Chapter 7

Moving Beyond Conditional Expectations: Weighted Least Squares, Heteroskedasticity, Local Polynomial Regression

So far, all our estimates have been based on the mean squared error, giving equal importance to all observations. This is appropriate for looking at conditional expectations. In this chapter, we'll start to work with giving more or less weight to different observations. On the one hand, this will let us deal with other aspects of the distribution beyond the conditional expectation, especially the conditional variance. First we look at weighted least squares, and the effects that ignoring heteroskedasticity can have. This leads naturally to trying to estimate variance functions, on the one hand, and generalizing kernel regression to local polynomial regression, on the other.

7.1 Weighted Least Squares

When we use ordinary least squares to estimate linear regression, we (naturally) minimize the mean squared error:

$$MSE(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \vec{x}_i \cdot \beta)^2 \tag{7.1}$$

The solution is of course

$$\widehat{\beta}_{OLS} = (\mathbf{x}^T\mathbf{x})^{-1}\mathbf{x}^T\mathbf{y} \tag{7.2}$$

We could instead minimize the *weighted* mean squared error,

$$WMSE(\beta, \vec{w}) = \frac{1}{n}\sum_{i=1}^{n} w_i(y_i - \vec{x}_i \cdot \beta)^2 \tag{7.3}$$

This includes ordinary least squares as the special case where all the weights $w_i = 1$. We can solve it by the same kind of linear algebra we used to solve the ordinary linear least squares problem. If we write $\mathbf{w}$ for the matrix with the w_i on the diagonal and zeroes everywhere else, the solution is

$$\widehat{\beta}_{WLS} = (\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \mathbf{y} \tag{7.4}$$

But why would we want to minimize Eq. 7.3?

1. *Focusing accuracy.* We may care very strongly about predicting the response for certain values of the input — ones we expect to see often again, ones where mistakes are especially costly or embarrassing or painful, etc. — than others. If we give the points $\vec{x_i}$ near that region big weights w_i, and points elsewhere smaller weights, the regression will be pulled towards matching the data in that region.

2. *Discounting imprecision.* Ordinary least squares is the maximum likelihood estimate when the ϵ in $Y = \vec{X} \cdot \beta + \epsilon$ is IID Gaussian white noise. This means that the variance of ϵ has to be constant, and we measure the regression curve with the same precision elsewhere. This situation, of constant noise variance, is called **homoskedasticity**. Often however the magnitude of the noise is not constant, and the data are **heteroskedastic**.

   When we have heteroskedasticity, even if each noise term is still Gaussian, ordinary least squares is no longer the maximum likelihood estimate, and so no longer efficient. If however we know the noise variance σ_i^2 at each measurement i, and set $w_i = 1/\sigma_i^2$, we get the heteroskedastic MLE, and recover efficiency. (See below.)

   To say the same thing slightly differently, there's just no way that we can estimate the regression function as accurately where the noise is large as we can where the noise is small. Trying to give equal attention to all parts of the input space is a waste of time; we should be more concerned about fitting well where the noise is small, and expect to fit poorly where the noise is big.

3. *Doing something else.* There are a number of other optimization problems which can be transformed into, or approximated by, weighted least squares. The most important of these arises from generalized linear models, where the mean response is some nonlinear function of a linear predictor; we will look at them in Chapters 12 and 13.

In the first case, we decide on the weights to reflect our priorities. In the third case, the weights come from the optimization problem we'd really rather be solving. What about the second case, of heteroskedasticity?

7.2 Heteroskedasticity

Suppose the noise variance is itself variable. For example, the figure shows a simple linear relationship between the input X and the response Y, but also a nonlinear

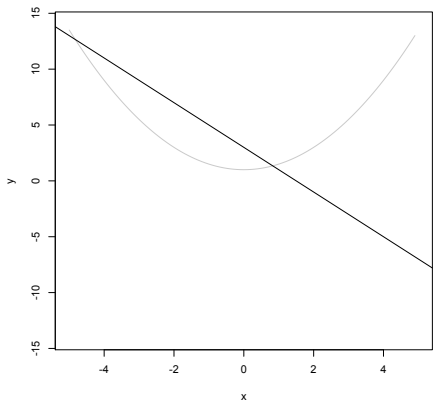


Figure 7.1: Black line: Linear response function ($y = 3 - 2x$). Grey curve: standard deviation as a function of x ($\sigma(x) = 1 + x^2/2$).

relationship between X and $\mathrm{Var}[Y]$.

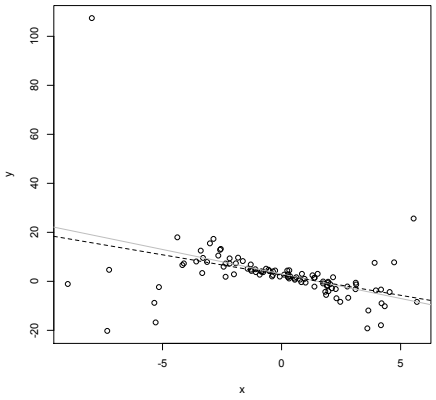
In this particular case, the ordinary least squares estimate of the regression line is $2.56 - 1.65x$, with R reporting standard errors in the coefficients of ± 0.52 and 0.20, respectively. Those are however calculated under the assumption that the noise is homoskedastic, which it isn't. And in fact we can see, pretty much, that there is heteroskedasticity — if looking at the scatter-plot didn't convince us, we could always plot the residuals against x, which we should do anyway.

To see whether that makes a difference, let's re-do this many times with different draws from the same model (Example 22).

Running `ols.heterosked.error.stats(100)` produces 10^4 random samples which all have the same x values as the first one, but different values of y, generated however from the same model. It then uses those samples to get the standard error of the ordinary least squares estimates. (Bias remains a non-issue.) What we find is the standard error of the intercept is only a little inflated (simulation value of 0.64 versus official value of 0.52), but the standard error of the slope is much larger than what R reports, 0.46 versus 0.20. Since the intercept is fixed by the need to make the regression line go through the center of the data, the real issue here is that our estimate of the slope is much less precise than ordinary least squares makes it out to be. Our estimate is still consistent, but not as good as it was when things were homoskedastic. Can we get back some of that efficiency?

