Using Nonparametric Smoothing in Regression

36-402, Data Analysis

20 January 2011

Contents

We are still talking about using smoothing methods in regression.

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:

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

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?

1 How Much Should We Smooth?

When we smooth very little ($h \to 0$), then we can match very small or sharp aspects of the true regression function, if there are such. Smoothing less leads to less bias. At the same time, each of our predictions is going to be an average

over (in effect) just a few observations, so it becomes noisier. So smoothing less increases the variance of our estimate. Since

$$(\text{totalerror}) = (\text{noise}) + (\text{bias})^2 + (\text{variance})$$

(see lecture 1, section 4.1), if we plot the different components of error as a function of h, we typically get something that looks like Figure 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 (as explained last time) cross-validation comes in.

You should note, at this point, that the optimal amount of smoothing depends on (1) the real regression curve, (2) our smoothing method, and (3) 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 1 to Figure 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.

2 Adapting to Unknown Roughness

Consider Figure 3, which graphs two functions, f and g. Both are "smooth" functions in the qualitative, mathematical sense (C^∞: they're not only continuous, their derivatives exist and are continuous to all orders). We could Taylor expand both functions to approximate their values anywhere, just from knowing enough derivatives at one point x_0.[2] 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. Can we now recover the curves?

If we had multiple measurements at the same x, then we could recover the expectation value by averaging: since the regression curve $r(x) = \mathbf{E}[Y|X = x]$,

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

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

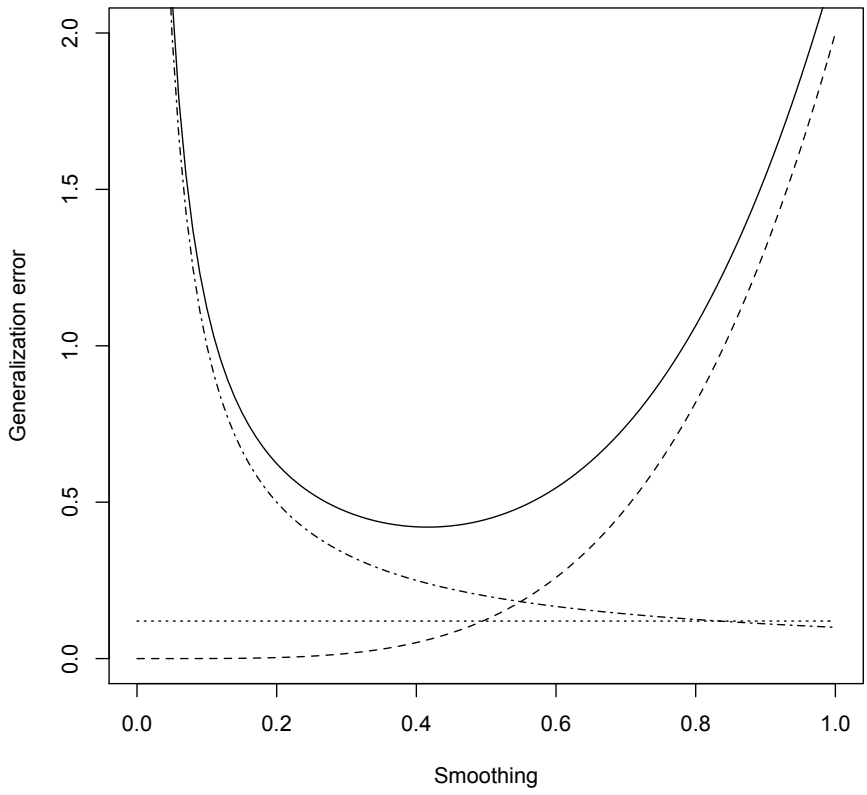


Figure 1: Over-all generalization error for different amounts of smoothing (solid curve) decomposed into intrinsic noise (dotted line), approximation error introduced by smoothing (=squared bias; dashed curve), and estimation variance (dot-and-dash curve). The numerical values here are arbitrary, but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

