Chapter 9

Additive Models

9.1 Partial Residuals and Back-fitting for Linear Models

The general form of a linear regression model is

$$\mathbf{E}\left[Y|\vec{X}=\vec{x}\right] = \beta_0 + \vec{\beta}\cdot\vec{x} = \sum_{j=0}^{p}\beta_j x_j \tag{9.1}$$

where for $j \in 1:p$, the x_j are the components of $\vec{x}$, and x_0 is always the constant 1. (Adding this fictitious constant variable lets us handle 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?

$$\begin{aligned}
\mathbf{E}\left[Y|X_k=x_k\right] &= \mathbf{E}\left[\mathbf{E}\left[Y|X_1,X_2,\ldots X_k,\ldots X_p\right]|X_k=x_k\right] && (9.2)\\
&= \mathbf{E}\left[\sum_{j=0}^{p}\beta_j X_j|X_k=x_k\right] && (9.3)\\
&= \beta_k x_k + \mathbf{E}\left[\sum_{j\neq k}\beta_j X_j|X_k=x_k\right] && (9.4)
\end{aligned}$$

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

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

9.1. Turned around,

$$\begin{aligned}
\beta_k x_k &= \mathbf{E}\left[Y|X_k = x_k\right] - \mathbf{E}\left[\sum_{j\neq k}\beta_j X_j|X_k = x_k\right] && (9.5)\\
&= \mathbf{E}\left[Y - \left(\sum_{j\neq k}\beta_j X_j\right)|X_k = x_k\right] && (9.6)
\end{aligned}$$

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 $Y^{(k)}$.

$$\beta_k x_k = \mathbf{E}\left[Y^{(k)}|X_k = x_k\right] \qquad (9.7)$$

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 **back-fitting**; the pseudo-code is in Example 24.

This is an iterative approximation algorithm. Initially, we look at how far each point is from the global mean, and do a simple regression of those deviations on the first input variable. This then gives us a better idea of what the regression surface really is, and we use the deviations from *that* surface in a simple regression on the next variable; this should catch relations between Y and X_2 that weren't already caught by regressing on X_1. We then go on to the next variable in turn. At each step, each coefficient is adjusted to fit in with what we have already guessed about the other coefficients — that's why it's called "back-fitting". 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.

Back-fitting is not usually how we fit linear models any more, 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 back-fitting is that it doesn't actually rely on the model being *linear*.

9.2 Additive Models

The **additive model** for regression is

$$\mathbf{E}\left[Y|\vec{X} = \vec{x}\right] = \alpha + \sum_{j=1}^{p} f_j(x_j) \qquad (9.8)$$

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

```
Given: $n \times (p+1)$ inputs $\mathbf{x}$ ($0^{\text{th}}$ column all 1s)
        $n \times 1$ responses $\mathbf{y}$
        small tolerance $\delta > 0$
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 24: Pseudocode for back-fitting linear models. Assume we make at least one pass through the `until` loop. Recall from Chapter 1 that centering the data does not change the β_j; this way the intercept only have to be calculated once, at the end.

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 $\mathbf{E}\left[f_j(X_j)\right] = 0$.[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 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, conveys more information, and imposes fewer systematic distortions on the data.

Now, one of the nice properties which additive models share with linear ones has

[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 way collinearity keeps us from defining true slopes in linear regression. But it's less harmful than collinearity because we really can fix it by the convention given above.

```
Given: $n \times p$ inputs $\mathbf{x}$
        $n \times 1$ responses $\mathbf{y}$
        small tolerance $\delta > 0$
        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 25: Pseudo-code for back-fitting additive models. Notice the extra step, as compared to back-fitting linear models, which keeps each partial response function centered.

to do with the partial residuals. Defining

$$Y^{(k)} = Y - \left(\alpha + \sum_{j \neq k} f_j(x_j)\right) \tag{9.9}$$

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

$$\mathbf{E}\left[Y^{(k)}|X_k = x_k\right] = f_k(x_k) \tag{9.10}$$

If we knew how to estimate arbitrary one-dimensional regressions, we could now use back-fitting to estimate additive models. But we have spent a lot of time talking about how to use smoothers to fit one-dimensional regressions! We could use nearest neighbors, or splines, or kernels, or local-linear regression, or anything else we feel like substituting here.

Our new, improved back-fitting algorithm in Example 25. Once again, while it's not obvious that this converges, it does converge. Also, the back-fitting 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.

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 in Chapters 12–13 after we've looked at generalized linear models. The last section of this chapter illustrates using these packages to fit an additive model.

9.3 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 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,

$$MSE_{\text{linear}} = \sigma^2 + a_{\text{linear}} + O(n^{-1}) \tag{9.11}$$

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.[4] 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. We saw in Chapter 4 that the mean-squared error of kernel regression in one dimension is $\sigma^2 + O(n^{-4/5})$. Splines, k-nearest-neighbors (with growing k), etc., all attain the same rate. But in p dimensions, this becomes (Wasserman, 2006, §5.12)

$$MSE_{\text{nonpara}} - \sigma^2 = O(n^{-4/(p+4)}) \tag{9.12}$$

There's no ultimate approximation bias term here. Why does the rate depend on p? Well, to give a very hand-wavy explanation, think of the smoothing methods, where $\hat{r}(\vec{x})$ 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. The appearance of the 4s is a little more

[4]See Appendix B you are not familiar with "big O" notation.

mysterious, but can be resolved from an error analysis of the kind we did for kernel density estimation in Chapter 4[5].