7.2.1 Weighted Least Squares as a Solution to Heteroskedasticity

Suppose we visit the Oracle of Regression (Figure 7.4), who tells us that the noise has a standard deviation that goes as $1 + x^2/2$. We can then use this to improve our regression, by solving the weighted least squares problem rather than ordinary least



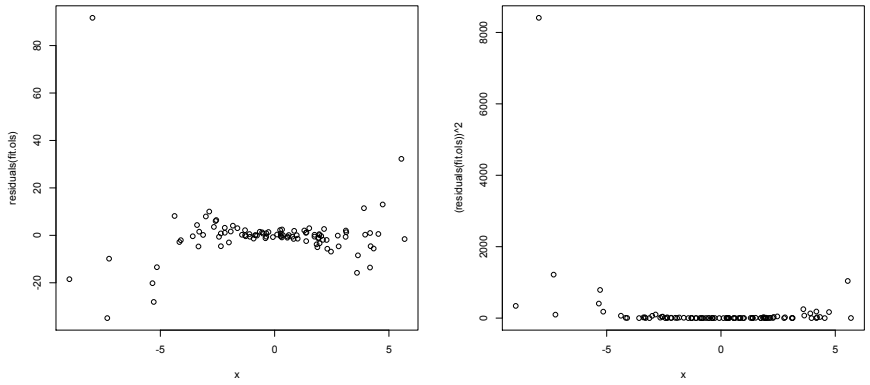

```
x = rnorm(100,0,3)
y = 3-2*x + rnorm(100,0,sapply(x,function(x){1+0.5*x^2}))
plot(x,y)
abline(a=3,b=-2,col="grey")
fit.ols = lm(y~x)
abline(fit.ols,lty=2)
```

Figure 7.2: Scatter-plot of $n = 100$ data points from the above model. (Here $X \sim \mathcal{N}(0,9)$.) Grey line: True regression line. Dashed line: ordinary least squares regression line.

```
ols.heterosked.example = function(n) {
  y = 3-2*x + rnorm(n,0,sapply(x,function(x){1+0.5*x^2}))
  fit.ols = lm(y~x)
  # Return the errors
  return(fit.ols$coefficients - c(3,-2))
}

ols.heterosked.error.stats = function(n,m=10000) {
  ols.errors.raw = t(replicate(m,ols.heterosked.example(n)))
  # transpose gives us a matrix with named columns
  intercept.sd = sd(ols.errors.raw[,"(Intercept)"])
  slope.sd = sd(ols.errors.raw[,"x"])
  return(list(intercept.sd=intercept.sd,slope.sd=slope.sd))
}
```

Code Example 22: Functions to generate heteroskedastic data and fit OLS regression to it, and to collect error statistics on the results.




```
plot(x,residuals(fit.ols))
plot(x,(residuals(fit.ols))^2)
```

Figure 7.3: Residuals (left) and squared residuals (right) of the ordinary least squares regression as a function of x. Note the much greater range of the residuals at large absolute values of x than towards the center; this changing dispersion is a sign of heteroskedasticity.

squares (Figure 7.5).

This not only looks better, it is better: the estimated line is now $2.67 - 1.91x$, with reported standard errors of 0.29 and 0.18. Does this check out with simulation? (Example 23.)

The standard errors from the simulation are 0.22 for the intercept and 0.23 for the slope, so R's internal calculations are working very well.

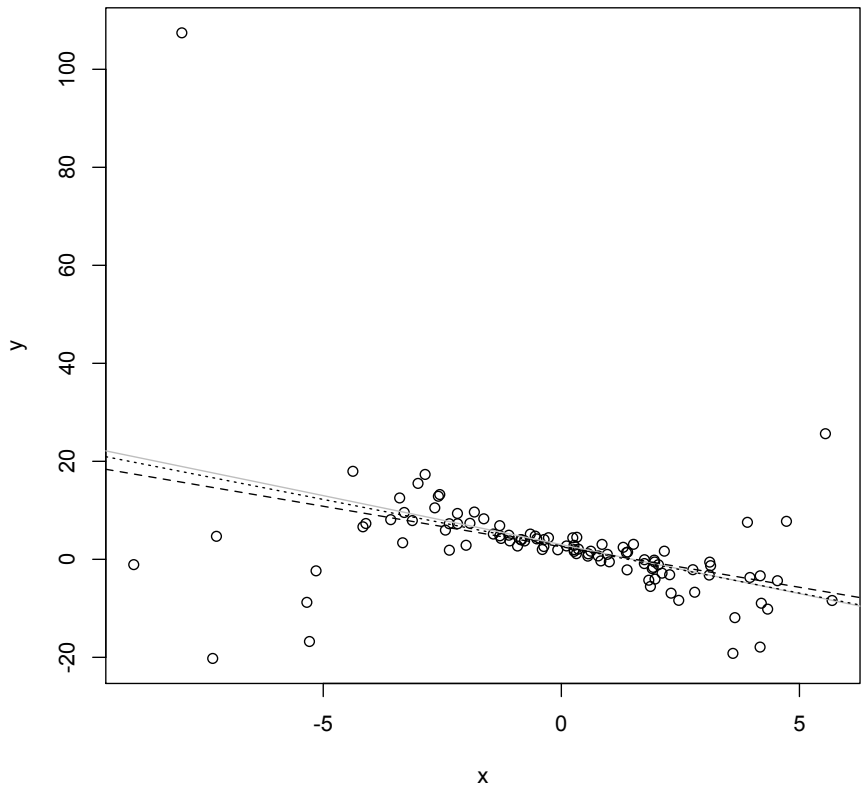
Why does putting these weights into WLS improve things?

7.2.2 Some Explanations for Weighted Least Squares

Qualitatively, the reason WLS with inverse variance weights works is the following. OLS tries equally hard to match observations at each data point.[1] Weighted least squares, naturally enough, tries harder to match observations where the weights are big, and less hard to match them where the weights are small. But each y_i contains not only the true regression function $r(x_i)$ but also some noise ϵ_i. The noise terms have large magnitudes where the variance is large. So we should want to have small weights where the noise variance is large, because there the data tends to be far from the true regression. Conversely, we should put big weights where the noise variance is small, and the data points are close to the true regression.

[1]Less anthropomorphically, the objective function in Eq. 7.1 has the same derivative with respect to the squared error at each point, $\frac{\partial MSE}{\partial (y_i - \vec{x}_i \cdot \beta)^2} = \frac{1}{n}$.

Figure 7.4: Statistician (right) consulting the Oracle of Regression (left) about the proper weights to use to overcome heteroskedasticity. (Image from http://en.wikipedia.org/wiki/Image:Pythia1.jpg.)




```
fit.wls = lm(y~x, weights=1/(1+0.5*x^2))
abline(fit.wls,lty=3)
```

Figure 7.5: Figure 7.2, with addition of weighted least squares regression line (dotted).

```
wls.heterosked.example = function(n) {
  y = 3-2*x + rnorm(n,0,sapply(x,function(x){1+0.5*x^2}))
  fit.wls = lm(y~x,weights=1/(1+0.5*x^2))
  # Return the errors
  return(fit.wls$coefficients - c(3,-2))
}

wls.heterosked.error.stats = function(n,m=10000) {
  wls.errors.raw = t(replicate(m,wls.heterosked.example(n)))
  # transpose gives us a matrix with named columns
  intercept.sd = sd(wls.errors.raw[,"(Intercept)"])
  slope.sd = sd(wls.errors.raw[,"x"])
  return(list(intercept.sd=intercept.sd,slope.sd=slope.sd))
}
```

Code Example 23: Linear regression of heteroskedastic data, using weighted least-squared regression.

