Model Selection and Assessment

DS-6030 | Spring 2026

selection.pdf

Table of contents

1 Predictive Performance

1.1 Model Complexity

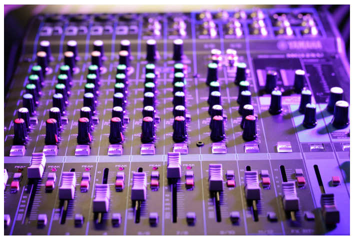
Most model families can be tuned to have a complexity (or edf):

- number of parameters in a linear model (polynomials, interactions, subset selection)
- neighborhood size (the k in knn models)
- penalty λ or constraint (t) for regularized models
- number of trees, tree depth (classification and regression trees, random forest, boosting)
- We will cover many more models with tuning parameters later in the course

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

> The goal is to *tune* the model structure so that it has just enough flexibility (complexity) to capture the reproducible (true) structure, but not too much that it overfits.
>
> - Note: the optimal complexity for a given training data set depends on the sample size.

1.2 Tuning parameters



Tuning parameters are the "control knobs" of a model.

- Some tuning parameters directly control the complexity/flexibility of a model (e.g., the k in knn).
- Other tuning parameters impact the structure or algorithm of a model (e.g., using Euclidean or Manhattan distance in knn)
- Other ancillary aspects like pre-processing, feature engineering, and imputation approaches can also be considered as tuning steps.
  - Some implementations can scale data internally before estimating parameters (e.g., `glmnet()`)
  - Dropping observations with missing data or using median imputation

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

This discussion will focus on the *complexity parameters*, but the same framework will help you choose the best values of the other configurable aspects of a model. We represent the tuning parameters with ω. For example,:

- 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 $\text{Pen}(\beta) = \lambda \sum_{j=1}^{p} \beta_j^2$

1.3 Predictive Performance

Because our focus is on predictive performance (not interpretation or inference), the optimal tuning parameter(s), given training data $\mathcal{D}_{\text{train}}$, 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:

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

Ideally, we want to pick tuning parameter(s) ω to minimize EPE, given the training data

$$\omega^{opt} = \underset{\omega}{\arg\min}\ \text{EPE}(\omega \mid \mathcal{D}_{\text{train}})$$

Note

The optimal tuning parameters are a function of the training data size. Usually, as the training size increases, a more flexible/complex (higher edf) model will give better predictions.

1.4 Training Error vs. Testing Error

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

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

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

Using the ω that minimizes the training error:

$$\omega^* = \underset{\omega}{\arg\min}\ \text{TrE}(\omega, \mathcal{D}_{\text{train}})$$

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

Figure Taken from: ESL Fig 7.1

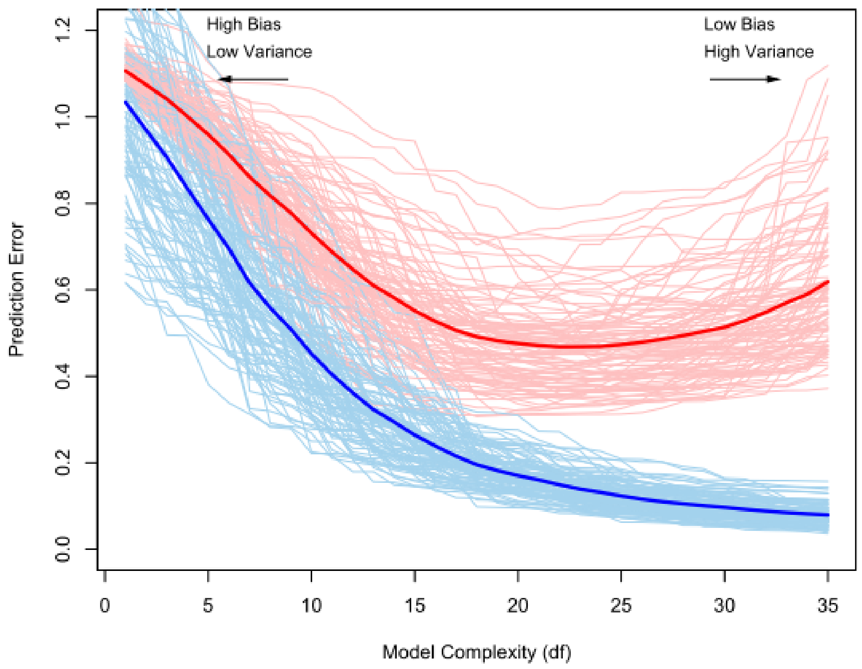


FIGURE 7.1. *Behavior of test sample and training sample error as the model complexity is varied. The light blue curves show the training error* $\overline{\mathrm{err}}$, *while the light red curves show the conditional test error* $\mathrm{Err}_{\mathcal{T}}$ *for* 100 *training sets of size* 50 *each,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

2 Estimating Predictive Performance

2.1 Model Selection

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

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

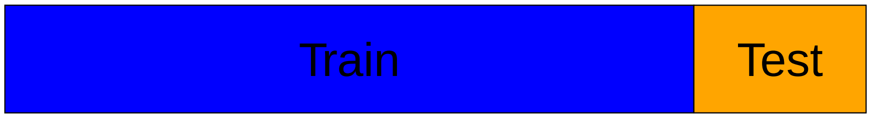
There are two main approaches to estimating the predictive performance (EPE) of a model:

1. Predict on hold-out data.
2. Make a mathematical adjustment to the training error that better estimates the test error.

2.2 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: $\mathcal{D} = (\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}})$.



Repeat the following steps for each tuning parameter combination (or model family):

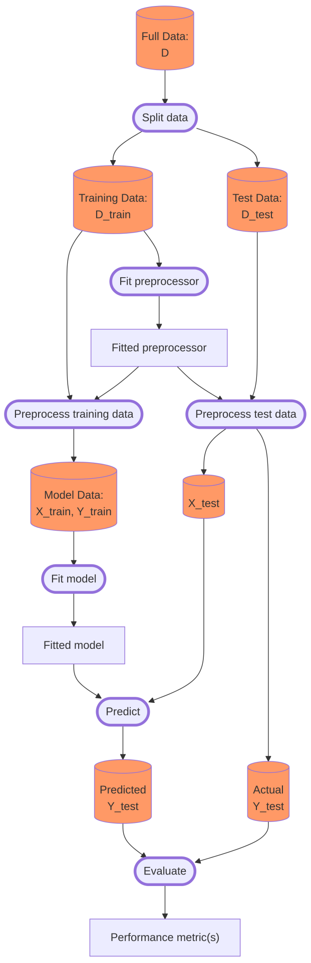
1. Pre-process the training data $\mathcal{D}_{\text{train}}$ so it is suitable for the chosen model (e.g., scaling, log transform). This produces transformed traing data $\mathcal{D}'_{\text{train}}$.
   - Record any parameters were learned/estimated (e.g., column standard deviations, principal component loadings).
2. Fit/Train models using the (possibly transformed) training data $\mathcal{D}'_{\text{train}}$. This estimates the model parameters $\hat{\theta}_\omega = \hat{\theta}_\omega(\mathcal{D}_{\text{train}})$ and produces a prediction function $\hat{f}_\omega(\cdot) = f(\cdot; \hat{\theta}_\omega)$.

- The ω refers to the *tuning parameters*.
- Be sure to include pre-processing steps in the model fitting stage.

3. Apply the same pre-processing steps to the test data $\mathcal{D}_{\text{test}}$ from step 1. Be careful to use the learned parameters from training.
4. Make predictions on the (possibly transformed) test data $\mathcal{D}'_{\text{test}}$.
   - $\hat{y}_j = \hat{f}_\omega(x_j)$ is the prediction for test observation j.
5. Evaluate predictive performance on the test data.

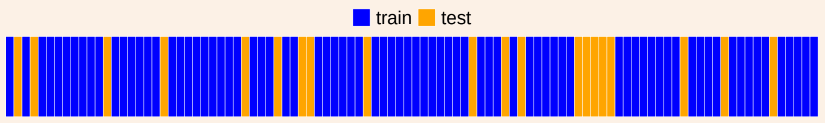
$$\widehat{\text{PE}}(\omega, \mathcal{D}_{\text{test}}) = \frac{1}{m}\sum_{j=1}^{m} \ell(\tilde{y}_j, \hat{f}_\omega(\tilde{x}_j))$$

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



Train/Test Split

a. In practice, the observations should be assigned **randomly** to the train/test sets (not sequentially like in the above figure)
   - This will reduce the influence of potential correlation in the data



b. Caution: all models should be fit and evaluated on the **same training and test split**. Be careful not to evaluate different models on different data!

Your Turn #1

1. How large should the training/test set be?
2. What estimates suffers where there is not enough data in Train Set? Test Set?
3. How can you estimate the uncertainty/variability in the predictive performance metric(s)?

2.2.1 Problems with using a single Hold-Out set

1. Need to decide on how to split the data.
   - 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.

2.2.2 Uncertainty in Predictive Performance

Consider *test* data with m independent observations and a fitted model that makes predictions $\hat{f}(x)$.

The average test error is

$$\text{Avg. Prediction Error} = \widehat{\text{PE}} = \frac{1}{m}\sum_{j=1}^{m} \ell(\tilde{y}_j, \hat{f}(\tilde{x}_j))$$

The $\widehat{\text{PE}}$ is a point estimate of the true EPE. We can use standard statistical tools to assess the variability. E.g.,

- The standard error (SE) is $\hat{\sigma}/\sqrt{m}$, where $\hat{\sigma}$ is the estimated standard deviation of the loss.
- A 95% confidence interval is (approximately) $\widehat{\text{PE}} \pm 2\widehat{\text{SE}}$.

Here is an example with $m = 10$ and absolute error as the loss.

row	y	$\hat{y}$	abs_error
1	6	-0.63	6.63
2	5	3.36	1.64
3	3	2.72	0.28
4	4	5.25	1.25
5	5	3.94	1.06
6	1	4.09	3.09
7	5	3.37	1.63
8	7	4.50	2.50
9	3	4.50	1.50
10	5	-0.70	5.70
—-	—	——	———-
			$\hat{\mu} = 2.53$
			$\hat{\sigma} = 2.07$
			CI $= \hat{\mu} \pm 2 \cdot 2.07/\sqrt{10}$

Bootstrap

An alternative is to use bootstrapping. Resample the test data (with replacement) and recalculate the loss many times. Use quantiles of the bootstrap losses to create the confidence interval.

2.3 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
- Generalized Cross-Validation

2.3.1 AIC/BIC

- AIC/BIC are appropriate when the loss function is a negative log-likelihood (MSE, MAE, log-loss, poisson loss, Brier, etc.)
- Let $\mathcal{M}(\theta)$ represent a model (i.e., model family and tuning parameters) with model parameters θ
- Let $\hat{\mathcal{M}} = \arg\max_\theta \; \log L(\mathcal{M}(\theta))$ be the parameters that maximize the log-likelihood (or penalized log-likelihood).
- $L(\hat{\mathcal{M}})$ is the likelihood of model $\hat{\mathcal{M}}$
  - note: to use AIC/BIC a distributional form 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)

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

Information Criterion (generic)

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

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 $\text{MSE}(\hat{\beta})$ is the *training* MSE and C is a constant that doesn't depend on any model parameters or tuning parameters.
Therefore, the AIC for linear regression is usually reported as:

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

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

2.3.2 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, $\text{RSS}(\bar{y})$ is the residual sum of squares from an intercept only model, and $\text{RSS}(\hat{\beta})$ is the residual sum of squares from the model.

