Lecture 11: Splines

36-402, Advanced Data Analysis

15 February 2011

Reading: Chapter 11 in Faraway; chapter 2, pp. 49–73 in Berk.

Contents

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**

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

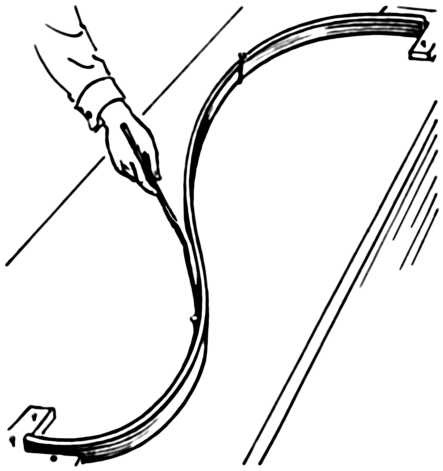


Figure 1: A craftsman's spline, from Wikipedia, s.v. "Flat spline".

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,

$$\hat{r}_\lambda = \operatorname*{argmin}_m \mathcal{L}(m, \lambda) \tag{2}$$

is a function of x, or curve, called a **smoothing spline**, or **smoothing spline function**[1].

[1]The name "spline" actually comes from a simple tool used by craftsmen to draw smooth curves, which was a thin strip of a flexible material like a soft wood, as in Figure 1. (A few years ago, when the gas company dug up my front yard, the contractors they hired to put the driveway back used a plywood board to give a smooth, outward-curve edge to the new driveway. The "knots" metal stakes which the board was placed between, and the curve of the board was a spline, and they poured concrete to one side of the board, which they left standing until the concrete dried.) Bending such a material takes energy — the stiffer the material, the more energy has to go into bending it through the same shape, and so the straighter the curve it will make between given points. For smoothing splines, using a stiffer material corresponds to increasing λ.

It is possible to show that all solutions, no matter what the initial data are, are piecewise cubic polynomials which are continuous and have continuous first and second derivatives — i.e., not only is $\hat{r}$ continuous, so are $\hat{r}'$ and $\hat{r}''$. The boundaries between the pieces are located at the original data points. These are called, somewhat obscure, the **knots** of the spline. The function continuous beyond the largest and smallest data points, but it is always linear on those regions.[2] I will not attempt to prove this.

I will also assert, without proof, that such piecewise cubic polynomials can approximate any well-behaved function arbitrarily closely, given enough pieces. Finally, I will assert that smoothing splines are linear smoothers, in the sense given in earlier lectures: predicted values are always linear combinations of the original response values y_i.

1.1 The Meaning of the Splines

Look back to the optimization problem. As $\lambda \to \infty$, having any curvature at all becomes infinitely penalized, and only linear functions are allowed. But we know how to minimize mean squared error with linear functions, that's OLS. So we understand that limit.

On the other hand, as $\lambda \to 0$, we decide that we don't care about curvature. In that case, we can always come up with a function which just interpolates between the data points, an **interpolation spline** passing exactly through each point. More specifically, of the infinitely many functions which interpolate between those points, we pick the one with the minimum average curvature.

At intermediate values of λ, $\hat{r}_\lambda$ becomes a function which compromises between having low curvature, and bending to approach all the data points closely (on average). The larger we make λ, the more curvature is penalized. There is a bias-variance trade-off here. As λ grows, the spline becomes less sensitive to the data, with lower variance to its predictions but more bias. As λ shrinks, so does bias, but variance grows. For consistency, we want to let $\lambda \to 0$ as $n \to \infty$, just as, with kernel smoothing, we let the bandwidth $h \to 0$ while $n \to \infty$.

We can also think of the smoothing spline as the function which minimizes the mean squared error, subject to a constraint on the average curvature. This turns on a general corresponds between penalized optimization and optimization under constraints, which is explored in the appendix. The short version is that each level of λ corresponds to imposing a cap on how much curvature the function is allowed to have, on average, and the spline we fit with that λ is the MSE-minimizing curve subject to that constraint. As we get more data, we have more information about the true regression function and can relax the constraint (let λ shrink) without losing reliable estimation.

