STAT340 Lecture 11: cross validation and model selection

Brian Powers

Updated 12/2/2024

Introduction

In our discussions of regression the past few weeks, we have encountered many situations in which we needed to make a choice about the model that we fit to data. For example, suppose we have a data set with hundreds or even thousands of predictor variables. This is frequently the case in applications to genetics, where we have thousands of genes, and we want to predict some outcome (e.g., disease status). How do we decide which variables to include in linear regression (or any other prediction model)?

This is an example of *model selection*, our subject for this week.

Learning objectives

After this lecture, you will be able to

- Explain the problem of variable selection in the context of linear regression
- Explain and apply cross-validation methods, including leave-one-out cross-validation and K-fold cross-validation.
- Explain subset selection methods, including forward and backward stepwise selection.
- Explain and apply regularization and shrinkage methods, including ridge regression and the LASSO.

Model selection: overview

Our focus this week will be on model selection for regression problems. Still, we should note that similar ideas apply in many other situations. For example, when *clustering* data, we use model selection to choose how many clusters to group the data into.

The unifying idea is that we have to choose among many different *similar* ways of describing the data. That's what model selection helps us do.

Variable selection

Suppose that we have a collection of p predictors, and that p is very large (say, $p \approx n$). If we try to fit a model using all of these predictors, we will end up over-fitting to the data.

We have n equations in p unknowns. When p is of a similar size to the number of observations n, the system is over-determined (or close to it).

In situations like this, we would like to choose just a few of these predictors for inclusion in a statistical model (e.g., linear regression). This is an example of model selection: we have a bunch of different models under consideration (i.e., a different possible model for each set of variables we might choose), and we want to pick the best one.

The natural question, then, is: how do we compare models?

Example: `mtcars` I

Let's consider a very simple example, adapted from Section 3.3.2 and Section 5.1 in ISLR, revisiting our old friend the `mtcars` data set.

```
data('mtcars');
head(mtcars);
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

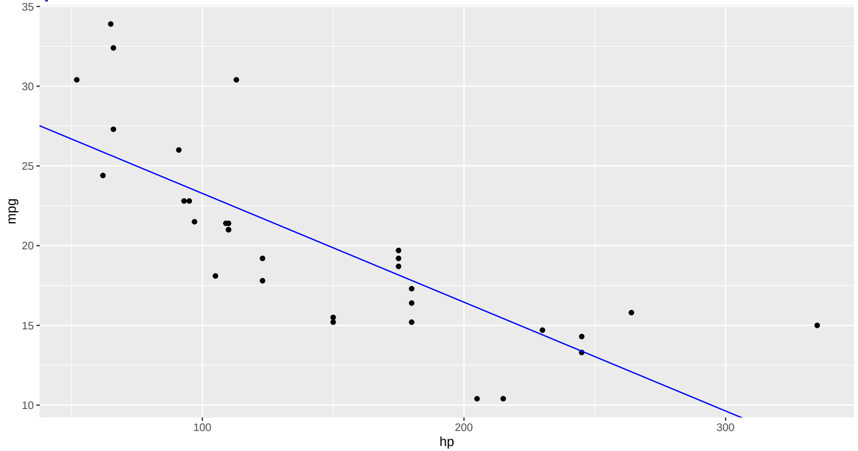
Let's confine our attention to the `mpg` (miles per gallon) and `hp` (horsepower) variables. Can we predict gas mileage from the horsepower?

Example: `mtcars` II

Let's try fitting linear regression.

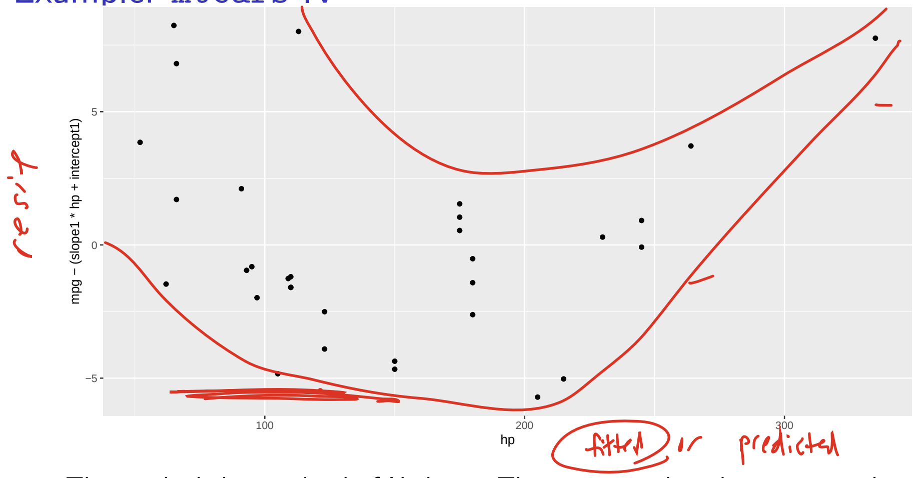
```r
model1 <- lm(mpg ~ 1 + hp, mtcars);
intercept1 <- model1$coefficients[1];
slope1 <- model1$coefficients[2];

# Plot the data itself
pp <- ggplot( mtcars, aes(x=hp, y=mpg)) + geom_point();
pp <- pp + geom_abline(intercept=intercept1, slope=slope1, colour='blue' );
```

Example: `mtcars` III



Okay, it looks reasonable, but you might notice that the residuals have a bit of a weird behavior.

Example: `mtcars` IV



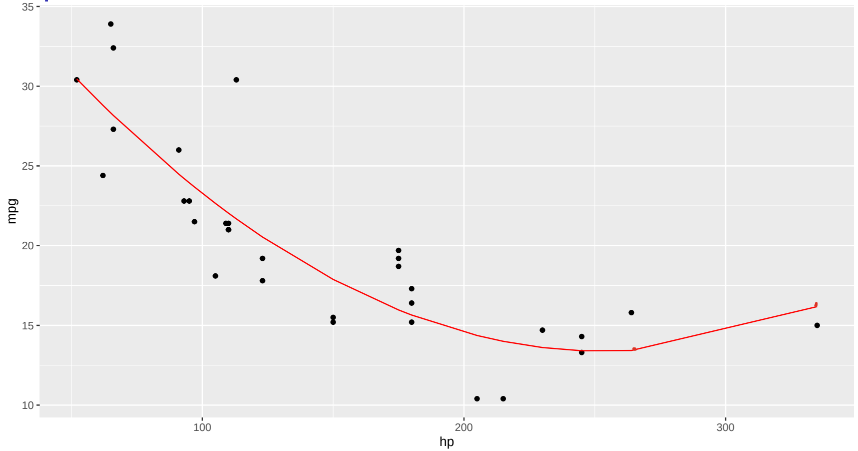
The residuals have a kind of U-shape. This suggests that there is a non-linearity in the data that we are failing to capture.

Example: `mtcars` V

Let's try adding another predictor: the squared horsepower.

```r
model2 <- lm(mpg ~ 1 + hp + I(hp^2), mtcars);
intercept2 <- model2$coefficients[1];
slope2_1 <- model2$coefficients[2];
slope2_2 <- model2$coefficients[3];

# Plot the data itself
pp <- ggplot( mtcars, aes(x=hp, y=mpg)) + geom_point();
# As usual, there are cleaner ways to do this plot, but this is the quick and easy way to make it.
# If we were doing this more carefully, we would evaluate the curve in the plot at more x-values than
pp <- pp + geom_line(data=mtcars, aes(x=hp, y=intercept2 + slope2_1*hp + slope2_2*hp^2 ), color='red'
```

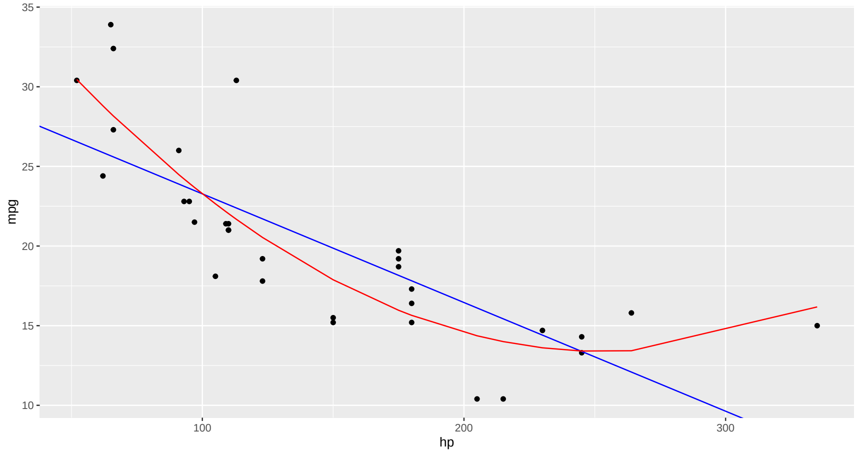
Example: `mtcars` VI



That looks like quite an improvement!

Example: `mtcars` VII

Just for comparison:



Example: `mtcars` VIII

We can also compare the squared residuals to confirm that adding the feature `hp^2` actually decreased our error:

