`predict` and Friends: Common Methods for Predictive Models in R

36-402, Spring 2015

Handout No. 1, 25 January 2015

R has lots of functions for working with different sort of predictive models. This handout reviews how they work with `lm`, and how they generalize to other sorts of models. We'll use the data from the first homework for illustration throughout:

```r
mob <- read.csv("http://www.stat.cmu.edu/~cshalizi/uADA/15/hw/01/mobility.csv")
```

Estimation Functions and Formulas

You know how to estimate a linear model in R: you use `lm`. For instance, this will regress the variable `Mobility` in the data frame `mob` on `Population`, `Seg_racial`, `Commute`, `Income` and `Gini`:

```r
mob.lm1 <- lm(mob$Mobility ~ mob$Population + mob$Seg_racial + mob$Commute + mob$Income + mob$Gini)
```

What `lm` returns is a complex object containing the estimated coefficients, the fitted values, a lot of diagnostic statistics, and a lot of information about exactly what work R did to do the estimation. We will come back to some of this later. The thing to focus on for now is the argument to `lm` in the line of code above, which tells the function exactly what model to estimate — it **specifies** the model. The R jargon term for that sort of specification is that it is the **formula** of the model.

While the line of code above works, it's not very elegant, because we have to keep typing `mob$` over and over. More abstractly, it runs specifying which variables we want to use (and how we want to use them) together with telling R where to look up the variables. This gets annoying if we want to, say, compare estimates of the same model on two different data sets (in this example, perhaps from different years). The solution is to separate the formula from the data source:

```r
mob.lm2 <- lm(Mobility ~ Population + Seg_racial + Commute + Income + Gini, data=mob)
```

(You should convince yourself, at this point, that `mob.lm1` and `mob.lm2` have the same coefficients, residuals, etc.)

The `data` argument tells `lm` to look up variable names appearing in the formula (the first argument) in a dataframe called `mob`. It therefore works even if there aren't variables in our workspace called `Mobility`, `Population`, etc., those just have to be column names in `mob`. In addition to being easier to write, read and re-use than our first effort, this format works better when we use the model for prediction, as explained below.

Transformations can also be part of the formula:

```r
mob.lm3 <- lm(Mobility ~ log(Population) + Seg_racial + Commute + Income + Gini, data=mob)
```

Formulas are so important that R knows about them as a special data type. They *look* like ordinary strings, but they *act* differently, so there are special functions for converting strings (or potentially other things) to formulas, and for manipulating them. For instance, if we want to keep around the formula with log-transformed population, we can do as follows:

```r
form.logpop <- "Mobility ~ log(Population) + Seg_racial + Commute + Income + Gini"
form.logpop <- as.formula(form.logpop)
mob.lm4 <- lm(form.logpop, data=mob)
```

(Again, convince yourself at this point that `mob.lm3` and `mob.lm4` are completely equivalent.)

(Being able to turn strings into formulas is very convenient if we want to try out a bunch of different model specifications, because R has lots of tools for building strings according to regular patterns, and then we can turn all those into formulas. There are some examples of this in the online code for lecture 3.)

If we have already estimated a model and want the formula it used as the specification, we can extract that with the `formula` function:

```r
formula(mob.lm3)
```

```
## Mobility ~ log(Population) + Seg_racial + Commute + Income +
##     Gini
```

```r
formula(mob.lm3) == form.logpop
```

```
## [1] TRUE
```

Extracting Coefficients, Confidence Intervals, Fitted Values, Residuals, etc.

If we want the coefficients of a model we've estimated, we can get that with the `coefficients` function:

```r
coefficients(mob.lm3)
```

```
##     (Intercept) log(Population)      Seg_racial         Commute
##       8.339e-02      -2.894e-03      -5.657e-02       1.451e-01
##          Income            Gini
##       1.772e-06      -1.622e-01
```

If we want confidence intervals for the coefficients, we can use `confint`:

```r
confint(mob.lm3,level=0.90) # default confidence level is 0.95
```

```
##                        5 %       95 %
## (Intercept)      3.611e-02  1.307e-01
## log(Population) -5.982e-03  1.934e-04
## Seg_racial      -8.479e-02 -2.835e-02
## Commute          1.132e-01  1.769e-01
## Income           1.298e-06  2.246e-06
## Gini            -1.988e-01 -1.255e-01
```

(This calculates confidence intervals assuming independent, constant-variance Gaussian noise everywhere, etc., etc., so it's not to be taken too seriously unless you've checked those assumptions somehow; see Chapter 2 of the notes, and Chapter 6 for alternatives.)

