Smoothing Splines

Regression splines have much to recommend them:

- We estimate them using linear regression, which is easy and makes it easy to include other explanatory variables.
- These can include all sorts of dummy variables and/or polynomials in other predictors.
- Natural cubic splines handle boundaries well.

However, choosing the number and location of the knots involves quite a bit of judgement. Bad choices can have unfortunate consequences.

A more automated procedure is to use **smoothing splines**.

The idea is to minimize a sum of squares plus a penalty function that penalizes roughness.

The penalty function shrinks the coefficients towards linearity.

The minimization problem is

$$\min_g \left(\sum_{i=1}^{n} \big(y_i - g(x_i)\big)^2 + \lambda \int \big(g''(t)\big)^2 dt \right), \tag{1}$$

where λ is a smoothing (tuning) parameter.

The first term in (1) is a **loss function**. It will be small when all the $g(x_i)$ fit the data well.

The second term in (1) is a **penalty term**. It will be small whenever the second derivative of $g(t)$ is small for all t.

Here t is just the argument over which we integrate.

If all the second derivatives were zero, then $g(x)$ would be linear.

- When $\lambda = \infty$, minimizing (1) yields an OLS regression line.
- When $\lambda = 0$, minimizing (1) yields a perfect fit if every x_i is unique, because $g(x)$ can be any function.

It can be shown that the minimizer of (1) is a natural cubic spline with knots at all of the (unique) x_i.

- However, it is not the natural cubic spline that we would get if we used `ns()` to create N splines and then used OLS.
- Instead, the spline coefficients are shrunk towards the linear least squares fit because of the penalty term.
- This is a bit like ridge regression.
- As with Shiller lags and seasonal dummies, it is second derivatives not coefficients that are being penalized.

What we are actually minimizing can be written as

$$(\boldsymbol{y} - \boldsymbol{N\theta})^\top(\boldsymbol{y} - \boldsymbol{N\theta}) + \lambda\boldsymbol{\theta}^\top\boldsymbol{\Omega}_{\boldsymbol{N}}\boldsymbol{\theta}. \tag{2}$$

Here $\boldsymbol{N}$ is an $n \times n$ matrix with typical element $N_j(x_i)$, where the $N_j(x)$ form an n-dimensional set of basis functions for the family of natural cubic splines.

The matrix $\mathbf{\Omega_N}$ can be defined as

$$(\mathbf{\Omega_N})_{jk} \equiv \int N_j''(t) N_k''(t) dt. \tag{3}$$

This involves the second derivatives of the basis functions.

- As λ increases from 0 to ∞, the **effective degrees of freedom** (or **edf**) decreases from n_u to 2.
- Here n_u is the number of unique values of x_i. In the earnings dataset, there are 63 unique values of `age`.
- When edf $= n_u$, we are setting $g(x_i) = \bar{y}_i$ for all observations, where $\bar{y}_i$ is the average value of y_i for all observations with the same x_i as the observation (x_i, y_i).
- Thus, for the earnings dataset, the most flexible model would be equivalent to regressing log earnings on 63 age dummies.
- When edf $= 2$, we are estimating a linear regression by OLS.

In practice, we do not actually need to put knots at every value of x_i. We can get away with a much smaller number.

If there are n_u unique values, we actually need a number on the order of $O(n_u^{1/5})$ for $n_u > 49$.

The function `smooth.spline()` will estimate smoothing splines, using a variety of computational tricks.

It actually uses *B*-splines instead of natural cubic splines, for computational reasons.

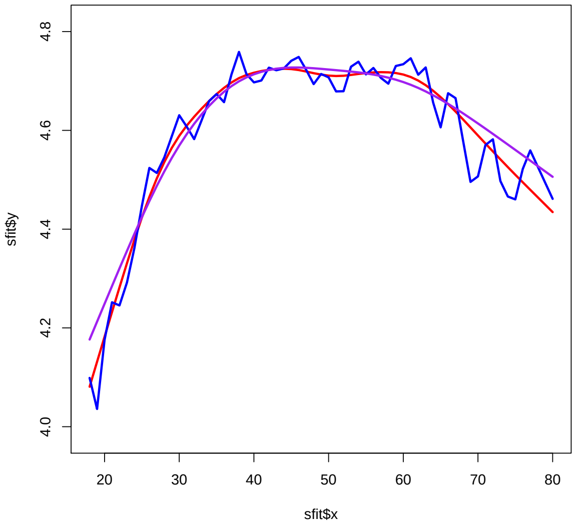
Although `smooth.spline()` lets you specify λ, it prefers to use a parameter called `spar` that is typically in the 0-1 interval.

