# 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:

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

Most smoothing methods have a control setting, here written $h$, that says *how much* to smooth. 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.

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

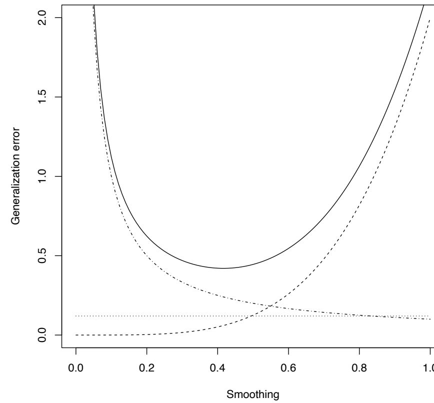
$$\text{(total error)} = \text{(noise)} + \text{(bias)}^2 + \text{(variance)} \tag{4.2}$$

(Eq. 1.28), 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



11:21 Wednesday 24th January, 2024
Copyright ©Cosma Rohilla Shalizi; do not distribute without permission
updates at http://www.stat.cmu.edu/~cshalizi/ADAfaEPoV/



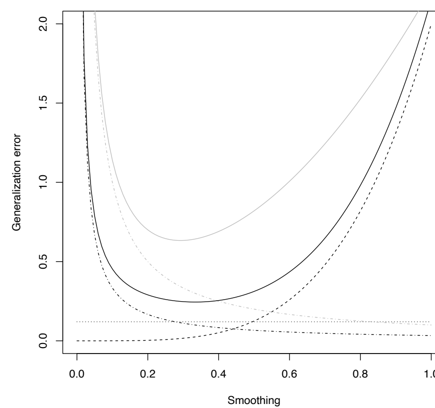

```
curve(2 * x^4, from = 0, to = 1, lty = 2, xlab = "Smoothing", ylab = "Generalization error")
curve(0.12 + x - x, lty = 3, add = TRUE)
curve(1/(10 * x), lty = 4, add = TRUE)
curve(0.12 + 2 * x^4 + 1/(10 * x), add = TRUE)
```

**Figure 4.1** Decomposition of the generalization error of smoothing: the total error (solid) equals process noise (dotted) plus approximation error from smoothing (=squared bias, dashed) and estimation variance (dot-and-dash). The numerical values here are arbitrary, but the functional forms (squared bias $\propto h^4$, variance $\propto n^{-1}h^{-1}$) are representative of kernel regression (Eq. 4.12).

on the real regression curve, on our smoothing method, *and* on 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.

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




```
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) don't change, 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).

## 4.2 Adapting to Unknown Roughness

Figure 4.3, which graphs two functions, $r$ and $s$. Both are "smooth" functions in the 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] 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, $r$ is less smooth than $s$ — it changes much more rapidly, with many reversals of direction. For the same degree of accuracy in the interpolation $r$ needs more, and more closely spaced, training points $x_i$ than does $s$.

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

As remarked in Chapter 1, if we had many measurements at the same $x$, then we could find the expectation value by averaging: the regression function $\mu(x) = \mathbb{E}[Y|X = x]$, so with multiple observations $x_i = x$, the mean of the corresponding $y_i$ would (by the law of large numbers) converge on $\mu(x)$. 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 would always be $\sigma^2$, the noise variance. However, our estimate would be unbiased.

Smoothing methods try to use 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, $\mu(x_i)$ will be close to $\mu(x)$. 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 independent terms together also reduces variance. If smoothing gets rid of more variance than it adds 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 B.1), so

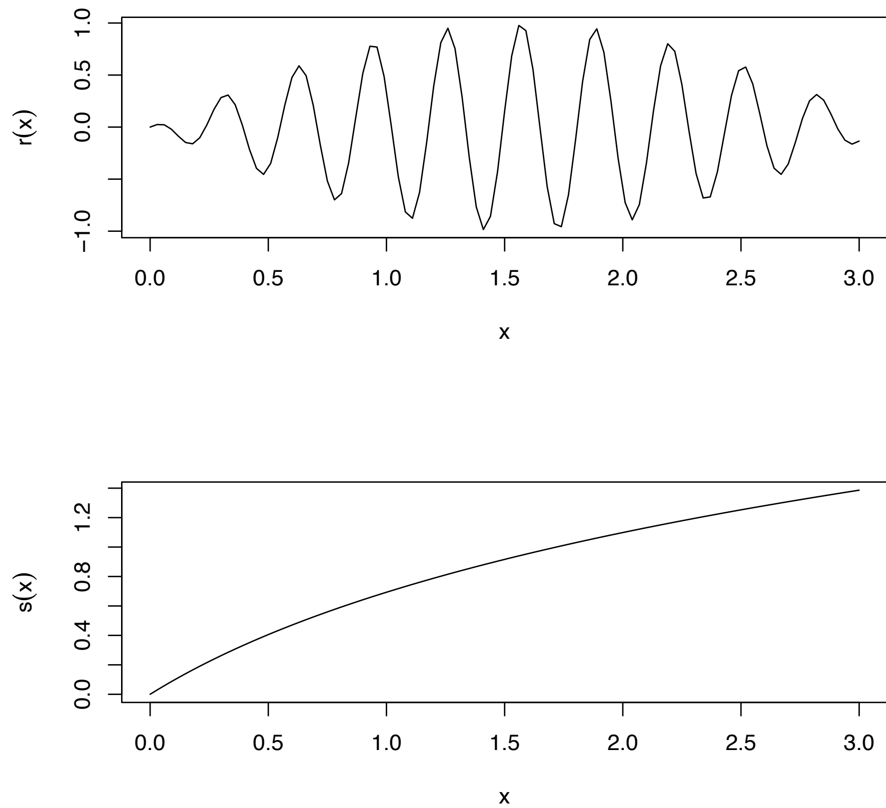
$$\mu(x_i) \approx \mu(x) + (x_i - x)\mu'(x) \tag{4.3}$$

and

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

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

[2] They are "$C^\infty$": continuous, with continuous derivatives to all orders.
[3] See App. B for a refresher on Taylor expansions.




