02 - Cross-Validation and Model Selection

SYS 6018 | Fall 2020

02-crossval.pdf

Contents

1 Predictive Performance

1.1 Model Complexity

Models can be of varying complexity:

- number of parameters / edof (polynomials, interactions)
- penalty λ or constraint (t) for regularized models
- neighborhood size (knn)
- number of trees, tree depth (classification and regression trees, random forest, boosting)
- etc. (we will cover more models later in course)

Highly adaptable model families can accommodate complex relationships, but easily overemphasize patterns that are not reproducible (e.g. noise or statistical fluctuation)

Goal is to be flexible (complex) enough to find reproducible (true) structure, but not overfit

1.2 Tuning parameters

Tuning parameters are the "control knobs" of a model. It may be better to refer to these as *complexity parameters* because they are usually connected to the flexibility/complexity of a model. E.g., the λ in lasso and ridge are tuning parameters

We will represent the tuning parameters with ω:

- For knn regression, $\omega = k$
- For polynomial regression, $\omega = J$ in the model $\hat{f}(x) = \sum_{j=0}^{J} \beta_j x^j$
- In best subsets regression, $\omega = p$, the total number of features allowed in the model
- In ridge regression, $\omega = \lambda$ in the ridge penalty $\text{Pen}(\beta) = \lambda \sum_{\text{j}=1}^{\text{p}} \beta_{\text{j}}^2$

Given the value of the tuning parameter, it is often straightforward to estimate the *model parameters* from the data (e.g., $\hat{\beta}$)

1.3 Predictive Performance

Because our focus is on predictive performance (not interpretation or inference), the optimal tuning parameter(s), given training data D, is the value(s) that minimizes the expected prediction error (PE):

$$\text{EPE}(\omega) = \text{E}[\text{PE}(\omega)] = \text{E}_{\tilde{X},\tilde{Y}}[\ell(\tilde{Y}, \hat{f}_\omega(\tilde{X}))]$$

where:

- $\ell()$ is the loss function (e.g., RSS/MSE)
- $\tilde{X}, \tilde{Y}$ are the *test data* drawn from the same distribution (hopefully) as the training data
- $\hat{f}_\omega(x) = f_\omega(x; \hat{\theta}_\omega(\mathcal{D}))$ is the prediction function which has been estimated under the tuning parameter ω
  - $\hat{\theta}_\omega(\mathcal{D})$ are the **model parameters** estimated from the training data D *given* the tuning value ω.

Ideally, we want to pick tuning parameter(s) ω to minimize EPE

$$\omega^{opt} = \arg\min_{\omega} \text{EPE}(\omega)$$

1.4 Training Error

There are a few ways to estimate EPE. A *not* very good way is to use the training error (TE):

$$\text{TE}(\omega, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, \hat{f}_\omega(x_i))$$

where $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ is the training data.

Using the ω that minimizes the training error:

$$\omega^* = \arg\min_{\omega} \ \text{TE}(\omega, \mathcal{D})$$

Will always lead to a model that overfits (as long as f is flexible enough) Why?

1.5 Training Error vs. Testing Error

Figure Taken from: ESL Fig 7.1

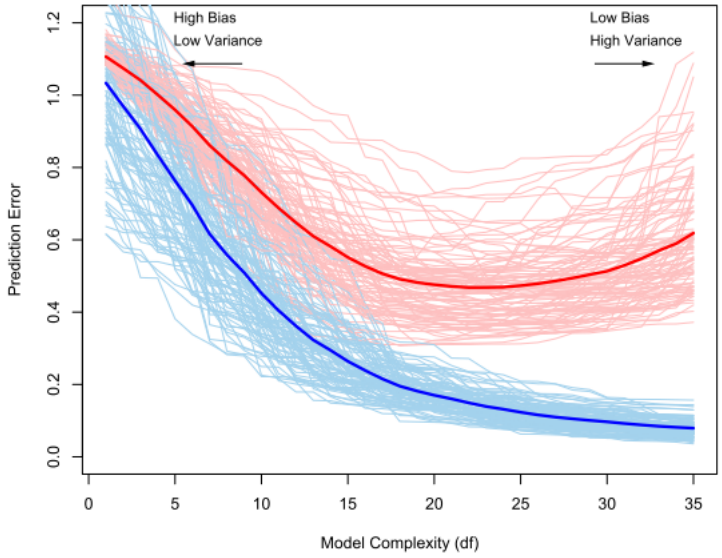


FIGURE 7.1. *Behavior of test sample and training sample error as the model complexity is varied. The light blue curves show the training error* $\overline{\text{err}}$*, while the light red curves show the conditional test error* $\text{Err}_{\mathcal{T}}$ *for* 100 *training sets of size* 50 *each, as the model complexity is increased. The solid curves show the expected test error* Err *and the expected training error* $\text{E}[\overline{\text{err}}]$.

1.6 Model Selection and Assessment

- **Model selection:** estimating the performance of different models in order to choose the best one.

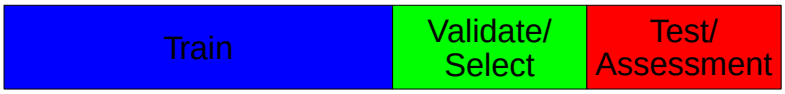
$$\omega^* = \arg\min_{\omega} \text{EPE}(\omega)$$

- **Model assessment:** having chosen a final model, estimating its prediction error on new data (i.e., estimating EPE for final model).

$$\text{EPE}(\omega^*) = \text{E}_{\tilde{X},\tilde{Y}}[\ell(\tilde{Y}, \hat{f}_{\omega^*}(\tilde{X}))]$$

1.7 Train/Validate/Test

If it were possible to have lots of data (all from the same distribution), then we would split up the data into three pieces:



- Train: Estimate model parameters θ (for given tuning ω) for many models ($\omega_1, \omega_2, \ldots$)
  - Use the training dataset to estimate the *model parameters* (e.g., β) for a set of *tuning parameters* (e.g., ω) (and possibly different model families)
  - The output from this step will be a set of fitted models, $\hat{f}_1, \hat{f}_2, \ldots$
- Validate/Select: Choose optimal tuning parameters ω (i.e., model selection step)
  - Evaluate the performance of each fitted model on the Validate/Select data
  - Choose the best/final model based on the performance
- Test/Assessment: Estimate EPE (i.e., final model assessment)
  - Never use the Test/Assessment data until **all** model fitting, tuning, selection is finished.
  - Then the performance of the best/final model can be assessed (without bias)

Your Turn #1

1. What estimates suffers where there is not enough data in each group?
2. Is the performance of a model on the validation data reflective of its performance on the Test/Assessment data? When/Why do we need Test/Assessment data?