```
c( sum( model1$residuals^2 ), sum( model2$residuals^2 ) );
## [1] 447.6743 274.6317
```

Why stop there? Why not add `hp^3` as well, or even `hp^4`? Well, funny enough, that is precisely the idea behind *polynomial regression*, which you can learn more about in ISLR (Section 3.3.2; more substantial discussion in Chapter 7) or in a regression course.

But that raises the question: how do we know when to stop?

-You'll find that if you add `hp^3` to the model above, that the sum of squared residuals does indeed improve.

-But how do we know if that improvement is worth it?

Model Comparison Statistics: Adjusted R^2, AIC and BIC

Rather than comparing the RSS of two models, which only compares the reduction to residuals with no regards to the number of predictors (the complexity) in the model, there are some statistics that are often used. Note that these statistics are relevant when comparing models of *different* complexity - two models with the same number of predictors would just as well be compared using RSS. These statistics are useful to balance the benefit of reduced RSS with the cost of additional model complexity.

In each case, k is the number of parameters being estimated - including the intercept.

Adjusted R^2

$$R^2 = 1 - \frac{RSS}{TSS}$$

$$R^2_{adj} = 1 - \frac{RSS/(n-k)}{TSS/(n-1)} = \boxed{1 - \frac{RSS}{TSS}\left(\frac{n-1}{n-k}\right)}$$

The fraction multiplied will be > 1, and grows with model complexity. In an extreme case, this penalty can result in a negative R^2_{adj}, so it's important to remember that this statistic is not meaningful by itself, only when used to compare models. Thus $R^2_{adj} < R^2$, and applies a penalty that grows with the number of predictors. When comparing two models using R^2_{adj} we prefer the model that has the higher value.

Akaike information criterion (AIC)

$$AIC = -2\ln(L) + 2k$$

If you work out the math (we won't here) for a linear model this can be expressed in terms of *RSS*

$$AIC = n\ln(RSS/n) + 2k$$

For a logistic model

$$AIC = \text{Residual Deviance} + 2k$$

When comparing models we prefer the model with the **lower** AIC.

Bayesian information criterion (BIC)

Compared to AIC, The Bayesian information criterion imposes a steeper penalty on the number of estimated coefficients *as long as n is large enough*.

$$BIC = -2\ln(L) + \underline{\ln(n)k} = n\ln(RSS/n) + \ln(n)k$$

$\ln(n) > 2$ when $n > 7$, which will most often be the case.

The first term in AIC and BIC is the residual deviance, which we want to be as low as possible. While a more complex model will reduce residual deviance, both AIC and BIC add a penalty. BIC adds a more severe penalty per predictor (if $n > e^2 \approx 7.4$).

The bottom line: between these three model comparison statistics, BIC more heavily favors simpler models, R^2_{adj} allows for more complex models and AIC is somewhere in the middle.

Overfitting and Unseen Data

If we keep adding more predictors to our model, the residuals will continue to decrease, but this will not actually mean that our model is better. Instead, what we will be doing is *over-fitting* to the data. That is, our model will really just be "memorizing" the data itself rather than learning a model.

The true test of model quality is how well it does at predicting for data that we *didn't* see.

That is, if we fit our model on data (X_i, Y_i) for $i = 1, 2, \dots, n$, how well does our model do on a previously unseen data point (X_{n+1}, Y_{n+1})?

Specifically, in the case of regression, we want our model to minimize

$$\mathbb{E}\left(\hat{Y}_{n+1} - Y_{n+1}\right)^2,$$

where $\hat{Y}_{n+1}$ is our model's prediction based on coefficients estimated from our n training observations.

Validation Sets I

So rather than focusing on how well our model fits our training data, we should be trying to determine how well our model does when it gets applied to data that we haven't seen before.

Specifically, we would like to know the mean squared error (MSE),

$$\mathbb{E}\left(\hat{Y}_{n+1} - Y_{n+1}\right)^2 .$$

We always train ("fit" in the language of statistics) our model on a *training set*, and then assess how well the model performs on a *test set* that our model hasn't seen before.

The trouble is that in most statistical problems, we have at most a few hundred data points to work with. It can seem painful to set aside some of our data just to use as a test set.

Validation Sets II

Following the logic of the train/test split idea in ML, though, a natural approach is to do the following:

partition

1. Split our data into two parts, say $\underline{S_1, S_2}$, such that $S_1 \cup S_2 = \{1, 2, \ldots, n\}$ and $S_1 \cap S_2 = \emptyset$.
2. Obtain estimate $\hat{\beta}_1$ by fitting a model on the observations in $\underline{S_1}$ — training
3. Evaluate the error of our fitted model on S_2,

esimate of MSE ←
$$\hat{E}_1 = \frac{1}{|S_2|} \sum_{i \in S_2} \Big(Y_i - \overset{\hat{y}_i}{\cancel{\hat{\beta}_1}} X_i \Big)^2 .$$
— test data

Typically, we call S_2, the set that we make predictions for, the *validation set*, because it is validating our model's performance.

Example: mtcars revisited I

each of size 16

Let's see this in action on the mtcars data set.

We randomly split the data set into two groups. For each model order 1, 2, 3, 4 and 5, we fit the model to the training set and then measure the sum of squared residuals of that model when applied to the validation set.

One run of this experiment is summarized in resids_onerun. For details, refer to mtcars_poly.R, which is included among the supplementary files for this lecture.

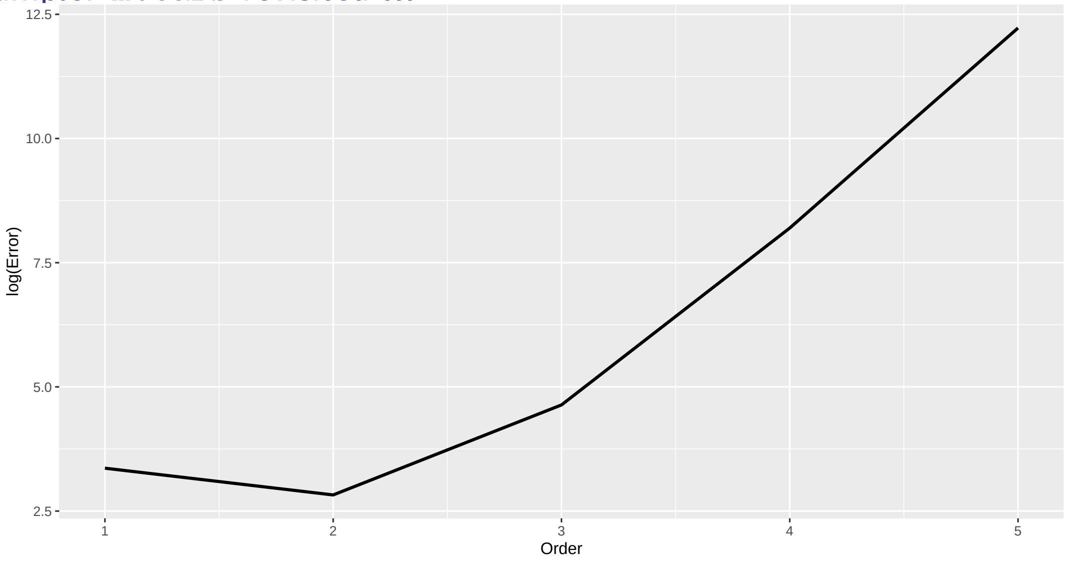
```
source('mtcars_poly.R');

head(resids_onerun);
##   Order        Error
## 1     1     28.88989
## 2     2     16.85463
## 3     3    103.17567
## 4     4   3633.52482
## 5     5 203810.03011
```

Example: `mtcars` revisited II

