Programming

DJM

January 16, 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.
2. Don't write anything more than once. This has three corollaries:
   1. If you are tempted to copy/paste, don't.
   2. Don't use *magic numbers*. Define all constants at the top of the script.
   3. Write functions.
3. The third is **very important**. Functions are easy to test. You give different inputs and check whether the output is as expected. This helps catch mistakes.
4. There are two kinds of errors: syntax and function.
   The first R can find (missing close parenthesis, wrong arguments, etc.
   The second you can only catch by thorough testing (see the HW)
5. Don't use **magic numbers**.
6. Use meaningful names. Don't do this:

```
data("ChickWeight")
out = lm(weight~Time+Chick+Diet, data=ChickWeight)
```

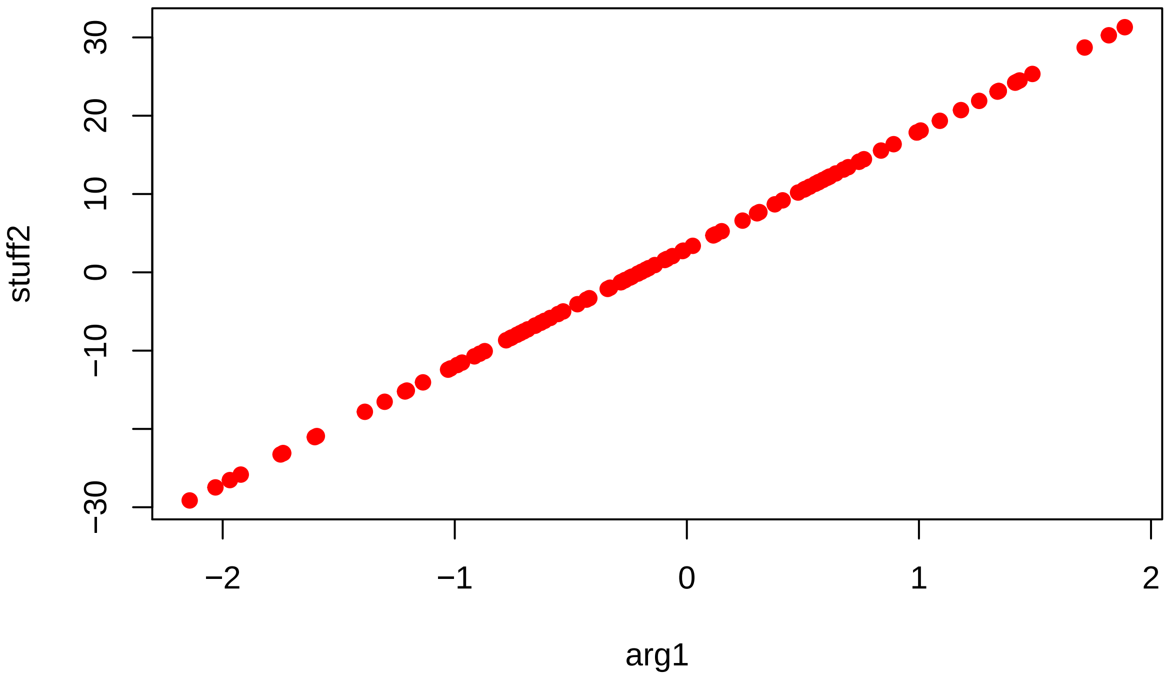
7. Comment things that aren't clear from the (meaningful) names
8. Comment long formulas that don't immediately make sense:

```
garbage = with(ChickWeight,
               by(weight, Chick,
                  function(x) (x^2+23)/length(x))) ## WTF???
```