2 Model Selection

The goal of model selection is to pick the model that will provide the best *predictive performance* on test data (i.e, model with smallest EPE).

- Note: "model" means model family plus tuning parameter (e.g., polynomial with degree = 3 or knn with k = 12)

There are three main approaches to estimating the predictive performance (EPE) of a model:

1. Predict on hold-out data
2. Make a mathematical adjustment to the training error that better estimates the test error
3. Resampling methods (cross-validation, bootstrap)
    - i.e., use *multiple* hold-out sets

2.1 Hold out set

An obvious way to assess how well a model will perform on test data is to evaluate it on hold-out data.

Split the data into a training and test set: $\mathcal{D} = (\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}})$.

- Fit models with $\mathcal{D}_{\text{train}}$ to get estimates $\hat{\theta}_\omega = \hat{\theta}_\omega(\mathcal{D}_{\text{train}})$ and $\hat{f}_\omega(\cdot) = f(\cdot; \hat{\theta}_\omega)$.
- Use $\mathcal{D}_{\text{test}}$ to calculate:

$$\text{PE}(\omega, \mathcal{D}_{\text{test}}) = \frac{1}{m} \sum_{j=1}^{m} \ell(\tilde{y}_j, \hat{f}_\omega(\tilde{x}_j))$$

  where $(\tilde{x}_j, \tilde{y}_j) \in \mathcal{D}_{\text{test}}$ for $j = 1, 2, \ldots, m$.

2.1.1 Problems with using a single Hold-Out set

1. Need to decide how much data into each set?
    - Too little in training and poor parameter estimates
    - Too little in test and poor performance estimates and model selection
2. We may happen to choose a split that uses a train or test set that poorly represents the data generating process.
    - The chance of *bad* split is reduced when $n - m$ and m are both large.

2.2 Adjustments to Training Error (AIC/BIC)

Another approach is to use all the data for training, but adjust the training error to account for potential over-fitting

- The adjustment is usually a function of model complexity (e.g., edf)

Examples:

- AIC/BIC
- Adjusted R^2
- Mallow's Cp

2.2.1 AIC/BIC

- Let $\mathcal{M}$ be a model (e.g., model family and ω)
- Let $\hat{\mathcal{M}} = \arg\max_\theta \mathcal{M}(\theta)$ be the parameters that maximize the likelihood (or penalized log-likelihood).
- $L(M)$ is likelihood of the model
  - Thus, to use AIC/BIC a distributional assumption must be specified
- Let $d(\hat{\mathcal{M}})$ be the *effective degrees of freedom (edf)* (e.g., number of estimated model parameters) of the model

Akaike information criterion (AIC)

$$\text{AIC}(\mathcal{M}) = -2\log L(\hat{\mathcal{M}}) + 2d(\hat{\mathcal{M}})$$

Bayesian information criterion (BIC)

$$\begin{aligned}\text{BIC}(\mathcal{M}) &= -2\log L(\hat{\mathcal{M}}) + d(\hat{\mathcal{M}})\log n \\ &= \text{AIC}(\hat{\mathcal{M}}) + d(\hat{\mathcal{M}})\left(\log(n) - 2\right)\end{aligned}$$

Information Criterion (generic)

$$\begin{aligned}\text{IC}(\mathcal{M}) &= \ell(\hat{\mathcal{M}}) + P(\hat{\mathcal{M}}) \\ &= \text{Loss} + \text{Penalty}\end{aligned}$$

Example

For a linear regression model with $d = p + 1$ estimated coefficients, $\mathcal{M}_d = f(x;\beta) = \sum_{j=0}^{p}\beta_j x_j$,

- d is the tuning parameter, β are the model parameters, Gaussian distribution:

$$\log L(\hat{\mathcal{M}}_d) = -\frac{n}{2}\log(\text{MSE}(\hat{\beta})) + C$$

where C is a constant that doesn't depend on any model parameters or tuning parameters

$$\text{AIC}(\hat{\mathcal{M}}_\text{d}) = \text{n}\log(\text{MSE}(\hat{\beta})) + 2\text{d}$$

- Choose the model that *minimizes* the AIC/BIC
- The AIC/BIC can be used for any likelihood (e.g., logistic regression)

2.2.2 Adjusted R^2

$$R^2_{\text{adj}} = 1 - \frac{\text{RSS}(\hat{\beta})}{\text{RSS}(\bar{y})}\frac{n-1}{n-d}$$

where d is the number of estimated parameters in the model.

- R^2_{adj} is only appropriate under a squared error loss function.
- Choose the model with the *largest* R^2_{adj}

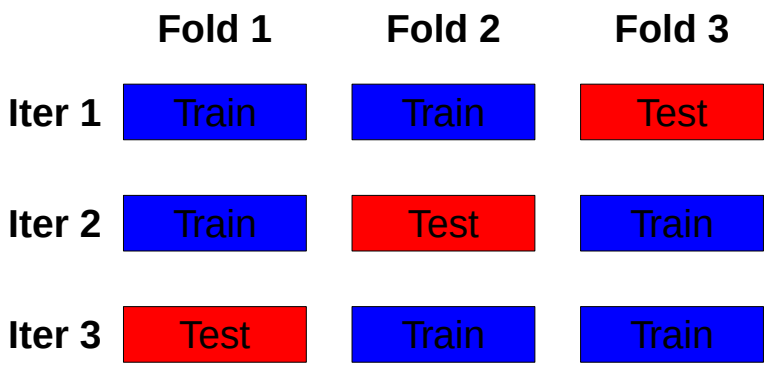
3 Cross-Validation and other Resampling based model selection procedures

3.1 Cross-Validation

Cross-validation is a way to use more of the data for **both** training and testing

- Randomly divide the set of observations into K groups, or folds, of approximately equal size.
- The first fold is treated as a validation set, and the model parameters are estimated from the remaining $K-1$ folds. And predictions are made on the hold-out set.
- The performance on each fold is combined to get a more accurate assessment of model performance on future data.

