Data-based Statistical Decision Model

Lecture 8 - Smoothing methods

Sungkyu Jung

Recall the simple linear regression

From $Y = m(X) + \epsilon$, $m(X) = \beta_0 + \beta_1 X$, and a given data set,

$$\hat{\beta}_1 = \frac{c_{XY}}{s_X^2} = \sum_i \frac{(x_i - \bar{x})}{s_X^2} y_i,$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.$$

Thus, the conditional mean estimate of Y at $X = x$ is:

$$\begin{aligned}
\hat{m}(x) &= \hat{\beta}_0 + \hat{\beta}_1 x \\
&= \bar{y} + \hat{\beta}_1 (x - \bar{x}) \\
&= \sum_i \frac{1}{n} y_i + \sum_i \frac{(x_i - \bar{x})(x - \bar{x})}{s_X^2} y_i \\
&= \sum_i \left(\frac{1}{n} + \frac{(x_i - \bar{x})(x - \bar{x})}{s_X^2} \right) y_i
\end{aligned}$$

Linear smoothers

- From the above, we can see that the linear regression prediction is of the form

$$m(x) = \sum_{i=1}^{n} w(x, x_i) \cdot y_i$$

with $w(x, x_i) = \frac{1}{n} + \frac{(x_i - \bar{x})(x - \bar{x})}{s_X^2}$.

In general, we call a regression function of this form a linear smoother (note this is so-named because it is linear in y, but need not behave linearly as a function of x!)

K-nearest-neighbors regression

- Another common, more flexible linear smoother is *k-nearest-neighbors regression*; for this we take

$$w(x, x_i) = \begin{cases} 1/k, & \text{if } x_i \text{ is one of the } k \text{ nearest points to } x; \\ 0, & \text{otherwise.} \end{cases}.$$

- In other words

$$\hat{m}(x) = \frac{1}{k} \sum_{i \in N_k(x)} y_i,$$

where $N_k(x)$ gives the k nearest neighbors of x

- To use this in practice, we're going to need to choose k. What are the tradeoffs at either end? For small k, we risk picking up noisy features of the sample that don't really have to do with the true regression function $m(x)$, called *overfitting*; for large k, we may miss important details, due to averaging over too many y_i values, called *underfitting*.

Quantifying the amount of over/underfitting

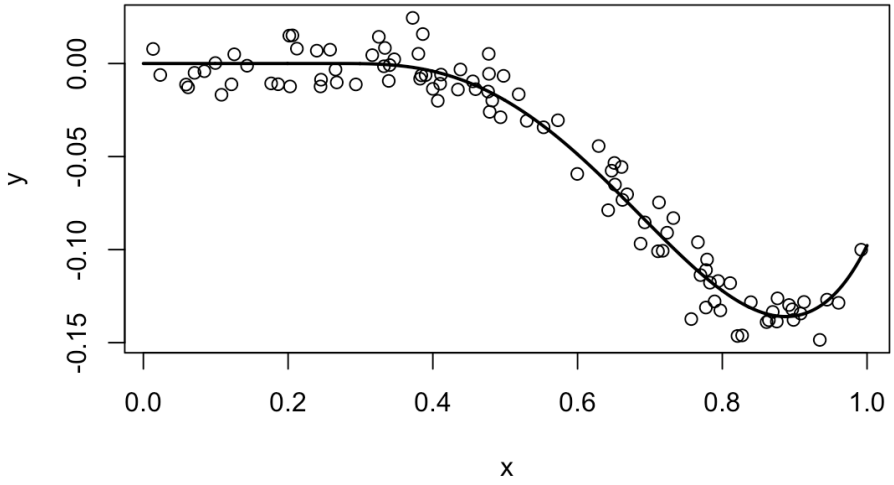
- How are we going to quantify this (overfitting vs underfitting)?
- Use Training MSE!
  - Fit $\hat{m}$ for a choice of k, using training data;
  - Evaluate the out-of-sample MSE, using testing data ((x_i', y_i')).
- We could then look at the expected test error,

$$E(\mathrm{TestErr}^2(\hat{m})) = E(\frac{1}{n}\sum_i (y_i' - \hat{m}(x_i'))^2)$$

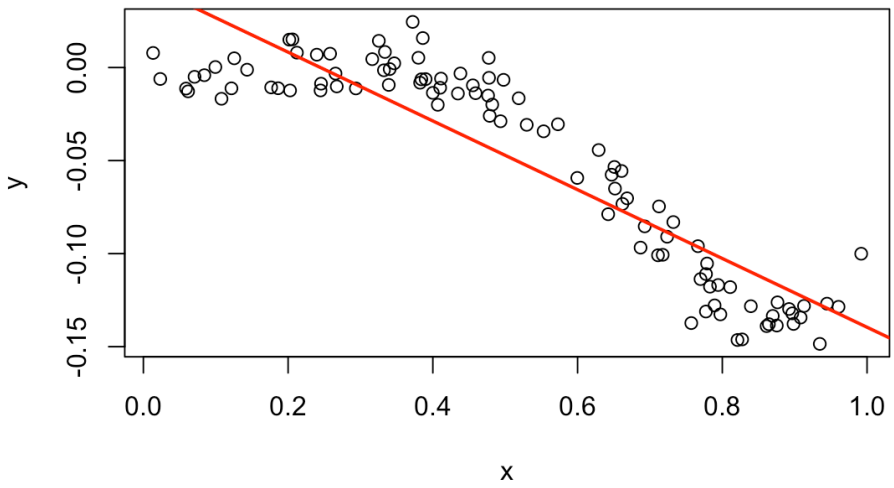
Note that the expectation here is taken over all that is random (both training and test samples). This really does capture what we want, and has the right behavior with k.

A toy data example

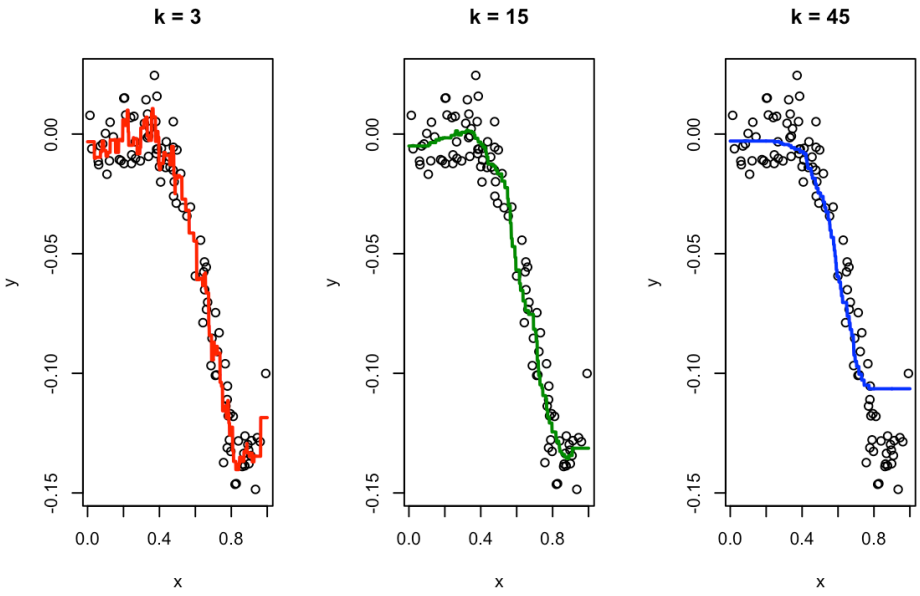
```
load("nonlin.Rdata")

# Plot the first training set
x = xtrain[,1]
y = ytrain[,1]
plot(x,y)
lines(x0,r0,lwd=2)
```



