7 Dimensionality and Its Reduction

"A mind that is stretched by a new idea can never go back to its original dimensions." (Oliver Wendell Holmes)

With the dramatic increase in data available from a new generation of astronomical telescopes and instruments, many analyses must address the question of the complexity as well as size of the data set. For example, with the SDSS imaging data we could measure arbitrary numbers of properties or features for any source detected on an image (e.g., we could measure a series of progressively higher moments of the distribution of fluxes in the pixels that make up the source). From the perspective of efficiency we would clearly rather measure only those properties that are directly correlated with the science we want to achieve. In reality we do not know the correct measurement to use or even the optimal set of functions or bases from which to construct these measurements. This chapter deals with how we can learn which measurements, properties, or combinations thereof carry the most information within a data set. The techniques we will describe here are related to concepts we have discussed when describing Gaussian distributions (§3.5.2), density estimation (§6.1), and the concepts of information content (§5.2.2).

We will start in §7.1 with an exploration of the problems posed by high-dimensional data. In §7.2 we will describe the data sets used in this chapter, and in §7.3, we will introduce perhaps the most important and widely used dimensionality reduction technique, principal component analysis (PCA). In the remainder of the chapter, we will introduce several alternative techniques which address some of the weaknesses of PCA.

7.1. The Curse of Dimensionality

Imagine that you have decided to purchase a car and that your initial thought is to purchase a "fast" car. Whatever the exact implementation of your search strategy, let us assume that your selection results in a fraction $r < 1$ of potential matches. If you expand the requirements for your perfect car such that it is "red," has "8 cylinders," and a "leather interior" (with similar selection probabilities), then your selection probability would be r^4. If you also throw "classic car" into the mix, the selection probability becomes r^5. The more selection conditions you adopt, the tinier is the

chance of finding your ideal car! These selection conditions are akin to dimensions in your data set, and this effect is at the core of the phenomenon known as the "curse of dimensionality"; see [2].

From a data analysis point of view, the curse of dimensionality impacts the size of data required to constrain a model, the complexity of the model itself, and the search times required to optimize the model. Considering just the first of these issues, we can quantitatively address the question of how much data is needed to estimate the probability of finding points within a high-dimensional space. Imagine that you have drawn N points from a D-dimensional uniform distribution, described by a hypercube (centered at the origin) with edge length 2 (i.e., each coordinate lies within the range $[-1, 1]$). Under a Euclidean distance metric, what proportion of points fall within a unit distance of the origin?

If the points truly are uniformly distributed throughout the volume, then a good estimate of this is the ratio of the volume of a unit hypersphere centered at the origin, to the volume of the side-length=2 hypercube centered at the origin. For two dimensions, this gives

$$f_2 = \frac{\pi r^2}{(2r)^2} = \pi/4 \approx 78.5\%. \tag{7.1}$$

For three dimensions, this becomes

$$f_3 = \frac{(4/3)\pi r^3}{(2r)^3} = \pi/6 \approx 52.3\%. \tag{7.2}$$

Generalizing to higher dimensions requires some less familiar formulas for the hypervolumes of D-spheres. It can be shown that the hypervolume of a D-dimensional hypersphere with radius r is given by

$$V_D(r) = \frac{2r^D \pi^{D/2}}{D\,\Gamma(D/2)}, \tag{7.3}$$

where $\Gamma(z)$ is the complete gamma function. The reader can check that evaluating this for $D = 2$ and $D = 3$ yield the familiar formulas for the area of a circle and the volume of a sphere. With this formula we can compute

$$f_D = \frac{V_D(r)}{(2r)^D} = \frac{\pi^{D/2}}{D2^{D-1}\Gamma(D/2)}. \tag{7.4}$$

One can quite easily show that

$$\lim_{D\to\infty} f_D = 0. \tag{7.5}$$

That is, in the limit of many dimensions, *not a single point* will be within a unit radius of the origin! In other words, the fraction of points within a search radius r (relative to the full space) will tend to zero as the dimensionality grows and, therefore, the number of points in a data set required to evenly sample this hypervolume will grow exponentially with dimension. In the context of astronomy, the SDSS [1] comprises

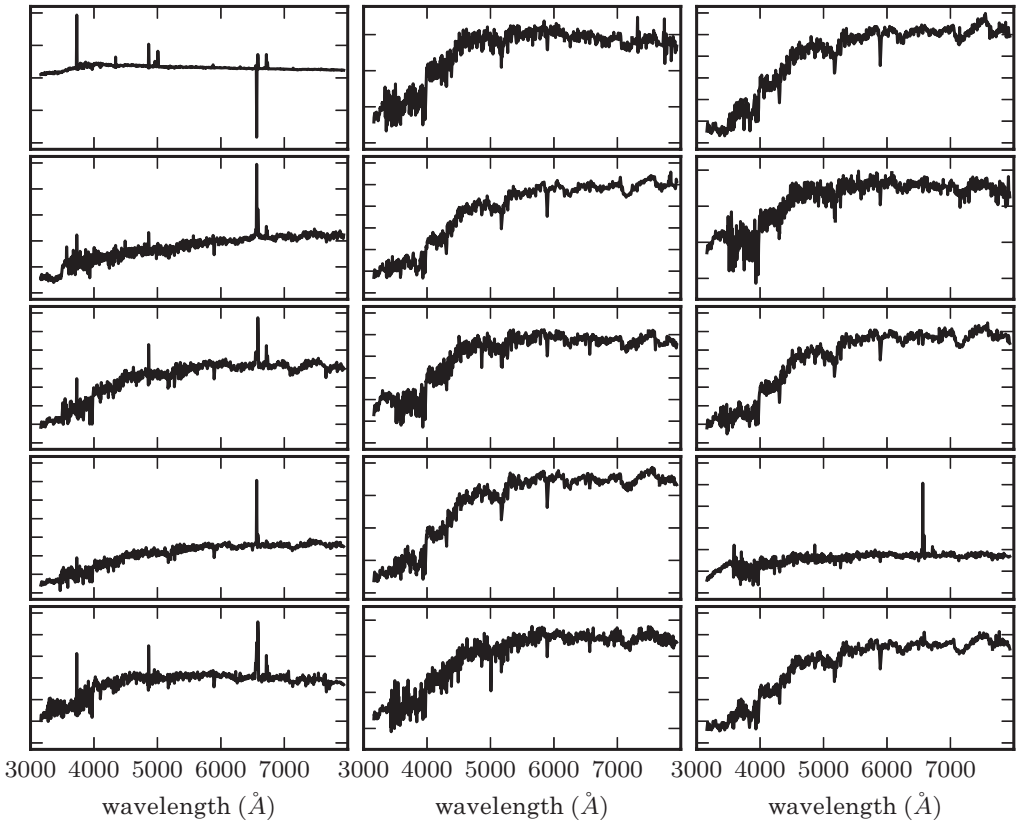

Figure 7.1. A sample of 15 galaxy spectra selected from the SDSS spectroscopic data set (see §1.5.5). These spectra span a range of galaxy types, from star-forming to passive galaxies. Each spectrum has been shifted to its rest frame and covers the wavelength interval 3000–8000 Å. The specific fluxes, $F_\lambda(\lambda)$, on the ordinate axes have an arbitrary scaling.

a sample of 357 million sources. Each source has 448 measured attributes (e.g., measures of flux, size, shape, and position). If we used our physical intuition to select just 30 of those attributes from the database (e.g., a subset of the magnitude, size, and ellipticity measures) and normalized the data such that each dimension spanned the range -1 to 1, the probability of having one of the 357 million sources reside within the unit hypersphere would be only one part in 1.4×10^5.

Given the dimensionality of current astronomical data sets, how can we ever hope to find or characterize any structure that might be present? The underlying assumption behind our earlier discussions has been that all dimensions or attributes are created equal. We know that is, however, not true. There exist projections within the data that capture the principal physical and statistical correlations between measured quantities (this idea lies behind the *intrinsic dimensionality* discussed in §2.5.2). Finding these dimensions or axes efficiently and thereby reducing the dimensionality of the underlying data is the subject of this chapter.

7.2. The Data Sets Used in This Chapter

Throughout this chapter we use the SDSS galaxy spectra described in §1.5.4 and §1.5.5, as a proxy for high-dimensional data. Figure 7.1 shows a representative sample

of these spectra covering the interval 3200–7800 Å in 1000 wavelength bins. While a spectrum defined as $x(\lambda)$ may not immediately be seen as a point in a high-dimensional space, it can be represented as such. The function $x(\lambda)$ is in practice sampled at D discrete flux values, and written as a D-dimensional vector. And just as a three-dimensional vector is often visualized as a point in a three-dimensional space, this spectrum (represented by a D-dimensional vector) can be thought of as a single point in D-dimensional space. Analogously, a $D = N \times K$ image may also be expressed as a vector with D elements, and therefore a point in a D-dimensional space. So, while we use spectra as our proxy for high-dimensional space, the algorithms and techniques described in this chapter are applicable data as diverse as catalogs of multivariate data, two-dimensional images, and spectral hypercubes.

7.3. Principal Component Analysis

Figure 7.2 shows a two-dimensional distribution of points drawn from a Gaussian centered on the origin of the x- and y-axes. While the points are strongly correlated along a particular direction, it is clear that this correlation does not align with the initial choice of axes. If we wish to reduce the number of features (i.e., the number of axes) that are used to describe these data (providing a more compact representation) then it is clear that we should rotate our axes to align with this correlation (we have already encountered this rotation in eq. 3.82). Any rotation preserves the relative ordering or configuration of the data so we choose our rotation to maximize the ability to discriminate between the data points. This is accomplished if the rotation maximizes the variance along the resulting axes (i.e., defining the first axis, or principal component, to be the direction with maximal variance, the second principal component to be orthogonal to the first component that maximizes the residual variance, and so on). As indicated in figure 7.2, this is mathematically equivalent to a regression that minimizes the square of the orthogonal distances from the points to the principal axes.

This dimensionality reduction technique is known as a principal component analysis (PCA). It is also referred to in the literature as a Karhunen–Loéve [21, 25] or Hotelling transform. PCA is a linear transform, applied to multivariate data, that defines a set of uncorrelated axes (the principal components) ordered by the variance captured by each new axis. It is one of the most widely applied dimensionality reduction techniques used in astrophysics today and dates back to Pearson who, in 1901, developed a procedure for fitting lines and planes to multivariate data; see [28].

There exist a number of excellent texts on PCA that review its use across a broad range of fields and applications (e.g., [19] and references therein). We will, therefore, focus our discussion of PCA on a brief description of its mathematical formalism then concentrate on its application to astronomical data and its use as a tool for classification, data compression, regression, and signal-to-noise filtering of high-dimensional data sets.

Before progressing further with the application of PCA, it is worth noting that many of the applications of PCA to astronomical data describe the importance of the orthogonal nature of PCA (i.e., the ability to project a data set onto a set of uncorrelated axes). It is often forgotten that the observations themselves are already a

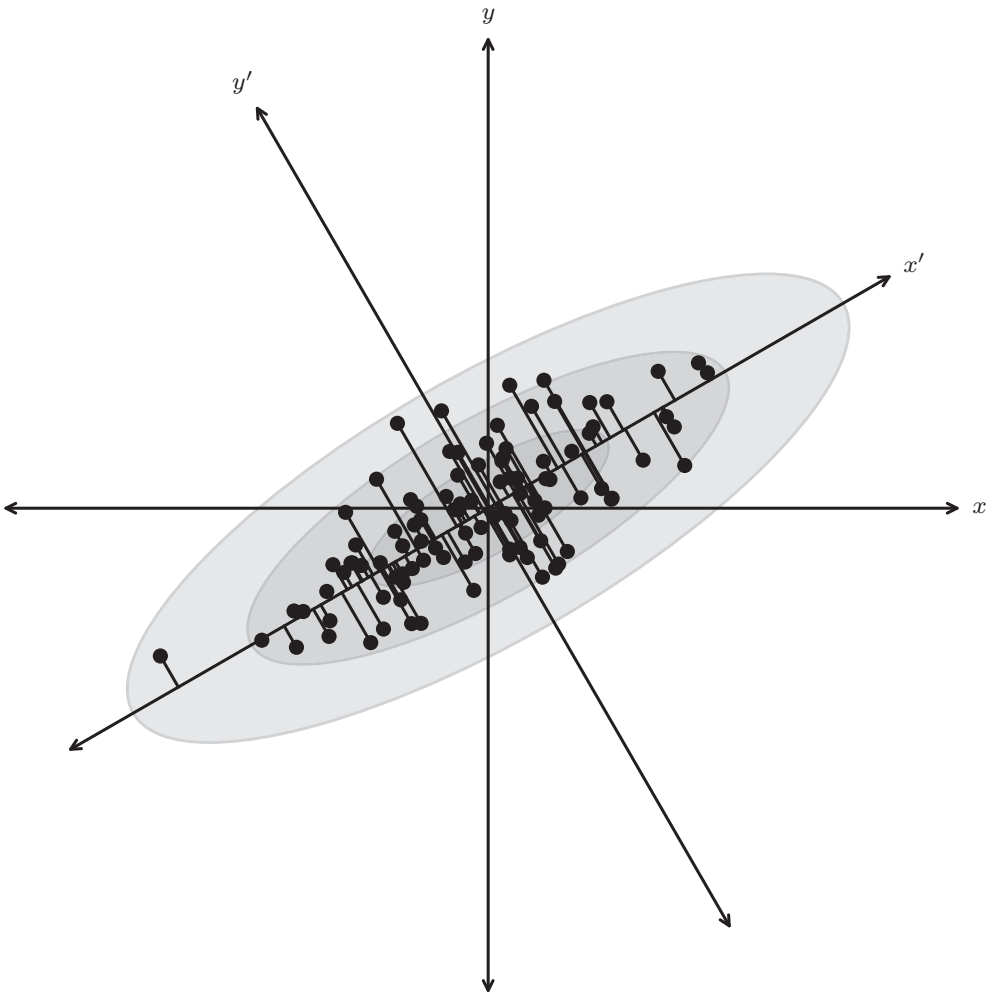

Figure 7.2. A distribution of points drawn from a bivariate Gaussian and centered on the origin of x and y. PCA defines a rotation such that the new axes (x' and y') are aligned along the directions of maximal variance (the principal components) with zero covariance. This is equivalent to minimizing the square of the perpendicular distances between the points and the principal components.

representation of an orthogonal basis (e.g., the axes {1,0,0,0, ...}, {0,1,0,0,0, ...}, etc.). As we will show, the importance of PCA is that *the new axes are aligned with the direction of maximum variance within the data* (i.e., the direction with the maximum signal).

7.3.1. The Derivation of Principal Component Analyses

Consider a set of data, $\{x_i\}$, comprising a series of N observations with each observation made up of K measured features (e.g., size, color, and luminosity, or the wavelength bins in a spectrum). We initially center the data by subtracting the mean of each feature in $\{x_i\}$ and then write this $N \times K$ matrix as X.[1] The covariance

[1]Often the opposite convention is used: that is, N points in K dimensions are stored in a $K \times N$ matrix rather than an $N \times K$ matrix. We choose the latter to align with the convention used in Scikit-learn and AstroML.

of the centered data, C_X, is given by

$$C_X = \frac{1}{N-1} X^T X, \tag{7.6}$$

where the $N - 1$ term comes from the fact that we are working with the sample covariance matrix (i.e., the covariances are derived from the data themselves).

Nonzero off-diagonal components within the covariance matrix arise because there exist correlations between the measured features (as we saw in figure 7.2; recall also the discussion of bivariate and multivariate distributions in §3.5). PCA wishes to identify a projection of $\{x_i\}$, say, R, that is aligned with the directions of maximal variance. We write this projection as $Y = XR$ and its covariance as

$$C_Y = R^T X^T X R = R^T C_X R \tag{7.7}$$

with C_X the covariance of X as defined above.

The first principal component, r_1, of R is defined as the projection with the maximal variance (subject to the constraint that $r_1^T r_1 = 1$). We can derive this principal component by using Lagrange multipliers and defining the cost function, $\phi(r_1, \lambda)$, as

$$\phi(r_1, \lambda_1) = r_1^T C_X r_1 - \lambda_1 (r_1^T r_1 - 1). \tag{7.8}$$

Setting the derivative of $\phi(r_1, \lambda)$ (with respect to r_1) to zero gives

$$C_X r_1 - \lambda_1 r_1 = 0. \tag{7.9}$$

λ_1 is, therefore, the root of the equation $\det(C_X - \lambda_1 \mathbf{I}) = 0$ and is an eigenvalue of the covariance matrix. The variance for the first principal component is maximized when

$$\lambda_1 = r_1^T C_X r_1 \tag{7.10}$$

is the largest eigenvalue of the covariance matrix. The second (and further) principal components can be derived in an analogous manner by applying the additional constraint to the cost function that the principal components are uncorrelated (e.g., $r_2^T C_X r_1 = 0$).

The columns of R are then the eigenvectors or principal components, and the diagonal values of C_Y define the amount of variance contained within each component. With

$$C_X = R C_Y R^T \tag{7.11}$$

and ordering the eigenvectors by their eigenvalue we can define the set principal components for X.

Efficient computation of principal components

One of the most direct methods for computing the PCA is through the eigenvalue decomposition of the covariance or correlation matrix, or equivalently through the

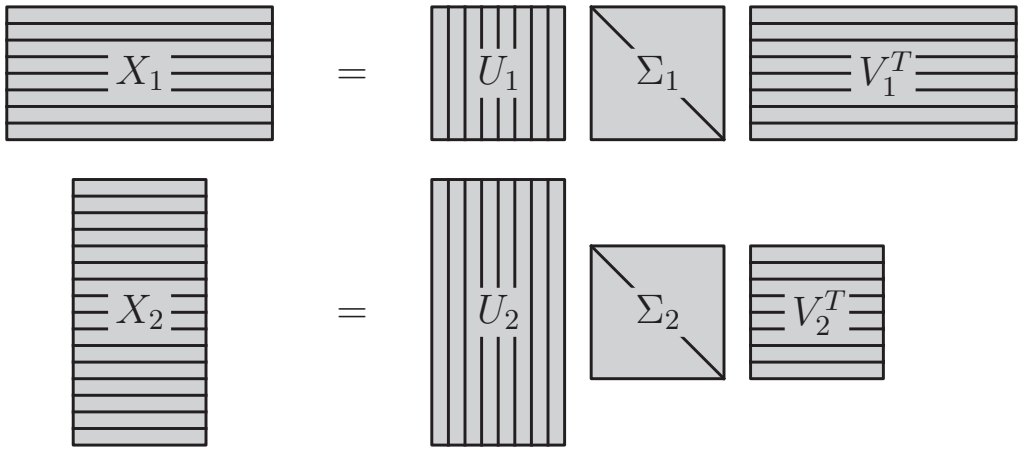

Figure 7.3. Singular value decomposition (SVD) can factorize an $N \times K$ matrix into $U\Sigma V^T$. There are different conventions for computing the SVD in the literature, and this figure illustrates the convention used in this text. The matrix of singular values Σ is always a square matrix of size $[R \times R]$ where $R = \min(N, K)$. The shape of the resulting U and V matrices depends on whether N or K is larger. The columns of the matrix U are called the left-singular vectors, and the columns of the matrix V are called the right-singular vectors. The columns are orthonormal bases, and satisfy $U^T U = V^T V = I$.

singular value decomposition (SVD) of the data matrix itself. The scaled SVD can be written

$$U\Sigma V^T = \frac{1}{\sqrt{N-1}} X, \tag{7.12}$$

where the columns of U are the *left-singular vectors*, and the columns of V are the *right-singular vectors*. There are many different conventions for the SVD in the literature; we will assume the convention that the matrix of singular values Σ is always a square, diagonal matrix, of shape $[R \times R]$ where $R = \min(N, K)$ is the rank of the matrix X (assuming all rows and columns of X are independent). U is then an $[N \times R]$ matrix, and V^T is an $[R \times K]$ matrix (see figure 7.3 for a visualization of this SVD convention). The columns of U and V form orthonormal bases, such that $U^T U = V^T V = I$.

Using the expression for the covariance matrix (eq. 7.6) along with the scaled SVD (eq. 7.12) gives

$$\begin{aligned} C_X &= \left[\frac{1}{\sqrt{N-1}} X\right]^T \left[\frac{1}{\sqrt{N-1}} X\right] \\ &= V\Sigma U^T U\Sigma V^T \\ &= V\Sigma^2 V^T. \end{aligned} \tag{7.13}$$

Comparing to eq 7.11, we see that the right singular vectors V correspond to the principal components R, and the diagonal matrix of eigenvalues C_Y is equivalent to the square of the singular values,

$$\Sigma^2 = C_Y. \tag{7.14}$$

Thus the eigenvalue decomposition of C_X, and therefore the principal components, can be computed from the SVD of X, without explicitly constructing the matrix C_X.

NumPy and SciPy contain powerful suites of linear algebra tools. For example, we can confirm the above relationship using `svd` for computing the SVD, and `eigh` for computing the symmetric (or in general Hermitian) eigenvalue decomposition:

```
>>> import numpy as np
>>> X = np.random.random((100, 3))
>>> CX = np.dot(X.T, X)
>>> U, Sdiag, VT = np.linalg.svd(X, full_matrices=
    False)
>>> CYdiag, R = np.linalg.eigh(CX)
```

The `full_matrices` keyword assures that the convention shown in figure 7.3 is used, and for both Σ and C_Y, only the diagonal elements are returned. We can compare the results, being careful of the different ordering conventions: `svd` puts the largest singular values first, while `eigh` puts the smallest eigenvalues first:

```
>>> np.allclose(CYdiag, Sdiag[::-1] ** 2)
 # [::-1] reverses the array
True
>>> np.set_printoptions(suppress=True)
 # clean output for below
>>> VT[::-1].T / R
array([[-1., -1.,  1.],
       [-1., -1.,  1.],
       [-1., -1.,  1.]])
```

The eigenvectors of C_X and the right singular vectors of X agree up to a sign, as expected. For more information, see appendix A or the documentation of `numpy.linalg` and `scipy.linalg`.

The SVD formalism can also be used to quickly see the relationship between the covariance matrix C_X, and the correlation matrix,

$$\begin{aligned} M_X &= \frac{1}{N-1} X X^T \\ &= U \Sigma V^T V \Sigma U^T \\ &= U \Sigma^2 U^T \end{aligned} \tag{7.15}$$

in analogy with above. The left singular vectors, U, turn out to be the eigenvectors of the correlation matrix, which has eigenvalues identical to those of the covariance matrix. Furthermore, the orthonormality of the matrices U and V means that if U is known, V (and therefore R) can be quickly determined using the linear algebraic

eigenvalues, σ_i, that are the diagonals of the matrix Σ:

$$\frac{\sum_i^{i=r} \sigma_i}{\sum_i^{i=R} \sigma_i} < \alpha. \tag{7.20}$$

Typical values for α range from 0.70 to 0.95, though the choice of threshold is sensitive to the shape of the scree plot (figure 7.5); for a shallow slope in the scree plot, whether r or $r + 1$ crosses the threshold is somewhat arbitrary.

The shape of the scree plot can be used to define the level of truncation. Cattell [6], using factor analysis, proposed that a sharp change in the gradient of the eigenvalues (i.e., a knee in the scree plot) could be used to define the cutoff value, r. The knee is defined by [6] as $\Sigma_r^2 - \Sigma_{r+1}^2$. If no clearly defined break in the scree plot exists, the definition of this cutoff becomes problematic. A modification of the technique in [6] is the LEV diagram, which plots the logarithm of the eigenvalue against the number of eigenvectors. The rationale for this modification is that if noise decays geometrically, variation in the eigenvalues should drop as a linear function.

For the correlation matrix PCA, Kaiser's rule [20] or the Guttman–Kaiser criterion can be applied. This states that, if all of the elements of x are independent then all principal components would have unit variance. In this case, r can be set to the limit where the eigenvalues in the scree plot fall below unity. In the context of the covariance matrix this can be reformulated as the setting of r to the number of components at which the eigenvalue falls to the average of all eigenvalues. Jolliffe [19] proposed a modification of this truncation to 70% of the average eigenvalues to allow for sample variance (which increases the number of components returned). Experiments with Kaiser's rule show that it tends to overpredict the number of components that remain after truncation.

Each of the criteria described above are sensitive to the shape of the scree plot and the choice of truncation rule remains application specific.

7.3.3. PCA with Missing Data

Until now we have assumed that the data we are working with are complete, without gaps or censored elements. In real-world applications, the presence of detector glitches, variable noise, or masking effects (e.g., sky lines in astronomical spectra) can make these assumptions invalid. Truncation of the expansion provides a signal-to-noise filtering of the data. The PCA bases should also be able to correct for missing elements within the data: because the PCA components encode the correlation of each flux with the other measured fluxes, these components should provide a natural way to determine these missing values.

One complication we must address is that eigenspectra are only defined to be orthogonal over the data range on which they are constructed. If a data vector does not fully cover that space then projecting the data onto the eigenbases will result in a biased set of expansion coefficients. Everson and Sirovich [12] have, however, shown that, when we know how the input data are masked or censored, we can correct for the nonorthogonality of the eigenbases. Following Connolly and Szalay [9], we consider an observed spectrum, $\boldsymbol{x}^o$, as the combination of the true spectrum (i.e., without gaps), $\boldsymbol{x}$, and a wavelength-dependent weight, $\boldsymbol{w}$. This weight is zero where

data are missing and $1/\boldsymbol{\sigma}^2$ for the remaining spectral range (with $\boldsymbol{\sigma}^2$ the variance of the spectral elements). Minimizing the quadratic deviation between the original spectrum, $\boldsymbol{x}^o$, and its truncated reconstruction, $\sum_i \theta_i \boldsymbol{e}_i$ and solving for θ_i gives

$$\sum_k \theta_i \boldsymbol{w}(k)\boldsymbol{e}_i(k)\boldsymbol{e}_j(k) = \sum_k \boldsymbol{w}(k)\boldsymbol{x}^o(k)\boldsymbol{e}_j(k), \tag{7.21}$$

where $\sum_k$ represents the sum over the length of the vector $\boldsymbol{x}(k)$ (i.e., over wavelength for the case of the spectra). If we define $M_{ij} = \sum_k \boldsymbol{w}(k)\boldsymbol{e}_i(k)\boldsymbol{e}_j(k)$ and $F_i = \sum_k \boldsymbol{w}(k)\boldsymbol{x}^o(k)\boldsymbol{e}_i(k)$ then this simplifies to

$$\theta_i = \sum_j M_{ij}^{-1} F_j, \tag{7.22}$$

where F_j represent the coefficients derived from the gappy data and M_{ij}^{-1} expresses how correlated the eigenvectors are over the missing regions.

Figure 7.7 shows the reconstruction of missing data using the PCA components. The gray regions represent intervals in wavelength space where the spectra are censored (the underlying data values within these censored regions are shown by the black line). The gray lines represent the reconstruction of these spectral regions using the eigenbases. An estimate of the uncertainty on the reconstruction coefficients is given by

$$\mathrm{Cov}(\theta_i, \theta_j) = M_{ij}^{-1}. \tag{7.23}$$

The accuracy of this reconstruction will depend on the distribution of the gaps within the data vector (though this is reflected in $\mathrm{Cov}(\theta_i, \theta_j)$). For an uncorrelated set of gaps, reconstruction with the number of sampled points, $N_{\mathrm{samples}} > r$, is possible. This observation is at the heart of the fields of lossy compression and compressed sensing.

7.3.4. Scaling to Large Data Sets

There are a number of limitations of PCA that can make it impractical for application to very large data sets. Principal to this are the computational and memory requirements of the SVD, which scale as $\mathcal{O}(D^3)$ and $\mathcal{O}(2 \times D \times D)$, respectively. In §7.3.1 we derived the PCA by applying an SVD to the covariance and correlation matrices of the data X. Thus, the computational requirements of the SVD are set by the rank of the data matrix, X, with the covariance matrix the preferred route if $K < N$ and the correlation matrix if $K > N$. Given the symmetric nature of both the covariance and correlation matrix, eigenvalue decompositions (EVD) are often more efficient than SVD approaches.

Even given these optimizations, with data sets exceeding the size of the memory available per core, applications of PCA can be very computationally challenging. This is particularly the case for real-world applications when the correction techniques for missing data are iterative in nature. One approach to address these limitations is to make use of online algorithms for an iterative calculation of the mean. As shown in

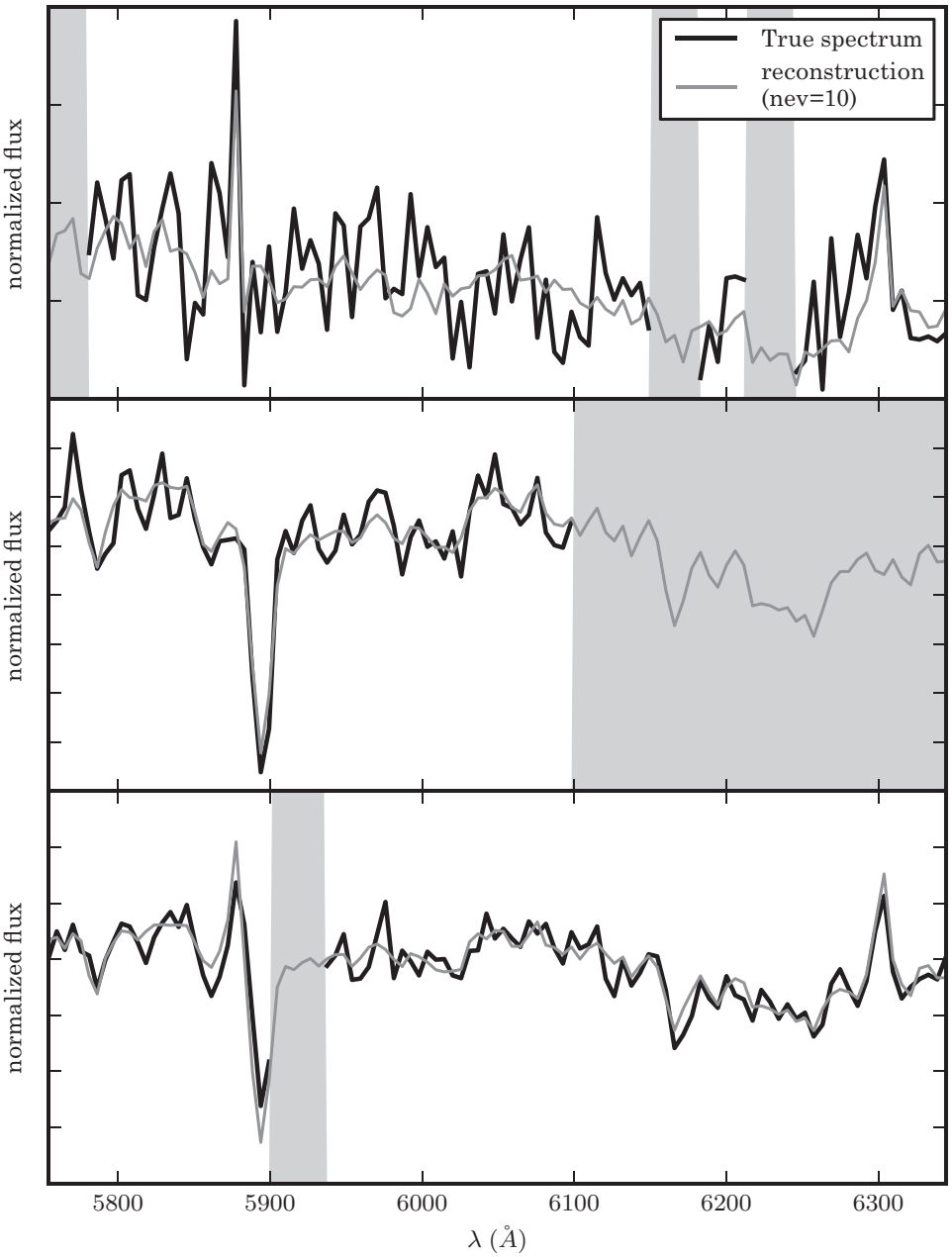

Figure 7.7. The principal component vectors defined for the SDSS spectra can be used to interpolate across or reconstruct missing data. Examples of three masked spectral regions are shown comparing the reconstruction of the input spectrum (black line) using the mean and the first ten eigenspectra (gray line) The gray bands represent the masked region of the spectrum.

[5], the sample covariance matrix can be defined as

$$C = \gamma C_{\text{prev}} + (1 - \gamma) x^T x \tag{7.24}$$

$$\sim \gamma Y_p D_p Y_p^T + (1 - \gamma) x^T x, \tag{7.25}$$

where C_{prev} is the covariance matrix derived from a previous iteration, x is the new observation, Y_p are the first p eigenvectors of the previous covariance matrix, D_p

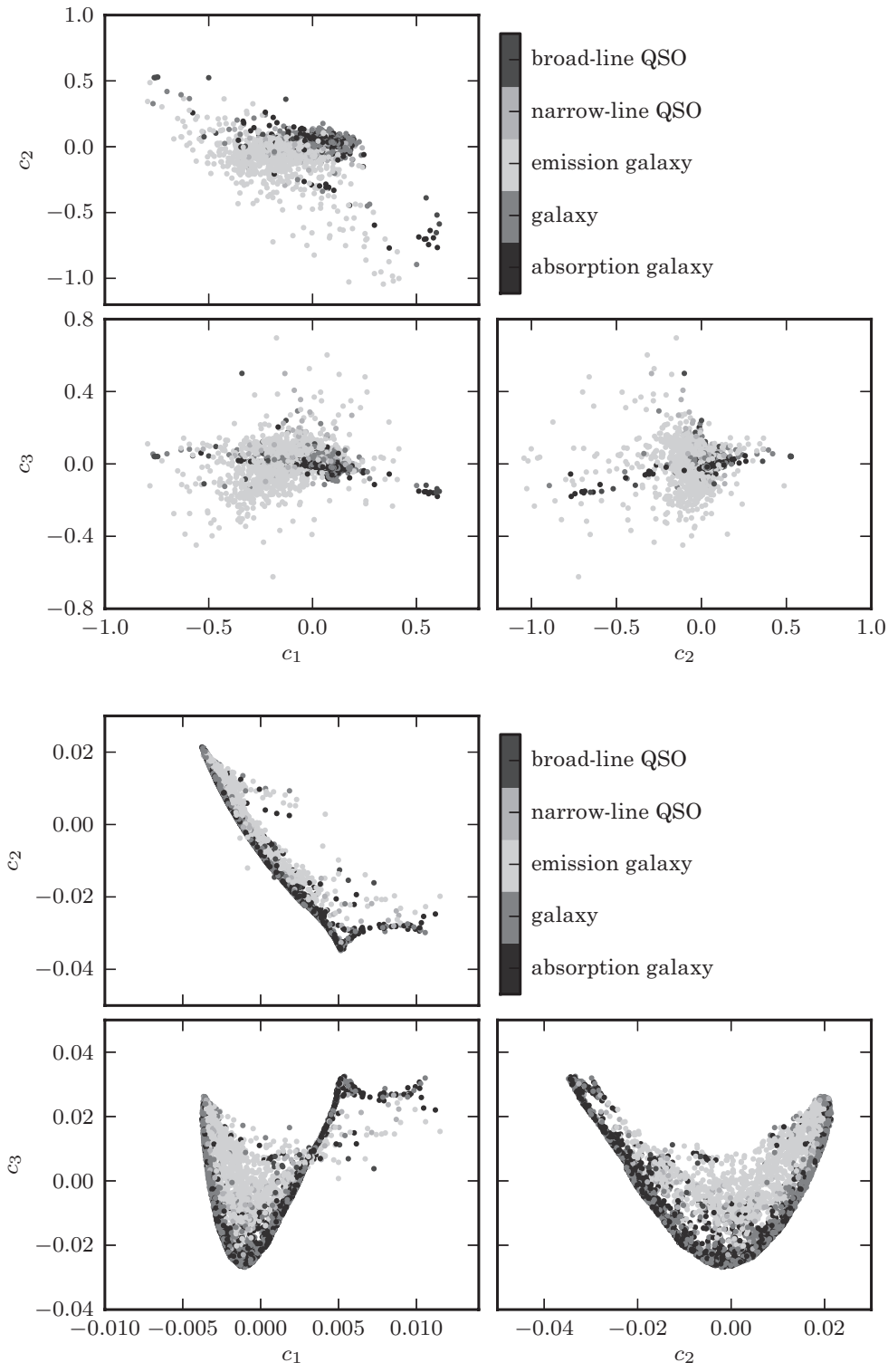

Figure 7.9. A comparison of the classification of quiescent galaxies and sources with strong line emission using LLE and PCA. The top panel shows the segregation of galaxy types as a function of the first three PCA components. The lower panel shows the segregation using the first three LLE dimensions. The preservation of locality in LLE enables nonlinear features within a spectrum (e.g., variation in the width of an emission line) to be captured with fewer components. This results in better segregation of spectral types with fewer dimensions. See color plate 6.