```
par(mfcol = c(2, 1))
true.r <- function(x) {
    sin(x) * cos(20 * x)
}
true.s <- function(x) {
    log(x + 1)
}
curve(true.r(x), from = 0, to = 3, xlab = "x", ylab = expression(r(x)))
curve(true.s(x), from = 0, to = 3, xlab = "x", ylab = expression(s(x)))
par(mfcol = c(1, 1))
```

**Figure 4.3** Two curves for the running example. Above, $r(x) = \sin x \cos 20x$ ; below, $s(x) = \log 1 + x$ (we will not use this information about the exact functional forms).

by just $w_i$.

$$\widehat{\mu}(x) = \sum_{i=1}^{n} y_i w_i \tag{4.5}$$

$$= \sum_{i=1}^{n} (\mu(x) + (x_i - x)\mu'(x) + \epsilon_i) w_i \tag{4.6}$$

$$= \mu(x) + \sum_{i=1}^{n} w_i \epsilon_i + \mu'(x) \sum_{i=1}^{n} w_i (x_i - x) \tag{4.7}$$

$$\widehat{\mu}(x) - \mu(x) = \sum_{i=1}^{n} w_i \epsilon_i + \mu'(x) \sum_{i=1}^{n} w_i (x_i - x) \tag{4.8}$$

$$\mathbb{E}\left[(\widehat{\mu}(x) - \mu(x))^2\right] = \sigma^2 \sum_{i=1}^{n} w_i^2 + \mathbb{E}\left[\left(\mu'(x) \sum_{i=1}^{n} w_i (x_i - x)\right)^2\right] \tag{4.9}$$

(Remember that: $\sum w_i = 1$; $\mathbb{E}[\epsilon_i] = 0$; $\epsilon$ is uncorrelated with everything; and $\mathbb{V}[\epsilon_i] = \sigma^2$.)

The first term on the final right-hand side is an estimation variance, which will tend to shrink as $n$ grows. (If we just did a simple global mean, $w_i = 1/n$ for all $i$, so we'd get $\sigma^2/n$, just like in baby stats.) The second term, an expectation, is bias, which grows as $x_i$ gets further from $x$, and as the magnitudes of the derivatives grow, i.e., this term's growth varies with *how* smooth or wiggly the regression function is. For smoothing to work, $w_i$ had better shrink as $x_i - x$ and $\mu'(x)$ 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 $r(1.6)$ and $s(1.6)$ from the noisy observations. We'll try a simple approach, just averaging all values of $r(x_i) + \epsilon_i$ and $s(x_i) + \eta_i$ for $1.5 < x_i < 1.7$ with equal weights. For $r$, this gives 0.54, while $r(1.6) = 0.83$. For $g$, this gives 0.94, with $s(1.6) = 0.96$. (See figure 4.5.) The same window size creates a much larger bias with the rougher, more rapidly changing $r$ than with the smoother, more slowly changing $s$. 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

$$\mathbb{E}\left[\widehat{\mu}(x) - \mu(x) | X_1 = x_1, \ldots X_n = x_n\right] = h^2 \left[\frac{1}{2}\mu''(x) + \frac{\mu'(x) f'(x)}{f(x)}\right] \sigma_K^2 + o(h^2) \tag{4.10}$$

where $f$ is the density of $x$, and $\sigma_K^2 = \int u^2 K(u) du$, the variance of the probability density corresponding to the kernel[5]. The $\mu''$ term just comes from the second-

[4] The higher derivatives of $\mu$ also matter, since we should really keep 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.

[5] If you are not familiar with the "order" symbols $O$ and $o$, see Appendix A.

order part of the Taylor expansion. To see where the $\mu' f'$ 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 $\widehat{\mu}(x)$, and their distribution will be roughly symmetric around $x$ (at least once $h$ is sufficiently small). So, at mode, $\mathbb{E}[w(X_i, x, h)(X_i - x)\widehat{\mu}(x)] \approx 0$. 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,

$$\mathbb{V}[\widehat{\mu}(x)|X_1 = x_1, \ldots X_n = x_n] = \frac{\sigma^2(x)R(K)}{nhf(x)} + o((nh)^{-1}) \tag{4.11}$$

where $R(K) \equiv \int K^2(u)du$. 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 $\widehat{\mu}(x)$. Each of these has a $y_i$ value, equal to $\mu(x)$ 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 $x$:

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

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

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

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

Of course, this expression for the optimal $h$ involves the unknown derivatives $\mu'(x)$ and $\mu''(x)$, 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

[6] Exercise 4.1 sketches the demonstration for the special case of the uniform ("boxcar") kernel.

[7] You may be wondering why I keep talking about *the* optimal bandwidth, when Eq. 4.14 makes it seem that the bandwidth should vary with $x$. One can 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.

*would* work, if an Oracle were to tell us the derivatives, or (to cut to the chase) the optimal bandwidth $h_{\text{opt}}$. Since most of us do not have access to such oracles, we need to *estimate* $h_{\text{opt}}$. Once we have this estimate, $\widehat{h}$, then we get our 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 $h_{\text{opt}}$, 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 $h_{\text{opt}}$. 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 4 shows how it would work in R, with a one predictor variable, borrowing 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 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 4.7 plots the CV estimate of the (root) mean-squared error versus bandwidth for our two curves. Figure 4.8 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 (which would look like Figure 4.1), so optimizing the CV error is close to optimizing the generalization error.

Notice, by the way, in Figure 4.7, that the rougher curve is more sensitive to the choice of bandwidth, and that the smoother curve always has a lower

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

[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.

[10] The package has methods for automatically selecting bandwidth by cross-validation — see §4.6 below.

```
cv_bws_npreg <- function(x, y, bandwidths = (1:50)/50, nfolds = 10) {
    require(np)
    n <- length(x)
    stopifnot(n > 1, length(y) == n)
    stopifnot(length(bandwidths) > 1)
    stopifnot(nfolds > 0, nfolds == trunc(nfolds))

    fold_MSEs <- matrix(0, nrow = nfolds, ncol = length(bandwidths))
    colnames(fold_MSEs) = bandwidths

    case.folds <- sample(rep(1:nfolds, length.out = n))
    for (fold in 1:nfolds) {
        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 4: *Cross-validation for univariate kernel regression. 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 is* just *demo of how cross-validation for bandwidth selection works in principle;* don't *use it blindly on data, or in assignments. (That goes double for the vector of default bandwidths.)*

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 14, 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

[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.

symbols, which also absorb the little-$o$ remainder terms:

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

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:

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

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 proportional to the product of sample size and bandwidth. If we kept $h$ constant and just let $n \to \infty$, 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 $h_n \to 0$ as $n \to \infty$, but $nh_n \to \infty$. Then, by Eq. 4.16, the risk (generalization error) of kernel smoothing is approaching that of the ideal predictor.