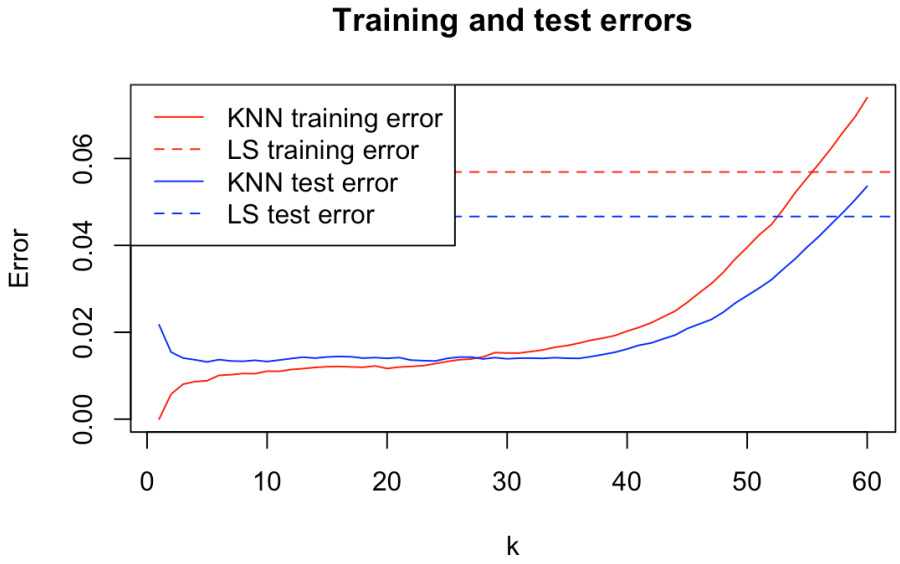
```
# Look at linear regression the first training set
linmodel = lm(y~x)
plot(x,y)
abline(a=linmodel$coef[1],b=linmodel$coef[2],col="red",lwd=2)
```




```
# Look at knn regression on the first training set
library(FNN)
ks = c(3,15,45)
knnmodel1 = knn.reg(train=matrix(x,ncol=1),test=matrix(x0,ncol=1),y=y,k=ks[1])
knnmodel2 = knn.reg(train=matrix(x,ncol=1),test=matrix(x0,ncol=1),y=y,k=ks[2])
knnmodel3 = knn.reg(train=matrix(x,ncol=1),test=matrix(x0,ncol=1),y=y,k=ks[3])

par(mfrow=c(1,3))
plot(x,y,main=paste("k =",ks[1]))
lines(x0,knnmodel1$pred,col="red",lwd=2)
plot(x,y,main=paste("k =",ks[2]))
lines(x0,knnmodel2$pred,col="green4",lwd=2)
plot(x,y,main=paste("k =",ks[3]))
lines(x0,knnmodel3$pred,col="blue",lwd=2)
```



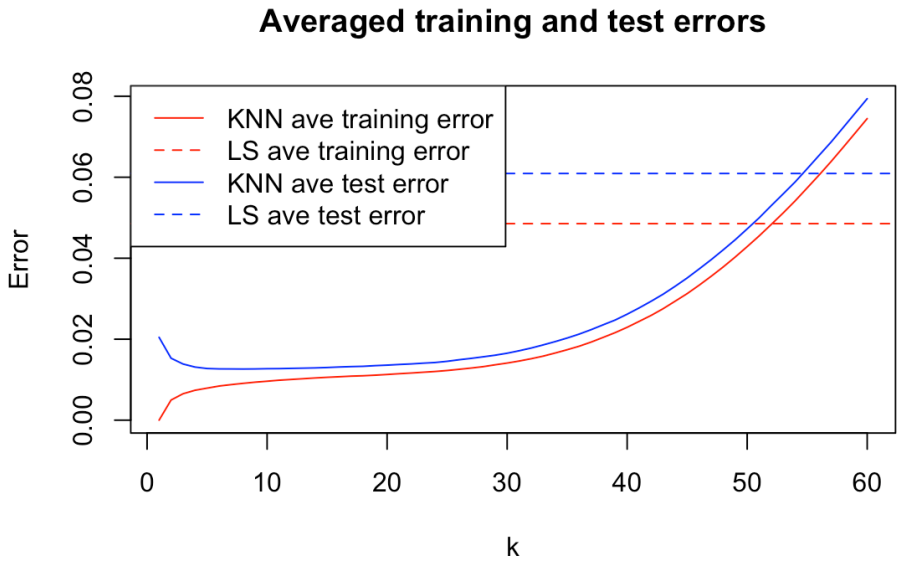
```
# Examine training and test errors
errtrain.lin = sum((y - linmodel$fitted)^2)
errtest.lin = sum((ytest[,1] - predict(linmodel,newx=xtest[,1]))^2)

nk = 60
ks = 1:nk
errtrain.knn = errtest.knn = numeric(nk)
for (i in 1:nk) {
  knnmodel = knn.reg(matrix(x,ncol=1),matrix(x,ncol=1),y=y,k=ks[i])
  errtrain.knn[i] = sum((y-knnmodel$pred)^2)
  knnmodel = knn.reg(matrix(x,ncol=1),matrix(xtest[,1],ncol=1),y=y,k=ks[i])
  errtest.knn[i] = sum((ytest[,1]-knnmodel$pred)^2)
}

ylim = range(c(errtrain.knn,errtest.knn))

par(mfrow=c(1,1))
plot(ks,errtrain.knn,type="l",ylim=ylim,col="red",
     main="Training and test errors",xlab="k",ylab="Error")
abline(h=errtrain.lin,lty=2,col="red")
lines(ks,errtest.knn,col="blue")
abline(h=errtest.lin,lty=2,col="blue")
legend("topleft",col=c("red","red","blue","blue"),lty=c(1,2,1,2),
       legend=c("KNN training error","LS training error",
                "KNN test error","LS test error"))
```



```
# Average these results over the 50 simulations
p = 50
errtrain.lin.all = numeric(p)
errtest.lin.all = numeric(p)
errtrain.knn.all = matrix(0,nk,p)
errtest.knn.all = matrix(0,nk,p)
for (j in 1:p) {
  cat(paste(j,",",sep=""))
  x = xtrain[,j]
  y = ytrain[,j]

  linmodel = lm(y~x)
  errtrain.lin.all[j] = sum((y - linmodel$fitted)^2)
  errtest.lin.all[j] = sum((ytest[,j] - predict(linmodel,newx=xtest[,j]))^2)

  for (i in 1:nk) {
    knnmodel = knn.reg(matrix(x,ncol=1),matrix(x,ncol=1),y,k=ks[i])
    errtrain.knn.all[i,j] = sum((ytrain[,j]-knnmodel$pred)^2)
    knnmodel = knn.reg(matrix(x,ncol=1),matrix(xtest[,j],ncol=1),y,k=ks[i])
    errtest.knn.all[i,j] = sum((ytest[,j]-knnmodel$pred)^2)
  }
}
```

```
## 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,
37,38,39,40,41,42,43,44,45,46,47,48,49,50,
```

```
errtrain.lin.ave = mean(errtrain.lin.all)
errtest.lin.ave = mean(errtest.lin.all)
errtrain.knn.ave = rowMeans(errtrain.knn.all)
errtest.knn.ave = rowMeans(errtest.knn.all)
ylim = range(c(errtrain.knn.ave,errtest.knn.ave))

par(mfrow=c(1,1))
plot(ks,errtrain.knn.ave,type="l",ylim=ylim,col="red",
     main="Averaged training and test errors",xlab="k",ylab="Error")
abline(h=errtrain.lin.ave,lty=2,col="red")
lines(ks,errtest.knn.ave,col="blue")
abline(h=errtest.lin.ave,lty=2,col="blue")
legend("topleft",col=c("red","red","blue","blue"),lty=c(1,2,1,2),
       legend=c("KNN ave training error","LS ave training error",
                "KNN ave test error","LS ave test error"))
```



Smoother linear smoothers

- In k-nearest-neighbors regression, the weights $w(x, x_i)$ are either $1/k$ or 0, depending on x_i. This will often produce jagged looking fits. Why? Are these weights smooth over x?
- How about choosing

$$w(x, x_i) = \frac{\exp(-(x - x_i)^2/(2h))}{\sum_{j=1}^{n} \exp(-(x - x_j)^2/(2h))}?$$

- This is an example of kernel regression, using what's called a Gaussian kernel.
- The parameter h here is called the bandwidth, like k in k-nearest-neighbors regression, it controls the level of adaptivity/flexibility of the fit.
- One reason to prefer kernel regression is that it can produce smoother (less jagged) looks fits than k-nearest-neighbor regresison.

Kernel regression

What is a kernel?

