Bootstrap and Splines

SYS 6018 | Spring 2025

bootstrap.pdf

Contents

1 Introduction to the Bootstrap

1.1 Required R Packages

We will be using the R packages of:

- `broom` for tidy extraction of model components
- `splines` for working with B-splines
- `tidyverse` for data manipulation and visualization
- `tidymodels` for data optional modeling framework

```
library(broom)
library(splines)
library(tidyverse)
library(tidymodels)
```

1.2 Uncertainty in a test statistic

There is often interest in understanding the uncertainty in the estimated value of a test statistic.

- For example, let p be the actual/true proportion of customers who will use your company's coupon.
- To estimate p, you decide to take a sample of $n = 200$ customers and find that $x = 10$ or $\hat{p} = 10/200 = 0.05 = 5\%$ redeemed the coupon.

1.2.1 Confidence Interval

- It is common to calculate the 95% *confidence interval (CI)*

$$\begin{aligned}
\mathrm{CI}(p) &= \hat{p} \pm 2 \cdot \mathrm{SE}(\hat{p}) \\
&= \hat{p} \pm 2\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \\
&= 0.05 \pm 0.03
\end{aligned}$$

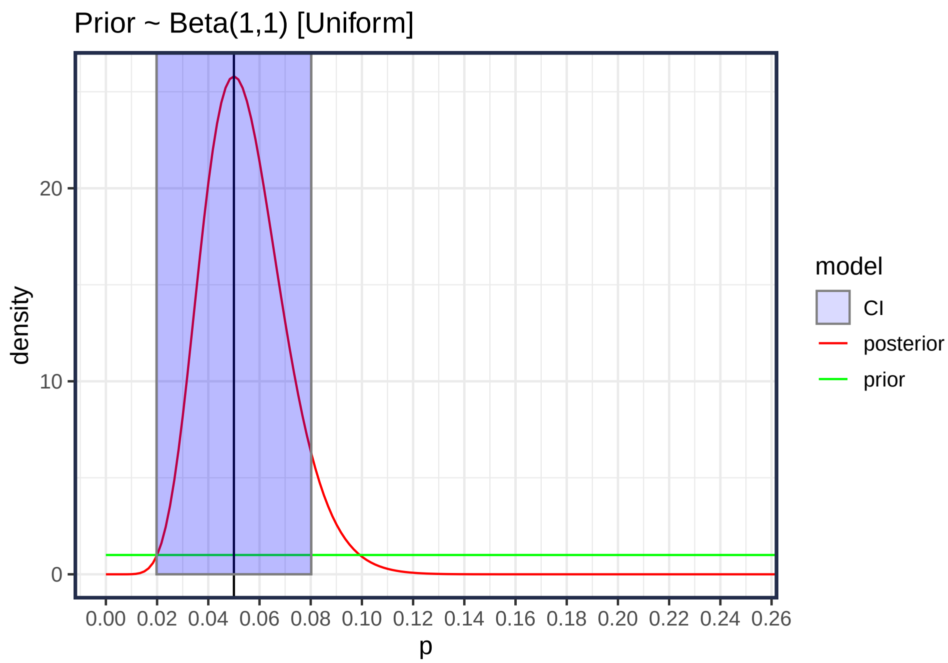
- This calculation is based on the assumption that $\hat{p}$ is approximately normally distributed with the mean equal to the *unknown* true p, i.e., $\hat{p} \sim N(p, \sqrt{\frac{p(1-p)}{n}})$.

Sample Size and Confidence

Notice that the width of the confidence interval (and Margin of Error) is inversely proportional to $\sqrt{n}$. The larger the sample size, the less uncertainty there is in the estimate.

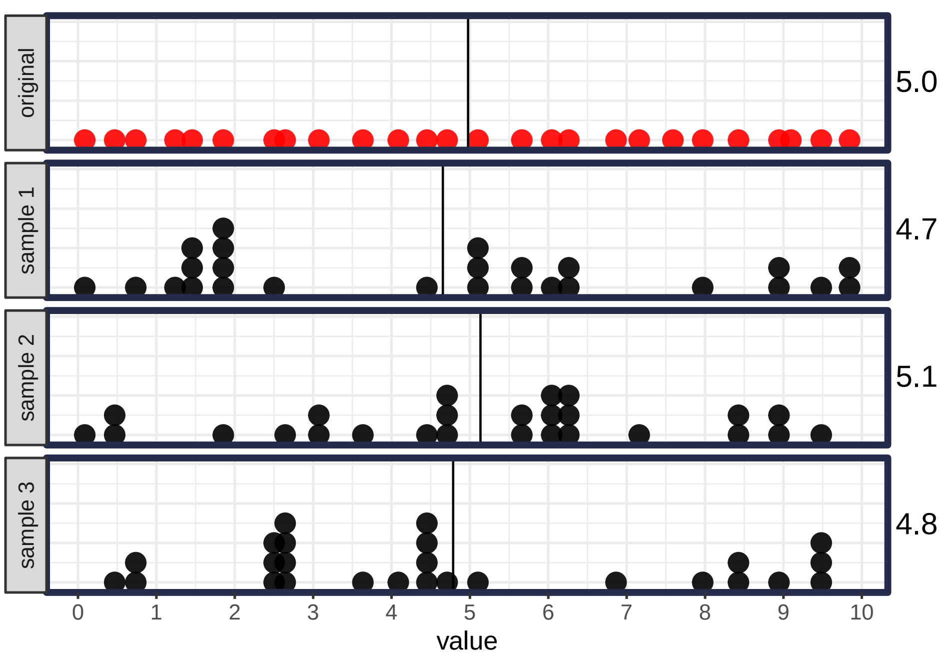
1.2.2 Bayesian Posterior Distribution

In the Bayesian world, you'd probably specify a Beta *prior* for p, i.e., $p \sim \mathrm{Beta}(a, b)$ and calculate the *posterior* distribution $p \mid x = 10 \sim \mathrm{Beta}(a + x, b + n - x)$ which would fully characterize the uncertainty.



1.2.3 The Bootstrap

- The Boostrap is a way to assess the uncertainty in a test statistic using *resampling*.
- The idea is to simulate the data from the *empirical distribution*, which puts a point mass of $1/n$ at each observed data point (i.e., sample the original data **with replacement**).
  - It is important to simulate n observations (same size as original data) because the uncertainty in the test statistic is a function of n



- Then, calculate the test statistic for each bootstrap sample. The variability in the collection of bootstrap test statistics should be similar to the variability in the test statistic.

Algorithm: Nonparametric/Empirical Bootstrap

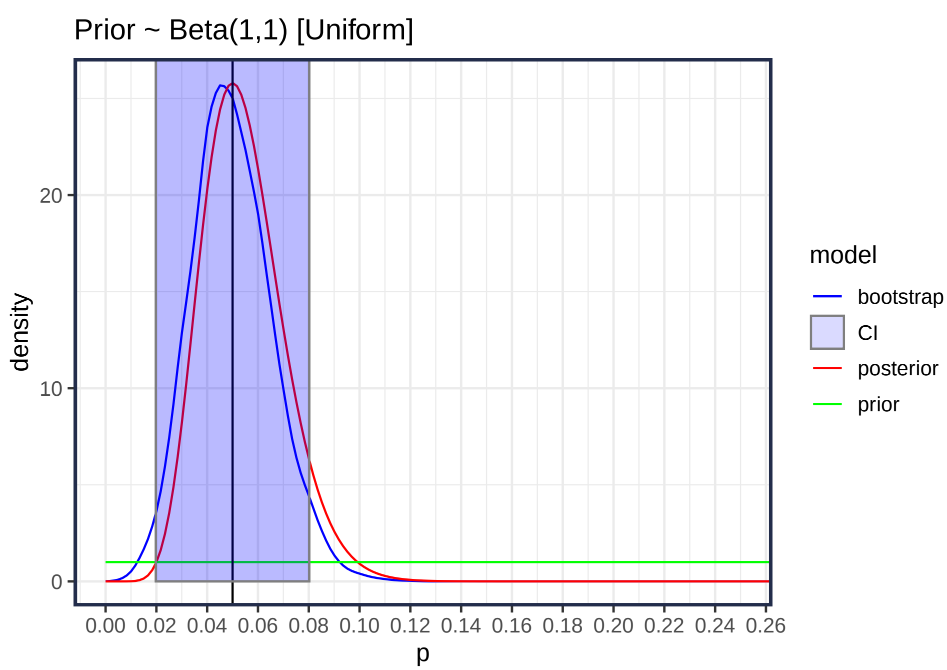
Observe data $D = [X_1, X_2, \dots, X_n]$ (n observations).
Calculate a test statistic $\hat{\theta} = \hat{\theta}(D)$, which is a function of D.
Repeat steps 1 and 2 M times:

1. Simulate D^*, a new data set of n observations by sampling from D with replacement.
2. Calculate the bootstrap test statistic $\hat{\theta}^* = \hat{\theta}(D^*)$

The bootstrapped samples $\hat{\theta}^*_1, \hat{\theta}^*_2, \dots, \hat{\theta}^*_M$ can be used to estimate the distribution of $\hat{\theta}$.

