Using Nonparametric Smoothing: Adaptation and Testing Parametric Models

36-350, Data Mining

24 October 2008

We are still talking about regression, i.e., a supervised learning problem.

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:

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

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. (See section 4 of lecture 16 for refreshers on both of these.)

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

1 How Much Smoothing? Adapting to Unknown Roughness

Consider Figure 1, 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. 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 $g(\cdot)$.

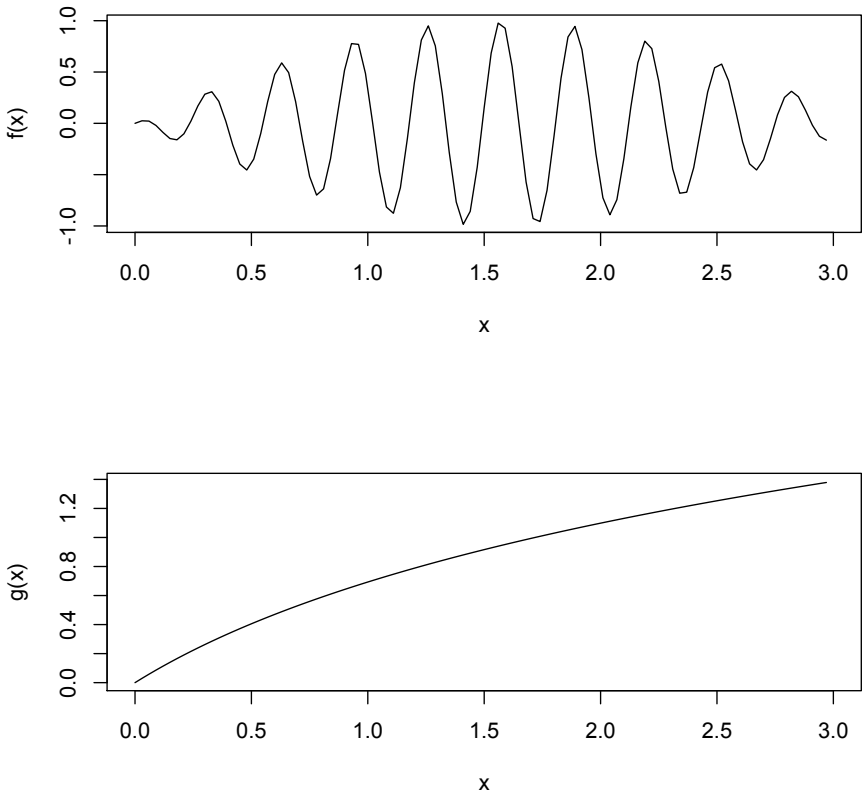


Figure 1: Two curves for the running example. Above, $f(x)$; below, $g(x)$. (As it happens, $f(x) = \sin x \cos 20x$, and $g(x) = \log x + 1$, but that doesn't really matter.)

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. (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 2. 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 $r(x) = \mathbf{E}[Y|X = x]$, 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

$$r(x_i) \approx r(x) + (x_i - x)r'(x)$$

and

$$y_i \approx r(x) + (x_i - x)r'(x) + \epsilon_i$$

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 $\mathbf{E}[\epsilon_i] = \sigma^2$.)

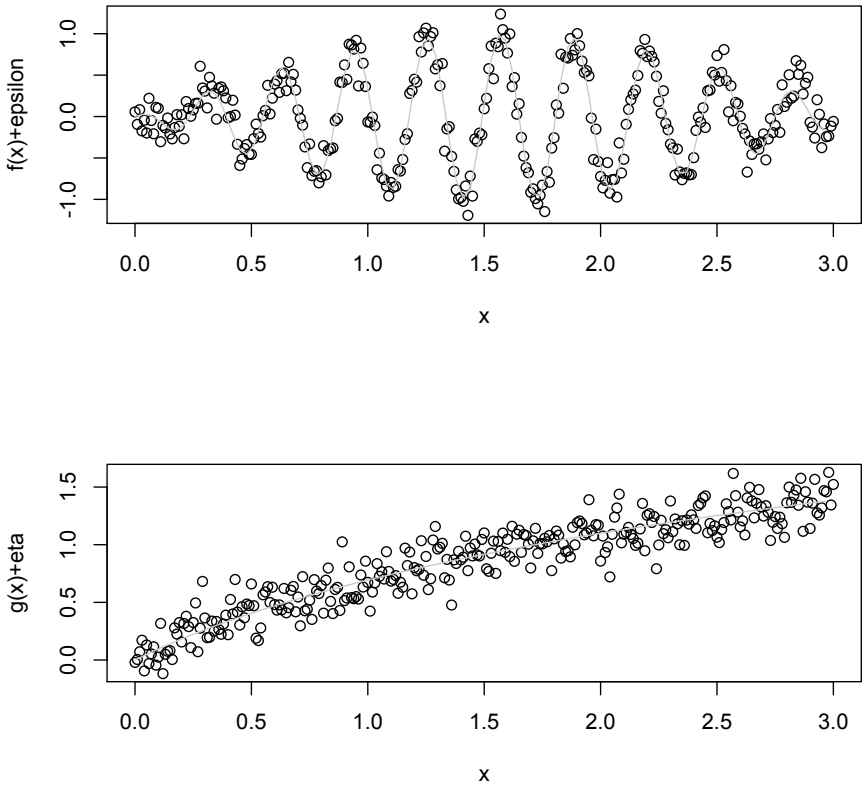


Figure 2: The same two curves as before, but corrupted by IID Gaussian noise with mean zero and standard deviation 0.15. (Different noise realizations for the two curves.) The light grey line shows the noiseless curves.

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.[1] 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 $g(x_i) + \eta_i$ for $1.5 < x_i < 1.7$ with equal weights. For f, this gives 0.41, while $f(1.6) = 0.83$. For g, this gives 0.98, with $g(1.6) = 0.95$. (See figure 3). 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}.[2] 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[3] substitute for prior knowledge.