There is a relationship between `spar` and λ, but it is quite complicated.

LOO cross-validation can be used, and it works very efficiently; see Subsection 7.5.2. But it only works for unique x_i.

Figure 13.1 shows the results of estimating the relationship between age and the log of earnings using `smooth.spline()`.

Figure 13.1 — Smoothing Splines for Log Earnings



The red curve was obtained by letting the function pick `spar`. It chose 0.669875 using cross-validation.

It refused to use leave-one-out cross-validation because the x_i were not unique.

Instead, it used **generalized cross-validation**; see ESL, Chapter 7.

The blue curve used $\texttt{spar} = 0.2$, which is evidently much too small.

The purple curve used $\texttt{spar} = 0.8$, which appears to be somewhat too big. The fitted values are too large at both ends.

How to obtain confidence intervals will be discussed in the context of generalized additive models.

- The main advantage of using smoothing splines is that there is no need to choose the number and location of the knots.
- One could use a smoothing spline first, then pick the knots so that the natural cubic spline yields similar results.

Smoothing Splines for Logit

Instead of $g(x)$ denoting the fitted value conditional on x, it now denotes the log of the odds:

$$\log \frac{\Pr(y = 1 \mid x)}{\Pr(y = 0 \mid x)} = g(x). \tag{4}$$

Therefore

$$p(x) \equiv \Pr(y = 1 \mid x) = \frac{\exp\big(g(x)\big)}{1 + \exp\big(g(x)\big)}. \tag{5}$$

The penalized criterion function is based on the loglikelihood function:

$$\sum_{i=1}^{n} \Big(y_i \log\left(p(x_i)\right) + (1 - y_i) \log\left(1 - p(x_i)\right) \Big) - \frac{1}{2}\lambda \int \left(g''(t)\right)^2 dt. \tag{6}$$

This is minimized by making g a natural cubic spline with knots at every unique value of x.

The predicted log odds at x is simply

$$\hat{g}(x) = \sum_{j=1}^{n} \hat{\theta}_j N_j(x). \tag{7}$$

Of course, if we set $\lambda = 0$, maximizing (6) would yield the maximum likelihood logit estimates.

- Maximizing (6) requires nonlinear optimization, which would be prohibitively expensive for large n if we actually used n basis functions.
- But, as in the regression case, we can get away with far fewer than n basis functions when n is large.

Of course, we could also use ordinary or natural cubic splines with a binary dependent variable.

We would need to choose how many knots and their locations. The basis functions could then be constructed using the `bs()` or `ns()` functions in the usual way.

Kernel Regression

In economics, the most widely-used approach to nonparametric regression is **kernel regression**.

Suppose that two random variables y and x are jointly distributed with joint density $f(y, x)$. We want to estimate $\mu(x_0) \equiv \mathrm{E}(y \mid x_0)$ for (many) points x_0. This is a regression function.

The **Nadaraya-Watson**, or **locally constant** (LC), estimator of $\mu(x_0)$ is

$$\hat{\mu}_h(x_0) = \frac{\sum_{i=1}^{N} y_i k_i(x_0)}{\sum_{i=1}^{N} k_i(x_0)}, \quad k_i(x_0) \equiv k\left(\frac{x_i - x_0}{h}\right), \tag{8}$$

where $k(\cdot)$ is a kernel function.

If both were divided by nh, the numerator of (8) would be a weighted average of the values of y_i in the neighborhood of x_0, and the denominator would be a kernel estimate of the density of x at x_0.

The Nadaraya-Watson estimator $\hat{\mu}_h(x_0)$ is the solution to the estimating equation

$$\sum_{i=1}^{n} k_i(x_0)\big(y_i - \hat{\mu}_h(x_0)\big) = 0. \tag{9}$$

This is the empirical counterpart of setting a weighted average of the $y_i - \mu(x_0)$ to zero.