It will not surprise you to learn that we select λ by cross-validation. Ordinary k-fold CV is entirely possible, but leave-one-out CV works quite well for splines. In fact, the default in most spline software is either leave-one-out CV, or an

[2] Can you explain why it is linear outside the data range, in terms of the optimization problem?

even faster approximation called "generalized cross-validation" or GCV. The details of how to rapidly compute the LOOCV or GCV scores are not especially important for us, but can be found, if you want them, in many books, such as Simonoff (1996, §5.6.3).

2 An Example

The default R function for fitting a smoothing spline is called `smooth.spline`. The syntax is

```
smooth.spline(x, y, cv=FALSE)
```

where `x` should be a vector of values for input variable, `y` is a vector of values for the response (in the same order), and the switch `cv` controls whether to pick λ by generalized cross-validation (the default) or by leave-one-out cross-validation. The object which `smooth.spline` returns has an `$x` component, *re-arranged in increasing order*, a `$y` component of fitted values, a `$yin` component of original values, etc. See `help(smooth.spline)` for more.

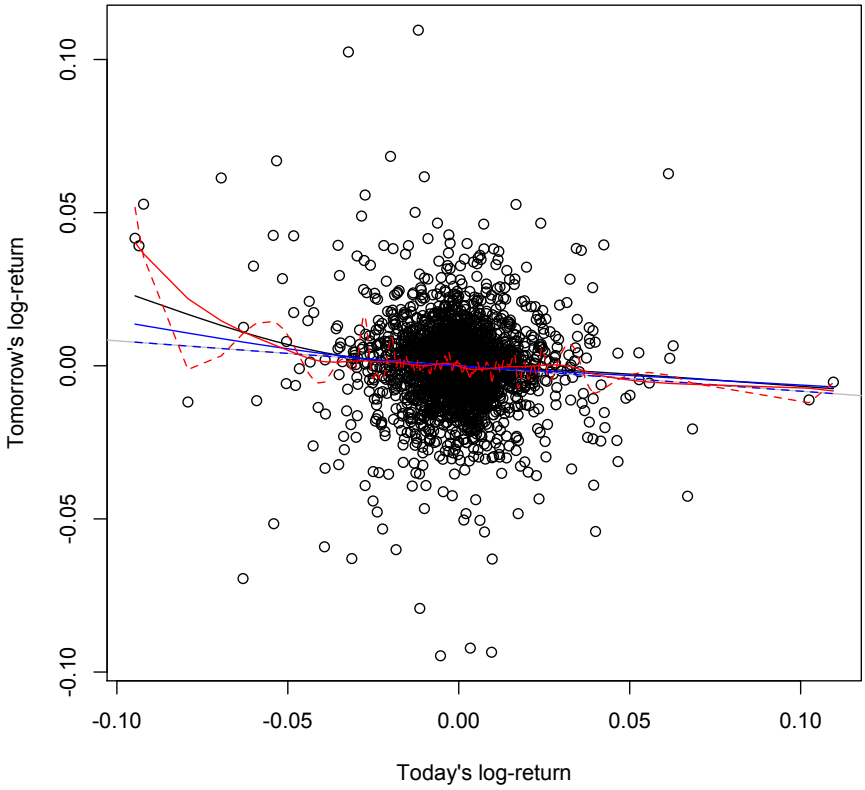
Figure 2 revisits the stock-market data from Homework 4 (and lecture 7). The vector `sp` contains the log-returns of the S & P 500 stock index on 2528 consecutive trading days; we want to use the log-returns on one day to predict what they will be on the next. The horizontal axis in the figure shows the log-returns for each of 2527 days t, and the vertical axis shows the corresponding log-return for the succeeding day $t - 1$q You saw in the homework that a linear model fitted to this data displays a slope of -0.0822 (grey line in the figure). Fitting a smoothing spline with cross-validation selects $\lambda = 0.0513$, and the black curve:

```
> sp.today <- sp[-length(sp)]
> sp.tomorrow <- sp[-1]
> sp.spline <- smooth.spline(x=sp[-length(sp)],y=sp[-1],cv=TRUE)
Warning message:
In smooth.spline(sp[-length(sp)], sp[-1], cv = TRUE) :
  crossvalidation with non-unique 'x' values seems doubtful
> sp.spline
Call:
smooth.spline(x = sp[-length(sp)], y = sp[-1], cv = TRUE)

Smoothing Parameter  spar= 1.389486  lambda= 0.05129822 (14 iterations)
Equivalent Degrees of Freedom (Df): 4.206137
Penalized Criterion: 0.4885528
PRESS: 0.0001949005
> sp.spline$lambda
[1] 0.05129822
```

