12 Dealing with more complicated problems

In this final chapter we will learn ways to tackle more complicated problems. I first introduce the concepts of cross-validation and regularization as means for controlling the complexity of fitted models, taking again the example of curve fitting. Global polynomials have some unpleasant properties, so I will introduce more practical methods of curve fitting using local basis functions, in particular splines and regression kernels. I will finish up with an outline of two approaches that are useful when computing the entire posterior is difficult or unnecessary: numerical optimization for finding the maximum of the posterior and bootstrapping for estimating its variance.

12.1 Cross-validation

Suppose we have a data set $\{x, y\}$ for which there is unknown noise ϵ in the y values. We want to fit a polynomial function to this such that $y = f_J(x) + \epsilon$, where

$$f_J(x) = \sum_{j=0}^{J} x^j \beta_j \tag{12.1}$$

but we do not have a good idea of what order polynomial – value of J – to use. We cannot select the J that gives the best fit, because we could just keep increasing J until we fitted the data exactly. This is not what we want when we know there is noise in the data.

One approach is to calculate the Bayesian evidence for different order polynomials and select the one with the highest evidence. We did this in section 11.3 to decide whether a line with zero gradient (polynomial order zero) or finite gradient (order one) was better. However, using the evidence has some disadvantages.

- It is hard to decide on the priors for polynomial parameters, because our prior information probably doesn't come in the form of constraints on a polynomial. The impact of different values of the high order coefficients on the function is not very intuitive. (We will see in section 12.3 how we can overcome this to some extent.)
- The evidence is relatively slow to compute, because we often have to make millions or more likelihood calculations (unless we have a strong prior or can chose one that permits an analytic integration).
- If we cannot specify the noise model (for the uncertainties in y), then the likelihood is not defined so the evidence cannot be calculated.

Recall that we do not need to know the standard deviation of the noise: we saw in section 11.3 that once we have a noise model, e.g. a zero mean Gaussian with standard deviation σ, we can specify a prior on σ and marginalize over it. Often we can define a reasonable noise model, so the last point above is not usually a restriction in practice. Nonetheless, the following method will allow us to fit a suitable function, with complexity control, without an explicit noise model.

An alternative to using the evidence is cross-validation. Suppose we have N data points. We divide these at random into two sets, one called the *training set* D_{train} containing N_{train} data points, the other called the *test set* D_{test} containing N_{test} data points. We choose a polynomial order J, fit the line using the training set, then calculate the quality of the fit using the test set. If we're using least squares to fit the line (chapter 4), then the quality of the fit is measured by the sum of squared residuals on just the points in the test set

$$\text{SS}_{\text{test}}(J) = \sum_{i \in D_{\text{test}}} [y_i - f_J(x_i; D_{\text{train}})]^2 \tag{12.2}$$

where what comes after the ";" in the function indicates here the data used to fit the model. We repeat this for a range of J, and then choose the model – value of J – that gives the smallest value of the test error. By evaluating the fit on a different set of data from what was used to make the fit, we avoid selecting a model that *overfits* the data. A very high order polynomial may fit the training data very well – better than a lower order polynomial – but if much of the variation it fits is due to noise rather than an intrinsic variation, then we will measure a large sum of squared residuals on the test set. This is because although the test set will show the same intrinsic variation, its noise will be different (because noise is random). By splitting the data into separate train and test sets we establish how well the model solution generalizes. This is a form of *regularization*, which we will discuss more in section 12.2.1.

A disadvantage of this approach is that by keeping back data from the fitting we will not get such a good fit. We mitigate this problem via a variation known as *leave-one-out cross-validation*. We now define N training sets such that the ith, call it D_{-i}, contains all data points *except* the single point (x_i, y_i). For a given model J we fit the curve separately for each training set, and for each of these calculate the residual on the single point that was not in the training set. The quality of fit is the sum of squared residuals for all these fits, i.e.

$$\text{SS}_{\text{CV}}(J) = \sum_{i=1}^{N} [y_i - f_J(x_i; D_{-i})]^2. \tag{12.3}$$

To put the sum of squares on the scale of the data and so make it more interpretable we can form the root mean square (RMS) residual instead, $\sqrt{\text{SS}/N}$, which is a measure of the residual per point (cf. equation 4.25).

One can also make an intermediate approach called *K-fold cross-validation*, in which there are $K < N$ train/test sets of data with more than one data point in each test set. This is useful if N is too small for us to construct sufficiently large train and test sets, but too large for us to want to (or to need to) construct N models to do leave-one-out cross-validation.

Cross-validation effectively enables us to sample the variation in the underlying function *and* the noise using just the available data. The method does not distinguish between the two; nor does it need to. By evaluating the quality of the fit on the omitted data we are able to monitor this fit quality as a function of the complexity of the model. Models that are either too simple (low order) or too complex (high order) will both perform badly. This can be understood in terms of a trade-off between the bias and variance of the fit. Suppose we had N different sets of training data.

- If the model we are fitting has low order, then the N different fits on the N training sets will all be quite similar. Hence they will all produce similar sized residuals on the test set, so the variance of these residuals will be low. But their bias will be large, because the model is not able to fit the true complexity of the data.
- If the model has high order, then the N fits will now have much lower bias due to the increased flexibility of the model. But their variance will be comparatively large, because a small change in the training data will produce a significantly different fit (due to the sensitive high order terms), and thus quite different residuals on the test data.

We saw in section 4.8 that the expected squared residual is a combination of bias and variance. Thus the optimal fit – smallest residuals on the test data – will be achieved by a model lying between the two extremes.

The R code at the end of this section demonstrates cross-validation by fitting lines of order $J = 0$ to $J = 9$ to a set of $N = 25$ data points. These data come from a fourth-order polynomial to which I have added zero-mean Gaussian noise with standard deviation 250. The data are shown in the top-left panel of figure 12.1.

As we saw in section 4.5, the model can be written in matrix form as $\mathbf{f} = X\boldsymbol{\beta}$. In the code I use the function `poly` to define the polynomials of various orders. Given a scalar value x, `poly` (with the specification `raw=TRUE`) creates the vector of values $(x, x^2, \ldots, x^J)$, where J is specified by the parameter `degree`. Note that `poly` does not include the power of zero in its output. Given a vector of N elements, `poly` operates on each element separately to produce an $N \times J$ matrix X. My function `polyeval` does this and then multiplies the result by $\boldsymbol{\beta}$ to evaluate the model $X\boldsymbol{\beta}$ (the operator `%*%` does matrix/vector multiplication). `poly` is also used together with `lm` to fit the polynomials via least squares. Note how I set `subset=-i` in `lm` to exclude just point i from the fit for the leave-one-out cross-validation.

Each model is fit by minimizing the sum of squared residuals, for which an estimate of the noise is not required. The fits for each J are shown in the panels of figure 12.1 labelled $J = 0$ to $J = 9$. The bottom-right panel plots the RMS of the fits for each J, as defined earlier. For $J = 0$ we only have the offset term in the model, so the fit is just the mean of the training data. Interestingly in this case, $J = 0$ is marginally favoured over $J = 1$. We then see a significant drop in the RMS as J increases from $J = 1$ to $J = 4$, the true value. Fifth- and sixth-order polynomials give much the same performance, but as J increases beyond this, the solutions begin to get considerably worse due to the large variance of the fits – overfitting – which we can see in the panels at large x. It's clear too that these higher order polynomials will give very poor extrapolations of the data.

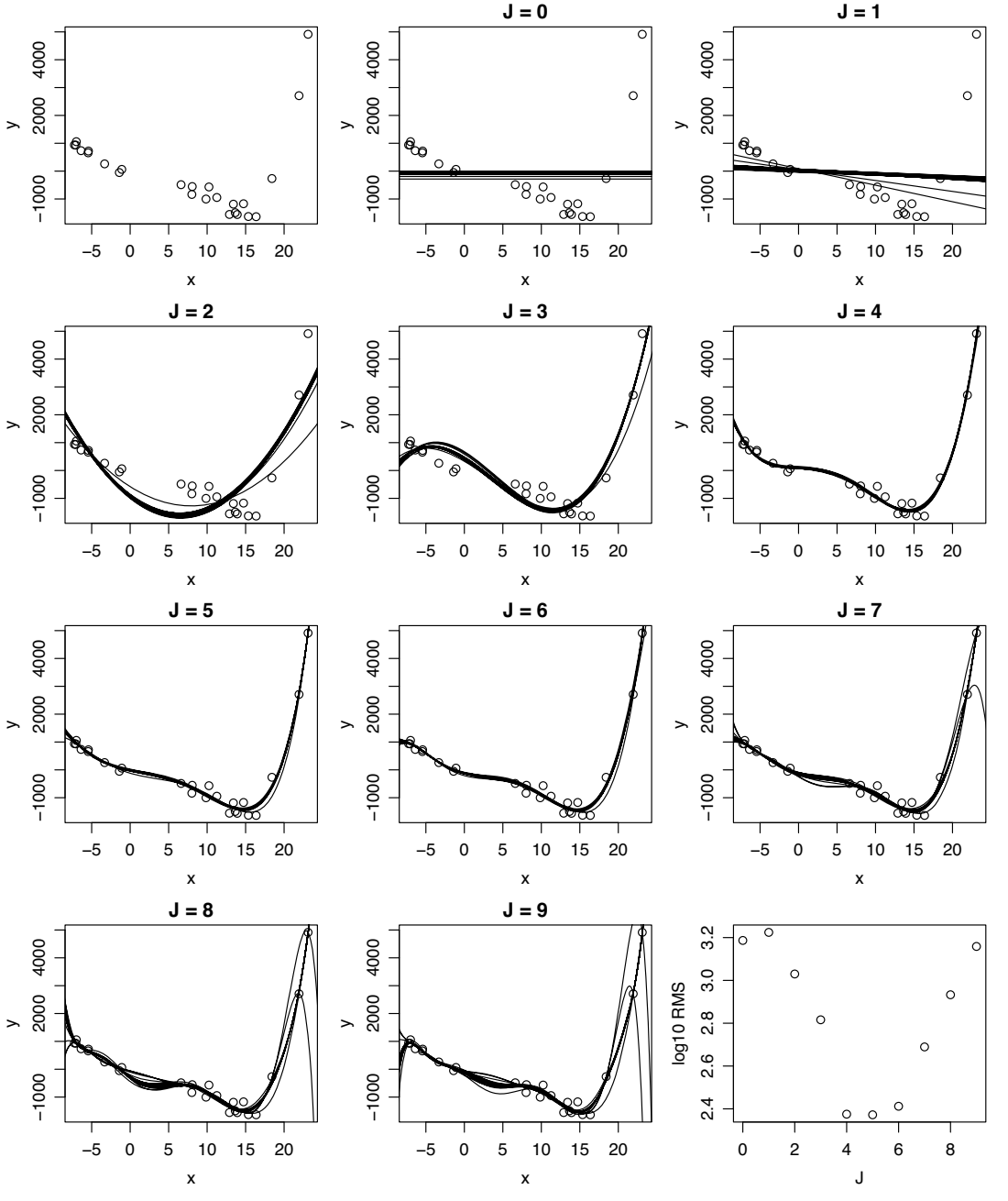

Fig. 12.1 Demonstration of leave-one-out cross-validation using polynomials. The top-left panel shows the data. The next ten panels labelled $J = 0$ to $J = 9$ show the $N = 25$ fits for each training data set for polynomials of order J. The bottom-right panel plots $\log \mathrm{RMS}$ vs J, where $\mathrm{RMS} = \sqrt{\mathrm{SS}_{\mathrm{CV}}/N}$ and $\mathrm{SS}_{\mathrm{CV}}$ is defined in equation 12.3.