- Or properties of the distribution, like standard deviation (standard error), percentiles, etc.

```
#: Original Data
x = c(rep(1, 10), rep(0, 190))      # 10 successes, 190 failures
n = length(x)                       # length of observed data

#: Bootstrap Distribution
M = 5000                            # number of bootstrap samples
p = numeric(M)                      # initialize vector for test statistic
set.seed(201910)                    # set random seed
for(m in 1:M){
  # sample from empirical distribution
  ind = sample(n, replace=TRUE)     # sample indices with replacement
  xboot = x[ind]                    # bootstrap sample
  # calculate proportion of successes
  p[m] = mean(xboot)                # calculate test statistic
}

#: Bootstrap Percentile based confidence Intervals
quantile(p, probs=c(.025, .975))    # 95% bootstrap interval
#>  2.5% 97.5%
#> 0.020 0.085
```



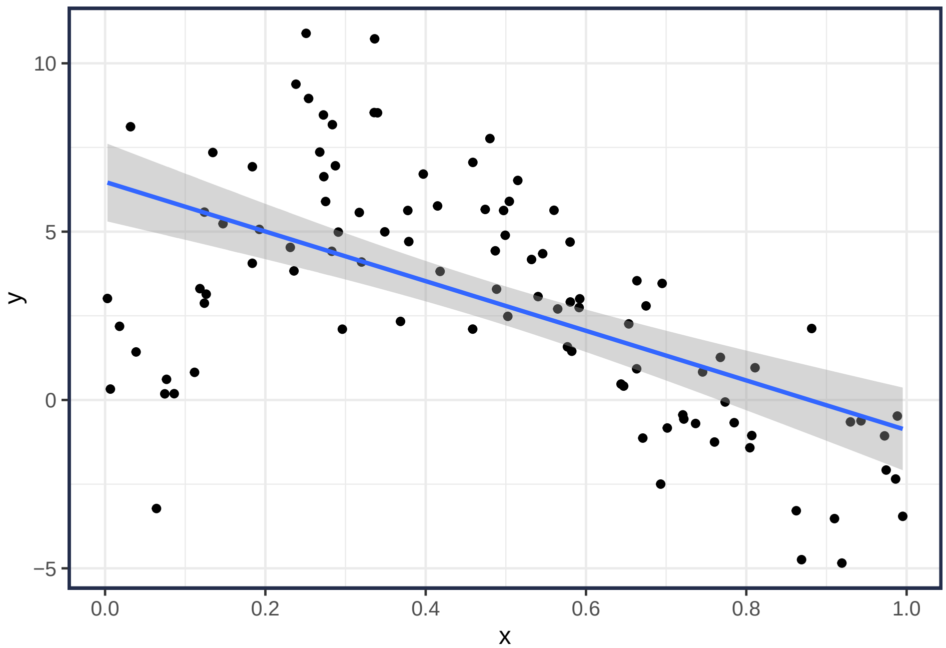
Note

- Notice that in the above example the bootstrap distribution is close to the Bayesian posterior distribution (using the uninformative Uniform prior).
- This is no accident, it turns out there is a close correspondence between the bootstrap derived distribution and the Bayesian posterior distribution under *uninformative priors*
  - See ESL 8.4 for more details

2 Bootstrapping Regression Parameters

The bootstrap is not limited to univariate test statistics. It can be used on multivariate test statistics.

Consider the uncertainty in estimates of the parameters (i.e., β coefficients) of a regression model.

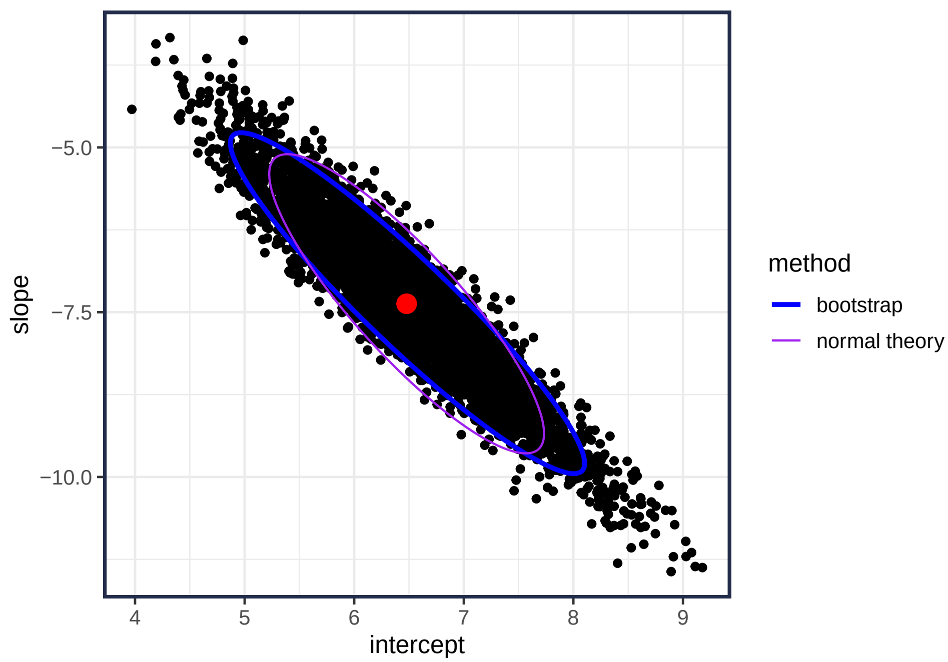



```
m1 = lm(y~x, data=data_train) # fit simple OLS
broom::tidy(m1, conf.int=TRUE)# OLS estimated coefficients
#> # A tibble: 2 x 7
#>   term        estimate std.error statistic  p.value conf.low conf.high
#>   <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
#> 1 (Intercept)     6.48     0.584     11.1  5.39e-19     5.32      7.64
#> 2 x              -7.37     1.06      -6.97 3.69e-10    -9.47     -5.27
vcov(m1) %>% round(2)         # variance matrix
#>             (Intercept)     x
#> (Intercept)        0.34 -0.54
#> x                 -0.54  1.12
vcov(m1) %>% diag() %>% sqrt() %>% round(2)   # standard errors
#> (Intercept)           x
#>        0.58        1.06
```

2.1 Bootstrap the β's

```
#: Bootstrap Distribution
n = nrow(data_train)                # size of training data
M = 5000                            # number of bootstrap samples
beta = list()                       # initialize list for test statistics
set.seed(201910)                    # set random seed
for(m in 1:M){
  # sample from empirical distribution
  ind = sample(n, replace=TRUE)     # sample indices with replacement
  data_boot = data_train[ind,]      # bootstrap sample
  # fit regression model
  m_boot = lm(y~x, data=data_boot) # fit simple OLS
  # save test statistics
  beta[[m]] = broom::tidy(m_boot) %>% select(term, estimate)
}
#: convert to tibble (and add column names)
beta = bind_rows(beta, .id = "iteration") %>%
  pivot_wider(names_from = term, values_from=estimate) %>%
  select(intercept = "(Intercept)", slope = "x", -iteration)
```

```
#: Plot
ggplot(beta, aes(intercept, slope)) +
  geom_point() +
  geom_point(data=tibble(intercept=coef(m1)[1], slope = coef(m1)[2]),
             color="red", size=4)
```




```
#: bootstrap estimate
var(beta) %>% round(2)             # variance matrix
#>           intercept slope
#> intercept      0.57 -0.85
#> slope         -0.85  1.45
apply(beta, 2, sd) %>% round(2)    # standard errors (sqrt of diagonal)
#> intercept     slope
#>      0.75      1.21
```

3 Non-linear Modeling via Basis Expansion

For a univariate x, a linear basis expansion is

$$\hat{f}(x) = \sum_j \hat{\theta}_j b_j(x)$$

where $b_j(x)$ is the value of the jth basis function at x and θ_j is the coefficient to be estimated.

- The $b_j(x)$ are sometimes pre-specified before modeling (i.e., not estimated). But other approaches use sample data to estimate (e.g., using quantiles for knot placement).
  - Just be sure to estimate everything from the training data so there is no data leakage!

Examples:

- **Linear Regression**

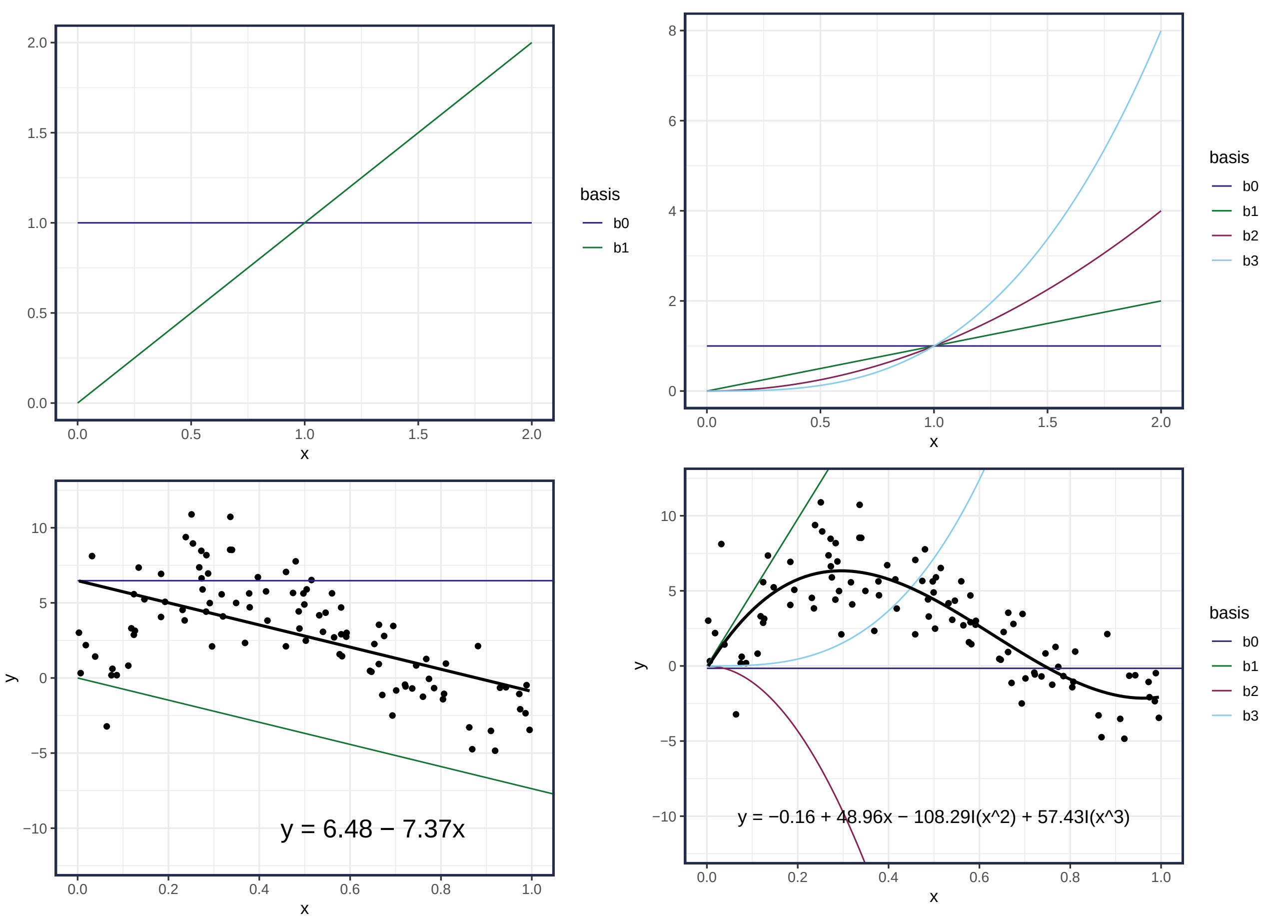
$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x$$

