STAT 213: Overfitting and Cross-Validation

Colin Reimer Dawson

Last Revised April 2, 2018

The purpose of this activity is two-fold: (1) for you to get a conceptual understanding and a feel for the phenomenon of overfitting and the tool of cross-validation, and (2) for you to learn how to do cross-validation in R.

I suggest creating a Markdown document or at least a script to hold the code for this activity.

Generating the Data

First use `set.seed()` with a "unique" integer of your choice (T numbers work well) so that you can get the same results every time you re-run or re-Knit your code, but so that you get different results from other groups.

As we did in the last activity which examined what happens to R^2 as we add either useful or useless predictors, we will again use a synthetic dataset generated based on a known "population" model. This time we will consider various polynomials.

The data has just one predictor, X, but the true relationship between X and the mean of Y is a third-degree (cubic) polynomial, given by the following formula:

$$f(X) = 1 - \frac{1}{2}X - \frac{1}{2}X^2 + \frac{1}{4}X^3$$

The Y values have been generated by adding zero-mean Normally distributed residuals $f(X)$, with a standard deviation of 4.

Run the following scripts from my website (after setting your random seed). The first one defines some useful helper functions to simplify our code later. The second one generates the data and plots it with the population model.

```
set.seed(00029747)
source("~/213site/code/helper_functions.R")
source("~/213site/code/gen_poly_data.R")
# source("http://colindawson.net/stat213/code/helper_functions.R")
# source("http://colindawson.net/stat213/code/gen_poly_data.R")
```

Fitting some Polynomial Models

Similar to what we did in class, let's fit a series of ten regression models to this data, this time using polynomial prediction functions, ranging from order 1 (linear) to order $n - 1$ (remember that the last model will have 0 error df).

The easiest thing to do is probably to use the `poly()` function for this so you can just change the degree for each new model.

For example, the model for degree 2 is

```
model2 <- lm(y ~ poly(x, degree = 2, raw = TRUE), data = FakePolyData)
```

To Do: Fit each model (degree 1 to degree n-2) and plot each model over the data. I've defined a helper function in the script you ran above to streamline things for you:

```
model2 <- fit.polynomial.model(
    degree = 2, linear.formula = y ~ x, data = FakePolyData)
plot.data.with.polynomial.model(model2, data = FakePolyData)
```

If you want, you can streamline the process even more using an R function called `lapply()`, which works as follows.

```
## lapply() takes a list of arguments and a function name and
## calls the function several times, setting the first argument to
## each entry in the list supplied, and holding the rest constant
models <- lapply(
    1:10, fit.polynomial.model,
    linear.formula = y ~ x, data = FakePolyData)
par(mfrow = c(3,4))  # sets up a 3 x 4 grid for plots
lapply(models, plot.data.with.polynomial.model, data = FakePolyData)
```

Checking R^2 Values

Notice that, as we did with our adjusted R^2 activity, for the the first three models, we are adding predictors that are actually related to the response, according to our known population model. For the terms of degree greater than 3, there is no relationship between the predictors we are adding and the response; although the curve is fitting the data better it is really just fitting the "noise". That is, it's **overfitting** the data.

To-Do: For your nine models, find the R^2 and adjusted R^2 values, and plot them as a function of the number of predictors. Assuming you used the `lapply()` trick above and now have a list of ten models, we can use the same trick again (along with another helper function) to extract the fit statistics from each one.

```
## applies the rsquared() function to each entry in `models`
## and collects the results in a variable
rsquared.values <- sapply(models, rsquared)
## applies the adj.rsquared() function
## This doesn't exist in R by default; I defined it
## in helper_functions.R
adj.rsquared.values <- sapply(models, adj.rsquared)
## Plots degrees on the x axis against r-squareds
## on the y axis.  type = "b" plots "b"oth points
## and connecting lines
plot(1:10, rsquared.values, type = "b")
## Adds adjusted r-squared with dashed lines (lty = 2)
## and filled circles (pch = 16)
points(1:10, adj.rsquared.values, type = "b", lty = 2, pch = 16)
legend("topleft", legend = c("Unadjusted", "Adjusted"),
       lty = c(1,2), pch = c(1,16))
```

Using Held Out Data

So far we are evaluating the fit of each model on the full dataset; that is, we are using the same data to *fit* and to *evaluate* the models.

