Chapter 10

Data Classification

"*Science is the systematic classification of experience.*"—George Henry Lewes

10.1 Introduction

The classification problem is closely related to the clustering problem discussed in Chaps. 6 and 7. While the clustering problem is that of determining similar groups of data points, the classification problem is that of *learning* the structure of a data set of examples, *already partitioned into groups*, that are referred to as *categories* or *classes*. The learning of these categories is typically achieved with a *model*. This model is used to estimate the group identifiers (or *class labels*) of one or more previously unseen data examples with unknown labels. Therefore, one of the inputs to the classification problem is an example data set that has already been partitioned into different classes. This is referred to as the *training data*, and the group identifiers of these classes are referred to as class labels. In most cases, the class labels have a clear semantic interpretation in the context of a specific application, such as a group of customers interested in a specific product, or a group of data objects with a desired property of interest. The model learned is referred to as the *training model*. The previously unseen data points that need to be classified are collectively referred to as the *test data set*. The algorithm that creates the training model for prediction is also sometimes referred to as the *learner*.

Classification is, therefore, referred to as *supervised* learning because an example data set is used to learn the structure of the groups, just as a teacher supervises his or her students towards a specific goal. While the groups learned by a classification model may often be related to the similarity structure of the feature variables, as in clustering, this need not necessarily be the case. In classification, the example training data is paramount in providing the guidance of how groups are defined. Given a data set of test examples, the groups created by a classification model on the test examples will try to mirror the number and structure of the groups available in the example data set of training instances. Therefore, the classification problem may be intuitively stated as follows:

Given a set of training data points, each of which is associated with a class label, determine the class label of one or more previously unseen test instances.

Most classification algorithms typically have two phases:

1. *Training phase:* In this phase, a training model is constructed from the training instances. Intuitively, this can be understood as a summary mathematical model of the labeled groups in the training data set.

2. *Testing phase:* In this phase, the training model is used to determine the class label (or group identifier) of one or more unseen test instances.

The classification problem is more powerful than clustering because, unlike clustering, it captures a *user-defined notion of grouping* from an example data set. Such an approach has almost direct applicability to a wide variety of problems, in which groups are defined naturally based on *external application-specific criteria*. Some examples are as follows:

1. *Customer target marketing:* In this case, the groups (or labels) correspond to the user interest in a particular product. For example, one group may correspond to customers interested in a product, and the other group may contain the remaining customers. In many cases, training examples of previous buying behavior are available. These can be used to provide examples of customers who may or may not be interested in a specific product. The feature variables may correspond to the demographic profiles of the customers. These training examples are used to learn whether or not a customer, with a known demographic profile, but unknown buying behavior, may be interested in a particular product.

2. *Medical disease management:* In recent years, the use of data mining methods in medical research has gained increasing traction. The features may be extracted from patient medical tests and treatments, and the class label may correspond to treatment outcomes. In these cases, it is desired to predict treatment outcomes with models constructed on the features.

3. *Document categorization and filtering:* Many applications, such as newswire services, require real-time classification of documents. These are used to organize the documents under specific topics in Web portals. Previous examples of documents from each topic may be available. The features correspond to the words in the document. The class labels correspond to the various topics, such as politics, sports, current events, and so on.

4. *Multimedia data analysis:* It is often desired to perform classification of large volumes of multimedia data such as photos, videos, audio, or other more complex multimedia data. Previous examples of particular activities of users associated with example videos may be available. These may be used to determine whether a particular video describes a specific activity. Therefore, this problem can be modeled as a binary classification problem containing two groups corresponding to the occurrence or nonoccurrence of a specific activity.

The applications of classification are diverse because of the ability to *learn by example.*

It is assumed that the training data set is denoted by $\mathcal{D}$ with n data points and d features, or dimensions. In addition, each of the data points in $\mathcal{D}$ is associated with a label drawn from $\{1 \dots k\}$. In some models, the label is assumed to be binary ($k = 2$) for simplicity. In the latter case, a commonly used convention is to assume that the labels are drawn from $\{-1, +1\}$. However, it is sometimes notationally convenient to assume that the labels are drawn from $\{0, 1\}$. This chapter will use either of these conventions depending on the classifier. A training model is constructed from $\mathcal{D}$, which is used to predict the label of unseen test instances. The output of a classification algorithm can be one of two types:

1. *Label prediction:* In this case, a label is predicted for each test instance.

2. *Numerical score:* In most cases, the learner assigns a score to each instance–label combination that measures the propensity of the instance to belong to a particular class. This score can be easily converted to a label prediction by using either the maximum value, or a cost-weighted maximum value of the numerical score across different classes. One advantage of using a score is that different test instances can be compared and ranked by their propensity to belong to a particular class. Such scores are particularly useful in situations where one of the classes is very rare, and a numerical score provides a way to determine the top *ranked* candidates belonging to that class.

A subtle but important distinction exists in the design process of these two types of models, especially when numerical scores are used for ranking different test instances. In the first model, the training model does not need to account for the relative classification propensity across different *test instances*. The model only needs to worry about the relative propensity towards different *labels* for a specific instance. The second model also needs to properly normalize the classification scores across different test instances so that they can be meaningfully compared for ranking. Minor variations of most classification models are able to handle either the labeling or the ranking scenario.

When the training data set is small, the performance of classification models is sometimes poor. In such cases, the model may describe the specific random characteristics of the training data set, and it may not *generalize* to the group structure of *previously unseen* test instances. In other words, such models might accurately predict the labels of instances used to construct them, but they perform poorly on unseen test instances. This phenomenon is referred to as *overfitting*. This issue will be revisited several times in this chapter and the next.

Various models have been designed for data classification. The most well-known ones include decision trees, rule-based classifiers, probabilistic models, instance-based classifiers, support vector machines, and neural networks. The modeling phase is often preceded by a feature selection phase to identify the most informative features for classification. Each of these methods will be addressed in this chapter.

This chapter is organized as follows. Section 10.2 introduces some of the common models used for feature selection. Decision trees are introduced in Sect. 10.3. Rule-based classifiers are introduced in Sect. 10.4. Section 10.5 discusses probabilistic models for data classification. Section 10.6 introduces support vector machines. Neural network classifiers are discussed in Sect. 10.7. Instance-based learning methods are explained in Sect. 10.8. Evaluation methods are discussed in Sect. 10.9. The summary is presented in Sect. 10.10.

10.2 Feature Selection for Classification

Feature selection is the first stage in the classification process. Real data may contain features of varying relevance for predicting class labels. For example, the gender of a person is less relevant for predicting a disease label such as "diabetes," as compared to his or her age. Irrelevant features will typically harm the accuracy of the classification model in addition to being a source of computational inefficiency. Therefore, the goal of feature selection algorithms is to select the most informative features with respect to the class label. Three primary types of methods are used for feature selection in classification.

1. *Filter models:* A crisp mathematical criterion is available to evaluate the quality of a feature or a subset of features. This criterion is then used to filter out irrelevant features.

2. *Wrapper models:* It is assumed that a classification algorithm is available to evaluate how well the algorithm performs with a particular subset of features. A feature search algorithm is then wrapped around this algorithm to determine the relevant set of features.

3. *Embedded models:* The solution to a classification model often contains useful hints about the most relevant features. Such features are isolated, and the classifier is retrained on the pruned features.

In the following discussion, each of these models will be explained in detail.

10.2.1 Filter Models

In filter models, a feature or a subset of features is evaluated with the use of a class-sensitive discriminative criterion. The advantage of evaluating a *group* of features at one time is that redundancies are well accounted for. Consider the case where two feature variables are perfectly correlated with one another, and therefore each can be predicted using the other. In such a case, it makes sense to use only one of these features because the other adds no *incremental* knowledge with respect to the first. However, such methods are often expensive because there are 2^d possible subsets of features on which a search may need to be performed. Therefore, in practice, most feature selection methods evaluate the features independently of one another and select the most discriminative ones.

Some feature selection methods, such as *linear discriminant analysis*, create a linear combination of the original features as a new set of features. Such analytical methods can be viewed either as stand-alone classifiers or as dimensionality reduction methods that are used *before* classification, depending on how they are used. These methods will also be discussed in this section.

10.2.1.1 Gini Index

The Gini index is commonly used to measure the discriminative power of a particular feature. Typically, it is used for categorical variables, but it can be generalized to numeric attributes by the process of discretization. Let $v_1 \ldots v_r$ be the r possible values of a particular categorical attribute, and let p_j be the fraction of data points containing attribute value v_i that belong to the class $j \in \{1 \ldots k\}$ for the attribute value v_i. Then, the Gini index $G(v_i)$ for the value v_i of a categorical attribute is defined as follows:

$$G(v_i) = 1 - \sum_{j=1}^{k} p_j^2. \tag{10.1}$$

When the different classes are distributed evenly for a particular attribute value, the value of the Gini index is $1 - 1/k$. On the other hand, if all data points for an attribute value

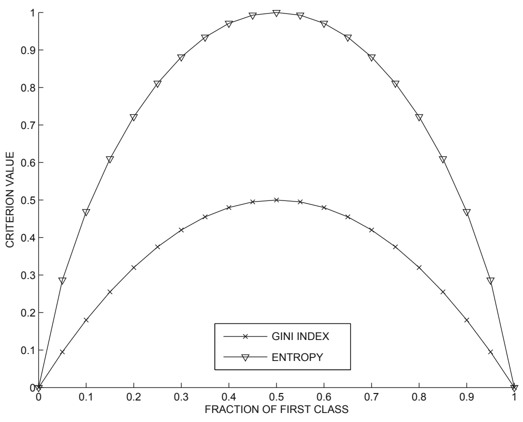


Figure 10.1: Variation of two feature selection criteria with class distribution skew

v_i belong to the same class, then the Gini index is 0. Therefore, lower values of the Gini index imply greater discrimination. An example of the Gini index for a two-class problem for varying values of p_1 is illustrated in Fig. 10.1. Note that the index takes on its maximum value at $p_1 = 0.5$.

The value-specific Gini index is converted into an attributewise Gini index. Let n_i be the number of data points that take on the value v_i for the attribute. Then, for a data set containing $\sum_{i=1}^{r} n_i = n$ data points, the overall Gini index G for the attribute is defined as the weighted average over the different attribute values as follows:

$$G = \sum_{i=1}^{r} n_i G(v_i)/n. \tag{10.2}$$

Lower values of the Gini index imply greater discriminative power. The Gini index is typically defined for a particular feature rather than a subset of features.

10.2.1.2 Entropy

The class-based entropy measure is related to notions of *information gain* resulting from fixing a specific attribute value. The entropy measure achieves a similar goal as the Gini index at an intuitive level, but it is based on sound information-theoretic principles. As before, let p_j be the fraction of data points belonging to the class j for attribute value v_i. Then, the class-based entropy $E(v_i)$ for the attribute value v_i is defined as follows:

$$E(v_i) = -\sum_{j=1}^{k} p_j \log_2(p_j). \tag{10.3}$$

The class-based entropy value lies in the interval $[0, \log_2(k)]$. Higher values of the entropy imply greater "mixing" of different classes. A value of 0 implies perfect separation, and, therefore, the largest possible discriminative power. An example of the entropy for a two-class problem with varying values of the probability p_1 is illustrated in Fig. 10.1. As in the case of the Gini index, the overall entropy E of an attribute is defined as the weighted average over the r different attribute values:

$$E = \sum_{i=1}^{r} n_i E(v_i)/n. \tag{10.4}$$

Here, n_i is the frequency of attribute value v_i.

