7 Splines etc

In ISLR Chapter 7, we go beyond linearity and model the nonlinear effect of a predictor. Some choices (e.g., polynomials, step functions) may not be good. Good choices are natural splines, smoothing splines, and local regression. The effects across predictors may be put together additively as in generalized additive models (GAMs).

High-dimensional versions of splines and local regression can also be defined for two or more predictors.

7.1 ISLR 7.1 Polynomial regression

In R, `poly(x, d)` can generate a d-degree polynomial for a variable x as an $n \times d$ matrix. The default is to generate orthogonal columns, which is good for fitting models on them. To generate columns $x, x^2, \ldots, x^d$, use `poly(x, d, raw=T)`.

Polynomial regression with a high degree (> 4) is generally not recommended, because

- A high-degree polynomial is difficult to interpret.
- A polynomial with K degrees has the same degrees of freedom as a natural spline with $K + 1$ knots, but the latter has a lot better flexibility and interpretation (ISLR 7.4).
- Polynomials allow high leverage data to have too much impact on the result.
- Data at one end could influence the shape of the fitted curve on the other end.

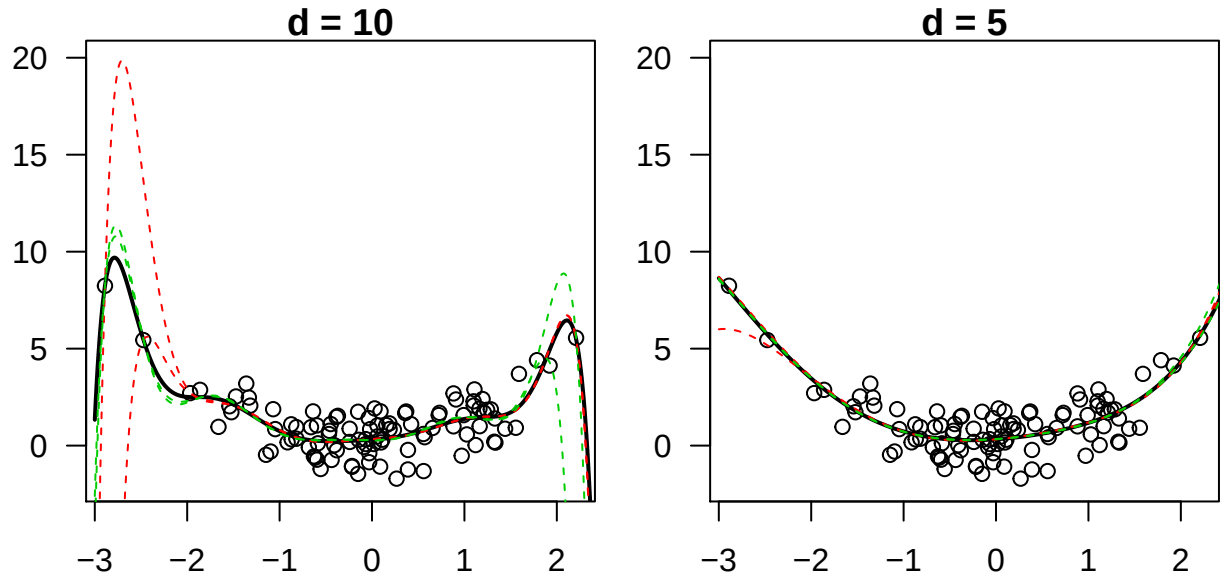
Below is an example to demonstrate the last two points. We first simulate some data.

```
set.seed(20)
n = 100
x = rnorm(n)
y = x^2 + rnorm(n)
xx = seq(-3, 3, 0.01)  ## a grid of x for drawing fitted curves
```

I will repeat the following twice, one for $d = 10$ and the other for $d = 5$. To make it easy to manage, I put these steps into a function.

```
myfun = function(x, y, d, nends=2) {
    modpoly = lm(y ~ poly(x, d))
    plot(x, y, ylim=c(-2, 20), main=paste("d =", d), las=1)
    points(xx, predict(modpoly, data.frame(x=xx)), type='l', lwd=2)

    plotremove = function(idx, col) {
        for(i in idx) {
            xtmp = x[-i]; ytmp = y[-i]
            predtmp = predict(lm(ytmp ~ poly(xtmp, d)), data.frame(xtmp=xx))
            points(xx, predtmp, type='l', lty=2, col=col)
        }
    }

    orderidx = order(x)
    plotremove(head(orderidx, nends), 2)  ## red
    plotremove(tail(orderidx, nends), 3)  ## green
}
```

```
par(mfrow=c(1,2), mar=c(2,2,1,1))
myfun(x, y, 10, 2)
myfun(x, y, 5, 2)
```



7.2 ISLR 7.2 Step functions

- Using a step function on a predictor results in a piecewise constant model.
  - Step functions allow nonlinear and non-monotonic effects (e.g., U-shape).
- A step function transforms a quantitative predictor into an ordered categorical variable.
  - Common in epidemiology. For example, even when age is available, "age group" is often used as an input variable in an analysis.
  - This can lead to both distortion and loss of information. For example, when using age group (by decade) as a predictor, we implicitly assume that ages 41 and 49 (8 years apart) have the same effect, while ages 49 and 51 (2 years apart) have different effects.
  - Assuming a constant effect over an interval we may miss a trend inside the interval (Figure 7.2 shows an example.)
  - Too few categories may over-simplify the relationship.
  - A step function with K categories has the same degrees of freedom as a natural spline with K knots, but the latter has a lot better flexibility and interpretation (ISLR 7.4).
- Step functions are "easy to interpret" due to oversimplification. This is similar to tree models.
  - Things always become "easy to interpret" if they are turned into black or white or if they are boiled down to a single number.
  - Categorization of a variable may be useful when reporting results. But it is often not helpful in analysis.
  - Dichotomization is the worst case of categorization.

