The builtenvir package for linear DLM's

(With intro to OOP)

Andrew Whiteman

August 21, 2017

Contents

Intro to OOP

Object Oriented Programming

- So-called by analogy to objects in the real world
- For example, a physical door has a set of *attributes* like a handle or knob, a hinge-mechanism, the dimensions of the doorway, ...
- When you encounter a door, you always know just how to use it—it doesn't matter if it's the door to SPH, the door to your car, or the door on your oven, some similar action allows you to *open* it
- Programmers use this kind of abstraction to design efficient and easy to use software

Intro to OOP

Object Oriented Programming

- So-called by analogy to objects in the real world
- For example, a physical door has a set of *attributes* like a handle or knob, a hinge-mechanism, the dimensions of the doorway, ...
- When you encounter a door, you always know just how to use it—it doesn't matter if it's the door to SPH, the door to your car, or the door on your oven, some similar action allows you to *open* it
- Programmers use this kind of abstraction to design efficient and easy to use software

Intro to OOP

Object Oriented Programming

- So-called by analogy to objects in the real world
- For example, a physical door has a set of *attributes* like a handle or knob, a hinge-mechanism, the dimensions of the doorway, ...
- When you encounter a door, you always know just how to use it—it doesn't matter if it's the door to SPH, the door to your car, or the door on your oven, some similar action allows you to *open* it
- Programmers use this kind of abstraction to design efficient and easy to use software

Intro to OOP

Object Oriented Programming

- So-called by analogy to objects in the real world
- For example, a physical door has a set of *attributes* like a handle or knob, a hinge-mechanism, the dimensions of the doorway, ...
- When you encounter a door, you always know just how to use it—it doesn't matter if it's the door to SPH, the door to your car, or the door on your oven, some similar action allows you to *open* it
- Programmers use this kind of abstraction to design efficient and easy to use software

Intro to OOP

Object Oriented Programming

- So-called by analogy to objects in the real world
- For example, a physical door has a set of *attributes* like a handle or knob, a hinge-mechanism, the dimensions of the doorway, ...
- When you encounter a door, you always know just how to use it—it doesn't matter if it's the door to SPH, the door to your car, or the door on your oven, some similar action allows you to *open* it
- Programmers use this kind of abstraction to design efficient and easy to use software

Intro to OOP: Objects in R

Whether or not you've realized it, if you've used R, you've probably interacted with OOP design

For instance, take the "object" returned by `lm()`:

```
## simulate data:
set.seed(12345)
n <- 100
p <- 2
x <- rnorm(n)
b <- rnorm(p)  # beta coefs
sigma.sq <- 1
y <- rnorm(n, cbind(1, x) %*% b, sqrt(sigma.sq))

fit <- lm(y ~ x)
```

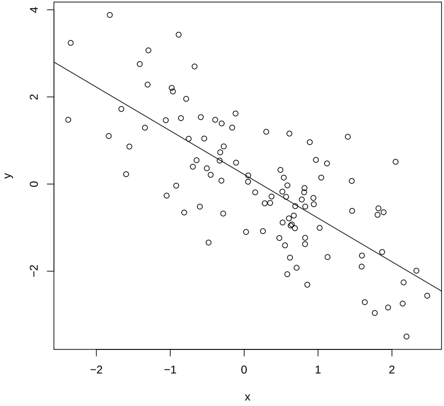
Intro to OOP: Objects in R

Whether or not you've realized it, if you've used R, you've probably interacted with OOP design
For instance, take the "object" returned by `lm()`:

```
## simulate data:
set.seed(12345)
n <- 100
p <- 2
x <- rnorm(n)
b <- rnorm(p)  # beta coefs
sigma.sq <- 1
y <- rnorm(n, cbind(1, x) %*% b, sqrt(sigma.sq))

fit <- lm(y ~ x)
```

