been refined throughout this century and has been applied across a vast array of situations. Application of the ideas of probability enables us to obtain "best" estimates of values, even in the face of data inadequacies, and even when only a sample has been measured. Moreover, application of these ideas also allows us to quantify our confidence in the results.

Later chapters of this book make heavy use of probabilistic arguments. They underlie many—perhaps even most—data mining tools, from global modeling to pattern identification.

4.9 Further Reading

Books containing discussions of different schools of probability, along with the consequences for inference, include those by DeFinetti (1974, 1975), Barnett (1982), and Bernardo and Smith (1994). References to other work on statistics and particular statistical models are given at the ends of chapters 6, 9, 10 and 11.

There are many excellent basic books on the calculus of probability, including those by Grimmett and Stirzaker (1992) and Feller (1968, 1971). The text by Hamming (1991) is oriented towards engineers and computer scientists (and contains many interesting examples), and Applebaum (1996) is geared toward undergraduate mathematics students. Probability calculus is a dynamic area of applied mathematics, and has benefited substantially from the different areas in which it has been applied. For example, Alon and Spencer (1992) give a fascinating tour of the applications of probability in modern computer science.

The idea of randomness as departure from the regular or predictable is discussed in work on Kolmogorov complexity (e.g., Li and Vitanyi, 1993). Whittaker (1990) provides an excellent treatment of the general principles of conditional dependence and independence in graphical models. Pearl (1988) is a seminal work in this area from the the artificial intelligence perspective.

There are numerous introductory texts on inference, such as those by Daly et al. (1995), as well as more advanced texts that contain a deeper discussion of inferential conscepts, such as Cox and Hinkley (1974), Schervish (1995), Lindsey (1996), and Lehmann and Casella (1998), and Knight (2000). A broad discussion of likelihood and its applications is provided by Edwards (1972). Bayesian methods are now the subjects of entire books. Gelman et al. (1995) provides an excellent general text on Bayesian approach. A comprehensive reference is given by Bernardo and Smith (1994) and a lighter introduction is give by Lee (1989). Nonparametric methods are described by Randles and Wolfe (1979) and Maritz (1981). Bootstrap methods are described by Efron and Tibshirani (1993).

Miller (1980) describes simultaneous test procedures. The methods we have outlined above are not the only approaches to the problem of inference about multiple parameters; Lindsey (1999) describes another.

Books on survey sampling discuss efficient strategies for drawing samples—see, for example, Cochran (1977) and Kish (1965).

Chapter 5: A Systematic Overview of Data Mining Algorithms

5.1 Introduction

This chapter will examine what we mean in a general sense by a *data mining algorithm* as well as what components make up such algorithms. A working definition is as follows:

A data mining algorithm is a well-defined procedure that takes data as input and produces output in the form of models or patterns.

We use the term *well-defined* indicate that the procedure can be precisely encoded as a finite set of rules. To be considered an *algorithm*, the procedure must always terminate after some finite number of steps and produce an output.

In contrast, a *computational method* has all the properties of an algorithm except a method for guaranteeing that the procedure will terminate in a finite number of steps. While specification of an algorithm typically involves defining many practical implementation details, a computational method is usually described more abstractly. For example, the search technique *steepest descent* is a computational method but is not in itself an algorithm (this search method repeatedly moves in parameter space in the direction that has the steepest decrease in the score function relative to the current parameter values). To specify an algorithm using the steepest descent method, we would have to give precise methods for determining where to begin descending, how to identify the direction of steepest descent (calculated exactly or approximated?), how far to move in the chosen direction, and when to terminate the search (e.g., detection of convergence to a local minimum).

As discussed briefly in chapter 1, the specification of a data mining algorithm to solve a particular task involves defining specific *algorithm components*:

1. the data mining *task* the algorithm is used to address (e.g., visualization, classification, clustering, regression, and so forth). Naturally, different types of algorithms are required for different tasks.
2. the *structure* (functional form) of the model or pattern we are fitting to the data (e.g., a linear regression model, a hierarchical clustering model, and so forth). The structure defines the boundaries of what we can approximate or learn. Within these boundaries, the data guide us to a particular model or pattern. In chapter 6 we will discuss in more detail forms of model and pattern structures most widely used in data mining algorithms.
3. the *score function* we are using to judge the quality of our fitted models or patterns based on observed data (e.g., misclassification error or squared error). As we will discuss in chapter 7, the score function is what we try to maximize (or minimize) when we fit parameters to our models and patterns. Therefore, it is important that the score function reflects the relative practical utility of different parameterizations of our model or pattern structures. Furthermore, the score function is critical for learning and generalization. It can be based on goodness-of-fit alone (i.e., how well the model can describe the observed data) or can try to capture *generalization* performance (i.e., how well will the model describe data we have not yet seen). As we will see in later chapters, this is a subtle issue.
4. the *search* or *optimization method* we use to search over parameters and structures, i.e., computational procedures and algorithms used to find the maximum (or minimum) of the score function for particular models or patterns. Issues here include computational methods used to optimize the score function (e.g., steepest descent) and search-related parameters (e.g., the maximum number of iterations or convergence specification for an iterative algorithm). If the model (or pattern) structure is a single fixed structure (such as a kth-order polynomial function of the inputs), the search is conducted in parameter space to optimize the score function relative to this fixed structural form. If the model (or pattern) structure consists of a set (or family) of different structures, there is a search over both structures and their associated parameter spaces. Optimization and search are traditionally at the heart of any data mining algorithm, and will be discussed in much more detail in chapter 8.
5. the *data management technique* to be used for storing, indexing, and retrieving data. Many statistical and machine learning algorithms do not specify any data management technique, essentially assuming that the data set is small enough to reside in main memory so that random access of any data point is free (in terms of time) relative to actual computational costs. However, massive data sets may exceed the capacity of available main memory and reside in secondary (e.g., disk) or tertiary (e.g., tape) memory. Accessing such data is typically orders of magnitude slower than accessing main memory, and thus, for massive data sets, the physical location of the data and the manner in which it is

accessed can be critically important in terms of algorithm efficiency. This issue of data management will be discussed in more depth in chapter 12.

