Additive Models

36-350, Data Mining, Fall 2009

2 November 2009

READINGS: *Principles of Data Mining*, pp. 393–395; Berk, ch. 2.

Contents

1 Partial Residuals and Backfitting for Linear Models

The general form of a linear regression model is

$$\mathbf{E}\left[Y|\vec{X}=\vec{x}\right] = \beta_0 + \vec{\beta}\cdot\vec{x} = \sum_{j=0}^{p}\beta_j x_j \tag{1}$$

where for $j \in 1:p$, the x_j are the components of $\vec{x}$, and x_0 is always the constant 1. (Adding a fictitious constant "feature" like this is a standard way of handling the intercept just like any other regression coefficient.)

Suppose we don't condition on all of $\vec{X}$ but just one component of it, say X_k. What is the conditional expectation of Y?

$$\begin{aligned}
\mathbf{E}\left[Y|X_k=x_k\right] &= \mathbf{E}\left[\mathbf{E}\left[Y|X_1,X_2,\ldots X_k,\ldots X_p\right]|X_k=x_k\right] &(2)\\
&= \mathbf{E}\left[\sum_{j=0}^{p}\beta_j X_j|X_k=x_k\right] &(3)\\
&= \beta_k x_k + \mathbf{E}\left[\sum_{j\neq k}\beta_j X_j|X_k=x_k\right] &(4)
\end{aligned}$$

where the first line uses the law of total expectation[1], and the second line uses Eq. 1. Turned around,

$$\begin{aligned} \beta_k x_k &= \mathbf{E}\left[Y|X_k = x_k\right] - \mathbf{E}\left[\sum_{j\neq k} \beta_j X_j | X_k = x_k\right] & (5) \\ &= \mathbf{E}\left[Y - \left(\sum_{j\neq k} \beta_j X_j\right) | X_k = x_k\right] & (6) \end{aligned}$$

The expression in the expectation is the k^{th} **partial residual** — the (total) residual is the difference between Y and its expectation, the partial residual is the difference between Y and what we expect it to be *ignoring* the contribution from X_k. Let's introduce a symbol for this, say $Y^{(k)}$.

$$\beta_k x_k = \mathbf{E}\left[Y^{(k)}|X_k = x_k\right] \quad (7)$$

In words, if the over-all model is linear, then the partial residuals are linear. And notice that X_k is the only input feature appearing here — if we could somehow get hold of the partial residuals, then we can find β_k by doing a simple regression, rather than a multiple regression. Of course to get the partial residual we need to know all the other β_js...

This suggests the following estimation scheme for linear models, known as the **Gauss-Seidel algorithm**, or more commonly and transparently as **backfitting**; the pseudo-code is in Example 1.

This is an iterative approximation algorithm. Initially, we look at how far each point is from the global mean, and do simple regressions of those deviations on the input features. This then gives us a better idea of what the regression surface really is, and we use the deviations from *that* surface in our next set of simple regressions. At each step, each coefficient is adjusted to fit in with what we already know about the other coefficients — that's why it's called "backfitting". It is not obvious[2] that this converges, but it does, and the fixed point on which it converges is the usual least-squares estimate of β.

Backfitting is not usually how we fit linear models, because with modern numerical linear algebra it's actually faster to just calculate $(\mathbf{x}^T\mathbf{x})^{-1}\mathbf{x}^T\mathbf{y}$. But the cute thing about backfitting is that it doesn't actually rely on the model being *linear*.

[1] As you learned in baby prob., this is the fact that $\mathbf{E}\left[Y|X\right] = \mathbf{E}\left[\mathbf{E}\left[Y|X,Z\right]|X\right]$ — that we can always condition on another variable or variables (Z), provided we then average over those extra variables when we're done.

[2] Unless, I suppose, you're Gauss.

Given: $n \times (p+1)$ inputs $\mathbf{x}$ (0^{th} column all 1s)
$n \times 1$ responses $\mathbf{y}$
tolerance $1 \gg \delta > 0$
center $\mathbf{y}$ and each column of $\mathbf{x}$
$\widehat{\beta}_j \leftarrow 0$ for $j \in 1:p$
`until` (all $|\widehat{\beta}_j - \gamma_j| \leq \delta$) {
`for` $k \in 1:p$ {
$y_i^{(k)} = y_i - \sum_{j \neq k} \widehat{\beta}_j x_{ij}$
$\gamma_k \leftarrow$ regression coefficient of $y^{(k)}$ on $x_{\cdot k}$
$\widehat{\beta}_k \leftarrow \gamma_k$
}
}
$\widehat{\beta}_0 \leftarrow \left(n^{-1}\sum_{i=1}^{n} y_i\right) - \sum_{j=1}^{p} \widehat{\beta}_j n^{-1} \sum_{i=1}^{n} x_{ij}$
Return: $(\widehat{\beta}_0, \widehat{\beta}_1, \ldots \widehat{\beta}_p)$

Code Example 1: Pseudocode for backfitting linear models. Assume we make at least one pass through the `until` loop. Recall from the handouts on linear models that centering the data does not change the β_j; this way the intercept only have to be calculated once, at the end.

