Curve Fitting

36-600

Fall 2021

The Setting

Assume that we have n data tuples,

$$(x_i, y_i, e_i)\,,$$

where x_i is the x-coordinate, y_i is the y-coordinate, and e_i is the estimated uncertainty for datum i. The goal: to draw a "curve" through the points, a line that whose mathematical form you specify, or perhaps one that is estimated from the data themselves.

But wait, isn't that just regression?

It is! It is not necessarily *linear* regression, but it is a (weighted) regression.

But...weighted?

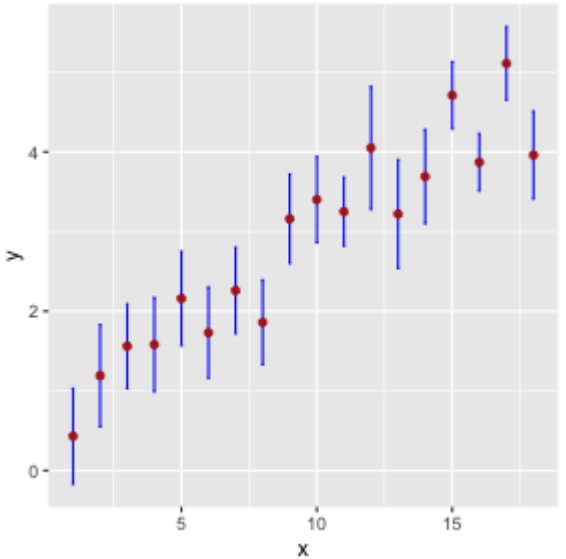
Unlike a typical linear regression setting, we have estimated uncertainties here (the e_i's). A data point with a smaller uncertainty should have more weight in the fitting process than one with a larger uncertainty. There is, as you might expect by now, no unique way to specify the weightings, but *inverse-variance* weighting is often used:

$$w_i = \frac{1}{e_i^2}\,.$$

Example

Here is a dataset that you have been given:

x	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00	11.00	12.00	13.00	14.00	15.00	16.00	17.00	18.00
y	0.43	1.19	1.56	1.58	2.16	1.73	2.26	1.86	3.16	3.40	3.25	4.05	3.22	3.69	4.71	3.87	5.11	3.96
e	0.60	0.64	0.53	0.59	0.59	0.57	0.54	0.53	0.56	0.54	0.43	0.77	0.68	0.59	0.42	0.36	0.46	0.55

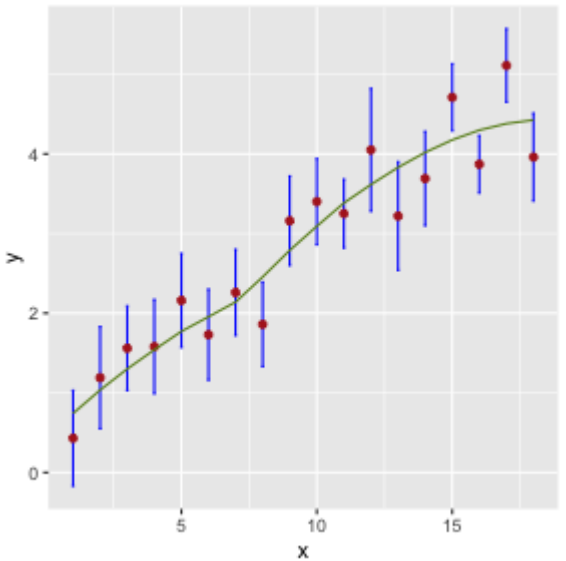


Your advisor tells you to "fit a model" to these data, and you dutifully agree.

Model 1: Local Polynomial Regression

This is what you might fit if you have no idea about what the underlying model is. The basic idea is that at each point, a polynomial is fit, with more weight being given to nearby data points and less to those farther away. (You might know of the concept as "smoothing.") The `loess` function by default does local quadratic regression, but you can change the polynomial degree.