```
set.seed(1)
# Plot these results
pp <- ggplot(resids_onerun, aes(x=Order, y=log(Error) ) );
pp <- pp + geom_line( size=1)
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
pp
```

Example: `mtcars` revisited III

Validation



Example: `mtcars` revisited IV

Let's pause to make sure that we understand what this plot actually shows. We split the `mtcars` dataset randomly into two sets, a "training" set and a "validation" set. For each order (1, 2, 3, 4, 5), we fit a model of that order to the training set. Then we use that model to try and predict the outcomes (`mpg`) on the validation set. So this is the performance of five different models, each trained on the same data S_1 and evaluated on the same data S_2, *different from the training data*.

Looking at the plot, we see that as we add higher-order powers of `hp`, we don't really gain much in terms of the error (i.e., sum of squared residuals) beyond order 2. Indeed, past the order-3 model, the error gets worse again!

Aside: this deteriorating performance is due largely to the fact that the `mtcars` data set is so small. Once we split it in half, we are fitting our model to just 16 observations. Estimating four or five coefficients from only about 15 observations is asking for trouble! This is a tell-tale sign of over-fitting of a model. This would be a good occasion for some kind of *regularization*, but we'll come back to that.

Variance in the residuals I

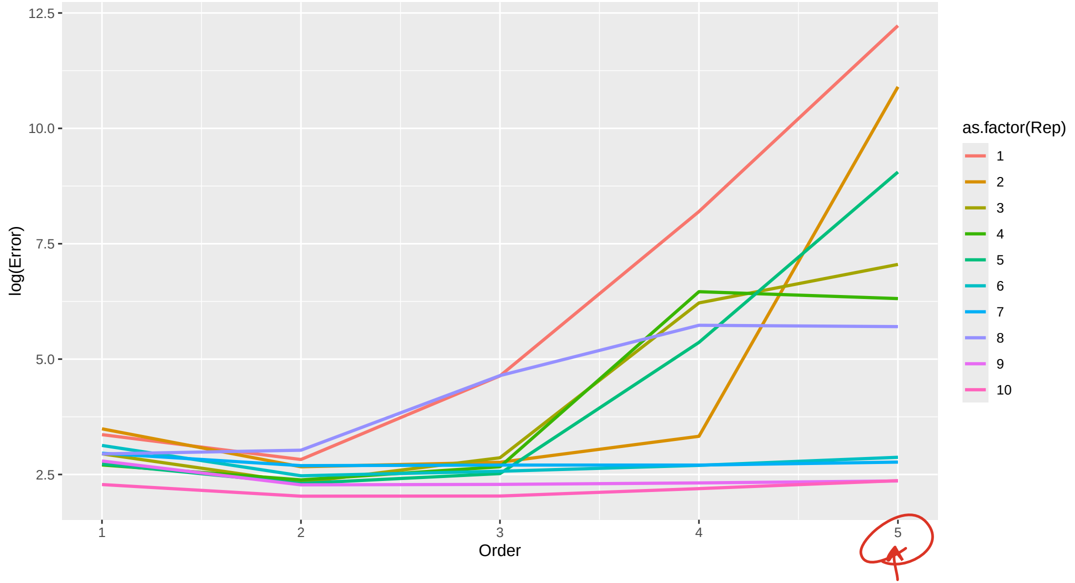
There's one problem, though, beyond matters of sample size. That plot shows the residuals as a function of model order for one particular random set S_1. Let's plot the same residuals for a few different random sets.

Note: the data frame `resids` contains multiple replicates of the above experiment. Once again, refer to the code in `mtcars_poly.R` for details.

```
head(resids)
##   Rep Order    Error
## 1   1     1 28.88989
## 2   2     1 32.77625
## 3   3     1 19.05170
## 4   4     1 14.98573
## 5   5     1 15.28959
## 6   6     1 22.85369
```

```
pp <- ggplot(resids, aes(x=Order, y=log(Error), color=as.factor(Rep) ) );
pp <- pp + geom_line( size=1)
pp
```

Variance in the residuals II



Variance in the residuals III

Hmm. There's quite a lot of variance among our different estimates of the prediction error. Note that the y-axis is on a log scale, so an increase from, say, 2 to 3 is an *order of magnitude* increase in error.

Each of these is supposed to be estimating the error

$$\mathbb{E}\left(\hat{Y}_{n+1} - Y_{n+1}\right)^2,$$

but there's so much variation among our estimates that it's hard to know if we can trust any one of them in particular!

Indeed, the variance is so high that we needed to plot the error on a log scale! Once in a while, we get unlucky and pick an especially bad train/validate split, and the error is truly awful!

Question: what explains the variance among the different lines in that plot?

Question: How might we reduce that variance?

Reducing the variance: Leave-one-out cross-validation I

One source of variance in our cross-validation plots above was the fact that each replicate involved splitting the data in half and training on only one of the two halves.

That means that on average, from one replicate to another, the data used to train the model changes quite a lot, and hence our estimated model changes a lot. That's where the variance comes from in the plot we just looked at!

There is also the related problem that we are training on only half of the available data. As statisticians and/or machine learners, we don't like not using all of our data!

So, here's one possible solution: instead of training on half the data and validating (i.e., evaluating the model) on the other half, let's train on all of our data except for one observation, then evaluate our learned model on that one held-out data point.

That is, instead of splitting our data into two halves, we

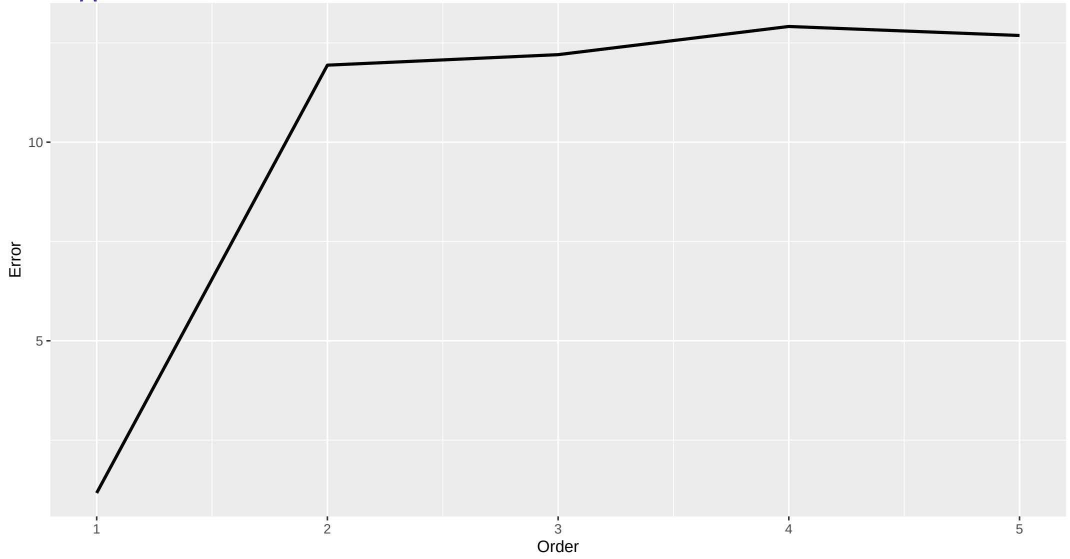
1. Take one observation and set it aside (i.e., hold it out)
2. Train our model on the other $n-1$ observations
3. Evaluate our model on the held-out observation.

Reducing the variance: Leave-one-out cross-validation II

This is called *leave-one-out cross-validation* (LOO-CV).

```r
set.seed(2)
# This R file implements the same experiment as we saw above,
# but this time doing LOO-CV instead of a naive two-set split.
source('mtcars_poly_loocv.R');

pp <- ggplot(resids_onerun, aes(x=Order, y=Error ) );
pp <- pp + geom_line( size=1)
pp
```

Reducing the variance: Leave-one-out cross-validation III

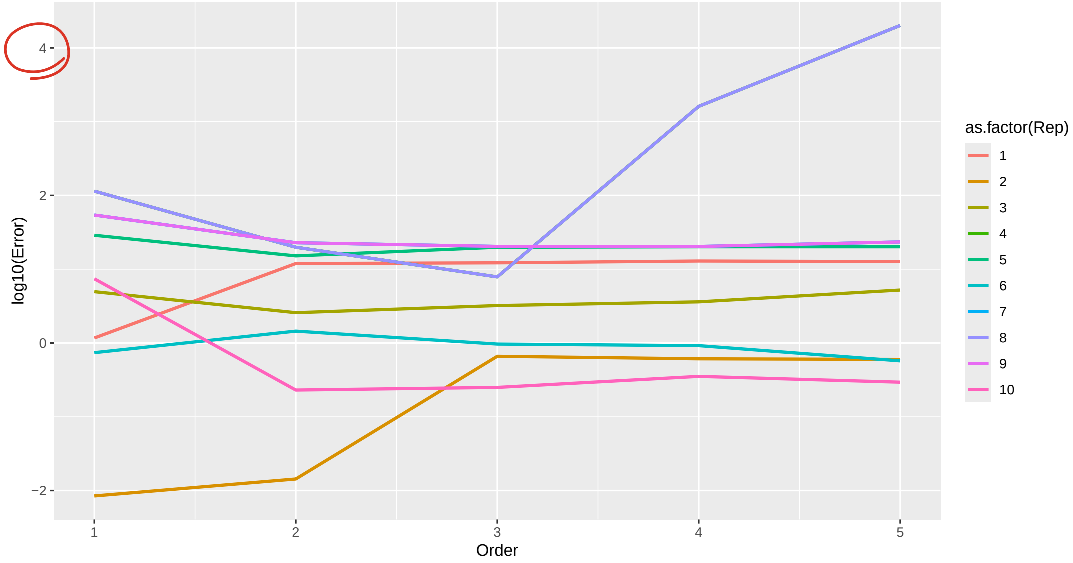


Reducing the variance: Leave-one-out cross-validation IV

But once again, that's just one run. Let's display several of them in one plot.