Cross-validation is frequently used with nonlinear models of high-dimensional data sets, such as neural networks, and many other machine learning methods.[1]

R file: `CV_polynomials.R`

```
##### Demonstration of cross-validation model selection using polynomials

# Function to evaluate polynomial with parameter beta at x.
# poly() as used here excludes x^0 - the constant, beta[1] - so I remove
# it from the matrix multiplication, then add it explicitly afterwards.
polyeval <- function(x, beta) {
  return(poly(x=x, degree=length(beta)-1, raw=TRUE) %*% beta[-1] + beta[1])
}

pdf("CV_polynomials.pdf", 10, 12)
par(mfrow=c(4,3), mar=c(3.5,3.5,1.5,0.5), oma=c(0.5,0.5,1,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
set.seed(63)

# Simulate data: 4th order polynomial
beta <- c(0, 5, 1, -2, 0.1)
Ndat  <- 25
sigma <- 250
x <-  runif(Ndat, min=-8, max=25)
y <-  polyeval(x, beta) + rnorm(Ndat, 0, sigma)
# xp, yp just for plotting
xp <- seq(from=-10, to=30, length.out=1e3)
yp <- polyeval(xp, beta)
plot(x,y)
#lines(xp, yp, col="red")

# Do CV. Plot all fits for each j in a separate panel
jmax <- 9 # evaluate all polynomials with order from 0 to jmax
# Do j=0 separately
rss0 <- 0
plot(x, y, main=c("J = 0"))
for(i in 1:Ndat) {
  pred <- mean(y[-i])
  rss0 <- rss0 + (pred-y[i])^2
  abline(h=pred)
}
# Do j=1:jmax
rss <- vector(mode="numeric", length=jmax)
for(j in 1:jmax) {
  rss[j] <- 0
  plot(x, y, main=paste("J =", j))
  for(i in 1:Ndat) {
    # poly() used in lm() does include x^0 term
    mod  <- lm(y ~ poly(x, j, raw=TRUE), subset=-i)
    pred <- predict(mod, newdata=data.frame(x=x[i]))
    rss[j] <- rss[j] + (pred - y[i])^2
    lines(xp, predict(mod, newdata=data.frame(x=xp)))
  }
```

[1] Indeed, classic feedforward neural networks are nothing more than rather flexible, nonlinear basis function models with parameters normally fit by optimization, with the complexity of the fit controlled by cross-validation.

```
}
plot(0:jmax, log10(sqrt(c(rss0, rss)/Ndat)), xlab="J", ylab="log10 RMS")
dev.off()
```

12.2 Regularization in regression

12.2.1 Regularization

A fundamental principle employed in science is the *principle of parsimony*. This states that we should try to explain data with the simplest model possible. One can think of arbitrarily complex and bizarre ways to explain data. Prior implausibility aside, experience shows that overly complex models make poor predictions. They fit the available data too precisely, so are unable to generalize to the broader situation. In other words, such models have high variance. In contrast, overly simple models will not explain the data well enough: they have a large bias. A model should be as simple as possible, but as complex as necessary.

Another name for this concept is *Occam's razor*: if each of a set of models explains the data equally well, choose the simplest, by which we mean the one with the fewest assumptions. This principle does not say we should always favour the simplest model. A complex problem may need a complex solution.

The cross-validation technique in the previous section controlled the complexity of the model (the order of the polynomial in the example used) by using the variance in the data, and seeing how well its solution generalized to data left out. We saw in section 11.5 how the Bayesian evidence automatically performs complexity control at the model level via the Occam factor, by trading-off model fit with model complexity. We can also apply this idea of regularization at the level of the individual model. We will see how to do this here and will then see that the procedure has a probabilistic interpretation.

Consider a data set $\{x_i, y_i\}$ (for $i = 1 \ldots N$) with unknown noise, to which we want to fit a polynomial of order J. We write this as $y_i = f(\mathbf{x}_i) + \epsilon$, where

$$f(\mathbf{x}_i) = \sum_{j=0}^{J} x_i^j \beta_j = \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta} \tag{12.4}$$

with $\mathbf{x}_i^\mathsf{T} = (1, x_i, x_i^2, \ldots, x_i^J)$. Note the 1 in this vector, which multiplies β_0 to provide the constant term in the fit. The larger the absolute values of the polynomial coefficients in the fitted model, the more complex the function, because large coefficients produce more bendy curves (see figure 12.1). Thus a term such as

$$\sum_{j=0}^{J} \beta_j^2 \tag{12.5}$$

is a measure of the complexity of the model. As it stands, this expression is inconsistent, because each β_j is a coefficient of a different power of x and thus has different units. It makes no sense to sum terms with different units. We remedy this by *standardizing* the

data. This means transforming each component of $\mathbf{x}$ to have zero mean and unit standard deviation over the N data points. Thus for $j = 1$ we do

$$x_i \rightarrow \frac{x_i - \overline{x}}{\sigma_x} \tag{12.6}$$

where $\overline{x}$ and σ_x are the mean and standard deviation (respectively) of $\{x_i\}$. We do the same for the other powers of x (components of $\mathbf{x}$). Once standardized, the components of $\mathbf{x}$ are unitless and each component of $\boldsymbol{\beta}$ has units y. If we also centre the $\{y\}$, $y_i \rightarrow y_i - \bar{y}$, then we don't need the constant term in the model, so we can set $\beta_0 = 0$ and drop the constant from the definition of $\mathbf{x}$.

A good fit demands that we minimize the residual sum of squares (RSS), but this will get smaller the larger J is. So instead of minimizing the RSS we minimize the objective function

$$\mathcal{E} = \sum_{i=1}^{N} [y_i - f_J(x_i)]^2 + \lambda \sum_{j=1}^{J} \beta_j^2 \tag{12.7}$$

where $\lambda \geq 0$. The first term is the RSS and the second term is our measure of model complexity. As we want to minimize $\mathcal{E}$, this corresponds to a complexity penalty. The size of the parameter λ determines the degree of the penalty: a larger λ means more penalty (more regularization), and so less complex solutions. As β_j^2 has units of y^2 (due to the standardization), λ is unitless. A relatively complex model will achieve a small value for the RSS by fitting the data closely, but it will have a large value of $\sum \beta_j^2$. The larger λ, the more this complexity term contributes to the objective function $\mathcal{E}$. By minimizing $\mathcal{E}$ with fixed λ, we achieve a set of coefficients $\{\beta_j\}$ that give us the optimal trade-off between fitting on the one hand and complexity on the other.

Fitting models with a penalty term is called *regularization*. In the context of least squares, this particular type of regularization is called *Tikhonov regularization*. Linear regression with this regularizer is called *ridge regression*, which we specify and solve in the next section.

12.2.2 Ridge regression

Section 4.5 showed the matrix formulation for ordinary least squares linear regression with J input variables. Section 4.6 showed how this can also be applied to polynomial regression in one-dimension with a Jth order polynomial.

Ridge regression can be seen as a generalization of these. Proceeding as in section 4.5, we first write the input data as an $N \times J$ design matrix X, in which each row is the vector of J features for input i. That is, the ith row is the vector of J input dimensions if we're doing linear multi-dimensional regression, or the vector $(x_i, x_i^2, \ldots, x_i^J)$ if we're doing polynomial regression. We standardize this, so each column of X has zero mean and unit standard deviation. $\mathbf{y}$ is the N-dimensional vector $(y_1, y_2, \ldots, y_N)$. This we centre, so it has zero mean. $\boldsymbol{\beta}$ is the J-dimensional vector of parameters, so the regularization term is $\boldsymbol{\beta}^\mathsf{T}\boldsymbol{\beta}$ (a scalar product).

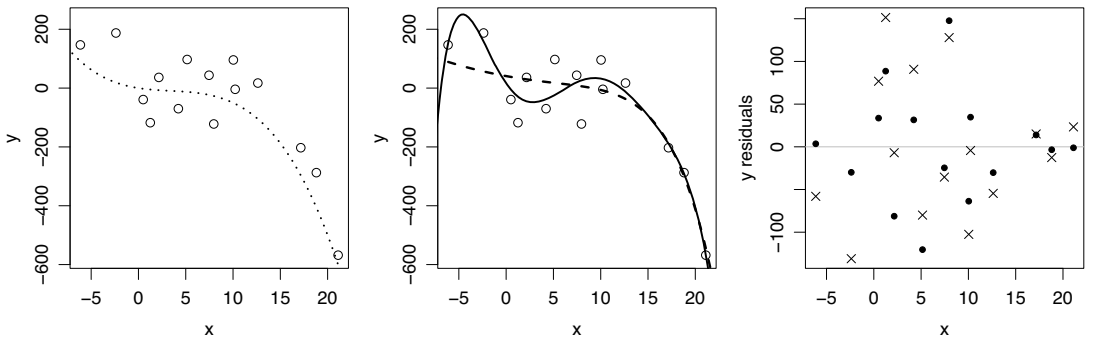

Fig. 12.2 Demonstration of ridge regression. Left: the data (open circles) have been computed from a third-order polynomial (dotted line) to which Gaussian noise have been added. Centre: ordinary least squares (solid line) and ridge regression (dashed line) fits to the data using sixth-order polynomials in both cases. Right: residuals for ordinary least squares (filled circles) and ridge regression (crosses).

Following from equation 12.7, we want to minimize

$$\mathcal{E} = (\mathbf{y} - X\boldsymbol{\beta})^{\mathsf{T}}(\mathbf{y} - X\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^{\mathsf{T}}\boldsymbol{\beta} \tag{12.8}$$

with respect to $\boldsymbol{\beta}$ for a given λ. Differentiating and setting to zero we get

$$\begin{aligned} \mathbf{0} &= -X^{\mathsf{T}}(\mathbf{y} - X\boldsymbol{\beta}) + \lambda\boldsymbol{\beta} \\ \mathbf{0} &= -X^{\mathsf{T}}\mathbf{y} + X^{\mathsf{T}}X\boldsymbol{\beta} + \lambda\boldsymbol{\beta} \\ \boldsymbol{\beta} &= (X^{\mathsf{T}}X + \lambda I)^{-1}X^{\mathsf{T}}\mathbf{y} \end{aligned} \tag{12.9}$$

where I is the identity matrix. Compare this with the ordinary least squares solution in equation 4.56. They are of course the same when $\lambda = 0$, as this corresponds to no regularization. Given that $|X^{\mathsf{T}}X + \lambda I| > |X^{\mathsf{T}}X|$ for $\lambda > 0$, we see that $|\boldsymbol{\beta}|$ is smaller in ridge regression than in ordinary regression. Also, the term $(X^{\mathsf{T}}X + \lambda I)$ will always be invertible for $\lambda > 0$, so we will always get a solution, unlike with ordinary least squares. Thus ridge regression gives solutions to ill-posed problems, which is another purpose of regularization.