For every data point in the original data set, we have both a fitted value ($\hat{y}$) and a residual ($y - \hat{y}$). These are vectors, and can be extracted with the `fitted` and `residuals` functions:

```r
head(fitted(mob.lm2))
```

```
##       1       2       3       4       5       6
## 0.07048 0.06300 0.06926 0.04928 0.05792 0.06456
```

```r
head(fitted(mob.lm3))
```

```
##       1       2       3       4       5       6
## 0.06708 0.06500 0.06774 0.05266 0.06633 0.07133
```

```r
tail(residuals(mob.lm2))
```

```
##       736       737       738       739       740       741
## -0.045252 -0.031707  0.004027  0.015472 -0.025058  0.007091
```

```r
tail(residuals(mob.lm4))
```

```
##      736      737      738      739      740      741
## -0.04975 -0.03300 -0.01938  0.01415 -0.03119  0.02998
```

(I use `head` and `tail` here to keep from have to see hundreds of values.)

You may be more used to accessing all these things as parts of the estimated model — writing something like `mob.lm2$coefficients` to get the coefficients. This is fine as far as it goes, but we will work with many different sorts of statistical models in this course, and those internal names can change from model to model. If the people implementing the models did their job, however, functions like `fitted`, `residuals`, `coefficients` and `confint` will all, to the extent they apply, work, and work in the same way.

Methods and Classes (R-Geeky But Important)

In R things like `residuals` or `coefficients` are a special kind of function, called **methods**. Other methods, which you've used a lot without perhaps realizing it, are `plot`, `print` and `summary`. These are a sort of generic or meta- function, which looks up the class of model being used, and then calls a specialized function which how to work with that class. The convention is that the specialized function is named *method*`.`*class*, e.g., `summary.lm`. If no specialized function is defined, R will try to use *method*`.default`.

The advantage of methods is that you, as a user, don't have to learn a totally new syntax to get the coefficients or residuals of every new model class; you just use `residuals(mdl)` whether `mdl` comes from a linear regression which could have been done two centuries ago, or is a Batrachian Emphasis Machine which won't be invented for another five years. (It also means that core parts of R don't have to be re-written every time someone comes up with a new model class.) The one draw-back is that the help pages for the generic methods tend to be pretty vague, and you may have to look at the help for the class-specific functions — compare `?summary` with `?summary.lm`. (If you are not sure what the class of your model, `mdl`, is called, use `class(mdl)`.)

Making Predictions

The point of a regression model is to do prediction, and the method for doing so is, naturally enough, called `predict`. It works like so:

```r
predict(object, newdata)
```

Here `object` is an already estimated model, and `newdata` is a data frame containing the new cases, real or imaginary, for which we want to make predictions. The output is (generally) a vector, with a predicted value for each row of `newdata`. If the rows of `newdata` have names, those will be carried along as names in the output vector. Here, as a little example, we take our first specification, and get predicted values for every community in Alabama:

```r
predict(mob.lm2, newdata=mob[which(mob$State=="AL"),])
```

```
##      89      90      91     136     140     147     151     152     153 
## 0.06303 0.05805 0.06326 0.07347 0.04584 0.06507 0.06885 0.01799 0.03774 
##     154     156     157     158     159 
## 0.05232 0.03188 0.06477 0.03255 0.06408
```

It is important to remember that making a prediction does *not* mean "changing the data and re-estimating the model"; it means taking the unchanged estimate of the model, and putting in new values for the covariates or independent variables. (In terms of the linear model, we change x, not $\hat{\beta}$.)

Notice that I used `mob.lm2` here, rather than the mathematically-equivalent `mob.lm1`. Because I specified `mob.lm2` with a formula that just referred to column names, `predict` looks up columns with those names in `newdata`, puts them into the function estimated in `mob.lm2`, and calculates the predictions. Had I tried to use `mob.lm1`, it would have completely ignored `newdata`. This is one crucial reason why it is best to use clean formulas and a `data` argument when estimating the model.

