This week we'll start out by reviewing some key concepts that we sped over last week. We'll also go over an example to demonstrate why any variable that we add to a regression will increase the R^2 (and why you should therefore be suspicious of any "R^2 maximizers"!

Last section

Using `apply`, round two: Simply put, `apply()` is a function that lets you perform operations on different parts of a matrix. I'll show you a few different ways to use `apply()`, borrowing heavily from Neil Saunders' excellent blog post on the topic[1].

First we use our `matrix()` command from last week to construct a 2×10 matrix.

```
(m <- matrix(c(1:10, 11:20), nrow = 2, ncol = 10, byrow=T))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
```

Now we'll use the apply function to get the mean of each row. Note that we could also do this manually, using `mean(m[1, ])` and `mean(m[2, ])`. But that's boring.

```
apply(m, MARGIN = 1, FUN = mean)
```

```
[1]  5.5 15.5
```

I've included the parameter names just to be clear about what's happening here. Per the `apply()` documentation, setting `MARGIN = 1` tells `R` to apply the given function over rows. Setting `FUN = mean` indicates that the given function is `mean`. If we actually want the column means, then we just have to set `MARGIN = 2`:

```
apply(m, 2, mean)
```

```
[1]  6  7  8  9 10 11 12 13 14 15
```

Note that including the parameter names is not strictly necessary, so I omitted it above. If you don't pass the parameter names, however, be sure to pass your arguments in the correct order! Otherwise you may get some very confusing results.

Finally, we can `apply()` a function to every cell in the matrix individually using `MARGIN = 1:2` (or just `1:2` as our second argument, for short).

```
apply(m, 1:2, function(x) {x/2})
```

[1] Available at `http://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/`. He goes into more detail on the entire family of `apply()` functions, if you're interested.

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]  0.5    1  1.5    2  2.5    3  3.5    4  4.5     5
[2,]  5.5    6  6.5    7  7.5    8  8.5    9  9.5    10
```

This example is a bit contrived since we could have achieved the same result with the command `m / 2`, but hopefully it makes the point clear.

OLS with matrices: As we discussed last week, the use of canned routines is not permitted for most of this class; you'll have to write the econometric routines from first principles. First, create matrices of the data, since we will be working mainly with matrix operations. Let $\mathbf{y}$ be the dependent variable, price, and let $\mathbf{X}$ be a matrix of the other car characteristics, along with a column of ones prepended. The `cbind()` function binds the columns horizontally and coerces the `matrix` class.

```
y <- matrix(data$price)
X <- cbind(1, data$mpg, data$weight)
head(X)
```

```
     [,1] [,2] [,3]
[1,]    1   22 2930
[2,]    1   17 3350
[3,]    1   22 2640
[4,]    1   20 3250
[5,]    1   15 4080
[6,]    1   18 3670
```

Last week I demo'ed how to use the `rep()` command to create an $n \times 1$ vector of ones. `rep()`, short for replicate, is an incredibly useful command. However, in this setting `cbind()` only needs to be passed a single 1 — it's smart enough to do the replication itself in order to ensure that the matrix is filled.

Just to make sure that our matrices will be conformable when we regress `y` on `X`, check that the number of observations are the same in both variables.

```
dim(X)[1] == nrow(y)
```

```
[1] TRUE
```

Using the matrix operations described in the previous section, we can quickly estimate the ordinary least squared parameter vector.

```
b <- solve(t(X) %*% X) %*% t(X) %*% y
b
```

```
            [,1]
[1,] 1946.068668
[2,]  -49.512221
[3,]    1.746559
```

That's it! And although you're not allowed to use it in your problem sets, `lm()` is a nice tool for checking our results.

```
lm(y ~ X - 1)
```

```
Call:
lm(formula = y ~ X - 1)

Coefficients:
      X1        X2        X3
1946.069   -49.512     1.747
```

They match! Thank goodness. If you're interested in knowing what `lm()` does, I highly recommend reading through relevant `R` documentation[2].

Centered R^2

First, we create a random matrix, where each element is drawn from a standard uniform distribution — another context to practice the `function()` structure. The function `randomMat()` generates a long vector of length $n \cdot k$ and then reshapes it into an $n \times k$ matrix.