To use ridge regression effectively we have to decide on a sensible value of λ. Although λ is unitless, we can see from equation 12.7 that its value will not be invariant under an increase in the size of the data set, for example. But if we offset or rescale x or y, we don't need to change λ (on account of the standardization). Finding an appropriate value for λ can be done to some degree by trial and error. Once in the right vicinity we could use cross-validation (section 12.1) to find the most appropriate value among some set. Several of the R methods use *generalized cross-validation* to do this. This is an approximation to leave-one-out cross-validation which is generally faster to compute.[2]

[2] Generalized cross-validation essentially replaces the individual elements of the smoother matrix – defined later – with the average of the trace of this matrix.

Table 12.1 Coefficients for the ordinary least squares and ridge regression solutions shown in figure 12.2.

order	β_{OLS}	β_{ridge}
1	−363	−45.7
2	1117	43.6
3	1538	−18.8
4	−9588	−36.9
5	11743	−54.4
6	−4654	−63.7

Figure 12.2 compares the solutions obtained with ridge regression (using $\lambda = 1$) and with ordinary least squares (OLS) regression, using a sixth-order polynomial in both cases. The data are $N = 15$ points computed from a third-order (cubic) polynomial to which Gaussian noise have been added. The plot shows the original data (prior to standardization) and the fits have been transformed back to the non-standardized scales. The values of the resulting coefficients in the two cases are shown in Table 12.1 (these apply to the standardized data.) All of the coefficients from the ridge regression have much smaller absolute values than those from the ordinary regression. The sum of squares of the coefficients is 2.6×10^8 and 1.3×10^4 for ordinary least squares and ridge regression respectively. The RMS of the residuals for the fits is 63.7 and 80.0 respectively. So ridge regression does what it should: it produces a fit with a slightly larger RMS but with significantly smaller coefficients.

The R below code implements the above and produces figure 12.2. I encourage you to experiment with the code by changing the value of λ (on the line where `betaRidge` is computed), the order of the models (`degree`), and the amount and complexity of the data. In the code I again use the function `poly` and `polyeval` as discussed in the previous section. In the line

```
X <- scale(poly(x, degree=6, raw=TRUE))
```

`poly` is applied to all N (`Ndat`) values of x to produce an $N \times J$ matrix. This is then standardized using the function `scale` to produce the $N \times J$ design matrix `X`. The vector `xp` is a dense vector of size N_p that I use to compute the fitted functions in order to plot them. To compute this, `xp` (and its powers up to J) must be standardized using the means and standard deviations computed on the data; we must not compute new means and standard deviations! The numbers required for this can be extracted from the standardized design matrix `X` using the `attr` function. R is a bit counter-intuitive when it comes to mixed matrix/vector algebra. One might think that

```
XP - attr(X,"scaled:center")
```

would subtract the $1 \times J$ vector `attr(X,"scaled:center")` from each row of the $N_p \times J$

matrix XP. But R does column-wise operations, so we first need to transpose XP, then subtract the vector of means, divide by the vector of standard deviations, and finally transpose back again. This is done with the following code.

```
t( (t(XP) - attr(X,"scaled:center")) / attr(X,"scaled:scale") )
```

Note that the plots are of the raw (non-standardized) data, and the resulting curves are converted back from the centered coordinates so they can be plotted with these raw data.

R file: ridge_regression.R

```
##### Demonstration of ridge regression for 1D polynomials

# Function to evaluate polynomial with parameter beta at x.
# poly() as used here excludes x^0 - the constant, beta[1] - so I remove
# it from the matrix multiplication, then add it explicitly afterwards.
polyeval <- function(x, beta) {
  return(poly(x=x, degree=length(beta)-1, raw=TRUE) %*% beta[-1] + beta[1])
}

# Function to solve for ridge regression parameters
# given data vector y of length N, N*J data matrix X,
# and lambda (scalar).
param.ridge <- function(y, X, lambda) {
  return( solve(t(X)%*%X + diag(lambda, nrow=ncol(X))) %*% t(X) %*% y )
}

pdf("ridge_regression.pdf", 12, 4)
par(mfrow=c(1,3), mar=c(3.5,3.5,0.5,0.5), oma=c(0.5,0.1,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.2)

set.seed(100)

# Simulate data: 3rd order (cubic) polynomial
beta <- c(0, -5, 1, -0.1)
Ndat  <- 15
sigma <- 100
x <-  runif(Ndat, min=-8, max=25)
y <-  polyeval(x, beta) + rnorm(Ndat, 0, sigma)
# xp, yp just for plotting
xp <- seq(from=-10, to=27, length.out=1e3)
yp <- polyeval(xp, beta)

# Plot data and true model
plot(x, y, ylim=c(-580, 240))
lines(xp, yp, lty=3, lwd=2)

# Build matrices, centre y, centre and scale each power of x. "degree" in
# poly defines the order of the polynomial we use in both solutions.
ys <- scale(y, scale=FALSE)
X  <- scale(poly(x, degree=6, raw=TRUE)) # Ndat * 6 matrix
XP <- poly(xp, degree=6, raw=TRUE)       # 1e3  * 6 matrix
XP <- t( (t(XP) - attr(X,"scaled:center")) / attr(X,"scaled:scale") )
# Solve for OLS and ridge regression parameters and calculate residuals
betaOLS  <- param.ridge(ys, X, lambda=0)
residOLS <- X %*% betaOLS - ys
```

```
betaRidge  <- param.ridge(ys, X, lambda=1)
residRidge <- X %*% betaRidge - ys

# Plot raw (non-standardized) data together with fitted curves.
# Latter is done by plotting de-centered model predictions at raw xp.
plot(x, y, ylim=c(-580, 240))
lines(xp, XP %*% betaOLS   + attr(ys, "scaled:center"), lwd=2)
lines(xp, XP %*% betaRidge + attr(ys, "scaled:center"), lwd=2.5, lty=2)

# Plot residuals
plot(x, y, type="n", ylim=range(c(residOLS, residRidge)),
     ylab="y residuals")
abline(h=0, col="grey")
points(x, residOLS,   pch=20)
points(x, residRidge, pch=4)

dev.off()

# Print coefficients, sum of squares of coefficients, and RMS of residuals
format(data.frame(betaOLS, betaRidge), digits=3)
cat("beta^2 = ", sum(betaOLS^2), sum(betaRidge^2), "\n")
cat("RMS =    ", sqrt(mean(residOLS^2)), sqrt(mean(residRidge^2)), "\n")
```

12.2.3 Probabilistic interpretation of regularization

We saw in section 4.4.2 how ordinary least squares was equivalent to maximum likelihood for a Gaussian likelihood. Following the same approach, we can interpret the regularizer in ridge regression as a prior on the model parameters. Taking the logarithm of Bayes' theorem gives

$$\ln P(\boldsymbol{\beta}\,|\,\mathbf{y}) \;=\; \ln P(\mathbf{y}\,|\,\boldsymbol{\beta}) + \ln P(\boldsymbol{\beta}) + \text{constant} \tag{12.10}$$

where for brevity I have dropped the conditioning on both the model M and the input data X. The constant (the evidence) is independent of $\boldsymbol{\beta}$. Suppose the likelihood $P(\mathbf{y}\,|\,\boldsymbol{\beta})$ is Gaussian with covariance matrix Σ (equation 4.58) and we adopt a prior

$$P(\boldsymbol{\beta}) \;=\; \exp\left(-\frac{1}{2}\lambda\boldsymbol{\beta}^{\mathsf{T}}\boldsymbol{\beta}\right). \tag{12.11}$$

Equation 12.10 then becomes

$$\ln P(\boldsymbol{\beta}\,|\,\mathbf{y}) \;=\; -\frac{1}{2}(\mathbf{y}-X\boldsymbol{\beta})^{\mathsf{T}}\Sigma^{-1}(\mathbf{y}-X\boldsymbol{\beta}) - \frac{1}{2}\lambda\boldsymbol{\beta}^{\mathsf{T}}\boldsymbol{\beta} + \text{constant} \tag{12.12}$$

where the constant still only includes terms independent of $\boldsymbol{\beta}$. Comparing this to equation 12.8 (with $\Sigma = I$), we see that the objective function used in ridge regression is equal to -2 times the logarithm of the posterior PDF over the model parameters (to within an additive constant). Thus by minimizing the objective function we maximize the posterior. This is true even when the data have (common) error bars σ, as the scaling provided by a diagonal covariance matrix ($\Sigma = \sigma^2 I$) simply changes the relative sizes of the posterior and prior terms, a freedom which can be absorbed into the regularization parameter λ. For

the more general case of different sized error bars and/or covariance, we can modify the objective function in ridge regression to be the negative of equation 12.12.

Thus we can interpret the regularizer as a prior on the solution that would otherwise be obtained with maximum likelihood. We have seen the effect of priors in several places in this book. Maximum likelihood estimates sometimes suffer from high variance on account of overfitting data, so the prior can play an important role by reducing this variance.

12.3 Regression with basis functions

A polynomial – equation 12.1 – is an example of a *basis function*. Another well-known one is a Fourier series, useful for representing periodic functions, which is

$$f(x) = a_0 + \sum_{j=1}^{J} a_j \cos\left(\frac{2\pi j x}{L}\right) + b_j \sin\left(\frac{2\pi j x}{L}\right) \tag{12.13}$$

for some constant L. The coefficients of this basis function are $\{a_j, b_j\}$. A basis function representation is anything that represents a function as a linear combination of functions, so in general we can write it as

$$f(x) = \sum_{j=1}^{J} \beta_j h_j(x). \tag{12.14}$$

12.3.1 Splines

Polynomials are not very nice functions to work with. Low order polynomials are not flexible enough to fit most data, yet once we use orders beyond a few, polynomials produce wild variations in a desperate attempt to fit the given data points. This is exacerbated by the fact that the polynomials we have been using are global: they extend across the whole data range. Often we would like to have a curve that gives increased curvature (flexibility) just in certain parts of the data space, without producing a big wobble somewhere else. This can be achieved using localized basis functions. A common example is splines.

The idea of a spline is to split the data into regions and to fit each separately. We consider them in one dimension. We define K points – called knots – along the x-axis at $x = \{\upsilon_k\}$. These knots divide the data space into $K + 1$ regions, whereby the first region spans the range $-\infty$ to υ_1 and the last region spans the range υ_K to $+\infty$. We now define $K + 1$ basis functions, each of which describes the function over just one region.

A simple spline is a piecewise constant spline, in which we fit a constant separately to each region. The next simplest polynomial is a piecewise linear spline, in which we fit a straight line to each region, and so on for higher orders. Examples of such splines are shown in figure 12.3. Such piecewise functions (of any order) have the problem that they are not continuous at the knots, because each region is fit separately. One usually wouldn't even call such functions "splines".