$$b_0(x) = 1$$
$$b_1(x) = x$$

- **Polynomial Regression**

$$\hat{f}(x) = \sum_{j=1}^{d} \hat{\beta}_j x^j$$

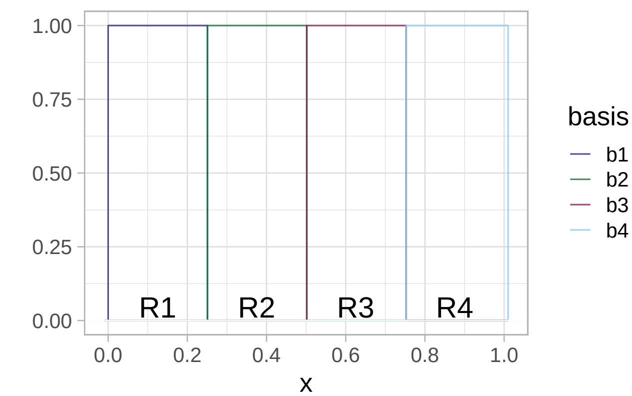
$$b_j(x) = x^j$$

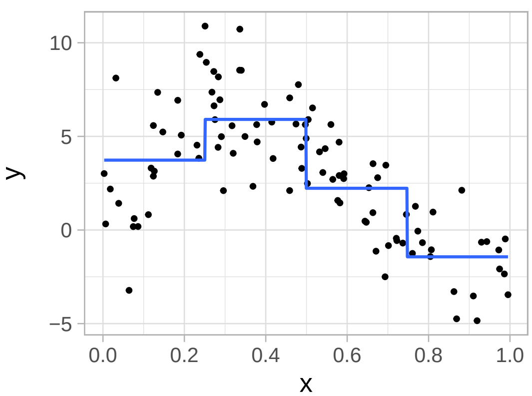


- **Piecewise Constant Regression (Regressogram)**

$$\hat{f}(x) = \sum_{j=1}^{p} \hat{\beta}_j \, \mathbb{1}(x \in R_j)$$

$$b_1(x) = \mathbb{1}(x \in R_1)$$
$$b_2(x) = \mathbb{1}(x \in R_2)$$
$$\vdots$$
$$b_p(x) = \mathbb{1}(x \in R_p)$$



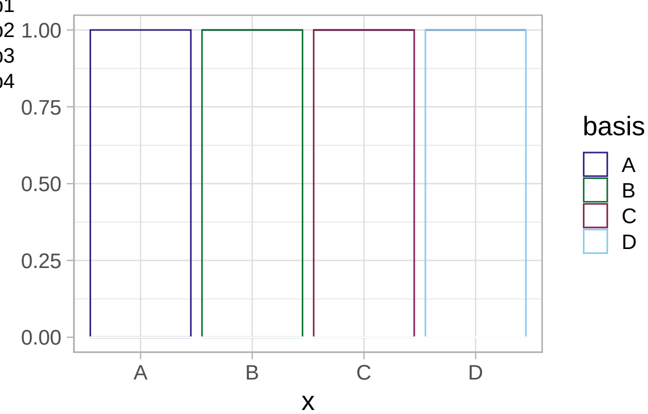


- **Categorical encoding (dummy, one-hot)**

$$x \in \{c_1, c_2, \ldots, c_p\}$$

$$\hat{f}(x) = \sum_{j=1}^{p} \hat{\beta}_j \, \mathbb{1}(x = c_j) \qquad \text{one-hot}$$
$$= \hat{\beta}_0 + \sum_{j=2}^{p} \hat{\beta}_j \, \mathbb{1}(x = c_j) \qquad \text{dummy}$$

$$b_1(x) = \mathbb{1}(x = c_1)$$
$$b_2(x) = \mathbb{1}(x = c_2)$$
$$\vdots$$
$$b_p(x) = \mathbb{1}(x = c_p)$$



3.1 Piecewise Polynomials

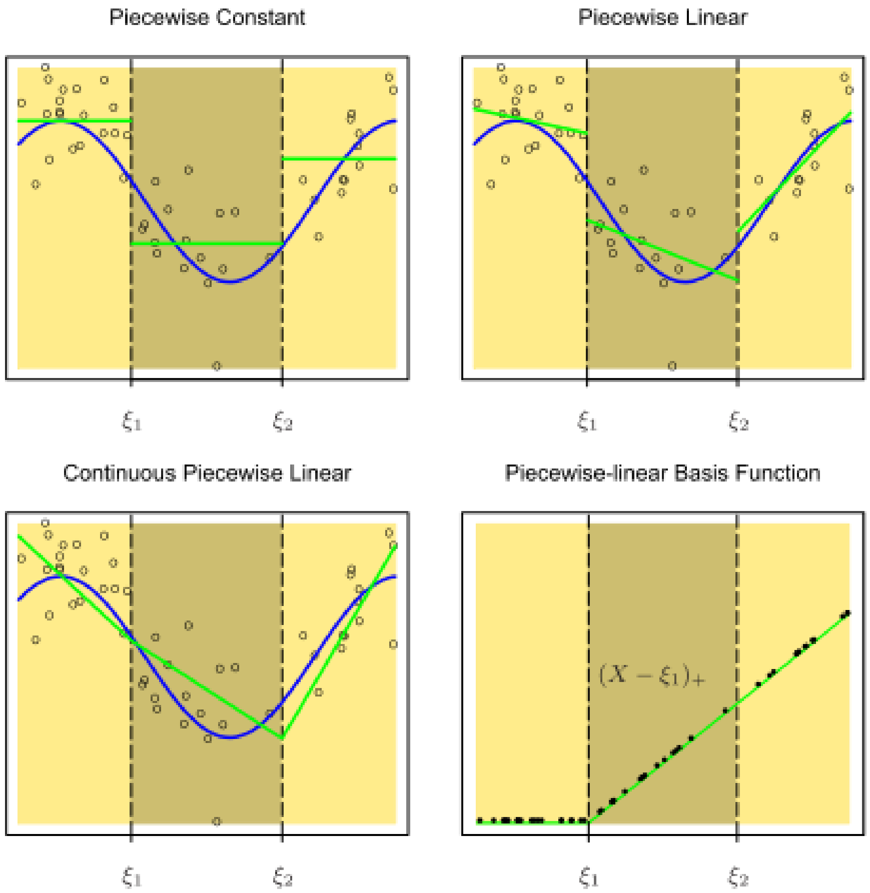


FIGURE 5.1. *The top left panel shows a piecewise constant function fit to some artificial data. The broken vertical lines indicate the positions of the two knots ξ_1 and ξ_2. The blue curve represents the true function, from which the data were generated with Gaussian noise. The remaining two panels show piecewise linear functions fit to the same data—the top right unrestricted, and the lower left restricted to be continuous at the knots. The lower right panel shows a piecewise-linear basis function, $h_3(X) = (X - \xi_1)_+$, continuous at ξ_1. The black points indicate the sample evaluations $h_3(x_i)$, $i = 1, \ldots, N$.*

Model Matrix

How can you make a model matrix (aka design matrix) for the piecewise polynomial predictor variables?

3.2 B-Splines

- A degree = 0 B-spline is a *regressogram* basis. Will lead to a piecewise constant fit.
- A degree = 3 B-spline (called *cubic* splines) is similar in shape to a Gaussian pdf. But the B-spline has finite support and facilitates quick computation (due to the induced sparseness).

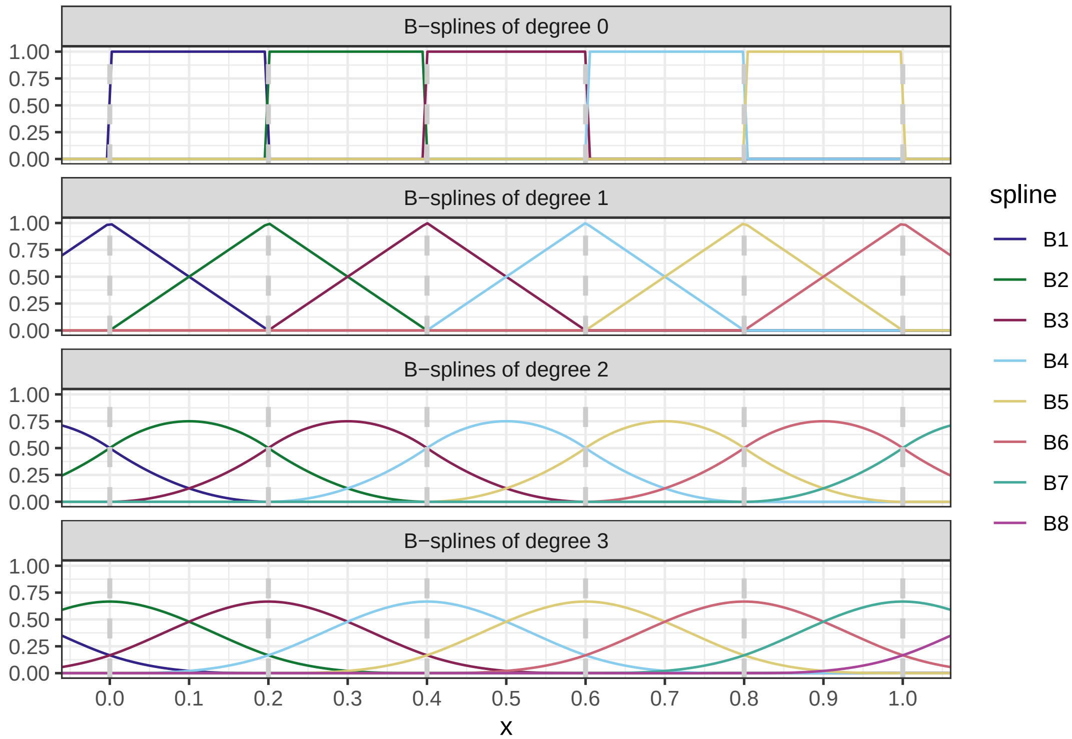


Figure 1: Like ESL Fig 5.20, B-splines (knots shown by vertical dashed lines)

3.2.1 Parameter Estimation

$$\hat{f}(x) = \sum_j \hat{\theta}_j b_j(x)$$

In matrix notation,

$$\hat{f}(X) = B\hat{\theta}$$

where B is the *basis matrix* and X is the model matrix.