(PRESS is the "prediction sum of squares", i.e., the sum of the squared leave-one-out prediction errors. Also, the warning about cross-validation, while well-intentioned, is caused here by there being just two days with log-returns of zero.) This is the curve shown in black in the figure. The curves shown in blue are for large values of λ, and clearly approach the linear regression; the curves shown in red are for smaller values of λ.

The spline can also be used for prediction. For instance, if we want to know what the return to expect following a day when the log return was $+0.01$,




```
sp.today <- sp[-length(sp)]
sp.tomorrow <- sp[-1]
plot(sp.today,sp.tomorrow,xlab="Today's log-return",
     ylab="Tomorrow's log-return")
abline(lm(sp.tomorrow ~ sp.today),col="grey")
sp.spline <- smooth.spline(x=sp.today,y=sp.tomorrow,cv=TRUE)
lines(sp.spline)
lines(smooth.spline(sp.today,sp.tomorrow,spar=1.5),col="blue")
lines(smooth.spline(sp.today,sp.tomorrow,spar=2),col="blue",lty=2)
lines(smooth.spline(sp.today,sp.tomorrow,spar=1.1),col="red")
lines(smooth.spline(sp.today,sp.tomorrow,spar=0.5),col="red",lty=2)
```

Figure 2: The S& P 500 log-returns data (circles), together with the OLS linear regression (grey line), the spline selected by cross-validation (solid black curve, $\lambda = 0.0513$), some more smoothed splines (blue, $\lambda = 0.322$ and 1320) and some less smooth splines (red, $\lambda = 4.15 \times 10^{-4}$ and 1.92×10^{-8}). Incoveniently, `smooth.spline` does not let us control λ directly, but rather a somewhat complicated but basically exponential transformation of it called `spar`. See `help(smooth.spline)` for the gory details. The equivalent λ can be extracted from the return value, e.g., `smooth.spline(sp.today,sp.tomorrow,spar=2)$lambda`.

```
> predict(sp.spline,x=0.01)
$x
[1] 0.01
$y
[1] -0.0007169499
```

i.e., a very slightly negative log-return. (Giving both an `x` and a `y` value like this means that we can directly feed this output into many graphics routines, like `points` or `lines`.)

2.1 Confidence Bands for Splines

Continuing the example, the smoothing spline selected by cross-validation has a negative slope everywhere, like the regression line, but it's asymmetric — the slope is more negative to the left, and then levels off towards the regression line. (See Figure 2 again.) Is this real, or might the asymmetry be a sampling artifact?

We'll investigate by finding confidence bands for the spline, much as we did in Lecture 8 for kernel regression. Again, we need to bootstrap, and we can do it either by resampling the residuals or resampling whole data points. Let's take the latter approach, which assumes less about the data. We'll need a simulator:

```
sp.frame <- data.frame(today=sp.today,tomorrow=sp.tomorrow)
sp.resampler <- function() {
  n <- nrow(sp.frame)
  resample.rows <- sample(1:n,size=n,replace=TRUE)
  return(sp.frame[resample.rows,])
}
```

This treats the points in the scatterplot as a complete population, and then draws a sample from them, with replacement, just as large as the original. We'll also need an estimator. What we want to do is get a whole bunch of spline curves, one on each simulated data set. But since the values of the input variable will change from one simulation to another, to make everything comparable we'll evaluate each spline function on a fixed grid of points, that runs along the range of the data.

```
sp.spline.estimator <- function(data,m=300) {
  # Fit spline to data, with cross-validation to pick lambda
  fit <- smooth.spline(x=data[,1],y=data[,2],cv=TRUE)
  # Set up a grid of m evenly-spaced points on which to evaluate the spline
  eval.grid <- seq(from=min(sp.today),to=max(sp.today),length.out=m)
    # Slightly inefficient to re-define the same grid every time we call this,
    # but not a big overhead
  # Do the prediction and return the predicted values
  return(predict(fit,x=eval.grid)$y)  # We only want the predicted values
}
```

