Nonparametric Smoothing

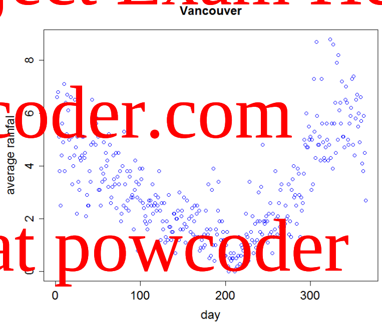
BTRY/STSCI 4520

- Smoothing with a Nadaraya-Watson estimator
- Choosing Bandwidths
- Non-parametric density estimation
- Local linear regression
- Basis expansion methods

Curve Fitting

In the most general set-up

- data are (t_i, y_i)
- want a function $y_i = g(t_i) + \epsilon_i$ with $\epsilon_i \sim N(0, \sigma^2)$
- don't want to make assumptions about g except that it is "smooth".



Try and produce methods that fit locally.

Local Fitting

Idea: use only points near the t of interest.

Binning:

- Divide $t_1, \ldots, t_n$ into bins.
- Use average of y_i who's t_i are in the same bin as t.

Produce locally constant $\hat{g}$. Not very pretty.

Use the average of a moving window centered at t:

$$\hat{g}(t) = \text{average of } y_i \text{ with } |t_i - t| < h$$

Or weight by distance from t.

Nadaraya-Watson Estimator

$$g(t, h) = \frac{\sum_{i=1}^{n} y_i K\left(\frac{t_i - t}{h}\right)}{\sum_{i=1}^{n} K\left(\frac{t_i - t}{h}\right)}$$

- $K((t - t_i)/h)$ weights each y_i by distance of t_i from t
- Denominator ensures that weights sum to 1 at each t.
- Bandwidth h spreads weight closer or further from t.


```
NW = function(t,TT,Y,bw){
  return( weighted.mean(Y, dnorm(TT-t,sd=bw) ) )
}
```

Larger h implies that points further away have influence on curve.

Bias-Variance Trade-off

Need to think about choosing h; what do we want to achieve? Possibly we would like to minimize expected squared error:

$$E\left(\hat{g}(t,h) - g(t)\right)^2$$

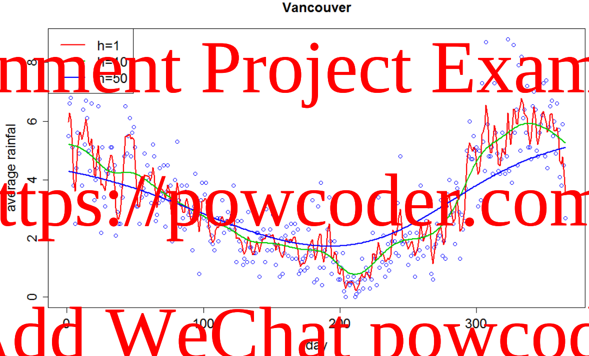
By adding and subtracting $E\hat{g}(t,h)$ we can write

$$\begin{aligned} E\left[\hat{g}(t,h) - g(t)\right]^2 &= \left[E\hat{g}(t,h) - g(t)\right]^2 + E\left[\hat{g}(t,h) - E\hat{g}(t,h)\right]^2 \\ &= \text{bias}^2 + \text{variance}. \end{aligned}$$

Intuitively: larger h leads to larger bias (use points further away), but smaller variance (use more points). We'll do this formally in density estimation.

This is generally referred to as a trade-off; want to balance these two.

Example: Vancouver Precipitation



- Small h: relies on points close to t: low bias, but high variance.
- Large h: spreads weight across many observations: smaller variance, higher bias.
- Finding an ideal compromise is difficult.

Choosing A Bandwidth

Rather than minimizing at just one value of t, we'll average across the interval:

$$\mathrm{IMSE}(h) = \int E\left(g(t) - \hat{g}(t, h)\right)^2 dt$$

This is the error we make for t drawn uniformly on the relevant interval.

We can also think of this as the error (minus residual variance) for predicting a new data point (t_{new}, y_{new}).

We will approximate this via cross-validation.

Cross Validation

Cross-validation = estimate on some data, evaluate on others.