Table 5.1 illustrates how three well-known data mining algorithms (CART, backpropagation, and the A Priori algorithm) can be described in terms of these basic components. Each of these algorithms will be discussed in detail later in this chapter. (One of the differences between statistical and data mining perspectives is evident from this table. Statisticians would regard CART as a model, and backpropagation as a parameter estimation algorithm. Data miners tend to see things more in terms of algorithms: processing the data using the algorithm to yield a result. The difference is really more one of perspective than substance.)

Table 5.1: Three Well-Known Data Mining Algorithms Broken Down in Terms of their Algorithm Components.

	CART	Backpropagation	A Priori
Task	Classification and Regression	Regression	Rule Pattern Discovery
Structure	Decision Tree	Neural Network (Nonlinear functions)	Association Rules
Score Function	Cross-validated Loss Function	Squared Error	Support/Accuracy
Search Method	Greedy Search over Structures	Gradient Descent on Parameters	Breath-First with Pruning
Data Management Technique	Unspecified	Unspecified	Linear Scans

Specification of the model (or pattern) structures and the score function typically happens "off-line" as part of the human-centered process of setting up the data mining problem. Once the data, the model (or pattern) structures, and the score function have been decided upon, the remainder of the problem—optimizing the score function—is largely computational. (In practice there may be several iterations of this process as models and score functions are revised in light of earlier results). Thus, the algorithmic core of a data mining algorithm lies in the computational methods used to implement the search and data management components.

The component-based description presented in this chapter provides a general high-level framework for both *analysis* and *synthesis* of data mining algorithms. From an analysis viewpoint, describing existing data mining algorithms in terms of their components clarifies the role of each component and makes it easier to compare competing algorithms. For example, do two algorithms differ in terms of their model structures, their score functions, their search techniques, or their data management strategies? From a synthesis viewpoint, by combining different components in different combinations we can build data mining algorithms with different properties. In chapters 9 through 14 we will discuss each of the components in much more detail in the context of specific algorithms. In this chapter we will focus on how the pieces fit together at a high level. The primary theme here is that the component-based view of data mining algorithms provides a parsimonious and structured "language" for description, analysis, and synthesis of data mining algorithms.

For the most part we will limit the discussion to cases in which we have a single form of model or pattern structure (e.g., trees, polynomials, etc.), rather than those in which we are considering multiple types of model structures for the same problem. The component viewpoint can be generalized to handle such situations, but typically the score functions, the search method, and the data management techniques all become more complex.

5.2 An Example: The CART Algorithm for Building Tree Classifiers

To clarify the general idea of viewing algorithms in terms of their components, we will begin by looking at one well-known algorithm for classification problems.
The CART (Classification And Regression Trees) algorithm is a widely used statistical procedure for producing classification and regression models with a tree-based structure. For the sake of simplicity we will consider only the classification aspect of CART, that is, mapping an input vector **x** to a categorical (class) output label *y* (see figure 5.1). (A more detailed discussion of CART is provided in chapter 10.) In the context of the components discussed above, CART can be viewed as the "algorithm-tuple" consisting of the following:

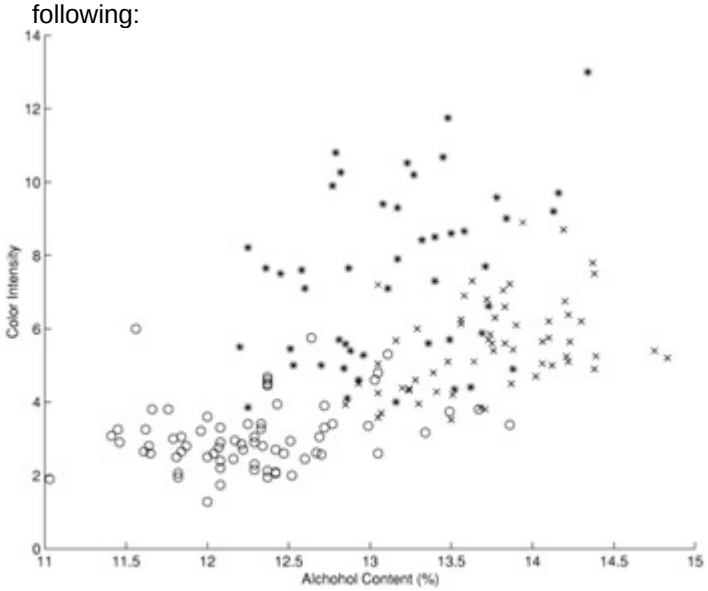

Figure 5.1: A Scatterplot of Data Showing Color Intensity versus Alcohol Content for a Set of Wines. The Data Mining Task is to Classify the Wines into One of Three Classes (Three Different Cultivars), Each Shown with a Different Symbol in the Plot. The Data Originate From a 13-Dimensional Data Set in Which Each Variable Measures of a Particular Characteristic of a Specific Wine.

1. *task = prediction (classification)*
2. *model structure = tree*
3. *score function = cross-validated loss function*
4. *search method = greedy local search*
5. *data management method = unspecified*

The fundamental distinguishing aspect of the CART algorithm is the *model structure* being used; the classification tree. The CART tree model consists of a hierarchy of univariate binary decisions. Figure 5.2 shows a simple example of such a classification tree for the data in figure 5.1. Each internal node in the tree specifies a binary test on a single variable, using thresholds on real and integer-valued variables and subset membership for categorical variables. (In general we use *b* branches at each node, *b* = 2.) A data vector **x** descends a unique path from the root node to a leaf node depending on how the values of individual components of **x** match the binary tests of the internal nodes. Each leaf node specifies the class label of the most likely class at that leaf or, more generally, a probability distribution on class values conditioned on the branch leading to that leaf.