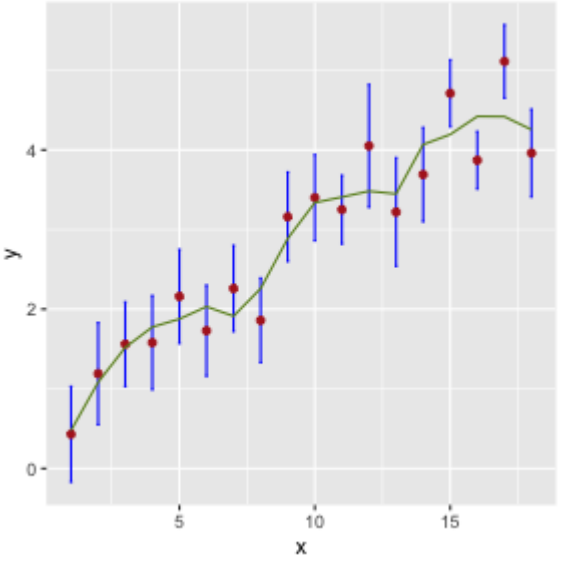
```
loess.out <- loess(y~x,data=df,weights=1/(df$e)^2)
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") +
  geom_point(color="firebrick") +
  geom_line(mapping=aes(x=loess.out$x,y=loess.out$fitted),color="olivedrab")
```



Model 1: Local Polynomial Regression

An important argument is `span` (default: 0.75). A larger value `span` means more neighboring data weigh in on the estimate made at x_o. Large `span` values produce overly smooth functions, while smaller ones produce overly noisy functions. It's the whole flexibility thing all over again.

```
loess.out <- loess(y~x,data=df,weights=1/(df$e)^2,span=0.4)
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") +
  geom_point(color="firebrick") +
  geom_line(mapping=aes(x=loess.out$x,y=loess.out$fitted),color="olivedrab")
```

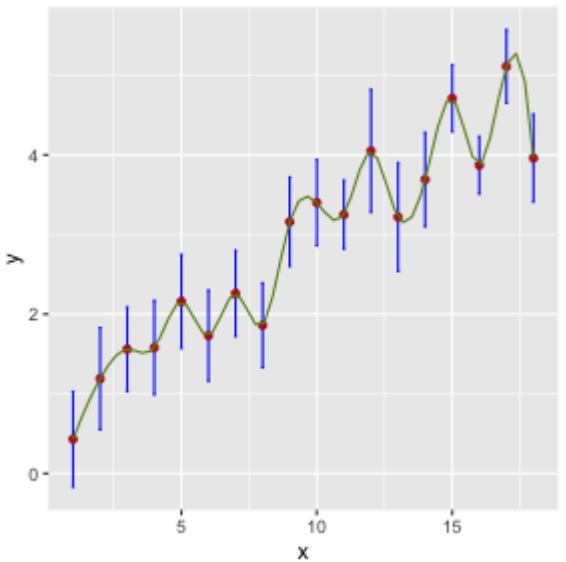


Digression: What About (Cubic) Splines?

"Because I've heard of these."

The main issue with splines is that they are not regression functions per se: they are *interpolators* that by definition *pass through every point*. That's *not* what we want here. Splines are what you use when your data points have no error and you need to interpolate y values between the data points. (Also, weighting is not involved...since you know where the data points are exactly, supposedly.)

```
spl.out <- spline(df$x,df$y)
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") +
  geom_point(color="firebrick") +
  geom_line(data=data.frame("x"=spl.out$x,"y"=spl.out$y),mapping=aes(x=spl.out$x,y=spl.out$y),color="olivedr
```



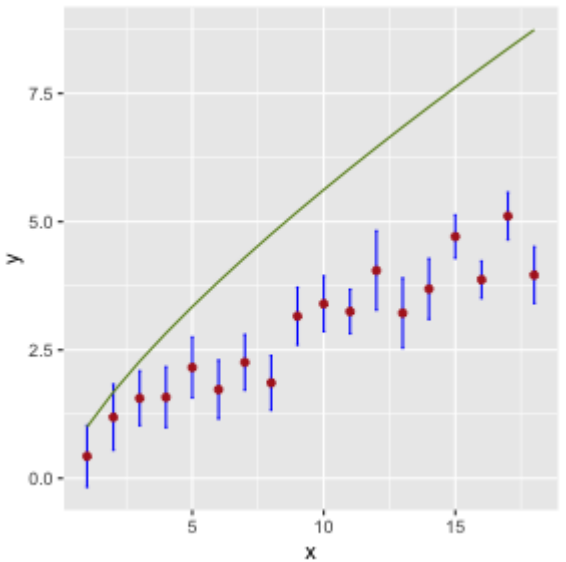
Model 2: A Specified Function

Suppose you read somewhere that for these data, the underlying model is apparently

$$f(x) = x^{\alpha}\,,$$

where α is an unknown quantity that has to be determined. OK...let's start by picking what we think might be a good value for α: 0.75. Just because.

```
mod.x <- 1:18 ; mod.y <- x^0.75
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") + geom_point(color="firebrick") +
  geom_line(mapping=aes(x=mod.x,y=mod.y),color="olivedrab")
```

