[i,], beta[1,])"
[6] "    }"
[7] "    beta[1,1:np] <- mvn[1:np]"
[8] "    mvn[1:np] ~ dmnorm(coefs[], prec[,])"
[9] "}"
attr(,"class")
[1] "custommodel"

R> custommodel(glm.pred, c(3,5))

[1] "model {"
[2] "    for (i in 1:n) {"
[3] "        Y[i] ~ dbin(z[i], k)"
[4] "        logit(z[i]) <- mu[i]"
[5] "        mu[i] <- inprod(X[i,], beta[1,])"
[6] "    }"
[7] "    beta[1,1:np] <- mvn[1:np]"
[8] "    mvn[1:np] ~ dmnorm(coefs[], prec[,])"
[9] "}"
attr(,"class")
[1] "custommodel"
```

Let's consider the Poisson case only (the Binomial differs from it only by the specification of the `model` argument based on the `custommodel` approach, and the fitted model used). The prediction can be done by `jags.fit`, only the data specification is somewhat different. We will define the model parameters based on the MLE (`coefs`) and the variance-covariance matrix. We define a Multivariate Normal node for all the model parameters, by using the inverse of the variance-covariance matrix as a precision matrix (`prec`). Be careful, the check for symmetry in JAGS is stricter than the usual numerical precision in R, consequently we ensure that this condition is met by using the `make.symmetric` function. The data specification will look like (note, we are using the observed data in `X`, but algorithmically, this doesn't make any difference):

```
R> prec <- make.symmetric(solve(vcov(mod1)))
R> coefs <- coef(mod1)
R> prdat <- list(n = nrow(X), X = X,
+       np = ncol(X), k = 1, coefs = coefs, prec = prec)
```

We use the `jags.fit` function. One chain is usually enough:

```
R> prval <- jags.fit(prdat, "z", custommodel(glm.pred, c(4,6)), n.chains = 1)
```

The resuling `mcmc.list` object contains the conditional posterior distribution for our Poisson GLM based prediction with prediction intervals.

9. Methods for prediction

For our convenience, we can write a `vcov` method. We simply use the `vcov` method defined for the `mcmc.list` part of the fitted model object and do some cosmetics on the names:

```
R> vcov.dcglm <- function(object, ...) {
+       rval <- vcov(object$mcmc, ...)
+       rownames(rval) <- colnames(rval) <- names(coef(object))
+       rval
+   }
```

Comparison of the `glm` and `dcglm` approaches:

```
R> vcov(m1)

            (Intercept)           x
(Intercept) 0.008990272 0.009590252
x           0.009590252 0.025655809

R> vcov(dcm1)

            (Intercept)          x
(Intercept) 0.008729083 0.00914548
x           0.009145480 0.02509039
```

```
R> vcov(m2)

            (Intercept)         x
(Intercept)   0.2520969 0.1053385
x             0.1053385 1.0374523

R> vcov(dcm2)

            (Intercept)         x
(Intercept)   0.2584138 0.1168975
x             0.1168975 1.0792020
```

Quite similar as we expected.

The `predict` function will look like:

```
R> predict.dcglm <- function(object, newdata = NULL, type = c("link", "response"), se = FALSE, ...
+      glm.pred <- c("model {",
+            "    for (i in 1:n) {",
+            "        Y[i] ~ dpois(z[i])",
+            "        Y[i] ~ dbin(z[i], k)",
+            "        log(z[i]) <- mu[i]",
+            "        logit(z[i]) <- mu[i]",
+            "        mu[i] <- inprod(X[i,], beta[1,])",
+            "    }",
+            "    beta[1,1:np] <- mvn[1:np]",
+            "    mvn[1:np] ~ dmnorm(coefs[], prec[,])",
+            "    }")
+      prec <- make.symmetric(solve(vcov(object)))
+      coefs <- coef(object)
+      if (is.null(newdata)) {
+          X <- object$model
+      } else {
+          rhs <- model.frame(object$formula, newdata)
+          X <- model.matrix(attr(rhs, "terms"), rhs)
+      }
+      type <- match.arg(type)
+      params <- switch(type,
+          "link" = "mu",
+          "response" = "z")
+      model <- switch(object$family,
+          "poisson" = custommodel(glm.pred, c(4,6)),
+          "binomial" = custommodel(glm.pred, c(3,5)))
+      prdat <- list(n = nrow(X), X = X,
+          np = ncol(X), k = 1, coefs = coefs, prec = prec)
+      prval <- jags.fit(prdat, params, model, ...)
+      if (!se) {
+          rval <- coef(prval)
+      } else {
+          rval <- list(fit = coef(prval),
+              se.fit = mcmcapply(prval, sd))
+      }
+      rval
+  }
```

The pseudo-code for `predict` is:

1. `glm.predict` is the familiar `custommodel` specification.
2. `prec` and `coefs` is needed for the data specification.
3. If `newdata` is `NULL`, we use the extracted design matrix ou our fitted model (`object`). Else, we create the design matrix corresponding to our model from `newdata` (a data frame, containing the same covariates, but possibly with different values). For this extraction, we use the `formula` of the fitted model `object`.
4. Based on the `type` argument, we will monitor (sample) the nodes `mu` (if `type = "link"`) or `z` (if `type = "response"`). `mu` corresponds to the values on the scale of the linear predictors, while `z` corresponds to the values on the response scale.
5. `model` is determined by the `family` of the fitted model `object`.
6. `prdat` is the data, `prmod` is the fitted MCMC object.
7. If the `se` argument is `FALSE`, the return value will be the point estimate vector of the prediction. If the `se` argument is `TRUE`, the return value will be a list including point estimates (`fit`) and standard errors (`se.fit`). Then, return thevalue.

Now let's do the prediction for a range of `x` values from -1 to 1 (call it `px`):