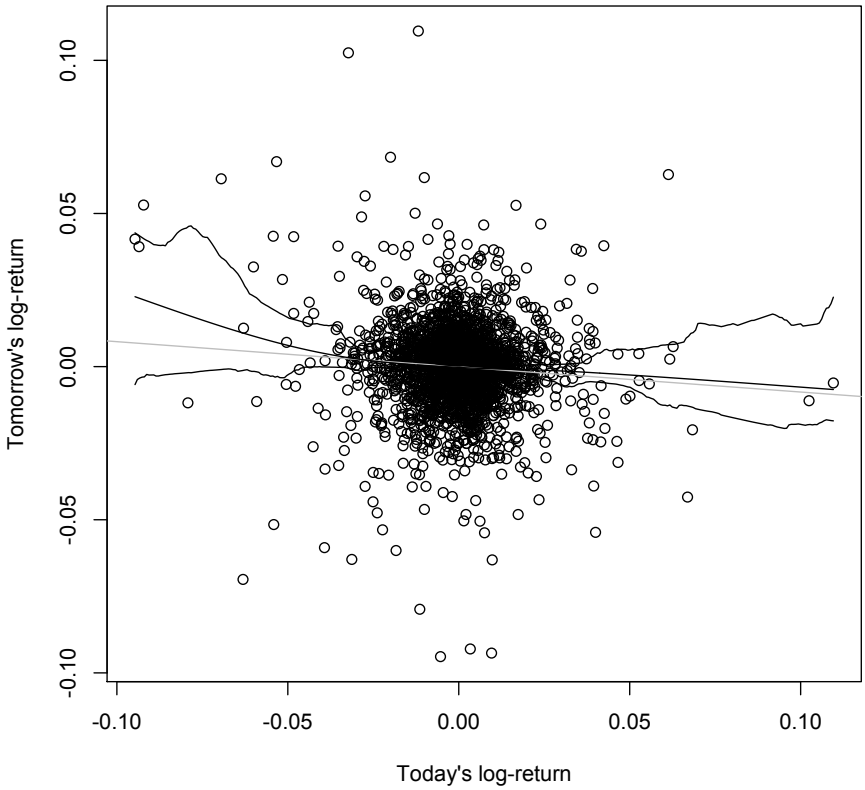
This sets the number of evaluation points to 300, which is large enough to give visually smooth curves, but not so large as to be computationally unwieldly.

Now put these together to get confidence bands:

```
sp.spline.cis <- function(B,alpha,m=300) {
  spline.main <- sp.spline.estimator(sp.frame,m=m)
  # Draw B boottrap samples, fit the spline to each
  spline.boots <- replicate(B,sp.spline.estimator(sp.resampler(),m=m))
    # Result has m rows and B columns
  cis.lower <- 2*spline.main - apply(spline.boots,1,quantile,probs=1-alpha/2)
  cis.upper <- 2*spline.main - apply(spline.boots,1,quantile,probs=alpha/2)
  return(list(main.curve=spline.main,lower.ci=cis.lower,upper.ci=cis.upper,
              x=seq(from=min(sp.today),to=max(sp.today),length.out=m)))
}
```

The return value here is a list which includes the original fitted curve, the lower and upper confidence limits, and the points at which all the functions were evaluated.

Figure 3 shows the resulting 95% confidence limits, based on `B=1000` bootstrap replications. (Doing all the bootstrapping took 45 seconds on my laptop.) These are pretty clearly asymmetric in the same way as the curve fit to the whole data, but notice how wide they are, and how they get wider the further we go from the center of the distribution in either direction.




```
sp.cis <- sp.spline.cis(B=1000,alpha=0.05)
plot(sp.today,sp.tomorrow,xlab="Today's log-return",
     ylab="Tomorrow's log-return")
abline(lm(sp.tomorrow ~ sp.today),col="grey")
lines(x=sp.cis$x,y=sp.cis$main.curve)
lines(x=sp.cis$x,y=sp.cis$lower.ci)
lines(x=sp.cis$x,y=sp.cis$upper.ci)
```

Figure 3: Bootstrapped pointwise confidence band for the smoothing spline of the S & P 500 data, as in Figure 2. The 95% confidence limits around the main spline estimate are based on 1000 bootstrap re-samplings of the data points in the scatterplot.