- For example, a polynomial matrix is

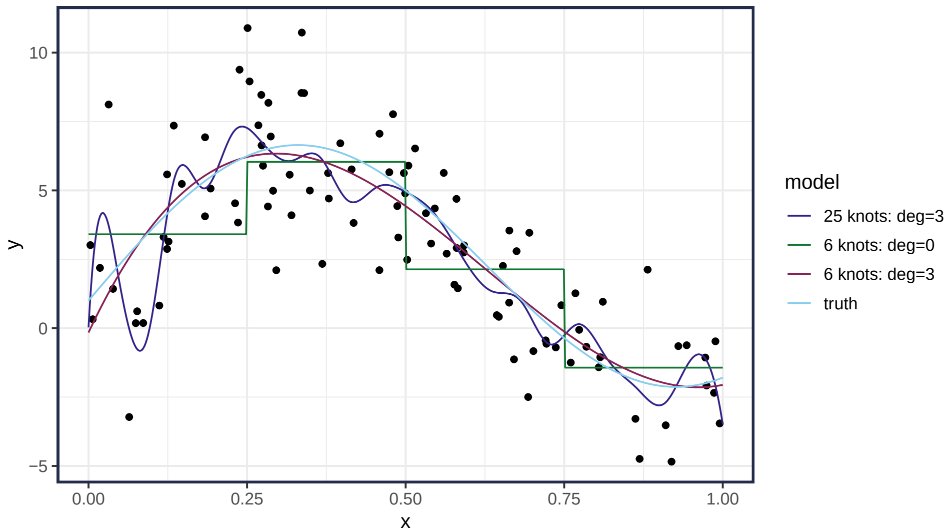
$$B = \begin{bmatrix} 1 & X_1 & X_1^2 & \dots & X_1^J \\ 1 & X_2 & X_2^2 & \dots & X_2^J \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & X_n & X_n^2 & \dots & X_n^J \end{bmatrix}$$

- More generally,

$$B = \begin{bmatrix} b_1(x_1) & b_2(x_1) & \dots & b_J(x_1) \\ b_1(x_2) & b_2(x_2) & \dots & b_J(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ b_1(x_n) & b_2(x_n) & \dots & b_J(x_n) \end{bmatrix}$$

- This is in a mathematical form just like linear regression! Estimate with OLS is familiar:

$$\hat{\theta} = (B^{\mathsf{T}}B)^{-1}B^{\mathsf{T}}Y$$



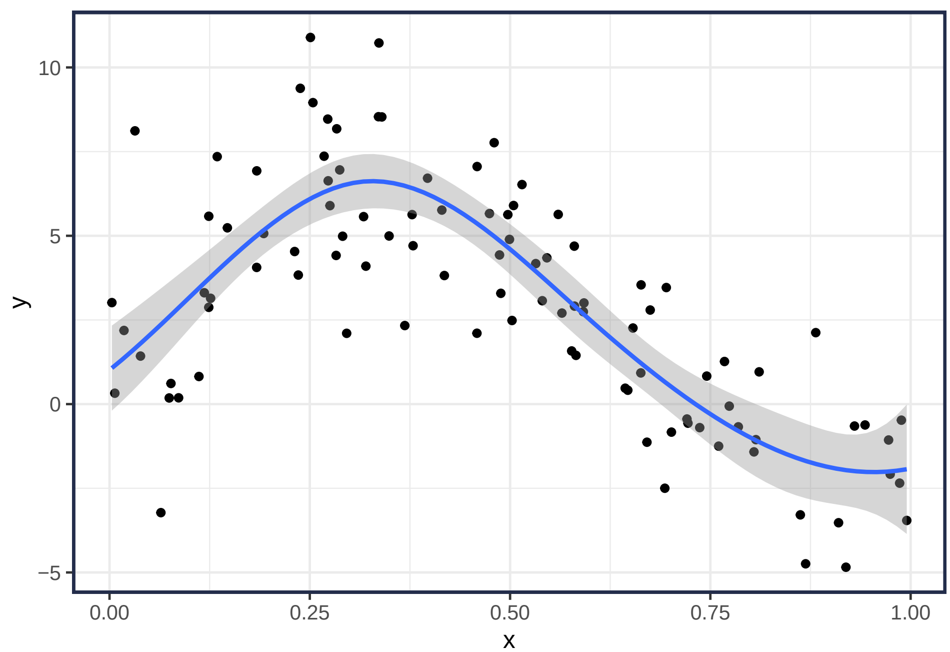
- It may be helpful to think of a basis expansion as similar to a dummy coding for categorical variables.
  - This expands the single variable x into df new variables.
- In R, the function `bsp()`, from the `splines2` package can be put directly into the formula to create the B-spline basis.
  - Using `-1` in the formula to remove the intercept (optional)

```
#- fit a 5 df cubic B-spline
library(splines2)
# function to fit a cubic (deg=3) b-spline regression from .data with columns
#  x and y. The ... allow additional arguments passed into the bsp() function.
# The Boundary.knots argument lets you control the outer knots -
#  which should help control variability in the tails
fit_cubic_bspline <- function(.data, df = 5, ...){
  lm(y ~ bsp(x, df=df, deg = 3, Boundary.knots = c(-.2, 1.2), ...)-1,
     data = .data)
}
```

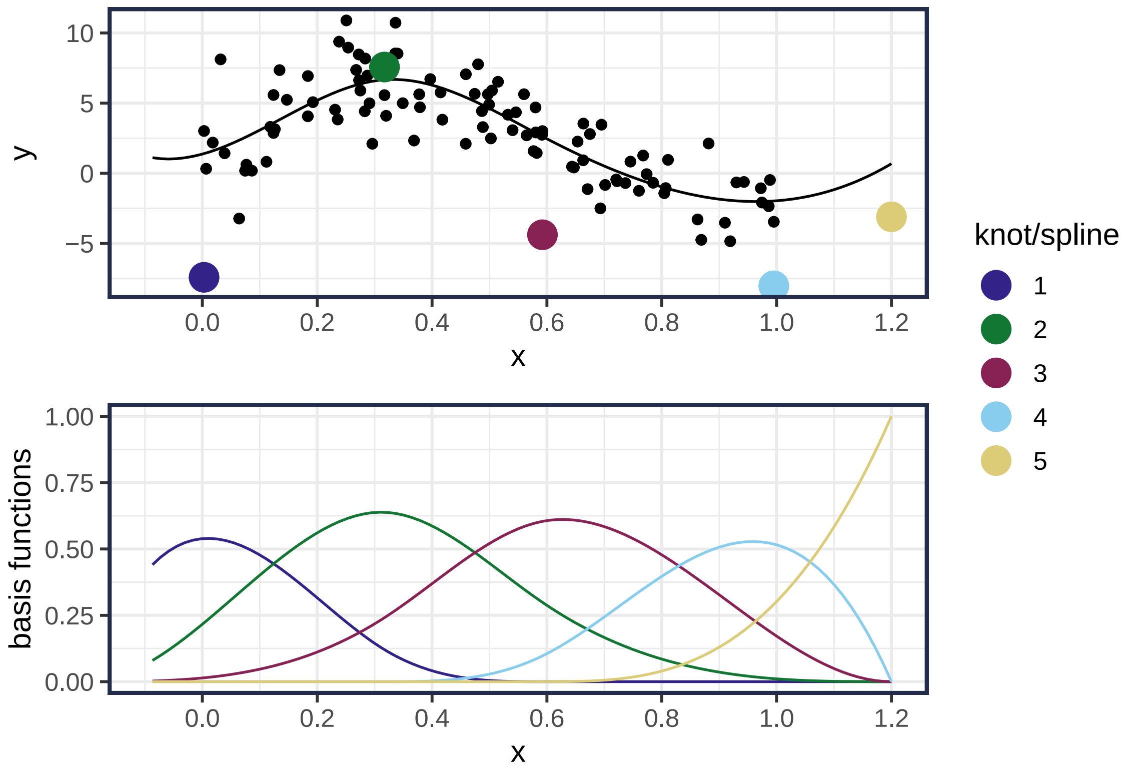
```
model_bs = fit_cubic_bspline(data_train, df = 5)
summary(model_bs)
#>
#> Call:
#> lm(formula = y ~ bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2,
#>     1.2), ...) - 1, data = .data)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -5.584 -1.414 -0.182  1.416  6.463
#>
#> Coefficients:
#>                                                               Estimate
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1   -2.501
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2   10.921
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3   -0.241
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4   -4.706
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5    1.453
#>                                                               Std. Error
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1      1.515
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2      1.268
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3      1.534
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4      3.068
```

```
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5       6.901
#>                                                              t value Pr(>|t|)
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1   -1.65     0.10
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2    8.61  1.5e-13
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3   -0.16     0.88
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4   -1.53     0.13
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5    0.21     0.83
#>
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2 ***
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.09 on 95 degrees of freedom
#> Multiple R-squared:  0.804,  Adjusted R-squared:  0.794
#> F-statistic:   78 on 5 and 95 DF,  p-value: <2e-16
```

```
ggplot(data_train, aes(x,y)) +
  geom_point() +
  geom_smooth(
    method='lm',
    formula='y~splines2::bsp(x, df=5, deg=3, Boundary.knots = c(-.2, 1.2))-1'
  )
```



