OLS Matrix Notation, Cross Validation

Miles Chen, PhD

Department of Statistics

Week 1 Friday

UCLA

Section 1

OLS in Matrix Notation

Useful gradient relations

Important: The gradient will always have the same dimensions as the input vector.

In the following examples, $\mathbf{w}$ and $\mathbf{x}$ are 2 x 1 matrices.

$f(\mathbf{w})$ (1×1)	$\frac{\partial f}{\partial \mathbf{w}}$
$\mathbf{w^T x}$ $(1\times2)\times(2\times1)$	$\mathbf{x}$ (2×1)
$\mathbf{x^T w}$ $(1\times2)\times(2\times1)$	$\mathbf{x}$ (2×1)
$\mathbf{w^T w}$ $(1\times2)\times(2\times1)$	$\mathbf{2w}$ (2×1)
$\mathbf{w^T C w}$ for symmetric $\mathbf{C}$ $(1\times2)\times(2\times2)\times(2\times1)$	$\mathbf{2Cw}$ $(2\times2)\times(2\times1)$

Back to OLS Regression Loss (Matrix notation)

Earlier, we found the total OLS regression loss function to be:

$$\mathcal{L} = \frac{1}{N}\mathbf{t}^T\mathbf{t} - \frac{2}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{t} + \frac{1}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}$$

We will take the gradient of the loss with respect to the vector $\mathbf{w}$, set the gradient equal to the 0 vector, and then solve for $\mathbf{w}$.

If $\mathbf{w}$ is a 2 x 1 vector, then ∇f must also be 2 x 1.

$$\begin{aligned}\underset{(2\times 1)}{\frac{\partial \mathcal{L}}{\partial \mathbf{w}}} &= \underset{(2\times 1)}{\mathbf{0}} - \underset{(2\times N)\times(N\times 1)}{\frac{2}{N}\mathbf{X}^T\mathbf{t}} + \underset{(2\times N)\times(N\times 2)\times(2\times 1)}{\frac{2}{N}\mathbf{X}^T\mathbf{X}\mathbf{w}} \\ &= \underset{(2\times 1)}{\frac{-2}{N}\mathbf{X}^T\mathbf{t}} + \underset{(2\times 1)}{\frac{2}{N}\mathbf{X}^T\mathbf{X}\mathbf{w}}\end{aligned}$$

Solving for the $\mathbf{w}$ that minimizes the OLS Loss

Set the gradient equal to 0 and solve for $\mathbf{w}$:

$$\begin{aligned} \mathbf{0} &= \frac{-2}{N}\mathbf{X}^T\mathbf{t} + \frac{2}{N}\mathbf{X}^T\mathbf{X}\mathbf{w} \\ \mathbf{X}^T\mathbf{t} &= \mathbf{X}^T\mathbf{X}\mathbf{w} \\ \underset{(2\times N)(N\times 2)}{(\mathbf{X}^T\mathbf{X})^{-1}}\underset{(2\times N)(N\times 1)}{\mathbf{X}^T\mathbf{t}} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\mathbf{w} \\ \underset{(2\times 1)}{(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}} &= \mathbf{w} \end{aligned}$$

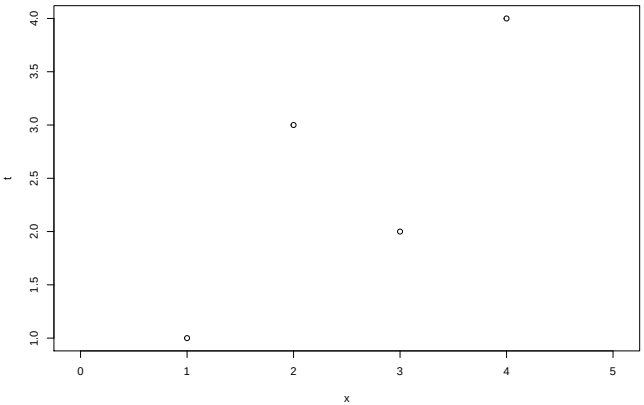
OLS Result

The estimate of $\mathbf{w}$ that minimizes the loss function of OLS regression is $\hat{\mathbf{w}}$:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