[1] 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.

[2] Notice that h is not a property of the data-generating process, like most parameters we estimate, but rather something which depends on both that process (here, the roughness of the regression function) *and* on our particular prediction method. The best bandwidth for Gaussian-kernel regression isn't the best bandwidth for box-car-kernel regression, and neither is the best number of neighbors for k-NN regression. In particular, the optimal h is going to change with n.

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

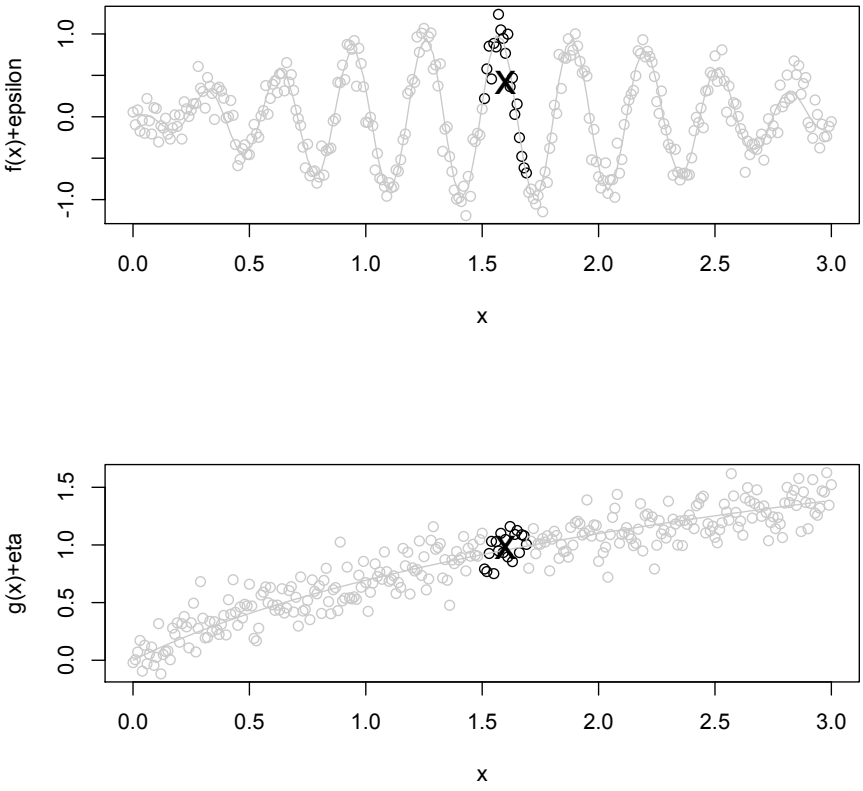


Figure 3: Relationship between smoothing and function roughness. In both the upper and lower panel we are trying to estimate the value of the regression function at $x = 1.6$ from averaging observations taken with $1.5 < x_i < 1.7$ (black points, others are "ghosted" in grey). The location of the average 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).

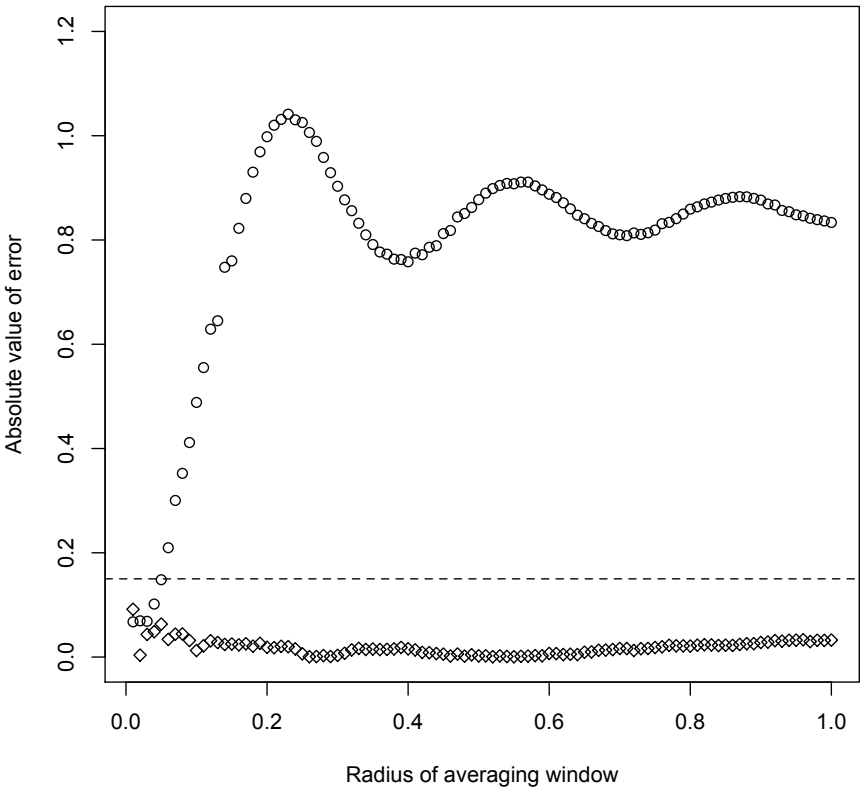


Figure 4: Estimating $f(1.6)$ and $g(1.6)$ from averaging observed values at $1.6 - h < x < 1.6 + h$, for different radii h. Circles: error of estimates of $f(1.6)$; diamonds: error of estimates of $g(1.6)$; dashed line: σ, the standard deviation of the noise.