2 Additive Models

The **additive model** for regression is

$$\mathbf{E}\left[Y|\vec{X} = \vec{x}\right] = \alpha + \sum_{j=1}^{p} f_j(x_j)$$

This includes the linear model as a special case, where $f_j(x_j) = \beta_j x_j$, but it's clearly more general, because the f_js can be pretty arbitrary nonlinear functions. The idea is still that each input feature makes a separate contribution to the response, and these just add up, but these contributions don't have to be strictly proportional to the inputs. We do need to add a restriction to make it identifiable; without loss of generality, say that $\mathbf{E}[Y] = \alpha$ and $\mathbf{E}[f_j(X_j)] = 0$.[3]

Additive models keep a lot of the nice properties of linear models, but are more flexible. One of the nice things about linear models is that they are fairly straightforward to interpret: if you want to know how the prediction changes as you change x_j, you just need to know β_j. The partial response function f_j plays the same role in an additive model: of course the change in prediction from changing x_j will generally depend on the level x_j had before perturbation, but since that's also true of reality that's really a feature rather than a bug. It's

[3]To see why we need to do this, imagine the simple case where $p = 2$. If we add constants c_1 to f_1 and c_2 to f_2, but subtract $c_1 + c_2$ from α, then nothing *observable* has changed about the model. This degeneracy or lack of identifiability is a little like the rotation problem for factor analysis, but less harmful because we really can fix it by the convention given above.

true that a set of plots for f_js takes more room than a table of β_js, but it's also nicer to look at. There are really very, very few situations when, with modern computing power, linear models are superior to additive models.

Now, one of the nice properties which additive models share with linear ones has to do with the partial residuals. Defining

$$Y^{(k)} = Y - \left(\alpha + \sum_{j \neq k} f_j(x_j) \right)$$

a little algebra along the lines of the last section shows that

$$\mathbf{E}\left[Y^{(k)} | X_k = x_k \right] = f_k(x_k)$$

If we knew how to estimate arbitrary one-dimensional regressions, we could now use backfitting to estimate additive models. But we have spent a lot of time talking about how to use smoothers to fit one-dimensional regressions!

Our new, improved backfitting algorithm in Example 2. Once again, while it's not obvious that this converges, it does converge. Also, the backfitting procedure works well with some complications or refinements of the additive model. If we know the function form of one or another of the f_j, we can fit those parametrically (rather than with the smoother) at the appropriate points in the loop. (This would be a **semiparametric** model.) If we think that there is an interaction between x_j and x_k, rather than their making separate additive contributions, we can smooth them together; etc.

You will write a simple back-fitting algorithm in the homework. In practice, however, it will be better to use existing packages, since their authors have already made, and fixed, all of the mistakes you will make. There are actually *two* packages standard packages for fitting additive models in R: `gam` and `mgcv`. Both have commands called `gam`, which fit **generalized** additive models — the generalization is to use the additive model for things like the probabilities of categorical responses, rather than the response variable itself. If that sounds obscure right now, don't worry — we'll come back to this when we look at advanced classification methods. The last section of this hand-out illustrates using these packages to fit an additive model.

3 The Curse of Dimensionality

Before illustrating how additive models work in practice, let's talk about why we'd want to use them. So far, we have looked at two extremes for regression models; additive models are somewhere in between.

On the one hand, we had linear regression, which is a parametric method (with $p + 1$) parameters. Its weakness is that the true regression function r is hardly ever linear, so even with infinite data it will always make systematic mistakes in its predictions — there's always some approximation bias, bigger or

Given: $n \times p$ inputs $\mathbf{x}$
$n \times 1$ responses $\mathbf{y}$
tolerance $1 \gg \delta > 0$
one-dimensional smoother $\mathcal{S}$
$\widehat{\alpha} \leftarrow n^{-1} \sum_{i=1}^{n} y_i$
$\widehat{f}_j \leftarrow 0$ for $j \in 1 : p$
`until` (all $|\widehat{f}_j - g_j| \leq \delta$) {
`for` $k \in 1 : p$ {
$y_i^{(k)} = y_i - \sum_{j \neq k} \widehat{f}_j(x_{ij})$
$g_k \leftarrow \mathcal{S}(y^{(k)} \sim x_{\cdot k})$
$g_k \leftarrow g_k - n^{-1} \sum_{i=1}^{n} g_k(x_{ik})$
$\widehat{f}_k \leftarrow g_k$
}
}
Return: $(\widehat{\alpha}, \widehat{f}_1, \ldots \widehat{f}_p)$

Code Example 2: Pseudo-code for backfitting additive models. Notice the extra step, as compared to backfitting linear models, which keeps each partial response function centered.

smaller depending on how non-linear r is. The strength of linear regression is that it converges very quickly as we get more data. Generally speaking,

$$MSE_{\text{linear}} = \sigma^2 + a_{\text{linear}} + O(n^{-1})$$

where the first term is the intrinsic noise around the true regression function, the second term is the (squared) approximation bias, and the last term is the estimation variance. Notice that the rate at which the estimation variance shrinks doesn't depend on p — factors like that are all absorbed into the big O. Other parametric models generally converge at the same rate.