10.2.1.3 Fisher Score

The Fisher score is naturally designed for numeric attributes to measure the ratio of the average interclass separation to the average intraclass separation. The larger the Fisher score, the greater the discriminatory power of the attribute. Let μ_j and σ_j, respectively, be the mean and standard deviation of data points belonging to class j for a particular feature, and let p_j be the fraction of data points belonging to class j. Let μ be the global mean of the data on the feature being evaluated. Then, the Fisher score F for that feature may be defined as the ratio of the interclass separation to intraclass separation:

$$F = \frac{\sum_{j=1}^{k} p_j(\mu_j - \mu)^2}{\sum_{j=1}^{k} p_j \sigma_j^2}. \tag{10.5}$$

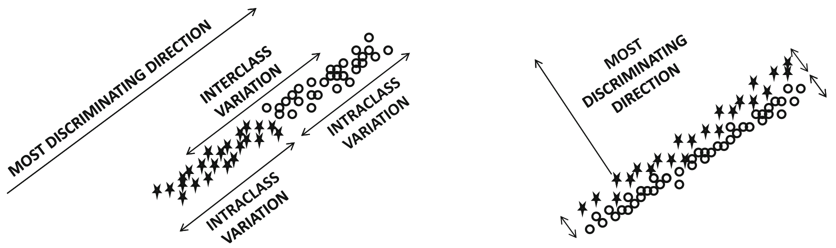
The numerator quantifies the average interclass separation, whereas the denominator quantifies the average intraclass separation. The attributes with the largest value of the Fisher score may be selected for use with the classification algorithm.

10.2.1.4 Fisher's Linear Discriminant

Fisher's linear discriminant may be viewed as a generalization of the Fisher score in which newly created features correspond to linear combinations of the original features rather than a subset of the original features. This direction is designed to have a high level of discriminatory power with respect to the class labels. Fisher's discriminant can be viewed as a *supervised* dimensionality reduction method in contrast to *PCA*, which maximizes the preserved variance in the feature space but does not maximize the *class-specific* discrimination. For example, the most discriminating direction is aligned with the highest variance direction in Fig. 10.2a, but it is aligned with the lowest variance direction in Fig. 10.2b. In each case, if the data were to be projected along the most discriminating direction $\overline{W}$, then the *ratio* of interclass to intraclass separation is maximized. How can we determine such a d-dimensional vector $\overline{W}$?

The selection of a direction with high discriminative power is based on the same quantification as the Fisher score. Fisher's discriminant is naturally defined for the two-class scenario, although generalizations exist for the case of multiple classes. Let $\overline{\mu_0}$ and $\overline{\mu_1}$ be the d-dimensional row vectors representing the means of the data points in the two classes, and let Σ_0 and Σ_1 be the corresponding $d \times d$ covariance matrices in which the (i, j)th entry represents the covariance between dimensions i and j for that class. The fractional presence of the two classes are denoted by p_0 and p_1, respectively. Then, the equivalent Fisher score $FS(\overline{W})$ for a d-dimensional row vector $\overline{W}$ may be written in terms of *scatter* matrices, which are weighted versions of covariance matrices:

$$\begin{aligned} FS(\overline{W}) &= \frac{\text{Between Class Scatter along } \overline{W}}{\text{Within Class Scatter along } \overline{W}} \propto \frac{(\overline{W} \cdot \overline{\mu_1} - \overline{W} \cdot \overline{\mu_0})^2}{p_0[\text{Variance(Class 0)}] + p_1[\text{Variance(Class 1)}]} \\ &= \frac{\overline{W}[(\overline{\mu_1} - \overline{\mu_0})^T(\overline{\mu_1} - \overline{\mu_0})]\overline{W}^T}{p_0[\overline{W}\Sigma_0\overline{W}^T] + p_1[\overline{W}\Sigma_1\overline{W}^T]} = \frac{[\overline{W} \cdot (\overline{\mu_1} - \overline{\mu_0})]^2}{\overline{W}(p_0\Sigma_0 + p_1\Sigma_1)\overline{W}^T}. \end{aligned}$$



(a) Discriminating direction is aligned with high-variance direction (b) Discriminating direction is aligned with low-variance direction

Figure 10.2: Impact of class distribution on Fisher's discriminant direction

Note that the quantity $\overline{W}\Sigma_i\overline{W}^T$ in one of the aforementioned expressions represents the variance of the projection of a data set along $\overline{W}$, whose covariance matrix is Σ_i. This result is derived in Sect. 2.4.3.1 of Chap. 2. The rank-1 matrix $S_b = [(\overline{\mu_1} - \overline{\mu_0})^T(\overline{\mu_1} - \overline{\mu_0})]$ is also referred[1] to as the (scaled) *between-class scatter-matrix* and the matrix $S_w = (p_0\Sigma_0 + p_1\Sigma_1)$ is the (scaled) *within-class scatter matrix*. The quantification $FS(\overline{W})$ is a direct generalization of the *axis-parallel* score in Eq. 10.5 to an arbitrary direction $\overline{W}$. The goal is to determine a direction $\overline{W}$ that maximizes the Fisher score. It can be shown[2] that the optimal direction $\overline{W^*}$, expressed as a row vector, is given by the following:

$$\overline{W^*} \propto (\overline{\mu_1} - \overline{\mu_0})(p_0\Sigma_0 + p_1\Sigma_1)^{-1}. \tag{10.6}$$

If desired, successive orthogonal directions may be determined by iteratively projecting the data into the orthogonal subspace to the optimal directions found so far, and determining the Fisher's discriminant in this reduced subspace. The final result is a new representation of lower dimensionality that is more discriminative than the original feature space. Interestingly, the matrix $S_w + p_0p_1S_b$ can be shown to be invariant to the values of the class labels of the data points (see Exercise 21), and it is equal to the covariance matrix of the data. Therefore, the top-k eigenvectors of $S_w + p_0p_1S_b$ yield the basis vectors of *PCA*.

This approach is often used as a stand-alone classifier, which is referred to as *linear discriminant analysis*. A perpendicular hyperplane $\overline{W^*}\cdot\overline{X}+b = 0$ to the most discriminating direction is used as a binary class separator. The optimal value of b is selected based on the accuracy with respect to the training data. This approach can also be viewed as projecting the training points along the most discriminating vector $\overline{W^*}$, and then selecting the value of b to decide the point on the line that best separates the two classes. The Fisher's discriminant for binary classes can be shown to be a special case of *least-squares regression* for numeric classes, in which the response variables are set to $-1/p_0$ and $+1/p_1$, respectively, for the two classes (cf. Sect. 11.5.1.1 of Chap. 11).

[1]The unscaled versions of the two scatter matrices are $np_0p_1S_b$ and nS_w, respectively. The sum of these two matrices is the total scatter matrix, which is n times the covariance matrix (see Exercise 21).

[2]Maximizing $FS(\overline{W}) = \frac{\overline{W}S_b\overline{W}^T}{\overline{W}S_w\overline{W}^T}$ is the same as maximizing $\overline{W}S_b\overline{W}^T$ subject to $\overline{W}S_w\overline{W}^T = 1$. Setting the gradient of the Lagrangian relaxation $\overline{W}S_b\overline{W}^T - \lambda(\overline{W}S_w\overline{W}^T - 1)$ to 0 yields the generalized eigenvector condition $S_b\overline{W}^T = \lambda S_w\overline{W}^T$. Because $S_b\overline{W}^T = (\overline{\mu_1}^T - \overline{\mu_0}^T)\left[(\overline{\mu_1} - \overline{\mu_0})\overline{W}^T\right]$ always points in the direction of $(\overline{\mu_1}^T - \overline{\mu_0}^T)$, it follows that $S_w\overline{W}^T \propto \overline{\mu_1}^T - \overline{\mu_0}^T$. Therefore, we have $\overline{W} \propto (\overline{\mu_1} - \overline{\mu_0})S_w^{-1}$.

10.2.2 Wrapper Models

Different classification models are more accurate with different sets of features. Filter models are agnostic to the particular classification algorithm being used. In some cases, it may be useful to leverage the characteristics of the specific classification algorithm to select features. As you will learn later in this chapter, a linear classifier may work more effectively with a set of features where the classes are best modeled with linear separators, whereas a distance-based classifier works well with features in which distances reflect class distributions.

Therefore, one of the inputs to wrapper-based feature selection is a specific classification induction algorithm, denoted by $\mathcal{A}$. Wrapper models can optimize the feature selection process to the classification algorithm at hand. The basic strategy in wrapper models is to iteratively refine a current set of features F by successively adding features to it. The algorithm starts by initializing the current feature set F to $\{\}$. The strategy may be summarized by the following two steps that are executed iteratively:

1. Create an augmented set of features F by adding one or more features to the current feature set.

2. Use a classification algorithm $\mathcal{A}$ to evaluate the accuracy of the set of features F. Use the accuracy to either accept or reject the augmentation of F.

The augmentation of F can be performed in many different ways. For example, a greedy strategy may be used where the set of features in the previous iteration is augmented with an additional feature with the greatest discriminative power with respect to a filter criterion. Alternatively, features may be selected for addition via random sampling. The accuracy of the classification algorithm $\mathcal{A}$ in the second step may be used to determine whether the newly augmented set of features should be accepted, or one should revert to the set of features in the previous iteration. This approach is continued until there is no improvement in the current feature set for a minimum number of iterations. Because the classification algorithm $\mathcal{A}$ is used in the second step for evaluation, the final set of identified features will be sensitive to the choice of the algorithm $\mathcal{A}$.

10.2.3 Embedded Models

The core idea in embedded models is that the solutions to many classification formulations provide important hints about the most relevant features to be used. In other words, knowledge about the features is *embedded* within the solution to the classification problem. For example, consider a linear classifier that maps a training instance $\overline{X}$ to a class label y_i in $\{-1, 1\}$ using the following linear relationship:

$$y_i = \text{sign}\{\overline{W} \cdot \overline{X} + b\}. \tag{10.7}$$

Here, $\overline{W} = (w_1, \ldots w_d)$ is a d-dimensional vector of coefficients, and b is a scalar that is learned from the training data. The function "sign" maps to either -1 or $+1$, depending on the sign of its argument. As we will see later, many linear models such as Fisher's discriminant, support vector machine (SVM) classifiers, logistic regression methods, and neural networks use this model.

Assume that all features have been normalized to unit variance. If the value of $|w_i|$ is relatively[3] small, the ith feature is used very weakly by the model and is more likely to be noninformative. Therefore, such dimensions may be removed. It is then possible to train the

[3]Certain variations of linear models, such as L_1-regularized SVMs or *Lasso* (cf. Sect. 11.5.1 of Chap. 11), are particularly effective in this context. Such methods are also referred to as *sparse learning* methods.

Table 10.1: Training data snapshot relating the salary and age features to charitable donation propensity

Name	**Age**	**Salary**	**Donor?**
Nancy	21	37,000	N
Jim	27	41,000	N
Allen	43	61,000	Y
Jane	38	55,000	N
Steve	44	30,000	N
Peter	51	56,000	Y
Sayani	53	70,000	Y
Lata	56	74,000	Y
Mary	59	25,000	N
Victor	61	68,000	Y
Dale	63	51,000	Y

same (or a different) classifier on the data with the pruned feature set. If desired, statistical tests may be used to decide when the value of $|w_i|$ should be considered sufficiently small. Many decision tree classifiers, such as *ID3*, also have feature selection methods embedded in them.

In *recursive* feature elimination, an iterative approach is used. A small number of features are removed in each iteration. Then, the classifier is retrained on the pruned set of features to re-estimate the weights. The re-estimated weights are used to again prune the features with the least absolute weight. This procedure is repeated until all remaining features are deemed to be sufficiently relevant. Embedded models are generally designed in an ad hoc way, depending on the classifier at hand.