The most straight-forward way to pick a bandwidth, and one which generally manages to be adaptive, is in fact cross-validation; either leave-one-out or k-fold CV usually works pretty well. 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` and the response in the vector `y`, and using the R-builtin function `ksmooth` with a Gaussian kernel.

```
bandwidths = (1:50)/50
num.bws = length(bandwidths)
MSEs = vector(length=num.bws)
MSEs = rep(0,num.bws)
names(MSEs) = bandwidths
num.folds=10
training.fraction = 0.9
n = length(x)
n.train = floor(training.fraction*n)
n.test = n - n.train
for (i in 1:num.folds) {
  train.rows = sample(1:n,size=n.train,replace=FALSE)
  x.train = x[train.rows]
  y.train = y[train.rows]
  x.test = x[-train.rows]
  y.test = y[-train.rows]
  # ksmooth returns predictions in order of increasing x values;
  # we may as well sort things into that order now
  y.test = y.test[order(x.test)]
  x.test = sort(x.test)
  for (bw in bandwidths) {
    predicted = ksmooth(x.train,y.train,kernel="normal",bandwidth=bw,
                        n.points=n.test,x.points=x.test)$y
    MSEs[paste(bw)] <- MSEs[paste(bw)] + mean((predicted-y.test)^2)
  }
}
MSEs = MSEs/num.folds
```

At the end of this, the vector `MSEs` contains the cross-validated mean-squared errors of all the different bandwidths in the vector `bandwidths`.[4] The best bandwidth can then be selected by `which.min`. All of this is encapsulated in a function, `cv.ksmooth`, on a later page.

[4] R Trickery Explanation: Notice that `MSEs` is first created as a vector of all 0s, and then its components are given names, which happen to the bandwidths. This allows us to access them later by those names. However, `bw` is a vector of numbers, and names are always character strings. When we say `names(MSEs) = bandwidths`, the `names` function is smart enough to automatically convert `bandwidths` into strings, but later when we want to access specific elements of `MSEs`, we have to do the conversion ourselves. The `paste` function is the easiest way to do this.

Figure 5 plots the CV estimate of the (root) mean-squared error versus bandwidth for our two curves. Figure 6 shows the data, the actual regression functions and the estimated curves with the CV-selected bandwidths.

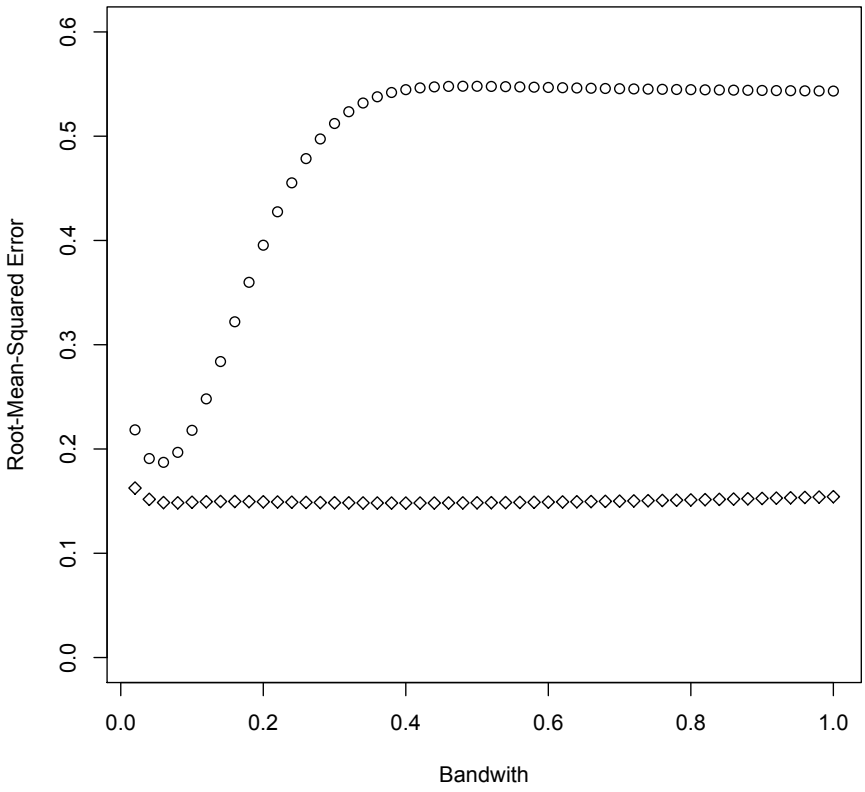


Figure 5: Cross-validated estimate of the (root) mean-squard error as a function of the bandwidth. Circles: data from $f(x)$; diamonds: data from $g(x)$. Notice that the rougher curve is more sensitive to the choice of bandwidth, and that the smoother curve is more preditable at every choice of bandwidth. CV selects bandwidths of 0.06 for f and 0.4 for g.