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

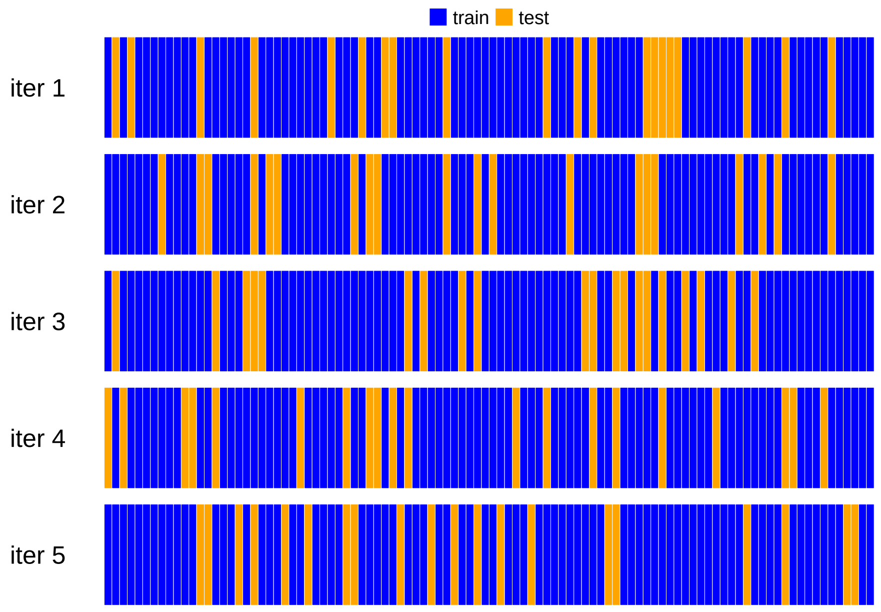
3 Cross-Validation for Model Selection

3.1 Monte Carlo Cross-Validation (Repeated Hold-Outs)

A *single* train/test split may not be ideal due to too much variability:

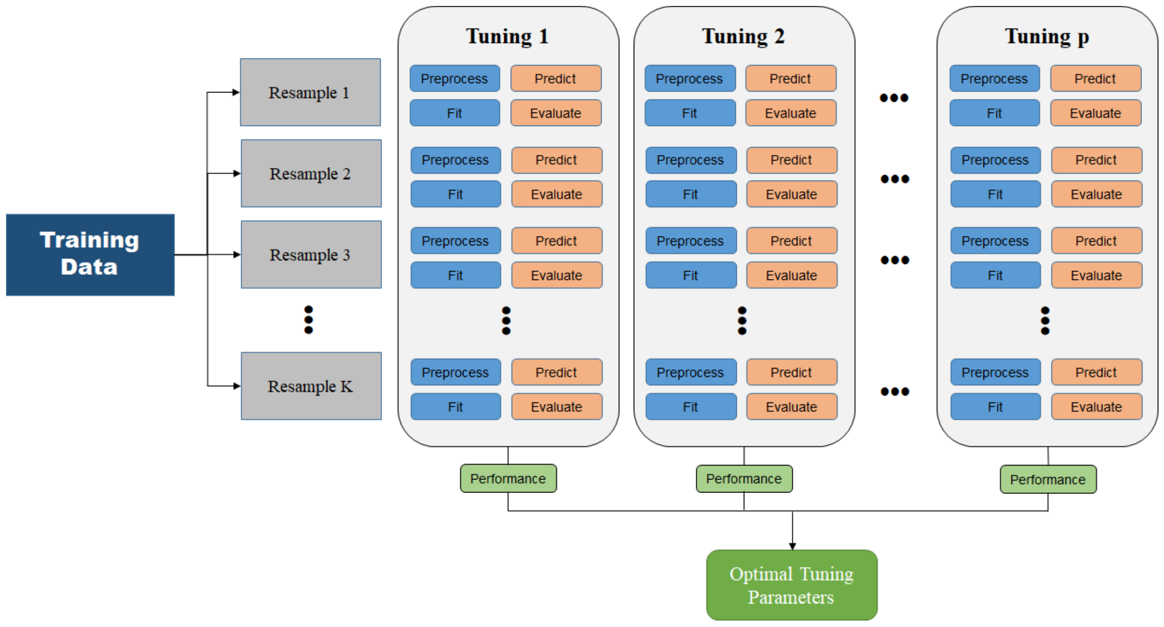
- Too few observations in training set and poor parameter estimates
- Too few observations in test set and poor performance estimates and model selection
- Increased likelihood of choosing a "bad" split.
  - The chance of *bad* split is reduced when $n - m$ and m are both large.

But we can reduce the variability by **repeating the hold-out analysis multiple times**.



Thoughts:

- Can control exactly how much data used for estimating model parameters and how much for model evaluation.
- Can use lots of iterations to reduce variability
- Notice that some observations are used for training (or testing) multiple times, while other observations are never used.

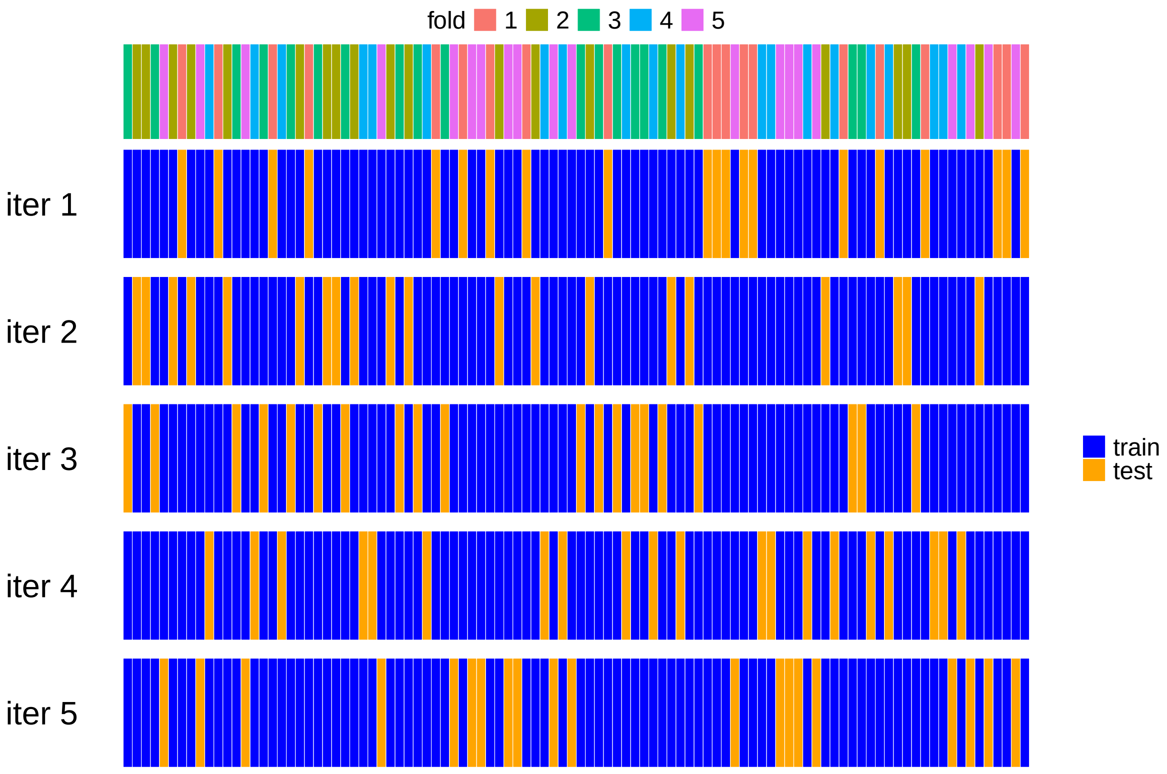

Training Data
Resample 1
Resample 2
Resample 3
Resample K
Tuning 1
Tuning 2
Tuning p
Preprocess
Predict
Fit
Evaluate
Performance
Optimal Tuning Parameters

3.2 K-fold Cross-Validation

K-fold (or V-fold) cross-validation is a special implementation of repeated hold-outs.

- Randomly divide the set of observations into K groups, or folds, of approximately equal size.
- One fold is treated as a validation/test 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.
- Every observation is used exactly $K-1$ times for training and once for evaluation.
  - $(K-1)/K$ proportion of data used for training
  - $1/K$ proportion of data used for evaluation

3.2.1 Example: 5-fold cross-validation



3.3 Cross-Validation Algorithms for Model/Tuning Parameter Selection

Cross-validation techniques create multiple training and test sets and thus multiple estimates of predictive performance. There are a few common approaches to aggregate the performance estimates and choose the best model (or best tuning parameters).

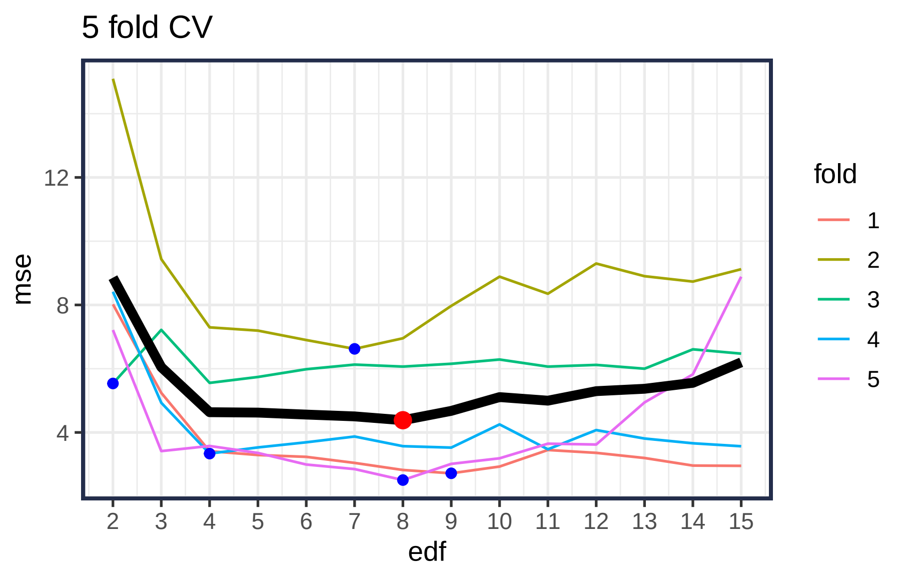
3.3.1 Approach #1: Maximum Average CV Performance

Algorithm: K-fold Cross-Validation

0. Inputs:
    - Set of models or tuning parameters (denote by ω).
    - Data $\mathcal{D}$.
    - Number of folds, K.
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 $\mathcal{D} \setminus \mathcal{F}_k$ (training data) to estimate the model $\hat{f}_\omega^{-k}$
    b. Make predictions for data in $\mathcal{F}_k$ (hold-out data)
    $$\hat{y}_i(\omega) = \hat{f}_\omega^{-k}(x_i) \quad \text{for all } i \in \mathcal{F}_k$$
    c. Calculate the average loss for fold k:
    $$L_k(\omega) = \frac{1}{n_k} \sum_{i \in \mathcal{F}_k} \ell(y_i, \hat{y}_i(\omega))$$
    where n_k is the number of observations in fold k.
3. Model Selection: Choose the tuning parameter(s) (or model) that minimize the cross-validation loss
    $$\hat{\omega} = \arg\min_\omega \text{CV}(\omega)$$
    where $\text{CV}(\omega) = \frac{1}{n} \sum_{k=1}^K n_k L_k(\omega)$.
    - Because the folds are about the same size ($n_k/n \approx 1/K$) you will often see the CV loss replaced by $\text{CV}(\omega) = \frac{1}{K} \sum_{k=1}^K L_k(\omega)$.
4. Final Model: Refit the best model $\hat{\omega}$ using *all data* $\mathcal{D}$ to get the final model.
    - 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 $\{L_k(\omega)\}$!

