Chapter 4

Using Nonparametric Smoothing in Regression

Having spent long enough running down linear regression, and thought through evaluating predictive models, it is time to turn to constructive alternatives, which are (also) based on smoothing.

Recall the basic kind of smoothing we are interested in: we have a response variable Y, some input variables which we bind up into a vector X, and a collection of data values, $(x_1, y_1), (x_2, y_2), \ldots x_n, y_n)$. By "smoothing", I mean that predictions are going to be weighted averages of the observed responses in the training data:

$$\hat{r}(x) = \sum_{i=1}^{n} y_i w(x, x_i, h) \tag{4.1}$$

Most smoothing methods have a control setting, which here I write h, that determines *how much* smoothing we do. With k nearest neighbors, for instance, the weights are $1/k$ if x_i is one of the k-nearest points to x, and $w = 0$ otherwise, so large k means that each prediction is an average over many training points. Similarly with kernel regression, where the degree of smoothing is controlled by the bandwidth h.

Why do we want to do this? How do we pick how much smoothing to do?

4.1 How Much Should We Smooth?

When we smooth very little ($h \to 0$), then we can match very small, fine-grained or sharp aspects of the true regression function, if there are such. That is, less smoothing leads to less bias. At the same time, less smoothing means that each of our predictions is going to be an average over (in effect) fewer observations, making the prediction noisier. Smoothing less increases the variance of our estimate. Since

$$(\text{total error}) = (\text{noise}) + (\text{bias})^2 + (\text{variance}) \tag{4.2}$$

(Eq. 1.26), if we plot the different components of error as a function of h, we typically get something that looks like Figure 4.1. Because changing the amount of smoothing has opposite effects on the bias and the variance, there is an optimal amount of smoothing, where we can't reduce one source of error without increasing the other. We therefore want to find that optimal amount of smoothing, which is where cross-validation comes in.

You should note, at this point, that the optimal amount of smoothing depends on the real regression curve, our smoothing method, *and* how much data we have. This is because the variance contribution generally shrinks as we get more data.[1] If we get more data, we go from Figure 4.1 to Figure 4.2. The minimum of the over-all error curve has shifted to the left, and we should smooth less.

Strictly speaking, **parameters** are properties of the data-generating process alone, so the optimal amount of smoothing is not really a parameter. If you do think of it as a parameter, you have the problem of why the "true" value changes as you get more data. It's better thought of as a setting or control variable in the smoothing method, to be adjusted as convenient.

4.2 Adapting to Unknown Roughness

Consider Figure 4.3, which graphs two functions, f and g. Both are "smooth" functions in the qualitative, mathematical sense[2]. We could Taylor-expand both functions to approximate their values anywhere, just from knowing enough derivatives at one point x_0.[3] Alternately, if instead of knowing the derivatives at x_0, we have the values of the functions at a sequence of points $x_1, x_2, \ldots x_n$, we could use interpolation to fill out the rest of the curve. Quantitatively, however, $f(x)$ is less smooth than $g(x)$ — it changes much more rapidly, with many reversals of direction. For the same degree of inaccuracy in the interpolation $f(\cdot)$ needs more, and more closely spaced, training points x_i than goes $g(\cdot)$.

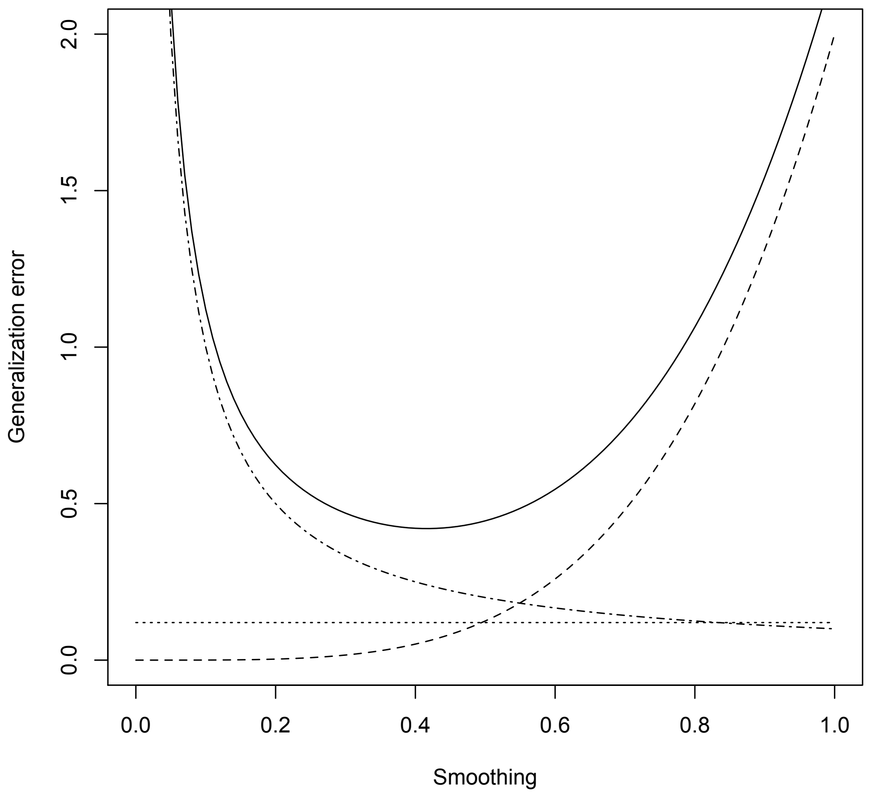
Now suppose that we don't get to actually get to see $f(x)$ and $g(x)$, but rather just $f(x)+\epsilon$ and $g(x)+\eta$, where ϵ and η are noise. (To keep things simple I'll assume they're the usual mean-zero, constant-variance, IID Gaussian noises, say with $\sigma = 0.15$.) The data now look something like Figure 4.4. Can we now recover the curves?

As remarked in Chapter 1, if we had multiple measurements at the same x, then we could recover the expectation value by averaging: since the regression function $r(x) = \mathbf{E}[Y|X=x]$, if we had many observations all with $x_i = x$, the average of the corresponding y_i would (by the law of large numbers) converge on $r(x)$. Generally, however, we have at most one measurement per value of x, so simple averaging won't work. Even if we just confine ourselves to the x_i where we have observations, the mean-squared error will always be σ^2, the noise variance. However, our estimate would be unbiased.

[1]Sometimes bias changes as well. Noise does not (why?).

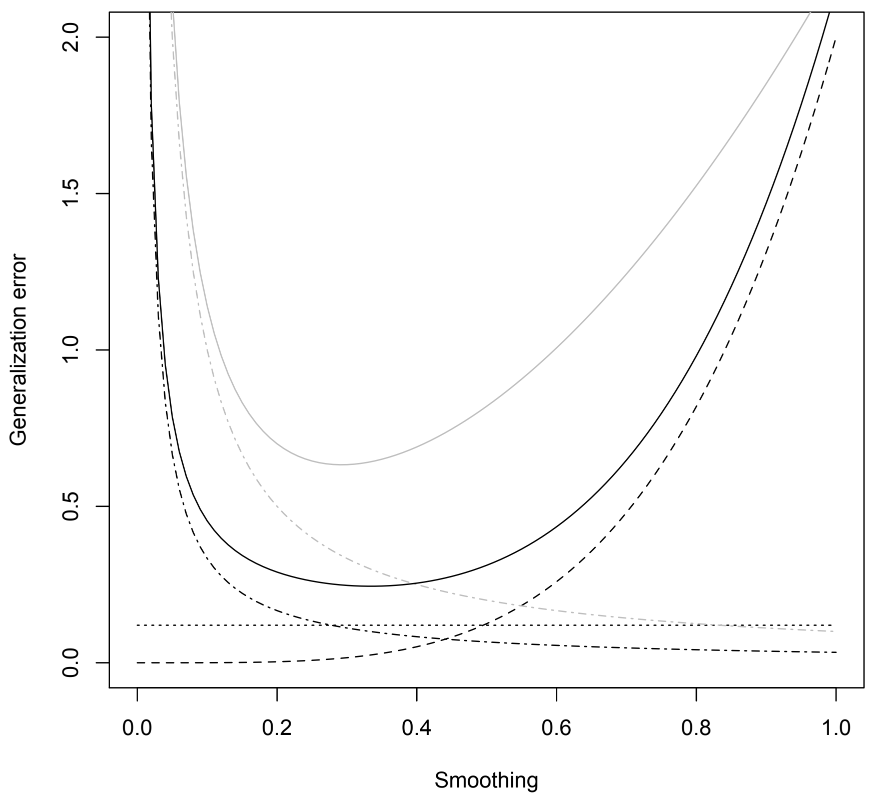
[2]They are "C^∞": they're not only continuous, but their derivatives exist and are continuous to all orders.

[3]Technically, a function whose Taylor series converges everywhere is **analytic**.



