Lecture 3: SQUARESVILLE
Last time

We started with a review of R, discussing some of its history and some basic functionality; we got as far one of the basic data types, vectors

We then reviewed so-called variance inflation factors and started looking at modeling with polynomials; we didn’t get very far because we spent too much time with R
Today

We will reverse the order of the material so we don’t end up sinking too much time into R; we will start with orthogonality, its meaning and its usefulness in regression

This will lead us to a process for generating orthogonal predictors called successive orthogonalization or the Gram-Schmidt procedure

We will then find a new interpretation for a regression coefficient, relating its stability to how much information it shares with other variables

We close with orthogonal polynomials and an introduction to the bias-variance tradeoff
Orthogonality

Two vectors \( \mathbf{x}_1 = (x_{11}, \ldots, x_{1n}) \) and \( \mathbf{x}_2 = (x_{21}, \ldots, x_{2n}) \) are said to be orthogonal if they are perpendicular.

Let \( \theta \) denote the angle between \( \mathbf{x}_1 \) and \( \mathbf{x}_2 \); then from linear algebra or from basic geometry (for \( n=2 \) or \( 3 \)) we have the formula

\[
\cos \theta = \frac{\sum_{i=1}^{n} x_{i1} x_{i2}}{\sqrt{\sum_{i=1}^{n} x_{i1}^2} \sqrt{\sum_{i=1}^{n} x_{i2}^2}}
\]

Therefore, \( \mathbf{x}_1 \) and \( \mathbf{x}_2 \) are orthogonal if \( \sum_{i=1}^{n} x_{i1} x_{i2} = 0 \).
Orthogonality

We are going to apply this property in several places over the course of the quarter; today, we are going to see how an assumption of orthogonality simplifies the least squares problem.

Along the way, we will learn how R fits a linear model (specifically, we'll see something called the QR decomposition), and then see some examples where the orthogonality property holds.

Today will be a mix of computation, methodology and application...
Regression revisited

First, recall the basic normal linear model: We have n observations $y_1, \ldots, y_n$ where each response $y_i$ is associated with the predictors $x_{i1}, x_{i2}, \ldots, x_{ip}$; then, we assume that

$$y_i = \beta_1^* x_{i1} + \beta_2^* x_{i2} + \cdots + \beta_p^* x_{ip} + \epsilon_i$$

where the $\epsilon_i$ are independent, each having a normal distribution with mean 0 and variance $\sigma^2$.
Regression revisited

To estimate the coefficients $\beta_1^*, \ldots, \beta_p^*$ we had been turing to OLS (ordinary least squares, or the maximum likelihood estimate under the normal linear model)

We want to choose $\hat{\beta}_1, \ldots, \hat{\beta}_p$ to be the value of $\beta_1, \ldots, \beta_p$ that minimizes the OLS criterion

$$\sum_{i=1}^{n} [y_i - \hat{\beta}_1 x_{i1} - \cdots - \hat{\beta}_p x_{ip}]^2$$
Regression revisited

Last quarter, you saw that you could write the solution to the least squares problem in matrix form $\hat{\beta} = (X^TX)^{-1}X^Ty$ where $\hat{\beta} = (\hat{\beta}_1, \ldots, \hat{\beta}_p)$

where $y = (y_1, \ldots, y_n)$, and the $n \times p$ design matrix $X$ is given by

$$X = \begin{pmatrix} x_{11} & x_{21} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ x_{31} & x_{32} & \cdots & x_{3p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix}$$
# standardizing variables
# let's work with the cancer data

cancer = read.csv(url("http://www.stat.ucla.edu/~cocteau/cancer.csv"))

names(cancer)

# in R, the matrix X is referred to as the model matrix

X = model.matrix(lpsa~.,data=cancer)

class(X)
dim(X)

attributes(X)

# now have a look at the fish data again

fish = read.csv(url("http://www.stat.ucla.edu/~cocteau/stat101c/data/fish.csv"))

fish$station = factor(fish$station)
fish$river = factor(fish$river)

X = model.matrix(mercury~.,data=fish)

class(X)
dim(fish)
dim(X)

attributes(X)
Orthogonal predictors

In certain special cases, our predictors are orthogonal; that means, for example, that the first and second predictors satisfy

$$\sum_{i=1}^{n} x_{i1}x_{i2} = 0$$

What does this mean for our regression? In particular, what happens to $X'X$?
Orthogonal predictors

When our predictors are orthogonal, the matrix $X'X$ becomes diagonal and the diagonal entries are just sums of squares; for example, the first and second diagonal entries are

$$\sum_{i=1}^{n} x_{i1}^2 \quad \text{and} \quad \sum_{i=1}^{n} x_{i2}^2$$

What does all this mean in terms of our least squares solutions?
Orthogonal predictors

Since we have that $\hat{\beta} = (X'X)^{-1}X'y$, if our predictors are orthogonal, then the least squares estimates have a simple form.