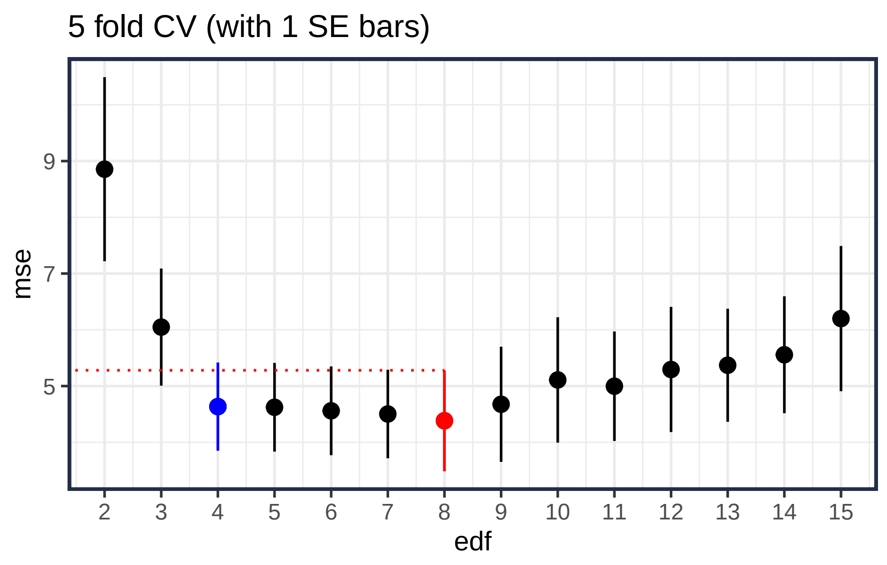
- What is the variability across folds?
- Does performance differ over folds?
- Do some folds cause outliers?



3.3.2 Approach #2: 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 $\text{CV}(\hat{\omega})$.

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



The standard error in the cross-validated performance $\text{CV}(\omega)$ can be estimated in two main ways. The two estimates will not be equal, but should be close.

1. Using the average performance from each fold:

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

2. Using the performance for each observation:

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

where $L_i(\omega) = \ell(y_i, \hat{y}_i(\omega))$.

truth	fold	Model 1	Loss 1	Model 2	Loss 2
y_1	5	$\hat{y}_1^1$	$L_1(1)$	$\hat{y}_1^2$	$L_1(2)$
y_2	2	$\hat{y}_2^1$	$L_2(1)$	$\hat{y}_2^2$	$L_2(2)$
y_3	2	$\hat{y}_3^1$	$L_3(1)$	$\hat{y}_3^2$	$L_3(2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
y_n	9	$\hat{y}_n^1$	$L_n(1)$	$\hat{y}_n^2$	$L_n(2)$

Correlation between folds

These estimates of standard error are too small (they underestimate the uncertainty) because they don't account for the correlation between folds. More specifically, because the training data for any two folds has $(K-2)/(K-1)\%$ overlap, the performance metrics will be correlated.
Because correctly accounting for the correlation is not straightforward, you will often find in practice that the correlation effects are just ignored. However, other approaches like *nested cross-validation* and *mixed/hierarchical anova models* can help reduce the correlation effects.

3.3.3 Approach #3: Mixed-Model/Multilevel-Model Approach

This approach attempts to directly account for the variability due to the folds. The big idea is to model the cross-validation results using a mixed effects (or hierarchical Bayesian) statistical model. Specifically, the model for the loss/error for model m in fold k is:

$$\begin{aligned} L_{km} &= \text{Intercept} + \text{Fold}(k) + \text{Model}(m) + \epsilon_{km} \\ &= \beta_0 + b_k + \sum_m \mathbb{1}(\text{Model} = m)\beta_m + \epsilon_{km} \end{aligned}$$

where b_k is a *random* intercept effect and β_m is the fixed model effect. The details for fitting such models are beyond the scope of this class (but may be included in Bayesian ML), but there are existing R packages that make it easy to implement.

First, let's consider the 5-fold cross-validation results:

fold	df	mse
1	2	8.021
2	2	15.095
3	2	5.536
4	2	8.413
5	2	7.213
1	3	5.242
2	3	9.437

Using approach #1, ignoring correlation between folds, we get estimates (in performance order):

df	n	mean	se	ConfInt_95
8	5	4.38	0.90	[2.59, 6.18]
7	5	4.50	0.79	[2.93, 6.08]
6	5	4.56	0.79	[2.98, 6.14]
5	5	4.62	0.79	[3.05, 6.20]
4	5	4.63	0.78	[3.07, 6.20]
9	5	4.68	1.02	[2.63, 6.72]
11	5	5.00	0.97	[3.05, 6.94]
10	5	5.11	1.11	[2.88, 7.34]
12	5	5.29	1.11	[3.07, 7.52]
13	5	5.37	1.01	[3.36, 7.38]
14	5	5.56	1.04	[3.48, 7.63]
3	5	6.05	1.04	[3.97, 8.13]
15	5	6.20	1.29	[3.62, 8.78]
2	5	8.86	1.64	[5.58, 12.13]

But as mentioned, the standard errors and confidence interval are too narrow because they aren't accounting for correlation between folds.

Because the CV results data is small (only 5 folds and 15 tuning parameters), we can quickly estimate a fully Bayesian model. For this, I'll use the `rstanarm` package using all default priors.

```r
library(rstanarm)
perf_model =
  stan_lmer(
    mse ~ df + (1|fold),
    data = CV_results,
    seed = 12345, # set seed so reproducible
    refresh = 0 # don't print progress
  )
```

df	mean	sd	CredInt_95
8	4.45	1.09	[2.35, 6.62]
7	4.58	1.09	[2.45, 6.80]
6	4.65	1.08	[2.57, 6.91]
5	4.71	1.08	[2.67, 6.91]
4	4.72	1.08	[2.64, 6.91]
9	4.75	1.09	[2.63, 7.07]
11	5.08	1.07	[3.03, 7.26]
10	5.19	1.08	[3.12, 7.48]
12	5.39	1.09	[3.32, 7.60]
13	5.46	1.09	[3.33, 7.74]
14	5.64	1.08	[3.55, 7.88]
3	6.13	1.09	[4.01, 8.41]
15	6.28	1.07	[4.17, 8.56]

Notice that the standard deviation and credible intervals are wider than the estimates from the basic approach.

Tidymodels package ‘tidyposterior‘

The `tidyposterior` function `perf_mod()` makes it even easier to implement these and choose the best predictive model from cross-validation. More details are provided in Tidy Modeling with R Chapter 11 and code examples in the appendix of these notes.

3.4 Choice of K in K-fold cross-validation

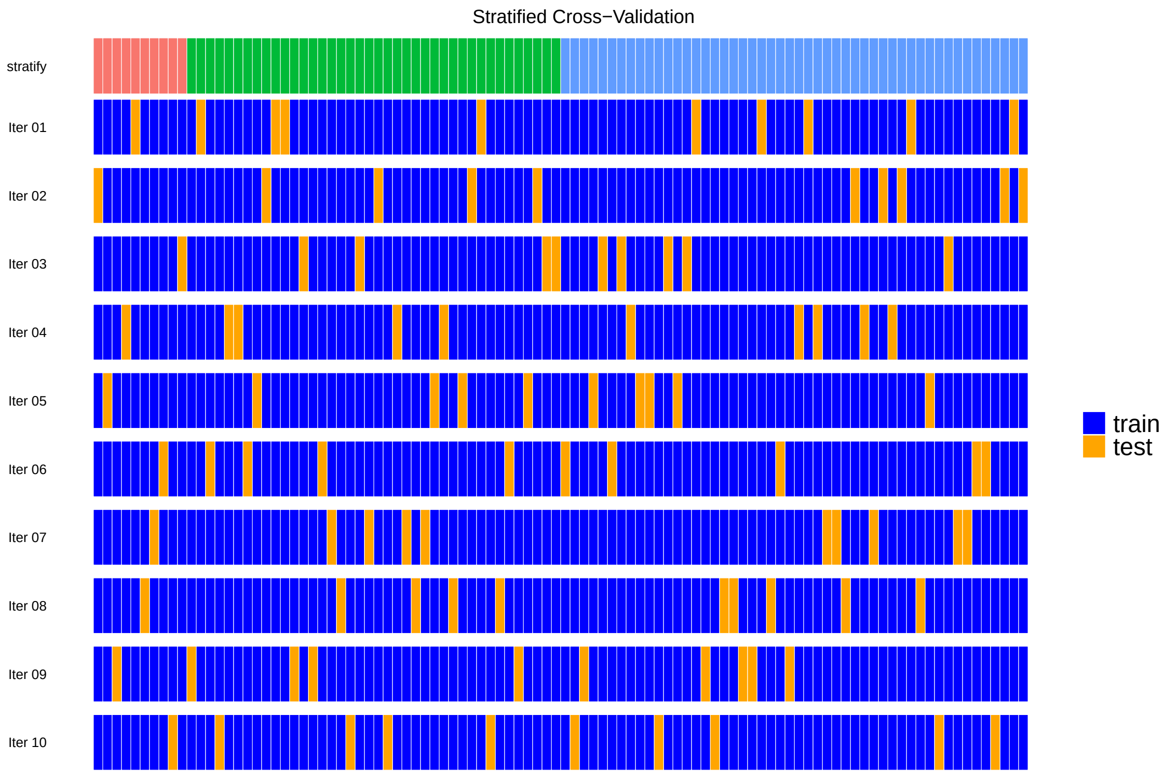
- $K = 5, 10, n$ are common choices
- $K = n$ is called *leave-one-out (LOOCV)*
  - There exists a closed form solution for models that are linear in y

3.4.1 Performance Bias/Variance Trade-off

- Around $(K-1)/K$ of the data is used for training and $1/K$ for testing.
  - Calculate the size of training $n(K-1)/K$ and size of testing n/K. **Remember uncertainty is a function of sample size, not proportions.**
- 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. Note special exceptions possible for $K = n$ with some models (e.g., linear predictors).
- Note: A single run of K-fold cross-validation will still have high uncertainty. Consider *repeated* cross-validation instead.