The qualitative reasoning in the last paragraph doesn't explain why the weights should be inversely proportional to the variances, $w_i \propto 1/\sigma^2_{x_i}$ — why not $w_i \propto 1/\sigma_{x_i}$, for instance? Look at the equation for the WLS estimates again:

$$\widehat{\beta}_{WLS} = (\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \mathbf{y} \tag{7.5}$$

Imagine holding $\mathbf{x}$ constant, but repeating the experiment multiple times, so that we get noisy values of $\mathbf{y}$. In each experiment, $Y_i = \vec{x}_i \cdot \beta + \epsilon_i$, where $\mathbf{E}\left[\epsilon_i\right] = 0$ and $\mathrm{Var}\left[\epsilon_i\right] = \sigma^2_{x_i}$. So

$$\begin{aligned} \widehat{\beta}_{WLS} &= (\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \mathbf{x} \beta + (\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \epsilon & (7.6) \\ &= \beta + (\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \epsilon & (7.7) \end{aligned}$$

Since $\mathbf{E}\left[\epsilon\right] = 0$, the WLS estimator is unbiased:

$$\mathbf{E}\left[\widehat{\beta}_{WLS}\right] = \beta \tag{7.8}$$

In fact, for the j^{th} coefficient,

$$\begin{aligned} \widehat{\beta}_j &= \beta_j + \left[(\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w} \epsilon\right]_j & (7.9) \\ &= \beta_j + \sum_{i=1}^{n} H_{ji}(w) \epsilon_i & (7.10) \end{aligned}$$

where in the last line I have bundled up $(\mathbf{x}^T \mathbf{w} \mathbf{x})^{-1} \mathbf{x}^T \mathbf{w}$ as a matrix $\mathbf{H}(w)$, with the argument to remind us that it depends on the weights. Since the WLS estimate is

unbiased, it's natural to want it to also have a small variance, and

$$\mathrm{Var}\left[\widehat{\beta}_j\right] = \sum_{i=1}^{n} H_{ji}(w)\sigma^2_{x_i} \tag{7.11}$$

It can be shown — the result is called the **generalized Gauss-Markov theorem** — that picking weights to minimize the variance in the WLS estimate has the unique solution $w_i = 1/\sigma^2_{x_i}$. It does not require us to assume the noise is Gaussian, but the proof is a bit tricky (see Appendix D).

A less general but easier-to-grasp result comes from adding the assumption that the noise around the regression line is Gaussian — that

$$Y = \vec{x}\cdot\beta + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2_x) \tag{7.12}$$

The log-likelihood is then (EXERCISE 1)

$$-\frac{n}{2}\ln 2\pi - \frac{1}{2}\sum_{i=1}^{n}\log\sigma^2_{x_i} - \frac{1}{2}\sum_{i=1}^{n}\frac{(y_i - \vec{x}_i\cdot\beta)^2}{\sigma^2_{x_i}} \tag{7.13}$$

If we maximize this with respect to β, everything except the final sum is irrelevant, and so we minimize

$$\sum_{i=1}^{n}\frac{(y_i - \vec{x}_i\cdot\beta)^2}{\sigma^2_{x_i}} \tag{7.14}$$

which is just weighted least squares with $w_i = 1/\sigma^2_{x_i}$. So, if the probabilistic assumption holds, WLS is the efficient maximum likelihood estimator.

7.2.3 Finding the Variance and Weights

All of this was possible because the Oracle told us what the variance function was. What do we do when the Oracle is not available (Figure 7.6)?

Under some situations we can work things out for ourselves, without needing an oracle.

- We know, empirically, the precision of our measurement of the response variable — we know how precise our instruments are, or each value of the response is really an average of several measurements so we can use their standard deviations, etc.
- We know how the noise in the response must depend on the input variables. For example, when taking polls or surveys, the variance of the proportions we find should be inversely proportional to the sample size. So we can make the weights proportional to the sample size.

Both of these outs rely on kinds of background knowledge which are easier to get in the natural or even the social sciences than in many industrial applications. However, there are approaches for other situations which try to use the observed residuals to get estimates of the heteroskedasticity; this is the topic of the next section.

Figure 7.6: The Oracle may be out (left), or too creepy to go visit (right). What then?
(Left, the sacred oak of the Oracle of Dodona, copyright 2006 by Flickr user "essayen", http://flickr.com/photos/essayen/245236125/; right, the entrace to the cave of the Sibyl of Cumæ, copyright 2005 by Flickr user "pverdicchio", http://flickr.com/photos/occhio/17923096/. Both used under Creative Commons license.)

7.3 Conditional Variance Function Estimation

Remember that there are two equivalent ways of defining the variance:

$$\mathrm{Var}[X] = \mathbf{E}\left[X^2\right] - (\mathbf{E}[X])^2 = \mathbf{E}\left[(X - \mathbf{E}[X])^2\right] \tag{7.15}$$

The latter is more useful for us when it comes to estimating variance functions. We have already figured out how to estimate means — that's what all this previous work on smoothing and regression is for — and the deviation of a random variable from its mean shows up as a residual.

There are two generic ways to estimate conditional variances, which differ slightly in how they use non-parametric smoothing. We can call these the **squared residuals method** and the **log squared residuals method**. Here is how the first one goes.

1. Estimate $r(x)$ with your favorite regression method, getting $\hat{r}(x)$.
2. Construct the **squared residuals**, $u_i = (y_i - \hat{r}(x_i))^2$.
3. Use your favorite *non-parametric* method to estimate the conditional mean of the u_i, call it $\widehat{q}(x)$.
4. Predict the variance using $\widehat{\sigma}^2_x = \widehat{q}(x)$.

The log-squared residuals method goes very similarly.[2]

1. Estimate $r(x)$ with your favorite regression method, getting $\hat{r}(x)$.
2. Construct the **log squared residuals**, $z_i = \log(y_i - \hat{r}(x_i))^2$.
3. Use your favorite *non-parametric* method to estimate the conditional mean of the z_i, call it $\hat{s}(x)$.

[2] I learned it from Wasserman (2006, pp. 87–88).

4. Predict the variance using $\widehat{\sigma}^2_x = \exp \widehat{s}(x)$.

The quantity $y_i - \hat{r}(x_i)$ is the i^{th} residual. If $\widehat{r} \approx r$, then the residuals should have mean zero. Consequently the variance of the residuals (which is what we want) should equal the expected squared residual. So squaring the residuals makes sense, and the first method just smoothes these values to get at their expectations.

What about the second method — why the log? Basically, this is a convenience — squares are necessarily non-negative numbers, but lots of regression methods don't easily include constraints like that, and we really don't want to predict negative variances.[3] Taking the log gives us an unbounded range for the regression.

Strictly speaking, we don't need to use non-parametric smoothing for either method. If we had a parametric model for σ^2_x, we could just fit the parametric model to the squared residuals (or their logs). But even if you think you know what the variance function should look like it, why not check it?