In R, `cut()` can create a factor variable from an input variable. For example:

```
x = rnorm(100)
y = x^2 + rnorm(100)
lm(y ~ cut(x, breaks=seq(-4,4)))
```

7.3 ISLR 7.3 Basis functions

The **basis vectors** for a space are a set of vectors that serve as the basis to span the whole space through linear combination. For example, R^3, the 3-dimensional Euclidean space has basis vectors $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$, $e_3 = (0, 0, 1)$. Every point in R^3 can be written as a linear combination of them: $(x, y, z) = xe_1 + ye_2 + ze_3$. They are not unique. For example $f_1 = (\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0)$, $f_2 = (-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0)$, and $f_3 = (0, 0, 1)$ are a set of basis vectors.

Similarly, **basis functions** for a family of functions are a set of functions that span the whole family through linear combination. For example,

- Polynomials of degree d have basis functions $f_0(x) = 1, f_1(x) = x, \ldots, f_d(x) = x^d$, because any d-degree polynomial can be expressed as a linear combination of these functions: $a_0 + a_1 x + \cdots + a_d x^d = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_d f_d(x)$.
- Step functions over k intervals $[a_0, a_1), [a_1, a_2), \ldots, [a_{k-1}, a_k)$ have basis functions $f_1(x) = I(a_0 \le x < a_1)$, $f_2(x) = I(a_1 \le x < a_2)$, $\ldots$, $f_k(x) = I(a_{k-1} \le x < a_k)$.
- Basis functions are not unique. A family of functions can have several sets of basis functions.

7.4 ISLR 7.4 Regression splines

A **spline** is a piecewise polynomial function, with constraints at the **knots** to ensure the function is smooth. There are often three constraints at a knot t: (1) f is continuous at t, (2) first derivative of f is continuous at t, and (3) second derivative of f is continuous at t.

For example, at knot t, the interval to its left may be modeled with $a_1 x^3 + b_1 x^2 + c_1 x + d_1$, and the interval to its right may be modeled with $a_2 x^3 + b_2 x^2 + c_2 x + d_2$. Then the three constraints at t are

$$\begin{aligned} a_1 t^3 + b_1 t^2 + c_1 t + d_1 &= a_2 t^3 + b_2 t^2 + c_2 t + d_2, \\ 3a_1 t^2 + 2b_1 t + c_1 &= 3a_2 t^2 + 2b_2 t + c_2, \\ 6a_1 t + 2b_1 &= 6a_2 t + 2b_2. \end{aligned}$$

The rationale of these constraints is shown in ESL Figure 5.2.

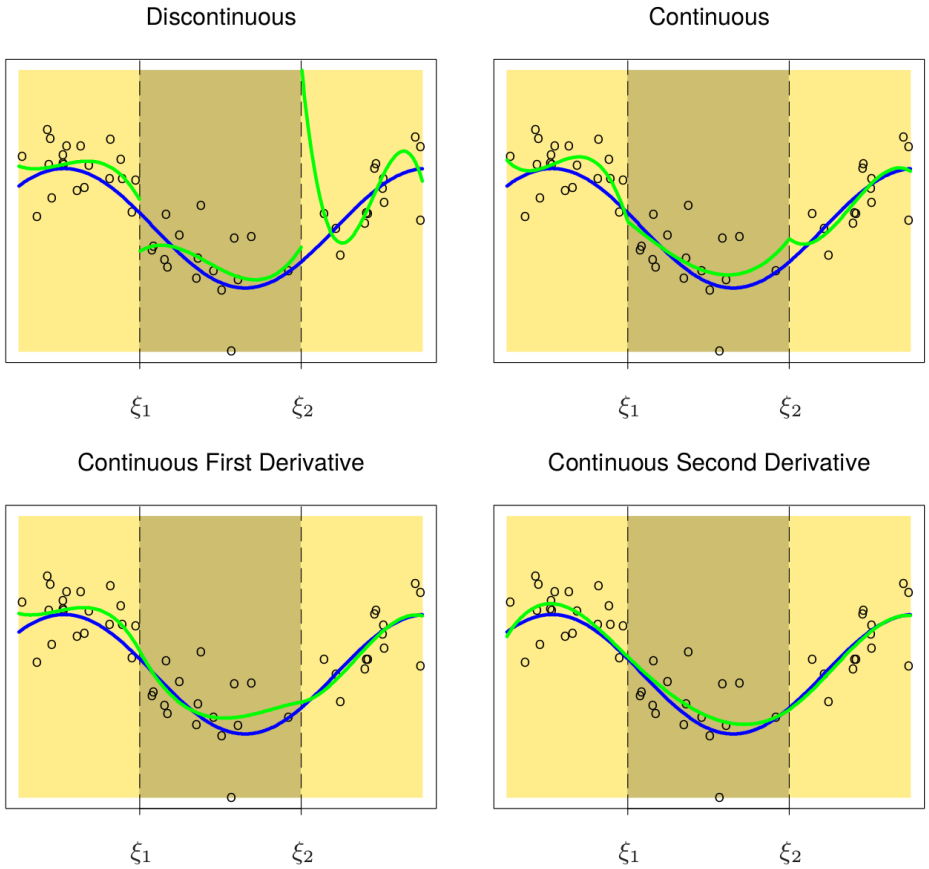


FIGURE 5.2. *A series of piecewise-cubic polynomials, with increasing orders of continuity.*

Cubic splines are splines that are piecewise cubic functions.

