Lab 1: Cross Validation

PSTAT131-231

Week 1

Objectives

In this lab, you'll practice:

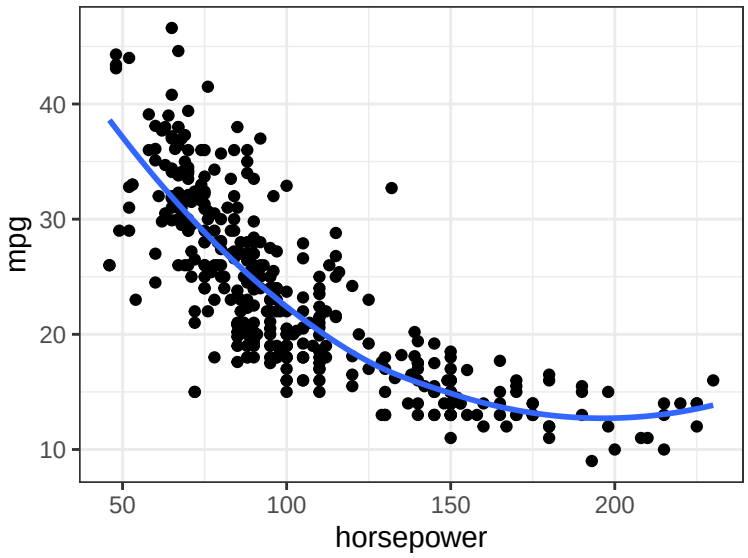
- Create training-test, and cross-validation partitions using `modelr` commands
- Fit linear, polynomial, and spline models in R
- Estimate prediction errors for regression models
- Implement efficient cross validation using `purrr` and `modelr`

Exploration of Auto data

You will use the `auto` dataset from ISLR to work through an example of model building and cross validation. First load the data. It comprises measurements related to mileage taken on 392 autos.

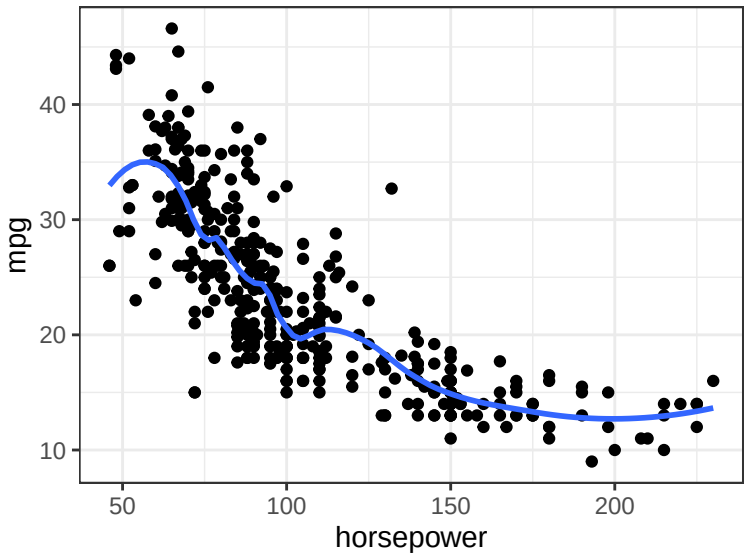
```
# select data columns
auto <- as_tibble(Auto)
```

Consider first regressing miles per gallon on horsepower.



Caution about smooth trend lines: The trend line above is known as a LOESS curve, and is a local smoothing of the data scatter. There is a 'bandwidth' parameter that controls how large or small a neighborhood is used in smoothing. This can have a big effect on perceptions of the trend. Below is a more local

smoothing; the only difference from the previous plot is the trend line (the data are the same). Consider (no need to answer in writing): do you perceive a different pattern in this plot than in the previous?