3.1.1 3-fold cross-validation



In practice, the observations should be assigned **randomly** to the folds (not sequentially like in the above figure)

- This will reduce the influence of potential correlation in the data

3.2 Cross-Validation Algorithm

Algorithm: Cross-Validation

1. Split data into K folds (of roughly equal size)
   - $\mathcal{F}_1, \ldots, \mathcal{F}_K$
2. For $k = 1, \ldots, K$ and for all models (e.g., for all ω) :
   a. Use the data in $\mathcal{D} \setminus \mathcal{F}_k$ to estimate the model $\hat{f}_\omega^{-k}$
   b. Predict for data in $\mathcal{F}_k$ and calculate the average loss

$$L_k(\omega) = \frac{1}{n_k} \sum_{i \in \mathcal{F}_k} \ell(y_i, \hat{f}_\omega^{-k}(x_i))$$

   where n_k is the number of observations in fold k
3. Choose tuning parameters (model selection) that minimize cross-validation loss

$$\hat{\omega} = \arg\min_\omega \text{CV}(\omega)$$

   where $\text{CV}(\omega) = \frac{1}{n} \sum_{k=1}^K n_k L_k(\omega)$
4. Refit all data using $\hat{\omega}$ to get the final model ($\hat{\theta}$).
   - An alternative is to use an *ensemble* model by combining the models from all folds with weights $1/K$

Note: in step 3, do not blindly choose the tuning parameter without looking at the $\{L_k(\omega)\}$! - What is the variability across folds? - Does performance differ over folds? - Do some folds cause outliers?

3.3 Alternative CV error approach

Instead of summarizing the performance in each fold, we can predict the response and summarize after all folds are predicted.

truth	fold	Model 1	Model 2
y_1	5	$\hat{y}_1^1$	$\hat{y}_1^2$
y_2	2	$\hat{y}_1^1$	$\hat{y}_1^2$
y_3	2	$\hat{y}_1^1$	$\hat{y}_1^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
y_n	9	$\hat{y}_1^1$	$\hat{y}_1^2$

Then we can calculate the loss for every observation, e.g., $L_i(\omega) = (y_i - \hat{y}_i(\omega))^2$ and overall cross-validation error is $\text{CV}(\omega) = \frac{1}{n} \sum_{i=1}^n L_i$.

3.4 1 SE Rule

One Standard Error Rule suggests that instead of using $\hat{\omega} = \arg\min_\omega \text{CV}(\omega)$, we should use the least complex model that is within one standard error of $\text{CV}(\hat{\omega})$.

- To adjust for the uncertainty in the evaluation results. "Given everything equal, choose the least complex model" (*parsimony*)

The standard error of the average cross-validation error can be estimated by:

$$\text{SE}(\omega) = \frac{\text{standard deviation of } \{L_k\}}{\sqrt{K}}$$

Using the alternative approach, the standard error is estimated as

$$\text{SE}(\omega) = \frac{\text{standard deviation of } \{L_i\}}{\sqrt{n}}$$

The two estimates will not be equal, but should be close.

3.5 Choice of K

- $K = 5, 10, n$ are common choices
- $K = n$ is called *leave-one-out (LOOCV)*

3.5.1 Performance Bias/Variance Trade-off

- Note: around $(K-1)/K$ of the data is used for training and $1/K$ for testing
- if K is too small, then not enough training data and poor cv error estimate (**bias** in prediction error)
- if K is too large, then the f_ω^{-k} are correlated and variance is not reduced (**variance** in prediction error)
  - because the training data is similar across folds
- Computational: need to fit each model K times

3.6 Balanced Data Splitting for Cross-Validation

- The most basic approach is to use random sampling to assign observations to folds
- But this can be problematic if there are outliers or classes with small frequency
  - Especially a problem for categorical data. Some levels are not included in training set.
  - So how to predict when they show up in test set?
- Stratified sampling can be used to ensure similar distributions in each fold
  - e.g., equal number of observations in each fold from same quartile of y
- Or based on predictor values: clustering the training data and then assigning into folds such that an equal number of observations from each cluster are in each fold

3.7 Different Model Families

Most of the discussion has been on finding the optimal complexity/tuning parameter for a given *model family*.

But cross-validation can be used to compare across model families.

Consider some options and their corresponding complexity/tuning parameters:

- linear regression with stepwise variable selection
  - number of parameters p, or acceptance/rejection criteria
- elastic net (includes lasso and ridge)
  - α and λ
- k-nearest neighbor
  - k

Use CV to pick best-of-the-best

Be careful to ensure that all aspects of estimation (e.g., variable selection, tuning parameter selection) are **inside** the cross-validation.

3.8 Repeated Cross-Validation

If you have the patience (or computing resources), you can be more certain about the model performance if you repeat cross-validation several times.

- if you repeat K-fold cross-validation 5 times, then you will need to fit each model $5K$ times
- Take the average of the cross-validation score as the performance measure.

3.9 Repeated Train/Test splits

But why even bother about the added code complexity involved in making the folds?

- Just repeatedly split the data into a training and test set
- This will be similar to the repeated cross-validation, but is just simpler to code
  - holding out 10% of the data is equivalent to $K = 10$ fold cross-validation.

3.10 Out-of-Bag

We already saw how the bootstrap can be used for model selection.

- use the out-of-bag observations to estimate predictive performance

Because about 37% of the data will be used for model assessment (testing data), it is similar to $K = 3$ cross-validation (if repeated 3 times).