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 slow? We can compare it to what would happen with a parametric model, say with parameter $\theta$. (For linear regression, $\theta$ would be the vector of slopes and the intercept.) The optimal value of the parameter, $\theta_0$, minimizes the mean-squared error. At $\theta_0$, 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 $\theta_0$ is unknown and must be estimated, one typically has $\widehat{\theta} - \theta_0 = O(1/\sqrt{n})$. Because the error is minimized at $\theta_0$, the first derivatives of $MSE$ at $\theta_0$ are 0. Doing a second-order Taylor expansion of the parametric model contributes an error $O((\widehat{\theta} - \theta_0)^2)$, so altogether

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

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

[12] When the model is wrong, the optimal parameter value $\theta_0$ 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

This doesn't change much if we use cross-validation. Writing $\widehat{h_{CV}}$ for the bandwidth picked by cross-validation, it turns out (Simonoff, 1996, ch. 5) that

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

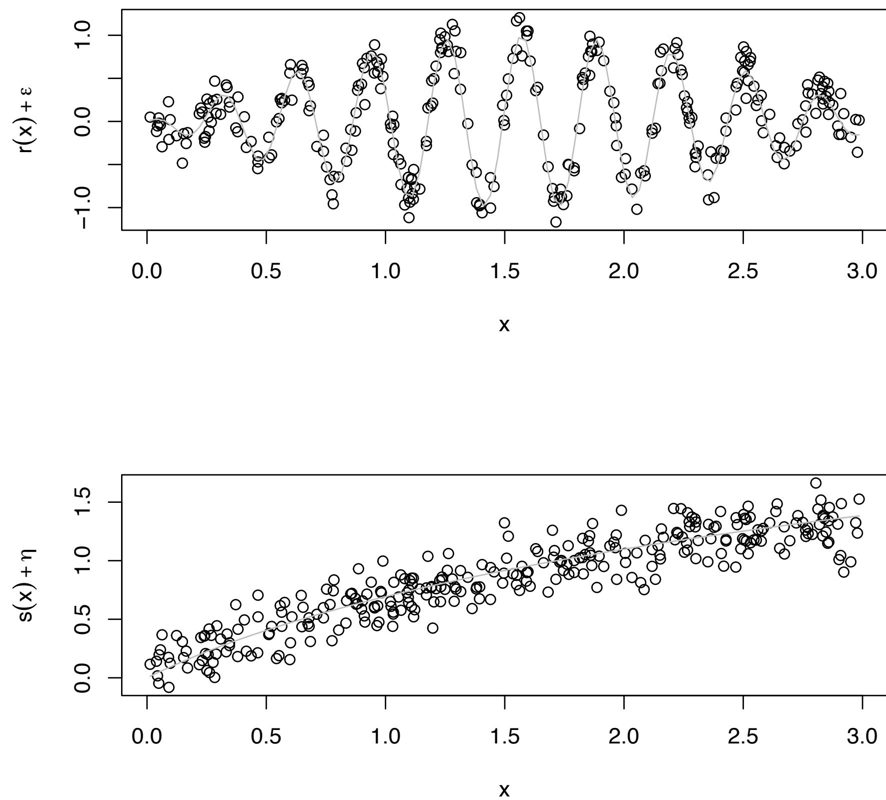
Given this, one concludes (Exercise 4.2) that the MSE of using $\widehat{h_{CV}}$ is also $O(n^{-4/5})$.

### 4.2.3 Summary on Kernel Smoothing in 1D

Suppose that $X$ and $Y$ are both one-dimensional, and the true regression function $\mu(x) = \mathbb{E}[Y|X = x]$ 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 $O((1/nh)^{-1})$. 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 $O(n^{-4/5})$.

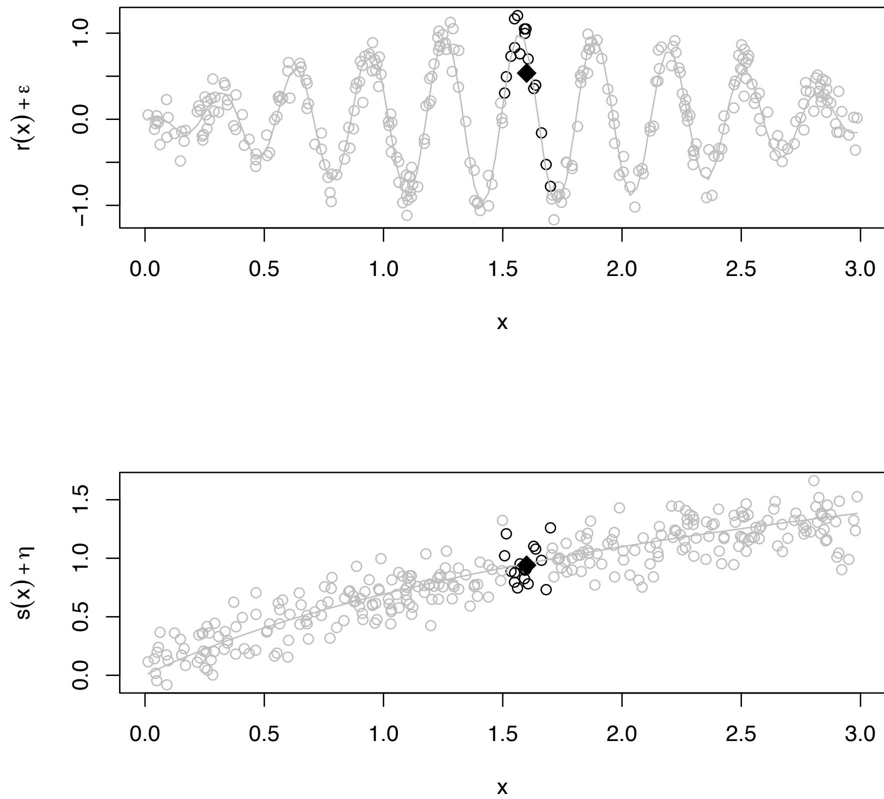
following sense (Wasserman, 2006). Any curve-fitting method which can learn arbitrary smooth regression functions will have some curves where it cannot converge any faster than $O(n^{-4/5})$. (In the jargon, that is the **minimax rate**.) Methods which converge faster than this for some kinds of curves have to converge more slowly for others. So this is the best rate we can hope for on truly unknown curves.

