Computational Statistics with Application to Bioinformatics

Prof. William H. Press
Spring Term, 2008
The University of Texas at Austin

Unit 20: Multidimensional Interpolation on Scattered Data
Unit 20: Multidimensional Interpolation on Scattered Data (Summary)

• One dimensional interpolation is more or less a black-box solved problem
• In higher dimensions, the explosion of volume makes things hard
  – rarely if ever can populate a full grid
  – has more the character of a machine learning problem
  – we’ll illustrate on a simple, smooth function in 4 dimensions
  – generate a training and a testing set to use comparatively

• Shepard Interpolation
  – smoother version of “nearest neighbor”
  – it’s fast, but woefully inaccurate compared to other methods
  – generally use inverse-power weights
    • minimum exponent is $D+1$

• Radial Basis Function (RBF) Interpolation
  – solve linear equations to put interpolant exactly through the data
  – it’s slow, $O(N^3)$ one time work + $O(N)$ per interpolation
  – various basis functions: multiquadric, inverse multiquadric, thin plate spline, Gaussian
    • they usually have a scale factor that must be determined empirically
    • controls over- vs. under-smoothing
• Laplace Interpolation restores missing data values on a grid
  – solve Laplace’s equation with (interior) boundary conditions on known values
  – implement by sparse linear method such as biconjugate gradient
  – can do remarkably well when as much as 90% of the values are missing
• Gaussian Process Regression (aka Linear Prediction, aka Kriging)
  – like Shepard and RBF, a weighted average of observed points
  – but the weights now determined by a statistical model, the variogram
    • equivalent to estimating the covariance
  – can give error estimates
  – can be used both for interpolation (“honor the data”) and for filtering (“smooth the data”)
  – it’s a kind of Wiener filtering in covariance space
    • function covariance is the signal, noise covariance is the noise
  – cost is $O(N^3)$ one time work, $O(N)$ per interpolation, $O(N^2)$ per variance estimate
Interpolation on Scattered Data in Multidimensions

In 1-dimension, interpolation is basically a solved problem, OK to view as a black box (at least in this course):

```matlab
pts = sort(rand(20,1));
vals = exp(-3*pts.^2);
ivals = interp1(pts,vals,[0:.001:1],'spline');
tvals = exp(-3*[0:.001:1].^2);
hist(ivals-tvals,50);
```

Some smooth function of order unity

Note scale

Shape from oscillatory behavior of error term
In higher numbers of dimensions, especially >2, the explosion of volume makes things more interesting.

Rarely enough data for any kind of mesh.

Lots of near-ties for nearest neighbor points (none very near).

The problem is more like a machine learning problem:

Given a training set $x_i$ with “responses” $y_i$, $i = 1 \ldots N$
Predict $y(x)$ for some new $x$
Let's try some different methods on 500 data points in 4 dimensions.

$500^{1/4} \approx 4.7$, so the data is sparse, but not ridiculous

\[ \text{testfun} = @(x) 514.1890 \times \exp(-2.0 \times \text{norm}(x- [.3 .3 .3 .3])) \times \ldots \times x(1) \times (1-x(1)) \times x(2) \times (1-x(2)) \times x(3) \times (1-x(3)) \times x(4) \times (1-x(4)); \]

\[ [x1 \ x2] = \text{meshgrid}(0:0.01:1,0:0.01:1); \]
\[ z = \text{arrayfun}( @(s1,s2) \text{testfun}([s1 \ s2 .3 .3]), x1, x2); \]
\[ \text{contour}(z) \]

a Gaussian, off-center in the unit cube, and tapered to zero at its edges

\[ z = \text{arrayfun}( @(s1,s2) \text{testfun}([s1 \ s2 .7 .7]), x1, x2); \]
\[ \text{contour}(z) \]
Generate training and testing sets of data. The points are chosen randomly in the unit cube.

```matlab
npts = 500;
pts = cell(npts, 1);
for j=1:npts, pts{j} = rand(1, 4); end;
vals = cellfun(testfun,pts);
hist(vals, 50)
```

```matlab
tpts = cell(npts, 1);
for j=1:npts, tpts{j} = rand(1, 4); end;
tvals = cellfun(testfun,tpts);
hist(tvals, 50)
```

If you have only one sample of real data, you can test by leave-one-out, but that is a lot more expensive since you have to repeat the whole interpolation, including one-time work, each time.
Shepard Interpolation

The prediction is a weighted average of all the observed values, giving (much?) larger weights to those that are closest to the point of interest.