```
randomMat <- function(n, k) {
  v <- runif(n*k)
  matrix(v, nrow=n, ncol=k)
}
```

You might notice that I didn't include a return statement in this function. That's okay! `R` automatically returns the output from the last command entered by default. So, the function `randomMat()` behaves as we would expect:

```
randomMat(3,2)
```

```
          [,1]      [,2]
[1,] 0.3672254 0.1453050
[2,] 0.1924655 0.3152535
[3,] 0.5996757 0.4909152
```

Another useful function for this section will be to create a square demeaning matrix **A** of dimension n. The following function just wraps a few algebraic maneuvers, so that subsequent code is easier to read.

```
  demeanMat <- function(n) {
  ones <- rep(1, n)
  diag(n) - (1/n) * ones %*% t(ones)
}
```

[2]Remember, you can do this using `?lm`.

As is described in the notes, pre-multiplying a matrix $\mathbf{B}$ by $\mathbf{A}$ will result in a matrix $\mathbf{C} = \mathbf{AB}$ of deviations from the column means of $\mathbf{B}$. Check that this is true. This may seem like a roundabout way to check the equivalence of the matrices; but it provides the opportunity to practice the `apply` function.

```
A <- demeanMat(3)
B <- matrix(1:9, nrow=3)
col.means <- apply(B, 2, mean)
C <- apply(B, 1, function(x) {x - col.means})
all.equal(A %*% B, t(C))
```

```
[1] TRUE
```

Alright, we're ready to apply the functions to real data in order to calculate the centered $\mathbf{R}^2$. First, read in the data to conform to equation (2.37) on page 14 of the lecture notes, and identify the number of observations n for later use:

```
data <- read.csv("auto.csv", header=TRUE)
names(data) <- c("price", "mpg", "weight")
y <- matrix(data$price)
X2 <- cbind(data$mpg, data$weight)
n <- nrow(X2)
```

The centered $\mathbf{R}^2$ is defined according to equation (2.41) as follows:

$$\mathbf{R}^2 = \frac{\mathbf{b}_2'\mathbf{X}_2^{*\prime}\mathbf{X}_2^*\mathbf{b}_2}{\mathbf{y}^{*\prime}\mathbf{y}^*}, \tag{1}$$

where $\mathbf{y}^* = \mathbf{Ay}$, $\mathbf{X}_2^* = \mathbf{AX}_2$, and $\mathbf{b}_2 = (\mathbf{X}_2^{*\prime}\mathbf{X}_2^*)^{-1}\mathbf{X}_2^{*\prime}\mathbf{y}^*$. Noting that $\mathbf{A}$ is both symmetric and idempotent, we can rewrite Equation (1) in terms of matrices already defined, thereby simplifying the subsequent code dramatically. From my limited experience with programming, the best code is that which reflects the core idea of the procedure; more time spent with a pen and paper and not in `R` will almost always yield more readable code, and more readable code yields fewer errors and suggests quick extensions. That said, note that $\mathbf{X}_2^{*\prime}\mathbf{X}_2^* = \mathbf{X}_2'\mathbf{A}'\mathbf{AX}_2 = \mathbf{X}_2'\mathbf{AAX}_2 = \mathbf{X}_2'\mathbf{AX}_2$ and similarly that $\mathbf{y}^{*\prime}\mathbf{y}^* = \mathbf{y}'\mathbf{Ay}$ and $\mathbf{X}_2^{*\prime}\mathbf{y}^* = \mathbf{X}_2'\mathbf{Ay}$. If we write a more general function, though, we can apply it to an arbitrary dependent vector and associated cofactor matrix:

```
R.squared <- function(y, X) {
  n <- nrow(X)
  A <- demeanMat(n)
  xtax <- t(X) %*% A %*% X
  ytay <- t(y) %*% A %*% y
  b2 <- solve(xtax) %*% t(X) %*% A %*% y
  R2 <- t(b2) %*% xtax %*% b2 / ytay
  return(R2)
  }