For example, the first and second estimates are

\[
\hat{\beta}_1 = \frac{\sum_{i=1}^{n} x_{i1}y_i}{\sum_{i=1}^{n} x_{i1}^2} \quad \text{and} \quad \hat{\beta}_2 = \frac{\sum_{i=1}^{n} x_{i2}y_i}{\sum_{i=1}^{n} x_{i2}^2}
\]
Orthogonal predictors

Next, what does this mean in terms of the variance inflation factors considered last time? In particular, what do we get if we form a regression of one predictor onto the rest?

Suppose, for example, we regress the first predictor on the remaining p-1; what values do you get for the coefficients? What is the value of $R^2$?
Orthogonal predictors

For those of you who are uncomfortable with the matrix notation, we can derive these expression directly by expanding the least squares criterion (we would never ever ever ever do this in a real context)

\[
\sum_{i} \left[ y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip} \right]^2 = \\
\sum_{i} y_i^2 + \beta_1^2 \sum_{i} x_{i1}^2 + \cdots + \beta_p^2 \sum_{i} x_{ip}^2 \\
-2\beta_1 \sum_{i} y_i x_{i1} - \cdots - 2\beta_p \sum_{i} y_i x_{ip} \\
-2\beta_1 \beta_2 \sum_{i} x_{i1} x_{i2} - \cdots - \beta_{p-1} \beta_p \sum_{i} x_{i,p-1} x_{ip}
\]
Orthogonal predictors

For orthogonal predictors, the cross terms in the previous expression are all zero; here are two views of that relationship:

An expression in R

\[
> \text{sum}(X[,1]*X[,2])
\]

[1] 0

Analytical expression

\[
\sum_{i} x_{i1}x_{i2} = 0
\]

This will simplify things considerably....
Orthogonal predictors

Collecting terms, we arrive at the following expression

\[
\sum_{i} [y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}]^2
\]

\[= \sum_{i} y_i^2 + \beta_1^2 \sum_{i} x_{i1}^2 + \cdots + \beta_p^2 \sum_{i} x_{ip}^2 - 2\beta_1 \sum_{i} y_i x_{i1} - \cdots - 2\beta_p \sum_{i} y_i x_{ip}\]

which means minimizing the sum of squares is now trivial
Orthogonal predictors

Differentiating with respect to $\beta_1$ gives us the OLS estimate

$$\hat{\beta}_1 = \frac{\sum_i y_i x_{i1}}{\sum_i x_{i1}^2}$$

which is what we expected
Orthogonal predictors

From these expressions, we can also easily derive the distribution of the regression estimates under the normal linear model; recall we have assumed

\[ y_i = \beta_1^* x_{i1} + \beta_2^* x_{i2} + \cdots + \beta_p^* x_{ip} + \epsilon_i \]

Substituting in the expression for \( \hat{\beta}_1 \) we get

\[ \hat{\beta}_1 = \sum_{i=1}^{n} y_i x_{in} = \sum_{i=1}^{n} (\beta_1^* x_{i1} + \cdots + \beta_p^* x_{ip} + \epsilon_i) x_{i1} \]

and by orthogonality we find

\[ \hat{\beta}_1 = \beta_1^* + \frac{\sum_{i=1}^{n} \epsilon_i x_{i1}}{\sum_{i=1}^{n} x_{i1}^2} \]
Orthogonal predictors

This last expression tells us everything! Since the $\epsilon_i$ are assumed normal and independent, then, conditional on the values of our predictors, our least squares estimates are normally distributed.

Even without normality, if $\epsilon_i$ are independent with mean 0 and common variance $\sigma^2$, then

$$\hat{\beta}_2 = \beta_2^* + \frac{\sum_i \epsilon_i x_{i1}}{\sum_i x_{i1}^2}$$

has mean $\beta_2^*$ and variance

$$\text{var}(\hat{\beta}_1) = \frac{\sum_i x_{i1}^2 \text{var} \epsilon_i}{(\sum_i x_{i1})^2} = \sigma^2 \frac{\sum_i x_{i1}^2}{(\sum_i x_{i1})^2} = \frac{\sigma^2}{\sum_i x_{i1}^2}$$
Orthogonal predictors

To prepare us for the next lecture, suppose instead of
\[ y_i = \beta_1^* x_{i1} + \beta_2^* x_{i2} + \cdots + \beta_p^* x_{ip} + \epsilon_i \]
we have guessed wrong and our model is really \( y_i = f(x_{i1}, \ldots, x_{ip}) + \epsilon_i \)