For each data point (t_i, y_i)

- Remove (t_i, y_i) from the data.
- Estimate $\hat{g}^{-i}(t, h)$ from the remaining data.
- Predict $\hat{g}^{-i}(t_i, h)$

The cross-validated score is

$$\text{OCV}(h) = \sum_{i=1}^{n} \left(y_i - \hat{g}^{-i}(t_i, h)\right)^2$$

Treats each i as a "new" observation. Choose h to minimize $\text{OCV}(h)$.

Cross Validation

Leave each point out in turn and try to predict it

```
CV = function(bw,TT,Y)
{
  err = rep(0,length(Y))   # Prediction Errors
  for(i in 1:length(err)){
    err[i] = Y[i] - NW(TT[i],TT[-i],Y[-i],bw)
  }
  return( sum(err^2) )
}
```

We'll search over values, often best by powers of 10

```
hs = 1:20
for(j in 1:length(hs)){
  CVs[j] = CV(hs[j],TT,Y)
}
```

Calculating Cross Validation Efficiently

- Leaving one point out at a time is computationally expensive.
- But, $\hat{g}(t)$ is linear in the y_i; we can write out in vectors

$$\hat{\mathbf{y}} = S(h)\mathbf{y}$$

where $\mathbf{y}$ is the vector of observations and $\hat{\mathbf{y}}$ predicted values and

$$[S(h)]_{ij} = \frac{K\left(\frac{t_j - t_i}{h}\right)}{\sum_{k=1}^{n} K\left(\frac{t_i - t_k}{h}\right)}$$

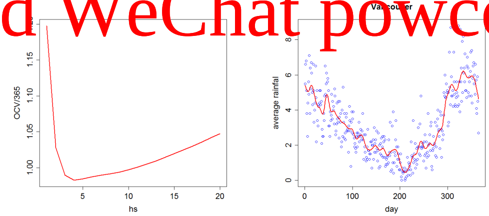
- Some calculation (not shown) gives us that

$$\mathrm{OCV}(h) = \sum_{i=1}^{n} \frac{(y_i - \hat{g}(t_i, h))^2}{(1 - [S(h)]_{ii})^2}$$

This identity means that we can calculate $\mathrm{OCV}(h)$ as efficiently as $\hat{\mathbf{y}}$.

In Code

```
hs = 1:20        # Bandwidths
OCV = 0*hs       # OCV scores

# Matrix of time-points:
X = matrix(TT,365,365)

for(i in 1:length(hs)){
  Kvals = dnorm( X - t(X),sd=hs[i] )              # Kernel distances
  Sh = diag( 1/apply(Kvals,1,sum) )%*%Kvals  # Divide by weights

  hatY = Sh%*%Y                                # Predictions
  OCV[i] = sum( (Y-hatY)^2/(1-diag(Sh))^2 )  # OCV formula
}
```

Generalized Cross Validation

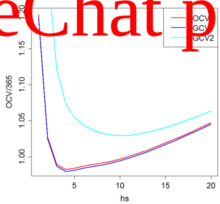
OCV tends to under-smooth and has some mathematically undesirable properties.

Instead, *Generalized Cross Validation* is often used

$$\mathrm{GCV}(h) = \frac{\sum_{i=1}^{n} (y_i - \hat{g}(t_i, h))^2}{\sum_{i=1}^{n} (1 - [S(h)]_{ii})^2}$$

```
GCV[i] = sum( (Y-hatY)^2 )/sum( (1-diag(Sh))^2 )
```

within loop above.



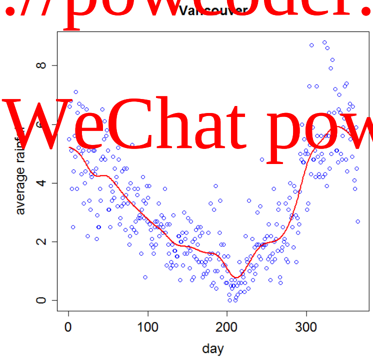
Undersmoothing

GCV and OCV generally produce curves that are rougher than is visually appealing.