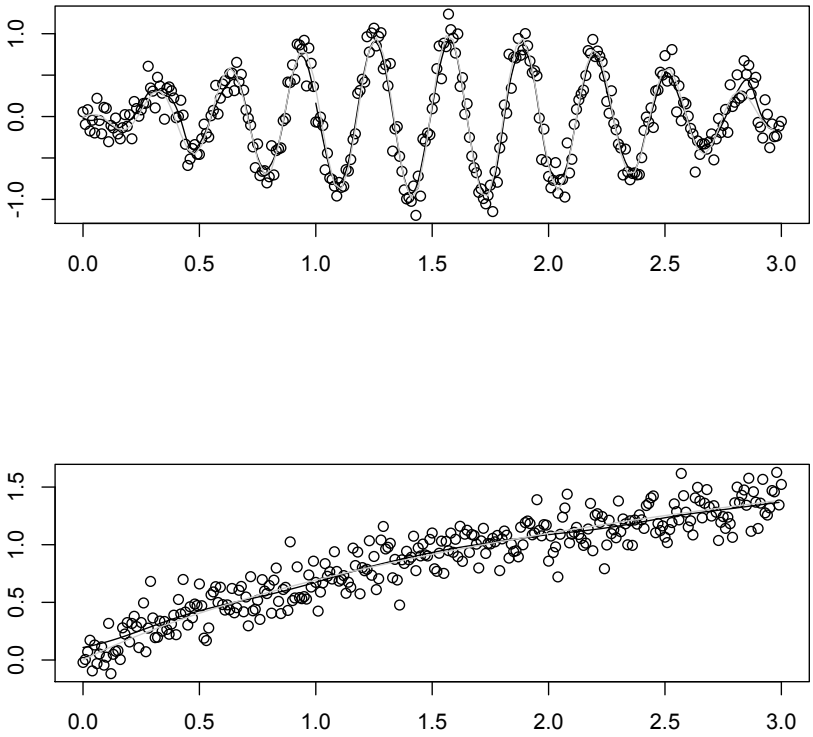


Figure 6: Data from the running examples (circles), actual regression functions (grey curves) and kernel estimates of regression functions with CV-selected bandwidths (black curves).

```
cv.ksmooth = function(x,y,bandwidths,num.folds=10,training.fraction=0.9) {
  num.bws = length(bandwidths)
  MSEs = vector(length=num.bws)
  MSEs = rep(0,num.bws)
  names(MSEs) = bandwidths
  n = length(x)
  n.train = floor(training.fraction*n)
  n.test = n - n.train
  for (i in 1:num.folds) {
    train.rows = sample(1:n,size=n.train,replace=FALSE)
    x.train = x[train.rows]
    y.train = y[train.rows]
    x.test = x[-train.rows]
    y.test = y[-train.rows]
    y.test = y.test[order(x.test)]
    x.test = sort(x.test)
    for (bw in bandwidths) {
      predicted = ksmooth(x.train,y.train,kernel="normal",bandwidth=bw,
                          n.points=n.test,x.points=x.test)$y
      MSEs[paste(bw)] <- MSEs[paste(bw)] + mean((predicted-y.test)^2)
    }
  }
  MSEs = MSEs/num.folds
  best.bw = as.numeric(names(which.min(MSEs)))
  fit = ksmooth(x,y,kernel="normal",bandwidth=best.bw,n.points=n,
                x.points=x)
  return(fit)
}
```

2 Testing Functional Forms

One important, but under-appreciated, use of nonparametric regression is in testing whether parametric regressions are well-specified.

The typical parametric regression model is something like

$$Y = f(X;\theta) + \epsilon$$

where f is some function which is completely specified except for the adjustable parameters θ, and ϵ, as usual, is uncorrelated noise. Overwhelmingly, f is linear in the variables in X, or perhaps includes some interactions between them.

How can we tell if the specification is right? If, for example, it's a linear model, how can we check whether there might not be some nonlinearity? One common approach is to modify the specification by adding in *specific* departures from the modeling assumptions — say, adding a quadratic term — and seeing whether the coefficients that go with those terms are significantly non-zero, or

whether the improvement in fit is significant.[5] For example, one might compare the model

$$Y = \theta_1 x_1 + \theta_2 x_2 + \epsilon$$

to the model

$$Y = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \epsilon$$

by checking whether the estimated θ_3 is significantly different from 0, or whether the residuals from the second model are significantly smaller than the residuals from the first.

This can work, *if* you have chosen the right nonlinearity to test. It has the power to detect certain mis-specifications, if they exist, but not others. (What if the departure from linearity is not quadratic but cubic?) If you have good reasons to think that if the model is wrong, it can only be wrong in certain ways, fine; if not, though, why only check for those errors?

Nonparametric regression effectively lets you check for *all* kinds of systematic errors, rather than singling out a particular one. Here is the basic procedure (though there are many variants).

1. Get data $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$.
2. Fit the parametric model, getting an estimate $\widehat{\theta}$, and mean-squared error $MSE_p(\widehat{\theta})$.
3. Fit your favorite nonparametric regression (using cross-validation to pick control settings as necessary), getting curve $\widehat{r}$ and mean-squared error $MSE_{np}(\widehat{r})$.
4. Calculate $t_{obs} = MSE_p(\widehat{\theta}) - MSE_{np}(\widehat{r})$. Generally (but not always!) $t_{obs} > 0$, since the nonparametric model has higher capacity than the parametric one does.
5. Simulate from the parametric model $\widehat{\theta}$ to get faked data $(x'_1, y'_1), \ldots (x'_n, y'_n)$.
6. Fit the parametric model to the simulated data, getting estimate $\tilde{\theta}$ and $MSE_p(\tilde{\theta})$.
7. Fit the nonparametric model to the simulated data, getting estimate $\tilde{r}$ and $MSE_{np}(\tilde{r})$.
8. Calculate $T = MSE_p(\tilde{\theta}) - MSE_{np}(\tilde{r})$.
9. Repeat steps 5–8 many times to get an estimate of the distribution of T.
10. The p-value is $\#\{T > t_{obs}\}/\#T$.

[5] In my experience, this is second in popularity only to ignoring the issue.

What's the logic here? We know that the nonparametric model has higher capacity than the parametric one, so it should generally be able to fit the data more closely, *even if* the parametric model is correct. So $t_{obs} > 0$ is positive isn't evidence against the parametric model by itself. But the *size* of the difference in fits matters. By simulating from the parametric model, we generating surrogate data which looks just like reality ought to, if the model is true. We then see how much better we could expect the nonparametric model to fit *under the parametric model*. If the non-parametric model fits the actual data much better than this, we can reject the parametric model with high confidence: it's really unlikely that we'd see that big an improvement from using the nonparametric model just by luck.[6]

Here is an example to show this in action. First, let's detect a reasonably subtle nonlinearity. The function $g(x)$ from the previous example is nonlinear. (Actually, $g(x) = \log x + 1$.) Figure 7 shows the regression function and the data (yet again). The nonlinearity is clear with the curve to guide the eye, but fairly subtle.