It’s a smoother version of “value of nearest neighbor” or “mean of few nearest neighbors”.

\[ y(x) = \frac{\sum_{i=0}^{N-1} y_i \phi(|x - x_i|)}{\sum_{i=0}^{N-1} \phi(|x - x_i|)} \]

\[ \phi(r) = r^{-p} \]

the power-law form has the advantage of being scale-free, so you don’t have to know a scale in the problem

In D dimensions, you’d better choose \( p \geq D+1 \), otherwise you’re dominated by distant, not close, points: volume \( \sim \) no. of points \( \sim r^D \)

Shepard interpolation is relatively fast, \( O(N) \) per interpolation. The problem is that it’s usually not very accurate.
Shepard performance on our training/testing set:

```matlab
function val = shepinterp(x, p, vals, pts)
phi = cellfun(@(y) (norm(x-y)+1.e-40).^(-p), pts);
val = (vals' * phi)./sum(phi);

shepvals = cellfun(@(x) shepinterp(x, 6, vals, pts), tpts);
plot(tvals, shepvals, '.')

hist(shepvals-tvals,50)
```

note value of p

note biases
Want to see what happens if we choose too small a value of p for this D?

```
shepvals = cellfun(@(x) shepinterp(x,3,vals,pts), tpts);
plot(tvals,shepvals,'.
hist(shepvals-tvals,50)
```

small values are getting pulled up by the (distant) peak

If you choose too **large** a value for p, you get ~ “value of nearest neighbor”
Radial Basis Function Interpolation

This looks superficially like Shepard, but it is typically much more accurate.

However, it is also much more expensive:
O(N^3) one time work + O(N) per interpolation.

Like Shepard, the interpolator is a linear combination of identical kernels, centered on the known points

\[ y(\mathbf{x}) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|) \]

But now we solve N linear equations to get the weights, by requiring the interpolator to go exactly through the data:

\[ y_j = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \quad \text{or} \quad \Phi \mathbf{w} = \mathbf{y} \]

There is now no requirement that the kernel \( \phi(r) \) falls off rapidly, or at all, with \( r \).
Commonly used Radial Basis Functions (RBFs)

\[ \phi(r) = (r^2 + r_0^2)^{1/2} \]  
“multiquadric”

you have to pick a scale factor

\[ \phi(r) = (r^2 + r_0^2)^{-1/2} \]  
“inverse multiquadric”

\[ \phi(r) = r^2 \log(r/r_0) \]  
“thin plate spline”

\[ \phi(r) = \exp\left(-\frac{1}{2} \frac{r^2}{r_0^2}\right) \]  
“Gaussian” Typically very sensitive to the choice of \( r_0 \), and therefore less often used. (Remember the problems we had getting Gaussians to fit outliers!)

The choice of scale factor is a trade-off between over- and under-smoothing. (Bigger \( r_0 \) gives more smoothing.) The optimal \( r_0 \) is usually on the order of the typical nearest-neighbor distances.
Let’s try a multiquadric with $r_0 = 0.1$

\[
\begin{align*}
    r_0 &= 0.1; \\
    \phi &= @(x) \sqrt{\text{norm}(x)^2 + r_0^2}; \\
    \text{phimat} &= \text{zeros}(\text{npts}, \text{npts}); \\
    \text{for } i=1: \text{npts}, \text{ for } j=1: \text{npts}, \text{ phimat}(i,j) &= \phi(\text{pts}(i) - \text{pts}(j)); \text{ end}; \text{ end}; \\
    \text{wgts} &= \text{phimat}\backslash\text{vals}; \\
    \text{valinterp} &= @(x) \text{wgts}' \times \text{cellfun}(@(y) \phi(\text{norm}(x-y)), \text{pts}); \\
    \text{ivals} &= \text{cellfun}(\text{valinterp}, \text{tpts}); \\
    \text{hist(ivals-\text{vals},50)} \\
    \text{stdev} &= \text{std(ivals-\text{vals})}
\end{align*}
\]

Matlab “solve linear equations” operator!

\[\text{stdev} = 0.0145\]
Automate the test process somewhat

```matlab
function [stdev,ivals] = TestAnRBF(phi,pts,vals,tpts,tvals)
npts = numel(vals);
phiMat = zeros(npts, npts);
for i=1:npts, for j=1:npts, phiMat(i,j) = phi(pts{i}-pts{j}); end; end;
wghts = phiMat' * vals;
valinterp = @(x) wghts' * cellfun(@(y) phi(norm(x-y)),pts);
ivals = cellfun(valinterp,tpts);
stdev = std(ivals-tvals);
```

