Programming Guidelines and Linear Models R Basics (A review... Hopefully)

DJM, Revised: NAK

January 15, 2018

General advice

When writing R code (or any code), there are some important rules

1. Write script files (which you save) and source them. Don't do everything in the console. You probably know this already.
2. Don't write anything more than once. This has three corollaries:
   i. If you are tempted to copy/paste, don't.
   ii. Don't use *magic numbers*. Define all constants at the top of the script.
   iii. Write functions.
3. The third is **very important**. Functions are easy to test. You give different inputs and chec kwhether the output is as expected. This helps catch mistakes.
4. There are two kinds of errors: syntax and incorrect output.
   i. Syntax Errors: R can find (missing close parenthesis, wrong arguments, etc.
   ii. Incorrect Output: you can only catch by thorough testing.
5. Use meaningful names. Don't do this:

```
data("ChickWeight") # Default dataset in R

# Experience tells us what this DOES, but citing 'out' later in the code
# is going to be very confusing. I had to learn this issue the hard way.
out <- lm(weight~Time+Chick+Diet, data=ChickWeight)
```

7. Comment things that aren't clear from the (meaningful) names
8. Comment long formulas that don't immediately make sense:

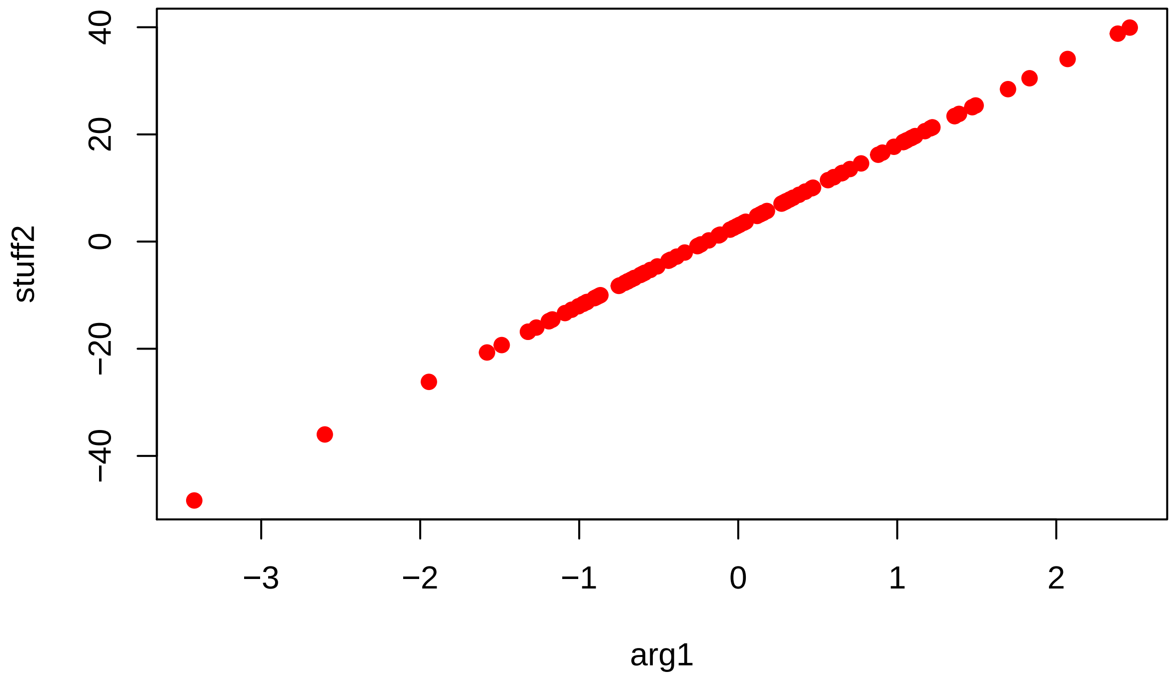
```
garbage <- with(ChickWeight,
                by(weight, Chick,
                   function(x) (x^2+23)/length(x))) ## Why is this being done?
```