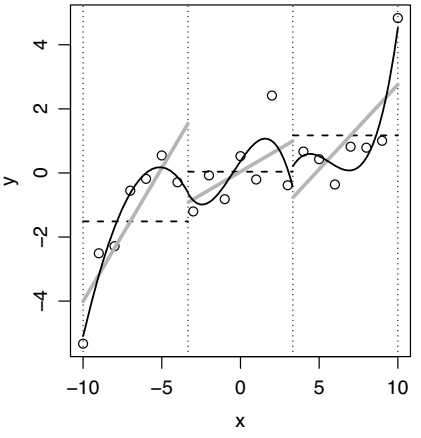

Fig. 12.3 Piecewise constant (dashed black line), linear (grey line), and cubic (solid black curve) functions fitted to the data (open circles) independently in each region between the knots (indicated by the vertical dotted lines). This region-independent fitting causes the functions to be discontinuous at the knots (even for the cubic function at the second knot: we just got lucky that it is nearly continuous in this example).

We can force the function to be continuous at the knots by defining a different set of basis functions. This will be easier if I first define the following notation. For any scalar a let

$$(a)_+ = \begin{cases} a & \text{if} \quad a > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{12.15}$$

One possible basis function set to produce a *continuous* piecewise linear function is

$$\begin{aligned} h_1(x) &= 1 \\ h_2(x) &= x \\ h_{k+2}(x) &= (x - \upsilon_k)_+ \quad \text{where} \quad k = 1 \ldots K. \end{aligned} \tag{12.16}$$

This defines $K + 2$ basis functions, essentially one for each of the $K + 1$ regions plus a constant offset. In fact the last basis function spans the last region, the one before last the last two regions, etc., but this amounts to (and achieves) the same thing. The function itself is given by equation 12.14. $K + 1$ independent linear fits would require $2(K + 1)$ parameters. But with K knots we have K continuity constraints and so K fewer free parameters. Thus we have $2(K + 1) - K = K + 2$ parameters, which is the number of coefficients of the basis function in equation 12.16 ($J = K + 2$). We can then proceed to infer these parameters using, for example, least squares.

The grey line in figure 12.4 shows an example of such a continuous piecewise linear spline with four knots. We can see how the continuity conditions at the knots modify the fit from comparison to figure 12.3: the boundary conditions force the fit in the central region

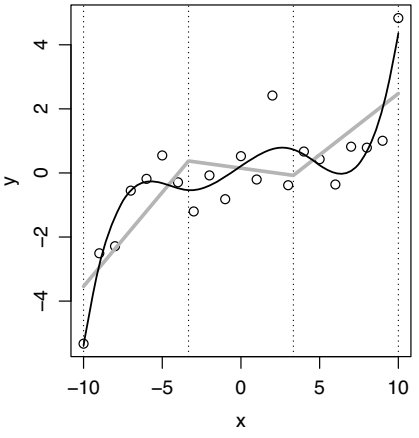

Fig. 12.4 Continuous linear spline (grey line) and continuous cubic spline (black line), each with four knots at the positions shown by the vertical dotted lines, fitted to 21 data points (open circles).

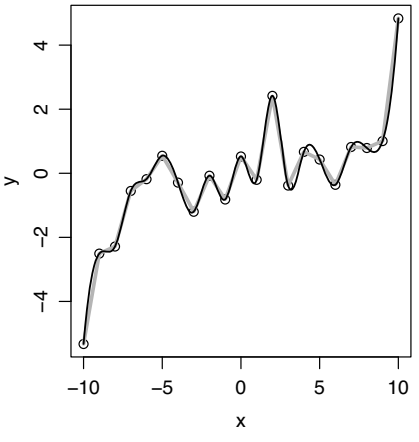

Fig. 12.5 As figure 12.4 but now with a knot at every data point, so that the function interpolates the points exactly.

to have a negative gradient, for example. The grey line in figure 12.5 shows the case when there is a knot at each data point. The function of course fits the data exactly.

Although such functions are continuous at the knots, they are not smooth, because their derivatives are not continuous. We can enforce this by using higher order piecewise functions. A quadratic spline has continuous gradients (first derivatives) but still has discontinuous second derivatives at the knots. Such fits look odd. A cubic spline – using cubic polynomials – is continuous in the zeroth, first, and second derivatives, and is the lowest order spline that looks smooth to the human brain (aliens may think otherwise). The basis

functions are

$$\begin{aligned} h_1(x) &= 1 \\ h_2(x) &= x \\ h_3(x) &= x^2 \\ h_4(x) &= x^3 \\ h_{k+4}(x) &= (x - \upsilon_k)_+^3 \quad \text{where} \quad k = 1 \ldots K. \end{aligned} \tag{12.17}$$

An arbitrary cubic equation would require four parameters in each region, thus requiring a total of $4(K+1)$ parameters. The cubic spline has three continuity constrains for each knot, and thus $3K$ fewer parameters, leaving a total of $4(K+1)-3K = K+4$ parameters, which is equal to the number of basis functions ($J = K+4$). The black line figure 12.4 shows an example of this with four knots.

The more knots we choose, the better the curve will fit the data. If we set the number of knots equal to the number of data points, then the spline will fit the data exactly, as can be seen in figure 12.5.

12.3.2 Smoothing splines

A practical issue with splines is deciding how many knots to use. The more we use the better the fit, to the point of an exact fit when $K = N$, the number of data points. As the data are presumably noisy, this will be an overfit. A solution to this is to use *smoothing splines*. We put a knot at every data point ($K = N$) but then penalize complex solutions through the addition of a regularizer (section 12.2.1). A measure of the complexity of a function is its degree of curvature, which in turn is measured by the magnitude of the second derivative. Thus a suitable regularizer is

$$\int \left(\frac{d^2 f}{dx^2} \right)^2 dx. \tag{12.18}$$

A smoothing spline uses a cubic spline basis (equation 12.17) to minimize

$$\mathcal{E} = \sum_i [y_i - f(x_i)]^2 + \lambda \int \left(\frac{d^2 f}{dt^2} \right)^2 dt. \tag{12.19}$$

where t is a dummy variable for x. Compare this with the ridge regression objective function (equation 12.8). Instead of penalizing the magnitude of the polynomial coefficients equally for all components of the basis function, a smoothing spline penalizes the overall curvature of the function. It is instructive to write this objective function in matrix format. For N data points the basis function expansion (equation 12.14) can be written $\mathbf{f} = H\boldsymbol{\beta}$ where H is the $N \times J$ matrix of basis function evaluations. Equation 12.19 can then be written as

$$\mathcal{E} = (\mathbf{y} - H\boldsymbol{\beta})^\mathsf{T}(\mathbf{y} - H\boldsymbol{\beta}) + \lambda \boldsymbol{\beta}^\mathsf{T} \Omega \boldsymbol{\beta} \tag{12.20}$$

where Ω is a $J \times J$ matrix with elements

$$\Omega_{j,j'} = \int \frac{d^2 h_j}{dt^2} \frac{d^2 h_{j'}}{dt^2} \, dt. \tag{12.21}$$

The regularizer may be written like this because the function is a linear combination of the basis functions.[3] Differentiating the objective function with respect to $\boldsymbol{\beta}$ and setting to zero we get the solution for $\boldsymbol{\beta}$

$$\begin{aligned} \mathbf{0} &= -H^{\mathsf{T}}(\mathbf{y} - H\boldsymbol{\beta}) + \lambda\Omega\boldsymbol{\beta} \\ \mathbf{0} &= -H^{\mathsf{T}}\mathbf{y} + (H^{\mathsf{T}}H + \lambda\Omega)\boldsymbol{\beta} \\ \boldsymbol{\beta} &= (H^{\mathsf{T}}H + \lambda\Omega)^{-1}H^{\mathsf{T}}\mathbf{y}. \end{aligned} \tag{12.22}$$

This has the same form as the solution for ridge regression (equation 12.9), except that the data are now expressed through basis functions, and the regularizer is not the identity matrix but the matrix of integrals of the products of the second derivatives of the basis functions.

The values of the model function at the N inputs are

$$\mathbf{f}(\mathbf{x}) = H\boldsymbol{\beta} \tag{12.23}$$

which can be written

$$\begin{aligned} \mathbf{f}(\mathbf{x}) &= S\mathbf{y} \quad \text{where} \\ S &= H(H^{\mathsf{T}}H + \lambda\Omega)^{-1}H^{\mathsf{T}}. \end{aligned} \tag{12.24}$$

The $N \times N$ matrix S is called the *smoother matrix*. It depends only on the fixed input data $\{x\}$ and the regularization constant λ, but not on $\mathbf{y}$. Equation 12.24 tells us that the fitted curve is therefore linear in $\mathbf{y}$.

We saw in section 4.5 that the model predictions arising from ordinary least squares could be written in a similar form to this, namely $\mathbf{f}(\mathbf{x}) = X\boldsymbol{\beta}$, and its corresponding smoother matrix is $S_{\text{OLS}} = X(X^{\mathsf{T}}X)^{-1}X^{\mathsf{T}}$ (see equation 4.57). Assuming $(X^{\mathsf{T}}X)$ is not a singular matrix, this has a unique solution with $N - J$ degrees of freedom, where J is the number of parameters in the model. It turns out that the degrees of freedom is also given by the trace of the matrix, trace(S_{OLS}). So for any model fit that can be written as $S\mathbf{y}$ where S does not depend on $\mathbf{y}$, we can define the *effective degrees of freedom* as trace(S). For splines, S depends only on $\{x\}$ and λ, so for given input data, specifying the effective degrees of freedom is equivalent to specifying λ. This can be a more intuitive way of setting the degree of regularization.

Figure 12.6 shows smoothing splines with different degrees of smoothing.

12.3.3 R code

You can use the following code to experiment with the splines discussed in the previous section. Run as is, it will create the plots shown in figures 12.3 to 12.6. The data are `x` and `y`. Curves are plotted by evaluating them on a grid of x points defined as the vector `xp`.

To fit the discontinuous splines (figure 12.3), I explicitly fit functions separately to the regions between the knots. I define these regions using two lists: `xSeg` to partition the data `x` and `xpSeg` to partition the plotting points `xp`. `xSeg[[s]]` indexes the points in `x`

[3] This also takes care of the potentially different units of the different basis functions h_j, because β_j has units y/h_j. The regularizer λ has units x^3.

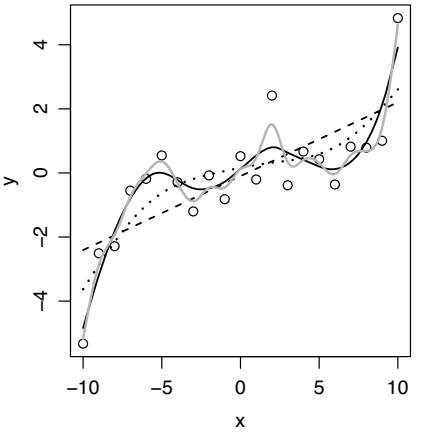

Fig. 12.6 A smoothing spline fitted to 21 data points (open circles) using four different values for the effective degrees of freedom (amounts of smoothing): 2 (dashed black line), 4 (dotted black line), 8 (solid black line), 15 (grey line). A spline with 21 effective degrees of freedom would fit the points exactly.

corresponding to region `s` (1, 2, or 3), and likewise for `xpSeg[[s]]`. The data are fitted using `lm` on the `subset` of the `x` data in that region. `xp[xpSeg[[s]]]` gives the x values of the points in the region at which we wish to evaluate the function, which is done with `predict`. The plot is then made with `lines`.