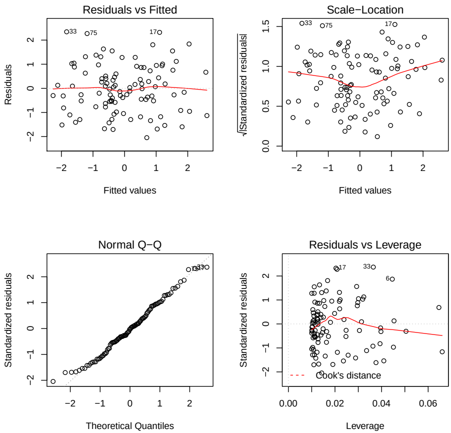
Intro to OOP: Objects in R

```
> fit

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.2201      -1.0024
```



Intro to OOP: Objects in R

```
> plot(fit)
```

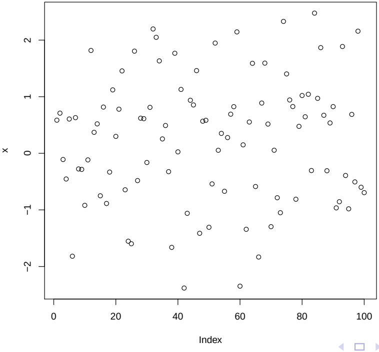


Intro to OOP: Objects in R

And many more functions exist to interact with the objects `lm()` returns...

How does R know exactly what to do with these commands? And why does the output of `plot(fit)` look different from ⇓?

```
> plot(x)
```



Intro to OOP: Objects in R

The answer is because the R developers chose to organize these kinds of functions as *methods* that act on different objects [e.g. *open(door)*]

```
> class (fit)
[1] "lm"
>
> class (x)
[1] "numeric"
```

Intro to OOP: Objects in R

```
> methods(class = "lm")
 [1] add1           alias          anova          case.names     coerce
 [6] confint        cooks.distance deviance       dfbeta         dfbetas
[11] drop1          dummy.coef     effects        extractAIC     family
[16] formula        hatvalues      influence      initialize     kappa
[21] labels         logLik         model.frame    model.matrix   nobs
[26] plot           predict        print          proj           qr
[31] residuals      rstandard      rstudent       show           simulate
[36] slotsFromS3    summary        variable.names vcov
see '?methods' for accessing help and source code
```

Intro to OOP: Objects in R

Internally, when you type

```
> fit

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.2201      -1.0024
```

```
> getAnywhere(print.lm)
A single object matching print.lm was found
It was found in the following places
  registered S3 method for print from namespace stats
  namespace:stats
with value

function (x, digits = max(3L, getOption("digits") - 3L), ...)
{
    cat("\nCall:\n", paste(deparse(x$call), sep = "\n", collapse = "\n"),
        "\n\n", sep = "")
    if (length(coef(x))) {
        cat("Coefficients:\n")
        print.default(format(coef(x), digits = digits), print.gap = 2L,
            quote = FALSE)
    }
    else cat("No coefficients\n")
    cat("\n")
    invisible(x)
}
<bytecode: 0x7f8ccc3d5e00>
<environment: namespace:stats>
```

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Objects in R

- And the same with:
  - `plot(fit)` → `plot.lm`
  - `summary(fit)` → `summary.lm`
  - ...
- We can and should take advantage of this organization structure when we write R code
- Users should rightly expect `plot(`*fitted-model*`)` to behave similarly regardless of the specific program or type of model
- Not only can this approach produce user-friendly code, the structure can be more efficient in terms of what the *language* expects as well (more on this later)

Intro to OOP: Defining new R classes

- There are multiple ways of doing this in R, including: `S3`, `S4`, and Reference Classes
- Won't go into all these, but you can read more about them for example here
- Most of base R uses `S3` classes and methods; newer packages often use some combination of the three

Contents

builtenvir: Class Structure

The primary purpose of `builtenvir` at the moment is to provide users a way to fit Distributed Lag Models

