

Midterm Examination

36-350, Fall 2011

SOLUTIONS

These solutions are more detailed, especially for questions 1–8, than your answers needed to be.

1. SOLUTION: (c). The intent seems to be to produce result (a), but `rbind()` (like all well-designed functions) does not alter its arguments, so `all.cases` remains empty. Something like `all.cases <- rbind(all.cases, case.1)` would have worked.
2. SOLUTION: `r <- m %*% n`. The code is just explicitly writing out the definition of matrix multiplication (“add up each row times each column”).
3. SOLUTION: (b). Because `x` and `y` are names of arguments to the functions `f` and `g` (respectively), the values we give to those arguments when calling them over-ride any values they might have in the global environment. So `f(1)` is the same as `g(1^2)`, or `g(1)`, which is `1*2^(-1)`, or 0.5, and the previous assignments to `x` and `y` don’t matter.
4. SOLUTION: (d). The first argument to `curve()`, named `expr`, needs to be an expression involving the name `x` which R can evaluate by substituting in a vector for `x`. This is true in `B`, and R will draw a parabola. In `A`, however, instead of an expression, we have the creation of an anonymous function. (See lecture 11 and the different versions of `surface()`.)
5. SOLUTION: (b). Since $5 < 10 \not< 9$, the first point is out of the limits, but $20 < 32 < 50$, so the second point is, and the right answer is 0.5. The code, however, returns 0. The first comparison, `x < upper.limits`, evaluates to `FALSE TRUE`, and the second, `x > lower.limits`, to `TRUE TRUE`. But `&&` is designed to *always* return a single Boolean value, so when it is applied to Boolean vectors, it just looks at their first element. (See lecture 3.) Thus we get the logical-and of `FALSE` and `TRUE`, which is `FALSE`, as value of `in.limits`. The code would be fixed by replacing `&&` with `&`.
6. SOLUTION: (b). Since the first argument to `f` is called `x`, running `f(2)` means 2 is the value of `x` inside `f`. The default value of `y` inside `f` is `x*2`, but this expression is evaluated inside the environment of `f`, producing 4, rather than in the global environment, which would have produced 20.

7. SOLUTION: (d). The call to `sapply()` is incorrect – it is calling `mypower` with different values of the `alpha` argument, rather than `n`.
8. SOLUTION: (b). The matrix `result` needs to be set up inside the function. The counter `i` runs over the rows of `result`, so the number of rows should equal `length(x)`. Similarly, the counter `j` runs over the columns of `result`, so the number of columns has to match `length(y)`.
9. (a) SOLUTION: The bug is in line 8, which should read

```
variance.of.eststs <- apply(jackknifed.eststs,1,var)
```

Each time `estimator(omit.one.case(data,omitted))` is run in line 6, it produces a vector, which is bound on to `jackknifed.eststs` as a column. Looping over lines 5 and 6 builds an array where each row is one of the components of the output of `estimator`, and each column corresponds to a different case in `data`. The right thing to do is to take the variance of each row of `jackknifed.eststs`. Line 8, however, takes the variance of each column. Changing the second argument of `apply` from 2 to 1 fixes the code completely.

- (b) SOLUTION: (b). The estimator being applied to the data is `mean`. Recall from intro. statistics that the standard error of the mean is s/\sqrt{n} , where s is the sample standard deviation. The data is a random draw of size 400 from a normal distribution with population mean 7 and population standard deviation 5. With $n = 400$, the sample standard deviation should be close to the population standard deviation, so $s \approx 5$, and $5/\sqrt{400} = 5/20 = 1/4$.
- (c) SOLUTION: Because the data are generated randomly.
10. (a) SOLUTION: The filled-in code should read

```
mse <- function(f, x = x.test, y = y.test) {
  y.hat <- apply(x,1,f)
  return(mean((y - y.hat)^2))
}
```

- (b) SOLUTION: The filled-in code should read

```
fitmany <- function(x = x.train, y = y.train,
  tp = seq(from=0, to=100, by=0.1)) {
  predictors <- lapply(tp,superpredictor,x=x,y=y)
  return(predictors)
}
```

The first two arguments to `lapply` are a data object and a function; it returns a list produced by applying the function over and over to the components of the data object. Extra named arguments get passed along to the function. Thus this code has the desired effect of running `superpredictor()` multiple times, with fixed `x` and `y` and changing tuning parameters.

(c) SOLUTION: The filled-in code should read

```
supersimplecv <- function(x.train, y.train, x.test, y.test,
                          tuning.param = seq(from=0, to=100, by=0.1)) {

  mse <- function(f, x, y) {
    ### OMITTED ###
  }
  fitmany <- function(x, y, tp) {
    ### OMITTED
  }

  predictors <- fitmany(x=x.train,y=y.train,tp=tuning.param)
  test.mses <- sapply(predictors,mse,x=x.test,y=y.test)
  f.best <- predictors[[which.min(test.mses)]]
  return(list(tuning.param=tuning.param,mse=test.mses,f.best=f.best))
}
```

Note the double braces `[[...]]` for accessing a single element of a list by position (rather than, say, returning a list of length 1, or accessing by name.)

If you did not know about `which.min()`, you could fake it with

```
f.best <- predictors[min(which(test.mses == min(test.mse)))]
```

(The outer `min()` handles exact ties for the smallest MSE among different predictors.) You could also write a loop. The best thing, however, is to use `which.min()`.

(d) SOLUTION:

```
cv <- supersimplecv(x.train, y.train, x.test, y.test)
best.tp <- cv$tuning.param[which.min(cv$mse)]
all.x <- rbind(x.train,x.test)
all.y <- c(y.train,y.test)
cv.selected.f <- superpredictor(x=all.x,y=all.y,tuning.param=best.tp)
```

Again, `which.min()` can be replaced by a combination of `which()` and `min()`.