To fit and plot the continuous splines (figure 12.4) I again use `predict` and `lm`, but now with the spline basis functions defined by `bs`. Combined with `lm` this automatically divides the data up into the regions specified by the knots, fits them with the boundary conditions appropriate to the order (`degree`), and predicts only at the `xp` values within the region.

For the exact splines (figure 12.5) I use the function `approxfun`, which does exact linear (or constant) interpolation, and `splinefun`, which does exact cubic spline interpolation.

For the smoothing splines (figure 12.6) I use `smooth.spline`, which allows us to specify the amount of smoothing using the degrees of freedom. If you want to specify the number of knots instead, the syntax (for four knots) is as follows.

```
smooth.spline(x, y, spar=0, cv=FALSE, all.knots=FALSE, nknots=4)
```

R file: `splines.R`

```
##### Demonstration of 1D splines

library(splines) # for bs

# Simulate data
set.seed(101)
x <- -10:10
y <- sin(2*pi*x/10) + 0.005*x^3 + 1*rnorm(length(x), 0, 1)
xp <- seq(from=min(x), to=max(x), length.out=1e3) # for plotting splines
```

```
# Discontinuous splines of orders 0,1,3 with definable knots (here 4)
# I use lm() to explicitly fit within each region.
pdf("splines_discontinuous.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.5,3.5,0.5,1), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x,y)
knots=c(-3.33, 3.33)
abline(v=c(knots, range(x)), lty=3)
xSeg <- list(which(x<=knots[1]), which(x>knots[1] & x<knots[2]),
             which(x>=knots[2]))
xpSeg <- list(which(xp<=knots[1]), which(xp>knots[1] & xp<knots[2]),
              which(xp>=knots[2]))
for(s in 1:3) { # loop over the three subregions
  lines(xp[xpSeg[[s]]], predict(lm(y ~ 1, subset=xSeg[[s]]),
                                newdata=data.frame(x=xp[xpSeg[[s]]])),
        lwd=1.5, lty=2)
  lines(xp[xpSeg[[s]]], predict(lm(y ~ x, subset=xSeg[[s]]),
                                newdata=data.frame(x=xp[xpSeg[[s]]])),
        lwd=3, col="grey70")
  lines(xp[xpSeg[[s]]], predict(lm(y ~ x + I(x^2) + I(x^3),
                                   subset=xSeg[[s]]),
                                newdata=data.frame(x=xp[xpSeg[[s]]])),
        lwd=1.5, col="black")
}
dev.off()

# Continuous splines of orders 1,3 with definable knots (here 4)
# bs() automatically uses extreme data points as additional knots.
pdf("splines_continuous.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.5,3.5,0.5,1), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x,y)
knots=c(-3.33, 3.33)
abline(v=c(knots, range(x)), lty=3)
lines(xp, predict(lm(y ~ bs(x, knots=knots, degree=1)),
                  newdata=data.frame(x=xp)), lwd=3, col="grey70")
lines(xp, predict(lm(y ~ bs(x, knots=knots, degree=3)),
                  newdata=data.frame(x=xp)), lwd=1.5, col="black")
dev.off()

# Exact splines (i.e knot at each point) of orders 1,3
pdf("splines_exact.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.5,3.5,0.5,1), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x,y)
# Plots exact linear spline then exact cubic spline
lines(xp, approxfun(x, y, method="linear")(xp), lwd=3, col="grey70")
lines(xp, splinefun(x, y)(xp), lwd=1.5, col="black")
dev.off()

# Smoothing splines with various degrees of freedom
pdf("splines_smoothing.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.5,3.5,0.5,1), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)
plot(x,y)
```

```
lines(predict(smooth.spline(x, y, df= 2), xp), lwd=1.5, lty=2, col="black")
lines(predict(smooth.spline(x, y, df= 4), xp), lwd=2,   lty=3, col="black")
lines(predict(smooth.spline(x, y, df= 8), xp), lwd=1.5, lty=1, col="black")
lines(predict(smooth.spline(x, y, df=15), xp), lwd=2,   lty=1, col="grey70")
dev.off()
```

12.4 Regression kernels

The smoothing spline achieves flexibility by defining a knot at every data point and then fitting a cubic function to each with two sets of constraints: (1) continuity of the zeroth, first, and second derivatives; (2) a regularization term. There are other approaches to achieve a smooth fit, and one is to use kernels. I introduced kernels in section 7.2.2 as a way of doing density estimation. A kernel is a weighted function of data in which the weights depend on the distance of the data points from the point of evaluation. Let's see how we can use kernels for doing local regression.

Consider a set of N data points $\{x_i, y_i\}$ for which we want to get an estimate of the regression function $f(x)$. A simple kernel estimate of the function at any x is just the mean of all those points within a distance λ, i.e.

$$f(x) = \frac{1}{N_\lambda} \sum_{i \in D_\lambda(x)} y_i \tag{12.25}$$

where $D_\lambda(x)$ is that set of points, and N_λ is its size. While this gives an estimate anywhere, we do not infer a global parametric function for the data, and also not an additive model like a basis function expansion. Rather we evaluate the function at specific points. To plot our estimate of the function, we simply evaluate it on a sufficiently dense and broad grid of points. For this reason, this approach to regression is often called a *non-parametric model*. We can think of moving a box over the data and taking the average in it as we go. This will smooth out the variations in the data. We can also use a variable size box by using a nearest neighbour kernel instead, as defined in section 7.2.2.

We are not limited to taking the mean of the points within the kernel. We could take the median instead, in which case we end up with a *median filter*. We could also fit a straight line, or quadratic, or higher order polynomial to the points within the kernel, all of which are examples of *local polynomial regression*. In practice we would not usually go beyond a quadratic function, and linear often suffices. Note that even if we fit a linear function locally, the resulting function is not globally linear.

The problem with these kinds of functions/filters is that they are often not very smooth, because as we move the filter, points jump in and out of the kernel, changing the mean, median, or polynomial fit, sometimes by large amounts. A solution to this is to assign weights to data points that decrease smoothly with increasing distance from the centre of the kernel. Thus we can generalize the above considerations to estimate the function as

$$f(x) = \frac{\sum_{i=1}^{N} K(x, x_i)\, y_i}{\sum_{i=1}^{N} K(x, x_i)} \tag{12.26}$$

which is the *Nadaraya–Watson kernel estimator*. Defining

$$u = \frac{x - x_i}{\lambda} \tag{12.27}$$

we normally write the kernel as $K(u)$, where λ is its bandwidth. One choice for the kernel is a Gaussian

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}u^2\right). \tag{12.28}$$

We don't actually need the normalization constant for regression because it will cancel out in equation 12.26, but I will write all kernels here as normalized. The Gaussian kernel has the disadvantage of infinite support (it is non-zero everywhere): distant points can influence the function, which is often undesirable. For this reason we often use truncated kernels. A popular choice is the Epanechnikov kernel (see figure 12.7)

$$K(u) = \begin{cases} \frac{3}{4}(1-u^2) & \text{if} \quad |u| \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{12.29}$$

The larger the bandwidth the greater the number of points in the kernel, so the more the estimator will smooth out the data (smaller variance, but larger bias). A small kernel, in contrast, will follow the variations in the data more closely (larger variance, but smaller bias). We saw in section 7.2.2 one method for determining the bandwidth from the data using the L^2 risk function. We could also use a kernel with a variable-sized bandwidth, which is what we get with the k nearest neighbours kernel. This can be useful if the density of the data along with x-axis is very non-uniform. With a fixed bandwidth (and kernel with finite support), λ must be at least as large as half the separation between the two most-separated neighbouring points in order to get an estimate everywhere. Yet this may be so large that it would over smooth the data where it is dense (leading to a large bias).

The following R code experiments with kernels for one-dimensional regression, by applying three different kernel methods to smooth the data points (open circles) shown in figure 12.8. The code first uses `ksmooth` (from the `stats` library) to do zeroth-order ($f(x) = b_0$) kernel regression with a constant Nadaraya–Watson kernel (`kernel="box"`). We see from the figure that the resulting curve is discontinuous. If you experiment with the size of the bandwidth you will find that these discontinuities are determined by the density of the data, and do not really vanish even if you increase the bandwidth. If you change the kernel to do Gaussian (`kernel="normal"`) rather than constant weighting, you get a much smoother curve.

The code then uses `locpoly` (from the `KernSmooth` library) to do first-order ($f(x) = b_0 + b_1 x$) kernel regression, again with constant weighting. Increasing the order of this polynomial (with `degree`) does not have much impact on the shape of the curve.

The third method is `loess` (from the `stats` library). This smooths the data by fitting a first-order polynomial to the 20 nearest neighbours (set by `span`). The `loess` function

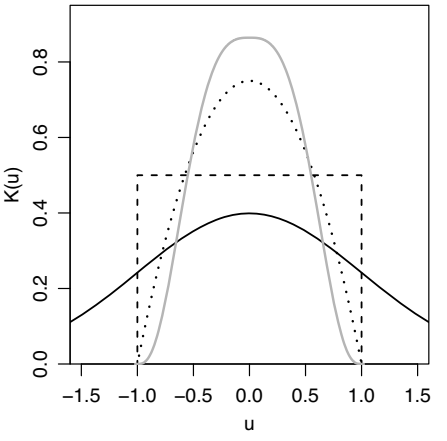

Fig. 12.7 Four different normalized kernels: uniform (dashed black line), Gaussian (solid black line), Epanechnikov (dotted black line), and tricubic (grey line). Apart from the Gaussian, these are non-zero only for $|u| < 1$.

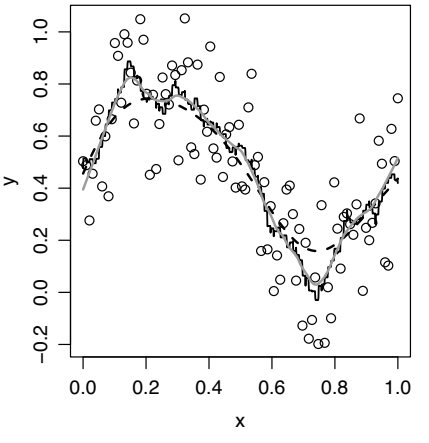

Fig. 12.8 Three different kernel smoothers fitted to some data points (open circles). Solid black line: constant kernel with a bandwith of 0.1 (using `ksmooth`). Dashed black line: local linear function regression with Gaussian kernel weighting with a bandwith of 0.1 (using `locpoly`). Grey line: local linear function regression with constant kernel weighting using nearest 20 neighbours (using `loess`).

weights the points using a tricubic kernel

$$K(u) = \begin{cases} \frac{70}{81}(1 - |u|^3)^3 & \text{if} \quad |u| \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{12.30}$$

This is also shown in figure 12.7. It has a slightly nicer behaviour near to $|u| = 1$ than the Epanechnikov kernel.