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 the error of $n = 100$ one-dimensional observations requires $O(4 \times 10^{41})$ hundred-dimensional 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 functions, and seeing only a trillion points from the function 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

$$MSE_{\text{additive}} - \sigma^2 = a_{\text{additive}} + O(n^{-4/5}) \tag{9.13}$$

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.[6]

[5]More exactly, remember that in one dimension, the bias of a kernel smoother with bandwidth h is $O(h^2)$, and the variance is $O(1/nh)$, because only samples falling in an interval about h across contribute to the prediction at any one point, and when h is small, the number of such samples is proportional to nh. Adding bias squared to variance gives an error of $O(h^4) + O(1/nh)$, solving for the best bandwidth gives $h_{\text{opt}} = O(n^{-1/5})$, and the total error is then $O(n^{-4/5})$. Suppose for the moment that in p dimensions we use the same bandwidth along each dimension. (We get the same end result with more work if we let each dimension have its own bandwidth.) The bias is still $O(h^2)$, because the Taylor expansion still goes through. But now only samples falling into a region of volume $O(h^d)$ around x contribute to the prediction at x, so the variance is $O(1/nh^d)$. The best bandwidth is now $h_{\text{opt}} = O(n^{-1/(p+4)})$, yielding an error of $O(n^{-4/(p+4)})$ as promised.

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

9.4 Example: California House Prices Revisited

As an example, we'll revisit the housing price data from the homework. This has both California and Pennsylvania, but it's hard to visually see patterns with both states; I'll do California, and let you replicate this all on Pennsylvania, and even on the combined data.

Start with getting the data:

```
housing <- na.omit(read.csv("http://www.stat.cmu.edu/~cshalizi/uADA/13/hw/01/calif_penn_2011.csv"
calif <- housing[housing$STATEFP==6,]
```

(How do I know that the `STATEFP` code of 6 corresponds to California?)

We'll fit a linear model for the log price, on the thought that it makes some sense for the factors which raise or lower house values to multiply together, rather than just adding.

```
calif.lm <- lm(log(Median_house_value) ~ Median_household_income
  + Mean_househould_income + POPULATION + Total_units + Vacant_units + Owners
  + Median_rooms + Mean_household_size_owners + Mean_household_size_renters
  + LATITUDE + LONGITUDE, data = calif)
```

This is very fast — about a fifth of a second on my laptop.

Here are the summary statistics[7]:

```
> print(summary(calif.lm),signif.stars=FALSE,digits=3)

Call:
lm(formula = log(Median_house_value) ~ Median_household_income +
    + Mean_househould_income + POPULATION + Total_units + Vacant_units +
    Owners + Median_rooms + Mean_household_size_owners
    + Mean_household_size_renters + LATITUDE + LONGITUDE, data = calif)

Residuals:
   Min     1Q Median     3Q    Max
-3.855 -0.153  0.034  0.189  1.214

Coefficients:
                             Estimate Std. Error t value Pr(>|t|)
(Intercept)                 -5.74e+00   5.28e-01  -10.86  < 2e-16
Median_household_income      1.34e-06   4.63e-07    2.90   0.0038
Mean_househould_income       1.07e-05   3.88e-07   27.71  < 2e-16
POPULATION                  -4.15e-05   5.03e-06   -8.27  < 2e-16
Total_units                  8.37e-05   1.55e-05    5.41  6.4e-08
Vacant_units                 8.37e-07   2.37e-05    0.04   0.9719
Owners                      -3.98e-03   3.21e-04  -12.41  < 2e-16
```

[7]I have suppressed the usual stars on "significant" regression coefficients, because, as discussed in Chapter 2, those are not, in fact, the most important variables, and I have reined in R's tendency to use far too many decimal places.

```
predlims <- function(preds,sigma) {
  prediction.sd <- sqrt(preds$se.fit^2+sigma^2)
  upper <- preds$fit+2*prediction.sd
  lower <- preds$fit-2*prediction.sd
  lims <- cbind(lower=lower,upper=upper)
  return(lims)
}
```

Code Example 26: Function for calculating quick-and-dirty prediction limits from a prediction object (preds) containing fitted values and their standard errors, and an estimate of the over-all noise level. Because those are two (independent) sources of noise, we need to combine the standard deviations by "adding in quadrature".

```
Median_rooms                -1.62e-02   8.37e-03   -1.94   0.0525
Mean_household_size_owners   5.60e-02   7.16e-03    7.83  5.8e-15
Mean_household_size_renters -7.47e-02   6.38e-03  -11.71  < 2e-16
LATITUDE                    -2.14e-01   5.66e-03  -37.76  < 2e-16
LONGITUDE                   -2.15e-01   5.94e-03  -36.15  < 2e-16

Residual standard error: 0.317 on 7469 degrees of freedom
Multiple R-squared: 0.639,Adjusted R-squared: 0.638
F-statistic: 1.2e+03 on 11 and 7469 DF,  p-value: <2e-16
```