More associated with human visual perception than statistics

One suggested change is a little extra penalty in GCV:

```
GCV2[i] = sum( (Y-hatY)^2 )/sum( (1-1.4*diag(Sh))^2 )
```

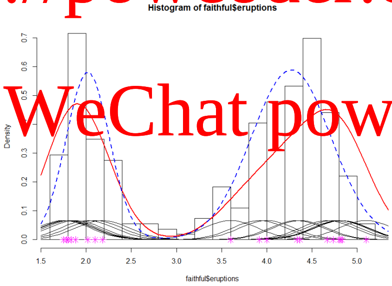
Extensions 1: Kernel Density Estimation (Rizzo Ch 10)

If $X_1, \ldots, X_n \sim f(\cdot)$ but we don't know f.

Estimate for f is

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{X_i - x}{h}\right)$$

places a "bump" K with width h over each data point X_i.

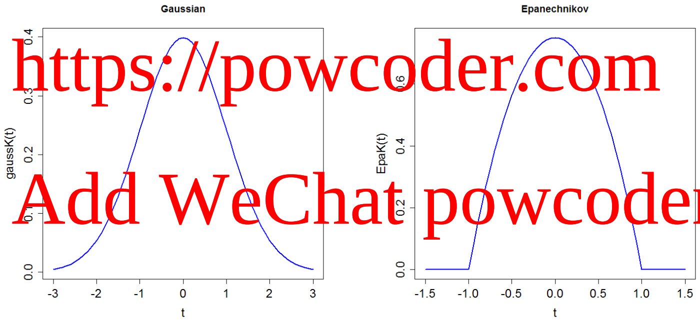


Using 20 points at random for ease of illustration.

Alternative Kernels

Examples here use the normal density; but any symmetric bump that integrates to 1 will do.

Gaussian Density $\max(1 - x^2, 0)$

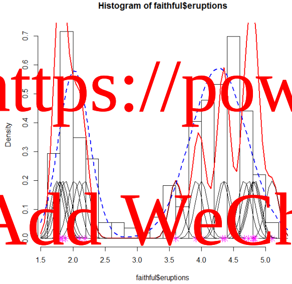


Epanechnikov kernel often favored for theoretical reasons.

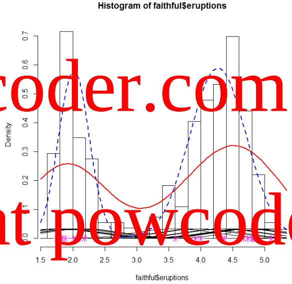
Bandwidth h

Bandwidth controls the “wigglyness” of the estimate.

$h = 0.1$

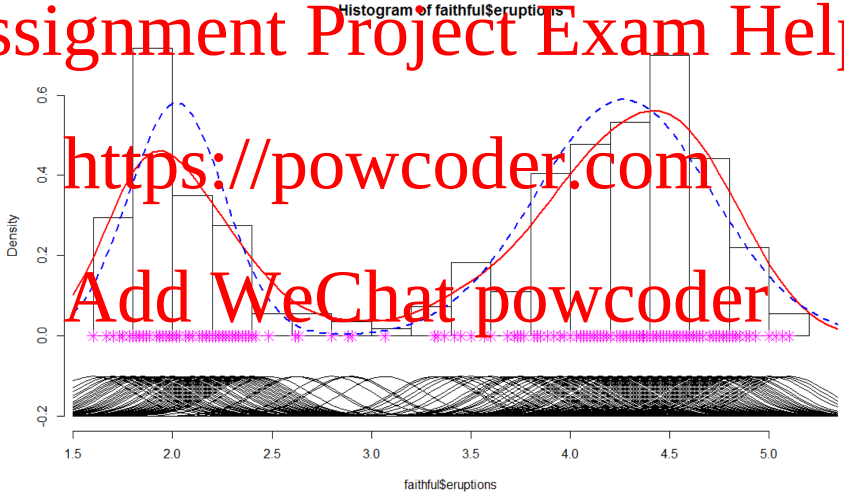


$h = 0.6$



Intuitively: smaller h makes estimate more sensitive to data $\Rightarrow$ higher variance.

It works better with all the data



Asymptotics Sketch (details unimportant)

We will look at the *expected* error between $\hat{f}(x)$ and $f(x)$.