I encourage you to experiment with changing the parameters in this code, in particular the degree of polynomials (by setting `degree` to 0, 1, or 2) and the bandwidth of the kernels (by setting `bandwidth`). Check the help pages of the methods to see how the bandwidths are defined. For all three cases the smoothing function is evaluated on a dense grid of points (much denser than the data). I then use `lines` to connect these in the plot to give the appearance of a continuous line. For `ksmooth` and `loess` I define this grid as `xp`, with 1000 equally spaced points. For `locpoly` the grid is defined internally to the function; the default is 400 equally spaced points.

R file: `regression_kernels.R`

```
##### Demonstration of regression kernels

library(KernSmooth) # for locpoly

pdf("regression_kernels.pdf", 4, 4)
par(mfrow=c(1,1), mar=c(3.5,3.5,0.5,1), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)

# Simulate data
x <- seq(from=0, to=1, length.out=100)
f <- function(x){0.5 + 0.4*sin(2*pi*x)}
set.seed(10)
y <- f(x) + rnorm(n=x, mean=0, sd=0.2)
plot(x, y)

# Constant regression using a constant kernel of specified bandwidth.
xp <- seq(from=min(x), to=max(x), length.out=1e3)
lines(xp, ksmooth(x=x, y=y, kernel="box", bandwidth=0.1, x.points=xp)$y,
      lwd=1.5)

# Local polynomial (oder=degree) regression using a Gaussian kernel
# with specified bandwidth and polynomial degree (here constant)
mod <- locpoly(x=x, y=y, degree=1, bandwidth=0.1, gridsize=1e3)
lines(mod$x, mod$y, lwd=2, lty=2)

# Local polynomial (oder=degree) regression using fraction span of data
# around each prediction point, i.e. span*length(x) nearest neighbours.
# It won't work if span is too small.
xp <- seq(from=min(x), to=max(x), length.out=1e3)
yp <- predict(loess(y ~ x, span=0.2, degree=1), newdata=data.frame(x=xp))
lines(xp, yp, lwd=2, col="grey60")

dev.off()
```

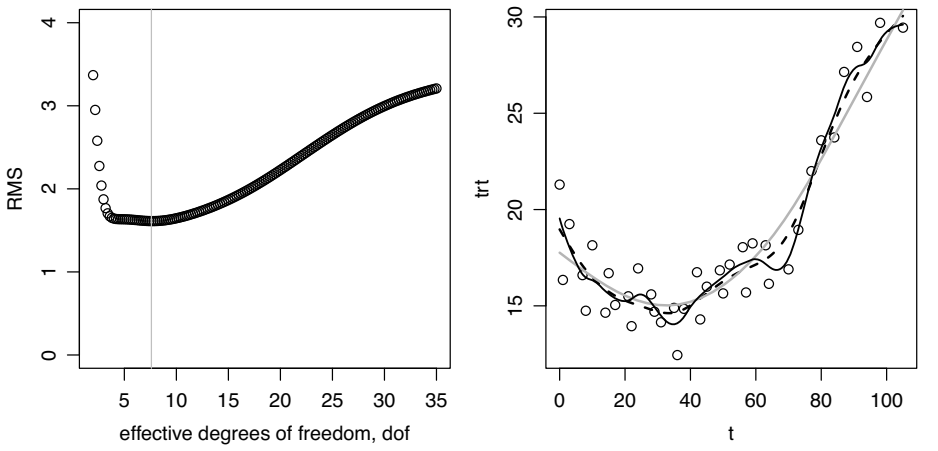

Fig. 12.9 Applying a smoothing spline to the rat diet data. The left panel shows how the root mean square (RMS) of the residuals varies as a function of the degrees of freedom in the smoothing spline, computed using cross-validation. The minimum, marked with a vertical dashed line, is at 7.6. The black dashed line in the right panel is the corresponding smoothing spline. The grey line and solid black line are smoothers with half and double this number of degrees of freedom respectively.

12.5 A non-parametric smoothing problem

I'll wrap up this analysis of models for curve fitting by applying both smoothing splines and a kernel regression smoother to the same data and comparing the results. For this we will use the "rat diet" (!) data set embedded in the `fields` package in R. The scientific content doesn't concern us here. We are just interested in getting a smooth, plausible fit of the variable `rat.diet$trt` to the variable `rat.diet$t` for the $N = 39$ data points.

I start with the smoothing spline (section 12.3.2), whereby I now use the `sreg` function in the package `fields`. I control the amount of smoothing through the effective degrees of freedom (dof). I vary this from 2 to 35 in steps of 0.2. For each value of dof, I use leave-one-out cross-validation to calculate the RMS of the fits. That is, for each dof I fit N cubic splines, each using a different set of $N - 1$ data points, and I calculate the residual on the one point left out for that N. These are combined to form the RMS. The results of this are shown in the left panel of figure 12.9. The smallest RMS of 1.61 is achieved with dof $= 7.6$. The data are shown in the right panel, overplotted with this best fitting smoother, as well as that smoother with dof equal to half and twice the optimal value. Fewer degrees of freedom corresponds to a smoother line. The splines are plotted by computing them on a dense grid of 1000 points. The R code to do this investigation and to make this plot is below.

R file: `ratdiet_splines.R`

```
##### Application of smoothing splines to the rat.diet data

library(fields) # for sreg and rat.diet
Ndat <- nrow(rat.diet)

# Calculate RMS using LOO-CV
dofValues <- seq(2,35,0.2)
Ndof <- length(dofValues)
rss  <- rep(x=0, times=Ndof)
for (k in 1:Ndof) {
  for(i in 1:Ndat) {
    mod  <- sreg(rat.diet$t[-i], rat.diet$trt[-i], df=dofValues[k])
    pred <- predict(mod, rat.diet$t[i])
    rss[k] <- rss[k] + (pred - rat.diet$trt[i])^2
  }
}
rms <- sqrt(rss/Ndat)

pdf("ratdiet_splines.pdf", 8, 4)
par(mfrow=c(1,2), mar=c(3.5,3.5,0.5,0.5), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)

# Plot RMS vs dof
plot(dofValues, rms, xlab="effective degrees of freedom, dof", ylab="RMS",
     ylim=c(0,4))
bd <- dofValues[which.min(rms)]
abline(v=bd, col="grey")
text(bd, 5, bd, pos=4)
cat(bd, rms[which.min(rms)])

# Plot data with three different spline smoothers
plot(rat.diet$t, rat.diet$trt, xlab="t", ylab="trt")
xp <- seq(from=min(rat.diet$t), to=max(rat.diet$t), length.out=1e3)
lines(xp, predict(sreg(rat.diet$t, rat.diet$trt, df=bd),   xp), lwd=2,
      lty=2, col="black")
lines(xp, predict(sreg(rat.diet$t, rat.diet$trt, df=bd/2), xp), lwd=2,
      lty=1, col="grey70")
lines(xp, predict(sreg(rat.diet$t, rat.diet$trt, df=2*bd), xp), lwd=1.5,
      lty=1, col="black")

dev.off()
```

I now apply a kernel smoother (section 12.4) to the same data, using the nearest neighbour kernel and doing local linear function regression. I again use the function `loess`. In its tricubic kernel (equation 12.30), λ is the distance to the furthest neighbour. The amount of smoothing is controlled via the number of nearest neighbours k. Again I use leave-one-out cross-validation to calculate the RMS of the fits, and plot this now against the number of neighbours. The number of neighbours is specified using `span`, as the fraction of the data available (i.e. `k/Ndat`). The result is shown in the left panel of figure 12.10. We see that the minimum is much flatter than the corresponding plot from the smoothing splines (figure 12.9). The optimal value is at $k = 16$ neighbours and achieves an RMS of 1.48, slightly smaller than with the smoothing splines. The data are shown in the right panel, overplotted with this best fitting smoother, as well as the smoother with half and twice as many neigh-

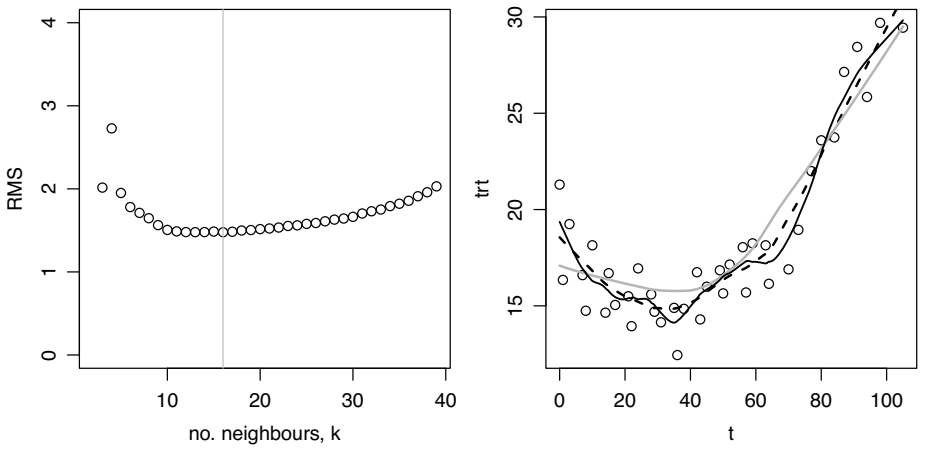

Fig. 12.10 Applying the kernel regression method to the rat diet data. The left panel shows how the root mean square of the residuals varies as a function of the number of neighbours used in the kernel. The vertical range is the same as in figure 12.9 to ease comparison. The point at $k = 2$ is off the scale with an RMS of 18.4 (it would be the left-most point). The minimum, marked with a vertical dashed line, is at 16 neighbours. The black dashed line in the right panel is the corresponding regression curve. The solid black line and grey line are smoothers with half (8) and double (32) this number of neighbours respectively.

bours. More neighbours corresponds to a smoother line. As before, the smoother is plotted by computing it on a dense grid of 1000 points. This is all done by the R code below.

