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 [14]. 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 galactic 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" [23]). 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 [15]. 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 [33]. The sample is further constrained to a smaller region of the overall color–color space following [15] ($0.7 < u - g < 1.35$, $-0.15 < g - r < 0.4$, $-0.15 < r - i < 0.22$, 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 Catalogs. 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' 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 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*).

[1] If, however, you can enlist thousands of citizen scientists in this proposition then you can fundamentally change this aspect; see [21].

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 approach (see §5.2.4).

The task of learning the best classifier then becomes the task of estimating p_k. 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 and the π quantities. Furthermore, the Bayes classifier can be shown to be optimal if the p_k and the π quantities are chosen to be the true distributions: that is, lower error cannot 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.

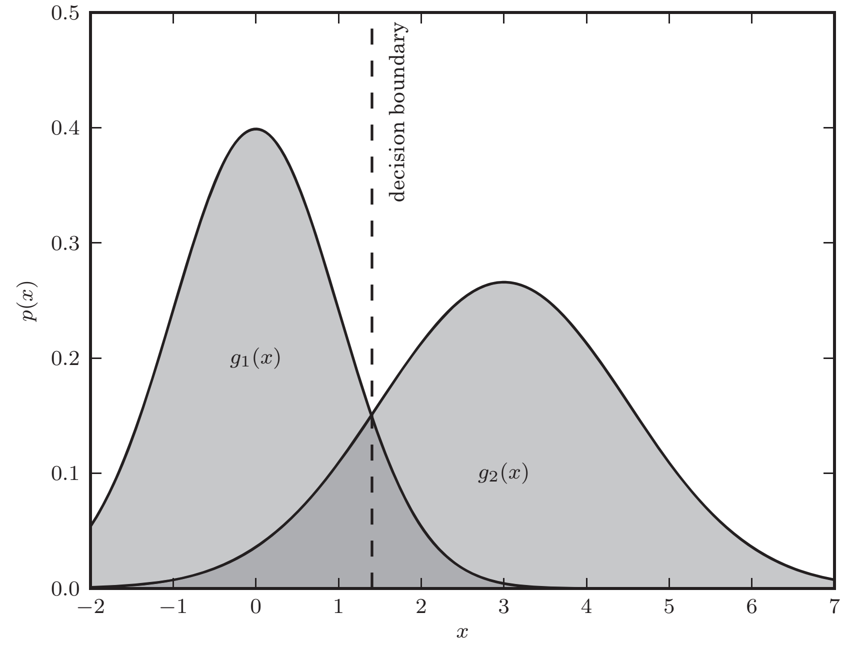

Figure 9.1. An illustration of a decision boundary between two Gaussian distributions.

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, $g_1(\mathbf{x}) = g_2(\mathbf{x})$, $g_1(\mathbf{x}) - g_2(\mathbf{x}) = 0$, and $g(\mathbf{x}) = 1/2$ 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

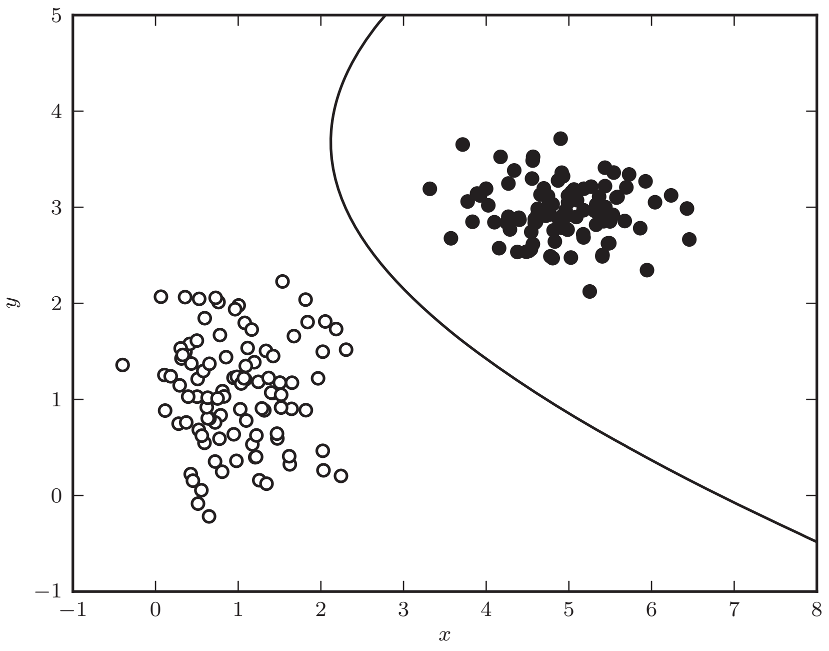

Figure 9.2. A decision boundary computed for a simple data set using Gaussian naive Bayes classification. The line shows the decision boundary, which corresponds to the curve where a new point has equal posterior probability of being part of each class. In such a simple case, it is possible to find a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

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

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

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.

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))  # 2 pts in 100 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.

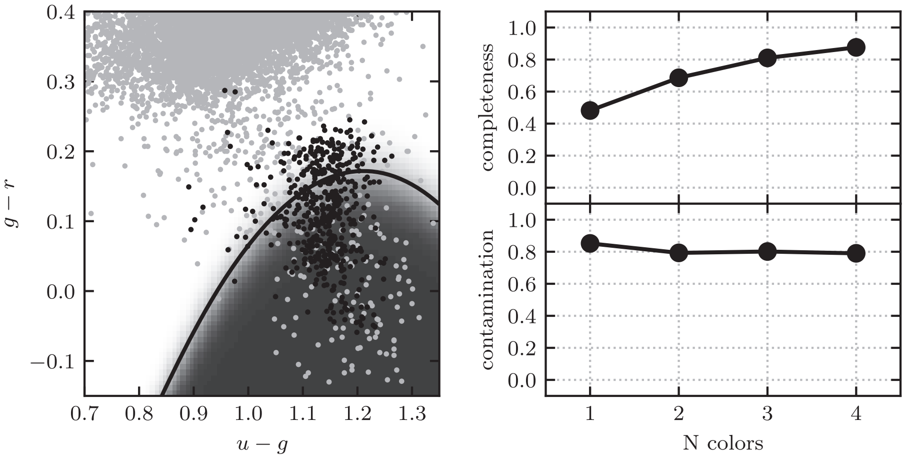

Figure 9.3. Gaussian naive Bayes classification method used to separate variable RR Lyrae stars from nonvariable main sequence stars. In the left panel, the light gray points show nonvariable sources, while the dark points show variable sources. The classification boundary is shown by the black line, 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.

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. This is apparent in the distribution of points and labels in the RR Lyrae data set.

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 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)

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

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

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

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:

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

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 **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.discriminant_analysis import (LinearDiscriminant
...                      Analysis, QuadraticDiscriminantAnalysis)

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

>>> lda = LinearDiscriminantAnalysis()
>>> lda.fit(X, y)
>>> y_pred = lda.predict(X)

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

For more details see the Scikit-learn documentation.

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) to ease the computational cost of model fitting. Note that the number of Gaussian components K must be chosen, ideally, for

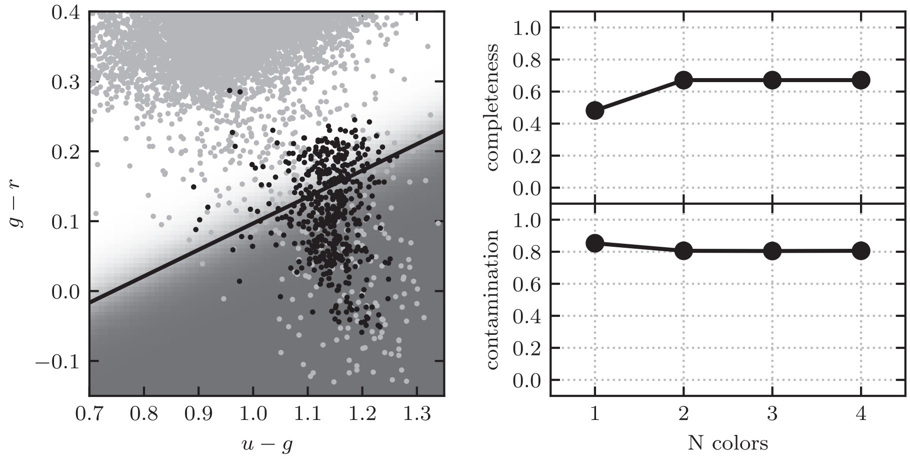

Figure 9.4. The linear discriminant boundary for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, LDA achieves a completeness of 0.672 and a contamination of 0.806.