We came to estimating the variance function because of wanting to do weighted least squares, but these methods can be used more generally. It's often important to understand variance in its own right, and this is a general method for estimating it. Our estimate of the variance function depends on first having a good estimate of the regression function

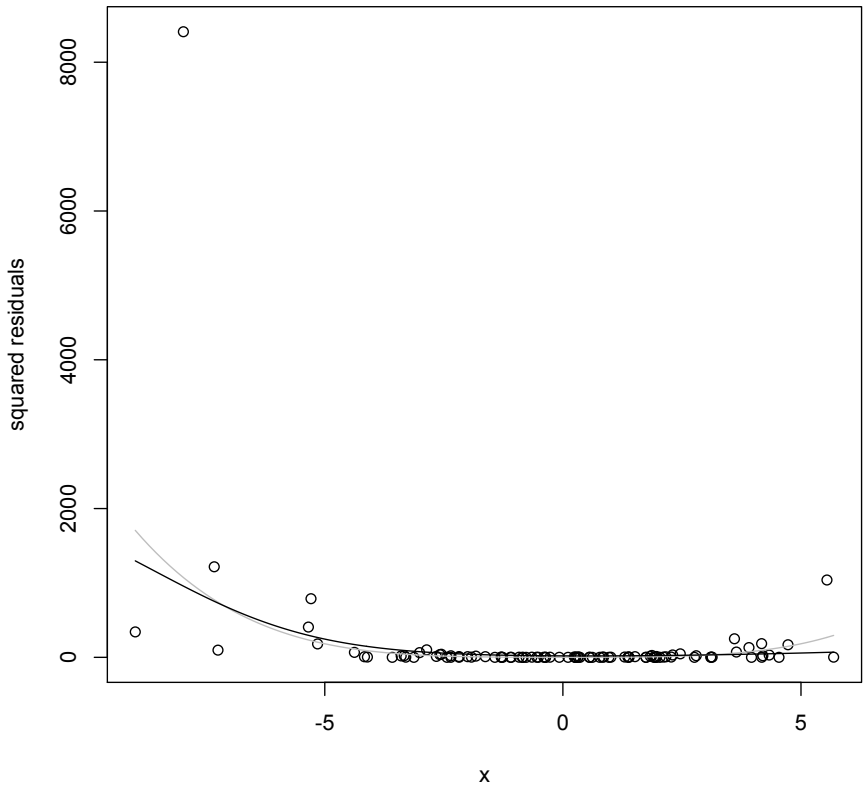
7.3.1 Iterative Refinement of Mean and Variance: An Example

The estimate $\widehat{\sigma}^2_x$ depends on the initial estimate of the regression function $\hat{r}(x)$. But, as we saw when we looked at weighted least squares, taking heteroskedasticity into account can change our estimates of the regression function. This suggests an iterative approach, where we alternate between estimating the regression function and the variance function, using each to improve the other. That is, we take either method above, and then, once we have estimated the variance function $\widehat{\sigma}^2_x$, we re-estimate $\hat{r}$ using weighted least squares, with weights inversely proportional to our estimated variance. Since this will generally change our estimated regression, it will change the residuals as well. Once the residuals have changed, we should re-estimate the variance function. We keep going around this cycle until the change in the regression function becomes so small that we don't care about further modifications. It's hard to give a strict guarantee, but *usually* this sort of iterative improvement will converge.

Let's apply this idea to our example. Figure 7.3b already plotted the residuals from OLS. Figure 7.7 shows those squared residuals again, along with the true variance function and the estimated variance function.

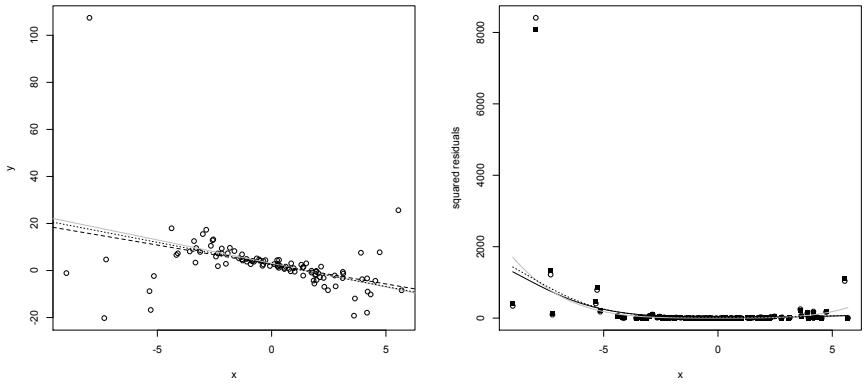
The OLS estimate of the regression line is not especially good ($\widehat{\beta}_0 = 2.56$ versus $\beta_0 = 3$, $\widehat{\beta}_1 = -1.65$ versus $\beta_1 = -2$), so the residuals are systematically off, but it's clear from the figure that kernel smoothing of the squared residuals is picking up on the heteroskedasticity, and getting a pretty reasonable picture of the variance function.

[3] Occasionally you do see people doing things like claiming that genetics explains more than 100% of the variance in some psychological trait, and so the contributions of environment and up-bringing have negative variance. Some of them — for instance, Alford *et al.* (2005) — manage to say this with a straight face.




```
plot(x,residuals(fit.ols)^2,ylab="squared residuals")
curve((1+x^2/2)^2,col="grey",add=TRUE)
require(np)
var1 <- npreg(residuals(fit.ols)^2 ~ x)
grid.x <- seq(from=min(x),to=max(x),length.out=300)
lines(grid.x,predict(var1,exdat=grid.x))
```

Figure 7.7: Points: actual squared residuals from the OLS line. Grey curve: true variance function, $\sigma_x^2 = (1 + x^2/2)^2$. Black curve: kernel smoothing of the squared residuals, using `npreg`.




```
fit.wls1 <- lm(y~x,weights=1/fitted(var1))
plot(x,y)
abline(a=3,b=-2,col="grey")
abline(fit.ols,lty=2)
abline(fit.wls1,lty=3)
plot(x,(residuals(fit.ols))^2,ylab="squared residuals")
points(x,(residuals(fit.wls1))^2,pch=15)
lines(grid.x,predict(var1,exdat=grid.x))
var2 <- npreg(residuals(fit.wls1)^2 ~ x)
curve((1+x^2/2)^2,col="grey",add=TRUE)
lines(grid.x,predict(var2,exdat=grid.x),lty=3)
```

Figure 7.8: Left: As in Figure 7.2, but with the addition of the weighted least squares regression line (dotted), using the estimated variance from Figure 7.7 for weights. Right: As in Figure 7.7, but with the addition of the residuals from the WLS regression (black squares), and the new estimated variance function (dotted curve).