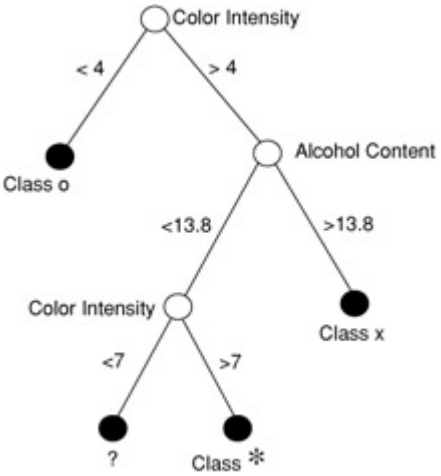

Figure 5.2: A Classification Tree for the Data in Figure 5.1 in Which the Tests Consist of Thresholds (Shown Beside the Branches) on Variables at Each Internal Node and Leaves Contain Class Decisions. Note that One Leaf is Denoted ? to Illustrate that there is Considerable Uncertainty About the Class Labels of Data Points in this Region of the Space.

The structure of the tree is derived from the data, rather than being specified *a priori* (this is where data mining comes in). CART operates by choosing the best variable for splitting the data into two groups at the root node. It can use any of several different splitting criteria; all produce the effect of partitioning the data at an internal node into two disjoint subsets (branches) in such a way that the class labels in each subset are as homogeneous as possible. This splitting procedure is then recursively applied to the data in each of the child nodes, and so forth. The size of the final tree is a result of a relatively complicated "pruning" process, outlined below. Too large a tree may result in overfitting, and too small a tree may have insufficient predictive power for accurate classification. The hierarchical form of the tree structure clearly separates algorithms like CART from classification algorithms based on non-tree structures (e.g., a model that uses a linear combination of all variables to define a decision boundary in the input space). A tree structure used for classification can readily deal with input data that contain *mixed* data types (i.e., combinations of categorical and real-valued data), since each internal node depends on only a simple binary test. In addition, since CART builds the tree using a single variable at a time, it can readily deal with large numbers of variables. On the other hand, the representational power of the tree structure is rather coarse: the decision regions for classifications are constrained to be hyper-rectangles, with boundaries constrained to be parallel to the input variable axes (as an example, see figure 5.3).

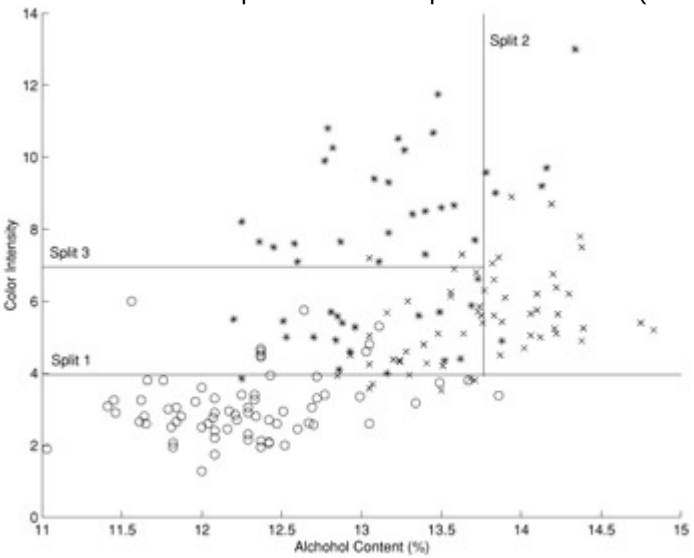

Figure 5.3: The Decision Boundaries From the Classification Tree in Figure 5.2 are Superposed on the Original Data. Note the Axis-Parallel Nature of the Boundaries.

The score function used to measure the quality of different tree structures is a general misclassification loss function, defined as

(5.1) $$\sum_{i=1}^{n} C\Big(y(i), \hat{y}(i)\Big),$$

where C ($y(i)$, $\hat{y}(i)$) is the loss incurred (positive) when the class label for the ith data vector, $y(i)$, is predicted by the tree to be $\hat{y}(i)$. In general, C is specified by an $m \times m$ matrix, where m is the number of classes. For the sake of simplicity we will assume here a loss of 1 is incurred whenever $\hat{y}(i) \neq y(i)$, and the loss is 0 otherwise. (This is known as the "0–1" loss function, or the misclassification rate if we normalize the sum above by dividing by n.)

CART uses a technique known as *cross-validation* to estimate this misclassification loss function. We will explain cross-validation in more detail in chapter 7. Basically, this method partitions the training data into a subset for building the tree and then estimates the misclassification rate on the remaining *validation* subset. This partitioning is repeated multiple times on different subsets, and the misclassification rates are then averaged to yield a *cross-validation estimate* of how well a tree of a particular size will perform on new, unseen data. The size of tree that produces the smallest cross-validated misclassification estimate is selected as the appropriate size for the final tree model. (This description captures the essence of tree selection via cross-validation, but in practice the process is a little more complex.)

Cross-validation allows CART to estimate the performance of any tree model on data not used in the construction of the tree—i.e., it provides an estimate of generalization performance. This is critical in the tree-growing procedure, since the misclassification rate on the training data (the data used to construct the tree) can often be reduced by simply making the tree more complex; thus, the training data error is not necessarily indicative of how the tree will perform on new data.