- Setting `df=5` will create a B-spline model matrix with 5 columns (plus an intercept)
  - One column for each basis function

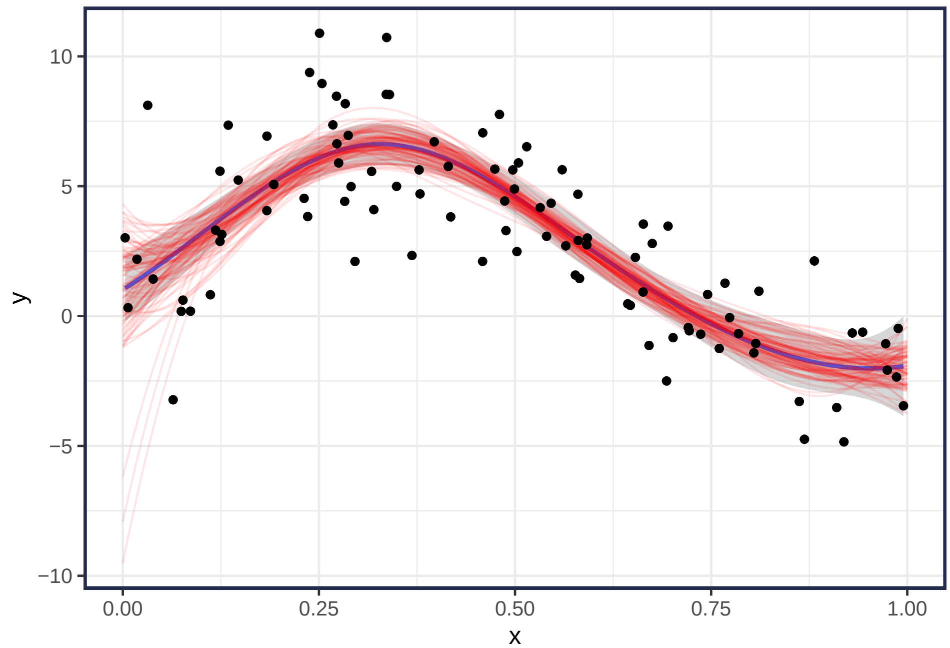


3.3 Bootstrap Confidence Interval for $f(x)$

Bootstrapping can be used to understand the uncertainty in the fitted values

```
#: Bootstrap CI (Percentile Method)
M = 100                                       # number of bootstrap samples
data_eval = tibble(x=seq(0, 1, length=300)) # evaluation points
YHAT = matrix(NA, nrow(data_eval), M)        # initialize matrix for fitted values

set.seed(201910)
for(m in 1:M){
  # sample indices/rows from empirical distribution (with replacement)
  ind = sample(n, replace=TRUE)
  # fit bspline model to those indices/rows
  m_boot = fit_cubic_bspline(data_train[ind,], # fit bootstrap data
                             df = 5)
  # predict from bootstrap model
  YHAT[,m] = predict(m_boot, data_eval)
}
```

```
#: Convert to tibble and plot
data_fitted = as_tibble(YHAT) %>%  # convert matrix to tibble
  bind_cols(data_eval) %>%      # add the eval points
  pivot_longer(-x, names_to="simulation", values_to="y") # convert to long format

ggplot(data_train, aes(x,y)) +
  geom_smooth(method='lm',
              formula='y~splines2::bsp(x, df=5, deg=3, Boundary.knots = c(-.2, 1.2))-1') +
  geom_line(data=data_fitted, color="red", alpha=.10, aes(group=simulation)) +
  geom_point()
```




```
#-- Calculate Confidence intervals
## for a 90% CI, find the upper and lower 5% values at every x location
## Homework Exercise
```

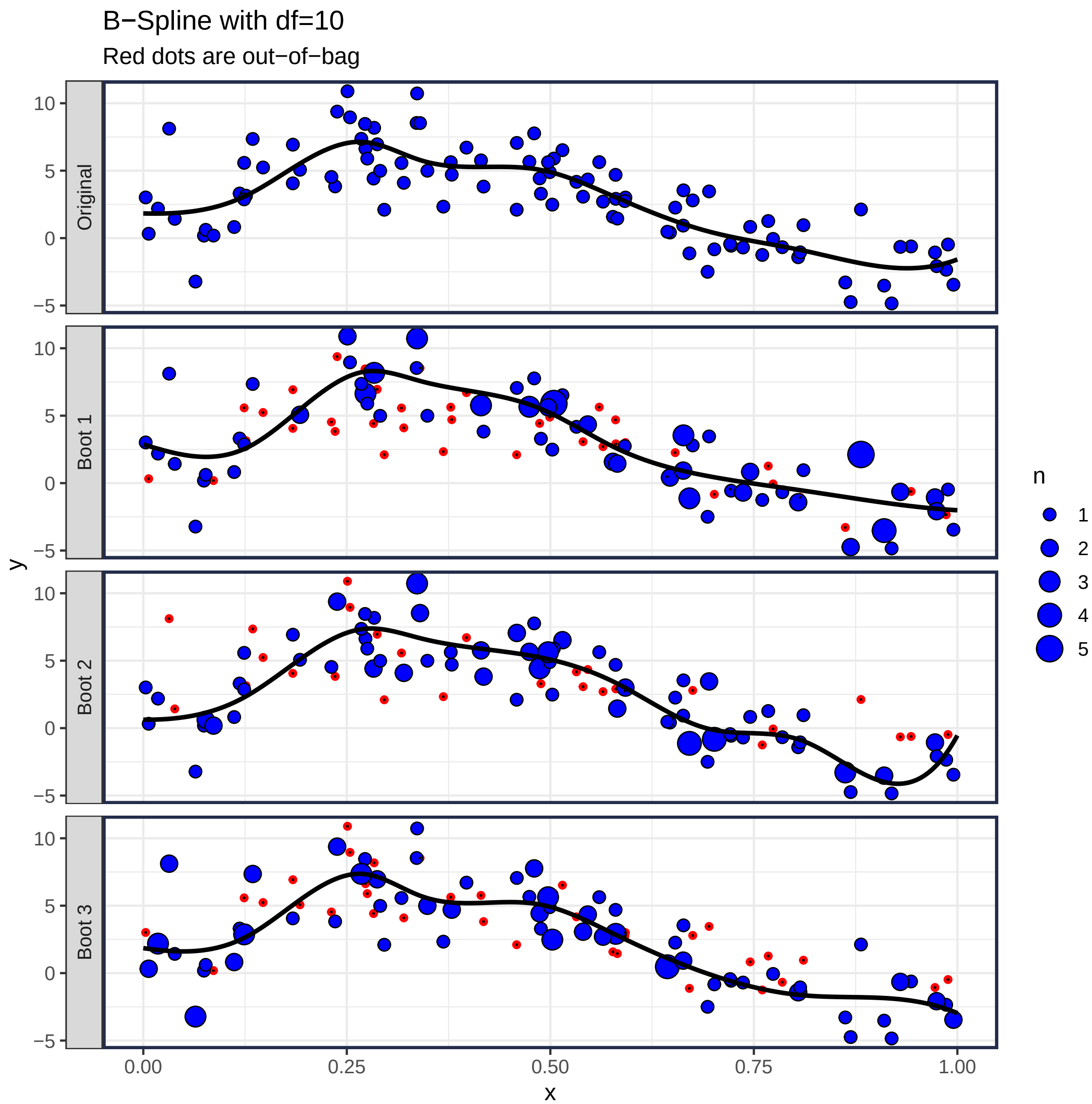
4 More Bagging

4.1 Out-of-Bag Samples

Your Turn #1 : Observations not in bootstrap sample

What is the expected proportion of observations that will *not* be in a bootstrap sample?

Let's look at a few bootstrap fits:

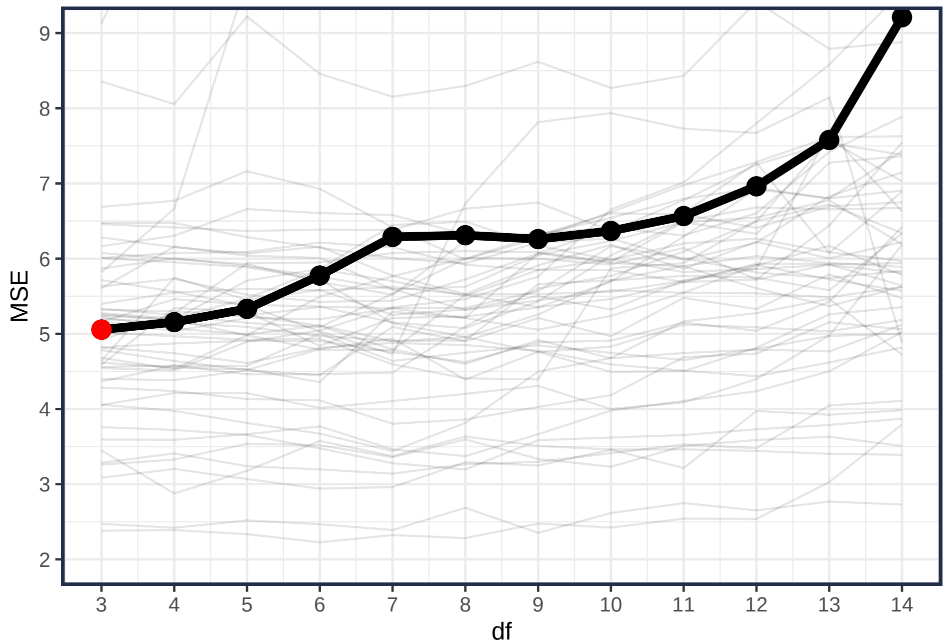


- Notice that each bootstrap sample excludes about 37% of the original observations.
- These are called *out-of-bag* (oob) samples and can be used to assess model fit
  - The out-of-bag observations were not used to estimate the model parameters, so will be sensitive to over/under fitting
- Below, we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