3 Basis Functions

This section is optional reading.

Splines, I said, are piecewise cubic polynomials. To see how to fit them, let's think about how to fit a global cubic polynomial. We would define four **basis functions**,

$$B_1(x) = 1 \tag{3}$$
$$B_2(x) = x \tag{4}$$
$$B_3(x) = x^2 \tag{5}$$
$$B_4(x) = x^3 \tag{6}$$

with the hypothesis being that the regression function is a weight sum of these,

$$r(x) = \sum_{j=1}^{4} \beta_j B_j(x) \tag{7}$$

That is, the regression would be linear in the *transformed* variable $B_1(x), \ldots B_4(x)$, even though it is nonlinear in x.

To estimate the coefficients of the cubic polynomial, we would apply each basis function to each data point x_i and gather the results in an $n \times 4$ matrix $\mathbf{B}$,

$$B_{ij} = B_j(x_i) \tag{8}$$

Then we would do OLS using the $\mathbf{B}$ matrix in place of the usual data matrix $\mathbf{x}$:

$$\hat{\beta} = (\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T\mathbf{y} \tag{9}$$

Since splines are piecewise cubics, things proceed similarly, but we need to be a little more careful in defining the basis functions. Recall that we have n values of the input variable x, $x_1, x_2, \ldots x_n$. For the rest of this section, I will assume that these are in increasing order, because it simplifies the notation. These n "knots" define $n+1$ pieces or segments $n-1$ of them between the knots, one from $-\infty$ to x_1, and one from x_n to $+\infty$. A third-order polynomial on each segment would seem to need a constant, linear, quadratic and cubic term per segment. So the segment running from x_i to x_{i+1} would need the basis functions

$$\mathbf{1}_{(x_i,x_{i+1})}(x),\ x\mathbf{1}_{(x_i,x_{i+1})}(x),\ x^2\mathbf{1}_{(x_i,x_{i+1})}(x),\ x^3\mathbf{1}_{(x_i,x_{i+1})}(x) \tag{10}$$

where as usual the indicator function $\mathbf{1}_{(x_i,x_{i+1})}(x)$ is 1 if $x \in (x_i, x_{i+1})$ and 0 otherwise. This makes it seem like we need $4(n+1) = 4n+4$ basis functions.

However, we know from linear algebra that the number of basis vectors we need is equal to the number of dimensions of the vector space. The number of dimensions for an arbitrary piecewise cubic with $n+1$ segments is indeed $4n+4$, but splines are constrained to be smooth. The spline must be continuous, which

means that at each x_i, the value of the cubic from the left, defined on (x_{i-1}, x_i), must match the value of the cubic from the right, defined on (x_i, x_{i+1}). This gives us one constraint per data point, reducing the dimensionality to at most $3n+4$. Since the first and second derivatives are also continuous, we come down to a dimensionality of just $n+4$. Finally, we know that the spline function is linear outside the range of the data, i.e., on $(-\infty, x_1)$ and on (x_n, ∞), lowering the dimension to n. There are no more constraints, so we end up needing only n basis functions. And in fact, from linear algebra, any set of n piecewise cubic functions which are linearly independent[3] can be used as a basis. One common choice is

$$
\begin{aligned}
B_1(x) &= 1 && (11)\\
B_2(x) &= x && (12)\\
B_{i+2}(x) &= \frac{(x-x_i)_+^3 - (x-x_n)_+^3}{x_n - x_i} - \frac{(x-x_{n-1})_+^3 - (x-x_n)_+^3}{x_n - x_{n-1}} && (13)
\end{aligned}
$$

where $(a)_+ = a$ if $a > 0$, and $= 0$ otherwise. This rather unintuitive-looking basis has the nice property that the second and third derivatives of each B_j are zero outside the interval (x_1, x_n).