One test looks like

```matlab
r0 = 0.2;
phi = @(x) sqrt(norm(x)^2+r0^2);
[stdev,ivals] = TestAnRBF(phi,pts,vals,tpts,tvals);
stdev
plot(tvals,ivals,'.')
hist(ivals-tvals,50)
```

And the results are…

You have to redefine phi each time, because, in Matlab semantics, the value of r0 used is that at the time of definition, not execution.
$r_0 = 0.2$

$\sigma = 0.0099$

so $r_0 \sim 0.6$ is the optimal choice, and it’s not too sensitive

$\sigma = 0.0059$

$\sigma = 0.0104$
Try an inverse multiquadric

\[
r_0 = 0.6; \\
\phi = @(x) 1./\sqrt{\text{norm}(x)^2 + r_0^2}; \\
[\text{stdev ivals}] = \text{TestAnRBF}(\phi, \text{pts}, \text{vals}, \text{tpts}, \text{tvals}); \\
\text{stdev} \\
\text{plot}(\text{tvals}, \text{ival}s, '.') \\
stdev = \\
0.0058
\]

(performance virtually identical to multiquadric on this example)

RBF interpolation is for interpolation on a smooth function, not for fitting a noisy data set.

By construction it exactly “honors the data” (meaning that it goes through the data points – it doesn’t smooth them).

If the data is in fact noisy, RBF will produce an interpolating function with spurious oscillations.
**Laplace Interpolation** is a specialized interpolation method for restoring missing data on a grid. It’s simple, but sometimes works astonishingly well.

Mean value theorem for solutions of Laplace’s equation (harmonic functions):

If \( y \) satisfies \( \nabla^2 y = 0 \) in any number of dimensions, then for any sphere not intersecting a boundary condition,

\[
\frac{1}{\text{area}} \int_{\text{surface } \omega} y \, d\omega = y(\text{center})
\]

So Laplace’s equation is, in some sense, the perfect interpolator. It turns out to be the one that minimizes the integrated square of the gradient,

\[
\int_{\Omega} |\nabla y|^2 \, d\Omega
\]

So the basic idea of Laplace interpolation is to set at every known data point, and solve \( \nabla^2 y = 0 \) at every unknown point.
Lots of linear equations (one for each grid point)!

\[ y_0 - \frac{1}{4} y_u - \frac{1}{4} y_d - \frac{1}{4} y_l - \frac{1}{4} y_r = 0 \]  
\[ y_0 = y_0\text{(measured)} \]

generic equation for an unknown point
note that this is basically the mean value theorem

generic equation for a known point

lots of special cases:
\[ y_0 - \frac{1}{2} y_u - \frac{1}{2} y_d = 0 \]  
(left and right boundaries)
\[ y_0 - \frac{1}{2} y_l - \frac{1}{2} y_r = 0 \]  
(top and bottom boundaries)
\[ y_0 - \frac{1}{2} y_r - \frac{1}{2} y_d = 0 \]  
(top-left corner)
\[ y_0 - \frac{1}{2} y_l - \frac{1}{2} y_d = 0 \]  
(top-right corner)
\[ y_0 - \frac{1}{2} y_r - \frac{1}{2} y_u = 0 \]  
(bottom-left corner)
\[ y_0 - \frac{1}{2} y_l - \frac{1}{2} y_u = 0 \]  
(bottom-right corner)

There is exactly one equation for each grid point, so we can solve this as a giant (sparse!) linear system, e.g., by the bi-conjugate gradient method.

Surprise! It’s in NR3, as Laplace_interp, using Linbcg for the solution.
Easy to embed in a mex function for Matlab

```c
#include "nr3_matlab.h"
#include "linbcg.h"
#include "interp_laplace.h"

/* Usage:
   outmatrix = laplaceinterp(inmatrix)
*/

Laplace_interp *mylap = NULL;
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    if (nrhs != 1 || nlhs != 1) throw("laplaceinterp.cpp: bad number of args");
    MatDoub ain(prhs[0]);
    MatDoub aout(ain.nrows(), ain.ncols(), plhs[0]);
    aout = ain; // matrix op
    mylap = new Laplace_interp(aout);
    mylap->solve();
    delete mylap;
    return;
}
```
Let’s try it on our favorite face for filtering
(But this is interpolation, not filtering: there is no noise!)

```matlab
IN = fopen('image-face.raw','r');
face = flipud(reshape(fread(IN),256,256)');
fclose(IN);
bwcolormap = [0:1/256:1; 0:1/256:1; 0:1/256:1]';
image(face)
colormap(bwcolormap);
axis('equal')
```
facemiss = face;
ranface = rand(size(face));
facemiss(ranface < 0.1) = 255;
image(facemiss)
colormap(bwcolormap)
axis('equal')
delete a random 10% of pixels
facemiss(facemiss > 254) = 9.e99;
newface = laplaceinterp(facemiss);
image(newface)
colormap(bwcolormap)
axis('equal')