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




```
x = runif(300, 0, 3)
yr = true.r(x) + rnorm(length(x), 0, 0.15)
ys = true.s(x) + rnorm(length(x), 0, 0.15)
par(mfcol = c(2, 1))
plot(x, yr, xlab = "x", ylab = expression(r(x) + epsilon))
curve(true.r(x), col = "grey", add = TRUE)
plot(x, ys, xlab = "x", ylab = expression(s(x) + eta))
curve(true.s(x), col = "grey", add = TRUE)
```

**Figure 4.4** The curves of Fig. 4.3 (in grey), plus IID Gaussian noise with mean 0 and standard deviation 0.15. The two curves are sampled at the same $x$ values, but with different noise realizations.




```
par(mfcol = c(2, 1))
x.focus <- 1.6
x.lo <- x.focus - 0.1
x.hi <- x.focus + 0.1
colors = ifelse((x < x.hi) & (x > x.lo), "black", "grey")
plot(x, yr, xlab = "x", ylab = expression(r(x) + epsilon), col = colors)
curve(true.r(x), col = "grey", add = TRUE)
points(x.focus, mean(yr[(x < x.hi) & (x > x.lo)]), pch = 18, cex = 2)
plot(x, ys, xlab = "x", ylab = expression(s(x) + eta), col = colors)
curve(true.s(x), col = "grey", add = TRUE)
points(x.focus, mean(ys[(x < x.hi) & (x > x.lo)]), pch = 18, cex = 2)
par(mfcol = c(1, 1))
```

**Figure 4.5** Relationship between smoothing and function roughness. In both panels we estimate the value of the regression function at $x = 1.6$ by averaging observations where $1.5 < x_i < 1.7$ (black points, others are "ghosted" in grey). The location of the average in shown by the large black diamond. This works poorly for the rough function $r$ in the upper panel (the bias is large), but much better for the smoother function in the lower panel (the bias is small).

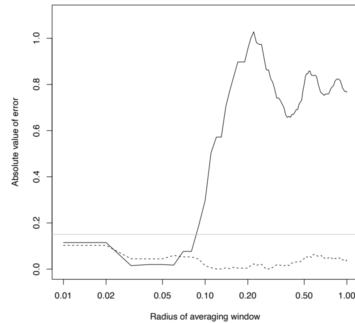
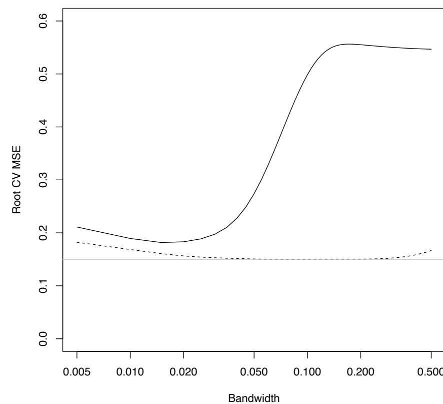
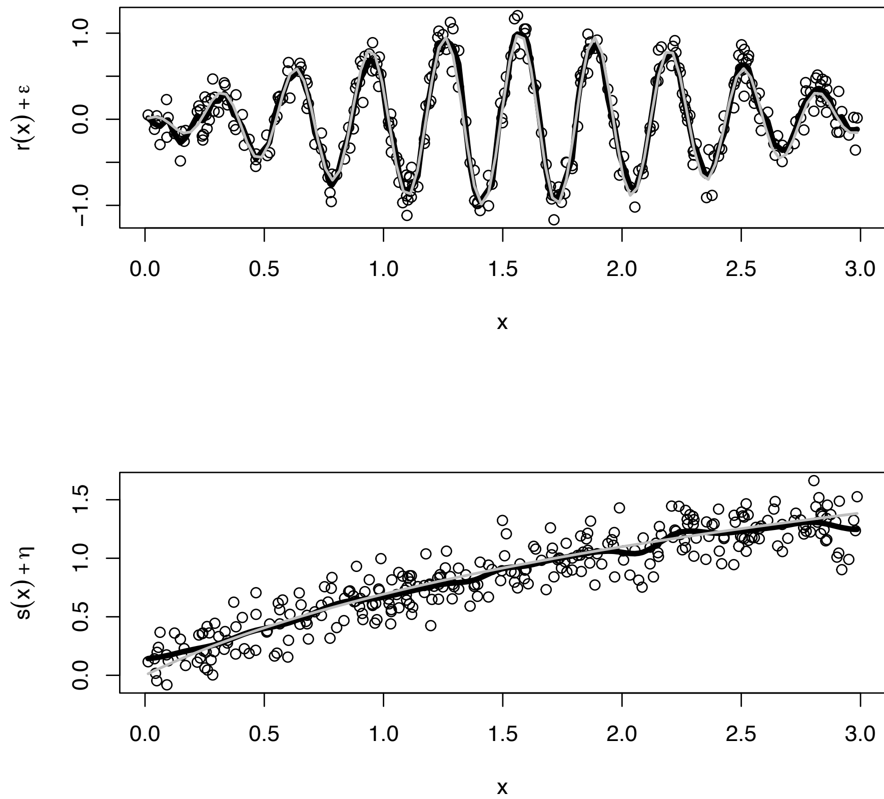



**Figure 4.6** Error of estimating $r(1.6)$ (solid line) and $s(1.6)$ (dashed) from averaging observed values at $1.6 - h < x < 1.6 + h$, for different radii $h$. The grey is $\sigma$, the standard deviation of the noise — how can the estimation error be smaller than that?




```
rbws <- cv_bws_npreg(x, yr, bandwidths = (1:100)/200)
sbws <- cv_bws_npreg(x, ys, bandwidths = (1:100)/200)
plot(1:100/200, sqrt(rbws$CV_MSEs), xlab = "Bandwidth", ylab = "Root CV MSE", type = "l",
    ylim = c(0, 0.6), log = "x")
lines(1:100/200, sqrt(sbws$CV_MSEs), lty = "dashed")
abline(h = 0.15, col = "grey")
```

**Figure 4.7** Cross-validated estimate of the (root) mean-squard error as a function of the bandwidth (solid curve, $r$ data; dashed, $s$ data; grey line, true noise $\sigma$). 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. CV selects bandwidths of 0.015 for $r$ and 0.075 for $s$.




```
x.ord = order(x)
par(mfcol = c(2, 1))
plot(x, yr, xlab = "x", ylab = expression(r(x) + epsilon))
rhat <- npreg(bws = rbws$best.bw, txdat = x, tydat = yr)
lines(x[x.ord], fitted(rhat)[x.ord], lwd = 4)
curve(true.r(x), col = "grey", add = TRUE, lwd = 2)
plot(x, ys, xlab = "x", ylab = expression(s(x) + eta))
shat <- npreg(bws = sbws$best.bw, txdat = x, tydat = ys)
lines(x[x.ord], fitted(shat)[x.ord], lwd = 4)
curve(true.s(x), col = "grey", add = TRUE, lwd = 2)
par(mfcol = c(1, 1))
```

**Figure 4.8** Data from the running examples (circles), true regression functions (grey) and kernel estimates of regression functions with CV-selected bandwidths (black). R NOTES: The `x` values aren't sorted, so we need to put them in order before drawing lines connecting the fitted values; then we need to put the fitted values in the same order. Alternately, we could have used `predict` on the sorted values, as in §4.3.