Let's take a different approach, and choose a random subset of the data as a "training set" to use when fitting the models, "holding out" the remaining points to serve as independent validation of the quality of the model. Instead of evaluating the fit to the training set, we'll evaluate its predictions on the held out set. The following code does this step-by-step so you can understand the logic, since this is the "meat" of the activity, so don't just copy it blindly; try to understand what each line is doing.

In practice we can use prewritten functions for this.

```
n <- nrow(FakePolyData)
## use all but one for training
to.hold.out <- rep(0:1, times = c(n - 1, 1)) %>%
    sample()  # this randomizes where the 1s go
ValidationSet <- filter(FakePolyData, to.hold.out == 1)
TrainSet <- setdiff(FakePolyData, ValidationSet)

### Now fit a polynomial model on the training set only
model2.train <- lm(y ~ poly(x, degree = 2, raw = TRUE), data = TrainSet)
```

We can plot the prediction function over the full dataset, highlighting the held out point:

```
## Need mosaic for this
library(mosaic)
plot(y ~ x, data = TrainSet, pch = 16, col = 1)
points(y ~ x, data = ValidationSet, pch = 1, col = 2)
f.hat <- makeFun(model2.train)
curve(f.hat(x), add = TRUE)
```

Now we'll compute the squared prediction error for the held out point.

```
predictions.vs <- predict(model2.train, newdata = ValidationSet)
mse.test <- with(ValidationSet, mean((y - predictions.vs)^2))
mse.test
```

Compare this to the mean squared prediction error on the training set itself:

```
predictions.ts <- predict(model2.train, newdata = TrainSet)
mse.train <- with(TrainSet, mean((y - predictions.ts)^2))
mse.train
```

Most likely the (mean) error on *new* data is higher than the mean error on the data we used to estimate the coefficients. This makes sense; evaluating on the training set is like giving an exam where all the questions were on the practice test: Sure you might do well in part because you learned the material, but you might also do well in part because you memorized the ins and outs of those problems. It would not be a surprise if the grades were unrealistically good.

Let's check out polynomials of degree 1 to 9 (we can't fit degree 10, since that would have 11 coefficients and the training set only has 10 data points).

```
## I've provided two more
## helper functions: one to create the plot, and another to
## compute the (mean) squared prediction error:
plot.fit.with.test(model2.train, plot.formula = y ~ x,
                   train.set = TrainSet, test.set = ValidationSet)
get.held.out.error(
    model2.train,
    test.set = ValidationSet,
    response.name = "y",
    error.metric = mse)
get.held.out.error(
    model2.train,
    test.set = TrainSet,
    response.name = "y",
    error.metric = mse)
```

We can use these with some `lapply` constructions for efficiency:

```
## First the plots
models <- lapply(1:9, fit.polynomial.model,
                 linear.formula = y ~ x, data = TrainSet)
par(mfrow = c(3,4))
lapply(models, plot.fit.with.test, plot.formula = y ~ x,
       train.set = TrainSet, test.set = ValidationSet)
```

```
## Then the MSE computations
validation.mse.list <-
    lapply(models, get.held.out.error,
           test.set = ValidationSet,
           response.name = "y",
           error.metric = mse
           )
validation.mse.list
train.mse.list <-
    lapply(models, get.held.out.error,
           test.set = TrainSet,
           response.name = "y",
           error.metric = mse)
```

Likely you see that the *generalization* error, unlike the training set error, is better for some simpler models than for the most complex models.

Swapping Held-Out Points

We picked a single random point to "hold out", so our results will be sensitive to which point that is. We might expect more stable results if, instead of letting one point dictate our scores, we let every point have a turn to be the "held out" point; fitting the model each time to the remaining points and evaluating on that one, then averaging the results.

This method is called **leave-one-out cross validation**: each time we fit the model, we leave one point out to validate that model, and we do this "back and forth" (that's the "cross" part).