Similarly to `stats::lm`, where the output is an object of class ``lm'', `builtenvir::dlm` outputs an object of class ``Dlm''. The goal is to make the syntax as classically R as possible:

```
library (builtenvir)
data (simdata)

lag <- seq(0.1, 10, length.out = 100)  # distances
X <- simdata[, -(1:3)]  # measurements binned by lag distance

fit <- dlm(Y ~ Age * Gender + cr(lag, X), data = simdata)
```

builtenvir: Class Structure

```
> fit
Call: dlm(formula = Y ~ Age * Gender + cr(lag, X), data = simdata)
(frequentist) distributed lag model fit via REML
Number of Observations: 1000
Log-Likelihood: -1261.647
Fixed Effects:
     (Intercept)       Age   Gender Age:Gender
[1,]     0.33080 -0.49092  1.15477     -0.009

          Random Effects Residuals
Std Error       0.016283    0.8299
>
> class(fit)
[1] "FreqDlm"
attr(,"package")
[1] "builtenvir"

> inherits(fit, "Dlm")
[1] TRUE
```

builtenvir: Class Structure

```
> fit
Call: dlm(formula = Y ~ Age * Gender + cr(lag, X), data = simdata)
(frequentist) distributed lag model fit via REML
Number of Observations: 1000
Log-Likelihood: -1261.647
Fixed Effects:
     (Intercept)      Age   Gender Age:Gender
[1,]     0.33080 -0.49092  1.15477     -0.009

          Random Effects Residuals
Std Error       0.016283    0.8299
>
> class(fit)
[1] "FreqDlm"
attr(,"package")
[1] "builtenvir"

> inherits(fit, "Dlm")
[1] TRUE
```

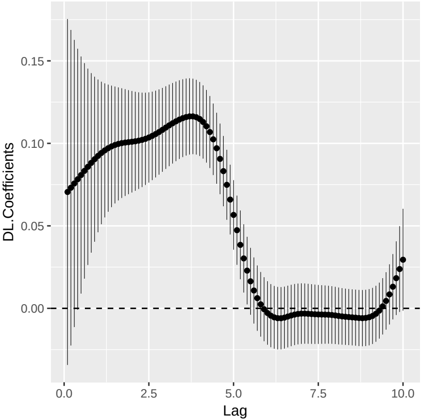
builtenvir: Class Structure

```
> summary(fit)
(frequentist) distributed lag model fit via REML
Call: dlm(formula = Y ~ Age * Gender + cr(lag, X), data = simdata)

Number of observations: 1000
Standardized Residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-3.25001 -0.70212 -0.02999  0.00000  0.68596  2.87362

    Random Effects Residuals
Var     0.00026513    0.6888

Fixed Effects:
              Estimate Std. Error   t value  Pr(>|t|)
(Intercept)  0.3307981  0.3600354    0.9188  0.358427
Age         -0.4909218  0.0048446 -101.3336 < 2.2e-16 ***
Gender       1.1547739  0.4331537    2.6660  0.007801 **
Age:Gender  -0.0089778  0.0066326   -1.3536  0.176175
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Correlation:
           (Intercept)      Age   Gender
Age           -0.88016
Gender        -0.65179  0.72642
Age:Gender     0.64604 -0.73193 -0.9926
```

builtenvir: Class Structure

Plot doesn't work exactly the same way (no diagnostics), but that's by design:

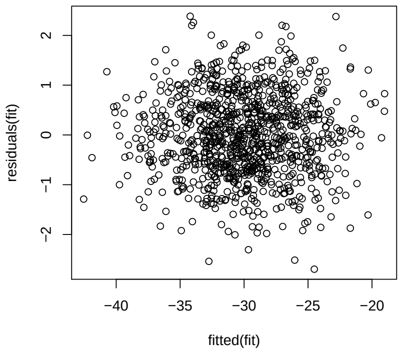
```
> plot(fit)
```



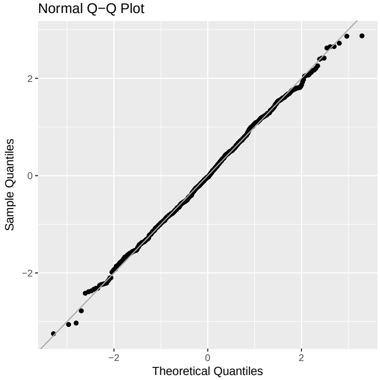
builtenvir: Class Structure

Can still get residuals and QQ plots easily from similar methods:

```
> plot(fitted(fit), residuals(fit))
```




```
> qqnorm(fit)
```



builtenvir: Lazy Programming

One of the first rules of good programming is to do no more work than necessary. This goes hand in hand with using a system of classes to be more consistent with what the *language* expects