Now we use the estimated variance function to re-estimate the regression line, with weighted least squares.

```
> fit.wls1 <- lm(y~x,weights=1/fitted(var1))
> coefficients(fit.wls1)
(Intercept)           x
   2.595860   -1.876042
> var2 <- npreg(residuals(fit.wls1)^2 ~ x)
```

The slope has changed substantially, and in the right direction (Figure 7.8a). The residuals have also changed (Figure 7.8b), and the new variance function is closer to the truth than the old one.

Since we have a new variance function, we can re-weight the data points and re-estimate the regression:

```
> fit.wls2 <- lm(y~x,weights=1/fitted(var2))
> coefficients(fit.wls2)
(Intercept)           x
   2.625295   -1.914075
> var3 <- npreg(residuals(fit.wls2)^2 ~ x)
```

Since we know that the true coefficients are 3 and -2, we know that this is moving in the right direction. If I hadn't told you what they were, you could still observe that the difference in coefficients between `fit.wls1` and `fit.wls2` is smaller than that between `fit.ols` and `fit.wls1`, which is a sign that this is converging.

I will spare you the plot of the new regression and of the new residuals. When we update a few more times:

```
> fit.wls3 <- lm(y~x,weights=1/fitted(var3))
> coefficients(fit.wls3)
(Intercept)           x
   2.630249   -1.920476
> var4 <- npreg(residuals(fit.wls3)^2 ~ x)
> fit.wls4 <- lm(y~x,weights=1/fitted(var4))
> coefficients(fit.wls4)
(Intercept)           x
   2.631063   -1.921540
```

By now, the coefficients of the regression are changing in the fourth significant digit, and we only have 100 data points, so the imprecision from a limited sample surely swamps the changes we're making, and we might as well stop.

Manually going back and forth between estimating the regression function and estimating the variance function is tedious. We could automate it with a function, which would look something like this:

```
iterative.wls <- function(x,y,tol=0.01,max.iter=100) {
  iteration <- 1
  old.coefs <- NA
  regression <- lm(y~x)
  coefs <- coefficients(regression)
  while (is.na(old.coefs) ||
         ((max(coefs - old.coefs) > tol) && (iteration < max.iter))) {
    variance <- npreg(residuals(regression)^2 ~ x)
    old.coefs <- coefs
    iteration <- iteration+1
    regression <- lm(y~x,weights=1/fitted(variance))
    coefs <- coefficients(regression)
  }
  return(list(regression=regression,variance=variance,iterations=iteration))
}
```

This starts by doing an unweighted linear regression, and then alternates between WLS for the getting the regression and kernel smoothing for getting the variance. It

stops when no parameter of the regression changes by more than `tol`, or when it's gone around the cycle `max.iter` times.[4] This code is a bit too inflexible to be really "industrial strength" (what if we wanted to use a data frame, or a more complex regression formula?), but shows the core idea.

7.3.2 Real Data Example: Old Heteroskedastic

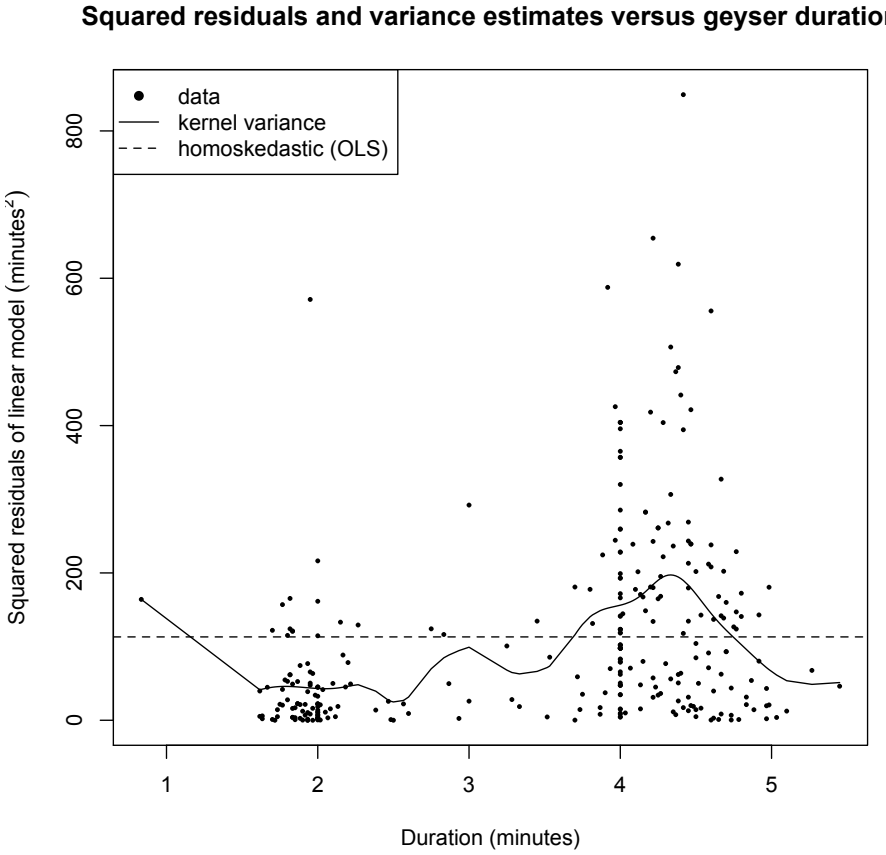
§5.3.2 introduced the `geyser` data set, which is about predicting the waiting time between consecutive eruptions of the "Old Faithful" geyser at Yellowstone National Park from the duration of the latest eruption. Our exploration there showed that a simple linear model (of the kind often fit to this data in textbooks and elementary classes) is not very good, and raised the suspicion that one important problem was heteroskedasticity. Let's follow up on that, building on the computational work done in that section.

The estimated variance function `geyser.var` does not look particularly flat, but it comes from applying a fairly complicated procedure (kernel smoothing with data-driven bandwidth selection) to a fairly limited amount of data (299 observations). Maybe that's the amount of wiggliness we should *expect* to see due to finite-sample fluctuations? To rule this out, we can make surrogate data from the homoskedastic model, treat it the same way as the real data, and plot the resulting variance functions (Figure 7.10). The conditional variance functions estimated from the homoskedastic model are flat or gently varying, with much less range than what's seen in the data.

While that sort of qualitative comparison is genuinely informative, one can also be more quantitative. One might measure heteroskedasticity by, say, evaluating the conditional variance at all the data points, and looking at the ratio of the interquartile range to the median. This would be zero for perfect homoskedasticity, and grow as the dispersion of actual variances around the "typical" variance increased. For the data, this is `IQR(fitted(geyser.var))/median(fitted(geyser.var))` $= 0.86$. Simulations from the OLS model give values around 10^{-15}.