At the other extreme, we've seen a number of completely non-parametric regression methods, such as kernel regression, local polynomials, k-nearest neighbors, etc. Here the limiting approximation bias is actually *zero*, at least for any reasonable regression function r. The problem is that they converge more slowly, because we need to use the data not just to figure out the coefficients of a parametric model, but the sheer shape of the regression function. Again generally speaking, the rate of convergence for these models is (Wasserman, 2006, §5.12)

$$MSE_{\text{nonpara}} - \sigma^2 = O(n^{-4/(p+4)})$$

There's no approximation bias term here, just estimation variance.[4] Why does the rate depend on p? Well, to give a very hand-wavy explanation, think of

[4] To be careful: if we use, say, kernel regression, then at any *finite* n and bandwidth there is some approximation bias, but this can be made arbitrarily small, and is actually absorbed into the remaining big-O.

the smoothing methods, where $\widehat{r}(\vec{x})$ is an average over y_i for $\vec{x}_i$ near $\vec{x}$. In a p dimensional space, the volume within ϵ of $\vec{x}$ is $O(\epsilon^p)$, so to get the same density (points per unit volume) around $\vec{x}$ takes exponentially more data as p grows. This doesn't explain where the 4s come from, but that's honestly too long and complicated a story for this class. (Take 10-702.)

For $p = 1$, the non-parametric rate is $O(n^{-4/5})$, which is of course slower than $O(n^{-1})$, but not all that much, and the improved bias usually more than makes up for it. But as p grows, the non-parametric rate gets slower and slower, and the fully non-parametric estimate more and more imprecise, yielding the infamous **curse of dimensionality**. For $p = 100$, say, we get a rate of $O(n^{-1/26})$, which is not very good at all. Said another way, to get the same precision with p inputs that n data points gives us with one input takes $n^{(4+p)/5}$ data points. For $p = 100$, this is $n^{20.8}$, which tells us that matching $n = 100$ requires $O(4 \times 10^{41})$ observations.

So completely unstructured non-parametric regressions won't work very well in high dimensions, at least not with plausible amounts of data. The trouble is that there are just *too many* possible high-dimensional surfaces, and seeing only a million or a trillion points from the surface doesn't pin down its shape very well at all.

This is where additive models come in. Not every regression function is additive, so they have, even asymptotically, some approximation bias. But we can estimate each f_j by a simple one-dimensional smoothing, which converges at $O(n^{-4/5})$, almost as good as the parametric rate. So overall

$$MSE_{\text{additive}} - \sigma^2 = a_{\text{additive}} + O(n^{-4/5})$$

Since linear models are a sub-class of additive models, $a_{\text{additive}} \leq a_{\text{lm}}$. From a purely predictive point of view, the only time to prefer linear models to additive models is when n is so small that $O(n^{-4/5}) - O(n^{-1})$ exceeds this difference in approximation biases; eventually the additive model will be more accurate.[5]

[5] Unless the best additive approximation to r really is linear; then the linear model has no more bias and better variance.

4 Example: California House Prices Revisited

As an example, we'll revisit the California house price data from the homework.

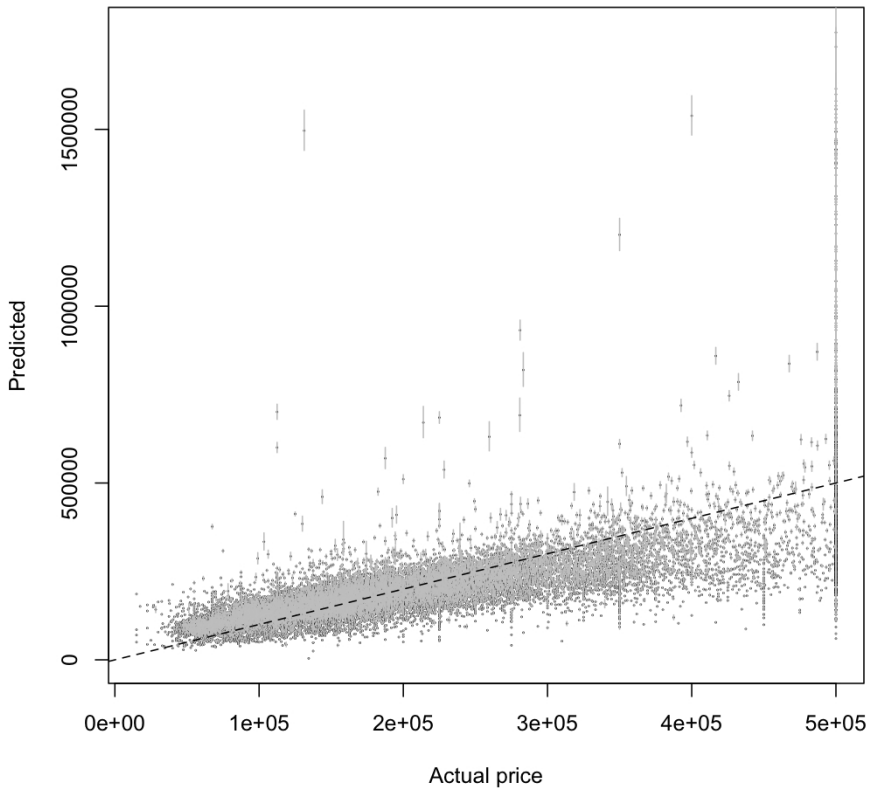
```
calif = read.table("~/teaching/350/hw/06/cadata.dat",header=TRUE)
```