Numeric Example

```
x <- c(1, 2, 3, 4); t <- c(1, 3, 2, 4)
plot(x, t, asp = 1)
```



```r
X <- cbind(1, x) # x matrix, n by 2
print(X)
```

```
##         x
## [1,] 1 1
## [2,] 1 2
## [3,] 1 3
## [4,] 1 4
```

```r
t(X) %*% X  # X-Transpose times X is 2 by 2
```

```
##       x
##    4 10
## x 10 30
```

```
solve(t(X) %*% X)  # (xTx)^-1 is 2 by 2
```

```
##           x
##    1.5 -0.5
## x -0.5  0.2
```

```
solve(t(X) %*% X) %*% t(X) # X-transpose X inverse times X-transpose is 2 by N
```

```
##   [,1] [,2] [,3] [,4]
##    1.0  0.5  0.0 -0.5
## x -0.3 -0.1  0.1  0.3
```

```
w_hat <- solve(t(X) %*% X) %*% t(X) %*% t
w_hat # w-hat is 2 by 1
```

```
##   [,1]
##    0.5
## x  0.8
```

The coefficient estimates are 0.5 for the intercept, and 0.8 for the slope.

```
lm(t ~ x)
```

```
##
## Call:
## lm(formula = t ~ x)
##
## Coefficients:
## (Intercept)            x
##         0.5          0.8
```

Using R's lm function produces the same results as our matrix multiplication operations.

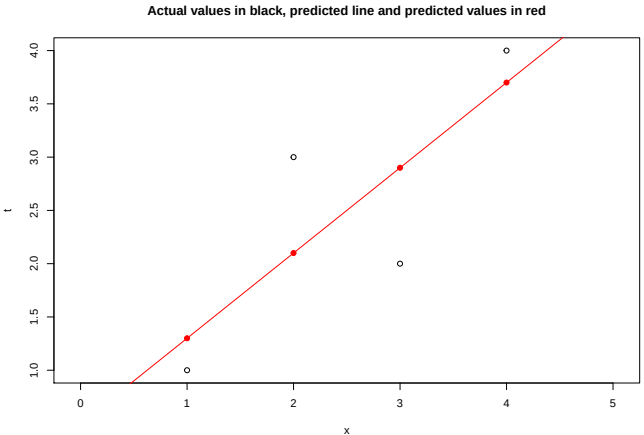
Predicted values

To see what the model predicts, we can just perform the matrix multiplication: $\mathbf{X}\hat{\mathbf{w}}$

```
t_hat <- X %*% w_hat  # predicted vals is product of X and est. coefficients
t_hat
```

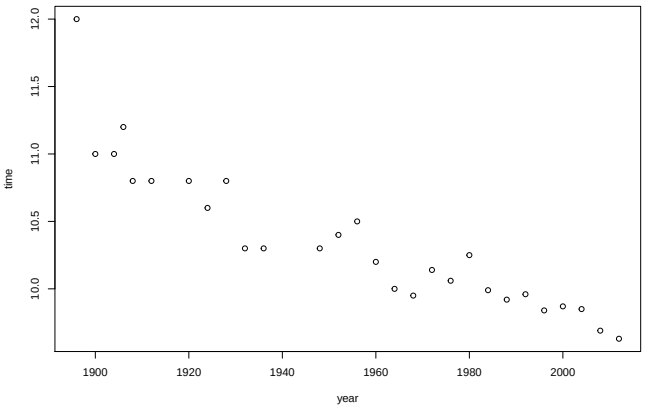
```
##      [,1]
## [1,]  1.3
## [2,]  2.1
## [3,]  2.9
## [4,]  3.7
```

```r
plot(x, t, asp = 1, main = "Actual values in black, predicted line and predicted values in red")
abline(w_hat[1], w_hat[2], col = 'red')
points(x, t_hat, col = "red", pch = 19)
```



Olympic Data

```
url <- "https://raw.githubusercontent.com/sdrogers/fcmlcode/master/R/data/olympics/male100.csv"
olympic <- read.csv(url, header = FALSE, col.names = c("year","time"))
plot(olympic)
```



Fitting the model with `lm()`

Here are the results of fitting the model using `lm()`

```
linear_fit <- lm(olympic$time ~ olympic$year)
linear_fit
```

```
##
## Call:
## lm(formula = olympic$time ~ olympic$year)
##
## Coefficients:
##  (Intercept)  olympic$year
##     36.30912      -0.01328
```