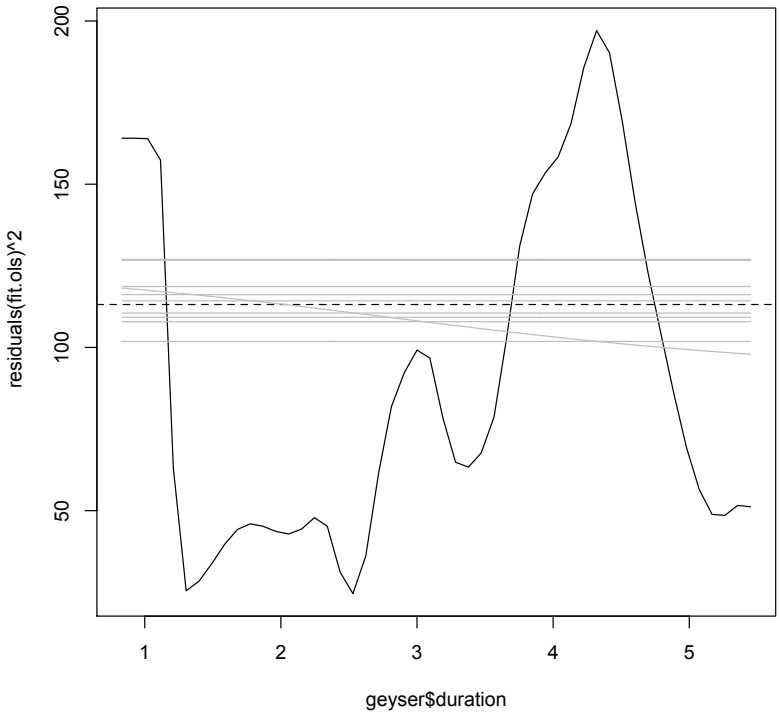
There is nothing particularly special about this measure of heteroskedasticity — after all, I just made it up. The broad point it illustrates is that whenever we have some sort of quantitttive summary statistic we can calculate on our real data, we can also calculate the same statistic on realizations of the model, and the difference will then tell us something about how close the simulations, and so the model, come to the data. In this case, we learn that the linear, homoskedastic model seriously understates the variability of this data. That leaves open the question of whether the problem is the linearity or the homoskedasticity; I will leave that question to EXERCISE 5.

[4] The condition in the `while` loop is a bit complicated, to ensure that the loop is executed at least once. Some languages have an `until` control structure which would simplify this.




```
plot(geyser$duration, residuals(fit.ols)^2, cex=0.5, pch=16,
  main="Squared residuals and variance estimates versus geyser duration",
  xlab="Duration (minutes)",
  ylab=expression("Squared residuals of linear model "(minutes^2)))
library(np)
geyser.var <- npreg(residuals(fit.ols)^2~geyser$duration)
duration.order <- order(geyser$duration)
lines(geyser$duration[duration.order],fitted(geyser.var)[duration.order])
abline(h=summary(fit.ols)$sigma^2,lty="dashed")
legend("topleft",
  legend=c("data","kernel variance","homoskedastic (OLS)"),
  lty=c(-1,1,2),pch=c(16,-1,-1))
```

Figure 7.9: Squared residuals from the linear model of Figure 5.4, plotted against duration, along with the unconditional, homoskedastic variance implicit in OLS (dashed), and a kernel-regression estimate of the conditional variance (solid).




```
plot(geyser.var)
abline(h=summary(fit.ols)$sigma^2,lty=2)
duration.grid <- seq(from=min(geyser$duration),to=max(geyser$duration),
  length.out=300)
one.var.func <- function() {
  fit <- lm(waiting ~ duration, data=rgeyser())
  var.func <- npreg(residuals(fit)^2 ~ geyser$duration)
  lines(duration.grid,predict(var.func,exdat=duration.grid),col="grey")
}
invisible(replicate(10,one.var.func()))
```

Figure 7.10: The actual conditional variance function estimated from the Old Faithful data (and the linear regression), in black, plus the results of applying the same procedure to simulations from the homoskedastic linear regression model (grey lines; see §5.3.2 for the `rgeyser()` function). The fact that the estimates from the simulations are all flat or gently sloped suggests that the changes in variance found in the data are too large to just be sampling noise.

7.4 Re-sampling Residuals with Heteroskedasticity

Re-sampling the residuals of a regression, as described in §6.4, assumes that the distribution of fluctuations around the regression curve is the same for all values of the input x. Under heteroskedasticity, this is of course not the case. Nonetheless, we can still re-sample residuals to get bootstrap confidence intervals, standard errors, and so forth, provided we define and scale them properly. If we have a conditional variance function $\hat{\sigma}^2(x)$, or a conditional standard deviation function $\hat{sigma}(x)$, as well as the estimated regression function $\hat{r}(x)$, we can combine them to re-sample heteroskedastic residuals.

1. Construct the standardized residuals, by dividing the actual residuals by the conditional standard deviation:
$$\eta_i = \epsilon_i / \hat{\sigma}(x_i) \tag{7.16}$$
The η_i should now be all the same size (in distribution!), no matter where x_i is in the space of predictors.
2. Re-sample the η_i with replacement, to get $\tilde{\eta}_1, \ldots \tilde{\eta}_n$.
3. Set $\tilde{x}_i = x_i$.
4. Set $\tilde{y}_i = \hat{r}(\tilde{x}_i) + \hat{\sigma}(\tilde{x}_i)\tilde{\eta}_i$.
5. Analyze the surrogate data $(\tilde{x}_1, \tilde{y}_1), \ldots (\tilde{x}_n, \tilde{y}_n)$ like it was real data.

Of course, this still assumes that the *only* difference in distribution for the noise at different values of x is its scale.

7.5 Local Linear Regression

Switching gears, recall from Chapter 2 that one reason it can be sensible to use a linear approximation to the true regression function $r(x)$ is that we can always[5] Taylor-expand the latter around any point x_0,

$$r(x) = r(x_0) + \sum_{k=1}^{\infty} \frac{(x - x_0)^k}{k!} \left. \frac{d^k r}{d r^k} \right|_{x=x_0} \tag{7.17}$$

and similarly with all the partial derivatives in higher dimensions. If we truncate the series at first order, $r(x) \approx r(x_0) + (x - x_0) r'(x_0)$, we see that the first-order coefficient $r'(x_0)$ is the best linear prediction coefficient, at least when x is sufficiently close to x_0. The snag in this line of argument is that if $r(x)$ isn't really linear, then r' isn't a constant, and the optimal linear predictor to use depends on where we want to make predictions.

[5] At least if $r(x)$ is differentiable.

However, statisticians are thrifty people, and having assembled all the machinery for linear regression, they are loathe to throw it away just because the fundamental model is wrong. If we can't fit one line, why not fit many? If each point has a different best linear regression, why not estimate them all? Thus the idea of **local** linear regression: fit a different linear regression everywhere, weighting the data points by how close they are to the point of interest[6].