Then, our regression coefficients become
\[
\hat{\beta}_1 = \frac{\sum_i y_i x_{i1}}{\sum_i x_{i1}^2} = \frac{\sum_i [f(x_{i1}, \ldots, x_{ip}) + \epsilon_i] x_{i1}}{\sum_i x_{i1}^2}
\]
Orthogonal predictors

Rewriting this a little, we find

\[ \hat{\beta}_1 = \frac{\sum_i f(x_{i1}, \ldots, x_{ip})x_{i1}}{\sum_i x_{i1}^2} + \frac{\sum_i \epsilon_i x_{i1}}{\sum_i x_{i1}^2} \]

This says that if we guess wrong in specifying our model, then our least squares coefficients are "centered" on the value obtained by regressing the true, unknown function onto the space of predictors.

We can obviously make a stronger statement here, covering the case that we have excluded important variables (and considering the matrix form of the solution, you can come up with the same result for non-orthogonal predictors), but we'll get to that a bit later.
Orthonormal predictors

In some cases, we go a bit farther and standardize variables so that they also have a sum of squares equal to 1; for the first predictor, this means

\[ \sum_{i=1}^{n} x_{i1}^2 = 1 \]

We call these predictors orthonormal; it simplifies our notation even further, so that our least squares estimates are just the “inner products”

\[ \hat{\beta}_1 = \sum_{i=1}^{n} y_i x_{i1} \]
Orthonormal predictors

With orthonormal predictors, our expressions get even cleaner; consider, for example, the residual sum of squares based on orthonormal predictors

\[
\sum_{i=1}^{n} [y_i - \hat{\beta}_1 x_{i1} - \cdots - \hat{\beta}_p x_{ip}]^2
\]

\[
= \sum_{i=1}^{n} y_i^2 + \hat{\beta}_1^2 + \cdots + \hat{\beta}_p^2 - 2\hat{\beta}_1 \sum_{i=1}^{n} x_{i1}y_i - \cdots - 2\hat{\beta}_p \sum_{i=1}^{n} x_{ip}y_i
\]

\[
= \sum_{i=1}^{n} y_i^2 + \hat{\beta}_1^2 + \cdots + \hat{\beta}_p^2 - 2\hat{\beta}_1 \sum_{i=1}^{n} x_{i1}y_i - \cdots - 2\hat{\beta}_p \sum_{i=1}^{n} x_{ip}y_i
\]

\[
= \sum_{i=1}^{n} y_i^2 - \sum_{j=1}^{p} \hat{\beta}_j^2
\]
Regression revisited

From this last expression we see that as we add more variables, our RSS goes down; every new orthonormal predictor drops it by the square of the associated OLS coefficient.

This means that finding the best model of size 5 in terms of RSS, say, is trivial; just find the five predictors with the largest coefficients!
Orthogonal predictors

While they seem to make life easier, improving the stability of our estimates in terms of the associated VIFs and making the overall least squares solution simpler, you’ve probably only seen such predictors in the context of a designed experiment (so called fractional factorial designs have this property).

So how general a construction is this?
Orthogonal predictors

Well, we have seen one important case of orthogonality in the context of least squares; that is, by design, the residuals from a regression are orthogonal to the predictors

Intuitively, this comes from the fact that regression is just a projection onto the space spanned by the columns of our design matrix X; the residuals are, therefore, orthogonal to that space

If you’re not comfortable with that line of reasoning (I am not sure how much of that you covered last quarter), you can also see it directly from the normal equations...
Orthogonal predictors

That is, taking partial derivatives of

$$\sum_{i=1}^{n} [y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}]^2$$

with respect to $\beta_1, \ldots, \beta_p$ leads to the $p$ equations

$$\sum_{i=1}^{n} (y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}) x_{i1} = 0$$
$$\sum_{i=1}^{n} (y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}) x_{i2} = 0$$
$$\vdots \quad \vdots \quad \vdots$$
$$\sum_{i=1}^{n} (y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}) x_{ip} = 0$$
Orthogonal predictors

This, then, gives us a clue about how we can create orthogonal predictors; in fact, if we were to solve the normal equations for two predictors -- an intercept so that $x_1 = (1, \ldots, 1)$, and a covariate $x_2 = (x_{12}, \ldots, x_{n2})$ -- we would find that

$$
\hat{\beta}_2 = \frac{\sum_i y_i (x_{1i} - \bar{x}_2)}{\sum_i (x_{1i} - \bar{x}_2)^2}
$$

We can understand this in terms of a two step process...
One, two...