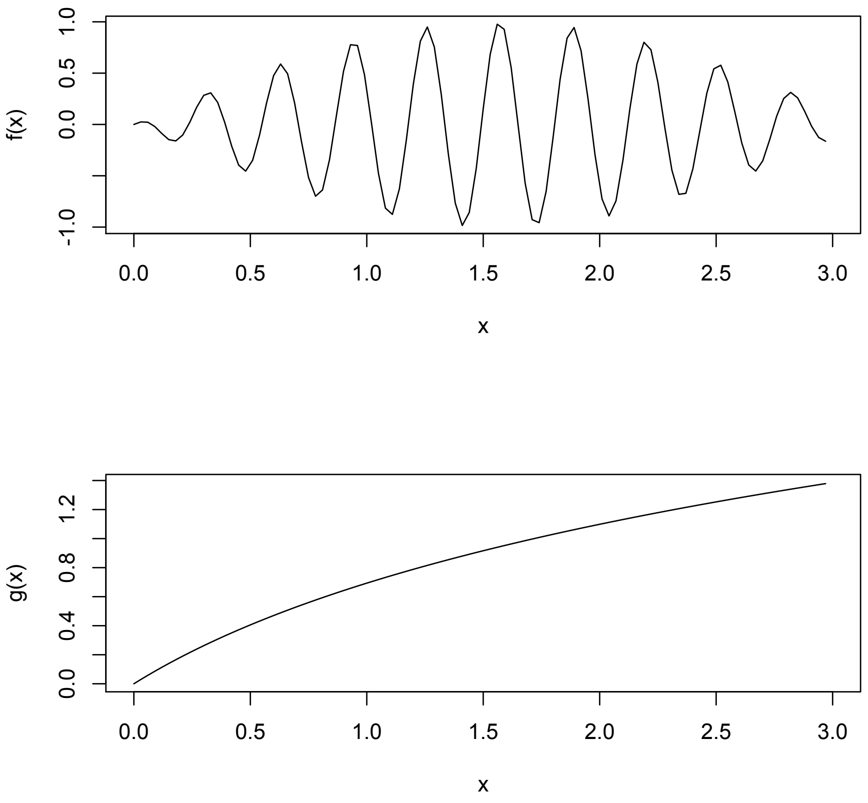
```
curve(2*x^4,from=0,to=1,lty=2,xlab="Smoothing",ylab="Generalization error")
curve(0.12+x-x,lty=3,add=TRUE)
curve(1/(10*x),lty=4,add=TRUE)
curve(0.12+2*x^4+1/(10*x),add=TRUE)
```

Figure 4.1: Over-all generalization error for different amounts of smoothing (solid curve) decomposed into process noise (dotted line), approximation error introduced by smoothing (=squared bias; dashed curve), and estimation variance (dot-and-dash curve). The numerical values here are arbitrary, but the functional forms (squared bias $\propto h^4$, variance $\propto n^{-1}h^{-1}$) are representative of typical results for non-parametric smoothing.



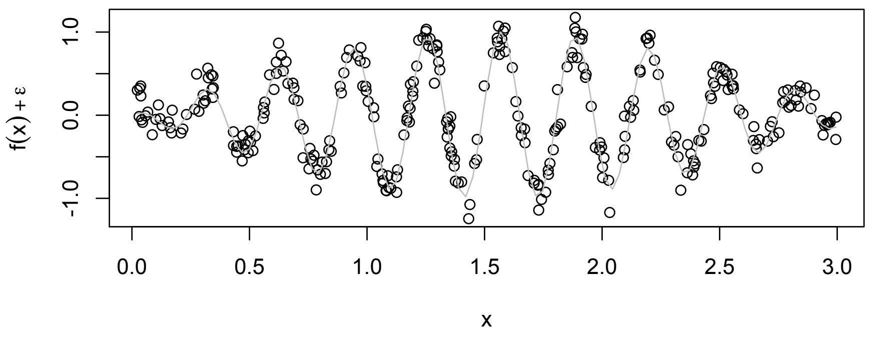
```
curve(2*x^4,from=0,to=1,lty=2,xlab="Smoothing",ylab="Generalization error")
curve(0.12+x-x,lty=3,add=TRUE)
curve(1/(10*x),lty=4,add=TRUE,col="grey")
curve(0.12+2*x^2+1/(10*x),add=TRUE,col="grey")
curve(1/(30*x),lty=4,add=TRUE)
curve(0.12+2*x^4+1/(30*x),add=TRUE)
```

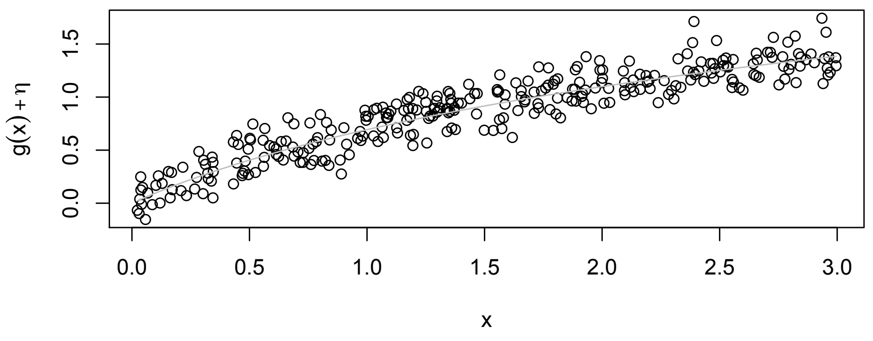
Figure 4.2: Consequences of adding more data to the components of error: noise (dotted) and bias (dashed) are unchanged, but the new variance curve (dotted and dashed, black) is to the left of the old (greyed), so the new over-all error curve (solid black) is lower, and has its minimum at a smaller amount of smoothing than the old (solid grey).



```
par(mfcol=c(2,1))
curve(sin(x)*cos(20*x),from=0,to=3,xlab="x",ylab=expression(f(x)))
curve(log(x+1),from=0,to=3,xlab="x",ylab=expression(g(x)))
```

Figure 4.3: Two curves for the running example. Above, $f(x)$; below, $g(x)$. (As it happens, $f(x) = \sin x \cos 20x$, and $g(x) = \log x + 1$, but that doesn't really matter.)





```
x = runif(300,0,3)
yf = sin(x)*cos(20*x)+rnorm(length(x),0,0.15)
yg = log(x+1)+rnorm(length(x),0,0.15)
par(mfcol=c(2,1))
plot(x,yf,xlab="x",ylab=expression(f(x)+epsilon))
curve(sin(x)*cos(20*x),col="grey",add=TRUE)
plot(x,yg,xlab="x",ylab=expression(g(x)+eta))
curve(log(x+1),col="grey",add=TRUE)
```

Figure 4.4: The same two curves as before, but corrupted by IID Gaussian noise with mean zero and standard deviation 0.15. (The x values are the same, but there are different noise realizations for the two curves.) The light grey line shows the noiseless curves.

What smoothing methods try to use is that we may have multiple measurements at points x_i which are *near* the point of interest x. If the regression function is smooth, as we're assuming it is, $r(x_i)$ will be close to $r(x)$. Remember that the mean-squared error is the sum of bias (squared) and variance. Averaging values at $x_i \neq x$ is going to introduce bias, but averaging many independent terms together also reduces variance. If by smoothing we get rid of more variance than we gain bias, we come out ahead.

Here's a little math to see it. Let's assume that we can do a first-order Taylor expansion (Figure 4.5), so

$$r(x_i) \approx r(x) + (x_i - x)r'(x) \tag{4.3}$$

and

$$y_i \approx r(x) + (x_i - x)r'(x) + \epsilon_i \tag{4.4}$$

Now we average: to keep the notation simple, abbreviate the weight $w(x_i, x, h)$ by just w_i.

$$\begin{aligned}
\widehat{r}(x) &= \frac{1}{n}\sum_{i=1}^{n} y_i w_i & (4.5)\\
&= \frac{1}{n}\sum_{i=1}^{n} (r(x) + (x_i - x)r'(x) + \epsilon_i)w_i & (4.6)\\
&= r(x) + \sum_{i=1}^{n} w_i \epsilon_i + r'(x)\sum_{i=1}^{n} w_i(x_i - x) & (4.7)\\
\widehat{r}(x) - r(x) &= \sum_{i=1}^{n} w_i \epsilon_i + r'(x)\sum_{i=1}^{n} w_i(x_i - x) & (4.8)\\
\mathbf{E}\left[(\widehat{r}(x) - r(x))^2\right] &= \sigma^2 \sum_{i=1}^{n} w_i^2 + \mathbf{E}\left[\left(r'(x)\sum_{i=1}^{n} w_i(x_i - x)\right)^2\right] & (4.9)
\end{aligned}$$

(Remember that: $\sum w_i = 1$; $\mathbf{E}\left[\epsilon_i\right] = 0$; the noise is uncorrelated with everything; and $\mathbf{E}\left[\epsilon_i\right] = \sigma^2$.)

The first term on the final right-hand side is an estimation variance, which will tend to shrink as n grows. (If $w_i = 1/n$, the unweighted averaging case, we get back the familiar σ^2/n.) The second term, expectation, on the other hand, is bias, which grows as x_i gets further from x, and as the magnitudes of the derivatives grow, i.e., *how* smooth or wiggly the regression function is. For this to work, w_i had better shrink as $x_i - x$ and $r'(x)$ grow.[4] Finally, all else being equal, w_i should also shrink with n, so that the over-all size of the sum shrinks as we get more data.