```
pp <- ggplot(resids, aes(x=Order, y=log10(Error), color=as.factor(Rep)));
pp <- pp + geom_line( size=1)
pp
```

Reducing the variance: Leave-one-out cross-validation V



Reducing the variance: Leave-one-out cross-validation VI

For each of our replicates, we are estimating our model based on $n - 1$ of the observations, and then evaluating our prediction on the one held-out observation.

But now we have a different kind of variance: our estimate of the error is at the mercy of the one observation that we chose to hold out. If we chose an especially "bad" or "challenging" observation to hold out, then our error might be especially high.

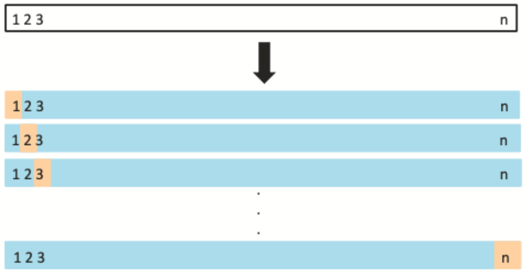
Leave-one-out cross-validation (LOO-CV) tries to bridge this gap (i.e., balancing the better stability of leaving one observation out with the variability induced by evaluating on a single point) by:

For each $i = 1, 2, \ldots, n$:

1. Train the model on $\{(X_j, Y_j) : i \neq i\}$.
2. Evaluate on (X_i, Y_i).
3. Average the model error over all $i = 1, 2, \ldots, n$.

Reducing the variance: Leave-one-out cross-validation VII

This illustration from ISLR should give you the general idea.



Let's see that in action. As we have done many times this semester, this code is optimized for clarity and readability, not for concision or "cleverness". There are much more "graceful" ways of doing this, and shortly we'll see R's built-in CV tools, which are what we would normally use for this. But here the goal is to illustrate the core ideas in a really obvious way, hence the "clumsy" code.

Reducing the variance: Leave-one-out cross-validation VIII

```r
data('mtcars'); # Still using mtcars data; reloading it just to remind us.

nrows <- nrow(mtcars); # Number of observations in the data
norder <- 5;
# For each choice of observation to hold out, we need to record the score
# (i.e., squared erro) for each of the five model orders.
errors <- data.frame( 'Row'=rep(1:nrows, each=norder),
                     'Order'=rep(1:norder, times=nrows),
                                 'Error'=rep(NA, nrows*norder));
```

Reducing the variance: Leave-one-out cross-validation IX

```r
for ( i in 1:nrow(mtcars) ) {
  train_data <- mtcars[-c(i),]; # Leave out the i-th observation
  leftout <- mtcars[c(i),]; # the row containing the left-out sample.

  # Fit the linear model, then evaluate.
  m1 <- lm(mpg ~ 1 + hp, train_data ); m1.pred <- predict( m1, leftout );
  idx <- (errors$Row==i & errors$Order==1);
  errors[idx,]$Error <- (m1.pred - leftout$mpg)^2 ;

  # Fit the quadratic model, then evaluate.
  m2 <- lm(mpg ~ 1 + hp + I(hp^2), train_data ); m2.pred <- predict( m2, leftout );
  idx <- (errors$Row==i & errors$Order==2);
  errors[idx,]$Error <- (m2.pred - leftout$mpg)^2;
}
```

Reducing the variance: Leave-one-out cross-validation X

```r
for ( i in 1:nrow(mtcars) ) {
  # Fit the cubic model, then evaluate.
  m3 <- lm(mpg ~ 1 + hp + I(hp^2) + I(hp^3), train_data );
  m3.pred <- predict( m3, leftout );
  idx <- (errors$Row==i & errors$Order==3);
  errors[idx,]$Error <- (m3.pred - leftout$mpg)^2;

  # Fit the 4-th order model, then evaluate.
  m4 <- lm(mpg ~ 1 + hp + I(hp^2) + I(hp^3) + I(hp^4), train_data );
  m4.pred <- predict( m4, leftout );
  idx <- (errors$Row==i & errors$Order==4);
  errors[idx,]$Error <- (m4.pred - leftout$mpg)^2;

  # Fit the 5-th order model, then evaluate.
  m5 <- lm(mpg ~ 1 + hp + I(hp^2) + I(hp^3) + I(hp^4) + I(hp^5), train_data );
  m5.pred <- predict( m5, leftout );
  idx <- (errors$Row==i & errors$Order==5);
  errors[idx,]$Error <- (m5.pred - leftout$mpg)^2;
}
```

Reducing the variance: Leave-one-out cross-validation XI

Okay, so let's make sure that we understand what is going on, here.

The data frame `errors` now has `nrows*norders` rows. So for each observation in the cars data set, there are five entries in the table `errors`, recording the squared error for the models of order 1, 2, 3, 4 and 5 when that data point was held out.

We said that when we do CV, we want to average across the n observations, so let's do that. We're going to use the `aggregate` function, which is one of the ways to perform "group-by" operations in R.

Group-by operations are where we pool our observations into subsets according to some criterion, and then compute a summary statistic over all of the observations in the same subset (i.e., the same "group").

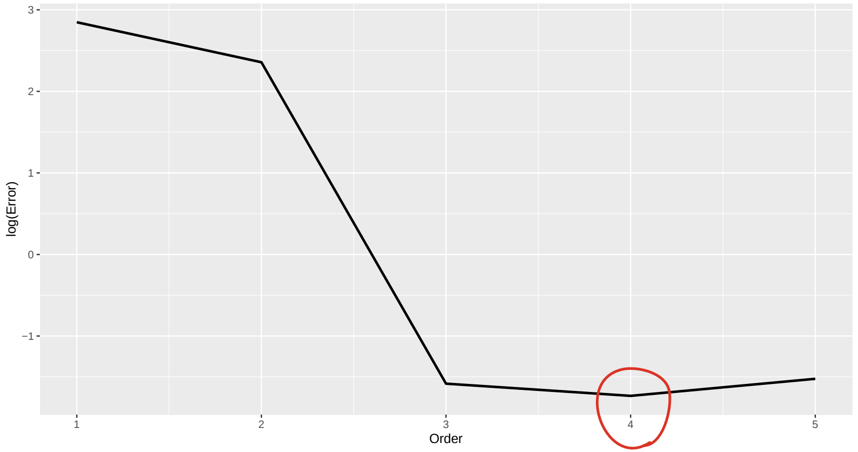
Using that language, we want to group the rows of `errors` according to model order, and take the average squared error within each order.

Reducing the variance: Leave-one-out cross-validation XII

```
# Error ~ Order tells R to group the data according to the Order column
# and that we want to summarize the Error column within observations
# of the same Order.
# Passing the FUN=mean argument tells R that the summary statistic we want to use
# is the function mean().
# We could pass other summary statistic functions in this argument.
# For example, we could use median, sd, var, max, etc.,
# though those would be a bit silly here.
err_agg <- aggregate(Error ~ Order, data=errors, FUN=mean);

head(err_agg)
##   Order      Error
## 1     1 17.2533029
## 2     2 10.5614300
## 3     3  0.2051574
## 4     4  0.1766283
## 5     5  0.2177162
```

Reducing the variance: Leave-one-out cross-validation XIII

And we can plot that just to drive the point home.



Recap: single split versus LOO-CV I

So far we have seen two different ways of estimating a model's performance on unseen data.

The first was to randomly split the data into two sets, train on one and evaluate on the other.

Pro: Only have to fit a model once (or just a few times, if we are going to repeat the operation and average)

Con: Only have half of the data available to fit the model, which leads to less accurate prediction (and thus high variance in estimated model).

The second is **leave-one-out cross-validation**.

Pro: Use all but one observation to fit the model, so model fit is almost as good as if we had used all of the data

Con: Have to fit the model anew for each held-out data point, results in fitting the model n different times, which can be expensive.

Con: Because any two training sets overlap in all but one of their elements, our fitted models are very highly correlated with one another, so we're doing a lot of work (n model fits) to get a bunch of highly correlated measurements.

Recap: single split versus LOO-CV II

So, the natural question is: can we bridge the gap between these two extremes.

The happy medium: K-fold cross validation I

Well, there are a few different ways to bridge this gap, for example using Monte Carlo methods. Let's discuss the most popular one here.

We'll borrow a bit from the LOO-CV idea, while lessening the correlatedness of the models fits.

K-fold CV randomly divides the data into K subsets, called *folds*. Then, one at a time, we hold out one of the folds, train our model on the $K-1$ remaining folds, and evaluate our model's prediction error on the held-out fold. Then, we can average the errors across the K folds.

That is, the "recipe" for K-fold cross-validation is

$2 \le K \le n$

1. Randomly partition the data into K (approximately) same-sized subsets, $S_1, S_2, \ldots, S_K$ such that $\cup_k S_k = \{1, 2, \ldots, n\}$ and $S_k \cap S_\ell = \emptyset$ for all $k \neq \ell$
2. For each $k = 1, 2, \ldots, K$, train a model on the observations indexed by $i \in \cup_{\ell \neq k} S_\ell$ and compute the prediction error

$$\hat{E}_k = \frac{1}{|S_k|} \sum_{i \in S_k} (\hat{y}_i - y_i)^2$$

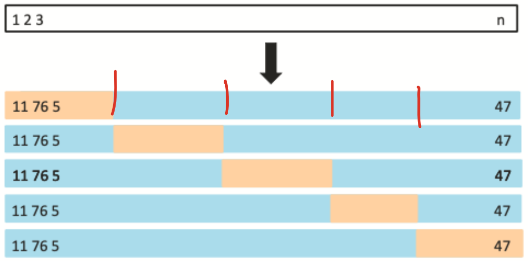
The happy medium: K-fold cross validation II

3. Estimate the true error $\mathbb{E}(\hat{y}_{n+1} - y_{n+1})^2$ as

$$\frac{1}{K}\sum_{k=1}^{K} \hat{E}_k,$$

The happy medium: K-fold cross validation III

Schematically, this looks something like this (with $K = 5$):



Let's implement this in R, just for the practice. Once again, R has built-in tools for making this easier, which we will discuss later, but this is a good opportunity to practice our R a bit.

The happy medium: K-fold cross validation IV