## 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

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

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 $\infty$) 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 "positive definite"[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:

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

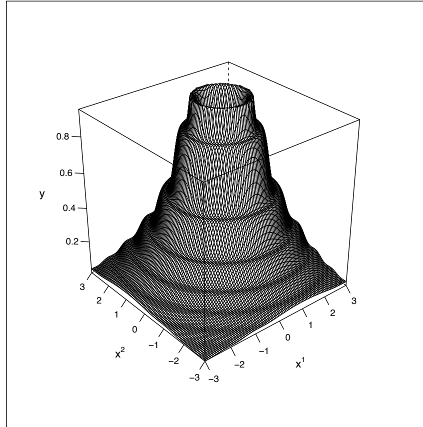
Now we just need to pick a bandwidth for each kernel, which in general should not be equal — say $\vec{h} = (h_1, h_2, \ldots h_d)$. Instead of having a one-dimensional error curve, as in Figure 4.1 or 4.2, we will have a $d$-dimensional error surface, but we can still use cross-validation to find the 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.

(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 14, 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 4.9 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 4.11 shows the estimate we get with a product of Gaussian kernels and only 1000 noisy data points. It's

[15] Remember that for a matrix $\mathbf{v}$ to be "positive definite", it must be the case that for any vector $\vec{a} \neq \vec{0}$, $\vec{a} \cdot \mathbf{v}\vec{a} > 0$. 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 with non-trivial correlation across the variables, but this is very rare in my experience.

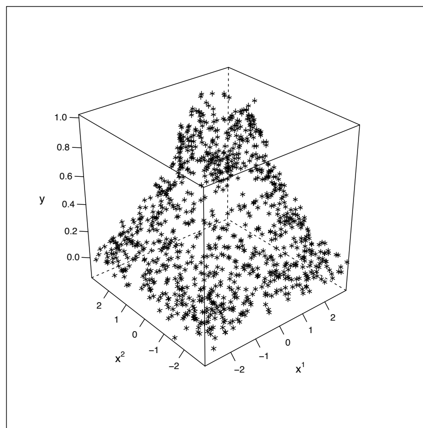



```
x1.points <- seq(-3, 3, length.out = 100)
x2.points <- x1.points
x12grid <- expand.grid(x1 = x1.points, x2 = x2.points)
y <- matrix(0, nrow = 100, ncol = 100)
y <- outer(x1.points, x2.points, f)
library(lattice)
wireframe(y ~ x12grid$x1 * x12grid$x2, scales = list(arrows = FALSE), xlab = expression(x^1),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

**Figure 4.9** 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`.

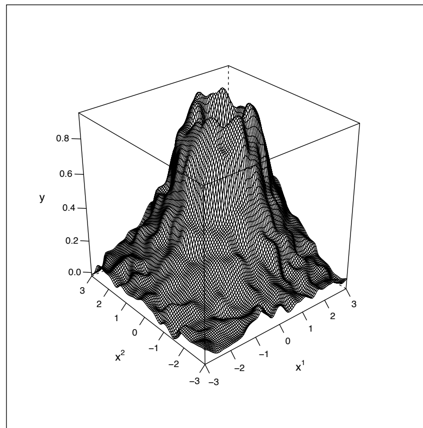
not perfect, of course (in particular the estimated contours aren't as perfectly smooth and round as the true ones), but the important thing is that we got this without having to know, and describe in Cartesian coordinates, the type of shape we were looking for. Kernel smoothing *discovered* the right 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 8) 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.




```
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(y ~ x1 * x2, data = noise, col = "black", scales = list(arrows = FALSE), xlab = expression(x^1
    ylab = expression(x^2), zlab = "y")
```

**Figure 4.10** 1000 points sampled from the surface in Figure 4.9, 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 4.11** Gaussian kernel regression of the points in Figure 4.10. Notice that the estimated function will make predictions at arbitrary points, not just the places where there was training data.

## 4.4 Interpreting Smoothers: Plots

In a linear regression without interactions, it is fairly easy to interpret the coefficients. The expected response changes by $\beta_i$ for a one-unit change in the $i^{\text{th}}$ 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).

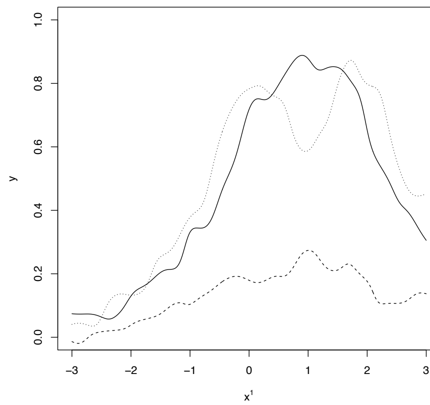
As soon as we introduce interactions, all this becomes harder, even for parametric regression. If there is an interaction between two components of the input, say $x^1$ and $x^2$, then we can't talk about the change in the expected response for a one-unit change in $x^1$ without saying what $x^2$ is. We might *average* over $x^2$ values, and in §4.5 below 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 $\beta_1$" is just false, no matter what number we fill in for $\beta_1$. Likewise for derivatives; we'll come back to them next time as well.

What about pictures? With only two input variables, we can make wireframe plots like Figure 4.11, or contour or level plots, which will show the predictions for different combinations of the two variables. But what if we want to look at one variable at a time, or there are more than two input variables?

A reasonable way to produce *a* curve for each input variable is to set all the others to some "typical" value, like their means or medians, and to then plot the predicted response as a function of the one remaining variable of interest (Figure 4.12). Of course, when there are interactions, changing the values of the other inputs will change the response to the input of interest, so it's a good idea to produce a couple of curves, possibly super-imposed (Figure 4.12 again).

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/or how much big a lie you are prepared to tell in order to keep your story simple.