Fitting regression models

The trend looks nonlinear, but we'll start with a linear model to illustrate fitting. Let's first randomly partition the data into 80% training and 20% test sets. The `resample_partition` function in the `modelr` library does this efficiently:

```
# partition
set.seed(40121)
auto_part <- resample_partition(data = auto, p = c(test = 0.2, train = 0.8))

# inspect
auto_part
```

```
## $test
## <resample [78 x 9]> 1, 3, 4, 5, 12, 18, 26, 29, 40, 43, ...
##
## $train
## <resample [314 x 9]> 2, 6, 7, 8, 9, 10, 11, 13, 14, 15, ...
```

The outputs are `resample` objects. Each resample is a list containing a copy of the `auto` data (a tibble) and an index set giving the row indices included in that partition. Let's look at one:

```
# inspect one partition
auto_part$train %>% str()
```

```
## List of 2
##  $ data: tibble [392 x 9] (S3: tbl_df/tbl/data.frame)
##   ..$ mpg          : num [1:392] 18 15 18 16 17 15 14 14 14 15 ...
##   ..$ cylinders    : num [1:392] 8 8 8 8 8 8 8 8 8 8 ...
```

```
##   ..$ displacement: num [1:392] 307 350 318 304 302 429 454 440 455 390 ...
##   ..$ horsepower  : num [1:392] 130 165 150 150 140 198 220 215 225 190 ...
##   ..$ weight      : num [1:392] 3504 3693 3436 3433 3449 ...
##   ..$ acceleration: num [1:392] 12 11.5 11 12 10.5 10 9 8.5 10 8.5 ...
##   ..$ year        : num [1:392] 70 70 70 70 70 70 70 70 70 70 ...
##   ..$ origin      : num [1:392] 1 1 1 1 1 1 1 1 1 1 ...
##   ..$ name        : Factor w/ 304 levels "amc ambassador brougham",..: 49 36 231 14 161 141 54 223 24
##  $ idx : int [1:314] 2 6 7 8 9 10 11 13 14 15 ...
##  - attr(*, "class")= chr "resample"
```

Notice that the dimensions of the `$data` element of the list are the same as the overall `auto` data: it is an exact copy. Many model fitting functions in R know how to handle resample objects 'under the hood' by combining the index indicating which rows to include with the data copy to fit only to the appropriate subset of rows, but if you ever need to recover a resample object as an explicit data set, coerce to a data frame:

```
# coerce a resample to a data frame
auto_part$train %>%
  as.data.frame() %>%
  str()
```

```
## tibble [314 x 9] (S3: tbl_df/tbl/data.frame)
##  $ mpg         : num [1:314] 15 15 14 14 14 15 15 15 14 24 ...
##  $ cylinders   : num [1:314] 8 8 8 8 8 8 8 8 8 4 ...
##  $ displacement: num [1:314] 350 429 454 440 455 390 383 400 455 113 ...
##  $ horsepower  : num [1:314] 165 198 220 215 225 190 170 150 225 95 ...
##  $ weight      : num [1:314] 3693 4341 4354 4312 4425 ...
##  $ acceleration: num [1:314] 11.5 10 9 8.5 10 8.5 10 9.5 10 15 ...
##  $ year        : num [1:314] 70 70 70 70 70 70 70 70 70 70 ...
##  $ origin      : num [1:314] 1 1 1 1 1 1 1 1 1 3 ...
##  $ name        : Factor w/ 304 levels "amc ambassador brougham",..: 36 141 54 223 241 2 101 57 30 274
```

Notice specifically the dimensions: this data frame has $314 \approx 0.8 \times 392$ observations, compared with all 392 observations in the `auto` data; these are only the rows included in that partition, comprising roughly 80% of the dataset.