```
data('mtcars'); # We'll continue to use the mtcars data set
K <- 5; # 5-fold regularization. K between 5 and 10 is a fairly standard choice

# The first thing we need to do is partition the data into K folds. There are many different # ways to
# But here's an approach using the R function split()
n <- nrow(mtcars);
# sample(n,n,replace=FALSE) really just randomly permutes the data.
# Then, passing that into the split function assigns these to the K different factors
# defined by as.factor(1:K).
Kfolds <- split( sample(1:n, n,replace=FALSE), as.factor(1:K));
## Warning in split.default(sample(1:n, n, replace = FALSE), as.factor(1:K)): data
## length is not a multiple of split variable
# Note that this will throw a warning in the event that K does not divide n evenly.
```

The happy medium: K-fold cross validation V

```
Kfolds
## $`1`
## [1] 11 17 21 23 18 31  1
##
## $`2`
## [1] 32 25 12 14 28  8 13
##
## $`3`
## [1] 26 19  5 22 20  2
##
## $`4`
## [1]  9  3  7 15  6 27
##
## $`5`
## [1] 29  4 10 24 30 16
```

Now, for each of these $K = 5$ folds, we'll set it aside, train on the remaining data, and evaluate on the fold.

The happy medium: K-fold cross validation VI

```r
# The file mtcars_Kfold.R defines a function that trains the five different-order
# models and evaluates each one according to the given holdout set.
# It largely repeats the structure of the LOO-CV code implemented above,
# hence why it is relegated to a file for your later perusal.
source('mtcars_Kfold.R');

# Set up a data frame to hold our residuals.
norder <- 5;
Kfold_resids <- data.frame( 'Order'=rep(1:norder, each=K),
                            'Fold'=rep(1:K, norder ),
                            'Error'=rep(NA, K*norder) );

for (k in 1:K ) {
  heldout_idxs <- Kfolds[[k]]; # The indices of the k-th hold-out set.

  # Now train the 5 different models and store their residuals.
  idx <- (Kfold_resids$Fold==k);
  Kfold_resids[idx, ]$Error <- mtcars_fit_models( heldout_idxs );

}
```

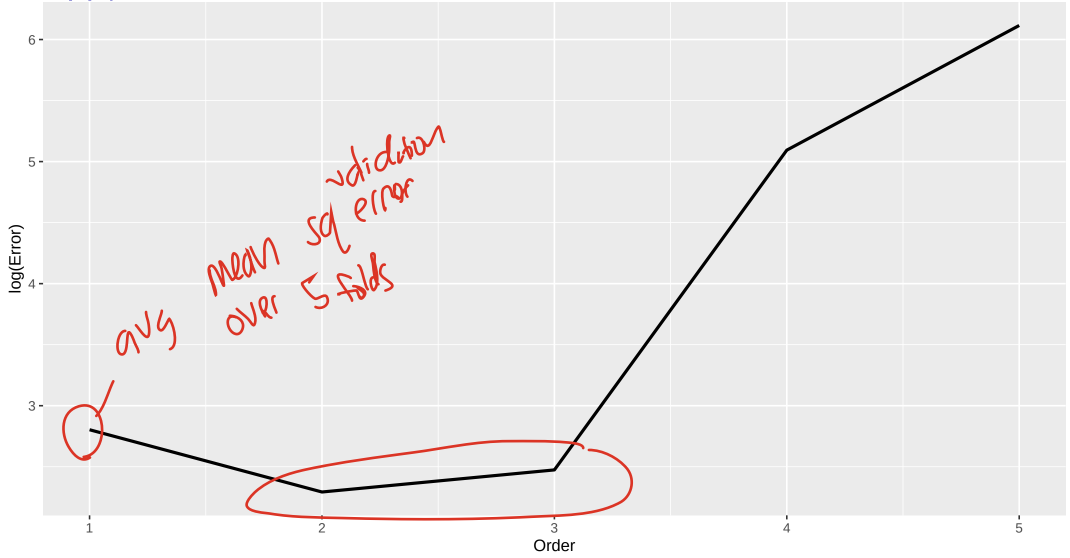
The happy medium: K-fold cross validation VII

```
head(Kfold_resids)
##   Order Fold     Error
## 1     1    1 27.149674
## 2     1    2 11.317275
## 3     1    3 23.811126
## 4     1    4 11.717761
## 5     1    5  8.485381
## 6     2    1  6.423633
```

Now, we need to aggregate over the $K = 5$ folds, and then we can plot the errors. Once again, we need to use a log scale for the errors, because the higher-order models cause some really bad prediction errors on a handful of "bad" examples.

```
KF_agg <- aggregate(Error ~ Order, data=Kfold_resids, FUN=mean);