As in smoothing, we divide expected error into

$$E(\hat{f}(x) - f(x))^2 = E(\hat{f}(x) - E\hat{f}(x))^2 + (E\hat{f}(x) - f(x))^2$$

which divides total error into *variance* and *bias*.

It is natural to think of $h \to 0$ as $n \to \infty$, but how quickly?

We will use change of variables:

$$\begin{aligned} E_{X_i} \frac{1}{h} K\left(\frac{X_i - x}{h}\right) &= \int \frac{1}{h} K\left(\frac{y - x}{h}\right) f(y) dy \\ &= \int \frac{1}{h} hK(u) f(x + hu) du = \int K(u) f(x + hu) du \end{aligned}$$

by setting $u = (y - x)/h$ or $y = x + hu$.

Bias

We observe that because each $K((X_i - x)/h)$ are i.i.d:

$$E\hat{f}(x) = \int \frac{1}{h} K\left(\frac{y-x}{h}\right) f(y)dy$$

bias "spreads out" f:

$$E\hat{f}(x) = \int K(u) f(x+hu)du$$

We take two terms of a Taylor expansion of $f(x+hu)$ about h

$$\begin{aligned}
E\hat{f}(x) &= \int K(u)\left[f(x) + huf'(x) + \frac{1}{2}h^2u^2 f''(x+hu^*)\right]du \\
&= f(x) + hf'(x)\int uK(u)du + \frac{1}{2}h^2 \int K(u)u^2 f''(x+hu^*)du \\
&= f(x) + o(h^2)
\end{aligned}$$

so bias decreases at rate h^2. Need to trade this off with variance.

Variance

Variance of $\hat{f}(x) = \frac{1}{n}\sum K((X_i - x)/h)/h$ is

$$
\begin{aligned}
\mathrm{var}(\hat{f}(x)) &= \frac{1}{n}\mathrm{Var}\frac{1}{h}K\left(\frac{X - x}{h}\right) \\
&= \frac{1}{n}\left(E\left(\frac{1}{h}K\left(\frac{X_i - x}{h}\right)\right)^2 - \left(E\frac{1}{h}K\left(\frac{X_i - x}{h}\right)\right)^2\right) \\
&= \frac{1}{n}\left(\frac{1}{h}\int\frac{1}{h}K\left(\frac{y - x}{h}\right)^2 f(y)dy - \left(\int K(u)f(x + hu)du\right)^2\right) \\
&= \frac{1}{nh}\int K(u)^2 f(x + hu)du - \frac{1}{n}\left(\int K(u)f(x + hu)du\right)^2 \\
&= O\left(\frac{1}{nh}\right)
\end{aligned}
$$

increases at rate $1/h$ for a fixed n.

Trade Off

We can do calculus with order notation by looking at

$$\begin{aligned}
E(\hat{f}(x) - f(x))^2 &= E(\hat{f}(x) - E\hat{f}(x))^2 + (E\hat{f}(x) - f(x))^2 \\
&= \frac{A}{nh} + (f(x) + Bh^2 - f(x))^2 \\
&= \frac{A}{nh} + B^2 h^4
\end{aligned}$$

which is minimized when

$$h = \left(\frac{A}{4B^2 n}\right)^{1/5} = O(n^{-1/5})$$

or plugging this back into MSE

$$E(\hat{f}(x) - f(x))^2 = O(n^{-4/5}).$$

Notice that this is slower than the $O(n^{-1})$ rate we usually get for squared error of parameter estimates.

Extensions 2: Local Linear Regression

Re-thinking Nadaraya-Watson; the locally-weighted mean solves

$$\hat{m}(t) = \underset{m}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - m)^2 K\left(\frac{t_i - t}{h}\right)$$

But we don't have to restrict to a "constant" model.

Local linear regression

$$(\hat{\beta}_0(t), \hat{\beta}_1(t)) = \underset{\beta_0, \beta_1}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1(t_i - t))^2 K\left(\frac{t_i - t}{h}\right)$$

has better statistical properties, especially at edge of data and if you want to take derivatives.

Predict $\hat{\beta}_0(t)$ but we can also get a slope from $\hat{\beta}_1(t)$
Local weighting can be applied to any model in this way.

