Practical 3: Smoothing with basis expansions and penalties

Smoothing and function estimation play important parts in applied statistics and data science. One approach combines basis expansions and penalized regression, both techniques with much wider application. In this practical you (working individually) will write R functions for smoothing x, y data. The idea is that you have a model:

$$y_i = f(x_i) + \epsilon_i, \qquad i = 1, \ldots, n$$

where x_i and y_i are observed, f is an unknown smooth function, and ϵ_i a zero mean error term with variance σ^2. To estimate f you approximate it using a *basis expansion*, $f(x) = \sum_{j=1}^k \beta_j b_j(x)$, where the $b_j(x)$ are known *basis functions* and the coefficients, β_j, are to be estimated. Here you will use evenly spaced *B-spline* basis functions. These are bell shaped functions, each simply translations of each other along the x axis. They are so constructed that, when evaluated at any x value within the range of the data, the evaluations sum to 1. Having chosen a basis, the model can be written as the linear model, $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $X_{ij} = b_j(x_i)$.

k is chosen generously, so that the approximation can accurately capture a wide range of function shapes and complexities, but that means that there is a danger of over-fitting noisy data (i.e. fitting the ϵ_i component of y_i, not just the $f(x_i)$ component). To avoid this we can impose a *smoothing penalty* to encourage β_js corresponding to neighbouring $b_j(x)$s to vary smoothly from one to the next. An example of such a penalty might be

$$\sum_{j=2}^{k-1} (\beta_{i-1} - 2\beta_i + \beta_{i+1})^2 = \boldsymbol{\beta}^T \mathbf{D}^T \mathbf{D} \boldsymbol{\beta}$$

where $\mathbf{D}$ is a $k - 2 \times k$ matrix of zeroes, except for $D_{i,i} = D_{i,i+2} = 1$ and $D_{i,i+1} = -2$ for $i = 1, \ldots, k-2$.

The model is then estimated by *penalized least squares*:

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\text{argmin}} \ \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \boldsymbol{\beta}^T \mathbf{D}^T \mathbf{D} \boldsymbol{\beta}$$

where λ is a *smoothing parameter* which we will need to choose somehow. High λ will give a straight line fit (whatever k is), while low λ will give a very wiggly fit. Smooth functions estimated like this are called *P-splines.*

It easy to show that $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{D}^T\mathbf{D})^{-1}\mathbf{X}^T\mathbf{y}$ (the fitted values are of course $\hat{\boldsymbol{\mu}} = \mathbf{X}\hat{\boldsymbol{\beta}}$). Given that our smooth function estimate can vary from something very wiggly to a simple straight line fit as λ increases, it does not make sense to treat its statistical degrees of freedom as k. Instead the *effective degrees of freedom*, $\kappa = \text{tr}\{(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{D}^T\mathbf{D})^{-1}\mathbf{X}^T\mathbf{X}\}$ is used. We can then immediately estimate the residual variance as $\hat{\sigma}^2 = \|\mathbf{y} - \hat{\boldsymbol{\mu}}\|^2/(n - \kappa)$, and it turns out that the (Bayesian) covariance matrix for the coefficients, $\boldsymbol{\beta}$, is $(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{D}^T\mathbf{D})^{-1}\hat{\sigma}^2$.

An effective way to choose λ is to find the value of λ minimizing the *generalized cross validation* criterion, GCV $= \hat{\sigma}^2/(n - \kappa)$. Computing GCV for many trial values of λ would be rather expensive if we directly used the expressions given above. But it turns out that if some matrix decompositions are done before we start searching for the optimal λ, then the search can be made very efficient. The key is to first form the QR decomposition $\mathbf{X} = \mathbf{QR}$. Then (defining eigen-decomposition $\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T = \mathbf{R}^{-T}\mathbf{D}^T\mathbf{D}\mathbf{R}^{-1}$)

$$\mathbf{X}^T\mathbf{X} + \lambda \mathbf{D}^T\mathbf{D} = \mathbf{R}^T\mathbf{R} + \lambda \mathbf{D}^T\mathbf{D} = \mathbf{R}^T(\mathbf{I} + \lambda \mathbf{R}^{-T}\mathbf{D}^T\mathbf{D}\mathbf{R}^{-1})\mathbf{R} = \mathbf{R}^T(\mathbf{I} + \lambda \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T)\mathbf{R} = \mathbf{R}^T\mathbf{U}(\mathbf{I} + \lambda\boldsymbol{\Lambda})\mathbf{U}^T\mathbf{R}$$