3.5 Stratified/Balanced Cross-validation

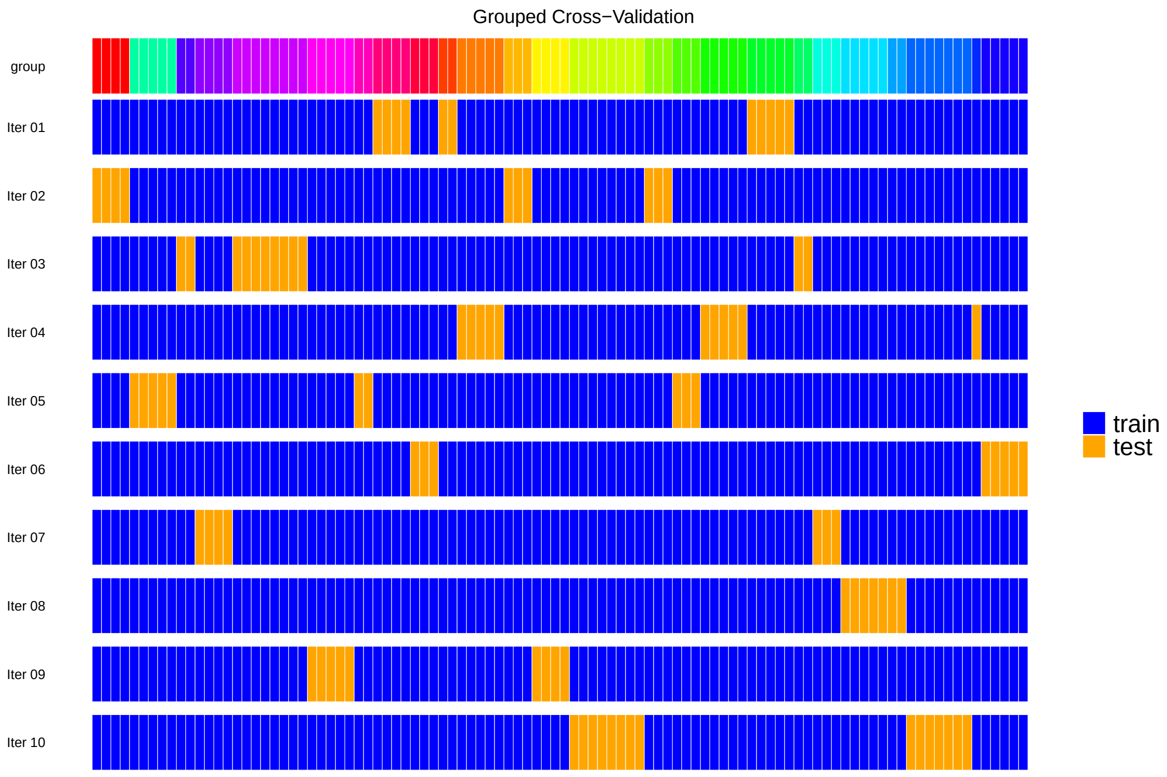
- The most basic approach is to use random sampling to assign observations to training and test sets (or folds). However due to the random assignment, the folds will probably not have the same empirical distribution of outcomes or predictors.
- But this can be problematic if there are outliers or categorical variables with small frequency.
  - This is especially a problem for categorical data. If some levels of a predictor variable are not included in training set, the model won't know how to predict if they appear in the test set.
- Stratified sampling can be used to force similar empirical distributions in each fold
  - e.g., equal number of observations in each fold from same quartile of y
  - e.g., equal number of observations in each fold from the same categorical y
  - e.g., equal number of observations in each fold from same race/ethnicity
  - e.g., cluster the training data and assign observations into folds such that an equal number of observations from each cluster are in each fold
- Stratify on outcome variable(s) or predictors (or both).



3.6 Grouped Cross-Validation

Sometimes the data will consist of *grouped observations*. All observations from the same group should be in the same fold. A common example of grouping is repeated measures, for example data collected on the same individual. We want all the data for the same individual either in the training set or test set, but not scattered across both.

This approach may produce an unequal number of observations in each fold (or hold-out set).



3.7 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.

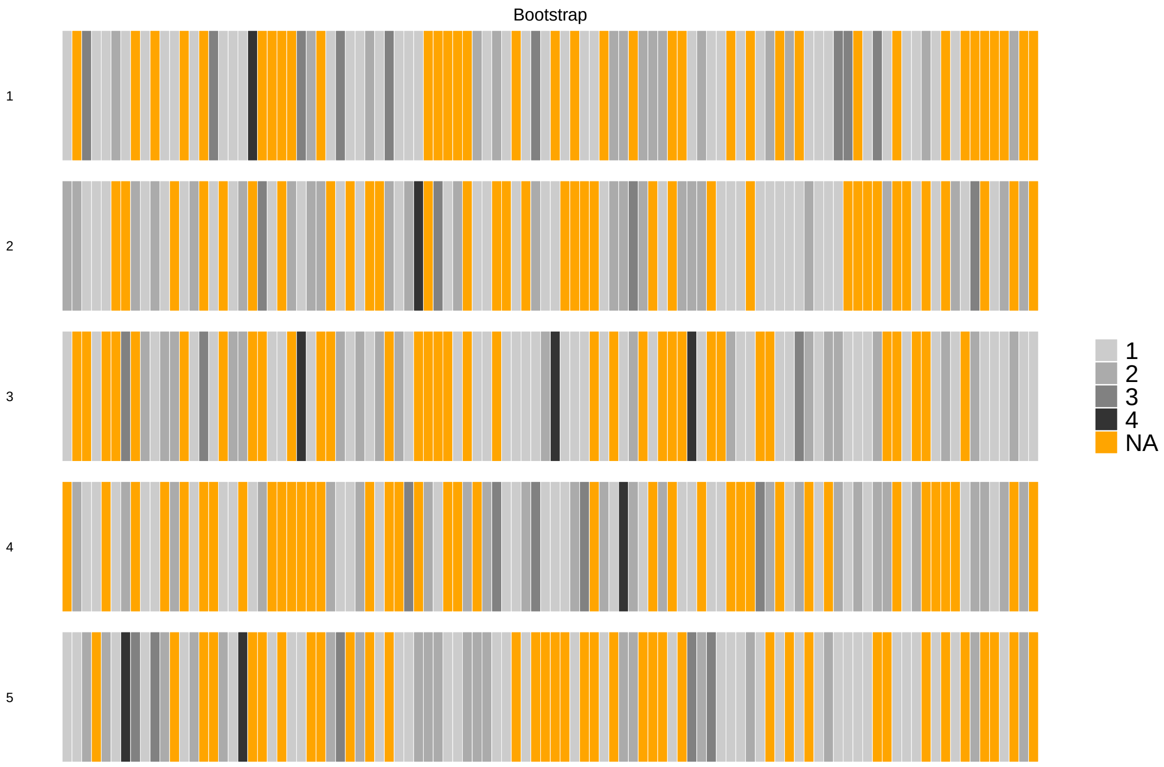
- if you repeat K-fold cross-validation 5 times, then you will need to fit each model $5K$ times

3.7.1 Monte Carlo CV (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 repeated cross-validation, but give much more control over the size and number of experiments.
  - holding out 20% of the data and repeating 5 times is comparable to $K = 5$ fold cross-validation.
  - but we could also hold out 20% of the data and repeating 8 times, a scenario that isn't possible with k-fold cross-validation.
- Stratification and grouping are also possible with MC-CV.

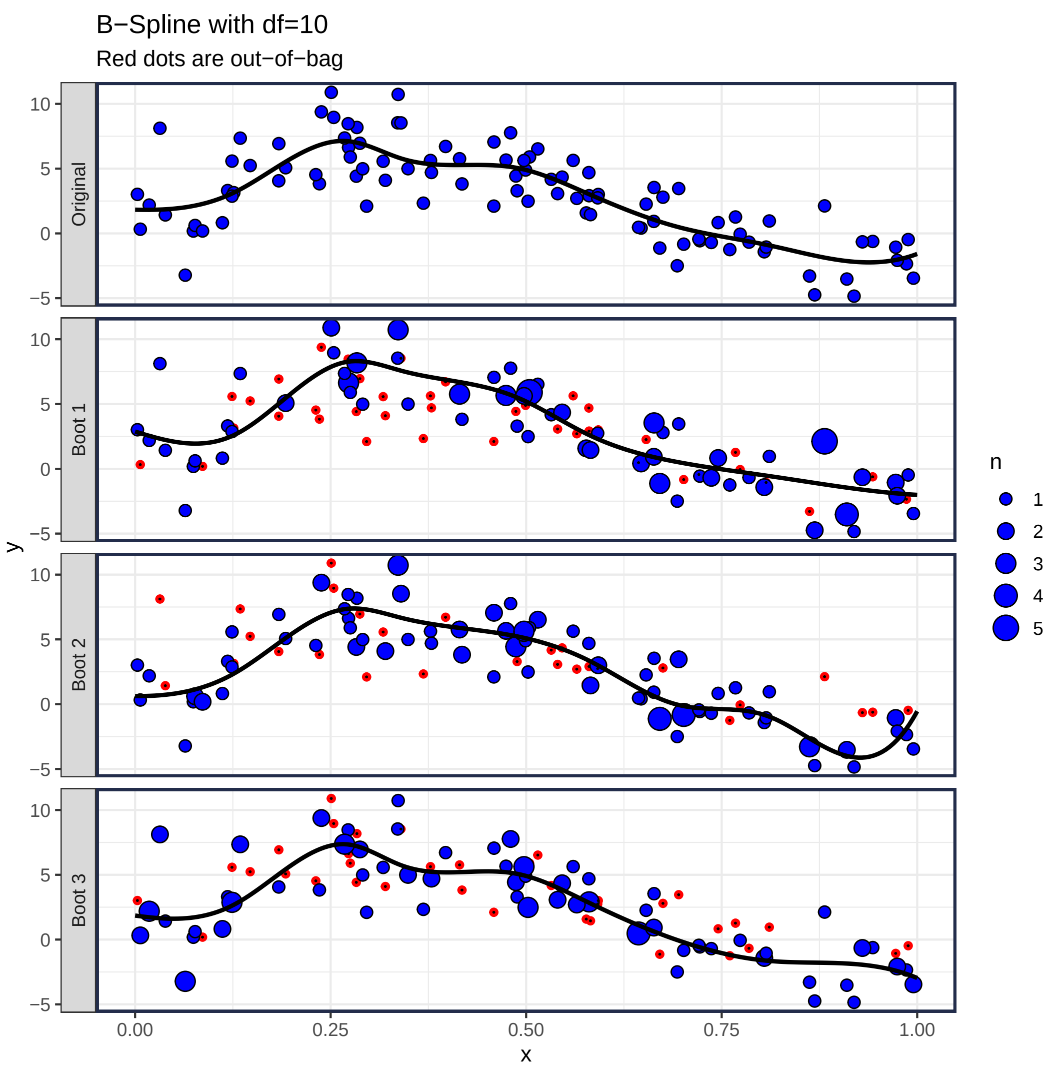
3.8 Bootstrap Out-of-Bag



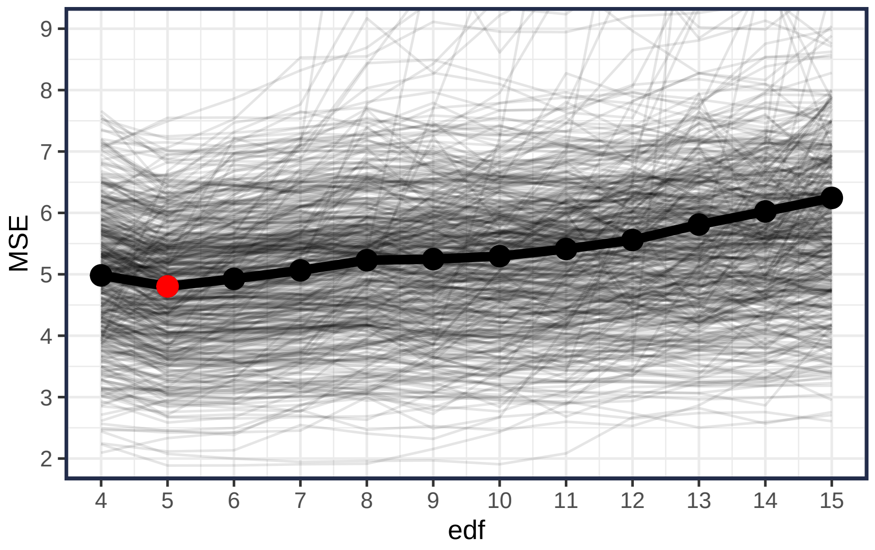
Because about 37% of the bootstrap data will be used for model assessment (testing data), it is *similar* to $K = 3$ cross-validation, if repeated 3 times. Some differences:

- Use n observations for model fitting (instead of $2/3 \cdot n$)
- Can repeat the bootstrap more than 3 times to reduce variability.