R Packages

(see Lab for examples, but) a number of packages provide local polynomial fitting

- locpol
  - choice of kernels
  - degree 1 through 10 (include regression on t^2, t^3 etc)
  - nice plotting features

- KernSmooth
  - only Gaussian kernels
  - also allows degree 0
  - density estimation etc

- locfit (not as clearly documented, but does multivariate local polynomials)

Generally, require evaluation points to be specified beforehand.

Basis Expansions

Alternative, to local methods, we can use regression approaches.

From multiple linear regression:

$$y_i = \beta_0 + x_{1i}\beta_1 + x_{2i}\beta_2 + \cdots + \epsilon_i$$

Or if there is curvature:

$$y_i = \beta_0 + x_i\beta_1 + x_i^2\beta_2 + x_i^3\beta_3 + \cdots + \epsilon_i$$

Which we will write as

$$y_i = \sum_{j=1}^{K} c_j\phi_j(t_i) + \epsilon_i = g(t_i) + \epsilon_i$$

or more compactly

$$g(t) = \mathbf{c}^T\Phi(t)$$

Splines

Using polynomials is generally a bad idea. Many other basis systems available. We'll illustrate with splines here.

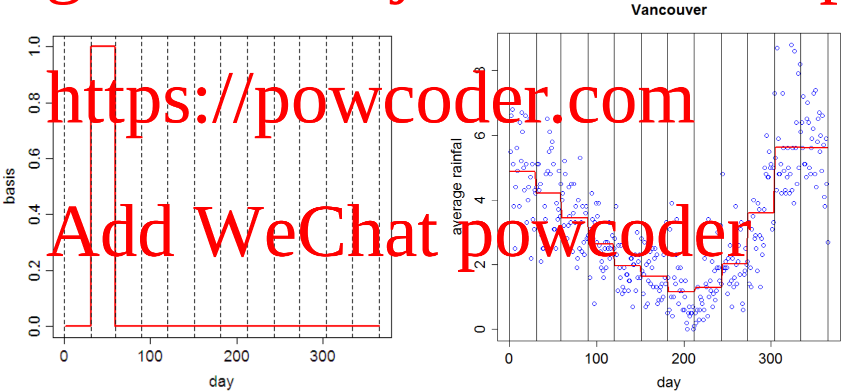
- Splines are polynomial segments joined end-to-end
- Segments are constrained to be smooth at the join
- The points at which the segments join are called *knots*
- The order m (order = degree+1) of the polynomial segments and
- the location of the knots define the system
- **Bsplines** are a particularly useful means of incorporating the constraints.

The package `splines` will be used for this.

Splines

Vancouver precipitation with knots at months.

Splines of order 1

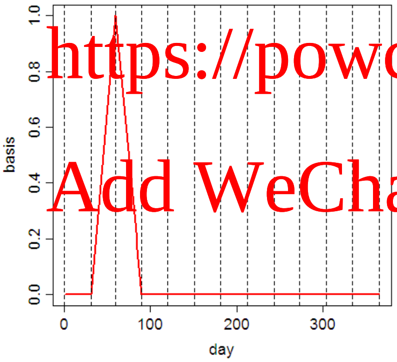


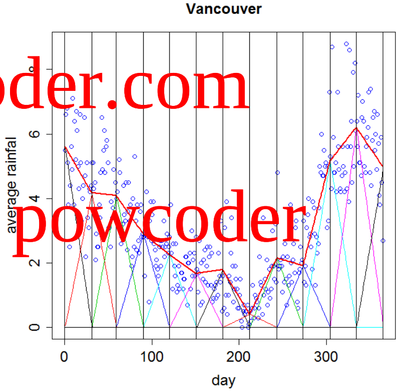
Same as dividing data into bins and and use mean in each bin.

Splines

Vancouver precipitation with knots at months.