Now let's fit a linear model. `pander()` formats the summary output nicely in a table.

```
# regress mpg on horsepower
fit_lm <- lm(mpg ~ horsepower, data = auto_part$train)
summary(fit_lm) %>% pander()
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	39.79	0.7909	50.32	9.189e-152
horsepower	-0.1545	0.007067	-21.86	6.748e-65

Table 2: Fitting linear model: mpg ~ horsepower

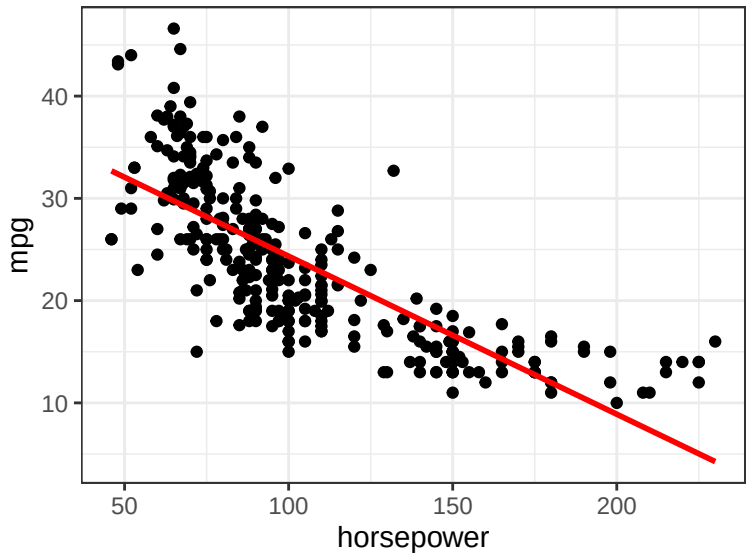
Observations	Residual Std. Error	R^2	Adjusted R^2
314	4.899	0.605	0.6037

The `modelr` package includes some useful `mutate`-like functions for adding predictions, residuals, and functions performing other computations using `lm` objects. The following is an example of adding predictions to the original dataset.

```
# store model predictions
fit_lm_df <- auto_part$train %>%
  as.data.frame() %>%
  add_predictions(fit_lm, var = 'pred')
fit_lm_df %>%
  select(mpg, horsepower, pred) %>%
  head(5) %>%
  pander()
```

mpg	horsepower	pred
15	165	14.3
15	198	9.202
14	220	5.803
14	215	6.576
14	225	5.031

By plotting a path through the predictions, we can visualize the model output.



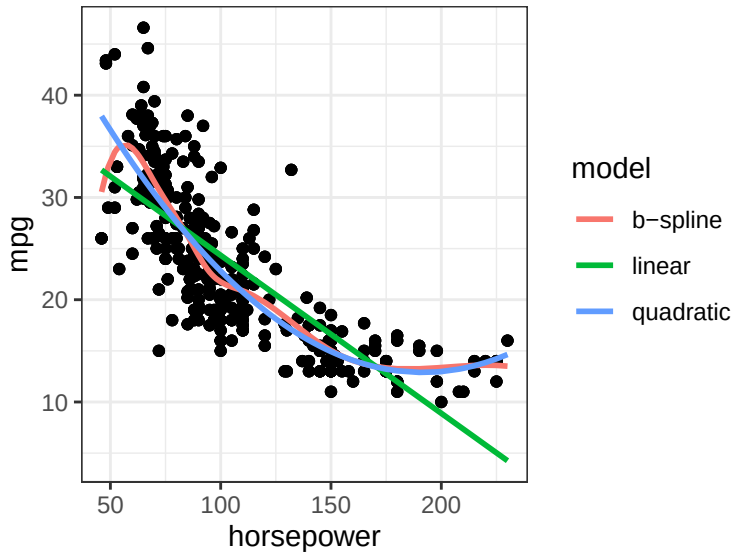
Now let's fit_lm some nonlinear models.

```
# polynomial model
fit_poly <- lm(mpg ~ poly(horsepower,
                          degree = 2,
                          raw = T),
               data = auto_part$train)

# basis expansion using b-spline bases
```

```
fit_spline <- lm(mpg ~ bs(horsepower, df = 8),
           data = auto_part$train)
```

The same approach of adding predictions and plotting paths can be used to superimpose several fit_lmted models on one plot. When you review the source code, notice the `arrange` step (`geom_path` plots a path through points in the order in which they appear).



Which do you think looks better? We can quantify fit_lm quality by computing the training MSE. This can be efficiently computed with the `mse()` function in `modelr`.

```
# store training mses
mse_train_lm <- mse(model = fit_lm, data = auto_part$train)
mse_train_poly <- mse(model = fit_poly, data = auto_part$train)
mse_train_spline <- mse(model = fit_spline, data = auto_part$train)

# print as a named vector
c(lm = mse_train_lm,
  poly = mse_train_poly,
  spline = mse_train_spline)
```

```
##       lm     poly   spline
## 23.84992 19.00195 17.76338
```

So the spline model has the best fit. But what about predictions? Recall from lecture that test set MSE is an estimate of prediction error.

Your Turn (1) Follow the example of computing training MSE and introduce appropriate modifications to compute test MSE. (Hint: all you need to do is change the data argument for each `mse` call.) Which model has the lowest estimated prediction error?