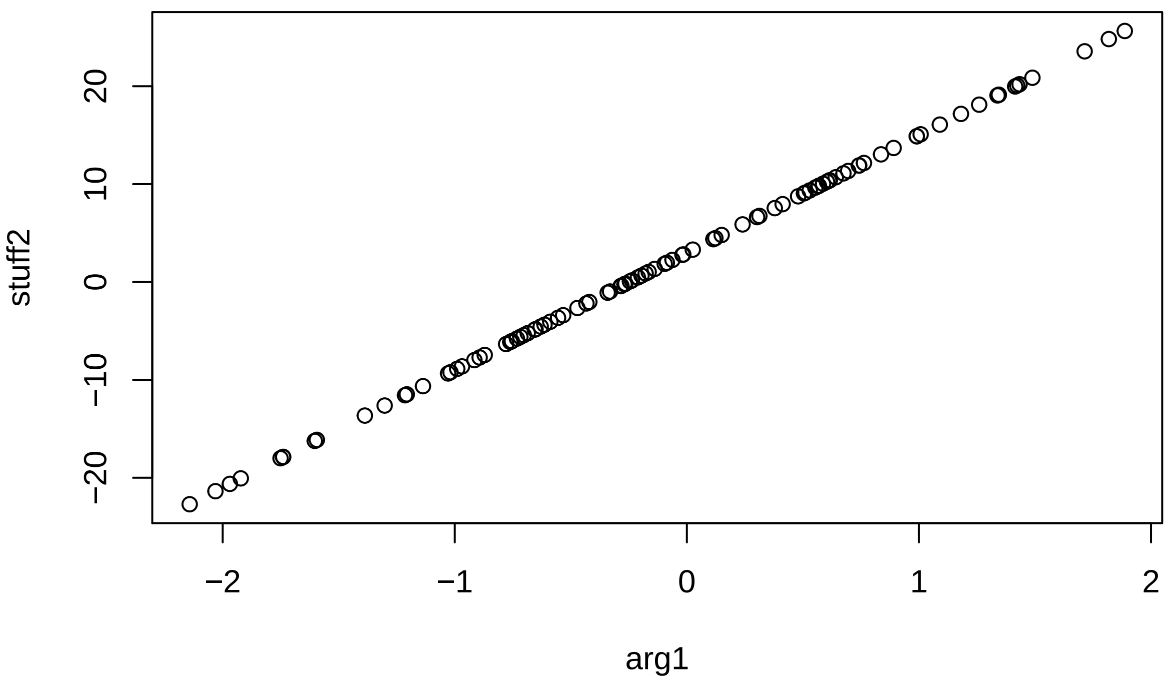
Functions

Write lots of functions. I can't emphasize this enough.

```
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)
```