pp <- ggplot(KF_agg, aes(x=Order, y=log(Error) ) );
pp <- pp + geom_line( size=1)
pp
```

The happy medium: K-fold cross validation VIII



The happy medium: K-fold cross validation IX

Once again, the order-2 model, `mpg ~ 1 + hp + hp^2`, does best (usually, anyway–occasionally the order-3 model is slightly better due to randomness on this small data set).

Aside: the bias-variance decomposition I

Suppose that we have a quantity θ that we want to estimate, and we have an estimator $\hat{\theta}$, the *mean squared error* is defined as

$$\text{MSE}(\hat{\theta}, \theta) = \mathbb{E}\left(\hat{\theta} - \theta\right)^2 .$$

For example, in our CV examples above, we wanted to estimate the squared error on a previously unseen data point, $\mathbb{E}(\hat{Y}_{n+1} - Y_{n+1})^2$. Note that even though this looks kind of like MSE, it is *not*. This quantity is θ in our MSE expression above. It is a thing we want to estimate. Our love of squared errors has caused us to have a whole mess of colliding notation. Such is life.

Important point: we are taking expectation here with respect to the random variable $\hat{\theta}$. Its randomness comes from the data itself (which we usually assume to depend on the true parameter θ in some way).

Aside: the bias-variance decomposition II

Now, let's expand the MSE by adding and subtracting $\mathbb{E}\hat{\theta}$ inside the square:

$$\begin{aligned}
\mathsf{MSE} &= \mathbb{E}\left(\hat{\theta} - \theta\right)^2 \\
&= \mathbb{E}\left(\hat{\theta} - \mathbb{E}\hat{\theta} + \mathbb{E}\hat{\theta} - \theta\right)^2 \\
&= \mathbb{E}\left[\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right)^2 + 2\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right) + \left(\mathbb{E}\hat{\theta} - \theta\right)^2\right] \\
&= \mathbb{E}\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right)^2 + \mathbb{E}2\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right)\left(\mathbb{E}\hat{\theta} - \theta\right) + \mathbb{E}\left(\mathbb{E}\hat{\theta} - \theta\right)^2 .
\end{aligned}$$

Now, let's notice that θ and $\mathbb{E}\hat{\theta}$ are not random, so they can get pulled out of the expectation (along with the factor of 2, which is also not random!). we can write (again, remember that the expectation is over $\hat{\theta}$, while θ is non-random)

$$\mathbb{E}2\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right)\left(\mathbb{E}\hat{\theta} - \theta\right) = 2\left(\mathbb{E}\hat{\theta} - \theta\right)\mathbb{E}\left(\hat{\theta} - \mathbb{E}\hat{\theta}\right) = 0,$$

Aside: the bias-variance decomposition III

because

$$\mathbb{E}\left(\hat{\theta}-\mathbb{E}\hat{\theta}\right)=\mathbb{E}\hat{\theta}-\mathbb{E}\hat{\theta}=0.$$

Plugging this into our equation above, we conclude that

$$\mathsf{MSE}=\mathbb{E}\left(\hat{\theta}-\mathbb{E}\hat{\theta}\right)^2+\mathbb{E}\left(\mathbb{E}\hat{\theta}-\theta\right)^2.$$

The first term on the right is just a variance– like $\mathbb{E}(X-\mathbb{E}X)^2$.

The second term on the right is the expectation of $(\mathbb{E}\hat{\theta}-\theta)^2$. But this term isn't random at all– θ is a fixed parameter, and $\mathbb{E}\hat{\theta}$ is just an expected value (i.e., not random!), so

$$\mathbb{E}\left(\mathbb{E}\hat{\theta}-\theta\right)^2=\left(\mathbb{E}\hat{\theta}-\theta\right)^2,$$

and notice that this is just the squared bias– the square of the difference between the expectation of our estimator and the thing it is supposed to estimate.

Aside: the bias-variance decomposition IV

So, to recap, we have shown that we can decompose the MSE as

$$\mathrm{MSE}(\hat{\theta}, \theta) = \mathrm{Var}\,\hat{\theta} + \mathrm{Bias}^2(\hat{\theta}, \theta).$$

In general, there will be many different estimators (i.e., many different choices of $\hat{\theta}$) that all obtain (approximately) the same MSE. The above equation means that once we are choosing among these different "similar" estimators (i.e., estimators that have similar MSE), we are really just trading off between bias and variance. That is, an estimator with smaller bias will have to "pay" for it with more variance. This is often referred to as the *bias-variance tradeoff*.

CV and the bias-variance tradeoff

Now, the purpose of cross-validation is to estimate the model error $\mathbb{E}(\hat{Y}_{n+1} - Y_{n+1})^2$. The bias-variance tradeoff says that, roughly speaking, different "reasonable" ways of estimating this quantity will all have about the same MSE, but will involve balancing bias against variance.

Bias in CV

Let's think back to the "naive" cross-validation approach, in which we split the data into two sets of similar sizes, train on one and evaluate on the other. When we do that, we train our model on a much smaller data set than if we used the full data. The result is that we (accidentally) over-estimate the error of our model, because models trained on less data simply tend to be less accurate.

That is to say, the "naive" cross-validation approach tends to yield a biased estimate of the true error of the model. Specifically, our estimate is biased upward.

On the other hand, LOOCV should be approximately unbiased as an estimate of the model error, because the difference between training on n and $n-1$ data points should not be especially large (at least once n is reasonably large).

It stands to reason that K-fold CV should sit at a kind of "happy medium" level of bias between LOOCV and "naive" CV.

Variance in CV

So LOOCV is the least biased estimate of model error, but the bias-variance trade-off predicts that we must "pay" for this in variance. It turns out that LOOCV has the most variance out of the three methods LOOCV, K-fold CV (for $K < n$) and "naive" CV.

Intuitively, the variance in LOOCV comes from the following fact: recall that for each $i = 1, 2, \dots, n$, we hold out the i-th data point and train a model on the rest.

This means that we have n different trained models, each trained on $n - 1$ data points, but each pair of training sets overlap in $n - 2$ of their data points. The result is that the trained models are highly correlated with one another. Changing just one data point in our data set doesn't change the fitted model much!

The result is that these estimated model errors are highly correlated with one another, with the result that our overall estimate of the model error has high variance.

The K models trained in K-fold CV are less correlated with one another, and hence we have (comparatively) less variance. It turns out in this case that K less-correlated error estimates have smaller correlation than n highly-correlated ones.

K-fold CV: the happy medium I

Thus, K-fold CV is a popular choice both because it is computationally cheaper than LOOCV (K model fits compared to n of them) and because it strikes a good balance between bias and variance.

3# Variable selection, for real this time

In our examples above, we concentrated on choosing among a family of linear regression models that varied in their orders, in the sense that they included as predictors all powers of the horsepower variable `hp`, up to some maximum power, to predict gas mileage. Hopefully it is clear how we could modify our approach to, say, choose which variables we do and don't include in a model (e.g., as in the Pima diabetes data set that we've seen a few times this semester).

Ultimately, our goal was to choose, from among a set of predictors that we *could* include in our model (e.g., powers of `hp`, in the case of the `mtcars` example), which predictors to actually include in the model. Again, this task is *variable selection*.

One thing that might be bugging us so far is that any way we slice it, cross-validation doesn't use all of the available data: we are always holding *something* out of our fitted model for the sake of estimating our error on unseen data.

K-fold CV: the happy medium II

Let's look at a few different approaches to variable selection that do not rely on cross-validation. These alternative methods have the advantage of not trying to estimate the unknown model error on unseen data. On the other hand, these methods can be more computationally intensive and tend to come with fewer theoretical guarantees.

This is not to suggest, however, that these methods are at odds with cross-validation. In actual research papers and in industry applications, you'll often see both CV and some of the methods presented below used in tandem to select the best model for the job.

Setup: linear regression and fitting I

Let's continue to focus on linear regression, bearing in mind that the ideas introduced here apply equally well to other regression and prediction methods (e.g., logistic regression). Let's recall that multiple linear regression models a response $Y \in \mathbb{R}$ as a linear (again, technically affine– linear plus an intercept!) function of a set of p predictors plus normal noise:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon.$$

Here, ϵ is mean-zero normal with unknown variance $\sigma^2 > 0$, and the variables $X_1, X_2, \ldots, X_p$ are the *predictors*. We often refer to p, the number of predictors, as the *dimension* of the problem, because the data (well, the vector of predictors, anyway), lies in p-dimensional space. Collecting the coefficients into a vector $(\beta_0, \beta_1, \ldots, \beta_p) \in \mathbb{R}^{p+1}$ and creating a vector $X = (1, X_1, X_2, \ldots, X_p) \in \mathbb{R}^p$, we can write this more succinctly as (if you have not taken linear algebra, you can safely ignore this, we're just including it because it's a common notation)

$$Y = \beta^T X + \epsilon.$$

Setup: linear regression and fitting II

In multiple linear regression, we observe a collection of predictor-response pairs (X_i, Y_i) for $i = 1, 2, \ldots, n$, with

$$X_i = (1, X_{i,1}, X_{i,2}, \ldots, X_{i,p}) \in \mathbb{R}^{p+1}.$$