- But since you still use n observations for model fitting, it will not be equivalent

3.11 Resampling Comparison

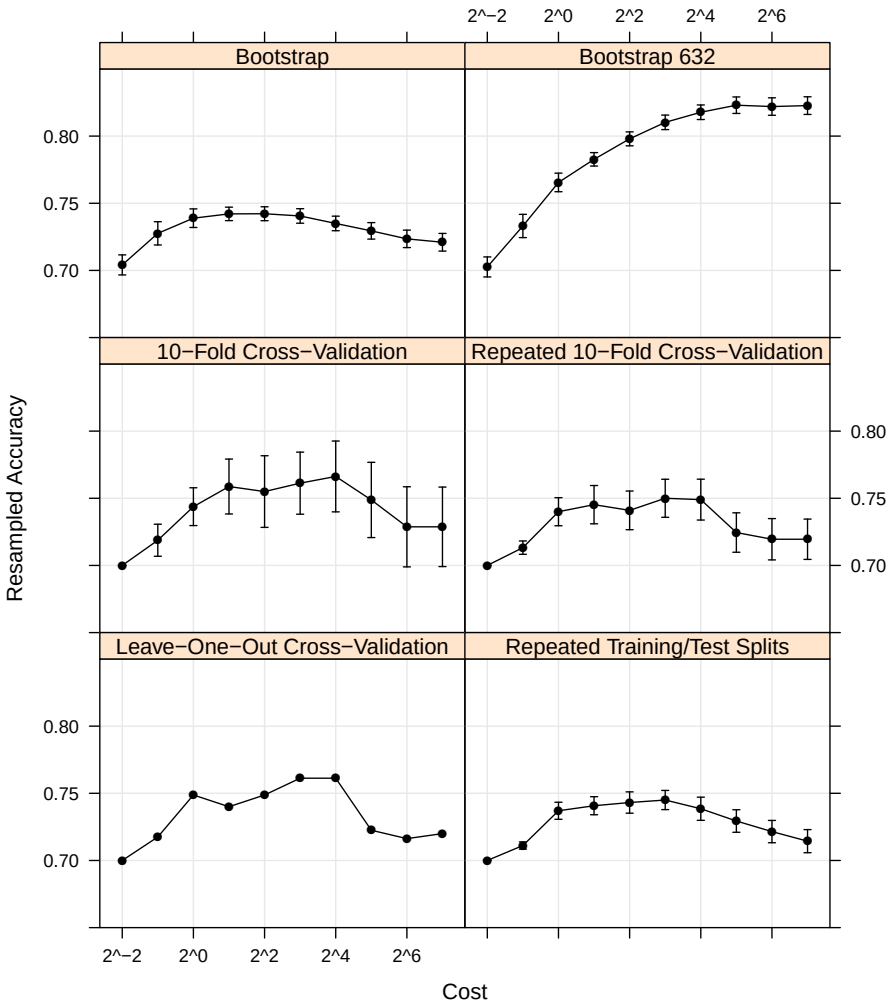


Figure 4.10 from Applied Predictive Modeling, by Kuhn and Johnson. Retrieved on Oct 22, 2019 from https://github.com/topepo/APM_Figures

3.12 Linear Smoothers and LOOCV

A *linear smoother* is a model that the fitted training data can be represented by the equation

$$\hat{Y} = HY$$

- For example, in linear regression $\hat{Y} = X\hat{\beta}$ and therefore, $H = X(X^\mathsf{T}X)^{-1}X^\mathsf{T}$.
  - H is called the *hat matrix* or *projection matrix*
  - The diagonal elements of H are called the *leverages* of the observations
  - High leverage points has a large impact on the model fit. Thus, the LOOCV will be especially sensitive to observations with high leverage.

It turns out that for linear smoothers, the LOOCV error (under squared error loss) is

$$\text{LOOCV}(\omega) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{f}_\omega(x_i)}{1 - h_{ii}}\right)^2$$

where h_{ii} is the i^{th} diagonal element of H.

- This takes away the computational burden of n model fits! The model is fit once to the full data and *corrected* for in the above equation.
- A similar, but even faster to compute, version is the *Generalized Cross-Validation*

$$\begin{aligned}\text{GCV}(\omega) &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{f}_\omega(x_i)}{1 - tr(H)/n}\right)^2 \\ &= \frac{\text{MSE}_\omega}{1 - tr(H)/n}\end{aligned}$$

  where $tr(H) = \sum_{i=1}^{n} h_{ii}$ is the *trace* of H.

  - In unpenalized linear regression $tr(H) = d$, the number of estimated parameters.

4 R Code

4.1 Simulate Data

```r
#-- Data Generation
n = 100                                 # number of observations
generate_x <- function(n) runif(n)      # U[0,1]
f <- function(x) 1 + 2*x + 5*sin(5*x)   # true mean function
sd = 2                                  # stdev for error

set.seed(825)                           # set seed for reproducibility
x = generate_x(n)                       # get x values
y = f(x) + rnorm(n, sd=sd)              # get y values
data_train = tibble(x,y)                # training data
```

4.2 Approach #1: Calculate MSE for each fold

First, we make a function that will fit a set of B-spline models (of varying complexity) to the training data, make predictions on the test data, and calculate the MSE.

```r
library(splines)  # for the bs() function
library(dplyr)    # for tibble() and bind_rows() functions

# sp_fit(): fit set of B-spline models and evaluate on test data
#-------------------------------------------------------------------------
# data_train, data_test: training and test data (requires column names x,y)
# DF: set of spline degrees (tuning parameters)
# kts.bdry: boundary knots for the splines
# output: tibble with df and associated mean squared error (MSE) on test data
sp_fit <- function(data_train, data_test, DF = seq(4, 15, by=1), kts.bdry = c(-.2, 1.2)) {

  MSE = numeric(length(DF))

  for(i in 1:length(DF)) {
    #- set tuning parameter value
    df = DF[i]
    #- fit with training data
    fit = lm(y~bs(x, df=df, Boundary.knots=kts.bdry)-1,
                    data=data_train)
    #- predict on test data
    yhat = predict(fit, newdata=data_test)
    #- get errors / loss
    MSE[i] = mean( (data_test$y - yhat)^2 )
  }

tibble(df=DF, mse=MSE)

}
```