First, conduct a regression where $x_2$ is the response and the only predictor is the intercept $x_1 = \{1, \ldots, 1\}$; it's not hard to show (and we've basically done it in the last few slides) that the coefficient is $\beta_2$

We then construct a new variable representing the residuals after this regression $z_2 = x_2 - \bar{x}_2 x_1$ (which is orthogonal to the constant term $x_1$) and regress the response $y$ on $z_2$ to give the coefficient $\beta_2$

Note: Here we've started using vector notation (suitable for direct use in R) so that $z_2 = x_2 - \bar{x}_2 x_1$ means $z_{2i} = x_{i2} - \bar{x}_2$ for $i = 1, \ldots, n$
Orthogonal predictors

For those of you versed in the projection approach, we haven’t changed the space spanned by $X_1$ and $X_2$; instead we’ve derived a new pair of orthogonal basis functions for the space $X_1$ and $X_2$.

This procedure generalizes to $p$ predictors...
Regression by successive orthogonalization (Gram-Schmidt)

1. Set $z_1 = x_1 = (1, \cdots, 1)$

2. For $j = 2, \ldots, p$

   Regress $x_j$ on $z_1, \ldots, z_{j-1}$ to produce the coefficients

   $$\hat{\gamma}_{lj} = \frac{\sum_l z_l x_{il}}{\sum_l z_l^2} \quad \text{for} \quad l = 1, \ldots, j - 1$$

   and the residual vector $z_j = x_j - (\hat{\gamma}_{j1} z_1 + \cdots + \hat{\gamma}_{j-1,j} z_{j-1})$

3. Regress $y$ on the residual $z_p$ to give $\hat{\beta}_p = \frac{\sum_i z_p y_i}{\sum_i z_p^2}$
Unrolling

Rearranging the residuals in step (2), we find

\[ x_1 = z_1 \]
\[ x_2 = z_2 - \hat{\gamma}_{12} z_1 \]
\[ x_3 = z_3 - \hat{\gamma}_{23} z_2 - \hat{\gamma}_{13} z_1 \]
\[ \vdots \]
\[ x_p = z_p - \hat{\gamma}_{p-1,p} z_{p-1} - \cdots - \hat{\gamma}_{1p} z_1 \]

And so each \( x_j \) is a linear combination of the \( z_j \) for \( 1 \leq j \); since the \( z_j \) are all orthogonal, they form a basis for the column space of \( X \).

Also, since \( x_p \) alone involves \( z_p \) (with coefficient 1), we see that the coefficient from step (3) is indeed the regression coefficient we’re after.
Two results

If $x_p$ is highly correlated with some of the other $x_i$'s, the residual vector will be small and the coefficient vector unstable; that is, we have

$$\text{var } \hat{\beta}_p = \frac{\sigma^2}{\sum z_{i}^{2}}$$

so that the precision with which we can estimate $\hat{\beta}_p$ depends on the sum of squares $\sum z_{i}^{2}$; this represents how much of $x_p$ is unexplained by the other $x_i$'s.
Two results

This, then, exposes two results

1. In general, the jth regression coefficient is the univariate regression of y on z where z is the residual after regressing $x_j$ on $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_p$

2. We can represent the GS algorithm in matrix form

$$X = Z\Gamma$$

where $Z$ has the columns $z_1, \ldots, z_p$ and $\Gamma$ is the upper-triangular matrix with entries $\Gamma_{ij}$; introducing a diagonal matrix $D$ where the jth element is $\sqrt{\sum_j z_j^2}$ and so we can write

$$X = (ZD^{-1})(D\Gamma) = QR$$
The QR decomposition

The QR decomposition breaks $X$ up into an $n$-by-$p$ orthogonal matrix $Q$ and a $p$-by-$p$ upper triangular matrix $R$; it represents one convenient orthogonal basis for the column space of $X$

Aside from the last regression coefficient, you can represent the full fit with the equation $\beta = R^{-1}Q^Ty$ (the inverse of $R$ being easy because $R$ is triangular)

This is how $R$ fits a linear model; rather than solve the normal equations via $X^tX^{-1}X^ty$, it instead constructs the QR decomposition of the model matrix
An example

We briefly introduced this data set last time; it concerns the amount of strontium isotopes in sea water.

Specifically, the idea was to examine the jump in strontium in sea water at the so-called Cretaceous-Tertiary boundary (65.5 million years ago) -- a moment associated with a mass extinction.

The size of the jump in strontium around the K-T boundary (K not C, from the German Kreidezeit) could tell us something about what caused the extinctions; was it a meteor? Was it volcanic activity? Both?
Sr ratio vs age (millions of years)
Modeling

As a function of time, we probably would not attempt to model strontium levels with a linear fit; instead we might try something more elaborate: Let's start with simple polynomials (we'll get more elaborate as the quarter progresses); on the next slide, we consider two ways to represent the space of cubic polynomials

While the first is certainly easiest, what do we notice about the variance inflation factors for this collection of predictors?