Fitting a linear model is very fast (about 1/5 of a second on my laptop). Here are the summary statistics:

```
> linfit = lm(log(MedianHouseValue) ~ ., data=calif)
> print(summary(linfit),signif.stars=FALSE)

Call:
lm(formula = log(MedianHouseValue) ~ ., data = calif)

Residuals:
      Min        1Q    Median        3Q       Max
-2.517974 -0.203797  0.001589  0.194947  3.464100

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)    -1.180e+01  3.059e-01 -38.570  < 2e-16
MedianIncome    1.782e-01  1.639e-03 108.753  < 2e-16
MedianHouseAge  3.261e-03  2.111e-04  15.446  < 2e-16
TotalRooms     -3.186e-05  3.855e-06  -8.265  < 2e-16
TotalBedrooms   4.798e-04  3.375e-05  14.215  < 2e-16
Population     -1.725e-04  5.277e-06 -32.687  < 2e-16
Households      2.493e-04  3.675e-05   6.783 1.21e-11
Latitude       -2.801e-01  3.293e-03 -85.078  < 2e-16
Longitude      -2.762e-01  3.487e-03 -79.212  < 2e-16

Residual standard error: 0.34 on 20631 degrees of freedom
Multiple R-squared: 0.6432,Adjusted R-squared: 0.643
F-statistic:  4648 on 8 and 20631 DF,  p-value: < 2.2e-16
```

Figure 1 plots the predicted prices, ± 2 standard errors, against the actual prices. The predictions are not all that accurate — the RMS residual is 0.340 on the log scale (i.e., 41%), and only 3.3% of the actual prices fall within the prediction bands.[6] On the other hand, they *are* quite precise, with an RMS standard error of 0.0071 (i.e., 0.71%). This linear model is pretty thoroughly converged.

[6] You might worry that the top-coding of the prices — all values over $500,000 are recorded as $500,001 — means we're not being fair to the model. After all, we see $500,001 and the model predicts $600,000, the prediction might be right — it's certainly right that it's over $500,000. To deal with this, I tried top-coding the predicted values, but it didn't change much — the RMS error for the linear model only went down to 0.332, and it was similarly inconsequential for the others. Presumably this is because only about 5% of the records are top-coded.




```
predictions = predict(linfit,se.fit=TRUE)
plot(calif$MedianHouseValue,exp(predictions$fit),cex=0.1,
     xlab="Actual price",ylab="Predicted")
segments(calif$MedianHouseValue,exp(predictions$fit-2*predictions$se.fit),
         calif$MedianHouseValue,exp(predictions$fit+2*predictions$se.fit),
         col="grey")
abline(a=0,b=1,lty=2)
```

Figure 1: Actual median house values (horizontal axis) versus those predicted by the linear model (black dots), plus or minus two standard errors (grey bars). The dashed line shows where actual and predicted prices would be equal. Note that I've exponentiated the predictions so that they're comparable to the original values.

Next, we'll fit an additive model, using the `gam` function from the `mgcv` package; this automatically sets the bandwidths using a fast approximation to leave-one-out CV called **generalized cross-validation**, or **GCV**.

```
> require(mgcv)
> system.time(addfit <- gam(log(MedianHouseValue) ~ s(MedianIncome)
                           + s(MedianHouseAge) + s(TotalRooms)
                           + s(TotalBedrooms) + s(Population) + s(Households)
                           + s(Latitude) + s(Longitude), data=calif))
   user  system elapsed
 41.144   1.929  44.487
```

(That is, it took almost a minute in total to run this.) The `s()` terms in the `gam` formula indicate which terms are to be smoothed — if we wanted particular parametric forms for some variables, we could do that as well. (Unfortunately `MedianHouseValue ~ s(.)` will not work.) The smoothing here is done by splines, and there are lots of options for controlling the splines, if you know what you're doing.