A kernel is a non-negative real-valued integrable function K. For most applications, it is desirable to define the function to satisfy two additional requirements:

1. Normalization:

$$\int_{-\infty}^{+\infty} K(u)\, du = 1\,;$$

2. Symmetry:

$$K(-u) = K(u) \text{ for all values of } u\,.$$

—

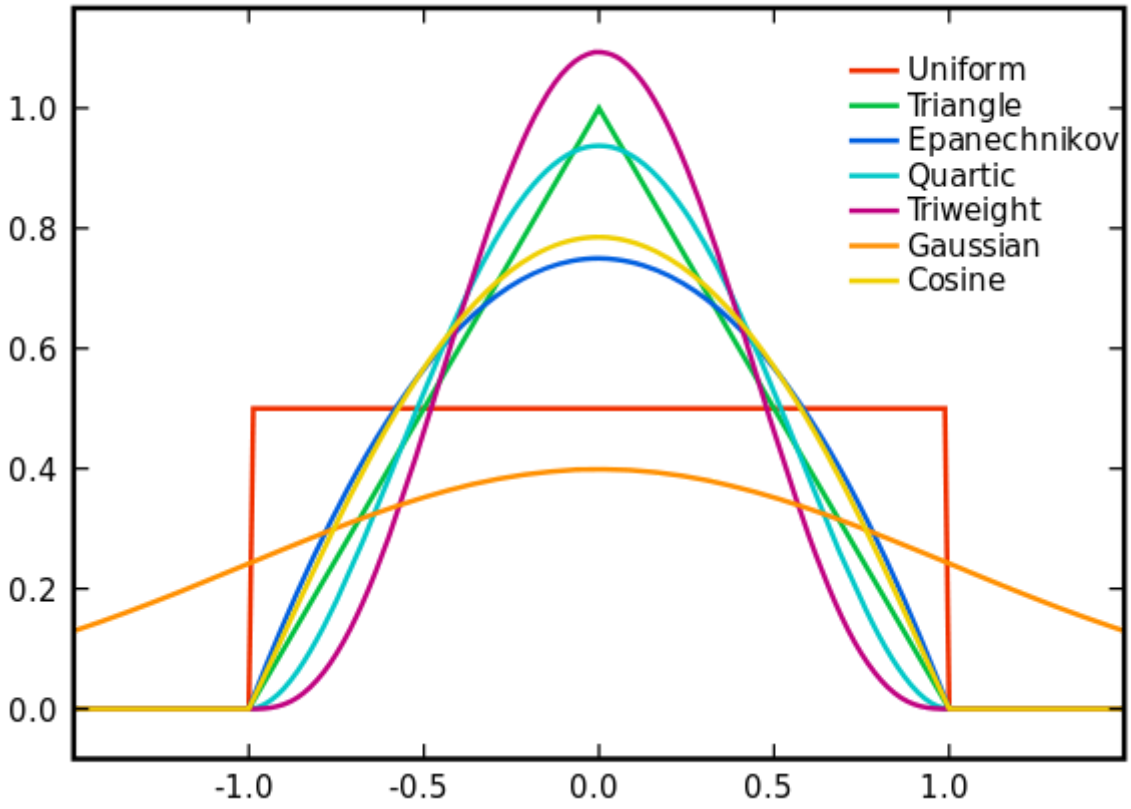
Most often used kernel is the Gaussian kernel:

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}$$

(It's just the density function of $N(0, 1)$ distribution.)

There are many other commonly-used kernels

- See https://en.wikipedia.org/wiki/Kernel_(statistics) (https://en.wikipedia.org/wiki/Kernel_(statistics)) for a list.



Kernel regression from a choice of a kernel

Given a choice of kernel K, and a bandwidth h, kernel regression is defined by taking

$$w(x, x_i) = \frac{K(\frac{x-x_i}{h})}{\sum_{j=1}^{n} K(\frac{x-x_j}{h})}$$

in the linear smoother form (1). In other words, the kernel regression estimator is

$$\hat{m}(x) = \sum_{i=1}^{n} w(x, x_i) y_i = \frac{\sum_{i=1}^{n} K(\frac{x-x_i}{h}) y_i}{\sum_{j=1}^{n} K(\frac{x-x_j}{h})}$$

- What is this doing? This is a weighted average of y_i values. Think about laying doing a Gaussian kernel around a specific query point x, and evaluating its height at each x_i in order to determine the weight associate with y_i. (It's just a *moving average*!)

- Because these weights are smoothly varying with x, the kernel regression estimator $\hat{m}(x)$ itself is also smoothly varying with x
- What's in the choice of kernel? Different kernels can give different results. But many of the common kernels tend to produce similar estimators; e.g., Gaussian vs. Epanechnikov, there's not a huge difference.
- A much bigger difference comes from choosing different bandwidth values h. What's the tradeoff present when we vary h? Hint: as we've mentioned before, you should always keep these two quantities in mind …

Biad and variance of a kernel regression predictor

- At a fixed query point x, the expected test error has a fundamental decomposition:

$$\begin{aligned} E[\text{TestErr}(\hat{m}(x))] &= E[(Y - \hat{m}(X))^2 \mid X = x] \\ &= \sigma^2 + \text{Bias}^2(\hat{m}(x)) + Var(\hat{m}(x)). \end{aligned}$$

  So what is the bias and variance of the kernel regression estimator?

- Under some smoothness assumptions on the true m,

$$\text{Bias}^2(\hat{m}(x)) = [E(\hat{m}(x)) - m(x)]^2 \leq C_1 h^2,$$

and

$$Var(\hat{m}(x)) \leq \frac{C_2}{nh}.$$

for some constants C_1 and C_2. Does this make sense? What happens to the bias and variance as h shrinks? As h grow?

- This means that

$$E[\text{TestErr}(\hat{m}(x))] = \sigma^2 + C_1 h^2 + \frac{C_2}{nh},$$

  which is minimized by choosing $h = \frac{C_2}{2C_1 n^{1/3}}$. Is this is a realistic choice for the bandwidth?

Practical considerations

- In practice, we tend to select h by, you guessed it, cross-validation.
- For simple dataset, we can use a rule-of-thumb selection of h.
- Kernels can actually suffer bad bias at the boundaries … why? Think of the asymmetry of the weights

---

Kernel regression in multiple dimensions

- In multiple dimensions, say, each $x_i \in R^p$, we can easily use kernels, we just replace $x_i - x$ in the kernel argument by $\|x_i - x\|_2$, so that the multivariate kernel regression estimator is

$$\hat{m}(x) = \frac{\sum_{i=1}^{n} K(\frac{\|x_i - x\|_2}{h}) y_i}{\sum_{j=1}^{n} K(\frac{\|x_i - x\|_2}{h})}$$

- The same calculations as those that went into producing the bias and variance bounds above can be done in this multivariate case, showing that

  $$\text{Bias}^2(\hat{m}(x)) \leq \tilde{C}_1 h^2,$$

  and

  $$Var(\hat{m}(x)) \leq \frac{\tilde{C}_2}{nh^p}.$$

  What is the optimal h, now?

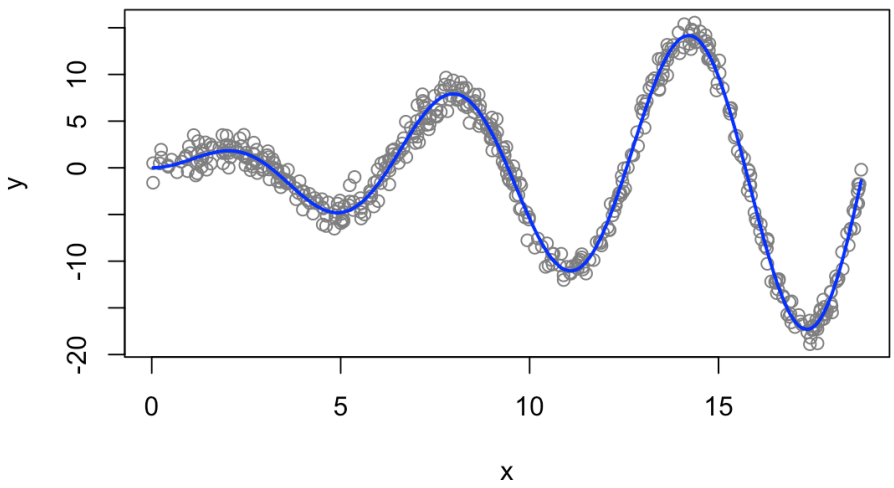
- (A version of) Curse of dimensionality: Nonparametric methods typically suffer from variance that scales exponentially with the number of predictors p; remember, this means that the test error for such methods also scales exponentially with p, which is an awful trend!

LOESS

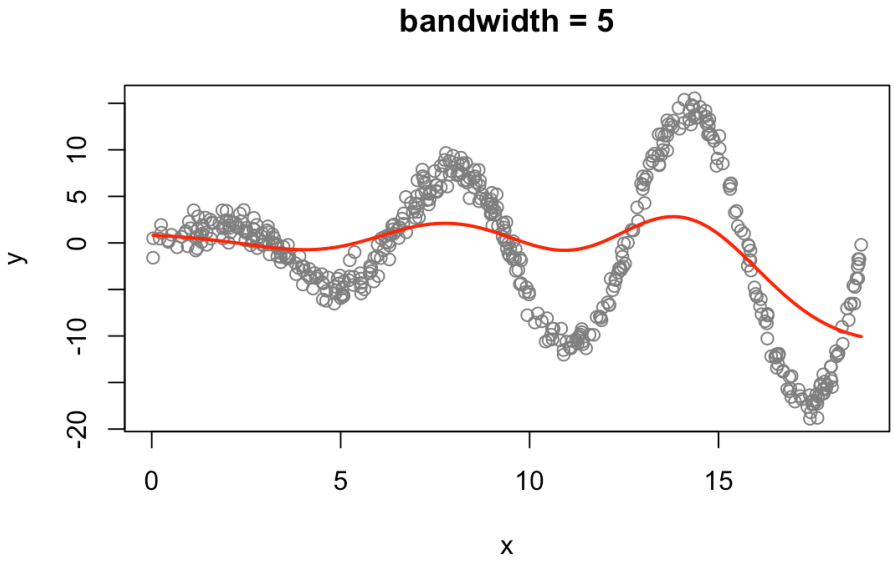
- A similar idea is the LOESS (LOcal regrESSion), LOWESS (Locally Weighted RegrESSion), or local polynomial regression.
- While kernel regression locally fits a weighted average (degree 0), LOESS fits linear regression lines (with local data, defined by the kernel weights).