7.5.2. IsoMap

IsoMap [30], short for isometric mapping, is another manifold learning method which, interestingly, was introduced in the same issue of *Science* in 2000 as was LLE. IsoMap is based on a multidimensional scaling (MDS) framework. Classical MDS is a method to reconstruct a data set from a matrix of pairwise distances (for a detailed discussion of MDS see [4]).

If one has a data set represented by an $N \times K$ matrix X, then one can trivially compute an $N \times N$ distance matrix D_X such that $[D_X]_{ij}$ contains the distance between points i and j. Classical MDS seeks to reverse this operation: given a distance matrix D_X, MDS discovers a new data set Y which minimizes the error

$$\mathcal{E}_{XY} = |\tau(D_X) - \tau(D_Y)|^2, \tag{7.32}$$

where τ is an operator with a form chosen to simplify the analytic form of the solution. In metric MDS the operator τ is given by

$$\tau(D) = \frac{HSH}{2}, \tag{7.33}$$

where S is the matrix of square distances $S_{ij} = D_{ij}^2$, and H is the "centering matrix" $H_{ij} = \delta_{ij} - 1/N$. This choice of τ is convenient because it can then be shown that the optimal embedding Y is identical to the top D eigenvectors of the matrix $\tau(D_X)$ (for a derivation of this property see [26]).

The key insight of IsoMap is that we can use this metric MDS framework to derive a nonlinear embedding by constructing a suitable stand-in for the distance matrix D_X. IsoMap recovers nonlinear structure by approximating geodesic curves which lie within the embedded manifold, and computing the distances between each point in the data set along these geodesic curves. To accomplish this, the IsoMap algorithm creates a connected graph G representing the data, where G_{ij} is the distance between point i and point j if points i and j are neighbors, and $G_{ij} = 0$ otherwise. Next, the algorithm constructs a matrix D'_X such that $[D'_X]_{ij}$ contains the length of the shortest path between point i and j traversing the graph G. Using this distance matrix, the optimal d-dimensional embedding is found using the MDS algorithm discussed above.

IsoMap has a computational cost similar to that of LLE if clever algorithms are used. The first step (nearest-neighbor search) and final step (eigendecomposition of an $N \times N$ matrix) are similar to those of LLE. IsoMap has one additional hurdle however: the computation of the pairwise shortest paths on an order-N sparse graph G. A brute-force approach to this sort of problem is prohibitively expensive: for each point, one would have to test every combination of paths, leading to a total computation time of $\mathcal{O}(N^2 k^N)$. There are known algorithms which improve on this: the Floyd–Warshall algorithm [13] accomplishes this in $\mathcal{O}(N^3)$, while the Dijkstra algorithm using Fibonacci heaps [14] accomplishes this in $\mathcal{O}(N^2(k + \log N))$: a significant improvement over brute force.

Scikit-learn has a fast implementation of the IsoMap algorithm, using either the Floyd–Warshall algorithm or Dijkstra's algorithm for shortest-path search. The neighbor search is implemented with a fast tree search, and the final eigenanalysis is implemented using the Scikit-learn ARPACK wrapper. It can be used as follows:

```
import numpy as np
from sklearn.manifold import Isomap

X = np.random.normal(size=(1000, 2))
    # 1000 pts in 2 dims
R = np.random.random((2, 10))  # projection matrix
X = np.dot(X, R)  # X is now a 2D manifold in
    # 10D space
k = 5  # number of neighbors used in the fit
n = 2  # number of dimensions in the fit
iso = Isomap(k, n)
iso.fit(X)
proj = iso.transform(X)  # 1000 x 2 projection of
       # data
```

For more details, see the documentation of Scikit-learn or the source code of the IsoMap figures in this chapter.

7.5.3. Weaknesses of Manifold Learning

Manifold learning is a powerful tool to recover low-dimensional nonlinear projections of high-dimensional data. Nevertheless, there are a few weaknesses that prevent it from being used as widely as techniques like PCA:

Noisy and gappy data: Manifold learning techniques are in general not well suited to fitting data plagued by noise or gaps. To see why, imagine that a point in the data set shown in figure 7.8 is located at $(x, y) = (0, 0)$, but not well constrained in the z direction. In this case, there are three perfectly reasonable options for the missing z coordinate: the point could lie on the bottom of the "S", in the middle of the "S", or on the top of the "S". For this reason, manifold learning methods will be fundamentally limited in the case of missing data. One may imagine, however, an iterative approach which would construct a (perhaps multimodal) Bayesian constraint on the missing values. This would be an interesting direction for algorithmic research, but such a solution has not yet been demonstrated.

Tuning parameters: In general, the nonlinear projection obtained using these techniques depends highly on the set of nearest neighbors used for each point. One may select the k neighbors of each point, use all neighbors within a radius r of each point, or choose some more sophisticated technique. There is currently no solid recommendation in the literature for choosing the optimal set of neighbors for a given embedding: the optimal choice will depend highly on the local density of each point, as well as the curvature of the manifold at each point. Once again, one may

imagine an iterative approach to optimizing the selection of neighbors based on these insights. This also could be an interesting direction for research.

Dimensionality: One nice feature of PCA is that the dimensionality of the data set can be estimated, to some extent, from the eigenvalues associated with the projected dimensions. In manifold learning, there is no such clean mapping. In fact, we have no guarantee that the embedded manifold is unidimensional! One can easily imagine a situation where a data set is drawn from a two-dimensional manifold in one region, and from a three-dimensional manifold in an adjacent region. Thus in a manifold learning setting, the choice of output dimensionality is a free parameter. In practice, either $d = 1$, $d = 2$, or $d = 3$ is often chosen, for the simple reason that it leads to a projection which is easy to visualize! Some attempts have been made to use local variance of a data set to arrive at an estimate of the dimensionality (see, e.g., [11]) but this works only marginally well in practice.

Sensitivity to outliers: Another weakness of manifold learning is its sensitivity to outliers. In particular, even a single outlier between different regions of the manifold can act to "short-circuit" the manifold so that the algorithm cannot find the correct underlying embedding.

Reconstruction from the manifold: Because manifold learning methods generally do not provide a set of basis functions, any mapping from the embedded space to the higher-dimensional input space must be accomplished through a reconstruction based on the location of nearest neighbors. This means that a projection derived from these methods cannot be used to compress data in a way that is analogous to PCA. The full input data set and the full projected data must be accessed in order to map new points between the two spaces.

With these weaknesses in mind, manifold learning techniques can still be used successfully to analyze and visualize large astronomical data sets. LLE has been applied successfully to several classes of data, both as a classification technique and an outlier detection technique [10, 27, 32]. These methods are ripe for exploration of the high-dimensional data sets available from future large astronomical surveys.

7.6. Independent Component Analysis and Projection Pursuit

Independent component analysis (ICA) [8] is a computational technique that has become popular in the biomedical signal processing community to solve what has often been referred to as the "cocktail party problem"; see [7]. In this problem, there are multiple microphones situated through out a room containing N people. Each microphone picks up a linear combination of the N voices. The goal of ICA is to use the concept of statistical independence to isolate (or unmix) the individual signals. In the context of astronomical problems we will consider the application of ICA to the series of galaxy spectra used for PCA (see §7.3.2). In this example, each galaxy spectrum is considered as the microphone picking up a linear combination of input signals from individual stars and HII regions.

Each spectrum, $x_i(k)$, can now be described by

$$x_1(k) = a_{11}s_1(k) + a_{12}s_2(k) + a_{13}s_3(k) + \cdots, \tag{7.34}$$

$$x_2(k) = a_{21}s_1(k) + a_{22}s_2(k) + a_{23}s_3(k) + \cdots, \tag{7.35}$$

$$x_3(k) = a_{31}s_1(k) + a_{32}s_2(k) + a_{33}s_3(k) + \cdots, \tag{7.36}$$

where $s_i(k)$ are the individual stellar spectra and a_{ij} the appropriate mixing amplitudes. In matrix format we can write this as

$$X = AS, \tag{7.37}$$

where X and S are matrices for the set of input spectra and stellar spectra, respectively. Extracting these signal spectra is equivalent to estimating the appropriate weight matrix, W, such that

$$S = WX. \tag{7.38}$$

The principle that underlies ICA comes from the observation that the input signals, $s_i(k)$, should be statistically independent. Two random variables are considered statistically independent if their joint probability distribution, $f(x, y)$, can be fully described by a combination of their marginalized probabilities, that is,

$$f(x^p, y^q) = f(x^p)f(y^q), \tag{7.39}$$

where p and q represent arbitrary higher-order moments of the probability distributions. For the case of PCA, $p = q = 1$ and the statement of independence simplifies to the weaker condition of uncorrelated data (see §7.3.1 on the derivation of PCA).

In most implementations of ICA algorithms the requirement for statistical independence is expressed in terms of the non-Gaussianity of the probability distributions. The rationale for this is that the sum of any two independent random variables will always be more Gaussian than either of the individual random variables (i.e., from the central limit theorem). This would mean that, for the case of the stellar components that make up a galaxy spectrum, if we identify an unmixing matrix, W, that maximizes the non-Gaussianity of the distributions, then we would be identifying the input signals. Definitions of non-Gaussianity range from the use of the kurtosis of a distribution (see §5.2.2), the negentropy (the negative of the entropy of a distribution), and mutual information.

Related to ICA is projection pursuit [15, 16]. Both techniques seek interesting directions within multivariate data sets. One difference is that in projection pursuit these directions are identified one at a time, while for ICA the search for these signals can be undertaken simultaneously. For each case, the definition of "interesting" is often expressed in terms of how non-Gaussian the distributions are after projections. Projection Pursuit can be considered as a subset of ICA.

Scikit-learn has an implementation of ICA based on the FastICA algorithm [18]. It can be used as follows:

```
import numpy as np
from sklearn.decomposition import FastICA

X = np.random.normal(size=(100, 2))
    # 100 pts in 2 dims
R = np.random.random((2, 5))  # mixing matrix
X = np.dot(X, R)  # X is now 2D data in 5D space
ica = FastICA(2)  # fit two components
ica.fit(X)
proj = ica.transform(X) # 100 x 2 projection of data

comp = ica.components_  # the 2 x 5 matrix of indep.
       # components
sources = ica.sources_  # the 100 x 2 matrix of
          # sources
```

There are several options to fine-tune the algorithm; for details refer to the Scikit-learn documentation.

7.6.1. The Application of ICA to Astronomical Data

In figure 7.1 we showed the set of spectra used to test PCA. From the eigenspectra and their associated eigenvalues it was apparent that each spectrum comprises a linear combination of basis functions whose shapes are broadly consistent with the spectral properties of individual stellar types (O, A, and G stars). In the middle panel of figure 7.4 we apply ICA to these same spectra to define the independent components. As with PCA, preprocessing of the input data is an important component of any ICA application. For each data set the mean vector is removed to center the data. Whitening of the distributions (where the covariance matrix is diagonalized and normalized to reduce it to the identity matrix) is implemented through an eigenvalue decomposition of the covariance matrix.

The cost function employed is that of FastICA [18] which uses an analytic approximation to the negentropy of the distributions. The advantage of FastICA is that each of the independent components can be evaluated one at a time. Thus the analysis can be terminated once a sufficient number of components has been identified.

Comparison of the components derived from PCA, NMF and ICA in figure 7.4 shows that each of these decompositions produces a set of basis functions that are broadly similar (including both continuum and line emission). The ordering of the importance of each component is dependent on technique: in the case of ICA, finding a subset of ICA components is not the same as finding all ICA components (see [17]), which means that the a priori assumption of the number of underlying components will affect the form of the resulting components. This choice of dimensionality is a problem common to all dimensionality techniques (from LLE to the truncation of

PCA components). As with many multivariate applications, as the size of the mixing matrix grows, computational complexity often makes it impractical to calculate the weight matrix W directly. Reduction in the complexity of the input signals through the use of PCA (either to filter the data or to project the data onto these basis functions) is often applied to ICA applications.

7.7. Which Dimensionality Reduction Technique Should I Use?

In chapter 6 we introduced the concept of defining four axes against which we can compare the techniques described in each chapter. Using the axes of "accuracy," "interpretability," "simplicity," and "speed" (see §6.6 for a description of these terms) we provide a rough assessment of each of the methods considered in this chapter.

What are the most *accurate* dimensionality reduction methods? First we must think about how we want to define the notion of accuracy for the task of dimension reduction. In some sense the different directions one can take here are what distinguishes the various methods. One clear notion of this is in terms of reconstruction error—some function of the difference between the original data matrix and the data matrix reconstructed using the desired number of components. When all possible components are used, every dimensionality reduction method, in principle, is an exact reconstruction (zero error), so the question becomes some function of the number of components—for example, which method tends to yield the most faithful reconstruction after K components. PCA is designed to provide the best square reconstruction error for any given K. LLE minimizes square reconstruction error, but in a nonlinear fashion. NMF also minimizes a notion of reconstruction error, under nonnegativity constraints. Other approaches, such as the multidimensional scaling family of methods ([4, 22]), of which IsoMap is a modern descendant and variant, attempt to minimize the difference between each pairwise distance in the original space and its counterpart in the reconstructed space. In other words, once one has made a choice of which notion of accuracy is desired, there is generally a method which directly maximizes that notion of accuracy. Generally speaking, for a fixed notion of error, a nonlinear model should better minimize error than one which is constrained to be linear (i.e., manifold learning techniques should provide more compact and accurate representations of the data). Additional constraints, such as nonnegativity, only reduce the flexibility the method has to fit the data, making, for example, NMF generally worse in the sense of reconstruction error than PCA. ICA can also be seen in the light of adding additional constraints, making it less aggressive in terms of reconstruction error.

NMF-type approaches do, however, have a significant performance advantage over PCA, ICA, and manifold learning for the case of low signal-to-noise data due to the fact that they have been expanded to account for heteroscedastic uncertainties; see [31]. NMF and its variants can model the *measured* uncertainties within the data (rather than assuming that the variance of the ensemble of data is representative of these uncertainties). This feature, at least in principle, provides a more accurate set of derived basis functions.

What are the most *interpretable* dimensionality reduction methods? In the context of dimensionality reduction methods, interpretability generally means the extent to which we can identify the meaning of the directions found by the method. For data sets where all values are known to be nonnegative, as in the case of spectra or image pixel brightnesses, NMF tends to yield more sensible components than PCA. This makes sense because the constraints ensure that reconstructions obtained from NMF are themselves valid spectra or images, which might not be true otherwise. Interpretability in terms of understanding each component through the original dimensions it captures can be had to some extent with linear models like PCA, ICA, and NMF, but is lost in the nonlinear manifold learning approaches like LLE and IsoMap (which do not provide the principal directions, but rather the positions of objects within a lower-dimensional space).

A second, and significant, advantage of PCA over the other techniques is the ability to estimate the importance of each principal component (in terms of its contribution to the variance). This enables simple, though admittedly ad hoc (see §7.3.2), criteria to be applied when truncating the projection of data onto a set of basis functions.

What are the most *scalable* dimensionality reduction methods? Faster algorithms can be obtained for PCA in part by focusing on the fact that only the top K components are typically needed. Approximate algorithms for SVD-like computations (where only parts of the SVD are obtained) based on sampling are available, as well as algorithms using similar ideas that yield full SVDs. Online algorithms can be fairly effective for SVD-like computations. ICA can be approached a number of ways, for example using the FastICA algorithm [18]. Fast optimization methods for NMF are discussed in [23]. In general, linear methods for dimensionality reduction are usually relatively tractable, even for high N. For nonlinear methods such as LLE and IsoMap, the most expensive step is typically the first one, an $\mathcal{O}(N^2)$ all-nearest-neighbor computation to find the K neighbors for each point. Fast algorithms for such problems, including one that is provably $\mathcal{O}(N)$, are discussed in §2.4 (see also WSAS). The second step is typically a kind of SVD computation. Overall LLE is $\mathcal{O}(N^2)$ and IsoMap is $\mathcal{O}(N^3)$, making them intractable on large data sets without algorithmic intervention.

What are the *simplest* dimensionality reduction methods? All dimensionality reduction methods are fiddly in the sense that they all require, to some extent, the selection of the number of components/dimensions to which the data should be reduced. PCA's objective is convex, meaning that there is no need for multiple random restarts, making it usable essentially right out of the box. This is not the case for ICA and NMF, which have nonconvex objectives. Manifold learning methods like LLE and IsoMap have convex objectives, but require careful evaluation of the parameter k which defines the number of nearest neighbors.

As mentioned in §6.6, selecting parameters can be done automatically in principle for any machine learning method for which cross-validation (see §8.11) can be applied, including unsupervised methods as well as supervised ones. Cross-validation can be applied whenever the model can be used to make predictions (in this case "predictions" means coordinates in the new low-dimensional space) on test data, that is, data not included in the training data. While PCA, NMF, and ICA can

TABLE 7.1.
Summary of the practical properties of the main dimensionality reduction techniques.

Method	Accuracy	Interpretability	Simplicity	Speed
Principal component analysis	H	H	H	H
Locally linear embedding	H	M	H	M
Nonnegative matrix factorization	H	H	M	M
Independent component analysis	M	M	L	L

be applied to such "out-of-sample" data, LLE and IsoMap effectively require, in order to get predictions, that test data be added to the existing training set and the whole model retrained. This requirement precludes the use of cross-validation to select k automatically and makes the choice of k a subjective exercise.

Other considerations, and taste. As we noted earlier in §7.5.3, other considerations can include robustness to outliers and missing values. Many approaches have been taken to making robust versions of PCA, since PCA, as a maximum-likelihood-based model, is sensitive to outliers. Manifold learning methods can also be sensitive to outliers. PCA, ICA, and NMF can be made to work with missing values in a number of ways: we saw one example for PCA in §7.3.3. There is no clear way to handle missing values with manifold learning methods, as they are based on the idea of distances, and it is not clear how to approach the problem of computing distances with missing values. Regarding taste, if we consider these criteria as a whole the simplest and most useful technique is principal component analysis (as has been recognized by the astronomical community with over 300 refereed astronomical publications between 2000 and 2010 that mention the use of PCA). Beyond PCA, NMF has the advantage of mapping to many astronomical problems (i.e., for positive data and in the low signal-to-noise regime) and is particularly suitable for smaller data sets. ICA has interesting roots in information theory and signal processing, and manifold learning has roots in geometry; their adoption within the astronomical community has been limited to date.

Simple summary. Table 7.1 is a simple summary of the trade-offs along our axes of accuracy, interpretability, simplicity, and speed in dimension reduction methods, expressed in terms of high (H), medium (M), and low (L) categories.

References

[1] Abazajian, K. N., J. K. Adelman-McCarthy, M. A. Agüeros, and others (2009). The Seventh Data Release of the Sloan Digital Sky Survey. *ApJS 182*, 543–558.

[2] Bellman, R. E. (1961). *Adaptive Control Processes*. Princeton, NJ: Princeton University Press.

[3] Blanton, M. R. and S. Roweis (2007). K-corrections and filter transformations in the ultraviolet, optical, and near-infrared. *AJ 133*, 734–754.

[4] Borg, I. and P. Groenen (2005). *Modern Multidimensional Scaling: Theory and Applications*. Springer.

[5] Budavári, T., V. Wild, A. S. Szalay, L. Dobos, and C.-W. Yip (2009). Reliable eigenspectra for new generation surveys. *MNRAS 394*, 1496–1502.

[6] Cattell, R. B. (1966). The scree test for the number of factors. *Multivariate Behavioral Research 1*(2), 245–276.

[7] Cherry, E. C. (1953). Some experiments on the recognition of speech, with one and with two ears. *Acoustical Society of America Journal 25*, 975.

[8] Comon, P. (1994). Independent component analysis—a new concept? *Signal Processing 36*(3), 287–314.

[9] Connolly, A. J. and A. S. Szalay (1999). A robust classification of galaxy spectra: Dealing with noisy and incomplete data. *AJ 117*, 2052–2062.

[10] Daniel, S. F., A. Connolly, J. Schneider, J. Vanderplas, and L. Xiong (2011). Classification of stellar spectra with local linear embedding. *AJ 142*, 203.

[11] de Ridder, D. and R. P. Duin (2002). Locally linear embedding for classification. *Pattern Recognition Group Technical Report Series PH-2002-01*.

[12] Everson, R. and L. Sirovich (1995). Karhunen-Loeve procedure for gappy data. *J. Opt. Soc. Am. A 12*, 1657–1664.

[13] Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM 5*, 345.

[14] Fredman, M. L. and R. E. Tarjan (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 596–615.

[15] Friedman, J. H. (1987). Exploratory projection pursuit. *Journal of the American Statistical Association 82*, 249–266.

[16] Friedman, J. H. and J. W. Tukey (1974). A projection pursuit algorithm for exploratory data analysis. *IEEE Transactions on Computers 23*, 881–889.

[17] Girolami, M. and C. Fyfe (1997). Negentropy and kurtosis as projection pursuit indices provide generalised ICA algorithms. In A. Cichocki and A.Back (Eds.), *NIPS-96 Blind Signal Separation Workshop*, Volume 8.

[18] Hyvaerinen, A. (1999). Fast and robust fixed-point algorithms for independent component analysis. *IEEE-NN 10*(3), 626.

[19] Jolliffe, I. T. (1986). *Principal Component Analysis*. Springer.

[20] Kaiser, H. F. (1960). The application of electronic computers to factor analysis. *Educational and Psychological Measurement 20*(1), 141–151.

[21] Karhunen, H. (1947). Uber Lineare Methoden in der Wahrscheinlichkeitsrechnung. *Ann. Acad. Sci. Fenn. Ser. A.I. 37*. See translation by I. Selin, The Rand Corp., Doc. T-131, 1960.

[22] Kruskal, J. and M. Wish (1978). *Multidimensional Scaling*. Number 11 in Quantitative Applications in the Social Sciences. SAGE Publications.

[23] Lee, D. D. and H. S. Seung (2001). Algorithms for non-negative matrix factorization. In T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.), *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, Cambridge, Massachusetts, pp. 556–562. MIT Press.

[24] Lehoucq, R. B., D. C. Sorensen, and C. Yang (1997). ARPACK users guide: Solution of large scale eigenvalue problems by implicitly restarted Arnoldi methods.

[25] Loéve, M. (1963). *Probability Theory*. New York: Van Nostrand.

[26] Mardia, K. V., J. T. Kent, and J. M. Bibby (1979). *Multivariate Analysis*. Academic Press.

[27] Matijevič, G., A. Prša, J. A. Orosz, and others (2012). Kepler eclipsing binary stars. III. Classification of Kepler eclipsing binary light curves with locally linear embedding. *AJ 143*, 123.

[28] Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science 2*, 559–572.

[29] Roweis, S. T. and L. K. Saul (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science 290*, 2323–2326.

[30] Tenenbaum, J. B., V. Silva, and J. C. Langford (2000). A global geometric framework for nonlinear dimensionality reduction. *Science 290*(5500), 2319–2323.

[31] Tsalmantza, P. and D. W. Hogg (2012). A data-driven model for spectra: Finding double redshifts in the Sloan Digital Sky Survey. *ApJ 753*, 122.

[32] Vanderplas, J. and A. Connolly (2009). Reducing the dimensionality of data: Locally linear embedding of Sloan Galaxy Spectra. *AJ 138*, 1365–1379.

[33] Yip, C. W., A. J. Connolly, A. S. Szalay, and others (2004). Distributions of galaxy spectral types in the Sloan Digital Sky Survey. *AJ 128*, 585–609.

[34] Yip, C. W., A. J. Connolly, D. E. Vanden Berk, and others (2004). Spectral classification of quasars in the Sloan Digital Sky Survey: Eigenspectra, redshift, and luminosity effects. *AJ 128*, 2603–2630.

8 Regression and Model Fitting

"Why are you trying so hard to fit in when you were born to stand out?" (Ian Wallace)

Regression is a special case of the general model fitting and selection procedures discussed in chapters 4 and 5. It can be defined as the relation between a dependent variable, y, and a set of independent variables, x, that describes the expectation value of y given x: $E[y|x]$. The purpose of obtaining a "best-fit" model ranges from scientific interest in the values of model parameters (e.g., the properties of dark energy, or of a newly discovered planet) to the predictive power of the resulting model (e.g., predicting solar activity). The usage of the word regression for this relationship dates back to Francis Galton, who discovered that the difference between a child and its parents for some characteristic is proportional to its parents' deviation from typical people in the whole population,[1] or that children "regress" toward the population mean. Therefore, modern usage of the word in a statistical context is somewhat different.

As we will describe below, regression can be formulated in a way that is very general. The solution to this generalized problem of regression is, however, quite elusive. Techniques used in regression tend, therefore, to make a number of simplifying assumptions about the nature of the data, the uncertainties of the measurements, and the complexity of the models. In the following sections we start with a general formulation for regression, list various simplified cases, and then discuss methods that can be used to address them, such as regression for linear models, kernel regression, robust regression and nonlinear regression.

8.1. Formulation of the Regression Problem

Given a multidimensional data set drawn from some pdf and the full error covariance matrix for each data point, we can attempt to infer the underlying pdf using either parametric or nonparametric models. In its most general incarnation, this is a

[1] If your parents have very high IQs, you are more likely to have a lower IQ than them, than a higher one. The expected probability distribution for your IQ if you also have a sister whose IQ exceeds your parents' IQs is left as an exercise for the reader. Hint: This is related to regression toward the mean discussed in § 4.7.1.

very hard problem to solve. Even with a restrictive assumption that the errors are Gaussian, incorporating the error covariance matrix within the posterior distribution is not trivial (cf. § 5.6.1). Furthermore, accounting for any selection function applied to the data can increase the computational complexity significantly (e.g., recall § 4.2.7 for the one-dimensional case), and non-Gaussian error behavior, if not accounted for, can produce biased results.

Regression addresses a slightly simpler problem: instead of determining the multidimensional pdf, we wish to infer the expectation value of y given x (i.e., the conditional expectation value). If we have a model for the conditional distribution (described by parameters $\boldsymbol{\theta}$) we can write this function[2] as $y = f(x|\boldsymbol{\theta})$. We refer to y as a scalar dependent variable and x as an independent vector. Here x does not need to be a random variable (e.g., x could correspond to deterministic sampling times for a time series). For a given model class (i.e., the function f can be an analytic function such as a polynomial, or a nonparametric estimator), we have k model parameters θ_p, $p = 1, \ldots, k$.

Figure 8.1 illustrates how the constraints on the model parameters, $\boldsymbol{\theta}$, respond to the observations x_i and y_i. In this example, we assume a simple straight-line model with $y_i = \theta_0 + \theta_1 x_i$. Each point provides a joint constraint on θ_0 and θ_1. If there were no uncertainties on the variables then this constraint would be a straight line in the (θ_0, θ_1) plane ($\theta_0 = y_i - \theta_1 x_i$). As the number of points is increased the best estimate of the model parameters would then be the intersection of all lines. Uncertainties within the data will transform the constraint from a line to a distribution (represented by the region shown as a gray band in figure 8.1). The best estimate of the model parameters is now given by the posterior distribution. This is simply the multiplication of the probability distributions (constraints) for all points and is shown by the error ellipses in the lower panel of figure 8.1. Measurements with upper limits (e.g., point x_4) manifest as half planes within the parameter space. Priors are also accommodated naturally within this picture as additional multiplicative constraints applied to the likelihood distribution (see § 8.2).

Computationally, the cost of this general approach to regression can be prohibitive (particularly for large data sets). In order to make the analysis tractable, we will, therefore, define several types of regression using three "classification axes":

- *Linearity.* When a parametric model is linear in all model parameters, that is, $f(x|\theta) = \sum_{p=1}^{k} \theta_p \, g_p(x)$, where functions $g_p(x)$ do not depend on any free model parameters (but can be nonlinear functions of x), regression becomes a significantly simpler problem, called linear regression. Examples of this include polynomial regression, and radial basis function regression. Regression of models that include nonlinear dependence on θ_p, such as $f(x|\theta) = \theta_1 + \theta_2 \sin(\theta_3 x)$, is called nonlinear regression.
- *Problem complexity.* A large number of independent variables increases the complexity of the error covariance matrix, and can become a limiting factor in nonlinear regression. The most common regression case found in practice is the $M = 1$ case with only a single independent variable (i.e., fitting a straight line to data). For linear models and negligible errors on the independent variables, the problem of dimensionality is not (too) important.

[2]Sometimes $f(x; \boldsymbol{\theta})$ is used instead of $f(x|\boldsymbol{\theta})$ to emphasize that here f is a function rather than pdf.

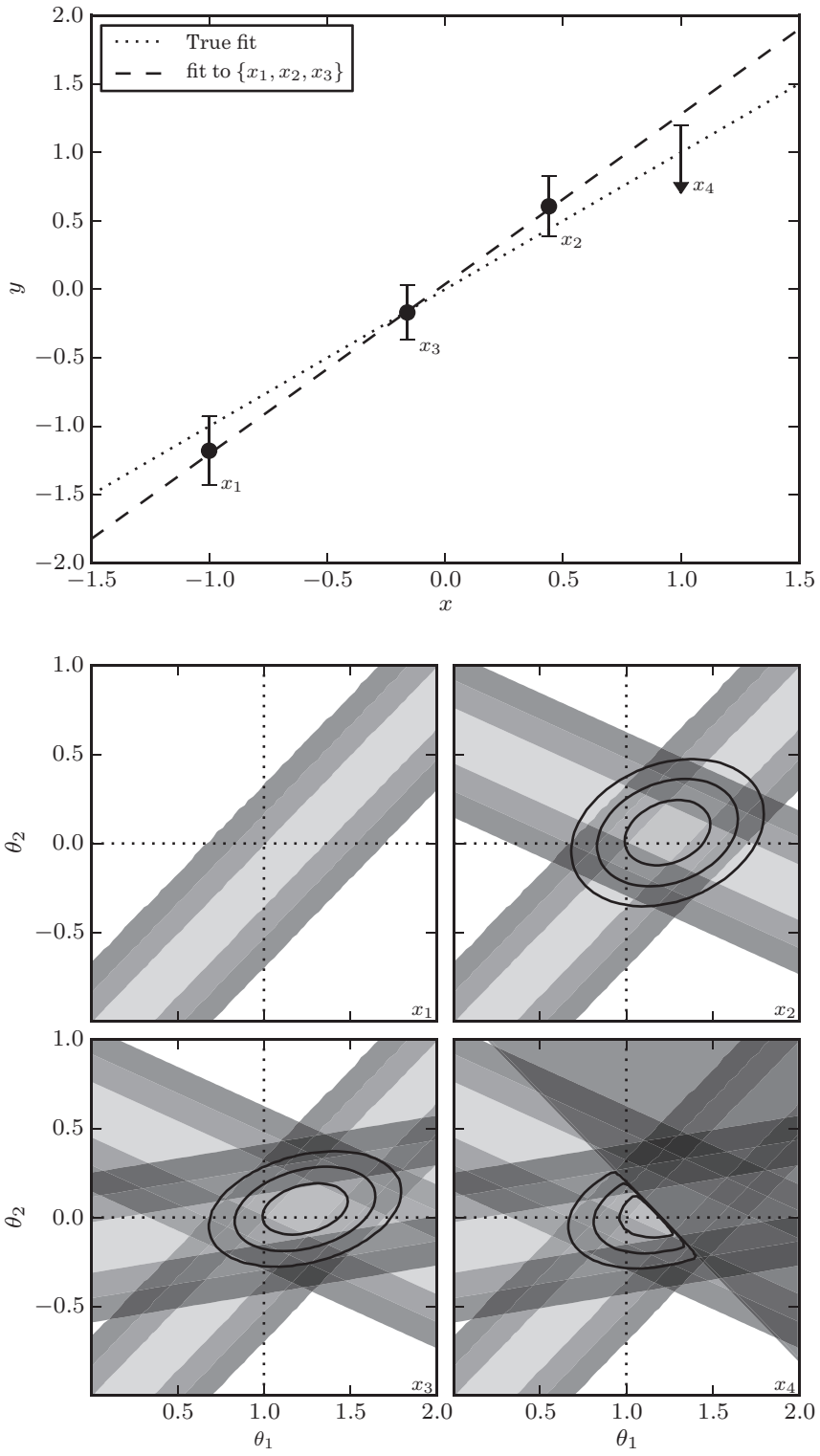

Figure 8.1. An example showing the online nature of Bayesian regression. The upper panel shows the four points used in regression, drawn from the line $y = \theta_1 x + \theta_0$ with $\theta_1 = 1$ and $\theta_0 = 0$. The lower panel shows the posterior pdf in the (θ_0, θ_1) plane as each point is added in sequence. For clarity, the implied dark regions for $\sigma > 3$ have been removed. The fourth point is an upper-limit measurement of y, and the resulting posterior cuts off half the parameter space.

- *Error behavior*. The uncertainties in the values of independent and dependent variables, and their correlations, are the primary factor that determines which regression method to use. The structure of the error covariance matrix, and deviations from Gaussian error behavior, can turn seemingly simple problems into complex computational undertakings. Here we will separately discuss the following cases:

    1. Both independent and dependent variables have negligible errors (compared to the intrinsic spread of data values); this is the simplest and most common "y vs. x" case, and can be relatively easily solved even for nonlinear models and multidimensional data.
    2. Only errors for the dependent variable (y) are important, and their distribution is Gaussian and homoscedastic (with σ either known or unknown).
    3. Errors for the dependent variable are Gaussian and known, but heteroscedastic.
    4. Errors for the dependent variable are non-Gaussian, and their behavior is known.
    5. Errors for the dependent variable are non-Gaussian, but their exact behavior is unknown.
    6. Errors for independent variables (x) are not negligible, but the full covariance matrix can be treated as Gaussian. This case is relatively straightforward when fitting a straight line, but can become cumbersome for more complex models.
    7. All variables have non-Gaussian errors. This is the hardest case and there is no ready-to-use general solution. In practice, the problem is solved on a case-by-case basis, typically using various approximations that depend on the problem specifics.