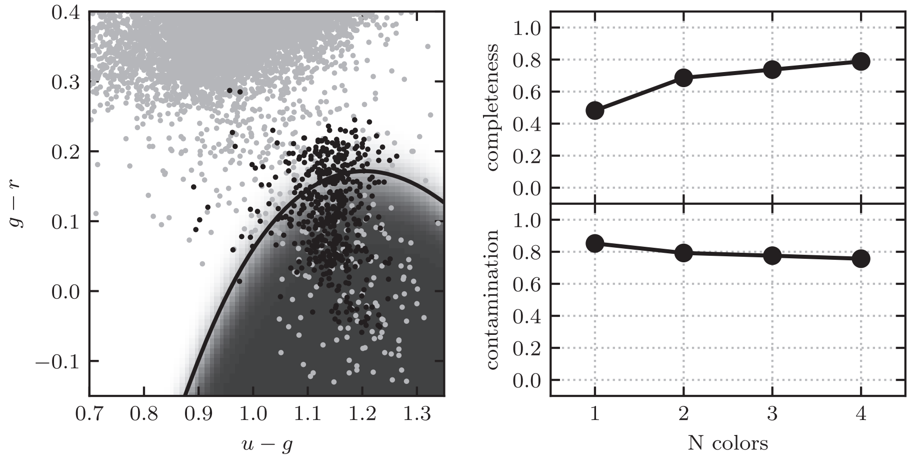

Figure 9.5. The quadratic discriminant boundary for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, QDA achieves a completeness of 0.788 and a contamination of 0.757.

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.

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.

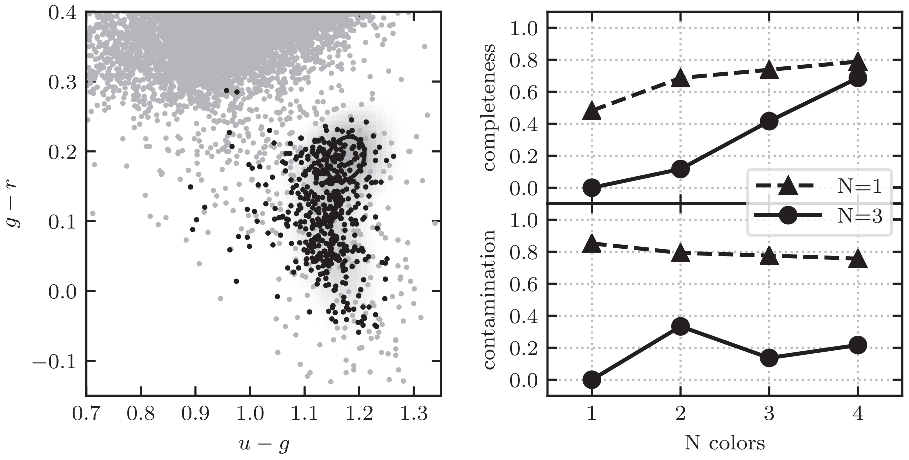

Figure 9.6. Gaussian mixture Bayes classifier for RR Lyrae stars (see caption of figure 9.3 for details). Here the left panel shows the decision boundary for the three-component model, and the right panel shows the completeness and contamination for both a one- and three-component mixture model. With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

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))  # 2 pts in 100 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.

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 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 [30] 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 *k*d-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))  # 2 pts in 100 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 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.

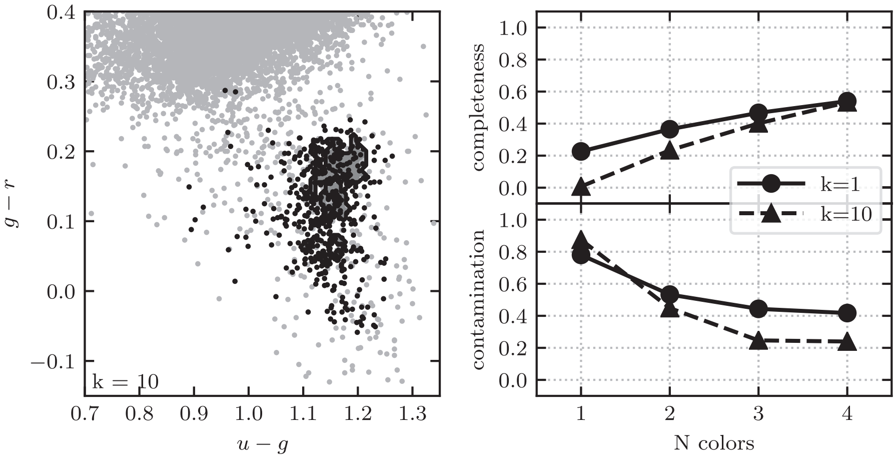

Figure 9.7. K-nearest-neighbor classification for RR Lyrae stars (see caption of figure 9.3 for details). Here the left panel shows the decision boundary for the model based on $K = 10$ neighbors, and the right panel shows the completeness and contamination for both $K = 1$ and $K = 10$. With all four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

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

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

where we define

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

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.

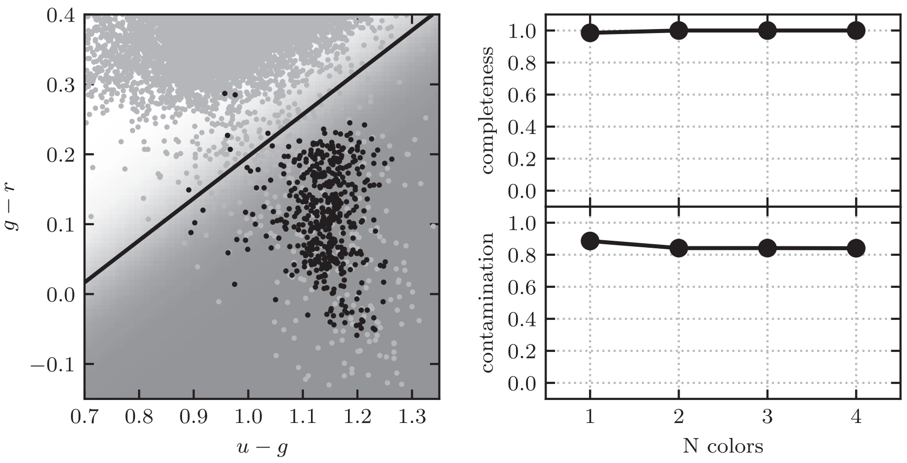

Figure 9.8. Logistic regression for RR Lyrae stars (see caption of figure 9.3 for details). With all four colors, logistic regression achieves a completeness of 0.993 and a contamination of 0.838.

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.

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

>>> logr = LogisticRegression(penalty='l2', solver='lbfgs')
>>> 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.

The hyperplane which maximizes the margin is given by finding

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

Equivalently, the constraints can be written as $y_i(\beta_0 + \beta^T x_i) \geq m||\beta||$. 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

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

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

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

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

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

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

we will now minimize

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

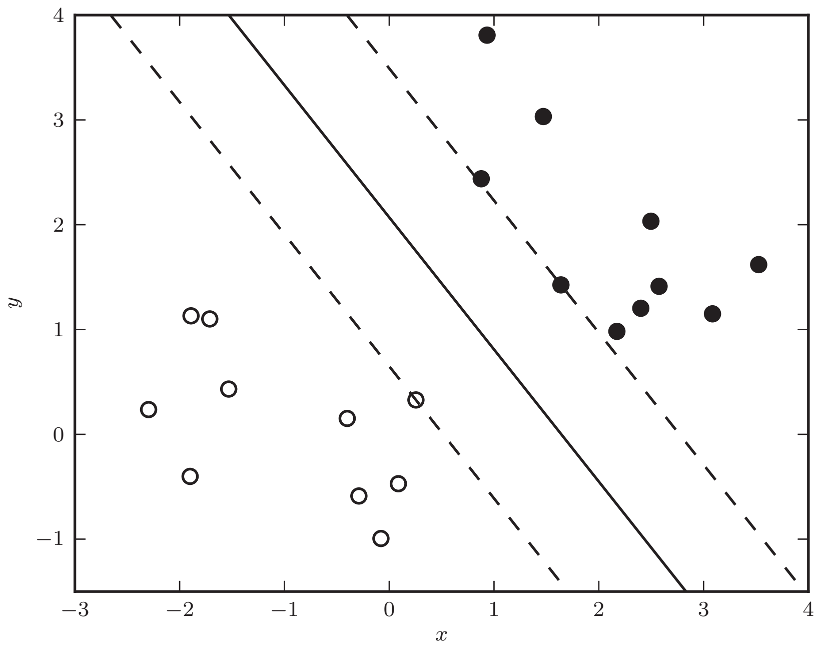

Figure 9.9. Illustration of SVM. The region between the dashed lines is the *margin*, and the points which the dashed lines touch are called the *support vectors*.