Now that we have our basis functions, we can once again write the spline as a weighted sum of them,

$$m(x) = \sum_{j=1}^{m} \beta_j B_j(x) \tag{14}$$

and put together the matrix $\mathbf{B}$ where $B_{ij} = B_j(x_i)$. We can write the spline objective function in terms of the basis functions,

$$n\mathcal{L} = (\mathbf{y} - \mathbf{B}\beta)^T(\mathbf{y} - \mathbf{B}\beta) + n\lambda\beta^T\Omega\beta \tag{15}$$

where the matrix Ω encodes information about the curvature of the basis functions:

$$\Omega_{jk} = \int dx B_j''(x) B_k''(x) \tag{16}$$

Notice that only the quadratic and cubic basis functions will contribute to Ω. With the choice of basis above, the second derivatives are non-zero on, at most, the interval (x_1, x_n), so each of the integrals in Ω is going to be finite. This is something we (or, realistically, R) can calculate *once*, no matter what λ is. Now we can find the smoothing spline by differentiating with respect to β:

$$
\begin{aligned}
0 &= -2\mathbf{B}^T\mathbf{y} + 2\mathbf{B}^T\mathbf{B}\hat{\beta} + 2n\lambda\Omega\hat{\beta} && (17)\\
\mathbf{B}^T\mathbf{y} &= (\mathbf{B}^T\mathbf{B} + n\lambda\Omega)\hat{\beta} && (18)\\
\hat{\beta} &= (\mathbf{B}^T\mathbf{B} + n\lambda\Omega)^{-1}\mathbf{B}^T\mathbf{y} && (19)
\end{aligned}
$$

Once again, if this were ordinary linear regression, the OLS estimate of the coefficients would be $(\mathbf{x}^T\mathbf{x})^{-1}\mathbf{x}^T\mathbf{y}$. In comparison to that, we've made two

[3] Recall that vectors $\vec{v}_1, \vec{v}_2, \ldots \vec{v}_d$ are linearly independent when there is no way to write any one of the vectors as a weighted sum of the others. The same definition applies to functions.

changes. First, we've substituted the basis function matrix $\mathbf{B}$ for the original matrix of independent variables, $\mathbf{x}$ — a change we'd have made already for plain polynomial regression. Second, the "denominator" is not $\mathbf{x}^T\mathbf{x}$, but $\mathbf{B}^T\mathbf{B}+n\lambda\Omega$. Since $\mathbf{x}^T\mathbf{x}$ is n times the covariance matrix of the independent variables, we are taking the covariance matrix of the spline basis functions and adding some extra covariance — how much depends on the shapes of the functions (through Ω) and how much smoothing we want to do (through λ). The larger we make λ, the less the actual data matters to the fit.

In addition to explaining how splines can be fit quickly (do some matrix arithmetic), this illustrates two important tricks. One, which we won't explore further here, is to turn a nonlinear regression problem into one which is linear in another set of basis functions. This is like using not just one transformation of the input variables, but a whole library of them, and letting the data decide which transformations are important. There remains the issue of selecting the basis functions, which can be quite tricky. In addition to the spline basis[4], most choices are various sorts of waves — sine and cosine waves of different frequencies, various wave-forms of limited spatial extent ("wavelets": see section 11.4 in Faraway), etc. The ideal is to chose a function basis where only a few non-zero coefficients would need to be estimated, but this requires some understanding of the data...

The other trick is that of stabilizing an unstable estimation problem by adding a penalty term. This reduces variance at the cost of introducing some bias. We will see much more of this in a later lecture.

4 Splines in Multiple Dimensions

Suppose we have *two* input variables, x and z, and a single response y. How could we do a spline fit?

One approach is to generalize the spline optimization problem so that we penalize the curvature of the spline surface (no longer a curve). The appropriate penalized least-squares objective function to minimize is

$$\mathcal{L}(m,\lambda) = \sum_{i=1}^{n}(y_i - m(x_i,z_i))^2 + \lambda \int dxdz \left[\left(\frac{\partial^2 m}{dx^2}\right)^2 + 2\left(\frac{\partial^2 m}{dxdz}\right)^2 + \left(\frac{\partial^2 m}{dz^2}\right)^2\right] \tag{20}$$

The solution is called a **thin-plate spline**. This is appropriate when the two input variables x and z should be treated more or less symmetrically.