Figure 5.4 illustrates this point with a hypothetical plot of typical error rates as a function the size of the tree. The error rate on the training data decreases monotonically (to an error rate of zero if the variables can produce leaves that each contain data from a only single class). The test error rate on new data (which is what we are typically interested in for prediction) also decreases at first. Very small trees (to the left) do not have sufficient predictive power to make accurate predictions. However, unlike the training error, the test error "bottoms out" and begins to increase again as the algorithm overfits the data and adds nodes that are merely predicting noise or random variation in the training data, and which is irrelevant to the predictive task. The goal of an algorithm like CART is to find a tree close to the optimal tree size (which is of course unknown ahead of time); it tries to find a model that is complex enough to capture any structure that exists, but not so complex that it overfits. For small to medium amounts of data it is preferable to do this without having to reserve some of our data to estimate this out-of-sample error. For very large data sets we can sometimes afford to simply partition the data into training and validation data sets and to monitor performance on the validation data.

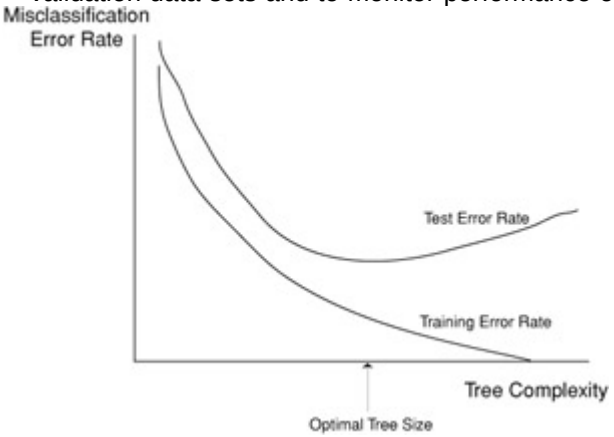

Figure 5.4: A Hypothetical Plot of Misclassification Error Rates for Both Training and Test Data as a Function of Tree Complexity (e.g., Number of Leaves in the Tree).

The use of a cross-validated score function distinguishes CART from most other data mining algorithms based on tree models. For example, the C4.5 algorithm (a widely used alternative to CART for building classification trees) judges individual tree structures by heuristically adjusting the estimated error rate on the training data to approximate the test error rate (in an attempt to correct for the fact that the training error rate is generally an underestimate of the out-of-sample error rate). The adjusted error rate is then used in a pruning phase to search for the tree that maximizes this score.

CART uses a greedy local search method to identify good candidate tree structures, recursively expanding the tree from a root node, and then gradually "pruning" back specific branches of this large tree. This heuristic search method is dictated by the combinatorially large search space (i.e., the space of all possible binary tree structures) and the lack of any tractable method for finding the single optimal tree (relative to a given score function). The folk wisdom in tree learning is that greedy local search in tree building works just about as well as any more sophisticated heuristic, and is much simpler to implement than more complex search methods. Thus, greedy local search is the method of choice in most practical tree learning algorithms.

In terms of data management, CART implicitly assumes that the data are all in main memory. To be fair to CART, very few algorithms published out-side the database literature provide any explicit guidance on data management for large data sets. For some algorithms, adding an appropriate data management technique is straightforward and can be done in a relatively modular fashion. For example, if each data point needs to be visited only once and the order does not matter, data management is trivial (just read the data points sequentially in subsets into main memory).

For tree algorithms, however, the model, the score function, and the search method are complex enough to make data management quite nontrivial. To understand why this is so, remember that a tree algorithm recursively partitions the observations (the rows of our data matrix) into subsets in a data-driven manner, requiring us to repeatedly find different subsets of observations in our database and determine various properties of these subsets. In a naive implementation of the algorithm for data sets too large to fit in main memory, this will involve many repeated scans of the secondary storage medium (such as a disk), leading to very poor time performance. Scalable versions of tree algorithms have been developed recently that use special purpose data structures to deal efficiently with data outside main memory.

To summarize our reductionist view of CART, we note that the algorithm consists of (1) a tree model structure, (2) a cross-validated score function, and (3) a two-phase greedy search over tree structures ("growing" and "pruning"). In this sense, CART is relatively straightforward to understand once one grasps the key ideas involved. Clearly, we could develop alternative algorithms that use the same tree structure, cross-validated score function, and search techniques, and that are similar in spirit to CART, but that are application-specific in details of implementation (such as how missing data are handled in both training and prediction). For a given data mining application, customizing the algorithm in this fashion might be well worth pursuing. In short, the power of an algorithm such as CART is in the fundamental concepts that it embodies, rather than in the specific details of implementation.

5.3 The Reductionist Viewpoint on Data Mining Algorithms

Repeating the basic mantra of this chapter, once we have a data set and a specific data mining task, a data mining algorithm can be thought of as a "tuple" consisting of {*model structure, score function, search method, data management technique*}. While this is a simple observation, it has some fairly profound implications. First, the number of different algorithms we can generate is very large! By combining different model structures with different score functions, different search methods, and different data management techniques, we can generate a potentially infinite number of different algorithms. (This point has not escaped academic researchers.)

However, the complexity of "algorithm space" is manageable once we realize the second implication: while there is a very large number of possible algorithms, there is only a relatively small number of fundamental "values" for each component in the tuple. Specifically, there are well-defined categories of models and patterns that we can use for problems such as regression, classification, or clustering; we will discuss these in detail in chapter 6. Similarly, as we will see in chapter 7, there are relatively few score functions (such as likelihood, sum-of-squared-errors, and classification rate) that have broad appeal. There are also just a few general classes of search and optimization methods that have wide applicability, and the essential principles of data management can be reduced to a relatively small number of different techniques (as discussed in chapters 8 and 12, respectively).

