Chapter 4

DJM

18 February 2020

Announcements

Workflow for doing statistics

1. Choose a family of models.
2. Split the data in half (randomly)
3. For each model:
    1. Use half the data to…
    2. Calculate CV to get estimates of the risk.
    3. Choose the tuning parameter that gets the lowest estimate of the risk.
4. Choose a model by picking the **model** with the lowest estimate of the risk.
5. Evaluate and describe your model. Make plots, interpret coefficients, make predictions, etc. Use the **other half**. Why?
6. If you see things if 5 you don't like, propose a new model(s) to handle these issues and return to step 3.

"Smoothers" and easy CV

Linear smoothers

- Recall S431:

  The "Hat Matrix" puts the hat on Y: $\widehat{Y} = HY$.

- If I want to get fitted values from the linear model

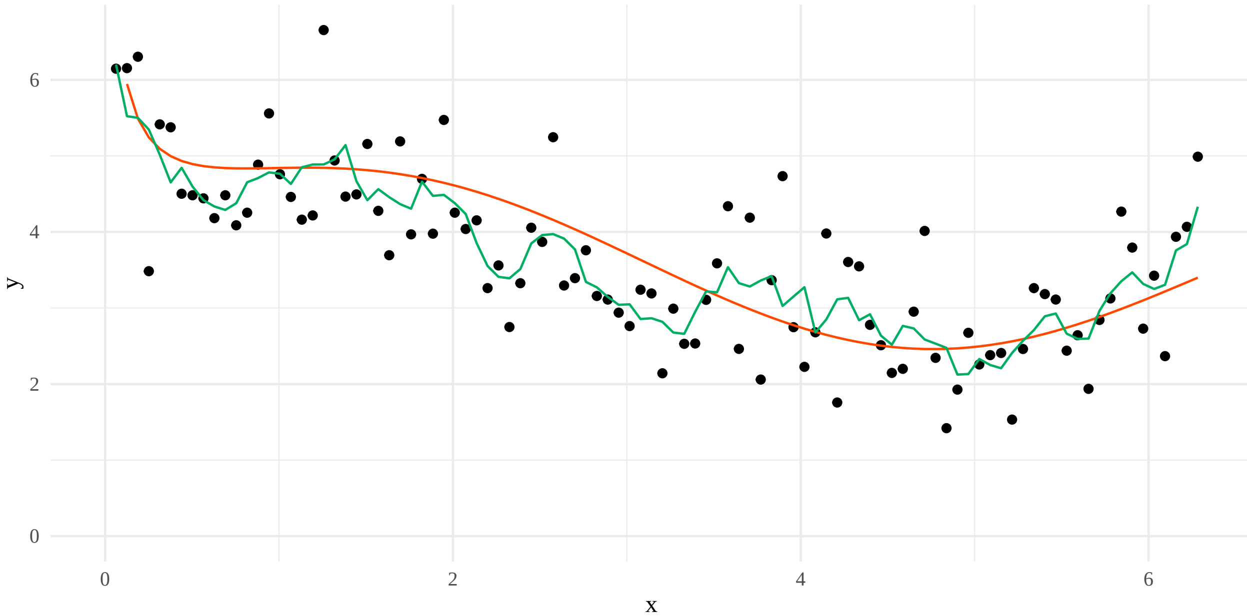
$$\widehat{Y} = X\widehat{\beta} = \left[X(X^\top X)^{-1}X^\top\right] Y = HY$$

- We generalize this to arbitrary matrices:

  A linear smoother is any predictor f that gives fitted values via $f(X) = WY$.

- Today, we will learn other ways of predicting Y from X.
- If I can get the fitted values at my original datapoints X by multiplying Y by a matrix, then that is a linear smoother.

Example



At each x, find 2 points on the left, and 2 on the right. Average their y values with that of your current point.

```
W = toeplitz(c(rep(1,3),rep(0,n-3)))
W = sweep(W, 1, rowSums(W), '/')
df$Yhat = W %*% df$y
geom_line(mapping = aes(x,Yhat), color=green)
```

This is a linear smoother. What is W?

What is W?