Functions

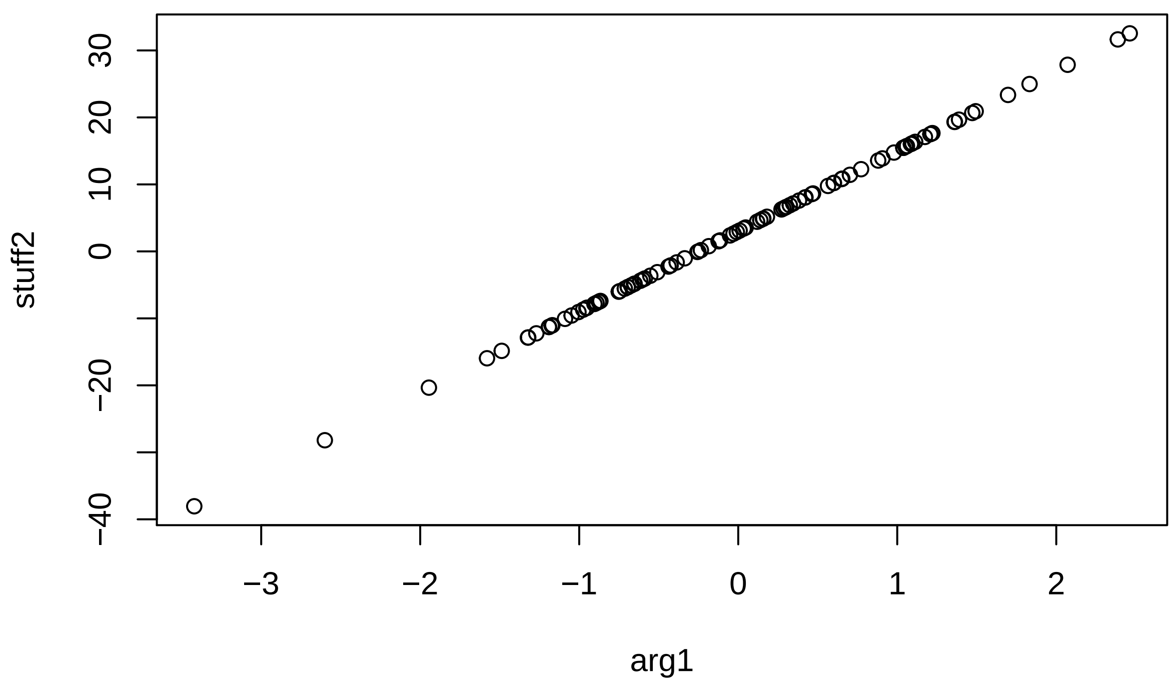
Write lots of functions. Any task you are doing more than once or twice should have a function. Otherwise you have to copy and paste code which makes it unecessarily long; the longer your code is, the harder it is to understand.

```
# Functions have arguments... Hopefully that isn't a surprise.
f <- function(arg1, arg2, arg3=12, ...){
  stuff <- arg1*arg3
  stuff2 <- stuff + arg2
  plot(arg1, stuff2, ...)
  return(stuff2)
}
```

```
x <- rnorm(100)
y1 <- f(x, 3, 15, col=2, pch=19)
```




```
f(x, 3)
```




```
##   [1]  -4.21045222   4.86445854  -8.40745112  15.46252619  -7.85283487
##   [6] -11.27267224  17.08710448  24.98586346   9.77248123  23.35879312
##  [11]  -3.63625268  10.85177152   4.42459382   6.26615516  -2.26237923
##  [16]  19.35017517   3.40629958   2.37915206  10.79578221   6.56113665
##  [21] -12.23682284  20.91854076 -10.07344738  -8.69789977   1.53913936
##  [26]  31.64631192  15.46351706   7.58388646  20.66319378  15.73330221
```

```
## [31]  -7.40068558   6.41472903  -5.08150895  10.22457188   0.18183339
## [36]  14.75217287 -14.84889685  13.56232420  -4.86235286  -5.33802148
## [41]  -7.69494253  -7.51245959   1.63627138  19.66156287  -4.82045676
## [46] -20.34346181  17.62919263  10.20570563   1.51944980  -5.55755397
## [51]  13.88306095  16.13869610   2.89269852  16.35156147  -2.10145521
## [56]  -5.93681608   7.14767217  -9.57489056  11.42833078  -9.05794373
## [61] -15.95785548 -11.08326298   8.64580132  10.83623782  -8.46167363
## [66]  -3.10733043   6.89103772  -0.01272829   0.77669323   3.56189532
## [71]   8.07433914 -38.05129543  15.62276274 -28.19723927  -1.03634613
## [76]   8.03731995  -4.39458438  19.32356271 -11.02447509  16.02943348
## [81]  -5.32896052 -12.87327443   4.89453332  17.65591505   6.64306682
## [86] -11.29372263  27.85833092  -4.04891131   8.55685528   3.10449093
## [91]  -6.01709818   5.16614328   4.66529900  17.49889420  32.54805862
## [96]  12.27109199  -1.64367112 -12.84867038   2.64475823  -0.08273064
```

Assignment Operator

What's up with '`<-`' and '`=`'? The answer is ridiculously vague.