Thus, many well-known data mining algorithms are composed of a combination of well-defined components. In other words algorithms tend to be relatively tightly clustered in "algorithm space" (as spanned by the "dimensions" of model structure, score function, search method, and data management technique).
The *reductionist* (i.e., a component-based) view for data mining algorithms is quite useful in practice. It clarifies the underlying operation of a particular data mining algorithm by reducing it to its essential components. In turn, this makes it easier to compare different algorithms, since we can clearly see similarities and differences at the component level (e.g., we were able to distinguish between CART and C4.5 primarily in terms of what score functions they use).
Even more important, this view places an emphasis on the fundamental properties of an algorithm avoiding the tendency to think of lists of algorithms. When faced with a data mining application, a data miner should think about which *components* fit the specifics of his or her problem, rather than which specific "off-the-shelf" algorithm to choose. In an ideal world, the data miners would have available a software environment within which they could *compose* components (from a library of model structures, score functions, search methods, etc.) to synthesize an algoithm customized for their specific applications. Unfortunately this remains a ideal state of affairs rather than the practical norm; current data analysis software packages often provide only a list of algorithms, rather than a component-based toolbox for algorithm synthesis. This is understandable given the aim of providing usable tools for data miners who do not have the background or the time to understand the underlying details at a component level. However these software tools may not be ideal for more skilled practitioners who wish to customize and synthesize problem-specific algorithms. The "cookbook" approach is also somewhat dangerous, since naive users of data mining tools may not fully understand the limitations (and underlying assumptions) of the particular black-box algorithms they are using. In contrast, a description based on components makes it relatively clear what is inside the black box.

To illustrate the general utility of the reductionist viewpoint, in the next three sections we will look at three well-known algorithms in terms of their components. These and related algorithms will addressed in more detail in chapters 9 through 14, where we discuss a more complete range of solutions for different data mining tasks.

5.3.1 Multilayer Perceptrons for Regression and Classification

Feedforward multilayer perceptrons (MLPs) are the most widely used models in the general class of artificial neural network models. The MLP structure provides a nonlinear mapping from a real-valued input vector **x** to a real-valued output vector **y**. As a result, an MLP can be used as a nonlinear model for regression problems, as well as for classification, through appropriate interpretation of the outputs. The basic idea is that a vector of p input values is multiplied by a $p \times d_1$ weight matrix, and the resulting d_1 values are each individually transformed by a nonlinear function to produce d_1 "hidden node" outputs. The resulting d_1 values are then multiplied by a $d_1 \times d_2$ weight matrix (another "layer" of weights), and the d_2 values are each put through a non-linear function. The resulting d_2 values can either be used as the outputs of the model or be put through another layer of weight multiplications and non-linear transformations, and so on (hence, the "multilayer" nature of the model; the term *perceptron* refers to the original model of

this form proposed in the 1960s, consisting of a single layer of weights followed by a threshold nonlinearity).

As an example, consider the simple network model in figure 5.5 with a single "hidden" layer. Two inner products, $s_1 = \sum_{i=1}^{4} \alpha_i x_i$ and $s_2 = \sum_{i=1}^{4} \beta_i x_i$, are calculated via the first layer of weights (the as and the ßs), and each in turn transformed by a nonlinear function at the hidden nodes to produce two scalar values: h_1 and h_2. The nonlinear logistic function, i.e., $h_1 = h(s_1) = 1/(1 + e^{-s_1})$, is widely used. Next h_1 and h_2 are weighted and combined to produce the output value $y = \sum_{i=1}^{2} w_i h_i$ (we could in principle perform a nonlinear transformation on *y* also). Thus, *y* is a *nonlinear function* of the input vector **x**. The *h*s can be viewed as nonlinear transformations of the four-dimensional input, a new set of two "basis functions," h_1 and h_2. The parameters of this model to be estimated from the data are the eight weights on the input layer (a_1, ..., a_4, $ß_1$, ..., $ß_4$) and the two weights on the output layer (w_1 and w_2). In general, with *p* inputs, a single hidden layer with *h* hidden nodes, and a single output, there are $(p + 1)h$ parameters (weights) in all to be estimated from the data. In general we can have multiple layers of such weight multiplications and nonlinear transformations, but a single hidden layer is used most often since multiple hidden layer networks can be slow to train. The weights of the MLP are the parameters of the model and must be determined from the data.

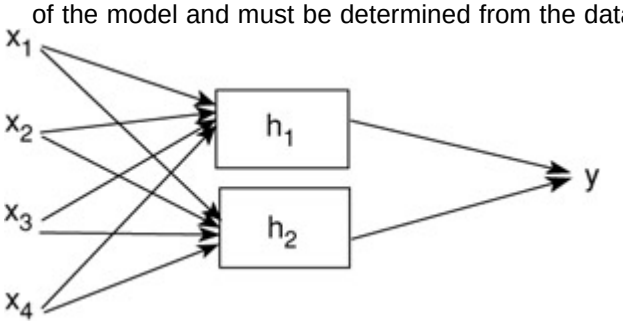

Figure 5.5: A Diagram of a Simple Multilayer Perceptron (or Neural Network) Model with Two Hidden Nodes ($d_1 = 2$) and a Single Output Node ($d_2 = 1$).

Note that if the output **y** is a scalar *y* (i.e., $d_2 = 1$) and is bounded between 0 and 1 (we can just choose a nonlinear transformation of the weighted values coming from the previous layer to ensure this condition), we can use *y* as an indicator of class membership for two-class problems and (for example) threshold at 0.5 to decide between class 1 and class 2. Thus, MLPs can easily be used for classification as well as for regression. Because of the nonlinear nature of the model, the decision boundaries between different classes produced by a network model can also be quite non-linear. Figure 5.6 provides an example of such decision boundaries. Note that they are highly nonlinear, in contrast to those produced by the classification tree in figure 5.3. Unlike the classification tree in figure 5.2, however, there is no simple summary form we can use to describe the workings of the neural network model.