Splines of order 2

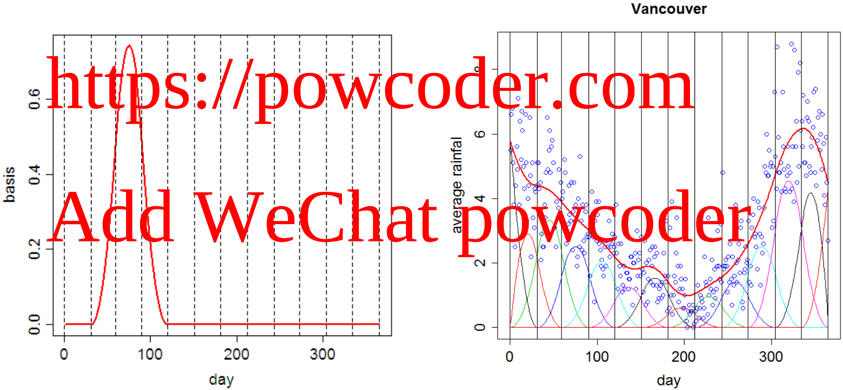




Splines

Vancouver precipitation with knots at months.

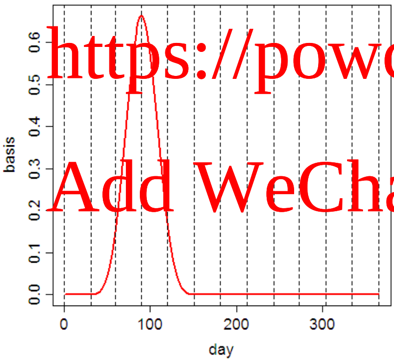
Splines of order 3

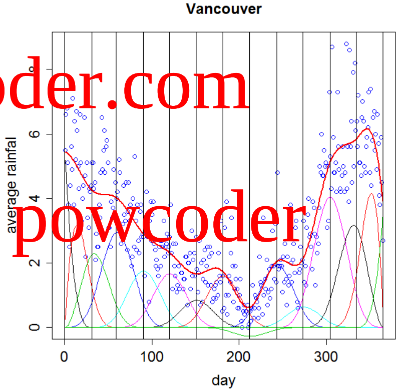


Splines

Vancouver precipitation with knots at months.

Splines of order 4

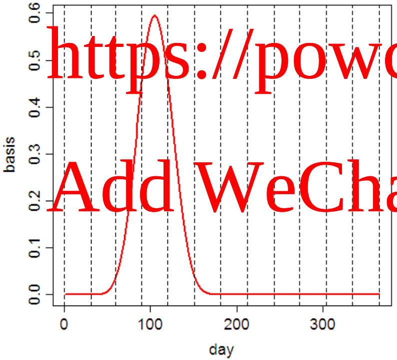


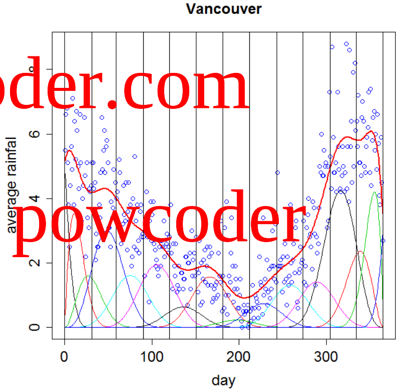


Splines

Vancouver precipitation with knots at months.

Splines of order 5

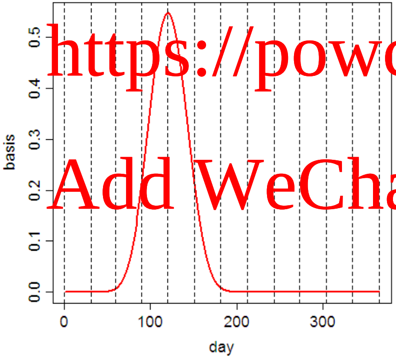


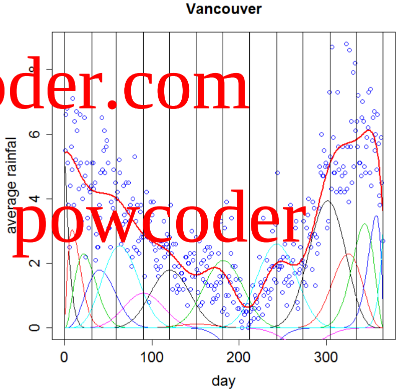


Splines

Vancouver precipitation with knots at months.