A simple linear regression actually looks pretty good:

```
> summary(lm(y~x))

Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-0.38063 -0.10262  0.00561  0.10158  0.43561

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.21141    0.01739   12.16   <2e-16 ***
x            0.42427    0.01003   42.29   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1512 on 299 degrees of freedom
Multiple R-squared: 0.8568,Adjusted R-squared: 0.8563
F-statistic:  1788 on 1 and 299 DF,  p-value: < 2.2e-16
```

R^2 is ridiculously high — over 85% of the variance is predicted by the line. The p-value reported by R is also very, very low, which seems good, but remember all this really means is "a sloped line is definitely better here than a flat line" (Figure 8)

The MSE of the linear fit is 0.0227:

[6] As usual with p-values, this is not symmetric. A high p-value might mean that the true regression function is very close to $f(x;\theta)$, or it might just mean that we don't have enough data to draw conclusions.

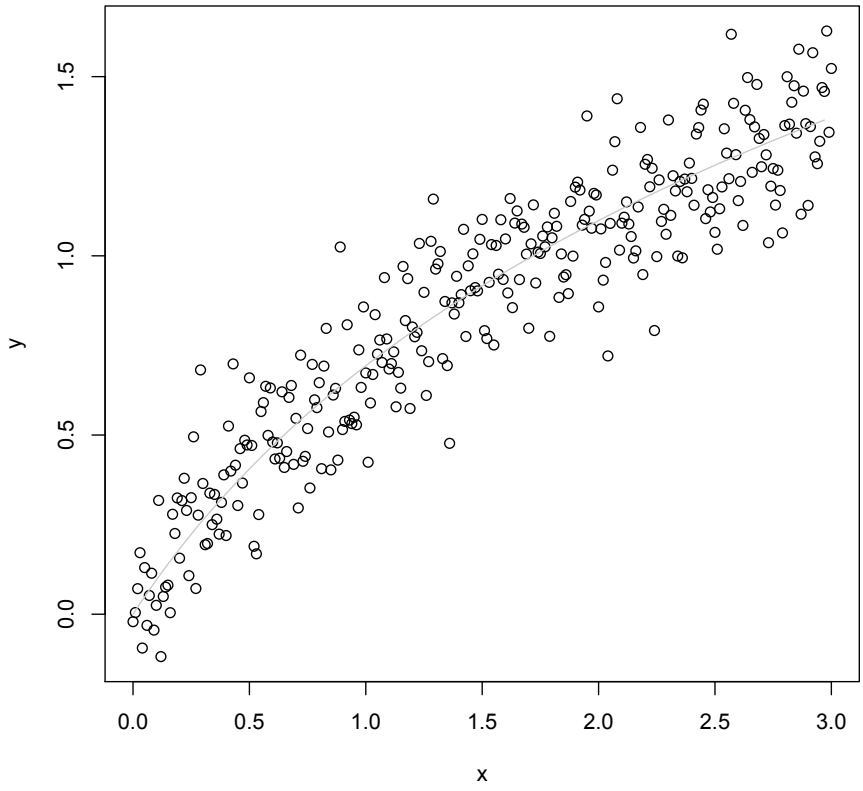


Figure 7: True regression curve (grey) and data points (circles). The curve $g(x) = \log x + 1$.

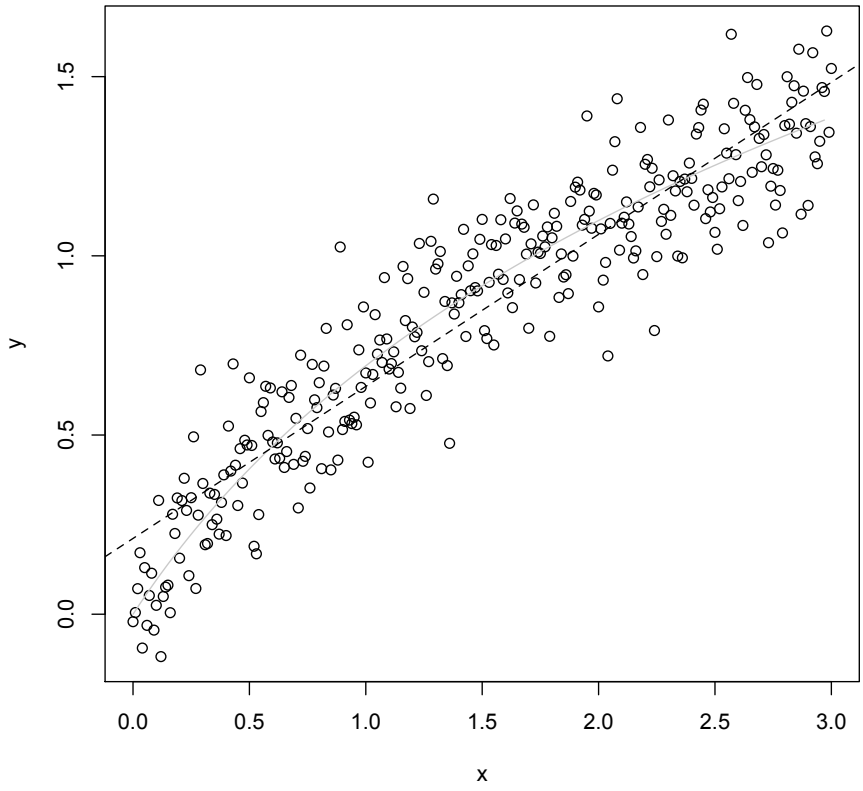


Figure 8: As previous figure, but adding the least-squares regression line (dashed). Compare to the bottom panel of Figure 6.

```
> mean((lm(y~x)$residuals)^2)
[1] 0.02272178
```

Using the same `ksmooth` fit as before for the nonparametric regression, the MSE is smaller, 0.0186.

```
> mean((y - ksmooth(x,y,"normal",0.4)$y)^2)
[1] 0.01860447
```

So $t_{obs} = 0.004$

```
> t.obs = mean((lm(y~x)$residuals)^2)
  - mean((y - ksmooth(x,y,"normal",0.4)$y)^2)
```