- A cubic spline with K knots has $K+1$ intervals. Each interval has 4 parameters (because a cubic function has 4 coefficients). There are 3 constraints at every knot. So, there are $4(K+1)-3K = K+4$ degrees of freedom (DFs), which include one DF for the intercept. That is, $K+3$ DFs are attributable to the variable.
- A set of basis functions are: 1, x, x^2, x^3, $(x-t_1)^3_+$, $\ldots$, $(x-t_K)^3_+$, where $t_1 < \ldots < t_K$ are the knots. The function $(x-t)^3_+$ is called *truncated power basis function* at t.
- B-splines are another set of basis functions (output of `splines::bs()`)
- In R function `splines::bs()`:
  - One can use `df=` to specify the DF attributable to the variable. For example, `bs(x, df=5)` will generate the 5 bases attributable to x (as an $n \times 5$ matrix), using 2 knots.
  - In `bs(x, df)`, by default, the $df-3$ knots are evenly spaced according to quantile. For example, `bs(x, df=5)` will have knots at the 33th and 66th percentiles.
  - One can instead specify the positions of the knots with the `knots=` argument.

```
library(splines)
attach(ISLR::Auto)

bs.mod1 = lm(mpg ~ bs(horsepower, df=5))
plot(horsepower, mpg, bty='n')
xx = seq(min(horsepower), max(horsepower), length.out=100)
points(xx, predict(bs.mod1, data.frame(horsepower=xx)), type='l', col=1, lwd=1)

summary(bs.mod1)$coef  ## not useful to look at
all.equal(bs(horsepower, df=5),
          bs(horsepower, knots=quantile(horsepower, probs=(1:2)/3)))  ## True

detach()
```

- In cubic splines, the internal intervals have constaints on both sides, but the two end intervals have constraints on only one side, leaving them too flexible. This is addressed by natural splines.

Natural splines (also called **restricted cubic splines**, RCS) are cubic splines but with linear functions at the two end intervals.

- A natural spline with K knots has 2 fewer parameters at each end interval than cubic splines. So, there are $4(K-1)+2\cdot 2-3K = K$ degrees of freedom, which include one DF for the intercept. That is, $K-1$ DFs are attributable to the variable.
- A set of basis functions are $N_1(x) = 1$, $N_2(x) = x$, and for $j = 1, \ldots, K-2$,

$$N_{j+2}(x) = (x-t_j)^3_+ - \frac{t_K - t_j}{t_K - t_{K-1}}(x-t_{K-1})^3_+ + \frac{t_{K-1}-t_j}{t_K-t_{K-1}}(x-t_K)^3_+.$$

- B-splines are another set of basis functions (output of `splines::ns()`)
- In R function `splines::ns()`:
  - One can use `df=` to specify the DF attributable to the variable. For example, `ns(x, df=5)` will generate the 5 bases attributable to x (as an $n \times 5$ matrix), using 6 knots.
  - In `ns(x, df)`, by default, the $df+1$ knots are $\min(x)$, $\max(x)$, and $df-1$ *internal knots* in between that are evenly spaced according to quantile. For example, `ns(x, df=5)` will have knots $\min(x)$, $\max(x)$, and 4 internal knots at the 20th, 40th, 60th, 80th percentiles.
  - One can instead specify the positions of the internal knots with the `knots=` argument.

```
library(splines)
attach(ISLR::Auto)

ns.mod1 = lm(mpg ~ ns(horsepower, df=5))
plot(horsepower, mpg, bty='n')
xx = seq(min(horsepower), max(horsepower), length.out=100)
points(xx, predict(ns.mod1, data.frame(horsepower=xx)), type='l', col=2, lwd=1)
```

```
summary(ns.mod1)$coef
all.equal(ns(horsepower, df=5),
          ns(horsepower, knots=quantile(horsepower, probs=1:4/5)))  ## True

detach()
```

Knot positions and number of knots:

- Ideally, knots should be placed at where the function may change rapidly.
- The number of knots is a hyperparameter, which can be selected through cross-validation, as shown in Figure 7.6.

It may not be desirable to have a visible linear portion at each end. This is addressed by the choice of boundary knots.