Creating our X Matrix

We'll also try estimating the slope and intercept using matrix operations. First, we create the X matrix.

```
x <- olympic$year
X <- cbind(1, x)
print(X)
```

```
##             x
##  [1,] 1 1896
##  [2,] 1 1900
##  [3,] 1 1904
##  [4,] 1 1906
##  [5,] 1 1908
##  [6,] 1 1912
##  [7,] 1 1920
##  [8,] 1 1924
##  [9,] 1 1928
## [10,] 1 1932
## [11,] 1 1936
## [12,] 1 1948
## [13,] 1 1952
## [14,] 1 1956
## [15,] 1 1960
## [16,] 1 1964
```

Estimating w

We estimate the coefficients with $\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$

```
w_hat <- solve( t(X) %*% X) %*% t(X) %*% matrix(olympic$time)
w_hat
```

```
##           [,1]
##   36.30912041
## x -0.01327532
```

Again, the coefficient estimates here match the results from `lm()`.

Predicted values

To see what the model predicts, we can just perform the matrix multiplication: $\mathbf{X}\hat{\mathbf{w}}$

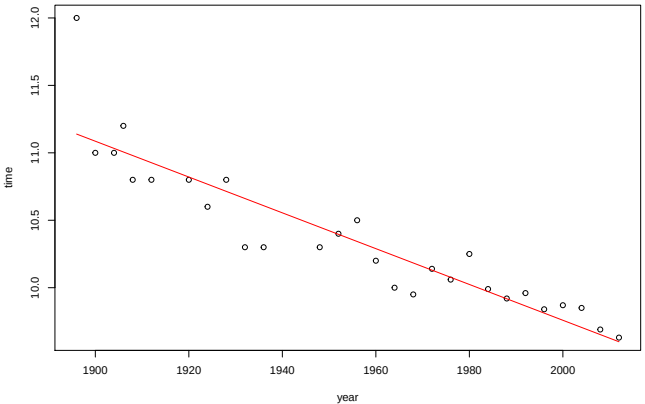
```
t_hat <- X %*% w_hat
print(t_hat)
```

```
##             [,1]
##  [1,] 11.139106
##  [2,] 11.086005
##  [3,] 11.032904
##  [4,] 11.006353
##  [5,] 10.979803
##  [6,] 10.926701
##  [7,] 10.820499
##  [8,] 10.767397
##  [9,] 10.714296
## [10,] 10.661195
## [11,] 10.608093
## [12,] 10.448790
## [13,] 10.395688
## [14,] 10.342587
## [15,] 10.289486
## [16,] 10.236384
## [17,] 10.183283
## [18,] 10.130182
```

Plot the fit

Here is a plot of the actual values and the fitted linear regression line.

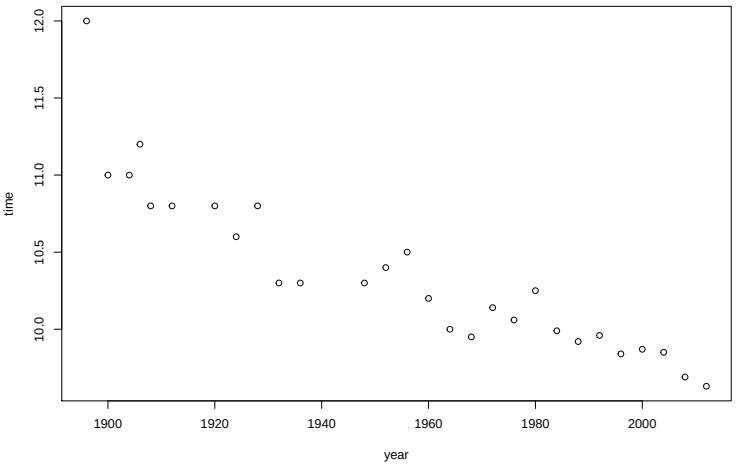
```
plot(olympic$year, olympic$time, ylab = 'time', xlab = 'year')
lines(olympic$year, t_hat, col = 'red')
```



Section 2

A Non-linear Response from Linear Model

Olympic Data



A Linear model with non-linear terms

We can perform Least Squares regression with polynomial values by expanding our $\mathbf{X}$ matrix with additional columns.