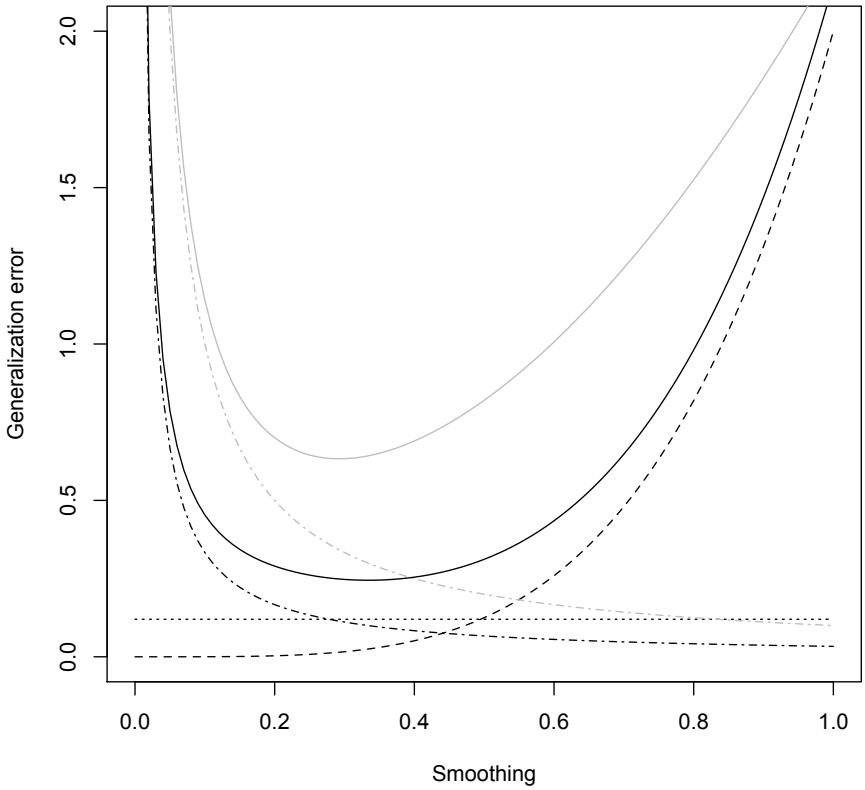
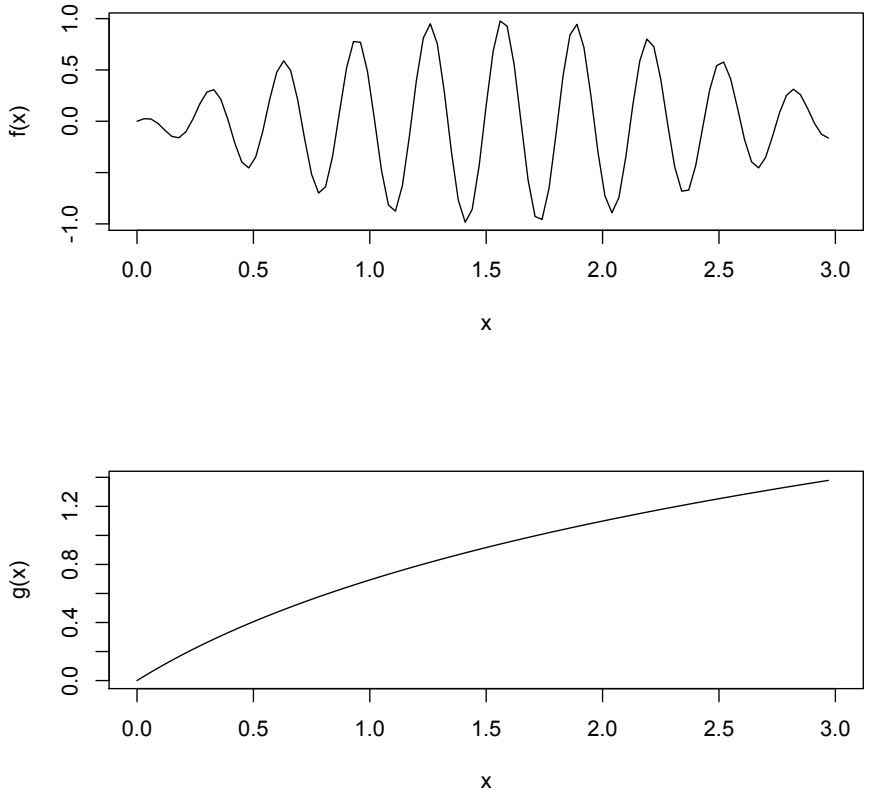
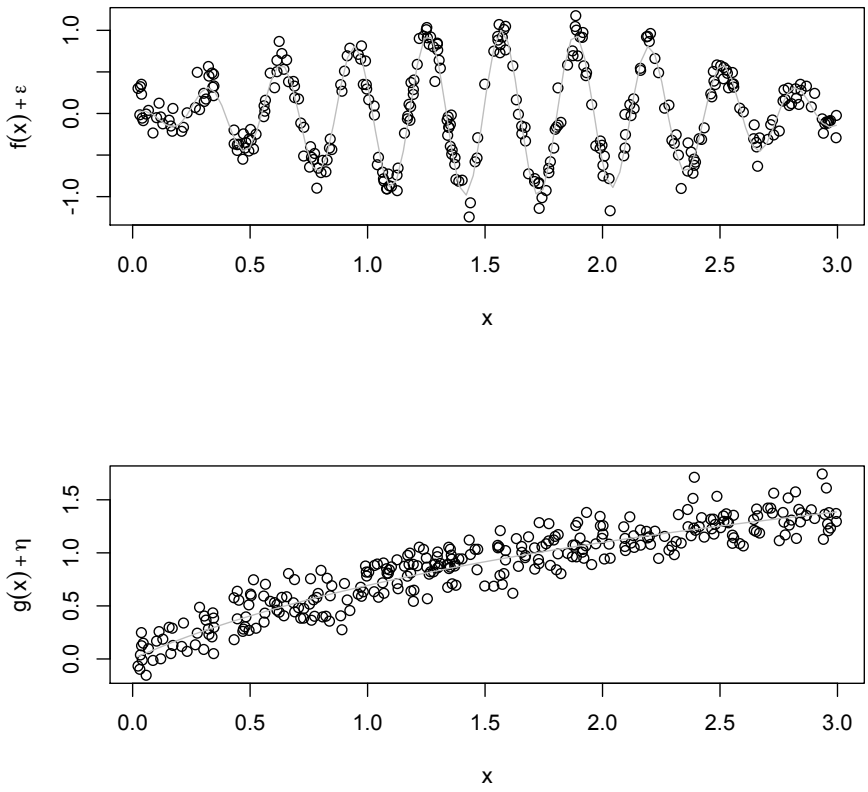


Figure 2: Consequences of adding more data to the components of error: noise (dotted) and bias (dashed) are unchanged, but the new variance curve (dotted and dashed, black) is to the left of the old (greyed), so the new over-all error curve (solid black) is lower, and has its minimum at a smaller amount 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 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: 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.

if we had many observations at the same x_i, 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 will be unbiased.