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

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

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.

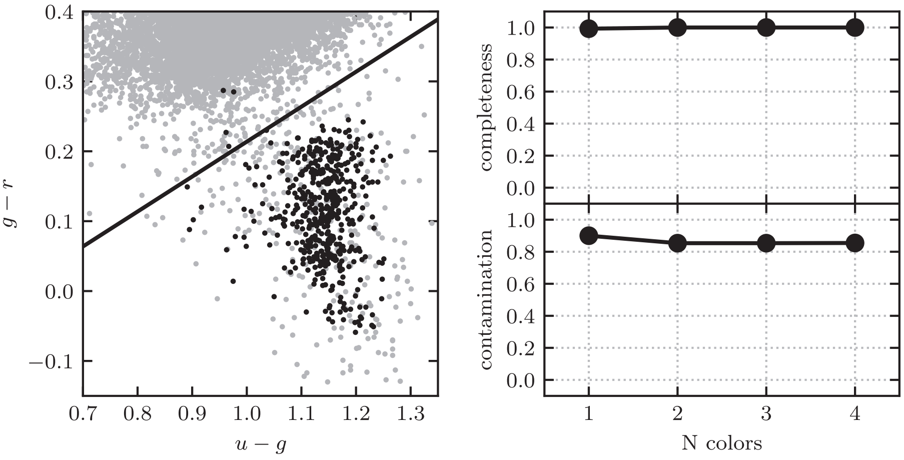

Figure 9.10. SVM applied to the RR Lyrae data (see caption of figure 9.3 for details). With all four colors, SVM achieves a completeness of 1.0 and a contamination of 0.854.

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

>>> model = LinearSVC(loss='squared_hinge')
>>> 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.

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

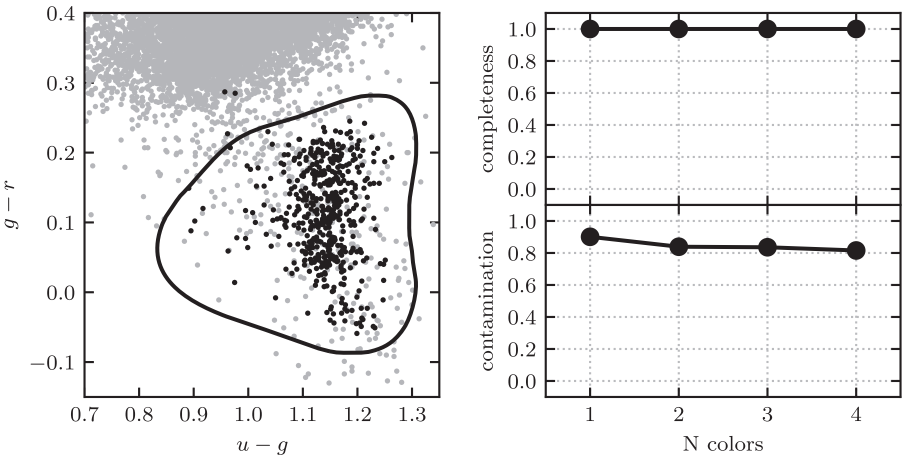

Figure 9.11. Kernel SVM applied to the RR Lyrae data (see caption of figure 9.3 for details). This example uses a Gaussian kernel with $\gamma = 20$. With all four colors, kernel SVM achieves a completeness of 1.0 and a contamination of 0.852.

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 criterion (see §9.7.1).

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

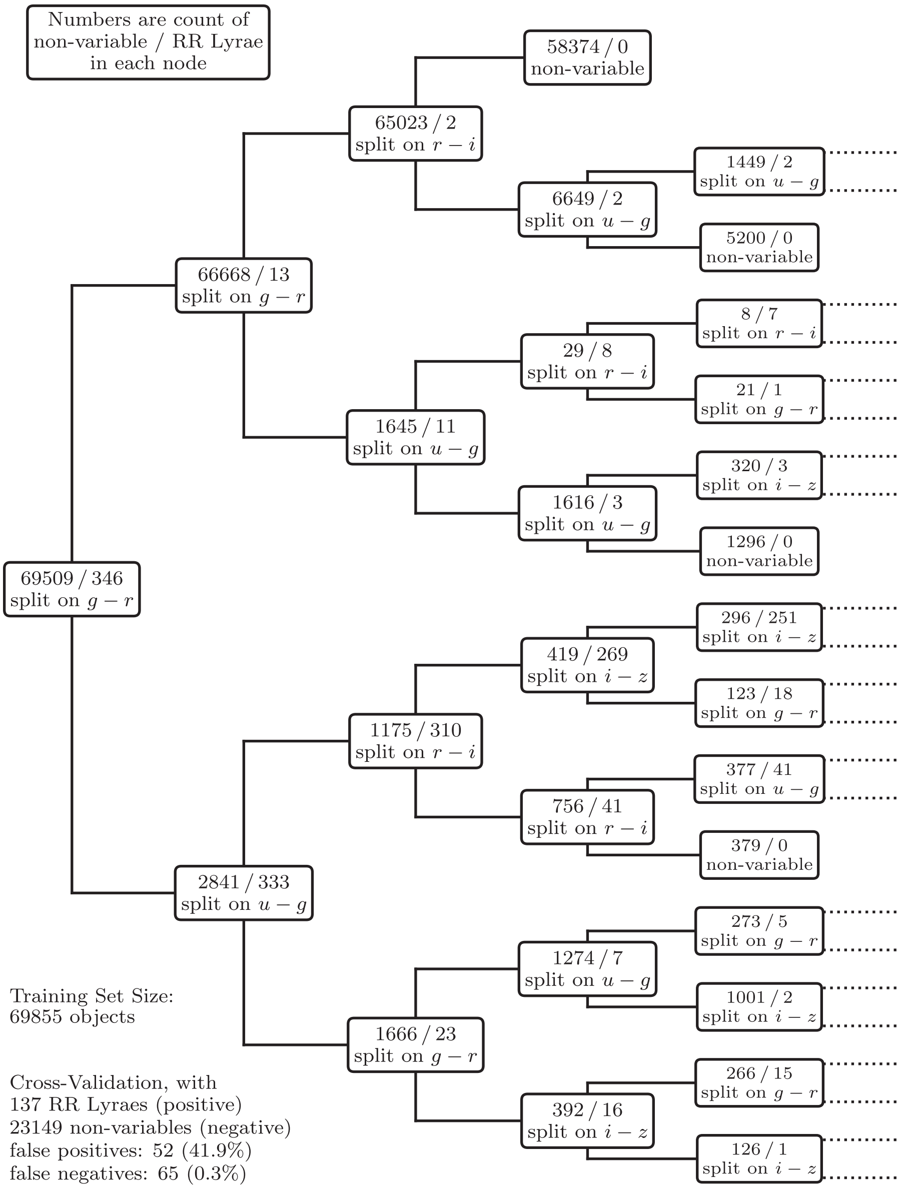
Numbers are count of non-variable / RR Lyrae in each node
69509 / 346 split on $g - r$
66668 / 13 split on $g - r$
65023 / 2 split on $r - i$
58374 / 0 non-variable
6649 / 2 split on $u - g$
1449 / 2 split on $u - g$
5200 / 0 non-variable
1645 / 11 split on $u - g$
29 / 8 split on $r - i$
8 / 7 split on $r - i$
21 / 1 split on $g - r$
1616 / 3 split on $u - g$
320 / 3 split on $i - z$
1296 / 0 non-variable
2841 / 333 split on $u - g$
1175 / 310 split on $r - i$
419 / 269 split on $i - z$
296 / 251 split on $i - z$
123 / 18 split on $g - r$
756 / 41 split on $r - i$
377 / 41 split on $u - g$
379 / 0 non-variable
1666 / 23 split on $g - r$
1274 / 7 split on $u - g$
273 / 5 split on $g - r$
1001 / 2 split on $i - z$
392 / 16 split on $i - z$
266 / 15 split on $g - r$
126 / 1 split on $i - z$
Training Set Size: 69855 objects
Cross-Validation, with
137 RR Lyraes (positive)
23149 non-variables (negative)
false positives: 52 (41.9%)
false negatives: 65 (0.3%)

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.