One alternative that is commonly used are the so-called orthogonal polynomials; at the right we use the R command `poly` to generate a basis set and then look at the VIFs
# load some new data
kt = read.csv(url("http://www.stat.ucla.edu/~cocteau/kt.csv"))

# check out the variables
names(kt)
dim(kt)

# have a look
plot(kt, xlab="age (millions of years)", ylab="Sr ratio")

# create the basis terms for
# a cubic polynomial

tmp = data.frame(a = kt$age,
                 a2 = kt$age^2,
                 a3 = kt$age^3)

# now, check their "stability"
vif(tmp)

# what do you see? do you recommend using
# this basis?
# let's try another one; the so-called
# orthogonal polynomials

tmp = poly(kt$age, deg=3)
vif(tmp)
Orthogonality

The VIFs for the orthogonal polynomials were all 1; that means the $R^2$ values associated with the various regressions were all zero.

In this case, we say that the predictors are orthogonal; to make sense of this, recall that for a regression to have an $R^2$ of zero, the predictors contain “no information” about the response.

At a technical level, these predictors are orthogonal in the sense that as vectors in an n-dimensional space (in this case n=45), they are perpendicular; put another way, their inner products are zero.

Check that for your orthogonal polynomials!
# form the orthogonal polynomials

tmp = poly(kt$age, deg=3)

# now have a look

plot(kt$age, tmp[,1])
plot(kt$age, tmp[,2])
plot(kt$age, tmp[,3])

matplot(kt$age, tmp)
Orthogonal polynomials

How do these compare to ordinary polynomials?

In particular, what seems to be happening as the degree gets larger?
Orthogonal polynomials

We can apply (essentially) the Gram-Schmit procedure

1. First, let $z_0$ represent the constant function $(1,...,1)$

2. Next, let $z_1 = x - \bar{x}$; put another way, we can think of $z_1$ as the residuals after fitting a with $x$ as the response and $z_0$ as the predictor -- an intercept-only model

3. Next, let $z_2$ be the residuals after a regression of $x^2$ using $z_0$ and $z_1$ as predictors

4. Finally, let $z_3$ be the residuals after a regression of $x^3$ using $z_0$, $z_1$ and $z_2$ as predictors
# constructing an orthogonal polynomial
# the hard way
# first, the linear term

x1 = kt$age
x1 = x1 - mean(x1)
x1 = x1/sqrt(sum(x1^2))

# then the quadratic

x2 = kt$age^2
x2 = residuals(lm(x2~x1))
x2 = x2/sqrt(sum(x2^2))

# and finally the cubic

x3 = kt$age^3
x3 = residuals(lm(x3~x1+x2))
x3 = x3/sqrt(sum(x3^2))

# check that we have a match

tmp = poly(kt$age,3)
round(sum((tmp[,1]-x1)^2),digits=6)
round(sum((tmp[,2]-x2)^2),digits=6)
round(sum((tmp[,3]-x3)^2),digits=6)
Orthogonal polynomials

This process is quite close to the Gram-Schmidt procedure, with the inclusion of a step to normalize the new predictors so that their sum of squares is 1 -- making this an orthonormal basis.

As we have seen, performing regressions with orthonormal predictors is simple; let's try out our closed form expressions from earlier in the lecture...
# fitting with orthogonal polynomials

# first, some data to make nice plots with

newkt = data.frame(age=seq(40,110,len=500))

fit = lm(sr.ratio~poly(age,deg=3),data=kt)

plot(kt$age,kt$sr.ratio)
lines(newkt$age,predict(fit,newdata=newkt),col=5)

fit = lm(sr.ratio~poly(age,deg=5),data=kt)
lines(newkt$age,predict(fit,newdata=newkt),col=6)

# let's check out the coefficients

summary(fit)

tmp = poly(kt$age,deg=5)
sum(tmp[,1]*kt$sr.ratio)
sum(tmp[,2]*kt$sr.ratio)
sum(tmp[,3]*kt$sr.ratio)
sum(tmp[,4]*kt$sr.ratio)
sum(tmp[,5]*kt$sr.ratio)

coefficients(fit)
Before we leave this...

Earlier in the lecture, we alluded to the fact that we might not have the right functional form or the right set of predictors in our regression equation; in the case of polynomials, this becomes particularly relevant.

We are resorting to polynomials here not because we believe that our response is a polynomial, but instead because polynomials are flexible tools that can approximate a function well -- by taking higher and higher degrees, it is a mathematical fact that you can produce better and better approximations.
Before we leave this...