- I actually built this one directly into the code.
- An example with a 10 x 10 matrix:

```
W = toeplitz(c(rep(1,3),rep(0,7)))
round(sweep(W, 1, rowSums(W), '/'), 2)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] 0.33 0.33 0.33 0.00  0.0  0.0 0.00 0.00 0.00  0.00
##  [2,] 0.25 0.25 0.25 0.25  0.0  0.0 0.00 0.00 0.00  0.00
##  [3,] 0.20 0.20 0.20 0.20  0.2  0.0 0.00 0.00 0.00  0.00
##  [4,] 0.00 0.20 0.20 0.20  0.2  0.2 0.00 0.00 0.00  0.00
##  [5,] 0.00 0.00 0.20 0.20  0.2  0.2 0.20 0.00 0.00  0.00
##  [6,] 0.00 0.00 0.00 0.20  0.2  0.2 0.20 0.20 0.00  0.00
##  [7,] 0.00 0.00 0.00 0.00  0.2  0.2 0.20 0.20 0.20  0.00
##  [8,] 0.00 0.00 0.00 0.00  0.0  0.2 0.20 0.20 0.20  0.20
##  [9,] 0.00 0.00 0.00 0.00  0.0  0.0 0.25 0.25 0.25  0.25
## [10,] 0.00 0.00 0.00 0.00  0.0  0.0 0.00 0.33 0.33  0.33
```

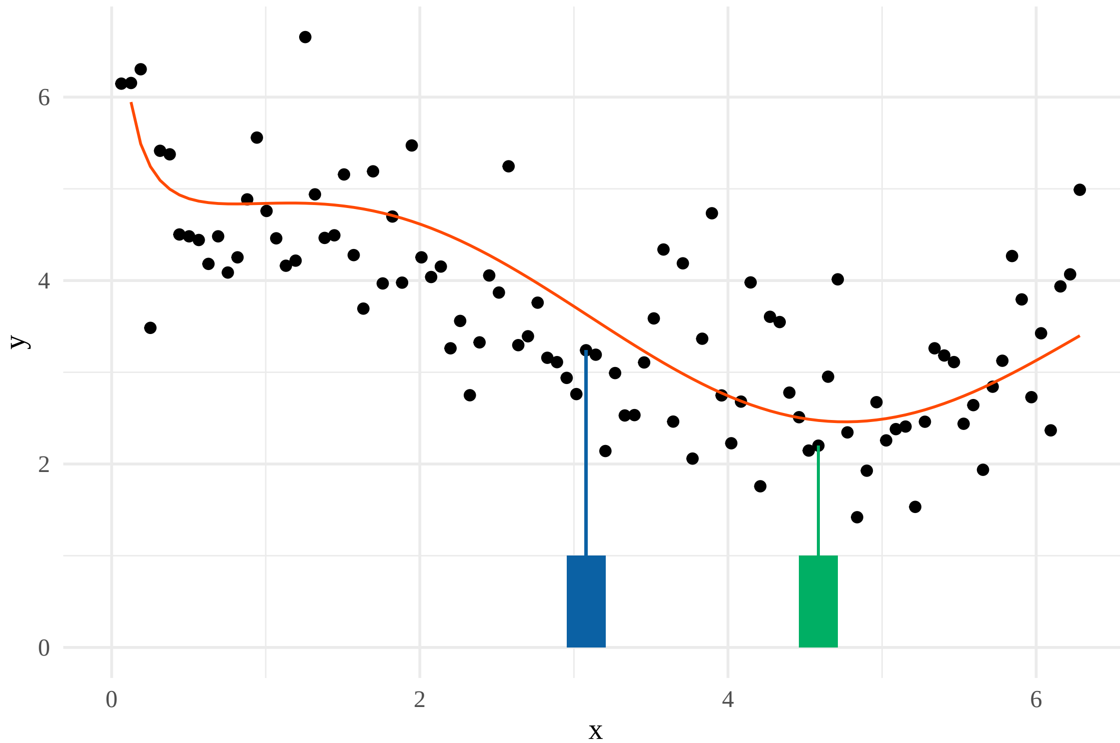
- This is a “kernel” smoother.

What is a "kernel" smoother?

- The mathematics:

  A kernel is any function K such that for any u, $K(u) \geq 0$, $\int duK(u) = 1$ and $\int uK(u)du = 0$.