To illustrate, let's try to estimate $f(1.6)$ and $g(1.6)$ from the noisy observations. We'll try a simple approach, just averaging all values of $f(x_i) + \epsilon_i$ and $g(x_i) + \eta_i$

[4]The higher derivatives of r also matter, since we should really be keeping more than just the first term in the Taylor expansion. The details get messy, but Eq. 4.12 below gives the upshot for kernel smoothing in particular.

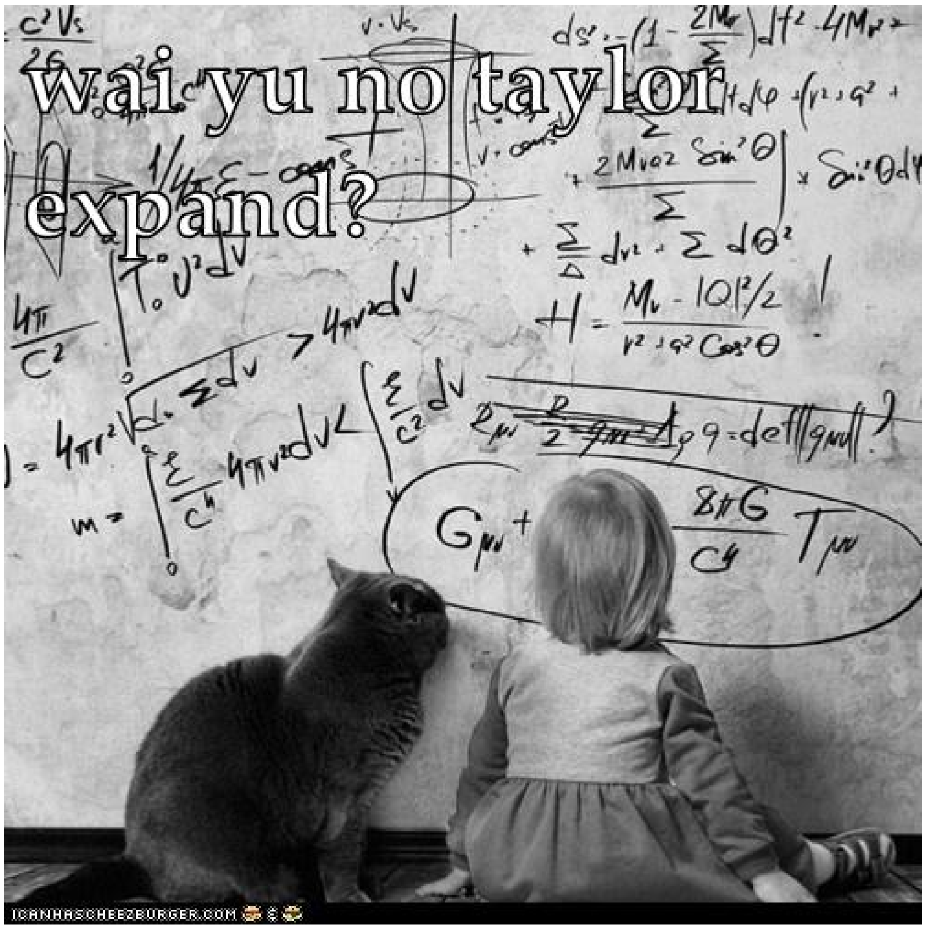


Figure 4.5: Sound advice when stuck on almost any problem in statistical theory.

for $1.5 < x_i < 1.7$ with equal weights. For f, this gives 0.46, while $f(1.6) = 0.89$. For g, this gives 0.98, with $g(1.6) = 0.95$. (See figure 4.6). The same size window introduces a much larger bias with the rougher, more rapidly changing f than with the smoother, more slowly changing g. Varying the size of the averaging window will change the amount of error, and it will change it in different ways for the two functions.

If one does a more careful second-order Taylor expansion like that leading to Eq. 4.9, specifically for kernel regression, one can show that the bias at x is

$$\mathbf{E}\left[\hat{r}(x) - r(x)|X_1 = x_1, \ldots X_n = x_n\right] = h^2\left[\frac{1}{2}r''(x) + \frac{r'(x)f'(x)}{f(x)}\right]\sigma^2_K + o(h^2) \quad (4.10)$$

where f is the density of x, and $\sigma^2_K = \int u^2 K(u)du$, the variance of the probability density corresponding to the kernel[5]. The r'' term just comes from the second-order part of the Taylor expansion. To see where the $r'f'$ term comes from, imagine first that x is a mode of the distribution, so $f'(x) = 0$. As h shrinks, only training points where X_i is very close to x will have any weight in $\hat{r}(x)$, and their distribution will be roughly symmetric around x (at least once h is sufficiently small). So, at mode, $\mathbf{E}\left[w(X_i, x, h)(X_i - x)\hat{r}(x)\right] \approx 0$. Away from a mode, there will tend to be more training points on one side or the other of x, depending on the sign of $f'(x)$, and this induces a bias. The tricky part of the analysis is concluding that the bias has exactly the form given above.[6]

One can also work out the variance of the kernel regression estimate,

$$\mathrm{Var}\left[\hat{r}(x)|X_1 = x_1, \ldots X_n = x_n\right] = \frac{\sigma^2(x)R(K)}{nhf(x)} + o((nh)^{-1}) \quad (4.11)$$

where $R(K) = \int K^2(u)du$. Roughly speaking, the width of the region where the kernel puts non-trivial weight is about h, so there will be about $nhf(x)$ training points available to estimate $\hat{r}(x)$. Each of these has a y_i value, equal to $r(x)$ plus noise of variance $\sigma^2(x)$. The final factor of $R(K)$ accounts for the average weight.

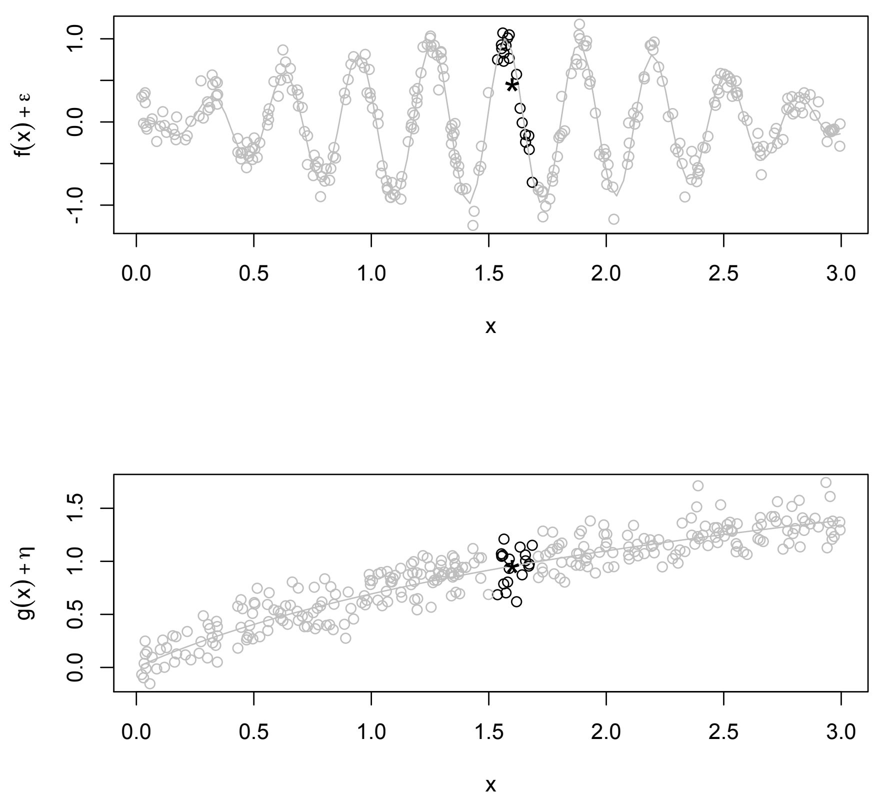
Putting the bias together with the variance, we get an expression for the mean squared error of the kernel regression at x:

$$MSE(x) = \sigma^2(x) + h^4\left[\frac{1}{2}r''(x) + \frac{r'(x)f'(x)}{f(x)}\right]^2 (\sigma^2_K)^2 + \frac{\sigma^2(x)R(K)}{nhf(x)} + o(h^4) + o(1/nh) \quad (4.12)$$

Eq. 4.12 tells us that, in principle, there is a single optimal choice of bandwidth h, an optimal degree of smoothing. We could find it by taking Eq. 4.12, differentiating with