Quick example: just by defining an R-consistent `logLik` method, we can get the functions `AIC` and `BIC` for free

```
logLik.Dlm <- function(object, ...) {
  ll <- object@logLik
  attr(ll, "nall") <- attr(ll, "nobs") <- object@N
  attr(ll, "df") <- object@K$fixed + 2
  ## fixed effects + 2 variance parameters
  class(ll) <- "logLik"
  return (ll)
}

> logLik(fit)
'log Lik.' -1261.647 (df=8)
>
> AIC(fit)
[1] 2539.294
> BIC(fit)
[1] 2578.556
```

builtenvir: Lazy Programming

One of the first rules of good programming is to do no more work than necessary. This goes hand in hand with using a system of classes to be more consistent with what the *language* expects

Quick example: just by defining an R-consistent `logLik` method, we can get the functions `AIC` and `BIC` for free

```
logLik.Dlm <- function(object, ...) {
  ll <- object@logLik
  attr(ll, "nall") <- attr(ll, "nobs") <- object@N
  attr(ll, "df") <- object@K$fixed + 2
  ## fixed effects + 2 variance parameters
  class(ll) <- "logLik"
  return (ll)
}

> logLik(fit)
'log Lik.' -1261.647 (df=8)
>
> AIC(fit)
[1] 2539.294
> BIC(fit)
[1] 2578.556
```

builtenvir: Lazy Programming

One of the first rules of good programming is to do no more work than necessary. This goes hand in hand with using a system of classes to be more consistent with what the *language* expects

Quick example: just by defining an R-consistent `logLik` method, we can get the functions `AIC` and `BIC` for free

```
logLik.Dlm <- function(object, ...) {
  ll <- object@logLik
  attr(ll, "nall") <- attr(ll, "nobs") <- object@N
  attr(ll, "df") <- object@K$fixed + 2
  ## fixed effects + 2 variance parameters
  class(ll) <- "logLik"
  return (ll)
}
```

```
> logLik(fit)
'log Lik.' -1261.647 (df=8)
>
> AIC(fit)
[1] 2539.294
> BIC(fit)
[1] 2578.556
```

builtenvir: Extending the Package

Ultimately, most of the methods defined for `stats::lm` and `nlme::lme` should probably be implemented for `builtenvir::Dlm` + a few extras

```
> methods(class = "Dlm")
 [1] coef       coerce<-   deviance   fitted     fixef      logLik     plot
 [8] predict    qqnorm     ranef      residuals scaleMat   se.fixef   se.ranef
[15] show       sigma      theta      vcov       vcovTheta
see '?methods' for accessing help and source code
```

builtenvir: Extending the Package

- We'll also want to have `gdlm`'s for logistic models, etc
- This should involve creation of `Gdlm` and `SummaryGdlm` classes and possibly sub-classes
- I've kept with the strategy of using `S4` classes and a combination of `S3` and `S4` methods defined on those classes

builtenvir: Extending the Package

- We'll also want to have `gdlm`'s for logistic models, etc
- This should involve creation of `Gdlm` and `SummaryGdlm` classes and possibly sub-classes
- I've kept with the strategy of using S4 classes and a combination of S3 and S4 methods defined on those classes

builtenvir: Extending the Package

- We'll also want to have `gdlm`'s for logistic models, etc
- This should involve creation of `Gdlm` and `SummaryGdlm` classes and possibly sub-classes
- I've kept with the strategy of using `S4` classes and a combination of `S3` and `S4` methods defined on those classes

I'd very much appreciate your help with beta-testing!

Thanks!