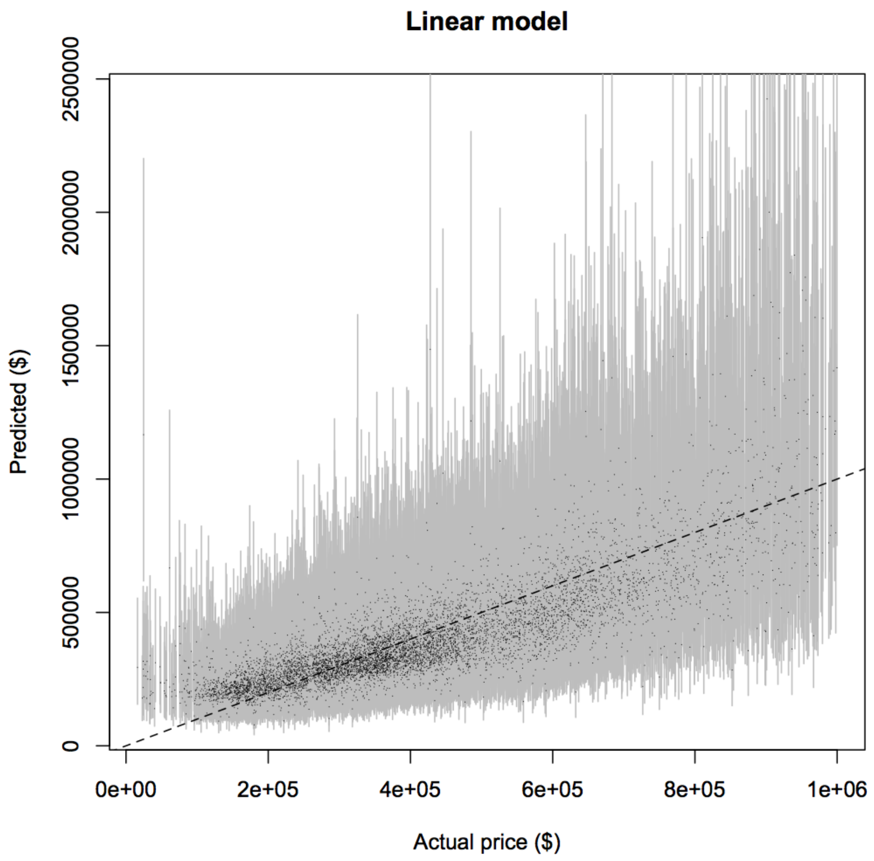
Figure 9.1 plots the predicted prices, ±2 standard errors, against the actual prices. The predictions are not all that *accurate* — the RMS residual is 0.317 on the log scale (i.e., 37%), but they do have pretty reasonable coverage; about 96% of actual precise fall within the prediction limits[8]

```
sqrt(mean(residuals(calif.lm)^2))
mean((log(calif$Median_house_value) <= predlims.lm[,"upper"])
  & (log(calif$Median_house_value) >= predlims.lm[,"lower"])
```

On the other hand, the predictions *are* quite precise, with the median of the calculated standard errors being 0.011 (i.e., 1.1%). This linear model *thinks* it knows what's going on.

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**.

[8]Remember from your linear regression class that there are two kinds of confidence intervals we might want to use for prediction. One is a confidence interval for the *conditional mean* at a given value of x; the other is a confidence interval for the *realized values of Y* at a given x. Earlier examples (and homework) have emphasized the former, but since we don't know the true conditional means here, we need to use the latter sort of intervals, prediction intervals proper, to evaluate coverage. The `predlims` function in Figure 9.1 calculates a rough prediction interval by taking the standard error of the conditional mean, combining it with the estimated standard deviation, and multiplying by 2. Strictly speaking, we ought to worry about using a t-distribution rather than a Gaussian here, but with 7469 residual degrees of freedom, this isn't going to matter much. (Assuming Gaussian noise is likely to be more of a concern, but this is only meant to be a rough cut anyway.)



```
preds.lm <- predict(calif.lm,se.fit=TRUE)
predlims.lm <- predlims(preds.lm,sigma=summary(calif.lm)$sigma)
plot(calif$Median_house_value,exp(preds.lm$fit),type="n",
  xlab="Actual price ($)",ylab="Predicted ($)", main="Linear model")
segments(calif$Median_house_value,exp(predlims.lm[,"lower"]),
  calif$Median_house_value,exp(predlims.lm[,"upper"]), col="grey")
abline(a=0,b=1,lty="dashed")
points(calif$Median_house_value,exp(preds.lm$fit),pch=16,cex=0.1)
```

Figure 9.1: Actual median house values (horizontal axis) versus those predicted by the linear model (black dots), plus or minus two *predictive* standard errors (grey bars). The dashed line shows where actual and predicted prices would be equal. Here `predict` gives both a fitted value for each point, and a standard error for that prediction. (There is no `newdata` argument in this call to `predict`, so it defaults to the training data used to learn `calif.lm`, which in this case is what we want.) I've exponentiated the predictions so that they're comparable to the original values (and because it's easier to grasp dollars than log-dollars).

```
require(mgcv)
system.time(calif.gam <- gam(log(Median_house_value)
  ~ s(Median_household_income) + s(Mean_househould_income) + s(POPULATION)
  + s(Total_units) + s(Vacant_units) + s(Owners) + s(Median_rooms)
  + s(Mean_household_size_owners) + s(Mean_household_size_renters)
  + s(LATITUDE) + s(LONGITUDE), data=calif))
   user  system elapsed
  5.481   0.441  17.700
```

(That is, it took almost eighteen seconds 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 we can't just write `MedianHouseValue` $\sim$ `s(.)`, we have to list all the variables on the right-hand side.) 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 9.2 compares the predicted to the actual responses. The RMS error has improved (0.27 on the log scale, or 30%, with 96% of observations falling with ± 2 standard errors of their fitted values), at only a fairly modest cost in the claimed precision (the RMS standard error of prediction is 0.020, or 2.0%). Figure 9.3 shows the partial response functions.