```
new.frame <- data.frame(x1 = seq(-3, 3, length.out = 300), x2 = median(x2.noise))
plot(new.frame$x1, predict(noise.np, newdata = new.frame), type = "l", xlab = expression(x^1),
    ylab = "y", ylim = c(0, 1))
new.frame$x2 <- quantile(x2.noise, 0.25)
lines(new.frame$x1, predict(noise.np, newdata = new.frame), lty = 2)
new.frame$x2 <- quantile(x2.noise, 0.75)
lines(new.frame$x1, predict(noise.np, newdata = new.frame), lty = 3)
```

**Figure 4.12** 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 $25^{\text{th}}$ percentile (dashed) and its $75^{\text{th}}$ percentile (dotted). Note that the changing shape of the partial response curve indicates an interaction between the two inputs. Also, note that the model can make predictions at arbitrary coordinates, whether or not there were any training points there.

## 4.5 Average Predictive Comparisons

Suppose we have a linear regression model

$$Y = \beta_1 X_1 + \beta_2 X_2 + \epsilon \quad (4.24)$$

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 $\beta_1$:

$$[\beta_1(X_1 + 1) + \beta_2 X_2] - [\beta_1 X_1 + \beta_2 X_2] = \beta_1 \quad (4.25)$$

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

$$Y = \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon \quad (4.26)$$

Now the effect of increasing $X_1$ by 1 is

$$[\beta_1(X_1+1)+\beta_2 X_2+\beta_3(X_1+1)X_2]-[\beta_1 X_1+\beta_2 X_2+\beta_3 X_1 X_2] = \beta_1+\beta_3 X_2 \quad (4.27)$$

The right answer to "how much does the response change when $X_1$ is increased by one unit?" depends on the value of $X_2$; it's certainly not just "$\beta_1$".

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

$$Y = \frac{e^{\beta X}}{1 + e^{\beta X}} + \epsilon \quad (4.28)$$

which looks like Figure 4.13, setting $\beta = 7$ (for luck). Moving $x$ from $-4$ to $-3$ increases the response by $7.57 \times 10^{-10}$, 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 11.)

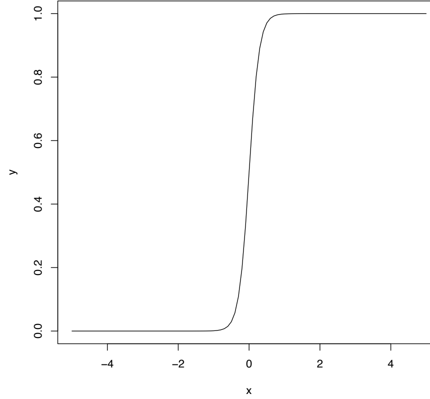
More generally, let's say we are regressing $Y$ on a vector $\vec{X}$, and want to assess the impact of one component of the input on $Y$. To keep the use of subscripts and superscripts to a minimum, we'll write $\vec{X} = (U, \vec{V})$, where $U$ is the coordinate we're really interested in. (It doesn't have to come first, of course.) We would like to know how much the prediction changes as we change $u$,

$$\mathbb{E}\left[Y|\vec{X} = (u^{(2)}, \vec{v})\right] - \mathbb{E}\left[Y|\vec{X} = (u^{(1)}, \vec{v})\right] \quad (4.29)$$

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

$$\frac{\mathbb{E}\left[Y|\vec{X} = (u^{(2)}, \vec{v})\right] - \mathbb{E}\left[Y|\vec{X} = (u^{(1)}, \vec{v})\right]}{u^{(2)} - u^{(1)}} \quad (4.30)$$

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




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

**Figure 4.13** The function of Eq. 4.28, with $\beta = 7$.

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 choose our starting point, ending point and context, and just plug in to Eq. 4.29 or Eq. 4.30. (Or problem 9 in problem set 11.) 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** for $u$

$$\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\widehat{\mu}(u_j, \vec{v}_i) - \widehat{\mu}(u_i, \vec{v}_i)\right)\mathrm{sign}(u_j - u_i)}{\sum_{i=1}^{n}\sum_{j=1}^{n}(u_j - u_i)\mathrm{sign}(u_j - u_i)} \tag{4.31}$$

where $i$ and $j$ run over data points, $\widehat{\mu}$ is our estimated regression function, and the sign function is defined by $\mathrm{sign}(x) = +1$ 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 is a reasonable summary of how rapidly we should expect the response to vary as $u$ changes slightly. But we need to remember that once the model is nonlinear or has interactions, it's just not possible to boil down the whole predictive relationship between $u$ and $y$ into one number. In particular, the value of Eq. 4.31 is going to depend on the distribution of $u$ (and possibly of $v$), even when the regression function is unchanged. (See Exercise 4.3.)

[17] Actually, they propose something a bit more complicated, which takes into account the uncertainty in our estimate of the regression function, via bootstrapping (Chapter 6).

```
make.demo.df <- function(n) {
    demo.func <- function(x, z, w) {
        20 * x^2 + ifelse(w == "A", z, 10 * exp(z)/(1 + exp(z)))
    }
    x <- runif(n, -1, 1)
    z <- rnorm(n, 0, 10)
    w <- sample(c("A", "B"), size = n, replace = TRUE)
    y <- demo.func(x, z, w) + rnorm(n, 0, 0.05)
    return(data.frame(x = x, y = y, z = z, w = w))
}

demo.df <- make.demo.df(100)
```

Code Example 5: *Generating data from Eq. 4.32*

## 4.6 Computational Advice: `npreg`

The homework will call for you to do nonparametric regression with the `np` package — which we've already looked at a little. It's a powerful bit of software, but it can take a bit of getting used to. This section is not a substitute for reading Hayfield and Racine (2008), but should get you started.

We'll look at a synthetic-data example with four variables: a quantitative response $Y$, two quantitative predictors $X$ and $Z$, and a categorical predictor $W$, which can be either "A" or "B". The true model is

$$Y = \epsilon + 20X^2 + \begin{cases} Z & \text{if } W = A \\ 10e^Z/(1+e^Z) & \text{if } W = B \end{cases} \tag{4.32}$$

with $\epsilon \sim \mathcal{N}(0, 0.05)$. Code Example 5 generates some data from this model for us.

The basic function for fitting a kernel regression in `np` is `npreg` — conceptually, it's the equivalent of `lm`. Like `lm`, it takes a `formula` argument, which specifies the model, and a `data` argument, which is a data frame containing the variables included in the formula. The basic idea is to do something like this:

```
demo.np1 <- npreg(y ~ x + z, data = demo.df)
```

The variables on the right-hand side of the formula are the predictors; we use `+` to separate them. Kernel regression will automatically include interactions between all variables, so there is no special notation for interactions. Similarly, there is no point in either including or excluding intercepts. If we wanted to transform either a predictor variable or the response, as in `lm`, we can do so. Run like this, `npreg` will try to determine the best bandwidths for the predictor variables, based on a sophisticated combination of cross-validation and optimization.

Let's look at the output of `npreg`:

```
summary(demo.np1)
##
## Regression Data: 100 training points, in 2 variable(s)
##                       x        z
## Bandwidth(s): 0.06227118 4.744557
##
```

```
## Kernel Regression Estimator: Local-Constant
## Bandwidth Type: Fixed
## Residual standard error: 2.584642
## R-squared: 0.9378975
##
## Continuous Kernel Type: Second-Order Gaussian
## No. Continuous Explanatory Vars.: 2
```

The main things here are the bandwidths. We also see the root mean squared error on the training data. Note that this is the in-sample root MSE; if we wanted the in-sample MSE, we could do

```
demo.np1$MSE
## [1] 6.680373
```

(You can check that this is the square of the residual standard error above.) If we want the cross-validated MSE used to pick the bandwidths, that's

```
demo.np1$bws$fval
## [1] 25.52361
```

The `fitted` and `residuals` functions work on these objects just like they do in `lm` objects, while the `coefficients` and `confint` functions do not. (Why?)

The `predict` function also works like it does for `lm`, expecting a data frame containing columns whose names match those in the formula used to fit the model:

```
predict(demo.np1, newdata = data.frame(x = -1, z = 5))
## [1] 26.04758
```

With two predictor variables, there is a nice three-dimensional default plot (Figure 4.14).

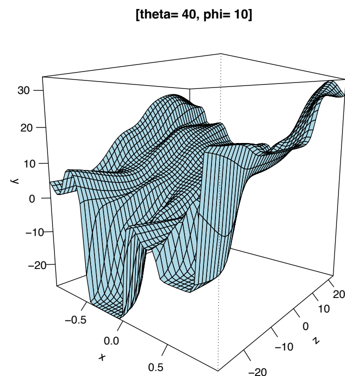
Kernel functions can also be defined for categorical and ordered variables. These can be included in the formula by wrapping the variable in `factor()` or `ordered()`, respectively:

```
demo.np3 <- npreg(y ~ x + z + factor(w), data = demo.df)
```

Again, there's no point, or need, to indicate interactions. Including the extra variable, not surprisingly, improves the cross-validated MSE:

```
demo.np3$bws$fval
## [1] 13.94945
```

With three or more predictor variables, we'd need a four-dimensional plot, which is hard. Instead, the default is to plot what happens as we sweep one variable with the others held fixed (by default, at their medians; see `help(npplot)` for changing that), as in Figure 4.15. We get something parabola-ish as we sweep $X$ (which is right), and something near a step function as we sweep $Z$ (which is right when $W = B$), so we're not doing badly for estimating a fairly complicated function of three variables with only 100 samples. We could also try fixing $W$ at one value or another and making a perspective plot — Figure 4.16.




```
plot(demo.np1, theta = 40, view = "fixed")
```

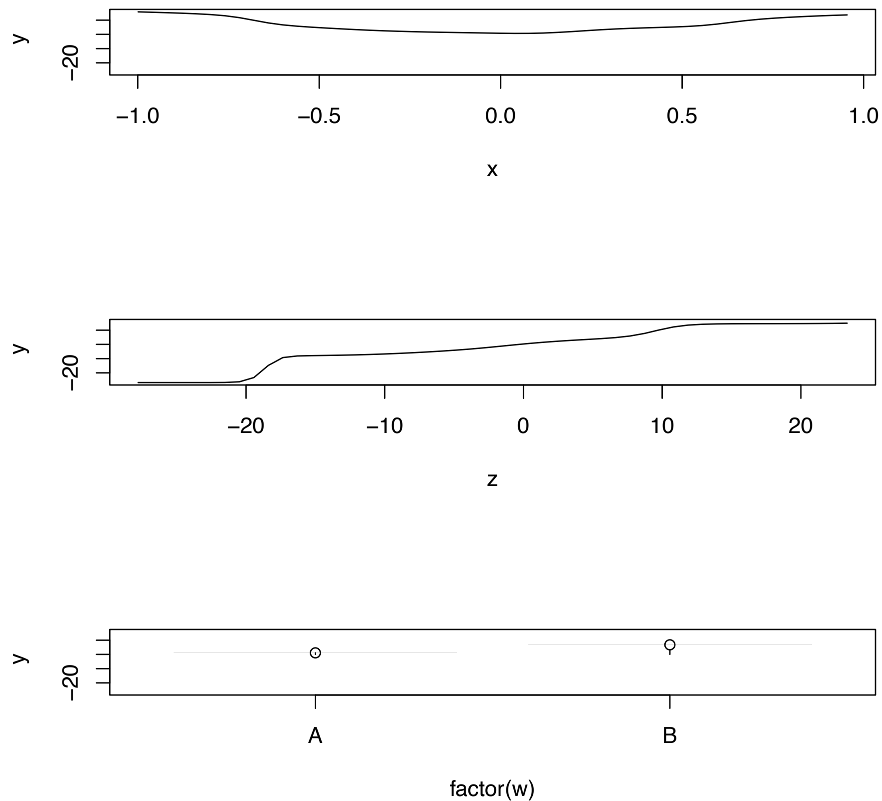
**Figure 4.14** Plot of the kernel regression with just two predictor variables. (See `help(npplot)` for plotting options.

The default optimization of bandwidths is *extremely* aggressive. It keeps adjusting the bandwidths until the changes in the cross-validated MSE are very small, or the changes in the bandwidths themselves are very small. The "tolerances" for what count as "very small" are controlled by arguments to `npreg` called `tol` (for the bandwidths) and `ftol` (for the MSE), which default to about $10^{-8}$ and $10^{-7}$, respectively. With a lot of data, or a lot of variables, this gets *extremely* slow. One can often make `npreg` run much faster, with no real loss of accuracy, by adjusting these options. A decent rule of thumb is to start with `tol` and `ftol` both at 0.01. One can use the bandwidth found by this initial coarse search to start a more refined one, as follows:

```
bigdemo.df <- make.demo.df(1000)
system.time(demo.np4 <- npreg(y ~ x + z + factor(w), data = bigdemo.df, tol = 0.01,
    ftol = 0.01))
##    user  system elapsed
##  31.314   0.330  32.160
```

This tells us how much time it took R to run `npreg`, dividing that between time spent exclusively on our job and on background system tasks. The result of the run is stored in `demo.np4`:

```
demo.np4$bws
##
## Regression Data (1000 observations, 3 variable(s)):
##
##                        x        z    factor(w)
```




```
plot(demo.np3)
```

**Figure 4.15** Predictions of `demo.np3` as each variable is swept over its range, with the others held at their medians.