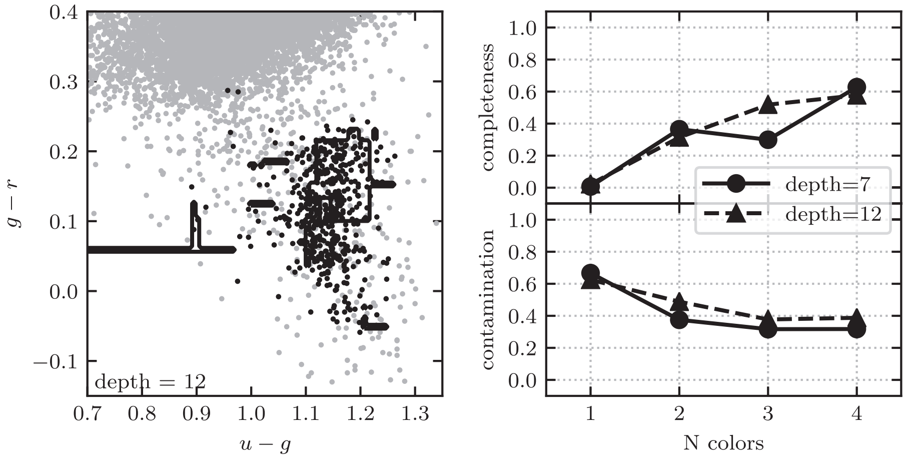

Figure 9.13. Decision tree applied to the RR Lyrae data (see caption of figure 9.3 for details). This example uses tree depths of 7 and 12. With all four colors, this decision tree achieves a completeness of 0.569 and a contamination of 0.386.

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.

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))  # 2 pts in 100 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 Criterion

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 criterion based on the information content or entropy of the data; see [27]. 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

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

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* [2]. 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

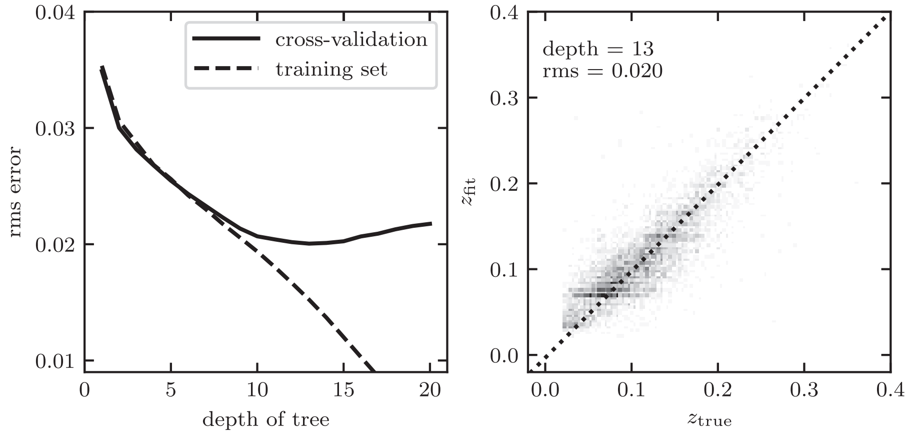

Figure 9.14. Photometric redshift estimation using decision-tree regression. The data are described in §1.5.5. The training set consists of u, g, r, i, z magnitudes of 60,000 galaxies from the SDSS spectroscopic sample. Cross-validation is performed on an additional 6000 galaxies. The left panel shows 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.

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 criterion 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.

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))  # 2 pts in 100 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.

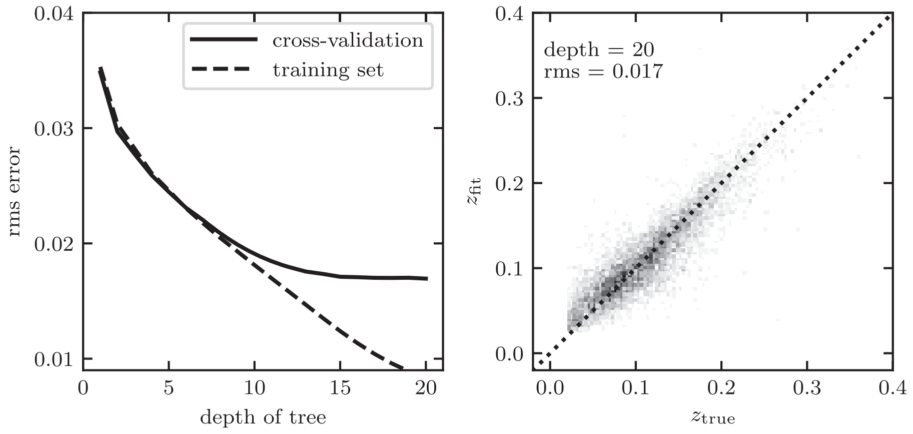

Figure 9.15. Photometric redshift estimation using random forest regression, with ten random trees. Comparison to figure 9.14 shows that random forests correct for the overfitting evident in very deep decision trees. Here the optimal depth is 20 or above, and a much better cross-validation error is achieved.

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 [3] 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* [8]. 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.

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.

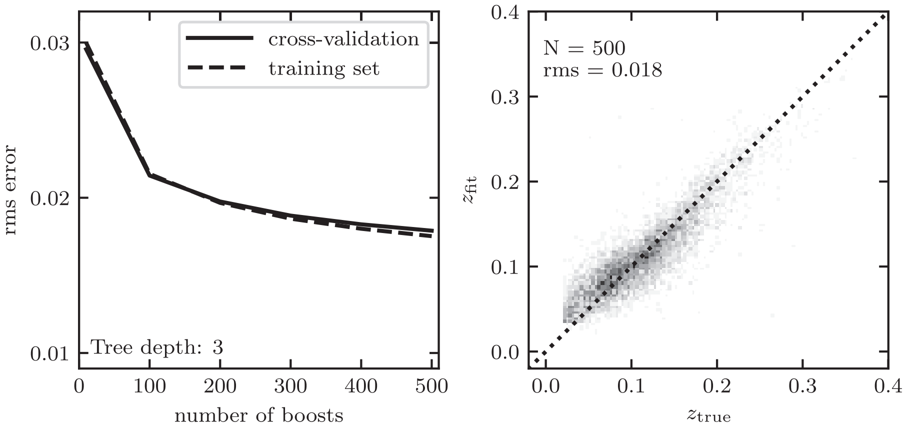

Figure 9.16. Photometric redshift estimation using gradient-boosted decision trees, with 100 boosting steps. As with random forests (figure 9.15), boosting allows for improved results over the single tree case (figure 9.14). Note, however, that the computational cost of boosted decision trees is such 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.

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 [9]. 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.

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. Deep Learning and Neural Networks

9.8.1. Neural Networks

Deep learning, an extension of the neural networks that were popularized in the 1990s [25], has become one of the principal techniques for machine learning in the sciences today [18]. The concepts behind neural or belief networks are inspired by the structure and function of the brain. A neuron in the brain is a core computational unit that takes

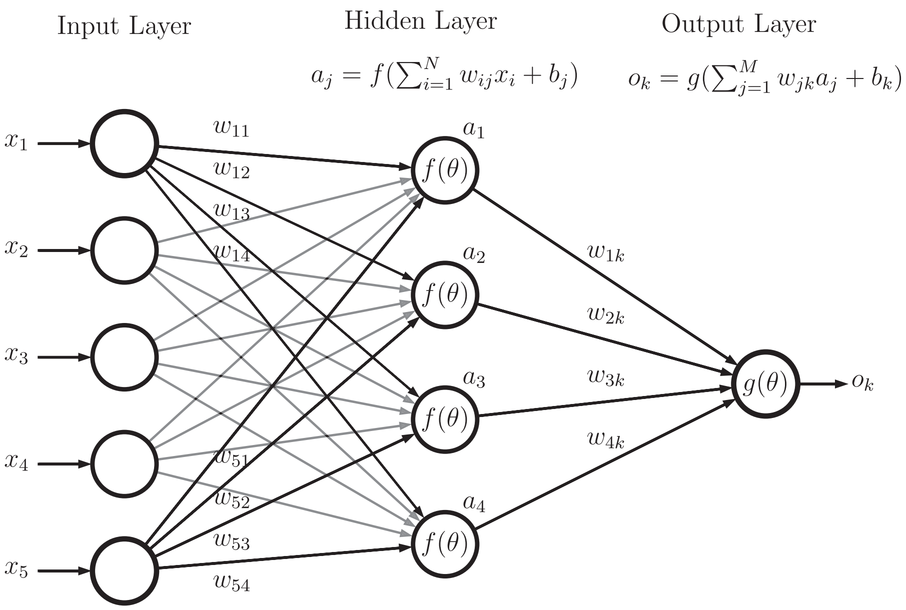

Figure 9.17. An illustration of the architecture of a neural network showing the connections between the input, output, and hidden layers.

a series of inputs from branched extensions of the neuron called dendrites, operates on these inputs, and generates an output that is transmitted along an axon to one or more other neurons. In the context of a neural network, as seen in figure 9.17, a neuron, j, takes a set of inputs, x_i, applies a, typically nonlinear, function to these inputs and generates an output value. Networks are then created by connecting multiple neurons or layers of neurons to one another.