Hence $(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{D}^T\mathbf{D})^{-1} = \mathbf{R}^{-1}\mathbf{U}(\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\mathbf{U}^T\mathbf{R}^{-T}$ and $\hat{\boldsymbol{\beta}} = \mathbf{R}^{-1}\mathbf{U}(\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\mathbf{U}^T\mathbf{Q}^T\mathbf{y}$. It is also easy to show that $\kappa = \text{tr}\{(\mathbf{I} + \lambda\boldsymbol{\Lambda})^{-1}\}$. The point being that inversion of diagonal matrix $\mathbf{I} + \lambda\boldsymbol{\Lambda}$ takes only $O(k)$ operations for each new λ value, as opposed to the $O(k^3)$ needed by the original expressions.

Your task is to write a function for fitting P-splines to x, y data, choosing the smoothing parameter by GCV. Your function will return objects of class `pspline`, and you will also write, `print`, `predict` and `plot` method functions for this class. In detail:

1. `pspline(x,y,k=20,logsp=c(-5,5),bord=3,pord=2,ngrid=100)` should be a function with arguments:

   `x` and `y` – the vectors of x, y data to smooth.

   `k` – the number of basis functions to use.

   `logsp` – the ends of the interval over which to search for the smoothing parameter (log λ scale). If only a single value is provided then no searching is done, and the spline is returned for the given log λ value.

`bord` – the B-spline order to use: 3 corresponds to cubic.

`pord` – the order of difference to use in the penalty. 2 is for the penalty given above.

`ngrid` – the number of smoothing parameter values to try. You should use even spacing of values on the log scale.

The function must return an object (a list) of class `pspline` defining the best fit spline smoother. The returned object should contain elements: `coef` ($\hat{\boldsymbol{\beta}}$), `fitted` ($\hat{\boldsymbol{\mu}}$) and `sig2` ($\hat{\sigma}^2$). Decide yourself what else this object should contain, in order to provide what will be needed by the method functions. But note that it will definitely need to contain the `knots` vector used to set up the B-spline basis (without these you will not be able to make prediction work properly).

You will use the `splineDesign` function from R's built in `splines` package to set up the basis ($\mathbf{X}$ matrix). This is rather fiddly, so here is the code to do this (all variables as listed above):

```
dk <- diff(range(x))/(k-bord) ## knot spacing
knots <- seq(min(x)-dk*bord,by=dk,length=k+bord+1)
X <- splines::splineDesign(knots,x,ord=bord+1,outer.ok=TRUE)
```

The easiest way to generate the $\mathbf{D}$ matrix is simply `D <- diff(diag(k),differences=pord)`. I suggest that you try out the code for `X` and `D` to make sure that you know what it produces (choose some example `x` vectors and `k` values). It is a good idea to plot each column of `X` against the `x` used in producing it, in order to visualize the basis functions.

2. `print.pspline(m)` should be a method function reporting some details of the model fit, `m`. The output should look like this (with the numbers obviously referring to whatever smooth fit is being reported)

```
Order 3 p-spline with order 2 penalty
Effective degrees of freedom: 11.22137   Coefficients: 20
residual std dev: 22.67567   r-squared: 0.7797937   GCV: 4.222302
```

   Note that $r^2 = 1 - (n-1)\hat{\sigma}^2 / \sum_i (y_i - \bar{y})^2$. The function should silently return a list containing elements: `gcv` (the generalized cross validation criterion of the fitted model), `edf` (κ) and `r2` (r^2).