It makes little sense to have latitude and longitude make separate additive contributions here; presumably they interact. We can just smooth them together[9]:

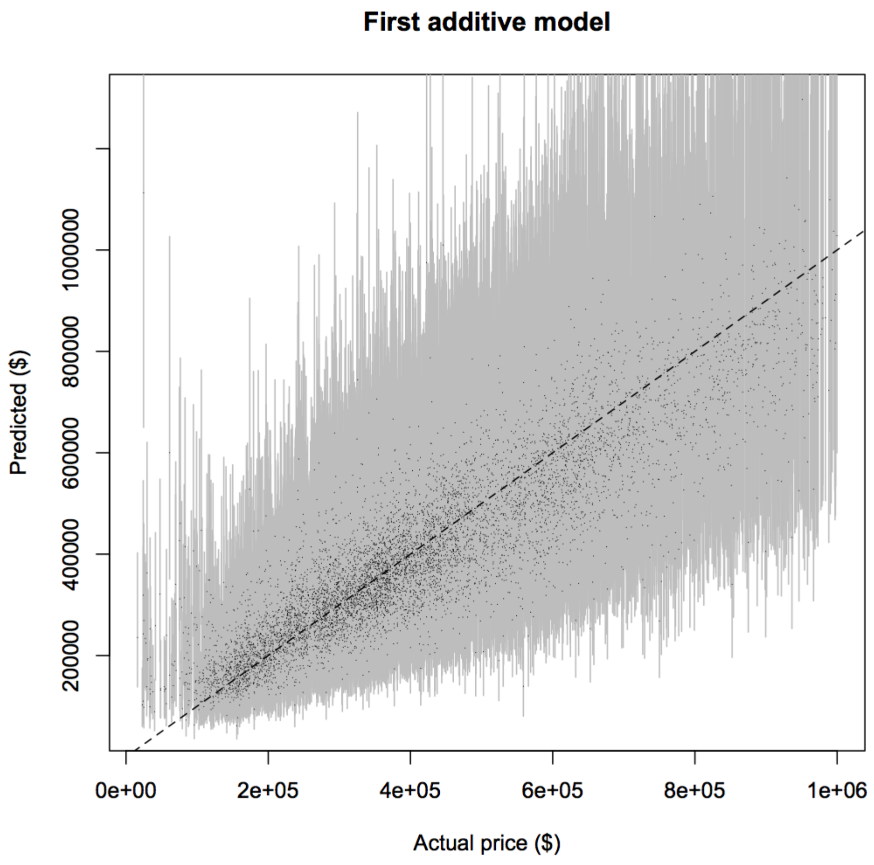
```
calif.gam2 <- gam(log(Median_house_value)
  ~ s(Median_household_income) + s(Mean_househould_income) + s(POPULATION)
  + s(Total_units) + s(Vacant_units) + s(Owners) + s(Median_rooms)
  + s(Mean_household_size_owners) + s(Mean_household_size_renters)
  + s(LONGITUDE,LATITUDE), data=calif)
```

This gives an RMS error of ± 0.25 (log-scale) and 96% coverage, with a median standard error of 0.021, so accuracy is improving (at least in sample), with little loss of precision.

```
preds.gam2 <- predict(calif.gam2,se.fit=TRUE)
predlims.gam2 <- predlims(preds.gam2,sigma=sqrt(calif.gam2$sig2))
mean((log(calif$Median_house_value) <= predlims.gam2[,"upper"]) &
  (log(calif$Median_house_value) >= predlims.gam2[,"lower"]))
```

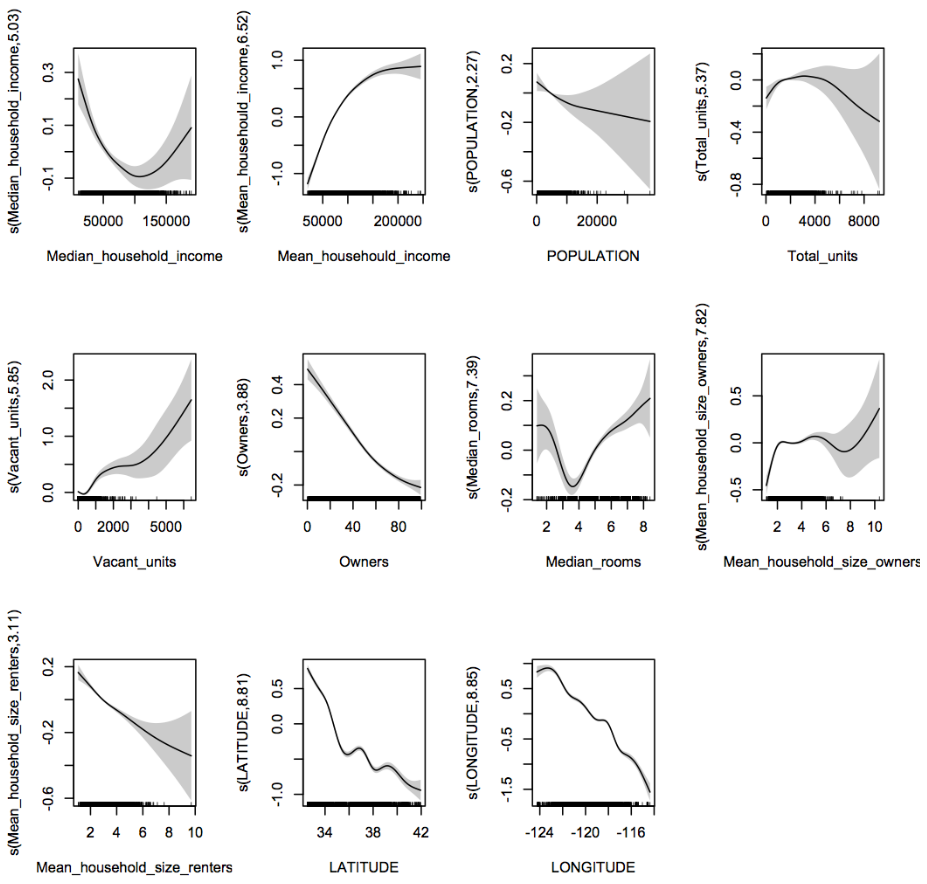
Figures 9.5 and 9.6 show two different views of the joint smoothing of longitude and latitude. In the perspective plot, it's quite clear that price increases specifically towards the coast, and even more specifically towards the great coastal cities. In the contour plot, one sees more clearly an inward 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.