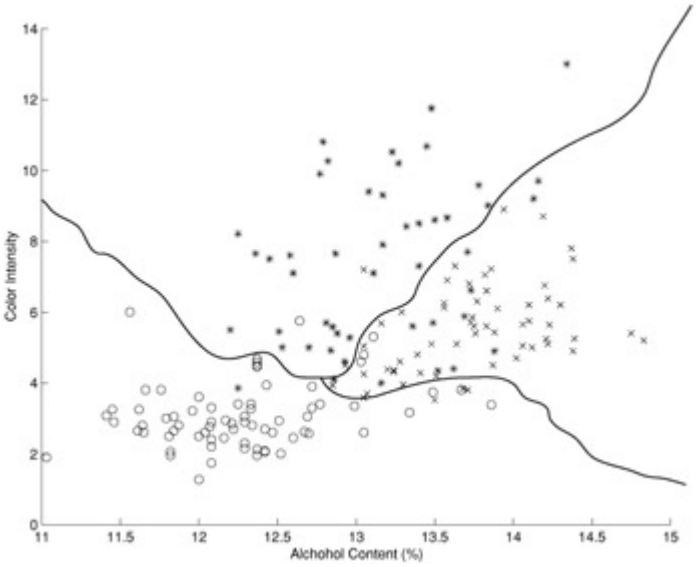

Figure 5.6: An Example of the Type of Decision Boundaries that a Neural Network Model Would Produce for the Two-Dimensional Wine Data of Figure 5.2(a).

The reductionist view of an MLP learning algorithm yields the following "algorithm-tuple":

1. *task = prediction: classification or regression*
2. *structure = multiple layers of nonlinear transformations of weighted sums of the inputs*
3. *score function = sum of squared errors*
4. *search method = steepest-descent from randomly chosen initial parameter values*
5. *data management technique = online or batch*

The distinguishing feature of this algorithm is the multilayer, nonlinear nature of its model structure (note both that the output *y* is a nonlinear function of the inputs and that the parameters ? (the weights) appear nonlinearly in the score function). This clearly sets a neural network apart from more traditional linear and polynomial functional forms for regression and from tree-based models for classification.

The sum of squared errors (SSE), the most widely used score function for MLPs, is defined as:

(5.2) $$S_{SSE} = \sum_{i}^{n} \Big(y(i) - \hat{y}(i) \Big)^2 ,$$

where *y*(*i*) and y(*i*) are the true target value and the output of the network, respectively, for the *i*th data point, and where y(*i*) is a function of the input vector **x**(*i*) and the MLP parameters (weights) ?. It is sometimes assumed that squared error is the *only* score function that can be used with a neural network model. In fact, as long as it is differentiable as a function of the model parameters (allowing us to determine the direction of steepest descent), *any* score function can be used as the basis for a steepest-descent search method such as backpropagation. For example, if we view squared error as just a special case of a more general log-likelihood function (as discussed in chapter 4), we can use a variety of other likelihood-based score functions in place of squared error, tailored for specific applications.

Training a neural network consists of minimizing S_{SSE} by treating it as a function of the unknown parameters ? (i.e., parameter estimation of ? given the data). Given that each y(*i*) is typically a highly nonlinear function of the parameters ?, the score function S_{SSE} is also highly nonlinear as a function of ?. Thus, there is no closed-form solution for finding the parameters ? that minimize S_{SSE} for an MLP. In addition, since there can be many local minima on the surface of S_{SSE} as a function of ?, training a neural network (i.e., finding the parameters that minimize S_{SSE} for a particular data set and model structure) is often a highly non-trivial multivariate optimization problem. Iterative local search techniques are required to find satisfactory local minima.

The original training method proposed for MLPs, known as backpropagation, is a relatively simple optimization method. It essentially performs steepest-descent on the score function (the sum of squared errors) in parameter space, solving this nonlinear optimization problem by descending to a local minimum given a randomly chosen starting point in parameter space. (In practice we usually descend from multiple starting points and select the best local minimum found overall.) In a more general context, there is a large family of optimization methods for such nonlinear optimization problems. It is often assumed that steepest-descent is the only optimization method that can be used to train an MLP, but in fact more powerful nonlinear optimization techniques such as conjugate gradient techniques can be brought to bear on this problem. We discuss some of these techniques in chapter 8.
In terms of data management, a neural network can be trained either online (updating the weights based on cycling through one data point at a time) or in batch mode (updating the weights after seeing all of the data points). The online updating version of the algorithm is a special case of a more general class of online estimation algorithms (see chapter 8 for further discussion of the trade-offs involved in using such algorithms). An important practical distinction between MLPs and classification trees is that a tree algorithm (such as CART) searches through models of different complexities in a relatively automated manner (e.g., finding the right-sized tree is a basic feature of the CART algorithm). In contrast, there is no widely accepted procedure for determining the appropriate structure for an MLP (i.e., determining how many layers and how many hidden nodes to include in the model). Numerous algorithms exist for constructing network structures automatically, including methods that start with small networks and add nodes and weights in an incremental "growing" manner, as well as methods that start with large networks and "prune" away weights and nodes that appear to be irrelevant. Incrementally growing a network structure can be subject to local minima problems (the best network with k hidden nodes may be quite different in parameter space from the best network with k - 1 hidden nodes). On the other hand, training an overly large network can be prohibitively expensive, especially when the model structure is large (e.g., with a large input dimensionality p). In practice, network structures are often determined by a trial-and-error procedure of manually adjusting the number of hidden nodes until satisfactory performance is reached on a validation data set (a set of data points not used in training).