For the first four cases, when error behavior for the dependent variable is known, and errors for independent variables are negligible, we can easily use the Bayesian methodology developed in chapter 5 to write the posterior pdf for the model parameters,

$$p(\boldsymbol{\theta}|\{x_i, y_i\}, I) \propto p(\{x_i, y_i\}|\boldsymbol{\theta}, I)\, p(\boldsymbol{\theta}, I). \tag{8.1}$$

Here the information I describes the error behavior for the dependent variable. The data likelihood is the product of likelihoods for the individual points, and the latter can be expressed as

$$p(y_i|x_i, \boldsymbol{\theta}, I) = e(y_i|y), \tag{8.2}$$

where $y = f(x|\boldsymbol{\theta})$ is the adopted model class, and $e(y_i|y)$ is the probability of observing y_i given the true value (or the model prediction) y. For example, if the y error distribution is Gaussian, with the width for ith data point given by σ_i, and the errors on x are negligible, then

$$p(y_i|x_i, \boldsymbol{\theta}, I) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left(\frac{-[y_i - f(x_i|\boldsymbol{\theta})]^2}{2\sigma_i^2}\right). \tag{8.3}$$

8.1.1. Data Sets Used in This Chapter

For regression and its application to astrophysics we focus on the relation between the redshifts of supernovas and their luminosity distance (i.e., a cosmological parametrization of the expansion of the universe [1]). To accomplish this we generate a set of synthetic supernova data assuming a cosmological model given by

$$\mu(z) = -5\log_{10}\left((1+z)\frac{c}{H_0}\int \frac{dz}{(\Omega_m(1+z)^3+\Omega_\Lambda)^{1/2}}\right), \tag{8.4}$$

where $\mu(z)$ is the distance modulus to the supernova, H_0 is the Hubble constant, Ω_m is the cosmological matter density and Ω_Λ is the energy density from a cosmological constant. For our fiducial cosmology we choose $\Omega_m = 0.3$, $\Omega_\Lambda = 0.7$ and $H_0 = 70\,\mathrm{km\,s^{-1}\,Mpc^{-1}}$, and add heteroscedastic Gaussian noise that increases linearly with redshift. The resulting $\mu(z)$ cannot be expressed as a sum of simple closed-form analytic functions, including low-order polynomials. This example addresses many of the challenges we face when working with observational data sets: we do not know the intrinsic complexity of the model (e.g., the form of dark energy), the dependent variables can have heteroscedastic uncertainties, there can be missing or incomplete data, and the dependent variables can be correlated. For the majority of techniques described in this chapter we will assume that uncertainties in the independent variables are small (relative to the range of data and relative to the dependent variables). In real-world applications we do not get to make this choice (the observations themselves define the distribution in uncertainties irrespective of the models we assume). For the supernova data, an example of such a case would be if we estimated the supernova redshifts using broadband photometry (i.e., photometric redshifts). Techniques for addressing such a case are described in § 8.8.1. We also note that this toy model data set is a simplification in that it does not account for the effect of K-corrections on the observed colors and magnitudes; see [7].

8.2. Regression for Linear Models

Given an independent variable x and a dependent variable y, we will start by considering the simplest case, a linear model with

$$y_i = \theta_0 + \theta_1 x_i + \epsilon_i. \tag{8.5}$$

Here θ_0 and θ_1 are the coefficients that describe the regression (or objective) function that we are trying to estimate (i.e., the slope and intercept for a straight line $f(x) = \theta_0 + \theta_1 x_i$), and ϵ_i represents an additive noise term.

The assumptions that underlie our linear regression model include the uncertainties on the independent variables that are considered to be negligible, and the dependent variables have known heteroscedastic uncertainties, $\epsilon_i = \mathcal{N}(0, \sigma_i)$. From eq. 8.3 we can write the data likelihood as

$$p(\{y_i\}|\{x_i\}, \boldsymbol{\theta}, I) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(\frac{-(y_i - (\theta_0 + \theta_1 x_i))^2}{2\sigma_i^2}\right). \tag{8.6}$$

For a flat or uninformative prior pdf, $p(\theta|I)$, where we have no knowledge about the distribution of the parameters θ, the posterior will be directly proportional to the likelihood function (which is also known as the error function). If we take the logarithm of the posterior then we arrive at the classic definition of regression in terms of the log-likelihood:

$$\ln(L) \equiv \ln((\boldsymbol{\theta}|\{x_i, y_i\}, I)) \propto \sum_{i=1}^{N} \left(\frac{-(y_i - (\theta_0 + \theta_1 x_i))^2}{2\sigma_i^2} \right). \tag{8.7}$$

Maximizing the log-likelihood as a function of the model parameters, $\boldsymbol{\theta}$, is achieved by minimizing the sum of the square errors. This observation dates back to the earliest applications of regression with the work of Gauss [6] and Legendre [14], when the technique was introduced as the "method of least squares."

The form of the likelihood function and the "method of least squares" optimization arises from our assumption of Gaussianity for the distribution of uncertainties in the dependent variables. Other forms for the likelihoods can be assumed (e.g., using the L_1 norm, see § 4.2.8, which actually precedes the use of the L_2 norm [2, 13], but this is usually at the cost of increased computational complexity). If it is known that measurement errors follow an exponential distribution (see § 3.3.6) instead of a Gaussian distribution, then the L_1 norm should be used instead of the L_2 norm and eq. 8.7 should be replaced by

$$\ln(L) \propto \sum_{i=1}^{N} \left(\frac{-|y_i - (\theta_0 + \theta_1 x_i)|}{\Delta_i} \right). \tag{8.8}$$

For the case of Gaussian homoscedastic uncertainties, the minimization of eq. 8.7 simplifies to

$$\theta_1 = \frac{\sum_i^N x_i y_i - \bar{x}\bar{y}}{\sum_i^N (x_i - \bar{x})^2}, \tag{8.9}$$

$$\theta_0 = \bar{y} - \theta_1 \bar{x}, \tag{8.10}$$

where $\bar{x}$ is the mean value of x and $\bar{y}$ is the mean value of y. As an illustration, these estimates of θ_0 and θ_1 correspond to the center of the ellipse shown in the bottom-left panel in figure 8.1. An estimate of the variance associated with this regression and the standard errors on the estimated parameters are given by

$$\sigma^2 = \sum_{i=1}^{N} (y_i - \theta_0 + \theta_1 x_i)^2, \tag{8.11}$$

$$\sigma_{\theta_1}^2 = \sigma^2 \frac{1}{\sum_i^N (x_i - \bar{x})^2}, \tag{8.12}$$

$$\sigma_{\theta_0}^2 = \sigma^2 \left(\frac{1}{N} + \frac{\bar{x}^2}{\sum_i^N (x_i - \bar{x})^2} \right). \tag{8.13}$$

For heteroscedastic errors, and in general for more complex regression functions, it is easier and more compact to generalize regression in terms of matrix notation. We, therefore, define regression in terms of a design matrix, M, such that

$$Y = M\boldsymbol{\theta}, \tag{8.14}$$

where Y is an N-dimensional vector of values y_i,

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ y_{N-1} \end{bmatrix}. \tag{8.15}$$

For our straight-line regression function, $\boldsymbol{\theta}$ is a two-dimensional vector of regression coefficients,

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}, \tag{8.16}$$

and M is a $2 \times N$ matrix,

$$M = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ . & . \\ 1 & x_{N-1} \end{bmatrix}, \tag{8.17}$$

where the constant value in the first column captures the θ_0 term in the regression.

For the case of heteroscedastic uncertainties, we define a covariance matrix, C, as an $N \times N$ matrix,

$$C = \begin{bmatrix} \sigma_0^2 & 0 & . & 0 \\ 0 & \sigma_1^2 & . & 0 \\ . & . & . & . \\ 0 & 0 & . & \sigma_{N-1}^2 \end{bmatrix} \tag{8.18}$$

with the diagonals of this matrix containing the uncertainties, σ_i, on the dependent variable, Y.

The maximum likelihood solution for this regression is

$$\boldsymbol{\theta} = (M^T C^{-1} M)^{-1} (M^T C^{-1} Y), \tag{8.19}$$

which again minimizes the sum of the square errors, $(Y - \boldsymbol{\theta} M)^T C^{-1} (Y - \boldsymbol{\theta} M)$, as we did explicitly in eq. 8.9. The uncertainties on the regression coefficients, $\boldsymbol{\theta}$, can now be expressed as the symmetric matrix

$$\Sigma_\theta = \begin{bmatrix} \sigma_{\theta_0}^2 & \sigma_{\theta_0\theta_1} \\ \sigma_{\theta_0\theta_1} & \sigma_{\theta_1}^2 \end{bmatrix} = [M^T C^{-1} M]^{-1}. \tag{8.20}$$

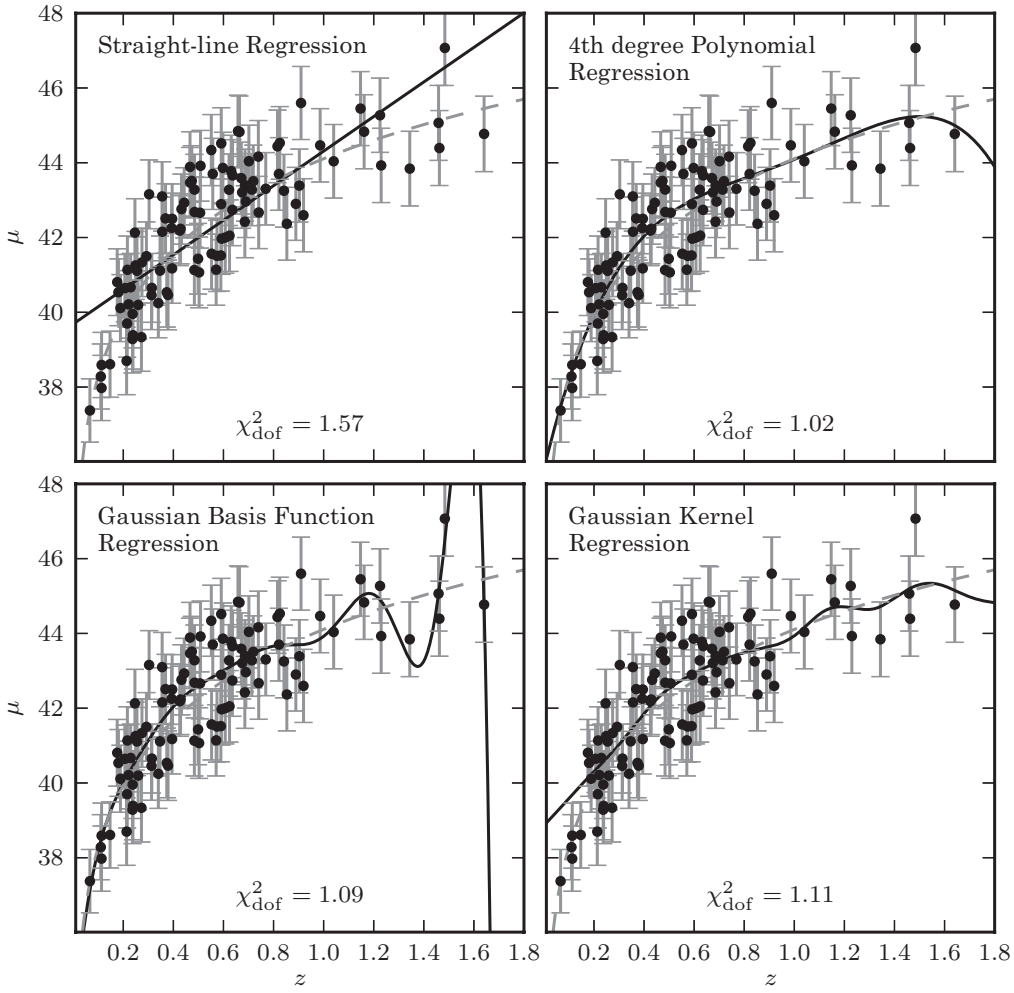

Figure 8.2. Various regression fits to the distance modulus vs. redshift relation for a simulated set of 100 supernovas, selected from a distribution $p(z) \propto (z/z_0)^2 \exp[(z/z_0)^{1.5}]$ with $z_0 = 0.3$. Gaussian basis functions have 15 Gaussians evenly spaced between $z = 0$ and 2, with widths of 0.14. Kernel regression uses a Gaussian kernel with width 0.1.

Whether we have sufficient data to constrain the regression (i.e., sufficient degrees of freedom) is defined by whether $M^T M$ is an invertible matrix.

The top-left panel of figure 8.2 illustrates a simple linear regression of redshift, z, against distance modulus, μ, for the set of 100 supernovas described in § 8.1.1. The solid line shows the regression function for the straight-line model and the dashed line the underlying cosmological model from which the data were drawn (which of course cannot be described by a straight line). It is immediately apparent that the chosen regression model does not capture the structure within the data at the high and low redshift limits—the model does not have sufficient flexibility to reproduce the correlation displayed by the data. This is reflected in the $\chi^2_{\rm dof}$ for this fit which is 1.54 (see § 4.3.1 for a discussion of the interpretation of $\chi^2_{\rm dof}$).

We now relax the assumptions we made at the start of this section, allowing not just for heteroscedastic uncertainties but also for correlations between the measures of the dependent variables. With no loss in generality, eq. 8.19 can be extended to allow for covariant data through the off-diagonal elements of the covariance matrix C.

8.2.1. Multivariate Regression

For multivariate data (where we fit a hyperplane rather than a straight line) we simply extend the description of the regression function to multiple dimensions, with $y = f(x|\boldsymbol{\theta})$ given by

$$y_i = \theta_0 + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_k x_{ik} + \epsilon_i \tag{8.21}$$

with θ_i the regression parameters and x_{ik} the kth component of the ith data entry within a multivariate data set. This multivariate regression follows naturally from the definition of the design matrix with

$$M = \begin{pmatrix} 1 & x_{01} & x_{02} & . & x_{0k} \\ 1 & x_{11} & x_{12} & . & x_{1k} \\ . & . & . & . & . \\ 1 & x_{N1} & x_{N2} & . & x_{Nk} \end{pmatrix}. \tag{8.22}$$

The regression coefficients (which are estimates of $\boldsymbol{\theta}$ and are often differentiated from the true values by writing them as $\hat{\boldsymbol{\theta}}$) and their uncertainties are, as before,

$$\boldsymbol{\theta} = (M^T C^{-1} M)^{-1} (M^T C^{-1} Y) \tag{8.23}$$

and

$$\Sigma_{\boldsymbol{\theta}} = [M^T C^{-1} M]^{-1}. \tag{8.24}$$

Multivariate linear regression with homoscedastic errors on dependent variables can be performed using the routine `sklearn.linear_model.LinearRegression`. For data with homoscedastic errors, AstroML implements a similar routine:

```
import numpy as np
from astroML.linear_model import LinearRegression
X = np.random.random((100, 2))  # 100 points in
     2 dimensions
dy = np.random.random(100)  # heteroscedastic errors
y = np.random.normal(X[:, 0] + X[:, 1], dy)

model = LinearRegression()
model.fit(X, y, dy)
y_pred = model.predict(X)
```

`LinearRegression` in Scikit-learn has a similar interface, but does not explicitly account for heteroscedastic errors. For a more realistic example, see the source code of figure 8.2.

8.2.2. Polynomial and Basis Function Regression

Due to its simplicity, the derivation of regression in most textbooks is undertaken using a straight-line fit to the data. However, the straight line can simply be interpreted as a first-order expansion of the regression function $y = f(x|\boldsymbol{\theta})$. In general we can express $f(x|\boldsymbol{\theta})$ as the sum of arbitrary (often nonlinear) functions as long as the model is linear in terms of the regression parameters, $\boldsymbol{\theta}$. Examples of these general linear models include a Taylor expansion of $f(x)$ as a series of polynomials where we solve for the amplitudes of the polynomials, or a linear sum of Gaussians with fixed positions and variances where we fit for the amplitudes of the Gaussians.

Let us initially consider polynomial regression and write $f(x|\boldsymbol{\theta})$ as

$$y_i = \theta_0 + \theta_1 x_i + \theta_2 x_i^2 + \theta_3 x_i^3 + \cdots . \tag{8.25}$$

The design matrix for this expansion becomes

$$M = \begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ . & . & . & . \\ 1 & x_N & x_N^2 & x_N^3 \end{pmatrix}, \tag{8.26}$$

where the terms in the design matrix are 1, x, x^2, and x^3, respectively. The solution for the regression coefficients and the associated uncertainties are again given by eqs. 8.19 and 8.20.

A fourth-degree polynomial fit to the supernova data is shown in the top-right panel of figure 8.2. The increase in flexibility of the model improves the fit (note that we have to be aware of overfitting the data if we just arbitrarily increase the degree of the polynomial; see § 8.11). The $\chi^2_{\rm dof}$ of the regression is 1.02, which indicates a much better fit than the straight-line case. At high redshift, however, there is a systematic deviation between the polynomial regression and the underlying generative model (shown by the dashed line), which illustrates the danger of extrapolating this model beyond the range probed by the data.

Polynomial regression with heteroscedastic errors can be performed using the `PolynomialRegression` function in AstroML:

```
import numpy as np
from astroML.linear_model import PolynomialRegression

X = np.random.random((100, 2)) # 100 points in 2 dims
y = X[:, 0] ** 2 + X[:, 1] ** 3
model = PolynomialRegression(3)
        # fit 3rd degree polynomial
model.fit(X, y)
y_pred = model.predict(X)
```

Here we have used homoscedastic errors for simplicity. Heteroscedastic errors in `y` can be used in a similar way to `LinearRegression`, above. For a more realistic example, see the source code of figure 8.2.

The number of terms in the polynomial regression grows exponentially with order. Given a data set with k dimensions to which we fit a p-dimensional polynomial, the number of parameters in the model we are fitting is given by

$$m = \frac{(p+k)!}{p!\,k!}, \tag{8.27}$$

including the intercept or offset. The number of degrees of freedom for the regression model is then $\nu = N - m$ and the probability of that model is given by a χ^2 distribution with ν degrees of freedom.

We can generalize the polynomial model to a basis function representation by noting that each row of the design matrix can be replaced with any series of linear or nonlinear functions of the variables x_i. Despite the use of arbitrary basis functions, the resulting problem remains linear, because we are fitting only the coefficients multiplying these terms. Examples of commonly used basis functions include Gaussians, trigonometric functions, inverse quadratic functions, and splines.

Basis function regression can be performed using the routine `BasisFunctionRegression` in AstroML. For example, Gaussian basis function regression is as follows:

```
import numpy as np
from astroML.linear_model import
     BasisFunctionRegression

X = np.random.random((100, 1))  # 100 points in 1
    # dimension dy = 0.1
y = np.random.normal(X[:, 0], dy)
mu = np.linspace(0, 1, 10)[:, np.newaxis]
     # 10 x 1 array of mu
sigma = 0.1

model = BasisFunctionRegression('gaussian', mu=mu,
        sigma=sigma)
model.fit(X, y, dy)
y_pred = model.predict(X)
```

For a further example, see the source code of figure 8.2.

The application of Gaussian basis functions to our example regression problem is shown in figure 8.2. In the lower-left panel, 15 Gaussians, evenly spaced between redshifts $0 < z < 2$ with widths of $\sigma_z = 0.14$, are fit to the supernova data. The $\chi^2_{\rm dof}$ for this fit is 1.09, comparable to that for polynomial regression.

8.3. Regularization and Penalizing the Likelihood

All regression examples so far have sought to minimize the mean square errors between a model and data with known uncertainties. The Gauss–Markov theorem states that this least-squares approach results in the minimum variance unbiased estimator (see § 3.2.2) for the linear model. In some cases, however, the regression problem may be ill posed and the best unbiased estimator is not the most appropriate regression. Instead, we can trade an increase in bias for a reduction in variance. Examples of such cases include data that are highly correlated (which results in ill-conditioned matrices), or when the number of terms in the regression model decreases the number of degrees of freedom such that we must worry about overfitting of the data.

One solution to these problems is to penalize or limit the complexity of the underlying regression model. This is often referred to as regularization, or shrinkage, and works by applying a penalty to the likelihood function. Regularization can come in many forms, but usually imposes smoothness on the model, or limits the numbers of, or the values of, the regression coefficients.

In § 8.2 we showed that regression minimizes the least-squares equation,

$$(Y - M\boldsymbol{\theta})^T(Y - M\boldsymbol{\theta}). \tag{8.28}$$

We can impose a penalty on this minimization if we include a regularization term,

$$(Y - M\boldsymbol{\theta})^T(Y - M\boldsymbol{\theta}) + \lambda|\boldsymbol{\theta}^T\boldsymbol{\theta}|, \tag{8.29}$$

where λ is the regularization or smoothing parameter and $|\boldsymbol{\theta}^T\boldsymbol{\theta}|$ is an example of the penalty function. In this example, we penalize the size of the regression coefficients (which is known as ridge regression as we will discuss in the next section). Solving for $\boldsymbol{\theta}$ we arrive at a modification of eq. 8.19,

$$\boldsymbol{\theta} = (M^TC^{-1}M + \lambda I)^{-1}(M^TC^{-1}Y), \tag{8.30}$$

where I is the identity matrix. One aspect worth noting about robustness through regularization is that, even if $M^TC^{-1}M$ is singular, solutions can still exist for $(M^TC^{-1}M + \lambda I)$.

A Bayesian implementation of regularization would use the prior to impose constraints on the probability distribution of the regression coefficients. If, for example, we assumed that the prior on the regression coefficients was Gaussian with the width of this Gaussian governed by the regularization parameter λ then we could write it as

$$p(\boldsymbol{\theta}|I) \propto \exp\left(\frac{-(\lambda\boldsymbol{\theta}^T\boldsymbol{\theta})}{2}\right). \tag{8.31}$$

Multiplying the likelihood function by this prior results in a posterior distribution with an exponent $(Y - M\boldsymbol{\theta})^T(Y - M\boldsymbol{\theta}) + \lambda|\boldsymbol{\theta}^T\boldsymbol{\theta}|$, equivalent to the MLE regularized regression described above. This Gaussian prior corresponds to ridge regression. For LASSO regression, described below, the corresponding prior would be an exponential (Laplace) distribution.

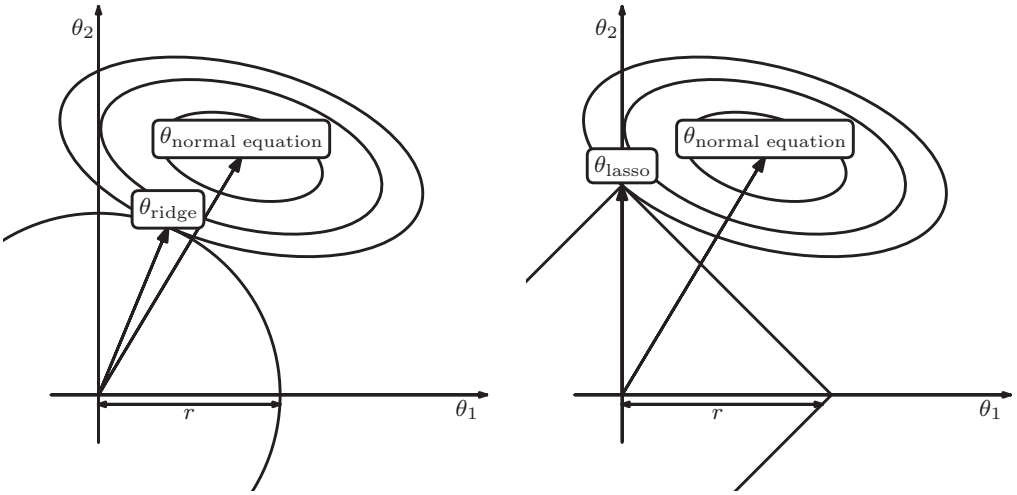

Figure 8.3. A geometric interpretation of regularization. The right panel shows L_1 regularization (LASSO regression) and the left panel L_2 regularization (ridge regularization). The ellipses indicate the posterior distribution for no prior or regularization. The solid lines show the constraints due to regularization (limiting $\boldsymbol{\theta}^2$ for ridge regression and $|\boldsymbol{\theta}|$ for LASSO regression). The corners of the L_1 regularization create more opportunities for the solution to have zeros for some of the weights.

8.3.1. Ridge Regression

The regularization example above is often referred to as ridge regression or Tikhonov regularization [22]. It provides a penalty on the sum of the squares of the regression coefficients such that

$$|\boldsymbol{\theta}|^2 < s, \tag{8.32}$$

where s controls the complexity of the model in the same way as the regularization parameter λ in eq. 8.29. By suppressing large regression coefficients this penalty limits the variance of the system at the expense of an increase in the bias of the derived coefficients.

A geometric interpretation of ridge regression is shown in figure 8.3. The solid elliptical contours are the likelihood surface for the regression with no regularization. The circle illustrates the constraint on the regression coefficients ($|\boldsymbol{\theta}|^2 < s$) imposed by the regularization. The penalty on the likelihood function, based on the squared norm of the regression coefficients, drives the solution to small values of $\boldsymbol{\theta}$. The smaller the value of s (or the larger the regularization parameter λ) the more the regression coefficients are driven toward zero.

The regularized regression coefficients can be derived through matrix inversion as before. Applying an SVD to the $N \times m$ design matrix (where m is the number of terms in the model; see § 8.2.2) we get $M = U\Sigma V^T$, with U an $N \times m$ matrix, V^T the $m \times m$ matrix of eigenvectors and Σ the $m \times m$ matrix of eigenvalues. We can now write the regularized regression coefficients as

$$\boldsymbol{\theta} = V\Sigma' U^T Y, \tag{8.33}$$

where Σ' is a diagonal matrix with elements $d_i/(d_i^2 + \lambda)$, with d_i the eigenvalues of MM^T.

As λ increases, the diagonal components are down weighted so that only those components with the highest eigenvalues will contribute to the regression. This relates directly to the PCA analysis we described in § 7.3. Projecting the variables onto the eigenvectors of MM^T such that

$$Z = MV, \tag{8.34}$$

with z_i the ith eigenvector of M, ridge regression shrinks the regression coefficients for any component for which its eigenvalues (and therefore the associated variance) are small.

The effective goodness of fit for a ridge regression can be derived from the response of the regression function,

$$\hat{y} = M(M^T M + \lambda I)^{-1} M^T y, \tag{8.35}$$

and the number of degrees of freedom,

$$\text{DOF} = \text{Trace}[M(M^T M + \lambda I)^{-1} M^T] = \sum_i \frac{d_i^2}{d_i^2 + \lambda}. \tag{8.36}$$

Ridge regression can be accomplished with the `Ridge` class in Scikit-learn:

```
import numpy as np
from sklearn.linear_model import Ridge

X = np.random.random((100, 10))
    # 100 points in 10 dims
y = np.dot(X, np.random.random(10))
    # random combination of X
model = Ridge(alpha = 0.05)  # alpha controls
        # regularization
model.fit(X, y)
y_pred = model.predict(X)
```

For more information, see the Scikit-learn documentation.

Figure 8.4 uses the Gaussian basis function regression of § 8.2.2 to illustrate how ridge regression will constrain the regression coefficients. The left panel shows the general linear regression for the supernovas (using 100 evenly spaced Gaussians with $\sigma = 0.2$). As we noted in § 8.2.2, an increase in the number of model parameters results in an overfitting of the data (the lower panel in figure 8.4 shows how the regression coefficients for this fit are on the order of 10^8). The central panel demonstrates how ridge regression (with $\lambda = 0.005$) suppresses the amplitudes of the regression coefficients and the resulting fluctuations in the modeled response.

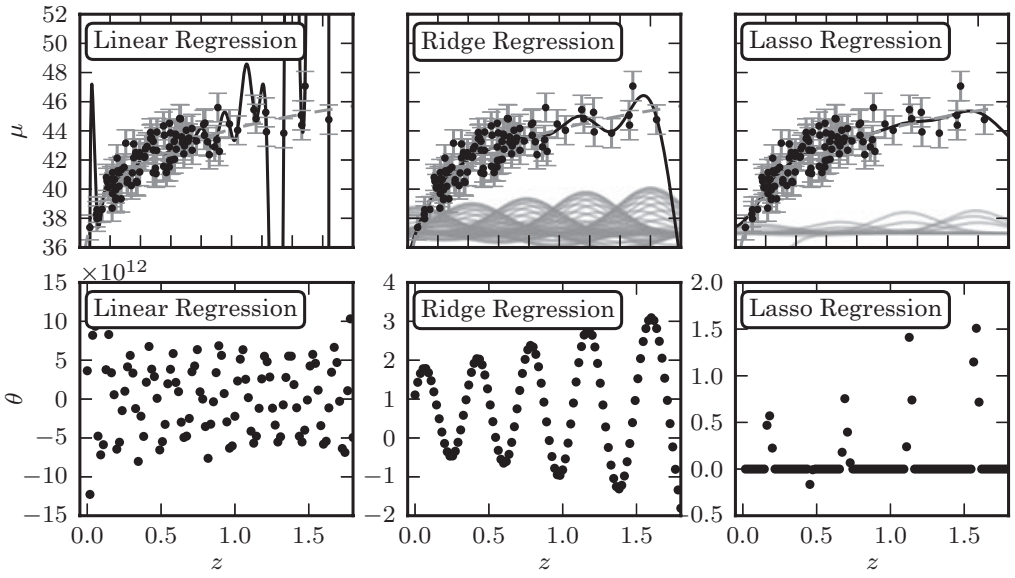

Figure 8.4. Regularized regression for the same sample as Fig 8.2. Here we use Gaussian basis function regression with a Gaussian of width $\sigma = 0.2$ centered at 100 regular intervals between $0 \leq z \leq 2$. The lower panels show the best-fit weights as a function of basis function position. The left column shows the results with no regularization: the basis function weights w are on the order of 10^8, and overfitting is evident. The middle column shows ridge regression (L_2 regularization) with $\lambda = 0.005$, and the right column shows LASSO regression (L_1 regularization) with $\lambda = 0.005$. All three methods are fit without the bias term (intercept).

8.3.2. LASSO Regression

Ridge regression uses the square of the regression coefficients to regularize the fits (i.e., the L_2 norm). A modification of this approach is to use the L_1 norm [2] to subset the variables within a model as well as applying shrinkage. This technique is known as LASSO (least absolute shrinkage and selection; see [21]). LASSO penalizes the likelihood as

$$(Y - M\boldsymbol{\theta})^T(Y - M\boldsymbol{\theta}) + \lambda|\boldsymbol{\theta}|, \tag{8.37}$$

where $|\boldsymbol{\theta}|$ penalizes the absolute value of $\boldsymbol{\theta}$. LASSO regularization is equivalent to least-squares regression with a penalty on the absolute value of the regression coefficients,

$$|\boldsymbol{\theta}| < s. \tag{8.38}$$

The most interesting aspect of LASSO is that it not only weights the regression coefficients, it also imposes sparsity on the regression model. Figure 8.3 illustrates the impact of the L_1 norm on the regression coefficients from a geometric perspective. The $\lambda|\boldsymbol{\theta}|$ penalty preferentially selects regions of likelihood space that coincide with one of the vertices within the region defined by the regularization. This corresponds to setting one (or more if we are working in higher dimensions) of the model attributes to zero. This subsetting of the model attributes reduces the underlying complexity of the model (i.e., we make zeroing of weights, or feature selection, more

aggressive). As λ increases, the size of the region encompassed within the constraint decreases.

Ridge regression can be accomplished with the `Lasso` class in Scikit-learn:

```
import numpy as np
from sklearn.linear_model import Lasso

X = np.random.random((100, 10))
    # 100 points in 10 dims
y = np.dot(X, np.random.random(10))
    # random comb. of X
model = Lasso(alpha = 0.05)  # alpha controls
        # regularization
model.fit(X, y)
y_pred = model.predict(X)
```

For more information, see the Scikit-learn documentation.

Figure 8.4 shows this effect for the supernova data. Of the 100 Gaussians in the input model, with $\lambda = 0.005$, only 13 are selected by LASSO (note the regression coefficients in the lower panel). This reduction in model complexity suppresses the overfitting of the data.

A disadvantage of LASSO is that, unlike ridge regression, there is no closed-form solution. The optimization becomes a quadratic programming problem (though it is still a convex optimization). There are a number of numerical techniques that have been developed to address these issues including coordinate-gradient descent [12] and least angle regression [5].

8.3.3. How Do We Choose the Regularization Parameter λ?

In each of the regularization examples above we defined a "shrinkage parameter" that we refer to as the regularization parameter. The natural question then is how do we set λ? So far we have only noted that as we increase λ we increase the constraints on the range regression coefficients (with $\lambda = 0$ returning the standard least-squares regression). We can, however, evaluate its impact on the regression as a function of its amplitude.

Applying the k-fold cross-validation techniques described in § 8.11 we can define an error (for a specified value of λ) as

$$\text{Error}(\lambda) = k^{-1} \sum_k N_k^{-1} \sum_i^{N_k} \frac{[y_i - f(x_i|\boldsymbol{\theta})]^2}{\sigma_i^2}, \tag{8.39}$$

where N_k^{-1} is the number of data points in the kth cross-validation sample, and the summation over N_k represents the sum of the squares of the residuals of the fit. Estimating λ is then simply a case of finding the λ that minimizes the cross-validation error.

8.4. Principal Component Regression

For the case of high-dimensional data or data sets where the variables are collinear, the relation between ridge regression and principal component analysis can be exploited to define a regression based on the principal components. For centered data (i.e., zero mean) we recall from § 7.3 that we can define the principal components of a system from the data covariance matrix, $X^T X$, by applying an eigenvalue decomposition (EVD) or singular value decomposition (SVD),

$$X^T X = V \Sigma V^T, \tag{8.40}$$

with V^T the eigenvectors and Σ the eigenvalues.

Projecting the data matrix onto these eigenvectors we define a set of projected data points,

$$Z = X V^T, \tag{8.41}$$

and truncate this expansion to exclude those components with small eigenvalues. A standard linear regression can now be applied to the data transposed to this principal component space with

$$Y = M_z \boldsymbol{\theta} + \epsilon, \tag{8.42}$$

with M_z the design matrix for the projected components z_i. The PCA analysis (including truncation) and the regression are undertaken as separate steps in this procedure. The distinction between principal component regression (PCR) and ridge regression is that the number of model components in PCR is ordered by their eigenvalues and is absolute (i.e., we weight the regression coefficients by 1 or 0). For ridge regression the weighting of the regression coefficients is continuous.

The advantages of PCR over ridge regression arise primarily for data containing independent variables that are collinear (i.e., where the correlation between the variables is almost unity). For these cases, the regression coefficients have large variance and their solutions can become unstable. Excluding those principal components with small eigenvalues alleviates this issue. At what level to truncate the set of eigenvectors is, however, an open question (see § 7.3.2). A simple approach to take is to truncate based on the eigenvalue with common cutoffs ranging between 1% and 10% of the average eigenvalue (see § 8.2 of [10] for a detailed discussion of truncation levels for PCR). The disadvantage of such an approach is that an eigenvalue does not always correlate with the ability of a given principal component to predict the dependent variable. Other techniques, including cross-validation [17], have been proposed yet there is no well-adopted solution to this problem.

Finally, we note that in the case of ill-conditioned regression problems (e.g., those with collinear data), most implementations of linear regression will implicitly perform a form of principal component regression when inverting the singular matrix $M^T M$. This comes through the use of the robust pseudoinverse, which truncates small singular values to prevent numerical overflow in ill-conditioned problems.

8.5. Kernel Regression

The previous sections found the regression or objective function that "best fits" a set of data assuming a linear model. Before we address the question of nonlinear optimization we will describe a number of techniques that make use of locality within the data (i.e., local regression techniques).

Kernel or Nadaraya–Watson [18, 23] regression defines a kernel, $K(x_i, x)$, local to each data point, with the amplitude of the kernel depending only on the distance from the local point to all other points in the sample. The properties of the kernel are such that it is positive for all values and asymptotes to zero as the distance approaches infinity. The influence of the kernel (i.e., the region of parameter space over which we weight the data) is determined by its width or bandwidth, h. Common forms of the kernel include the top-hat function, and the Gaussian distribution.

The Nadaraya–Watson estimate of the regression function is given by

$$f(x|K) = \frac{\sum_{i=1}^{N} K\left(\frac{||x_i - x||}{h}\right) y_i}{\sum_{i=1}^{N} K\left(\frac{||x_i - x||}{h}\right)}, \tag{8.43}$$

which can be viewed as taking a weighted average of the dependent variable, y. This gives higher weight to points near x with a weighting function,

$$w_i(x) = \frac{K\left(\frac{||x_i - x||}{h}\right)}{\sum_{i=1}^{N} K\left(\frac{||x_i - x||}{h}\right)}. \tag{8.44}$$

Nadaraya–Watson kernel regression can be performed using AstroML in the following way:

```
import numpy as np
from astroML.linear_model import NadarayaWatson

X = np.random.random((100, 2))# 100 points in 2 dims
y = X[:, 0] + X[:, 1]
model = NadarayaWatson('gaussian', 0.05)
model.fit(X, y)
y_pred = model.predict(X)
```

Figure 8.2 shows the application of Gaussian kernel regression to the synthetic supernova data compared to the standard linear regression techniques introduced in § 8.2. For this example we use a Gaussian kernel with $h = 0.1$ that is constant across the redshift interval. At high redshift, where the data provide limited support for the model, we see that the weight function drives the predicted value of y to that of the nearest neighbor. This prevents the extrapolation problems common when fitting polynomials that are not constrained at the edges of the data (as we saw in the top panels of figure 8.2). The width of the kernel acts as a smoothing function.