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



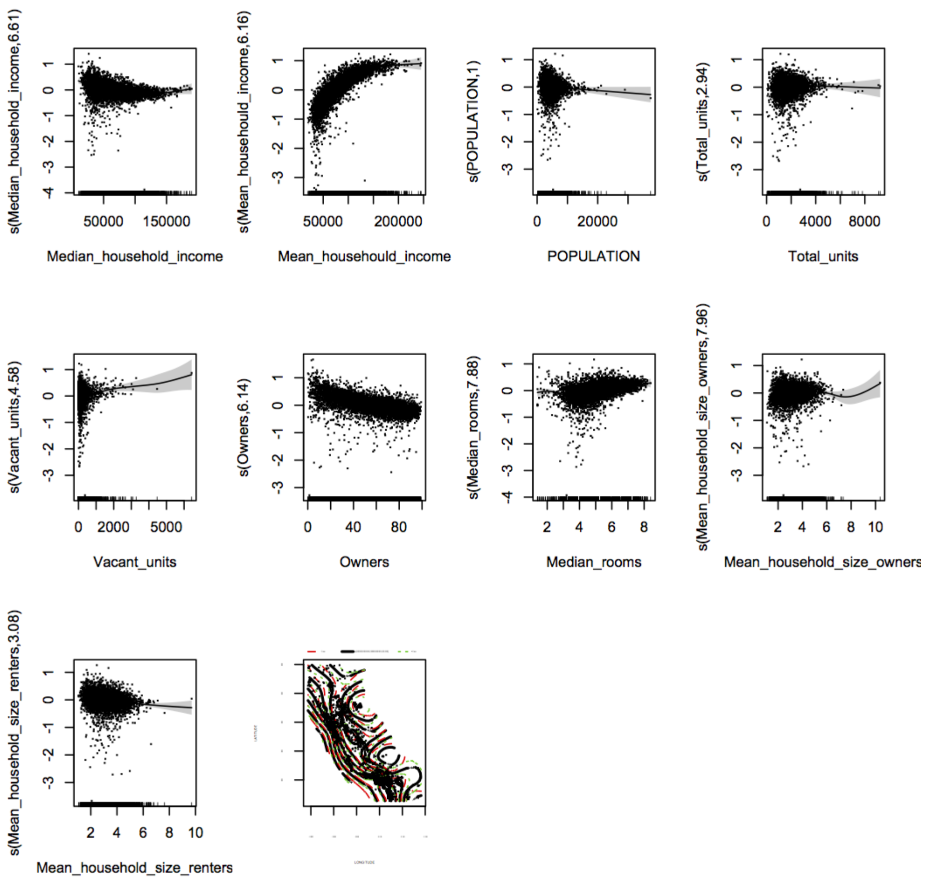
```
preds.gam <- predict(calif.gam,se.fit=TRUE)
predlims.gam <- predlims(preds.gam,sigma=sqrt(calif.gam$sig2))
plot(calif$Median_house_value,exp(preds.gam$fit),type="n",
  xlab="Actual price ($)",ylab="Predicted ($)", main="First additive model")
segments(calif$Median_house_value,exp(predlims.gam[,"lower"]),
  calif$Median_house_value,exp(predlims.gam[,"upper"]), col="grey")
abline(a=0,b=1,lty="dashed")
points(calif$Median_house_value,exp(preds.gam$fit),pch=16,cex=0.1)
```

Figure 9.2: Actual versus predicted prices for the additive model, as in Figure 9.1. Note that the `sig2` attribute of a model returned by `gam()` is the estimate of the noise around the regression surface (σ^2).



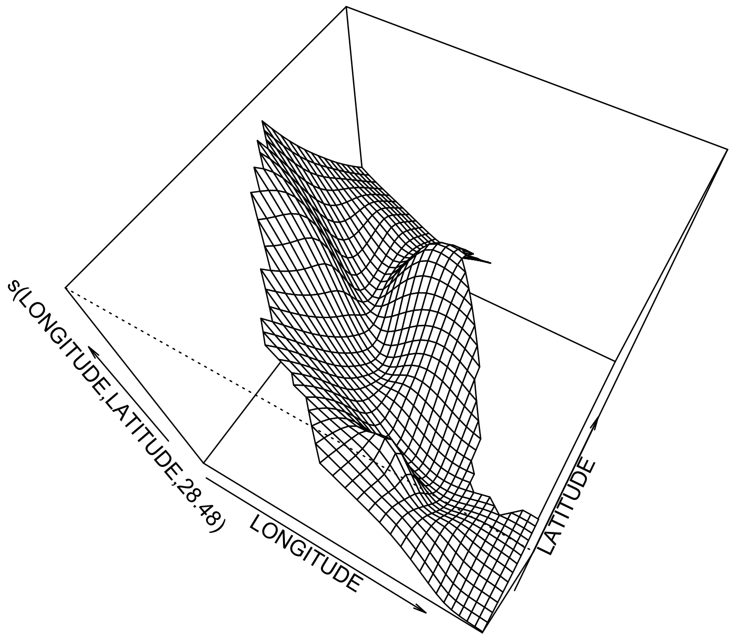
```
plot(calif.gam,scale=0,se=2,shade=TRUE,pages=1)
```