The component-based view of MLPs illustrates that the general approach is not very far removed from more traditional statistical estimation and optimization techniques. Many of these techniques (e.g., the incorporation of Bayesian priors into the score function to drive small weights to zero (to "regularize" the model) or the use of more sophisticated multivariate optimization procedures such as conjugate gradient techniques during weight search) can be used in training network models. In the 1980s, when neural network models were first introduced, the connections to the statistical literature were not at all obvious (although they seem quite clear in retrospect). There is no doubt that the primary contribution of the neural modeling approach lies in the nonlinear multilayer nature of the underlying model structure.

5.3.2 The A Priori Algorithm for Association Rule Learning

Association rules are among the most popular representations for local patterns in data mining. Chapter 13 provides a more in-depth description, but here we sketch the general idea and briefly describe a generic association rule algorithm in terms of its components. (This description is loosely based on the well-known A Priori algorithm, which was one of the earliest algorithms for finding association rules.)

An association rule is a simple probabilistic statement about the co-occurrence of certain events in a database, and is particularly applicable to sparse transaction data sets. For the sake of simplicity we assume that all variables are binary. An association rule takes the following form:

(5.3) $$\text{IF } A = 1 \text{ AND } B = 1 \text{ THEN } C = 1 \text{ with probability } p$$

where A, B, and C are binary variables and $p = p(C = 1|A = 1, B = 1)$, i.e., the conditional probability that $C = 1$ given that $A = 1$ and $B = 1$. The conditional probability p is sometimes referred to as the "accuracy" or "confidence" of the rule, and $p(A = 1, B = 1,$

$C = 1$) is referred to as the "support." This *pattern structure* or *rule structure* is quite simple and interpretable, which helps explain the general appeal of this approach. Typically the goal is to find all rules that satisfy the constraint that the accuracy p is greater than some threshold p_a and the support is greater than some threshold p_s (for example, to find all rules with support greater than 0.05 and accuracy greater than 0.8). Such rules comprise a relatively weak form of knowledge; they are really just summaries of co-occurrence patterns in the observed data, rather than strong statements that characterize the population as a whole. Indeed, in the sense that the term "rule" usually implies a causal interpretation (from the left to the right hand side), the term "association rule" is strictly speaking a misnomer since these patterns are inherently correlational but need not be causal.

The general idea of finding association rules originated in applications involving "market-basket data." These data are usually recorded in a database in which each observation consists of an actual basket of items (such as grocery items), and the variables indicate whether or not a particular item was purchased. We can think of this type of data in terms of a data matrix of n rows (corresponding to baskets) and p columns (corresponding to grocery items). Such a matrix can be very large, with n in the millions and p in the tens of thousands, and is generally very sparse, since a typical basket contains only a few items. Association rules were invented as a way to find simple patterns in such data in a relatively efficient computational manner.

In our reductionist framework, a typical data mining algorithm for association rules has the following components:

1. *task = description: associations between variables*
2. *structure = probabilistic "association rules" (patterns)*
3. *score function = thresholds on accuracy and support*
4. *search method = systematic search (breadth-first with pruning)*
5. *data management technique = multiple linear scans*

The score function used in association rule searching is a simple binary function. There are two thresholds: p_s is a lower bound on the support of the rule (e.g., $p_s = 0.1$ when we want only those rules that cover at least 10% of the data) and p_a is a lower bound on the accuracy of the rule (e.g., $p_a = 0.9$ when we want only rules that are at least 90% accurate). A pattern gets a score of 1 if it satisfies both of the threshold conditions, and gets a score of 0 otherwise. The goal is find all rules (patterns) with a score of 1.

The search problem is formidable given the exponential number of possible association rules—namely, $O(p2^{p-1})$ for binary variables if we limit our attention to rules with positive propositions (e.g., $A = 1$) in the left and right-hand sides. Nonetheless, by taking advantage of the nature of the score function, we can reduce the average run-time of the algorithm to much more manageable proportions. Note that if either $p(A = 1) = p_s$ or $p(B = 1) = p_s$, clearly $p(A = 1, B = 1) = p_s$. We can use this observation in our search for association rules by first finding all of the individual events (such as $A = 1$) that have a probability greater than the threshold p_s (this takes one linear scan of the entire database). An event (or set of events) is called "frequent" if the probability of the event(s) is greater than the support threshold p_s. We consider all possible pairs of these frequent events to be candidate frequent sets of size 2.

In the more general case of going from frequent sets of size $k - 1$ to frequent sets of size k, we can *prune* any sets of size k that contain a subset of $k - 1$ items that themselves are not frequent at the $k - 1$ level. For example, if we had only frequent sets $\{A = 1, B = 1\}$ and $\{B = 1, C = 1\}$, we could combine them to get the candidate $k = 3$ frequent set $\{A = 1, B = 1, C = 1\}$. However, if the subset of items $\{A = 1, C = 1\}$ was not frequent (i.e., this item set were not on the list of frequent sets of size $k = 2$), then $\{A = 1, B = 1, C = 1\}$ could not be frequent either, and it could safely be pruned. Note that this pruning can take place without searching the data directly, resulting in a considerable computational speedup for large data sets.

Given the pruned list of candidate frequent sets of size k, the algorithm performs another linear scan of the database to determine which of these sets are in fact frequent. The confirmed frequent sets of size k (if any) are combined to generate all possible frequent sets containing $k + 1$ events, followed by pruning, and then another scan of the database, and so on—until no more frequent sets can be generated. (In the worst case, all possible sets of events are frequent and the algorithm takes exponential time.

However, since in practice the data are often very sparse for the types of transaction data sets analyzed by these algorithms, the cardinality of the largest frequent set is usually quite small (relative to n), at least for relatively large support values.) The algorithm then makes one final linear scan through the data set, using the list of all frequent sets that have been found. It determines which subset combinations of the frequent sets also satisfy the accuracy threshold when expressed as a rule, and then returns the corresponding association rules.