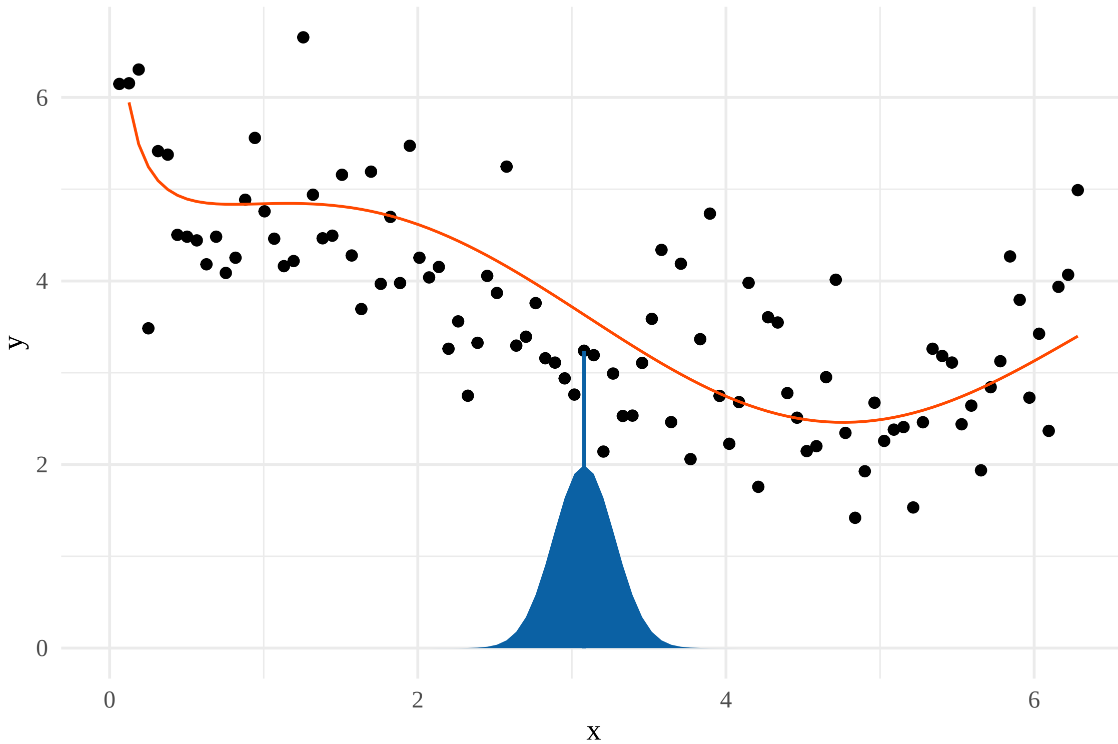
- The idea: a kernel is a nice way to take weighted averages. The kernel function gives the weights.
- The previous example is called the **boxcar** kernel. It looks like this:



- Notice that the kernel gets centered at each x. The weights of the average are determined by the shape of the kernel.
- For the boxcar, all the points inside the box get the same weight, all the rest get 0.

Other kernels

- Most of the time, we don't use the boxcar because the weights are weird.
- A more common one is the Gaussian kernel:



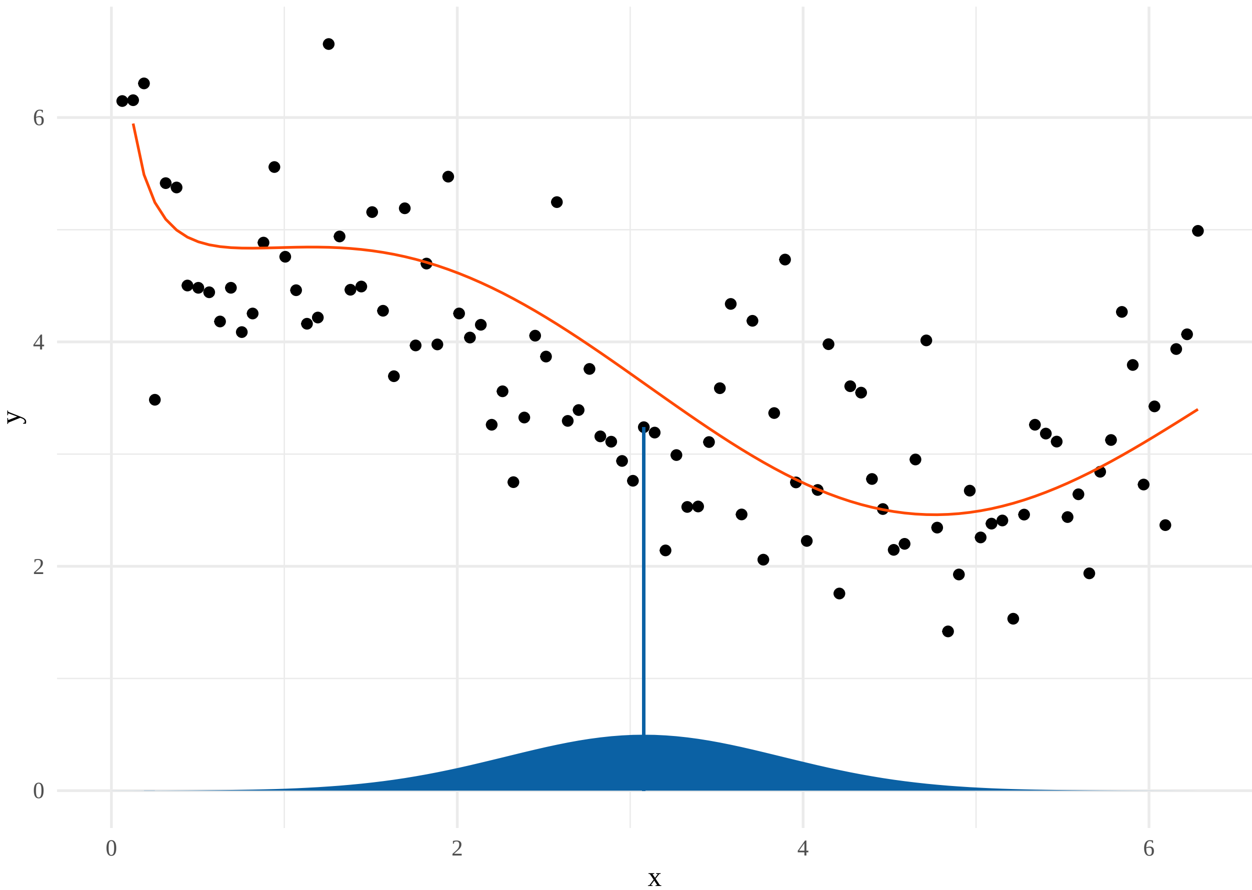
- Let's look at row 49 of the W matrix here:

$$W_{49,j} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x_j - x_{49})^2\right)$$

- For the plot, I made $\sigma = .2$.

Other kernels

- What if I made $\sigma = 0.8$?