A toy example

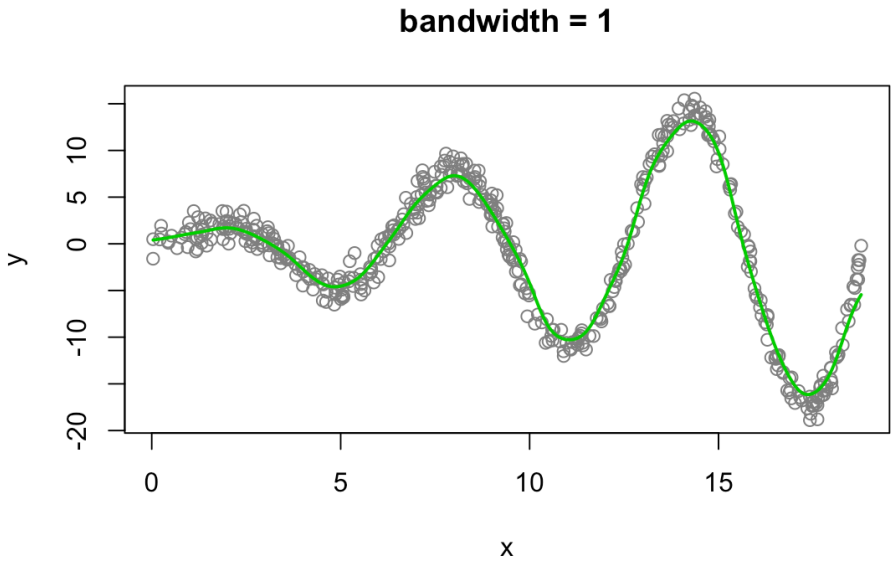
```
# Create a simple nonlinear example
set.seed(0)
n = 500
x = sort(runif(n,0,6*pi))
r = x*sin(x)
y = r + rnorm(n)

plot(x,y,col="gray50")
lines(x,r,col="blue",lwd=2)
```



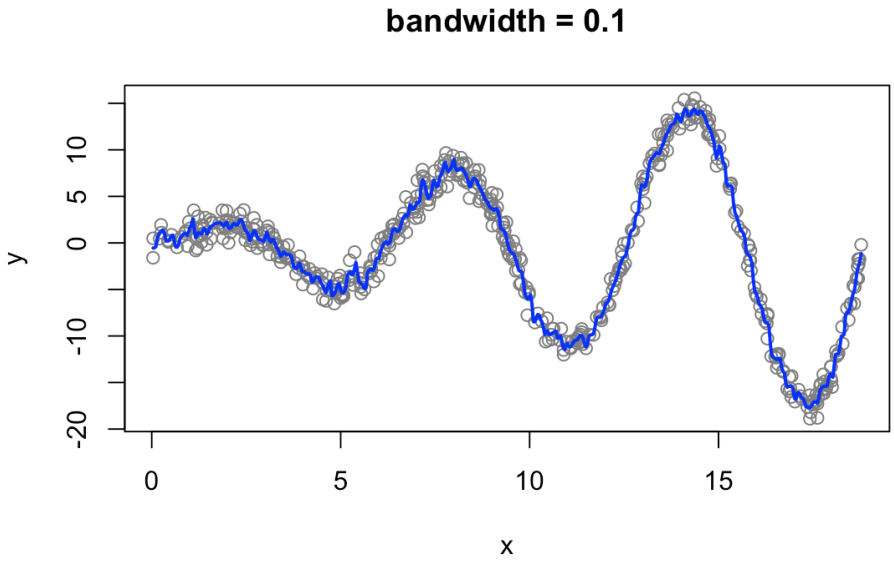

```
# Kernel regression
bws = c(5,1,0.1)
kernmod1 = ksmooth(x,y,kernel="normal",bandwidth=bws[1])
kernmod2 = ksmooth(x,y,kernel="normal",bandwidth=bws[2])
kernmod3 = ksmooth(x,y,kernel="normal",bandwidth=bws[3])

#par(mfrow=c(3,1))
plot(x,y,col="gray50",main=paste("bandwidth =",bws[1]))
lines(kernmod1$x,kernmod1$y,col="red",lwd=2)
```




```
plot(x,y,col="gray50",main=paste("bandwidth =",bws[2]))
lines(kernmod2$x,kernmod2$y,col=3,lwd=2)
```



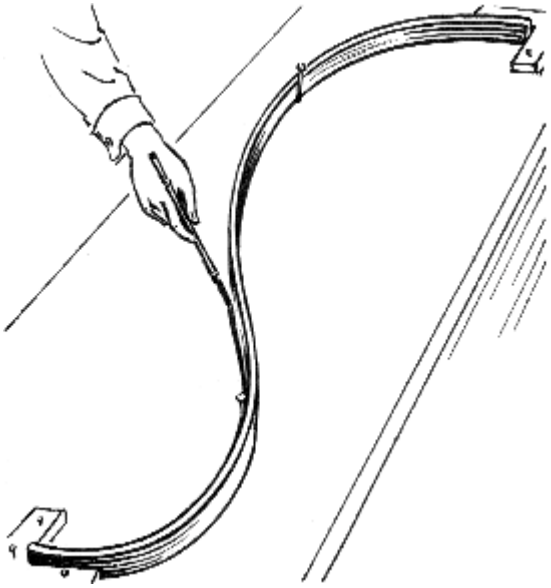

```
plot(x,y,col="gray50",main=paste("bandwidth =",bws[3]))
lines(kernmod3$x,kernmod3$y,col="blue",lwd=2)
```



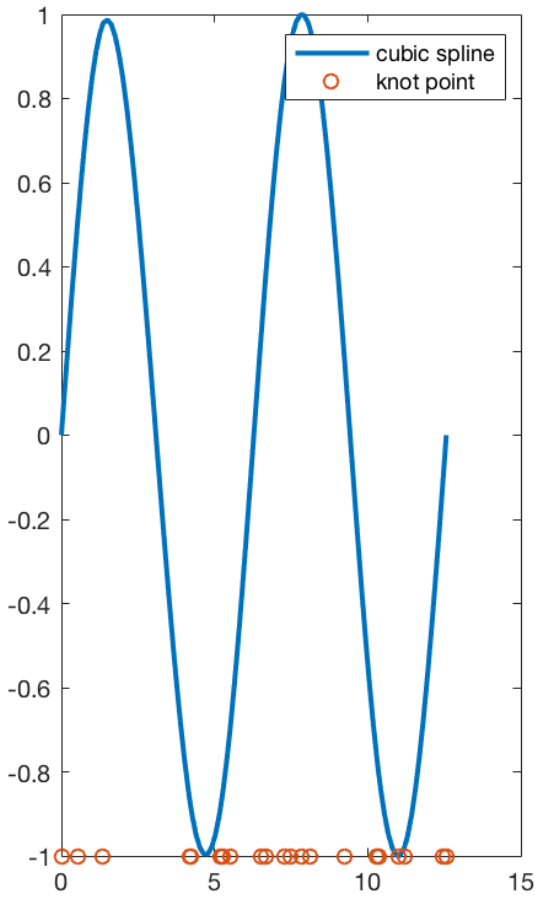
Smoothing Splines

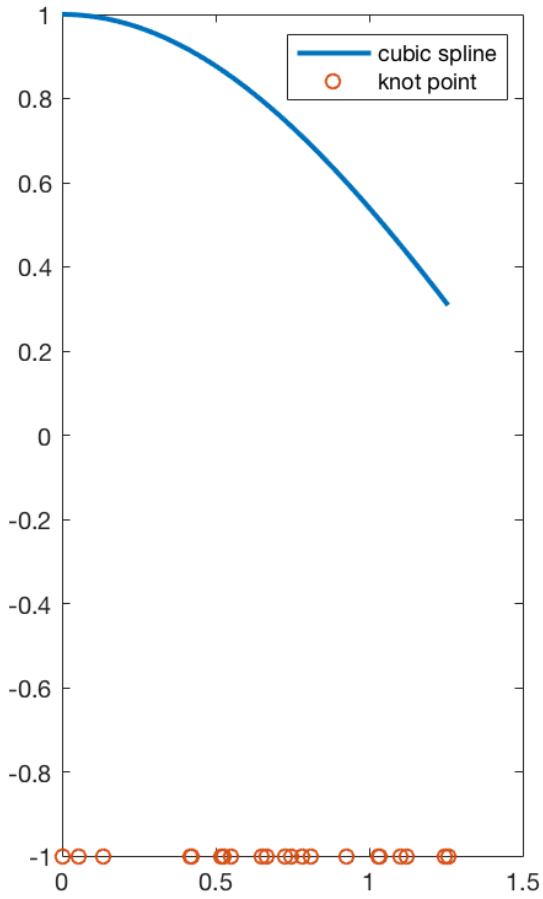
- Smoothing splines, like kernel regression and k-nearest-neigbors regression, provide a flexible way of estimating the underlying regression function $m(x) = E(Y|X = x)$. Though they can be defined for higher dimensions, we'll assume for simplicity throughout that $X \in R$, i.e., there is only one predictor variable.