Figure 2 compares the predicted to the actual responses. The RMS error has improved (0.29 on the log scale, or 33%, with 9.5% of observations falling with ± 2 standard errors of their fitted values), at only a fairly modest cost in precision (the RMS standard error of prediction is 0.016, or 1.6%). Figure 3 shows the partial response functions.

It seems silly to have latitude and longitude make separate additive contributions here; presumably they interact. We can just smooth them together[7]:

```
addfit2 <- gam(log(MedianHouseValue) ~ s(MedianIncome) + s(MedianHouseAge)
               + s(TotalRooms) +s(TotalBedrooms) + s(Population) + s(Households)
               + s(Longitude,Latitude), data=calif)
```

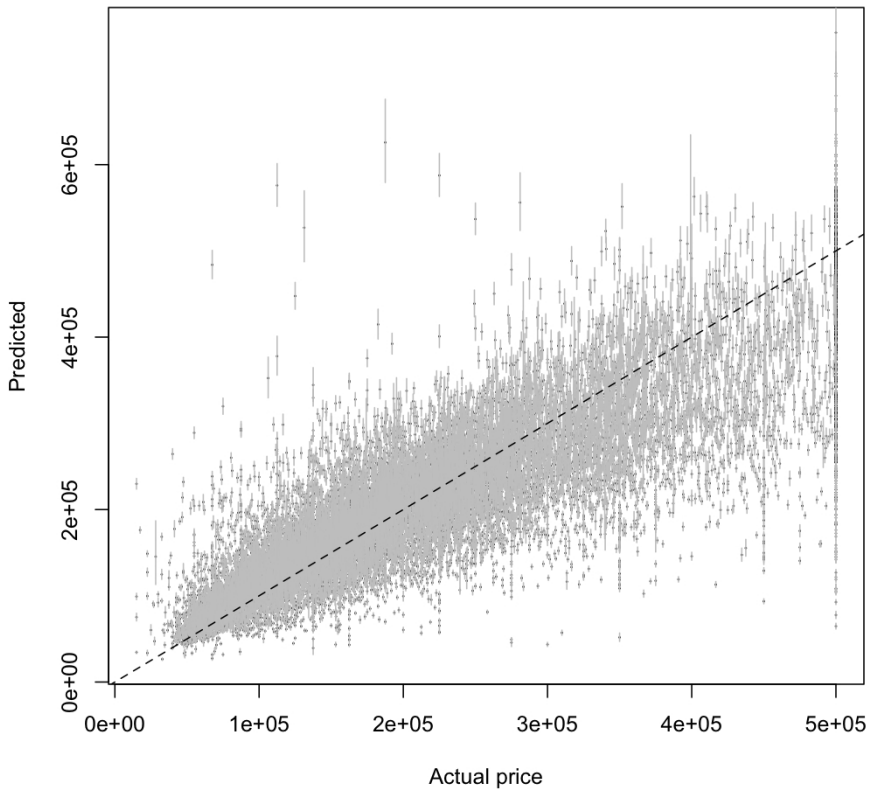
This gives an RMS error of $\pm 31\%$ (with 11% coverage), with no decrease in the precision of the predictions (at least to two figures).

Figures 5 and 6 show two different views of the joint smoothing of longitude and latitude. In the perspective plot, it's quite clear that price increases specifically towards the coast, and even more specifically towards the great coastal cities. In the contour plot, one sees more clearly an inward bulge of a negative, but not too very negative, contour line (between -122 and -120 longitude) which embraces Napa, Sacramento, and some related areas, which are comparatively more developed and more expensive than the rest of central California, and so more expensive than one would expect based on their distance from the coast and San Francisco.

The fact that the prediction intervals have such bad coverage is partly due to their being based on Gaussian approximations. Still, ± 2 standard errors should cover at least 25% of observations[8], which is manifestly failing here. This