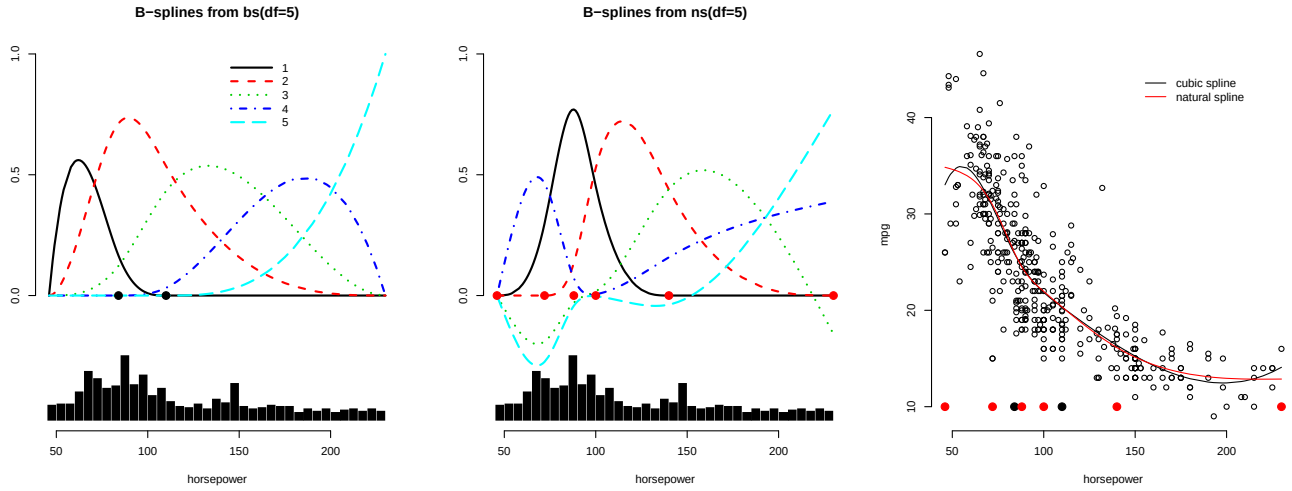
Boundary knots: In `bs()`, the `Boundary.knots = range(x)` argument provides two boundary knots for B-spline calculation, which has an effect on the resulting matrix, but does not have any effect on the resulting regression model. In `ns()`, the `Boundary.knots = range(x)` argument provides two real knots for defining the natural spline. This can be seen below.

```
library(splines)
attach(ISLR::Auto)

range(horsepower)  ## 46, 230
dim(bs(horsepower, knots=c(80, 120)))
dim(bs(horsepower, knots=c(80, 120), Boundary.knots=c(20,250)))
all.equal(bs(horsepower, knots=c(80, 120)),
          bs(horsepower, knots=c(80, 120), Boundary.knots=c(20,250)))  ## Not the same matrix

bs.mod1 = lm(mpg ~ bs(horsepower, knots=c(80, 120)))
bs.mod2 = lm(mpg ~ bs(horsepower, knots=c(80, 120), Boundary.knots=c(20,250)))
all.equal(predict(bs.mod1), predict(bs.mod2))  ## But the same model

ns.mod1 = lm(mpg ~ ns(horsepower, knots=c(80, 120)))
ns.mod2 = lm(mpg ~ ns(horsepower, knots=c(80, 120), Boundary.knots=c(20,250)))
all.equal(predict(ns.mod1), predict(ns.mod2))  ## Not the same model

detach()
```

To visualize the B-spline functions from `bs()` and `ns()`:

```
attach(ISLR::Auto)

library(splines)
idx = order(horsepower)
xhist = hist(horsepower, breaks=50, plot=FALSE)

bsmatrix = bs(horsepower, df=5)
matplot(horsepower[idx], bsmatrix[idx,], type='l', lwd=2, bty='n', ylim=c(-0.5,1), yaxt='n',
        xlab='horsepower', ylab='', main='B-splines from bs(df=5)')
axis(2, seq(0,1,.5))
points(quantile(horsepower, probs=1:2/3), c(0,0), col=1, pch=19, cex=1.5)
with(xhist, segments(mids, -0.5, mids, counts/200-0.5, lwd=8, lend=2))
legend(140, 1, col=1:5, lty=1:5, lwd=2, bty='n', seg.len=4, legend=1:5)

nsmatrix = ns(horsepower, df=5)
matplot(horsepower[idx], nsmatrix[idx,], type='l', lwd=2, bty='n', ylim=c(-0.5,1), yaxt='n',
        xlab='horsepower', ylab='', main='B-splines from ns(df=5)')
axis(2, seq(0,1,.5))
```

```
points(c(range(horsepower), quantile(horsepower, probs=1:4/5)), rep(0,6),
       col=2, pch=19, cex=1.5)
with(xhist, segments(mids, -0.5, mids, counts/200-0.5, lwd=8, lend=2))

bs.mod1 = lm(mpg ~ bs(horsepower, df=5))
ns.mod1 = lm(mpg ~ ns(horsepower, df=5))
plot(horsepower, mpg, bty='n')
points(horsepower[idx], fitted(bs.mod1)[idx], type='l', col=1)
points(horsepower[idx], fitted(ns.mod1)[idx], type='l', col=2)
points(quantile(horsepower, probs=1:2/3), c(10,10), col=1, pch=19, cex=1.5)
points(c(range(horsepower), quantile(horsepower, probs=1:4/5)), rep(10,6),
       col=2, pch=19, cex=1.5)
legend(150, 45, col=1:2, lty=1, bty='n',
       legend=c("cubic spline", "natural spline"))

detach()
```



Here both splines have 5 DFs attributable to the predictor. In the cubic spline, there are 2 knots. We fit 3 cubic functions for the 3 intervals, and put 3 constraints on each of the 2 knots. In the natural spline, there are 6 knots, two of which on the ends. Thus we fit 5 cubic functions for the 5 internal intervals and 2 linear functions for the two intervals outside of the two ends. Since the two end intervals do not have any data to support them, their existence effectively ensures that the cubic functions for the first and last internal intervals approach the ends in a mild way, with no change in the slope at around the ends (i.e., zero second derivatives at the ends).

7.5 ISLR 7.5

Smoothing splines start with a very different motivation. Consider all smooth functions $g(x)$ such that $g''(x)$ exists. We optimize the following

$$\text{minimize}_g \sum_i (y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt. \tag{7.11}$$

- A high $|g''(t)|$ reflects a quick change of $g'(t)$ at t, which happens when $g(t)$ is bumpy at t. So $\int g''(t)^2 dt$ is a way to measure the overall level of bumpiness/roughness of the function $g(t)$.
  - When $g(t) = \beta_0 + \beta_1 t$, $g''(t) = 0$ and $\int g''(t)^2 dt = 0$.
  - When $g(t) = \beta_0 + \beta_1 t + \beta_2 t^2$, $g''(t) = 2\beta_2$ and $\int_{x_{(1)}}^{x_{(n)}} g''(t)^2 dt = 4\beta_2^2 (x_{(n)} - x_{(1)})$. When $|\beta_2| = \frac{1}{\sqrt{8}} = 0.35$, the overall "roughness" of a quadratic function is similar to that of $\sin(t)$.