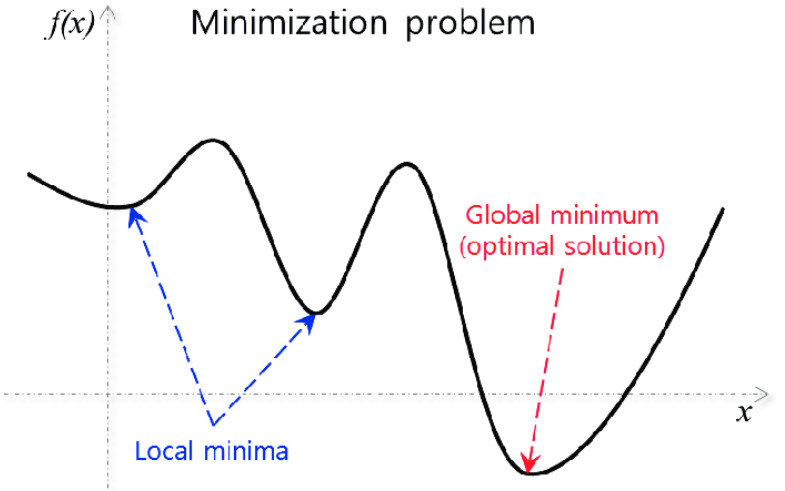


Hmm...this doesn't seem like a good choice. Let's change it...over and over again...

Digression: Optimization

...actually, let's not.

What we want to do is code an *optimizer*, a piece of code that determines the optimal value for α.



In the figure, x represents the quantity being optimized (for us, α), and $f(x)$ represents some function of x (the *objective function*) that quantifies how good the fit is. (Think of $f(x)$ as being like the MSE: smaller values are better.) You want to determine the value of x that (hopefully globally) minimizes the objective function, in a computationally efficient manner. (Trying every possible α as specified in a dense grid of values is not efficient!)

So, what is the objective function $f(x)$, in practice?

Digression: Optimization

The workhorse objective function is real-life applications is, IMHO, the *chi-square function*:

$$\chi^2 = \sum_{i=1}^{n} \frac{(y_i - \hat{y}_i)^2}{\hat{\sigma}_i^2} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \frac{1}{\hat{\sigma}_i^2} \,,$$

where $\hat{y}_i$ is the predicted value for y_i and $\hat{\sigma}_i^2$ is the estimated variance (square of the standard deviation, i.e., square of the estimated uncertainty).

Look at the way we write χ^2 to the right: it is the sum of the squared errors, weighted...and the weight is the inverse-variance weighting we were applying earlier in `loess`. So in the context of our problem, we would write χ^2 as

$$\chi^2 = \sum_{i=1}^{n} \frac{(y_i - g(x_i))^2}{e_i^2} = \sum_{i=1}^{n} \frac{(y_i - x_i^{\alpha})^2}{e_i^2} \,.$$

OK, now that we have that set, let's code an optimizer...

Digression: Optimization

...OK, slow down.

What are some of the useful properties of χ^2?

1. Because $|y_i - \hat{y}_i|$ should be $\approx e_i$ for a good model, we can take the optimized value of χ^2 and divide by $n - p$ (where p is the number of free parameters in $g(x)$...here, $p = 1$, specifically, α) and check to see if the value is approximately 1. If yes, the model is a good one (maybe not the best one, but a good one), whereas if the value is much larger than 1, the model is probably no good.

2. We can avail ourselves of a particular hypothesis test, the χ^2 goodness-of-fit test. For this test, the null hypothesis is that the optimized model provides an acceptable fit to the data, and to quantify it, we check to see if the value χ^2_{min} is consistent with a chi-square distribution for $n - p$ degrees of freedom. As written in `R` code, the p-value would be

   ```
   1 - pchisq(chi2.min,n-p)
   ```

   i.e., the integral of a chi-square distribution for $n - p$ degrees of freedom from χ^2_{min} to ∞.