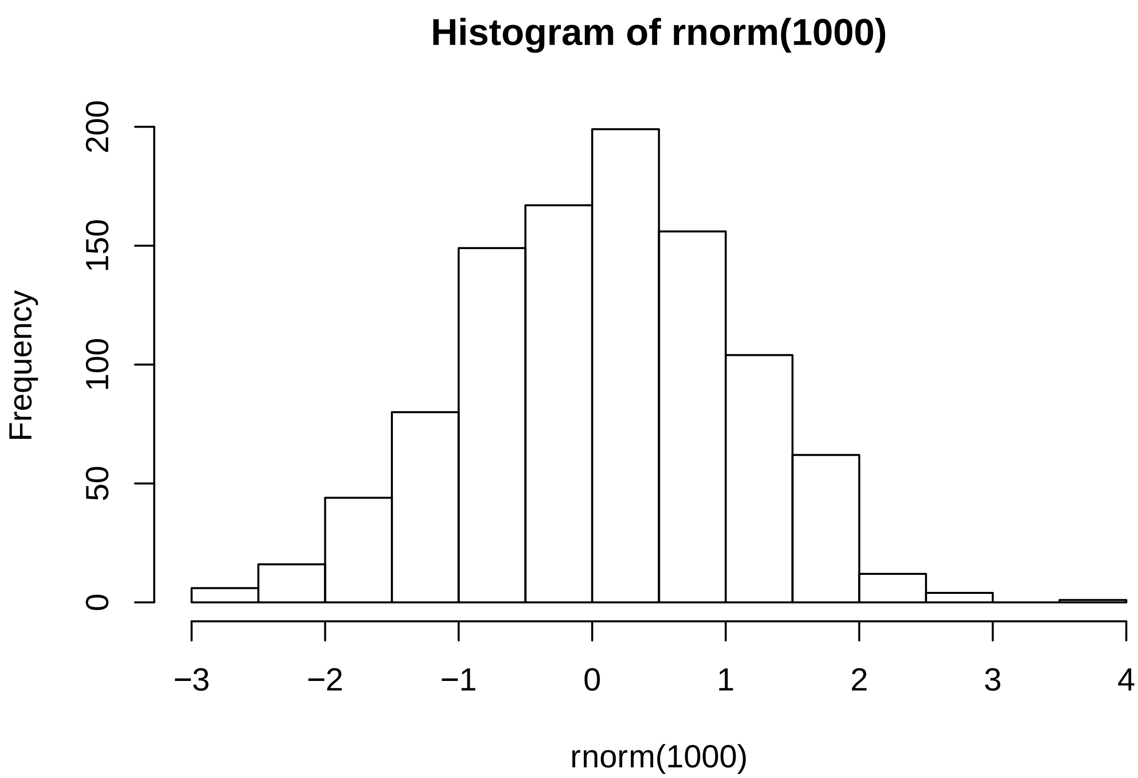

```
##  [1] -18.00788298   0.86889109  19.06211376   6.63049777 -20.62894734
##  [6]  17.17300427   1.02570542  -5.60971072   5.88181509   6.75248362
## [11]  -4.86292087  10.39075262  -4.82326015  18.11583439  -7.71489819
## [16] -13.64942445  11.91191814   1.34090350 -16.23748174   0.65292488
## [21]  24.81983502  -4.55622648 -16.13082235   9.13253251  -4.36195439
## [26]   3.30670097  -9.22311188  16.08133645  19.97581932   8.75128022
## [31]  -3.39521167   9.35203342  -0.17804018   7.54624990  -6.34442193
## [36]   9.82903423 -21.37446720  -2.65728762  10.24529674 -11.57308107
## [41]   4.47633443  12.15684602  -6.04920675 -22.70744406   9.64216753
```

```
## [46]   9.05627829 -10.64324667  -0.22725826   0.15864925  19.12411100
## [51]   1.85515391  20.86768996   2.25654979 -12.62483643  19.13081143
## [56]   0.05755745  -0.38468721  -7.98561576  10.70009178  15.08832512
## [61]   4.37113595   4.79959549   0.46715491  -5.22850149   7.95287759
## [66]  -8.62439168  10.04107748   2.81554601  14.89140059  -5.43256560
## [71]  -2.19182939  -8.87463726  14.88936583  13.04015475   1.96681072
## [76]  13.69306859 -17.87183475  -7.45198008  -5.78507681  11.34125400
## [81]   2.77216732  -0.97114266  -6.12853665  -3.66068033  20.20333776
## [86]   9.79154045  -4.06043218  -0.42093316 -17.88096061  11.09871701
## [91]  25.64430390  -2.04302230  -1.09536807  -9.34522277 -11.47958659
## [96]  10.35676582  11.89429227  20.08158215  23.56870074 -20.06565103
```

Outputs vs. Side effects

- Side effects are things a function does, outputs can be assigned to variables
- A good example is the `hist` function
- You have probably only seen the side effect which is to plot the histogram

```
myHistogram = hist(rnorm(1000))
```



The output

```
myHistogram
```

```
## $breaks
##  [1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5
## [15]  4.0
##
```

```
## $counts
##  [1]   6  16  44  80 149 167 199 156 104  62  12   4   0   1
##
## $density
##  [1] 0.012 0.032 0.088 0.160 0.298 0.334 0.398 0.312 0.208 0.124 0.024
## [12] 0.008 0.000 0.002
##
## $mids
##  [1] -2.75 -2.25 -1.75 -1.25 -0.75 -0.25  0.25  0.75  1.25  1.75  2.25
## [12]  2.75  3.25  3.75
##
## $xname
## [1] "rnorm(1000)"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

Assignment

What's up with `<-` and `=`?

- 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`.

- Technically, this 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 `<-`.
- My personal preference is to use `=` most of the time, and `<-` when naming functions, or when trying to do something like on the previous slide.
- 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
```

Flow control

```
x = 1
y = c(2,3,4,1,-1,0)
# bad  if(x=1) print(x)
if(x==1) print(x)
```

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

```
y > x
```

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

```
any(y>x)
```

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

```
! x
```

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

```
all(y>x)
```

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

```
while(x < 4){
  print(x)
  x = x+1
}
```