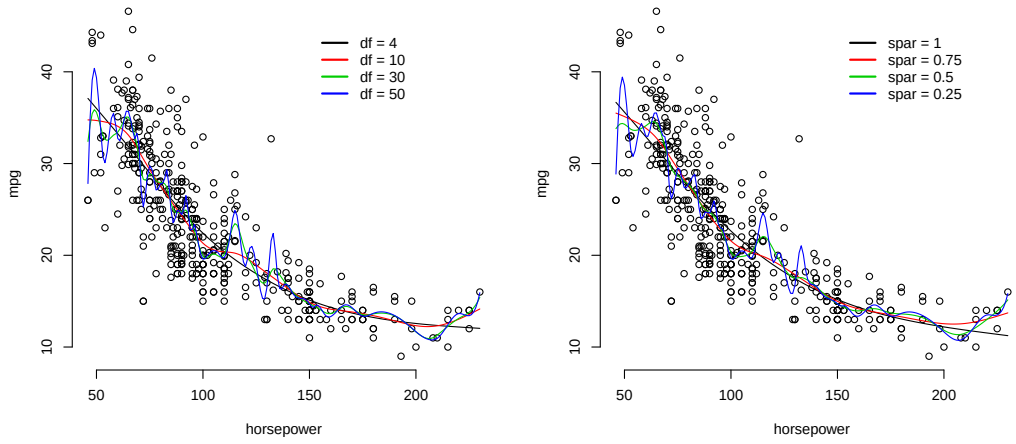
  - When $g(t) = \sin(t)$, $g''(t) = -\sin(t)$ and $\int_{x_{(1)}}^{x_{(n)}} g''(t)^2 dt = \int_{x_{(1)}}^{x_{(n)}} \sin^2(t) dt \approx \frac{1}{2}(x_{(n)} - x_{(1)})$. (The antiderivative of $\sin^2(t)$ is $\frac{1}{2}[t - \frac{1}{2}\sin(2t)]$.)
- It can be shown that the solution $g(x)$ to (7.11) must be a natural spline with all x_i being knots. So, **a smoothing spline is a regularized natural spline**. We thus can obtain a closed-form solution for (7.11) using the natural spline framework developed above.
- (7.11) is a version of generalized ridge regression.

Effective degrees of freedom: The closed-form solution leads to a formula for fitted values: $\hat{y}_\lambda = S_\lambda y$, where S_λ is an $n \times n$ matrix. The effective DF is defined as $\text{trace}(S_\lambda) = \sum_{i=1}^n \{S_\lambda\}_{ii}$. (Rationale: In linear regression, let X be the $n \times p$ design matrix. Then $\hat{y} = Hy$, where $H = X(X'X)^{-1}X'$. The trace of H is p, the degrees of freedom.)

- The effective DF may not be an integer. The value increases gradually as the penalty λ decreases.
- Instead of specifying λ, one can specify a desired effective DF or desired smoothness.

R has a function `smooth.spline(x, y)`. Note the order of `x` and `y`. Use `df=` to specify the desired effective DF, or `spar=` to specify the desired smoothness (a value in $(0, 1]$). The higher `df` the rougher function, and the lower `spar` the rougher function.

```
require(ISLR)
sms = with(Auto, smooth.spline(horsepower, mpg, df=4))
names(sms)

par(mfrow=c(1,2), cex=.7)
xgrid = seq(46,230)  ## for drawing fitted curves
colseq = 1:4

with(Auto, plot(horsepower, mpg, bty='n'))
dfseq = c(4, 10, 30, 50)
for(i in 1:4) {
  tmp = with(Auto, smooth.spline(horsepower, mpg, df=dfseq[i]))
  lines(predict(tmp, xgrid), col=colseq[i])
}
legend(150, 45, col=colseq, lty=1, lwd=2, bty='n', legend = paste("df =", dfseq))

with(Auto, plot(horsepower, mpg, bty='n'))
sparseq = (4:1)/4
for(i in 1:4) {
  tmp = with(Auto, smooth.spline(horsepower, mpg, spar=sparseq[i]))
  lines(predict(tmp, xgrid), col=colseq[i])
}
legend(150, 45, col=colseq, lty=1, lwd=2, bty='n', legend = paste("spar =", sparseq))
```



Choosing hyperparameter lambda:

- Cross-validation always works. This is a computational approach.
- Traditional formula-based CV approaches: Let S_λ be the matrix for λ such that $\hat{y}_\lambda = S_\lambda y$.
  - Leave-one-out CV (LOOCV): Select λ that minimizes $\text{err}_\lambda = \sum_i (y_i - \hat{y}_{i|-i})^2 = \sum_i \left[\frac{y_i - \hat{y}_{\lambda,i}}{1-\{S_\lambda\}_{ii}}\right]^2$.
  - Generalized CV (GCV): Select λ that minimizes $V(\lambda) = \frac{\frac{1}{n}\sum (y_i - \hat{y}_{\lambda,i})^2}{[\frac{1}{n} tr(I-S_\lambda)]^2}$. (Motivation: $V(\lambda) = \frac{1}{n}\sum_i (y_i - \hat{y}_{i|-i})^2 w_i^2(\lambda)$, where $w_i(\lambda) = \frac{1-\{S_\lambda\}_{ii}}{\frac{1}{n} tr(I-S_\lambda)}$. Note that $\frac{1}{n}\sum_i w_i(\lambda) = 1$.)

In `smooth.spline()`, when both `df` and `spar` are not specified, a formula-based CV will be performed. The default (`cv=F`) is the GCV. When `cv=T`, the LOOCV is performed. The document of `smooth.spline()` suggests to use GCV when there are duplicate values in `x`, which is the case in the example below.