restore them by Laplace interpolation
this is the convention expected by laplaceinterp for missing data

pretty amazing!
facemiss = face;
ranface = rand(size(face));
facemiss(ranface < 0.5) = 255;
image(facemiss)
colormap(bwcolormap)
axis('equal')

delete a random 50% of pixels
facemiss(facemiss > 254) = 9. e99;
newface = laplaceinterp(facemiss);
image(newface)
colorbar(bwcolormap)
axis('equal')

restore them by Laplace interpolation

starting to see some degradation
facemiss = face;
ranface = rand(size(face));
facemiss(ranface < 0.9) = 255;
image(facemiss)
colormap(bwcolormap)
axis('equal')
delete a random 90% of pixels
(well, it's cheating a bit, because your eye can't see
the shades of grey in the glare of all that white)
This is a bit more fair…
facemiss(facemiss > 254) = 9.e99;
newface = laplaceinterp(facemiss);
image(newface)
colormap(bwcolormap)
axis('equal')

restore by Laplace interpolation

still pretty amazing (e.g., would you have thought that the individual teeth were present in the sparse image?)
Linear Prediction
a.k.a. Gaussian Process Regression
a.k.a. Kriging

What is “linear” about it, is that the interpolant is a linear combination of the data \( y \) values, like Shepard interpolation (but on steroids!)

The weights, however, are highly nonlinear function of the \( x \) values. They are based on a statistical model of the function’s smoothness instead of Shepard’s fixed (usually power law) radial functions. That’s where the “Gaussian” comes from, not from any use of Gaussian-shaped functions!

The weights can either honor the data (interpolation), or else smooth the data with a model of how much unsmoothness is due to noise (fitting).

The underlying (noise free) model need not even be smooth.

You can get error estimates.

This is actually pretty cool, but it’s concepts can be somewhat hard to understand the first time you see them, so bear with me!

Also, since this is spread out over three sections of NR3, the notation will keep hopping around (apologies!)

Danie G. Krige
measured
signal
noise

residual (we’ll try to minimize it)

an interpolated value

using \( \langle y_\alpha n_\beta \rangle = 0 \)

these quantities have values that come from the autocorrelation structure of the signal

this is the autocorrelation structure of the noise

\[
y'_\alpha = y_\alpha + n_\alpha
\]

\[
y_\star = \sum_\alpha d_{\star \alpha} y'_\alpha + x_\star
\]
Now take the derivative of the mean square residual w.r.t. the weights, and set it to zero. Immediately get

\[ \sum_{\beta} \left[ \phi_{\alpha\beta} + \eta_{\alpha\beta} \right] d_{\star\beta} = \phi_{\star\alpha} \]

where

\[ \phi_{\alpha\beta} \equiv \langle y_\alpha y_\beta \rangle \quad \phi_{\star\alpha} \equiv \langle y_{\star} y_\alpha \rangle \quad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta \rangle \quad \text{or} \quad \langle n_\alpha^2 \rangle \delta_{\alpha\beta} \]

So,

\[ y_{\star} \approx \sum_{\alpha\beta} \phi_{\star\alpha} \left[ \phi_{\mu\nu} + \eta_{\mu\nu} \right]_{\alpha\beta}^{-1} y_{\beta} \]

This should remind you of Wiener filtering. In fact, it is Wiener filtering in the principal component basis:

\[ y_\gamma = \frac{\phi_{\gamma\gamma}}{\phi_{\gamma\gamma} + \eta_{\gamma\gamma}} y_{\gamma}' \]

So, this actually connects the lecture on PCA to the lecture on Wiener filtering!
Substituting back, we also get an estimate of the m.s. discrepancy

\[
\langle x^2 \rangle_0 = \langle y^2 \rangle - \sum_{\beta} d_{*\beta} \phi_{*\beta} = \langle y^2 \rangle - \sum_{\alpha\beta} \phi_{*\alpha} \left[ \phi_{\mu\nu} + \eta_{\mu\nu} \right]_{\alpha\beta}^{-1} \phi_{*\beta}
\]

That’s the “basic version” of linear prediction.