Next we create the folds and loop over the folds calculating the test MSE on each fold.

```r
library(splines)  # for the bs() function
library(dplyr)    # for tibble() and bind_rows() functions

#- Get K-fold partition
set.seed(2019)                      # set seed for replicability
n.folds = 10                        # number of folds for cross-validation
fold = sample(rep(1:n.folds, length=n))  # vector of fold labels
# notice how this is different than:  sample(1:K,n,replace=TRUE),
```

```r
#  which won't give almost equal group sizes

#- initialize
DF = seq(3, 15, by=1)      # edfs for spline
kts.bdry = c(-.2, 1.2)
results = tibble()

#- Iterate over folds
for(j in 1:n.folds){

  #-- Set training/val data
  val = which(fold == j)     # indices of validation data
  train = which(fold != j)   # indices of training data
  n.val = length(val)        # number of observations in validation

  #-- fit set of spline models
  results_j = sp_fit(data_train[train,], data_train[val,], DF=DF)
  results = bind_rows(results,
                      results_j %>% mutate(n.val, fold=j))
}
```

Now we can calculate the summary values and make a nice plot.

```r
#- Summary Measures
R = results %>%
  mutate(sse = mse*n.val) %>% # use this because sample size of folds may not be equal
  group_by(df) %>%
  summarize(K=n(), sse = sum(sse), MSE=sse/nrow(data_train),
            mse_mu = mean(mse), mse_sd = sd(mse), se = mse_sd/sqrt(K))
R %>% knitr::kable(digits=2)
```

df

K

sse

MSE

mse_mu

mse_sd

se

3

10

580.9

5.81

5.81

2.99

0.95

4

10

482.9

4.83

4.83

2.67

0.85

5

10

458.3

4.58

4.58

2.37

0.75

6

10

473.0

4.73

4.73

2.46

0.78

7

10

474.6

4.75

4.75

2.51

0.79

8

10

475.2

4.75

4.75

2.50

0.79

9

10

469.7

4.70

4.70

2.41

0.76

10

10

470.7

4.71

4.71

2.44

0.77

11

10

472.2

4.72

4.72

2.40

0.76

12

10

481.2

4.81

4.81

2.40

0.76

13

10

488.6

4.89

4.89

2.21

0.70

14

10

484.6

4.85

4.85

2.25

0.71

15

10

492.1

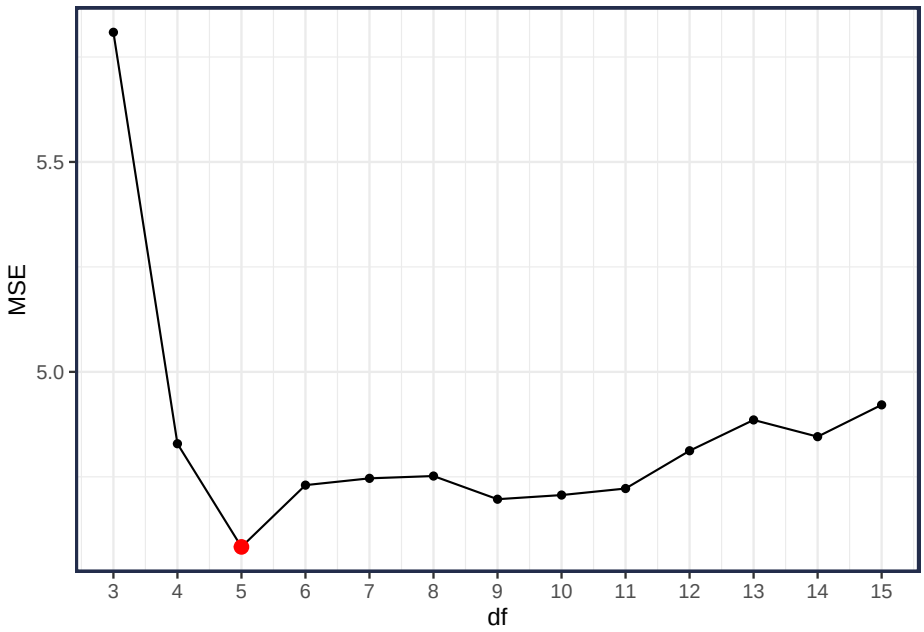
4.92

4.92

2.07

0.65

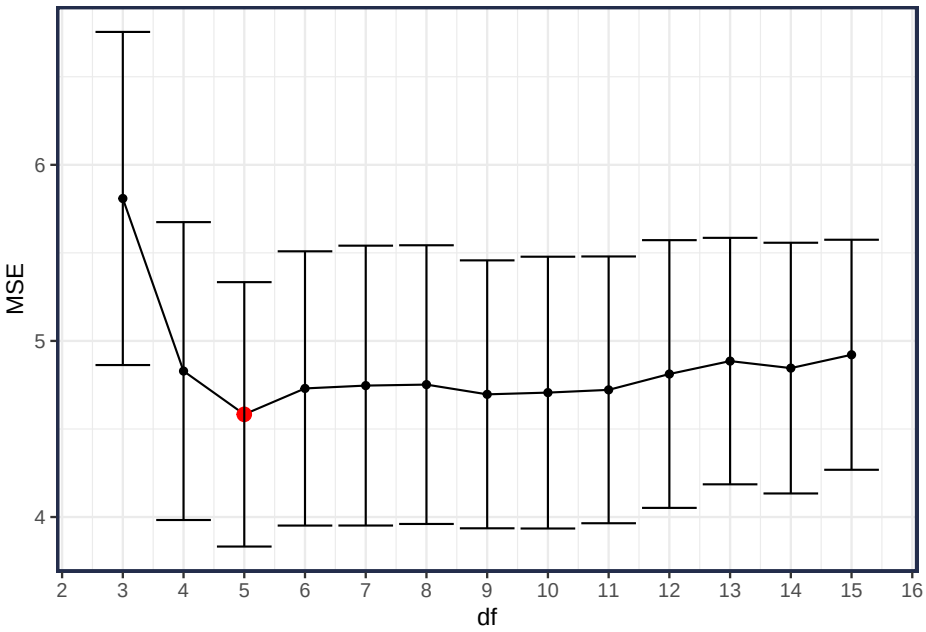
```
#- Plot results
R %>%
  ggplot(aes(df, MSE)) + geom_point() + geom_line() +
  geom_point(data=. %>% filter(MSE==min(MSE)), color="red", size=3) +
  scale_x_continuous(breaks=1:20)
```



- The minimum cross-validation error occurs at `df=5`. This matches the optimal complexity from the out-of-bag bootstrap analysis and the polynomial fit from the previous lecture notes.