But the conditional expectation of y_i is not $\mu(x_0)$. It is $\mu(x_i)$. So we are taking a weighted average of $\mu(x_0)$, plus noise, instead of $\mu(x_i)$ plus noise. This causes bias.

A better approximation would be the two-term Taylor expansion $\mu(x_0) + \mu'(x_0)(x_i - x_0)$, in which both $\mu(x_0)$ and $\mu'(x_0)$ are unknown.

Both of these unknowns can be estimated simultaneously by solving two estimating equations jointly.

The first estimating equation is

$$\sum_{i=1}^{n} k_i(x_0)\big(y_i - \mu(x_0) - \mu'(x_0)(x_i - x_0)\big) = 0, \tag{10}$$

which involves setting a weighted sum of the residuals to zero. The second is

$$\sum_{i=1}^{n} k_i(x_0)(x_i - x_0)\big(y_i - \mu(x_0) - \mu'(x_0)(x_i - x_0)\big) = 0. \tag{11}$$

which involves making the weighted residuals orthogonal to the weighted regressor.

The simplest way to solve these equations is to run the linear regression

$$k_i^{1/2}(x_0)y_i = k_i^{1/2}(x_0)\mu(x_0) + k_i^{1/2}(x_0)\mu'(x_0)(x_i - x_0) + \text{residual}. \tag{12}$$

so as to obtain the **locally linear estimator** of $\mu(x_0)$, which is just the first estimated coefficient, say $\hat{\mu}_h^{\mathrm{LL}}(x_0)$.

Regression (12) is called a **local(ly) linear regression** (LL).

We must run regression (12) for every value of x_0 at which we wish to evaluate $\mu(x_0)$.

- How many observations this regression has depends on the kernel and the bandwidth.
- For a Gaussian kernel, regression (12) always has n observations.
- For an Epanechnikov kernel, it typically has a smaller number, perhaps much smaller when h is small.
- Observations near x_0 get large weights, and observations far away get small weights. With kernels such as Epanechnikov, the latter actually get zero weights.

We could add additional, higher-order terms, such as $(x_i - x_0)^2$, $(x_i - x_0)^3$, and so on, in order to estimate $\mu''(x_0)$, $\mu'''(x_0)$, and so on.

If we do that, we get a **locally polynomial estimator** (LP).

Adding additional terms reduces bias but can increase variance, because we have to estimate more parameters.

- If $\mu(x)$ were actually linear, then using locally linear regression would involve no bias.
- If $\mu(x)$ were actually quadratic, then using locally quadratic regression would involve no bias.
- Unless the unknown function is very nonlinear, with a second derivative that changes sign at least once, (or n is really large), we probably don't want to go beyond the locally quadratic estimator.

In many cases, the locally linear estimator seems to be the best choice.

The optimal bandwidth h depends on d, the degree of the polynomial. The bigger is d, the larger we want h to be.

Making h larger reduces variance, and doing so causes a smaller increase in bias the larger is d.

Thus we expect h to be larger for LL than for LC kernel regression.

In theory, the optimal bandwidth depends on a lot of things. For given d, the optimal h depends on

- the sample size, with a factor of $n^{-1/5}$;
- the density of x, the regressor;
- the variance of the disturbances;
- whether the regression is LC, LL, or LP;
- the shape of the regression function; and
- the kernel.

Instead of trying to figure out the optimal h based on (estimates of) these things, it is common to use LOO cross-validation.

We simply choose h to minimize

$$\mathrm{LSCV}(h) = \sum_{i=1}^{n} \bigl(y_i - \hat{\mu}_{-i}(x_i)\bigr)^2, \tag{13}$$

where $\hat{\mu}_{-i}(x_i)$ is the leave-one-out estimate.

In the case of the locally constant (LC) estimator,

$$\hat{\mu}_{-i}(x_i) = \frac{\sum_{j \neq i}^{n} y_j k_j(x_i, h)}{\sum_{j \neq i}^{n} k_j(x_i, h)}, \tag{14}$$

where $k_j(x_i, h)$ is the kernel function with bandwidth h for the point x_i, evaluated at the point x_j.

For each value of i, $\hat{\mu}_{-i}(x_i)$ in (14) is just the kernel-weighted average of all the y_j for $j \neq i$.