R file: `ratdiet_kernelregression.R`

```
##### Application of kernel regression to the rat.diet data

library(fields) # for rat.diet
attach(rat.diet)
Ndat <- length(t)

# Calculate RMS using LOO-CV
# loess does not work with just one nearest neighbour, so I just do 2:Ndat.
# It also does not permit extrapolation, so I don't include the two test
# sets which would each be just the two extreme points. One could set
# surface="direct" as a loess option, but this gives worse fits for low k.
rss <- rep(NA, Ndat)
kRange <- 2:Ndat
for(k in kRange) {
  rss[k] <- 0
  for(i in 2:(Ndat-1)){ # RSS will be sum of squares of Ndat-2 residuals
    pred   <- predict(loess(trt ~ t, span=k/Ndat, degree=1, subset=-i),
                      newdata=data.frame(t=t[i]))
    rss[k] <- rss[k] + (pred - trt[i])^2
  }
```

```
}
rms <- sqrt((rss[kRange])/(Ndat-2))

pdf("ratdiet_kernelregression.pdf", 8, 4)
par(mfrow=c(1,2), mar=c(3.5,3.5,0.5,0.5), oma=c(0.5,0.5,0.5,0.5),
    mgp=c(2.2,0.8,0), cex=1.0)

# Plot RMS vs number of neighbours
plot(kRange, rms, xlab="no. neighbours, k", ylab="RMS", ylim=c(0,4))
bd <- kRange[which.min(rms)]
abline(v=bd, col="grey")
text(bd, 5, bd, pos=4)
cat(bd, rms[which.min(rms)])

# Plot data with three different kernel smoothers
plot(t, trt)
xp <- seq(from=min(t), to=max(t), length.out=1e3)
yp <- predict(loess(trt ~ t, span=16/Ndat, degree=1),
              newdata=data.frame(t=xp))
lines(xp, yp, lwd=2,   lty=2, col="black")
yp <- predict(loess(trt ~ t, span=32/Ndat, degree=1),
              newdata=data.frame(t=xp))
lines(xp, yp, lwd=2,   lty=1, col="grey70")
yp <- predict(loess(trt ~ t, span=8/Ndat, degree=1),
              newdata=data.frame(t=xp))
lines(xp, yp, lwd=1.5, lty=1, col="black")

dev.off()
detach(rat.diet)
```

12.6 Numerical optimization (mode finding)

When faced with a parameter inference problem, we ideally want to find the full posterior PDF over the model parameters. But for high-dimensional problems this may be too difficult or too time consuming. For other problems the posterior may be so sharply peaked that finding the full posterior is unnecessary: the maximum plus some estimate of its width is a sufficiently good approximation (as obtained by the quadratic approximation in section 7.1, for example). This can occur when the data are highly informative.

In such situations we will want to find the maximum of the posterior (or likelihood), which we discussed in section 4.4. In principle this is found by differentiation, but for most problems there will not be an analytic solution. We must instead proceed iteratively. The principle is straightforward. We define a starting point, take a sensible step in the parameter space, re-evaluate the function, and either terminate because we're sufficiently close to a maximum, or we iterate. Like Monte Carlo (chapter 8) this is a step-wise approach, but now we are no longer interested in a representative sampling of the distribution; we just want to get to a maximum (within some tolerance) as fast as possible. It is important to realise that generic posteriors are multimodal; an optimization method will find a local maximum, which is not necessarily the global maximum. In general it is impossible to know, without

evaluating the posterior everywhere, whether the maximum found is the global one or just a local one. Nonetheless, if the maximum found is "high enough" it may be an adequate solution. Posteriors can have complex shapes, especially in high dimensions, so a local or global maximum could lie far from the mean or median.

There is a vast literature on numerical optimization methods so I will just give a brief and selective introduction here.

Let $f(\theta)$ be the function we wish to maximize with respect to θ. If θ is a scalar, then straightforward methods like the golden section search can be used to find the maximum once it is known to lie within some interval. Here I consider the more general (and difficult) problem in which $\boldsymbol{\theta}$ is an N-dimensional vector (so I write it in boldface to emphasize this) in which case it is no longer possible to bracket the maximum with a finite number of points.

A method that just uses evaluations of the function is the *simplex method*, also known as the Nelder-Mead or amoeba method. The simplex in N dimensions is an object with $N+1$ vertices. In two dimensions this is a triangle, in three a tetrahedron, and so on. We first initialize the problem by selecting $N+1$ values of $\boldsymbol{\theta}$ and computing $f(\boldsymbol{\theta})$ at these. We identify the "worst" of these (the one with the lowest value of f) and compute a new point by reflecting this point through the centroid of the other N points. If the value of f at the new point is higher than at the other points, then it replaces the worst point. If not, there is a more involved scheme for deciding how update the points. This procedure is iterated, with the result that the simplex steps slowly towards the maximum. Various criteria exist to determine exactly how the simplex behaves and how the points are updated, with the goal of achieving faster convergence.

Methods which take advantage of knowledge of the gradient $\boldsymbol{\nabla} f$ may be more efficient. The simplest of these is *gradient ascent*. If the current position is $f(\boldsymbol{\theta}_i)$, then we take a small step in the direction of positive gradient to give the next position

$$\boldsymbol{\theta}_{i+1} \;=\; \boldsymbol{\theta}_i + \alpha\, \boldsymbol{\nabla} f(\boldsymbol{\theta}_i) \tag{12.31}$$

where α is a small positive quantity. (The positive sign in front of α should be switched to a negative sign if we wish to minimize f.) Note that α is not dimensionless: an optimal size depends both on the scale of the parameters and the gradient. It could be adapted at each iteration. A larger α will make larger steps and potentially achieve faster convergence, but if the gradient is changing rapidly this could lead to stepping over the maximum and then continuing the search in the wrong part of the parameter space. While simple, gradient ascent can be notoriously slow to converge if the search finds itself on a nearly flat ridge. Like all gradient-based methods it can also get stuck at local maxima. Various ways exist to circumvent this, such as adding to the update a proportion of the gradient from the previous iteration, $\boldsymbol{\nabla} f(\boldsymbol{\theta}_{i-1})$, or restarting the algorithm at a different point. I will use gradient descent in section 12.7.

Gradient methods can be accelerated further by using the second derivatives of f, if they exist. This is done by *Newton's method*. If Ω_i is the Hessian matrix – the matrix of all second partial derivatives – evaluated at $\boldsymbol{\theta}_i$, then the update rule is

$$\boldsymbol{\theta}_{i+1} \;=\; \boldsymbol{\theta}_i - \alpha\, \Omega_i^{-1} \boldsymbol{\nabla} f(\boldsymbol{\theta}_i) \tag{12.32}$$

again for a small positive α. (Note the negative sign, which is present for both maximization and minimization.) There are variations on this approach, such as the Gauss–Newton or Levenberg–Marquardt algorithms.

All numerical optimization methods face the problem of when to declare convergence. The *exact* maximum is unattainable, and anyway not necessary. But we need to get close enough. Common convergence criteria are that the absolute or relative value of the change in f compared to the previous iteration is below some pre-defined value.

There exist more advanced optimization methods, such as BFGS, genetic algorithms, simulated annealing, and particle swarm. The methods vary according to what information is available (e.g. gradients), whether constraints are involved, how the data are represented, etc. Even if gradients are not available in an analytic form it may be possible to calculate them numerically. The primary goal of these methods is always the same, however: how to find the highest (local) optimum with the fewest function evaluations.

12.7 Bootstrap resampling

The very fact that we use a numerical method to find the maximum of the posterior often implies that we don't have a simple expression for the width of the posterior, and so don't have a measure of the precision of estimate of the maximum.

A conceptually simple way to determine the accuracy of an estimator is to recalculate it using different subsets of the data, and then to calculate the standard deviation or a confidence interval over the set of estimators. Such an approach is purely empirical, in the sense that it does not assume a model for the distribution of the data. It is therefore rather general. Suppose we have a set of N measurements $\{x_i\}$. We draw N points at random from this set *with replacement*. This sample of data is called the *bootstrap sample*. We then calculate our estimator from this sample; call it $\hat{x}$. It could be something straightforward, like the mean or mode, or it could be a more complicated function. We then repeat this K times, to give K bootstrap samples and a corresponding set of K estimators $\{\hat{x}_k\}$. The standard deviation of this set is a measure of the uncertainty of our estimator, with the estimator itself being obtained from the original set of data.

In this process our data $\{x_i\}$ are essentially acting as a discrete distribution that approximates the true, unknown distribution of x (assuming that the N measurements are independent). This is why we make the draws with replacement: we don't change a distribution just because we've drawn from it. Any technique that involves estimating a quantity by sampling with replacement is called *bootstrapping*, although it is usually used to obtain an empirical estimate of the uncertainty in an estimator.

One application could be to find the uncertainty on the result of numerically optimizing a posterior. We repeat the optimization K times, each time using a different bootstrap sample of the data to estimate the posterior. The standard deviation of these maxima is a bootstrap estimate.

To illustrate this, let us return to the problem introduced in section 3.5, of estimating the distance to a star from its parallax. Suppose that instead of a single parallax measurement

we now have N independent parallax measurements of the same star, the set $D = \{\varpi_i\}$. Let us again adopt a Gaussian noise model with a common standard deviation σ_ϖ for all measurements. The likelihood of each measurement is given by equation 3.20, and the likelihood of the complete data set is the product of such terms. I adopt the prior in equation 3.31. The unnormalized posterior is

$$P^*(r\,|\,D) \;=\; r^2 \exp(-r/L) \prod_{i=1}^{N} \exp\left[-\frac{1}{2\sigma_\varpi^2}\left(\varpi_i - \frac{1}{r}\right)^2\right] \tag{12.33}$$

$$\ln P^*(r\,|\,D) \;=\; 2\ln r - \frac{r}{L} - \frac{1}{2\sigma_\varpi^2}\sum_{i=1}^{N}\left(\varpi_i - \frac{1}{r}\right)^2 \tag{12.34}$$

where the neglected normalization constant is a function of D, σ_ϖ, and L, but not of r. We will take the maximum of the posterior as our estimator of r (for which the normalization constant is not required). It's easier to find this by maximizing $\ln P^*$. Differentiating equation 12.34 gives

$$\frac{d\ln P^*}{dr} \;=\; \frac{2}{r} - \frac{1}{L} - \frac{N}{\sigma_\varpi^2 r^2}\left(\langle\varpi\rangle - \frac{1}{r}\right) \tag{12.35}$$

where $\langle\varpi\rangle = (1/N)\sum_i \varpi_i$ is the mean of the parallax measurements. We could solve this algebraically by setting the gradient to zero, because multiplying by r^3 gives us a cubic equation.[4] But for illustration purposes I will use the numerical gradient ascent method mentioned in the previous section.

The R code at the end of this section performs the bootstrapping for a set of $N = 10$ measurements. As before, parallaxes are in arcseconds and distances are in parsecs. The script first defines simple functions for returning the prior, likelihood, unnormalized posterior, and gradient of the logarithm of the unnormalized posterior. The function `post.grid` evaluates the posterior on a regular grid, scaling it so the maximum is unity. This is only used for plotting. `draw.sample` draws a bootstrap sample of size N from the data. The `gradient.ascent` function is generic for any one-dimensional function. A suitable value of the α parameter depends on both the scale of the gradient and the size of sensible steps to make in r. In this simple case it is best found by trial and error. It turns out that a value of order of 10^3 gives good convergence. Convergence is defined as having been achieved when the fractional change in gradient from the previous iteration is less than `rel.tol`. Its value is therefore scale independent. The maximum number of iterations is set to be so large that the convergence criterion is almost always reached first. Having defined these functions, the script goes on to define some data, set the parameters, and then to find numerically the maximum of the posterior for each of a large number K (`Kboot` in the code) of bootstrap samples. I set the initial value of the parameter in the optimization procedure to the inverse of the median of the bootstrapped data sample, which is a reasonable guess at the star's distance.