Note that here we are including the intercept term 1 in the vector of predictors for ease of notation. This is a common notational choice, so we're including it here to get you used to seeing this. Of course, this is not universal– it's one of those conventions that you have to be careful of and check what you are reading.

Recap: variable selection

So we have p variables (plus an intercept term), and we want to select which ones to include in our model. There are many reasons to want to do this, but let's just highlight three of them:

- If there are many "useless" variables (i.e., ones that are not good predictors of the response), then including them in the model can make our predictions less accurate. Thus, we would like to proactively identify which variables are not useful, and avoid including them in the model in the first place.
- A model with fewer variables is simpler, and we like simple models! Explaining, say, heart attack risk as a function of two or three factors is a lot easier to use than a model that uses ten or twenty factors.
- If the number of predictors p is too large (say, larger than n– a common occurrence in genomic studies, for example), our estimates of the coefficients are very unstable. Variable selection and related tools give us a way to introduce stability in the form of *regularization*, which we will talk about below.

Best subset selection I

So, we have p predictor variables available to us, and we want to choose which of them to actually include in our model.

Well, the most obvious solution is to just try all possible combinations of features, train a model using each combination, and keep the best one (measured by, say, residual sum of squares).

This would have an obvious drawback: we have already seen that we can trivially improve the RSS of our model by adding variables. So the models that include more variables would do better, even if those variables did not actually lead to better model error on unseen data.

The solution to this is to do the following:

1. For each $k = 1, 2, \ldots, p$, for every set of k different variables, fit a model and keep the model that best fits the data (measured by RSS). Call this model M_k.
2. Use CV (or some other tool like AIC or adjusted R^2, which we'll discuss below) to select among the models $M_1, M_2, \ldots, M_p$.

Best subset selection II

This is called *best subset selection*. It is implemented in R in, for example, the `leaps` library the function `regsubsets`, which gets called in more or less the same way as `lm`. See (here)[https://cran.r-project.org/web/packages/leaps/index.html] for documentation if you're interested.

There is one rather glaring problem with best subset selection, though:

Question: if there are p predictors, how many models does best subset selection fit before it makes a decision?

So once p is even moderately large, best subset selection is computationally expensive, and we need to do something a little more clever.

Stepwise selection

So best subset selection is expensive because we have to try every possible model, and then choose among the best "size-k" model for each $k = 1, 2, \dots, p$. How might we cut down on the computational expense?

Stepwise selection methods avoid exhaustively checking all 2^p possible models by starting with a particular model and adding or removing one variable at a time (i.e., in "steps").

The important part is in how we decide which predictor to add or remove from the model at a particular time.

Forward stepwise selection I

The most obvious (to me, anyway) way to avoid checking every possible model is to start with a "null" model (i.e., no predictors, just an intercept term), then repeatedly add the "best" predictor not already in the model. That is,

1. Start by fitting the "null" model, with just an intercept term. Call it M_0.
2. For each $k = 1, 2, \ldots, p$, among the $p - k$ predictors not already in the model, add the one that yields the biggest improvement in RSS. Call this model, which includes k predictors and the intercept term, M_k.
3. Use CV or some other method (e.g., an information criterion; see ISLR Section 6.1) to choose among $M_0, M_1, M_2, \ldots, M_p$.

Forward stepwise selection II

The important thing is that in Step 2 above, for each $k = 1, 2, \ldots, p$, we need to fit $p - k$ different models. Thus, in total (i.e., summing over $k = 0, 1, 2, \ldots, p$), we end up fitting

$$1 + \sum_{k=0}^{p-1}(p-k) = 1 + p^2 - \frac{(p-1)p}{2} = 1 + \frac{2p^2 - p^2 + p}{2} = 1 + \frac{p(p+1)}{2}$$

different models.

To get a sense of what a big improvement this is, when p is large, this right-hand side is approximately $p^2/2$. Compare that with 2^p, which is a MUCH larger number. For example, when $p = 10$, $2^{10} \approx 1000$, while $10^2/2 \approx 50$. When $p = 20$, $2^{20} \approx 1,000,000$ while $20^2/2 \approx 200$.

Of course, the drawback is that forward stepwise selection might “miss” the optimal model, since it does not exhaustively fit every possible model the way that best subset selection does.

Backward stepwise selection

Well, if we can do forward stepwise selection, why not go backwards?

In *backward stepwise selection*, we start with the full model (i.e., a model with all p predictors), and iteratively remove one predictor at a time, always removing the predictor that decreases RSS the least.

Just like forward stepwise regression, this decreases the number of models we have to fit from 2^p to something more like (approximately) $p^2/2$.

Cautionary note: backward selection will only work if the number of observations n is larger than p. If $n < p$, the "full" model cannot be fit, because we have an *overdetermined system of linear equations*– n equations in p unknowns, and $p > n$. This is a setting where *regularization* can help a lot (see below), but the details are best left to your regression course(s).

Hybrid approaches: the best of both worlds?

It is outside the scope of this course, but there do exist stepwise selection methods that try to combine forward and backward stepwise selection. For example, we can alternately add and remove variables as needed. This can be helpful when, for example, a predictor is useful "early" in the selection process, but becomes a less useful predictor once other predictors have been included.

Shrinkage and Regularization

The variable selection methods we just discussed involved trying out different subsets of the predictors and seeing how the model performance changed as a result.

Let's consider an alternative approach. What if instead of trying lots of different models with different numbers of predictors, we went ahead and fit a model with all p available predictors, but we modify our loss function in such a way that we will set the coefficients of "unhelpful" predictors to zero? This is usually called "shrinkage", because we shrink the coefficients toward zero. You will also often hear the term *regularization*, which is popular in machine learning, and means more or less the same thing.

Let's briefly discuss two such methods, undoubtedly two of the most important tools in the statistical toolbox: ridge regression and the LASSO.

Ridge regression I

By now you are bored to death of seeing the linear regression least squares objective, but here it is again:

$$\sum_{i=1}^{n}\left(Y_i - \beta_0 - \sum_{j=1}^{p}\beta_j X_{i,j}\right)^2$$

Here we are assuming that we have p predictors, so each (X_i, Y_i) pair has a vector of predictors $X_i = (X_{i,1}, X_{i,2}, \dots, X_{i,p}) \in \mathbb{R}^p$ and response $Y_i \in \mathbb{R}$.

Remember, we're trying to minimize this RSS by choosing the coefficients β_j, $j = 0, 1, 2, \dots, p$ in a clever way.

Ridge regression shrinks these estimated coefficients toward zero by changing the loss slightly. Instead of minimizing the RSS alone, we add a penalty term:

$$\sum_{i=1}^{n}\left(Y_i - \beta_0 - \sum_{j=1}^{p}\beta_j X_{i,j}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = \text{RSS} + \lambda\sum_{j=1}^{p}\beta_j^2$$

Ridge regression II

where $\lambda \geq 0$ is a *tuning parameter* (which we have to choose– more on that soon).

Our cost function now has two different terms:

1. Our old friend RSS, which encourages us to choose coefficients that reproduce the observed responses accurately
2. The *shrinkage penalty* $\lambda \sum_{j=1}^{p} \beta_j^2$, which encourages us to choose all our coefficients (other than β_0) equal to zero. That is, it *shrinks* our solution toward the origin.

The tuning parameter λ controls how much we care about this shrinkage penalty compared to the RSS term. When λ is big, we "pay" more for large coefficients, so we will prefer coefficients closer to zero. When $\lambda = 0$, we recover plain old least squares regression.

For each value of λ that we choose, we get a different solution to our (regularized) regression, say, $\hat{\beta}^{(\lambda)}$. In this sense, whereas least squares linear regression gives us just one solution $\hat{\beta}$, shrinkage methods give us a whole family of solutions, corresponding to different choices of λ.

For this reason, choosing the tuning parameter λ is crucial, but we will have only a little to say about this matter, owing to time constraints. Luckily, you already know a family of methods for choosing λ– cross validation is a very common approach!

Ridge regression on the `mtcars` data set I

Let's try this out on the `mtcars` data set, trying to predict `mpg` from all the of the available predictors, this time. One thing to bear in mind: the data set is only 32 observations, so our fits are going to be a little unstable (but this is precisely why we use regularization!).

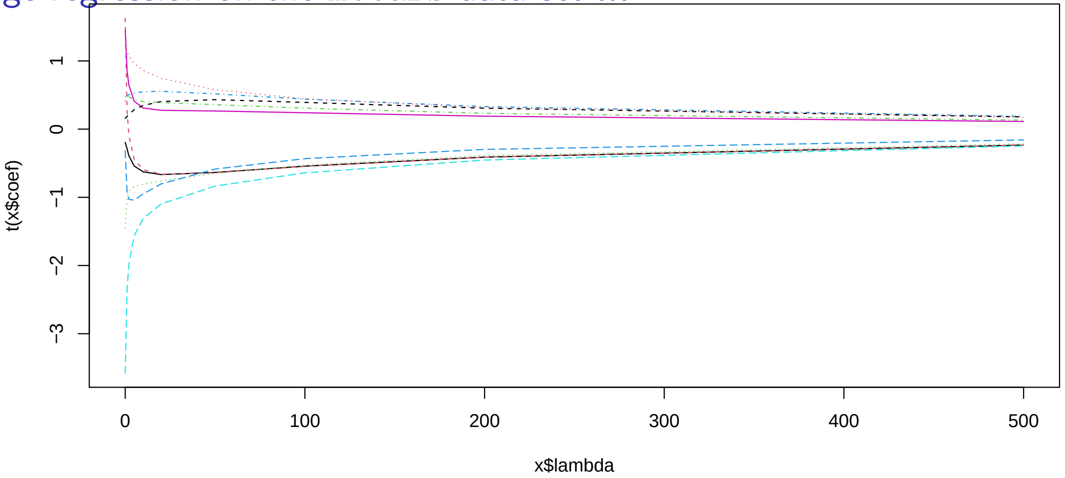
```
names(mtcars);
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Ridge regression is available in the `MASS` library in R.