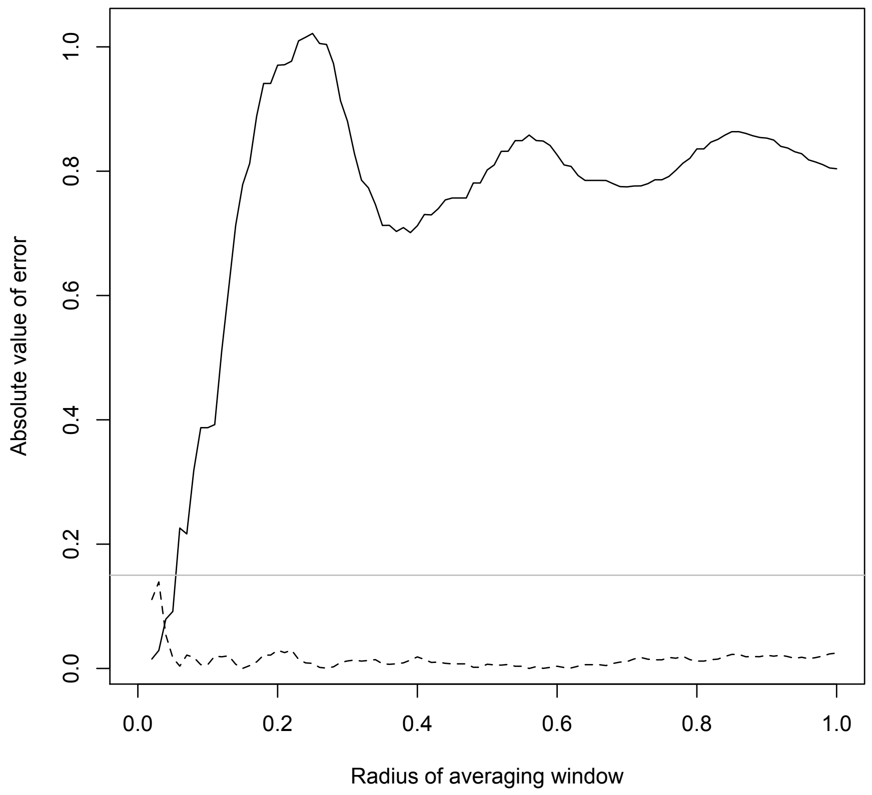
[5]If you are not familiar with the "order" symbols O and o, now would be a good time to read Appendix B.

[6]Exercise 1 shows how to do a bit of the demonstration for the special case of the uniform (boxcar) kernel.



```
par(mfcol=c(2,1))
colors=ifelse((x<1.7)&(x>1.5),"black","grey")
plot(x,yf,xlab="x",ylab=expression(f(x)+epsilon),col=colors)
curve(sin(x)*cos(20*x),col="grey",add=TRUE)
points(1.6,mean(yf[(x<1.7)&(x>1.5)]),pch="*",cex=2)
plot(x,yg,xlab="x",ylab=expression(g(x)+eta),col=colors)
curve(log(x+1),col="grey",add=TRUE)
points(1.6,mean(yg[(x<1.7)&(x>1.5)]),pch="*",cex=2)
```

Figure 4.6: Relationship between smoothing and function roughness. In both the upper and lower panel we are trying to estimate the value of the regression function at $x = 1.6$ from averaging observations taken with $1.5 < x_i < 1.7$ (black points, others are "ghosted" in grey). The location of the average in shown by the large black X. Averaging over this window works poorly for the rough function $f(x)$ in the upper panel (the bias is large), but much better for the smoother function in the lower panel (the bias is small).



```
loc_ave_err <- function(h,y,y0) {abs(y0-mean(y[(1.6-h < x) & (1.6+h>x)]))}
yf0=sin(1.6)*cos(20*1.6)
yg0=log(1+1.6)
f.LAE = sapply(0:100/100,loc_ave_err,y=yf,y0=yf0)
g.LAE = sapply(0:100/100,loc_ave_err,y=yg,y0=yg0)
plot(0:100/100,f.LAE,xlab="Radius of averaging window",
     ylab="Absolute value of error",type="l")
lines(0:100/100,g.LAE,lty=2)
abline(h=0.15,col="grey")
```

Figure 4.7: Estimating $f(1.6)$ and $g(1.6)$ from averaging observed values at $1.6 - h < x < 1.6 + h$, for different radii h. Solid line: error of estimates of $f(1.6)$; dashed line: error of estimates of $g(1.6)$; grey line: σ, the standard deviation of the noise.

respect to the bandwidth, and setting everything to zero (neglecting the o terms):

$$0 = 4h^3 \left[\frac{1}{2}r''(x) + \frac{r'(x)f'(x)}{f(x)}\right]^2 (\sigma_K^2)^2 - \frac{\sigma^2(x)R(K)}{nh^2 f(x)} \quad (4.13)$$

$$h = \left(n \frac{4f(x)(\sigma_K^2)^2 \left[\frac{1}{2}r''(x) + \frac{r'(x)f'(x)}{f(x)}\right]^2}{\sigma^2(x)R(K)} \right)^{-1/5} \quad (4.14)$$

Of course, this expression for the optimal h involves the unknown derivatives $r'(x)$ and $r''(x)$, plus the unknown density $f(x)$ and its unknown derivative $f'(x)$. But if we knew the derivative of the regression function, we would basically know the function itself (just integrate), so we seem to be in a vicious circle, where we need to know the function before we can learn it.[7]

One way of expressing this is to talk about how well a smoothing procedure *would* work, if an Oracle were to tell us the derivatives, or (to cut to the chase) the optimal bandwidth h_{opt}. Since most of us do not have access to such oracles, we need to *estimate* h_{opt}. Once we have this estimate, $\widehat{h}$, then we get out weights and our predictions, and so a certain mean-squared error. Basically, our MSE will be the Oracle's MSE, plus an extra term which depends on how far $\widehat{h}$ is to h_{opt}, and how sensitive the smoother is to the choice of bandwidth.

What would be really nice would be an **adaptive** procedure, one where our actual MSE, using $\widehat{h}$, approaches the Oracle's MSE, which it gets from h_{opt}. This would mean that, in effect, we are *figuring out* how rough the underlying regression function is, and so how much smoothing to do, rather than having to guess or be told. An adaptive procedure, if we can find one, is a partial[8] substitute for prior knowledge.

4.2.1 Bandwidth Selection by Cross-Validation

The most straight-forward way to pick a bandwidth, and one which generally manages to be adaptive, is in fact cross-validation; k-fold CV is usually somewhat better than leave-one-out, but the latter often works acceptably too. The usual procedure is to come up with an initial grid of candidate bandwidths, and then use cross-validation to estimate how well each one of them would generalize. The one with the lowest error under cross-validation is then used to fit the regression curve to the whole data[9].

Code Example 2 shows how it would work in R, with the values of the input variable being in the vector x (one dimensional) and the response in the vector y

[7]You may be wondering, at this point, why I keep talking about *the* optimal bandwidth, when it would seem, from Eq. 4.14, that the bandwidth should vary with x. One can actually go through pretty much the same sort of analysis in terms of the *expected* values of the derivatives, and the qualitative conclusions will be the same, but the notational overhead is even worse. Alternatively, there are techniques for variable-bandwidth smoothing.

[8]Only partial, because we'd *always* do better if the Oracle would just tell us h_{opt}.

[9]Since the optimal bandwidth is $\propto n^{-1/5}$, and the training sets in cross-validation are smaller than the whole data set, one might adjust the bandwidth proportionally. However, if n is small enough that this makes a big difference, the sheer noise in bandwidth estimation usually overwhelms this.

(also one dimensional), using the `npreg` function from the `np` library (Hayfield and Racine, 2008).[10]

The return value has three parts. The first is the actual best bandwidth. The second is a vector which gives the cross-validated mean-squared mean-squared errors of all the different bandwidths in the vector `bandwidths`. The third component is an array which gives the MSE for each bandwidth on each fold. It can be useful to know things like whether the difference between the CV score of the best bandwidth and the runner-up is bigger than their fold-to-fold variability.

Figure 4.8 plots the CV estimate of the (root) mean-squared error versus bandwidth for our two curves. Figure 4.9 shows the data, the actual regression functions and the estimated curves with the CV-selected bandwidths. This illustrates *why* picking the bandwidth by cross-validation works: the curve of CV error against bandwidth is actually a pretty good approximation to the true curve of generalization error against bandwidth (which would look like Figure 4.1), and so optimizing over the CV curve is close to optimizing over the generalization error curve. If we had a problem where cross-validation didn't give good estimates of generalization error, this wouldn't work.

Notice, by the way, in Figure 4.8, that the rougher curve is more sensitive to the choice of bandwidth, and that the smoother curve always has a lower mean-squared error. Also notice that, at the minimum, one of the cross-validation estimates of generalization error is smaller than the true system noise level; this shows that cross-validation doesn't completely correct for optimism[11].

We still need to come up with an initial set of candidate bandwidths. For reasons which will drop out of the math in Chapter 15, it's often reasonable to start around $1.06 s_X / n^{1/5}$, where s_X is the sample standard deviation of X. However, it is hard to be very precise about this, and good results often require some honest trial and error.