Figure 9.3: The estimated partial response functions for the additive model, with a shaded region showing ±2 standard errors. 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 scales are logarithmic.



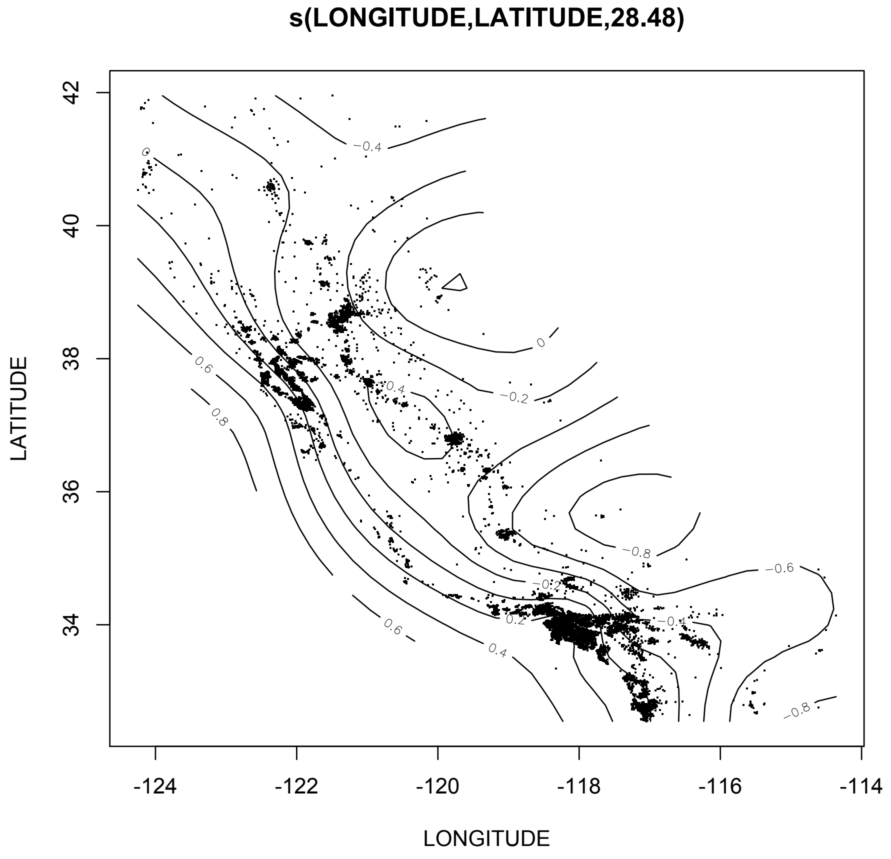
```
plot(calif.gam2,scale=0,se=2,shade=TRUE,resid=TRUE,pages=1)
```

Figure 9.4: Partial response functions and partial residuals for `addfit2`, as in Figure 9.3. See subsequent figures for the joint smoothing of longitude and latitude, which here is an illegible mess. See `help(plot.gam)` for the plotting options used here.



```
plot(calif.gam2,select=10,phi=60,pers=TRUE)
```

Figure 9.5: The result of the joint smoothing of longitude and latitude.



```
plot(calif.gam2,select=10,se=FALSE)
```