- These two work mostly the same but not always.
- The code `<-` means to "assign" the stuff on the right to the name on the left:

```
x <- 12
x; rm(x)
```

```
## [1] 12
```

This gives `x` the value `12`.

- Note that `rm(x)` removes `x` from the workspace. This serves two purposes:
  i. Keeps the workspace tidy. A clean workspace is a happy workspace.
  ii. For commonly used variable names, like `x`, this prevents compatibility issues when `x` might be used again for a completely different purpose.
- Technically using '`x <- 12`' is the same as

```
assign('x',12)
x; rm(x)
```

```
## [1] 12
```

Versatility

- In that simple case `=` does the same thing. However, `<-` is more versatile. Consider:

```
median(x=1:10)
```

```
## [1] 5.5
```

```
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
median(x <- 1:10)
```

```
## [1] 5.5
```

```
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

General practice

- Many style guides say to **always** use `<-`. However, these are *guides*.
- My personal preference is to use `<-` most of the time, and `=` when specifying constants, specically constants at the top of the script (so I don't have magic numbers).
- If you use `<-`, you should put a space on both sides. This avoids issues like

```
x< -3
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

when you meant

```
x <- 3
```

- One reason to avoid `=` is due to confusion with logical operators like "Does x=1?"

```
x=1
x==1
```

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

Function Demonstration

Say we wanted to use quantiles from the Pareto distribution.

$$f(x) = \frac{\alpha\sigma^\alpha}{x^{\alpha+1}}; \quad F(x) = 1 - (\sigma/x)^\alpha; \quad F^{-1}(p) = \sigma\,(1-p)^{-1/\alpha}$$

This function is not in any base R package. You can search for a package that has the desireds functions… Or make your own because some tiny little thing about the function you found makes you irritated.

```
qpareto.1 <- function(p, shape, scale) scale*((1-p)^(-1/(shape)))
qpareto.3 <- function(p, shape, scale, lower.tail=TRUE) {
  if(lower.tail==FALSE) p <- 1-p
  q <- qpareto.1(p, shape, scale)
  return(q)
}
```

```
qpareto.1(.4,2,2)
```

```
## [1] 2.581989
```

```
qpareto.3(.4,2,2)
```

```
## [1] 2.581989
```

```
qpareto.3(.4,2,2,FALSE)
```

```
## [1] 3.162278
```

```
qpareto.3(.6,2,2)
```

```
## [1] 3.162278
```

Traceback, Finding Issues…

From https://www.rdocumentation.org/packages/base/versions/3.5.2/topics/traceback

By default `traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message. It can also be used to print the current stack or arbitrary lists of deparsed calls.

```
qpareto.4 <- function(p, shape, scale, lower.tail=TRUE) {
  stopifnot(p >= 0, p <= 1, shape > 0, scale > 0)
  q <- qpareto.3(p, shape, scale, lower.tail)
  return(q)}
rpareto <- function(n, shape, scale) {
  x <- vector(length=n)
  for (i in 1:n) x[i] <- qpareto.4(p=rnorm(1), shape=shape, scale=scale)
  return(x)}

rpareto(10)
```

```
## Error in shape > 0: argument "shape" is missing, with no default
```

- We have to go to RStudio and run this code to see how traceback works.

Vectorizing

- Many functions are **vectorized**, but not all.
- Arithmetic functions **are**, but you have to understand what happens when you use non-conformable (different dimension) vectors.

```
1+1
```

```
## [1] 2
```

```
c(1,2,3) + c(4,5,6)
```

```
## [1] 5 7 9
```

```
c(1,2,3) + 1
```

```
## [1] 2 3 4
```

- Some strange ones

```
min(5:1,pi)
```

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

```
pmin(5:1,pi)
```

```
## [1] 3.141593 3.141593 3.000000 2.000000 1.000000
```