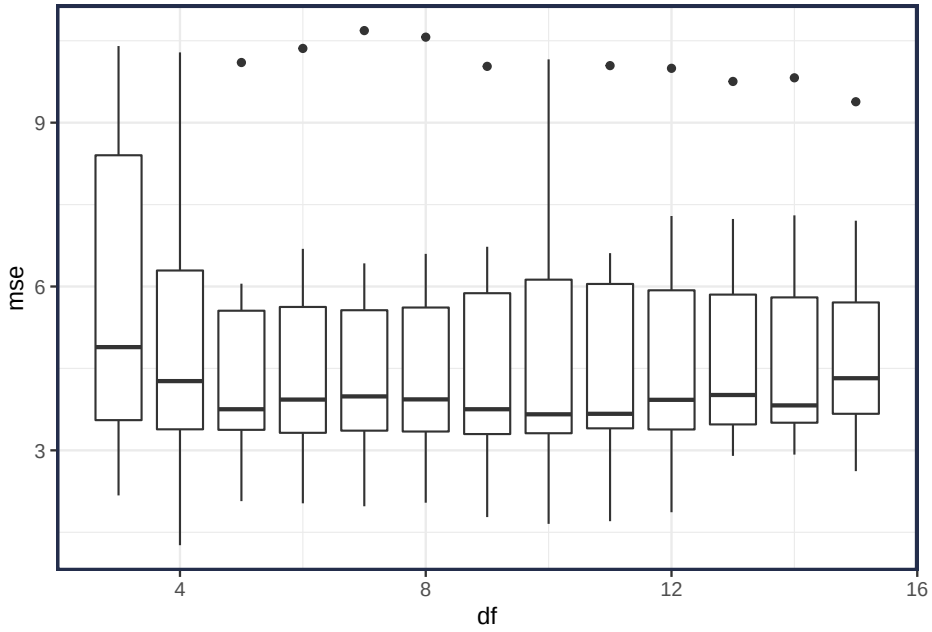
All seems good, no? What if we add the uncertainty - here is the same plot with error bars (1 standard error)

```
R %>%
  ggplot(aes(df, MSE)) + geom_point() + geom_line() +
  geom_point(data=. %>% filter(MSE==min(MSE)), color="red", size=3) +
  geom_errorbar(aes(ymin=MSE-se, ymax=MSE+se)) +
  scale_x_continuous(breaks=1:20)
```



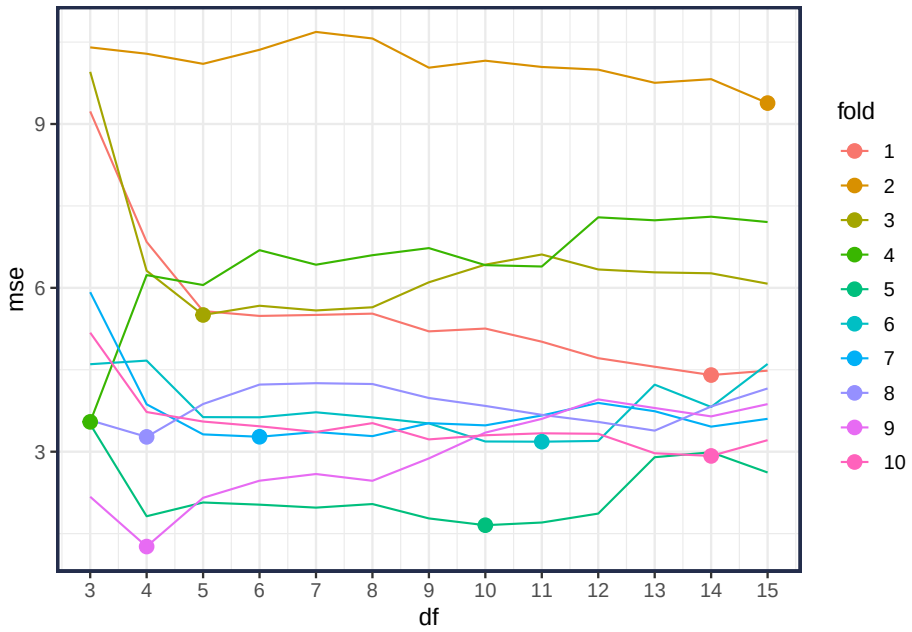
Notice how large the errorbars are! Let's dig into this a bit:

```
#- Boxplots
results %>%
  ggplot(aes(df, mse, group=df)) + geom_boxplot()
```



The boxplots show some outliers. How about a line plot, with one line per fold

```
results %>%
  ggplot(aes(df, mse, color=factor(fold))) + geom_line() +
  geom_point(data=. %>% group_by(fold) %>% filter(mse==min(mse)), size=3) +
  labs(color="fold") +
  scale_x_continuous(breaks=1:20)
```



Ah, this shows that fold 2 had unusually bad performance and that the optimal df parameter varies substantially over the folds. So while I'd pick `df=5` because it had the best aggregate performance, I'm not very confident that there isn't a better option; this leads me think more training data would be important for this problem.