```
# store test mses
mse_test_lm <- mse(model = fit_lm,data = auto_part$test)
mse_test_poly <- mse(model = fit_poly, data = auto_part$test)
mse_test_spline <- mse(model = fit_spline, data = auto_part$test)
# print as a named vector
c(lm = mse_test_lm,
  poly = mse_test_poly,
  spline = mse_test_spline)
```

```
##       lm     poly   spline
## 24.61910 19.09420 19.62622
```

Cross validation

An alternate way to estimate prediction error for the three models is to perform cross validation. The function `modelr::crossv_kfold()` handles this nicely:

```
# randomly split data into folds
n_folds <- 4
set.seed(10521) # for reproducibility
auto_cv <- auto %>% crossv_kfold(k = n_folds, id = 'fold')
```

Have a look at the result. The `kfold_crossv()` creates k random partitions with non-overlapping test sets, and returns a tibble with 'list-columns':

```
## # A tibble: 4 x 3
##   train          test           fold
##   <named list>   <named list>   <chr>
## 1 <resample[,9]> <resample[,9]> 1
## 2 <resample[,9]> <resample[,9]> 2
## 3 <resample[,9]> <resample[,9]> 3
## 4 <resample[,9]> <resample[,9]> 4
```

The elements of each list-column are `resample` objects, organized as follows:

- each row is one partitioning (one 'fold');
- the `train` column gives the training partition;
- the `test` column gives the test partition.

To efficiently fit_lm models to each set of training data, we can use `purrr:map`. However, we'll need to write functions that fit_lm each model taking training data as input.

```
# define fitting function
fit_fn <- function(resamp, form){
  out <- lm(formula = form, data = resamp)
  return(out)
}
```

Here is an example of usage for our function:

```r
# extract one resample
example_resamp <- auto_cv %>%
  pull(train) %>%
  simplify() %>%
  first()

# specify formula
linear_form <- formula('mpg ~ horsepower')

# example use of fit_fn()
fit_fn(resamp = example_resamp, form = linear_form)
```

```
##
## Call:
## lm(formula = form, data = resamp)
##
## Coefficients:
## (Intercept)   horsepower
##     39.7690      -0.1588
```

We can 'map' the training partitions to our function using the following chain. This efficiently fits a model to each fold.

```r
# fit model to each training partition via `map`
auto_cv %>%
  mutate(fit_lm = map(train, ~ fit_fn(resamp = .x, form = linear_form)))
```

```
## # A tibble: 4 x 4
##   train          test           fold  fit_lm
##   <named list>   <named list>   <chr> <named list>
## 1 <resample[,9]> <resample[,9]> 1     <lm>
## 2 <resample[,9]> <resample[,9]> 2     <lm>
## 3 <resample[,9]> <resample[,9]> 3     <lm>
## 4 <resample[,9]> <resample[,9]> 4     <lm>
```

Adding extra mutate-map commands, it is possible to fit multiple models:

```r
# define another formula
poly_form <- formula('mpg ~ poly(horsepower, degree = 2, raw = T)')

# fit model to each training partition via `map`
auto_cv %>%
  mutate(fit_lm = map(train, ~ fit_fn(resamp = .x, form = linear_form)),
         fit_poly = map(train, ~ fit_fn(resamp = .x, form = poly_form)))
```

```
## # A tibble: 4 x 5
##   train          test           fold  fit_lm       fit_poly
##   <named list>   <named list>   <chr> <named list> <named list>
## 1 <resample[,9]> <resample[,9]> 1     <lm>         <lm>
## 2 <resample[,9]> <resample[,9]> 2     <lm>         <lm>
## 3 <resample[,9]> <resample[,9]> 3     <lm>         <lm>
## 4 <resample[,9]> <resample[,9]> 4     <lm>         <lm>
```

Your turn (2) Append another `mutate-map` command to the chain above so that the code simultaneously fits all three models: linear, quadratic, and spline. Store the result as `cv_out`.