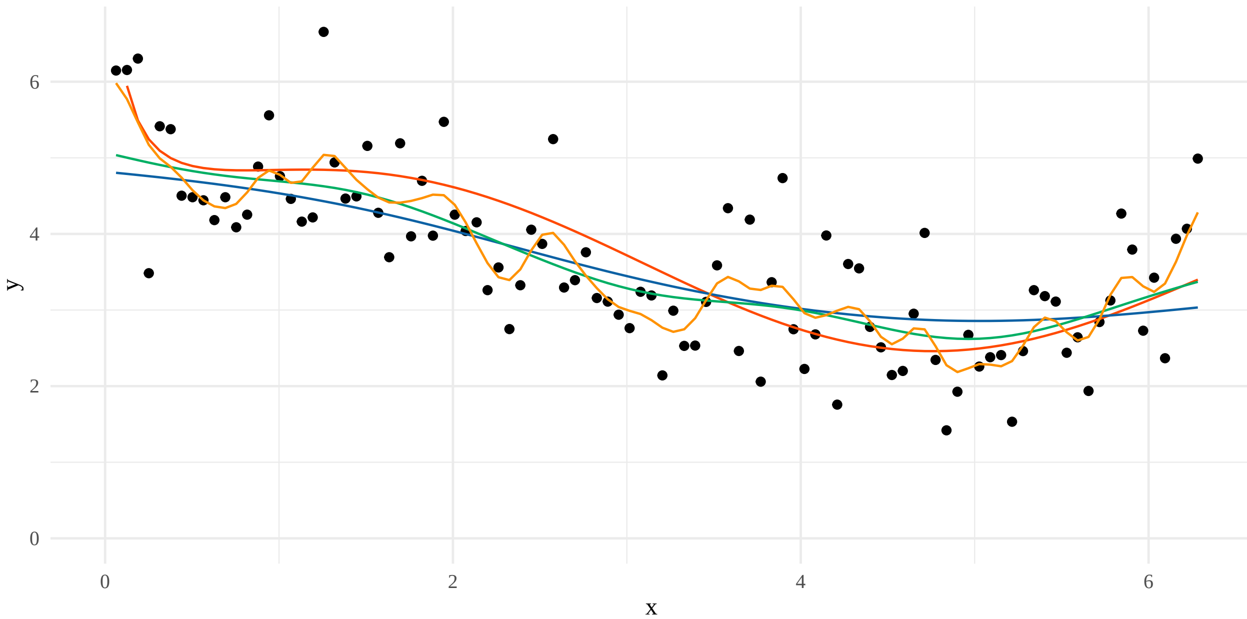


- Before, points far from x_{49} got very small weights for predicting at x_{49}, now they have more influence.
- For the Gaussian kernel, σ determines something like the "range" of the smoother.

Many Gaussians

- Using my formula for W, I can calculate different linear smoothers with different σ

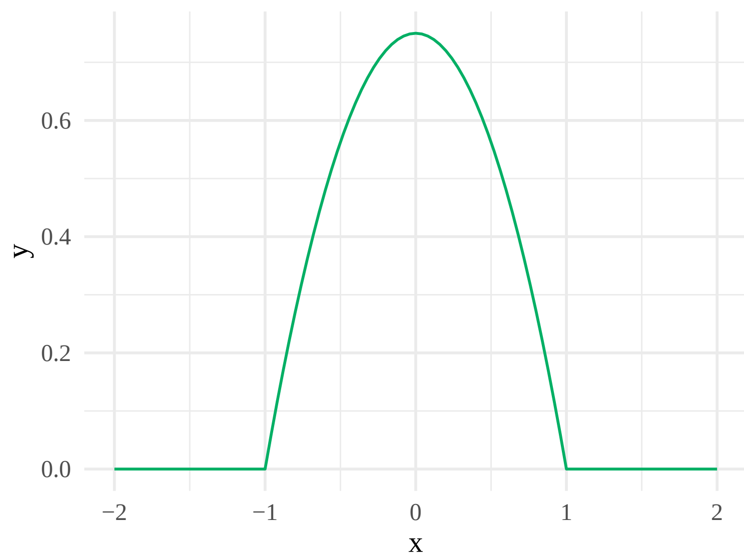
```
dmat = as.matrix(dist(x))
Wgauss <- function(sig){
  gg = exp(-dmat^2/(2*sig^2)) / (sig * sqrt(2*pi))
  sweep(gg, 1, rowSums(gg),'/')
}
df$W1 = with(df, Wgauss(1) %*% y)
df$W.5 = with(df, Wgauss(.5) %*% y)
df$W.1 = with(df, Wgauss(.1) %*% y)
ggplot(df, aes(x, y)) + geom_point() + xlim(0,2*pi) + ylim(0,max(df$y)) +
  stat_function(fun=trueFunction, color=red) +
  geom_line(aes(x, W1), color=blue) +
  geom_line(aes(x, W.5), color=green) +
  geom_line(aes(x, W.1), color=orange)
```



The bandwidth

- Choosing σ is **very** important.
- This "range" parameter is called the **bandwidth**.
- Most practitioners will tell you that it is way more important than which kernel you use.
- The default kernel is something called 'Epanechnikov':

```
epan <- function(x) 3/4*(1-x^2)*(abs(x)<1)
ggplot(data.frame(x=c(-2,2)), aes(x)) + stat_function(fun=epan,color=green)
```



How do you choose the bandwidth?

- Cross validation of course!

- Now the trick:

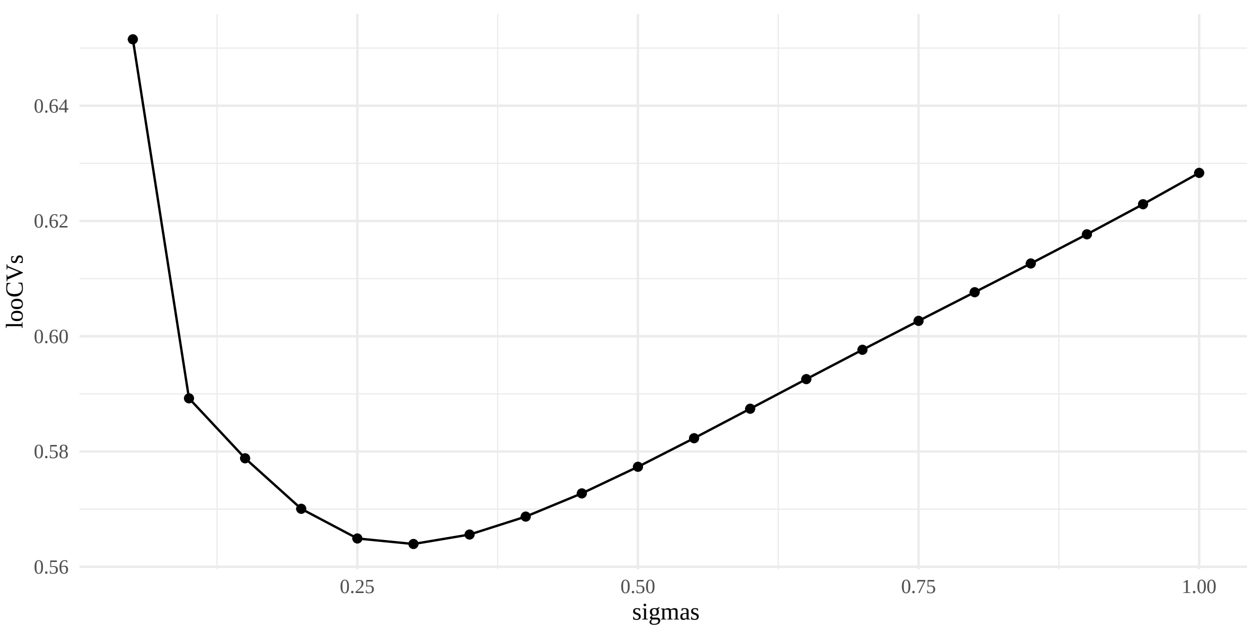
  For linear smoothers, one can show (after pages of tedious algebra which I wouldn't wish on my worst enemy, but might, in a fit of rage assign to a belligerent graduate student) that for $\widehat{Y} = WY$,

$$\text{LOO-CV} = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \widehat{y}_i)^2}{(1-w_{ii})^2} = \frac{1}{n}\sum_{i=1}^{n}\frac{\widehat{e}_i^2}{(1-w_{ii})^2}.$$

- This trick means that you only have to fit the model once rather than n times!
- You still have to calculate this for each model!

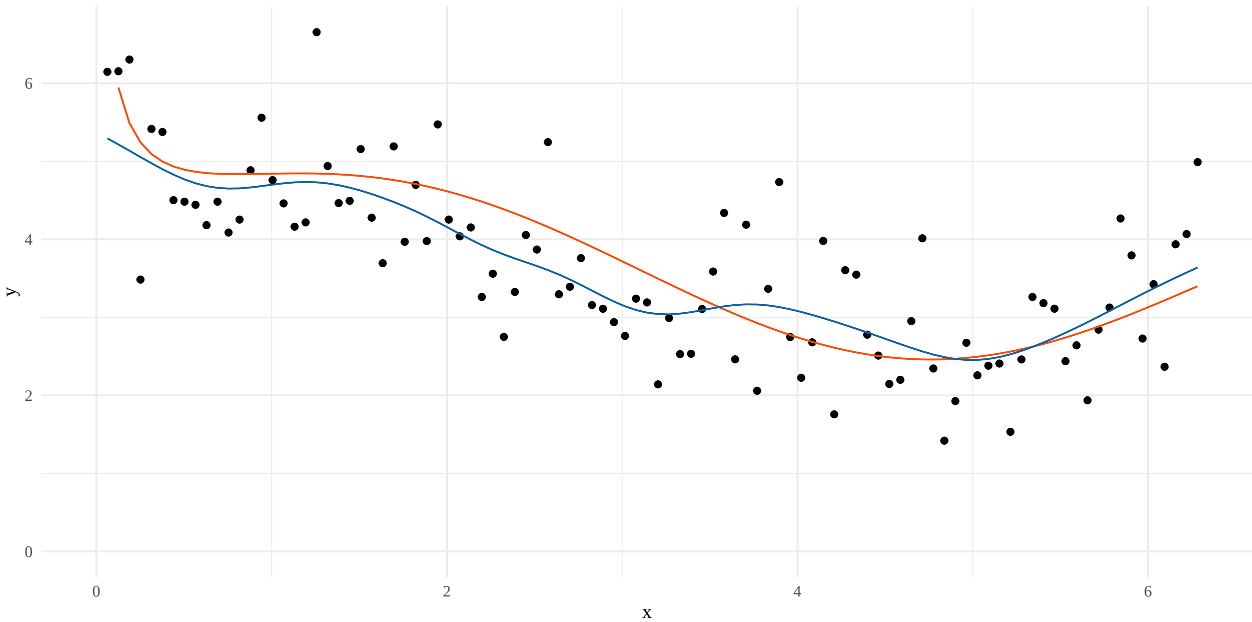
Back to my Gaussian example

```
looCV <- function(y, W){
  n = length(y)
  resids2 = ((diag(n)-W) %*% y)^2
  denom = (1-diag(W))^2
  return(mean(resids2/denom))
}

looCV.forNiceModels <- function(mdl){
  mean(residuals(mdl)^2/(1-hatvalues(mdl))^2)
}

looCVs = double(20)
sigmas = seq(.05, 1, length.out=length(looCVs))
for(i in 1:length(looCVs)){
  W = Wgauss(sigmas[i])
  looCVs[i] = looCV(df$y, W)
}
ggplot(data.frame(sigmas,looCVs),aes(sigmas,looCVs)) + geom_point() + geom_line()
```