An alternative is use the spline basis functions from section 3. We write

$$m(x) = \sum_{j=1}^{M_1}\sum_{k=1}^{M_2}\beta_{jk}B_j(x)B_k(z) \tag{21}$$

[4] Or, really, bases; there are multiple sets of basis functions for the splines, just like there are multiple sets of basis vectors for the plane. If you see the phrase "B splines", it refers to a particular choice of spline basis functions.

Doing all possible multiplications of one set of numbers or functions with another is said to give their **outer product** or **tensor product**, so this is known as a **tensor product spline** or **tensor spline**. We have to chose the number of terms to include for each variable (M_1 and M_2), since using n for each would give n^2 basis functions, and fitting n^2 coefficients to n data points is asking for trouble.

5 Smoothing Splines versus Kernel Regression

For one input variable and one output variable, smoothing splines can basically do everything which kernel regression can do[5]. The advantages of splines are their computational speed and (once we've calculated the basis functions) simplicity, as well as the clarity of controlling curvature directly. Kernels however are easier to program (if slower to run), easier to analyze mathematically[6], and extend more straightforwardly to multiple variables, and to combinations of discrete and continuous variables.

Further Reading

In addition to the sections in the textbooks mentioned on p. 1, there are good discussions of splines in Simonoff (1996, §5), Hastie *et al.* (2009, ch. 5) and Wasserman (2006, §5.5). The classic reference, by one of the people who really developed splines as a useful statistical tool, is Wahba (1990), which is great if you already know what a Hilbert space is and how to manipulate it.

References

Hastie, Trevor, Robert Tibshirani and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Berlin: Springer, 2nd edn. URL `http://www-stat.stanford.edu/~tibs/ElemStatLearn/`.

Simonoff, Jeffrey S. (1996). *Smoothing Methods in Statistics*. Berlin: Springer-Verlag.

Wahba, Grace (1990). *Spline Models for Observational Data*. Philadelphia: Society for Industrial and Applied Mathematics.

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

[5] In fact, as $n \to \infty$, smoothing splines approach the kernel regression curve estimated with a specific, rather non-Gaussian kernel. See Simonoff (1996, §5.6.2).

[6] Most of the bias-variance analysis for kernel regression can be done with basic calculus, much as we did for kernel density estimation in lecture 6. The corresponding analysis for splines requires working in infinite-dimensional function spaces called "Hilbert spaces". It's a pretty theory, if you like that sort of thing.

A Constraints, Lagrange multipliers, and penalties

Suppose we want to minimize[7] a function $L(u, v)$ of two variables u and v. (It could be more, but this will illustrate the pattern.) Ordinarily, we know exactly what to do: we take the derivatives of L with respect to u and to v, and solve for the u^*, v^* which makes the derivatives equal to zero, i.e., solve the system of equations

$$\frac{\partial L}{\partial u} = 0 \tag{22}$$

$$\frac{\partial L}{\partial v} = 0 \tag{23}$$

If necessary, we take the second derivative matrix of L and check that it is positive.

Suppose however that we want to impose a constraint on u and v, to demand that they satisfy some condition which we can express as an equation, $g(u, v) = c$. The old, unconstrained minimum u^*, v^* generally will not satisfy the constraint, so there will be a different, constrained minimum, say $\hat{u}, \hat{v}$. How do we find it?

We could attempt to use the constraint to eliminate either u or v — take the equation $g(u, v) = c$ and solve for u as a function of v, say $u = h(v, c)$. Then $L(u, v) = L(h(v, c), v)$, and we can minimize this over v, using the chain rule:

$$\frac{dL}{dv} = \frac{\partial L}{\partial v} + \frac{\partial L}{\partial u}\frac{\partial h}{\partial v} \tag{24}$$

which we then set to zero and solve for v. Except in quite rare cases, this is messy.