Let's look at a few bootstrap fits:



- Notice that each bootstrap sample does not include about 37% of the original observations.
- These are called *out-of-bag* samples and can be used to assess model fit
  - The out-of-bag observations were not used to estimate the model parameters, so will be sensitive to over/under fitting
- Below, we evaluate the oob error over the spline complexity (df = number of estimated coefficients)

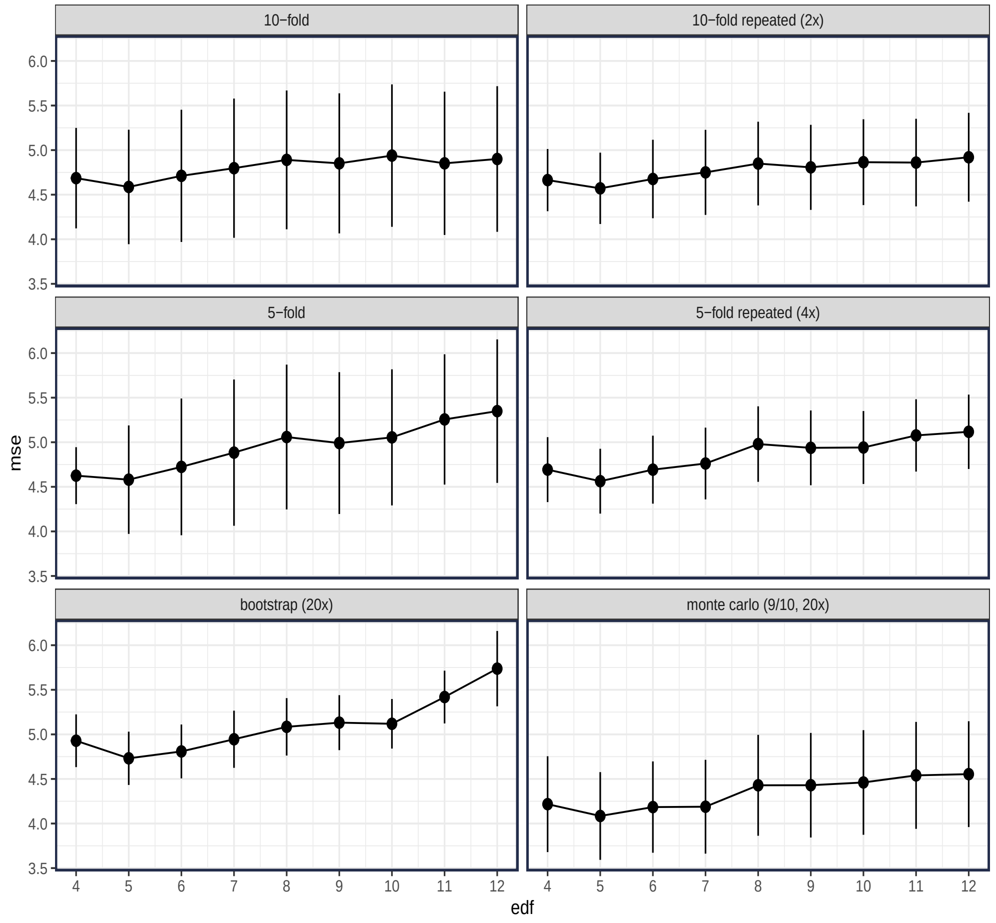


3.9 Prevent data leakage

Be careful to ensure that all aspects of estimation (e.g., pre-processing, variable selection, tuning parameter selection) are done **inside** the cross-validation for each training set.

3.10 Resampling Comparison

Here, we example the variability of a few resampling methods.



The bars are 1 standard error.

3.11 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*

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

  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 Model Assessment

We have been concerned with **model selection**, which means choosing the best model (or tuning parameter(s)). But let's turn our attention to **model assessment**, which involves understanding how well our chosen model(s) will perform on new, unseen data (i.e., estimating the EPE).

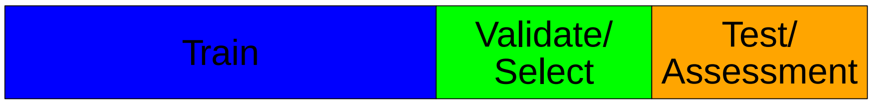
While the results from cross-validation (K-fold and Monte Carlo) do give some sense of performance it suffers from some issues:

1. It could be biased
   - can be too optimistic due to using CV to pick best tuning parameter or model.
   - for small samples, can be too pessimistic due to fitting models with $< n$ data.
2. There is still uncertainty/variance (especially for single cross-validation).
3. The new data is not from the same distribution as the training data.

One good way to check how well the selected model(s) perform on new data is to use new data to evaluate it. Consider the original version of this idea:

4.1 Train/Validate/Test

If it were possible to have lots of data (all from the same distribution), then we could 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.
- Refit optimal model using all available data (Training and Validation).
  - Then the performance of the best/final model can be assessed (without bias)

4.2 Resampling Based Model Assessment

Here is a resampling based improvement on this concept:

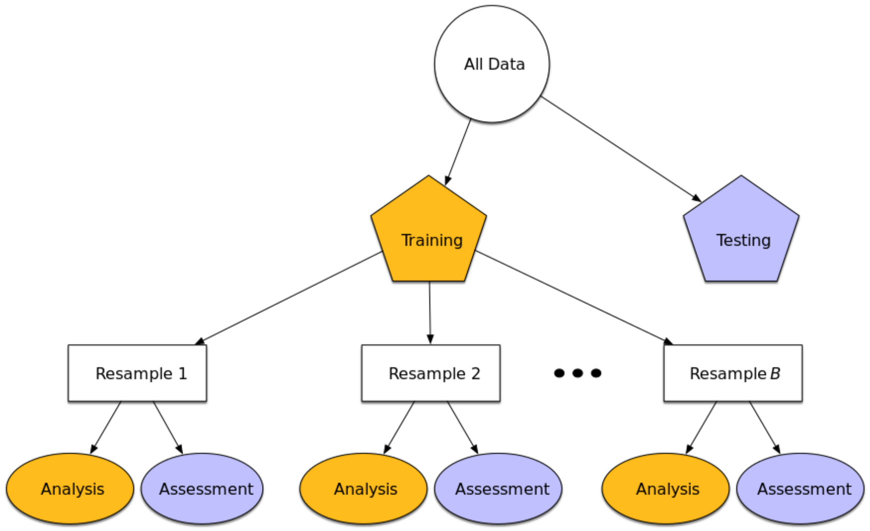
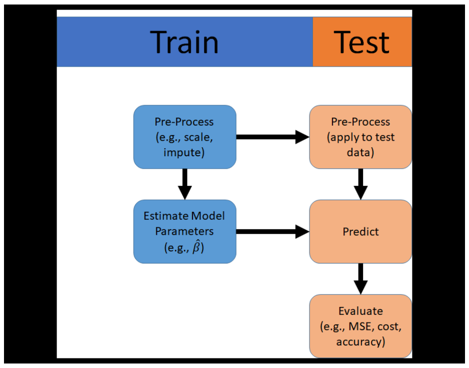
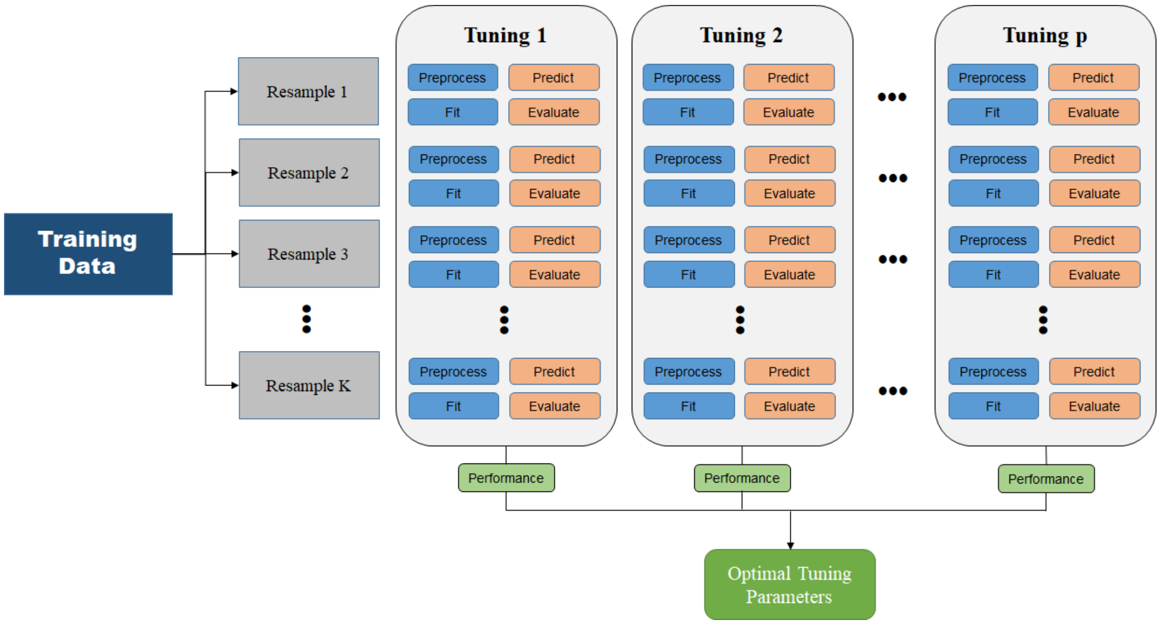


Figure taken from: **Feature Engineering and Selection: A Practical Approach for Predictive Models** by Max Kuhn and Kjell Johnson (2019-06-21)

If you don't have enough data to be confident about your test set performance, you could create a *nested* cross-validation: an outer loop over the initial train/test split to evaluate final performance; with an inner cross-validation loop to select the best model for each training set.