3. `predict.pspline(m,x,se=TRUE)` should make predictions from the smooth fit, for new `x` values (within the range of the original data). If `se=FALSE` then the vector of predictions is to be returned. If `se=TRUE` then a 2 item named list is to be returned, with predicted values in the `fit` item, and the corresponding standard errors in the `se` item. Predictions are made by creating a new model matrix for the new `x` data, using `splineDesign` (with the original knot positions and other settings). Let this matrix be called `Xp`. If `V` is the covariance matrix for the coefficients, then the required standard errors come from the square roots of the leading diagonals of the predicted value covariance matrix `Xp %*% V %*% t(Xp)`. However you definitely do not want to compute the latter matrix (that can end up costing more than model fitting). The standard errors you need are computable efficiently using `rowSums(Xp*(Xp%*%V))^.5`.

4. `plot.pspline`(m) should produce 3 plots. The first should be a plot of the original x,y data, with the estimated smooth function overlaid as a line, along with approximate 95% credible intervals for the smooth. The second should plot the model residuals against fitted values. The third should be a `qqplot` of the residuals. The function should silently return a list containing elements: `ll`, `ul` and `x` — vectors defining the the lower and upper confidence limits (as plotted) and the corresponding x values. `plot.pspline` should not reset graphical parameters (e.g. using `par`).

Note that the `mcycle` data from the `MASS` library provides one set of x, y (`times`, `accel`) data to try.

Working individually you should aim to produce well commented[1], clear and efficient code for the task. The code should be written in a plain text file called `proj3.r`, and is what you will submit. There should be only

[1] Good comments give an overview of what the code does, as well as line-by-line information to help the reader understand the code. Generally the code file should start with an overview of what the code in that file is about, and a high level outline of what it is doing. Similarly each function should start with a description of its inputs outputs and purpose plus a brief outline of how it works. Line-by-line comments aim to make the code easier to understand in detail.

functions in what you submit, so that when the file is sourced, functions are defined, but nothing else is run. Your solutions should use only the functions available in base R (or packages supplied with base R - no other packages). The work must be completed individually, and you must not share code for this task (you can discuss concepts of course). Note that the university has some automatic tools for detecting when code has been copied between groups (and also from online sources, and from work of students at other universities which has been submitted to a the same checking platform). Also, students tend to make very distinctive errors when coding that stand out even if obvious things are done to hide that code has been copied.

The first comment in your code should list your name and university user name. Your code file should not refer to this sheet, and should be sufficiently well commented that someone reading it can tell what it is about and the strategies you are using without reference to this sheet. However, your initial overview comments should be brief - reproducing all the maths in this sheet is not expected, just the bare essentials.

Your code will be subject to some auto-marking. For that reason it is **essential** that you stick exactly to the specification of the functions given, and that your submitted code does nothing but load functions when loaded into R using `source`. *You will lose marks if I have to edit your code in order to* `source` *it or run it.* Any tests or examples in your code file should be written into functions, but those functions should not be run when the file is `source`d.

One piece of work - the text file containing your commented R code - is to be submitted on Learn by 12:00 Friday 4th November 2022. Format the code and comments tidily, so that they display nicely on an 80 character width display, without line wraps (or only occasional ones). No extensions are available on this course, because of the frequency with which work has to be submitted. So late work will automatically attract a hefty penalty (of 100% after work has been marked and returned). Technology failures will not be accepted as a reason for lateness (unless it is provably the case that Learn was unavailable for an extended period), so aim to submit ahead of time.

Marking Scheme: Full marks will be obtained for code that:

1. does what it is supposed to do, and has been coded in R approximately as indicated (that is marks will be lost for simply finding a package or online code that simplifies the task for you).
2. produces correct results under a variety of automatic tests that functions written according to the specification should pass.
3. is carefully commented, so that someone reviewing the code can easily tell exactly what it is for, what it is doing and how it is doing it without having read this sheet, or knowing anything else about the code. Note that *easily tell* implies that the comments must also be as clear and *concise* as possible. You should assume that the reader knows basic R, but not that they know exactly what every function in R does.
4. is well structured and laid out, so that the code itself, and its underlying logic, are easy to follow.
5. is reasonably efficient, meaning that care has been taken to use the matrix decomposition approach given above to keep down the cost of fitting, with some care taken to avoid excessively costly matrix computations.
6. contains no evidence of having been copied, in whole or in part, from other students on this course, students at other universities (there are now tools to help detect this, including internationally), online sources etc.