# APTS Statistical Computing Assessment 2023: Thin Plate Splines

The work provided here is intended to take up to half a week to complete. Students should talk to their supervisors to find out whether or not their department requires this work as part of any formal accreditation process. Departments decide on the appropriate *level* of assessment locally, and may choose to drop some (or indeed all) of the parts, accordingly. In order to avoid undermining institutions' local assessment procedures the module lecturer will not respond to enquiries from students about this assignment.

Thin plate splines are a non-parametric way of estimating a function of several variables from data. The idea is that you have a model:

$$y_i = f(\mathbf{x}_i) + \epsilon_i, \qquad i = 1, \ldots, n$$

where $\mathbf{x}_i$ and $y_i$ are observed, $f$ is an unknown smooth function, and $\epsilon_i$ a zero mean error term with variance $\sigma^2$. $f$ can be represented using a *thin plate spline* function. In principle we can do this for any dimension of $\mathbf{x}_i$, but for now let's stick to dimension 2, bivariate smoothing.

To keep computation efficient we start by choosing $k$ points $\mathbf{x}_j^*$ spread 'nicely' throughout the the $\mathbf{x}_i$ points (if $n$ is not too large we might just set $k = n$ and set $\mathbf{x}_i^* = \mathbf{x}_i$, otherwise we might randomly sample $k$ of the $\mathbf{x}_i$ points to use as $\mathbf{x}_j^*$ points). Then define a function

$$\eta(r) = \begin{cases} r^2 \log(r) & r > 0 \\ 0 & \text{otherwise} \end{cases}$$

and define $\eta_j(\mathbf{x}) = \eta(\|\mathbf{x} - \mathbf{x}_j^*\|)$. We can represent $f$ as

$$f(\mathbf{x}) = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \sum_{j=1}^{k} \eta_j(\mathbf{x}) \delta_j \tag{1}$$

where $\boldsymbol{\delta}$ and $\boldsymbol{\alpha}$ are parameter vectors to be estimated. Typically we choose $k$ to be fairly large, to avoid biasing estimates by using an overly restrictive model, but this may in turn allow the model to overfit. To avoid this the model is usually estimated by minimizing a weighted sum of lack of fit and wiggliness of $f$:

$$\sum_{i=1}^{n} \{y_i - f(\mathbf{x}_i)\}^2 + \lambda \int \frac{\partial^2 f}{\partial x_1^2}^2 + 2\frac{\partial^2 f}{\partial x_1 \partial x_2}^2 + \frac{\partial^2 f}{\partial x_2^2}^2 dx_1 dx_2$$

$\lambda$ controls the smoothness of the estimated $f$. The $\boldsymbol{\delta}$ and $\boldsymbol{\alpha}$ minimizing this turn out to be the minimizers of

$$\|\mathbf{y} - \mathbf{T}\boldsymbol{\alpha} - \mathbf{E}\boldsymbol{\delta}\|^2 + \lambda \boldsymbol{\delta}^T \mathbf{E}^* \boldsymbol{\delta} \text{ subject to } \mathbf{T}^{*\mathsf{T}} \boldsymbol{\delta} = \mathbf{0},$$

where $E_{ij}^* = \eta_j(\mathbf{x}_i^*)$, $E_{ij} = \eta_j(\mathbf{x}_i)$ and the norm is Euclidean. The ith row of $n \times 3$ matrix $\mathbf{T}$ is $(1, \mathbf{x}_i)$ and the ith row of $k \times 3$ matrix $\mathbf{T}^*$ is $(1, \mathbf{x}_i^*)$. The constraint is inconvenient, but can be eliminated by re-parameterization. Let

$$\mathbf{T}^* = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

and let $\mathbf{Z}$ be the last $k-3$ columns of $\mathbf{Q}$ (something like `Z <- qr.Q(qr(Ts),complete=TRUE)[,-(1:3)]` would find `Z` explicitly). Then $\boldsymbol{\delta} = \mathbf{Z}\boldsymbol{\delta}_z$ will always meet the constraint for any $\boldsymbol{\delta}_z$ vector. Hence we can re-write the objective to minimize as

$$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \boldsymbol{\beta}^T \mathbf{S} \boldsymbol{\beta} \text{ where } \mathbf{X} = [\mathbf{EZ}, \mathbf{T}], \mathbf{S} = \begin{bmatrix} \mathbf{Z}^T \mathbf{E}^* \mathbf{Z} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \text{ and } \boldsymbol{\beta} = \begin{bmatrix} \boldsymbol{\delta}_z \\ \boldsymbol{\alpha} \end{bmatrix}$$

Minimizing this w.r.t. $\boldsymbol{\beta}$ we get

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{S})^{-1} \mathbf{X}^T \mathbf{y}, \quad \hat{\boldsymbol{\mu}} = \mathbf{X}\hat{\boldsymbol{\beta}} \text{ and EDF} = \text{trace}\{(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{S})^{-1} \mathbf{X}^T \mathbf{X}\}$$

where $\hat{\boldsymbol{\mu}}$ is the model prediction of $E(\mathbf{y})$ and EDF is the *effective degrees of freedom* of the model, which is between 3 and $k$, reflecting the fact that we have constrained the model fit by imposing the penalty during fitting.