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, so

$$r(x_i) \approx r(x) + (x_i - x)r'(x) \tag{2}$$

and

$$y_i \approx r(x) + (x_i - x)r'(x) + \epsilon_i \tag{3}$$

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

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

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

The first term on the final right-hand side is 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, on the other hand, is bias, which grows with how far the x_i are from x, and the magnitude of the derivative, 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.[3] Finally, all else being equal, w_i should also shrink with

[3]The higher derivatives of r also matter, since we should really be keeping more than just the first term in the Taylor expansion, but you get the idea.

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 $g(x_i) + \eta_i$ 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 5). 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 we look at the expression for the mean-squared error of the smoother, we can see that it's quadratic in the weights w_i. However, once we pick the smoother and take our data, the weights w_i are all functions of h, the control setting which determines the degree of smoothing. So in principle there will be an optimal choice of h. We can find this through calculus — take the derivative of the MSE with respect to h (via the chain rule) and set it equal to zero — but the expression for the optimal h involves the derivative $r'(x)$ of the regression function. Of course, 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.

One way of expressing this is to talk about how well a smoothing procedure *would* work, if an Oracle were to tell us the derivative, 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.

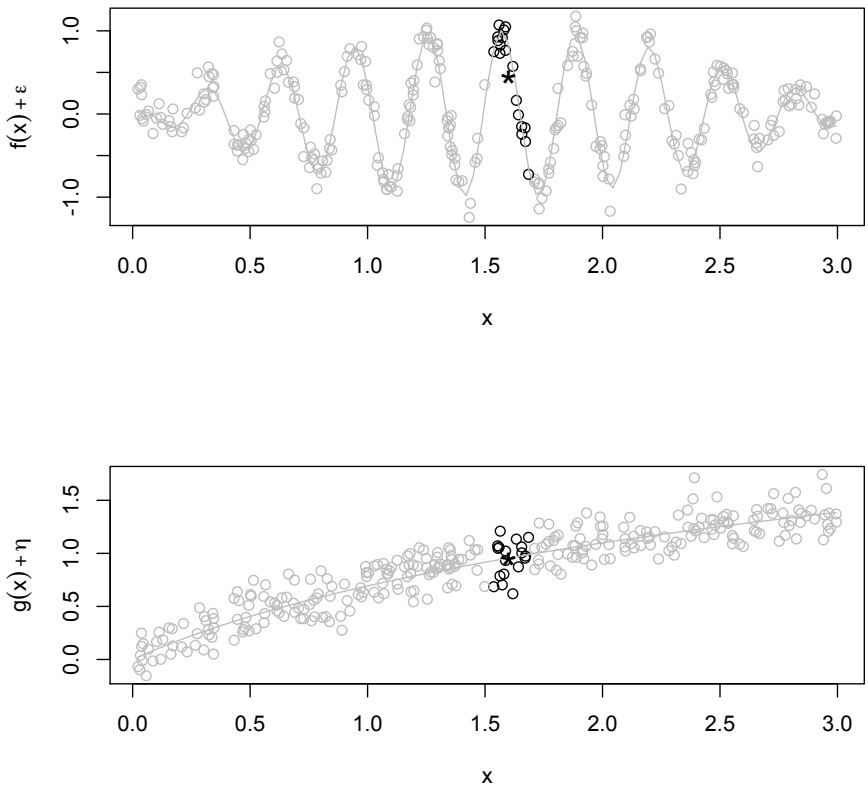
What would be really nice would be an **adaptive** procedure, one where our actual MSE, using $\widehat{h}$, approaches the Oracle's MSE, which it gets from h_{opt}. This would mean that, in effect, we are *figuring out* how rough the underlying regression function is, and so how much smoothing to do, rather than having to guess or be told. An adaptive procedure, if we can find one, is a partial[4] substitute for prior knowledge.

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 random-division CV would work in the usual way, going over a grid of possible bandwidths. Here is how it would work with the input variable being in the vector `x` (one dimensional) and the response in the vector `y` (one dimensional), and using the `npreg` function from the `np` library (`?`).[5]