How does our fit to the strontium data look? Are cubic polynomials sufficient? What about quintic?

How should we proceed? Is the following reasonable? Why? Why not?

```r
plot(kt$age, kt$sr.ratio)
fit = lm(sr.ratio~poly(age, deg=7), data=kt)
lines(newkt$age, predict(fit, newdata=newkt), col=1)
fit = lm(sr.ratio~poly(age, deg=9), data=kt)
lines(newkt$age, predict(fit, newdata=newkt), col=5)
fit = lm(sr.ratio~poly(age, deg=11), data=kt)
lines(newkt$age, predict(fit, newdata=newkt), col=6)
```
Before we leave this...

And here we have our first difference between approximation and estimation; while numerical analysis says we can make a set of polynomials behave and produce a good approximation when we know the full function, when we only have noisy samples, the situation is much different.

Tukey used to say that high-degree polynomials were sharp tools; that it was easy to cut yourself on them -- and now you can see why.
Bias and variance

And so with this polynomial example, we admit that there might be something missing from our model, that we might not have the exact expression that created the data.

In this class we’ll need to make sense of the tradeoffs involved in missing variables, having the wrong functional form, and so on -- our first concept in this regard will be the so-called bias-variance tradeoff.

Specifically, we want to know how close our estimated regression function \( \hat{f} \) can be to \( f^* \), the true function we’re after; how much will \( \hat{f} \) vary if we collect new data?
Bias and variance

To answer this question, we can appeal to a pretty general framework; suppose we have an estimate $\hat{\theta}$ and we want to know how close it is, on average, to a true parameter $\theta$.

One measure of closeness is the mean squared error $E(\hat{\theta} - \theta)^2$ where the expectation is taken over repeated experiments that give rise to new data and new values of the estimator $\hat{\theta}$. 
Bias and variance

If we add and subtract the mean of \( \hat{\theta} \) (again, taken over repeated experiments), we have that

\[
E(\hat{\theta} - \theta)^2 = E(\hat{\theta} - E\hat{\theta} + E\hat{\theta} - \theta)^2
\]

Expanding and canceling a cross term, we find that the mean squared error can be written in two pieces

\[
E(\hat{\theta} - E\hat{\theta})^2 + (E\hat{\theta} - \theta)^2
\]

The first is a variance term (measuring the deviations of \( \hat{\theta} \) around its mean) and the second is a bias term, illustrating how far the center of the distribution of \( \hat{\theta} \) is from the true parameter \( \theta \).
Bias and variance

As we will see in the next few lectures, these concepts motivate much of the work on variable selection, on alternative estimation procedures, and so on.

Next time we will focus on subset selection and on so-called shrinkage estimation, comparing different ways to construct a regression model -- along the way, we will keep returning to the orthogonal model for intuition.
R review - Matrices
Matrices

We can have matrices of the same basic types as vectors, and like vectors they can only hold one type of data

```r
> x # characters
[1,] "a" "f" "k" "p" "u"
[2,] "b" "g" "l" "q" "v"
[3,] "c" "h" "m" "r" "w"
[4,] "d" "i" "n" "s" "x"
[5,] "e" "j" "o" "t" "y"

> y # integers
   [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6
```
Matrices

Note that a different printing convention is followed for matrices; we get an indication of the dimension of the matrix (its rows and columns)

\[
\begin{array}{cccccc}
[1,] & 0.379 & 0.724 & 0.705 & 0.494 & 0.197 & 0.368 & 0.647 \\
[2,] & 0.477 & 0.935 & 0.121 & 0.475 & 0.699 & 0.046 & 0.469 \\
[3,] & 0.316 & 0.306 & 0.875 & 0.387 & 0.029 & 0.156 & 0.960 \\
[4,] & 0.172 & 0.860 & 0.836 & 0.492 & 0.624 & 0.294 & 0.950 \\
[5,] & 0.890 & 0.475 & 0.624 & 0.508 & 0.062 & 0.495 & 0.914 \\
\end{array}
\]

\[
\begin{array}{cccc}
[1,] & TRUE & TRUE & FALSE \\
[2,] & TRUE & TRUE & TRUE \\
\end{array}
\]
Matrices

You can create matrices from vectors

```r
> x = 1:3
> y = c(7,9,11)
> w = cbind(x,y)       # bind together as columns
> w
 [,1] [,2]
[1,] 1  7
[2,] 2  9
[3,] 3 11

> w = rbind(x,y)       # bind together as rows
> w
 [,1] [,2] [,3]
[1,] 1  2  3
[2,] 7  9 11
```
Matrices