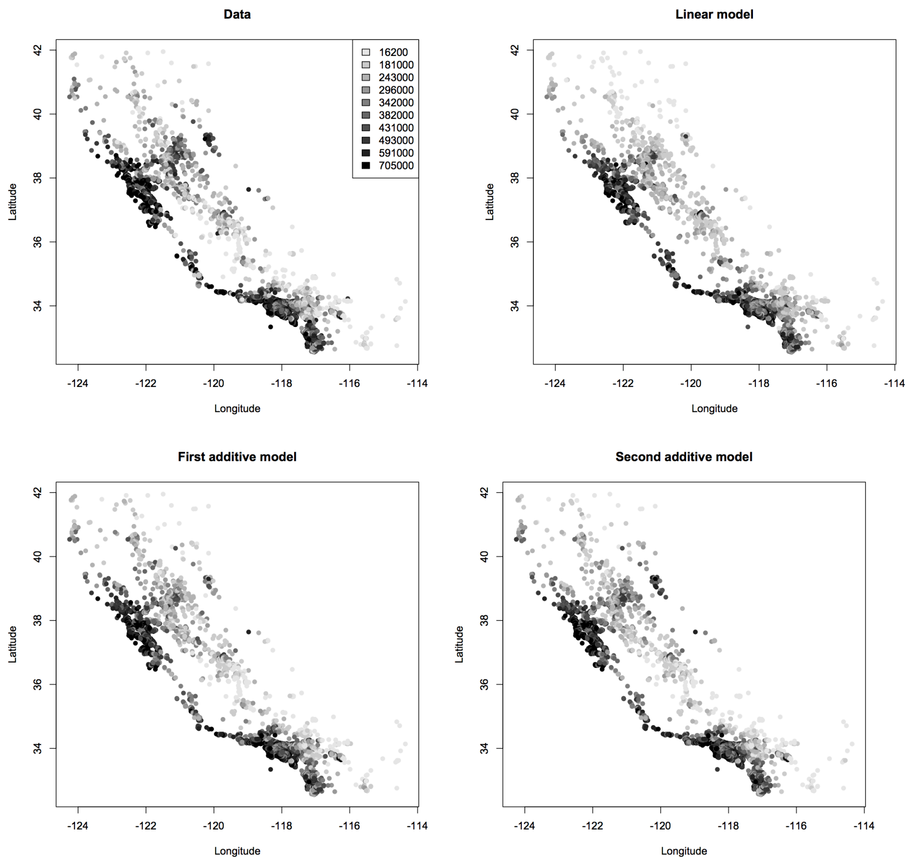
Figure 9.6: The result of the joint smoothing of longitude and latitude. Setting se=TRUE, the default, adds standard errors for the contour lines in multiple colors. Again, note that these are log units.

```
graymapper <- function(z, x=calif$LONGITUDE,y=calif$LATITUDE, n.levels=10,
  breaks=NULL,break.by="length",legend.loc="topright",digits=3,...) {
  my.greys = grey(((n.levels-1):0)/n.levels)
  if (!is.null(breaks)) {
    stopifnot(length(breaks) == (n.levels+1))
  }
  else {
    if(identical(break.by,"length")) {
      breaks = seq(from=min(z),to=max(z),length.out=n.levels+1)
    } else {
      breaks = quantile(z,probs=seq(0,1,length.out=n.levels+1))
    }
  }
  z = cut(z,breaks,include.lowest=TRUE)
  colors = my.greys[z]
  plot(x,y,col=colors,bg=colors,...)
  if (!is.null(legend.loc)) {
    breaks.printable <- signif(breaks[1:n.levels],digits)
    legend(legend.loc,legend=breaks.printable,fill=my.greys)
  }
  invisible(breaks)
}
```

Code Example 27: Map-making code. In its basic use, this takes vectors for x and y coordinates, and draws gray points whose color depends on a third vector for z, with darker points indicating higher values of z. Options allow for the control of the number of gray levels, setting the breaks between levels automatically, and using a legend. Returning the break-points makes it easier to use the same scale in multiple maps. See online for commented code.

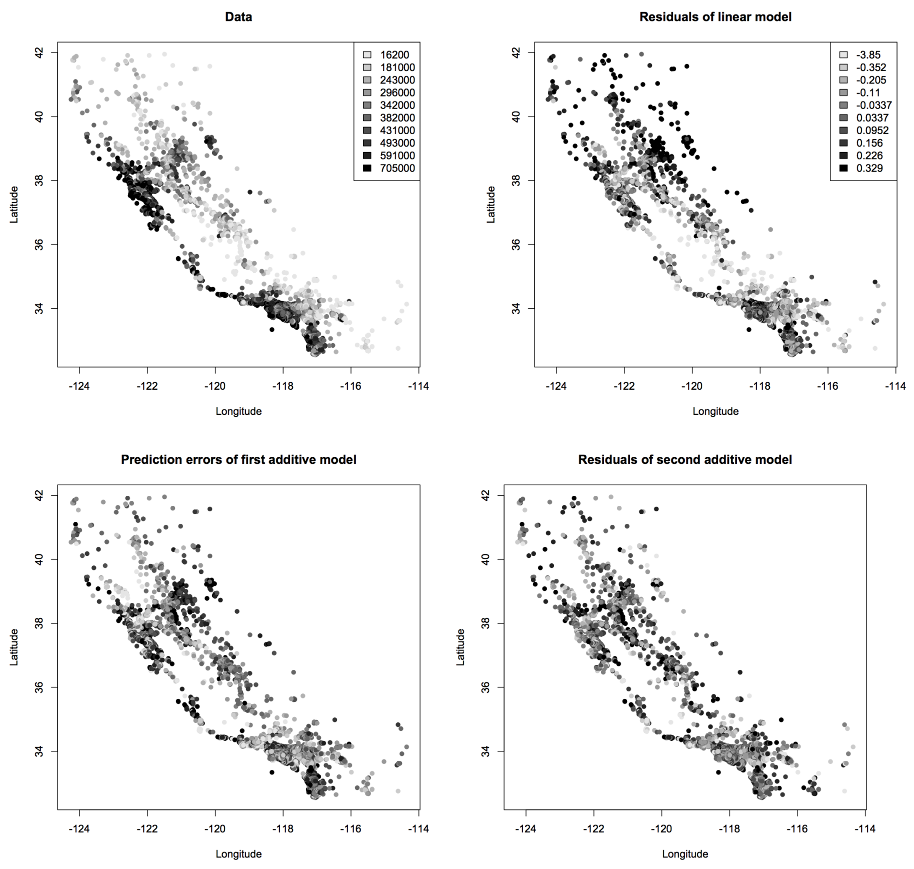
As you will recall from the homework, one of the big things wrong with the linear model was that its errors, i.e., the residuals, were highly structured and very far from random. In essence, it totally missed the existence of cities, and the fact that urban real estate is more expensive. It's a good idea, therefore, to make some maps, showing the actual values, and then, by way of contrast, the residuals of the models. Rather than do the plotting by hand over and over, let's write a function (Code Example 27).