The left panel of figure 12.11 shows the unnormalized posterior (equation 12.33) using all of the data as well as posteriors for seven example bootstrap samples. For each of the

[4] Equation 12.35 is the same as equation 19 in Bailer-Jones (2015) – which is for the single measurement case – when replacing ϖ with $\langle\varpi\rangle$ and σ_ϖ^2 with σ_ϖ^2/N in the latter equation.

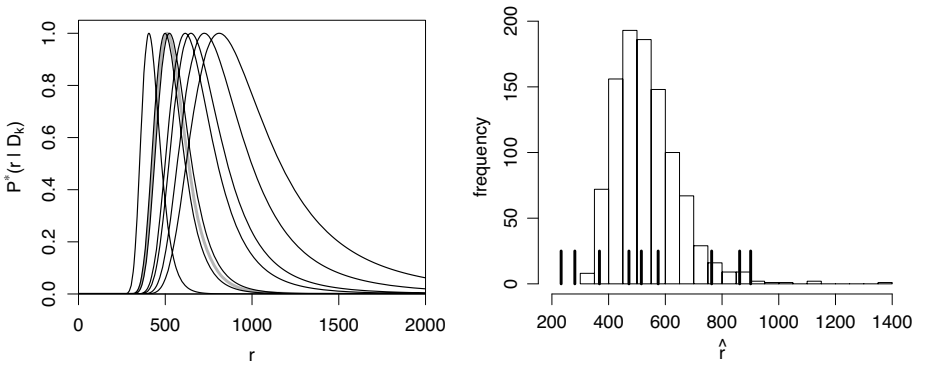

Fig. 12.11 Use of bootstrapping to estimate the standard deviation of the maximum of a distribution, in this case $P^*(r\,|\,D)$, the (unnormalized) posterior PDF of equation 12.33. Left: the thick grey line is the posterior calculated using all the data D. The black lines are posteriors calculated using seven example bootstrap samples of the data D_k ($k = 1 \ldots 7$). All distributions are scaled to have their maximum at one. Right: histogram of the maxima of the posterior for $K = 1000$ different bootstrap samples. For comparison the data D (plotted as $1/\varpi_i$) are shown with thick black vertical lines (the tenth point at $1/\varpi_i = 2780$ is not shown).

$k = 1 \ldots K$ bootstrap samples, the code locates the maximum of the posterior $\hat{r}_k$ by gradient ascent. The initial and final parameter values as well as the number of iterations done are written to the screen, so you can see whether convergence was reached in each case. The set of all K such maxima is $\{\hat{r}_k\}$ (here $K = 1000$). A histogram of $\{\hat{r}_k\}$ is shown in the right panel of figure 12.11. The standard deviation σ of this set is the bootstrap estimate of the standard deviation in our distance estimate. We find $\sigma = 113.4$.

Our distance estimate is the mode[5] of the posterior for the original data set, which is 511.3. The quantity σ is a measure of how accurately we can determine r from the data. It is *not* a measure of the "standard error in the mean" of $\{\hat{r}_k\}$ (see section 2.4). Increasing K will increase the precision with which we can estimate σ. That is, the variance in σ will reduce as $1/K$ in accordance with the central limit theorem (section 2.3), something we can verify by bootstrapping too. But increasing K will not continuously reduce the value of σ itself, because the scale of σ is determined by the ten data points we have. The bootstrap resampling quantifies how much the posterior mode varies on account of us having a finite amount of noisy data.

The quantity σ is also not a measure of the standard deviation of the posterior computed on the complete data set, even though in this particular case they have a similar size.[6] That they are not conceptually the same should be clear from the fact that our σ is related to

[5] This was found by gradient ascent and is the same as the analytic solution to equation 12.35 to the first decimal place.

[6] It is estimated numerically at the end of the R code below to be 115.6. This requires that we integrate the posterior. I do this by computing it on a dense grid, and then use equations 5.7 and 5.8 to calculate the variance.

the mode. We could instead have chosen to make bootstrap estimates of the variance of the mean, or of the 95% quantile, or of something else.

It is instructive to repeat this experiment with a different sized data set. Selecting just the first three data points of `data` and re-running the code, gradient ascent estimates the mode of the posterior to be 1567. This is far from the previous value due to the small amount of noisy data. The standard deviation of the bootstrap sample is now $\sigma = 276$. For comparison, the standard deviation of the posterior (computed from these three data points) is 370. As bootstrapping is done with replacement, the number of unique bootstrap data samples is given by equation 1.78 with $n = r = N$. For $N = 3$ the number of unique bootstrap samples is just ten. As the numerical optimization is deterministic and the starting point is always the same for a given bootstrap sample, there are likewise now just ten unique values in $\{\hat{r}_k\}$ (which you can verify with `unique(rMode)`). It doesn't make much sense to use bootstrap on such small samples. For $N = 10$ the number of unique samples rises to 92 378.

When we have a functional form for the posterior PDF – as in this example – it would be preferable to compute its standard deviation or a confidence interval (section 5.5) and use this a measure of the uncertainty our distance estimate, rather than using the bootstrap. I only use the bootstrap here to illustrate it on a problem for which we can verify the solution by a more direct method. Bootstrap is most useful when we only have a set of data, and do not have the functional form of the distribution the samples were drawn from. For example, if we only had the ten parallax measurements, and didn't know the likelihood or prior, we could estimate their mean and then estimate the uncertainty in this using the standard deviation of a set of bootstrap samples. This is done by the following, whereby I simply use the inverse parallax as the distance estimator.

```
rMean <- numeric(Kboot)
for(k in 1:Kboot) {
  rMean[k] <- 1/mean(draw.sample(data))
}
cat(1/mean(data), "+/-", sd(rMean), "\n")
```

This gives 492 ± 97. The mode and its bootstrapped standard deviation computed above was 511 ± 113.

R file: `bootstrap.R`

```
##### Demonstration of bootstrapping and gradient ascent

### Define functions

# Prior (normalized)
d.prior3 <- function(r, rlen) ifelse(r>0, (1/(2*rlen^3))*r^2*exp(-r/rlen), 0)

# Likelihood of one data point
d.like <- function(w, r, wsd) dnorm(x=w, mean=1/r, sd=wsd)

# Posterior (unnormalized) of set of data
d.post <- function(data, r, wsd, rlen) {
  prod(d.like(w=data, r, wsd))*d.prior3(r, rlen)
}
```

```
# Gradient of natural logarithm of d.post w.r.t r
grad.log.d.post <- function(r, d.post, data, wsd, rlen) {
  N <- length(data)
  dlnPdr <- 2/r - 1/rlen - (mean(data) - 1/r)*N/(wsd^2*r^2)
  return(dlnPdr)
  #return(dlnPdr * d.post(data, r, wsd, rlen))
}

# Calculate posterior on dense grid, scaled so mode=1.
# Return dataframe of r, posterior
post.grid <- function(data, wsd, rlen) {
  r <- seq(from=0, to=2*rlen, length.out=1e4)
  post <- numeric(length(r))
  for(i in 1:length(r)) {
    post[i] <- d.post(data=data, r=r[i], wsd, rlen)
  }
  return(data.frame(r=r, post=post/max(post)))
}

# Draw bootstrap sample
draw.sample <- function(data) sample(x=data, size=length(data),
                                     replace=TRUE)

# Gradient ascent (generic)
# Returns two element list:
# - par is optimized value of parameter
# - lastIt is iteration number reached
# gradFunc is a function which returns the gradient of the function to
# maximize. Its first argument must be the (scalar) value of the parameter.
# param is the intitial value of the parameter.
# '...' is used to pass anything else gradFunc needs (data etc).
# Search until relative change in gradient in less than rel.tol
# or maxIts iterations is reached.
grad.ascent <- function(gradFunc, param, ...) {
  rel.tol <- 1e-5
  alpha   <- 1e3
  maxIts  <- 1e4
  for(i in 1:maxIts) {
    paramNext <- param + alpha*gradFunc(param, ...)
    #cat(i, param, paramNext, "\n")
    if(abs(paramNext/param - 1) < rel.tol) {
      break
    } else {
      param <- paramNext
    }
  }
  return(list(par=paramNext, lastIt=i))
}

### Apply method

# Data (parallaxes in arcseconds; so distances are in parsecs)
wsd  <- 1.0e-3
data <- 1e-3*c(0.36, 1.11, 1.16, 1.31, 1.74, 1.94, 2.12, 2.72, 3.56, 4.30)
#data <- rnorm(n=10, mean=1/500, sd=wsd) # Above data were drawn like this
```

```
# Set up parameters
rlen  <- 1000 # prior length scale
Kboot <- 1000 # no. bootstrap samples

# Plot some example posteriors
pdf("bootstrap_posteriors.pdf", width=5, height=4)
par(mfrow=c(1,1), mar=c(3.5,4.2,1,1.25), oma=0.1*c(1,1,1,1),
    mgp=c(2.3,0.9,0), cex=1.15)
z <- post.grid(data, wsd, rlen)
plot(z$r, z$post, type="l", lwd=3, col="grey70",
     xlab="r", ylab=expression(paste(P^symbol("*"), "(r | ", D[k], ")")),
     ylim=c(0,1.05), xaxs="i", yaxs="i")
set.seed(555)
for(j in 1:7) {
  dataSample <- draw.sample(data)
  z <- post.grid(dataSample, wsd, rlen)
  lines(z$r, z$post, type="l", lwd=1.2)
}
dev.off()

# Draw bootstrap samples and for each calculate the maximum of the
# log posterior using gradient ascent
rMode <- numeric(Kboot)
for(k in 1:Kboot) {
  dataSample <- draw.sample(data)
  rInit <- 1/median(dataSample) # initial distance for optimization
  opt <- grad.ascent(gradFunc=grad.log.d.post, param=rInit, d.post,
                     dataSample, wsd, rlen)
  cat(rInit, opt$par, opt$lastIt, "\n") # print initial, final, no. its
  rMode[k] <- opt$par
}
cat("sd of bootstrap samples =", sd(rMode), "\n")
pdf("bootstrap_histogram.pdf", 5,4)
par(mfrow=c(1,1), mar=c(3.5,3.5,1,1), oma=0.1*c(1,1,1,1), mgp=c(2.2,0.8,0),
    cex=1.15)
hist(rMode, breaks=25, xlim=c(200, 1400), main="",
     xlab=expression(hat(r)), ylab="frequency")
segments(x0=1/data, y0=0, x1=1/data, y1=25, lwd=3)
dev.off()

# Estimate distance to the star using the original data set
opt <- grad.ascent(gradFunc=grad.log.d.post, param=rInit, d.post, data,
                   wsd, rlen)
cat("Numerical estimate of mode =", opt$par, "\n")

# For comparison, compute sd of posterior from all data (need to normalize
# it first). r must extend to where post is very small
r <- seq(from=0, to=2*rlen, by=0.1)
post <- numeric(length(r))
for(i in 1:length(r)) {
  post[i] <- d.post(data=data, r=r[i], wsd, rlen)
}
rMean <- sum(post*r)/sum(post)
plot(r, post, type="l") # ensure we have sampled the full posterior
cat("sd of posterior =", sqrt(sum(post*(r-rMean)^2)/sum(post)), "\n")
```