More generally, we can perform K**-fold cross validation**, where we randomly split the dataset into K roughly equal subsets, or "folds", and let one subset at a time serve as the validation set, fitting the model on the other $K-1$ "folds". Leave-one-out cross-validation is equivalent to using N folds. Two-fold and ten-fold cross-validation are other popular choices.

Here's the algorithm:

- Choose an appropriate measure of prediction performance (e.g., mean squared error)

- Split the data randomly into K roughly equal subsets (folds)
- Iterating over models under consideration, m from 1 to M:
  - Iterating over folds, k from 1 to K:
    - Set aside fold k as the validation set
    - Fit model m on the other $K - 1$ folds
    - Evaluate the performance of the model on fold k
  - Evaluate model m's average performance over the K folds
- Prefer the model(s) with the best average generalization performance

Let's see how two-fold cross-validation does with our fake data. The following code is very step-by-step, so the algorithm is clear. In practice we can shorten this.

```
## Splitting the data (setting seed for reproducibility)
set.seed(00029747)
K <- 2
N <- nrow(FakePolyData)
fold.labels <- rep(1:K, length = N) %>% sample()
Fold1 <- filter(FakePolyData, fold.labels == 1)
Fold2 <- filter(FakePolyData, fold.labels == 2)
```

```
## Here's a loop over folds for model 1
m1.without.f2 <- lm(y ~ poly(x, degree = 1, raw = TRUE),
                    data = setdiff(FakePolyData, Fold2))
m1.fold2.error <-
    get.held.out.error(
        m1.without.f2,
        test.set = Fold2,
        response.name = "y",
        error.metric = mse)
m1.without.f1 <- lm(y ~ poly(x, degree = 1, raw = TRUE),
                    data = setdiff(FakePolyData, Fold1))
m1.fold1.error <-
    get.held.out.error(
        m1.without.f1,
        test.set = Fold1,
        response.name = "y",
```

```
        error.metric = mse)
m1.error <- mean(c(m1.fold1.error, m1.fold2.error))
m1.error

## Here's a loop over folds for model 2
m2.without.f2 <- lm(y ~ poly(x, degree = 2, raw = TRUE),
                    data = setdiff(FakePolyData, Fold2))
m2.fold2.error <-
    get.held.out.error(
        m2.without.f2,
        test.set = Fold2,
        response.name = "y",
        error.metric = mse)
m2.without.f1 <- lm(y ~ poly(x, degree = 2, raw = TRUE),
                    data = setdiff(FakePolyData, Fold1))
m2.fold1.error <-
    get.held.out.error(
        m2.without.f1,
        test.set = Fold1,
        response.name = "y",
        error.metric = mse)
m2.error <- c(m2.fold1.error, m2.fold2.error) %>% mean()
m2.error
```

We could continue copying and pasting, but I've provided a function to do this more quickly.

```
set.seed(00029748)
do.k.fold.cv(model2, data = FakePolyData, num.folds = 2, error.metric = mse)
```

Iterating over models:

```
## Note that we are limited to degree 4 since one fold has only
## 5 datapoints
models <- lapply(1:4, fit.polynomial.model,
                 linear.formula = y ~ x, data = FakePolyData)
lapply(models, do.k.fold.cv,
       data = FakePolyData,
       num.folds = 2,
```

```
       error.metric = mse)
```

We can get even more reliable results if we repeat the process several times with different random splits of the data.

```
models <- lapply(1:4, fit.polynomial.model,
                 linear.formula = y ~ x, data = FakePolyData)
lapply(models, do.k.fold.cv,
       data = FakePolyData,
       num.folds = 2,
       error.metric = mse,
       iterations = 100)
```

What about leave-one-out cross-validation?

```
models <- lapply(1:(N-2), fit.polynomial.model,
                 linear.formula = y ~ x, data = FakePolyData)
lapply(models, do.k.fold.cv,
       data = FakePolyData,
       num.folds = N,
       error.metric = mse)
```

Which model does each algorithm suggest is best? How do these results compare to using adjusted R^2?