What is a spline?



- Before introducing smoothing splines, however, we first have to understand what a spline is.
- In words, a kth order spline is a piecewise polynomial function of degree k, that is continuous and has continuous derivatives of orders $1, \ldots, k-1$, at its knot points.
- Formally, a function $f : R \to R$ is a kth order spline with knot points at $t_1 < \ldots < t_m$, if
  - f is a polynomial of degree k on each of the intervals $(-\infty, t_1], [t_1, t_2], \ldots, [t_m, \infty)$, and
  - the jth derivative of f, is continuous at $t_1, \ldots, t_m$, for each $j = 0, 1, \ldots, k-1$.
- The most common case considered is $k = 3$, i.e., that of *cubic splines*. These are piecewise cubic functions that are continuous, and have continuous first, and second derivatives. Note that the continuity in all of their lower order derivatives makes splines very smooth.



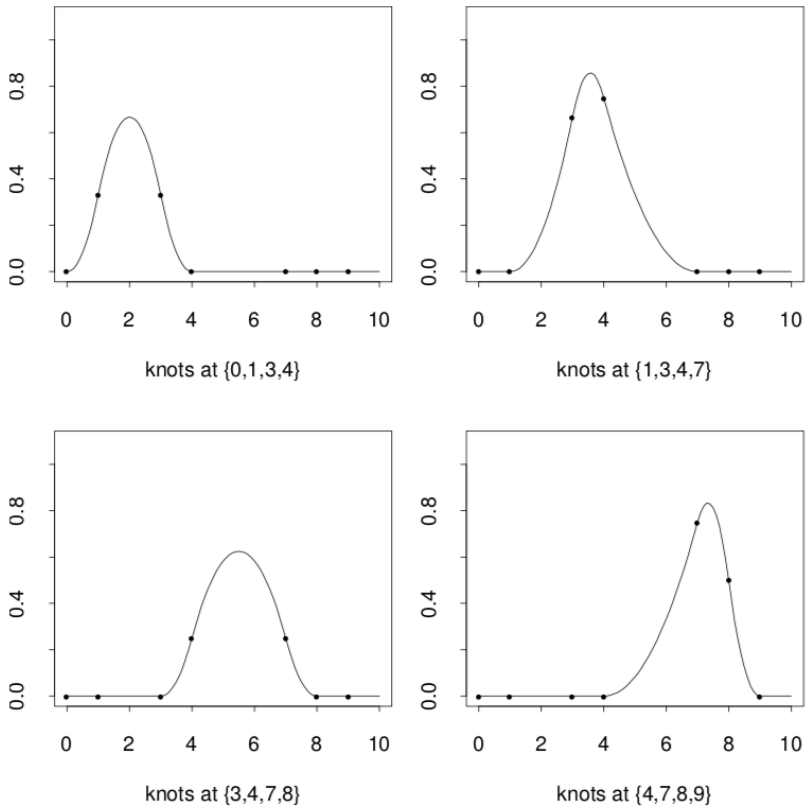


- A bit of statistical folklore: it is said that a cubic spline is so smooth, that one cannot detect the locations of its knots by eye!
- How can we parametrize the set of a splines with knots at a given set of points $t_1, \ldots, t_m$? The most natural way is to use the truncated power basis, $g_1, \ldots, g_{m+k+1}$,

$$g_1(x) = 1, g_2(x) = x, \ldots g_{k+1}(x) = x^k, g_{k+1+j}(x) = (x - t_j)_+^k, j = 1, \ldots, m$$

  Here x_+ denotes the positive part of x, i.e., $x_+ = maxx, 0$
- While these basis functions are natural, a much better computational choice, both for speed and numerical accuracy, is the *B-spline* basis. This was a major development in spline theory and is now pretty much the standard in software; we won't cover these, but it doesn't hurt to be aware of them.

B-splines of order 3: Plot with knots at 0, 1, 3, 4, 7, 8, 9



Regression splines

- So, what can you do with splines? Well, for one, we can perform regression on them!
- Given samples $(x_i, y_i), i = 1, \ldots, n$, we can consider estimating the regression function $m(x) = E(Y|X = x)$ by fitting a kth order spline with knots at some prespecified locations $t_1, \ldots, t_m$.
- This means considering functions of the form $\sum_{j=1}^{m+k+1} \beta_j g_j$ where β_j's are coefficients, and $g_j(\cdot)$'s are the truncated power basis functions for kth order splines over the knots $t_1, \ldots, t_m$.
- As the model $Y = m(X) + \epsilon$ suggests, fit β_j's by minimizing the sum of squared errors:

$$\min_m \frac{1}{n}\sum_{i=1}^{n}(y_i - m(x_i))^2,$$

equivalently,

$$\min_{\beta_j} \frac{1}{n}\sum_{i=1}^{n}(y_i - \sum_{j=1}^{m+k+1}\beta_j g_j(x_i))^2,$$

Regression spline is simple a multivariate regression

- The expression above looks more familiar after a change in notation.
- Write $\mathbf{y} = (y_1, \dots y_n)'$,and define the basis matrix $\mathbf{G}$ (of size $(n \times m + k + 1)$) by

$$G_{ij} = g_j(x_i).$$

- Then the minimization problem is to minimize $(\mathbf{y} - \mathbf{G}\beta)'(\mathbf{y} - \mathbf{G}\beta)$, and the solution is

$$\hat{\beta} = (\mathbf{G}'\mathbf{G})^{-1}\mathbf{G}'\mathbf{y}.$$

Regression spline is a linear smoother

To see this, denote $g(x) = (g_1(x), \dots g_{m+k+1}(x))'$, and then the regression spline estimate at x is

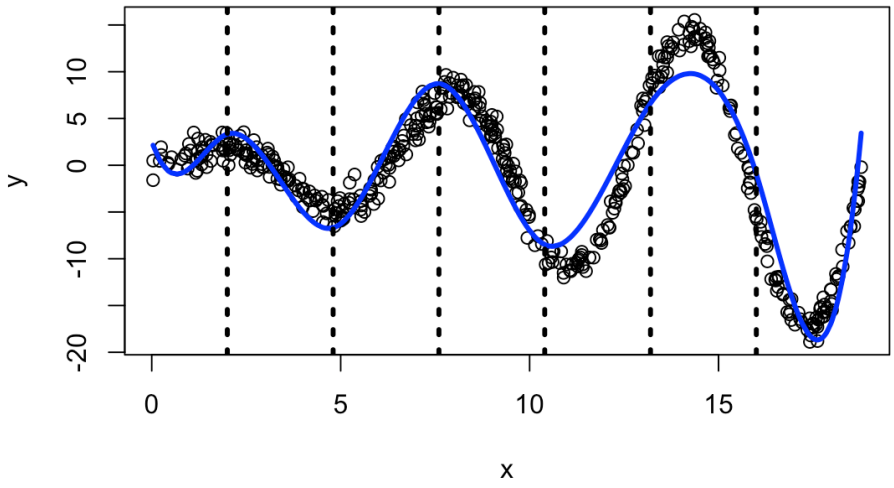
$$\hat{m}(x) = g(x)'\hat{\beta} = g(x)'(\mathbf{G}'\mathbf{G})^{-1}\mathbf{G}'\mathbf{y}.$$