At low redshift, the increase in the density of points at $z \approx 0.25$ biases the weighted estimate of $\hat{y}$ for $z < 0.25$. This is because, while the kernel amplitude is small for the higher redshift points, the increase in the sample size results in a higher than average weighting of points at $z \approx 0.25$. Varying the bandwidth as a function of the independent variable can correct for this. The rule of thumb for kernel-based regression (as with kernel density estimation described in § 6.3) is that the bandwidth is more important that the exact shape of the kernel.

Estimation of the optimal bandwidth of the kernel is straightforward using cross-validation (see § 8.11). We define the CV error as

$$\mathrm{CV}_{L_2}(h) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - f\left(x_i|K\left(\frac{||x_i - x_j||}{h}\right)\right)\right)^2. \tag{8.45}$$

We actually do not need to compute separate estimates for each leave-one-out cross-validation subsample if we rewrite this as

$$\mathrm{CV}_{L_2}(h) = \frac{1}{N}\frac{\sum_{i=1}^{N}(y_i - f(x_i|K))^2}{\left(1 - \frac{K(0)}{\sum_{j=1}^{N}K\left(\frac{||x_i - x_j||}{h}\right)}\right)^2}. \tag{8.46}$$

It can be shown that, as for kernel density estimation, the optimal bandwidth decreases with sample size at a rate of $N^{-1/5}$.

8.6. Locally Linear Regression

Related to kernel regression is *locally linear regression*, where we solve a separate weighted least-squares problem at each point x, finding the $w(x)$ which minimizes

$$\sum_{i=1}^{N} K\left(\frac{||x - x_i||}{h}\right)(y_i - w(x)\,x_i)^2. \tag{8.47}$$

The assumption for locally linear regression is that the regression function can be approximated by a Taylor series expansion about any local point. If we truncated this expansion at the first term (i.e., a locally constant solution) we recover kernel regression. For locally linear regression the function estimate is

$$\begin{aligned} f(x|K) &= \theta(x)x && (8.48)\\ &= x^T\left(\mathbf{X}^T W(x)\mathbf{X}\right)^{-1}\mathbf{X}^T W(x)\mathbf{Y} && (8.49)\\ &= \sum_{i=1}^{N} w_i(x)y_i, && (8.50) \end{aligned}$$

where $W(x)$ is an $N \times N$ diagonal matrix with the ith diagonal element given by $K||x_i - x||/h$.

A common form for $K(x)$ is the tricubic kernel,

$$K(x_i, x) = \left(1 - \left(\frac{|x - x_i|}{h}\right)^3\right)^3 \tag{8.51}$$

for $|x_i - x| < h$, which is often referred to as lowess (or loess; locally weighted scatter plot smoothing); see [3].

There are further extensions possible for local linear regression:

- *Local polynomial regression*. We can consider any polynomial order. However there is a bias–variance (complexity) trade-off, as usual. The general consensus is that going past linear increases variance without decreasing bias much, since local linear regression captures most of the boundary bias.
- *Variable-bandwidth kernels*. Let the bandwidth for each training point be inversely proportional to its kth nearest neighbor's distance. Generally a good idea in practice, though there is less theoretical consensus on how to choose the parameters in this framework.

None of these modifications improves the convergence rate.

8.7. Nonlinear Regression

Forcing data to correspond to a linear model through the use of coordinate transformations is a well-used trick in astronomy (e.g., the extensive use of logarithms in the astronomical literature to linearize complex relations between attributes; fitting $y = A\exp(Bx)$ becomes a linear problem with $z = K + Bx$, where $z = \log y$ and $K = \log A$). These simplifications, while often effective, introduce other complications (including the non-Gaussian nature of the uncertainties for low signal-to-noise data). We must, eventually, consider the case of nonlinear regression and model fitting.

In the cosmological examples described previously we have fit a series of parametric and nonparametric models to the supernova data. Given that we know the theoretical form of the underlying cosmological model, these models are somewhat ad hoc (e.g., using a series of polynomials to parameterize the dependence of distance modulus on cosmological redshift). In the following we consider directly fitting the cosmological model described in eq. 8.4. Solving for Ω_m and Ω_Λ is a nonlinear optimization problem requiring that we maximize the posterior,

$$p(\Omega_m, \Omega_\Lambda | z, I) \propto \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(\frac{-(\mu_i - \mu(z_i|\Omega_m, \Omega_\Lambda))^2}{2\sigma_i^2}\right) p(\Omega_m, \Omega_\Lambda) \tag{8.52}$$

with μ_i the distance modulus for the supernova and z_i the redshift.

In § 5.8 we introduced Markov chain Monte Carlo as a sampling technique that can be used for searching through parameter space. Figure 8.5 shows the resulting likelihood contours for our cosmological model after applying the Metropolis–Hastings algorithm to generate the MCMC chains and integrating the chains over the parameter space.

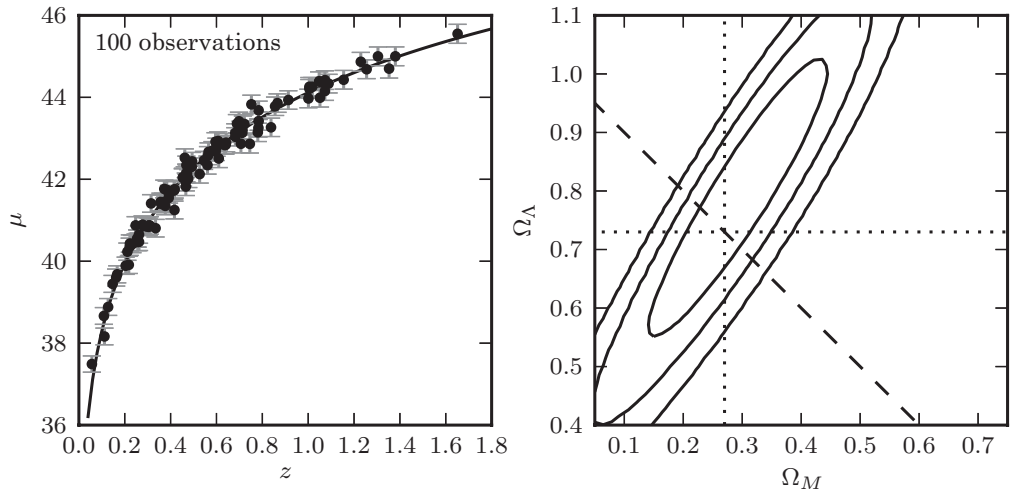

Figure 8.5. Cosmology fit to the standard cosmological integral. Errors in μ are a factor of ten smaller than for the sample used in figure 8.2. Contours are 1σ, 2σ, and 3σ for the posterior (uniform prior in Ω_M and Ω_Λ). The dashed line shows flat cosmology. The dotted lines show the input values.

An alternate approach is to use the Levenberg–Marquardt (LM) algorithm [15, 16] to optimize the maximum likelihood estimation. LM searches for the sum-of-squares minima of a multivariate distribution through a combination of gradient descent and Gauss–Newton optimization. Assuming that we can express our regression function as a Taylor series expansion then, to first order, we can write

$$f(x_i|\boldsymbol{\theta}) = f(x_i|\boldsymbol{\theta}_0) + J\, d\boldsymbol{\theta}, \tag{8.53}$$

where $\boldsymbol{\theta}_0$ is an initial guess for the regression parameters, J is the Jacobian about this point ($J = \partial f(x_i|\boldsymbol{\theta})/\partial\boldsymbol{\theta}$), and $d\boldsymbol{\theta}$ is a small change in the regression parameters. LM minimizes the sum of square errors,

$$\sum_i (y_i - f(x_i|\boldsymbol{\theta}_0) - J_i d\boldsymbol{\theta})^2 \tag{8.54}$$

for the perturbation $d\boldsymbol{\theta}$. This results in an update relation for $d\boldsymbol{\theta}$ of

$$(J^T C^{-1} J + \lambda\, \mathrm{diag}(J^T C^{-1} J))\, d\boldsymbol{\theta} = J^T C^{-1}(Y - f(X|\boldsymbol{\theta})), \tag{8.55}$$

with C the standard covariance matrix introduced in eq. 8.18. In this expression the λ term acts as a damping parameter (in a manner similar to ridge regression regularization discussed in § 8.3.1). For small λ the relation approximates a Gauss–Newton method (i.e., it minimizes the parameters assuming the function is quadratic). For large λ the perturbation $d\boldsymbol{\theta}$ follows the direction of steepest descent. The $\mathrm{diag}(J^T C^{-1} J)$ term, as opposed to the identity matrix used in ridge regression, ensures that the update of $d\boldsymbol{\theta}$ is largest along directions where the gradient is smallest (which improves convergence).

LM is an iterative process. At each iteration LM searches for the step $d\boldsymbol{\theta}$ that minimizes eq. 8.54 and then updates the regression model. The iterations cease when

the step size, or change in likelihood values reaches a predefined value. Throughout the iteration process the damping parameter, λ, is adaptively varied (decreasing as the minimum is approached). A common approach involves decreasing (or increasing) λ by ν^k, where k is the number of the iteration and ν is a damping factor (with $\nu > 1$). Upon successful convergence the regression parameter covariances are given by $[J^T \boldsymbol{\theta} J]^{-1}$. It should be noted that the success of the LM algorithm often relies on the initial guesses for the regression parameters being close to the maximum likelihood solution (in the presence of nonlocal minima), or the likelihood function surface being unimodal.

The submodule `scipy.optimize` includes several routines for optimizing linear and nonlinear equations. The Levenberg–Marquardt algorithm is used in `scipy.optimize.leastsq`. Below is a brief example of using the routine to estimate the first six terms of the Taylor series for the function $y = \sin x \approx \sum_n a_n x^n$:

```
import numpy as np
from scipy import optimize
x = np.linspace(-3, 3, 100) # 100 values between -3
                            # and 3
def taylor_err(a, x, f):
    p = np.arange(len(a))[:, np.newaxis]
        # column vector
    return f(x) - np.dot(a, x ** p)
a_start = np.zeros(6)  # starting guess
a_best, flag = optimize.leastsq(taylor_err, a_start,
                                args=(x, np.sin))
```

8.8. Uncertainties in the Data

In the opening section we introduced the problem of regression in its most general form. Computational complexity (particularly for the case of multivariate data) led us through a series of approximations that can be used in optimizing the likelihood and the prior (e.g., that the uncertainties have a Gaussian distribution, that the independent variables are error-free, that we can control complexity of the model, and that we can express the likelihood in terms of linear functions). Eventually we have to face the problem that many of these assumptions no longer hold for data in the wild. We have addressed the question of model complexity; working from linear to nonlinear regression. We now return to the question of *error behavior* and consider the uncertainty that is inherent in any analysis.

8.8.1. Uncertainties in the Dependent and Independent Axes

In almost all real-world applications, the assumption that one variable (the independent variable) is essentially free from any uncertainty is not valid. Both the dependent and independent variables will have measurement uncertainties. For our example data set we have assumed that the redshifts of the supernovas are known to a very high level of accuracy (i.e., that we measure these redshifts spectroscopically). If, for example, the redshifts of the supernovas were estimated based on the colors of the supernova (or host galaxy) then the errors on the redshift estimate can be significant. For our synthetic data we assume fractional uncertainties on the independent variable of 10%.

The impact of errors on the "independent" variables is a bias in the derived regression coefficients. This is straightforward to show if we consider a linear model with a dependent and independent variable, y^* and x^*. We can write the objective function as before,

$$y_i^* = \theta_0 + \theta_1 x_i^*. \tag{8.56}$$

Now let us assume that we observe y and x, which are noisy representations of y^* and x^*, i.e.,

$$x_i = x_i^* + \delta_i, \tag{8.57}$$
$$y_i = y^* + \epsilon_i, \tag{8.58}$$

with δ and ϵ centered normal distributions.

Solving for y we get

$$y = \theta_0 + \theta_1(x_i - \delta_i) + \epsilon_i. \tag{8.59}$$

The uncertainty in x is now part of the regression equation and scales with the regression coefficients (biasing the regression coefficient). This problem is known in the statistics literature as *total least squares* and belongs to the class of "errors-in-variables" problems; see [4]. A very detailed discussion of regression and the problem of uncertainties from an astronomical perspective can be found in [8, 11].

How can we account for the measurement uncertainties in both the independent and dependent variables? Let us start with a simple example where we assume the errors are Gaussian so we can write the covariance matrix as

$$\Sigma_i = \begin{bmatrix} \sigma_{x_i}^2 & \sigma_{xy_i} \\ \sigma_{xy_i} & \sigma_{y_i}^2 \end{bmatrix}. \tag{8.60}$$

For a straight-line regression we express the slope of the line, θ_1, in terms of its normal vector,

$$\mathbf{n} = \begin{bmatrix} -\sin\alpha \\ \cos\alpha \end{bmatrix}, \tag{8.61}$$

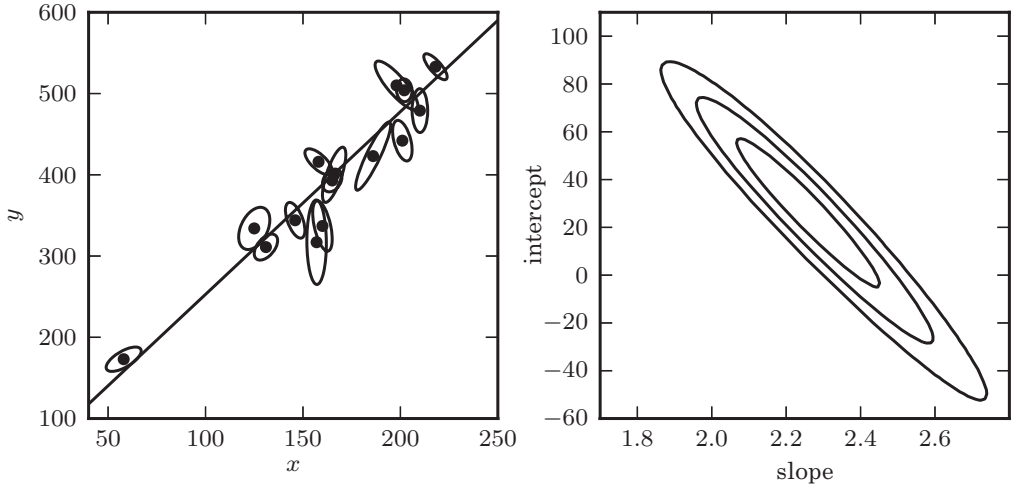

Figure 8.6. A linear fit to data with correlated errors in x and y. In the literature, this is often referred to as *total least squares* or *errors-in-variables* fitting. The left panel shows the lines of best fit; the right panel shows the likelihood contours in slope/intercept space. The points are the same set used for the examples in [8].

where $\theta_1 = \arctan(\alpha)$ and α is the angle between the line and the x-axis. The covariance matrix projects onto this space as

$$S_i^2 = \mathbf{n}^T \Sigma_i \mathbf{n} \tag{8.62}$$

and the distance between a point and the line is given by (see [8])

$$\Delta_i = \mathbf{n}^T z_i - \theta_0 \cos\alpha, \tag{8.63}$$

where z_i represents the data point (x_i, y_i). The log-likelihood is then

$$\ln \mathrm{L} = -\sum_i \frac{\Delta_i^2}{2S_i^2}. \tag{8.64}$$

Maximizing this likelihood for the regression parameters, θ_0 and θ_1 is shown in figure 8.6, where we use the data from [8] with correlated uncertainties on the x and y components, and recover the underlying linear relation. For a single parameter search (θ_1) the regression can be undertaken in a brute-force manner. As we increase the complexity of the model or the dimensionality of the data, the computational cost will grow and techniques such as MCMC must be employed (see [4]).

8.9. Regression That Is Robust to Outliers

A fact of experimental life is that if you can measure an attribute you can also measure it incorrectly. Despite the increase in fidelity of survey data sets, any regression or model fitting must be able to account for outliers from the fit. For the standard least-squares regression the use of an L_2 norm results in outliers that have substantial leverage in any fit (contributing as the square of the systematic deviation). If we knew

$e(y_i|y)$ for all of the points in our sample (e.g., they are described by an exponential distribution where we would use the L_1 norm to define the error) then we would simply include the error distribution when defining the likelihood. When we do not have a priori knowledge of $e(y_i|y)$, things become more difficult. We can either model $e(y_i|y)$ as a mixture model (see § 5.6.7) or assume a form for $e(y_i|y)$ that is less sensitive to outliers. An example of the latter would be the adoption of the L_1 norm, $\sum_i ||y_i - w_i x_i||$, which we introduced in § 8.3, which is less sensitive to outliers than the L_2 norm (and was, in fact, proposed by Rudjer Bošković prior to the development of least-squares regression by Legendre, Gauss, and others [2]). Minimizing the L_1 norm is essentially finding the median. The drawback of this least absolute value regression is that there is no closed-form solution and we must minimize the likelihood space using an iterative approach.

Other approaches to robust regression adopt an approach that seeks to reject outliers. In the astronomical community this is usually referred to as "sigma clipping" and is undertaken in an iterative manner by progressively pruning data points that are not well represented by the model. Least-trimmed squares formalizes this, somewhat ad hoc approach, by searching for the subset of K points which minimize $\sum_i^K (y_i - \theta_i x_i)^2$. For large N the number of combinations makes this search expensive.

Complementary to outlier rejection are the Theil–Sen [20] or the Kendall robust line-fit method and associated techniques. In these cases the regression is determined from the median of the slope, θ_1, calculated from all pairs of points within the data set. Given the slope, the offset or zero point, θ_0, can be defined from the median of $y_i - \theta_1 x_i$. Each of these techniques is simple to estimate and scales to large numbers.

M estimators (M stands for "maximum-likelihood-type") approach the problem of outliers by modifying the underlying likelihood estimator to be less sensitive than the classic L_2 norm. M estimators are a class of estimators that include many maximum-likelihood approaches (including least squares). They replace the standard least squares, which minimizes the sum of the squares of the residuals between a data value and the model, with a different function. Ideally the M estimator has the property that it increases less than the square of the residual and has a unique minimum at zero.

Huber loss function

An example of an M estimator that is common in robust regression is that of the Huber loss (or cost) function [9]. The Huber estimator minimizes

$$\sum_{i=1}^{N} e(y_i|y), \tag{8.65}$$

where $e(y_i|y)$ is modeled as

$$\phi(t) = \begin{cases} \frac{1}{2}t^2 & \text{if } |t| \leq c, \\ c|t| - \frac{1}{2}c^2 & \text{if } |t| \geq c, \end{cases} \tag{8.66}$$

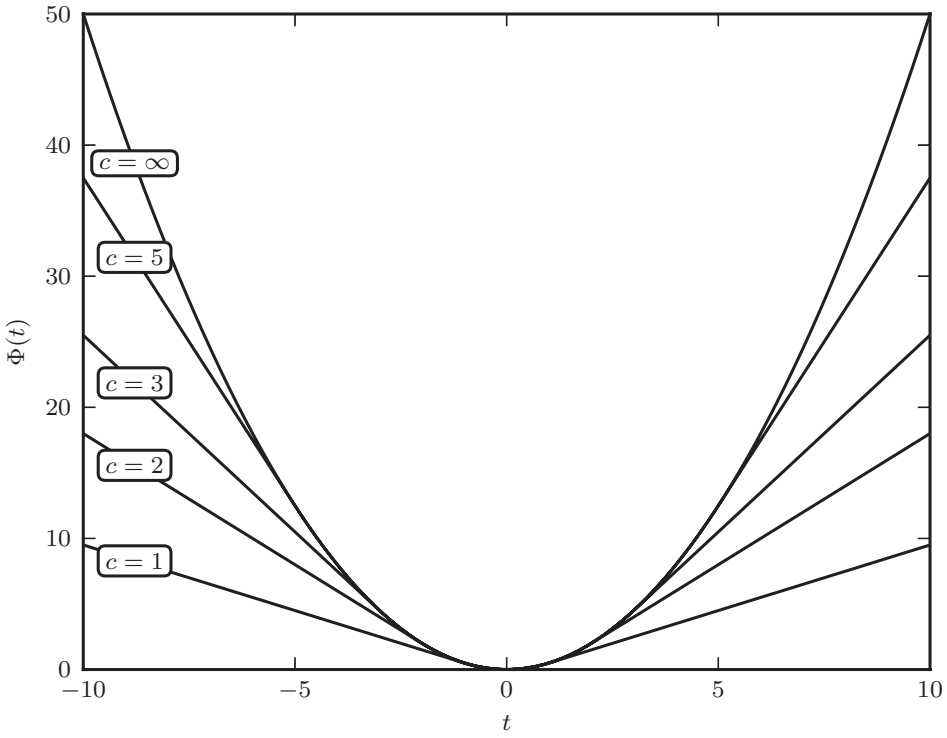

Figure 8.7. The Huber loss function for various values of c.

and $t = y_i - y$ with a constant c that must be chosen. Therefore, $e(t)$ is a function which acts like t^2 for $|t| \leq c$ and like $|t|$ for $|t| > c$ and is continuous and differentiable (see figure 8.7). The transition in the Huber function is equivalent to assuming a Gaussian error distribution for small excursions from the true value of the function and an exponential distribution for large excursions (its behavior is a compromise between the mean and the median). Figure 8.8 shows an application of the Huber loss function to data with outliers. Outliers do have a small effect, and the slope of the Huber loss fit is closer to that of standard linear regression.

8.9.1. Bayesian Outlier Methods

From a Bayesian perspective, one can use the techniques developed in chapter 5 within the context of a regression model in order to account for, and even to individually identify outliers (recall § 5.6.7). Figure 8.9 again shows the data set used in figure 8.8, which contains three clear outliers. In a standard straight-line fit to the data, the result is strongly affected by these points. Though this standard linear regression problem is solvable in closed form (as it is in figure 8.8), here we compute the best-fit slope and intercept using MCMC sampling (and show the resulting contours in the upper-right panel).

The remaining two panels show two different Bayesian strategies for accounting for outliers. The main idea is to enhance the model such that it can naturally explain the presence of outliers. In the first model, we account for the outliers through the use of a mixture model, adding a background Gaussian component to our data. This is the regression analog of the model explored in § 5.6.5, with the difference that here we are modeling the background as a wide Gaussian rather than a uniform distribution.

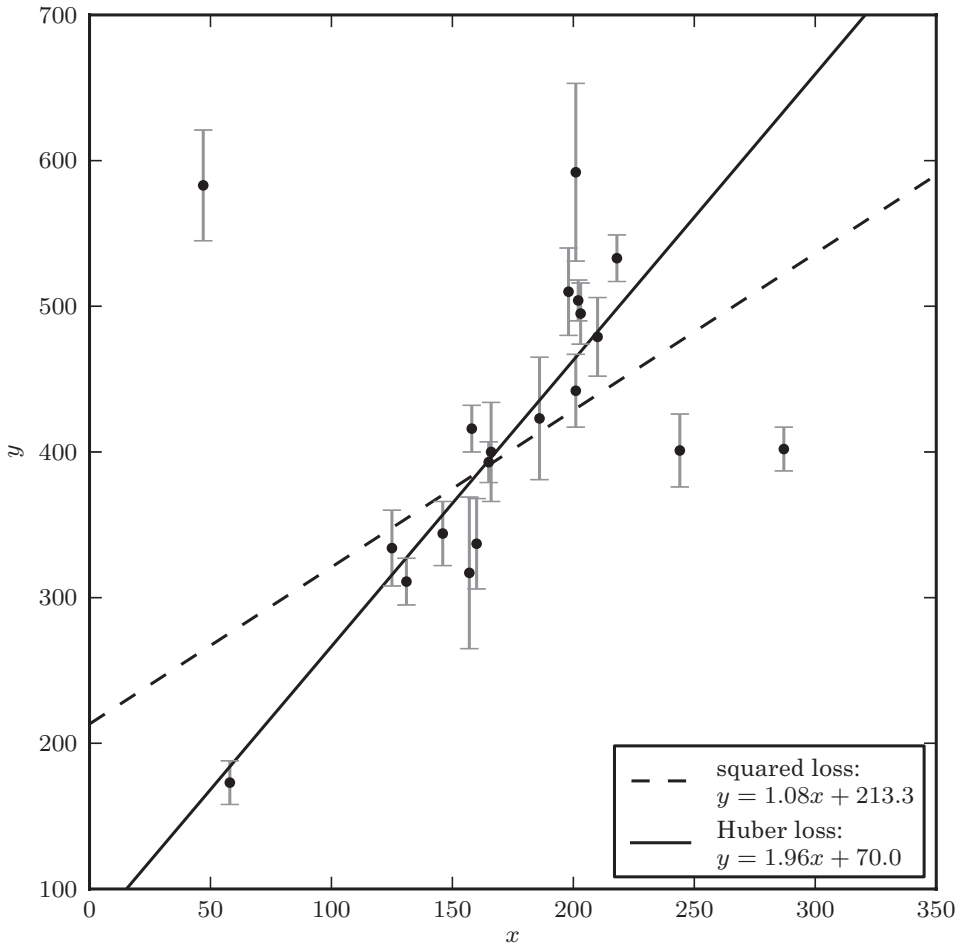

Figure 8.8. An example of fitting a simple linear model to data which includes outliers (data is from table 1 of [8]). A comparison of linear regression using the squared-loss function (equivalent to ordinary least-squares regression) and the Huber loss function, with $c = 1$ (i.e., beyond 1 standard deviation, the loss becomes linear).

The mixture model includes three additional parameters: μ_b and V_b, the mean and variance of the background, and p_b, the probability that any point is an outlier. With this model, the likelihood becomes (cf. eq. 5.83; see also [8])

$$p(\{y_i\}|\{x_i\}, \{\sigma_i\}, \theta_0, \theta_1, \mu_b, V_b, p_b) \propto \prod_{i=1}^{N} \left[\frac{1 - p_b}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(y_i - \theta_1 x_i - \theta_0)^2}{2\sigma_i^2} \right) + \frac{p_b}{\sqrt{2\pi(V_b + \sigma_i^2)}} \exp\left(-\frac{(y_i - \mu_b)^2}{2(V_b + \sigma_i^2)} \right) \right]. \tag{8.67}$$

Using MCMC sampling and marginalizing over the background parameters yields the dashed-line fit in figure 8.9. The marginalized posterior for this model is shown in the lower-left panel. This fit is much less affected by the outliers than is the simple regression model used above.

Finally, we can go further and perform an analysis analogous to that of § 5.6.7, in which we attempt to identify bad points individually. In analogy with eq. 5.94 we

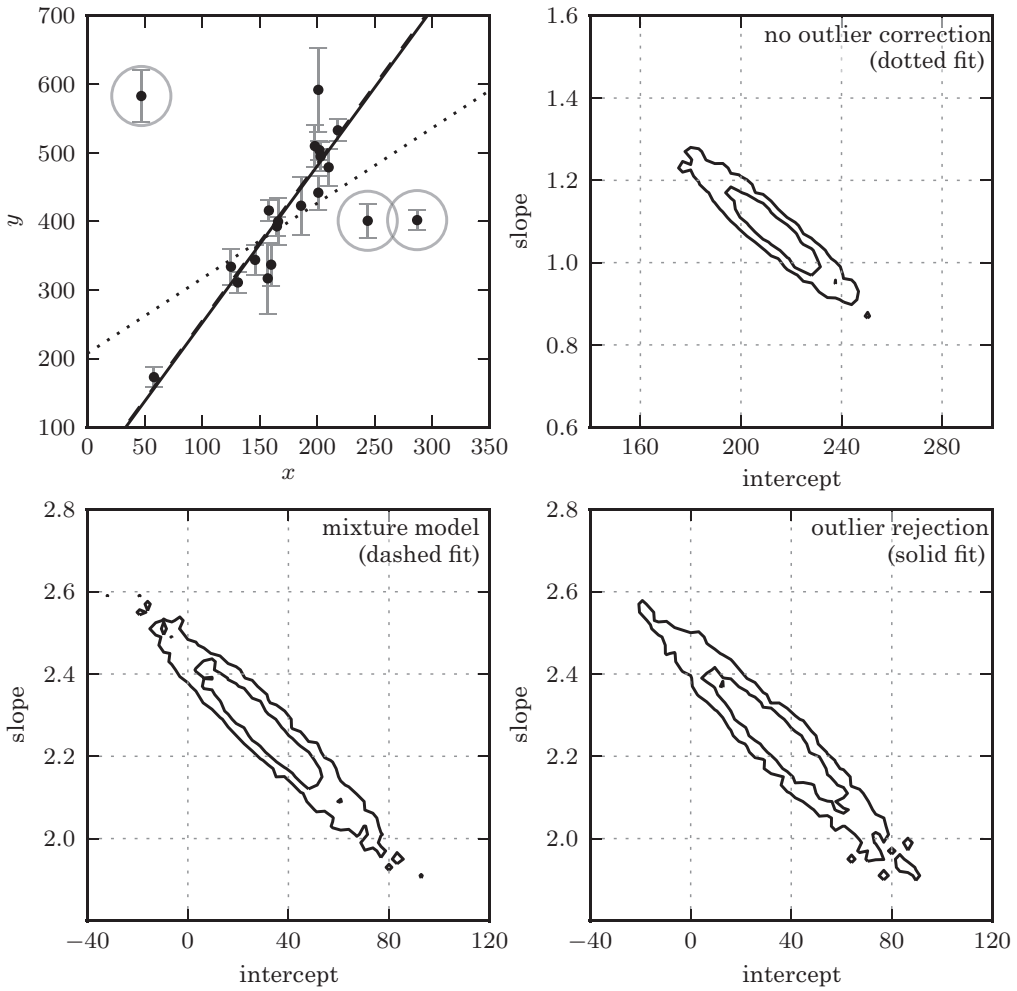

Figure 8.9. Bayesian outlier detection for the same data as shown in figure 8.8. The top-left panel shows the data, with the fits from each model. The top-right panel shows the 1σ and 2σ contours for the slope and intercept with no outlier correction: the resulting fit (shown by the dotted line) is clearly highly affected by the presence of outliers. The bottom-left panel shows the marginalized 1σ and 2σ contours for a mixture model (eq. 8.67). The bottom-right panel shows the marginalized 1σ and 2σ contours for a model in which points are identified individually as "good" or "bad" (eq. 8.68). The points which are identified by this method as bad with a probability greater than 68% are circled in the first panel.

can fit for nuisance parameters g_i, such that if $g_i = 1$, the point is a "good" point, and if $g_i = 0$ the point is a "bad" point. With this addition our model becomes

$$p(\{y_i\}|\{x_i\},\{\sigma_i\},\{g_i\},\theta_0,\theta_1,\mu_b,V_b) \propto \prod_{i=1}^{N}\left[\frac{g_i}{\sqrt{2\pi\sigma_i^2}}\exp\left(-\frac{(y_i-\theta_1 x_i-\theta_0)^2}{2\sigma_i^2}\right)\right.$$
$$\left.+\frac{1-g_i}{\sqrt{2\pi(V_b+\sigma_i^2)}}\exp\left(-\frac{(y_i-\mu_b)^2}{2(V_b+\sigma_i^2)}\right)\right]. \tag{8.68}$$

This model is very powerful: by marginalizing over all parameters but a particular g_i, we obtain a posterior estimate of whether point i is an outlier. Using this procedure,

the "bad" points have been marked with a circle in the upper-left panel of figure 8.9. If instead we marginalize over the nuisance parameters, we can compute a two-dimensional posterior for the slope and intercept of the straight-line fit. The lower-right panel shows this posterior, after marginalizing over $\{g_i\}$, μ_b, and V_b. In this example, the result is largely consistent with the simpler Bayesian mixture model used above.

8.10. Gaussian Process Regression

Another powerful class of regression algorithms is Gaussian process regression. Despite its name, Gaussian process regression is widely applicable to data that are not generated by a Gaussian process, and can lead to very flexible regression models that are more data driven than other parametric approaches. There is a rich literature on the subject, and we will only give a cursory treatment here. An excellent book-length treatment of Gaussian processes can be found in [19].

A Gaussian process is a collection of random variables in parameter space, any subset of which is defined by a joint Gaussian distribution. It can be shown that a Gaussian process can be completely specified by its mean and covariance function. Gaussian processes can be defined in any number of dimensions for any positive covariance function. For simplicity, in this section we will consider one dimension and use a familiar squared-exponential covariance function,

$$\mathrm{Cov}(x_1, x_2; h) = \exp\left(\frac{-(x_1 - x_2)^2}{2h^2}\right), \tag{8.69}$$

where h is the bandwidth. Given a chosen value of h, this covariance function specifies the statistics of an infinite set of possible functions $f(x)$. The upper-left panel of figure 8.10 shows three of the possible functions drawn from a zero-mean Gaussian process[3] with $h = 1.0$. This becomes more interesting when we specify constraints on the Gaussian process: that is, we select only those functions $f(x)$ which pass through particular points in the space. The remaining panels of figure 8.10 show this Gaussian process constrained by points without error (upper-right panel), points with error (lower-left panel), and a set of 20 noisy observations drawn from the function $f_{\mathrm{true}}(x) = \cos x$. In each, the shaded region shows the 2σ contour in which 95% of all possible functions $f(x)$ lie.

These constrained Gaussian process examples hint at how these ideas could be used for standard regression tasks. The Gaussian process regression problem can be formulated similarly to the other regression problems discussed above. We assume our data is drawn from an underlying model $f(x)$: that is, our observed data is $\{x_i, y_i = f(x_i) + \sigma_i\}$. Given the observed data, we desire an estimate of the mean value $\bar{f}_j^*$ and variance Σ_{jk}^* for a new set of measurements x_j^*. In Bayesian terms, we want to compute the posterior pdf

$$p(f_j | \{x_i, y_i, \sigma_i\}, x_j^*). \tag{8.70}$$

[3]These functions are drawn by explicitly creating the $N \times N$ covariance matrix C from the functional form in eq. 8.69, and drawing N correlated Gaussian random variables with mean 0 and covariance C.

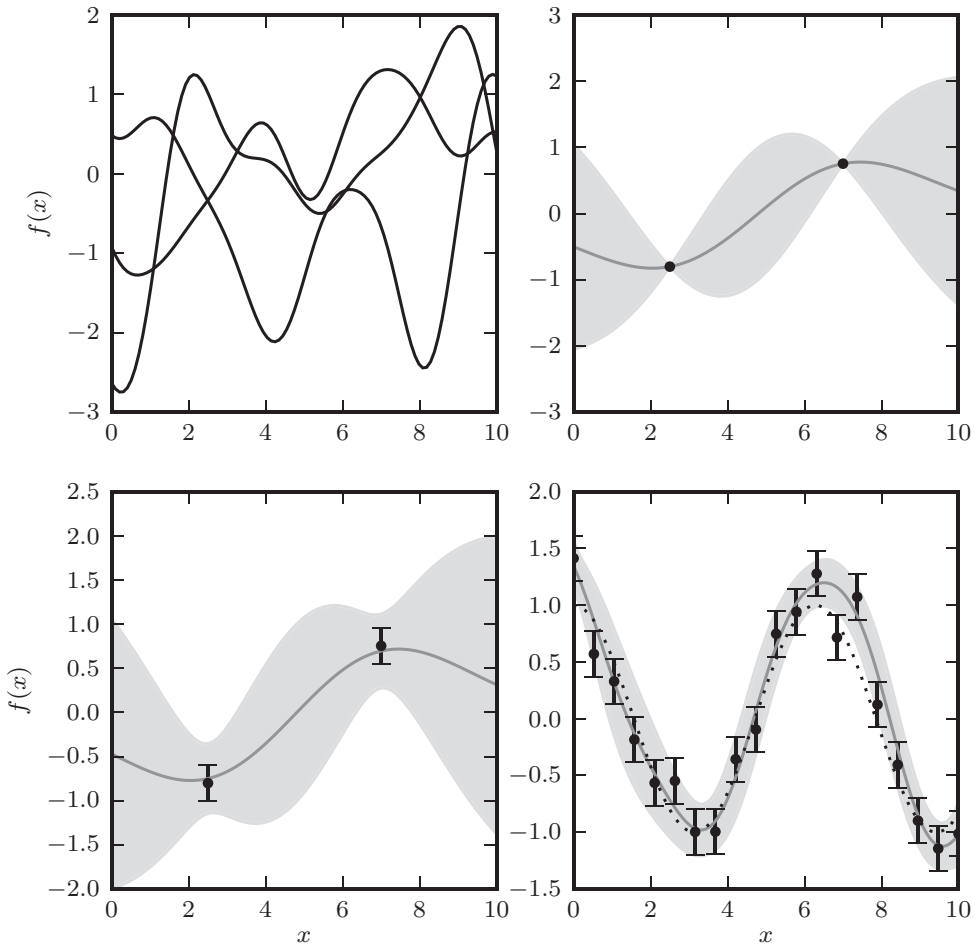

Figure 8.10. An example of Gaussian process regression. The upper-left panel shows three functions drawn from an unconstrained Gaussian process with squared-exponential covariance of bandwidth $h = 1.0$. The upper-right panel adds two constraints, and shows the 2σ contours of the constrained function space. The lower-left panel shows the function space constrained by the points with error bars. The lower-right panel shows the function space constrained by 20 noisy points drawn from $f(x) = \cos x$.