Model 2: Code and Run the Optimizer

The function `optim()` is a often-used optimization function within `R`. The best way to explain what it does is to write the code first, then look at it:

```
fit.fun <- function(par,data)
{
  return( sum((data$y-(data$x)^par[1])^2/(data$e)^2) ) # chi-square, coded
}
par <- c(0.75)                       # initial guess for alpha
op.out <- suppressWarnings(optim(par,fit.fun,data=df))
op.out$value ; op.out$par[1]       # the minimum of chi-square ; the estimated alpha (truth=0.5)
```

```
## [1] 16.80373
```

```
## [1] 0.5103149
```

- In the call to `optim()`, the first two arguments are `par` (a vector of parameter values, which here is of length 1 and represents α) and `fit.fun`. We'll get back to the third argument.
- `fit.fun()` itself expects `par` as its first argument. Fine. However, we need to pass in more information for the function to actually work; we need to pass in a data frame which contains x, y, and e. So we do. But that means...
- ...back in the call to `optim()`, we need to explicit say what the parameter `data` represents. So we tack on a third argument, `data=df`, where `df` is the data frame defined earlier that has the values of x, y, and e.
- Finally, note how we make an initial guess for α (`par <- c(0.75)`). The optimizer works fine with this guess here, but beware that in more complicated problems we run the risk of not finding the global minimum for χ^2 if our initial guess is "bad."

Model 2: Is This an Acceptable Model?

- $\chi^2/(n-p) = \chi^2/(n-1) = 16.804/17 \approx 1 \Rightarrow$ Yes
- We run the chi-square GOF test:

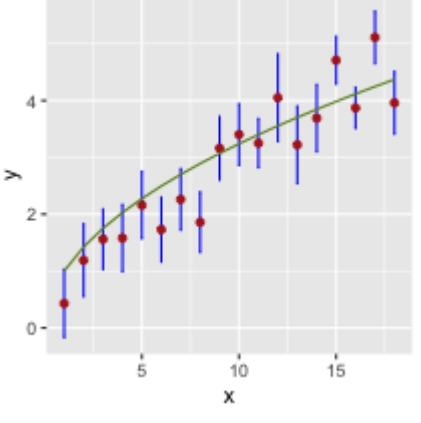
```
1 - pchisq(op.out$value,nrow(df)-1)
```

```
## [1] 0.4677384
```

The p-value is 0.46; we fail to reject the null hypothesis that our model is an acceptable one. $\Rightarrow$ Yes

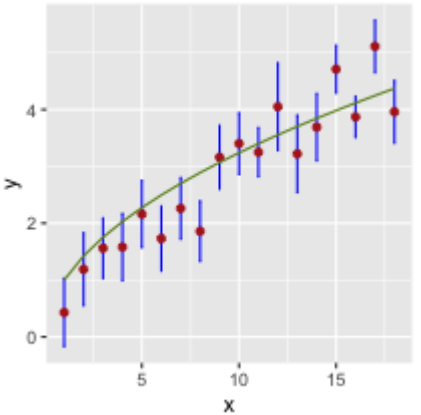
- Let's plot the result:

```
mod.x <- 1:18 ; mod.y <- x^op.out$par[1]
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") + geom_point(color="firebrick") +
  geom_line(mapping=aes(x=mod.x,y=mod.y),color="olivedrab")
```



Model 2: Wait...Let's Look at That Model Again

```
mod.x <- 1:18 ; mod.y <- x^op.out$par[1]
ggplot(data=df,mapping=aes(x=x,y=y)) +
  geom_errorbar(aes(ymin=y-e, ymax=y+e), width=.1,color="blue") + geom_point(color="firebrick") +
  geom_line(mapping=aes(x=mod.x,y=mod.y),color="olivedrab")
```



"Wait," you say. "Not all the blue lines overlap the model. That's an indication the model is bad, right?"

No. Think of the uncertainties as being estimates of the standard deviation: given x, the sampling of y has this amount of uncertainty. If the normal distribution governs the uncertainty, then we expect $\approx$ 68% of the blue lines to overlap the model. If we doubled the length of the blue lines, then we would expect $\approx$ 95% of the blue lines to overlap the model. Etc.

What do we see: 14 of 18 blue lines overlap the model, or 77.8%. Given the small sample size, this is completely consistent with an expected 68% overlap rate...the model (and the data) are just fine, thank you.