For regression analysis of `mpg` on `horsepower`, these two versions of CV give quite different results.

```
library(ISLR)
sms.cvT = with(Auto, smooth.spline(horsepower, mpg, cv=T))  ## LOOCV
sms.cvF = with(Auto, smooth.spline(horsepower, mpg, cv=F))  ## GCV, the default

sms.cvT$df  ## 5.78
sms.cvF$df  ## 42.15
sms.cvT$lambda; sms.cvF$lambda  ## very different lambda
sms.cvT$lambda / sms.cvF$lambda
```

The two models are quite different.

```
xgrid = seq(46,230)  ## for drawing fitted curves
with(Auto, plot(horsepower, mpg, bty='n'))
lines(predict(sms.cvT, xgrid), col=1)
lines(predict(sms.cvF, xgrid), col=2)
```

The result from 10-fold cross-validation is consistent with that of the GCV.

```
cv.part = function(n, K) {
    s1 = n %/% K; s2 = n %% K
    pos2 = ifelse(rep(s2==0,K), (1:K)*s1, (1:K)*s1 + c(rep(0,K-s2), 1:s2))
    pos1 = c(1, pos2[-K]+1)
    part = sample(n,n)
    out=list()
    for(k in 1:K) out[[k]] = part[pos1[k]:pos2[k]]
    out
}

cv.ss = function(x, y, dfrange, K) {
  n = length(y)
  part = cv.part(n, K)

  RSS = NULL
  for(dd in 1:length(dfrange)) {
    rss = 0
    for(ii in 1:K) {
      idxtest = part[[ii]]
      tmp = smooth.spline(x[-idxtest], y[-idxtest], df=dfrange[dd])
      rss = rss + sum((predict(tmp, x[idxtest])$y - y[idxtest])^2)
    }
    RSS[dd] = rss
  }
  names(RSS) = dfrange
```

```
  dfminrss = min(dfrange[which(RSS == min(RSS))])
  list(RSS=RSS, df=dfminrss)
}

library(ISLR)
set.seed(201)
dfseq = round(exp(seq(1, 4, 0.02)), 2)
res = cv.ss(Auto$horsepower, Auto$mpg, dfseq, 10)
res$df  ## 39.65
plot(dfseq, res$RSS, type='l')
```

7.6 ISLR 7.6

Local regression: Moving-window weighted regression.

- Similar to kNN. Both are *memory-based* (in contrast to *formula-based*), because the training data are needed when computing a prediction. (Or, a fine grid of results need to be stored.)
- Things to specify:
  - **Span**: Fraction of data used for every point x_0.
  - **Kernel** (weight function): How data are weighted (as a function of relative distance between x and x_0).
  - **Regression model** (and fitting criterion)
- All these factors are hyperparameters and can be selected using CV, although in practice they are often specified by users.

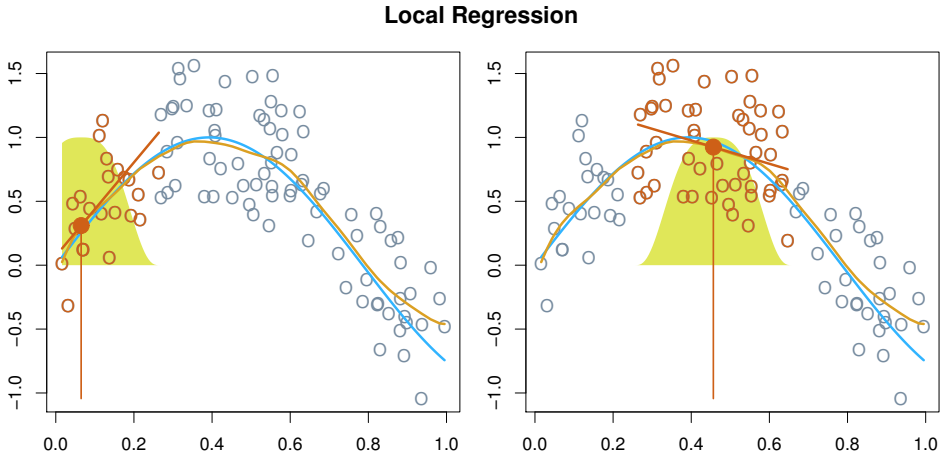


Figure 7.9 and Algorithm 7.1 in ISLR describe local *linear* regression (i.e., degree=1). In the R function `loess()`, the default is local *quadratic* model fitted with least squares (`degree=2` and `family="gaussian"`).

In `loess()`, the default neighborhood is `span = 0.75` (75% neighboring data are used for every x_0), which yields a very smooth function but which may not be what you want. One can also specify the desired DF (approximate "equivalent number of parameters") with the `enp.target=` option. When doing moving average (`degree=0`), span should be much smaller than the default of 0.75.

```
attach(ISLR::Auto)
aa = loess(mpg ~ horsepower)
aa  ## Note the ENP is provided
summary(aa)

idx = order(horsepower)
plot(horsepower, mpg)
lines(horsepower[idx], predict(loess(mpg ~ horsepower))[idx], col=1)   ## quadratic
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=1))[idx], col=2)  ## linear
```