This amounts to averaging over the *entire set* of possible functions $f(x)$ which pass through our constraints. This seemingly infinite calculation can be made tractable using a "kernel trick," as outlined in [19], transforming from the infinite function space to a finite covariance space. The result of this mathematical exercise is that the Gaussian process regression problem can be formulated as follows.

Using the assumed form of the covariance function (eq. 8.69), we compute the covariance matrix

$$K = \begin{pmatrix} K_{11} & K_{12} \\ K_{12}^T & K_{22} \end{pmatrix}, \tag{8.71}$$

where K_{11} is the covariance between the input points x_i with observational errors σ_i^2 added in quadrature to the diagonal, K_{12} is the cross-covariance between the input points x_i and the unknown points x_j^*, and K_{22} is the covariance between the

unknown points x_j^*. Then for observed vectors $\mathbf{x}$ and $\mathbf{y}$, and a vector of unknown points $\mathbf{x}^*$, it can be shown that the posterior in eq. 8.70 is given by

$$p(f_j|\{x_i, y_i, \sigma_i\}, x_j^*) = \mathcal{N}(\mu, \Sigma), \tag{8.72}$$

where

$$\mu = K_{12}K_{11}^{-1}\mathbf{y}, \tag{8.73}$$

$$\Sigma = K_{22} - K_{12}^T K_{11}^{-1} K_{12}. \tag{8.74}$$

Then μ_j gives the expected value $\tilde{f}_j^*$ of the result, and Σ_{jk} gives the error covariance between any two unknown points. Note that the physics of the underlying process enters through the assumed form of the covariance function (e.g., eq. 8.69; for an analysis of several plausible covariance functions in the astronomical context of quasar variability; see [24]).

In figure 8.11, we show a Gaussian process regression analysis of the supernova data set used above. The model is very well constrained near $z = 0.6$, where there is a lot of data, but not well constrained at higher redshifts. This is an important feature of Gaussian process regression: the analysis produces not only a best-fit model, but an uncertainty at each point, as well as a full covariance estimate of the result at unknown points.

A powerful and general framework for Gaussian process regression is available in Scikit-learn. It can be used as follows:

```
import numpy as np
from sklearn.gaussian_process import GaussianProcess

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = np.sin(10 * X[:, 0] + X[:, 1])
gp = GaussianProcess(corr='squared_exponential')
gp.fit(X, y)
y_pred, dy_pred = gp.predict(X, eval_MSE=True)
```

The `predict` method can optionally return the mean square error, which is the diagonal of the covariance matrix of the fit. For more details, see the source code of the figures in this chapter, as well as the Scikit-learn documentation.

The results shown in figure 8.11 depend on correctly tuning the correlation function bandwidth h. If h is too large or too small, the particular Gaussian process model will be unsuitable for the data and the results will be biased. Figure 8.11 uses a simple cross-validation approach to decide on the best value of h: we will discuss cross-validation in the following section. Sometimes, the value of h is scientifically an interesting outcome of the analysis; such an example is discussed in the context of the damped random walk model in § 10.5.4.

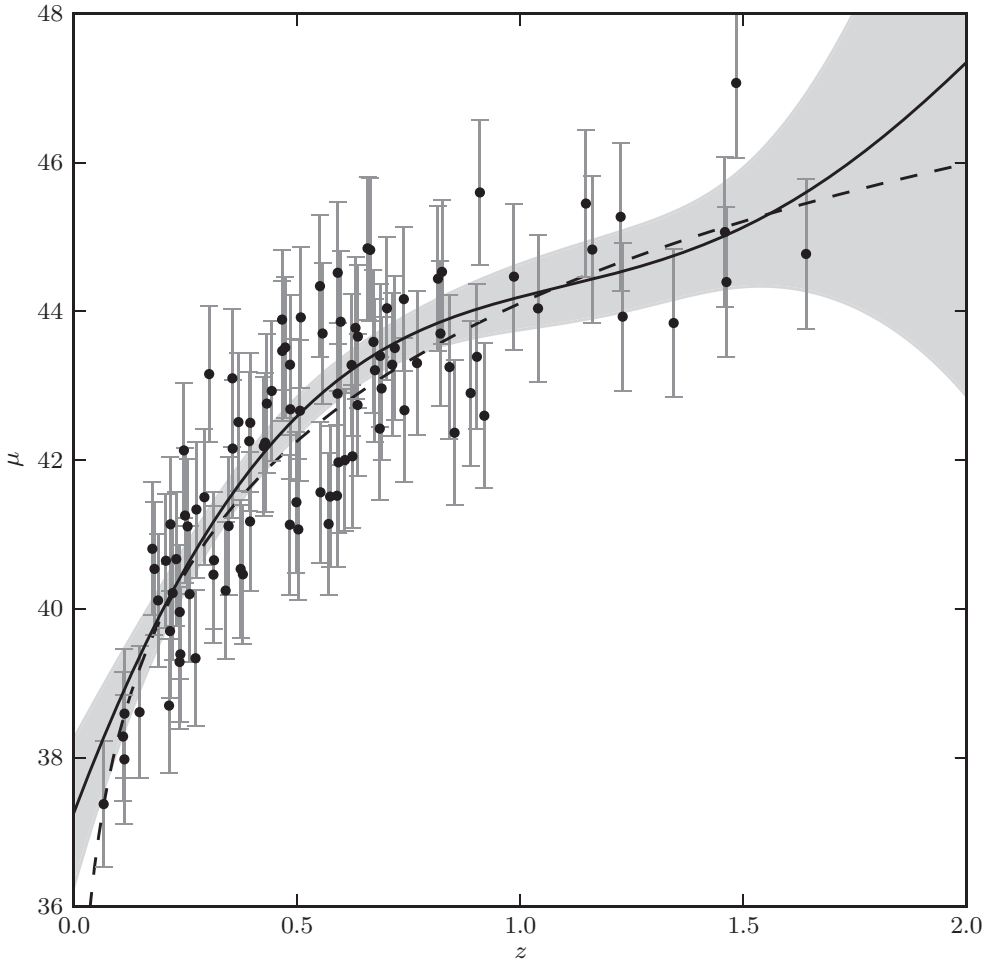

Figure 8.11. A Gaussian process regression analysis of the simulated supernova sample used in figure 8.2. This uses a squared-exponential covariance model, with bandwidth learned through cross-validation.

8.11. Overfitting, Underfitting, and Cross-Validation

When using regression, whether from a Bayesian or maximum likelihood perspective, it is important to recognize some of the potential pitfalls associated with these methods. As noted above, the optimality of the regression is contingent on correct model selection. In this section we explore cross-validation methods which can help determine whether a potential model is a good fit to the data. These techniques are complementary to the model selection techniques such as AIC and BIC discussed in § 4.3. This section will introduce the important topics of overfitting and underfitting, bias and variance, and introduces the frequentist tool of cross-validation to understand these.

Here, for simplicity, we will consider the example of a simple one-dimensional model with homoscedastic errors, though the results of this section naturally generalize to more sophisticated models. As above, our observed data is x_i, and we're trying to predict the dependent variable y_i. We have a training sample in which we have observed both x_i and y_i, and an unknown sample for which only x_i is measured.

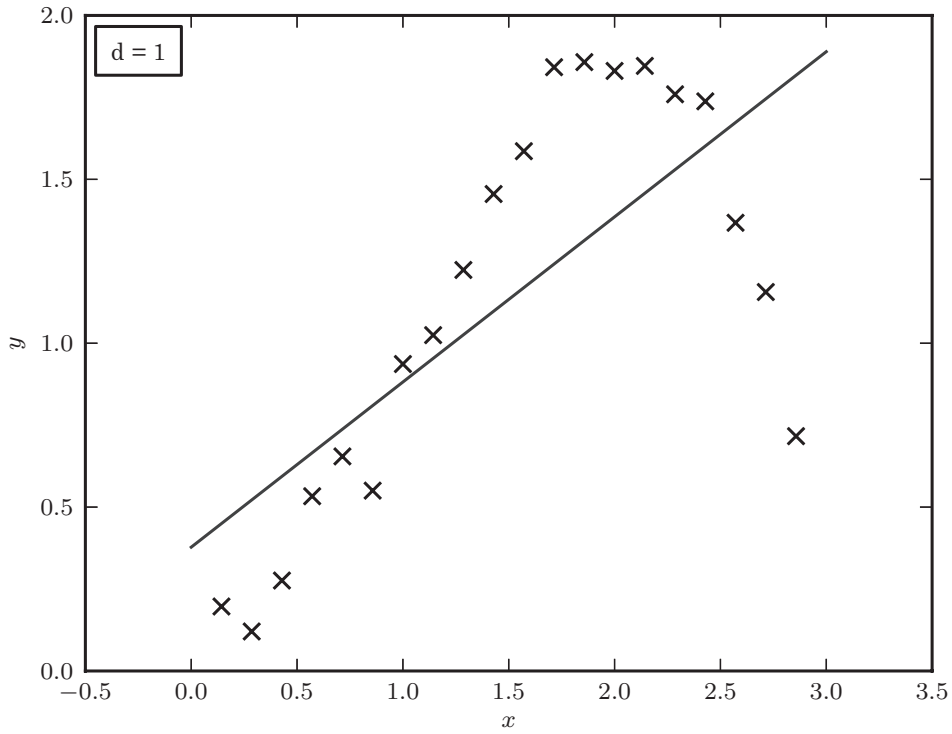

Figure 8.12. Our toy data set described by eq. 8.75. Shown is the line of best fit, which quite clearly underfits the data. In other words, a linear model in this case has high bias.

For example, you may be looking at the fundamental plane for elliptical galaxies, and trying to predict a galaxy's central black hole mass given the velocity dispersion and surface brightness of the stars. Here y_i is the mass of the black hole, and x_i is a vector of a length two consisting of velocity dispersion and surface brightness measurements.

Throughout the rest of this section, we will use a simple model where x and y satisfy the following:

$$\begin{aligned} 0 \leq x_i \leq 3, \\ y_i = x_i \sin(x_i) + \epsilon_i, \end{aligned} \tag{8.75}$$

where the noise is drawn from a normal distribution $\epsilon_i \sim \mathcal{N}(0, 0.1)$. The values for 20 regularly spaced points are shown in figure 8.12.

We will start with a simple straight-line fit to our data. The model is described by two parameters, the slope of the line, θ_1, and the y-axis intercept, θ_0, and is found by minimizing the mean square error,

$$\epsilon = \frac{1}{N} \sum_{i=1}^{N} (y_i - \theta_0 - \theta_1 x_i)^2. \tag{8.76}$$

The resulting best-fit line is shown in figure 8.12. It is clear that a straight line is not a good fit: it does not have enough flexibility to accurately model the data. We say in this case that the model is biased, and that it underfits the data.

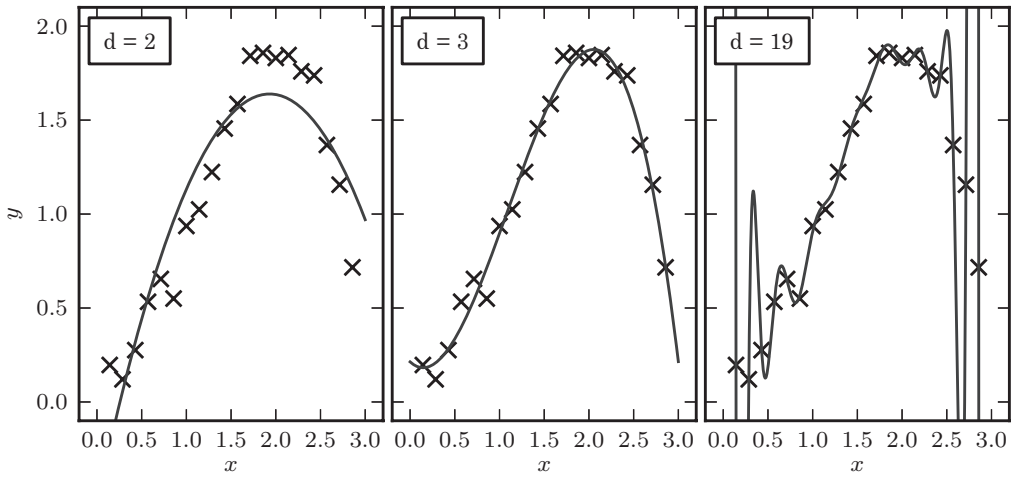

Figure 8.13. Three models of increasing complexity applied to our toy data set (eq. 8.75). The $d = 2$ model, like the linear model in figure 8.12, suffers from high bias, and underfits the data. The $d = 19$ model suffers from high variance, and overfits the data. The $d = 3$ model is a good compromise between these extremes.

What can be done to improve on this? One possibility is to make the model more sophisticated by increasing the degree of the polynomial (see § 8.2.2). For example, we could fit a quadratic function, or a cubic function, or in general a d-degree polynomial. A more complicated model with more free parameters should be able to fit the data much more closely. The panels of figure 8.13 show the best-fit polynomial model for three different choices of the polynomial degree d.

As the degree of the polynomial increases, the best-fit curve matches the data points more and more closely. In the extreme of $d = 19$, we have 20 degrees of freedom with 20 data points, and the training error given by eq. 8.76 can be reduced to zero (though numerical issues can prevent this from being realized in practice). Unfortunately, it is clear that the $d = 19$ polynomial is not a better fit to our data as a whole: the wild swings of the curve in the spaces between the training points are not a good description of the underlying data model. The model suffers from high variance; it overfits the data. The term variance is used here because a small perturbation of one of the training points in the $d = 19$ model can change the best-fit model by a large magnitude. In a high-variance model, the fit varies strongly depending on the exact set or subset of data used to fit it.

The center panel of figure 8.13 shows a $d = 3$ model which balances the trade-off between bias and variance: it does not display the high bias of the $d = 2$ model, and does not display high variance like the $d = 19$ model. For simple two-dimensional data like that seen here, the bias/variance trade-off is easy to visualize by plotting the model along with the input data. But this strategy is not as fruitful as the number of data dimensions grows. What we need is a general measure of the "goodness of fit" of different models to the training data. As displayed above, the mean square error does not paint the whole picture: increasing the degree of the polynomial in this case can lead to smaller and smaller training errors, but this reflects overfitting of the data rather than an improved approximation of the underlying model.

An important practical aspect of regression analysis lies in addressing this deficiency of the training error as an evaluation of goodness of fit, and finding a

model which best compromises between high bias and high variance. To this end, the process of *cross-validation* can be used to quantitatively evaluate the bias and variance of a regression model.

8.11.1. Cross-Validation

There are several possible approaches to cross-validation. We will discuss one approach in detail here, and list some alternative approaches at the end of the section. The simplest approach to cross-validation is to split the training data into three parts: the training set, the cross-validation set, and the test set. As a rule of thumb, the training set should comprise 50–70% of the original training data, while the remainder is divided equally into the cross-validation set and test set.

The training set is used to determine the parameters of a given model (i.e., the optimal values of θ_j for a given choice of d). Using the training set, we evaluate the training error $\epsilon_{\rm tr}$ using eq. 8.76. The cross-validation set is used to evaluate the cross-validation error $\epsilon_{\rm cv}$ of the model, also via eq. 8.76. Because this cross-validation set was not used to construct the fit, the cross-validation error will be large for a high-bias (overfit) model, and better represents the true goodness of fit of the model. With this in mind, the model which minimizes this cross-validation error is likely to be the best model in practice. Once this model is determined, the test error is evaluated using the test set, again via eq. 8.76. This test error gives an estimate of the reliability of the model for an unlabeled data set.

Why do we need a test set as well as a cross-validation set? In one sense, just as the parameters (in this case, θ_j) are learned from the training set, the so-called hyperparameters—those parameters which describe the complexity of the model (in this case, d)—are learned from the cross-validation set. In the same way that the parameters can be overfit to the training data, the hyperparameters can be overfit to the cross-validation data, and the cross-validation error gives an overly optimistic estimate of the performance of the model on an unlabeled data set. The test error is a better representation of the error expected for a new set of data. This is why it is recommended to use both a cross-validation set and a test set in your analysis.

A useful way to use the training error and cross-validation error to evaluate a model is to look at the results graphically. Figure 8.14 shows the training error and cross-validation error for the data in figure 8.13 as a function of the polynomial degree d. For reference, the dotted line indicates the level of intrinsic scatter added to our data.

The broad features of this plot reflect what is generally seen as the complexity of a regression model is increased: for small d, we see that both the training error and cross-validation error are very high. This is the tell-tale indication of a high-bias model, in which the model underfits the data. Because the model does not have enough complexity to describe the intrinsic features of the data, it performs poorly for both the training and cross-validation sets.

For large d, we see that the training error becomes very small (smaller than the intrinsic scatter we added to our data) while the cross-validation error becomes very large. This is the telltale indication of a high-variance model, in which the model overfits the data. Because the model is overly complex, it can match subtle variations in the training set which do not reflect the underlying distribution. Plotting this sort of information is a very straightforward way to settle on a suitable model. Of course,

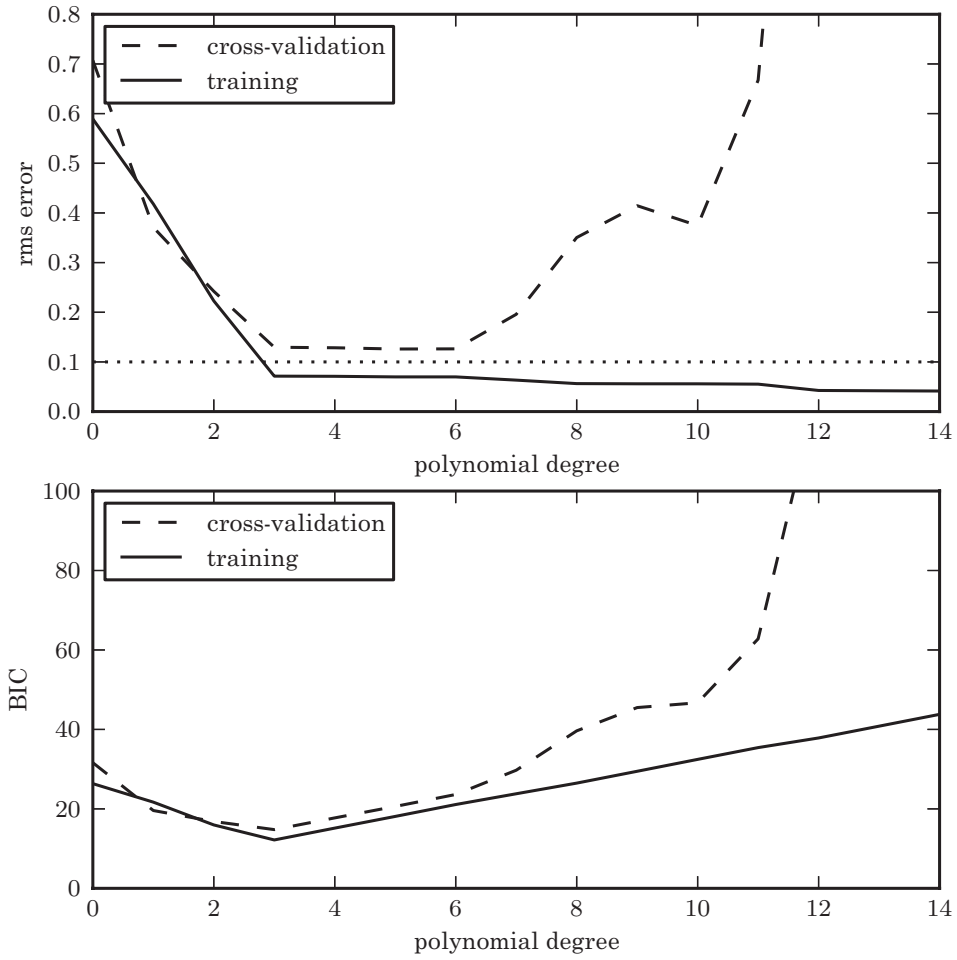

Figure 8.14. The top panel shows the root-mean-square (rms) training error and cross-validation error for our toy model (eq. 8.75) as a function of the polynomial degree d. The horizontal dotted line indicates the level of intrinsic scatter in the data. Models with polynomial degree from 3 to 5 minimize the cross-validation rms error. The bottom panel shows the Bayesian information criterion (BIC) for the training and cross-validation subsamples. According to the BIC, a degree-3 polynomial gives the best fit to this data set.

AIC and BIC provide another way to choose optimal d. Here both methods would choose the model with the best possible cross-validation error: $d = 3$.

8.11.2. Learning Curves

One question that cross-validation does not directly address is that of how to improve a model that is not giving satisfactory results (e.g., the cross-validation error is much larger than the known errors). There are several possibilities:

1. **Get more training data**. Often, using more data to train a model can lead to better results. Surprisingly, though, this is not always the case.
2. **Use a more/less complicated model**. As we saw above, the complexity of a model should be chosen as a balance between bias and variance.
3. **Use more/less regularization**. Including regularization, as we saw in the discussion of ridge regression (see § 8.3.1) and other methods above, can help

with the bias/variance trade-off. In general, increasing regularization has a similar effect to decreasing the model complexity.

4. **Increase the number of features**. Adding more observations of each object in your set can lead to a better fit. But this may not always yield the best results.

The choice of which route to take is far beyond a simple philosophical matter: for example, if you desire to improve your photometric redshifts for a large astronomical survey, it is important to evaluate whether you stand to benefit more from increasing the size of the training set (i.e., gathering spectroscopic redshifts for more galaxies) or from increasing the number of observations of each galaxy (i.e., reobserving the galaxies through other passbands). The answer to this question will inform the allocation of limited, and expensive, telescope time.

Note that this is a fundamentally different question than that explored above. There, we had a fixed data set, and were trying to determine the best model. Here, we assume a fixed model, and are asking how to improve the data set. One way to address this question is by plotting learning curves. A learning curve is the plot of the training and cross-validation error as a function of the number of training points. The details are important, so we will write this out explicitly.

Let our model be represented by the set of parameters $\boldsymbol{\theta}$. In the case of our simple example, $\boldsymbol{\theta} = \{\theta_0, \theta_1, \ldots, \theta_d\}$. We will denote by $\boldsymbol{\theta}^{(n)} = \{\theta_0^{(n)}, \theta_1^{(n)}, \ldots, \theta_d^{(n)}\}$ the model parameters which best fit the first n points of the training data: here $n \leq N_{\rm train}$, where $N_{\rm train}$ is the total number of training points. The truncated training error for $\theta^{(n)}$ is given by

$$\epsilon_{\rm tr}^{(n)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left[y_i - \sum_{m=0}^{d}\theta_0^{(n)}x_i^m\right]}. \tag{8.77}$$

Note that the training error $\epsilon_{\rm tr}^{(n)}$ is evaluated using only the n points on which the model parameters $\theta^{(n)}$ were trained, not the full set of $N_{\rm train}$ points. Similarly, the truncated cross-validation error is given by

$$\epsilon_{\rm cv}^{(n)} = \sqrt{\frac{1}{n}\sum_{i=1}^{N_{\rm cv}}\left[y_i - \sum_{m=0}^{d}\theta_0^{(n)}x_i^m\right]}, \tag{8.78}$$

where we sum over *all* of the cross-validation points. A learning curve is the plot of the truncated training error and truncated cross-validation error as a function of the size n of the training set used. For our toy example, this plot is shown in figure 8.15 for models with $d = 2$ and $d = 3$. The dotted line in each panel again shows for reference the intrinsic error added to the data.

The two panels show some common features, which are reflective of the features of learning curves for any regression model:

1. As we increase the size of the training set, the training error increases. The reason for this is simple: a model of a given complexity can better fit a small set of data than a large set of data. Moreover, aside from small random fluctuations, we expect this training error to always increase with the size of the training set.

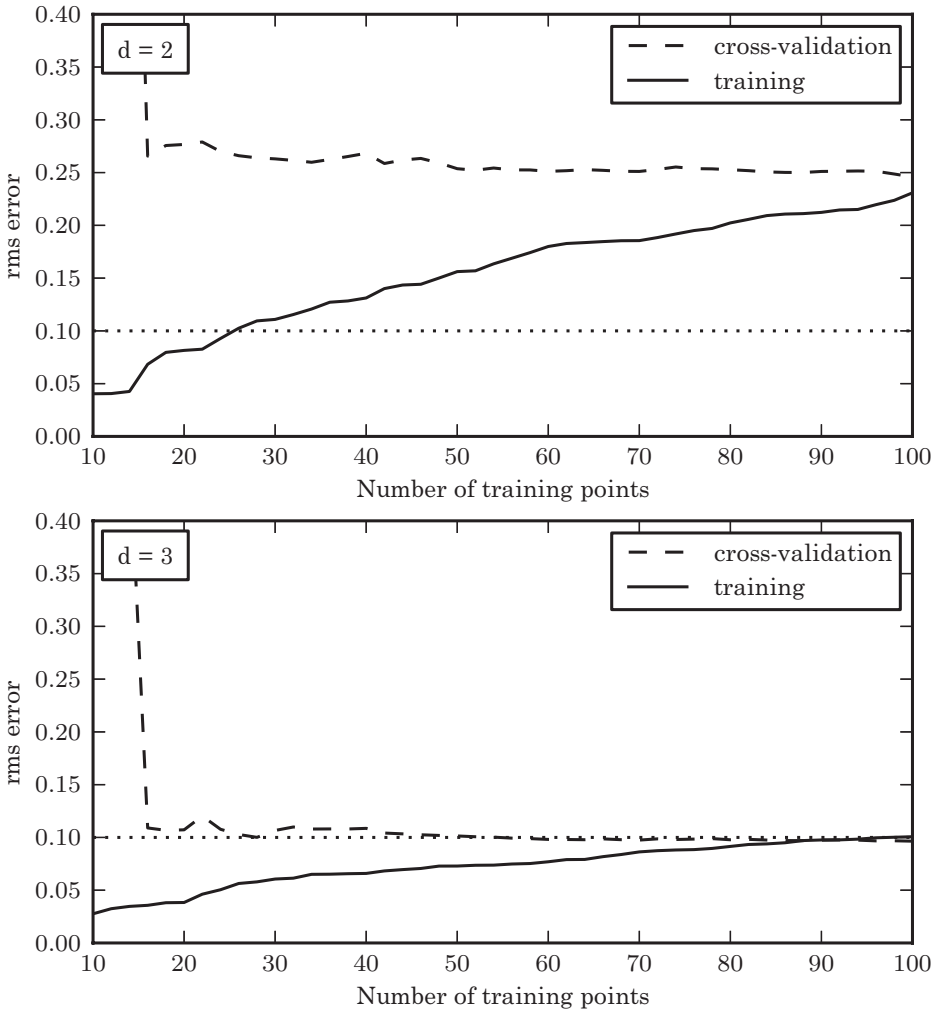

Figure 8.15. The learning curves for the data given by eq. 8.75, with $d = 2$ and $d = 3$. Both models have high variance for a few data points, visible in the spread between training and cross-validation error. As the number of points increases, it is clear that $d = 2$ is a high-bias model which cannot be improved simply by adding training points.

2. As we increase the size of the training set, the cross-validation error decreases. The reason for this is easy to see: a smaller training set leads to overfitting the model, meaning that the model is less representative of the cross-validation data. As the training set grows, overfitting is reduced and the cross-validation error decreases. Again, aside from random fluctuations, and as long as the training set and cross-validation set are statistically similar, we expect the cross-validation error to always decrease as the training set size grows.
3. The training error is everywhere less than or equal to the cross-validation error, up to small statistical fluctuations. We expect the model on average to better describe the data used to train it.
4. The logical outcome of the above three observations is that as the size N of the training set becomes large, the training and cross-validation curves will converge to the same value.

By plotting these learning curves, we can quickly see the effect of adding more training data. When the curves are separated by a large amount, the model error is dominated by variance, and additional training data will help. For example, in the lower panel of figure 8.15, at $N = 15$, the cross-validation error is very high and the training error is very low. Even if we only had 15 points and could not plot the remainder of the curve, we could infer that adding more data would improve the model.

On the other hand, when the two curves have converged to the same value, the model error is dominated by bias, and adding additional training data cannot improve the results *for that model*. For example, in the top panel of figure 8.15, at $N = 100$ the errors have nearly converged. Even without additional data, we can infer that adding data will not decrease the error below about 0.23. Improving the error in this case requires a more sophisticated model, or perhaps more features measured for each point.

To summarize, plotting learning curves can be very useful for evaluating the efficiency of a model and potential paths to improving your data. There are two possible situations:

1. **The training error and cross-validation error have converged**. In this case, increasing the number of training points under the same model is futile: the error cannot improve further. This indicates a model error dominated by bias (i.e., it is underfitting the data). For a high-bias model, the following approaches may help:

    - Add additional features to the data.
    - Increase the model complexity.
    - Decrease the regularization.

2. **The training error is much smaller than the cross-validation error**. In this case, increasing the number of training points is likely to improve the model. This condition indicates that the model error is dominated by variance (i.e., it is overfitting the data). For a high-variance model, the following approaches may help:

    - Increase the training set size.
    - Decrease the model complexity.
    - Increase the amplitude of the regularization.

Finally, we note a few caveats: first, the learning curves seen in figure 8.15 and the model complexity evaluation seen in figure 8.14 are actually aspects of a three-dimensional space. Changing the data and changing the model go hand in hand, and one should always combine the two diagnostics to seek the best match between model and data.

Second, this entire discussion assumes that the training data, cross-validation data, and test data are statistically similar. This analysis will fail if the samples are drawn from different distributions, or have different measurement errors or different observational limits.

8.11.3. Other Cross-Validation Techniques

There are numerous cross-validation techniques available which are suitable for different situations. It is easy to generalize from the above discussion to these various cross-validation strategies, so we will just briefly mention them here.

Twofold cross-validation

Above, we split the data into a training set d_1, a cross-validation set d_2, and a test set d_0. Our simple tests involved training the model on d_0 and cross-validating the model on d_1. In twofold cross-validation, this process is repeated, training the model on d_1 and cross-validating the model on d_0. The training error and cross-validation error are computed from the mean of the errors in each fold. This leads to more robust determination of the cross-validation error for smaller data sets.

K-fold cross-validation

A generalization of twofold cross-validation is K-fold cross-validation. Here we split the data into $K + 1$ sets: the test set d_0, and the cross-validation sets $d_1, d_2, \ldots, d_K$. We train K different models, each time leaving out a single subset to measure the cross-validation error. The final training error and cross-validation error can be computed using the mean or median of the set of results. The median can be a better statistic than the mean in cases where the subsets d_i contain few points.

Leave-one-out cross-validation

At the extreme of K-fold cross-validation is leave-one-out cross-validation. This is essentially the same as K-fold cross-validation, but this time our sets $d_1, d_2, \ldots, d_K$ have only one data point each. That is, we repeatedly train the model, leaving out only a single point to estimate the cross-validation error. Again, the final training error and cross-validation error are estimated using the mean or median of the individual trials. This can be useful when the size of the data set is very small, so that significantly reducing the number of data points leads to much different model characteristics.

Random subset cross-validation

In this approach, the cross-validation set and training set are selected by randomly partitioning the data, and repeating any number of times until the error statistics are well sampled. The disadvantage here is that not every point is guaranteed to be used both for training and for cross-validation. Thus, there is a finite chance that an outlier can lead to spurious results. For N points and P random samplings of the data, this situation becomes very unlikely for $N/2^P \ll 1$.

8.11.4. Summary of Cross-Validation and Learning Curves

In this section we have shown how to evaluate how well a model fits a data set through cross-validation. This is one practical route to the model selection ideas presented in chapters 4–5. We have covered how to determine the best model given a data set (§ 8.11.1) and how to address both the model and the data together to improve results (§ 8.11.2).

Cross-validation is one place where machine learning and data mining may be considered more of an art than a science. The optimal route to improving a model is not always straightforward. We hope that by following the suggestions in this chapter, you can apply this art successfully to your own data.

8.12. Which Regression Method Should I Use?

As we did in the last two chapters, we will use the axes of "accuracy," "interpretability," "simplicity," and "speed" (see § 6.6 for a description of these terms) to provide a rough guide to the trade-offs in choosing between the different regression methods described in this chapter.

What are the most* accurate *regression methods ? The starting point is basic linear regression. Adding ridge or LASSO regularization in principle increases accuracy, since as long as $\lambda = 0$ is among the options tried for λ, it provides a superset of possible complexity trade-offs to be tried. This goes along with the general principle that models with more parameters have a greater chance of fitting the data well. Adding the capability to incorporate measurement errors should in principle increase accuracy, assuming of course that the error estimates are accurate enough. Principal component regression should generally increase accuracy by treating collinearity and effectively denoising the data. All of these methods are of course linear—the largest leap in accuracy is likely to come when going from linear to nonlinear models. A starting point for increasing accuracy through nonlinearity is a linear model on nonlinear transformations of the original data. The extent to which this approach increases accuracy depends on the sensibility of the transformations done, since the nonlinear functions introduced are chosen manually rather than automatically. The sensibility can be diagnosed in various ways, such as checking the Gaussianity of the resulting errors. Truly nonlinear models, in the sense that the variable interactions can be nonlinear as well, should be more powerful in general. The final significant leap of accuracy will generally come from going to nonparametric methods such as kernel regression, starting with Nadaraya–Watson regression and more generally, local polynomial regression. Gaussian process regression is typically even more powerful in principle, as it effectively includes an aspect similar to that of kernel regression through the covariance matrix, but also learns coefficients on each data point.

What are the most* interpretable *regression methods ? Linear methods are easy to interpret in terms of understanding the relative importance of each variable through the coefficients, as well as reasoning about how the model will react to different inputs. Ridge or LASSO regression in some sense increase interpretability by identifying the most important features. Because feature selection happens as the result of an overall optimization, reasoning deeply about why particular features were kept or eliminated is not necessarily fruitful. Bayesian formulations, as in the case of models that incorporate errors, add to interpretability by making assumptions clear. PCR begins to decrease interpretability in the sense that the columns are no longer identifiable in their original terms. Moving to nonlinear methods, generalized linear models maintain the identity of the original columns, while general nonlinear

TABLE 8.1.
A summary of the practical properties of different regression methods.

Method	Accuracy	Interpretability	Simplicity	Speed
Linear regression	L	H	H	H
Linear basis function regression	M	M	M	M
Ridge regression	L	H	M	H
LASSO regression	L	H	M	L
PCA regression	M	M	M	M
Nadaraya–Watson regression	M/H	M	H	L/M
Local linear/polynomial regression	H	M	M	L/M
Nonlinear regression	M	H	L	L/M

models tend to become much less interpretable. Kernel regression methods can arguably be relatively interpretable, as their behavior can be reasoned about in terms of distances between points. Gaussian process regression is fairly opaque, as it is relatively difficult to reason about the behavior of the inverse of the data covariance matrix.