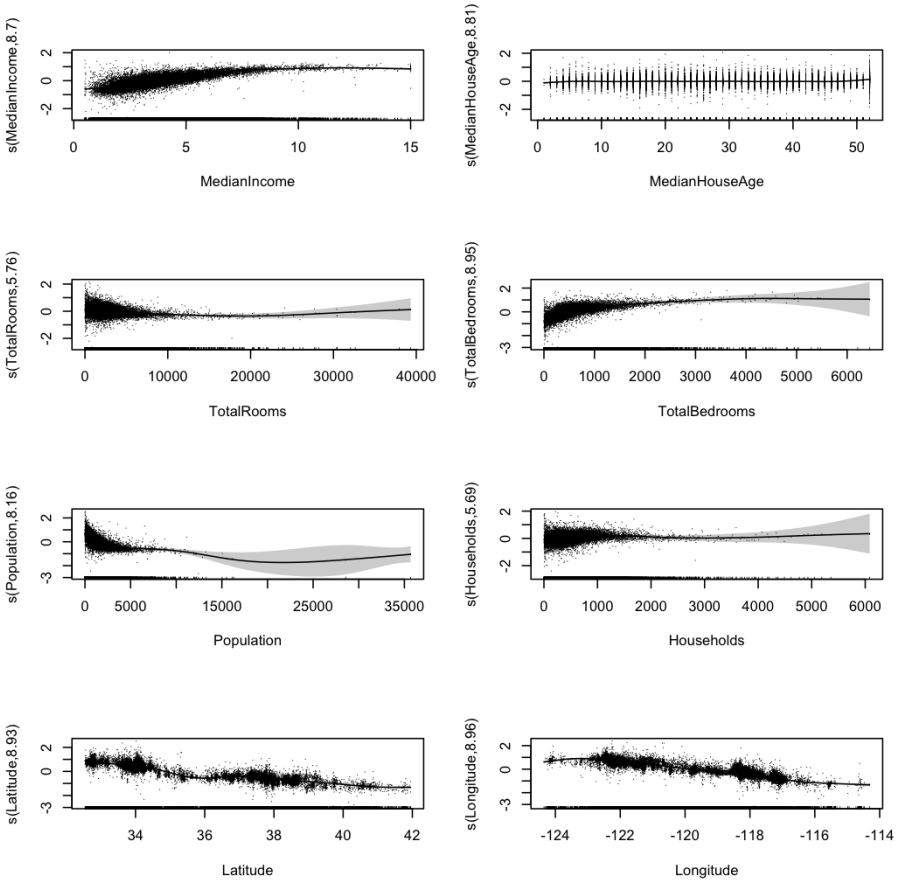
[7] If the two variables which interact are on very different scales, it's better to smooth them with a `te()` term than an `s()` term — see `help(gam.models)` — but here they are comparable.

[8] By Chebyshev's inequality: $\mathbb{P}(|X - \mathbf{E}[X]| \geq a\sigma) \leq 1/a^2$, where σ is the standard deviation of X.



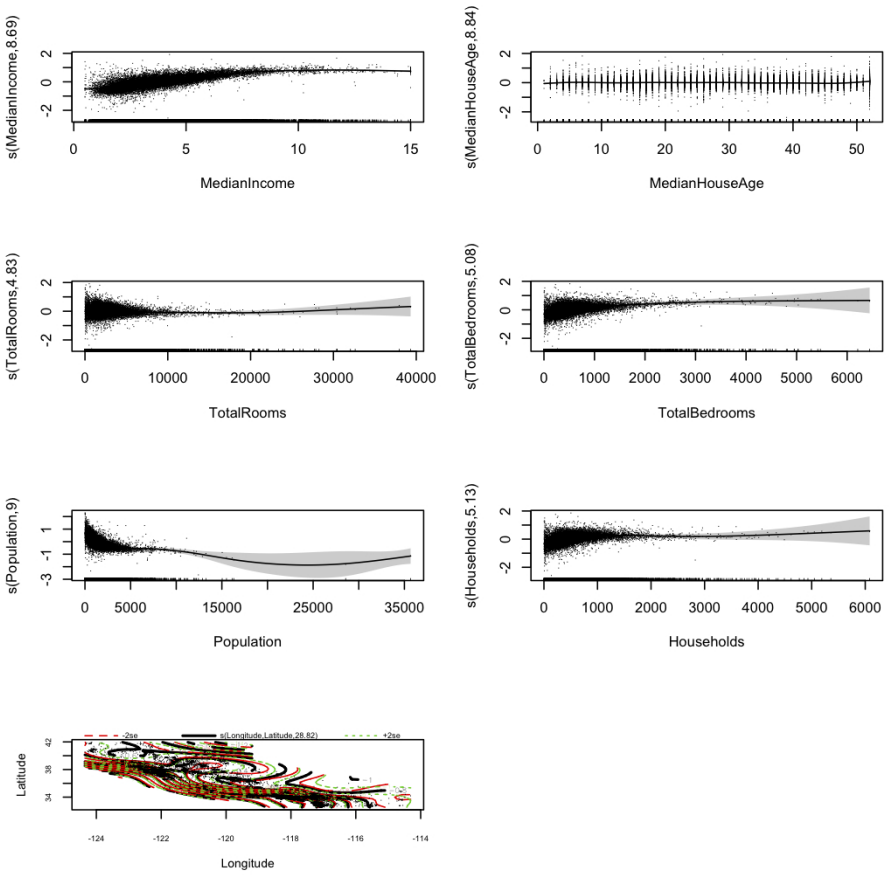

```
predictions = predict(addfit,se.fit=TRUE)
plot(calif$MedianHouseValue,exp(predictions$fit),cex=0.1,
     xlab="Actual price",ylab="Predicted")
segments(calif$MedianHouseValue,exp(predictions$fit-2*predictions$se.fit),
         calif$MedianHouseValue,exp(predictions$fit+2*predictions$se.fit),
         col="grey")
abline(a=0,b=1,lty=2)
```

Figure 2: Actual versus predicted prices for the additive model, as in Figure 1.