Unfortunately, the basic version is missing a couple of important tweaks, both related to the fact that, viewed as a Gaussian process, \( y \) may not have a zero mean, and/or may not be stationary. (Actually these two possibilities are themselves closely related.)

Tweak 1: We should constrain \( \sum_{\beta} d_{*\beta} = 1 \)
This is called “getting an unbiased estimator”.

Tweak 2: We should replace the covariance model \( \langle y_\alpha y_\beta \rangle \)
by an equivalent “variogram model” \( \langle (y_\alpha - y_\beta)^2 \rangle \)
(If you square this out, you immediately see that the relation involves \( \langle y_\alpha^2 \rangle \)
which is exactly the bad actor in a nonstationary process!)

Deriving the result is more than we want to do here, so we’ll jump to the answer. A (good content, but badly written) reference is Cressie, *Statistics for Spatial Data*. 
The answer is (changing notation!):

1. Estimate the variogram

\[ v(r) \sim \frac{1}{2} \left( [v(x + r) - v(x)]^2 \right) \]

Most often, this is done by assuming that it is isotropic (spherically symmetric), \( v(r) \)
Then you loop over all pairs of points for values from which to fit some parameterized form like

\[ v(r) = \alpha r^\beta \]

“power-law model”

There’s an NR3 object for doing exactly this.

2. Define:

\[ v_{ij} = v(|x_i - x_j|) \]
\[ Y = (y_0, y_1, \ldots, y_{N-1}, 0) \]
\[ V_* = (v_{*1}, v_{*2}, \ldots, v_{*N-1}, 1) \]
\[ \Sigma' = \begin{pmatrix} \Sigma & 0 \\ 0 & 0 \end{pmatrix} \]

\[ V = \begin{pmatrix} v_{00} & v_{01} & \cdots & v_{0,N-1} & 1 \\ v_{10} & v_{11} & \cdots & v_{1,N-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ v_{N-1,0} & v_{N-1,1} & \cdots & v_{N-1,N-1} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \]

(The weird augmentation of 1’s and 0’s trickily imposes the unbiased estimator constraint don’t ask how!)
3. Interpolate (as many times as you want) by

\[ \hat{y}_* = V_* \cdot (V - \Sigma')^{-1} \cdot Y \]

and (if you want) estimate the error by

\[ \text{Var}(\hat{y}_*) = V_* \cdot (V - \Sigma')^{-1} \cdot V_* \]

Note that each interpolation (if you precompute the last two matrix factors) costs \(O(N)\), while each variance costs \(O(N^2)\).

By the way, some other popular variogram models are

\[ v(r) = b[1 - \exp(-r/a)] \quad \text{“exponential model”} \]

\[ v(r) = \begin{cases} 
  b \left( \frac{3}{2} \frac{r}{a} - \frac{1}{2} \frac{r^3}{a^3} \right) & 0 \leq r \leq a \\
  b & a \leq r 
\end{cases} \quad \text{“spherical model”} \]
I don’t think that Matlab has Gaussian Process Regression (by any name), or at any rate I couldn’t find it. So, here is a wrapper for the NR3 class “Krig”.

Now it’s really easy:
Kriging results for our same 4-dimensional example:

```matlab
ptsmat = zeros(npts,4);
for j=1:npts, ptsmat(j,:) = pts{j}; end;
for j=1:npts, tptsmat(j,:) = tpts{j}; end;
kri g(ptsmat, vals, 0, 1.8);
kvals = zeros(npts,1);
for j=1:npts, kvals(j) = krig(tpts{j}); end;
plot(tvals, kvals, '.')
```

This is the exponent in the power-law model. 1.5 is usually good. Closer to 2 for smoother functions.

About 3x worse than our best RBF, but way better than Shepard.
How do the error estimates compare to the actual errors?

```matlab
kerrs = zeros(npts,1);
for j=1:npts, [kvals(j) kerrs(j)] = krig(tpts{j}); end;
plot(abs(kvals-tvals),kerrs,'.'
```

Note that we don’t expect, or get, a point-by-point correlation, but only that the actual error is plausibly drawn from a Normal distribution with the indicated standard deviation.
The real power of Kriging is its ability to incorporate an error model, taking us past interpolation and to fitting / smoothing / filtering.

interpolation: \((\sigma\text{'s} = 0)\)
filtering: (σ’s ≠ 0)

1-σ error band should contain true curve ~2/3 of the time.