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

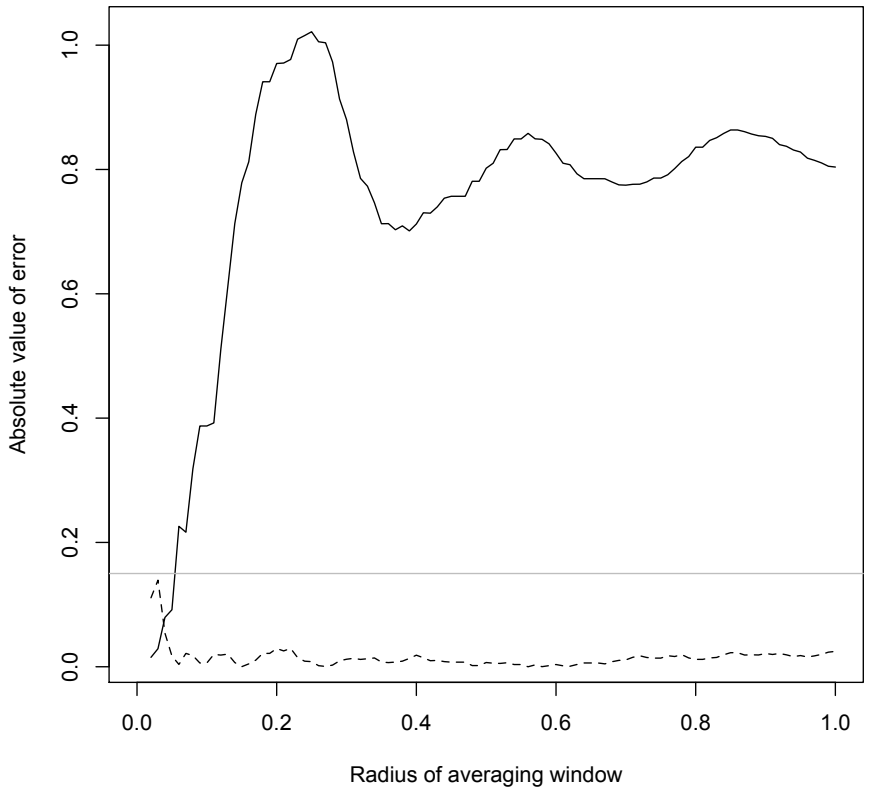
[4]Only partial, because we'd *always* do better if the Oracle would just tell us h_{opt}.

[5]The `np` package actually has a function, `npregbw`, which automatically selects bandwidths through a sophisticated combination of cross-validation and optimization techniques. It tends to be very slow.




```
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 5: 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")
```

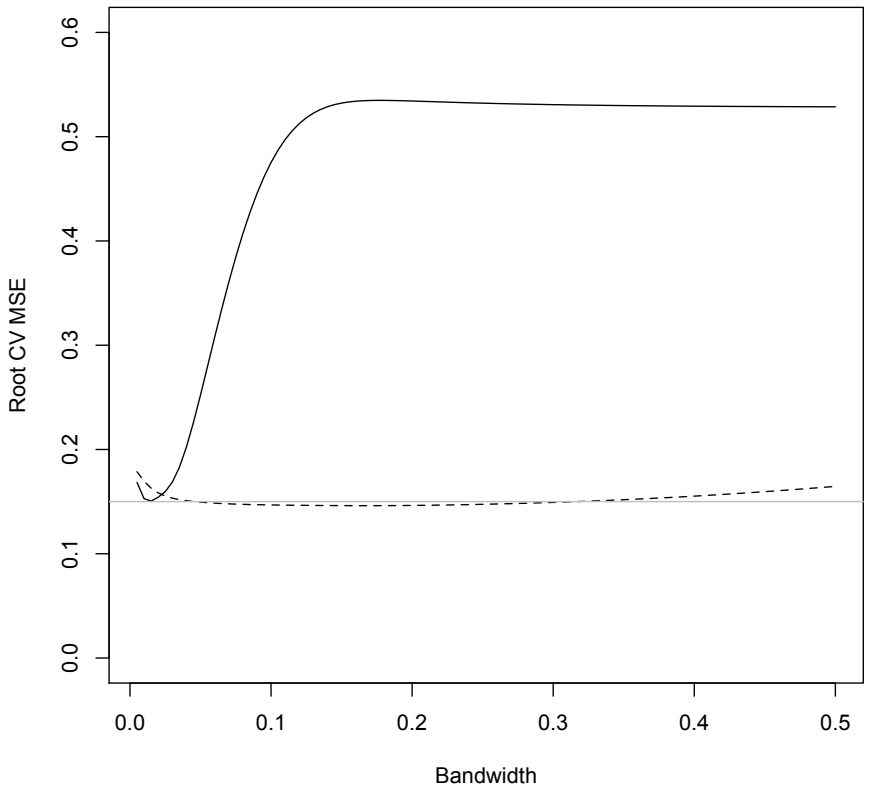
Figure 6: Estimating $f(1.6)$ and $g(1.6)$ from averaging observed values at $1.6 - h < x < 1.6 + h$, for different radii h. Solid line: error of estimates of $f(1.6)$; dashed line: error of estimates of $g(1.6)$; grey line: σ, the standard deviation of the noise.

```
# Multi-fold 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 1: 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 default vector of default bandwidths is pretty arbitrary; one could do better.

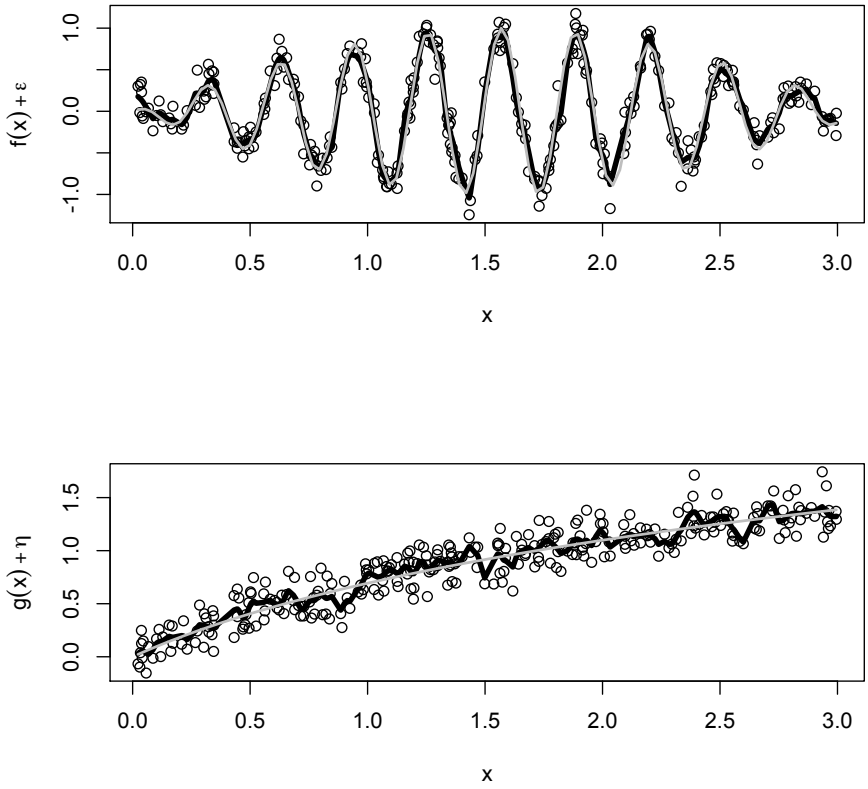
third component is an array which gives the MSE for each bandwidth on each fold. It can be useful to know things like whether the difference between the CV score of the best bandwidth and the runner-up is bigger than their fold-to-fold variability.

Figure 7 plots the CV estimate of the (root) mean-squared error versus bandwidth for our two curves. Figure 8 shows the data, the actual regression functions and the estimated curves with the CV-selected bandwidths.