Ridge regression on the `mtcars` data set II

```
library(MASS);
## Warning: package 'MASS' was built under R version 4.2.3

lambda_vals <- c(0,1,2,5,10,20,50,100,200,500); # Choose lambdas to try.
# lm.ridge needs:
# 1) a model (mpg~. says to model mpg as an intercept
#         plus a coefficient for every other variable in the data frame)
# 2) a data set (mtcars, of course)
# 3) a value for lambda. lambda=0 is the default,
#         and recovers classic linear regression.
#         But we can also pass a whole vector of lambdas, like we are about to do,
#         and lm.ridge will fit a separate model for each.
# See ?lm.ridge for details.
ridge_models <- lm.ridge(mpg~., mtcars, lambda=lambda_vals);

# Naively plotting this object shows us how the different coefficients
# change as lambda changes.
plot( ridge_models );
```

Ridge regression on the `mtcars` data set III



Each line in the above plot represents the coefficient of one of our predictors. The x-axis is our choice of λ (`lambda` in the code) and the y-axis is the actual value of the coefficients.

Ridge regression on the `mtcars` data set IV

Actually extracting those predictor labels to make a legend for this plot is annoying, and beside the point– refer to the documentation in `?lm.ridge`). The important point is that as we change λ, the coefficients change. Generally speaking, as λ gets bigger, more coefficients are closer to zero.

Indeed, if we make λ big enough, all of the coefficients will be zero (except the intercept, because it isn't multiplied by λ in the loss). That's shrinkage!

Just as a sanity check, let's fit plain old linear regression and verify that the coefficients with $\lambda = 0$ match.

```
lm_sanity_check <- lm(mpg~., mtcars);
lm_sanity_check$coefficients
## (Intercept)         cyl        disp          hp        drat          wt
## 12.30337416 -0.11144048  0.01333524 -0.02148212  0.78711097 -3.71530393
##        qsec          vs          am        gear        carb
##  0.82104075  0.31776281  2.52022689  0.65541302 -0.19941925
```

And compare that with

Ridge regression on the `mtcars` data set V

```
head( coef( ridge_models), 1 ); # the first row is the lambda=0.0 setting.
##                         cyl        disp           hp      drat         wt      qsec
##   0 12.30337 -0.1114405 0.01333524 -0.02148212 0.787111 -3.715304 0.8210407
##              vs        am     gear        carb
##   0 0.3177628 2.520227 0.655413 -0.1994193
```

They're the same, up to several digits of precision, anyway. Good!

Shrinkage and RSS I

Now, for each value of λ, we get a different fitted model. How do these different models do in terms of their fit (as measured by RSS)?

Well, annoyingly, the object returned by `lm.ridge` does not include a `residuals` attribute the same way that the `lm` object does:

```r
mean( lm_sanity_check$residuals^2 );
## [1] 4.609201
```

More annoyingly still, the object returned by `lm.ridge` also does not include a `predict` method, so we can just call something like `predict( model, data)` the way we would with the output of `lm`:

```r
mean( (predict( lm_sanity_check, mtcars) - mtcars$mpg )^2 )
## [1] 4.609201
```

Shrinkage and RSS II

So, we have to roll our own predict/residuals computation. This is going to be a bit complicated, but it's worth the detour to get some programming practice.

Our ridge regression model has coefficients: one set of coefficients for each valu of lambda that we passed in.

```
length( lambda_vals )
## [1] 10
```

Those estimated coefficients are stored in a matrix. Each column of this matrix corresponds to a coefficient (including the intercept, the first column. Each row corresponds to one λ value.

Shrinkage and RSS III

```
coef( ridge_models )
##                      cyl         disp           hp      drat         wt
##   0 12.30337 -0.1114405  0.0133352399 -0.021482119 0.7871110 -3.7153039
##   1 16.53766 -0.1624028  0.0023330776 -0.014934856 0.9246313 -2.4611460
##   2 18.55460 -0.2212878 -0.0007347273 -0.013481440 0.9597173 -2.0619305
##   5 20.72198 -0.3079969 -0.0036206803 -0.012460649 1.0060841 -1.6219933
##  10 21.27391 -0.3563891 -0.0048700433 -0.011966312 1.0409643 -1.3618221
##  20 20.88322 -0.3792044 -0.0054324669 -0.011248991 1.0545238 -1.1389693
##  50 19.85752 -0.3614183 -0.0052401829 -0.009609992 0.9828882 -0.8673112
## 100 19.37410 -0.3092845 -0.0044742424 -0.007814946 0.8353896 -0.6656981
## 200 19.29829 -0.2337711 -0.0033702943 -0.005732613 0.6286426 -0.4706006
## 500 19.54099 -0.1331434 -0.0019134597 -0.003201881 0.3567082 -0.2559721
##           qsec        vs        am      gear        carb
##   0 0.82104075 0.3177628 2.5202269 0.6554130 -0.19941925
##   1 0.49258752 0.3746517 2.3083758 0.6857159 -0.57579125
##   2 0.36772540 0.4389536 2.1835666 0.6545252 -0.64772938
##   5 0.23220486 0.5749017 1.9622086 0.5933014 -0.65548632
##  10 0.17615173 0.7007040 1.7537869 0.5573491 -0.59737336
##  20 0.15680180 0.8158626 1.5168704 0.5362044 -0.50497523
##  50 0.15120048 0.8706103 1.1764833 0.4942313 -0.36858373
## 100 0.13669747 0.7898063 0.9064651 0.4225302 -0.27224119
## 200 0.10798351 0.6187171 0.6407444 0.3195920 -0.18719020
## 500 0.06363418 0.3611011 0.3479026 0.1819390 -0.09983672
```

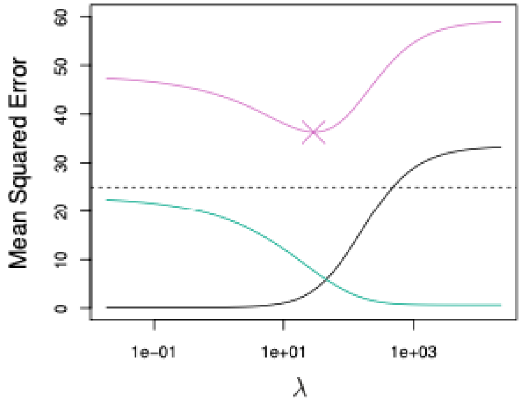
Why is ridge regression helpful? I

Well, the short answer is that ridge regression (and other shrinkage methods) prevents over-fitting. λ makes it more expensive to simply choose whatever coefficients we please, which in turn prevents us from over-fitting to the data.

In essence, this is the bias-variance tradeoff again! As λ increases, our freedom to choose the coefficients becomes more constrained, and the variance decreases (and the bias increases).

Here's an example from ISLR.

Why is ridge regression helpful? II



Notice that the variance decreases as λ increases, while squares bias decreases, but there is a "sweet spot" that minimizes the MSE. The whole point of model selection (CV, AIC, ridge regression, etc.) is to find this sweet spot (or spot close to it).

The LASSO I

Now, there's one issue with ridge regression, which becomes evident when we compare it with subset selection methods. Except when λ is truly huge (i.e., infinite), ridge regression fits a model that still has all of the coefficients in it (i.e., all of the coefficients are nonzero, though perhaps small). Said another way, we haven't simplified the model in the sense of reducing the number of predictors or only keeping the "useful" predictors around.

This isn't a problem for prediction. After all, more predictors often make prediction better, especially when we have regularization to prevent us from over-fitting.

But this *is* a problem if our goal is to simplify our model by selecting only some predictors to include in our model. One way to do this would be to make it so that coefficients that aren't "pulling their weight" in the sense of helping our prediction error will be set to zero. This is precisely the goal of the LASSO.

Note: I (Keith) usually write LASSO in all caps because it is an acronym for *least absolute shrinkage and selection operator*, but it is so wildly popular in statistics that we will often just write Lasso or lasso to refer to it. For those not familiar, a *lasso* is one of these.

The LASSO II

The LASSO looks a lot like ridge regression, except that the penalty is slightly different:

$$\sum_{i=1}^{n}\left(Y_i - \sum_{j=0}^{p}\beta_j X_{i,j}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = \text{RSS} + \lambda\sum_{j=1}^{p}|\beta_j|$$

The penalty term now involves a sum of absolute values of the coefficients instead of a sum of squares.

The interesting thing is that this small change has a big effect on what our estimated coefficients look like. The LASSO penalty encourages coefficients to be set *precisely* equal to zero if they aren't useful predictors (i.e., if they do not help to decrease the RSS).

There is an interesting geometric reason for this, though it is outside the scope of the course. See the end of Section 6.6.2 in ISLR.

The important point is that the LASSO performs *variable selection* for us by setting many coefficients to zero.

The LASSO III

The `glmnet` package has a very good LASSO implementation. This is generally a very useful package for doing all kinds of different penalized regression, and you'll likely use it extensively in your regression course(s). You'll get a bit of practice with it in discussion section.

How to choose λ? CV to the rescue!

A natural question in both ridge and the LASSO concerns how we should choose the term λ that controls the "strength" of the penalty term.

We said a few lectures ago that CV was useful beyond just variable selection, and here's the payoff.

CV is also a great way to choose λ in these kinds of *penalized* problems. We choose different values of λ, fit the corresponding models, and use CV to select among them!

Section 6.2.3 has a more detailed discussion of this, using 10-fold CV to compare different choices of λ.

Review

In these notes we covered

- The concept of over-fitting to training data
- Training / Validation data splits
- Single Split Validation
- Leave One Out Cross Validation
- K-Fold CV
- Bias - Variance Tradeoff
- Bias & Variance of CV Methods
- Bias & Variance of Model Error by Model Complexity
- Model Selection - Best Subset, Forward Stepwise, Backwards Stepwise
- Model Comparison Statistics (Adjusted R^2, AIC, BIC)
- Ridge Regression - shrinkage of coefficients
- LASSO regression - shrinkage and variable selection