Splines of order 6





Least-Squares

We have observations

$$y_i = g(t_i) + \epsilon$$

and we want to estimate

$$g(t) \approx \sum_{j=1}^{K} c_j \phi_j(t)$$

Minimize the sum of squared errors:

$$SSE = \sum_{i=1}^{n} (y_i - g(t_i))^2 = \sum_{i=1}^{n} (y_i - \mathbf{c}^T \Phi(t_i))^2$$

This is just linear regression!

The `splines` Package

To define a B-spline you need

1. A vector of knots
2. The order of the spline

On knots

- `ord-1` knots on each end are *boundary* points
- Splines cover `ord` inter-knot spacings – boundaries allow the edges.
- Can spread these out like regular knots, or pile up at the end points.

`splineDesign` gives values of basis functions

```
library(splines)
knots = c(rep(0,3),months,rep(365,3))   # Order 4 splines -
bvals = splineDesign(x=TT,knots=knots,ord=4)
```

Linear Regression on Basis Functions

- If the N by K matrix $\boldsymbol{\Phi}$ contains the values $\phi_k(t_j)$, and $\mathbf{y}$ is the vector $(y_1, \ldots, y_n)$, we can write

$$SSE(\mathbf{c}) = (\mathbf{y} - \boldsymbol{\Phi}\mathbf{c})^T (\mathbf{y} - \boldsymbol{\Phi}\mathbf{c})$$

- The error sum of squares is minimized by the *ordinary least squares estimate*

$$\hat{\mathbf{c}} = \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^T \mathbf{y}$$

- Then we have the estimate

$$\hat{g}(t) = \Phi(t)\hat{\mathbf{c}} = \Phi(t) \left(\boldsymbol{\Phi}^T \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^T \mathbf{y}$$

Penalization

- Using splines, or other basis expansion, bias/variance trade-off is in terms of number and order of basis functions.
- This is usually difficult to work with, and estimates have high variability.
- Instead, we add a penalty to squared error:

$$PENSSE(\lambda) = \sum_{i=1}^{n} (y_i - g(t_i))^2 + \lambda \int \left(g''(t)\right)^2 dt$$

  - Fit to data + penalty for roughness.
  - $\int g''(t)^2 dt = 0$ if $g(t) = a + bt$, but large if g is "wiggly"
  - λ governs trade-off between the two.
- Use more knots that needed (one each data point if possible) and then control variance with λ.

Penalty Matrices

Observe that

$$\lambda \int \left(g''(t)\right)^2 dt = \lambda \int \left(\sum c_i \phi_i''(t)\right)^2 dt$$
$$= \int \sum \sum c_i c_j \phi_i''(t) \phi_j''(t) dt$$
$$= \lambda \mathbf{c}^T P \mathbf{c}$$

for $P_{ij} = \int \phi_i''(t)\phi_j''(t)dt$.

We will approximate

$$\int \phi_i''(t)\phi_j''(t) \approx \delta \sum_k \phi_i''(k\delta)\phi_j''(k\delta)$$

(see numerical integration next) can write this as

$$P = \delta \mathbf{\Phi}''^T \mathbf{\Phi}''$$

`splineDesign` allows you to specify `derivs` for derivatives.

OCV and GCV

- Write

$$PENSSE(\lambda) = \sum (y_i - \Phi(t_i)\mathbf{c})^2 + \lambda \mathbf{c}^T P \mathbf{c}$$

- This gives formula for penalized $\mathbf{c}$ as

$$\hat{\mathbf{c}} = \left(\mathbf{\Phi}^T \mathbf{\Phi} + \lambda P \right)^{-1} \mathbf{\Phi}^T \mathbf{y}$$

or

$$\hat{\mathbf{y}} = \mathbf{\Phi} \left(\mathbf{\Phi}^T \mathbf{\Phi} + \lambda P \right)^{-1} \mathbf{\Phi}^T \mathbf{y} = S(\lambda)\mathbf{y}$$

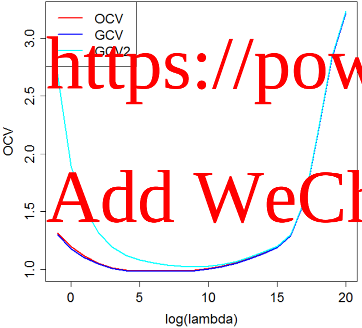
- Same formulae for OCV and GCV for λ.