```
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 7: Cross-validated estimate of the (root) mean-squard 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))
lines(x[x.ord],fitted(npreg(bws=fbws$best.bw,txdat=x,tydat=yf))[x.ord],lwd=4)
curve(sin(x)*cos(20*x),col="grey",add=TRUE,lwd=2)
plot(x,yg,xlab="x",ylab=expression(g(x)+eta))
lines(x[x.ord],fitted(npreg(bws=fbws$best.bw,txdat=x,tydat=yg))[x.ord],lwd=4)
curve(log(x+1),col="grey",add=TRUE,lwd=2)
```

Figure 8: 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.

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

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

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[6]. This leads to two popular choices: multivariate Gaussians (see the appendix), and product kernels.

Using the multivariate Gaussian, one would set the mean vector $\vec{\mu} = 0$, but one would still need to set the covariance matrix $\mathbf{\Sigma}$. The diagonal elements of $\mathbf{\Sigma}$ are variances, so picking them corresponds to picking bandwidths, but the off-diagonal parts, the covariances, correspond to guesses about how combinations of variables should trade off against each other. If the kernel was the Gaussian shown in Figure 13, and the two points marked + were two values of $\vec{x}_i$, we would give more weight to the point to the right of the center than the one above it, because of the *direction* of the displacement, even though the geometric distance is equal. This can be a powerful tool, if we know how to set it the covariance, but it is often hard to do well.