Vectorizing A Function Using Loops VS Using Vectorized Functions

```
rpareto <- function(n,shape,scale) {
  x <- vector(length=n)
  for (i in 1:n) x[i] <- qpareto.4(p=runif(1),shape=shape,scale=scale)
  return(x)}
rpareto2 <- function(n,shape,scale) {
  x=qpareto.4(p=runif(n),shape=shape,scale=scale)
  return(x)}
```

```r
# A of loops versus vectors
n = 1e6
rp1time <- system.time(rpareto(n,2,1))[[3]]
rp1time
```

```
## [1] 45.304
```

```r
rp2time <- system.time(rpareto2(n,2,1))[[3]]
rp2time
```

```
## [1] 0.066
```

```r
rp1time/rp2time
```

```
## [1] 686.4242
```

```r
# Does this ratio scale like we might think it would?
rp1time <- system.time(rpareto(n/10,2,1))[[3]]
rp1time
```

```
## [1] 4.507
```

```r
rp2time <- system.time(rpareto2(n/10,2,1))[[3]]
rp2time
```

```
## [1] 0.008
```

```r
rp1time/rp2time
```

```
## [1] 563.375
```

___Loops are bad… m'kay.

Loops in R can be very bad (SLOOOOOOOW)… But Why?

- The short answer is that `R` is not a **compiled** language.
- This means that whenever you write a loop, `R` has to re-read all the code within the loop each iteration
- This is may slow.
- The only thing slower, is if you don't preallocate.
- Remember that line `x <- vector(length(n))`?
- Without that line, `x` would get built within the loop, starting with length 1, then length 2, etc.
- Preallocation is the most improtant issue to address when writing loops.

`apply` Function And Its Variants (How to avoid loops!)

- `apply` and its variants try to do things where simple loops would suffice.
- `apply` is for matrices (or arrays). If you want to **apply** a function along a dimension

```r
(mat <- matrix(1:100,10))
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
```

```
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

```r
sum(mat)
```

```
## [1] 5050
```

```r
apply(mat,2,sum) # "applies" the function "sum" to each column (2nd dimension)
```

```
##  [1]  55 155 255 355 455 555 655 755 855 955
```

```r
for(i in seq_len(ncol(mat))) sum(mat[,i]) # same in a loop
```

lapply and sapply

- These work for lists

```r
(z <- list(a=1:5, b=matrix(rnorm(10),2), c=25))
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
##            [,1]        [,2]      [,3]       [,4]       [,5]
## [1,] -1.9150744 -0.08674015 -1.117439 -0.3002784 -0.5403839
## [2,]  0.1002514  0.21251901 -1.723551 -1.9794844 -0.9464680
##
## $c
## [1] 25
```

```r
lapply(z, sum)
```

```
## $a
## [1] 15
##
## $b
## [1] -8.296649
##
## $c
## [1] 25
```

```r
sapply(z, sum)
```

```
##         a         b         c
## 15.000000 -8.296649 25.000000
```

Linear models (Finally~!)

- R has lots of functions for working with different sorts of predictive models.
- We should review how they work with `lm`, and how they generalize to other sorts of models.
- We'll use the **Mobility** data from the book website:

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

Estimation Functions and Formulas

- To estimate a linear model in `R`: you use `lm`.

```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.

The data argument

- 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)
```

- 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

```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)
```

Why formulas?

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

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

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

```
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:

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

```
##     (Intercept) log(Population)      Seg_racial         Commute
##    8.338558e-02   -2.894236e-03   -5.656590e-02    1.450771e-01
##          Income            Gini
##    1.772105e-06   -1.621921e-01
```

```
mob.lm3$coefficients
```

```
##     (Intercept) log(Population)      Seg_racial         Commute
##    8.338558e-02   -2.894236e-03   -5.656590e-02    1.450771e-01
##          Income            Gini
##    1.772105e-06   -1.621921e-01
```

Or even

```
summary(mob.lm3)$coef
```

```
##                      Estimate   Std. Error   t value     Pr(>|t|)
## (Intercept)      8.338558e-02 2.870373e-02  2.905044 3.784114e-03
## log(Population) -2.894236e-03 1.874746e-03 -1.543802 1.230739e-01
## Seg_racial      -5.656590e-02 1.713493e-02 -3.301203 1.009994e-03
## Commute          1.450771e-01 1.934259e-02  7.500397 1.869467e-13
## Income           1.772105e-06 2.878660e-07  6.156006 1.236337e-09
## Gini            -1.621921e-01 2.225561e-02 -7.287695 8.277813e-13
```

Confidence Intervals

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

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

```
##                            5 %          95 %
## (Intercept)      0.036111577  1.306596e-01
## log(Population) -0.005981875  1.934023e-04
## Seg_racial      -0.084786513 -2.834528e-02
## Commute          0.113220542  1.769336e-01
## Income           0.000001298  2.246209e-06
## Gini            -0.198846318 -1.255379e-01
```

- **WARNGING**: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 Shalizi's book, and Chapter 6 for alternatives.