If we consider the simplified network in figure 9.17, inputs (e.g., broadband photometric observations of a galaxy or pixel values from a CCD image) are passed to the neurons in the network. Each input is weighted by a value, w_{ij}, and the sum of these weighted inputs is operated on by a response or activation function $f(\theta)$. Example activation functions include sigmoid, logistic, and ReLU (rectified linear units) functions, which transform the input signal so that it varies between 0 and 1 through the application of a nonlinear response. The output from any neuron is then given by

$$a_j = f\left(\sum_i w_{ij}x_i + b_j\right) \tag{9.56}$$

where b_j is a bias term that determines the input level at which the neuron becomes activated.

We refer to the neurons between the inputs and the output layers as the hidden layers. If the neurons from one layer connect to all neurons in a subsequent layer we call this a fully connected layer. When the outputs from the neurons only connect to subsequent layers (i.e., the graph is acyclic) we refer to this as a feed-forward network. In general the feed-forward network is the most common structure for a neural network used in classification. As we will see for autoencoders (§9.8.5), it is not required that each layer in a network have the same number of neurons as a previous layer nor that all outputs must connect to all neurons in the next layer. The art of defining a successful neural network often comes down to how we configure the number of and size of these layers.

The final layer in the network is the output layer. As with the hidden layer, an activation function, $g(\theta)$, in the output layer acts on the weighted sum of its inputs. In figure 9.17 we have a single output node but there can be multiple outputs if, for example, we are predicting multiple classes of galaxy morphology. For our example network the output from the final neuron, o_k, would be given by

$$o_k = g\left(\sum_j w_{jk} a_j + b_k\right) = g\left(\sum_j w_{jk} f\left(\sum_i w_{ij} x_i + b_j\right) + b_k\right) \tag{9.57}$$

In figure 9.18 we show the application of a simple neural network to the estimation of photometric redshifts. As in figure 9.17, this network has five inputs (the u, g, r, i, z magnitudes for galaxies from the SDSS spectroscopic sample) and a single output node (the redshift of the galaxy). The hidden layer comprises four neurons with a sigmoid activation function (the output neuron has a linear activation function). We implement the network using PyTorch[2], an open-source deep learning library, and use batch learning for the 330,799 galaxies in our training sample (§9.8.2).

Overall the performance of this simplified neural network is comparable to that from our random forests (§9.7.3) and boosting (§9.7.3) examples. The 0.026 rms error from the neural network is slightly worse than that achieved by the boosted decision trees (rms = 0.018), but there has been no attempt to optimize the depth or size of the network architecture for the redshift estimation. Increasing the number of nodes in a layer or number of layers in the network as a whole would signficantly improve its performance.

9.8.2. Training the Network

Training a neural network is conceptually simple. Given a labeled set of data and a loss function (see §4.2.8), we need to optimize the weights and biases within the network by minimizing the loss. One approach would be to use gradient descent with a forward propagation of the errors; the weights and biases are perturbed and the neuron activations propagated to the output layer where the loss is calculated. For large networks, the number of weights makes this approach computationally prohibitive. A solution for training large networks, proposed in [31], uses *backpropagation* to efficiently estimate the gradient of the loss function with respect to the weights and biases (the backpropagation technique dates back to the 1970s or earlier [20]) .

Backpropagation, or the backward propagation of errors, is an example of automated differentiation, where the chain rule is used to decompose the derivative of a complex function into a sequence of operations. Once we have the derivatives we can apply standard gradient descent approaches for learning the network. Given our simplified single-layer network in figure 9.17, we wish to calculate the gradient of the loss as a function of the weights (for this example we will ignore the bias). For our example

[2]See https://pytorch.org

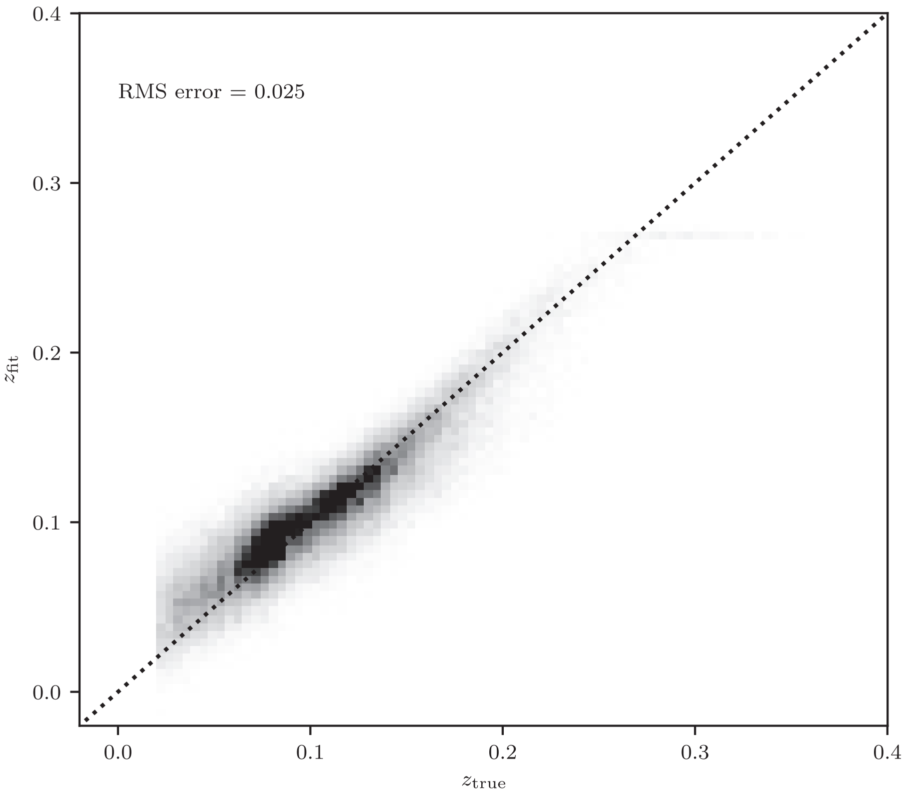

Figure 9.18. Photometric redshift estimation using a simple, single-layer neural network. The data are described in §1.5.5 and comprise u, g, r, i, z magnitudes of 330,799 galaxies from the SDSS spectroscopic sample. The network has five input neurons, a single hidden layer with four neurons, and a single output neuron. The rms for this simplified network is 0.026, comparable to that derived from random forests (§9.7.3) and boosted decision trees (§9.7.3).

network, and rewriting the input into a neuron as $z_k = \sum_j w_{jk} a_j$, we can use the chain rule to express the derivative of the loss, L, as a function of its input weights

$$\frac{\partial L}{\partial w_{jk}} = \frac{\partial L}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}}. \tag{9.58}$$

If we assume a sigmoid for the activation function and the L_2 norm for the loss function, this can be simplified for the final layer as

$$\frac{\partial L}{\partial w_{jk}} = (y - a_k) a_k (1 - a_k) a_j. \tag{9.59}$$

This can be written as $\delta_k a_j$ with $\delta_k = (y - a_k) a_k (1 - a_k)$ (i.e., expressing the gradient of the loss in terms of the activation values from the previous layer). To determine $\partial L/\partial w_{ij}$, the derivative of the loss with respect to the inputs to the hidden layer, we require

$$\frac{\partial L}{\partial w_{ij}} = \left[\sum_k \frac{\partial L}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial a_j}\right]\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}, \tag{9.60}$$

which simplifies to

$$\frac{\partial L}{\partial w_{ij}} = \left[\sum_k \delta_k w_{jk} \right] a_j(1 - a_j)a_i. \tag{9.61}$$

The gradient for each layer can, therefore, be estimated from the gradient and activation values of the subsequent layer. This is true for any activation function that is differentiable (i.e., we can take the error derivatives of one layer and compute the error derivatives from the layer before). The optimization of a network starts with randomly initialized weights and biases. It then proceeds to forward propagate these weights through the network, calculate the resulting loss, and then estimate the derivatives of the network via backpropagation. The weights are updated as

$$w'_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}, \tag{9.62}$$

where α is the called the *learning rate* and determines how much we perturb the weights in each iteration. This iterative procedure continues until convergence of the loss. As with many optimization techniques there is trade between the learning rate, convergence rate, and stability of the resulting solution. Large updates (a large learning rate) can produce rapid convergence but at the risk of a less stable solution. If the loss function can be expressed in terms of a sum over subsets of the training data (e.g., as is the case for the L_2 norm) the training can be undertaken either for the dataset as a whole, for subsets of the data (batch learning), or for individual entries (on-line or stochastic learning).