4.2.2 Convergence of Kernel Smoothing and Bandwidth Scaling

Go back to Eq. 4.12 for the mean squared error of kernel regression. As we said, it involves some unknown constants, but we can bury them inside big-O order symbols, which also absorb the little-o remainder terms:

$$MSE(h) = \sigma^2(x) + O(h^4) + O(1/nh) \tag{4.15}$$

The $\sigma^2(x)$ term is going to be there no matter what, so let's look at the excess risk over and above the intrinsic noise:

$$MSE(h) - \sigma^2(x) = O(h^4) + O(1/nh) \tag{4.16}$$

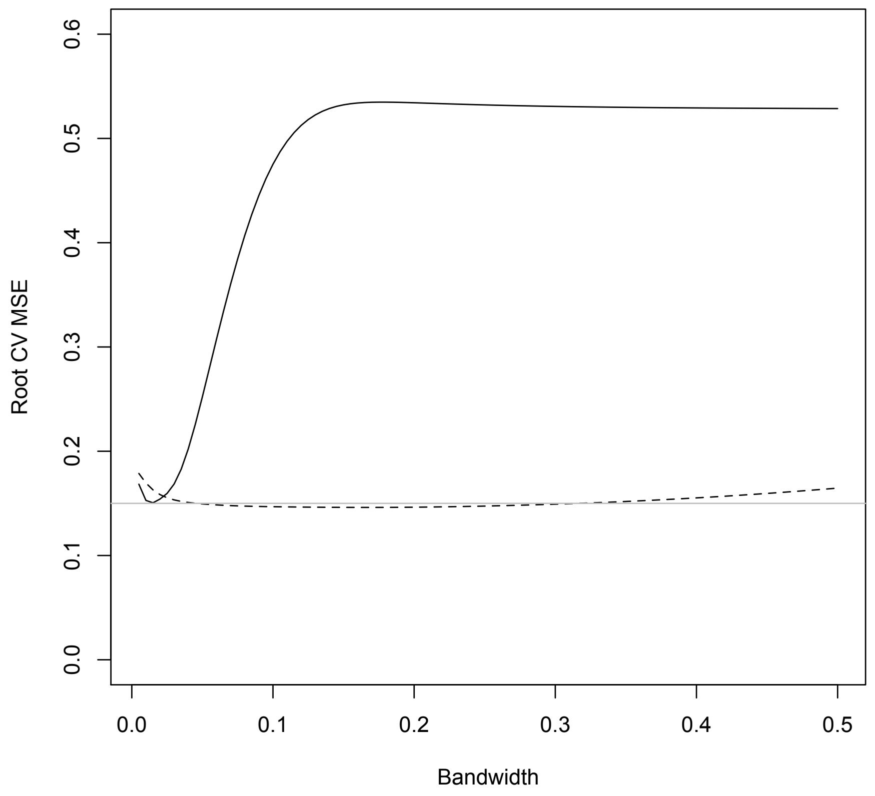
That is, the (squared) bias from the kernel's only approximately getting the curve is proportional to the fourth power of the bandwidth, but the variance is inversely

[10] The `np` package actually has a function, `npregbw`, which automatically selects bandwidths through a sophisticated combination of cross-validation and optimization techniques. The default settings for this function make it very slow, by trying very, very hard to optimize the bandwidth.

[11] Tibshirani and Tibshirani (2009) gives a fairly straightforward way to adjust the estimate of the generalization error for the selected model or bandwidth, but that doesn't influence the choice of the best bandwidth.

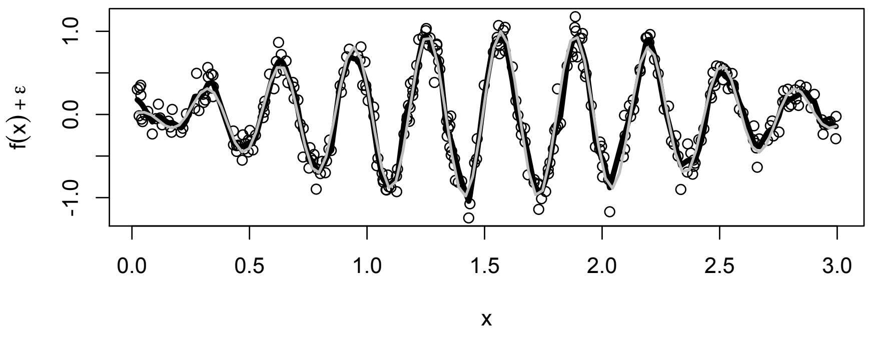
```
# Cross-validation for univariate kernel regression
cv_bws_npreg <- function(x,y,bandwidths=(1:50)/50,
  num.folds=10) {
  require(np)
  n <- length(x)
  stopifnot(n> 1, length(y) == n)
  stopifnot(length(bandwidths) > 1)
  stopifnot(num.folds > 0, num.folds==trunc(num.folds))

  fold_MSEs <- matrix(0,nrow=num.folds,
    ncol=length(bandwidths))
  colnames(fold_MSEs) = bandwidths

  case.folds <- rep(1:num.folds,length.out=n)
  case.folds <- sample(case.folds)
  for (fold in 1:num.folds) {
    train.rows = which(case.folds==fold)
    x.train = x[train.rows]
    y.train = y[train.rows]
    x.test = x[-train.rows]
    y.test = y[-train.rows]
    for (bw in bandwidths) {
      fit <- npreg(txdat=x.train,tydat=y.train,
                   exdat=x.test,eydat=y.test,bws=bw)
      fold_MSEs[fold,paste(bw)] <- fit$MSE
    }
  }
  CV_MSEs = colMeans(fold_MSEs)
  best.bw = bandwidths[which.min(CV_MSEs)]
  return(list(best.bw=best.bw,
    CV_MSEs=CV_MSEs,
    fold_MSEs=fold_MSEs))
}
```

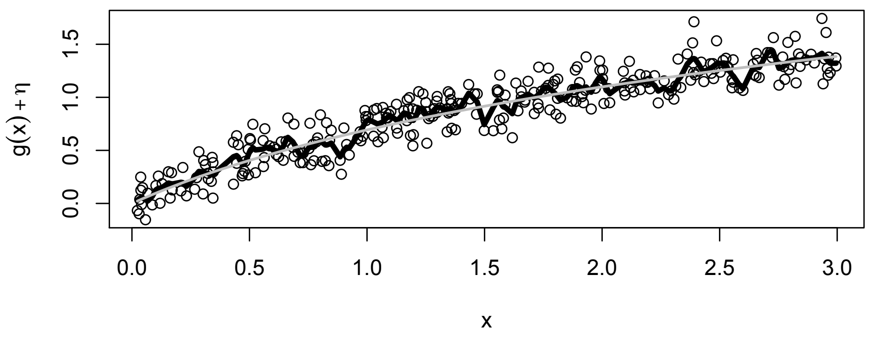
Code Example 2: Comments omitted here to save space; see the accompanying R file on the class website. The `colnames` trick: component names have to be character strings; other data types will be coerced into characters when we assign them to be names. Later, when we want to refer to a bandwidth column by its name, we wrap the name in another coercing function, such as `paste`. — The vector of default bandwidths is arbitrary and only provided for illustration; it should not be blindly copied and used on data (or on homework problems).



```
fbws <- cv_bws_npreg(x,yf,bandwidths=(1:100)/200)
gbws <- cv_bws_npreg(x,yg,bandwidths=(1:100)/200)
plot(1:100/200,sqrt(fbws$CV_MSEs),xlab="Bandwidth",
  ylab="Root CV MSE",type="l",ylim=c(0,0.6))
lines(1:100/200,sqrt(gbws$CV_MSEs),lty=2)
abline(h=0.15,col="grey")
```

Figure 4.8: Cross-validated estimate of the (root) mean-squared error as a function of the bandwidth. Solid curve: data from $f(x)$; dashed curve: data from $g(x)$; grey line: true σ. Notice that the rougher curve is more sensitive to the choice of bandwidth, and that the smoother curve is more predictable at every choice of bandwidth. Also notice that CV does not *completely* compensate for the optimism of in-sample fitting (see where the dashed curve falls below the grey line). CV selects bandwidths of 0.015 for f and 0.165 for g.





```
x.ord=order(x)
par(mfcol=c(2,1))
plot(x,yf,xlab="x",ylab=expression(f(x)+epsilon))
fhat <- npreg(bws=fbws$best.bw,txdat=x,tydat=yf)
lines(x[x.ord],fitted(fhat)[x.ord],lwd=4)
curve(sin(x)*cos(20*x),col="grey",add=TRUE,lwd=2)
plot(x,yg,xlab="x",ylab=expression(g(x)+eta))
ghat <- npreg(bws=fbws$best.bw,txdat=x,tydat=yg)
lines(x[x.ord],fitted(ghat)[x.ord],lwd=4)
curve(log(x+1),col="grey",add=TRUE,lwd=2)
```

Figure 4.9: Data from the running examples (circles), true regression functions (grey) and kernel estimates of regression functions with CV-selected bandwidths (black). The widths of the regression functions are exaggerated. Since the `x` values aren't sorted, we need to put them in order if we want to draw lines connecting the fitted values; then we need to put the fitted values in the same order. An alternative would be to use `predict` on the sorted values, as in the next section.

proportional to the product of sample size and bandwidth. If we kept h constant and just let $n \rightarrow \infty$, we'd get rid of the variance, but we'd be left with the bias. To get the MSE to go to zero, we need to let the bandwidth h change with n — call it h_n. Specifically, suppose $h_n \rightarrow 0$ as $n \rightarrow \infty$, but $nh_n \rightarrow \infty$. Then, by Eq. 4.16, the risk (generalization error) of kernel smoothing is approaching that of the ideal predictor or regression function.

