Midterm Examination

36-350, Fall 2011

INSTRUCTIONS: Provide all answers in your bluebook. Only work in the bluebook will be graded. Clearly indicate which problem each answer goes with.

There are 100 points in all.

Problems 1–8 are multiple choice and each worth 5 points. Problems 9 and 10 need longer answers and are worth 25 and 35 points. Leave yourself enough time to do the longer problems.

The exam will be curved.

Explain your reasoning when possible, even for the multiple choice questions; this will help us to give partial credit.

1. (5 points) What comes next?

```
> all.cases <- c()
> case.1 <- c(1610,4)
> rbind(all.cases,case.1)
> case.2 <- c(1877,2)
> rbind(all.cases,case.2)
> all.cases
```

(a)
```
     [,1] [,2]
[1,] 1610 1877
[2,]    4    2
```

(b)
```
[1] 1877    2
```

(c)
```
NULL
```

(d)
```
[1] 1610    4
```

(e)
```
     [,1] [,2]
[1,] 1610    4
[2,] 1877    2
```

2. (5 points) `m` and `n` are two square matrices of the same size. Consider the following:

```
r <- matrix(rep(0,times=ncol(m)*nrow(m)),nrow=ncol(m),ncol=ncol(m))
for (i in 1:ncol(m)) {
  for (j in 1:ncol(m)) {
    for (k in 1:ncol(n)) {
      r[i,j] <- r[i,j] + m[i,k]*n[k,j]
    }
  }
}
```

What is it equivalent to?

(a) `r <- m %*% n`

(b) `r <- colSums(m*n)`

(c) `r <- which.min(n*m)`

(d) `r <- n*m`

(e) None of the above; it contains a bug.

3. (5 points) What comes next?

```
g <- function(y) { return(y*(2^(-y))) }
f <- function(x) { return(g(x^2)) }
y <- 3
x <- 2
f(1)
```

(a) 0.375, because that is 3×2^{-3}

(b) 0.5, because that is 1×2^{-1}

(c) An error message about the value of `y` being unknown.

(d) 0.25, because that is $(2^2) \times 2^{-(2^2)}$

(e) 0.5, because that is 2×2^{-2}

4. (5 points) Are these equivalent?

A:
```
> s <- 2
> curve(function(x) { return(log((0.5/pi)*exp(-0.5*x^2/s^2))) },
        from=-1,to=1)
```

B:
```
> s <- 2
> curve(log((0.5/pi)*exp(-0.5*x^2/s^2)),
        from=-1,to=1)
```

Choose the most accurate answer.

(a) Yes, they both produce a plot of a parabola.

(b) Yes, they both produce a plot of a bell curve.

(c) No, they both result in an error message about `s`.

(d) No, **A** results in an error message about `expr`.

(e) No, **B** results in an error message about `expr`.

5. (5 points) Consider the following function, which is designed to check what fraction of a sequence of measurements are between tolerance limits:

```
prop.within.limits <- function(x,upper.limits,lower.limits) {
  stopifnot(length(x)==length(upper.limits),length(x)==length(lower.limits))
  in.limits <- (x < upper.limits) && (x > lower.limits)
  return(sum(in.limits)/length(x))
}
```

What does

```
prop.within.limits(c(10,32),c(9,50),c(5,20))
```

return?

(a) 0, and this is the right answer.

(b) 0, but the right answer is 0.5.

(c) 0.5, and this is the right answer.

(d) 0.5, but the right answer is 0.

(e) `[1] FALSE TRUE`

6. (5 points) What comes next?

```
> x <- 10
> f <- function(x, y = x * 2) { x + y }
> f(2)
```

(a) `[1] 2`

(b) `[1] 6`

(c) `[1] 22`

(d) `Error in x + y : 'y' is missing`

7. (5 points) The function `mypower()` calculates the power of some hypothesis test with a fixed level (`alpha`) for different sample sizes (`n`) in order to investigate the cost/benefit trade-off of increasing sample size. Below is an attempt to make a plot of the power for a few different values of sample size. The function `mypower()` works perfectly, however the code that comes after it does not. Why?

```
mypower <- function(alpha = .05, n) {
  ### OMITTED ###
}

mysamplesizes <- c(10, 20, 40, 100, 200, 400, 1000)
y <- sapply(mysamplesizes, mypower)
plot(x = mysamplesizes, y = y)
```

   (a) The lengths of `mysamplesizes` and `y` are not the same.

   (b) `mypower()` does not work when its argument `n` is a vector.

   (c) `y` is actually a list, not a numeric vector as intended.

   (d) The call to `sapply` is incorrect because `n` is not specified.

8. (5 points) The function `foo(x,y)` takes two vectors as arguments. What should be in the line that is marked below?

```
foo <- function(x, y) {
  ##### WHAT GOES HERE? #####
  for(i in 1:length(x)) {
    for(j in 1:length(y)) {
      result[i, j] <- g(x[i], y[j])
    }
  }
  return(result)
}
```

(a) `result <- matrix()`

(b) `result <- matrix(nrow = length(x), ncol = length(y))`

(c) `result <- matrix(nrow = length(y), ncol = length(x))`

(d) Nothing.

9. *The Jackknife in General* (25 points total) Recall that the jackknife is a way of approximating the standard error $se_{\hat{\theta}}$, or variance $se^2_{\hat{\theta}}$, of an estimate $\hat{\theta}$ of some quantity or parameter θ. It works by going over the data set, deleting the i^{th} observation, repeating the estimate to get $\hat{\theta}_{-i}$, and then scaling up the variance of the $\hat{\theta}_{-i}$ to find $se^2_{\hat{\theta}}$.