The very simplest approach we could take would be to divide up the range of x into so many bins, and fit a separate linear regression for each bin. This is unsatisfying for at least three reasons. First, it gives us weird discontinuities at the boundaries between bins. Second, it introduces an odd sort of bias, where our predictions near the boundaries of a bin depend strongly on data from the other side of the bin, and not at all on nearby data points just across the border, which is weird. Third, we need to pick the bins.

The next simplest approach would be to first figure out where we want to make a prediction (say x), and do a linear regression with all the data points which were sufficiently close, $|x_i - x| \leq h$ for some h. Now we are basically using a uniform-density kernel to weight the data points. This eliminates two problems from the binning idea — the examples we include are always centered on the x we're trying to get a prediction for, and we just need to pick one bandwidth h rather than placing all the bin boundaries. But still, each example point always has either weight 0 or weight 1, so our predictions change jerkily as training points fall into or out of the window. It generally works nicer to have the weights change more smoothly with the distance, starting off large and then gradually trailing to zero.

By now bells may be going off in your head, as this sounds very similar to the kernel regression. In fact, kernel regression is what happens when we truncate Eq. 7.17 at *zeroth* order, getting **locally constant** regression. Here's the problem we're setting up:

$$\underset{m(x)}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} w_i(x)\big(y_i - m(x)\big)^2 \tag{7.18}$$

which has the solution

$$\widehat{m}(x) = \frac{\sum_{i=1}^{n} w_i(x) y_i}{\sum_{j=1}^{n} w_j(x)} \tag{7.19}$$

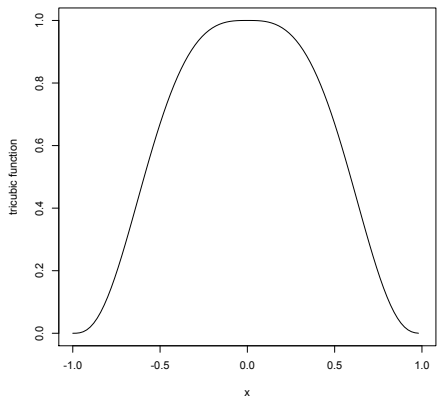
which just is our kernel regression, with the weights being proportional to the kernels, $w_i(x) \propto K(x_i, x)$. (Without loss of generality, we can take the constant of proportionality to be 1.)

What about **locally linear** regression? The optimization problem is

$$\underset{m,\beta}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} w_i(x)\big(y_i - m(x) - (x_i - x)\cdot\beta(x)\big)^2 \tag{7.20}$$

where again we can write $w_i(x)$ as proportional to some kernel function, $w_i(x) \propto K(x_i, x)$. To solve this, abuse notation slightly to define $z_i = (1, x_i - x)$, i.e., the

[6] Some people say "local linear" and some "local*ly* linear".




```
curve((1-abs(x)^3)^3,from=-1,to=1,ylab="tricubic function")
```

Figure 7.11: The tricubic kernel, with broad plateau where $|x| \approx 0$, and the smooth fall-off to zero at $|x| = 1$.

displacement from x, with a 1 stuck at the beginning to (as usual) handle the intercept. Now, by the machinery above,

$$\widehat{(m, \beta(x))} = (\mathbf{z}^T \mathbf{w}(x) \mathbf{z})^{-1} \mathbf{z}^T \mathbf{w}(x) \mathbf{y} \tag{7.21}$$

and the prediction is just the intercept, $\hat{m}$. If you need an estimate of the first derivatives, those are the $\widehat{\beta}$. Notice, from Eq. 7.21, that if the weights given to each training point change smoothly with x, then the predictions will also change smoothly.[7]

Using a smooth kernel whose density is positive everywhere, like the Gaussian, ensures that the weights will change smoothly. But we could also use a kernel which goes to zero outside some finite range, so long as the kernel rises gradually from zero inside the range. For locally linear regression, a common choice of kernel is therefore the **tri-cubic**,

$$K(x_i, x) = \left(1 - \left(\frac{|x_i - x_0|}{h}\right)^3\right)^3 \tag{7.22}$$

if $|x - x_i| < h$, and $= 0$ otherwise (Figure 7.11).

7.5.1 Advantages and Disadvantages of Locally Linear Regression

Why would we use locally linear regression, if we already have kernel regression?

[7] Notice that local linear predictors are still linear smoothers as defined in Chapter 1, (i.e., the predictions are linear in the y_i), but they are not, strictly speaking, *kernel* smoothers, since you can't re-write the last equation in the form of a kernel average.

1. You may recall that when we worked out the bias of kernel smoothers (Eq. 4.10 in Chapter 4), we got a contribution that was proportional to $r'(x)$. If we do an analogous analysis for locally linear regression, the bias is the same, *except* that this derivative term goes away.

2. Relatedly, that analysis we did of kernel regression tacitly assumed the point we were looking at was in the middle of the training data (or at least less than h from the border). The bias gets worse near the edges of the training data. Suppose that the true $r(x)$ is decreasing in the vicinity of the largest x_i. (See the grey curve in Figure 7.12.) When we make our predictions there, in kernel regression we can only average values of y_i which tend to be systematically larger than the value we want to predict. This means that our kernel predictions are systematically biased upwards, and the size of the bias grows with $r'(x)$. (See the black line in Figure 7.12 at the lower right.) If we use a locally linear model, however, it can pick up that there is a trend, and reduce the edge bias by extrapolating it (dashed line in the figure).

3. The predictions of locally linear regression tend to be smoother than those of kernel regression, simply because we are locally fitting a smooth line rather than a flat constant. As a consequence, estimates of the derivative $\frac{d\hat{r}}{dx}$ tend to be less noisy when $\hat{r}$ comes from a locally linear model than a kernel regression.

Of course, total prediction error depends not only on the bias but also on the variance. Remarkably enough, the variance for kernel regression and locally linear regression is the same. Since locally linear regression has smaller bias, the former is often predictively superior.

There are several packages which implement locally linear regression. Since we are already using `np`, one of the simplest is to set the `regtype="ll"` in `npreg`.[8] There are several other packages which support it, notably `KernSmooth` and `locpoly`.