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

```
for(i in 1:4) print(x+i)
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
```

```r
ifelse(any(y>x), 'yes', 'no')
```

```
## [1] "no"
```

qpareto.3 and qpareto.1

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

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

```
## [1] 3.333333
```

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

```
## [1] 3.333333
```

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

```
## [1] 5
```

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

```
## [1] 5
```

Traceback

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

```r
rpareto(10)
```

```
## Error in qpareto.4(p = rnorm(1), exponent = exponent, threshold = threshold): argument "exponent" is
```

- Demonstrate in Rstudio

Vectorizing

```r
rpareto <- function(n,exponent,threshold) {
  x <- vector(length=n)
  for (i in 1:n) x[i] <- qpareto.4(p=runif(1),exponent=exponent,threshold=threshold)
  return(x)}
rpareto2 <- function(n,exponent,threshold) {
  x=qpareto.4(p=runif(n),exponent=exponent,threshold=threshold)
  return(x)}
system.time(rpareto(1e6,2,1))
```

```
##    user  system elapsed 
##  15.812   0.165  16.134
```

```r
system.time(rpareto2(1e6,2,1))
```

```
##    user  system elapsed 
##   0.084   0.006   0.090
```

When might loops bad?

- 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.

lapply vs. apply vs. sapply

- Many functions are **vectorized**, but not all.
- Arithmetic functions **are**

```r
1+1
```

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

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

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

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

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

- Some strange ones

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

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

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

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

The apply variants

- These 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

```
(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
```

```
sum(mat)
```

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

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

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

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

lapply and sapply

- These work for lists

```
(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,] -0.290342 -1.2289455 1.04796532 -0.6959165 -0.95834745
## [2,]  1.507212 -0.0569639 0.09554912 -0.1396062  0.07445968
##
## $c
## [1] 25
```

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

```
## $a
## [1] 15
##
## $b
## [1] -0.6449349
##
## $c
## [1] 25
```

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

```
##          a          b          c
## 15.0000000 -0.6449349 25.0000000
```

lapply craziness

What does this do?

```
sapply(lapply(1:10, rnorm),mean)
```

```
##  [1] -0.33787355 -0.03705105 -0.08912076 -0.67261769 -0.37438853
##  [6]  0.39594545 -0.36554899 -0.13932612 -0.64264822  0.67081178
```

Linear models

Predict and Friends

- 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:

```
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`.

```
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:

```
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

```
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:

```
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

```r
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`:

```r
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
```

Warning!!

- 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.

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:

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

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

```r
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.

```
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 (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`.

Wherefore 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.

```r
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
```

Remember

- 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`.
- 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:

```r
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.

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

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

```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%
## 0.1122663 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`:

```r
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

```r
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
```

```r
# 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.

```r
#
library(np)
```

```
## Nonparametric Kernel Methods for Mixed Datatypes (version 0.60-5)
## [vignette("np_faq",package="np") provides answers to frequently asked questions]
## [vignette("np",package="np") an overview]
## [vignette("entropy_np",package="np") an overview of entropy-based methods]
```

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

(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.

We can look at a summary:

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

```
##
## Regression Data: 729 training points, in 5 variable(s)
##
## No. Complete Observations:  729
## No. Incomplete (NA) Observations:  12
## Observations omitted or excluded:  374 376 386 410 440 459 485 542 613 616 637 652
##                Population Seg_racial    Commute   Income      Gini
## Bandwidth(s):     1649603  0.1624437 0.03871639 2382.342 0.0318117
##
## Kernel Regression Estimator: Local-Constant
## Bandwidth Type: Fixed
## Residual standard error: 0.0302321
## R-squared: 0.6733646
##
## 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] 0.06430449 0.06742469 0.07513909 0.05630422 0.06187851 0.06751230
```

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

```
##           736           737           738           739           740
## -4.472859e-02 -3.445805e-02 -6.568906e-08  2.774485e-02 -7.634712e-03
##           741
##  1.801038e-02
```