10.3 Decision Trees

Decision trees are a classification methodology, wherein the classification process is modeled with the use of a set of *hierarchical* decisions on the feature variables, arranged in a tree-like structure. The decision at a particular node of the tree, which is referred to as the *split criterion*, is typically a condition on one or more feature variables in the training data. The split criterion divides the training data into two or more parts. For example, consider the case where *Age* is an attribute, and the split criterion is $Age \leq 30$. In this case, the left branch of the decision tree contains all training examples with age at most 30, whereas the right branch contains all examples with age greater than 30. The goal is to identify a split criterion so that the level of "mixing" of the class variables in each branch of the tree is reduced as much as possible. Each node in the decision tree logically represents a subset of the data space defined by the combination of split criteria in the nodes above it. The decision tree is typically constructed as a hierarchical partitioning of the training examples, just as a top-down clustering algorithm partitions the data hierarchically. The main difference from clustering is that the partitioning criterion in the decision tree is *supervised* with the class label in the training instances. Some classical decision tree algorithms include *C4.5*, *ID3*, and *CART*. To illustrate the basic idea of decision tree construction, an illustrative example will be used.

In Table 10.1, a snapshot of a hypothetical charitable donation data set has been illustrated. The two feature variables represent the age and salary attributes. Both attributes are related to the donation propensity, which is also the class label. Specifically, the likelihood of an individual to donate is positively correlated with his or her age and salary. However, the best separation of the classes may be achieved only by combining the two attributes. The goal in the decision tree construction process is to perform a sequence of splits in top-down fashion to create nodes at the leaf level in which the donors and nondonors are separated well. One way of achieving this goal is depicted in Fig. 10.3a. The figure illustrates a hierarchical arrangement of the training examples in a treelike structure. The first-level split uses the age attribute, whereas the second-level split for both branches uses the salary attribute. Note that different splits at the same decision tree level need not be on the same attribute. Furthermore, the decision tree of Fig. 10.3a has two branches at each node, but this need not always be the case. In this case, the training examples in all leaf nodes belong to the same class, and, therefore, there is no point in growing the decision tree beyond the leaf nodes. The splits shown in Fig. 10.3a are referred to as *univariate* splits because they use a single attribute. To classify a test instance, a single relevant path in the tree is traversed in top-down fashion by using the split criteria to decide which branch to follow at each node of the tree. The dominant class label in the leaf node is reported as the relevant class. For example, a test instance with age less than 50 and salary less than 60,000 will traverse the leftmost path of the tree in Fig. 10.3a. Because the leaf node of this path contains only nondonor training examples, the test instance will also be classified as a nondonor.

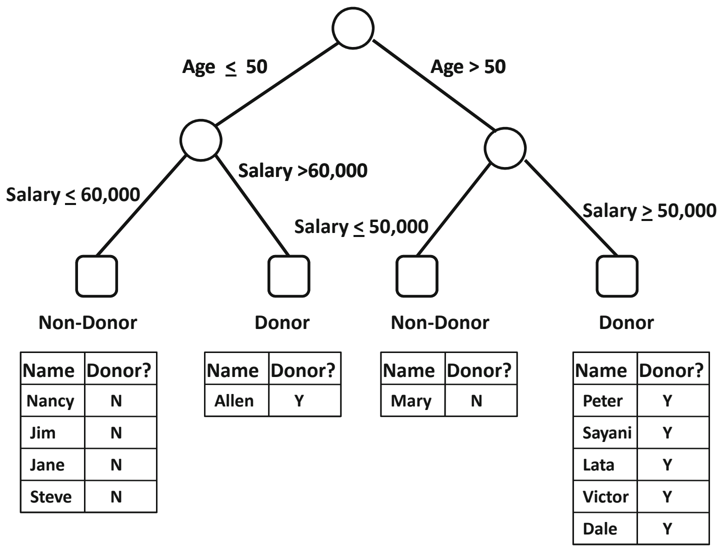
Multivariate splits use more than one attribute in the split criteria. An example is illustrated in Fig. 10.3b. In this particular case, a single split leads to full separation of the classes. This suggests that multivariate criteria are more powerful because they lead to shallower trees. For the same level of class separation in the training data, shallower trees are generally more desirable because the leaf nodes contain more examples and, therefore, are statistically less likely to overfit the noise in the training data.

A decision tree induction algorithm has two types of nodes, referred to as the *internal nodes* and *leaf nodes*. Each leaf node is labeled with the dominant class at that node. A special internal node is the root node that corresponds to the entire feature space. The generic decision tree induction algorithm starts with the full training data set at the root node and recursively partitions the data into lower level nodes based on the split criterion. Only nodes that contain a mixture of different classes need to be split further. Eventually, the decision tree algorithm stops the growth of the tree based on a *stopping criterion*. The simplest stopping criterion is one where all training examples in the leaf belong to the same class. One problem is that the construction of the decision tree to this level may lead to overfitting, in which the model fits the noisy nuances of the training data. Such a tree will not generalize to *unseen* test instances very well. To avoid the degradation in accuracy associated with overfitting, the classifier uses a *postpruning* mechanism for removing overfitting nodes. The generic decision tree training algorithm is illustrated in Fig. 10.4.

After a decision tree has been constructed, it is used for classification of unseen test instances with the use of top-down traversal from the root to a unique leaf. The split condition at each internal node is used to select the correct branch of the decision tree for further traversal. The label of the leaf node that is reached is reported for the test instance.

10.3.1 Split Criteria

The goal of the split criterion is to maximize the separation of the different classes among the children nodes. In the following, only univariate criteria will be discussed. Assume that



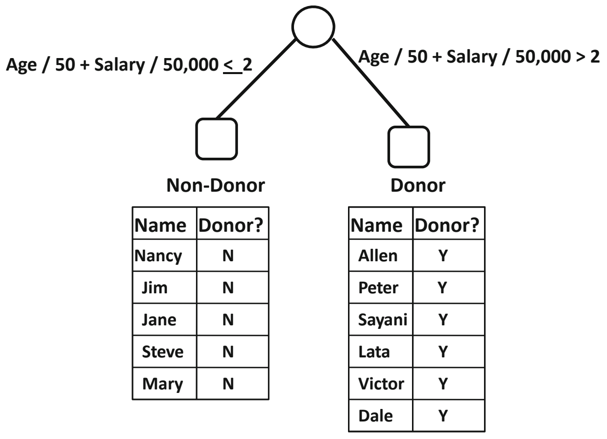
Name	Donor?
Nancy	N
Jim	N
Jane	N
Steve	N

Name	Donor?
Allen	Y

Name	Donor?
Mary	N

Name	Donor?
Peter	Y
Sayani	Y
Lata	Y
Victor	Y
Dale	Y

(a) Univariate split


Age / 50 + Salary / 50,000 ≤ 2
Age / 50 + Salary / 50,000 > 2
Non-Donor
Donor

Name	Donor?
Nancy	N
Jim	N
Jane	N
Steve	N
Mary	N

Name	Donor?
Allen	Y
Peter	Y
Sayani	Y
Lata	Y
Victor	Y
Dale	Y

(b) Multivariate split

Figure 10.3: Illustration of univariate and multivariate splits for decision tree construction

a quality criterion for evaluating a split is available. The design of the split criterion depends on the nature of the underlying attribute:

1. *Binary attribute:* Only one type of split is possible, and the tree is always binary. Each branch corresponds to one of the binary values.

2. *Categorical attribute:* If a categorical attribute has r different values, there are multiple ways to split it. One possibility is to use an r-way split, in which each branch of the split corresponds to a particular attribute value. The other possibility is to use a binary split by testing each of the $2^r - 1$ combinations (or groupings) of categorical attributes, and selecting the best one. This is obviously not a feasible option when the value of r is large. A simple approach that is sometimes used is to convert categorical

```
Algorithm GenericDecisionTree(Data Set: $\mathcal{D}$)
begin
  Create root node containing $\mathcal{D}$;
  repeat
    Select an eligible node in the tree;
    Split the selected node into two or more nodes
        based on a pre-defined split criterion;
  until no more eligible nodes for split;
  Prune overfitting nodes from tree;
  Label each leaf node with its dominant class;
end
```

Figure 10.4: Generic decision tree training algorithm

data to binary data with the use of the binarization approach discussed in Chap. 2. In this case, the approach for binary attributes may be used.

3. *Numeric attribute:* If the numeric attribute contains a small number r of ordered values (e.g., integers in a small range $[1, r]$), it is possible to create an r-way split for each distinct value. However, for continuous numeric attributes, the split is typically performed by using a binary condition, such as $x \leq a$, for attribute value x and constant a.

   Consider the case where a node contains m data points. Therefore, there are m possible split points for the attribute, and the corresponding values of a may be determined by sorting the data in the node along this attribute. One possibility is to test all the possible values of a for a split and select the best one. A faster alternative is to test only a smaller set of possibilities for a, based on equi-depth division of the range.

Many of the aforementioned methods requires the determination of the "best" split from a set of choices. Specifically, it is needed to choose from multiple attributes and from the various alternatives available for splitting each attribute. Therefore, quantifications of split quality are required. These quantifications are based on the same principles as the feature selection criteria discussed in Sect. 10.2.

1. *Error rate:* Let p be the fraction of the instances in a set of data points S belonging to the dominant class. Then, the error rate is simply $1-p$. For an r-way split of set S into sets $S_1 \dots S_r$, the overall error rate of the split may be quantified as the weighted average of the error rates of the individual sets S_i, where the weight of S_i is $|S_i|$. The split with the lowest error rate is selected from the alternatives.

2. *Gini index:* The Gini index $G(S)$ for a set S of data points may be computed according to Eq. 10.1 on the class distribution $p_1 \dots p_k$ of the training data points in S.

$$G(S) = 1 - \sum_{j=1}^{k} p_j^2 \tag{10.8}$$

   The overall Gini index for an r-way split of set S into sets $S_1 \dots S_r$ may be quantified as the weighted average of the Gini index values $G(S_i)$ of each S_i, where the weight of S_i is $|S_i|$.

$$\text{Gini-Split}(S \Rightarrow S_1 \ldots S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} G(S_i) \tag{10.9}$$

The split with the lowest Gini index is selected from the alternatives. The *CART* algorithm uses the Gini index as the split criterion.

3. *Entropy:* The entropy measure is used in one of the earliest classification algorithms, referred to as *ID3*. The entropy $E(S)$ for a set S may be computed according to Eq. 10.3 on the class distribution $p_1 \ldots p_k$ of the training data points in the node.

$$E(S) = -\sum_{j=1}^{k} p_j \log_2(p_j) \tag{10.10}$$

As in the case of the Gini index, the overall entropy for an r-way split of set S into sets $S_1 \ldots S_r$ may be computed as the weighted average of the Gini index values $G(S_i)$ of each S_i, where the weight of S_i is $|S_i|$.

$$\text{Entropy-Split}(S \Rightarrow S_1 \ldots S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} E(S_i) \tag{10.11}$$

Lower values of the entropy are more desirable. The entropy measure is used by the *ID3* and *C4.5* algorithms.

The information gain is closely related to entropy, and is equal to the *reduction* in the entropy $E(S) - \text{Entropy-Split}(S \Rightarrow S_1 \ldots S_r)$ as a result of the split. Large values of the reduction are desirable. At a conceptual level, there is no difference between using either of the two for a split although a normalization for the degree of the split is possible in the case of information gain. Note that the entropy and information gain measures should be used only to compare two splits of the same degree because both measures are naturally biased in favor of splits with larger degree. For example, if a categorical attribute has many values, attributes with many values will be preferred. It has been shown by the *C4.5* algorithm that dividing the overall information gain with the normalization factor of $-\sum_{i=1}^{r} \frac{|S_i|}{|S|} \log_2(\frac{|S_i|}{|S|})$ helps in adjusting for the varying number of categorical values.

The aforementioned criteria are used to select the choice of the split attribute and the precise criterion on the attribute. For example, in the case of a numeric database, different split points are tested for each numeric attribute, and the best split is selected.