If the formula specifies transformations, those will also be done on `newdata`; we don't have to do the transformations ourselves:

```r
predict(mob.lm3, newdata=mob[which(mob$State=="AL"),])
```

```
##      89      90      91     136     140     147     151     152     153 
## 0.06907 0.06257 0.06773 0.07561 0.05137 0.06849 0.07060 0.02782 0.04428 
##     154     156     157     158     159 
## 0.05772 0.03861 0.06774 0.04121 0.06765
```

The `newdata` does not have to be a subset of the original data used for estimation, or related to it in any way at all; it just has to have columns whose names match those in the right-hand side of the formula.

```r
predict(mob.lm3, newdata=data.frame(Population=1.5e6, Seg_racial=0, Commute=0.5,
                                    Income=3e4, Gini=median(mob$Gini)))
```

```
##      1 
## 0.1034
```

```r
predict(mob.lm3, newdata=data.frame(Population=1.5e6, Seg_racial=0, Commute=0.5,
                                    Income=quantile(mob$Income,c(0.05,0.5,0.95)),
                  Gini=quantile(mob$Gini,c(0.05,0.5,0.95))))
```

```
##     5%    50%    95% 
## 0.1123 0.1076 0.1025
```

(Explain what that last line does.)

A very common programming error is to run `predict` and get out a vector whose length equals the number of rows in the original estimation data, and which doesn't change no matter what you do to `newdata`. This is because if `newdata` is missing, or if R cannot find all the variables it needs in it, it defaults to giving us the predictions of the model on the original data. An even more annoying form of this error consists of forgetting that the argument is called `newdata` and not `data`:

```r
head(predict(mob.lm3)) # Equivalent to head(fitted(mob.lm3))
```

```
##       1       2       3       4       5       6
## 0.06708 0.06500 0.06774 0.05266 0.06633 0.07133
```

```r
head(predict(mob.lm3),data=data.frame(Population=1.5e6, Seg_racial=0, Commute=0.5,
                                Income=3e4, Gini=median(mob$Gini))) # Don't do this!
```

```
##       1       2       3       4       5       6
## 0.06708 0.06500 0.06774 0.05266 0.06633 0.07133
```

Returning the original fitted values when `newdata` is missing or messed up is not what I would have chosen, but nobody asked me.

Because `predict` is a method, the generic help file is fairly vague, and many options are only discussed on the help pages for the class-specific functions — compare `?predict` with `?predict.lm`. Common options include giving standard errors for predictions (as well point forecasts), and giving various sorts of intervals.

Using Different Model Classes

All of this carries over to different model classes, at least if they've been well-designed. For instance, suppose we want to estimating a kernel regression (as in chapter 4) to the same data, using the same variables. Here's how we do so:

```r
library(np)
# Pick bandwidth by automated cross-validation first
mob.npbw <- npregbw(formula=formula(mob.lm2), data=mob, tol=1e-2, ftol=1e-2)
# Now actually estimate
mob.np <- npreg(mob.npbw, data=mob)
# Would usually just do npreg(formula=formula(mob.lm2), data=mob, tol=1e-2, ftol=1e-2)
# but Markdown (not the command line!) didn't like it
mob.np <- npreg(mob.npbw, data=mob) # Now actually estimate
```

(See chapter 4 on the `tol` and `ftol` settings.)

We can re-use the formula, because it's just saying what the input and target variables of the regression are, and we want that to stay the same. More importantly, both `lm` and `npreg` use the same mechanism, of separating the formula specifying the model from the data set containing the actual values of the variables. (Of course, some models have variations in allowable formulas — interactions make sense for `lm` but not for `npreg`, the latter has a special way of dealing with ordered categorical variables that `lm` doesn't, etc.)

After estimating the model, we can do most of the same things to it that we could do to a linear model. We can look at a summary:

```r
summary(mob.np)
```

```
##
## Regression Data: 729 training points, in 5 variable(s)
##
## No. Complete Observations:  729 No. NA Observations:  12
## Observations omitted:  374 376 386 410 440 459 485 542 613 616 637 652
##               Population Seg_racial Commute Income    Gini
## Bandwidth(s):     185719     0.1625 0.04097   2343 0.03418
##
## Kernel Regression Estimator: Local-Constant
## Bandwidth Type: Fixed
## Residual standard error: 0.02996
## R-squared: 0.6795
##
## Continuous Kernel Type: Second-Order Gaussian
## No. Continuous Explanatory Vars.: 5
```

We can look at fitted values and residuals:

```r
head(fitted(mob.np))
```

```
## [1] 0.06519 0.06703 0.07541 0.05335 0.05767 0.06793
```

```r
tail(residuals(mob.np))
```

```
##        736        737        738        739        740        741
## -4.536e-02 -3.548e-02 -1.990e-07  2.508e-02 -8.133e-03  4.193e-05
```

and we can make predictions:

```r
predict(mob.np, newdata=data.frame(Population=1.5e6, Seg_racial=0, Commute=0.5,
                                Income=3e4, Gini=median(mob$Gini)))
```

```
## [1] 0.07756
```