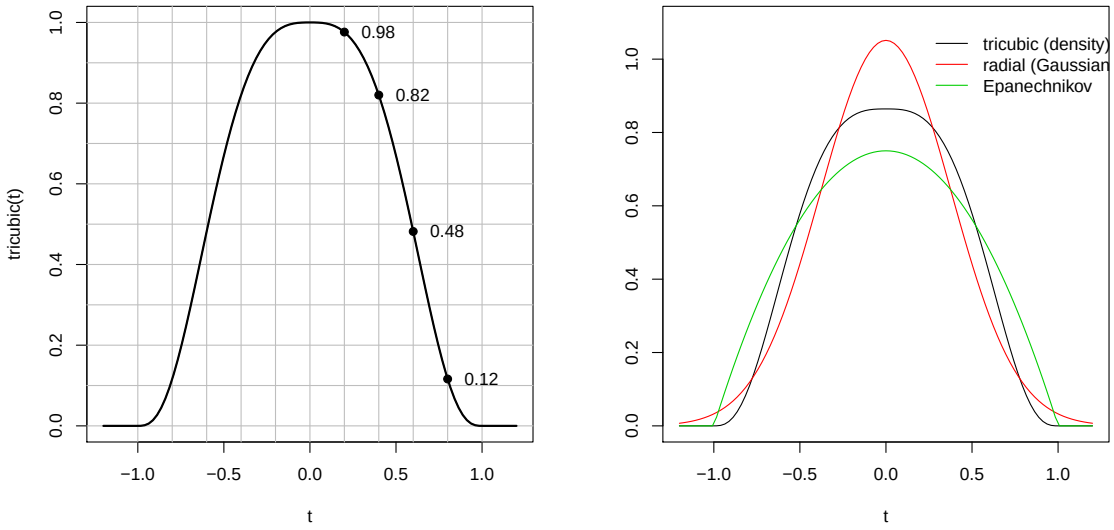
```
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=0))[idx], col=3)  ## span=.75 too large
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=0, span=.2))[idx], col=3, lty=2)
lines(horsepower[idx], predict(loess(mpg ~ horsepower, enp.target=10))[idx], col=1, lty=2)
detach()
```

Note that `loess()` is prefered to the older version `lowess()`. These functions have different defaults.

High-dimensional loess: The default `span=0.75` is often too high. For example, suppose x_1 and x_2 are two predictors with values uniformly distributed in $[0,1]$. A span of 75% neighboring points in the space of $[0,1] \times [0,1]$ is effectively a span of 86.6% on x_1 and x_2 (because $0.866^2 = 0.75$). For 3-dimensional predictors, a span of 75% in 3D is effectively a span of 90.9% on each predictor (because $0.909^3 = 0.75$).

Tricubic function: The weight function used in `loess()` is tricubic: $f(t) = (1 - |t|^3)^3 I(|t| \leq 1)$, where $t = dist(x, x_0)/\max(d)$ is the relative distance of x from x_0, and $\max(d)$ is the maximum distance from x_0 in the neighborhood of x_0. The left figure is the tricubic function drawn with the following code.

```
tricubic = function(x) (1 - abs(x)^3)^3 *(abs(x)<=1)
curve(tricubic(x), xlim=c(-1.2,1.2), xlab='t', ylab='tricubic(t)')
abline(h=seq(0,1,0.1), v=seq(-1,1,0.2), col='grey')
curve(tricubic(x), lwd=2, add=T)
xgrid = seq(.2, .8, .2)
points(xgrid, tricubic(xgrid), pch=19)
text(xgrid, tricubic(xgrid), round(tricubic(xgrid), 2), adj=-.5)
```



The right figure shows the comparison with two other kernel functions: Gaussian and the Epanechnikov kernel $h(t) = .75(1 - t^2)I(|t| \leq 1)$. The tricubic function has AUC 1.157. We draw a rescaled version $f(t)/1.157$ so that its AUC is 1 (a density function with mean 0 and variance 0.144). The Epanechnikov kernel has AUC 1 (a density function with mean 0 and variance 0.2). The Epanechnikov kernel is not smooth at -1 and 1. We also draw the Gaussian kernel $N(0, 0.144)$.

```
integrate(tricubic, -1, 1)  ## AUC is 1.157143
integrate(function(x) x^2 * tricubic(x)/1.157143, -1, 1)  ## VAR is 0.1440329

curve(tricubic(x)/1.157143, xlim=c(-1.2, 1.2), ylim=c(0,1.1), xlab='t', ylab='')
curve(dnorm(x, 0, sqrt(0.1440329)), add=T, col=2)  ## Gaussian kernel with same variance
curve(.75*(1-x^2) *(abs(x)<=1), add=T, col=3)  ## Epanechnikov kernel
legend(1,1, col=1:3, lty=1, bty='n',
       legend=c("tricubic", "radial (Gaussian)", "Epanechnikov"))
```

7.7 ISLR 7.7 Generalized additive models (GAMs)

$$y = \beta_0 + f_1(x_1) + \cdots + f_p(x_p) + \epsilon, \tag{7.15}$$

where $f_1, \ldots, f_p$ can be different functions with different levels of smoothness.

- Easy to interpret.
- One can plan on the DFs spent on the predictors.
- Can be fit using backfitting (or LS when the functions are explicit).
- Flexible modeling of the effects of individual predictors. The functions can be global or local or piecewise. A function can also be 2-dim over two predictors or 3-dim over three predictors.

Backfitting is an iterative algorithm for fitting additive models. For example, suppose our model is $y = \beta_0 + f_1(x_1) + f_2(x_2) + f_3(x_3) + \epsilon$. Given the current estimates $\hat{\beta}_0$, $\hat{f}_1$, and $\hat{f}_2$, we calculate partial residuals $r_i = y_i - \hat{\beta}_0 - \hat{f}_1(x_i) - \hat{f}_2(x_i)$ and then fit r_i to $f_3(x_i)$ to obtain a new estimate $\hat{f}_3$. We then repeat this process to estimate another component in the model. Repeat several cycles until convergence.