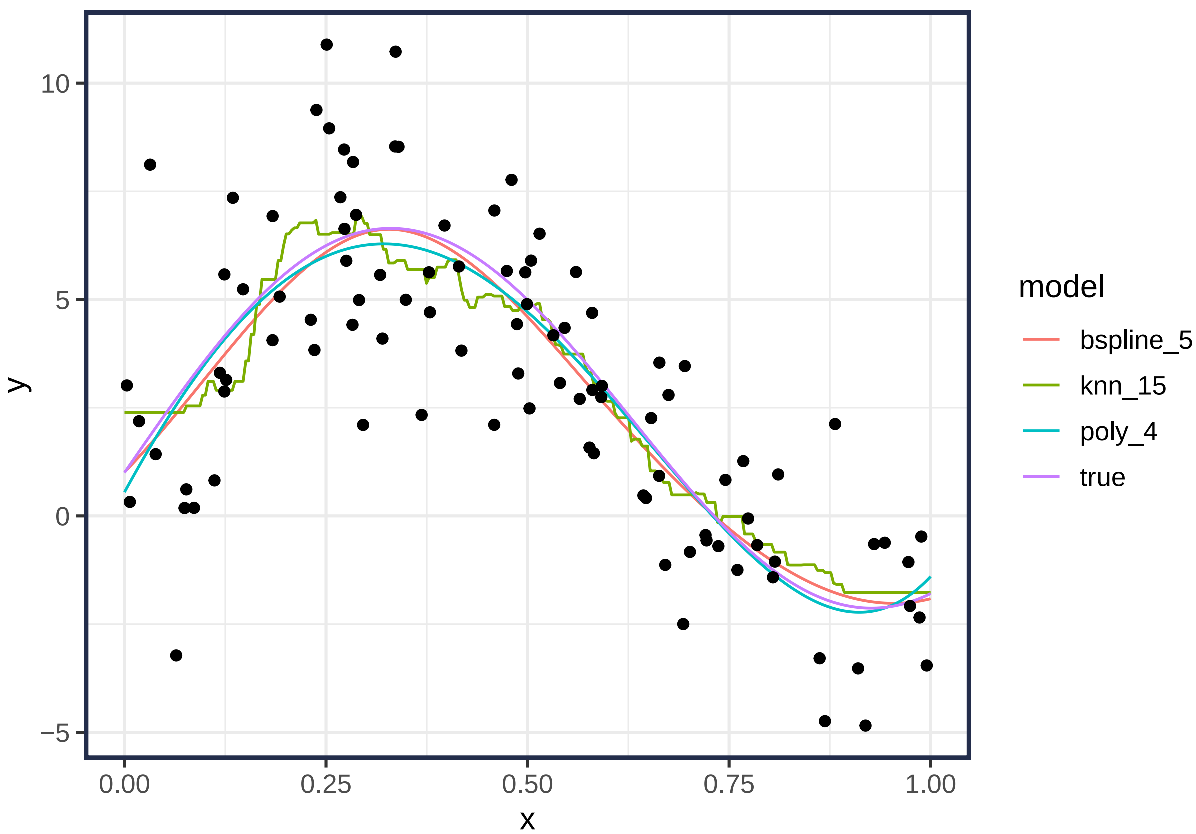
Bootstrapping B-spline fits

```
#: Settings
M = 50                        # number of bootstrap samples
DF = seq(3, 14, by=1)         # edfs for spline
n = nrow(data_train)

#: set-up
results = list()               # initialize results list
set.seed(2019)                 # set seed so reproducible

#: loop over M bootstraps
for(m in 1:M){

  #: sample from empirical distribution
  ind = sample(n, replace=TRUE)        # sample indices with replacement
  oob.ind = setdiff(1:n, ind)          # out-of-bag samples

  #: fit bspline models to all df in DF
  for(df in DF){
    if(length(oob.ind) < 1) next  # protection in case of no OOB
    #: fit with bootstrap data
    m_boot = fit_cubic_bspline(data_train[ind,], df = df)
    #: predict on oob data
    yhat.oob = predict(m_boot, data_train[oob.ind, ])
    #: evaluate
    sse = sum( (data_train$y[oob.ind] - yhat.oob)^2 )
    n.oob = length(oob.ind)
    #: save results
    results = c(results, list(tibble(m, df = df, sse, n.oob)))
  }

}
results = bind_rows(results)     # convert from list to tibble
```

```
avg = results %>% group_by(df) %>% summarize(mse = sum(sse)/sum(n.oob))
plot1 = results %>%
  ggplot(aes(x=df, y=sse/n.oob)) +
  geom_line(aes(group=m), alpha=.10) +
  coord_cartesian(ylim=c(2, 9)) +
  scale_x_continuous(breaks=1:20) + scale_y_continuous(breaks=1:20) +
  labs(x = "df", y="MSE")

plot1 +
  geom_point(data=avg, aes(df,mse), size=4) +
  geom_line(data=avg, aes(df,mse), linewidth=2) +
  geom_point(data=avg %>% slice_min(mse), aes(df, mse), color="red", size=4)
```



- The minimum out-of-bag error occurs at `df=5`. This matches the optimal complexity in a polynomial fit from the previous lecture notes.



4.2 Number of Bootstrap Simulations

Hesterberg recommends using $M \geq 15{,}000$ for real applications to remove most of the Monte Carlo variability.

- For the examples in class I used much less to demonstrate the principles.

5 More Resources

- Bootstrap
  - ISL 5.2
  - ESL 7.11
- Splines
  - ISL 7.2-7.5
  - ESL 5.1-5.4
- What Teachers Should Know About the Bootstrap: Resampling in the Undergraduate Statistics Curriculum, by Tim C. Hesterberg
- The `boot` package and `boot()` function provides some more advanced options for bootstrapping
- R's `tidymodels` package
  - Bootstrap resampling and tidy regression models
  - `rsample` for resampling
  - `yardstick` for evaluation metrics
  - `broom` for extracting properties (e.g., estimated parameters) of fitted models in a tidy form

5.1 Variations of the Bootstrap

- We have discussed only one type of bootstrap, *nonparametric/empirical/ordinary* where the observations are resampled
- Another option is to simulate from the *fitted model*. This is called the *parametric* bootstrap.
  - For example, in the regression setting, estimate $\hat{\theta}$ and $\hat{\sigma}$
  - Then given the original X's simulate new $y_i^* \mid x_i \sim f(x_i; \hat{\theta}) + \epsilon(\hat{\sigma})$

6 Appendix: R Code

6.1 Simulate Data

```
library(tidyverse)
```

```
n = 100                                      # number of observations
sim_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 = sim_x(n)                                 # get x values
y = f(x) + rnorm(n, sd=sd)                   # get y values
data_train = tibble(x,y)                     # create a data frame/tibble
```

6.2 Fit Linear Model; get coefficients

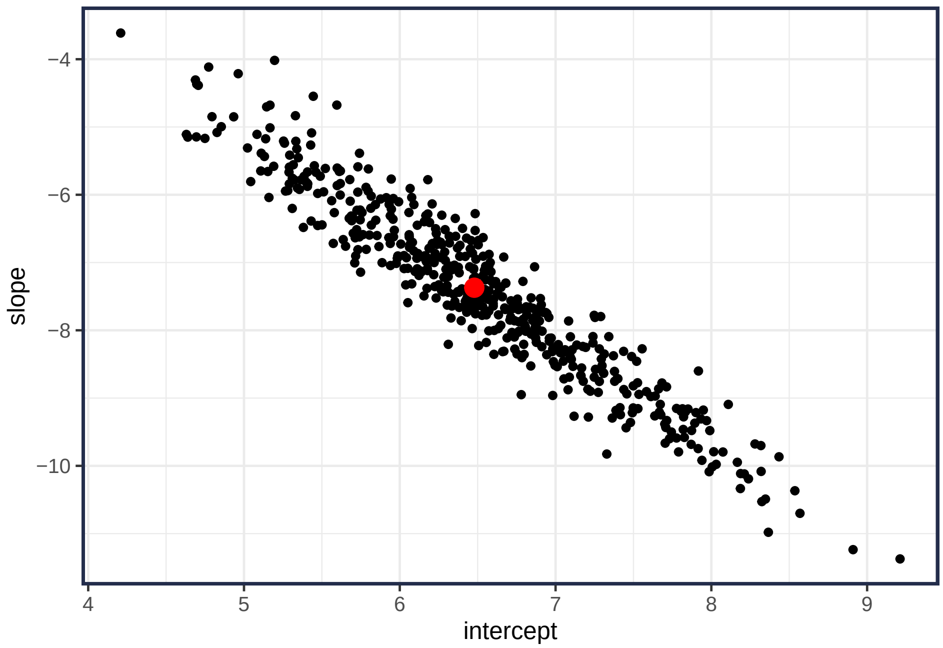
```
m1 = lm(y~x, data=data_train)  # fit simple OLS
broom::tidy(m1, conf.int=TRUE) # OLS estimated coefficients
#> # A tibble: 2 x 7
#>   term        estimate std.error statistic  p.value conf.low conf.high
#>   <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
#> 1 (Intercept)     6.48     0.584     11.1  5.39e-19     5.32      7.64
#> 2 x              -7.37     1.06      -6.97 3.69e-10    -9.47     -5.27
vcov(m1)     # normal theory based uncertainty
#>             (Intercept)       x
#> (Intercept)      0.3414 -0.5359
#> x               -0.5359  1.1183
```

Note that the linear model is poorly fitting, so don't expect good results for coefficients.