10.3.2 Stopping Criterion and Pruning

The stopping criterion for the growth of the decision tree is intimately related to the underlying pruning strategy. When the decision tree is grown to the very end until every leaf node contains only instances belonging to a particular class, the resulting decision tree exhibits 100 % accuracy on instances belonging to the training data. However, it often generalizes poorly to unseen test instances because the decision tree has now *overfit* even to the random characteristics in the training instances. Most of this noise is contributed by the lower level nodes, which contain a smaller number of data points. In general, simpler models (shallow decision trees) are preferable to more complex models (deep decision trees) if they produce the same error on the training data.

To reduce the level of overfitting, one possibility is to stop the growth of the tree early. Unfortunately, there is no way of knowing the correct point at which to stop the growth of the tree. Therefore, a natural strategy is to prune overfitting portions of the decision tree and convert internal nodes to leaf nodes. Many different criteria are available to decide whether a node should be pruned. One strategy is to explicitly penalize model complexity with the use of the *minimum description length principle (MDL)*. In this approach, the cost of a tree is defined by a weighted sum of its (training data) error and its complexity (e.g., the number of nodes). Information-theoretic principles are used to measure tree complexity. Therefore, the tree is constructed to optimize the cost rather than only the error. The main problem with this approach is that the cost function is itself a heuristic that does not work consistently well across different data sets. A simpler and more intuitive strategy is to a hold out a small fraction (say 20 %) of the training data and build the decision tree on the remaining data. The impact of pruning a node on the classification accuracy is tested on the holdout set. If the pruning improves the classification accuracy, then the node is pruned. Leaf nodes are iteratively pruned until it is no longer possible to improve the accuracy with pruning. Although such an approach reduces the amount of training data for building the tree, the impact of pruning generally outweighs the impact of training-data loss in the tree-building phase.

10.3.3 Practical Issues

Decision trees are simple to implement and highly interpretable. They can model arbitrarily complex decision boundaries, *given sufficient training data*. Even a univariate decision tree can model a complex decision boundary with piecewise approximations by building a sufficiently deep tree. The main problem is that the amount of training data required to properly approximate a complex boundary with a treelike model is very large, and it increases with data dimensionality. With limited training data, the resulting decision boundary is usually a rather coarse approximation of the true boundary. Overfitting is common in such cases. This problem is exacerbated by the sensitivity of the decision tree to the split criteria at the higher levels of the tree. A closely related family of classifiers, referred to as *rule-based classifiers*, is able to alleviate these effects by moving away from the strictly hierarchical structure of a decision tree.

10.4 Rule-Based Classifiers

Rule-based classifiers use a set of "if–then" rules $\mathcal{R} = \{R_1 \dots R_m\}$ to match *antecedents* to *consequents*. A rule is typically expressed in the following form:

IF *Condition* THEN *Conclusion*.

The condition on the left-hand side of the rule, also referred to as the antecedent, may contain a variety of logical operators, such as $<$, $\leq$, $>$, $=$, $\subseteq$, or $\in$, which are applied to the feature variables. The right-hand side of the rule is referred to as the consequent, and it contains the class variable. Therefore, a rule R_i is of the form $Q_i \Rightarrow c$ where Q_i is the antecedent, and c is the class variable. The "$\Rightarrow$" symbol denotes the "THEN" condition. The rules are generated from the training data during the training phase. The notation Q_i represents a precondition on the feature set. In some classifiers, such as association pattern classifiers, the precondition may correspond to a pattern in the feature space, though this may not always be the case. In general, the precondition may be any arbitrary condition on the feature variables. These rules are then used to classify a test instance. A rule is said to *cover* a training instance when the condition in its antecedent matches the training instance.

A decision tree may be viewed as a special case of a rule-based classifier, in which each path of the decision tree corresponds to a rule. For example, the decision tree in Fig. 10.3a corresponds to the following set of rules:

$$\begin{aligned}
&Age \leq 50 \text{ AND } Salary \leq 60,000 \Rightarrow \neg Donor\\
&Age \leq 50 \text{ AND } Salary > 60,000 \Rightarrow Donor\\
&Age > 50 \text{ AND } Salary \leq 50,000 \Rightarrow \neg Donor\\
&Age > 50 \text{ AND } Salary > 50,000 \Rightarrow Donor
\end{aligned}$$

Note that each of the four aforementioned rules corresponds to a path in the decision tree of Fig. 10.3a. The logical expression on the left is expressed in conjunctive form, with a set of "AND" logical operators. Each of the primitive conditions in the antecedent, (such as $Age \leq 50$) is referred to as a *conjunct*. The rule set from a training data set is not unique and depends on the specific algorithm at hand. For example, only two rules are generated from the decision tree in Fig. 10.3b.

$$\begin{aligned}
&Age/50 + Salary/50,000 \leq 2 \Rightarrow \neg Donor\\
&Age/50 + Salary/50,000 > 2 \Rightarrow Donor
\end{aligned}$$

As in decision trees, succinct rules, both in terms of the cardinality of the rule set and the number of conjuncts in each rule, are generally more desirable. This is because such rules are less likely to overfit the data, and will generalize well to unseen test instances. Note that the antecedents on the left-hand side always correspond to a rule *condition*. In many rule-based classifiers, such as association-pattern classifiers, the logical operators such as "$\subseteq$" are implicit and are omitted from the rule antecedent description. For example, consider the case where the age and salary are discretized into categorical attribute values.

$$Age\ [50:60],\quad Salary\ [50,000:60,000] \Rightarrow Donor$$

In such a case, the discretized attributes for age and salary will be represented as "items," and an association pattern-mining algorithm can discover the itemset on the left-hand side. The operator "$\subseteq$" is implicit in the rule antecedent. Associative classifiers are discussed in detail later in this section.

The training phase of a rule-based algorithm creates a set of rules. The classification phase for a test instance discovers all rules that are *triggered* by the test instance. A rule is said to be triggered by the test instance when the logical condition in the antecedent is satisfied by the test instance. In some cases, rules with conflicting consequent values are triggered by the test instance. In such cases, methods are required to resolve the conflicts in class label prediction. Rule sets may satisfy one or more of the following properties:

1. *Mutually exclusive rules:* Each rule covers a disjoint partition of the data. Therefore, at most one rule can be triggered by a test instance. The rules generated from a decision tree satisfy this property. However, if the extracted rules are subsequently modified to reduce overfitting (as in some classifiers such as *C4.5rules*), the resulting rules may no longer remain mutually exclusive.

2. *Exhaustive rules:* The entire data space is covered by at least one rule. Therefore, every test instance triggers at least one rule. The rules generated from a decision tree

also satisfy this property. It is usually easy to construct an exhaustive rule set by creating a single catch-all rule whose consequent contains the dominant class in the portion of the training data not covered by other rules.

It is relatively easy to perform the classification when a rule set satisfies both the aforementioned properties. The reason for this is that each test instance maps to exactly one rule, and there are no conflicts in class predictions by multiple rules. In cases where rule sets are not mutually exclusive, conflicts in the rules triggered by a test instance can be resolved in one of two ways:

1. *Rule ordering:* The rules are ordered by priority, which may be defined in a variety of ways. One possibility is to use a quality measure of the rule for ordering. Some popular classification algorithms, such as *C4.5rules* and *RIPPER*, use class-based ordering, where rules with a particular class are prioritized over the other. The resulting set of ordered rules is also referred to as a *decision list*. For an arbitrary test instance, the class label in the consequent of the top triggered rule is reported as the relevant one for the test instance. Any other triggered rule is ignored. If no rule is triggered then a default catch-all class is reported as the relevant one.

2. *Unordered rules:* No priority is imposed on the rule ordering. The dominant class label among *all* the triggered rules may be reported. Such an approach can be more robust because it is not sensitive to the choice of the *single* rule selected by a rule-ordering scheme. The training phase is generally more efficient because all rules can be extracted simultaneously with pattern-mining techniques without worrying about relative ordering. Ordered rule-mining algorithms generally have to integrate the rule ordering into the rule generation process with methods such as *sequential covering*, which are computationally expensive. On the other hand, the testing phase of an unordered approach can be more expensive because of the need to compare a test instance against all the rules.

How should the different rules be ordered for test instance classification? The first possibility is to order the rules on the basis of a quality criterion, such as the confidence of the rule, or a weighted measure of the support and confidence. However, this approach is rarely used. In most cases, the rules are ordered by class. In some rare class applications, it makes sense to order all rules belonging to the rare class first. Such an approach is used by *RIPPER*. In other classifiers, such as *C4.5rules*, various accuracy and information-theoretic measures are used to prioritize classes.

10.4.1 Rule Generation from Decision Trees

As discussed earlier in this section, rules can be extracted from the different paths in a decision tree. For example, *C4.5rules* extracts the rules from the *C4.5* decision tree. The sequence of split criteria on each path of the decision tree corresponds to the antecedent of a corresponding rule. Therefore, it would seem at first sight that rule ordering is not needed because the generated rules are exhaustive and mutually exclusive. However, the rule-extraction process is followed by a rule-pruning phase in which many conjuncts are pruned from the rules to reduce overfitting. Rules are processed one by one, and conjuncts are pruned from them in greedy fashion to improve the accuracy as much as possible on the covered examples in a separate holdout validation set. This approach is similar to decision tree pruning except that one is no longer restricted to pruning the conjuncts at the lower levels of the decision tree. Therefore, the pruning process is more flexible than that of a decision tree, because it is not restricted by an underlying tree structure. Duplicate rules may result from pruning of conjuncts. These rules are removed. The rule-pruning phase increases the coverage of the individual rules and, therefore, the mutually exclusive nature of the rules is lost. As a result, it again becomes necessary to order the rules.

In *C4.5rules*, all rules that belong to the class whose rule set has the smallest description length are prioritized over other rules. The total description length of a rule set is a weighted sum of the number of bits required to encode the size of the model (rule set) and the number of examples covered by the class-specific rule set in the training data, which belong to a different class. Typically, classes with a smaller number of training examples are favored by this approach. A second approach is to order the class first whose rule set has the least number of false-positive errors on a separate holdout set. A rule-based version of a decision tree generally allows the construction of a more flexible decision boundary with limited training data than the base tree from which the rules are generated. This is primarily because of the greater flexibility in the model which is no longer restrained by the straitjacket of an exhaustive and mutually exclusive rule set. As a result, the approach generalizes better to unseen test instances.

10.4.2 Sequential Covering Algorithms

Sequential covering methods are used frequently for creating ordered rule lists. Thus, in this case, the classification process uses the top triggered rule to classify unseen test instances. Examples of sequential covering algorithms include *AQ*, *CN2*, and *RIPPER*. The sequential covering approach iteratively applies the following two steps to grow the rules from the training data set $\mathcal{D}$ until a stopping criterion is met:

1. (*Learn-One-Rule*) Select a particular class label and determine the "best" rule from the current training instances $\mathcal{D}$ with this class label as the consequent. Add this rule to the bottom of the ordered rule list.

2. (*Prune training data*) Remove the training instances in $\mathcal{D}$ that are covered by the rule learned in the previous step. All training instances matching the *antecedent* of the rule must be removed, whether or not the class label of the training instance matches the consequent.

The aforementioned generic description applies to all sequential covering algorithms. The various sequential covering algorithms mainly differ in the details of how the rules are ordered with respect to each other.

1. *Class-based ordering:* In most sequential covering algorithms such as *RIPPER*, all rules corresponding to a particular class are generated and placed contiguously on the ordered list. Typically, rare classes are ordered first. Therefore, classes that are placed earlier on the list may be favored more than others. This can sometimes cause artificially lower accuracy for test instances belonging to the less favored class.