4.3 Approach #2: Predict $\hat{y}$ and then calculate MSE and standard error

First, we make a function that will fit a set of B-spline models (of varying complexity) to the training data and make predictions on the test data.

```
library(splines)  # for the bs() function
library(dplyr)    # for tibble() and bind_rows() functions

# sp_pred(): fit set of B-spline models and predict for test data
#--------------------------------------------------------------------
# data_train, data_test: training and test data (requires column names x,y)
# DF: set of spline degrees (tuning parameters)
# kts.bdry: boundary knots for the splines
# output: matrix of predicted values (one column per DF)
sp_pred <- function(data_train, data_test, DF = seq(4, 15, by=1), kts.bdry = c(-.2, 1.2)) {

  yhat = matrix(NA, nrow(data_test), length(DF))

  for(i in 1:length(DF)) {
    #- set tuning parameter value
    df = DF[i]
    #- fit with training data
    fit = lm(y~bs(x, df=df, Boundary.knots=kts.bdry)-1,
                     data=data_train)
    #- predict on test data
    yhat[,i] = predict(fit, newdata=data_test)
  }
return(yhat)
}
```

Next we create the folds and loop over the folds predicting the response variables using the out-of-sample data.

```r
library(splines)  # for the bs() function
library(dplyr)    # for tibble() and bind_rows() functions

#- Get K-fold partition
set.seed(2019)                       # set seed for replicability
n.folds = 10                         # number of folds for cross-validation
fold = sample(rep(1:n.folds, length=n))  # vector of fold labels
# notice how this is different than:  sample(1:K,n,replace=TRUE),
#  which won't give almost equal group sizes

#- initialize
DF = seq(3, 15, by=1)      # edfs for spline
kts.bdry = c(-.2, 1.2)
yhat = matrix(NA, nrow(data_train), length(DF))
colnames(yhat) = DF

#- Iterate over folds
for(j in 1:n.folds){

  #-- Set training/val data
  val = which(fold == j)     # indices of validation data
  train = which(fold != j)   # indices of training data
  n.val = length(val)        # number of observations in validation

  #-- fit set of spline models
  yhat[val, ] = sp_pred(data_train[train,], data_train[val,], DF=DF)
}
```

Now we can calculate the summary values and make a plot.

```r
#- matrix of squared error (one column per DF)
sq.error = apply(yhat, 2, function(est) (data_train$y - est)^2)

#- aggregate results
R = tibble(df = colnames(sq.error) %>% as.numeric(),
       MSE = colMeans(sq.error),
       sd = apply(sq.error, 2, sd),
       se = sd / sqrt(nrow(sq.error)))
R %>% knitr::kable(digits=2)
```