Back to my Gaussian example

```
df$Wstar = with(df, Wgauss(sigmas[which.min(looCVs)]) %*% y)
ggplot(df, aes(x, y)) + geom_point() + xlim(0,2*pi) + ylim(0,max(df$y)) +
  stat_function(fun=trueFunction, color=red) +
  geom_line(aes(x, Wstar), color=blue)
```



Heads up on Ch. 4

Ugly formulas

- These are things like (4.10)-(4.12) and (4.14)
- The purpose of these formulas is to illustrate **VERY GENERALLY** how to trade bias and variance with Kernel smoothers.
- The highest level overview is equation (4.16):

$$MSE - \sigma^2(x) = O(h^4) + O(1/nh).$$

- Note: we have moved **irreducible noise** to the left of `=`.
- The first term on the right is the **squared bias** while the second term on the right is the **variance**.
- The "big-Oh" notation means we have removed a bunch of constants that don't depend on n or h.

[They DO depend on the properties of the Kernel, and the distribution which generated the data.]

- The **Optimal Bandwidth** minimizes the MSE:

$$\begin{aligned} h_{opt} &= \arg\min_h C_1 h^4 + \frac{C_2}{nh} \\ \Rightarrow 0 &\overset{set}{=} 4C_1 h^3 - \frac{C_2}{nh^2} \\ \Rightarrow h^5 &= O\left(\frac{1}{n}\right) \\ \Rightarrow h_{opt} &= O\left(\frac{1}{n^{1/5}}\right). \end{aligned}$$

- If we plug this in, we get the **Oracle MSE**—the MSE for the optimal, though unavailable estimator.

$$\begin{aligned} MSE - \sigma^2 &= O(h_{opt}^4) + O(1/nh_{opt}) \\ &= O(n^{-4/5}) + O(1/n^{4/5}) \\ &= O\left(\frac{1}{n^{4/5}}\right) \end{aligned}$$

Ok, you asked for the algebra.

- You don't want the algebra.
- Like the formula for LOO-CV, if I were a horrible, soul destroying person, I would wade through it for the next two hours (to get (4.10)).
- Believe me, I've done it. Not fun. The hand wavy, "big-Oh" stuff is what you should keep in mind.
- If you really want it, I will write up a document with all the work.

Kernels and interactions

- In multivariate kernel regressions, you estimate a **surface** over the input variables.
- This is trying essentially to find $\widehat{f}(x_1, \ldots, x_p)$.
- Therefore, this function **by construction** includes interactions, handles categorical data, etc. etc.
- This is contrast with **linear models** which need you to specify these things.
- This extra complexity (automatically including interactions, as well as other things) comes with tradeoffs.

Issue 1

- More complicated functions (smooth Kernel regressions vs. linear models) tend to have **lower bias** but **higher variance**.
- For $p = 1$, equations (4.19) and (4.20) show this:
- **Bias**
  1. The bias of using a linear model when it is wrong is a number $b(x, \theta_0)$ which doesn't depend on n.
  2. The bias of using kernel regression is $O(1/n^{4/5})$. This goes to 0 as $n \to \infty$.
- **Variance**
  1. The variance of using a linear model is $O(1/n)$

2. The variance of using kernel regression is $O(1/n^{4/5})$.

- To conclude: bias of kernels goes to zero (not for lines) but variance of lines goes to zero faster than for kernels.
- If the linear model is right, you win. But if it's wrong, you (eventually) lose.
- How do you know if you have enough data? Do model selection (CV to choose models).
- Compare of the kernel version with CV-selected tuning parameter (the CV estimate of the risk), with the CV estimate of the risk for the linear model.