Now we need to simulate from the fitted parametric model. To do this we need the coefficients and the estimated noise level. We get them thus:

```
coefs = lm(y~x)$coefficients
sigma.hat = summary(lm(y~x))$sigma
```

Now we define simulation functions:

```
simul.lm = function(x) {coefs[1]+x*coefs[2] + rnorm(length(x),0,sigma.hat)}
```

This takes a vector of real numbers, treats them as inputs to the linear model, and gives us the corresponding (random) outputs. We don't really care about these values, however, just about the fits.

```
calc.T = function(x) {
  y.sim = simul.lm(x)
  MSE.p = mean((lm(y.sim~x)$residuals)^2)
  nonparam.fit = cv.ksmooth(x,y.sim,bandwidths=(1:50)/50)
  MSE.np = mean((y.sim - nonparam.fit$y)^2)
  return(MSE.p - MSE.np)
}
```

This uses the `cv.ksmooth` function defined earlier to pick the bandwidth by cross-validation on the *simulated* data.

Now we just call `calc.T` a lot to get a sampling distribution for T under the null hypothesis.

```
null.dist.T = replicate(1000,calc.T(x))
```

This takes some time, because each replication involves not just generating a new simulation sample, but also using ten-fold cross-validation to pick a bandwidth, which means that the simulated data undergoes ten random divisions into training and testing sets.

(While the computer is thinking, look at the command a little more closely. It leaves the `x` values alone, and only uses simulation to generate new y values. This is appropriate here because x was chosen deterministically, on a regular

grid. If the model we were testing specified a distribution for x, we should generate x each time we invoke `calc.T`. If the specification is vague, like "x is IID" but of no particular distribution, then use the bootstrap. The command would be something like

```
replicate(1000,calc.T(sample(x,size=length(x),replace=TRUE)))
```

to draw a different bootstrap sample of x each time.)

When it's done, we can plot the distribution and see that the observed value t_{obs} is pretty far out along the right tail (Figure 9). This tells us that it's very unlikely that `ksmooth` would improve so much on the linear model if the latter were true. In fact, we'd see that big an improvement only

```
> sum(null.dist.T > t.obs)/1000
[1] 0.018
```

1.8% of the time. We can thus reject the linear model pretty confidently.

As a second example, let's suppose that the linear model *is* right — then the test should give us a high p-value. So let us stipulate that in reality

$$Y = 0.2 + 0.5x + \eta$$

with $\eta \sim \mathcal{N}(0, 0.15^2)$. Figure 10 shows data from this, of the same size as before.

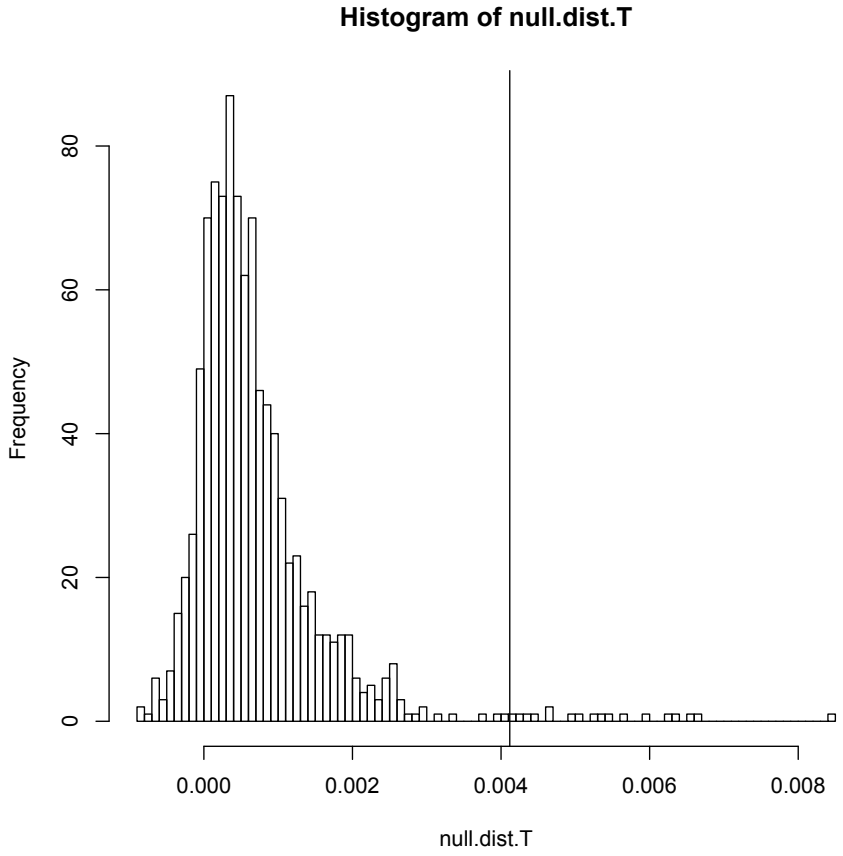
Repeating the same exercise as before, we get that $t_{obs} = 0.0015$, together with a slightly different null distribution (Figure 11). Now the p-value is a much higher 12%, which one would be quite rash to reject.

2.1 Why Use Parametric Models At All?

It might seem by this point that there is little point to using parametric models at all. Either our favorite parametric model is right, or it isn't. If it is right, then a consistent nonparametric estimate will eventually approximate it arbitrarily closely. If the parametric model is wrong, it will not self-correct, but the non-parametric estimate will eventually show us that the parametric model doesn't work. Either way, the parametric model seems superfluous.

There is an escape from this dilemma, by means of the bias-variance trade-off.[7] Low-dimensional parametric models have potentially high bias (if the real regression curve is very different from what the model posits), but low variance (because there isn't that much to estimate). Non-parametric regression models have low bias (they're flexible) but high variance (they're flexible). If the parametric model is true, it can converge *faster* than the non-parametric one. Even if the parametric model isn't quite true, the lower variance can still make it beat out the non-parametric model in over-all generalization error.