   When class-based ordering is used, the rules for a particular class are generated contiguously. The addition of rules for each class has a stopping criterion that is algorithm dependent. For example, *RIPPER* uses an MDL criterion that stops adding rules when further addition increases the description length of the model by at least a predefined number of units. Another simpler stopping criterion is when the error rate of the next generated rule on a separate validation set exceeds a predefined threshold. Finally, one might simply use a threshold on the number of uncovered training instances remaining for a class as the class-specific stopping criterion. When the number of uncovered training instances remaining for a class falls below a threshold, rules for that class consequent are no longer grown. At this point, rules corresponding to the next class are grown. For a k-class problem, this approach is repeated $(k-1)$ times. Rules for the kth class are not grown. The least prioritized rule is a single catch-all rule with its consequent as the kth class. When the test instance does not fire rules belonging to the other classes, this class is assumed as the relevant label.

2. *Quality-based ordering:* In some covering algorithms, class-based ordering is not used. A quality measure is used to select the next rule. For example, one might generate the rule with the highest confidence in the remaining training data. The catch-all rule corresponds to the dominant class among remaining test instances. Quality-based ordering is rarely used in practice because of the difficulty in interpreting a quality criterion which is defined only over the *remaining* test instances.

Because class-based ordering is more common, the Learn-One-Rule procedure will be described under this assumption.

10.4.2.1 Learn-One-Rule

The Learn-One-Rule procedure grows rules from the general to the specific, in much the same way a decision tree grows a tree hierarchically from general nodes to specific nodes. Note that a path in a decision tree is a rule in which the antecedent corresponds to the conjunction of the split criteria at the different nodes, and the consequent corresponds to the label of the leaf nodes. While a decision tree grows many different disjoint paths at one time, the Learn-One-Rule procedure grows a single "best" path. This is yet another example of the close relationship between decision trees and rule-based methods.

The idea of Learn-One-Rule is to successively add conjuncts to the left-hand side of the rule to grow a single decision *path* (rather than a decision tree) based on a quality criterion. The root of the tree corresponds to the rule $\{\} \Rightarrow c$. The class c represents the consequent of the rule being grown. In the simplest version of the procedure, a single path is grown at one time by successively adding conjuncts to the antecedent. In other words, conjuncts are added to increase the *quality* as much as possible. The simplest quality criterion is the accuracy of the rule. The problem with this criterion is that rules with high accuracy but very low coverage are generally not desirable because of overfitting. The precise choice of the quality criterion that regulates the trade-off between accuracy and coverage will be discussed in detail later. As in the case of a decision tree, various logical conditions (or split choices) must be tested to determine the best conjunct to be added. The process of enumeration of the various split choices is similar to a decision tree. The rule is grown until a particular stopping criterion is met. A natural stopping criterion is one where the quality of the rule does not improve by further growth.

One challenge with the use of this procedure is that if a mistake is made early on during tree growth, it will lead to suboptimal rules. One way of reducing the likelihood of suboptimal rules is to always maintain the m best paths during rule-growth rather than a single one. An example of rule growth with the use of a single decision path, for the donor example of Table 10.1, is illustrated in Fig. 10.5. In this case, the rule is grown for the donor class. The first conjunct added is $Age > 50$, and the second conjunct added is $Salary > 50,000$. Note the intuitive similarity between the decision tree of Figs. 10.3a and 10.5.

It remains to describe the quality criterion for the growth of the paths during the Learn-One-Rule procedure. On what basis is a particular path selected over the others? The

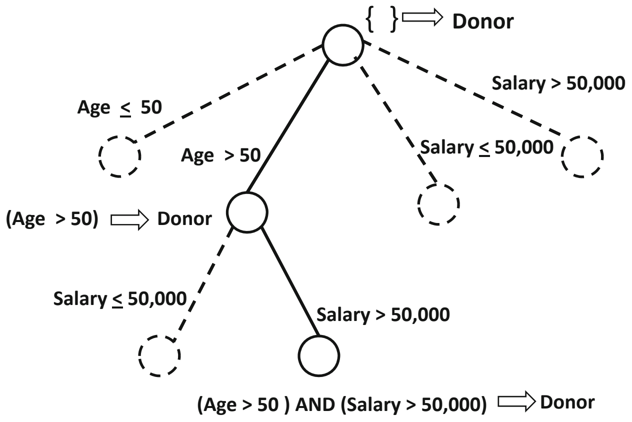


Figure 10.5: Rule growth is analogous to decision tree construction

similarity between rule growth and decision trees suggests the use of analogous measures such as the accuracy, entropy, or the Gini index, as used for split criteria in decision trees.

The criteria do need to be modified because a rule is relevant only to the training examples covered by the antecedent and the single class in the consequent, whereas decision tree splits are evaluated with respect to all training examples at a given node and all classes. Furthermore, decision tree split measures do not need to account for issues such as the coverage of the rule. One would like to determine rules with high coverage in order to avoid overfitting. For example, a rule that covers only a single training instance will always have 100 % accuracy, but it does not usually generalize well to unseen test instances. Therefore, one strategy is to combine the accuracy and coverage criteria into a single integrated measure.

The simplest combination approach is to use Laplacian smoothing with a parameter β that regulates the level of smoothing in a training data set with k classes:

$$\text{Laplace}(\beta) = \frac{n^+ + \beta}{n^+ + n^- + k\beta}. \tag{10.12}$$

The parameter $\beta > 0$ controls the level of smoothing, n^+ represents the number of correctly classified (positive) examples covered by the rule and n^- represents the number of incorrectly classified (negative) examples covered by the rule. Therefore, the total number of covered examples is $n^+ + n^-$. For cases where the absolute number of covered examples $n^+ + n^-$ is very small, Laplacian smoothing penalizes the accuracy to account for the unreliability of low coverage. Therefore, the measure favors greater coverage.

A second possibility is the *likelihood ratio statistic*. Let n_j be the observed number of training data points covered by the rule that belong to class j, and let n_j^e be the expected number of covered examples that would belong to class j, if the class distribution of the covered examples is the same as the full training data. In other words, if $p_1 \ldots p_k$ be the fraction of examples belonging to each class in the full training data, then we have:

$$n_i^e = p_i \sum_{i=1}^{k} n_i. \tag{10.13}$$

Then, for a k-class problem, the likelihood ratio statistic R may be computed as follows:

$$R = 2\sum_{j=1}^{k} n_j \log(n_j/n_j^e). \tag{10.14}$$

When the distribution of classes in the covered examples is significantly different than that in the original training data, the value of R increases. Therefore, the statistic tends to favor covered examples whose distributions are very different from the original training data. Furthermore, the presence of raw frequencies $n_1 \ldots n_k$ as multiplicative factors of the individual terms in the right-hand side of Eq. 10.14 ensures that larger rule coverage is rewarded. This measure is used by the *CN2* algorithm.

Another criterion is *FOIL's information gain*. The term "FOIL" stands for *first order inductive learner*. Consider the case where a rule covers n_1^+ positive examples and n_1^- negative examples, where positive examples are defined as training examples matching the class in the consequent. Consider the case where the addition of a conjunct to the antecedent changes the number of positive examples and negative examples to n_2^+ and n_2^-, respectively. Then, FOIL's information gain FG is defined as follows:

$$FG = n_2^+ \left(\log_2 \frac{n_2^+}{n_2^+ + n_2^-} - \log_2 \frac{n_1^+}{n_1^+ + n_1^-} \right). \tag{10.15}$$

This measure tends to select rules with high coverage because n_2^+ is a multiplicative factor in FG. At the same time, the information gain increases with higher accuracy because of the term inside the parentheses. This particular measure is used by the *RIPPER* algorithm.

As in the case of decision trees, it is possible to grow the rules until 100 % accuracy is achieved on the training data, or when the added conjunct does not improve the accuracy of the rule. Another criterion used by *RIPPER* is that the minimum description length of the rule must not increase by more than a certain threshold because of the addition of a conjunct. The description length of a rule is defined by a weighted function of the size of the conjuncts and the misclassified examples.

10.4.3 Rule Pruning

Rule-pruning is relevant not only for rules generated by the Learn-One-Rule method, but also for methods such as *C4.5rules* that extract the rules from a decision tree. Irrespective of the approach used to extract the rules, overfitting may result from the presence of too many conjuncts. As in decision tree pruning, the MDL principle can be used for pruning. For example, for each conjunct in the rule, one can add a penalty term δ to the quality criterion in the rule-growth phase. This will result in a *pessimistic error rate*. Rules with many conjuncts will therefore have larger aggregate penalties to account for their greater model complexity. A simpler approach for computing pessimistic error rates is to use a separate holdout validation set that is used for computing the error rate (without a penalty) but is not used by Learn-One-Rule during rule generation.

The conjuncts successively added during rule growth (in sequential covering) are then tested for pruning in reverse order. If pruning reduces the pessimistic error rate on the training examples covered by the rule, then the generalized rule is used. While some algorithms such as *RIPPER* test the most recently added conjunct first for rule pruning, it is not a strict requirement to do so. It is possible to test the conjuncts for removal in any order, or in greedy fashion, to reduce the pessimistic error rate as much as possible. Rule pruning may result in some of the rules becoming identical. Duplicate rules are removed from the rule set before classification.

10.4.4 Associative Classifiers

Associative classifiers are a popular strategy because they rely on association pattern mining, for which many efficient algorithmic alternatives exist. The reader is referred to Chap. 4 for algorithms on association pattern mining. The discussion below assumes binary attributes, though any data type can be converted to binary attributes with the process of discretization and binarization, as discussed in Chap. 2. Furthermore, unlike sequential covering algorithms in which rules are always ordered, the rules created by associative classifiers may be either ordered or unordered, depending upon application-specific criteria. The main characteristic of class-based association rules is that they are mined in the same way as regular association rules, except that they have a single class variable in the consequent. The basic strategy for an associative classifier is as follows:

1. Mine all class-based association rules at a given level of minimum support and confidence.

2. For a given test instance, use the mined rules for classification.

A variety of choices exist for the implementation of both steps. A naive way of implementing the first step would be to mine all association rules and then filter out only the rules in which the consequent corresponds to an individual class. However, such an approach is rather wasteful because it generates many rules with nonclass consequents. Furthermore, there is significant redundancy in the rule set because many rules that have 100 % confidence are special cases of other rules with 100 % confidence. Therefore, pruning methods are required during the rule-generation process.

The *classification based on associations (CBA)* approach uses a modification of the *Apriori* method to generate associations that satisfy the corresponding constraints. The first step is to generate *1-rule-items*. These are newly created items corresponding to combinations of items and class attributes. These rule items are then extended using traditional *Apriori*-style processing. Another modification is that, when patterns are generated corresponding to rules with 100 % confidence, those rules are not extended in order to retain greater generality in the rule set. This broader approach can be used in conjunction with almost any tree enumeration algorithm. The bibliographic notes contain pointers to several recent algorithms that use other frequent pattern mining methods for rule generation.

The second step of associative classification uses the generated rule set to make predictions for unseen test instances. Both ordered or unordered strategies may be used. The ordered strategy prioritizes the rules on the basis of the support (analogous to coverage), and the confidence (analogous to accuracy). A variety of heuristics may be used to create an integrated measure for ordering, such as using a weighted combination of support and confidence. The reader is referred to Chap. 17 for discussion of a representative rule-based classifier, *XRules*, which uses different types of measures. After the rules have been ordered, the top m matching rules to the test instance are determined. The dominant class label from the matching rules is reported as the relevant one for the test instance. A second strategy does not order the rules but determines the dominant class label from all the triggered rules. Other heuristic strategies may weight the rules differently, depending on their support and confidence, for the prediction process. Furthermore, many variations of associative classifiers do not use the support or confidence for mining the rules, but directly use class-based discriminative methods for pattern mining. The bibliographic notes contain pointers to these methods.