And you can use a special constructor

```r
> x = matrix(runif(10),5,2)  # 5 rows and 2 columns
> x
     [,1]       [,2]
[1,] 0.8892423 0.73841712
[2,] 0.3158160 0.09425744
[3,] 0.7490556 0.93721217
[4,] 0.8831179 0.47290106
[5,] 0.2369386 0.96440986
```
Matrices

And you can use a special constructor; you can leave out either the rows or the columns if the data are the right length

```r
> x = matrix(runif(10),5)  # 5 rows and 2 columns
> x = matrix(runif(10),5,2) # 5 rows and 2 columns

R will run through the data, arranging it into the proper shape, filling in columns first; if you want row order, you specify that with

```r
> x = matrix(1:10,5,2,byrow=T)
> x

```

```
[,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```
When R thinks about a matrix, it's really thinking about a vector with some additional information about the dimensions of the matrix; these are stored as an integer vector of length 2.

As we mentioned before, this kind of information is an attribute of the object; attributes can be associated with any object in R.

We can get at the dimension via:

```r
> x = matrix(runif(10), 5, 2)  # 5 rows and 2 columns
> dim(x)
[1] 5 2

> class(x)
[1] "matrix"

> attributes(x)
$dim
[1] 5 2
```
Matrices

Finally, you can change the shape of a matrix

```r
> x = 1:10
> dim(x) = c(2,5)      # remember this syntax also!
> x
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```
Matrices

You can also assign names to the rows and columns of a matrix

```r
> x = matrix(letters[1:10],5,2)
> rownames(x) = c("rumsfeld","bush","rice","gates","cheney")
> x

[,1] [,2]
rumsfeld  "a"  "f"
bush       "b"  "g"
rice       "c"  "h"
gates      "d"  "i"
cheney     "e"  "j"

> colnames(x) = c("var1","var2")
> x

  var1 var2
rumsfeld  "a"  "f"
bush       "b"  "g"
rice       "c"  "h"
gates      "d"  "i"
cheney     "e"  "j"
```
Matrices

These names are also stored as the attributes of the matrix; attributes are stored as a list -- something we will get to shortly

```r
> attributes(x)
$dim
[1] 5 2

$dimnames
$dimnames[['1']]
[1] "rumsfeld" "bush" "rice" "gates"
[5] "cheney"

$dimnames[['2']]
[1] "var1" "var2"
```
### Vectorized operations

As with vectors, R can operate on entire matrices at once

```r
> x
      var1      var2
rumsfeld 0.3344132 0.8754728
bush     0.1028031 0.1464921
rice     0.9924235 0.1076151
gates    0.1934795 0.1459303
cheney   0.3309968 0.7684184

> (sqrt(x) + exp(x))^0.1
      var1      var2
rumsfeld 1.070448 1.128024
bush     1.036335 1.044158
rice     1.139592 1.037257
gates    1.051564 1.044065
cheney   1.070029 1.117343
```
Operations on matrices

R also has a large number of built-in functions to compute with matrices (multiplication, addition) and to form various decompositions

```r
> x %*% y   # matrix multiplication
> w = t(x)   # transpose of x
> xinv = solve(x)  # inverse of x
> d = diag(x)   # extract the diagonal of x
> x = diag(1,10,5)  # create a 10x5 matrix, 1's on the diagonal
> eigen(x)  # the eigenvalue decomposition, output as a list
> qr(x)     # the QR decomposition of X
```
Matrices

Finally, an operator that makes life a lot easier...

> x = matrix(1:50,10)
> dim(x)
[1] 10 5

> apply(x,1,sum)
[1] 105 110 115 120 125 130 135 140 145 150

> apply(x,2,mean)
[1]  5.5 15.5 25.5 35.5 45.5
Subsetting

We have seen the basic syntax for subsetting already but let’s spell it out completely; R has 5 rules for selecting certain elements from a vector; they all use the operator “[”

Indexing by position

Indexing by exclusion

Indexing by name

Indexing with a logical mask

Empty subsetting
Subsetting

Indexing by position

```r
> x = letters
> x[1]
[1] "a"

> x[c(10,11)]
[1] "j" "k"

> x[c(10,11,43)]
[1] "j" "k" NA

> x[0]
character(0)

> x[c(0,10,11)]
[1] "j" "k"
```
Subsetting

Exclusion by position

> letters[-c(10,11)]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "l" "m"
[12] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
[23] "y" "z"

> letters[c(-10,11,43)]
Error: only 0's may mix with negative subscripts
Subsetting

Indexing by name

```r
> x = 1:3
> names(x) = c("a","b","c")

> x["a"]
a1
> x["a","b"]
Error in x["a", "b"] : incorrect number of dimensions

> x[!"a"]
Error in !"a" : Invalid argument to unary operator
```
Subsetting

Indexing with a logical mask