Association rule algorithms comprise an interesting class of data mining algorithms in that the search and data management components are their most critical components. In particular, association rule algorithms use a systematic breadth-first, general-to-specific search method that explicitly tries to minimize the number of linear scans through the database. While there exist numerous other rule-finding algorithms in the machine learning literature (with similar rule-based representations), association rule algorithms are designed specifically to operate on very large data sets in a relatively efficient manner. Thus, for example, research papers on association rule algorithms tend to emphasize computational efficiency rather than interpretation of the rules that the algorithms produce.

5.3.3 Vector-Space Algorithms for Text Retrieval

The general task of "retrieval by content" is loosely described as follows: we have a query object and a large database of objects, and we would like to find the k objects in the database that are most similar to the query object. We are all familiar with this problem in the context of searching through online collections of text. For example, our query could be a short set of keywords and the "database" could correspond to a large set of Web pages. Our task in this case would be to find the Web pages that are most relevant to our keywords.

Chapter 14 discusses this retrieval task in greater depth. Here we look at a generic text retrieval algorithm in terms of its components. One of the most important aspects of this problem is how similarity is defined. Text documents are of different lengths and structure. How can we compare such diverse documents? A key idea in text retrieval is to reduce all documents to a uniform vector representation, as follows. Let t_1, ..., t_p be p *terms* (words, phrases, etc.). We can think of these as variables, or columns in our data matrix. A document (a row in our data matrix) is represented by a vector of length p, where the ith component contains the count of how often term t_i appears in the document. As with market-basket data, in practice we can have a very large data matrix (n in the millions, p in the tens of thousands) that is very sparse (most documents will have many zeros). Again, of course, we normally would not actually store the data as a large $n \times p$ matrix: a more efficient representation is to store a list for each term t_i of all the documents containing that term.

Given this "vector-space" representation, we can now readily define similarity. One simple definition is to make the similarity distance a function of the angle between the two vectors in p-space. The angle measures similarity in a given direction in "term-space" and factors out any differences arising from the fact that large documents tend to have more occurrences of a word than small documents. The vector-space representation and the angle similarity measure may seem relatively primitive, but in practice this scheme works surprisingly well, and there exists a multitude of variations on this basic theme in text retrieval.

With this information, we are ready to define the components of a simple generic text-retrieval algorithm that takes one document and finds the k most similar documents:

1. *task = retrieval of the k most similar documents in a database relative to a given query*
2. *representation = vector of term occurrences*
3. *score function = angle between two vectors*
4. *search method = various techniques*
5. *data management technique = various fast indexing strategies*

There are many variations on the specific definitions of the components given above. For example, in defining the score function, we can specify similarity metrics more general than the angle function. In specifying the search method, various heuristic search techniques are possible. Note that search in this context is *real-time search*, since the

algorithm has to retrieve the patterns in realtime for a user (unlike the data mining algorithms we looked at earlier, for which search meant off-line searching for the optimal parameters and model structures).
Different applications may call for different components to be used in a retrieval algorithm. For example, in searching through legal documents, the absence of particular terms might be significant, and we might want to reflect this in our definition of a score function. In a different context we might want the opposite effect, i.e., to downweight the fact that two documents do *not* contain certain terms (relative to the terms they have in common).
It is clear, however, that the model representation is really the key idea here. Once the use vector representation has been established, we can define a wide range of similarity metrics in vector-space, and we can use standard search and indexing techniques to find near neighbors in sparse *p*-dimensional space. Different retrieval algorithms may vary in the details of the score function or search methods, but most share the same underlying vector representation of the data. Were we to define a different representation for a document (say a generative model for the data based on some form of grammar), we would probably have to come up with fundamentally different score functions and search methods.

5.4 Discussion

For the novice and the seasoned researcher alike, wandering through the jungle of data mining algorithms can be somewhat bewildering. We hope that the component-based view presented in this chapter provides a useful practical tool for the reader in evaluating algorithms. The process is as follows: try to strip away the jargon and marketing spin that are inevitable in any research paper or product literature, and reduce the algorithm to its basic components. The component-based description provides a well-defined and "calibrated" framework on which to base comparisons—e.g., we can compare a new algorithm to other well-known algorithms and see precisely how it differs in terms of its components, if it differs at all.
It is interesting to note the different emphases placed on algorithmic aspects of data mining in different research communities. A cursory glance through most statistical journals will reveal plenty of equations specifying models, score functions, and computational methods, with relatively few detailed algorithmic specifications of how the models will be fit in practice. Conversely, computer science journals on machine learning and pattern recognition often emphasize the computational methods and algorithms, with little emphasis on the appropriateness of either the structure of the model or the score function being used to fit it. For example, it is not uncommon to see empirical comparisons being made among *algorithms*, rather than among the underlying models or score functions. In the context of data mining, the different emphases in the two research areas have led to the development of quite different (and often complementary) methodologies. Statistical approaches often place significant emphasis on theoretical aspects of inference procedures (e.g., parameter estimation and model selection) and less emphasis on computational issues. Computer science approaches to data mining tend to do the reverse, focusing more on efficient search and data management and less on the appropriateness of the model (and pattern) structures, or on the relevance of the score function. This "cultural" difference is worth keeping in mind throughout this text, as it helps to explain the factors that motivated the development of specific models, inference methods, and algorithms within these two research communities.

For both the statistical and the computer science schools of thought, it is probably fair to say that the typical research paper is not very clear on what the underlying components of a particular algorithm are. The literature is replete with fancy-sounding names and acronyms for different algorithms. In many papers, the descriptions of the model structure, the score function, and the search method are abstrusely intertwined.
In practice, *all* components of a data mining algorithm are essential. The *relative* importance of the model, the score function, and the computational implementation varies from problem to problem. For small data sets, the interpretability and predictive power of the model may be (relatively speaking) a much more important factor than any