10.5 Probabilistic Classifiers

Probabilistic classifiers construct a model that quantifies the relationship between the feature variables and the target (class) variable as a probability. There are many ways in which such a modeling can be performed. Two of the most popular models are as follows:

1. *Bayes classifier:* The Bayes rule is used to model the probability of each value of the target variable for a given set of feature variables. Similar to mixture modeling in clustering (cf. Sect. 6.5 in Chap. 6), it is assumed that the data points within a class are generated from a specific probability distribution such as the Bernoulli distribution or the multinomial distribution. A *naive Bayes assumption* of class-conditioned feature independence is often (but not always) used to simplify the modeling.

2. *Logistic regression:* The target variable is assumed to be drawn from a Bernoulli distribution whose mean is defined by a parameterized logit function on the feature variables. Thus, the probability distribution of the *class* variable is a parameterized function of the feature variables. This is in contrast to the Bayes model that assumes a specific generative model of the *feature* distribution of each class.

The first type of classifier is referred to as a *generative classifier*, whereas the second is referred to as a *discriminative classifier*. In the following, both classifiers will be studied in detail.

10.5.1 Naive Bayes Classifier

The Bayes classifier is based on the Bayes theorem for conditional probabilities. This theorem quantifies the conditional probability of a random variable (class variable), given known observations about the value of another set of random variables (feature variables). The Bayes theorem is used widely in probability and statistics. To understand the Bayes theorem, consider the following example, based on Table 10.1:

Example 10.5.1 *A charitable organization solicits donations from individuals in the population of which* $6/11$ *have age greater than 50. The company has a success rate of* $6/11$ *in soliciting donations, and among the individuals who donate, the probability that the age is greater than 50 is* $5/6$. *Given an individual with age greater than 50, what is the probability that he or she will donate?*

Consider the case where the event E corresponds to $(Age > 50)$, and event D corresponds to an individual being a donor. The goal is to determine the *posterior* probability $P(D|E)$. This quantity is referred to as the "posterior" probability because it is conditioned on the observation of the event E that the individual has age greater than 50. The "prior" probability $P(D)$, before observing the age, is $6/11$. Clearly, knowledge of an individual's age influences posterior probabilities because of the obvious correlations between age and donor behavior.

Bayes theorem is useful for estimating $P(D|E)$ when it is hard to estimate $P(D|E)$ directly from the training data, but other conditional and prior probabilities such as $P(E|D)$, $P(D)$, and $P(E)$ can be estimated more easily. Specifically, Bayes theorem states the following:

$$P(D|E) = \frac{P(E|D)P(D)}{P(E)}. \tag{10.16}$$

Each of the expressions on the right-hand side is already known. The value of $P(E)$ is 6/11, and the value of $P(E|D)$ is 5/6. Furthermore, the prior probability $P(D)$ before knowing the age is 6/11. Consequently, the posterior probability may be estimated as follows:

$$P(D|E) = \frac{(5/6)(6/11)}{6/11} = 5/6. \tag{10.17}$$

Therefore, if we had 1-dimensional training data containing only the *Age*, along with the class variable, the probabilities could be estimated using this approach. Table 10.1 contains an example with training instances satisfying the aforementioned conditions. It is also easy to verify from Table 10.1 that the fraction of individuals above age 50 who are donors is 5/6, which is in agreement with Bayes theorem. In this particular case, the Bayes theorem is not really essential because the classes can be predicted directly from a *single* attribute of the training data. A question arises, as to why the indirect route of using the Bayes theorem is useful, if the posterior probability $P(D|E)$ could be estimated directly from the training data (Table 10.1) in the first place. The reason is that the conditional event E usually corresponds to a *combination of* constraints on d different feature variables, rather than a single one. This makes the direct estimation of $P(D|E)$ much more difficult. For example, the probability $P(Donor|Age > 50, Salary > 50{,}000)$ is harder to robustly estimate from the training data because there are fewer instances in Table 10.1 that satisfy *both* the conditions on age and salary. This problem increases with increasing dimensionality. In general, for a d-dimensional test instance, with d conditions, it may be the case that not even a single tuple in the training data satisfies all these conditions. Bayes rule provides a way of expressing $P(Donor|Age > 50, Salary > 50{,}000)$ in terms of $P(Age > 50, Salary > 50{,}000|Donor)$. The latter is much easier to estimate with the use of a product-wise approximation known as the *naive Bayes approximation*, whereas the former is not.

For ease in discussion, it will be assumed that all feature variables are categorical. The numeric case is discussed later. Let C be the random variable representing the class variable of an unseen test instance with d-dimensional feature values $\overline{X} = (a_1 \dots a_d)$. The goal is to estimate $P(C = c|\overline{X} = (a_1 \dots a_d))$. Let the random variables for the individual dimensions of $\overline{X}$ be denoted by $\overline{X} = (x_1 \dots x_d)$. Then, it is desired to estimate the conditional probability $P(C = c|x_1 = a_1, \dots x_d = a_d)$. This is difficult to estimate directly from the training data because the training data may not contain even a single record with attribute values $(a_1 \dots a_d)$. Then, by using Bayes theorem, the following equivalence can be inferred:

$$P(C = c|x_1 = a_1, \dots x_d = a_d) = \frac{P(C = c)P(x_1 = a_1, \dots x_d = a_d|C = c)}{P(x_1 = a_1, \dots x_d = a_d)} \tag{10.18}$$

$$\propto P(C = c)P(x_1 = a_1, \dots x_d = a_d|C = c). \tag{10.19}$$

The second relationship above is based on the fact that the term $P(x_1 = a_1, \dots x_d = a_d)$ in the denominator of the first relationship is independent of the class. Therefore, it suffices to only compute the numerator to determine the class with the maximum conditional probability. The value of $P(C = c)$ is the prior probability of the class identifier c and can be *estimated* as the fraction of the training data points belonging to class c. The key usefulness of the Bayes rule is that the terms on the right-hand side can now be effectively approximated from the training data with the use of a naive Bayes approximation. The naive Bayes approximation assumes that the values on the different attributes $x_1 \dots x_d$ are independent of one another conditional on the class. When two random events A and B are independent of one another conditional on a third event F, it follows that $P(A \cap B|F) = P(A|F)P(B|F)$. In the case of the naive Bayes approximation, it is assumed that the feature values are independent of one another conditional on a fixed value of the class variable. This implies the following for the conditional term on the right-hand side of Eq. 10.19.

$$P(x_1 = a_1, \ldots x_d = a_d | C = c) = \prod_{j=1}^{d} P(x_j = a_j | C = c) \tag{10.20}$$

Therefore, by substituting Eq. 10.20 in Eq. 10.19, the Bayes probability can be estimated within a constant of proportionality as follows:

$$P(C = c | x_1 = a_1, \ldots x_d = a_d) \propto P(C = c) \prod_{j=1}^{d} P(x_j = a_j | C = c). \tag{10.21}$$

Note that each term $P(x_j = a_j | C = c)$ is much easier to estimate from the training data than $P(x_1 = a_1, \ldots x_d = a_d | C = c)$ because enough training examples will exist in the former case to provide a robust estimate. Specifically, the *maximum likelihood estimate* for the value of $P(x_j = a_j | C = c)$ is the fraction of training examples taking on value a_j, conditional on the fact, that they belong to class c. In other words, if $q(a_j, c)$ is the number of training examples corresponding to feature variable $x_j = a_j$ and class c, and $r(c)$ is the number of training examples belonging to class c, then the estimation is performed as follows:

$$P(x_j = a_j | C = c) = \frac{q(a_j, c)}{r(c)}. \tag{10.22}$$

In some cases, enough training examples may still not be available to estimate these values robustly. For example, consider a rare class c with a *single training example* satisfying $r(c) = 1$, and $q(a_j, c) = 0$. In such a case, the conditional probability is estimated to 0. Because of the productwise form of the Bayes expression, the entire probability will be estimated to 0. Clearly, the use of a small number of training examples belonging to the rare class cannot provide robust estimates. To avoid this kind of overfitting, Laplacian smoothing is used. A small value of α is added to the numerator, and a value of $\alpha \cdot m_j$ is added to the denominator, where m_j is the number of distinct values of the jth attribute:

$$P(x_j = a_j | C = c) = \frac{q(a_j, c) + \alpha}{r(c) + \alpha \cdot m_j}. \tag{10.23}$$

Here, α is the Laplacian smoothing parameter. For the case where $r(c) = 0$, this has the effect of estimating the probability to an unbiased value of $1/m_j$ for all m_j distinct attribute values. This is a reasonable estimate in the absence of any training data about class c. Thus, the training phase only requires the estimation of these conditional probabilities $P(x_j = a_j | C = c)$ of each class–attribute–value combination, and the estimation of the prior probabilities $P(C = c)$ of each class.

This model is referred to as the *binary* or *Bernoulli* model for Bayes classification when it is applied to categorical data with only two outcomes of each feature attribute. For example, in text data, the two outcomes could correspond to the presence or absence of a word. In cases where more than two outcomes are possible for a feature variable, the model is referred to as the *generalized* Bernoulli model. The implicit generative assumption of this model is similar to that of mixture modeling algorithms in clustering (cf. Sect. 6.5 of Chap. 6). The features within each class (mixture component) are independently generated from a distribution whose probabilities are the productwise approximations of Bernoulli distributions. The estimation of model parameters in the training phase is analogous to the M-step in expectation–maximization (EM) clustering algorithms. Note that, unlike EM clustering algorithms, the labels on only the *training* data are used to compute the maximum likelihood estimates of parameters in the training phase. Furthermore, the E-step (or the iterative approach) is not required because the (deterministic) assignment "probabilities" of labeled data are already known. In Sect. 13.5.2.1 of Chap. 13, a more sophisticated model, referred to as the *multinomial model*, will be discussed. This model can address sparse frequencies associated with attributes, as in text data. In general, the Bayes model can assume any parametric form of the conditional feature distribution $P(x_1 = a_1, \dots x_d = a_d|C = c)$ of each class (mixture component), such as a Bernoulli model, a multinomial model, or even a Gaussian model for numeric data. The parameters of the distribution of each class are estimated in a data-driven manner. The approach discussed in this section, therefore, represents only a single instantiation from a wider array of possibilities.

The aforementioned description is based on categorical data. It can also be generalized to numeric data sets by using the process of discretization. Each discretized range becomes one of the possible categorical values of an attribute. Such an approach can, however, be sensitive to the granularity of the discretization. A second approach is to assume a specific form of the probability distribution of each mixture component (class), such as a Gaussian distribution. The mean and variance parameters of the Gaussian distribution of each class are estimated in a data-driven manner, just as the class conditioned feature probabilities are estimated in the Bernoulli model. Specifically, the mean and variance of each Gaussian can be estimated directly as the mean and variance of the training data for the corresponding class. This is similar to the M-step in EM clustering algorithms with Gaussian mixtures. The conditional class probabilities in Eq. 10.21 for a test instance are replaced with the class-specific Gaussian densities of the test instance.

10.5.1.1 The Ranking Model for Classification

The aforementioned algorithms predict the labels of *individual* test instances. In some scenarios, a *set* of test instances is provided to the learner, and it is desired to *rank* these test instances by their propensity to belong to a particularly important class c. This is a common scenario in rare-class learning, which will be discussed in Sect. 11.3 of Chap. 11.

As discussed in Eq. 10.21, the probability of a test instance $(a_1 \dots a_d)$ belonging to a particular class can be estimated within a constant of proportionality as follows:

$$P(C = c|x_1 = a_1, \dots x_d = a_d) \propto P(C = c) \prod_{j=1}^{d} P(x_j = a_j|C = c). \tag{10.24}$$

The constant of proportionality is irrelevant while comparing the scores across different *classes* but is not irrelevant while comparing the scores across different *test instances*. This is because the constant of proportionality is the inverse of the generative probability of the specific test instance. An easy way to estimate the proportionality constant is to use normalization so that the sum of probabilities across different classes is 1. Therefore, if the class label c is assumed to be an integer drawn from the range $\{1 \dots k\}$ for a k-class problem, then the Bayes probability can be estimated as follows:

$$P(C = c|x_1 = a_1, \dots x_d = a_d) = \frac{P(C = c) \prod_{j=1}^{d} P(x_j = a_j|C = c)}{\sum_{c=1}^{k} P(C = c) \prod_{j=1}^{d} P(x_j = a_j|C = c)}. \tag{10.25}$$

These normalized values can then be used to rank different test instances. It should be pointed out that most classification algorithms return a numerical score for each class, and therefore an analogous normalization can be performed for virtually any classification algorithm. However, in the Bayes method, it is more natural to intuitively interpret the normalized values as probabilities.