Natural splines

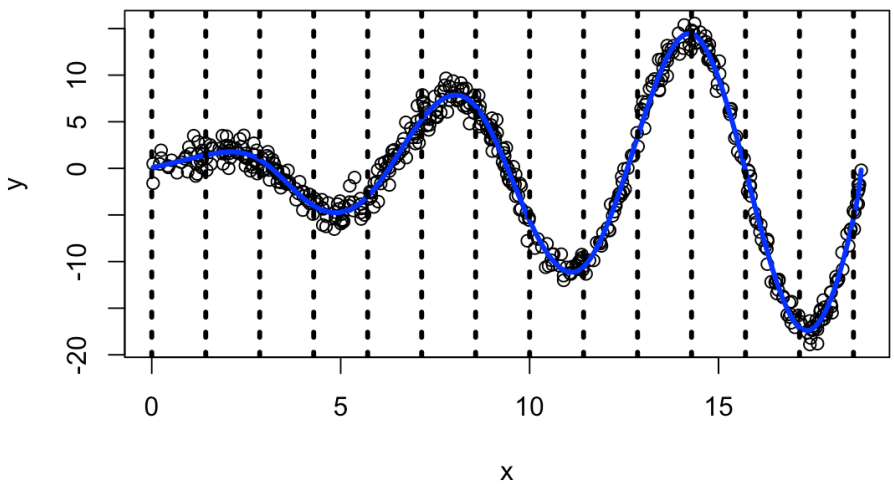
- One problem with regression splines is that the estimates tend to display erractic behavior, i.e., they have high variance, at the boundaries of the domain of $x_1, \dots x_n$. This gets worse as the order k gets larger
- A way to remedy this problem is to force the piecewise polynomial function to have a lower degree to the left of the leftmost knot, and to the right of the rightmost knot-this is exactly what natural splines do. A natural spline of order k, with knots at $t_1 <. . . < t_m$, is a piecewise polynomial function f such that
  - f is a polynomial of degree k on each of $[t_1, t_2], \dots [t_{m-1}, t_m]$,
  - f is a polynomial of degree $(k-1)/2$ on $(-\infty, t_1]$ and $[t_m, \infty)$,
  - f is continuous and has continuous derivatives of orders $1, \dots k-1$ at its knots.
- It is implicit here that natural splines are only defined for odd orders k. The most common case: $k = 3$, i.e., *cubic natural splines*, which are linear beyond the boundaries
- Note that there is a variant of the truncated power basis for natural splines (and a variant of the B-spline basis for natural splines). This time, though, we only need m basis functions to span the space of kth order natural splines with knots at $t_1, \dots t_m$

Toy example, revisited

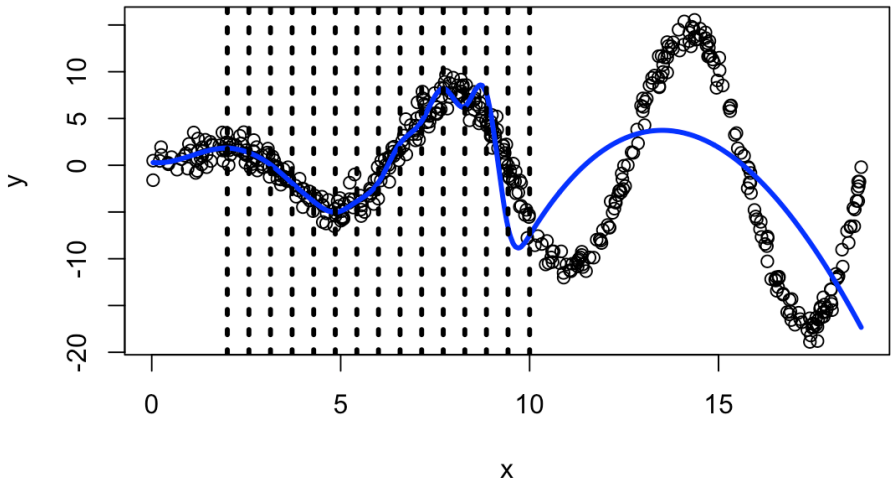
```
# Regression splines
library(splines)

knots = seq(2,16,length=6)
G = bs(x,knots=knots,degree=3)
regmod = lm(y~G)
plot(x,y)
lines(x,regmod$fitted,lwd=3,col="blue")
abline(v=knots,lty=3,lwd=3)
```




```
knots = seq(0,20,length=15)
G = bs(x,knots=knots,degree=3)
regmod = lm(y~G)
plot(x,y)
lines(x,regmod$fitted,lwd=3,col="blue")
abline(v=knots,lty=3,lwd=3)
```



```
knots = seq(2,10,length=15)
G = bs(x,knots=knots,degree=3)
regmod = lm(y~G)
plot(x,y)
lines(x,regmod$fitted,lwd=3,col="blue")
abline(v=knots,lty=3,lwd=3)
```



Knot locations??

- Regression splines are a classic tool, and can work well provided we choose good knot points. But in general choosing knots is a tricky business
- A natural choice is to place knots at every data point.
  - This will lead to a problem as the number of coefficient $p = n$ is just the same as the number of observations
  - (Near) multicollinearity will happen!
  - Use regularized regression to circumvent multicollinearity

Smoothing splines

- Smoothing splines are an interesting creature: these estimators perform a regularized regression over the natural spline basis, placing knots at all points $x_1, \ldots, x_n$.
- Smoothing splines circumvent the problem of knot selection (as they just use the inputs as knots), and simultaneously, they control for overfitting by shrinking the coefficients of the estimated function (in its basis expansion).
- We will focus on cubic smoothing splines.
- We consider functions of the form

$$m(\cdot) = \sum_{j=1}^{n} \beta_j g_j(\cdot),$$

where g_j's are the truncated power basis functions for natural cubic splines with knots at $x_1, \ldots, x_n$.

Cubic smoothing spline regression

- The coefficients are chosen to minimize

$$(\mathbf{y} - \mathbf{G}\beta)'(\mathbf{y} - \mathbf{G}\beta) + \lambda\beta'\Omega\beta.$$

and Ω is the penalty matrix defined as $\Omega_{ij} = \int g_i''(t)g_j''(t)dt$.

- There is an extra term $\lambda\beta'\Omega\beta$ compared to the usual criterion for regression splines; this is called a *regularization term*, and it has the effect of shrinking the components of the solution $\hat{\beta}$ towards zero. The parameter $\lambda \geq 0$ is a tuning parameter, often called the smoothing parameter, and the higher the value of λ, the more shrinkage.
- The exact form of the penalty matrix Ω is actually not so important. It imparts more shrinkage on the coefficients β_j that correspond to wigglier functions g_j . Hence, as we increase λ, we are shrinking away the wiggler basis functions.

Cubic smoothing spline regression: Estimation

- It should be not so surprising that the fitted values of β are:

$$\hat{\beta}_\lambda = (\mathbf{G}'\mathbf{G} + \lambda\Omega)^{-1}\mathbf{G}'\mathbf{y}.$$

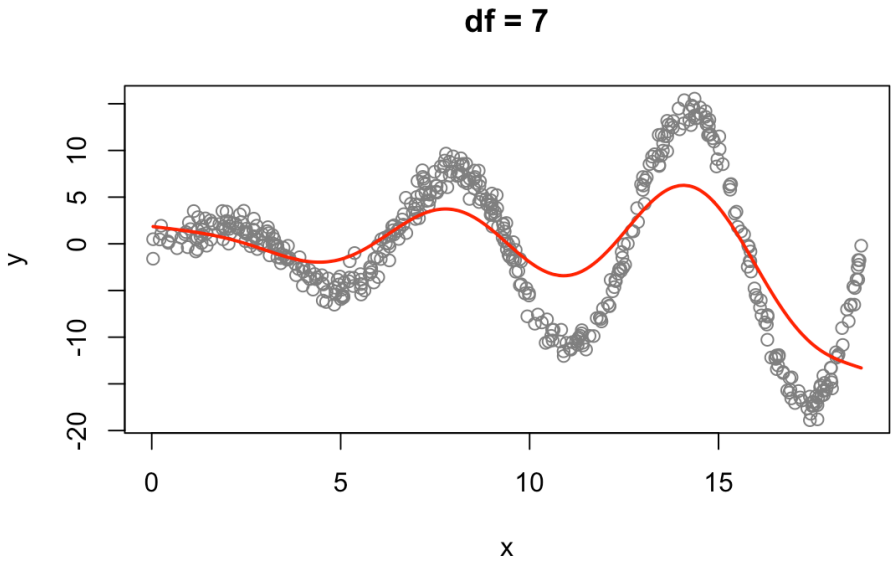
- Conditional mean estimators at $X = x$ is, for $g(x) = (g_1(x), \ldots g_m(x))'$,

$$\hat{m}(x) = g(x)'\hat{\beta}_\lambda = g(x)'(\mathbf{G}'\mathbf{G} + \lambda\Omega)^{-1}\mathbf{G}'\mathbf{y}.$$