As usual with cross-validation, we are computing fitted values for each observation omitting that observation from the sample.

This means that

$$\mathrm{E}\big(y_i - \hat{\mu}_{-i}(x_i)\big)^2 > \mathrm{E}\big(y_i - \hat{\mu}(x_i)\big)^2. \tag{15}$$

The l.h.s. is an over-estimate of σ^2, and the r.h.s. is probably an under-estimate, unless there is a lot of bias.

The np package, by Tristan Hayfield and Jeff Racine, implements many types of kernel density, distribution, and regression estimators.

See Jeffrey S. Racine, *An Introduction to the Advanced Theory of Nonparametric Econometrics: A Replicable Approach using R*, 2019.

The npreg function estimates many kernel regression models.

Remember the automobile data, where we want to use horsepower (hp) to predict gallons per mile (gpm).

The command

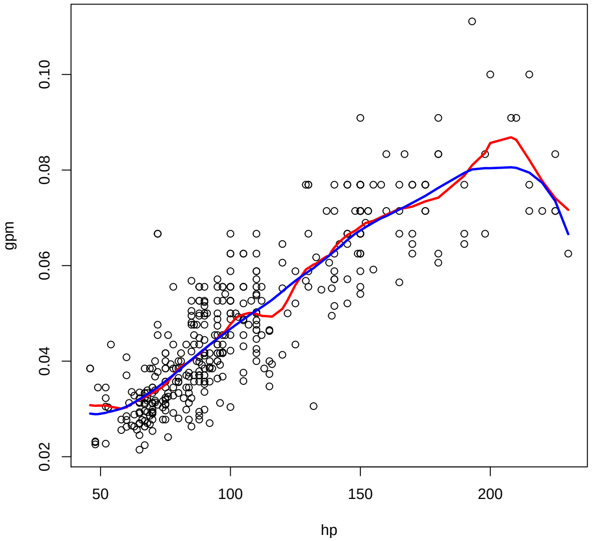
```
npfit <- npreg(gpm~hp, ckertype="epanechnikov",
                    regtype="lc")
```

runs a locally constant kernel regression of gpm on hp, using an Epanechnikov kernel, with the bandwidth chosen by cross-validation.

If we replace "lc" with "ll", it runs a locally linear regression instead.

Figure 13.2 shows the 392 data points, and both sets of predictions.

Figure 13.2 — LC (red) and LL (blue) Estimates



The LL fits look much better than the LC ones, but they both do strange things for large values of `hp`.

The `np` package does a great many things. In particular, the dependent variable can be binary, and there can be several regressors.

With two or more regressors, they should be standardized. Otherwise, whichever is in smaller units will get more weight.

It is impossible to maintain both low bias and low variance, unless the sample has a great many points near every interesting value of $\boldsymbol{x}_0$.

For $p >> 1$, this is extremely difficult to achieve unless n is very large.

- For bias to be small, we need all the points that get much weight to be near $\boldsymbol{x}_0$, which implies that h must be small.
- For variance to be small, we need there to be a lot of points that receive large weights, which usually implies that h must be large, unless n is very large or the data tend to be dense around $\boldsymbol{x}_0$.

The "curse of dimensionality" is the same as for KNN.

Local Likelihood

Any parametric model can be converted to a local one by using weights that vary across observations according to the value of $\boldsymbol{x}$.

In particular, it is easy to turn any globally linear model into a locally linear one by using kernel weights.

Suppose the model has a loglikelihood function

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \ell(y_i, \boldsymbol{x}_i^\top \boldsymbol{\beta}). \tag{16}$$

An obvious example is a logit or probit model. Then we can estimate a locally linear version by maximizing

$$\ell\big(\boldsymbol{\beta}(\boldsymbol{x}_0)\big) = \sum_{i=1}^{n} k(\boldsymbol{x}_i - \boldsymbol{x}_0, h)\,\ell\big(y_i, \boldsymbol{x}_i^\top \boldsymbol{\beta}(\boldsymbol{x}_0)\big). \tag{17}$$

Observe that each local maximization estimates a different parameter vector $\boldsymbol{\beta}(\boldsymbol{x}_0)$.

In (17), we weight contributions to the loglikelihood instead of squared residuals.