```
R> px <- data.frame(x=seq(-1, 1, len = 10))
R> px

            x
1  -1.0000000
2  -0.7777778
3  -0.5555556
4  -0.3333333
5  -0.1111111
6   0.1111111
7   0.3333333
8   0.5555556
9   0.7777778
10  1.0000000
```

The `glm` based predictions are:

```
R> pm1link <- predict(m1, newdata=px, type="link", se=TRUE)
R> pm1resp <- predict(m1, newdata=px, type="response", se=TRUE)
R> pm2link <- predict(m2, newdata=px, type="link", se=TRUE)
R> pm2resp <- predict(m2, newdata=px, type="response", se=TRUE)
```

The `dcglm` based predictions are:

```
R> pdcm1link <- predict(dcm1, newdata=px, type="link", se=TRUE)
R> pdcm1resp <- predict(dcm1, newdata=px, type="response", se=TRUE)
R> pdcm2link <- predict(dcm2, newdata=px, type="link", se=TRUE)
R> pdcm2resp <- predict(dcm2, newdata=px, type="response", se=TRUE)
```

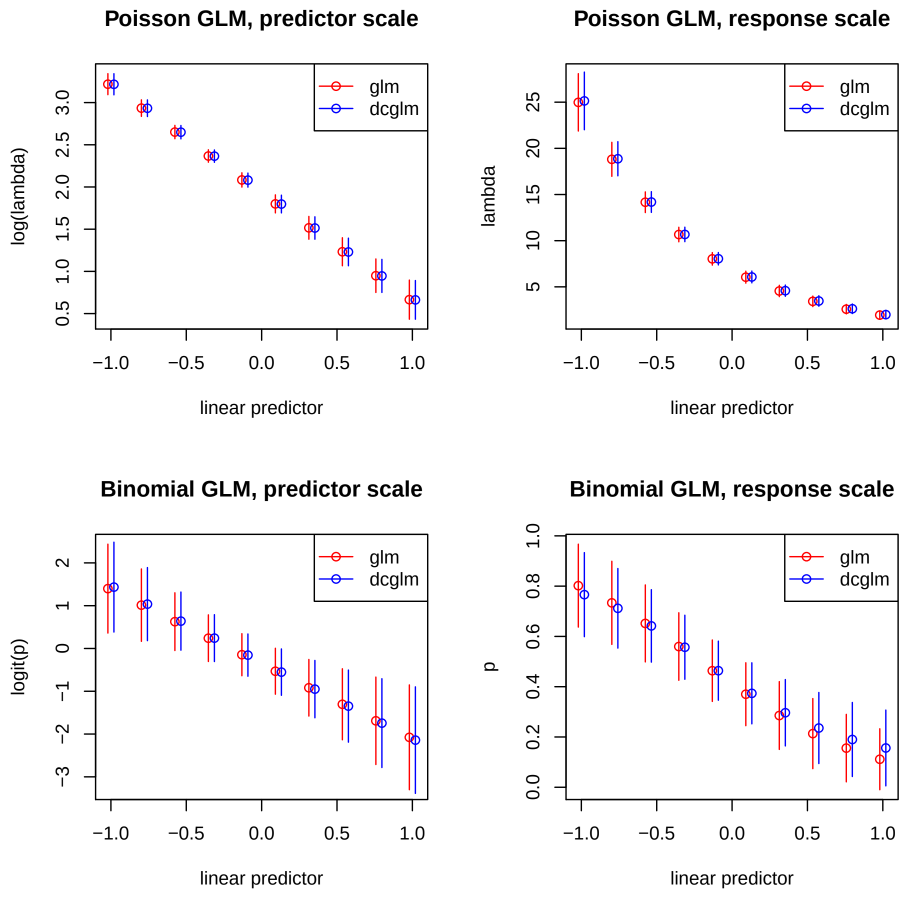

Figure 1: Prediction results based on the `glm` and `dcglm` approaches for the Poisson GLM. Points are prediction estimates, whiskers are prediction standard errors.

Fig. 1 shows prediction results.

10. Making the dcglm package

The easiest part now comes:

```
R> package.skeleton("dcglm", c("coef.dcglm","confint.dcglm","dcglm",
+        "fitted.dcglm","logLik.dcglm","predict.dcglm",
+        "print.dcglm","print.summary.dcglm","summary.dcglm","vcov.dcglm"))
```

Follow this workflow for your own model and estimating procedure, then edit the files (read the *Writing R Extensions* manual) in the package directory, run `R CMD check`, and ditribute your package.

References

Lele SR, Dennis B, Lutscher F (2007). "Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods." *Ecology Letters*, **10**, 551–563.

Ponciano JM, Taper ML, Dennis B, Lele SR (2009). "Hierarchical models in ecology: confidence intervals, hypothesis testing, and model selection using data cloning." *Ecology*, **90**, 356–362.

Sólymos P (2009). *dclone: Data Cloning and MCMC Tools for Maximum Likelihood Methods*. R package version 1.0-0, URL http://cran.r-project.org/packages=dclone.

Affiliation:

Péter Sólymos
Alberta Biodiversity Monitoring Institute
Department of Biological Sciences
CW 405, Biological Sciences Bldg
University of Alberta
Edmonton, Alberta, T6G 2E9, Canada
E-mail: solymos@ualberta.ca