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

This matrix can be constructed in R as follows:

```
x <- olympic$year      # little x for a vector of values
X <- cbind(1, x, x^2)  # big X for the matrix, which is created with cbind
colnames(X) <- c('1','x', 'x_sq')
head(X)
```

```
##      1    x    x_sq
## [1,] 1 1896 3594816
## [2,] 1 1900 3610000
## [3,] 1 1904 3625216
## [4,] 1 1906 3632836
## [5,] 1 1908 3640464
## [6,] 1 1912 3655744
```

Multivariable linear model

The formula for our predictions is:

$$f(x_n; w_0, w_1, w_2) = w_0 + w_1 x_n + w_2 x_n^2$$

In matrix notation, our predicted values are expressed with:

$$\mathbf{w}^T \mathbf{x}_n$$

which is exactly what it was for our simple linear model of one variable.

The column matrix of values predicted by the function $f(x_n; w_0, w_1, w_2)$ is:

$$\mathbf{Xw} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix} \times \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} w_0 + w_1 x_1 + w_2 x_1^2 \\ w_0 + w_1 x_2 + w_2 x_2^2 \\ \vdots \\ w_0 + w_1 x_N + w_2 x_N^2 \end{bmatrix}$$

Which matches what we want:

$$f(x_n; w_0, w_1, w_2) = w_0 + w_1 x_n + w_2 x_n^2$$

The Loss function

Because the predicted values are given by the same matrix operation ($\mathbf{Xw}$), the Loss function remains exactly as it was:

$$\mathcal{L} = \frac{1}{N}(\mathbf{t} - \mathbf{Xw})^T(\mathbf{t} - \mathbf{Xw})$$

We can see that by using matrix notation, we can add additional terms or columns in our X matrix, and the formulas remain unchanged.

Estimating the parameters

We can follow the exact same steps of derivation in matrix notation that we used for the OLS regression model of one variable. We will find that the best estimates of $\mathbf{w}$ will still be:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

Computational Issues

If we attempt to perform these calculations in R for our Olympic data, however, we run into an error.

```
solve(t(X) %*% X) %*% t(X) %*% matrix(olympic$time)
```

```
## Error in solve.default(t(X) %*% X): system is computationally singular: reciprocal condition number =
```

What's the issue?

Let's look at the matrix: $(\mathbf{X}^T\mathbf{X})$ before we try to find its inverse.

```
t(X) %*% X
```

```
##                1            x         x_sq
## 1             28        54726 1.069985e+08
## x          54726    106998484 2.092712e+11
## x_sq   106998484 209271175176 4.094385e+14
```

Floating-point limits

The error comes when we try to inverse the matrix $(\mathbf{X}^T\mathbf{X})$.

One of the columns in $\mathbf{X}$ is the year squared, which produces very large numbers. For year 2000, the square is 4,000,000.

When we do the operation of multiplying the matrix by its tranpose, we are multiplying that column against itself and summing it up, which produces a huge value 10^{14}.

Finding the inverse produces some extremely small values when multiplying the constant (1/determinant). When R detects that the value is smaller than Machine Epsilon, it throws an error to avoid computational instability. The computer then declares that the matrix inverse does not exist (which is not true).

Scale down the numbers

To deal with this computational issue, we can try a trick: simply multiply the x^2 column by some factor to reduce its size (the opposite can be done if the values need to be made larger).

```
x <- olympic$year
X_sc <- cbind(1, x, 0.0001 * x^2)  # X_sc for X scaled
colnames(X_sc) <- c('1','x', 'x_sq')
head(X_sc)
```

```
##      1    x     x_sq
## [1,] 1 1896 359.4816
## [2,] 1 1900 361.0000
## [3,] 1 1904 362.5216
## [4,] 1 1906 363.2836
## [5,] 1 1908 364.0464
## [6,] 1 1912 365.5744
```

Estimating $\mathbf{w}$ with matrix operations

With the scaled values, we can now calculate the matrix inverse. We compute our best-fitting coefficients with matrix operations: $\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$

```
w_hat <- solve(t(X_sc) %*% X_sc) %*% t(X_sc) %*% matrix(olympic$time)
w_hat
```

```
##              [,1]
## 1    396.3285754
## x     -0.3820779
## x_sq   0.9441766
```

Fitting the model with `lm()` for comparison.

We can compare the values with what R calculates with `lm()`. When using `lm()`, we write `0 + X_sc` to avoid fitting a separate intercept term. The matrix has a column of 1s to handle the intercept term already.

```
quad_model <- lm(olympic$time ~ 0 + X_sc)
quad_model$coefficients
```

```
##       X_sc1       X_scx    X_scx_sq
## 396.3285758  -0.3820779   0.9441766
```

We see our matrix operations match R's coefficients exactly (even though R does not use the matrix operations to find its estimates).