[7] Actually, there is another way out as well. *If* the parametric model actually represents our idea about the mechanism generating the data, then its parameters are substantively, "physically" meaningful and interesting, and the non-parametric smoothing doesn't capture that. However, this situation is so rare in data mining, as opposed to scientific inference, that I won't go further into this.




```
hist(null.dist.T,n=101)
abline(v=t.obs)
```

Figure 9: Distribution of $T = MSE_p - MSE_{np}$ for data simulated from the parametric model. The vertical line mark the observed value. Notice that the mode is positive and the distribution is right-skewed; this is typical.

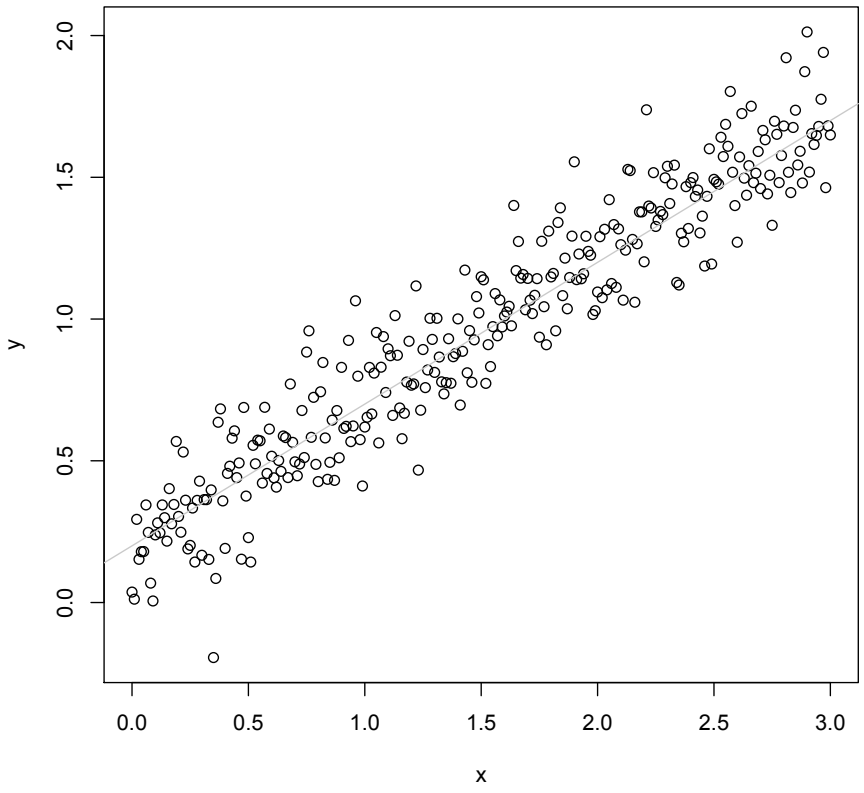


Figure 10: Data from the linear model (true regression line in grey).

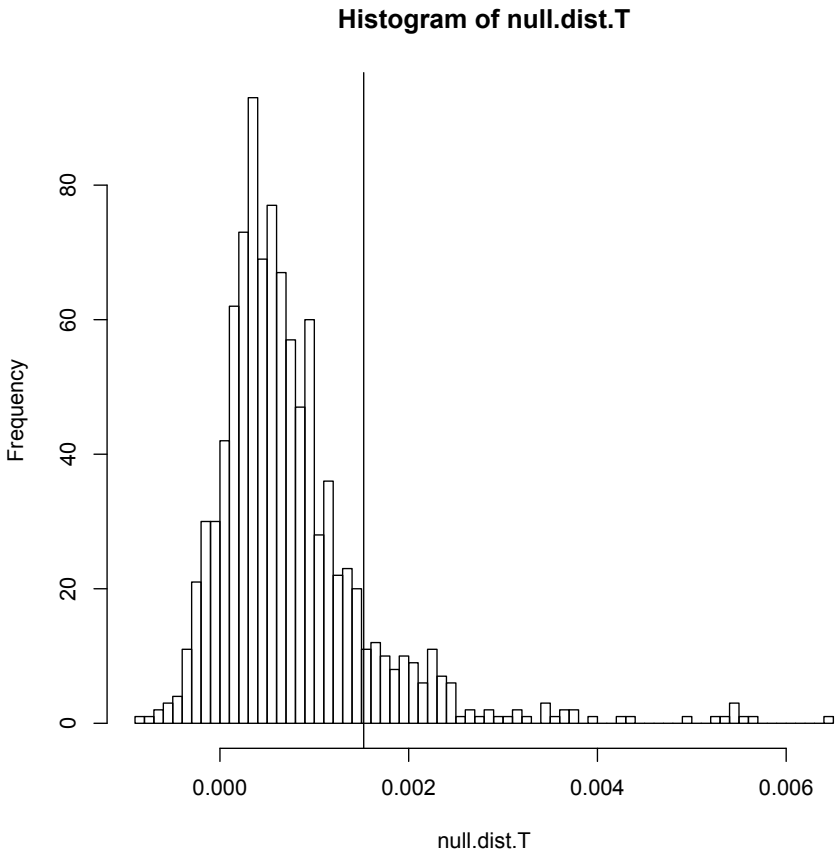


Figure 11: As in Figure 9, but using the data and fits from Figure 10.