What are the* simplest *regression methods ? Basic linear regression has no tunable parameters, so it qualifies as the simplest out-of-the-box method. Generalized linear regression is similar, aside from the manual preparation of the nonlinear features. Ridge and LASSO regression require that only one parameter is tuned, and the optimization is convex, so that there is no need for random restarts, and cross-validation can make learning fairly automatic. PCR is similar in that there is only one critical parameter and the optimization is convex. Bayesian formulations of regression models which result in MCMC are subject to our comments in § 5.9 regarding fiddliness (as well as computational cost). Nadaraya–Watson regression typically has only one critical parameter, the bandwidth. General local polynomial regression can be regarded as having another parameter, the polynomial order. Basic Gaussian process regression has only one critical parameter, the bandwidth of the covariance kernel, and is convex, though extensions with several additional parameters are often used.

What are the most* scalable *regression methods ? Methods based on linear regression are fairly tractable using state-of-the-art linear algebra methods, including basic linear regression, ridge regression, and generalized linear regression, assuming the dimensionality is not excessively high. LASSO requires the solution of a linear program, which becomes expensive as the dimension rises. For PCR, see our comments in § 8.4 regarding PCA and SVD. Kernel regression methods are naively $\mathcal{O}(N^2)$ but can be sped up by fast tree-based algorithms in ways similar to those discussed in § 2.5.2 and chapter 6. Certain approximate algorithms exist for Gaussian process regression, which is naively $\mathcal{O}(N^3)$, but GP is by far the most expensive among the methods we have discussed, and difficult to speed up algorithmically while maintaining high predictive accuracy.

Other considerations, and taste Linear methods can be straightforwardly augmented to handle missing values. The Bayesian approach to linear regression

incorporates measurement errors, while standard versions of machine learning methods do not incorporate errors. Gaussian process regression typically comes with posterior uncertainty bands, though confidence bands can be obtained for any method, as we have discussed in previous chapters. Regarding taste, LASSO is embedded in much recent work on sparsity and is related to work on compressed sensing, and Gaussian process regression has interesting interpretations in terms of priors in function space and in terms of kernelized linear regression.

Simple summary . We summarize our discussion in table 8.1, in terms of "accuracy," "interpretability," "simplicity," and "speed" (see § 6.6 for a description of these terms), given in simple terms of high (H), medium (M), and low (L).

References

[1] Astier, P., J. Guy, N. Regnault, and others (2005). The supernova legacy survey: Measurement of ω_m, ω_λ and w from the first year data set. *Astronomy & Astrophysics 447*(1), 24.

[2] Boscovich, R. J. (1757). De litteraria expeditione per pontificiam ditionem, et synopsis amplioris operis, ac habentur plura ejus ex exemplaria etiam sensorum impressa. *Bononiensi Scientarum et Artum Instituto Atque Academia Commentarii IV*, 353–396.

[3] Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association 74*(368), 829–836.

[4] Dellaportas, P. and D. A. Stephens (1995). Bayesian analysis of errors-in-variables regression models. *Biometrics 51*(3), 1085–1095.

[5] Efron, B., T. Hastie, I. Johnstone, and R. Tibshirani (2004). Least angle regression. *Annals of Statistics 32*(2), 407–451.

[6] Gauss, K. F. (1809). *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium*. Hamburg: Sumtibus F. Perthes et I. H. Besser.

[7] Hogg, D. W., I. K. Baldry, M. R. Blanton, and D. J. Eisenstein (2002). The K correction. *ArXiv:astro-ph/0210394*.

[8] Hogg, D. W., J. Bovy, and D. Lang (2010). Data analysis recipes: Fitting a model to data. *ArXiv:astro-ph/1008.4686*.

[9] Huber, P. J. (1964). Robust estimation of a local parameter. *Annals of Mathematical Statistics 35*, 73–101.

[10] Jolliffe, I. T. (1986). *Principal Component Analysis*. Springer.

[11] Kelly, B. C. (2011). Measurement error models in astronomy. *ArXiv:astro-ph/1112.1745*.

[12] Kim, J., Y. Kim, and Y. Kim (2008). A gradient-based optimization algorithm for LASSO. *Journal of Computational and Graphical Statistics 17*(4), 994–1009.

[13] Krajnović, D. (2011). A Jesuit anglophile: Rogerius Boscovich in England. *Astronomy and Geophysics 52*(6), 060000–6.

[14] Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*. Paris: Courcier.

[15] Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly Applied Mathematics II(2)*, 164–168.

[16] Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics 11*(2), 431–441.

[17] Mertens, B., T. Fearn, and M. Thompson (1995). The efficient cross-validation of principal components applied to principal component regression. *Statistics and Computing 5*, 227–235. 10.1007/BF00142664.

[18] Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability and its Applications 9*, 141–142.

[19] Rasmussen, C. and C. Williams (2005). *Gaussian Processes for Machine Learning*. Adaptive Computation And Machine Learning. MIT Press.

[20] Theil, H. (1950). A rank invariant method of linear and polynomial regression analysis, I, II, III. *Proceedings of the Koninklijke Nederlandse Akademie Wetenschappen, Series A – Mathematical Sciences 53*, 386–392, 521–525, 1397–1412.

[21] Tibshirani, R. J. (1996). Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society, Series B 58*(1), 267–288.

[22] Tikhonov, A. N. (1995). *Numerical Methods for the Solution of Ill-Posed Problems*, Volume 328 of *Mathematics and Its Applications*. Kluwer.

[23] Watson, G. S. (1964). Smooth regression analysis. *Sankhyā Ser. 26*, 359–372.

[24] Zu, Y., C. S. Kochanek, S. Kozłowski, and A. Udalski (2012). Is quasar variability a damped random walk? *ArXiv:astro-ph/1202.3783*.

9 Classification

"One must always put oneself in a position to choose between two alternatives."
(Talleyrand)

In chapter 6 we described techniques for estimating joint probability distributions from multivariate data sets and for identifying the inherent clustering within the properties of sources. We can think of this approach as the *unsupervised classification* of data. If, however, we have labels for some of these data points (e.g., an object is tall, short, red, or blue) we can utilize this information to develop a relationship between the label and the properties of a source. We refer to this as *supervised classification* .

The motivation for supervised classification comes from the long history of classification in astronomy. Possibly the most well known of these classification schemes is that defined by Edwin Hubble for the morphological classification of galaxies based on their visual appearance; see [7]. This simple classification scheme, subdividing the types of galaxies into seven categorical subclasses, was broadly adopted throughout extragalactic astronomy. Why such a simple classification became so predominant when subsequent works on the taxonomy of galaxy morphology (often with a better physical or mathematical grounding) did not, argues for the need to keep the models for classification simple. This agrees with the findings of George Miller who, in 1956, proposed that the number of items that people are capable of retaining within their short term memory was 7 ± 2 ("The magical number 7 ± 2" [10]). Subsequent work by Herbert Simon suggested that we can increase seven if we implement a partitioned classification system (much like telephone numbers) with a chunk size of three. Simple schemes have more impact—a philosophy we will adopt as we develop this chapter.

9.1. Data Sets Used in This Chapter

In order to demonstrate the strengths and weaknesses of these classification techniques, we will use two astronomical data sets throughout this chapter.

RR Lyrae

First is the set of photometric observations of RR Lyrae stars in the SDSS [8]. The data set comes from SDSS Stripe 82, and combines the Stripe 82 standard stars (§1.5.8),

which represent observations of nonvariable stars; and the RR Lyrae variables, pulled from the same observations as the standard stars, and selected based on their variability using supplemental data; see [16]. The sample is further constrained to a smaller region of the overall color–color space following [8] ($0.7 < u - g < 1.35$, $-0.15 < g - r < 0.4$, $-0.15 < r - i < 0.22$, and $-0.21 < i - z < 0.25$). These selection criteria lead to a sample of 92,658 nonvariable stars, and 483 RR Lyraes. Two features of this combined data set make it a good candidate for testing classification algorithms:

1. The RR Lyrae stars and main sequence stars occupy a very similar region in u, g, r, i, z color space. The distributions overlap slightly, which makes the choice of decision boundaries subject to the completeness and contamination trade-off discussed in §4.6.1 and §9.2.1.
2. The extreme imbalance between the number of sources and the number of background objects is typical of real-world astronomical studies, where it is often desirable to select rare events out of a large background. Such unbalanced data aptly illustrates the strengths and weaknesses of various classification methods.

We will use these data in the context of the classification techniques discussed below.

Quasars and stars

As a second data set for photometric classification, we make use of two catalogs of quasars and stars from the SDSS Spectroscopic Catalog. The quasars are derived from the DR7 Quasar Catalog (§1.5.6), while the stars are derived from the SEGUE Stellar Parameters Catalog (§1.5.7). The combined data has approximately 100,000 quasars and 300,000 stars. In this chapter, we use the $u - g$, $g - r$, $r - i$, and $i - z$ colors to demonstrate photometric classification of these objects. We stress that because of the different selection functions involved in creating the two catalogs, the combined sample does not reflect a real-world sample of the objects: we use it for purposes of illustration only.

Photometric redshifts

While photometric redshifts are technically a regression problem which belongs to chapter 8, they offer an excellent test case for decision trees and random forests, introduced in §9.7. The data for the photometric redshifts come from the SDSS spectroscopic database (§1.5.5). The magnitudes used are the model magnitudes mentioned above, while the true redshift measurements come from the spectroscopic pipeline.

9.2. Assigning Categories: Classification

Supervised classification takes a set of features and relates them to predefined sets of classes. Choosing the optimal set of features was touched on in the discussion of dimensionality reduction in §7.3. We will not address how we define the labels or taxonomy for the classification other than noting that the time-honored system of

having a graduate student label data does not scale to the size of today's data.[1] We start by assuming that we have a set of predetermined labels that have been assigned to a subset of the data we are considering. Our goal is to characterize the relation between the features in the data and their classes and apply these classifications to a larger set of unlabeled data.

As we go we will illuminate the connections between classification, regression, and density estimation. Classification can be posed in terms of density estimation—this is called *generative classification* (so-called since we will have a full model of the density for each class, which is the same as saying we have a model which describes how data could be generated from each class). This will be our starting point, where we will visit a number of methods. Among the advantages of this approach is a high degree of interpretability.

Starting from the same principles we will go to classification methods that focus on finding the decision boundary that separates classes directly, avoiding the step of modeling each class's density, called *discriminative classification*, which can often be better in high-dimensional problems.

9.2.1. Classification Loss

Perhaps the most common loss (cost) function in classification is *zero-one loss*, where we assign a value of one for a misclassification and zero for a correct classification. With $\widehat{y}$ representing the best guess value of y, we can write this classification loss, $L(y, \widehat{y})$, as

$$L(y, \widehat{y}) = \delta(y \neq \widehat{y}), \tag{9.1}$$

which means

$$L(y, \widehat{y}) = \begin{cases} 1 & \text{if } y \neq \widehat{y}, \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

The classification *risk* of a model (defined to be the expectation value of the loss—see §4.2.8) is given by

$$\mathbb{E}\left[L(y, \widehat{y})\right] = p(y \neq \widehat{y}), \tag{9.3}$$

or the probability of misclassification. This can be compared to the case of regression, where the most common loss function is $L(y, \widehat{y}) = (y - \widehat{y})^2$, leading to the risk $\mathbb{E}[(y - \widehat{y})^2]$. For the zero-one loss of classification, the risk is equal to the *misclassification rate* or *error rate*.

One particularly common case of classification in astronomy is that of "detection," where we wish to assign objects (i.e., regions of the sky or groups of pixels on a CCD) into one of two classes: a detection (usually with label 1) and a nondetection (usually with label 0). When thinking about this sort of problem, we may wish to

[1] If, however, you can enlist thousands of citizen scientists in this proposition then you can fundamentally change this aspect; see [9].

distinguish between the two possible kinds of error: assigning a label 1 to an object whose true class is 0 (a "false positive"), and assigning the label 0 to an object whose true class is 1 (a "false negative").

As in §4.6.1, we will define the *completeness*,

$$\text{completeness} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}, \tag{9.4}$$

and *contamination*,

$$\text{contamination} = \frac{\text{false positives}}{\text{true positives} + \text{false positives}}. \tag{9.5}$$

The completeness measures the fraction of total detections identified by our classifier, while the contamination measures the fraction of detected objects which are misclassified. Depending on the nature of the problem and the goal of the classification, we may wish to optimize one or the other.

Alternative names for these measures abound: in some fields the completeness and contamination are respectively referred to as the "sensitivity" and the "Type I error." In astronomy, one minus the contamination is often referred to as the "efficiency." In machine learning communities, the efficiency and completeness are respectively referred to as the "precision" and "recall."

9.3. Generative Classification

Given a set of data $\{\mathbf{x}\}$ consisting of N points in D dimensions, such that x_i^j is the jth feature of the ith point, and a set of discrete labels $\{y\}$ drawn from K classes, with values y_k, Bayes' theorem describes the relation between the labels and features:

$$p(y_k|\mathbf{x}_i) = \frac{p(\mathbf{x}_i|y_k)p(y_k)}{\sum_i p(\mathbf{x}_i|y_k)p(y_k)}. \tag{9.6}$$

If we knew the full probability densities $p(\mathbf{x}, y)$ it would be straightforward to estimate the classification likelihoods directly from the data. If we chose not to fully sample $p(\mathbf{x}, y)$ with our training set we can still define the classifications by drawing from $p(y|\mathbf{x})$ and comparing the likelihood ratios between classes (in this way we can focus our labeling on the specific, and rare, classes of source rather than taking a brute-force random sample).

In generative classifiers we are modeling the class-conditional densities explicitly, which we can write as $p_k(\mathbf{x})$ for $p(\mathbf{x}|y = y_k)$, where the class variable is, say, $y_k = 0$ or $y_k = 1$. The quantity $p(y = y_k)$, or π_k for short, is the probability of any point having class k, regardless of which point it is. This can be interpreted as the *prior* probability of the class k. If these are taken to include subjective information, the whole approach is Bayesian (chapter 5). If they are estimated from data, for example by taking the proportion in the training set that belong to class k, this can be considered as either a frequentist or as an empirical Bayes (see §5.2.4).

The task of learning the best classifier then becomes the task of estimating the p_k's. This approach means we will be doing multiple separate *density estimates*

using many of the techniques introduced in chapter 6. The most powerful (accurate) classifier of this type then, corresponds to the most powerful density estimator used for the p_k models. Thus the rest of this section will explore various models and approximations for the $p_k(\mathbf{x})$ in eq. 9.6. We will start with the simplest kinds of models, and gradually build the model complexity from there. First, though, we will discuss several illuminating aspects of the generative classification model.

9.3.1. General Concepts of Generative Classification

Discriminant function

With slightly more effort, we can formally relate the classification task to two of the major machine learning tasks we have seen already: density estimation (chapter 6) and regression (chapter 8). Recall, from chapter 8, the regression function $\widehat{y} = f(y|\mathbf{x})$: it represents the best guess value of y given a specific value of $\mathbf{x}$. Classification is simply the analog of regression where y is categorical, for example $y = \{0, 1\}$. We now call $f(y|\mathbf{x})$ the *discriminant function*:

$$g(\mathbf{x}) = f(y|\mathbf{x}) = \int y\, p(y|\mathbf{x})\, dy \tag{9.7}$$

$$= 1 \cdot p(y = 1|\mathbf{x}) + 0 \cdot p(y = 0|\mathbf{x}) = p(y = 1|\mathbf{x}). \tag{9.8}$$

If we now apply Bayes' rule (eq. 3.10), we find (cf. eq. 9.6)

$$g(\mathbf{x}) = \frac{p(\mathbf{x}|y = 1)\, p(y = 1)}{p(\mathbf{x}|y = 1)\, p(y = 1) + p(\mathbf{x}|y = 0)\, p(y = 0)} \tag{9.9}$$

$$= \frac{\pi_1 p_1(\mathbf{x})}{\pi_1 p_1(\mathbf{x}) + \pi_0 p_0(\mathbf{x})}. \tag{9.10}$$

Bayes classifier

Making the discriminant function yield a binary prediction gives the abstract template called a *Bayes classifier*. It can be formulated as

$$\widehat{y} = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 1/2, \\ 0 & \text{otherwise,} \end{cases} \tag{9.11}$$

$$= \begin{cases} 1 & \text{if } p(y = 1|\mathbf{x}) > p(y = 0|\mathbf{x}), \\ 0 & \text{otherwise,} \end{cases} \tag{9.12}$$

$$= \begin{cases} 1 & \text{if } \pi_1 p_1(\mathbf{x}) > \pi_0 p_0(\mathbf{x}), \\ 0 & \text{otherwise.} \end{cases} \tag{9.13}$$

This is easily generalized to any number of classes K, since we can think of a $g_k(\mathbf{x})$ for each class (in a two-class problem it is sufficient to consider $g(\mathbf{x}) = g_1(\mathbf{x})$). The Bayes classifier is a template in the sense that one can plug in different types of model for the p_k's and the π's. Furthermore, the Bayes classifier can be shown to be optimal if the p_k's and π's are chosen to be the true distributions: that is, lower error cannot

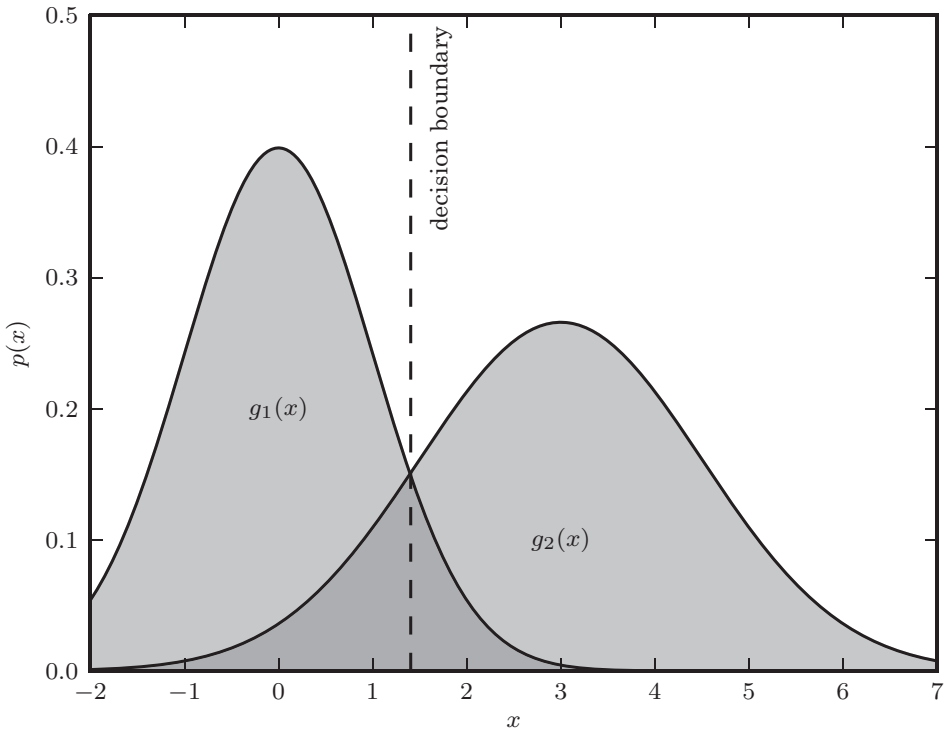

Figure 9.1. An illustration of a decision boundary between two Gaussian distributions.

be achieved. The Bayes classification template as described is an instance of empirical Bayes (§5.2.4).

Again, keep in mind that so far this is "Bayesian" only in the sense of utilizing Bayes' rule, an identity based on the definition of conditional distributions (§3.1.1), not in the sense of Bayesian inference. The interpretation/usage of the π_k quantities is what will make the approach either Bayesian or frequentist.

Decision boundary

The *decision boundary* between two classes is the set of x values at which each class is equally likely; that is,

$$\pi_1 p_1(\mathbf{x}) = \pi_2 p_2(\mathbf{x}); \tag{9.14}$$

that is, $g_1(\mathbf{x}) = g_2(\mathbf{x})$; that is, $g_1(\mathbf{x}) - g_2(\mathbf{x}) = 0$; that is, $g(\mathbf{x}) = 1/2$; in a two-class problem. Figure 9.1 shows an example of the decision boundary for a simple model in one dimension, where the density for each class is modeled as a Gaussian. This is very similar to the concept of hypothesis testing described in §4.6.

9.3.2. Naive Bayes

The Bayes classifier formalism presented above is conceptually simple, but can be very difficult to compute: in practice, the data $\{\mathbf{x}\}$ above may be in many dimensions, and have complicated probability distributions. We can dramatically reduce the complexity of the problem by making the assumption that all of the attributes we

measure are conditionally independent. This means that

$$p(x^i, x^j|y_k) = p(x^i|y)p(x^j|y_k), \tag{9.15}$$

where, recall, the superscript indexes the feature of the vector $\mathbf{x}$. For data in many dimensions, this assumption can be expressed as

$$p(x^0, x^1, x^2, \ldots, x^N|y_k) = \prod_i p(x^i|y_k). \tag{9.16}$$

Again applying Bayes' rule, we rewrite eq. 9.6 as

$$p(y_k|x^0, x^1, \ldots, x^N) = \frac{p(x^0, x^1, \ldots, x^N|y_k)p(y_k)}{\sum_j p(x^0, x^1, \ldots, x^N|y_j)p(y_j)}. \tag{9.17}$$

With conditional independence this becomes

$$p(y_k|x^0, x^1, \ldots, x^N) = \frac{\prod_i p(x^i|y_k)p(y_k)}{\sum_j \prod_i p(x^i|y_j)p(y_j)}. \tag{9.18}$$

Using this expression, we can calculate the most likely value of y by maximizing over y_k,

$$\widehat{y} = \arg\max_{y_k} \frac{\prod_i p(x^i|y_k)p(y_k)}{\sum_j \prod_i p(x^i|y_j)p(y_j)}, \tag{9.19}$$

or, using our shorthand notation,

$$\widehat{y} = \arg\max_{y_k} \frac{\prod_i p_k(x^i)\pi_k}{\sum_j \prod_i p_j(x^i)\pi_j}. \tag{9.20}$$

This gives a general prescription for the naive Bayes classification. Once sufficient models for $p_k(x^i)$ and π_k are known, the estimator $\widehat{y}$ can be computed very simply. The challenge, then, is to determine $p_k(x^i)$ and π_k, most often from a set of training data. This can be accomplished in a variety of ways, from fitting parametrized models using the techniques of chapters 4 and 5, to more general parametric and nonparametric density estimation techniques discussed in chapter 6.

The determination of $p_k(x^i)$ and μ_k can be particularly simple when the features x^i are categorical rather than continuous. In this case, assuming that the training set is a fair sample of the full data set (which may not be true), for each label y_k in the training set, the maximum likelihood estimate of the probability for feature x^i is simply equal to the number of objects with a particular value of x^i, divided by the total number of objects with $y = y_k$. The prior probabilities π_k are given by the fraction of training data with $y = y_k$.

Almost immediately, a complication arises. If the training set does not cover the full parameter space, then this estimate of the probability may lead to $p_k(x^i) = 0$ for some value of y_k and x^i. If this is the case, then the posterior probability in eq. 9.20 is

$p(y_k|\{x^i\}) = 0/0$ which is undefined! A particularly simple solution in this case is to use *Laplace smoothing*: an offset α is added to the probability of each bin $p_k(x^i)$ for all i, k, leading to well-defined probabilities over the entire parameter space. Though this may seem to be merely a heuristic trick, it can be shown to be equivalent to the addition of a Bayesian prior to the naive Bayes classifier.

9.3.3. Gaussian Naive Bayes and Gaussian Bayes Classifiers

It is rare in astronomy that we have discrete measurements for $\mathbf{x}$ even if we have categorical labels for y. The estimator for $\widehat{y}$ given in eq. 9.20 can also be applied to continuous data, given a sufficient estimate of $p_k(x^i)$. In Gaussian naive Bayes, each of these probabilities $p_k(x^i)$ is modeled as a one-dimensional normal distribution, with means μ_k^i and widths σ_k^i determined, for example, using the frequentist techniques in §4.2.3. In this case the estimator in eq. 9.20 can be expressed as

$$\widehat{y} = \arg\max_{y_k} \left[\ln \pi_k - \frac{1}{2} \sum_{i=1}^{N} \left(2\pi(\sigma_k^i)^2 + \frac{(x^i - \mu_k^i)^2}{(\sigma_k^i)^2} \right) \right], \tag{9.21}$$

where for simplicity we have taken the log of the Bayes criterion, and omitted the normalization constant, neither of which changes the result of the maximization.

The Gaussian naive Bayes estimator of eq. 9.21 essentially assumes that the multivariate distribution $p(\mathbf{x}|y_k)$ can be modeled using an axis-aligned multivariate Gaussian distribution. In figure 9.2, we perform a Gaussian naive Bayes classification on a simple, well-separated data set. Though examples like this one make classification straightforward, data in the real world is rarely so clean. Instead, the distributions often overlap, and categories have hugely imbalanced numbers. These features are seen in the RR Lyrae data set.

Scikit-learn has an estimator which performs fast Gaussian Naive Bayes classification:

```
import numpy as np
from sklearn.naive_bayes import GaussianNB

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
# simple division
gnb = GaussianNB()
gnb.fit(X, y)
y_pred = gnb.predict(X)
```

For more details see the Scikit-learn documentation.

In figure 9.3, we show the naive Bayes classification for RR Lyrae stars from SDSS Stripe 82. The completeness and contamination for the classification are shown in the right panel, for various combinations of features. Using all four colors, the Gaussian

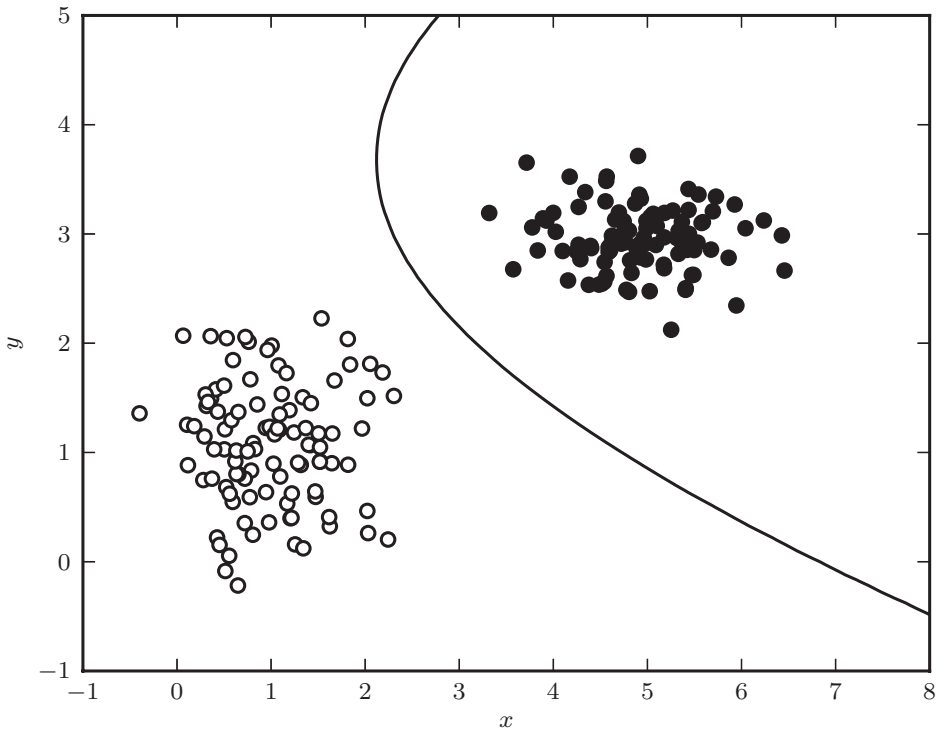

Figure 9.2. A decision boundary computed for a simple data set using Gaussian naive Bayes classification. The line shows the decision boundary, which corresponds to the curve where a new point has equal posterior probability of being part of each class. In such a simple case, it is possible to find a classification with perfect completeness and contamination. This is rarely the case in the real world.

naive Bayes classifier in this case attains a completeness of 87.6%, at the cost of a relatively high contamination rate of 79.0%.

A logical next step is to relax the assumption of conditional independence in eq. 9.16, and allow the Gaussian probability model for each class to have arbitrary correlations between variables. Allowing for covariances in the model distributions leads to the *Gaussian Bayes classifier* (i.e., it is no longer naive). As we saw in §3.5.4, a multivariate Gaussian can be expressed as

$$p_k(\mathbf{x}) = \frac{1}{|\Sigma_k|^{1/2}(2\pi)^{D/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu_\mathbf{k})^T \Sigma_k^{-1}(\mathbf{x}-\mu_\mathbf{k})\right\}, \tag{9.22}$$

where Σ_k is a $D \times D$ symmetric covariance matrix with determinant $\det(\Sigma_k) \equiv |\Sigma_k|$, and $\mathbf{x}$ and $\mu_\mathbf{k}$ are D-dimensional vectors. For this generalized Gaussian Bayes classifier, the estimator $\widehat{y}$ is (cf. eq. 9.21)

$$\widehat{y} = \arg\max_k \left\{-\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(\mathbf{x}-\mu_\mathbf{k})^T \Sigma_k^{-1}(\mathbf{x}-\mu_\mathbf{k}) + \log\pi_k\right\} \tag{9.23}$$

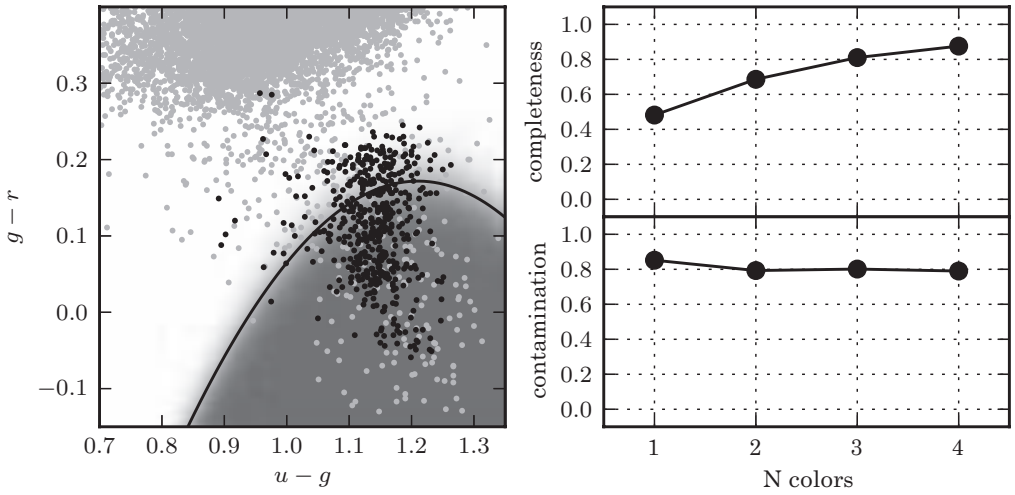

Figure 9.3. Gaussian naive Bayes classification method used to separate variable RR Lyrae stars from nonvariable main sequence stars. In the left panel, the light gray points show nonvariable sources, while the dark points show variable sources. The classification boundary is shown by the black line, and the classification probability is shown by the shaded background. In the right panel, we show the completeness and contamination as a function of the number of features used in the fit. For the single feature, $u-g$ is used. For two features, $u-g$ and $g-r$ are used. For three features, $u-g$, $g-r$, and $r-i$ are used. It is evident that the $g-r$ color is the best discriminator. With all four colors, naive Bayes attains a completeness of 0.876 and a contamination of 0.790.

or equivalently,

$$\widehat{y} = \begin{cases} 1 & \text{if } m_1^2 < m_0^2 + 2\log\left(\frac{\pi_1}{\pi_0}\right) + \left(\frac{|\Sigma_1|}{|\Sigma_0|}\right), \\ 0 & \text{otherwise,} \end{cases} \tag{9.24}$$

where $m_k^2 = (x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)$ is known as the *Mahalanobis distance*.

This step from Gaussian naive Bayes to a more general Gaussian Bayes formalism can include a large jump in computational cost: to fit a D-dimensional multivariate normal distribution to observed data involves estimation of $D(D+3)/2$ parameters, making a closed-form solution (like that for $D = 2$ in §3.5.2) increasingly tedious as the number of features D grows large. One efficient approach to determining the model parameters $\mu_{\mathbf{k}}$ and Σ_k is the expectation maximization algorithm discussed in §4.4.3, and again in the context of Gaussian mixtures in §6.3. In fact, we can use the machinery of Gaussian mixture models to extend Gaussian naive Bayes to a more general Gaussian Bayes formalism, simply by fitting to each class a "mixture" consisting of a single component. We will explore this approach, and the obvious extension to multiple component mixture models, in §9.3.5 below.

9.3.4. Linear Discriminant Analysis and Relatives

Linear discriminant analysis (LDA), like Gaussian naive Bayes, relies on some simplifying assumptions about the class distributions $p_k(\mathbf{x})$ in eq. 9.6. In particular, it assumes that these distributions have identical covariances for all K classes. This makes all classes a set of shifted Gaussians. The optimal classifier can then be derived

from the log of the class posteriors to be

$$g_k(\mathbf{x}) = \mathbf{x}^T \Sigma^{-1} \mu_\mathbf{k} - \frac{1}{2} \mu_\mathbf{k}{}^T \Sigma^{-1} + \log \pi_k, \tag{9.25}$$

with $\mu_\mathbf{k}$ the mean of class k and Σ the covariance of the Gaussians (which, in general, does not need to be diagonal). The class dependent covariances that would normally give rise to a quadratic dependence on $\mathbf{x}$ cancel out if they are assumed to be constant. The Bayes classifier is, therefore, linear with respect to $\mathbf{x}$.

The discriminant boundary between classes is the line that minimizes the overlap between Gaussians:

$$g_k(\mathbf{x}) - g_\ell(\mathbf{x}) = \mathbf{x}^T \Sigma^{-1}(\mu_k - \mu_\ell) - \frac{1}{2}(\mu_k - \mu_\ell)^T \Sigma^{-1}(\mu_k - \mu_\ell) + \log\left(\frac{\pi_k}{\pi_\ell}\right) = 0. \tag{9.26}$$

If we were to relax the requirement that the covariances of the Gaussians are constant, the discriminant function for the classes becomes quadratic in x:

$$g(\mathbf{x}) = -\frac{1}{2} \log |\Sigma_k| - \frac{1}{2}(\mathbf{x} - \mu_k)^T C^{-1}(\mathbf{x} - \mu_k) + \log \pi_k. \tag{9.27}$$

This is sometimes known as *quadratic discriminant analysis* (QDA), and the boundary between classes is described by a quadratic function of the features $\mathbf{x}$.

A related technique is called *Fisher's linear discriminant* (FLD). It is a special case of the above formalism where the priors are set equal but without the requirement that the covariances be equal. Geometrically, it attempts to project all data onto a single line, such that a decision boundary can be found on that line. By minimizing the loss over all possible lines, it arrives at a classification boundary. Because FLD is so closely related to LDA and QDA, we will not explore it further.

Scikit-learn has estimators which perform both LDA and QDA. They have a very similar interface:

```
import numpy as np
from sklearn.lda import LDA
from sklearn.qda import QDA

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division
lda = LDA()
lda.fit(X, y)
y_pred = lda.predict(X)

qda = QDA()
qda.fit(X, y)
y_pred = qda.predict(X)
```

For more details see the Scikit-learn documentation.