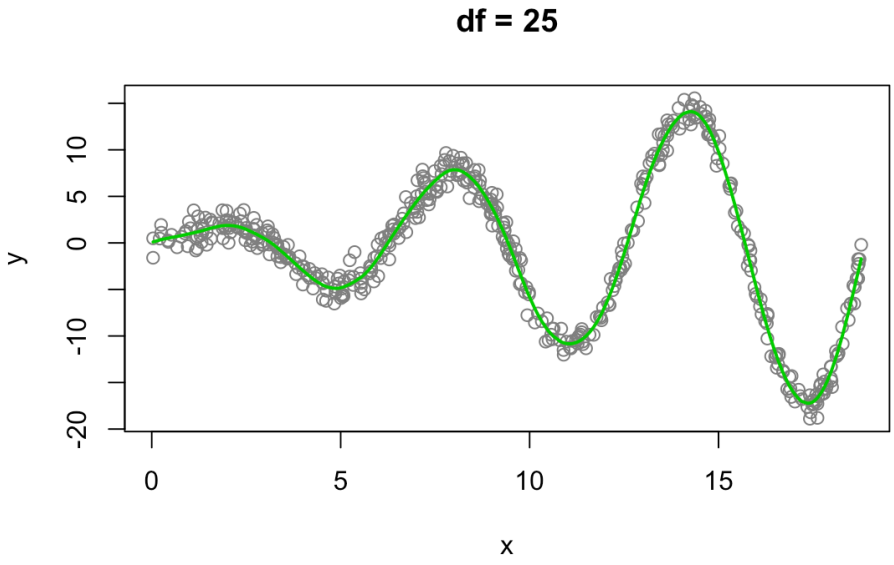
- This is also a linear smoother.

Toy example, revisited

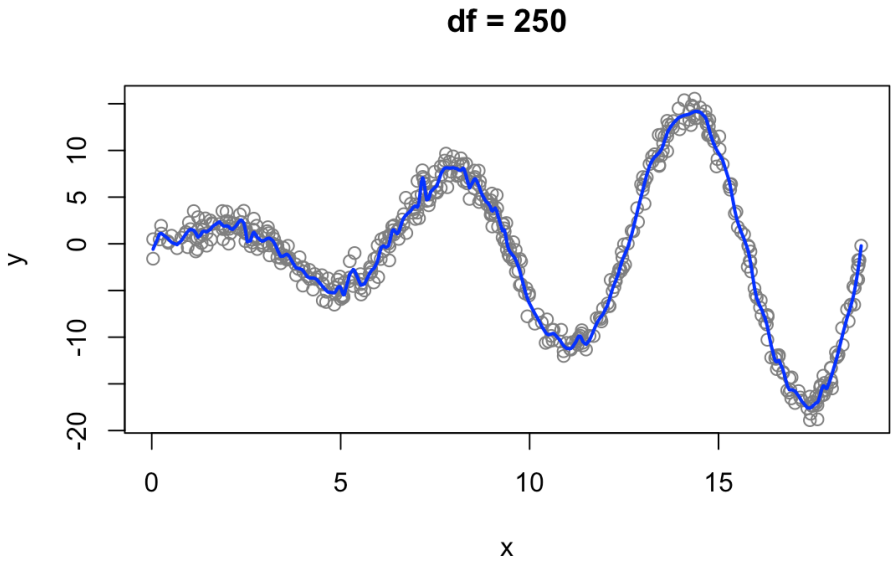
```
# Regression splines
# Smoothing splines
dfs = c(7,25,250)
splinemod1 = smooth.spline(x,y,df=dfs[1])
splinemod2 = smooth.spline(x,y,df=dfs[2])
splinemod3 = smooth.spline(x,y,df=dfs[3])

plot(x,y,col="gray50",main=paste("df =",dfs[1]))
lines(splinemod1$x,splinemod1$y,col="red",lwd=2)
```




```
plot(x,y,col="gray50",main=paste("df =",dfs[2]))
lines(splinemod2$x,splinemod2$y,col=3,lwd=2)
```




```
plot(x,y,col="gray50",main=paste("df =",dfs[3]))
lines(splinemod3$x,splinemod3$y,col="blue",lwd=2)
```



Inference with linear smoothers

We will come back to this after introducing bootstrap.

- Since all of the nonparametric regression fits are of the form $m(x) = \sum_i w(x, x_i) y_i$, the mean and variance of $m(x)$ are obtained by the parallel arguments we used in linear regression.

Additive models

Nonparametric regression vs parametric regression

- Assume for know that $X \in \mathbb{R}$. A model of the form

$$Y = m(X) + \varepsilon,$$

  where we don't make any assumptions about the form of the true underlying regression function $m(x) = E(Y|X = x)$, is called a nonparametric regression model.

- Contrast this with a parametric regression model, e.g., linear regression, in which we assume that $m(X) = \beta_0 + \beta_1 X$, so that the model becomes

$$Y = \beta_0 + \beta_1 X + \varepsilon.$$

The bias-variance tradeoff

- Generally speaking, a parametric estimator (e.g., linear regression) will have a lower variance than a nonparametric one (e.g., k-nearest-neighbors, kernel regression, or smoothing splines), because it is more restrictive.
- Meanwhile, the bias depends on the true underlying model. Nonparametric estimators are generally flexible enough that they will have a low bias for a wide range of underlying regression functions, but a parametric estimator (such as linear regression) will only have a low bias if the parametric assumption is approximately correct (i.e., the true model is approximately linear), and can otherwise suffer from high bias.

- As we know, expected test error is composed of bias and variance, so both of these quantities are important for predictive performance.

Multiple dimensions

When $X \in \mathbb{R}^p$, i.e., we have p predictors instead of 1, extending the linear model is straightforward;

- Writing $X = (X_1, \dots X_p)$, the multi-dimensional linear model is

$$Y = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p + \varepsilon.$$

  Estimation proceeds just as in the univariate case, but the true model is not linear, the bias might be high.

- Given a multi-dimensional nonparametric model

$$Y = m(X_1, \dots X_p) + \varepsilon,$$

  how do we construct fully nonparametric estimates for m?

- Actually, this is possible for each of the methods we discussed: k-nearest-neighbors, kernel regression, and smoothing splines.

- In multiple dimensions, the variance of nonparametric estimators becomes a real problem. Nonparametric methods typically suffer from variance that scales exponentially with the number of predictors p!

A middle ground

The additive compromise

Enter additive models, a framework that lies somewhere in between the fully parametric and nonparametric settings.

1. Parametric: $Y = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p + \varepsilon$;
2. Fully nonparametric $Y = m(X_1, \dots X_p) + \varepsilon$;
3. Additive model: $Y = \beta_0 + m_1(X_1) + \cdots m_p(X_p) + \varepsilon$;

- Additive estimates tend to balance the strengths of the fully nonparametric and parametric estimates. I.e., additive estimates tend have a lower variance than fully nonparametric ones, and can have a lower bias than parametric ones.

Estimation by backfitting

- Given pairs $(\mathbf{x}_i, y_i) \in \mathbb{R}^p \times \mathbb{R}$, $i = 1, \dots, n$, with each $\mathbf{x}_i = (x_{i1}, \dots, x_{ip}) \in \mathbb{R}^p$, the additive model becomes

$$y_i = \beta_0 + m_1(x_{i1}) + \cdots + m_p(x_{ip}) + \varepsilon$$

  subject to the identifiability assumptions $E(y_i) = \beta_0$, and $E(m_j(x_{ij})) = 0$ for $j = 1, \dots, p$.

- Computing an additive estimate from the model is now done by a simple procedure called *backfitting*. We first let

$$\hat{\beta}_0 = \bar{y}.$$

- The idea is for the rest is just to cycle through estimating each of $m_1, \ldots, m_p$ one at a time, by univariate smoothing, and repeat this until convergence.

- The intuition for backfitting just comes from rearranging the model. Supposing that we have good estimates for m_1 to m_{p-1} and β_0, except m_p, then the partially fitted additive model is

$$y_i^{(p)} \equiv y_i - \hat{\beta}_0 - \sum_{j=1}^{p-1} \hat{m}_j(x_{ij}) = m_p(x_{ip}) + \varepsilon$$

- Treating the left-hand side as a response, m_p is fitted by regressing the new response $y_i^{(p)}$ on x_{ip} by, either smoothing splines or even a simple linear regression.