A superior alternative is the method of **Lagrange multipliers**. We introduce a new variable λ, the Lagrange multiplier, and a new objective function, the **Lagrangian**,

$$\mathcal{L}(u, v, \lambda) = L(u, v) + \lambda(g(u, v) - c) \tag{25}$$

which we minimize with respect to λ, u and v and λ. That is, we solve

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0 \tag{26}$$

$$\frac{\partial \mathcal{L}}{\partial u} = 0 \tag{27}$$

$$\frac{\partial \mathcal{L}}{\partial v} = 0 \tag{28}$$

Notice that minimize $\mathcal{L}$ with respect to λ always gives us back the constraint equation, because $\frac{\partial \mathcal{L}}{\partial \lambda} = g(u, v) - c$. Moreover, when the constraint is satisfied,

[7] Maximizing L is of course just minimizing $-L$.

$\mathcal{L}(u, v, \lambda) = L(u, v)$. Taken together, these facts mean that the $\hat{u}, \hat{v}$ we get from the *unconstrained* minimization of $\mathcal{L}$ is equal to what we would find from the *constrained* minimization of L. We have encoded the constraint into the Lagrangian.

Practically, the value of this is that we know how to solve unconstrained optimization problems. The derivative with respect to λ yields, as I said, the constraint equation. The other derivatives are however yields

$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial L}{\partial u} + \lambda \frac{\partial g}{\partial u} \tag{29}$$

$$\frac{\partial \mathcal{L}}{\partial v} = \frac{\partial L}{\partial v} + \lambda \frac{\partial g}{\partial v} \tag{30}$$

Together with the constraint, this gives us as many equations as unknowns, so a solution exists.

If $\lambda = 0$, then the constraint doesn't matter — we could just as well have ignored it. When $\lambda \neq 0$, the size (and sign) of the constraint tells us about how it affects the value of the objective function at the minimum. The value of the objective function L at the constrained minimum is bigger than at the unconstrained minimum, $L(\hat{u}, \hat{v}) > L(u^*, v^*)$. Changing the level of constraint c changes how big this gap is. As we saw, $\mathcal{L}(\hat{u}, \hat{v}) = L(\hat{u}, \hat{v})$, so we can see how much influence the level of the constraint on the value of the objective function by taking the derivative of $\mathcal{L}$ with respect to c,

$$\frac{\partial L}{\partial c} = -\lambda \tag{31}$$

That is, at the constrained minimum, increasing the constraint level from c to $c + \delta$ would change the value of the objective function by $-\lambda\delta$. (Note that λ might be negative.) This makes λ the "price", in units of L, which we would be willing to pay for a marginal increase in c — what economists would call the **shadow price**[8].

If there is more than one constraint equation, then we just introduce more multipliers, and more terms, into the Lagrangian. Each multiplier corresponds to a different constraint. The size of each multiplier indicates how much lower the objective function L could be if we relaxed that constraint — the set of shadow prices.

What about inequality constraints, $g(u, v) \leq c$? Well, either the unconstrained minimum exists in that set, in which case we don't need to worry about it, or it does not, in which case the constraint is "binding", and we can treat this as an equality constraint[9].

So much for constrained optimization; how does this relate to penalties? Well, once we fix λ, the (u, v) which minimizes the full Lagrangian

$$L(u, v) + \lambda g(u, v) + \lambda c \tag{32}$$

[8] In economics, shadow prices are internal to each decision maker, and depend on their values and resources; they are distinct from market prices, which depend on exchange and are common to all decision makers.

[9] A full and precise statement of this idea is the Karush-Kuhn-Tucker theorem of optimization, which you can look up.

has to be the same as the one which minimizes

$$L(u, v) + \lambda g(u, v) \tag{33}$$

This is a *penalized* optimization problem. Changing the magnitude of the penalty λ corresponds to changing the level c of the constraint. Conversely, if we start with a penalized problem, it implicitly corresponds to a constraint on the value of the penalty function $g(u, v)$. So, generally speaking, constrained optimization corresponds to penalized optimization, and vice versa.

For splines, λ is the shadow price of curvature in units of mean squared error. Turned around, each value of λ corresponds to constraining the average curvature not to exceed some level.