```
## Bandwidth(s): 0.06101409 2.441987 9.649056e-08
##
## Regression Type: Local-Constant
## Bandwidth Selection Method: Least Squares Cross-Validation
## Formula: y ~ x + z + factor(w)
## Bandwidth Type: Fixed
## Objective Function Value: 1.517143 (achieved on multistart 1)
##
## Continuous Kernel Type: Second-Order Gaussian
```

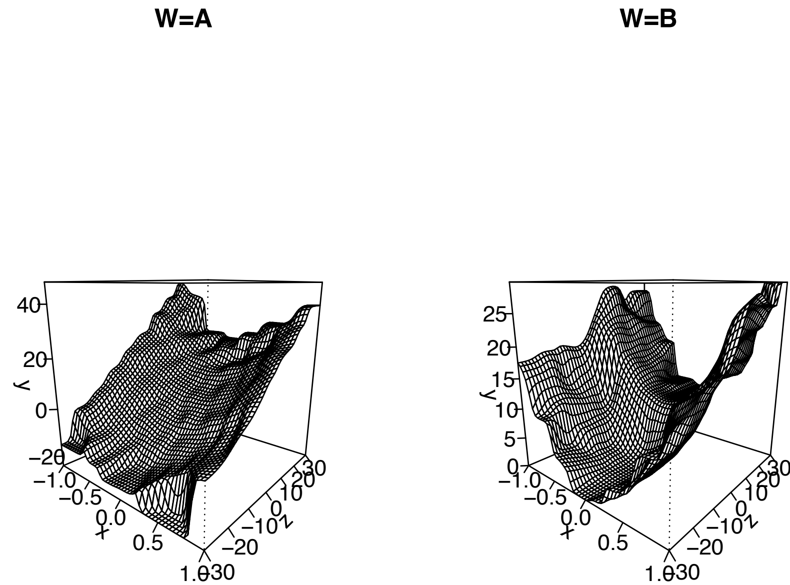
```
## No. Continuous Explanatory Vars.: 2
##
## Unordered Categorical Kernel Type: Aitchison and Aitken
## No. Unordered Categorical Explanatory Vars.: 1
```

The bandwidths have all shrunk (as they should), and the cross-validated MSE is also much smaller (1.5 versus 14 before). Figure 4.16 shows the estimated regression surfaces for both values of the categorical variable.

The package also contains a function, `npregbw`, which takes a formula and a data frame, and just optimizes the bandwidth. This is called automatically by `npreg`, and many of the relevant options are documented in its help page. One can also use the output of `npregbw` as an argument to `npreg`, in place of a formula.

As a final piece of computational advice, you will notice when you run these commands yourself that the bandwidth-selection functions by default print out lots of progress-report messages. This can be annoying, especially if you are embedding the computation in a document, and so can be suppressed by setting a global option at the start of your code:

```
options(np.messages = FALSE)
```




```
x.seq <- seq(from = -1, to = 1, length.out = 50)
z.seq <- seq(from = -30, to = 30, length.out = 50)
grid.A <- expand.grid(x = x.seq, z = z.seq, w = "A")
grid.B <- expand.grid(x = x.seq, z = z.seq, w = "B")
yhat.A <- predict(demo.np4, newdata = grid.A)
yhat.B <- predict(demo.np4, newdata = grid.B)
par(mfrow = c(1, 2))
persp(x = x.seq, y = z.seq, z = matrix(yhat.A, nrow = 50), theta = 40, main = "W=A",
    xlab = "x", ylab = "z", zlab = "y", ticktype = "detailed")
persp(x = x.seq, y = z.seq, z = matrix(yhat.B, nrow = 50), theta = 40, main = "W=B",
    xlab = "x", ylab = "z", zlab = "y", ticktype = "detailed")
```

**Figure 4.16** The regression surfaces learned for the demo function at the two different values of the categorical variable. Note that holding $z$ fixed, we always see a parabolic shape as we move along $x$ (as we should), while whether we see a line or something close to a step function at constant $x$ depends on $w$, as it should.

## 4.7 Further Reading

Simonoff (1996) is a good practical introduction to kernel smoothing and related methods. Wasserman (2006) provides more theory. Li and Racine (2007) is a detailed treatment of nonparametric methods for econometric problems, overwhelmingly focused on kernel regression and kernel density estimation (which we'll get to in Chapter 14); Racine (2008) summarizes.

While kernels are a nice, natural method of non-parametric smoothing, they are not the only one. We saw nearest-neighbors in §1.5.1, and will encounter splines (continuous piecewise-polynomial models) in Chapter 7 and trees (piecewise-constant functions, with cleverly chosen pieces) in Chapter 13; local linear models (§10.5) combine kernels and linear models. There are many, many more options.

### Historical Notes

Kernel regression was introduced, independently, by Nadaraya (1964) and Watson (1964); both were inspired by kernel density estimation.

In the mid-2010s, kernel smoothing was re-invented by computer scientists working on large language models, under the curious name of "attention". This turned out to be a key technical step in creating language models like GPT (Vaswani *et al.*, 2017). The first people to realize that "attention", in this sense, was a kind of kernel smoothing seem to have been Tsai *et al.* (2019).

## Exercises

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

$$\mathbb{E}\left[\widehat{\mu}(0)\right] = \mathbb{E}\left[\mu(X) \middle| X \in \left(-\frac{h}{2}, \frac{h}{2}\right)\right] \tag{4.33}$$

$$= \mu(0) + \mu'(0)\mathbb{E}\left[X \middle| X \in \left(-\frac{h}{2}, \frac{h}{2}\right)\right] \tag{4.34}$$

$$+ \frac{\mu''(0)}{2}\mathbb{E}\left[X^2 \middle| X \in \left(-\frac{h}{2}, \frac{h}{2}\right)\right] + o(h^2)$$

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

4.2 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 $O(n^{-4/5})$.

4.3 Generate 1000 data points where $X$ is uniformly distributed between $-4$ and 4, and $Y = e^{7x}/(1 + e^{7x}) + \epsilon$, with $\epsilon$ Gaussian and with variance 0.01. Use non-parametric regression to estimate $\widehat{\mu}(x)$, 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?