Figures 9.7 and 9.8 show that allowing for the interaction of latitude and longitude (the smoothing term plotted in Figures 9.5–9.6) leads to a much more *random* and less systematic clumping of residuals. This is desirable in itself, even if it does little to improve the mean prediction error. Essentially, what that smoothing term is doing is picking out the existence of California's urban regions, and their distinction from the rural background. Examining the plots of the interaction term should suggest to you how inadequate it would be to just put in a `LONGITUDE`$\times$`LATITUDE` term in a linear model.



```
calif.breaks <- graymapper(calif$Median_house_value, pch=16, xlab="Longitude",
  ylab="Latitude",main="Data",break.by="quantiles")
graymapper(exp(preds.lm$fit), breaks=calif.breaks, pch=16, xlab="Longitude",
  ylab="Latitude",legend.loc=NULL, main="Linear model")
graymapper(exp(preds.gam$fit), breaks=calif.breaks, legend.loc=NULL,
  pch=16, xlab="Longitude", ylab="Latitude",main="First additive model")
graymapper(exp(preds.gam2$fit), breaks=calif.breaks, legend.loc=NULL,
  pch=16, xlab="Longitude", ylab="Latitude",main="Second additive model")
```

Figure 9.7: Maps of real prices (top left), and those predicted by the linear model (top right), the purely additive model (bottom left), and the additive model with interaction between latitude and longitude (bottom right). Categories are deciles of the actual prices.



```
graymapper(calif$Median_house_value, pch=16, xlab="Longitude",
  ylab="Latitude", main="Data", break.by="quantiles")
errors.in.dollars <- function(x) { calif$Median_house_value - exp(fitted(x)) }
lm.breaks <- graymapper(residuals(calif.lm), pch=16, xlab="Longitude",
  ylab="Latitude", main="Residuals of linear model",break.by="quantile")
graymapper(residuals(calif.gam), pch=16, xlab="Longitude",
  ylab="Latitude", main="Residuals errors of first additive model",
  breaks=lm.breaks, legend.loc=NULL)
graymapper(residuals(calif.gam2), pch=16, xlab="Longitude",
  ylab="Latitude", main="Residuals of second additive model",
  breaks=lm.breaks, legend.loc=NULL)
```

Figure 9.8: Actual housing values (top left), and the residuals of the three models. (The residuals are all plotted with the same color codes.) Notice that both the linear model and the additive model without spatial interaction systematically mis-price urban areas. The model with spatial interaction does much better at having randomly-scattered errors, though hardly perfect. — How would you make a map of the magnitude of regression errors?

Including an interaction between latitude and longitude in a spatial problem is pretty obvious. There are other potential interactions which might be important here — for instance, between the two measures of income, or between the total number of housing units available and the number of vacant units. We could, of course, just use a completely unrestricted nonparametric regression — going to the opposite extreme from the linear model. In addition to the possible curse-of-dimensionality issues, however, getting something like `npreg` to run with 7000 data points and 11 predictor variables requires a lot of patience. Other techniques, like nearest neighbor regression or regression trees, may run faster, though cross-validation can be demanding even there.

9.5 Closing Modeling Advice

With modern computing power, there are very few situations in which it is actually better to do linear regression than to fit an additive model. In fact, there seem to be only two good reasons to prefer linear models.

1. Our data analysis is guided by a credible scientific theory which asserts linear relationships *among the variables we measure* (not others, for which our observables serve as imperfect proxies).
2. Our data set is so massive that either the extra processing time, or the extra computer memory, needed to fit and store an additive rather than a linear model is prohibitive.

Even when the first reason applies, and we have good reasons to believe a linear theory, the truly scientific thing to do would be to *check* linearity, by fitting a flexible non-linear model and seeing if it looks close to linear. (We will see formal tests based on this idea in Chapter 10.) Even when the second reason applies, we would like to know how much bias we're introducing by using linear predictors, which we could do by randomly selecting a subset of the data which is small enough for us to manage, and fitting an additive model.

In the vast majority of cases when users of statistical software fit linear models, neither of these reasons applies: theory doesn't tell us to expect linearity, and our machines don't compel us to use it. Linear regression is then employed for no better reason than that users know how to type `lm` but not `gam`. *You* now know better, and can spread the word.

9.6 Further Reading

Simon Wood, who wrote the `mgcv` package, has a very nice book about additive models and their generalizations, Wood (2006); at this level it's your best source for further information. Buja *et al.* (1989) is a thorough theoretical treatment.

Ezekiel (1924) seems to be the first publication advocating the use of additive models as a general method, which he called "curivilinear multiple correlation". His paper was complete with worked examples on simulated data (with known answers) and real data (from economics)[10]. He was explicit that any reasonable smoothing or regression technique could be used to determine the partial response functions. He also gave a successive-approximation algorithm for finding partial response functions: start with an initial guess about all the partial responses; plot all the partial residuals; refine the partial responses simultaneously; repeat. This differs from back-fitting in that the partial response functions are updating in parallel within each cycle, not one after the other. This is a subtle difference, and Ezekiel's method will often work, but can run into trouble with correlated predictor variables, when back-fitting will not.

[10]"Each of these curves illustrates and substantiates conclusions reached by theoretical economic analysis. Equally important, they provide definite quantitative statements of the relationships. The method of ...curvilinear multiple correlation enable[s] us to use the favorite tool of the economist, *caeteris paribus*, in the analysis of actual happenings equally as well as in the intricacies of theoretical reasoning" (p. 453).