This week we'll start out by reviewing some key concepts that we sped over last week. Next, we'll briefly learn how to use loops in `R`. We'll then spend most of section thinking about the different kinds of $\mathbf{R}^2$ measures and how to construct them. Finally, we'll work through an example that uses many of the tools we've developed in section to demonstrate why you should be suspicious of any "$\mathbf{R}^2$ 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!

Finally, we can `apply()` a function to every cell in the matrix individually using `MARGIN = 1:2`:

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

```
     [,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
```

[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.

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 **y** be the dependent variable, price, and let **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 × 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.

```
coefficients(lm(y ~ 0 + X))
```

```
         X1          X2          X3
1946.068668  -49.512221    1.746559
```

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

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

Loops in R

Loops are a familiar concept in programming. In general, a loop is a programming statement that allows repeated execution of some piece of code. Let's take a look at one of the most common loop types, a `for` loop:

```r
for (i in 1:5) {
  print(i*i)
}
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

The structure of the `for` loop always the same. First, we specify the counter variable, `i`, with `for (i in 1:5)`. This tells `R` to run the loop five times, first with `i=1`, second with `i=2`, and so on. Then, we specify the commands we want to run repeatedly. In this case, we want `R` to print i^2 for each value of `i`. And that's it!

I should mention that using loops is somewhat discouraged in `R`. `R` is a "vector-based" language, so in theory loops are much less efficient than vectorized functions like `%*%` or `apply()`. Of course, the datasets we will work with in this class are small, so any practical difference will be neglible until you get involved with real data. But if you're interested, you can try your hand at writing the equivalent vectorized function using the `apply()` family.

Calculating R^2

We've now laid enough groundwork to start getting into some detailed examples of employing `R` to learn and do applied econometrics. In other words, I can finally stop torturing you with pointless matrix algebra and start torturing you with matrix algebra *with a purpose*! The first such example has to do with R^2 values.

First, we'll go over how to calculate the R^2 values. Since it came up in office hours, we're going to write the code so that we can retrieve the uncentered $\mathbf{R}^2_{\text{uc}}$, the centered $\mathbf{R}^2$, and the adjusted $\overline{\mathbf{R}}^2$ with *or* without an intercept in our model. However, computing $\mathbf{R}^2$ values can be trickier than it seems, particularly since many statistical packages (including `R`) are not fully transparent about which $\mathbf{R}^2$ they are displaying. To ground ourselves before we start coding, we'll first go through the intuiton behind the $\mathbf{R}^2$ measures, then the math, and finally the code.

Intuition: Some intuition may be helpful here. The uncentered $\mathbf{R}^2_{\text{uc}}$ is a measure of how much of the variance in the data our model explains relative to a hyperplane[3] where $y = 0$. The centered $\mathbf{R}^2$ is a measure of how much of the variance in our model we can explain relative to a hyperplane where $y = \bar{y}$. The adjusted $\overline{\mathbf{R}}^2$ is the $\mathbf{R}^2$ with a penalty applied for adding more covariates.

[3] When you see "hyperplane," think "line." When you see "Separating Hyperplane Theorem," sprint the opposite direction.

Math: Enough of that pesky intution. Let's go to the math. First, the uncentered $\mathbf{R}^2_{\text{uc}}$:

$$\mathbf{R}^2_{\text{uc}} \equiv \frac{\hat{\mathbf{y}}'\hat{\mathbf{y}}}{\mathbf{y}'\mathbf{y}} = 1 - \frac{\mathbf{e}'\mathbf{e}}{\mathbf{y}'\mathbf{y}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - 0)^2} = 1 - \frac{\text{SSR}}{\text{SST}_0} \tag{1}$$

The above is perfectly general. The first equality comes from $\mathbf{y}'\mathbf{y} = \hat{\mathbf{y}}'\hat{\mathbf{y}} + \mathbf{e}'\mathbf{e}$, the second is matrix decomposition, and the third is just my favorite way to think about this measure. SSR is the sum of squared residuals, and SST_0 is the total sum of squares *relative to zero*. The latter is new notation to you, so be careful: it is NOT the SST that Max refers to in his notes — we'll use that soon, when we define the centered $\mathbf{R}^2$. Which we'll do now.

The centered $\mathbf{R}^2$ looks very similar to the uncentered $\mathbf{R}^2_{\text{uc}}$. The only difference is the denominator:

$$\mathbf{R}^2 \equiv 1 - \frac{\mathbf{e}'\mathbf{e}}{\mathbf{y}^{*\prime}\mathbf{y}^*} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} = 1 - \frac{\text{SSR}}{\text{SST}} \tag{2}$$

Remember that $\mathbf{y}^* = \mathbf{A}\mathbf{y}$, the demeaned version of $\mathbf{y}$. I diverge a bit from the lecture notes here. My definition is a bit more general, since it will allow us to calculate the $\mathbf{R}^2$ for models without an intercept[4]. Here, the SST is as it was in the lecture notes — it is sum of the squared differences between all of the values of y and the mean of y.

The adjusted $\overline{\mathbf{R}}^2$ is just the $\mathbf{R}^2$ with a penalty applied for additional covariates:

$$\overline{\mathbf{R}}^2 \equiv 1 - \frac{n-1}{n-k}(1 - \mathbf{R}^2) \tag{3}$$

What happens when `k=1`?

Coding: Hopefully you now feel sufficiently prepared (or bored) to start coding. Remember, there are many ways to calculate these values, this is just one of them. However, this is most general. We'll be using the `auto.csv` data, as usual. To keep things simple, we'll define a univariate model, where `y` is price, `X` contains an intercept and `mpg`, and `X2` is just `mpg`. We'll also define an `OLS()` function:

```
data <- read.csv("auto.csv", header=TRUE)
names(data) <- c("price", "mpg", "weight")
y <- matrix(data$price)
X <- cbind(1, data$mpg)
X2 <- cbind(data$mpg)
n <- nrow(data)

OLS <- function(y,X) {
  b <- solve(t(X) %*% X) %*% t(X) %*% y
}
```

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

[4] If you're interested in exploring why the definition in the lecture notes won't work for models without an intercept, see Greene 3.5.2.

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

Now, the main event! We'll define a function that runs OLS, computes all of our $\mathbf{R}^2$ values, and returns them in a matrix. There is a lot of code here to process, but if you refer to the math above you'll see that it all matches up. Note that using a semi-colon ; indicates the end of a statement, just like a line break.

```
R.squared <- function(y,X) {
  n <- nrow(X); k <- ncol(X)
  b <- OLS(y,X); yh <- X %*% b; e <- y - yh # yh is y hat, the predicted value for y

  SSR <- t(e) %*% e
  SST.0 <- t(y) %*% y # == sum(y^2)
  R2.unc <- 1 - SSR / SST.0

  A <- demeanMat(n)
  ys <- A %*% y # this is ystar
  SST.yb <- t(ys) %*% ys # ==  sum((y - mean(y)^2))
  R2.cen <- 1 - SSR / SST.yb

  R2.adj <- 1 - ((n-1)/(n-k))*(1-R2.cen)

  return(cbind(R2.unc,R2.cen,R2.adj))
}
```

We'll test our function by calling it with `X`, our data matrix that contains an intercept and `mpg`. We'll also confirm our results with `lm()`.

```
(Rsq.X2 <- R.squared(y,X))
summary(lm(y ~ X))$r.squared
summary(lm(y ~ X))$adj.r.squared
```

```
         [,1]      [,2]      [,3]
[1,] 0.856253 0.2195829 0.2087437
[1] 0.2195829
[1] 0.2087437
```

`lm()` does not, as far as I know, have a method for returning the uncentered R^2 for this model. Now let's try with `X2`, the data matrix with `mpg` but *no intercept*. If this were a horror movie, ominous music would be playing right now[5]:

```
(Rsq.X2 <- R.squared(y,X2))
summary(lm(y ~ 0 + X2))$r.squared
summary(lm(y ~ 0 + X2))$adj.r.squared
```

[5] Don't open that door!!