```r
> x = 1:5
> x[c(TRUE,TRUE,FALSE,FALSE,FALSE)]
[1] 1 2

> x >= 3  # remember the recycle rule!
[1] FALSE FALSE TRUE TRUE TRUE

> x[x >= 3]
[1] 3 4 5

> x[x >= 2 & x < 4]  # combining w/ logical operators
[1] 2 3
```
Logical operators

These will appear both in subsetting as well as to control program flow (with loops, if-statements, and so on)

They include operators like <, >, <=, >=, == and !=

On vectors, they apply elementwise (vectorized) and return a vector of logicals (TRUE or FALSE depending on whether the element satisfied the condition or not)

You can combine them with & for AND, and | for OR; the function any(x) returns TRUE if any element in x is true and all(x) returns TRUE if all the elements in x are true
Subsetting

Empty subsetting

> x = 1:5
> x[ ]
[1] 1 2 3 4 5
Subsetting

Uh, why the empty subsetting rule?

The power comes when you think about assignments for vectors, and in particular a slightly more elaborate data structure, the matrix
Assignments for vectors

We can modify specific elements in a vector using our subsetting rules

```r
> x = 1:5
> x[c(2,4)] = 0
> x
 [1] 1 0 3 0 5

> x[x < 2] = 10
> x
 [1] 10 10  3 10  5

> x[] = 5
> x
 [1] 5 5 5 5 5
```
Subsetting

The subsetting syntax for matrices comes right from vectors and we can apply the same basic rules

Indexing by position

```r
> x = matrix(letters[1:10],5,2)
> x[1,2]
[1] “f”
> x[c(3,4,5),c(1,2)]
[,1] [,2]
[1,] “c” “h”
[2,] “d” “i”
[3,] “e” “j”
```
Subsetting

Indexing by exclusion

```r
> x = matrix(letters[1:10],5,2)
> x[1,-2]
[1] “a”
> x[-c(3,4,5),c(1,2)]
[,1] [,2]
[1,] “a” “f”
[2,] “b” “g”
```

Indexing with a logical mask

```r
> x = matrix(letters[1:10],5,2)
> x[c(T,T,T,F,F,F),c(F,T)]
[1] “a” “g” “h”
```
Subsetting

Indexing by name

```r
> x = matrix(letters[1:10],5,2)
> rownames(x) =
+ c("bush","rumsfeld","rice","gates","cheney")
> colnames(x) = c("var1","var2")
> x
  var1 var2
bush   "a"  "f"
rumsfeld "b"  "g"
rice    "c"  "h"
gates   "d"  "i"
cheney    "e"  "j"
> x[c("rice","gates"),c("var1","var2")]
  var1 var2
rice    "c"  "h"
gates   "d"  "i"
```

Subsetting

Empty subsetting

```r
> x var1 var2
  patrick "a" "f"
  matt "b" "g"
  amy "c" "h"
  masanao "d" "i"
  nathan "e" "j"
  > x[,1]
      patrick    matt   amy   masanao    nathan
        "a"     "b"   "c"       "d"       "e"
  > x[,1,drop=F]
    var1
  patrick "a"  
  matt "b"  
  amy "c"  
  masanao "d" 
  nathan "e" 
```
Subsetting

Recall that the empty subset represented the entire vector; so when we read something like \( x[,1] \), we want all of the elements in the empty position (in this case rows).

Note that in the second case, R returned a vector rather than a matrix (all notion of \( \text{var1} \) disappeared); to keep that from happening use the \( \text{drop=F} \) parameter.
Subsetting

Since R views a matrix as a fancy vector, you can make some indexing shortcuts

```r
> x = matrix(letters[1:10],5,2)
> x
[,1] [,2]
[1,] "a"  "f"
[2,] "b"  "g"
[3,] "c"  "h"
[4,] "d"  "i"
[5,] "e"  "j"
> x[rbind(c(1,2),c(1,1),c(3,2))]  # matrix for an index
[1] "f" "a" "h"
> x[1:7]                           # vector for an index
[1] "a" "b" "c" "d" "e" "f" "g"
```
Subsetting

Or what about this?

```r
> x = matrix(letters[1:10],5,2)
> x
 [,1] [,2]
[1,]  "a"  "f"
[2,]  "b"  "g"
[3,]  "c"  "h"
[4,]  "d"  "i"
[5,]  "e"  "j"

> y = matrix(runif(10),5,2)
> y
 [,1]        [,2]
[1,] 0.63061690 0.99150369
[2,] 0.06743318 0.57479874
[3,] 0.94084094 0.95612238
[4,] 0.32397860 0.01915618
[5,] 0.06459324 0.33232415

> x[y>0.5]
 [1]  "a"  "c"  "f"  "g"  "h"
```