It is more common to use **product kernels**, which is to say to use a different kernel for each component, and take their product:

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

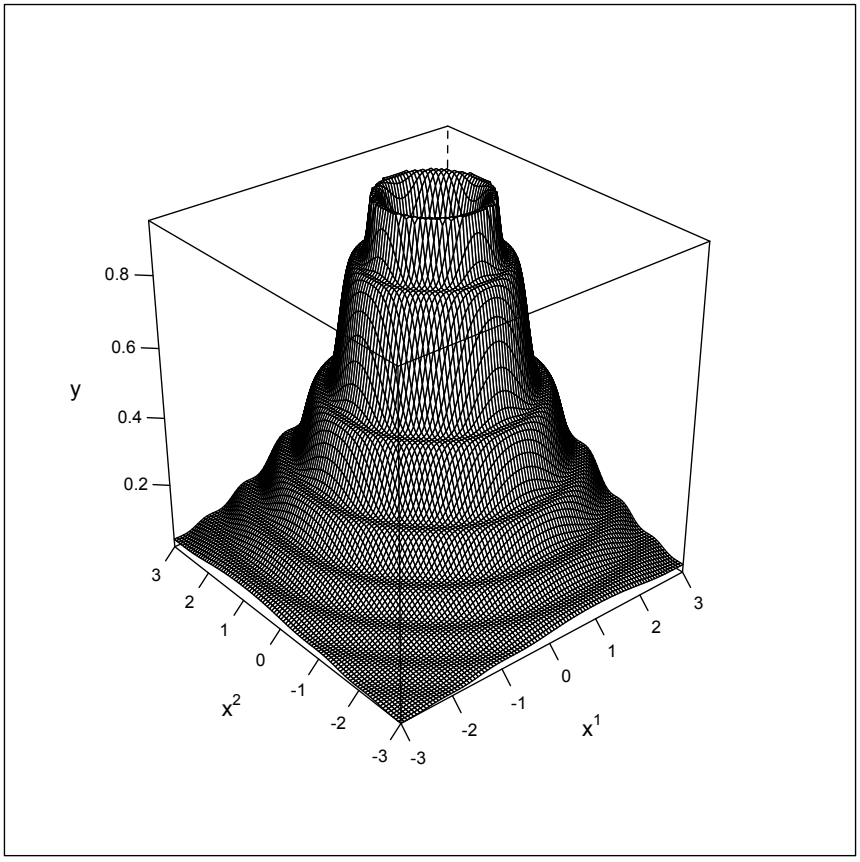
Now we just need to pick a bandwidth for each kernel, which in general should not be equal — say $\vec{h} = (h_1, h_2, \ldots h_d)$. Now instead of having a one-dimensional error curve, as in Figure 1 or 1, we will have a d-dimensional error surface, but we can still use cross-validation to find the 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

[6] 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.

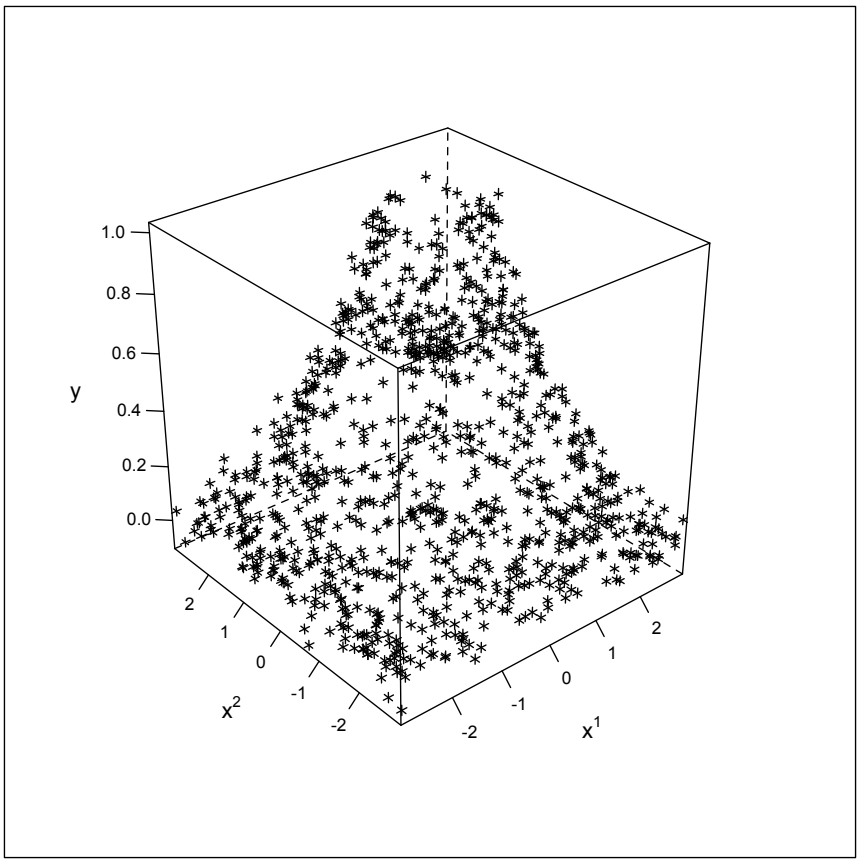
the others, cross-validation will tend to give them the maximum possible bandwidth, and smooth away their influence. We will look, later, at formal tests based on this idea.)

Whether we use product kernels or some multivariate distribution including correlations, 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 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 11 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 analytic geometry, the type of shape we were looking for. Kernel smoothing *discovered* the right general form.



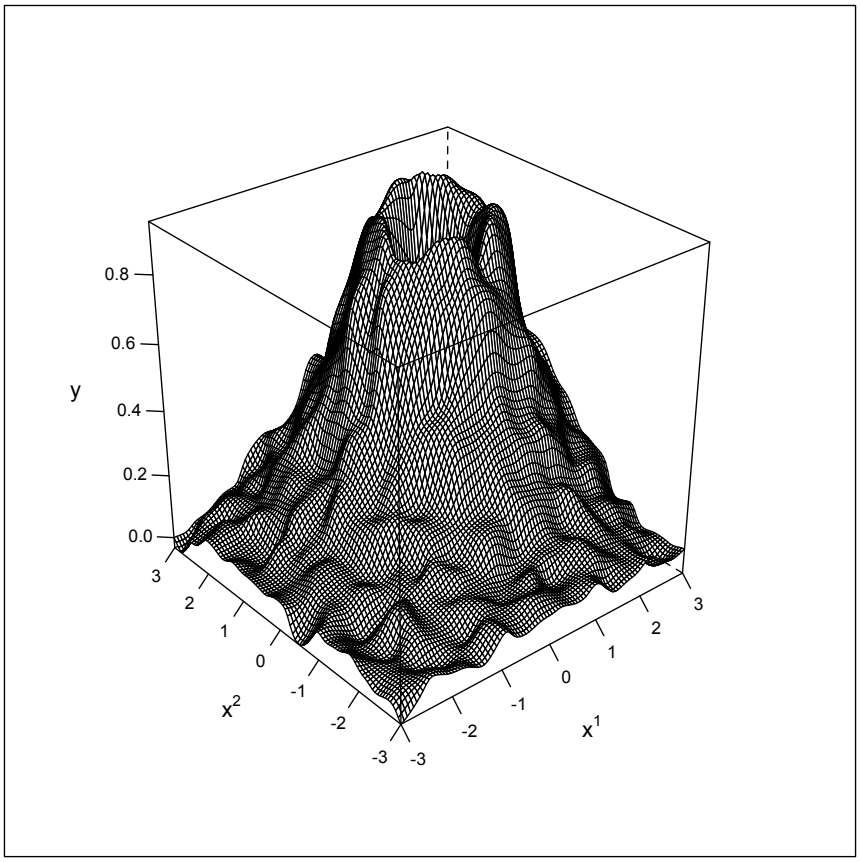