```
          [,1]       [,2]       [,3]
[1,] 0.6718225 -0.7817091 -0.7817091
[1] 0.6718225
[1] 0.6673269
```

Our results for $\mathbf{R}^2$ and $\overline{\mathbf{R}}^2$ are totally insane, but for some reason our $\mathbf{R}^2_{\text{uc}}$ matches the output from `lm()`. Does this make any sense? Actually, yes! Remember that the centered $\mathbf{R}^2$ compares our model, $\mathbf{y} = \mathbf{X}\beta + \varepsilon$ (with no intercept) to $\mathbf{y} = \bar{\mathbf{y}}$. Since $\mathbf{y} = \bar{\mathbf{y}}$ is actually a much better predictor of $\mathbf{y}$ than $\mathbf{Xb}$, we get that SSR > SST. Using that $\mathbf{R}^2 = 1 - \frac{\text{SST}}{\text{SST}}$, we can see why we get a negative $\mathbf{R}^2$ and $\overline{\mathbf{R}}^2$.

But, our results don't match the results from `lm()`. This is because `lm()` is actually pulling a bait-and-switch here: since we aren't passing it an intercept, it computes the uncentered $\mathbf{R}^2_{\text{uc}}$ instead of the centered $\mathbf{R}^2$ [6]. So rather than comparing our model $\mathbf{y} = \mathbf{Xb} + \varepsilon$ to $\mathbf{y} = \bar{\mathbf{y}} + \varepsilon$, it compares it to $\mathbf{y} = \mathbf{0} + \varepsilon$. If this doesn't make any sense to you, don't worry about it too much. Just remember to include an intercept in your model and everything will be fine.

More $\mathbf{R}^2$ shenanigans

Let's continue looking at the fishy behavior of $\mathbf{R}^2$. Suppose we were to add a few "variables" to our model. Suppose these "variables" were just columns of random numbers. What would that do to our $\mathbf{R}^2$? Let's find out.

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,] 0.03139583 0.1974591
[2,] 0.55584698 0.8773440
[3,] 0.21074879 0.2793260
```

Now, putting together everything we learned today[7], we'll use the code we wrote earlier and a `for` loop to calculate the $\mathbf{R}^2$ and $\overline{\mathbf{R}}^2$ as we add additional (random) covariates. To avoid negative values for our $\mathbf{R}^2$, we'll use `X`, which is our data matrix that contains an intercept and `mpg`:

[6] For more information, see here: `http://bit.ly/1c0nA6N`.

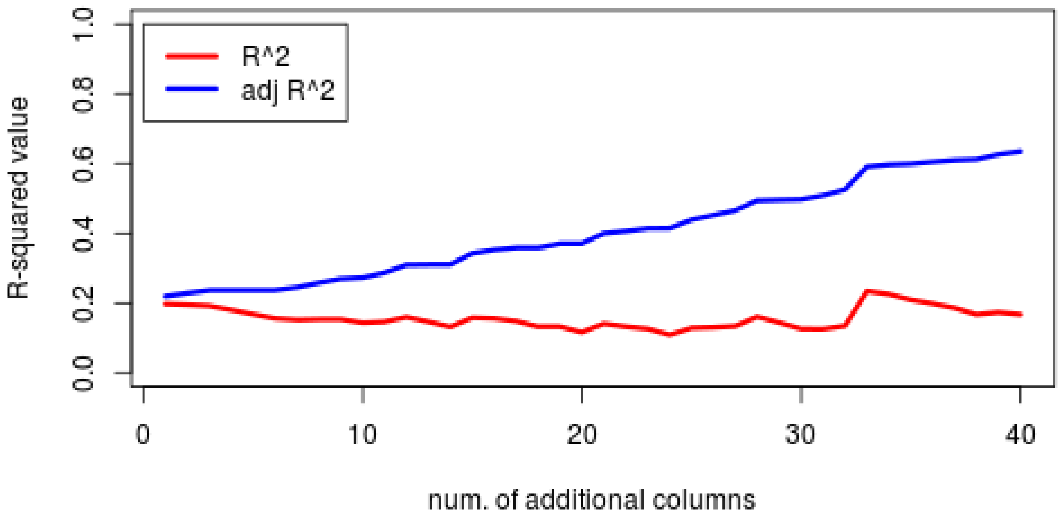
[7] See what I did there?

```
k.max <- 40
X.rnd <- randomMat(n, k.max)
R2.out <- matrix(rep(0, k.max*3), ncol = 3)

for (i in 1:k.max) {
  X.ext <- cbind(X, X.rnd[, seq(i)])
  R2.out[i, ] <-  R.squared(y, X.ext)
}
```

Next, we'll plot our $\mathbf{R}^2$ and $\overline{\mathbf{R}}^2$ values using a line plot:

```
plot(R2.out[ ,2], type = "l", lwd = 3, col = "blue",
xlab = "num. of additional columns", ylab = "R-squared value", ylim=c(0,1))
lines(R2.out[ ,3], 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"))
```



As you can see, our $\mathbf{R}^2$ has gone from a puny 0.2 to a robust 0.6, so our model must be much better, right? Wrong! This is another example of the classic "overfitting" problem: in a regression with 74 observations and 40+ covariates, *something* is always going to explain some portion of the variation. The $\overline{\mathbf{R}}^2$, on the other hand, stays mostly at the same level. But what if we set `k.max <- 70`? `k.max <- 100`? Why?

That's it for this section. Next week, we'll turn our attention to hypothesis testing.

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.