Train
Test
Pre-Process (e.g., scale, impute)
Pre-Process (apply to test data)
Estimate Model Parameters (e.g., β)
Predict
Evaluate (e.g., MSE, cost, accuracy)


Training Data
Resample 1
Resample 2
Resample 3
Resample K
Tuning 1
Tuning 2
Tuning p
Preprocess
Predict
Fit
Evaluate
Performance
Optimal Tuning Parameters

5 Appendix: R Code

```
library(splines2)   # for fast splines
library(tidyverse)  # for data manipulation
library(tidymodels) # for optional tidymodels approaches
```

5.1 Simulate Data

```
#-- Data Generation
generate_x <- function(n) runif(n)       # U[0,1]
generate_y <- function(x, sd = 2){
  n = length(x)
f <- function(x) 1 + 2*x + 5*sin(5*x)    # true mean function
f(x) + rnorm(n, sd=sd)
}

#- Training data
n_train = 200                            # number of observations
set.seed(825)                            # set seed for reproducibility
x = generate_x(n_train)                  # get x values
y = generate_y(x)                        # get y values
data_train = tibble(x,y)                 # training data

#- Test data
n_test = 50                              # number of observations
set.seed(825)                            # set seed for reproducibility
x = generate_x(n_test)                   # get x values
y = generate_y(x)                        # get y values
data_test = tibble(x,y)                  # test data
```

5.2 Model Set-up

We are going to evaluate the choice of B-spline tuning parameter $df \in \{3, 4, \ldots\}$ using a variety of re-sampling methods. The `df` parameter in `bsp()` controls the number of B-splines (basis functions) and hence the number of estimated parameters (edf). In the formulation below, I'll use cubic B-splines (degree = 3) where the knots are selected from the quantiles of the training data.

We will use the `bsp()` function from the `splines2` package for the model. I'll make a function `sp_eval()` 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.

```
library(splines2)  # for the bsp() function
library(dplyr)    # for tibble() and bind_rows() functions

# sp_eval(): fit set of B-spline models and evaluate on test data
#-------------------------------------------------------------------------
# data_fit, data_eval: training and test data (requires column names x,y)
# df: set of spline degrees (tuning parameters)
# kts.bdry: boundary knots for the splines (to help extrapolate)
# output: tibble with df and associated mean squared error (MSE) on test data
sp_eval <- function(data_fit, data_eval, df = seq(3, 15, by=1), kts_bdry = c(-.1, 1.1)) {
    MSE = numeric(length(df)) # initialize
    for(i in 1:length(df)) {
    # set tuning parameter value
    df_i = df[i]
    # fit with training data (no intercept)
    fmla = formula(y~splines2::bsp(x, df = df_i, degree = 3,
                                   Boundary.knots = kts_bdry) - 1)
    fit = lm(fmla, data = data_fit)
    # predict on test data
    yhat = predict(fit, data_eval)
    # get errors / loss
    MSE[i] = mean( (data_eval$y - yhat)^2 )
  }
tibble(df, mse=MSE) # output
}
```

5.3 Single train/test split

Set the number of hold-out observations and random seed.

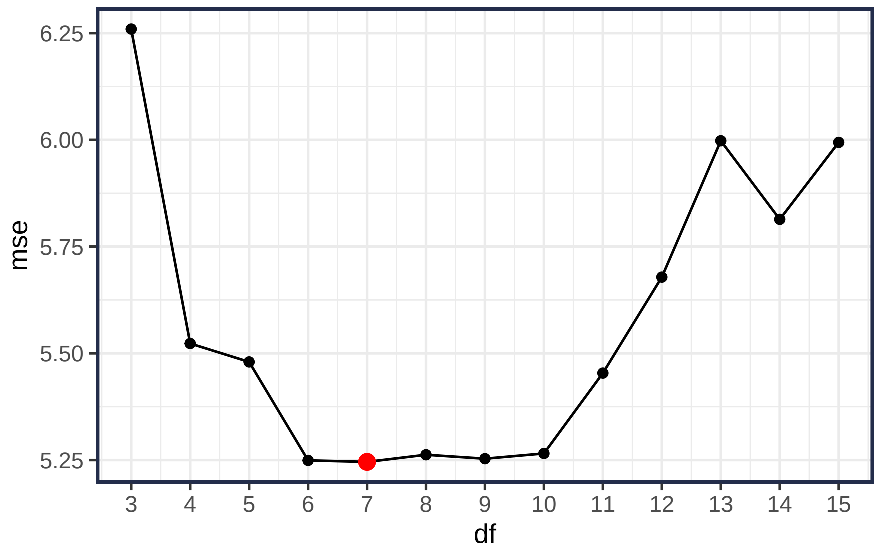
```
n_holdout = 20                        # size of hold-out set
set.seed(2021)                        # set seed for reproducibility
holdout = sample(nrow(data_train), size=n_holdout) # indices to include in holdout set
```

Evaluate the performance (over each tuning parameter)

```
results = sp_eval(
        data_fit = slice(data_train, -holdout),
        data_eval = slice(data_train, holdout)
        )
```

This visualizes the results showing that $\hat{df} = 7$ is the optimal value for this train/test split.

```
ggplot(results, aes(df, mse)) +
  geom_point() + geom_line() +
  geom_point(data = . %>% slice_min(mse, n=1), color="red", size=3) +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```



5.3.1 Using `rsample`

The `initial_split` function from the `rsample` package will a single train/test split:

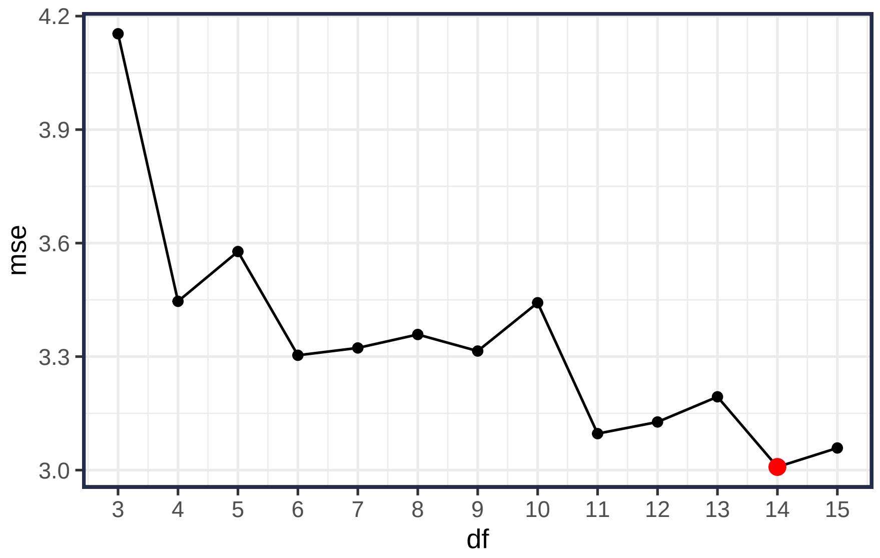
```
n_holdout = 20                                   # size of hold-out set
set.seed(2021)                                   # set seed for reproducibility
data_mc = initial_split(data_train, prop = 1 - n_holdout/nrow(data_train))
```

Evaluate the performance over each tuning parameter

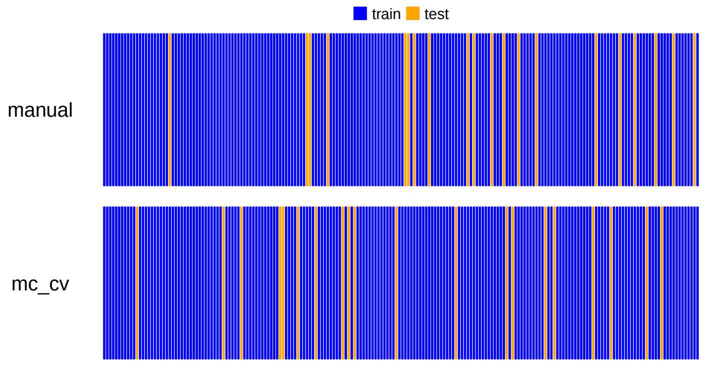
```
results2 = sp_eval(
        data_fit = training(data_mc),
        data_eval = testing(data_mc)
        )
```

This visualizes the results showing that $\hat{df} = 14$ is the optimal value for this train/test split.

```
ggplot(results2, aes(df, mse)) +
  geom_point() + geom_line() +
  geom_point(data = . %>% slice_min(mse, n=1), color="red", size=3) +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```



Note that we have a different result, even with same seed, because the `mc_cv()` function uses a different method to select holdout data.



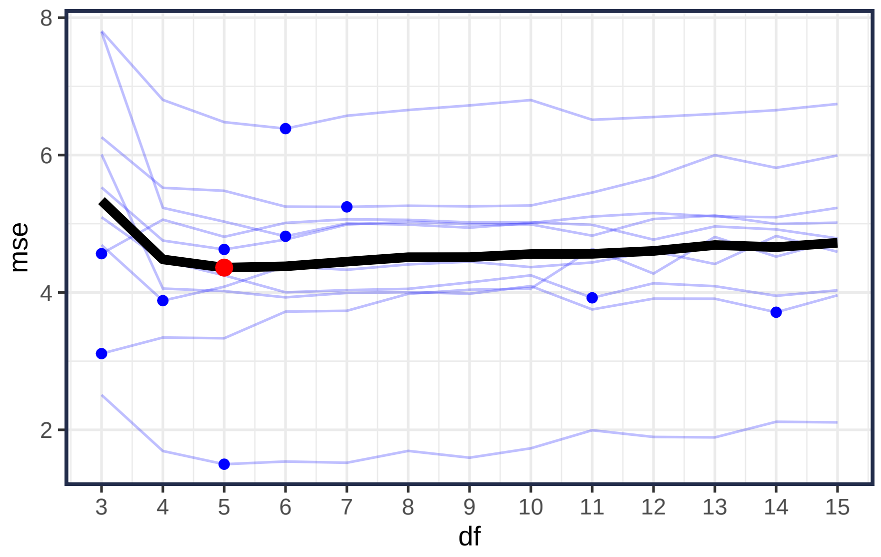
5.4 Repeated train/test splits (Monte Carlo Cross-Validation)

Now, let's repeat this procedure $M = 10$ times:

```
n_holdout = 20        # size of hold-out set
M = 10                # number of repeats
set.seed(2021)        # set seed for reproducibility

#-- Repeat the train/test split M times
results = vector("list", M)

for(m in 1:M) {
  #- select hold out observations
  holdout = sample(n_train, size=n_holdout) # indices to include in holdout set
  #- fit and evaluate models
  results[[m]] = sp_eval(
        data_fit = slice(data_train, -holdout),
        data_eval = slice(data_train, holdout)
        ) %>%
    mutate(iter = m)  # add iteration number
}
RESULTS = bind_rows(results)
```