Calculating the fitted values (matrix operations)

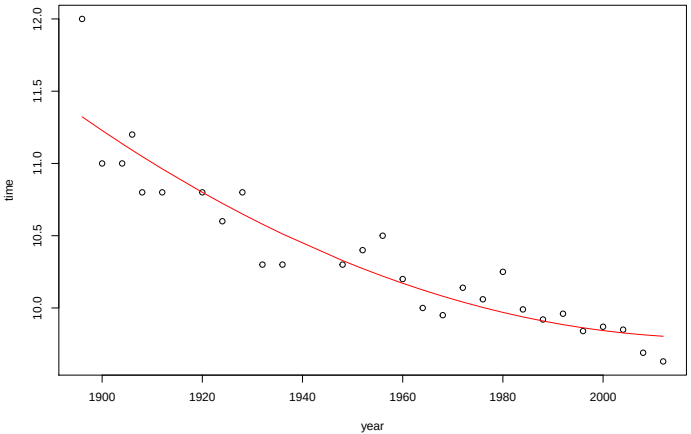
With the resulting `w_hat` matrix, we can see what values our model predicts for each observation (i.e. the fitted values) by performing a simple matrix multiplication: $\mathbf{X}\hat{\mathbf{w}}$

```
t_hat <- X_sc %*% w_hat
print(t_hat)
```

```
##             [,1]
##  [1,] 11.322942
##  [2,] 11.228268
##  [3,] 11.136616
##  [4,] 11.091923
##  [5,] 11.047985
##  [6,] 10.962375
##  [7,] 10.800219
##  [8,] 10.723673
##  [9,] 10.650149
## [10,] 10.579646
```

I can now plot those predicted values to see how our model fits.

```
plot(olympic$year, olympic$time, ylab = 'time', xlab = 'year')
lines(olympic$year, t_hat, col = 'red')
```



Function poly()

While scaling down big numbers works, for polynomials of high degree, our values can become very highly correlated, which introduces other problems. One solution is to use R's poly() function which will scale the values and calculate uncorrelated polynomial values.

```
x <- olympic$year
Xpoly <- poly(x, 2)   # create the X matrix
head(Xpoly)
```

```
##                1          2
## [1,] -0.3061317 0.33961254
## [2,] -0.2851996 0.26281252
## [3,] -0.2642675 0.19159407
## [4,] -0.2538015 0.15807793
## [5,] -0.2433355 0.12595719
## [6,] -0.2224034 0.06590188
```

Finding the coefficients with `lm()`

This time, there is no column of 1s so we do not need to add a `0 +` to the `lm()` call. R will fit an intercept for us automatically.

```
poly2model <- lm(olympic$time ~ Xpoly)
poly2model
```

```
##
## Call:
## lm(formula = olympic$time ~ Xpoly)
##
## Coefficients:
## (Intercept)       Xpoly1       Xpoly2
##     10.3625      -2.5368       0.5413
```

Seeing the fitted values

When using `lm()`, the fitted values are stored in the list as `..$fitted.values`. No need for an additional matrix multiplication step.

```
poly2model$fitted.values
```

```
##         1         2         3         4         5         6         7         8
## 11.322942 11.228268 11.136616 11.091923 11.047985 10.962375 10.800219 10.723673
##         9        10        11        12        13        14        15        16
## 10.650149 10.579646 10.512164 10.327847 10.272451 10.220076 10.170722 10.124390
##        17        18        19        20        21        22        23        24
## 10.081080 10.040790 10.003522  9.969275  9.938050  9.909846  9.884664  9.862502
##        25        26        27        28
##  9.843363  9.827244  9.814147  9.804071
```

A note

An unfortunate side effect of using `poly()` is that all of the X values get transformed, so the interpretability of coefficients is dramatically reduced. The coefficients for each term only make sense for the transformed values.

Using `poly()` is still useful, however, in that we can compare polynomial fits of different orders.

We can create several models and compare the resulting fits, even if we lose the interpretability of the coefficients.