The R `gam` package has the function `gam()`. In `gam()`, a smoothing spline on a predictor `x` is specified through `s(x)`, and a loess fit is specified through `lo(x)`. Other basis generators such as `ns()`, `bs()`, and `poly()` can be used. Traditional model terms are also allowed, such as `x` (linear effect if `x` is quantitative, or categorical effect if `x` is qualitative), `I(x>10)`, and interaction term `x1*x2`, etc.

```
library(gam)
library(ISLR)
gam.m3 = gam(wage ~ s(year,4) + s(age,5) + education, data=Wage)
names(gam.m3)
```

The `summary()` gives two regression tables. The table for "parametric effects" is for the traditional regression terms, the explicitly expressed terms from `poly()`, `bs()` and `ns()`, and the linear portion of the non-traditional terms from `s()` and `lo()`. The table for "nonparametric effects" is for the non-linear portion of the non-traditional terms.

```
summary(gam.m3)
gam.m3$coefficients
```

The model can be plotted nicely.

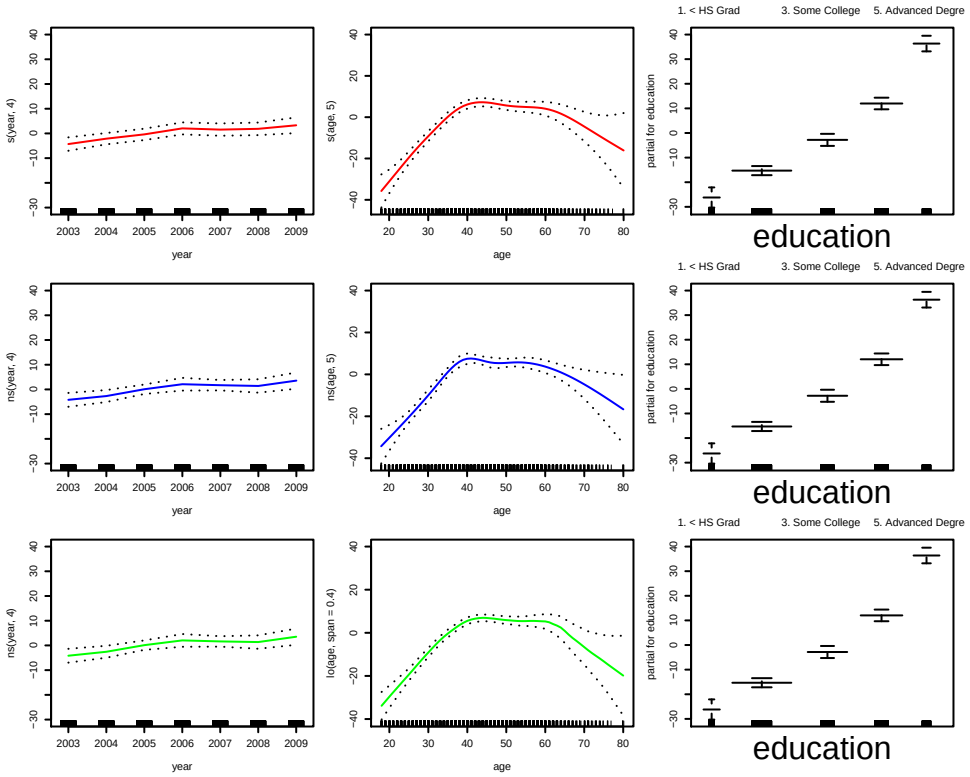
```
par(mfrow=c(1,3))
plot(gam.m3, se=T, col="blue", ylim=c(-30,40))  ## Figure 7.12
```

We can test for difference between nested models using `anova()`.

```
gam.m1 = gam(wage ~ s(age,5) + education, data=Wage)          ## no year effect
gam.m2 = gam(wage ~ year + s(age,5) + education, data=Wage)  ## linear in year
anova(gam.m1, gam.m2, gam.m3, test="F")
```

Compare `s()` with `ns()` and `lo()`. They give similar results.

```
require(gam)
library(ISLR)
gam.m3 = gam(wage ~ s(year,4) + s(age,5) + education, data=Wage)
gam.m3b = gam(wage ~ ns(year,4) + ns(age,5) + education, data=Wage)
gam.m4 = gam(wage ~ ns(year,4) + lo(age,span=0.4) + education, data=Wage)

par(mfrow=c(3,3), mar=c(4,4,2,1), cex=.3, cex.main=.2)
plot(gam.m3, se=T, col="red", ylim=c(-30,40))  ## Figure 7.12
plot(gam.m3b, se=T, col="blue", ylim=c(-30,40))  ## Figure 7.11
plot(gam.m4, se=T, col="green", ylim=c(-30,40))
```



A 2-dimensional loess fit (to capture 2D interaction) can be specified:

```
gam.lo.i = gam(wage ~ lo(year, age, span=0.25) + education, data=Wage)
par(mfrow=c(1,2))
plot(gam.lo.i)
```

The plot for `lo(year, age)` is not informative. We redraw it using the `rgl` package.

```
bb = preplot(gam.lo.i)
str(bb)

library(rgl)
with(bb[[1]], plot3d(x[[1]], x[[2]], y))
```

Traditional interaction between two variables such as `x1*x2` looks okay.

```
gam.m5 = gam(wage ~ s(age,5) + year * education, data=Wage)
anova(gam.m2, gam.m5, test="F")  ## df = 4, as expected
```

But "interaction with a spline" can be misleading. Here, `s(age,5) * year` means the traditional `age*year` (i.e., a term for `age`, a term for `year`, and their product term) plus the 4 addition terms for the non-linear portion of the smoothing spline for `age`.

```
gam.m6 = gam(wage ~ s(age,5) * year + education, data=Wage)
summary(gam.m6)
anova(gam.m2, gam.m6, test="F")  ## df = 1
```

7.8 Assignment

1. **Homework**: ISLR Chapter 7 Exercises 9