R.squared.adj <- function(y, X) {
  n <- nrow(X)
  k <- ncol(X)
  A <- demeanMat(n)
  xtax <- t(X) %*% A %*% X
  ytay <- t(y) %*% A %*% y
  b2 <- solve(xtax) %*% t(X) %*% A %*% y
  R2 <- t(b2) %*% xtax %*% b2 / ytay
  R2.adj <- 1 - ((n-1)/(n-k))*(1-R2)
  return(R2.adj)
  }

  R.squared(y, X2)
  R.squared.adj(y, X2)
```

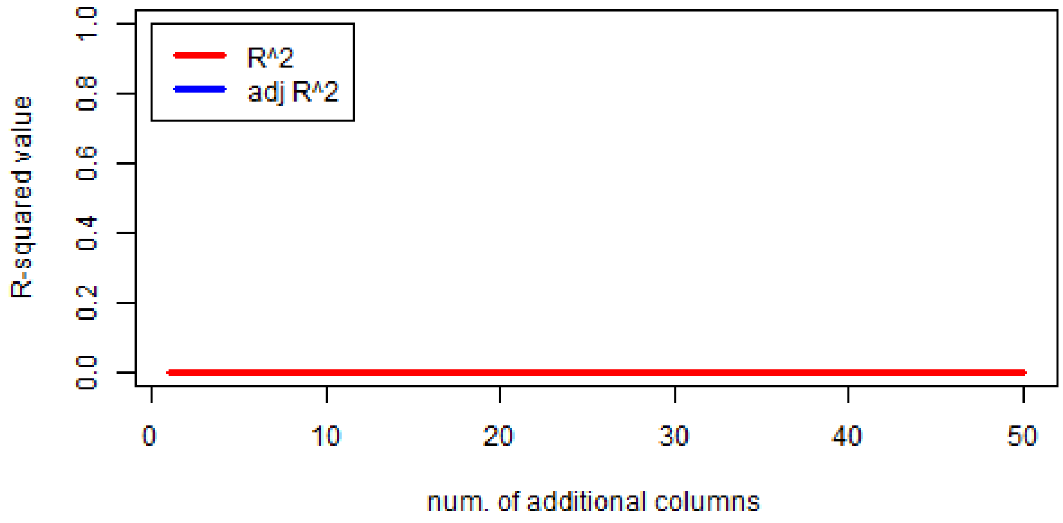
```
          [,1]
[1,] 0.2933891
          [,1]
[1,] 0.2835751
```

Without some penalty for addtional cofactors, the centered $\mathbf{R}^2$ will monotonically increase with the number of columns in the cofactor matrix $\mathbf{X}$, *even if all we do is add random covariates.* However, the $\bar{\mathbf{R}}^2$ behaves a little bit more reasonably. You can see this Figure (**??**).

```
n <- nrow(X2); k.max <- 50
X.rnd <- randomMat(n, k.max)
res.R2 <- rep(0, k.max)
res.adjR2 <- rep(0, k.max)

for (i in 1:k.max) {
  set.seed(
  X.ext <- cbind(X2, X.rnd[, seq(i)])
  res.R2[i] <- R.squared(y, X.ext)
  res.adjR2[i] <- R.squared.adj(y, X.ext)
}

plot(res.R2, type = "l", lwd = 3, col = "blue",
xlab = "num. of additional columns", ylab = "R-squared value", ylim=c(0,1))
lines(res.adjR2, type = "l", lwd = 3, col = "red")
legend(0,1,c("R^2","adj R^2"), lty = c(1,1), lwd = c(3,3), col = c("red","blue"))
```



It may be difficult to get a sense of the shape of the curve based on a single draw for the random matrix. We can calculate the relationship between $\mathbf{R}^2$ and the number of cofactors — or we can bootstrap an estimate for each index, which we will do in a subsequent section to illustrate bootstrapping in `R`.

Additional puzzles

1. Write a function `wt.coef()` that will return the OLS coefficient on weight from the regression of car price on the covariate matrix described above.
2. Adjust the function to return a list of coefficients from the same linear regression, appropriately named.
3. Find the estimate of the covariance matrix $\sigma^2(\mathbf{X}'\mathbf{X})^{-1}$ and show that the residuals and covariate matrix are orthogonal.