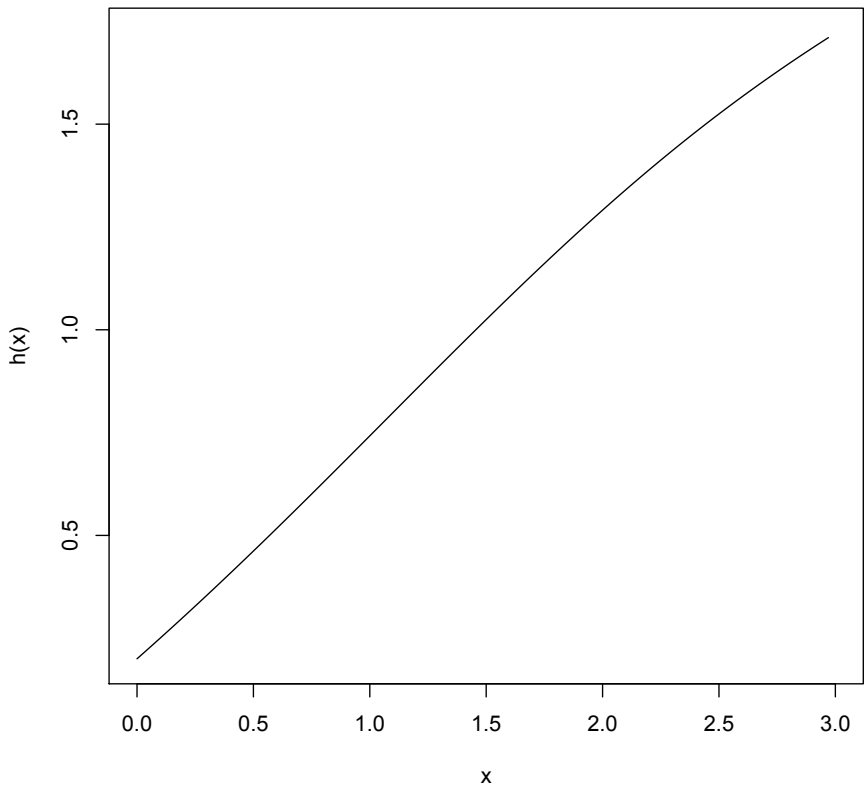


Figure 12: Graph of $0.2 + \frac{1}{2}\left(1 + \frac{\sin x}{10}\right) x$ over $[0, 3]$.

To illustrate, suppose that the true regression function is

$$\mathbf{E}[Y|X = x] = 0.2 + \frac{1}{2}\left(1 + \frac{\sin x}{10}\right) x$$

This is very nearly linear over small ranges — say $x \in [0, 3]$ (Figure 12).

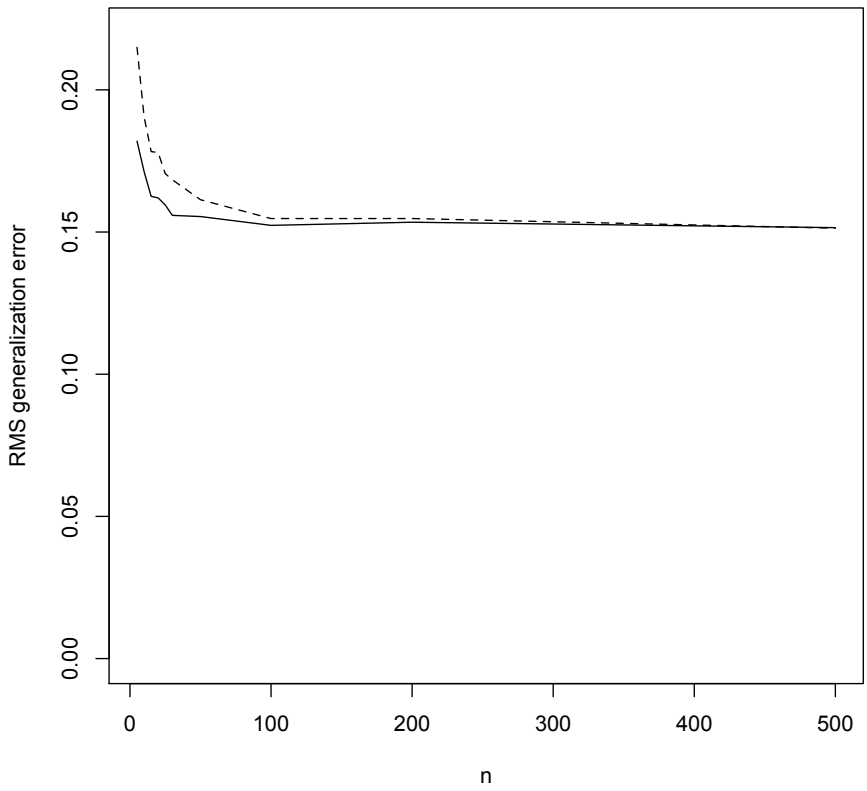
I will use the fact that I know the true model here to calculate the actual expected generalization error (by averaging over many samples).

```
nearly.linear.out.of.sample = function(n) {
  x=seq(from=0,to=3,length.out=n)
  y = h(x) + rnorm(n,0,0.15)
  y.new = h(x) + rnorm(n,0,0.15)
  lm.mse = mean((lm(y~x)$fitted.values - y.new )^2)
```

```
  np.mse = mean((cv.ksmooth(x,y,(1:50)/50)$y - y.new)^2)
  return(c(lm.mse,np.mse))
}

nearly.linear.generalization = function(n,m=100) {
  raw = replicate(m,nearly.linear.out.of.sample(n))
  reduced = rowMeans(raw)
  return(reduced)
}
```

Figure 13 shows that, out to a fairly substantial sample size (≈ 500), the lower bias of the non-parametric regression is systematically beaten by the lower variance of the linear model — though admittedly not by much.




```
sizes = c(5,10,15,20,25,30,50,100,200,500)
generalizations = sapply(sizes,nearly.linear.generalization)
plot(sizes,sqrt(generalizations[1,]),ylim=c(0,0.22),type="l",
     xlab="n",ylab="RMS generalization error")
lines(sizes,sqrt(results[2,]),lty=2)
```

Figure 13: Root-mean-square generalization error for linear model (solid line) and kernel smoother (dashed line), fit to the same sample of the indicated size. The true regression curve is as in 12, and observations are corrupted by IID Gaussian noise with $\sigma = 0.15$. The cross-over after which the nonparametric regressor has better generalization performance happens shortly before $n = 500$.