What is the best bandwidth? We saw in Eq. 4.14 that it is (up to constants)

$$h_{\text{opt}} = O(n^{-1/5}) \tag{4.17}$$

If we put this bandwidth into Eq. 4.16, we get

$$MSE(h) - \sigma^2(x) = O\left(\left(n^{-1/5}\right)^4\right) + O\left(n^{-1}\left(n^{-1/5}\right)^{-1}\right) = O\left(n^{-4/5}\right) + O\left(n^{-4/5}\right) = O\left(n^{-4/5}\right) \tag{4.18}$$

That is, the excess prediction error of kernel smoothing over and above the system noise goes to zero as $1/n^{0.8}$. Notice, by the way, that the contributions of bias and variance to the generalization error are both of the same order, $n^{-0.8}$.

Is this fast or small? We can compare it to what would happen with a parametric model, say with parameter θ. (Think, for instance, of linear regression, but not only linear regression.) There is an optimal value of the parameter, θ_0, which would minimize the mean-squared error. At θ_0, the parametric model has MSE

$$MSE(\theta_0) = \sigma^2(x) + b(x, \theta_0) \tag{4.19}$$

where b is the bias of the parametric model; this is zero when the parametric model is true[12]. Since θ_0 is unknown and must be estimated, one typically has $\widehat{\theta} - \theta_0 = O(1/\sqrt{n})$. A first-order Taylor expansion of the parametric model contributes an error $O(\widehat{\theta} - \theta_0)$, so altogether

$$MSE(\widehat{\theta}) - \sigma^2(x) = b(x, \theta_0) + O(1/n) \tag{4.20}$$

So parametric models converge more quickly (n^{-1} goes to zero faster than $n^{-0.8}$), but they will typically converge to the wrong answer ($b^2 > 0$). Kernel smoothing converges a bit more slowly, but always converges to the right answer[13].

How does all this change if h must be found by cross-validation? If we write $\widehat{h_{CV}}$ for the bandwidth picked by cross-validation, the one can show (Simonoff, 1996, ch. 5) that

$$\frac{\widehat{h_{CV}} - h_{\text{opt}}}{h_{\text{opt}}} - 1 = O(n^{-1/10}) \tag{4.21}$$

[12]When the parametric model is not true, the optimal parameter value θ_0 is often called the **pseudo-truth**.

[13]It is natural to wonder if one couldn't do better than kernel smoothing's $O(n^{-4/5})$ while still having no asymptotic bias. Resolving this is very difficult, but the answer turns out to be "no" in the following sense (Wasserman, 2006). Any curve-fitting method which can learn arbitrary smooth regression functions will have some curves where it cannot converge any faster than $O(n^{-4/5})$. (In the jargon, that is the **minimax rate**.) Methods which converge faster than this for some kinds of curves have to converge more slowly for others. So this is the best rate we can hope for on truly unknown curves.

Given this, one concludes (EXERCISE 2) that the MSE of using $\widehat{h_{CV}}$ is also $O(n^{-4/5})$.

4.2.3 Summary on Kernel Smoothing

Suppose that X and Y are both one-dimensional, and the true regression function $r(x) = \mathbf{E}[Y|X = x]$ is continuous and has first and second derivatives[14]. Suppose that the noise around the true regression function is uncorrelated between different observations. Then the bias of kernel smoothing, when the kernel has bandwidth h, is $O(h^2)$, and the variance, after n samples, is $O(1/nh)$. The optimal bandwidth is $O(n^{-1/5})$, and the excess mean squared error of using this bandwidth is $O(n^{-4/5})$. If the bandwidth is selected by cross-validation, the excess risk is still $O(n^{-4/5})$.

4.3 Kernel Regression with Multiple Inputs

For the most part, when I've been writing out kernel regression I have been treating the input variable x as a scalar. There's no reason to insist on this, however; it could equally well be a vector. If we want to enforce that in the notation, say by writing $\vec{x} = (x^1, x^2, \ldots x^d)$, then the kernel regression of y on $\vec{x}$ would just be

$$\hat{r}(\vec{x}) = \sum_{i=1}^{n} y_i \frac{K(\vec{x} - \vec{x_i})}{\sum_{j=1}^{n} K(\vec{x} - \vec{x_j})} \tag{4.22}$$

In fact, if we want to predict a vector, we'd just substitute $\vec{y_i}$ for y_i above.

To make this work, we need kernel functions for vectors. For scalars, I said that any probability density function would work so long as it had mean zero, and a finite, strictly positive (not 0 or ∞) variance. The same conditions carry over: any distribution over vectors can be used as a multivariate kernel, provided it has mean zero, and the variance matrix is finite and strictly positive[15]. In practice, the overwhelmingly most common and practical choice is to use **product kernels**[16].

A product kernel simply uses a different kernel for each component, and then multiplies them together:

$$K(\vec{x} - \vec{x_i}) = K_1(x^1 - x_i^1)K_2(x^2 - x_i^2)\ldots K_d(x^d - x_i^d) \tag{4.23}$$

Now we just need to pick a bandwidth for each kernel, which in general should not be equal — say $\vec{h} = (h_1, h_2, \ldots h_d)$. Instead of having a one-dimensional error curve, as in Figure 4.1 or 4.2, we will have a d-dimensional error surface, but we can still use cross-validation to find the vector of bandwidths that generalizes best. We generally can't, unfortunately, break the problem up into somehow picking the best bandwidth for each variable without considering the others. This makes it slower to select good bandwidths in multivariate problems, but still often feasible.

[14]Or can be approximated to arbitrarily closely by such functions.

[15]Remember that for a matrix $\mathbf{v}$ to be "strictly positive", it must be the case that for any vector $\vec{a}$, $\vec{a} \cdot \mathbf{v}\vec{a} > 0$. Covariance matrices are automatically non-negative, so we're just ruling out the case of some weird direction along which the distribution has zero variance.

[16]People do sometimes use multivariate Gaussians; we'll glance at this in Chapter 14.

(We can actually turn the need to select bandwidths together to our advantage. If one or more of the variables are irrelevant to our prediction given the others, cross-validation will tend to give them the maximum possible bandwidth, and smooth away their influence. In Chapter 15, we'll look at formal tests based on this idea.)

Kernel regression will recover almost any regression function. This is true even when the true regression function involves lots of interactions among the input variables, perhaps in complicated forms that would be very hard to express in linear regression. For instance, Figure 4.10 shows a contour plot of a reasonably complicated regression surface, at least if one were to write it as polynomials in x^1 and x^2, which would be the usual approach. Figure 4.12 shows the estimate we get with a product of Gaussian kernels and only 1000 noisy data points. It's not perfect, of course (in particular the estimated contours aren't as perfectly smooth and round as the true ones), but the important thing is that we got this without having to know, and describe in Cartesian coordinates, the type of shape we were looking for. Kernel smoothing *discovered* the right general form.

There are limits to these abilities of kernel smoothers; the biggest one is that they require more and more data as the number of predictor variables increases. We will see later (Chapter 9) exactly how much data is required, generalizing the kind of analysis done §4.2.2, and some of the compromises this can force us into.

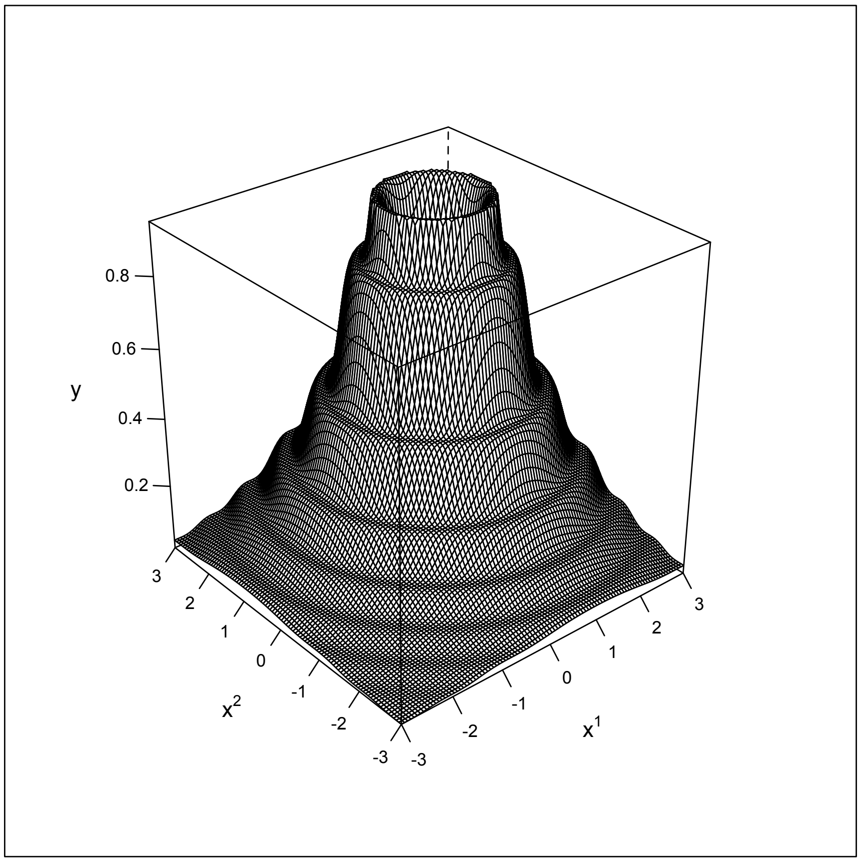
4.4 Interpreting Smoothers: Plots

In a linear regression without interactions, it is fairly easy to interpret the coefficients. The expected response changes by β_i for a one-unit change in the i^{th} input variable. The coefficients are also the derivatives of the expected response with respect to the inputs. And it is easy to draw pictures of how the output changes as the inputs are varied, though the pictures are somewhat boring (straight lines or planes).