9.8.3. How Many Layers and How Many Neurons?

The number of layers, number of neurons in a layer, and the connectivity of these layers is typically described as the *network architecture*. One of the central questions in deep learning applications is how to configure this network: how many neurons should we include in each layer and how many layers do we need? Too few and we run the risk of underfitting where the network is not capable of adapting to the structure within the data. Too many and we can overfit (see §8.11), resulting in a network that maps to the noise within the training data, and that will produce poor classifications when applied to new data sets. Cross-validation, as introduced in §8.11.1, is often used to evaluate the performance of the network, but given the computational cost of training a network it is useful to have a baseline strategy for defining the initial architecture.

When deciding on the number of layers within an architecture, it is worth considering the prior information we may have about the data. For data that can be represented by a linear model, no layers are required [22]. A single layer network can approximate any continuous function. Two layers can represent arbitrary decision boundaries for smooth functions [26]. More layers can represent noncontinuous or complex structure within data. With the work of [13] and the increase in computational resources through the use of GPUs or specialized TPUs (tensor processing units), we have the ability to train networks with many layers. These advances have resulted in approaches to define a network architecture becoming more trial and error than applying an underlying set of principles. For a starting point, however, there are relatively few problems that benefit significantly from more than two layers and we recommend starting with a single layer when training an initial network and using cross-validation to determine when additional layers lead to results in the data being overfit.

As with the number of layers, the number of neurons within a layer drives the computational cost (and requiring progressively larger training sets to avoid overfitting of the data). There are many proposals for rules of thumb for defining a network architecture [12]. These include

- the number of neurons should lie between the number of input and output nodes.
- the number of neurons should be equal to the number of outputs plus 2/3 the number of input nodes.
- the number of neurons in the hidden layer should be less than twice the size of the input layers.

9.8.4. Convolutional Networks

Convolutional neural networks [19] or (CNNs) are networks designed to work with images or with any regularly sampled dataset. They have been responsible for many of the improvements in the performance of image and video classification techniques over the last decade [17]. For the traditional deep network architectures described above, with a series of fully connected layers, image classification would require the output of each pixel in an image to be connected to every neuron in the first layer of the network. Given that the number of pixels can range from the tens of thousands to tens of millions, learning such a network would be extremely challenging. CNNs reduce the complexity of the network by requiring that neurons only respond to inputs from a subset of an image (the receptive field). This mimics the operation of the visual cortex where neurons only respond to a small part of the field of view.

There are four principal components to a CNN:

- a convolutional layer,
- a nonlinear activation function,
- a pooling or downsampling operation, and
- a fully connected layer for classification.

Dependent on the complexity of the network or structure of the data, these components can occur singularly or chained together in multiple sequences (see figure 9.19). In the following sections we will highlight the purpose and properties of each of these components but for details of this generic or more advanced CNN architectures we refer the reader to the books by [4, 11], the PyTorch (https://pytorch.org/tutorials) and TensorFlow (https://www.tensorflow.org/tutorials) tutorials, and the many online course on CNNs.

The inputs to a CNN are typically images or times series data (essentially a 1D image). For the case of photographs used in everyday life these images comprise multiple channels (e.g., the RGB channels of a jpeg or png image). For the case of astronomical data we will assume all images are grayscale with a single channel.

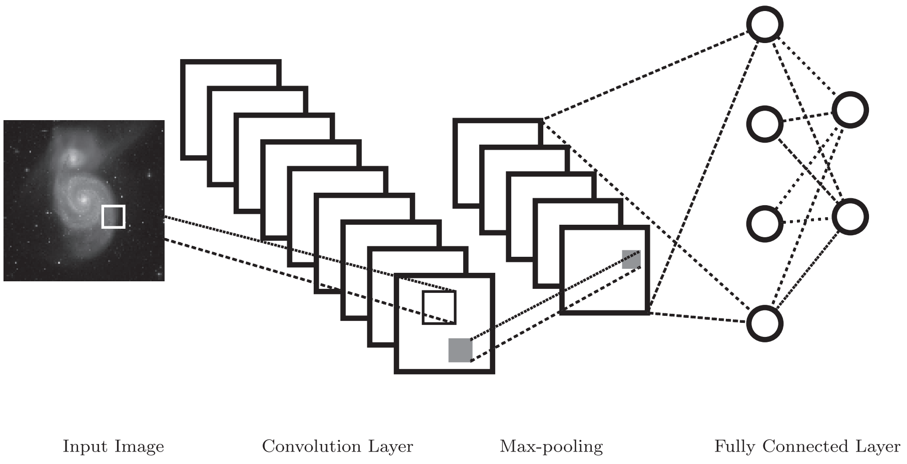

Figure 9.19. The network architecture for a convolutional neural network (CNN). This comprises a series of convolutional, activation, and pooling layers that end in a fully connected layer for the classification. One key aspect of the CNN is the concept of receptive field whereby neurons only respond to stimuli from small regions within an image (mimicking the response of the visual cortex). This reduces the complexity of the network that needs to be learned.

Convolution in a CNN refers to the convolution of the input data $I(x,y)$ with a kernel $K(x,y)$ which will produce a feature map $F(x,y)$

$$F(x,y) = K(x,y) * I(x,y) = \sum_{x_0} \sum_{y_0} I(x - x_0, y - y_0) K(x_0, y_0). \tag{9.63}$$

The kernel only responds to pixels within its receptive field (i.e., the size of the kernel), reducing the computational complexity of the resulting network. The kernels in the convolution are described by a *depth* (the number of kernels, K, applied to the image), and a *stride* (how many pixels a kernel shifts at each step in the convolution; typically one). Given an $N \times M$ image, the result of the convolution step is to transform a single image into a data cube of feature maps with a dimension $N \times M \times K$.

As with traditional neural networks, a nonlinear activation function is applied to the individual pixels in the resulting feature maps. This activation function can be a sigmoid as described previously or a hyperbolic tangent, but is most commonly a ReLU function which is defined as $\mathrm{ReLU}(x) = x$ if $x > 0$ and is 0 otherwise. The ReLU is popular because its derivative is easy to calculate (1 if the neuron is activated, 0 otherwise).

The pooling step in the CNN downsamples or subsamples the feature map data cubes. Pooling summarizes the feature map values within a region of interest (e.g., a 2x2 pixel window). The summary can be the average pixel value but more commonly the maximum pixel value is preserved (max pooling) in the downsampling. This pooling of the feature maps reduces the size of the resulting network and makes the network less sensitive to small translations or distortions between images.

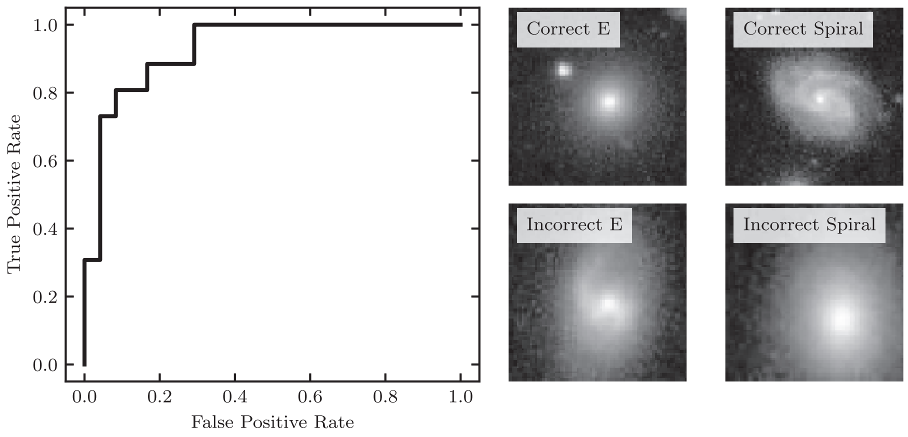

Figure 9.20. The accuracy of a multilayer convolutional neural network applied to a set of morphologically classified galaxy images taken from the SDSS. The configuration of the network is described in §9.8.4. The left panel shows the false-positive rate against the true-positive rate for the resulting network. The right side of the figure shows examples of images that were correctly and incorrectly classified.