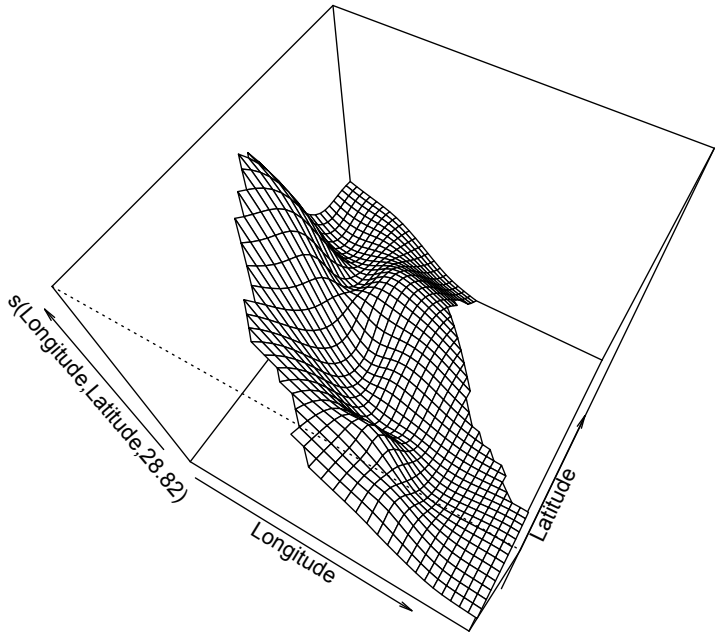

```
plot(addfit,scale=0,se=2,shade=TRUE,resid=TRUE,pages=1)
```

Figure 3: The estimated partial response functions for the additive model, with a shaded region showing ± 2 standard errors, and dots for the actual partial residuals. The tick marks along the horizontal axis show the observed values of the input variables (a **rug plot**); note that the error bars are wider where there are fewer observations. Setting `pages=0` (the default) would produce eight separate plots, with the user prompted to cycle through them. Setting `scale=0` gives each plot its own vertical scale; the default is to force them to share the same one. Finally, note that here the vertical scale is logarithmic.



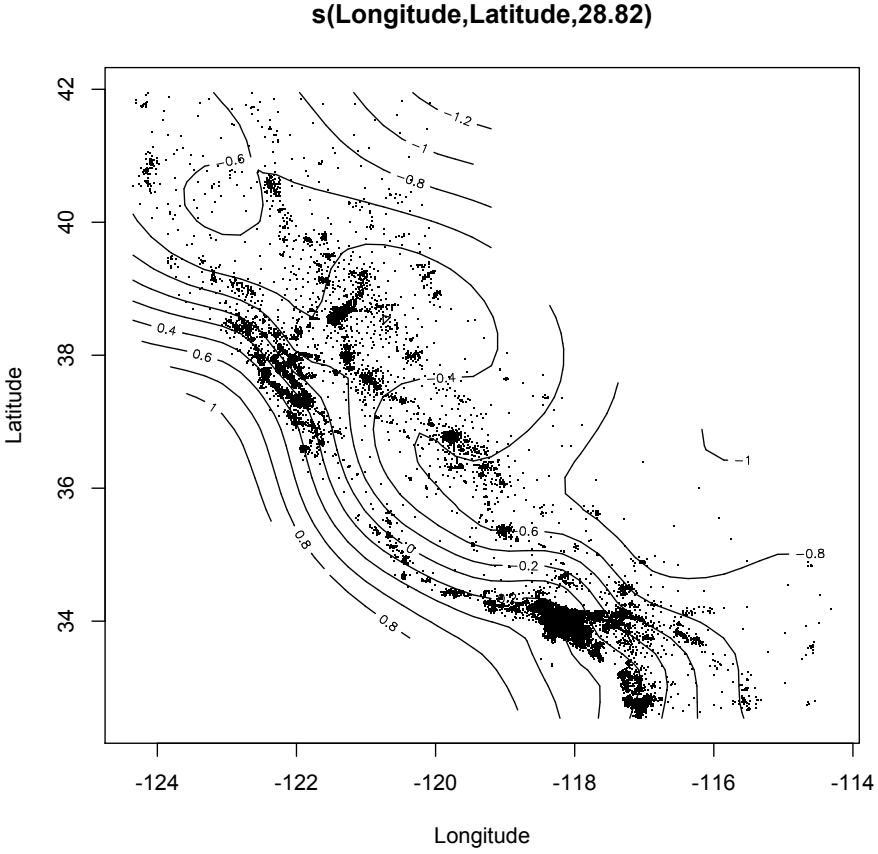

```
plot(addfit2,scale=0,se=2,shade=TRUE,resid=TRUE,pages=1)
```