Fitting Higher order polynomials

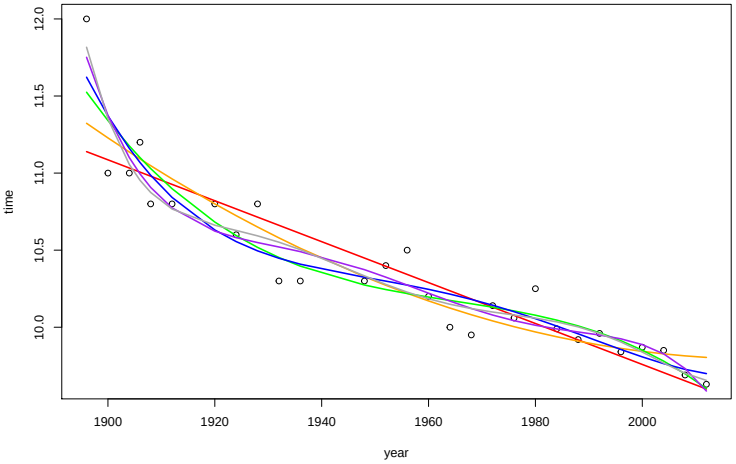
Here I will fit higher order polynomial models directly with `lm()` and `poly()`.

```
poly3model <- lm(olympic$time ~ poly(x, 3))
poly4model <- lm(olympic$time ~ poly(x, 4))
poly5model <- lm(olympic$time ~ poly(x, 5))
poly6model <- lm(olympic$time ~ poly(x, 6))
```

Plot the fitted models

```
plot(olympic$year, olympic$time, ylab = 'time', xlab = 'year')
lines(olympic$year, linear_fit$fitted.values, col = 'red', lwd = 2)
lines(olympic$year, poly2model$fitted.values, col = 'orange', lwd = 2)
lines(olympic$year, poly3model$fitted.values, col = 'green', lwd = 2)
lines(olympic$year, poly4model$fitted.values, col = 'blue', lwd = 2)
lines(olympic$year, poly5model$fitted.values, col = 'purple', lwd = 2)
lines(olympic$year, poly6model$fitted.values, col = 'darkgray', lwd = 2)
```

Plot the fitted models



Model selection - Which model should we use?

The classical approach would be to compare models by looking at the summary tables. Perhaps we would compare AIC or BIC values to help us decide which model is best.

```
summary(poly3model) # summary table for the cubic fit
```

```
##
## Call:
## lm(formula = olympic$time ~ poly(x, 3))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.34104 -0.09704  0.00006  0.07719  0.47550
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.36250    0.03452 300.186  < 2e-16 ***
## poly(x, 3)1 -2.53684    0.18266 -13.888 5.74e-13 ***
## poly(x, 3)2  0.54131    0.18266   2.963  0.00677 **
## poly(x, 3)3 -0.51534    0.18266  -2.821  0.00945 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1827 on 24 degrees of freedom
## Multiple R-squared:  0.8973, Adjusted R-squared:  0.8844
## F-statistic: 69.87 on 3 and 24 DF,  p-value: 5.297e-12
```

Model Selection

```
summary(poly6model) # summary table for the 6th order fit
```

```
##
## Call:
## lm(formula = olympic$time ~ poly(x, 6))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.35232 -0.06672 -0.01519  0.09227  0.27252
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.36250    0.03189 324.982  < 2e-16 ***
## poly(x, 6)1 -2.53684    0.16873 -15.035 1.03e-12 ***
## poly(x, 6)2  0.54131    0.16873   3.208  0.00422 **
## poly(x, 6)3 -0.51534    0.16873  -3.054  0.00602 **
## poly(x, 6)4  0.24248    0.16873   1.437  0.16543
## poly(x, 6)5 -0.31790    0.16873  -1.884  0.07347 .
## poly(x, 6)6  0.20758    0.16873   1.230  0.23220
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1687 on 21 degrees of freedom
## Multiple R-squared:  0.9233, Adjusted R-squared:  0.9014
## F-statistic: 42.13 on 6 and 21 DF,  p-value: 1.213e-10
```

Measuring model performance

Measuring model performance is not always straightforward.

If we look only at the R-squared value, more complicated models will always outperform simpler models.

If we compare summary tables of the 6th order polynomial fit and the 3rd order polynomial fit, the 4th, 5th, and 6th order components are not significant at the 0.05 level. Meanwhile the intercept, linear, quadratic, and cubic components are significant.

Does that mean we take the 3rd order polynomial fit and be done?

We'll take a closer look at model selection next week with cross-validation.