To be concrete, the backfitting algorithm is:

1. Initialize

$$\hat{\beta}_0 = 1/n \sum_{i=1}^{n} y_i, \quad \hat{m}_j \equiv 0, \forall j$$

2. Do until $\hat{m}_j$ converge:
    - For each predictor j:

$$\hat{m}_j \leftarrow \text{Smooth}[\{y_i - \hat{\beta}_0 - \sum_{k \neq j} \hat{m}_k(x_{ik}), x_{ij}\}_1^n] \quad \text{(backfitting step)}$$

$$\hat{m}_j \leftarrow \hat{m}_j - 1/n \sum_{i=1}^{n} \hat{m}_j(x_{ij}) \quad \text{(mean centering of estimated function)}$$

A worked example

As an example, we'll look at data on median house prices across Census tracts. This has both California and Pennsylvania, but it's hard to visually see patterns with both states.

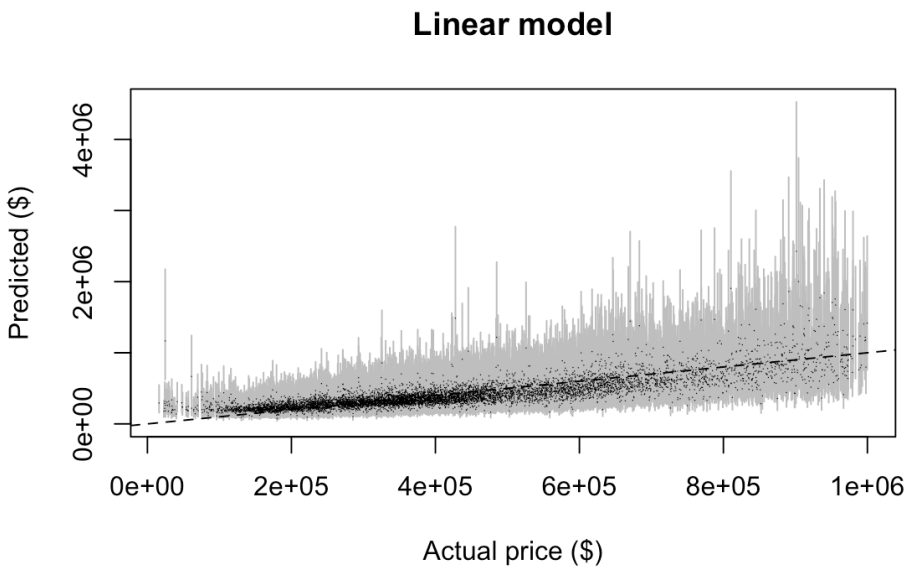
```
#
housing <- read.csv("calif_penn_2011.csv")
housing <- na.omit(housing)
calif <- housing[housing$STATEFP == 6, ] # 42 for Pennsylvania
```

We'll fit a linear model for the log price, on the thought that it makes some sense for the factors which raise or lower house values to multiply together, rather than just adding.

```
calif.lm <- lm(log(Median_house_value) ~ Median_household_income + Mean_household_income +
    POPULATION + Total_units + Vacant_units + Owners + Median_rooms + Mean_household_size_owners +
    Mean_household_size_renters + LATITUDE + LONGITUDE, data = calif)
# summary(calif.lm)
```

Below plots the predicted prices, 95% prediction intervals , against the actual prices. The predictions are not all that accurate, but they do have pretty reasonable coverage; about 96% of actual prices fall within the prediction limit.

```
preds.lm<-as_tibble(predict(calif.lm, interval="predict"))
plot(calif$Median_house_value, exp(preds.lm$fit), type = "n", pch = 22,xlab = "Actual price ($)",
    ylab = "Predicted ($)", main = "Linear model", ylim = c(0, exp(max(preds.lm$upr))))
segments(calif$Median_house_value, exp(preds.lm$lwr),
         calif$Median_house_value, exp(preds.lm$upr), col = "grey")
abline(a = 0, b = 1, lty = "dashed")
points(calif$Median_house_value, exp(preds.lm$fit), pch = 16, cex = 0.1)
```



Next, we'll fit an additive model, using the `gam` function from the `mgcv` package; this automatically sets the bandwidths using a fast approximation to leave-one-out CV called generalized cross-validation.

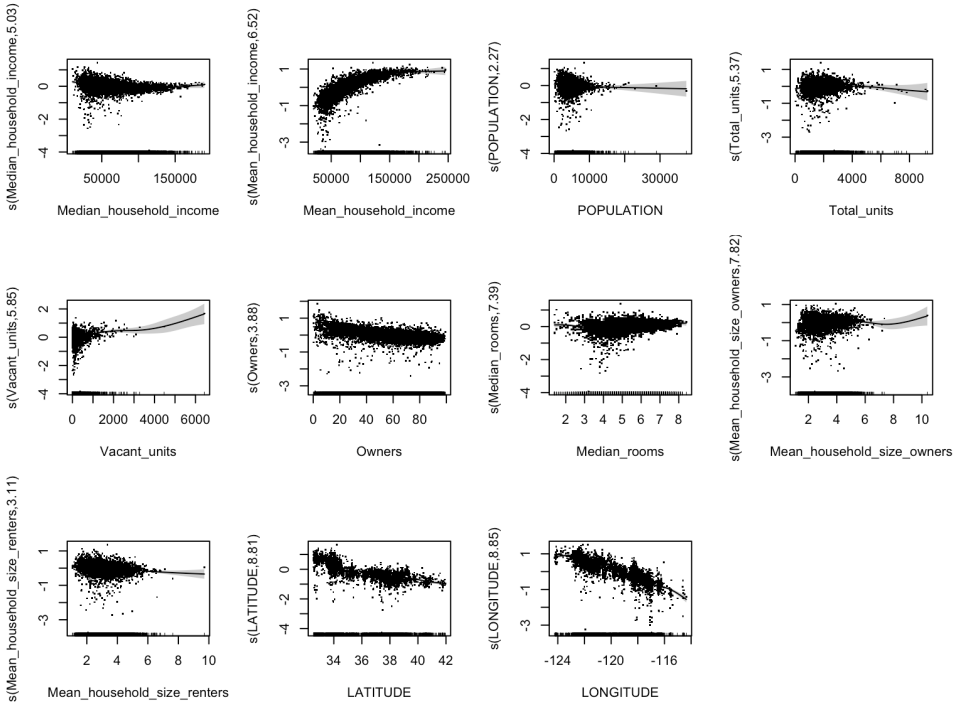
```
library(mgcv)
```

The `s()` terms in the `gam` formula indicate which terms are to be smoothed.

```
calif.gam <- gam(log(Median_house_value) ~ s(Median_household_income) +
    s(Mean_household_income) + s(POPULATION) + s(Total_units) + s(Vacant_units) +
    s(Owners) + s(Median_rooms) + s(Mean_household_size_owners) + s(Mean_household_size_renters) +
      s(LATITUDE) + s(LONGITUDE), data = calif)
summary(calif.gam)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## log(Median_house_value) ~ s(Median_household_income) + s(Mean_household_income) +
##     s(POPULATION) + s(Total_units) + s(Vacant_units) + s(Owners) +
##     s(Median_rooms) + s(Mean_household_size_owners) + s(Mean_household_size_renters) +
##     s(LATITUDE) + s(LONGITUDE)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.819325   0.003086    4155   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                                edf Ref.df       F  p-value
## s(Median_household_income)     5.027  6.288   7.591 2.67e-08 ***
## s(Mean_household_income)       6.517  7.686 165.075  < 2e-16 ***
## s(POPULATION)                  2.265  2.986   2.702   0.0440 *
## s(Total_units)                 5.368  6.568   2.501   0.0195 *
## s(Vacant_units)                5.853  6.972  20.914  < 2e-16 ***
## s(Owners)                      3.876  4.901 113.380  < 2e-16 ***
## s(Median_rooms)                7.393  8.267  18.056  < 2e-16 ***
## s(Mean_household_size_owners)  7.822  8.576  14.378  < 2e-16 ***
## s(Mean_household_size_renters) 3.113  4.015  36.824  < 2e-16 ***
## s(LATITUDE)                    8.806  8.989 180.208  < 2e-16 ***
## s(LONGITUDE)                   8.845  8.993 266.854  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.744   Deviance explained = 74.6%
## GCV = 0.071856  Scale est. = 0.071223  n = 7481
```

```
preds.gam<-as_tibble(predict(calif.gam, interval="predict"))
plot(calif.gam, scale = 0, se = 2, shade = TRUE, resid = TRUE, pages = 1)
```


s(Median_household_income,5.03)
Median_household_income
s(Mean_household_income,6.52)
Mean_household_income
s(POPULATION,2.27)
POPULATION
s(Total_units,5.37)
Total_units
s(Vacant_units,5.85)
Vacant_units
s(Owners,3.88)
Owners
s(Median_rooms,7.39)
Median_rooms
s(Mean_household_size_owners,7.82)
Mean_household_size_owners
s(Mean_household_size_renters,3.11)
Mean_household_size_renters
s(LATITUDE,8.81)
LATITUDE
s(LONGITUDE,8.85)
LONGITUDE