   This code is supposed to get a jackknife standard error for any estimation function which takes a data vector and returns a vector (of fixed length) of estimated quantities.

```
jackknife <- function(estimator,data,se=TRUE) {                          # 1
  n <- length(data)                                                      # 2
  jackknifed.ests <- c()                                                 # 3
  for (omitted in 1:n) {                                                 # 4
    jackknifed.ests <- cbind(jackknifed.ests,                            # 5
      estimator(omit.one.case(data,omitted)))                            # 6
  }                                                                      # 7
  variance.of.ests <- apply(jackknifed.ests,2,var)                       # 8
  jackknifed.vars <- ((n-1)^2/n)*variance.of.ests                        # 9
  return(if (se) { sqrt(jackknifed.vars)} else { jackknifed.vars } )     # 10
}                                                                        # 11
                                                                         # 12
omit.one.case <- function(data,i) {                                      # 13
  return(data[-i])                                                       # 14
}                                                                        # 15
```

   (Refer to code by line number as convenient.)

   (a) (15 points) `jackknife()` contains a bug. Find it, explain why it is a bug, and modify no more than two lines to fix it.

   (b) (5 points) Once `jackknife()` is fixed, approximately what should the result of this be?

```
jackknife(estimator=mean,data=rnorm(n=400,mean=7,sd=5))
```

   (a) 5 (b) 1/4 (c) 7/4 (d) 1/80 (e) 1/16 (f) something else?

   (c) (5 points) Why does the previous question only have an approximate answer?

10. *Tuning parameter selection by cross-validation* (35 points total) In a predictive modeling problem we are given a vector x (the predictor variable) and wish to predict the corresponding y (the response variable). These problems usually involve a training data set that is composed of examples of x and y pairs that go together. The goal is to use the training data to fit a model that will be able to predict y from x in novel cases outside the training data set. Many statistical procedures for doing this require the user to choose a tuning parameter during the fitting stage. Here is one example:

```
superpredictor <- function(x, y, tuning.param) { ... }
```

`superpredictor()` is a function that implements some statistical procedure for fitting a predictive model. It takes 3 arguments: a matrix `x` whose n rows correspond predictor variables, a vector `y` whose entries are the corresponding n response variables, and a number `tuning.param` that specifies how cautiously the procedure should fit the model. What `superpredictor()` returns is a function which makes predictions on new data.

Suppose the global environment contains the following 4 objects:

- `x.train` — numeric matrix with n rows,
- `y.train` — numeric vector of length n,
- `x.test` — numeric matrix with m rows, and
- `y.test` — numeric matrix of length m.

The following code snippet shows example usage for `superpredictor()`

```
f <- superpredictor(x = x.train, y = y.train,
                    tuning.param = 0.1)
y.hat1 <- f(x.train[1, ])
```

`f` is the fitted predictor function. It takes a vector as its only argument. Here it is being used to make a prediction for the first row of `x.train`. Calling `superpredictor()` with the same training data but changing `tuning.param` changes the fitted prediction function (as in `f` above). For example, `f2` (and `y.hat2`) below will be different from `f` (and `y.hat1`) above.

```
f2 <- superpredictor(x = x.train, y = y.train,
                     tuning.param = 0.2)
y.hat2 <- f2(x.train[1, ])
```

Your goal over the next few subproblems is to write a function to help find a good value for `tuning.param`.

(a) (10 points) We measure the quality of the model by having `f` make predictions for all of `x.test` and then computing the mean squared error (MSE) between the predictions and `y.test`. `f` will be a function returned by `superpredictor()`. You need to write code to compute predictions for each row of `x`. Fill-in the blank below.

```
mse <- function(f, x = x.test, y = y.test) {

  #### FILL-IN HERE ####

  return(mean((y - y.hat)^2))
}
```

(b) (10 points) You have seen two examples of using `superpredictor()` to fit a predictor function with two different values of the tuning parameters. Now we want to use the same training data to fit predictor functions over a range of values for the tuning parameter `tuning.param`. The function below should return a list of predictors each fit by calling `superpredictor()` with a corresponding value for `tuning.param` given by an entry of `tp`. Fill-in the blank below.

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

  #### FILL-IN HERE ####

}
```

(c) (10 points) We now have the main ingredients for writing a simple cross-validation procedure for selecting a good value of the tuning parameter. The function below, which you need to fill-in, should return a list with the following three components:

- `$tuning.param` — a vector of values for the tuning parameter
- `$mse` — a vector of the corresponding mean squared errors
- `$f.best` — the predictor function with the smallest MSE.

Assume that the functions `mse()` and `fitmany()` from the preceding parts are declared correctly within the `supersimplecv()` function below. Note that the default values for their arguments have been removed. You should only call these functions and the built-in R functions.

```
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
  }

  #### FILL-IN HERE ####

}
```

(d) (5 points) Fill-in code to take the results of `supersimplecv()` and fit a predictor function by calling `superpredictor()` with ALL of the data (`x.train`, `y.train`, `x.test`, and `y.test`) and the the best value of `tuning.param` found by `supersimplecv()`.

```
cv <- supersimplecv(x.train, y.train, x.test, y.test)

#### FILL-IN HERE ####
```