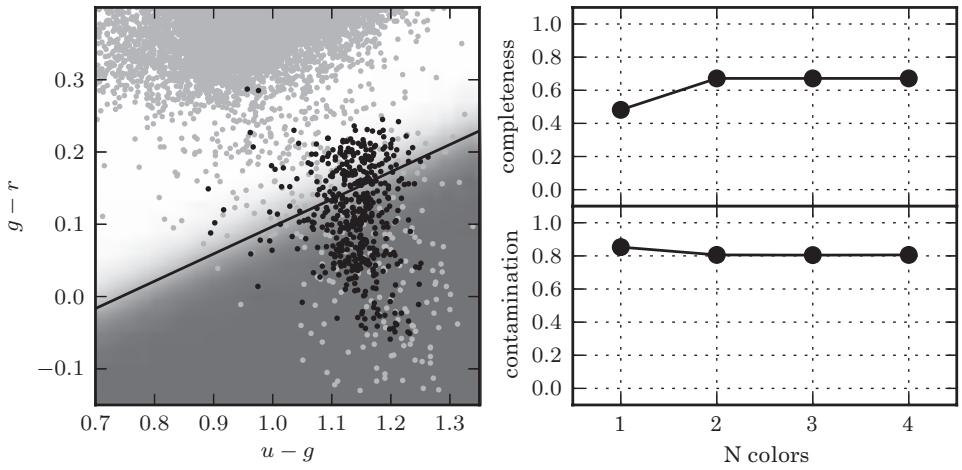

Figure 9.4. The linear discriminant boundary for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, LDA achieves a completeness of 0.672 and a contamination of 0.806.

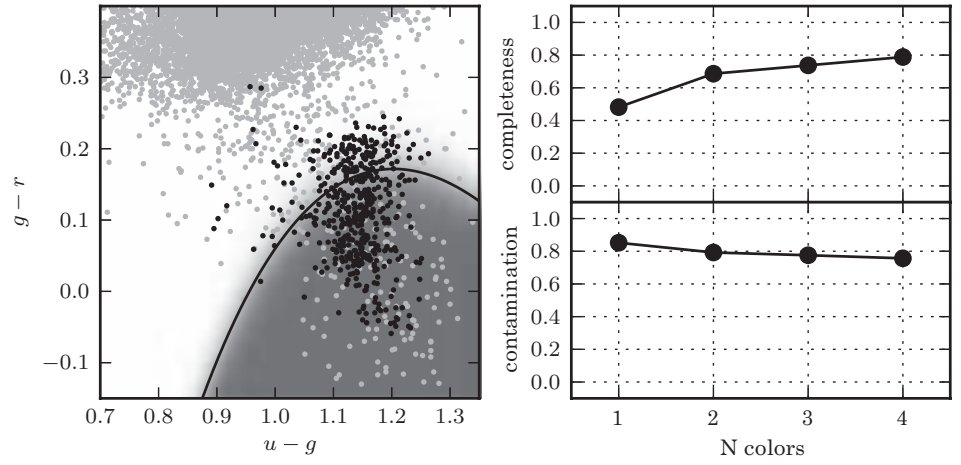

Figure 9.5. The quadratic discriminant boundary for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, QDA achieves a completeness of 0.788 and a contamination of 0.757.

The results of linear discriminant analysis and quadratic discriminant analysis on the RR Lyrae data from figure 9.3 are shown in figures 9.4 and 9.5, respectively. Notice that, true to their names, linear discriminant analysis results in a linear boundary between the two classes, while quadratic discriminant analysis results in a quadratic boundary. As may be expected with a more sophisticated model, QDA yields improved completeness and contamination in comparison to LDA.

9.3.5. More Flexible Density Models: Mixtures and Kernel Density Estimates

The above methods take the very general result expressed in eq. 9.6 and introduce simplifying assumptions which make the classification more computationally feasible. However, assumptions regarding conditional independence (as in naive Bayes) or Gaussianity of the distributions (as in Gaussian Bayes, LDA, and QDA) are not necessary parts of the model. With a more flexible model for the probability

distribution, we could more closely model the true distributions and improve on our ability to classify the sources. To this end, many of the techniques from chapter 6 can be applicable.

The next common step up in representation power for each $p_k(x)$, beyond a single Gaussian with arbitrary covariance matrix, is to use a Gaussian mixture model (GMM) (described in §6.3). Let us call this the *GMM Bayes classifier* for lack of a standard term. Each of the components may be constrained to a simple case (such as diagonal-covariance-only Gaussians etc.) to ease the computational cost of model fitting. Note that the number of Gaussian components K must be chosen, ideally, for each class independently, in addition to the cost of model fitting for each value of K tried. Adding the ability to account for measurement errors in Gaussian mixtures was described in §6.3.3.

AstroML contains an implementation of GMM Bayes classification based on the Scikit-learn Gaussian mixture model code:

```
import numpy as np
from astroML.classification import GMMBayes

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

gmmb = GMMBayes(3) # 3 clusters per class
gmmb.fit(X, y)
y_pred = gmmb.predict(X)
```

For more details see the AstroML documentation, or the source code of figure 9.6.

Figure 9.6 shows the GMM Bayes classification of the RR Lyrae data. The results with one component are similar to those of naive Bayes in figure 9.3. The difference is that here the Gaussian fits to the densities are allowed to have arbitrary covariances between dimensions. When we move to a density model consisting of three components, we significantly decrease the contamination with only a small effect on completeness. This shows the value of using a more descriptive density model.

For the ultimate in flexibility, and thus accuracy, we can model each class with a kernel density estimate. This *nonparametric* Bayes classifier is sometimes called *kernel discriminant analysis*. This method can be thought of as taking Gaussian mixtures to its natural limit, with one mixture component centered at each training point. It can also be generalized from the Gaussian to any desired kernel function. It turns out that even though the model is more complex (able to represent more complex functions), by going to this limit things become computationally simpler: unlike the typical GMM case, there is no need to optimize over the locations of the mixture components; the locations are simply the training points themselves. The optimization is over only one variable, the bandwidth of the kernel. One advantage of this approach is that when such flexible density models are used in the setting of

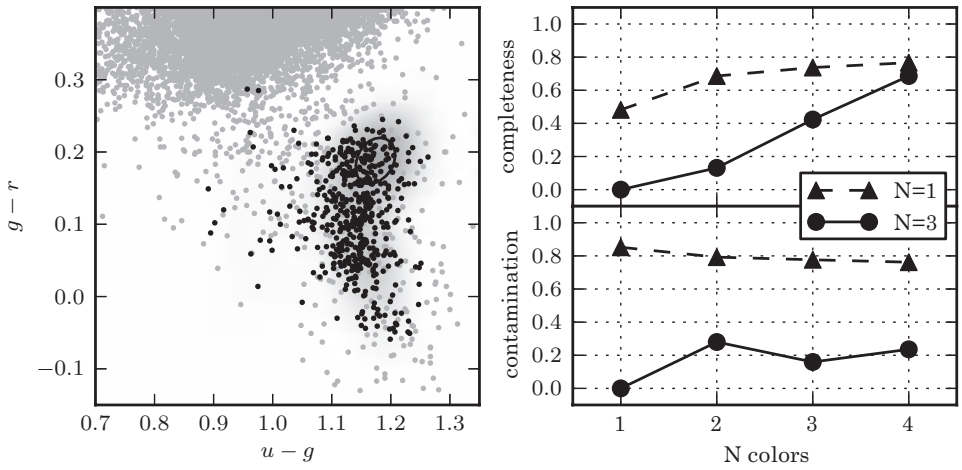

Figure 9.6. Gaussian mixture Bayes classifier for RR Lyrae stars (see caption of figure 9.3 for details). Here the left panel shows the decision boundary for the three-component model, and the right panel shows the completeness and contamination for both a one- and three-component mixture model. With all four colors and a three-component model, GMM Bayes achieves a completeness of 0.686 and a contamination of 0.236.

classification, their parameters can be chosen to maximize *classification* performance directly rather than density estimation performance.

The cost of the extra accuracy of kernel discriminant analysis is the high computational cost of evaluating kernel density estimates. We briefly discussed fast algorithms for kernel density estimates in §6.1.1, but an additional idea can be used to accelerate them in this setting. A key observation is that our computational problem can be solved more efficiently than by actually computing the full kernel summation needed for each class—to determine the class label for each query point, we need only determine the greater of the two kernel summations. The idea is to use the distance bounds obtained from tree nodes to bound $p_1(x)$ and $p_0(x)$: if at any point it can be proven that $\pi_1 p_1(x) > \pi_0 p_0(x)$ for all x in a query node, for example, then the actual class probabilities at those query points need not be evaluated.

Realizing this pruning idea most effectively motivates an alternate way of traversing the tree—a hybrid breadth and depth expansion pattern rather than the common depth-first traversal, where query nodes are expanded in a depth-first fashion and reference nodes are expanded in a breadth-first fashion. Pruning occurs when the upper and lower bounds are tight enough to achieve the correct classification, otherwise either the query node or all of the reference nodes are expanded. See [14] for further details.

9.4. K-Nearest-Neighbor Classifier

It is now easy to see the intuition behind one of the most widely used and powerful classifiers, the *nearest-neighbor* classifier: that is, just use the class label of the nearest point. The intuitive justification is that $p(y|x) \approx p(y|x')$ if x' is very close to x. It can also be understood as an approximation to kernel discriminant analysis where a variable-bandwidth (where the bandwidth is based on the distance to the nearest neighbor) kernel density estimate is used. The simplicity of K-nearest-neighbor is

that it assumes nothing about the form of the conditional density distribution, that is, it is completely nonparametric. The resulting decision boundary between the nearest-neighbor points is a Voronoi tessellation of the attribute space.

A smoothing parameter, the number of neighbors K, is typically used to regulate the complexity of the classification by acting as a smoothing of the data. In its simplest form a majority rule classification is adopted, where each of the K points votes on the classification. Increasing K decreases the variance in the classification but at the expense of an increase in the bias. Choosing K such that it minimizes the classification error rate can be achieved using cross-validation (see §8.11).

Weights can be assigned to the individual votes by weighting the vote by the distance to the nearest point, similar in spirit to kernel regression. In fact, the K-nearest-neighbor classifier is directly related to kernel regression discussed in §8.5, where the regressed value was weighted by a distance-dependent kernel.

In general a Euclidean distance is used for the distance metric. This can, however, be problematic when comparing attributes with no defined distance metric (e.g., comparing morphology with color). An arbitrary rescaling of any one of the axes can lead to a mixing of the data and alter the nearest-neighbor classification. Normalization of the features (i.e., scaling from [0–1]), weighting the importance of features based on cross-validation (including a 0/1 weighting which is effectively a feature selection), and use of the Mahalanobis distance $D(x, x_0) = (x - x_o)^T C^{-1}(x - x_0)$ which weights by the covariance of the data, are all approaches that have been adopted to account for this effect.

Like all nonparametric methods, nearest-neighbor classification works best when the number of samples is large; when the number of data is very small, parametric methods which "fill in the blanks" with model-based assumptions are often best. While it is simple to parallelize, the computational time for searching for the neighbors (even using kd-trees) can be expensive and particularly so for high-dimensional data sets. Sampling of the training samples as a function of source density can reduce the computational requirements.

Scikit-learn contains a fast K-neighbors classifier built on a ball-tree for fast neighbor searches:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

knc = KNeighborsClassifier(5) # use 5 nearest
      neighbors
knc.fit(X, y)
y_pred = knc.predict(X)
```

For more details see the AstroML documentation, or the source code of figure 9.7.

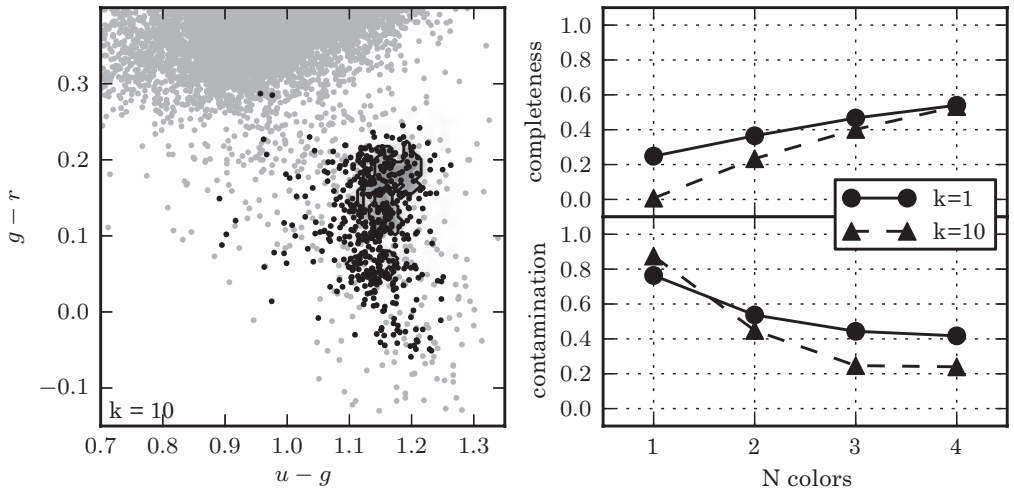

Figure 9.7. K-nearest-neighbor classification for RR Lyrae stars (see caption of figure 9.3 for details). Here the left panel shows the decision boundary for the model based on $K = 10$ neighbors, and the right panel shows the completeness and contamination for both $K = 1$ and $K = 10$. With all four colors and $K = 10$, K-neighbors classification achieves a completeness of 0.533 and a contamination of 0.240.

Figure 9.7 shows K-nearest-neighbor classification applied to the RR Lyrae data set with both $K = 1$ and $K = 10$. Note the complexity of the decision boundary, especially in regions where the density distributions overlap. This shows that even for our large data sets, K-nearest-neighbor may be overfitting the training data, leading to a suboptimal estimator. The discussion of overfitting in the context of regression (§8.11) is applicable here as well: KNN is unbiased, but is prone to very high variance when the parameter space is undersampled. This can be remedied by increasing the number of training points to better fill the space. When this is not possible, simple K-nearest-neighbor classification is probably not the best estimator, and other classifiers discussed in this chapter may provide a better solution.

9.5. Discriminative Classification

With nearest-neighbor classifiers we started to see a subtle transition—while clearly related to Bayes classifiers using variable-bandwidth kernel estimators, the class density estimates were skipped in favor of a simple classification decision. This is an example of *discriminative classification*, where we directly model the decision boundary between two or more classes of source. Recall, for $y \in \{0, 1\}$, the discriminant function is given by $g(x) = p(y = 1|x)$. Once we have it, no matter how we obtain it, we can use the rule

$$\widehat{y} = \begin{cases} 1 & \text{if } g(x) > 1/2, \\ 0 & \text{otherwise,} \end{cases} \tag{9.28}$$

to perform classification.

9.5.1. Logistic Regression

Logistic regression can be in the form of two (binomial) or more (multinomial) classes. For the initial discussion we will consider binomial logistic regression and consider the linear model

$$\begin{aligned} p(y=1|x) &= \frac{\exp\left[\sum_j \theta_j x^j\right]}{1+\exp\left[\sum_j \theta_j x^j\right]} \\ &= p(\boldsymbol{\theta}), \end{aligned} \tag{9.29}$$

where we define the logit function as

$$\text{logit}(p_i) = \log\left(\frac{p_i}{1-p_i}\right) = \sum_j \theta_j x_i^j. \tag{9.30}$$

The name logistic regression comes from the fact that the function $e^x/(1+e^x)$ is called the logistic function. Its name is due to its roots in regression, even though it is a method for classification.

Because y is binary, it can be modeled as a Bernoulli distribution (see §3.3.3), with (conditional) likelihood function

$$L(\beta) = \prod_{i=1}^{N} p_i(\beta)^{y_i}(1-p_i(\beta))^{1-y_i}. \tag{9.31}$$

Linear models we saw earlier under the generative (Bayes classifier) paradigm are related to logistic regression: linear discriminant analysis (LDA) uses the same model. In LDA,

$$\begin{aligned} \log\left(\frac{p(y=1|x)}{p(y=0|x)}\right) &= -\frac{1}{2}(\mu_0+\mu_1)^T\Sigma^{-1}(\mu_1-\mu_0) \\ &\quad + \log\left(\frac{\pi_0}{\pi_1}\right) + x^T\Sigma^{-1}(\mu_1-\mu_0) \\ &= \alpha_0 + \alpha^T x. \end{aligned} \tag{9.32}$$

In logistic regression the model is by assumption

$$\log\left(\frac{p(y=1|x)}{p(y=0|x)}\right) = \beta_0 + \beta^T x. \tag{9.33}$$

The difference is in how they estimate parameters—in logistic regression they are chosen to effectively minimize classification error rather than density estimation error.

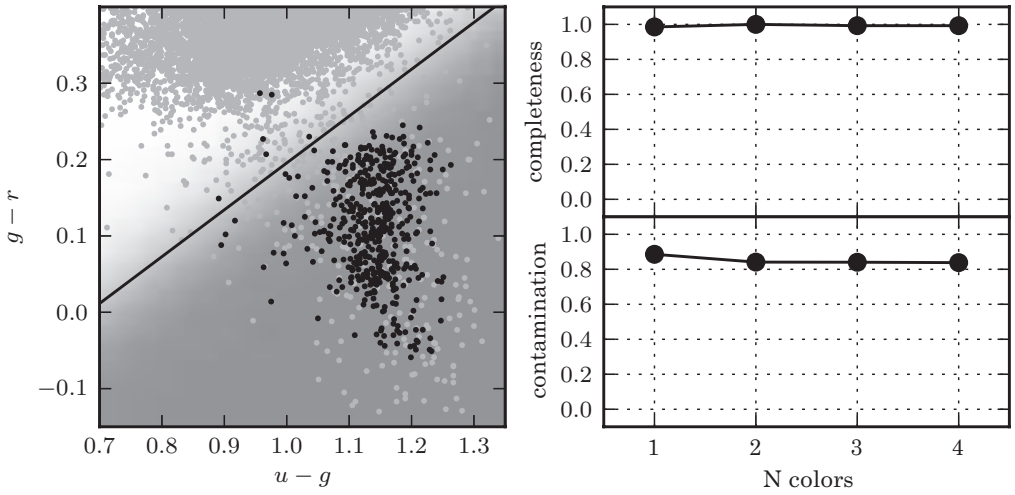

Figure 9.8. Logistic regression for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, logistic regression achieves a completeness of 0.993 and a contamination of 0.838.

Scikit-learn contains an implementation of logistic regression, which can be used as follows:

```
import numpy as np
from sklearn.linear_model import LogisticRegression

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

logr = LogisticRegression(penalty='l2')
logr.fit(X, y)
y_pred = logr.predict(X)
```

For more details see the Scikit-learn documentation, or the source code of figure 9.8.

Figure 9.8 shows an example of logistic regression on the RR Lyrae data sets.

9.6. Support Vector Machines

Now let us look at yet another way of choosing a linear decision boundary, which leads off in an entirely different direction, that of *support vector machines*.

Consider finding the hyperplane that maximizes the distance of the closest point from either class (see figure 9.9). We call this distance the *margin*. Points on the margin are called *support vectors*. Let us begin by assuming the classes are linearly separable. Here we will use $y \in \{-1, 1\}$, as it will make things notationally cleaner.

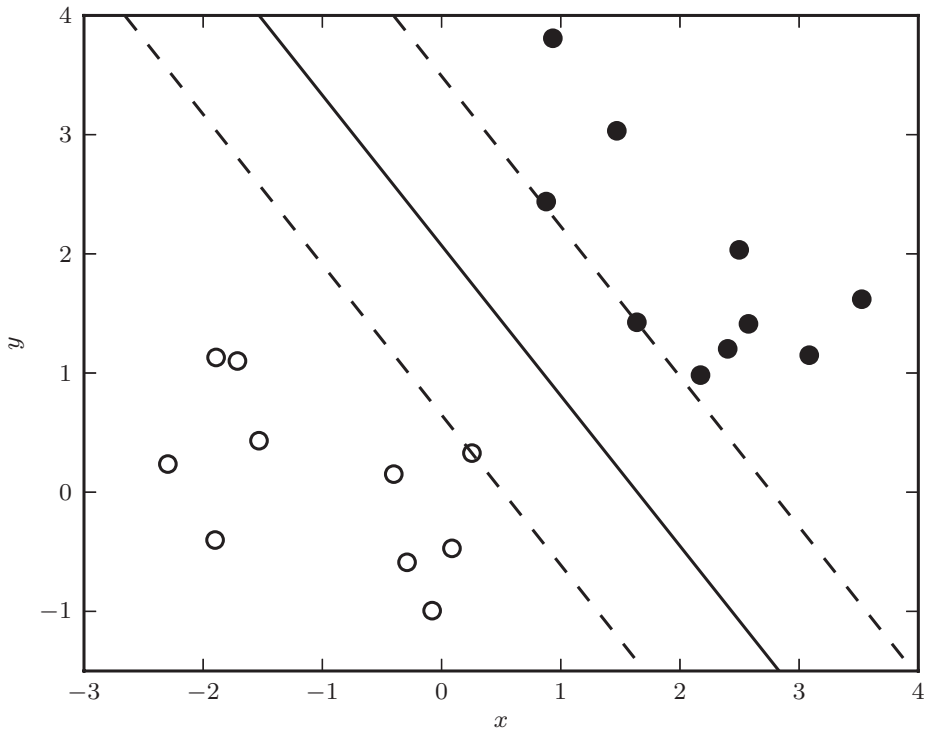

Figure 9.9. Illustration of SVM. The region between the dashed lines is the *margin*, and the points which the dashed lines touch are called the *support vectors*.

The hyperplane which maximizes the margin is given by finding

$$\max_{\beta_0,\beta}(m) \quad \text{subject to} \quad \frac{1}{\| \beta \|} y_i(\beta_0 + \beta^T x_i) \geq m \ \ \forall i. \tag{9.34}$$

Equivalently, the constraints can be written as $y_i(\beta_0 + \beta^T x_i) \geq m \parallel \beta \parallel$. Since for any β_0 and β satisfying these inequalities, any positively scaled multiple satisfies them too, we can arbitrarily set $\| \beta \| = 1/m$.

Thus the optimization problem is equivalent to minimizing

$$\frac{1}{2} \| \beta \| \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1 \ \ \forall i. \tag{9.35}$$

It turns out this optimization problem is a *quadratic programming* problem (quadratic objective function with linear constraints), a standard type of optimization problem for which methods exist for finding the global optimum. The theory of convex optimization tells us there is an equivalent way to write this optimization problem (its *dual formulation*).

Let $g^*(x)$ denote the optimal (maximum margin) hyperplane. Let $\langle x_i, x_{i'} \rangle$ denote the inner product of x_i and $x_{i'}$. Then

$$\beta_j^* = \sum_{i=1}^{N} \alpha_i \, y_i \, x_{ij}, \tag{9.36}$$

where α is the vector of weights that maximizes

$$\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{i'=1}^{N} \alpha_i \, \alpha_{i'} \, y_i \, y_{i'} \langle x_i, x_{i'} \rangle \tag{9.37}$$

$$\text{subject to} \quad \alpha_i \geq 0 \quad \text{and} \quad \sum_i \alpha_i y_i = 0. \tag{9.38}$$

For realistic problems, however, we must relax the assumption that the classes are linearly separable. In the primal formulation, instead of minimizing

$$\frac{1}{2} \| \beta \| \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1 \quad \forall i, \tag{9.39}$$

we will now minimize

$$\frac{1}{2} \| \beta \| \quad \text{subject to} \quad y_i(\beta_0 + \beta^T x_i) \geq 1 - \xi_i \quad \forall i, \tag{9.40}$$

where the ξ_i are called *slack variables* and we limit the amount of slack by adding the constraints

$$\xi_i \geq 0 \quad \text{and} \quad \sum_i \xi_i \leq C. \tag{9.41}$$

This effectively bounds the total number of misclassifications at C, which becomes a tuning parameter of the support vector machine. The points x_i for which $\alpha_i \neq 0$ are the support vectors.

The discriminant function can be rewritten as

$$g(x) = \beta_0 + \sum_{i=1}^{N} \alpha_i \, y_i \, \langle x, x_i \rangle \tag{9.42}$$

and the final classification rule is $\widehat{c}(x) = \text{sgn}[g(x)]$.

It turns out the SVM optimization is equivalent to minimizing

$$\sum_{i=1}^{N} (1 - y_i \, g(x_i))_+ + \lambda \, \| \beta \|^2, \tag{9.43}$$

where λ is related to the tuning parameter C and the index $+$ stands for $x_+ = \max(0, x)$. Notice the similarity here to the ridge regularization discussed in §8.3.1: the tuning parameter λ controls the strength of an L_2 regularization over the parameters β.

Figure 9.10 shows the SVM decision boundary computed for the RR Lyrae data sets. Note that because SVM uses a metric which maximizes the margin rather than a measure over all points in the data sets, it is in some sense similar in spirit to the rank-based estimators discussed in chapter 3. The median of a distribution is unaffected

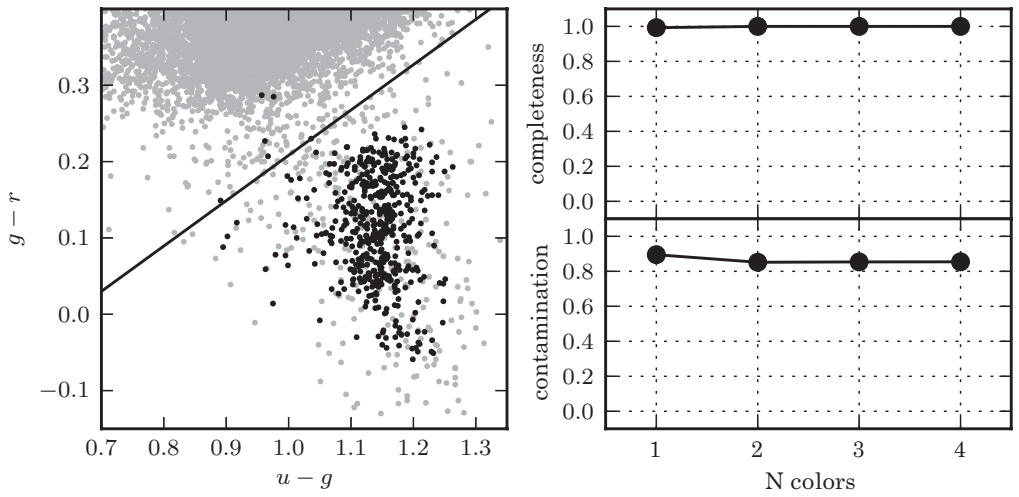

Figure 9.10. SVM applied to the RR Lyrae data (see caption of figure 9.3 for details). With all four colors, SVM achieves a completeness of 1.0 and a contamination of 0.854.

by even large perturbations of outlying points, as long as those perturbations do not cross the median. In the same way, once the support vectors are determined, changes to the positions or numbers of points beyond the margin will not change the decision boundary. For this reason, SVM can be a very powerful tool for discriminative classification. This is why figure 9.10 shows such a high completeness compared to the other methods discussed above: it is not swayed by the fact that the background sources outnumber the RR Lyrae stars by a factor of ∼200 to 1: it simply determines the best boundary between the small RR Lyrae clump and the large background clump. This completeness, however, comes at the cost of a relatively large contamination level.

Scikit-learn includes a fast SVM implementation for both classification and regression tasks. The SVM classifier can be used as follows:

```
import numpy as np
from sklearn.svm import LinearSVC

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

model = LinearSVC(loss='l2')
model.fit(X, y)
y_pred = model.predict(X)
```

For more details see the Scikit-learn documentation, or the source code of figure 9.11.

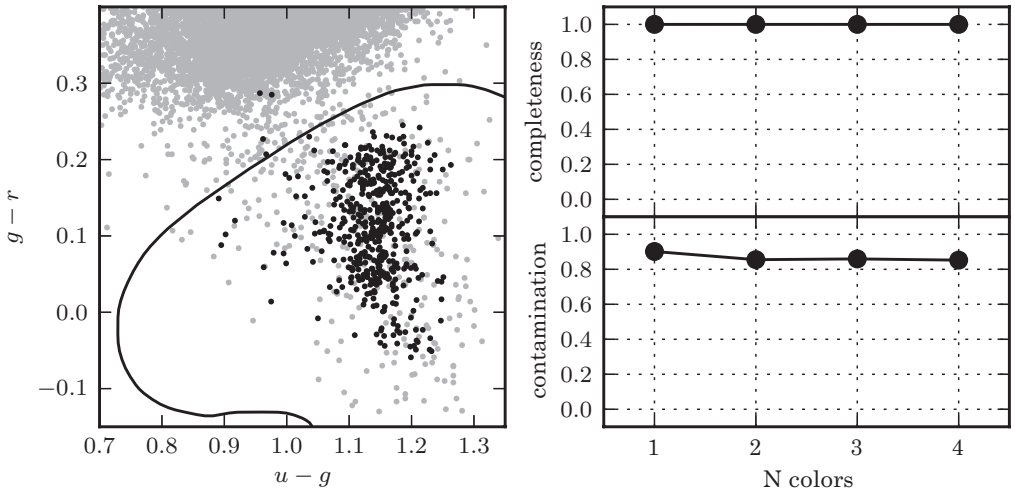

Figure 9.11. Kernel SVM applied to the RR Lyrae data (see caption of figure 9.3 for details). This example uses a Gaussian kernel with $\gamma = 20$. With all four colors, kernel SVM achieves a completeness of 1.0 and a contamination of 0.852.

One major limitation of SVM is that it is limited to linear decision boundaries. The idea of *kernelization* is a simple but powerful way to take a support vector machine and make it nonlinear—in the dual formulation, one simply replaces each occurrence of $\langle x_i, x_{i'} \rangle$ with a kernel function $K(x_i, x_{i'})$ with certain properties which allow one to think of the SVM as operating in a higher-dimensional space. One such kernel is the Gaussian kernel

$$K(x_i, x_{i'}) = e^{-\gamma ||x_i - x_{i'}||^2}, \tag{9.44}$$

where γ is a parameter to be learned via cross-validation. An example of applying kernel SVM to the RR Lyrae data is shown in figure 9.11. This nonlinear classification improves over the linear version only slightly. For this particular data set, the contamination is not driven by nonlinear effects.

9.7. Decision Trees

The decision boundaries that we discussed in §9.5 can be applied hierarchically to a data set. This observation leads to a powerful methodology for classification that is known as the *decision tree*. An example decision tree used for the classification of our RR Lyrae stars is shown in figure 9.12. As with the tree structures described in §2.5.2, the top node of the decision tree contains the entire data set. At each branch of the tree these data are subdivided into two child nodes (or subsets), based on a predefined decision boundary, with one node containing data below the decision boundary and the other node containing data above the decision boundary. The boundaries themselves are usually axis aligned (i.e., the data are split along one feature at each level of the tree). This splitting process repeats, recursively, until we achieve a predefined stopping criteria (see §9.7.1).

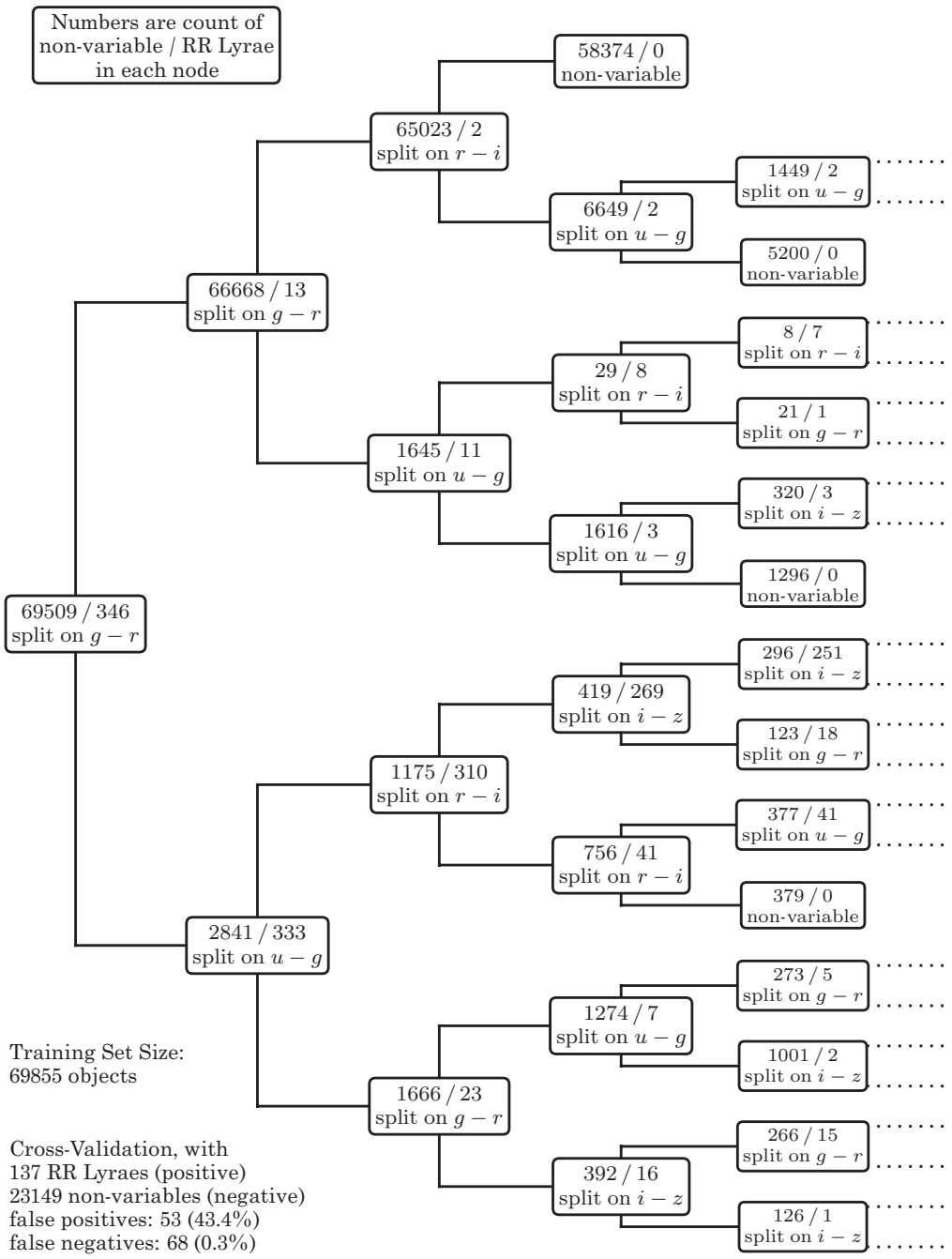

Figure 9.12. The decision tree for RR Lyrae classification. The numbers in each node are the statistics of the *training* sample of ∼70,000 objects. The cross-validation statistics are shown in the bottom-left corner of the figure. See also figure 9.13.

For the two-class decision tree shown in figure 9.12, the tree has been learned from a training set of standard stars (§1.5.8), and RR Lyrae variables with known classifications. The terminal nodes of the tree (often referred to as "leaf nodes") record the fraction of points contained within that node that have one classification or the other, that is, the fraction of standard stars or RR Lyrae.

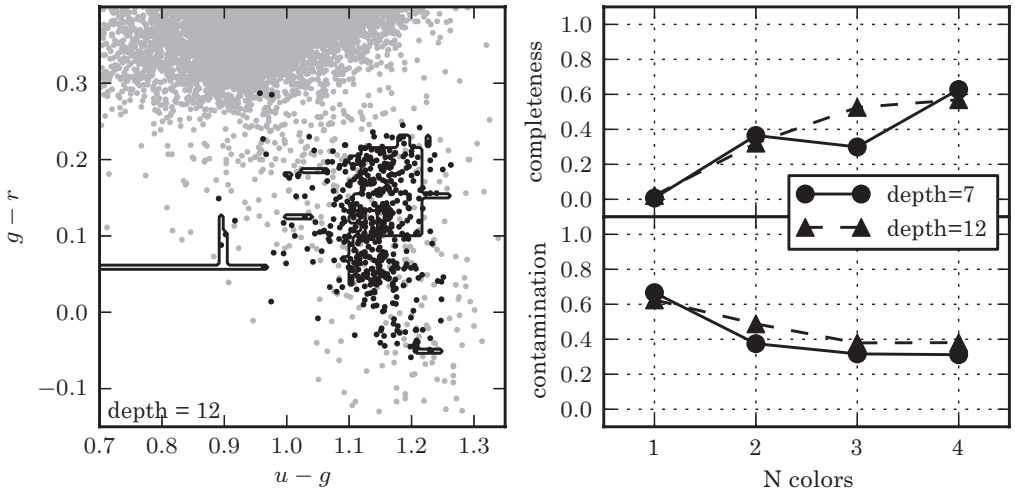

Figure 9.13. Decision tree applied to the RR Lyrae data (see caption of figure 9.3 for details). This example uses tree depths of 7 and 12. With all four colors, this decision tree achieves a completeness of 0.569 and a contamination of 0.386.

Scikit-learn includes decision-tree implementations for both classification and regression. The decision-tree classifier can be used as follows:

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

model = DecisionTreeClassifier(max_depth=6)
model.fit(X, y)
y_pred = model.predict(X)
```

For more details see the Scikit-learn documentation, or the source code of figure 9.11.