10.5.1.2 Discussion of the Naive Assumption

The Bayes model is referred to as "naive" because of the assumption of conditional independence. This assumption is obviously not true in practice because the features in real data sets are almost always correlated even when they are conditioned on a specific class. Nevertheless, in spite of this approximation, the naive Bayes classifier seems to perform quite well in practice in many domains. Although it is possible to implement the Bayes model using more general multivariate estimation methods, such methods can be computationally more expensive. Furthermore, the estimation of multivariate probabilities becomes inaccurate with increasing dimensionality, especially with limited training data. Therefore, significant practical accuracy is often not gained with the use of theoretically more accurate assumptions. The bibliographic notes contain pointers to theoretical results on the effectiveness of the naive assumption.

10.5.2 Logistic Regression

While the Bayes classifier assumes a specific form of the feature probability distribution for each class, logistic regression directly models the class-membership probabilities in terms of the feature variables with a discriminative function. Thus, the nature of the modeling assumption is different in the two cases. Both are, however, probabilistic classifiers because they use a specific modeling assumption to map the feature variables to a class-membership probability. In both cases, the parameters of the underlying probabilistic model need to be estimated in a data-driven manner.

In the simplest form of logistic regression, it is assumed that the class variable is binary, and is drawn from $\{-1, +1\}$, although it is also possible to model nonbinary class variables. Let $\overline{\Theta} = (\theta_0, \theta_1 \dots \theta_d)$ be a vector of $d+1$ different parameters. The ith parameter θ_i is a coefficient related to the ith dimension in the underlying data, and θ_0 is an offset parameter. Then, for a record $\overline{X} = (x_1 \dots x_d)$, the probability that the class variable C takes on the values of $+1$ or -1, is modeled with the use of a logistic function.

$$P(C = +1|\overline{X}) = \frac{1}{1 + e^{-(\theta_0 + \sum_{i=1}^{d} \theta_i x_i)}} \tag{10.26}$$

$$P(C = -1|\overline{X}) = \frac{1}{1 + e^{(\theta_0 + \sum_{i=1}^{d} \theta_i x_i)}} \tag{10.27}$$

It is easy to verify that the sum of the two aforementioned probability values is 1. Logistic regression can be viewed as either a probabilistic classifier or a *linear classifier*. In linear classifiers, such as Fisher's discriminant, a linear hyperplane is used to separate the two classes. Other linear classifiers such as SVMs and neural networks will be discussed in Sects. 10.6 and 10.7 of this chapter. In logistic regression, the parameters $\overline{\Theta} = (\theta_0 \dots \theta_d)$ can be viewed as the coefficients of a separating hyperplane $\theta_0 + \sum_{i=1}^{d} \theta_i x_i = 0$ between the two classes. The term θ_i is the linear coefficient of dimension i, and the term θ_0 is the constant term. The value of $\theta_0 + \sum_{i=1}^{d} \theta_i x_i$ will be either positive or negative, depending on the side of the separating hyperplane on which $\overline{X}$ is located. A positive value is predictive of the class $+1$, whereas a negative value is predictive of the class -1. In many other linear classifiers, the sign of this expression yields the class label of $\overline{X}$ from $\{-1, +1\}$. Logistic

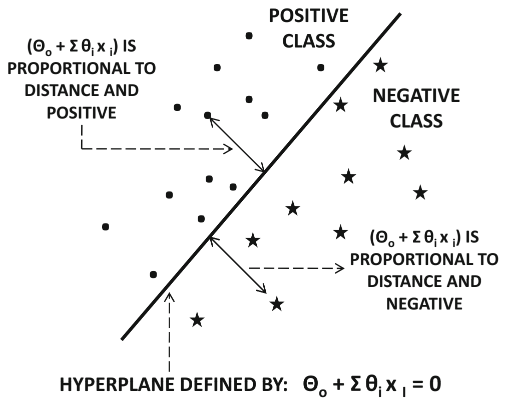

POSITIVE CLASS
($\Theta_o + \Sigma \theta_i x_i$) IS PROPORTIONAL TO DISTANCE AND POSITIVE
NEGATIVE CLASS
($\Theta_o + \Sigma \theta_i x_i$) IS PROPORTIONAL TO DISTANCE AND NEGATIVE
HYPERPLANE DEFINED BY: $\Theta_o + \Sigma \theta_i x_i = 0$

Figure 10.6: Illustration of logistic regression in terms of linear separators

regression achieves the same result in the form of probabilities defined by the aforementioned discriminative function.

The term $\theta_0 + \sum_{i=1}^{d} \theta_i x_i$, within the exponent of the logistic function is proportional to the distance of the data point from the separating hyperplane. When the data point lies exactly on this hyperplane, both classes are assigned the probability of 0.5 according to the logistic function. Positive values of the distance will assign probability values greater than 0.5 to the positive class. Negative values of the distance will assign (symmetrically equal) probability values greater than 0.5 to the negative class. This scenario is illustrated in Fig. 10.6. Therefore, the logistic function neatly exponentiates the distances, shown in Fig. 10.6, to convert them to intuitively interpretable probabilities in $(0, 1)$. The setup of logistic regression is similar to classical least-squares linear regression, with the difference that the logit function is used to estimate probabilities of class membership instead of constructing a squared error objective. Consequently, instead of the least-squares optimization in linear regression, a maximum likelihood optimization model is used for logistic regression.

10.5.2.1 Training a Logistic Regression Classifier

The maximum likelihood approach is used to estimate the best fitting parameters of the logistic regression model. Let $\mathcal{D}_+$ and $\mathcal{D}_-$ be the segments of the training data belonging to the positive and negative classes, respectively. Let the kth data point be denoted by $\overline{X_k} = (x_k^1 \ldots x_k^d)$. Then, the likelihood function $\mathcal{L}(\overline{\Theta})$ for the entire data set is defined as follows:

$$\mathcal{L}(\overline{\Theta}) = \prod_{\overline{X_k} \in \mathcal{D}_+} \frac{1}{1 + e^{-(\theta_0 + \sum_{i=1}^{d} \theta_i x_k^i)}} \prod_{\overline{X_k} \in \mathcal{D}_-} \frac{1}{1 + e^{(\theta_0 + \sum_{i=1}^{d} \theta_i x_k^i)}}. \tag{10.28}$$

This likelihood function is the product of the probabilities of all the training examples taking on their assigned labels according to the logistic model. The goal is to maximize this function to determine the optimal value of the parameter vector $\overline{\Theta}$. For numerical convenience, the log likelihood is used to yield the following:

$$\mathcal{LL}(\overline{\Theta}) = \log(\mathcal{L}(\overline{\Theta})) = -\sum_{\overline{X_k} \in \mathcal{D}_+} \log(1 + e^{-(\theta_0 + \sum_{i=1}^{d} \theta_i x_k^i)}) - \sum_{\overline{X_k} \in \mathcal{D}_-} \log(1 + e^{(\theta_0 + \sum_{i=1}^{d} \theta_i x_k^i)}). \tag{10.29}$$

There is no closed-form solution for optimizing the aforementioned expression with respect to the vector $\overline{\Theta}$. Therefore, a natural approach is to use a gradient ascent method to determine the optimal value of the parameter vector $\overline{\Theta}$ iteratively. The gradient vector is obtained by differentiating the log-likelihood function with respect to each of the parameters:

$$\nabla \mathcal{LL}(\overline{\Theta}) = \left(\frac{\partial \mathcal{LL}(\overline{\Theta}}{\partial \theta_0} \ldots \frac{\partial \mathcal{LL}(\overline{\Theta}}{\partial \theta_d} \right). \tag{10.30}$$

It is instructive to examine the ith component[4] of the aforementioned gradient, for $i > 0$. By computing the partial derivative of both sides of Eq. 10.29 with respect to θ_i, the following can be obtained:

$$\begin{aligned}
\frac{\partial \mathcal{LL}(\overline{\Theta})}{\partial \theta_i} &= \sum_{\overline{X_k} \in \mathcal{D}_+} \frac{x_k^i}{1 + e^{(\theta_0 + \sum_{i=1}^d \theta_i x_i)}} - \sum_{\overline{X_k} \in \mathcal{D}_-} \frac{x_k^i}{1 + e^{-(\theta_0 + \sum_{i=1}^d \theta_i x_i)}} && (10.31) \\
&= \sum_{\overline{X_k} \in \mathcal{D}_+} P(\overline{X_k} \in \mathcal{D}_-) x_k^i - \sum_{\overline{X_k} \in \mathcal{D}_-} P(\overline{X_k} \in \mathcal{D}_+) x_k^i && (10.32) \\
&= \sum_{\overline{X_k} \in \mathcal{D}_+} P(\text{Mistake on } \overline{X_k}) x_k^i - \sum_{\overline{X_k} \in \mathcal{D}_-} P(\text{Mistake on } \overline{X_k}) x_k^i. && (10.33)
\end{aligned}$$

It is interesting to note that the terms $P(\overline{X_k} \in \mathcal{D}_-)$ and $P(\overline{X_k} \in \mathcal{D}_+)$ represent the probability of an *incorrect prediction* of $\overline{X_k}$ in the positive and negative classes, respectively. Thus, the mistakes of the current model are used to identify the steepest ascent directions. This approach is generally true of many linear models, such as neural networks, which are also referred to as *mistake-driven methods*. In addition, the multiplicative factor x_k^i impacts the magnitude of the ith component of the gradient direction contributed by $\overline{X_k}$. Therefore, the update condition for θ_i is as follows:

$$\theta_i \leftarrow \theta_i + \alpha \left(\sum_{\overline{X_k} \in \mathcal{D}_+} P(\overline{X_k} \in \mathcal{D}_-) x_k^i - \sum_{\overline{X_k} \in \mathcal{D}_-} P(\overline{X_k} \in \mathcal{D}_+) x_k^i \right). \tag{10.34}$$

The value of α is the step size, which can be determined by using binary search to maximize the improvement in the objective function value. The aforementioned equation uses a batch ascent method, wherein all the training data points contribute to the gradient in a single update step. In practice, it is possible to cycle through the data points one by one for the update process. It can be shown that the likelihood function is concave. Therefore, a global optimum will be found by the gradient ascent method. A number of regularization methods are also used to reduce overfitting. A typical example of a regularization term, which is added to the log-likelihood function $\mathcal{LL}(\overline{\Theta})$ is $-\lambda \sum_{i=1}^d \theta_i^2/2$, where λ is the balancing parameter. The only difference to the gradient update is that the term $-\lambda\theta_i$ needs to be added to the ith gradient component for $i \geq 1$.

10.5.2.2 Relationship with Other Linear Models

Although the logistic regression method is a probabilistic method, it is also a special case of a broader class of *generalized linear models* (cf. Sect. 11.5.3 of Chap. 11). There are many ways of formulating a linear model. For example, instead of using a logistic function to set

[4]For the case where $i = 0$, the value of x_k^i is replaced by 1.

up a likelihood criterion, one might directly optimize the squared error of the prediction. In other words, if the class label for $\overline{X_k}$ is $y_k \in \{-1, +1\}$, one might simply attempt to optimize the squared error $\sum_{\overline{X_k} \in \mathcal{D}} (y_k - \text{sign}(\theta_0 + \sum_{i=1}^{d} \theta_i x_i^k))^2$ over all test instances. Here, the function "sign" evaluates to $+1$ or -1, depending on whether its argument is positive or negative. As will be evident in Sect. 10.7, such a model is (approximately) used by neural networks. Similarly, Fisher's linear discriminant, which was discussed at the beginning of this chapter, is also a linear least-squares model (cf. Sect. 11.5.1.1 of Chap. 11) but with a different coding of the class variable. In the next section, a linear model that uses the *maximum margin principle* to separate the two classes, will be discussed.