A popular method for estimating $\lambda$ is generalized cross validation, which chooses $\lambda$ to minimize

$$V(\lambda) = \|\mathbf{y} - \hat{\boldsymbol{\mu}}\|^2 / (n - \text{EDF})^2.$$

This gets rather expensive if we have to compute the EDF and $\hat{\boldsymbol{\mu}}$ using the above formulae, but things can be made much cheaper with some further transformations of the problem. First form a new QR decomposition $\mathbf{X} = \mathbf{QR}$ (here $\mathbf{Q}$ has the same dimension as $\mathbf{X}$, see `?qr.Q` to extract it from the result of `qr(X)`), and then the symmetric eigen decomposition $\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T = \mathbf{R}^{-T}\mathbf{S}\mathbf{R}^{-1}$. Then it easy to show that $\hat{\boldsymbol{\beta}} = \mathbf{R}^{-1}\mathbf{U}(\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\mathbf{U}^T\mathbf{Q}^T\mathbf{y}$ and $\text{EDF} = \text{tr}\{(\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\}$. Now $\mathbf{I} + \lambda\boldsymbol{\Lambda}$ is a diagonal matrix, so if $\mathbf{a} = (\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\mathbf{b}$ then $a_i = b_i/(1 + \lambda\Lambda_{ii})$ - i.e. there is no expensive matrix inversion to be done. The same goes for the EDF calculation. Hence the computation of the GCV score is now very cheap for each trial $\lambda$ value.

1. Show that the reparameterization $\boldsymbol{\delta} = \mathbf{Z}\boldsymbol{\delta}_z$ indeed results in $\boldsymbol{\delta}$ always meeting the constraint.
2. Show that the efficient expressions to the EDF and $\hat{\boldsymbol{\beta}}$ are computing the same thing as the original (inefficient) expressions.
3. Write a function, `fitTPS` for fitting thin plate splines to $\mathbf{x}, y$ data, choosing the smoothing parameter by GCV. Give the list object returned by your function a class `tps`, and also write a `plot.tps` method function for this class.

   In detail:

   - `fitTPS(x,y,k=100,lsp=c(-5,5))` should be a function with arguments

     `x` an $n \times 2$ matrix of location values.

     `y` the $n$ vector of values to smooth.

     `k` the number of basis functions to use.

     `lsp` the $\log\lambda$ limits between which to search for the optimal smoothing parameter value.

     To optimize the GCV score, simply evaluate it on a regular (log scale) grid of smoothing parameter values between the log limits given, and pick the value giving the best result. The function should return an object of class `tps`, which should be a list probably containing at least the following items:

     `beta` the best fit $\hat{\boldsymbol{\beta}}$ at the optimal $\lambda$ value.

     `mu` the corresponding $\hat{\boldsymbol{\mu}}$.

     `medf` the effective degrees of freedom at the optimal $\lambda$ value.

     `lambda` the vector of 100 $\lambda$ values searched over.

     `gcv` the corresponding GCV score vector.

     `edf` the corresponding vector of EDFs.

     Other items will be needed to enable predictions from the model.

   - `plot.tps` should have as its first argument an object of class `tps` as returned from `fitTPS`. `plot.tps` should plot your fitted thin plate spline as a contour plot or a perspective plot. Note that to predict from the fitted model you can transform from $\hat{\boldsymbol{\beta}}$ to $\hat{\boldsymbol{\alpha}}$ and $\hat{\boldsymbol{\delta}}$ and use (1). Alternatively produce the equivalent of $\mathbf{X}$ for the new $\mathbf{x}$ values at which you want to predict, but using the $\mathbf{Z}$ computed for fitting.

Here is some code to generate example test data and plot the true function (`ff`) underlying it.

```
ff <- function(x) exp(-(x[,1]-.3)^2/.2^2-(x[,2] - .3)^2/.3^2)*.5 +
                  exp(-(x[,1]-.7)^2/.25^2 - (x[,2] - .8 )^2/.3^2)
n <- 500
x <- matrix(runif(n*2),n,2)
y <- ff(x) + rnorm(n)*.1 ## generate example data to fit
## visualize test function ff...
m <- 50;x2 <- x1 <- seq(0,1,length=m)
xp <- cbind(rep(x1,m),rep(x2,each=m))
contour(x1,x2,matrix(ff(xp),m,m))
persp(x1,x2,matrix(ff(xp),m,m),theta=30,phi=30)
```