The result of the full decision tree as a function of the number of features used is shown in figure 9.13. This classification method leads to a completeness of 0.569 and a contamination of 0.386. The depth of the tree also has an effect on the precision and accuracy. Here, going to a depth of 12 (with a maximum of $2^{12} = 4096$ nodes) slightly overfits the data: it divides the parameter space into regions which are too small. Using fewer nodes prevents this, and leads to a better classifier.

Application of the tree to classifying data is simply a case of following the branches of the tree through a series of binary decisions (one at each level of the tree) until we reach a leaf node. The relative fraction of points from the training set classified as one class or the other defines the class associated with that leaf node.

Decision trees are, therefore, classifiers that are simple, and easy to visualize and interpret. They map very naturally to how we might interrogate a data set by hand (i.e., a hierarchy of progressively more refined questions).

9.7.1. Defining the Split Criteria

In order to build a decision tree we must choose the feature and value on which we wish to split the data. Let us start by considering a simple split criteria based on the information content or entropy of the data; see [11]. In §5.2.2, we define the entropy, $E(x)$, of a data set, x, as

$$E(x) = -\sum_i p_i(x) \ln(p_i(x)), \tag{9.45}$$

where i is the class and $p_i(x)$ is the probability of that class given the training data. We can define information gain as the reduction in entropy due to the partitioning of the data (i.e., the difference between the entropy of the parent node and the sum of entropies of the child nodes). For a binary split with $i = 0$ representing those points below the split threshold and $i = 1$ for those points above the split threshold, the information gain, $IG(x)$, is

$$IG(x|x_i) = E(x) - \sum_{i=0}^{1} \frac{N_i}{N} E(x_i), \tag{9.46}$$

where N_i is the number of points, x_i, in the ith class, and $E(x_i)$ is the entropy associated with that class (also known as Kullback–Leibler divergence in the machine learning community).

Finding the optimal decision boundary on which to split the data is generally considered to be a computationally intractable problem. The search for the split is, therefore, undertaken in a greedy fashion where each feature is considered one at a time and the feature that provides the largest information gain is split. The value of the feature at which to split the data is defined in an analogous manner, whereby we sort the data on feature i and maximize the information gain for a given split point, s,

$$IG(x|s) = E(x) - \arg\max_s \left(\frac{N(x|x < s)}{N} E(x|x < s) - \frac{N(x|x \geq s)}{N} E(x|x \geq s) \right). \tag{9.47}$$

Other loss functions common in decision trees include the Gini coefficient (see §4.7.2) and the misclassification error. The Gini coefficient estimates the probability that a source would be incorrectly classified if it was chosen at random from a data set and the label was selected randomly based on the distribution of classifications within the data set. The Gini coefficient, G, for a k-class sample is given by

$$G = \sum_i^k p_i(1 - p_i), \tag{9.48}$$

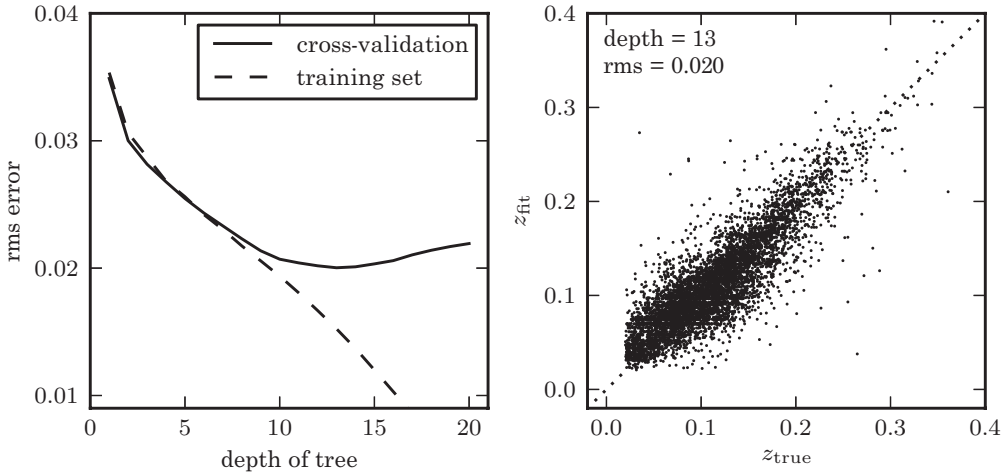

Figure 9.14. Photometric redshift estimation using decision-tree regression. The data is described in §1.5.5. The training set consists of u, g, r, i, z magnitudes of 60,000 galaxies from the SDSS spectroscopic sample. Cross-validation is performed on an additional 6000 galaxies. The left panel shows training error and cross-validation error as a function of the maximum depth of the tree. For a number of nodes $N > 13$, overfitting is evident.

where p_i is the probability of finding a point with class i within a data set. The misclassification error, MC, is the fractional probability that a point selected at random will be misclassified and is defined as

$$MC = 1 - \max_i(p_i). \tag{9.49}$$

The Gini coefficient and classification error are commonly used in classification trees where the classification is categorical.

9.7.2. Building the Tree

In principle, the recursive splitting of the tree could continue until there is a single point per node. This is, however, inefficient as it results in $\mathcal{O}(N)$ computational cost for both the construction and traversal of the tree. A common criterion for stopping the recursion is, therefore, to cease splitting the nodes when either a node contains only one class of object, when a split does not improve the information gain or reduce the misclassifications, or when the number of points per node reaches a predefined value.

As with all model fitting, as we increase the complexity of the model we run into the issue of overfitting the data. For decision trees the complexity is defined by the number of levels or depth of the tree. As the depth of the tree increases, the error on the training set will decrease. At some point, however, the tree will cease to represent the correlations within the data and will reflect the noise within the training set. We can, therefore, use the cross-validation techniques introduced in §8.11 and either the entropy, Gini coefficient, or misclassification error to optimize the depth of the tree. Figure 9.14 illustrates this cross-validation using a decision tree that predicts photometric redshifts. For a training sample of approximately 60,000 galaxies, with

the rms error in estimated redshift used as the misclassification criterion, the optimal depth is 13. For this depth there are roughly $2^{13} \approx 8200$ leaf nodes. Splitting beyond this level leads to overfitting, as evidenced by an increased cross-validation error.

A second approach for controlling the complexity of the tree is to grow the tree until there are a predefined number of points in a leaf node (e.g., five) and then use the cross-validation or test data set to prune the tree. In this method we take a greedy approach and, for each node of the tree, consider whether terminating the tree at that node (i.e., making it a leaf node and removing all subsequent branches of the tree) improves the accuracy of the tree. Pruning of the decision tree using an independent test data set is typically the most successful of these approaches. Other approaches for limiting the complexity of a decision tree include random forests (see §9.7.3), which effectively limits the number of attributes on which the tree is constructed.

9.7.3. Bagging and Random Forests

Two of the most successful applications of *ensemble learning* (the idea of combining the outputs of multiple models through some kind of voting or averaging) are those of *bagging* and *random forests* [1]. Bagging (from bootstrap aggregation) averages the predictive results of a series of bootstrap samples (see §4.5) from a training set of data. Often applied to decision trees, bagging is applicable to regression and many nonlinear model fitting or classification techniques. For a sample of N points in a training set, bagging generates K equally sized bootstrap samples from which to estimate the function $f_i(x)$. The final estimator, defined by bagging, is then

$$f(x) = \frac{1}{K} \sum_{i}^{K} f_i(x). \tag{9.50}$$

Random forests expand upon the bootstrap aspects of bagging by generating a set of decision trees from these bootstrap samples. The features on which to generate the tree are selected at random from the full set of features in the data. The final classification from the random forest is based on the averaging of the classifications of each of the individual decision trees. In so doing, random forests address two limitations of decision trees: the overfitting of the data if the trees are inherently deep, and the fact that axis-aligned partitioning of the data does not accurately reflect the potentially correlated and/or nonlinear decision boundaries that exist within data sets.

In generating a random forest we define n, the number of trees that we will generate, and m, the number of attributes that we will consider splitting on at each level of the tree. For each decision tree a subsample (bootstrap sample) of data is selected from the full data set. At each node of the tree, a set of m variables are randomly selected and the split criteria is evaluated for each of these attributes; a different set of m attributes are used for each node. The classification is derived from the mean or mode of the results from all of the trees. Keeping m small compared to the number of features controls the complexity of the model and reduces the concerns of overfitting.

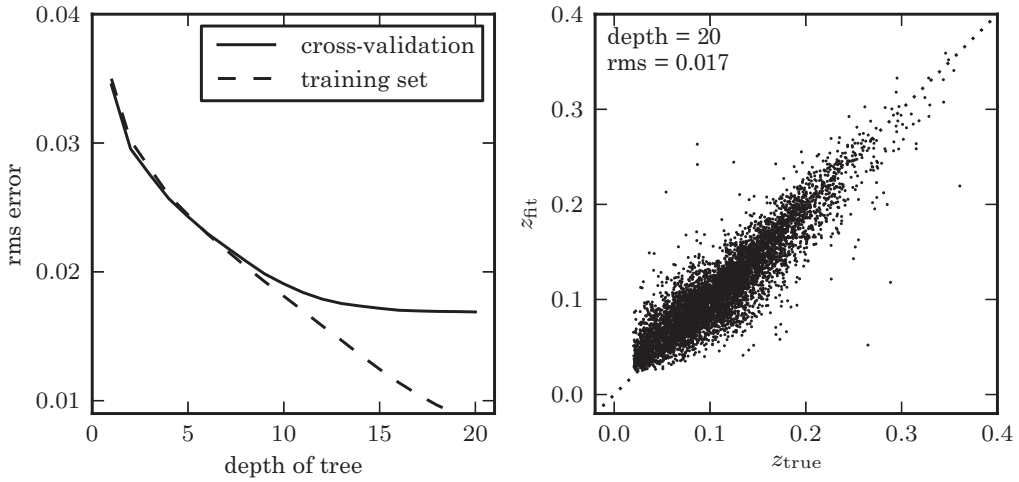

Figure 9.15. Photometric redshift estimation using random forest regression, with ten random trees. Comparison to figure 9.14 shows that random forests correct for the overfitting evident in very deep decision trees. Here the optimal depth is 20 or above, and a much better cross-validation error is achieved.

Scikit-learn contains a random forest implementation which can be used for classification or regression. For example, classification tasks can be approached as follows:

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier

X = np.random.random((100, 2)) # 100 pts in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

model = RandomForestClassifier(10)
# forest of 10 trees
model.fit(X, y)
y_pred = model.predict(X)
```

For more details see the Scikit-learn documentation, or the source code of figure 9.15.

Figure 9.15 demonstrates the application of a random forest of regression trees to photometric redshift data (using a forest of ten random trees—see [2] for a more detailed discussion). The left panel shows the cross-validation results as a function of the depth of each tree. In comparison to the results for a single tree (figure 9.14), the use of randomized forests reduces the effect of overfitting and leads to a smaller rms error.

Similar to the cross-validation technique used to arrive at the optimal depth of the tree, cross-validation can also be used to determine the number of trees, n, and the number of random features m, simply by optimizing over all free parameters. With

random forests, n is typically increased until the cross-validation error plateaus, and m is often chosen to be $\sim \sqrt{K}$, where K is the number of attributes in the sample.

9.7.4. Boosting Classification

Boosting is an ensemble approach that was motivated by the idea that combining many weak classifiers can result in an improved classification. This idea differs fundamentally from that illustrated by random forests: rather than create the models separately on different data sets, which can be done all in parallel, boosting creates each new model to attempt to correct the errors of the ensemble so far. At the heart of boosting is the idea that we reweight the data based on how incorrectly the data were classified in the previous iteration.

In the context of classification (boosting is also applicable in regression) we can run the classification multiple times and each time reweight the data based on the previous performance of the classifier. At the end of this procedure we allow the classifiers to vote on the final classification. The most popular form of boosting is that of adaptive boosting [4]. For this case, imagine that we had a weak classifier, $h(x)$, that we wish to apply to a data set and we want to create a strong classifier, $f(x)$, such that

$$f(x) = \sum_{m}^{K} \theta_m h_m(x), \tag{9.51}$$

where m indicates the number of the iteration of the weak classifier and θ_m is the weight of the mth iteration of the classifier.

If we start with a set of data, x, with known classifications, y, we can assign a weight, $w_m(x)$, to each point (where the initial weight is uniform, $1/N$, for the N points in the sample). After the application of the weak classifier, $h_m(x)$, we can estimate the classification error, e_m, as

$$e_m = \sum_{i=1}^{N} w_m(x_i) I(h_m(x_i) \neq y_i), \tag{9.52}$$

where $I(h_m(x_i) \neq y_i)$ is the indicator function (with $I(h_m(x_i) \neq y_i)$ equal to 1 if $h_m(x_i) \neq y_i$ and equal to 0 otherwise). From this error we define the weight of that iteration of the classifier as

$$\theta_m = \frac{1}{2} \log \left(\frac{1 - e_m}{e_m} \right) \tag{9.53}$$

and update the weights on the points,

$$w_{m+1}(x_i) = w_m(x_i) \times \begin{cases} e^{-\theta_m} & \text{if } h_m(x_i) = y_i, \\ e^{\theta_m} & \text{if } h_m(x_i) \neq y_i, \end{cases} \tag{9.54}$$

$$= \frac{w_m(x_i) e^{-\theta_m y_i h_m(x_i)}}{\sum_{i=1}^{N} w_m(x_i) e^{-\theta_m y_i h_m(x_i)}}. \tag{9.55}$$

The effect of updating $w(x_i)$ is to increase the weight of the misclassified data. After K iterations the final classification is given by the weighted votes of each classifier given by eq. 9.51. As the total error, e_m, decreases, the weight of that iteration in the final classification increases.

A fundamental limitation of the boosted decision tree is the computation time for large data sets. Unlike random forests, which can be trivially parallelized, boosted decision trees rely on a chain of classifiers which are each dependent on the last. This may limit their usefulness on very large data sets. Other methods for boosting have been developed such as gradient boosting; see [5]. Gradient boosting involves approximating a steepest descent criterion after each simple evaluation, such that an additional weak classification can improve the classification score and may scale better to larger data sets.

Scikit-learn contains several flavors of boosted decision trees, which can be used for classification or regression. For example, boosted classification tasks can be approached as follows:

```
import numpy as np
from sklearn.ensemble import
GradientBoostingClassifier

X = np.random.random((100, 2)) # 2 pts in 100 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
    # simple division

model = GradientBoostingClassifier()
model.fit(X, y)
y_pred = model.predict(X)
```

For more details see the Scikit-learn documentation, or the source code of figure 9.16.

Figure 9.16 shows the results for a gradient-boosted decision tree for the SDSS photometric redshift data. For the weak estimator, we use a decision tree with a maximum depth of 3. The cross-validation results are shown as a function of boosting iteration. By 500 steps, the cross-validation error is beginning to level out, but there are still no signs of overfitting. The fact that the training error and cross-validation error remain very close indicates that a more complicated model (i.e., deeper trees or more boostings) would likely allow improved errors. Even so, the rms error recovered with these suboptimal parameters is comparable to that of the random forest classifier.

9.8. Evaluating Classifiers: ROC Curves

Comparing the performance of classifiers is an important part of choosing the best classifier for a given task. “Best” in this case can be highly subjective: for some

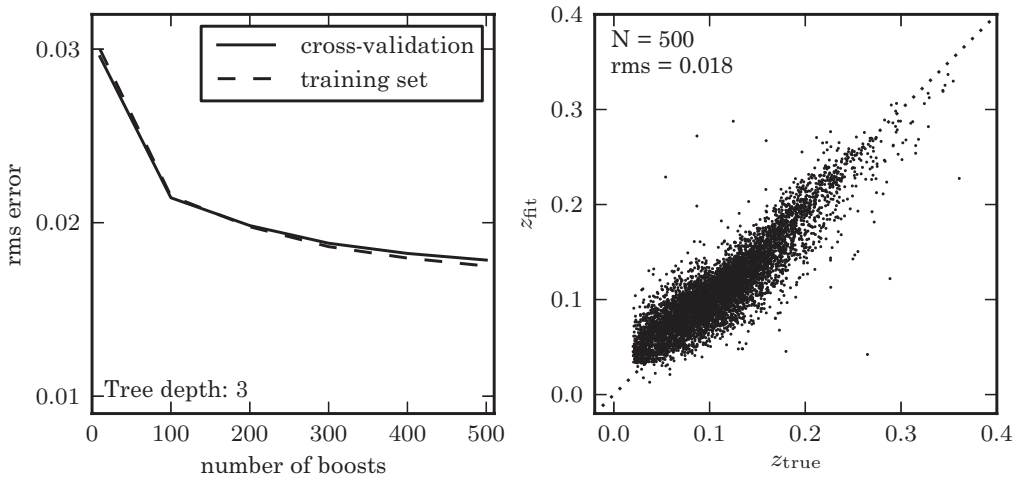

Figure 9.16. Photometric redshift estimation using gradient-boosted decision trees, with 100 boosting steps. As with random forests (figure 9.15), boosting allows for improved results over the single tree case (figure 9.14). Note, however, that the computational cost of boosted decision trees is such that it is computationally prohibitive to use very deep trees. By stringing together a large number of very naive estimators, boosted trees improve on the underfitting of each individual estimator.

problems, one might wish for high completeness at the expense of contamination; at other times, one might wish to minimize contamination at the expense of completeness. One way to visualize this is to plot receiver operating characteristic (ROC) curves (see §4.6.1). An ROC curve usually shows the true-positive rate as a function of the false-positive rate as the discriminant function is varied. How the function is varied depends on the model: in the example of Gaussian naive Bayes, the curve is drawn by classifying data using relative probabilities between 0 and 1.

A set of ROC curves for a selection of classifiers explored in this chapter is shown in the left panel of figure 9.17. The curves closest to the upper left of the plot are the best classifiers: for the RR Lyrae data set, the ROC curve indicates that GMM Bayes and K-nearest-neighbor classification outperform the rest. For such an unbalanced data set, however, ROC curves can be misleading. Because there are fewer than five sources for every 1000 background objects, a false-positive rate of even 0.05 means that false positives outnumber true positives ten to one! When sources are rare, it is often more informative to plot the efficiency (equal to one minus the contamination, eq. 9.5) vs. the completeness (eq. 9.5). This can give a better idea of how well a classifier is recovering rare data from the background.

The right panel of figure 9.17 shows the completeness vs. efficiency for the same set of classifiers. A striking feature is that the simpler classifiers reach a maximum efficiency of about 0.25: this means that at their best, only 25% of objects identified as RR Lyrae are actual RR Lyrae. By the completeness–efficiency measure, the GMM Bayes model outperforms all others, allowing for higher completeness at virtually any efficiency level. We stress that this is not a general result, and that the best classifier for any task depends on the precise nature of the data.

As an example where the ROC curve is a more useful diagnostic, figure 9.18 shows ROC curves for the classification of stars and quasars from four-color photometry (see the description of the data set in §9.1). The stars and quasars in

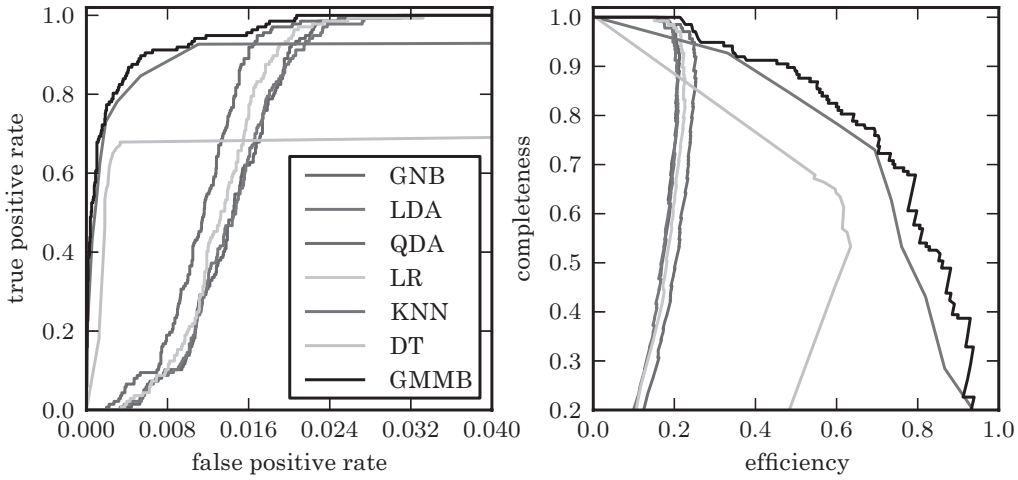

Figure 9.17. ROC curves (left panel) and completeness–efficiency curves (right panel) for the four-color RR Lyrae data using several of the classifiers explored in this chapter: Gaussian naive Bayes (GNB), linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), logistic regression (LR), K-nearest-neighbor classification (KNN), decision tree classification (DT), and GMM Bayes classification (GMMB). See color plate 7.

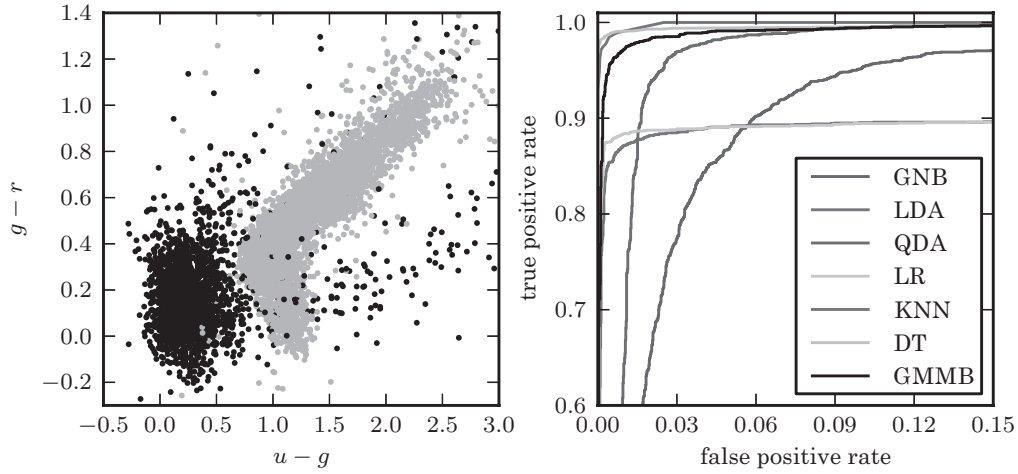

Figure 9.18. The left panel shows data used in color-based photometric classification of stars and quasars. Stars are indicated by gray points, while quasars are indicated by black points. The right panel shows ROC curves for quasar identification based on $u - g, g - r, r - i$, and $i - z$ colors. Labels are the same as those in figure 9.17. See color plate 8.

this sample are selected with differing selection functions: for this reason, the data set does not reflect a realistic sample. We use it for purposes of illustration only. The stars outnumber the quasars by only a factor of 3, meaning that a false-positive rate of 0.3 corresponds to a contamination of ∼50%. Here we see that the best-performing classifiers are the neighbors-based and tree-based classifiers, both of which approach 100% true positives with a very small number of false positives. An interesting feature is that classifiers with linear discriminant functions (LDA and logistic regression) plateau at a true-positive rate of 0.9. These simple classifiers, while useful in some situations, do not adequately explain these photometric data.

Scikit-learn has some built-in tools for computing ROC curves and completeness–efficiency curves (known as precision-recall curves in the machine learning community). They can be used as follows:

```
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

X = np.random.random((100, 2))# 100 points in 2 dims
y = (X[:, 0] + X[:, 1] > 1).astype(int)
  # simple boundary

gnb = GaussianNB().fit(X, y)
prob = gnb.predict_proba(X)

# Compute precision / recall curve
pr, re, thresh = metrics.precision_recall_curve
        (y, prob[:, 0])

# Compute ROC curve:true positives / false positives
tpr, fpr, thresh = metrics.roc_curve(y, prob[:, 0])
```

The thresholds are automatically determined based on the probability levels within the data. For more information, see the Scikit-learn documentation.

9.9. Which Classifier Should I Use?

Continuing as we have in previous chapters, we will answer this question by decomposing the notion of "best" along the axes of "accuracy," "interpretability," "simplicity," and "speed."

What are the most *accurate* classifiers? A bit of thought will lead to the obvious conclusion that no single type of model can be known in advance to be the best classifier for all possible data sets, as each data set can have an arbitrarily different underlying distribution. This is famously formalized in the no free lunch theorem [17]. The conclusion is even more obvious for regression, where the same theorem applies, though it was popularized in the context of classification. However, we can still draw upon a few useful rules of thumb.

As we have said in the summaries of earlier chapters, in general, the more parameters a model has, the more complex a function it can fit (whether that be the probability density functions, in the case of a generative classifier, or the decision boundary, in the case of a discriminative classifier), and thus the more likely it is to yield high predictive accuracy. Parametric methods, which have a fixed number of parameters with respect to the number of data points N, include (roughly in increasing order of typical accuracy) naive Bayes, linear discriminant

analysis, logistic regression, linear support vector machines, quadratic discriminant analysis, and linear ensembles of linear models. Nonparametric methods, which have a number of parameters that grows as the number of data points N grows, include (roughly in order of typical accuracy) decision trees, K-nearest-neighbor, neural networks, kernel discriminant analysis, kernelized support vector machines, random forests, and boosting with nonlinear methods (a model like K-nearest-neighbor for this purpose is considered to have $\mathcal{O}(N)$ parameters, as the model consists of the training points themselves—however, there is only one parameter, K, that in practice is *optimized* through cross-validation). While there is no way to know for sure which method is best for a given data set until it is tried empirically, generally speaking the most accurate method is most likely to be nonparametric.

Some models including neural networks and GMM Bayes classifiers are parametric for a fixed number of hidden units or components, but if that number is chosen in a way that can grow with N, such a model becomes nonparametric. The practical complication of trying every possible number of parameters typically prevents a disciplined approach to model selection, making such approaches difficult to call either nonparametric or strictly parametric. Among the nonparametric methods, the ensemble methods (e.g., bagging, boosting) are effectively models with many more parameters, as they combine hundreds or thousands of base models. Thus, as we might expect based on our general rule of thumb, an ensemble of models will generally yield the highest possible accuracy.

The generally simpler parametric methods can shine in some circumstances, however. When the dimensionality is very high compared to the number of data points (as is typical in text data where for each document the features are the counts of each of thousands of words, or in bioinformatics data where for each patient the features are the expression levels of each of thousands of genes), the points in such a space are effectively so dispersed that a linear hyperplane can (at least mostly) separate them. Linear classifiers have thus found great popularity in such applications, though this situation is fairly atypical in astronomy problems. Another setting that favors parametric methods is that of low sample size. When there are few data points, a simple function is all that is needed to achieve a good fit, and the principle of Occam's razor (§5.4.2) dictates favoring the simplest model that works. In such a regime, the paucity of data points makes the ability of domain knowledge-based assumptions to fill in the gaps compelling, leading one toward Bayesian approaches and carefully hand-constructed parametric models rather than relatively "blind," or assumption-free nonparametric methods. Hierarchical modeling, as possibly assisted by the formalism of Bayesian networks, becomes the mode of thinking in this regime.

Complexity control in the form of model selection is more critical for nonparametric methods—it is important to remember that the accuracy benefits of such methods are only realized assuming that model selection is done properly, via cross-validation or other measures (in order to avoid being misled by overly optimistic-looking accuracies). Direct comparisons between different ML methods, which are most common in the context of classification, can also be misleading when the amount of human and/or computational effort spent on obtaining the best parameter settings was not equal between methods. This is typically the case, for example, in a paper proposing one method over others.

What are the most *interpretable* classifiers? In many cases we would like to know *why* the classifier made the decision it did, or in general, what sort of discriminatory pattern the classifier has found in general, for example, which dimensions are the primary determinants of the final prediction. In general, this is where parametric methods tend to be the most useful. Though it is certainly possible to make a complex and unintelligible parametric model, for example by using the powerful general machinery of graphical models, the most popular parametric methods tend to be simple and easy to reason about. In particular, in a linear model, the coefficients on the dimensions can be interpreted to indicate the importance of each variable in the model.

Nonparametric methods are often too large to be interpretable, as they typically scale in size (number of parameters) with the number of data points. However, among nonparametric methods, certain ones can be intepretable, in different senses. A decision tree can be explained in plain English terms, by reading paths from the root to each leaf as a rule containing if-then tests on the variables (see, e.g., figure 9.12). The interpretability of decision trees was used to advantage in understanding star–galaxy classification [3]. A nearest-neighbor classifier's decisions can be understood by simply examining the k neighbors returned, and their class labels. A probabilistic classifier which explicitly implements Bayes' rule, such as kernel discriminant analysis, can be explained in terms of the class under which the test point was more likely—an explanation that is typically natural for physicists in particular. Neural networks, kernelized support vector machines, and ensembles such as random forests and boosting are among the least interpretable methods.

What are the most *scalable* classifiers? Naive Bayes and its variants are by far the easiest to compute, requiring in principle only one pass through the data. Learning a logistic regression model via standard unconstrained optimization methods such as conjugate gradient requires only a modest number of relatively cheap iterations through the data. Though the model is still linear, linear support vector machines are more expensive, though several fast algorithms exist. We have discussed K-nearest-neighbor computations in §2.5.2, but K-nearest-neighbor *classification* is in fact a slightly easier problem, and this can be exploited algorithmically. Kernel discriminant analysis can also be sped up by fast tree-based algorithms, reducing its cost from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Decision trees are relatively efficient, requiring $\mathcal{O}(N \log N)$ time to build and $\mathcal{O}(\log N)$ time to classify, as are neural networks. Random forests and boosting with trees simply multiplies the cost of decision trees by the number of trees, which is typically very large, but this can be easily parallelized. Kernelized support vector machines are $\mathcal{O}(N^3)$ in the worst case, and have resisted attempts at appreciably fast algorithms. Note that the need to use fast algorithms ultimately counts against a method in terms of its simplicity, at least in implementation.

What are the *simplest* classifiers? Naive Bayes classifiers are possibly the simplest in terms of both implementation and learning, requiring no tuning parameters to be tried. Logistic regression and decision tree are fairly simple to implement, with no tuning parameters. K-nearest-neighbor classification and KDA are simple methods that have only one critical parameter, and cross-validation over it is generally straightforward. Kernelized support vector machines also have only one

TABLE 9.1.
Summary of the practical properties of different classifiers.

Method	Accuracy	Interpretability	Simplicity	Speed
Naive Bayes classifier	L	H	H	H
Mixture Bayes classifier	M	H	H	M
Kernel discriminant analysis	H	H	H	M
Neural networks	H	L	L	M
Logistic regression	L	M	H	M
Support vector machines: linear	L	M	M	M
Support vector machines: kernelized	H	L	L	L
K-nearest-neighbor	H	H	H	M
Decision trees	M	H	H	M
Random forests	H	M	M	M
Boosting	H	L	L	L

or two critical parameters which are easy to cross-validate over, though the method is not as simple to understand as KNN or KDA. Mixture Bayes classifiers inherit the properties of Gaussian mixtures discussed in §6.6 (i.e., they are fiddly). Neural networks are perhaps the most fiddly methods. The onus of having to store and manage the typically hundreds or thousands of trees in random forests counts against it, though it is conceptually simple.

Other considerations, and taste. If obtaining good class probability estimates rather than just the correct class labels is of importance, KDA is a good choice as it is based on the most powerful approach for density estimation, kernel density estimation. For this reason, as well as its state-of-the-art accuracy and ability to be computed efficiently, once a fast KDA algorithm had been developed (see [14]), it produced a dramatic leap in the size and quality of quasar catalogs [12, 13], and associated scientific results [6, 15]. Optimization-based approaches, such as logistic regression, can typically be augmented to handle missing values, whereas distance-based methods such as K-nearest-neighbor, kernelized SVMs, and KDA cannot. When both continuous and discrete attributes are present, decision trees are one of the few methods that seamlessly handles both without modification.

Simple summary. Using our axes of "accuracy," "interpretability," "simplicity," and "speed," a summary of each of the methods considered in this chapter, in terms of high (H), medium (M), and low (L) is given in table 9.1.

Note that it is hard to say in advance how a method will behave in terms of accuracy, for example, on a particular data set—and no method, in general, always beats other methods. Thus our table should be considered an initial guideline based on extensive experience.

References

[1] Breiman, L. (2001). Random forests. *Mach. Learn. 45*(1), 5–32.

[2] Carliles, S., T. Budavári, S. Heinis, C. Priebe, and A. S. Szalay (2010). Random forests for photometric redshifts. *ApJ 712*, 511–515.

[3] Fayyad, U. M., N. Weir, and S. G. Djorgovski (1993). SKICAT: A machine learning system for automated cataloging of large scale sky surveys. In *International Conference on Machine Learning (ICML)*, pp. 112–119.

[4] Freund, Y. and R. E. Schapire (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci. 55*(1), 119–139. Special issue for EuroCOLT '95.

[5] Friedman, J. H. (2000). Greedy function approximation: A gradient boosting machine. *Annals of Statistics 29*, 1189–1232.

[6] Giannantonio, T., R. Crittenden, R. Nichol, and others (2006). A high redshift detection of the integrated Sachs-Wolfe effect. *Physical Review D 74*.

[7] Hubble, E. P. (1926). Extragalactic nebulae. *ApJ 64*, 321–369.

[8] Ivezić, Ž., A. K. Vivas, R. H. Lupton, and R. Zinn (2005). The selection of RR Lyrae stars using single-epoch data. *AJ 129*, 1096–1108.

[9] Lintott, C. J., K. Schawinski, A. Slosar, and others (2008). Galaxy zoo: morphologies derived from visual inspection of galaxies from the Sloan Digital Sky Survey. *MNRAS 389*, 1179–1189.

[10] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review 63*(2), 81–97.

[11] Quinlan, J. R. (1992). *C4.5: Programs for Machine Learning* (first ed.). Morgan Kaufmann Series in Machine Learning. Morgan Kaufmann.

[12] Richards, G., R. Nichol, A. G. Gray, and others (2004). Efficient photometric selection of quasars from the Sloan Digital Sky Survey: 100,000 $z < 3$ quasars from Data Release One. *ApJS 155*, 257–269.

[13] Richards, G. T., A. D. Myers, A. G. Gray, and others (2009). Efficient photometric selection of quasars from the Sloan Digital Sky Survey II. ∼1,000,000 quasars from Data Release Six. *ApJS 180*, 67–83.

[14] Riegel, R., A. G. Gray, and G. Richards (2008). Massive-scale kernel discriminant analysis: Mining for quasars. In *SDM*, pp. 208–218. SIAM.

[15] Scranton, R., B. Menard, G. Richards, and others (2005). Detection of cosmic magnification with the Sloan Digital Sky Survey. *ApJ 633*, 589–602.

[16] Sesar, B., Ž. Ivezić, S. H. Grammer, and others (2010). Light curve templates and galactic distribution of RR Lyrae stars from Sloan Digital Sky Survey Stripe 82. *ApJ 708*, 717–741.

[17] Wolpert, D. H. and W. G. Macready (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1*, 67–82.