6.3 Bootstrap distribution

```
M = 500                              # number of bootstrap samples
set.seed(2019)                       # set random seed

beta = vector("list", M)             # initialize list for test statistics
for(m in 1:M){
  #- sample from empirical distribution
  ind = sample(n, replace=TRUE)      # sample indices with replacement
  data_boot = data_train[ind,]       # bootstrap sample
  #- fit regression model
  m_boot = lm(y~x, data=data_boot) # fit simple OLS
  #- save test statistics
  beta[[m]] = broom::tidy(m_boot) %>% select(term, estimate)
}
```

```
#- convert to tibble (and add column names)
beta = bind_rows(beta, .id = "iteration") %>%
  pivot_wider(names_from = term, values_from=estimate) %>%
  select(intercept = "(Intercept)", slope = "x", -iteration)

#- Plot
ggplot(beta, aes(intercept, slope)) + geom_point() +
  geom_point(data=tibble(intercept=coef(m1)[1],
                         slope = coef(m1)[2]), color="red", size=4)
```




```
#- bootstrap estimate
var(beta)              # variance matrix
#>           intercept  slope
#> intercept    0.6989 -1.068
#> slope       -1.0676  1.792
apply(beta, 2, sd)    # standard errors (sqrt of diagonal)
#> intercept     slope
#>     0.836     1.338
```

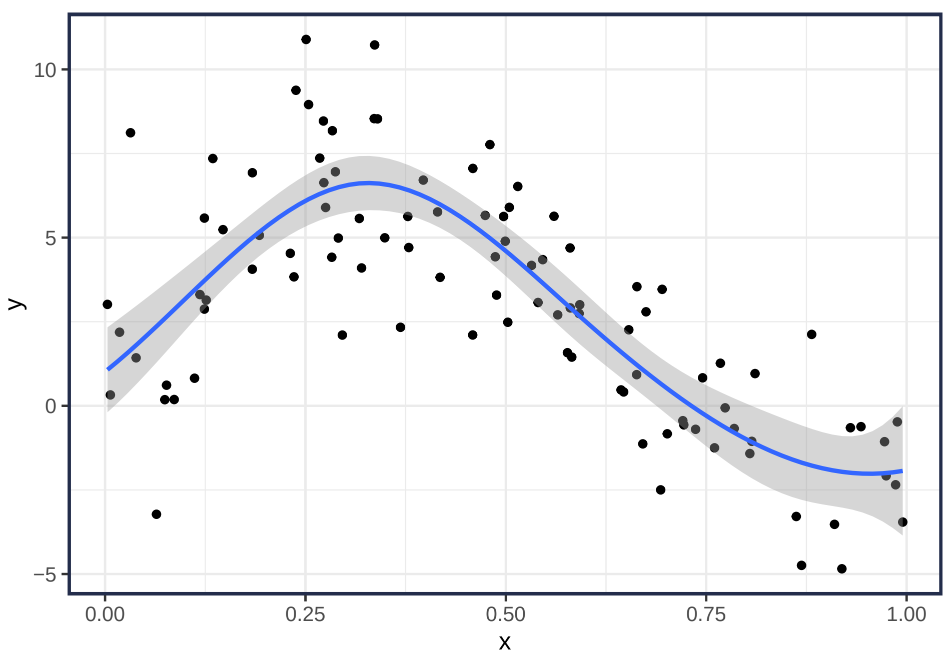
6.4 B-spline model

```
#- fit a 5 df cubic B-spline
library(splines2)
# function to fit a cubic (deg=3) b-spline regression from .data with columns
#  x and y. The ... allow additional arguments passed into the bsp() function.
# The Boundary.knots argument lets you control the outer knots -
#  which should help control variability in the tails
fit_cubic_bspline <- function(.data, df = 5, ...){
  lm(y ~ bsp(x, df=df, deg = 3, Boundary.knots = c(-.2, 1.2), ...)-1,
     data = .data)
}
```

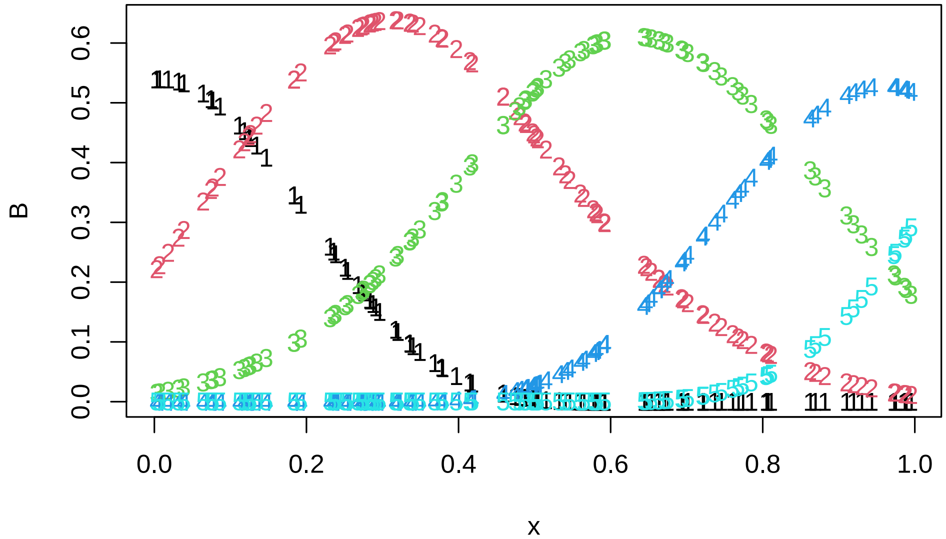
```
model_bs = fit_cubic_bspline(data_train, df = 5)
summary(model_bs)
#>
#> Call:
#> lm(formula = y ~ bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2,
#>     1.2), ...) - 1, data = .data)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -5.584 -1.414 -0.182  1.416  6.463
#>
#> Coefficients:
#>                                                               Estimate
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1   -2.501
```

```
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2   10.921
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3   -0.241
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4   -4.706
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5    1.453
#>                                                                Std. Error
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1      1.515
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2      1.268
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3      1.534
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4      3.068
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5      6.901
#>                                                                t value Pr(>|t|)
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1   -1.65     0.10
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2    8.61  1.5e-13
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3   -0.16     0.88
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4   -1.53     0.13
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5    0.21     0.83
#>
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)1
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)2 ***
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)3
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)4
#> bsp(x, df = df, deg = 3, Boundary.knots = c(-0.2, 1.2), ...)5
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.09 on 95 degrees of freedom
#> Multiple R-squared:  0.804,  Adjusted R-squared:  0.794
#> F-statistic:   78 on 5 and 95 DF,  p-value: <2e-16
```

```
ggplot(data_train, aes(x,y)) +
  geom_point() +
  geom_smooth(
    method='lm',
    formula='y~splines2::bsp(x, df=5, deg=3, Boundary.knots = c(-.2, 1.2))-1'
  )
```



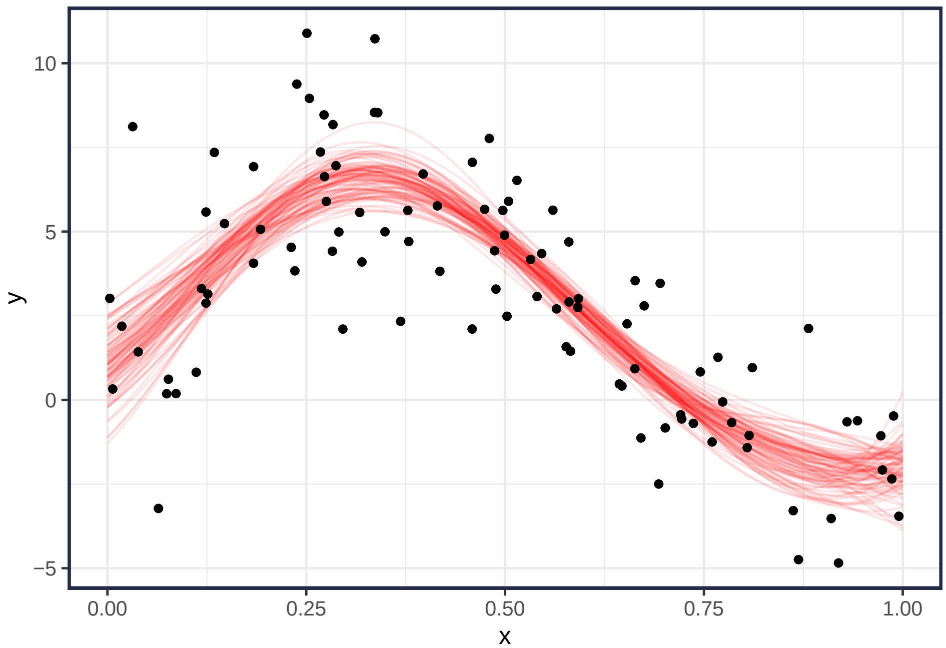
```
#: Evaluate the B-spline Basis
B = splines2::bsp(x, df=5, deg=3, Boundary.knots = c(-.2, 1.2))
matplot(x, B, type='p')
```



6.5 Bootstrap Uncertainty in B-spline Fit

```
M = 100                                        # number of bootstrap samples
data_eval = tibble(x=seq(0, 1, length=300)) # evaluation points
YHAT = matrix(NA, nrow(data_eval), M)        # initialize matrix for fitted values

#-- loop
set.seed(2019)
for(m in 1:M){
  # sample indices/rows from empirical distribution (with replacement)
  ind = sample(n, replace=TRUE)
  # fit bspline model to those indices/rows
  m_boot = fit_cubic_bspline(data_train[ind,], # fit bootstrap data
                             df = 5)
  #- predict from bootstrap model
  YHAT[,m] = predict(m_boot, data_eval)
}
```

```
#-- Convert to tibble and plot
data_fitted = as_tibble(YHAT) %>%  # convert matrix to tibble
  bind_cols(data_eval) %>%      # add the eval points
  pivot_longer(-x, names_to="simulation", values_to="y") # convert to long format

ggplot(data_train, aes(x,y)) +
  geom_smooth(method='lm',
              formula='y~bs(x, df=5, deg=3, Boundary.knots = kts_bdry)-1') +
  geom_line(data=data_fitted, color="red", alpha=.10, aes(group=simulation)) +
  geom_point()
#> Warning: Failed to fit group -1.
#> Caused by error:
#> ! object 'kts_bdry' not found
```