Here is the visualization of the results. There is one line for each repetition (colored blue). The solid black line is the mean.

```
ggplot(RESULTS, aes(df, mse)) +
  geom_line(aes(group = iter), color = "blue", alpha=.25) +
  geom_point(data=. %>% group_by(iter) %>% slice_min(mse, n=1), color="blue") +
  geom_line(data = . %>% group_by(df) %>% summarize(mse = mean(mse)),
            linewidth=2) +
  geom_point(data = . %>% group_by(df) %>% summarize(mse = mean(mse)) %>%
               slice_min(mse, n=1), size=3, color="red") +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```

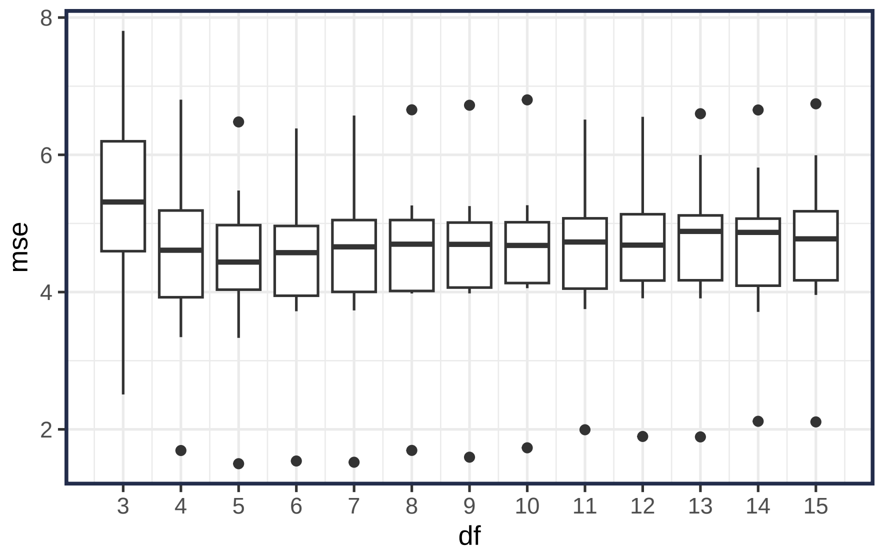


This suggests that $\hat{df} = 5$.

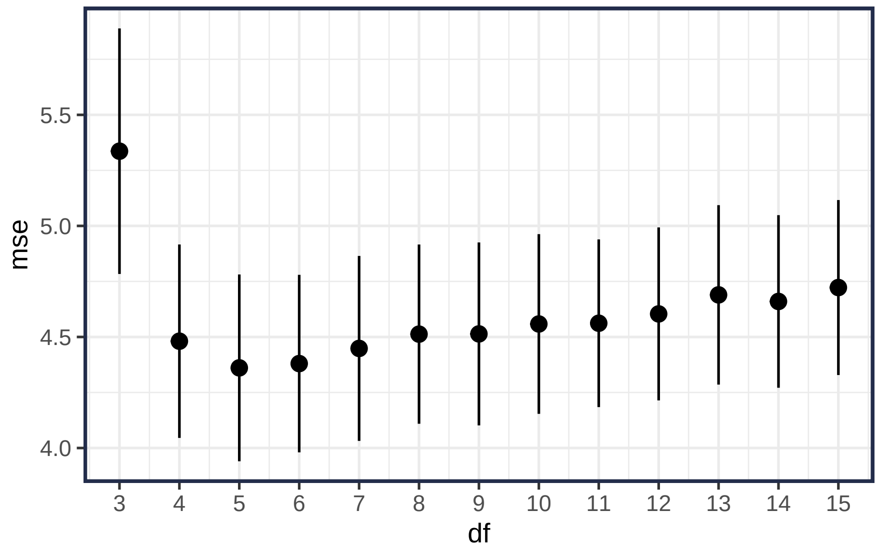
Notice that the iterations have different optimal df. Usually the average MSE is used to determine the optimal tuning parameters, but analysis of the individual optimal may be valuable at times.

You'll often see this information represented with boxplots or standard errorbar plots:

```r
#- Boxplots
RESULTS %>%
  ggplot(aes(df, mse, group=df)) + geom_boxplot() +
  scale_x_continuous(breaks = 3:15)
```



```r
#-- 1 standard error errorbars
RESULTS %>%
  group_by(df) %>%
  summarize(sd = sd(mse), mse = mean(mse), n = n(), se = sd/sqrt(n)) %>%
  ggplot(aes(df, mse)) + geom_pointrange(aes(ymin=mse-se, ymax=mse+se)) +
  scale_x_continuous(breaks = 3:15)
```



5.4.1 Using `rsample`

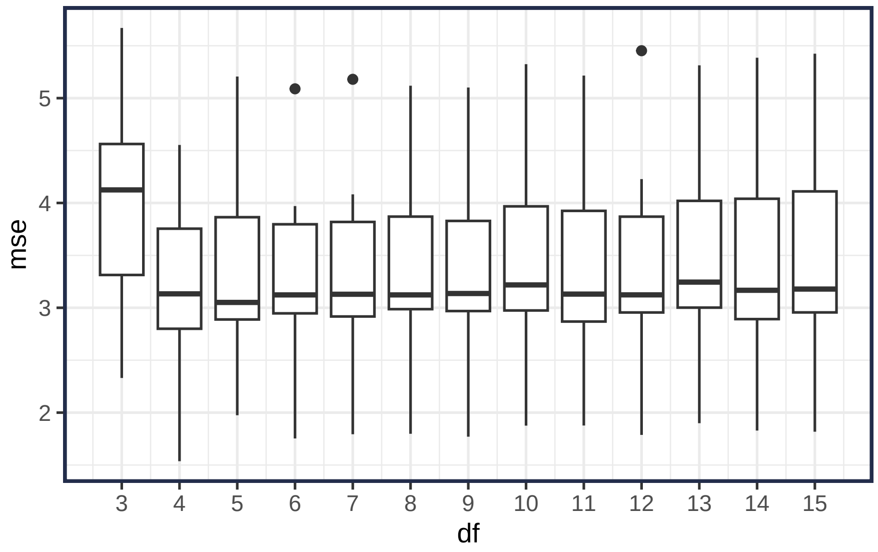
The `mc_cv` function from the `rsample` package will generate a monte carlo cross-validation.

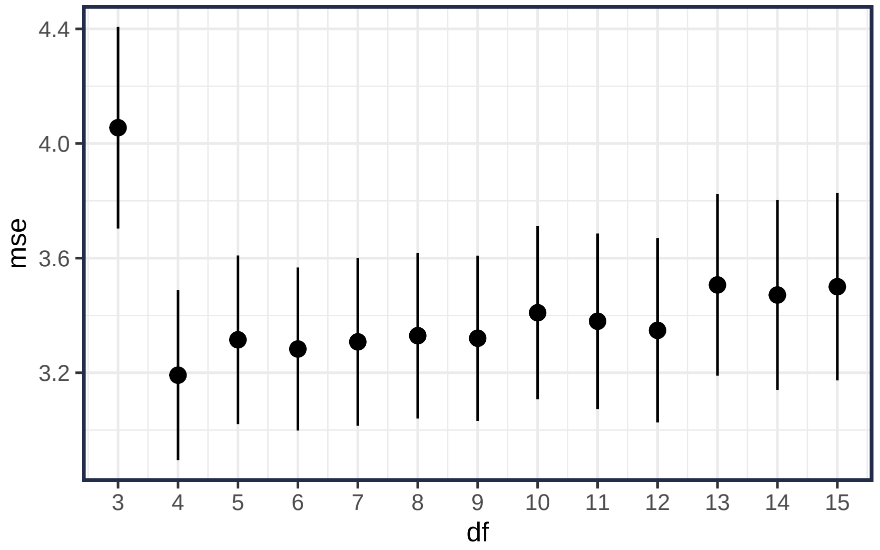
```
n_holdout = 20                      # size of hold-out set
M = 10                              # number of repeats
set.seed(2021)                      # set seed for reproducibility
data_mc = mc_cv(data_train,
                prop = 1 - n_holdout/n_train,
                times = M)
```

Evaluate the performance over each tuning parameter (`DF`)

```
RESULTS2 = map_df(
            .x = data_mc$splits,
            .f = ~sp_eval(
                    data_fit = training(.x),
                    data_eval = testing(.x)
                  ),
            .id = "iter"
        )
```

```
#- Boxplots
RESULTS2 %>%
  ggplot(aes(df, mse, group=df)) + geom_boxplot() +
  scale_x_continuous(breaks = 3:15)

#- 1 SE bars
RESULTS2 %>%
  group_by(df) %>%
  summarize(sd = sd(mse), mse = mean(mse), n = n(), se = sd/sqrt(n)) %>%
  ggplot(aes(df, mse)) + geom_pointrange(aes(ymin=mse-se, ymax=mse+se)) +
  scale_x_continuous(breaks = 3:15)
```





5.5 K-Fold Cross-Validation

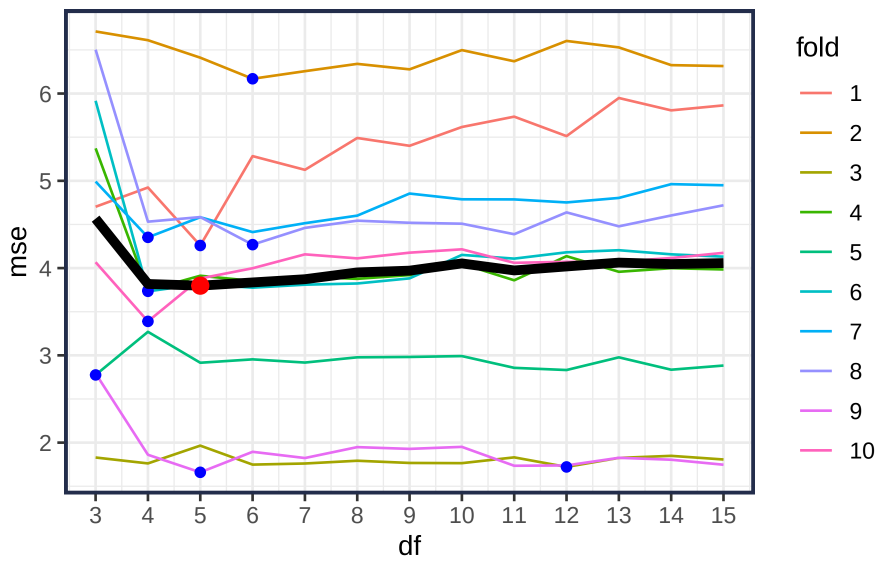
K-fold (or sometime called V-fold) cross-validation is just a special case of Monte Carlo CV where each observation is used for training K-1 times and in evaluation exactly once. To get folds of almost equal size I use the function `rep(1:n.folds, length=n)` and then shuffle (using `sample()`) to further limit potential correlation effects.

```r
#- Get K-fold partition
n.folds = 10          # number of folds for cross-validation
set.seed(2021)         # set seed for reproducibility
fold = sample(rep(1:n.folds, length=n_train))  # vector of fold labels
# notice how this is different than:  sample(1:K,n,replace=TRUE),
#  which won't necessarily give almost equal group sizes

results = vector("list", n.folds)
#- Iterate over folds
for(j in 1:n.folds){

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

  #- fit and evaluate models
  results[[j]] = sp_eval(
        data_fit = slice(data_train, train),
        data_eval = slice(data_train, val)
        ) %>%
    mutate(fold = j, n.val)  # add fold number and number in validation
}
RESULTS = bind_rows(results)
```

Notice that this is almost identical to the repeated hold-out process. The only difference is in how the train and hold-out sets are determined.

Here is the visualization of the results. There is one line for each repetition (colored blue). The solid black line is the mean.

```
RESULTS %>%
  mutate(fold = factor(fold)) %>%
  ggplot(aes(df, mse)) +
  geom_line(aes(color=fold)) +
  geom_point(data=. %>% group_by(fold) %>% slice_min(mse, n=1), color="blue") +
  geom_line(data = . %>% group_by(df) %>% summarize(mse = mean(mse)), linewidth=2) +
  geom_point(data = . %>% group_by(df) %>% summarize(mse = mean(mse)) %>%
               slice_min(mse, n=1), size=3, color="red") +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```



This suggests that $\hat{df} = 5$.

There is still much variability in the mean mse. This could be reduced by repeated cross-validation.