```
x.points <- seq(-3,3,length.out=100)
y.points <- x.points
xygrid <- expand.grid(x=x.points,y=y.points)
z <- matrix(0,nrow=100,ncol=100)
for (i in 1:100) {
  for (j in 1:100) {
     z[i,j] <- f(x.points[i],y.points[j])
  }
}
library(lattice)
wireframe(z~xygrid$x*xygrid$y,scales=list(arrows=FALSE),xlab=expression(x^1),
          ylab=expression(x^2),zlab="y")
```

Figure 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 function `f` is?) — `wireframe` is from the graphics library `lattice`.




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

Figure 10: 1000 data points, randomly sampled from the surface in Figure 9, plus independent Gaussian noise (s.d. = 0.05).




```
noise.np <- npreg(z~x+y,data=noise)
z.out <- matrix(0,100,100)
for (i in 1:100) {
  for (j in 1:100) {
    z.out[i,j] <- predict(noise.np,newdata=data.frame(x=x.points[i],
                                                      y=y.points[j]))
  }
}
wireframe(z.out~xygrid$x*xygrid$y,scales=list(arrows=FALSE),
          xlab=expression(x^1),ylab=expression(x^2),zlab="y")
```

Figure 11: Gaussian kernel regression of the points in Figure 10. Notice that the estimated function will make predictions are arbitrary points, not just the places where there was training data.

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).

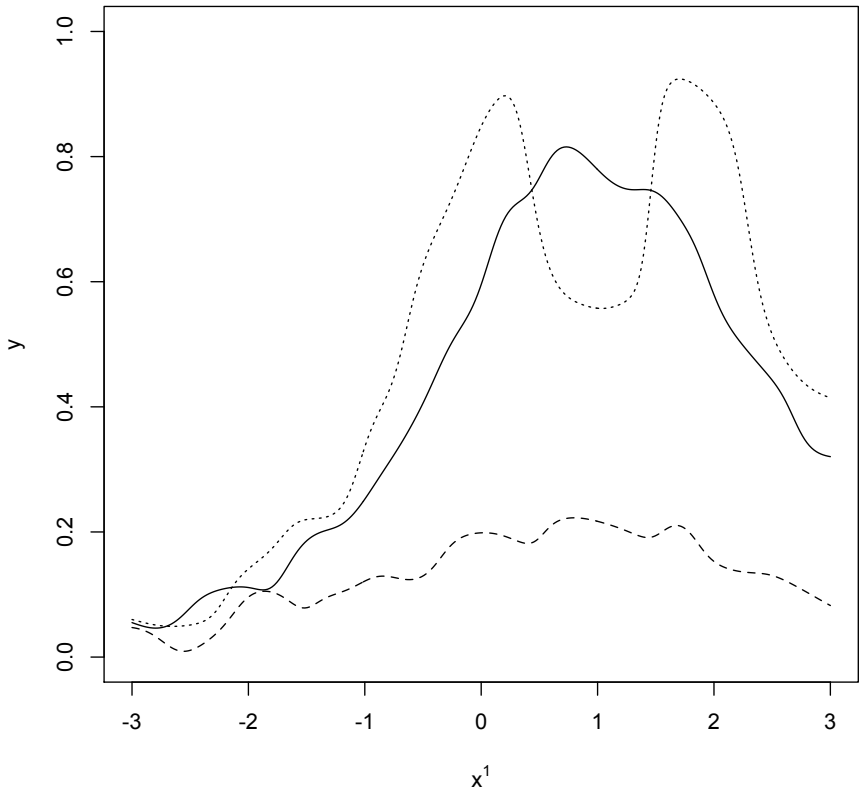
As soon as we introduce interactions, all this becomes harder, even for parametric regression. If there is an interaction between two components of the input, say x^1 and x^2, then we can't talk about the change in the expected response for a one-unit change in x^1 without saying what x^2 is. We might *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 12 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 Figure 12).

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.




```
new.frame <- data.frame(x=seq(-3,3,length.out=300),y=median(y.noise))
plot(new.frame$x,predict(noise.np,newdata=new.frame),type="l",xlab=expression(x^1),ylab="y"
new.frame$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 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^{th} percentile (dashed) and its 75^{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 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 25^{th} or the 75^{th} percentile for the second input.)