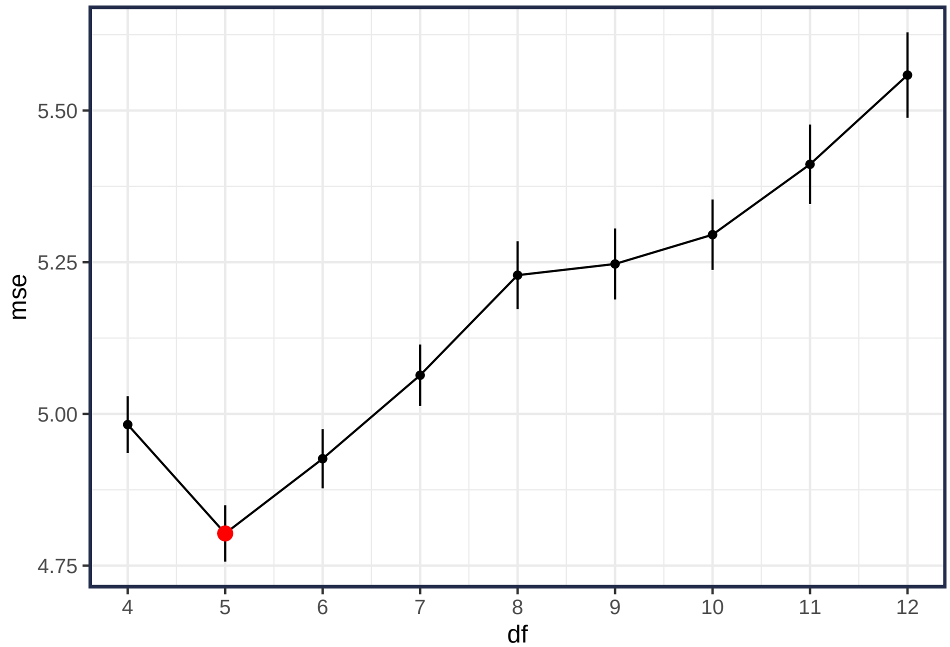


6.6 Out-of-bag performance evaluation

```
M = 500                       # number of bootstrap samples
DF = seq(4, 12, by=1)         # edfs for spline
results = list()              # initialize results list
set.seed(2019)                # set seed so reproducible

#-- Spline Settings
for(m in 1:M){
  #- sample from empirical distribution
  ind = sample(n, replace=TRUE)          # sample indices with replacement
  oob.ind = setdiff(1:n, ind)            # out-of-bag samples
  #- fit bspline models
  for(df in DF){
    if(length(oob.ind) < 1) next
    #- fit with bootstrap data
    m_boot = fit_cubic_bspline(data_train[ind,], # fit bootstrap data
                               df = df)
    #- predict on oob data
    yhat.oob = predict(m_boot, data_train[oob.ind, ])
    #- get errors
    sse = sum( (data_train$y[oob.ind] - yhat.oob)^2 )
    n.oob = length(oob.ind)
    #- save results
    results = c(results, list(tibble(m, df, sse, n.oob)))
  }
}
results = bind_rows(results)     # convert from list to tibble
```

```
results %>%
  group_by(df) %>%
  summarize(
    mse = sum(sse)/sum(n.oob),
    n_iter = n(),
    se = sd(sse/n.oob) / sqrt(n_iter)
```

```
  ) %>%
  ggplot(aes(df, mse)) +
  geom_point() + geom_line() +
  geom_linerange(aes(df, ymin = mse-se, ymax=mse+se)) + # 1 std error
  geom_point(data=. %>% slice_min(mse), color="red", size=3) +
  scale_x_continuous(breaks=1:20)
```



6.7 Using *tidymdodels* (`rsample` package)

6.7.1 Use rsample package for bootstrapping

The `rsample` package provides methods for creating a low memory set of bootstrap samples.

```
library(rsample)
set.seed(2019)
boots = rsample::bootstraps(data_train, times = 500)
```

The `boots` object is a tibble with two columns. The `splits` column contains the bootstrap (in-bag) and out-of-bag samples. The `id` gives the iteration. The bootstrap samples can be extracted from the `splits` column with `training()` and oob with `testing()`.

```
boots
#> # A tibble: 500 x 2
#>   splits            id
#>   <list>            <chr>
#> 1 <split [100/38]> Bootstrap001
#> 2 <split [100/41]> Bootstrap002
#> 3 <split [100/42]> Bootstrap003
#> 4 <split [100/37]> Bootstrap004
#> 5 <split [100/34]> Bootstrap005
#> 6 <split [100/37]> Bootstrap006
#> # i 494 more rows
```

6.7.2 bootstrap distribution of linear model coefficients

```
#: function to fit lm and extract coefficients
lm_get_coefs <- function(data){
  m = lm(y~x, data=data) # fit simple OLS
  broom::tidy(m) %>%
    select(term, estimate) %>%    # extract coefficients
    pivot_wider(names_from = term, values_from = estimate) %>% # wide
    rename(intercept = "(Intercept)", slope = "x") # rename to intercept,slope
}

#: use map() to implement loop
purrr::map_df(
  .x = boots$splits,
  .f = ~lm_get_coefs(training(.x)),
  .id = "iteration"
)
#> # A tibble: 500 x 3
#>   iteration intercept slope
#>   <chr>         <dbl> <dbl>
#> 1 1              6.47 -7.10
#> 2 2              7.21 -8.87
#> 3 3              8.03 -9.98
#> 4 4              6.18 -7.12
#> 5 5              6.65 -7.93
#> 6 6              7.41 -9.14
#> # i 494 more rows
```

6.7.3 Bootstrap Uncertainty in B-spline Fit

We will use the `bsp()` function from the `splines2` package for the model. I'll make a function `sp_predict()` that will fit a set of B-spline models (of varying complexity) to the training data, make predictions on the evaluation data.

```
library(splines2)  # for the bs() function

#: function to fit bspline and predict
sp_predict <- function(data_fit, data_eval, df = 5){
  fmla = "y ~ splines2::bsp(x, df=df, Boundary.knots=c(-.2, 1.2)) - 1"
  m = lm(as.formula(fmla), data=data_fit)
  data_eval %>% mutate(yhat = predict(m, .))
}
```

Now we can get predictions from all bootstrap fits:

```
data_eval = tibble(x=seq(0, 1, length=300)) # evaluation points

map_df(
  .x = boots$splits[1:50],
  .f = ~sp_predict(training(.x), data_eval),
  .id = "iter"
)
#> # A tibble: 15,000 x 3
#>   iter        x  yhat
#>   <chr>   <dbl> <dbl>
#> 1 1      0       2.50
#> 2 1      0.00334  2.55
#> 3 1      0.00669  2.60
#> 4 1      0.0100   2.65
#> 5 1      0.0134   2.70
#> 6 1      0.0167   2.75
#> # i 14,994 more rows
```

6.7.4 Out-of-bag performance evaluation

This function fits a b-spline using `data_fit`, make predictions on `data_eval`, and evaluates (using MSE). Uses a set of effective degrees of freedom `df`.

```
library(splines2)  # for the bsp() function

# 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(-.2, 1.2)) {

  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 = "y ~ splines2::bsp(x, df=df_i, Boundary.knots=c(-.2, 1.2)) - 1"
    fit = lm(as.formula(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, n_eval = nrow(data_eval), mse = MSE) # output
}
```

```
results = map_df(
  .x = boots$splits,
  .f = ~sp_eval(training(.x), testing(.x)),
  .id = "iter"
)
# Note that I used the default values of df and knots specified in sp_eval().
```

```
results %>%
  group_by(df) %>%
  summarize(
    mean_mse_1 = sum(n_eval*mse) / sum(n_eval), # accounts for different n.oob
    mean_mse = mean(mse),
    se = sd(mse)/sqrt(n())
  ) %>%
  arrange(mean_mse)
#> # A tibble: 13 x 4
#>      df mean_mse_1 mean_mse      se
#>   <dbl>      <dbl>    <dbl>   <dbl>
#> 1     5       4.80     4.80 0.0465
#> 2     6       4.93     4.93 0.0488
#> 3     4       4.98     4.98 0.0468
#> 4     7       5.06     5.06 0.0506
#> 5     8       5.23     5.22 0.0559
#> 6     9       5.25     5.24 0.0585
#> # i 7 more rows
```

6.7.5 Using `tune_grid()`

The `tune` package (part of `tidymodels`) provides a way to fit multiple models (or tuning parameters) over a common set of resamples.

First, create a workflow. Setting `deg_free = tune()` to allow a search over this tuning parameter.

```
library(tidymodels)
library(splines2)  # for the step_spline_b() function

# recipe
spline_rec = recipe(y~x, data=data_train) %>%
    step_spline_b(x, deg_free = tune(),
                  degree = 3,
                  options = list(Boundary.knots=c(-.2, 1.2)))

# workflow
spline_wf = workflow() %>%
  add_recipe(spline_rec) %>%
  add_model(linear_reg(), formula = y~.-1)
  # have to add another formula in last line to remove intercept
```

Note: I expected to remove the intercept (e.g., using formula `y ~ . - 1`) in the `recipe()` function, but it doesn't let you do that. The trick to removing the intercept in tidymodels is to add another formula at the end of the workflow inside the `add_model(..., formula = {here add -1})`.

Now we call `tune_grid()` supplying the grid of deg_free to try.

```
tmp = tune_grid(
  object = spline_wf,
```

```
  resamples = boots,
  grid = expand_grid(deg_free = 3:15),
  control = control_resamples(verbose=FALSE)
)
```

Now we can get the metrics

```
tmp %>%
  collect_metrics(summarize = TRUE) %>%
  filter(.metric == "rmse") %>%
  arrange(.metric)
#> # A tibble: 13 x 7
#>   deg_free .metric .estimator  mean     n std_err .config
#>      <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
#> 1        3 rmse    standard    2.45   500  0.0113 Preprocessor01_Model1
#> 2        4 rmse    standard    2.22   500  0.0107 Preprocessor02_Model1
#> 3        5 rmse    standard    2.18   500  0.0109 Preprocessor03_Model1
#> 4        6 rmse    standard    2.21   500  0.0112 Preprocessor04_Model1
#> 5        7 rmse    standard    2.24   500  0.0115 Preprocessor05_Model1
#> 6        8 rmse    standard    2.27   500  0.0122 Preprocessor06_Model1
#> # i 7 more rows
```