Training of the network, including learning the underlying kernels, is performed using backpropagation as described in §9.8.2. Once learned the kernels within the convolutional layer often appears as physically intuitive operations such as edge detection filters.

The final layer of a CNN is the classification layer which maps the output of the CNN to a set of labels. This is typically a fully connected layer as described in §9.8.2, where each output of the final pooling layer connects to all neurons in the classification layer. Other machine learning and classification methods could also be applied to the outputs of the pooling layer as the CNN is essentially learning a set of features from the input data.

In figure 9.20 we build a simple CNN [5]. We implement this network using the TensorFlow library,[3] an open source library for deep learning and use the high level Python interface provided by Keras.[4] For this example, we train a network to predict whether a galaxy is a spiral or an elliptical using 1000 SDSS jpeg images of galaxies and the morphologies measured by [7, 24]. Multiple convolution, activation, and pooling layers can be chained together within a CNN. For the current example, we have five convolution and activation layers, three max pooling layers and two fully connected layers for classification. After training, the network has a 0.875 probability of correctly classifying a galaxy image. In figure 9.20 we show the receiver operating characteristic (ROC) curve (see §9.9) for the CNN and illustrate some of the correctly and incorrectly classified galaxies. We note that the accuracy of the network is limited by the size of the training set (which itself is limited by the available time to train the network).

[3]See https://www.tensorflow.org

[4]See https://keras.io

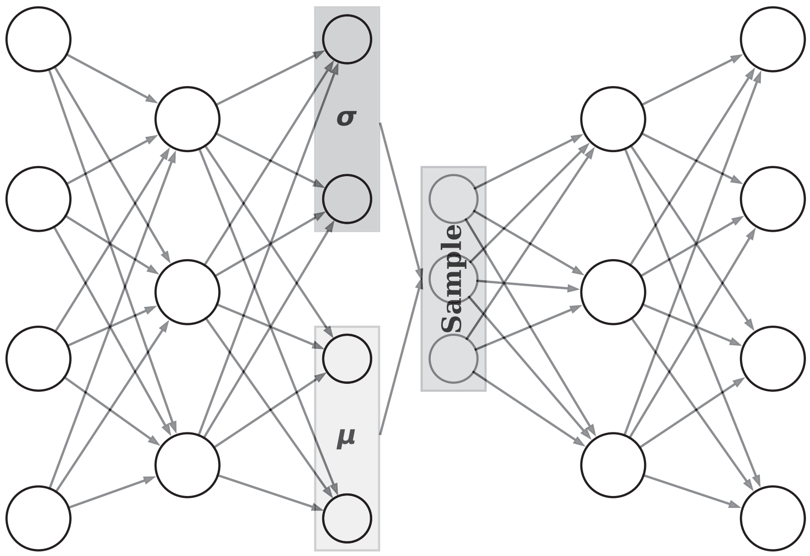

Figure 9.21. The structure of a variational autoencoder. The autoencoder comprises two networks, one that encodes the data into a lower-dimensional representation and a second that decodes the representation to reproduce the input data. The amount of compression is defined by the size of the smallest, or bottleneck, layer. In the variational autoencoder, the encoder learns a mean vector and variance for the input data representation, which ensures a continuous mapping between the data and latent space described by the encoded data.

9.8.5. Autoencoders

Autoencoders and the related variational autoencoders are variants of neural networks that learn the encoding or structure within data. They can be thought of as two networks, an encoder which learns a representation of the data and a decoder that reconstructs the input data from this representation. If the network that encodes the data includes a layer with fewer neurons than the dimensionality of the input data, then the autoencoders will compress the data (for linear autoencoders there is a direct relation to principal component analysis [1]). Training of autoencoders is undertaken with standard gradient descent and backpropagation techniques (§9.8.2).

The level of compressesion is defined by the number of neurons in the smallest (or bottleneck) layer. If we consider the activation of the neurons within this layer, they define a latent space where the progressive activation of individual or multiple neurons will result in new representations of the input data. For traditional autoencoders, one difficulty in applying them to astronomical problems is that the latent space does not always provide a continuous mapping to the data. Data can cluster within the latent space which means that we cannot assume a continuous interpolation of new data as we sample the latent space. The consequence of this is that traditional autoencoders have generally been only used for the denoising of data as opposed to generating new data samples.

More recently, variational autoencoders [16] have become popular as mechanisms for building generative models for a data set. They solve the interpolation issue by mapping the input data to a continuous distribution. The encoder maps the input

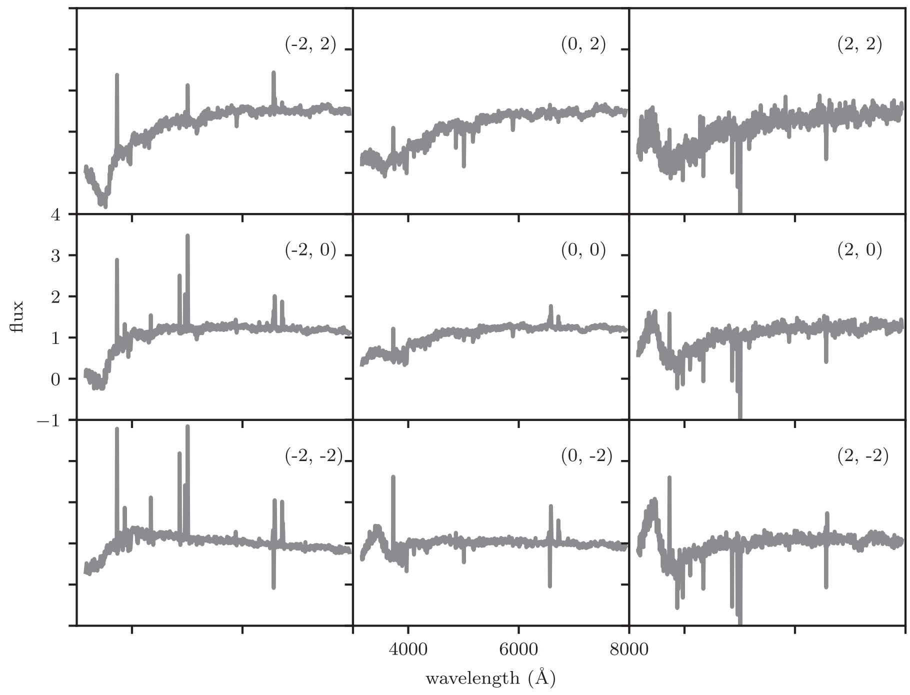

Figure 9.22. Spectra generated from a variational autoencoder applied to data from the SDSS survey. The encoder is limited to two components (a 2-dimensional latent space). As we progressively activate the neurons in this latent space we generate a smooth transition from spectra consistent with quiescent to star-forming galaxies. The numbers in each panel of the plot indicate the activation value of the two neurons in the latent space.

x to a latent space z with a distribution function $q(z|x)$. The decoder maps a point in the latent space z to a point in data space. The loss function for these operations comprises the usual log-likelihood of how well we reconstruct the input data, $\log p(x|z)$, and a second term that requires that encoder distribution function $q(z|x)$ reproduces the prior distribution function $p(z)$. We accomplish the second of these terms by calculating the Kullback-Leibler divergence between $q(z|x)$ and $p(z)$.

In a variational autoencoder, $p(z)$ is assumed to have the form of a Gaussian distribution with zero mean and unit variance. The loss for this network is then given by

$$L = \sum_i E_{q(z|x_i)}[\log p(x_i|z)] - KL(q(z|x_i)||p(z)), \tag{9.64}$$

where we sum over all data points x_i. In practice the Gaussian distribution for $q(z|x_i)$ is represented by a mean vector and a variance vector (of size N). The intuition for the variational autoencoder is that the mean vector centers the average encoding of a data point, and variance samples around this average encoding. The size of the variance determines how far from the average vector in the latent space an object may vary. As each latent component is represented by a probability distribution, if we randomly sample from these components and then pass these latent values through the decoder, we will generate a new output that will smoothly vary across the latent space.