The weights are given by the kernel function $k(\boldsymbol{x}_i - \boldsymbol{x}_0, h)$, which depends on the difference between $\boldsymbol{x}_i$ and $\boldsymbol{x}_0$ and the bandwidth h.

Recall that the weight given to x_i increases with h and decreases with $|x_i - x_0|$. So points far from x_0 receive little weight unless h is large.

For every specified value of $\boldsymbol{x}_0$, we obtain a different vector of estimates $\hat{\boldsymbol{\beta}}(\boldsymbol{x}_0)$.

The `np` package can estimate various models of this sort.

It can also do all sorts of other things. There is a lot of documentation to wade through.

Smoothing Splines versus Kernel Regression

It is perhaps not obvious, but smoothing splines, like kernel regression, are *local* methods.

The smoothing matrix $\boldsymbol{S}_\lambda$ gives large weights to points near x_0 and small weights to points far from it.

Figure 13.3, which is taken from Chapter 5 of ESL, illustrates this.

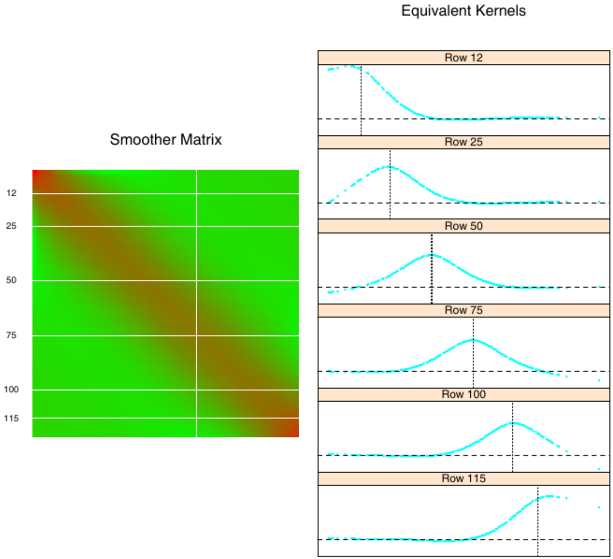
The colorful square shows $\boldsymbol{S}_\lambda$ for a particular problem with around 125 observations, where the x_i have been sorted.

The “equivalent kernels” on the right show the implied weights for a number of rows of the smoother matrix.

These equivalent kernels look very much like real kernels, although they change a bit as x_0 changes, which is a good thing.

They sometimes take on small negative values.

Figure 13.3 — Smoothing Splines vs. Kernels



Comparing Fully Automated Methods

Smoothing splines with λ chosen by some sort of cross-validation and kernel regression with h chosen in the same way are both fully automatic.

How do they compare for the earnings data? See Figure 13.4.

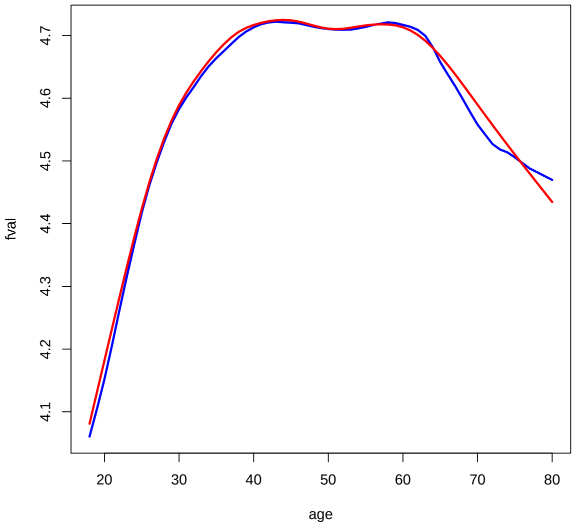
The smoothing spline chosen by `smooth.spline` is in red, and the locally linear kernel regression (Epanechnikov kernel) chosen by `npreg` is in blue.

The optimal `spar` for the smoothing spline was 0.670. This took just 0.0023 seconds.

The optimal bandwidth (h) for the kernel regression was 2.846. This took 5.10 seconds.

For large n, smoothing splines should be much cheaper than kernel regression.

Figure 13.4 — Smoothing Spline vs. Kernel Regression


fval
age