5.5.1 Using `rsample`

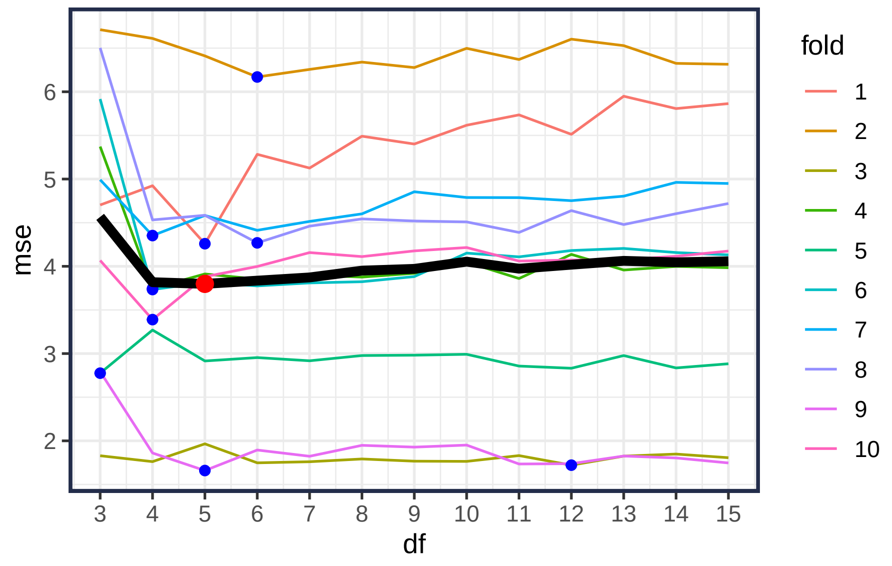
The `vfold_cv` function from the `rsample` package will generate v-fold cross-validation.

```
n_folds = 10          # number of folds for cross-validation
set.seed(2021)                        # set seed for reproducibility
data_cv = vfold_cv(data_train, v = n_folds)
```

Evaluate the performance over each tuning parameter (`DF`)

```
DF = seq(3, 10, by=1)      # set complexity (edf) values
RESULTS2 = map_df(data_cv$splits, ~sp_eval(
        data_fit = training(.x),
        data_eval = testing(.x)),
        .id = "fold"
        )
```

```
RESULTS2 %>%
  mutate(fold = factor(fold, 1:n_folds)) %>%
  ggplot(aes(df, mse)) +
  geom_line(aes(color=fold)) +
  geom_point(data=. %>% group_by(fold) %>% slice_min(mse, n=1), color="blue") +
  geom_line(data = . %>% group_by(df) %>% summarize(mse = mean(mse)), linewidth=2) +
  geom_point(data = . %>% group_by(df) %>% summarize(mse = mean(mse)) %>%
               slice_min(mse, n=1), size=3, color="red") +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```



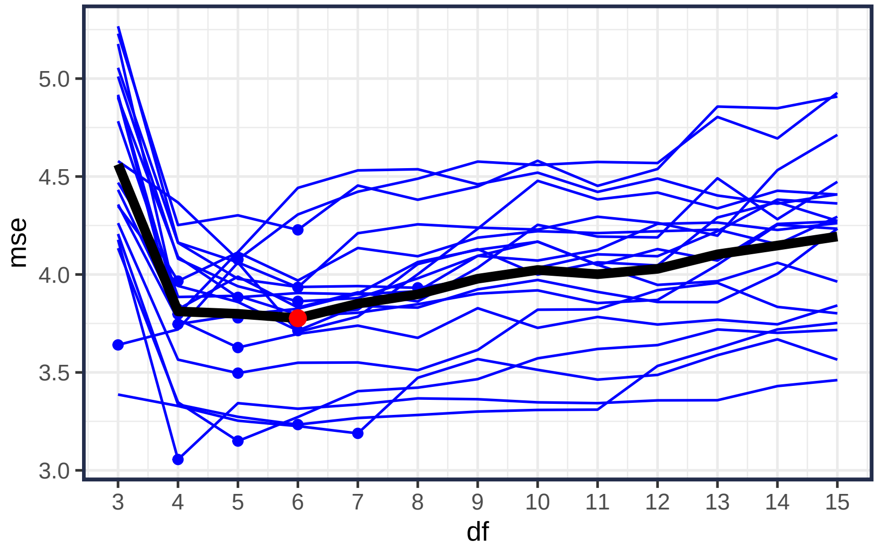
5.6 Out-of-Bag

We'll use $M = 20$ bootstraps:

```
n = nrow(data_train) # number of training observations
M = 20               # number of bootstraps
set.seed(2021)       # set seed for reproducibility

#-- Repeat the train/test split M times
RESULTS = vector("list", M)

for(m in 1:M) {

  #- bootstrap sampling
  boot = sample(n, size=n, replace=TRUE) # indices in bootstrap sample
  oob = setdiff(1:n, boot)               # out-of-bag indices

  #- fit and evaluate models
  results[[m]] = sp_eval(
        data_fit = slice(data_train, boot),
        data_eval = slice(data_train, oob)
        ) %>%
    mutate(iter = m, n.oob = length(oob))  # add fold number and number in oob
}
RESULTS = bind_rows(results)
```

Here is the visualization of the results. There is one line for each repetition (colored blue). The solid black line is the mean.

```
RESULTS %>%
  ggplot(aes(df, mse)) +
  geom_line(aes(group = iter), color="blue") +
  geom_point(data=. %>% group_by(iter) %>% slice_min(mse, n=1), color="blue") +
  geom_line(data = . %>% group_by(df) %>% summarize(mse = mean(mse)), size=2) +
  geom_point(data = . %>% group_by(df) %>% summarize(mse = mean(mse)) %>%
                slice_min(mse, n=1), size=3, color="red") +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```

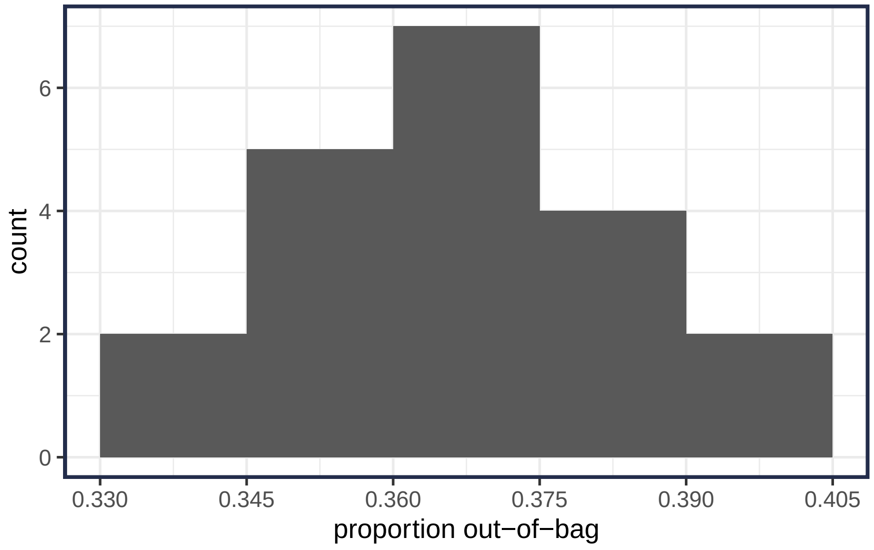


This suggests that $\hat{df} = 6$.

Notice that the bootstrap procedure uses a sample size of n to estimate the model parameters. The other approaches use a smaller sample size.

This procedure also uses a hold-out size of around 37%. Compare this with 10-fold cross-validation which only used 10% of the data for evaluation.

```
RESULTS %>% distinct(iter, n.oob) %>% mutate(p.oob = n.oob/n) %>%
  ggplot(aes(p.oob)) + geom_histogram(binwidth = .015, boundary=0) +
  scale_x_continuous(breaks = seq(0, 1, by=.015)) +
  labs(x= "proportion out-of-bag")
```



5.6.1 Using `rsample`

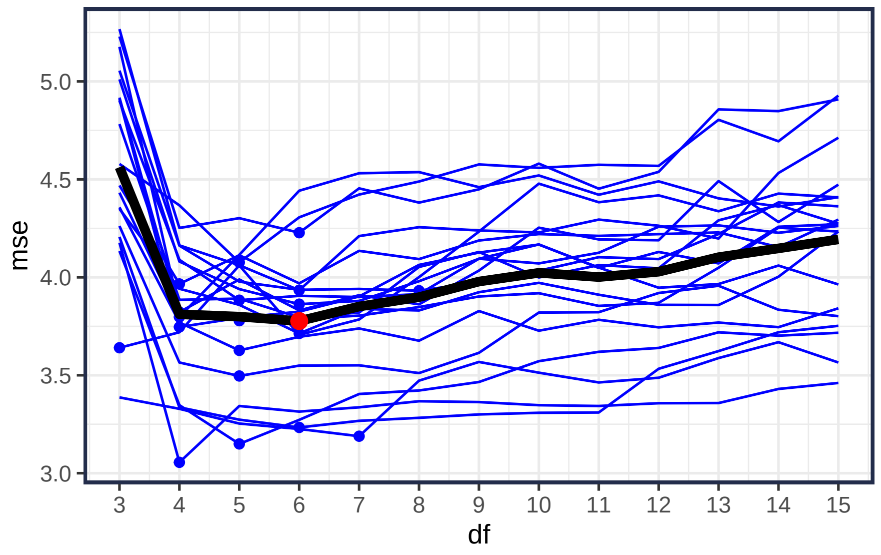
The `bootstraps()` function from the `rsample` package will generate bootstrap samples.

```
M = 20                    # number of bootstraps
set.seed(2021)                               # set seed for reproducibility
data_boot = bootstraps(data_train, times = M)
```

Evaluate the performance over each tuning parameter

```
RESULTS2 = map_df(data_boot$splits, ~sp_eval(
        data_fit = training(.x),
        data_eval = testing(.x)),
        .id = "iter"
        )
```

```
RESULTS2 %>%
  ggplot(aes(df, mse)) +
  geom_line(aes(group = iter), color="blue") +
  geom_point(data=. %>% group_by(iter) %>% slice_min(mse, n=1), color="blue") +
  geom_line(data = . %>% group_by(df) %>% summarize(mse = mean(mse)), size=2) +
  geom_point(data = . %>% group_by(df) %>% summarize(mse = mean(mse)) %>%
               slice_min(mse, n=1), size=3, color="red") +
  scale_x_continuous(breaks = seq(0, 20, by=1))
```



5.7 Evaluation on the test data

There are 200 training observations. For a model with edf = 7 there are only 28.6 observations per model parameter - which is not a lot. This suggests to me, and the plots confirm, that there is still much uncertainty in both the optimal tuning parameter (*df*) and the expected prediction error (EPE). For estimating *df*, I would repeat one of the approaches (cross-validation, monte carlo cross validation, out of bag) many more times to reduce the variability in the results.

This would also give an estimate of the EPE, but with bias. Once the optimal *df* is selected (`df_opt`), then a final model is fit, using all the data and predictions for the *test data* are made.

```
df_opt = 5  # best tuning parameter from cross-validation

sp_eval(data_fit = data_train,   # use all training data
        data_eval = data_test,   # predict on test data
        df = df_opt)             # use optimal tuning parameter(s)
#> # A tibble: 1 x 2
#>      df   mse
#>   <dbl> <dbl>
#> 1     5  5.05
```

The error on the test data, which hasn't been used in any way yet, will provide an unbiased estimate of performance (EPE). However, note the sample size impacts the remaining uncertainty.