df

MSE

sd

se

3

5.81

8.62

0.86

4

4.83

7.04

0.70

5

4.58

6.81

0.68

6

4.73

7.00

0.70

7

4.75

7.22

0.72

8

4.75

7.15

0.72

9

4.70

6.96

0.70

10

4.71

7.03

0.70

11

4.72

7.04

0.70

12

4.81

7.11

0.71

13

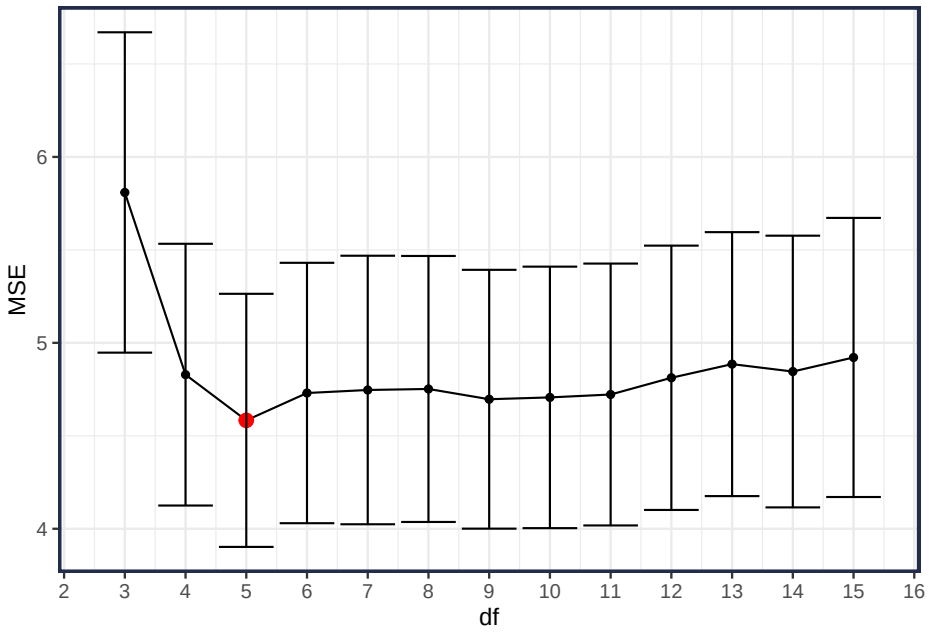
4.89

7.10

0.71

```
14
4.85
7.31
0.73
15
4.92
7.51
0.75
```

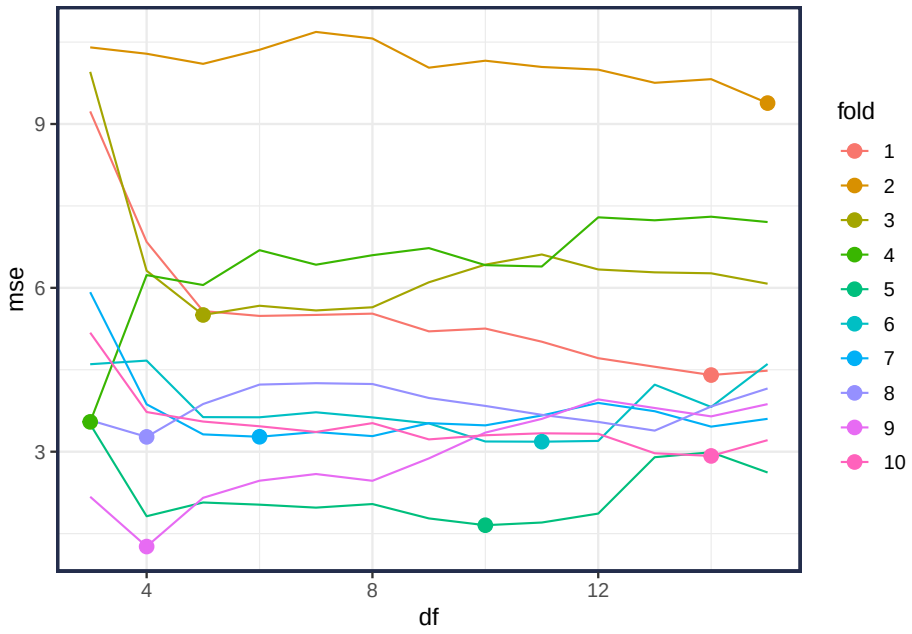
```
R %>%
  ggplot(aes(df, MSE)) + geom_point() + geom_line() +
  geom_point(data=. %>% filter(MSE==min(MSE)), color="red", size=3) +
  geom_errorbar(aes(ymin=MSE-se, ymax=MSE+se)) +
  scale_x_continuous(breaks=1:20)
```



Notice that the two methods give slightly different estimates of the standard errors. I would use this one, but they should be very similar in most situations.

The line plot can be made with `ggplot2` by first converting the `sq.error` to a tibble and doing some manipulation

```
sq.error %>%
  as_tibble() %>%          # convert the matrix into a tibble
  mutate(fold = fold, row=row_number()) %>% # add fold information and row number
  pivot_longer(cols=c(-fold, -row), names_to="df", values_to="sq.error") %>% # long format
  group_by(fold, df) %>% summarize(mse=mean(sq.error)) %>%
  mutate(df = as.numeric(df), fold = factor(fold)) %>%
  ggplot(aes(df, mse, color=fold)) + geom_line() +
  geom_point(data=. %>% group_by(fold) %>% filter(mse==min(mse)), size=3)
```



4.4 Repeated Hold-out

```r
library(splines)  # for the bs() function
library(dplyr)    # for tibble() and bind_rows() functions

#- Settings
set.seed(2019)                      # set seed for replicability
n.holdout = 10                      # number of hold-out samples
n.reps = 100                        # number of replications


#- initialize
DF = seq(3, 15, by=1)      # edfs for spline
kts.bdry = c(-.2, 1.2)
results = tibble()

#- Iterate over folds
for(j in 1:n.reps){

  #-- Set training/val data
  holdout = sample.int(n, size=n.holdout, replace=FALSE) # hold out indices
  train = -holdout                                       # training indices

  #-- fit set of spline models
  results_j = sp_fit(data_train[train,], data_train[holdout,], DF=DF)
  results = bind_rows(results,
                      results_j %>% mutate(iter=j))
}
```

```r
#- Summary Measures
R = results %>%
  group_by(df) %>%
  summarize(MSE = mean(mse),
            mse_sd = sd(mse), se = mse_sd/sqrt(n.reps))
R %>% knitr::kable(digits=2)
```

df

MSE

mse_sd

se

3

5.61

2.50

0.25

4

4.53

2.09

0.21

5

4.30

2.06

0.21

6

4.37

2.11

0.21

7

4.52

2.24

0.22

8

4.64

2.27

0.23

9

4.53

2.20

0.22

10

4.51

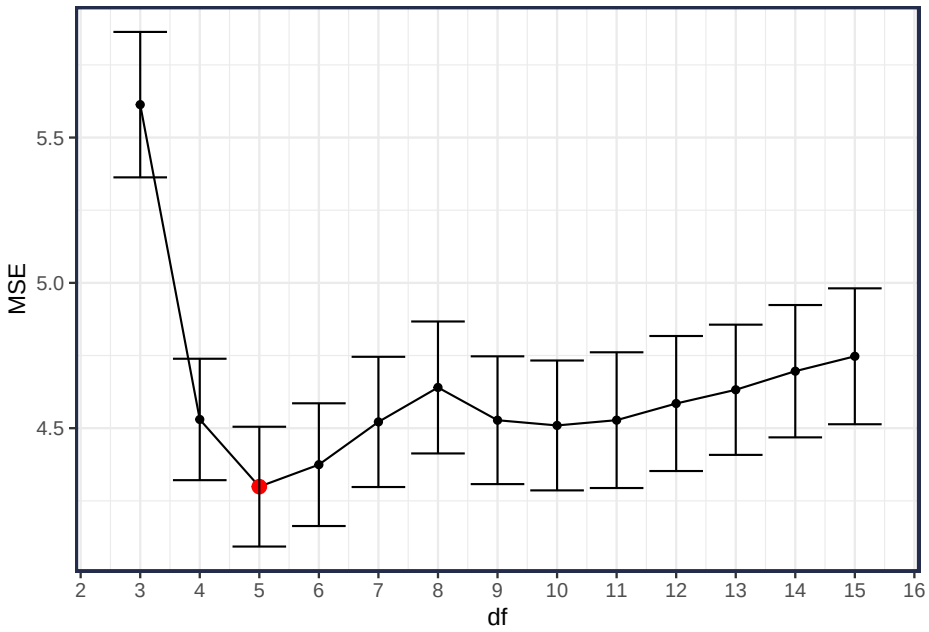
2.23

0.22

11
4.53
2.34
0.23
12
4.58
2.32
0.23
13
4.63
2.24
0.22
14
4.70
2.28
0.23
15
4.75
2.34
0.23

```

#- Plot results
R %>%
  ggplot(aes(df, MSE)) + geom_point() + geom_line() +
  geom_point(data=. %>% filter(MSE==min(MSE)), color="red", size=3) +
  geom_errorbar(aes(ymin=MSE-se, ymax=MSE+se)) +
  scale_x_continuous(breaks=1:20)
```



Notice the reduction in error bars. We can increase n.reps to achieve even smaller uncertainty or use smaller hold-out size to reduce the variance of the model parameters.