Figure 4: Partial response functions and partial residuals for `addfit2`, as in Figure 3. See subsequent figures for the joint smoothing of longitude and latitude, which here is an illegible mess.




```
plot(addfit2,select=7,phi=60,pers=TRUE)
```

Figure 5: The result of the joint smoothing of longitude and latitude.




```
plot(addfit2,select=7,se=FALSE)
```

Figure 6: The result of the joint smoothing of longitude and latitude. Setting `se=TRUE`, the default, adds standard errors for the contour lines in multiple colors. Again, note that these are log units.

suggests substantial remaining bias. One of the standard strategies for trying to reduce such bias is to allow more interactions. We will see automatic ways of doing this, in later lectures, where we can still get some sort of interpretation of results.

We could, of course, just use a completely unrestricted nonparametric regression — going to the opposite extreme from the linear model. I'll use `npreg` from the `np` package to fit a Nadaraya-Watson regression, using its built-in function `npregbw` to pick the bandwidths.[9]

```
library(np)
system.time(calif.bw <- npregbw(log(MedianHouseValue) ~., data=calif,type="ll"))
```

[[This is still running after 10 hours of processor time; I will update these notes later when it finishes.]]

References

Wasserman, Larry (2006). *All of Nonparametric Statistics*. Berlin: Springer-Verlag.

[9] `npregbw` takes formulas with the same syntax as `lm`. You can specify the kernel type, but the default for continuous data is Gaussian, which is fine here. You can also specify whether to do a local constant/Nadaraya-Watson regression (the default), or a local linear fit. While we haven't gone into it, a detailed examination of the approximations made in the two methods shows that local constant regressions have bigger finite-sample biases at the boundaries of the input region than do local linear regressions. (See Wasserman (2006, ch. 5).) I've been deliberately ignoring this, because usually there aren't many observations near the boundaries and this is a secondary issue. The California data, however, has a lot of top-coding, and so a lot of on-boundary observations.

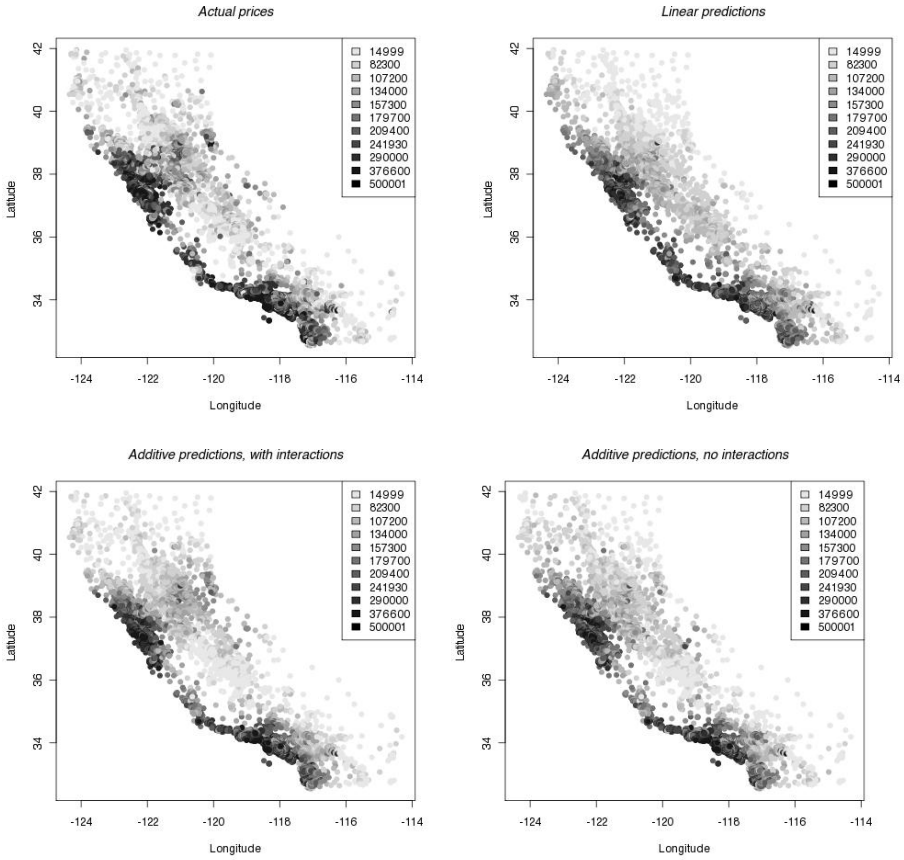


Figure 7: Maps of real or fitted prices: actual, top left; linear model, top right; first additive model, bottom right; additive model with interaction, bottom left. Categories are deciles of the actual prices; since those are the same for all four plots, it would have been nicer to make one larger legend, but that was beyond my graphical abilities.