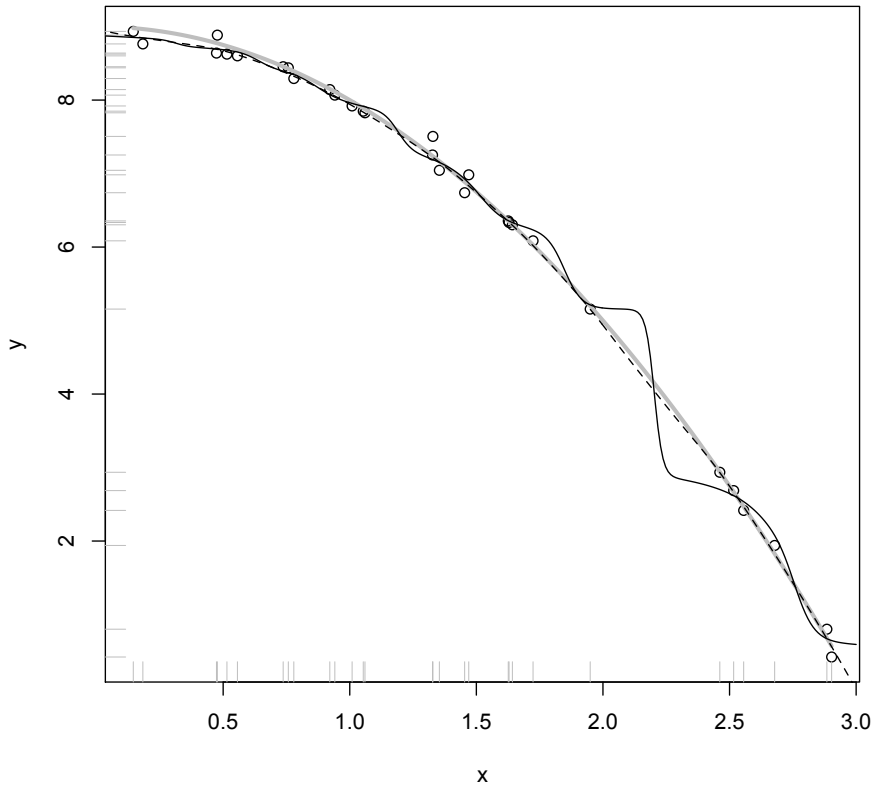
As the name of the latter suggests, there is no reason we *have* to stop at locally linear models, and we could use local polynomials of any order. The main reason to use a higher-order local polynomial, rather than a locally-linear or locally-constant model, is to estimate higher derivatives. Since this is a somewhat specialized topic, I will not say more about it.

7.5.2 Lowess

There is however one additional topic in locally linear models which is worth mentioning. This is the variant called **lowess** or **loess**.[9] The basic idea is to fit a locally linear model, with a kernel which goes to zero outside a finite window and rises gradually inside it, typically the tri-cubic I plotted earlier. The wrinkle, however, is that rather than solving a least *squares* problem, it minimizes a different and more

[8] `"ll"` stands for "locally linear", of course; the default is `regtype="lc"`, for "locally constant".

[9] I have heard this name explained as an acronym for both "locally weighted scatterplot smoothing" and "locally weight sum of squares".




```
x <- runif(30,max=3)
y <- 9-x^2 + rnorm(30,sd=0.1)
plot(x,y); rug(x,side=1, col="grey"); rug(y,side=2, col="grey")
curve(9-x^2,col="grey",add=TRUE,lwd=3)
grid.x <- seq(from=0,to=3,length.out=300)
np0 <- npreg(y~x); lines(grid.x,predict(np0, exdat=grid.x))
np1 <- npreg(y~x,regtype="ll"); lines(grid.x,predict(np1, exdat=grid.x),lty=2)
```

Figure 7.12: Points are samples from the true, nonlinear regression function shown in grey. The solid black line is a kernel regression, and the dashed line is a locally linear regression. Note that the locally linear model is smoother than the kernel regression, and less biased when the true curve has a non-zero bias at a boundary of the data (far right).

"robust" loss function,

$$\underset{\beta(x)}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} w_i(x)\ell(y - \vec{x}_i \cdot \beta(x)) \tag{7.23}$$

where $\ell(a)$ doesn't grow as rapidly for large a as a^2. The idea is to make the fitting less vulnerable to occasional large outliers, which would have very large squared errors, unless the regression curve went far out of its way to accommodate them. For instance, we might have $\ell(a) = a^2$ if $|a| < 1$, and $\ell(a) = 2|a| - 1$ otherwise[10]. We will come back to robust estimation later, but I bring it up now because it's a very common smoothing technique, especially for visualization.

Lowess smoothing is implemented in the default R packages through the function `lowess` (rather basic), and through the function `loess` (more sophisticated), as well as in the CRAN package `locfit` (more sophisticated still). The lowess idea can be combined with local fitting of higher-order polynomials; the `loess` and `locfit` commands both support this.

7.6 Exercises

To think through or experiment with, not to hand in.

1. Show that the model of Eq. 7.12 has the log-likelihood given by Eq. 7.13
2. Do the calculus to verify Eq. 7.4.
3. Is $w_i = 1$ a necessary as well as a sufficient condition for Eq. 7.3 and Eq. 7.1 to have the same minimum?
4. The text above looked at whether WLS gives better parameter estimates than OLS when there is heteroskedasticity, and we know and use the variance. Modify the code for to see which one has better generalization error.
5. COMPUTING §7.3.2 looked at the residuals of the linear regression model for the Old Faithful geyser data, and showed that they would imply lots of heteroskedasticity. This might, however, be an artifact of inappropriately using a linear model. Use either kernel regression (cf. §6.4.2) or local linear regression to estimate the conditional mean of waiting given duration, and see whether the apparent heteroskedasticity goes away.
6. Should local linear regression do better or worse than ordinary least squares under heteroskedasticity? What exactly would this mean, and how might you test your ideas?

[10] This is called the **Huber loss**; it continuously interpolates between looking like squared error and looking like absolute error. This means that when errors are small, it gives results very like least-squares, but it is resistant to outliers.

Chapter 8

Splines

8.1 Smoothing by Directly Penalizing Curve Flexibility

Let's go back to the problem of smoothing one-dimensional data. We imagine, that is to say, that we have data points $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$, and we want to find a function $\hat{r}(x)$ which is a good approximation to the true conditional expectation or regression function $r(x)$. Previously, we rather indirectly controlled how irregular we allowed our estimated regression curve to be, by controlling the bandwidth of our kernels. But why not be more direct, and directly control irregularity?

A natural way to do this, in one dimension, is to minimize the **spline objective function**

$$\mathscr{L}(m, \lambda) \equiv \frac{1}{n}\sum_{i=1}^{n}(y_i - m(x_i))^2 + \lambda \int dx (m''(x))^2 \tag{8.1}$$

The first term here is just the mean squared error of using the curve $m(x)$ to predict y. We know and like this; it is an old friend.

The *second* term, however, is something new for us. m'' is the second derivative of m with respect to x — it would be zero if m were linear, so this measures the **curvature** of m at x. The sign of m'' says whether the curvature is concave or convex, but we don't care about that so we square it. We then integrate this over all x to say how curved m is, on average. Finally, we multiply by λ and add that to the MSE. This is adding a **penalty** to the MSE criterion — given two functions with the same MSE, we prefer the one with less average curvature. In fact, we are willing to accept changes in m that increase the MSE by 1 unit if they also reduce the average curvature by at least λ.

The *solution* to this minimization problem,

$$\hat{r}_\lambda = \operatorname*{argmin}_m \mathscr{L}(m, \lambda) \tag{8.2}$$

is a function of x, or curve, called a **smoothing spline**, or **smoothing spline func-**