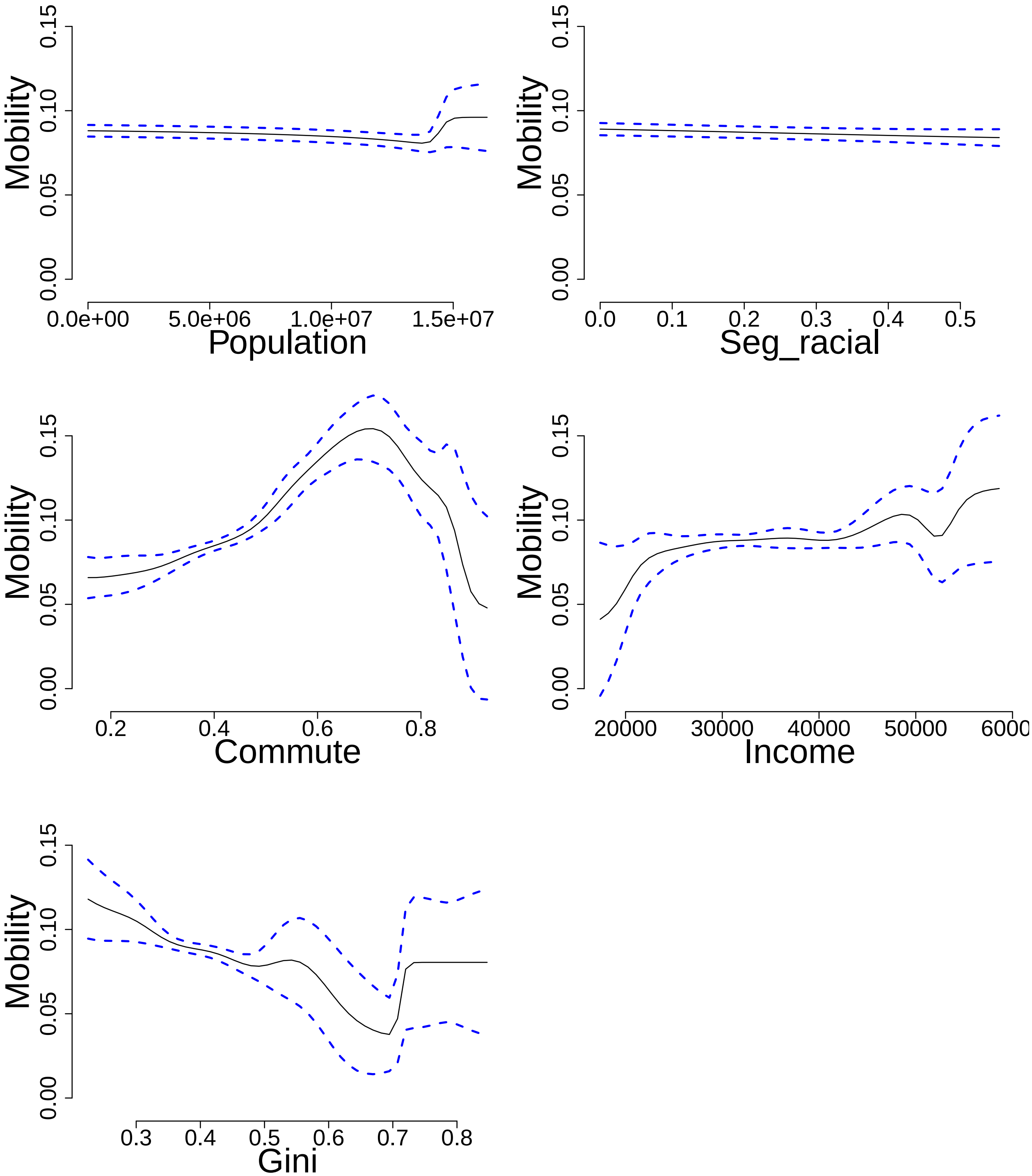
We can make predictions:

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

```
## [1] 0.09849096
```

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')
```


Mobility
0.00
0.05
0.10
0.15
Population
0.0e+00
5.0e+06
1.0e+07
1.5e+07
Seg_racial
0.0
0.1
0.2
0.3
0.4
0.5
Commute
0.2
0.4
0.6
0.8
Income
20000
30000
40000
50000
6000
Gini
0.3
0.4
0.5
0.6
0.7
0.8