```
# define a formula for the spline model
spline_form <- formula('mpg ~ bs(horsepower,df = 8)')
# fit linear, quadratic polynomial, and b-spline models to training partitions
cv_out <- auto_cv %>%
  mutate(fit_spline = map(train, ~ fit_fn(resamp = .x,form = spline_form)),
         fit_poly = map(train, ~ fit_fn(resamp = .x,form = poly_form)),
         fit_lm = map(train, ~ fit_fn(resamp = .x,form = linear_form)))
```

Now we also want to compute test-partition MSE for each fold. This can be done using `map2()`, which is exactly like `map()` but takes two arguments. An example of its use is shown below. (Notice that two *separate* mutate commands are used – they cannot be combined, because the fitted model column must be created first and then operated on to calculate MSEs.)

```
# fit linear model and then compute test mses
auto_cv %>%
  mutate(fit_lm = map(train, ~ fit_fn(resamp = .x, form = linear_form))) %>%
  mutate(mse_lm = map2(fit_lm, test, ~ mse(model = .x, data = .y)))
```

```
## # A tibble: 4 x 5
##   train          test           fold  fit_lm       mse_lm
##   <named list>   <named list>   <chr> <named list> <named list>
## 1 <resample[,9]> <resample[,9]> 1     <lm>         <dbl [1]>
## 2 <resample[,9]> <resample[,9]> 2     <lm>         <dbl [1]>
## 3 <resample[,9]> <resample[,9]> 3     <lm>         <dbl [1]>
## 4 <resample[,9]> <resample[,9]> 4     <lm>         <dbl [1]>
```

Your turn (3) Following the example above, fill in the additional commands below to compute test MSEs for all three models. Be sure to change the `eval = F` code chunk option to `eval = T` once you complete this part so that the result is shown in your knitted document!

```
## # A tibble: 4 x 9
##   train test  fold  fit_spline fit_poly fit_lm linear_mse quadratic_mse
##   <nam> <nam> <chr> <named li> <named > <name> <named li> <named list>
## 1 <res~ <res~ 1     <lm>       <lm>     <lm>   <dbl [1]>  <dbl [1]>
## 2 <res~ <res~ 2     <lm>       <lm>     <lm>   <dbl [1]>  <dbl [1]>
## 3 <res~ <res~ 3     <lm>       <lm>     <lm>   <dbl [1]>  <dbl [1]>
## 4 <res~ <res~ 4     <lm>       <lm>     <lm>   <dbl [1]>  <dbl [1]>
## # ... with 1 more variable: bspline_mse <named list>
```

To perform aggregation operations, it is necessary to `unnest` list columns. Averaging the MSE over fold can be done as follows. The code chunk below will work once you've completed the previous part.

```
# average mses over folds
cv_out %>%
  select(ends_with('mse')) %>%
  unnest(everything()) %>%
  summarize(across(everything(), mean))
```

```
## # A tibble: 1 x 3
##   linear_mse quadratic_mse bspline_mse
##        <dbl>         <dbl>       <dbl>
## 1       24.2          19.2        19.1
```

Your turn (4) Reflect on the results you've examined, specifically whether the cross-validation estimate of prediction error differs much from your vanilla test MSE beforehand, and which model has the lowest estimated prediction error. Do you think cross-validation is necessary to estimate prediction error in this particular problem? Why does the spline model have the best fit but not the lowest prediction error?

The cross-validation estimate of prediction error differs slightly to where it seems to make no big difference. Although the b-spline method has the lowest prediction error in cross-validation method, I would still pick the quadratic model because of how much more interpretable it is. I don't think cross-validation was necessary as with the vanilla test MSE, I would still pick the quadratic model. The spline model has the best fit because it is the most flexible, but in the cross-validation method, I actually got that the spline model DOES have the lowest prediction error.

Further exploration (optional – not for credit)

Choose your preferred model from the previous part for the relationship with horsepower, build a predictive model that incorporates the other variables in the dataset, and estimate its expected prediction error by 10-fold cross validation. This is very open-ended – it is not necessary to consider each variable separately, but you should make informed choices about model terms to include based on exploratory plots of the data.

Part one Make scatterplots of `mpg` against each variable.

Part two Decide on which variables to include in your model and how they will enter (linear, polynomial, on a transformed scale?). Partition the data into training and test sets, fit the model using our previous `fit_fn` to your training set, and compute test MSE.

Part three Adapt the code from the first part to estimate the expected prediction error by cross validation. Does the estimate differ much from your test set MSE?