Fitted values and residuals

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

```
head(fitted(mob.lm2)) # To let you know head() and tail() exist, jic
```

```
##          1          2          3          4          5          6
## 0.07048490 0.06299687 0.06926223 0.04927934 0.05791660 0.06455628
```

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

```
##          736          737          738          739          740
## -0.045252255 -0.031707484  0.004026805  0.015472295 -0.025058476
##          741
##  0.007091485
```

Using bits of the lm output

- 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.

```
# Example of all the different parts of a lm() object
names(mob.lm2)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "na.action"     "xlevels"       "call"          "terms"
## [13] "model"
```

Methods and Classes (Extra details, but possibly 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/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`.

Why methods?

- 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:

```
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.

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

```
##         89         90         91        136        140        147
## 0.06302814 0.05804528 0.06325527 0.07346574 0.04584468 0.06507174
##        151        152        153        154        156        157
## 0.06884769 0.01799403 0.03773926 0.05232423 0.03188207 0.06476723
##        158        159
## 0.03254932 0.06408194
```

Subtleties of Predict!

- 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 $\widehat{\beta}$.

- Notice that I used `mob.lm2` here, rather than the mathematically-equivalent `mob.lm1`. `-mob.lm1 <- lm(mob$Mobility ~ mob$Population + mob$Seg_racial + mob$Commute + mob$Income + mob$Gini)` `-mob.lm2 <- lm(Mobility ~ Population + Seg_racial + Commute + Income + Gini, data=mob)`
- 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.

Transformations

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

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

```
##         89         90         91        136        140        147
## 0.06907028 0.06256967 0.06773328 0.07560851 0.05136922 0.06848649
##        151        152        153        154        156        157
## 0.07059916 0.02782420 0.04427768 0.05771762 0.03861002 0.06773935
##        158        159
## 0.04120510 0.06764966
```

- 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

Fun with predict

- It just has to have columns whose names match those in the right-hand side of the formula.

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

```
##         1
## 0.1033759
```

```
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%
## 0.1122663 0.1075794 0.1024651
```

Problems w/ predict

- 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, the default is 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`:

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

```
##          1          2          3          4          5          6
## 0.06707724 0.06499898 0.06773945 0.05266410 0.06632751 0.07133333
```

More problems

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

```
##          1          2          3          4          5          6
## 0.06707724 0.06499898 0.06773945 0.05266410 0.06632751 0.07133333
```

```
# Don't do this!
```

- 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 estimate a kernel regression (as in chapter 4) to the same data, using the same variables.

```
#
library(np)
```

```
## Error in library(np): there is no package called 'np'
```

```
mob.npbw <- npregbw(formula=formula(mob.lm2), data=mob, tol=1e-2, ftol=1e-2)
```

```
## Error in npregbw(formula = formula(mob.lm2), data = mob, tol = 0.01, ftol = 0.01): could not find fu
```

```
mob.np <- npreg(mob.npbw, data=mob)
```

```
## Error in npreg(mob.npbw, data = mob): could not find function "npreg"
```

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

Why this is easy

- 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.

Putting It All Together

- We can look at a summary:

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

```
## Error in summary(mob.np): object 'mob.np' not found
```

- We can look at fitted values and residuals:

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

```
## Error in fitted(mob.np): object 'mob.np' not found
```

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

```
## Error in residuals(mob.np): object 'mob.np' not found
```

*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)))
```

```
## Error in predict(mob.np, newdata = data.frame(Population = 1500000, Seg_racial = 0, : object 'mob.np
```

- and we can plot things

```r
par(mar=c(5,5,1,1),cex.lab=3,cex.axis=2,lwd=2,col=4,bty='n')
plot(mob.np,plot.errors.method='bootstrap')
```

```
## Error in plot(mob.np, plot.errors.method = "bootstrap"): object 'mob.np' not found
```