An example variational autoencoder applied to the SDSS spectra used in chapter 7 is shown in figure 9.22. The variational autoencoder comprises a bottleneck layer with just two neurons. Figure 9.22 shows the decoded spectra as a function of the activation of these two neurons (i.e., we increase the output of each neuron from -2 to 2 and decode this compressed representation into the full spectrum). A number of aspects of this plot are worth noting. The autoencoder is limited to two components yet the spectra smoothly transition from quiescent galaxies to strongly line-emitting galaxy spectra. The reconstruction error for these spectra is comparable to that achieved with PCA using more components.

9.9. 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 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 9.23. 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 9.23 shows the completeness vs. the 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 9.24 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 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

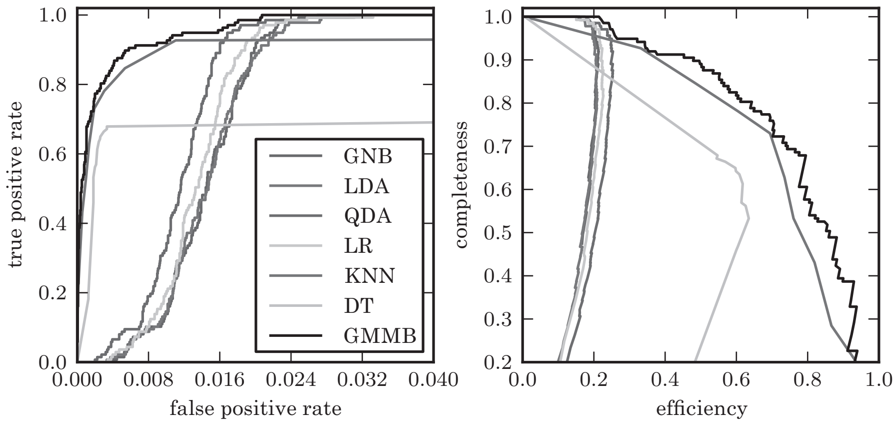

Figure 9.23. ROC curves (left panel) and completeness–efficiency curves (left panel) for the four-color RR Lyrae data using several of the classifiers explored in this chapter: Gaussian naive Bayes (GNB), linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), logistic regression (LR), K-nearest-neighbor (KNN) classification, decision tree classification (DT), and GMM Bayes classification (GMMB).

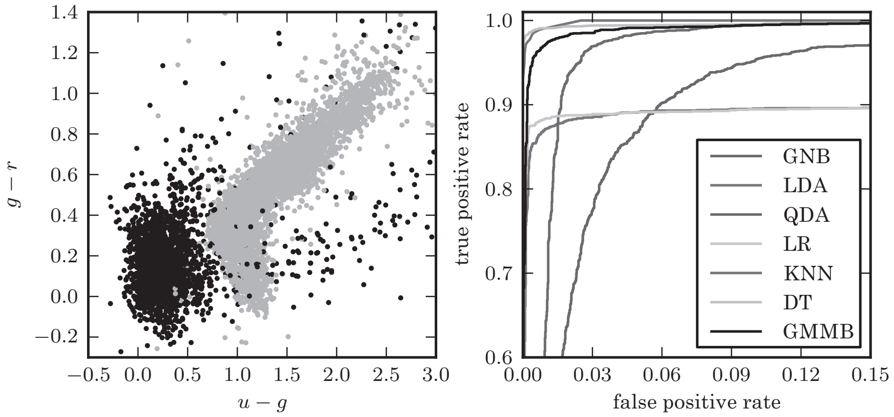

Figure 9.24. The left panel shows data used in color-based photometric classification of stars and quasars. Stars are indicated by gray points, while quasars are indicated by black points. The right panel shows ROC curves for quasar identification based on $u - g$, $g - r$, $r - i$, and $i - z$ colors. Labels are the same as those in figure 9.23.

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.

9.10. 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 [34]. 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. In recent years the flexibility of deep neural networks has led to them outperforming many other classification techniques, at least in the context of image and time series classification. For other classifiers increased flexibility in the function representation (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), is more likely 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 [6]. 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. Recently, however, significant work on saliency maps and feature attribution has improved the interpretability of deep networks. These approaches enable the visualization of how a neuron activates in response to stimuli from specific pixels within an image providing a mapping from the classification to the pixels contributing to the classification.

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 multiply 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 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). The onus of

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

Method	Accuracy	Interpretability	Simplicity	Speed
Deep neural networks	H	M	M	M
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

having to store and manage the typically hundreds or thousands of trees in random forests counts against it, though it is conceptually simple. The release of open source libraries such as TensorFlow and PyTorch have transformed the ease of use of deep learning applications. Building and configuring networks is relatively simple and, in many ways, is comparable in difficulty to generating random forests and mixture models (i.e., deciding how to configure a network is the challenge rather than coding it).

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 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 [30]), it produced a dramatic leap in the size and quality of quasar catalogs [28, 29], and associated scientific results [10, 32]. 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] Baldi, P. and K. Hornik (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks 2*(1), 53–58.

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

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

[4] Chollet, F. (2017). *Deep Learning with Python* (1st ed.). Greenwich, CT, USA: Manning Publications Co.

[5] Domínguez Sánchez, H., M. Huertas-Company, M. Bernardi, D. Tuccillo, and J. L. Fischer (2018, May). Improving galaxy morphologies for SDSS with Deep Learning. *MNRAS 476*, 3661–3676.

[6] 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.

[7] Fischer, J. L., M. Bernardi, H. Domínguez Sánchez, M. Huertas-Company, and D. Tuccillo (2018, 02). Improving galaxy morphologies for SDSS with Deep Learning. *Monthly Notices of the Royal Astronomical Society 476*(3), 3661–3676.

[8] 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.

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

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

[11] Goodfellow, I. J., Y. Bengio, and A. C. Courville (2016). *Deep Learning*. Adaptive computation and machine learning. MIT Press.

[12] Heaton, J. (2008). *Introduction to Neural Networks for Java* (2nd ed.). Heaton Research, Inc.

[13] Hinton, G. E., S. Osindero, and Y.-W. Teh (2006, July). A fast learning algorithm for deep belief nets. *Neural Comput. 18*(7), 1527–1554.

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

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

[16] Kingma, D. P. and M. Welling (2013, December). Auto-Encoding Variational Bayes. *arXiv e-prints*, arXiv:1312.6114.

[17] Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc.

[18] LeCun, Y., Y. Bengio, and G. Hinton (2015, 05). Deep learning. *Nature 521*, 436 EP –.

[19] LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation 1*(4), 541–551.

[20] Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki.

[21] 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.

[22] McCullagh, P. and J. Nelder (1989). *Generalized linear models* (2nd ed.). Chapman and Hall.

[23] 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.

[24] Nair, P. B. and R. G. Abraham (2010, February). A Catalog of Detailed Visual Morphological Classifications for 14,034 Galaxies in the Sloan Digital Sky Survey. *ApJS 186*, 427–456.

[25] Odewahn, S. C., E. B. Stockwell, R. L. Pennington, R. M. Humphreys, and W. A. Zumach (1992, January). Automated star/galaxy discrimination with neural networks. *AJ 103*, 318–331.

[26] P. Lippmann, R. (1987, 05). An introduction to computing with neural nets. *ASSP Magazine, IEEE 4*, 4– 22.

[27] Quinlan, J. R. (1992). *C4.5: Programs for Machine Learning* (first ed.). Morgan Kaufmann Series in Machine Learning. Morgan Kaufmann.

[28] 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.

[29] Richards, G. T., A. D. Myers, A. G. Gray, and others (2009). Efficient photometric selection of quasars from the Sloan Digital Sky Survey II. $\sim$1,000,000 quasars from Data Release Six. *ApJS 180*, 67–83.

[30] Riegel, R., A. G. Gray, and G. Richards (2008). Massive-scale kernel discriminant analysis: Mining for quasars. In *SDM*, pp. 208–218. SIAM.

[31] Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. Chapter Learning Internal Representations by Error Propagation, pp. 318–362. MIT Press.

[32] Scranton, R., B. Menard, G. Richards, and others (2005). Detection of cosmic magnification with the Sloan Digital Sky Survey. *ApJ 633*, 589–602.

[33] 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.

[34] Wolpert, D. H. and W. G. Macready (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1*, 67–82.