Issue 2

- For $p > 1$, there is more trouble.
- First, lets look again at
$$MSE(h) - \sigma^2(x) = O(1/n^{4/5}).$$
That is for $p = 1$. It's not **that much** slower than $O(1/n)$, the variance for linear models.
- If $p > 1$ similar calculations show,
$$MSE(h) - \sigma^2(x) = O(1/n^{4/(4+p)}) \qquad MSE(\theta_0) - \sigma^2(x) = b(x, \theta_0) + O(p/n).$$
- What if p is big?
    1. Then $O(1/n^{4/(4+p)})$ is still big.
    2. But $O(p/n)$ is small.
    3. So unless $b(x, \theta_0)$ is big, we should use the linear model.
- How do you tell? Use CV to decide.

Issue 3

- When p is big, `npreg` is slow.
- Not much to do about that.
- Chapter 8 has some compromises that people use.
- A **very, very** questionable rule of thumb: if $p > \log(n)$, this may not work.

Summary

- This is the lesson of the class (the second one)
- How to do data analysis:

1. Choose a family of models. Some parametric and some nonparametric
2. Split the data in half (randomly)
3. For each model:
    1. Use half the data to…
    2. Calculate CV get estimates of the risk.
    3. Choose any tuning parameters by using the one that has the lowest CV.
4. Choose a model by picking the **model** with the lowest CV.
5. Evaluate and describe your model. Make plots, interpret coefficients, make predictions, etc. Use the **other half**.

6. If you see things if 5 you don't like, propose a new model(s) to handle these issues and return to step 3.

- We like CV. It is good.
- Split your data to make reasonable inferences.

`npreg` computational advice

- Read section 4.6 carefully, it will make your life much easier
- `npreg` works like `lm`: `out = npreg(y~x1+x2)`
- The `+` just means "use these variables"
- There's no reason to use `I(x1^2)` or `x1*x2`, it already does that. (Why?)
- `npreg` takes a little while to run, be sure to set `cache=TRUE` so you need only run it once.
- You can use `ordered(x2)` or `factor(x2)`. This may improve the speed a bit.
- DO NOT CROSS VALIDATE. `npreg` does it automatically. The CV risk estimate is in `out$bws$fval`.

Some more `npreg` discussion

- `npreg` is using CV and optimization to try to choose the bandwidth(s) for you.
- The `tol` and `ftol` arguments control how close the solution needs to be to an optimum.
- Very basic minimization (called Gradient descent):
  - Suppose I want to minimize $f(x) = (x-6)^2$ numerically.
  - If I start at a point (say $x_1 = 23$), vaguely, I want to "go" in the negative direction of the gradient.
  - The gradient (at $x_1 = 23$) is $f'(23) = 2(23-6) = 34$.
  - Gradient descent says, ok go that way by some small amount: $x_2 = x_1 - \gamma 34$, for γ. small.
  - In general, $x_{n+1} = x_n - \gamma f'(x_n)$.

```
niter = 10
gam = 0.1
x = double(niter)
x[1] = 23
grad <- function(x) 2*(x-6)
for(i in 2:niter) x[i] = x[i-1] - gam*grad(x[i-1])
x
```

```
##  [1] 23.000000 19.600000 16.880000 14.704000 12.963200 11.570560 10.456448
##  [8]  9.565158  8.852127  8.281701
```

- How do I decide if I'm done? The easiest way is to check how much I'm moving.

Fixing my gradient descent code

```
maxiter = 1000
conv = FALSE
gam = 0.1
x = 23
tol = 1e-3
grad <- function(x) 2*(x-6)
for(iter in  1:maxiter){
```

```r
  x.new = x - gam * grad(x)
  conv = (x - x.new < tol)
  x = x.new
  if(conv) break
}
x
```

```
## [1] 6.003531
```

```r
iter
```

```
## [1] 38
```

- What happens if I change `tol` to `1e-7`?