As soon as we introduce interactions, all this becomes harder, even for parametric regression. If there is an interaction between two components of the input, say x^1 and x^2, then we can't talk about the change in the expected response for a one-unit change in x^1 without saying what x^2 is. We might *average* over x^2 values, and we'll see next time a reasonable way of doing this, but the flat statement "increasing x^1 by one unit increases the response by β_1" is just false, no matter what number we fill in for β_1. Likewise for derivatives; we'll come back to them next time as well.

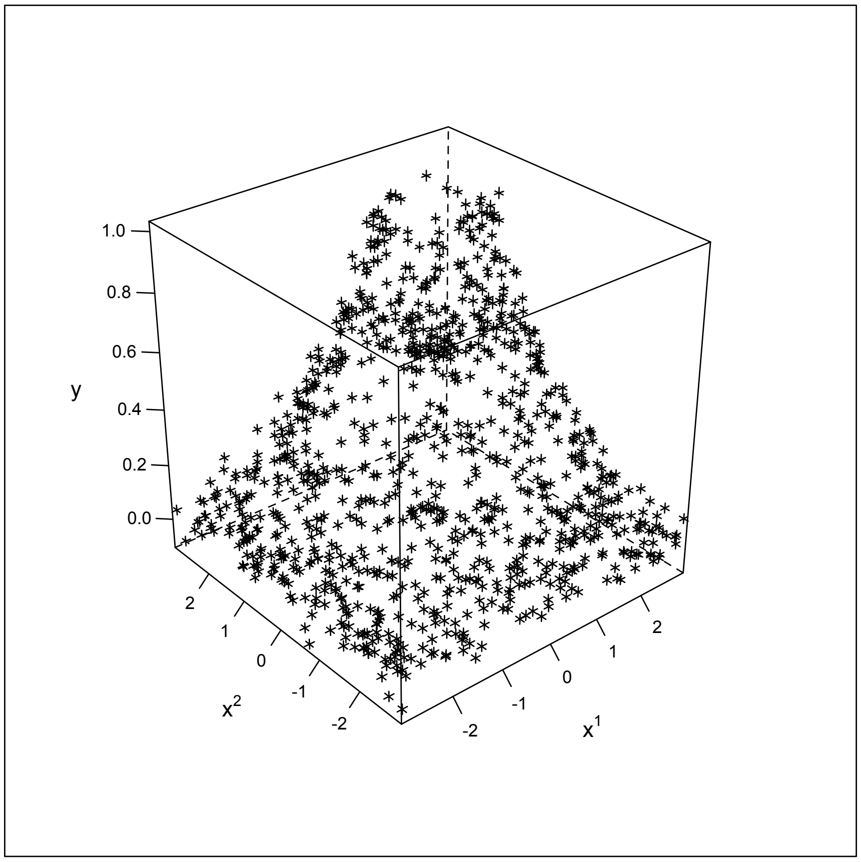
What about pictures? If there are only two input variables, then we can make plots like the wireframes in the previous section, or contour- or level- plots, which will show the predictions for different combinations of the two variables. But suppose we want to look at one variable at a time? Suppose there are more than two input variables?

A reasonable way of producing *a* curve for each input variable is to set all the others to some "typical" value, such as the mean or the median, and to then plot the predicted response as a function of the one remaining variable of interest. See Figure 4.13 for an example of this. Of course, when there are interactions, changing the values of the other inputs will change the response to the input of interest, so it may be a good idea to produce a couple of curves, possibly super-imposed (again, see



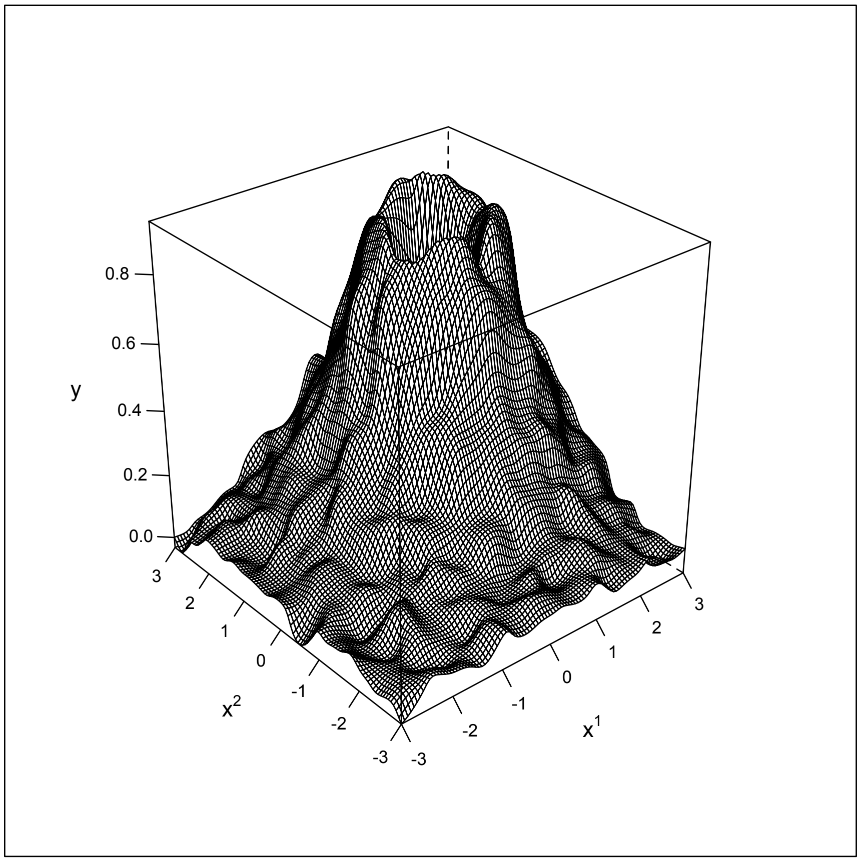
```
x1.points <- seq(-3,3,length.out=100)
x2.points <- x1.points
x12grid <- expand.grid(x1=x1.points,x2=x2.points)
y <- matrix(0,nrow=100,ncol=100)
y <- outer(x1.points,x2.points,f)
library(lattice)
wireframe(y~x12grid$x1*x12grid$x2,scales=list(arrows=FALSE),
  xlab=expression(x^1),ylab=expression(x^2),zlab="y")
```

Figure 4.10: An example of a regression surface that would be very hard to learn by piling together interaction terms in a linear regression framework. (Can you guess what the mystery function `f` is?) — `wireframe` is from the graphics library `lattice`.



```
x1.noise <- runif(1000,min=-3,max=3)
x2.noise <- runif(1000,min=-3,max=3)
y.noise <- f(x1.noise,x2.noise)+rnorm(1000,0,0.05)
noise <- data.frame(y=y.noise,x1=x1.noise,x2=x2.noise)
cloud(z~x*y,data=noise,col="black",scales=list(arrows=FALSE),
      xlab=expression(x^1),ylab=expression(x^2),zlab="y")
```

Figure 4.11: 1000 data points, randomly sampled from the surface in Figure 4.10, plus independent Gaussian noise (s.d. = 0.05).



```
noise.np <- npreg(y~x1+x2,data=noise)
y.out <- matrix(0,100,100)
y.out <- predict(noise.np,newdata=x12grid)
wireframe(y.out~x12grid$x1*x12grid$x2,scales=list(arrows=FALSE),
          xlab=expression(x^1),ylab=expression(x^2),zlab="y")
```

Figure 4.12: Gaussian kernel regression of the points in Figure 4.11. Notice that the estimated function will make predictions at arbitrary points, not just the places where there was training data.

Figure 4.13).

If there are three or more input variables, we can look at the interactions of any two of them, taken together, by fixing the others and making three-dimensional or contour plots, along the same principles.

The fact that smoothers don't give us a simple story about how each input is associated with the response may seem like a disadvantage compared to using linear regression. Whether it really is a disadvantage depends on whether there really is a simple story to be told — and, if there isn't, how big a lie you are prepared to tell in order to keep your story simple.