Fitting Vancouver

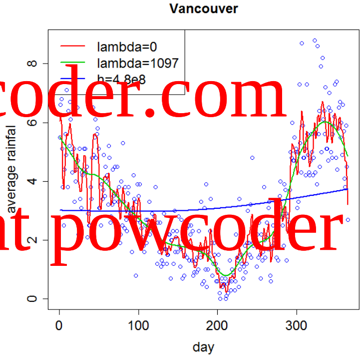
```
knots = c( rep(0,3),0:365,rep(365,3) ) # Knot every day
bvals = splineDesign(knots=knots,x=TT)  # Order 4 is default

pvals = splineDesign(knots=knots,x=seq(0,365,by=0.1),derivs=2)
Pen = 0.1*t(pvals)%*%pvals

# Now loop over values of lambda
lambda = exp(-1:20)

OCV = 0*lambda
GCV = 0*lambda

Yhats = matrix(0,365,length(lambda))

for(i in 1:length(lambda)){
  Slam = bvals%*%solve( t(bvals)%*%bvals+lambda[i]*Pen, t(bvals))

  Yhat = Slam%*%Y
  OCV[i] = mean( (Y-Yhat)^2/(1-diag(Slam))^2 )
  GCV[i] = sum( (Y-Yhat)^2)/sum( (1-diag(Slam))^2 )

  Yhats[,i] = Yhat
}
```

Fitting Vancouver

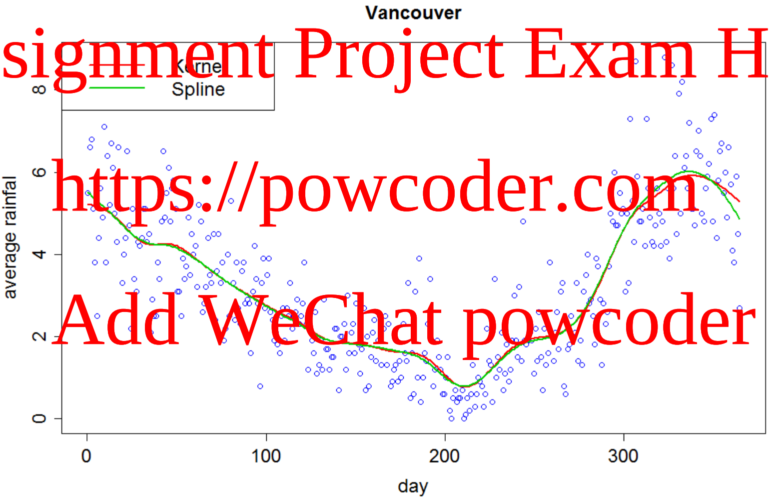
GCV/OCV Curves



Fits



A Comparison



Broader Uses

- Splines (and other bases) allow non-parametric terms to be included in any model.

- E.g. consider the partially linear model with a linear effect for x_1 and non-parametric effect for x_2:

$$y_i = x_{1i}\beta_1 + g(x_{2i}) + \epsilon$$

- Just represent $g(x_{2i}) = \Phi(x_{2i})\mathbf{c}$ and add $\mathbf{c}$ to coefficients.
- Works in generalized linear models, non-linear models, many other places.

- Many basis systems other than B-splines. Most noticeably Fourier bases

$$1, \cos(\omega t), \sin(\omega t), \cos(2\omega t), \sin(2\omega t), \cos(3\omega t), \ldots$$

and wavelets.

Summary

Two ways of adding non-parametric effects

1. Kernel methods
   - Weight points to account for those that are closest to t.
   - Requires whole model to change with t.
   - Relatively expensive, computationally.
   - Easy to work with mathematically.
2. Basis expansions
   - Represent non-parametric effect via a set of basis functions.
   - Add penalization to obtain smoothness.
   - Computationally efficient, can target terms you want.
   - Harder to demonstrate mathematical/statistical properties.

In both cases, smoothing parameters (h, λ) must be chosen by cross validation or other methods.