A The Multivariate Gaussian Distribution

The multivariate Gaussian is just the generalization of the ordinary Gaussian to vectors. Scalar Gaussians are parameterized by a mean μ and a variance σ^2, which we symbolize by writing $X \sim \mathcal{N}(\mu, \sigma^2)$. Multivariate Gaussians, likewise, are parameterized by a mean vector $\vec{\mu}$, and a variance-covariance matrix $\mathbf{\Sigma}$, written $\vec{X} \sim \mathcal{MVN}(\vec{\mu}, \mathbf{\Sigma})$. The components of $\vec{\mu}$ are the means of the different components of $\vec{X}$. The i, j^{th} component of $\mathbf{\Sigma}$ is the covariance between X^i and X^j (so the diagonal of $\mathbf{\Sigma}$ gives the component variances).

Just as the probability density of scalar Gaussian is

$$p(x) = \left(2\pi\sigma^2\right)^{-1/2} \exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\}$$

the probability density of the multivariate Gaussian is

$$p(\vec{x}) = (2\pi \det \mathbf{\Sigma})^{-d/2} \exp\left\{-\frac{1}{2}(\vec{x}-\vec{\mu}) \cdot \mathbf{\Sigma}^{-1}(\vec{x}-\vec{\mu})\right\}$$

Finally, remember that the parameters of a Gaussian change along with linear transformations

$$X \sim \mathcal{N}(\mu, \sigma^2) \Leftrightarrow aX + b \sim \mathcal{N}(a\mu + b, a^2\sigma^2)$$

and we can use this to "standardize" any Gaussian to having mean 0 and variance 1 (by looking at $\frac{X-\mu}{\sigma}$). Likewise, if

$$\vec{X} \sim \mathcal{MVN}(\vec{\mu}, \mathbf{\Sigma})$$

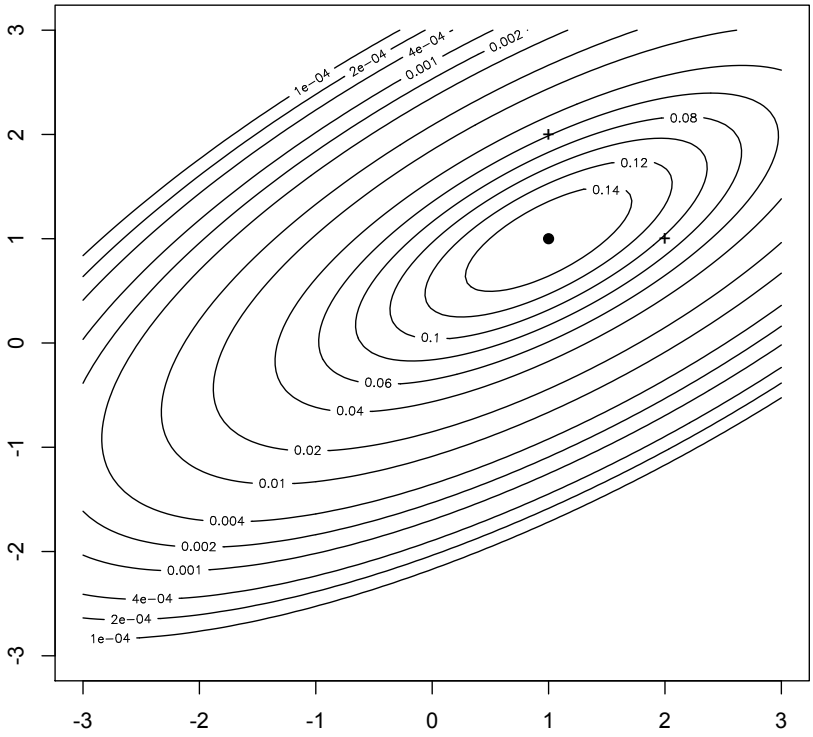
then

$$\mathbf{a}\vec{X} + \vec{b} \sim \mathcal{MVN}(\mathbf{a}\vec{\mu} + \vec{b}, \mathbf{a}\mathbf{\Sigma}\mathbf{a}^T)$$

In fact, the analogy between the ordinary and the multivariate Gaussian is so complete that it is very common to not really distinguish the two, and write $\mathcal{N}$ for both.

The multivariate Gaussian density is most easily visualized when $d = 2$, as in Figure 13. The probability contours are ellipses. The density changes comparatively slowly along the major axis, and quickly along the minor axis. The two points marked + in the figure have equal geometric distance from $\vec{\mu}$, but the one to its right lies on a higher probability contour, because of the direction of the separation.

For future reference, note that the axes of the probability-contour ellipses would be parallel to the coordinate axes if and only if the components of the vector were uncorrelated, and $\mathbf{\Sigma}$ was a diagonal matrix. We will see later that the axes of the ellipses correspond to the eigenvectors of $\mathbf{\Sigma}$, and the radii are proportional to the eigenvalues. (This is *not* obvious.)




```
library(mvtnorm)
x.points <- seq(-3,3,length.out=100)
y.points <- x.points
z <- matrix(0,nrow=100,ncol=100)
mu <- c(1,1)
sigma <- matrix(c(2,1,1,1),nrow=2)
for (i in 1:100) {
  for (j in 1:100) {
     z[i,j] <- dmvnorm(c(x.points[i],y.points[j]),mean=mu,sigma=sigma)
  }
}
contour(x.points,y.points,z)
```

Figure 13: Probability density contours for a two-dimensional multivariate Gaussian, with mean $\vec{\mu} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ (solid dot), and variance matrix $\mathbf{\Sigma} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$.