4.5 Average Predictive Comparisons

Suppose we have a linear regression model

$$Y = \beta_1 X^1 + \beta_2 X^2 + \epsilon \tag{4.24}$$

and we want to know how much Y changes, on average, for a one-unit increase in X^1. The answer, as you know very well, is just β_1:

$$[\beta_1(X^1+1) + \beta_2 X^2] - [\beta_1 X^1 + \beta_2 X^2] = \beta_1 \tag{4.25}$$

This is an interpretation of the regression coefficients which you are very used to giving. But it fails as soon as we have interactions:

$$Y = \beta_1 X^1 + \beta_2 X^2 + \beta_3 X^1 X^2 + \epsilon \tag{4.26}$$

Now the effect of increasing X^1 by 1 is

$$[\beta_1(X^1+1) + \beta_2 X^2 + \beta_3(X^1+1)X^2] - [\beta_1 X^1 + \beta_2 X^2 + \beta_3 X^1 X^2] = \beta_1 + \beta_3 X^2 \tag{4.27}$$

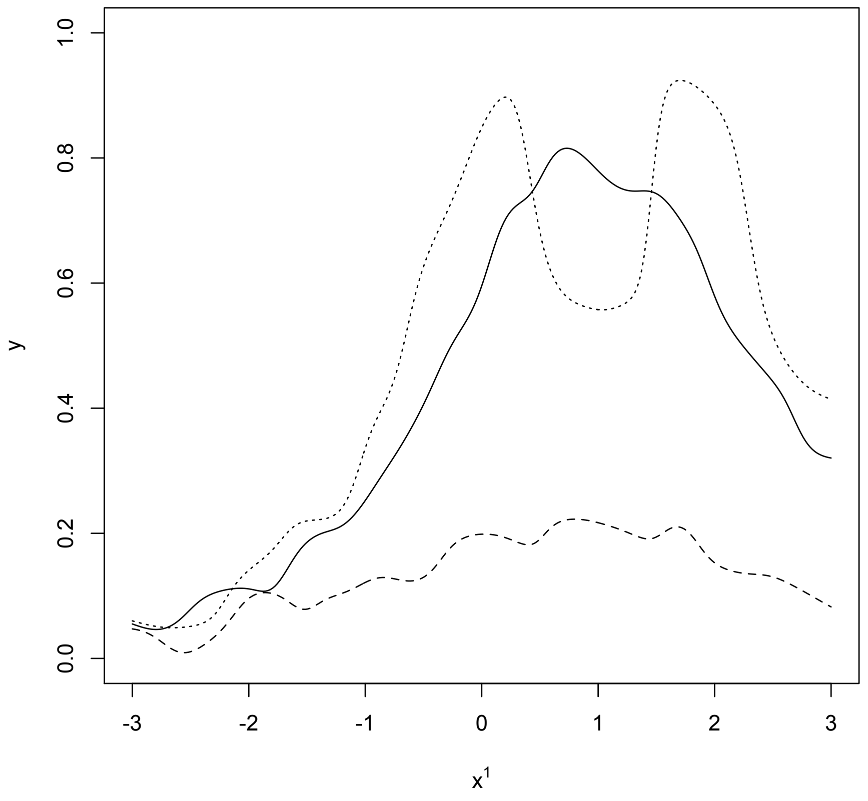
There just isn't one answer "how much does the response change when X^1 is increased by one unit?", it depends on the value of X^2. We certainly can't just answer "β_1".

We also can't give just a single answer if there are nonlinearities. Suppose that the true regression function is this:

$$Y = \frac{e^{\beta X}}{1 + e^{\beta X}} + \epsilon \tag{4.28}$$

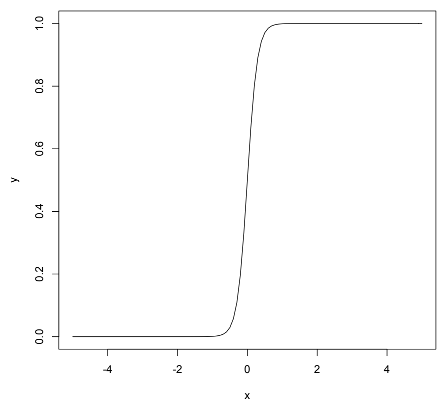
which looks like Figure 4.14, setting $\beta = 7$ (for luck). Moving x from -4 to -3 increases the response by 7.57×10^{-10}, but the increase in the response from $x = -1$ to $x = 0$ is 0.499. Functions like this are very common in psychology, medicine (dose-response curves for drugs), biology, etc., and yet we cannot sensibly talk about *the* response to a one-unit increase in x. (We will come back to curves which look like this in Chapter 12.)

More generally, let's say we are regressing Y on a vector $\vec{X}$, and want to assess the impact of one component of the input on Y. To keep the use of subscripts and



```
new.frame <- data.frame(x=seq(-3,3,length.out=300),y=median(y.noise))
plot(new.frame$x,predict(noise.np,newdata=new.frame),
  type="l",xlab=expression(x^1),ylab="y",ylim=c(0,1.0))
new.frame$y <- quantile(y.noise,0.25)
lines(new.frame$x,predict(noise.np,newdata=new.frame),lty=2)
new.frame$y <- quantile(y.noise,0.75)
lines(new.frame$x,predict(noise.np,newdata=new.frame),lty=3)
```

Figure 4.13: Predicted mean response as function of the first input coordinate x^1 for the example data, evaluated with the second coordinate x^2 set to the median (solid), its 25th percentile (dashed) and its 75th percentile (dotted). Note that the changing shape of the partial response curve indicates an interaction between the two inputs. Also, note that the model is able to make predictions at arbitrary coordinates, whether or not there were any training points there. (It happened that no observation was exactly at the median, the 25th or the 75th percentile for the second input.)



```
curve(exp(7*x)/(1+exp(7*x)),from=-5,to=5,ylab="y")
```

Figure 4.14: The function of Eq. 4.28, with $\beta = 7$.

superscripts to a minimum, we'll write $\vec{X} = (u, \vec{V})$, where u is the coordinate we're really interested in. (It doesn't have to come first, of course.) We would like to know how much the prediction changes as we change u,

$$EY|\vec{X} = (u^{(2)}, \vec{v}) - EY|\vec{X} = (u^{(1)}, \vec{v}) \tag{4.29}$$

and the change in the response per unit change in u,

$$\frac{EY|\vec{X} = (u^{(2)}, \vec{v}) - EY|\vec{X} = (u^{(1)}, \vec{v})}{u^{(2)} - u^{(1)}} \tag{4.30}$$

Both of these, but especially the latter, are called the **predictive comparison**. Note that both of them, as written, depend on $u^{(1)}$ (the starting value for the variable of interest), on $u^{(2)}$ (the ending value), and on $\vec{v}$ (the other variables, held fixed during this comparison). We have just seen that in a linear model without interactions, $u^{(1)}$, $u^{(2)}$ and $\vec{v}$ all go away and leave us with the regression coefficient on u. In nonlinear or interacting models, we can't simplify so much.

Once we have estimated a regression model, we can chose our starting point, ending point and context, and just plug in to Eq. 4.29 or Eq. 4.30. (Take a look again at problem 6 on Homework 2.) But suppose we do want to boil this down into a single number for each input variable — how might we go about this?

One good answer, which comes from Gelman and Pardoe (2007), is just to average 4.30 over the data[17] More specifically, we have as our **average predictive comparison**

[17]Actually, they propose something very slightly more complicated, which takes into account the uncertainty in our estimate of the regression function. We'll come back to this in a few lectures when we see how to quantify uncertainty in complex models.

for u

$$\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\hat{r}(u_j,\vec{v}_i)-\hat{r}(u_i,\vec{v}_j)\text{sign}(u_j-u_i)}{(u_j-u_i)\text{sign}(u_j-u_i)} \tag{4.31}$$

where i and j run over data points, $\hat{r}$ is our estimated regression function, and the sign function is defined by $\text{sign}(x)=+1$ if $x>0$, $=0$ if $x=0$, and $=-1$ if $x<0$. We use the sign function this way to make sure we are always looking at the consequences of *increasing* u.

The average predictive comparison provides a reasonable summary measure of how one should expect the response to vary as u changes slightly. But, once the model is nonlinear or allows interactions, it's just not possible to summarize the predictive relationship between u and y with a single number, and so the value of Eq. 4.31 is going to depend on the distribution of u (and possible of v), even when the regression function is unchanged. (See Exercise 3.)

4.6 Exercises

1. Suppose we use a uniform ("boxcar") kernel extending over the region $(-h/2, h/2)$. Show that

$$\begin{aligned}\mathbf{E}[\hat{r}(0)] &= \mathbf{E}\left[r(X)\,|-\frac{h}{2}<X<\frac{h}{2}\right] &(4.32)\\ &= r(0)+r'(0)\mathbf{E}\left[X\,|-\frac{h}{2}<X<\frac{h}{2}\right] &(4.33)\\ &\quad+\frac{r''(0)}{2}\mathbf{E}\left[X^2|-\frac{h}{2}<X<\frac{h}{2}\right]+o(h^2)\end{aligned}$$

   Show that $\mathbf{E}\left[X\,|-\frac{h}{2}<X<\frac{h}{2}\right]=O(r'(0)f'(0)h^2)$, and that $\mathbf{E}\left[X^2|-\frac{h}{2}<X<\frac{h}{2}\right]=O(h^2)$. Conclude that the over-all bias is $O(h^2)$.

2. Use Eqs. 4.21, 4.17 and 4.16 to show that the excess risk of the kernel smoothing, when the bandwidth is selected by cross-validation, is also $O(n^{-4/5})$.

3. Generate 1000 data points where X is uniformly distributed between -4 and 4, and $Y=e^{7x}/(1+e^{7x})+\epsilon$, with ϵ Gaussian and with variance 0.01. Use non-parametric regression to estimate $\hat{r}(x)$, and then use Eq. 4.31 to find the average predictive comparison. Now re-run the simulation with X uniform on the interval $[0, 0.5]$ and re-calculate the average predictive comparison. What happened?