10.6 Support Vector Machines

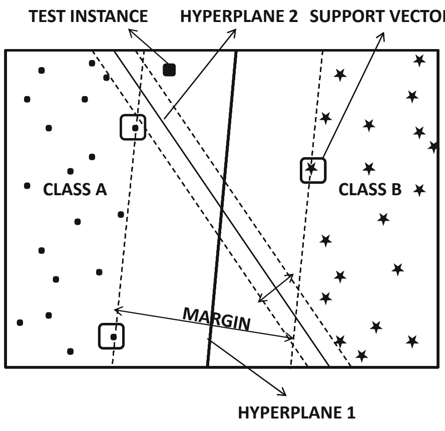
Support vector machines (SVMs) are naturally defined for binary classification of numeric data. The binary-class problem can be generalized to the multiclass case by using a variety of tricks discussed in Sect. 11.2 of Chap. 11. Categorical feature variables can also be addressed by transforming categorical attributes to binary data with the binarization approach discussed in Chap. 2.

It is assumed that the class labels are drawn from $\{-1, 1\}$. As with all linear models, SVMs use separating hyperplanes as the decision boundary between the two classes. In the case of SVMs, the optimization problem of determining these hyperplanes is set up with the notion of *margin*. Intuitively, a *maximum margin hyperplane* is one that cleanly separates the two classes, and for which a large region (or *margin*) exists on each side of the boundary with no training data points in it. To understand this concept, the very special case where the data is *linearly separable* will be discussed first. In linearly separable data, it is possible to construct a linear hyperplane which cleanly separates data points belonging to the two classes. Of course, this special case is relatively unusual because real data is rarely fully separable, and at least a few data points, such as mislabeled data points or outliers, will violate linear separability. Nevertheless, the linearly separable formulation is crucial in understanding the important principle of maximum margin. After discussing the linear separable case, the modifications to the formulation required to enable more general (and realistic) scenarios will be addressed.

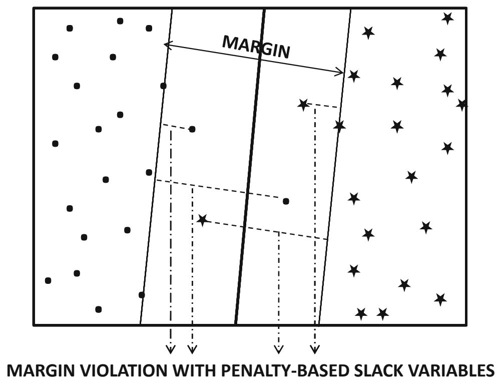
10.6.1 Support Vector Machines for Linearly Separable Data

This section will introduce the use of the maximum margin principle in linearly separable data. When the data is linearly separable, there are an infinite number of possible ways of constructing a linear separating hyperplane between the classes. Two examples of such hyperplanes are illustrated in Fig. 10.7a as hyperplane 1 and hyperplane 2. Which of these hyperplanes is better? To understand this, consider the test instance (marked by a square), which is very obviously much closer to class A than class B. The hyperplane 1 will correctly classify it to class A, whereas the hyperplane 2 will incorrectly classify it to class B.

The reason for the varying performance of the two classifiers is that the test instance is placed in a noisy and uncertain boundary region between the two classes, which is not easily *generalizable* from the available training data. In other words, there are few training data points in this uncertain region that are quite like the test instance. In such cases, a separating hyperplane like hyperplane 1, whose minimum perpendicular distance to training points from both classes is as large as possible, is the most robust one for correct classification. This distance can be quantified using the *margin* of the hyperplane.



(a) Hard separation



(b) Soft separation

Figure 10.7: Hard and soft SVMs

Consider a hyperplane that cleanly separates two linearly separable classes. The margin of the hyperplane is defined as the sum of its distances to the closest training points belonging to each of the two classes on the opposite side of the hyperplane. A further assumption is that the distance of the separating hyperplane to its closest training point of either class is the same. With respect to the separating hyperplane, it is possible to construct parallel hyperplanes that touch the training data of opposite classes on either side, and have no data point between them. The training data points on these hyperplanes are referred to as the *support vectors*, and the distance between the two hyperplanes is the *margin*. The separating hyperplane, or decision boundary, is precisely in the middle of these two hyperplanes in order to achieve the most accurate classification. The margins for hyperplane 1 and hyperplane 2 are illustrated in Fig. 10.7a by dashed lines. It is evident that the margin for hyperplane 1 is larger than that for hyperplane 2. Therefore, the former hyperplane provides better generalization power for unseen test instances in the "difficult" uncertain region separating the two classes where classification errors are most likely. This is also consistent with our earlier example-based observation about the more accurate classification with hyperplane 1.

How do we determine the maximum margin hyperplane? The way to do this is to set up a nonlinear programming optimization formulation that maximizes the margin by expressing it as a function of the coefficients of the separating hyperplane. The optimal coefficients can be determined by solving this optimization problem. Let the n data points in the training set $\mathcal{D}$ be denoted by $(\overline{X_1}, y_1) \ldots (\overline{X_n}, y_n)$, where $\overline{X_i}$ is a d-dimensional row vector corresponding to the ith data point, and $y_i \in \{-1, +1\}$ is the binary class variable of the ith data point. Then, the separating hyperplane is of the following form:

$$\overline{W} \cdot \overline{X} + b = 0. \tag{10.35}$$

Here, $\overline{W} = (w_1 \ldots w_d)$ is the d-dimensional row vector representing the normal direction to the hyperplane, and b is a scalar, also known as the *bias*. The vector $\overline{W}$ regulates the orientation of the hyperplane and the bias b regulates the distance of the hyperplane from the origin. The $(d + 1)$ coefficients corresponding to $\overline{W}$ and b need to be learned from the training data to maximize the margin of separation between the two classes. Because it is assumed that the classes are linearly separable, such a hyperplane can also be assumed to exist. All data points $\overline{X_i}$ with $y_i = +1$ will lie on one side of the hyperplane satisfying $\overline{W} \cdot \overline{X_i} + b \geq 0$. Similarly, all points with $y_i = -1$ will lie on the other side of the hyperplane satisfying $\overline{W} \cdot \overline{X_i} + b \leq 0$.

$$\overline{W} \cdot \overline{X_i} + b \geq 0 \quad \forall i : y_i = +1 \tag{10.36}$$
$$\overline{W} \cdot \overline{X_i} + b \leq 0 \quad \forall i : y_i = -1 \tag{10.37}$$

These constraints do not yet incorporate the margin requirements on the data points. A stronger set of constraints are defined using these margin requirements. It may be assumed that the separating hyperplane $\overline{W} \cdot \overline{X} + b = 0$ is located in the center of the two margin-defining hyperplanes. Therefore, the two symmetric hyperplanes touching the support vectors can be expressed by introducing another parameter c that regulates the distance between them.

$$\overline{W} \cdot \overline{X} + b = +c \tag{10.38}$$
$$\overline{W} \cdot \overline{X} + b = -c \tag{10.39}$$

It is possible to assume, without loss of generality, that the variables $\overline{W}$ and b are appropriately scaled, so that the value of c can be set to 1. Therefore, the two separating hyperplanes can be expressed in the following form:

$$\overline{W} \cdot \overline{X} + b = +1 \tag{10.40}$$
$$\overline{W} \cdot \overline{X} + b = -1. \tag{10.41}$$

These constraints are referred to as *margin constraints*. The two hyperplanes segment the data space into three regions. It is assumed that no training data points lie in the uncertain decision boundary region between these two hyperplanes, and all training data points for each class are mapped to one of the two remaining (extreme) regions. This can be expressed as pointwise constraints on the training data points as follows:

$$\overline{W} \cdot \overline{X_i} + b \geq +1 \quad \forall i : y_i = +1 \tag{10.42}$$

$$\overline{W} \cdot \overline{X_i} + b \leq -1 \quad \forall i : y_i = -1. \tag{10.43}$$

Note that the constraints for both the positive and negative classes can be written in the following succinct and algebraically convenient, but rather cryptic, form:

$$y_i(\overline{W} \cdot \overline{X_i} + b) \geq +1 \quad \forall i. \tag{10.44}$$

The distance between the two hyperplanes for the positive and negative instances is also referred to as the margin. As discussed earlier, the goal is to maximize this margin. What is the distance (or margin) between these two parallel hyperplanes? One can use linear algebra to show that the distance between two parallel hyperplanes is the normalized difference between their constant terms, where the normalization factor is the L_2-norm $||\overline{W}|| = \sqrt{\sum_{i=1}^{d} w_i^2}$ of the coefficients. Because the difference between the constant terms of the two aforementioned hyperplanes is 2, it follows that the distance between them is $2/||\overline{W}||$. This is the margin that needs to be maximized with respect to the aforementioned constraints. This form of the objective function is inconvenient because it incorporates a square root in the denominator of the objective function. However, maximizing $2/||\overline{W}||$ is the same as minimizing $||\overline{W}||^2/2$. This is a convex quadratic programming problem, because the quadratic objective function $||\overline{W}||^2/2$ needs to be minimized subject to a set of linear constraints (Eqs. 10.42–10.43) on the training points. Note that each training data point leads to a constraint, which tends to make the optimization problem rather large, and explains the high computational complexity of SVMs.

Such constrained nonlinear programming problems are solved using a method known as *Lagrangian relaxation*. The broad idea is to associate a nonnegative n-dimensional set of Lagrangian multipliers $\overline{\lambda} = (\lambda_1 \dots \lambda_n) \geq 0$ for the different constraints. The multiplier λ_i corresponds to the margin constraint of the ith training data point. The constraints are then relaxed, and the objective function is augmented by incorporating a Lagrangian penalty for constraint violation:

$$L_P = \frac{||\overline{W}||^2}{2} - \sum_{i=1}^{n} \lambda_i \left[y_i(\overline{W} \cdot \overline{X_i} + b) - 1 \right]. \tag{10.45}$$

For *fixed* nonnegative values of λ_i, margin constraint violations increase L_p. Therefore, the penalty term pushes the optimized values of $\overline{W}$ and b towards constraint nonviolation for minimization of L_P with respect to $\overline{W}$ and b. Values of $\overline{W}$ and b that satisfy the margin constraints will always result in a nonpositive penalty. Therefore, for any fixed nonnegative value of $\overline{\lambda}$, the minimum value of L_P will always be at most equal to that of the original optimal objective function value $||\overline{W^*}||^2/2$ because of the impact of the non-positive penalty term for any feasible $(\overline{W^*}, b^*)$.

Therefore, if L_P is minimized with respect to $\overline{W}$ and b for any particular $\overline{\lambda}$, and then maximized with respect to nonnegative Lagrangian multipliers $\overline{\lambda}$, the resulting *dual* solution L_D^* will be a lower bound on the optimal objective function $O^* = ||\overline{W^*}||^2/2$ of the SVM formulation. Mathematically, this *weak duality* condition can be expressed as follows:

$$O^* \geq L_D^* = \max_{\overline{\lambda} \geq 0} \min_{\overline{W}, b} L_P. \tag{10.46}$$

Optimization formulations such as SVM are special because the objective function is convex, and the constraints are linear. Such formulations satisfy a property known as *strong duality*. According to this property, the minimax relationship of Eq. 10.46 yields an optimal and feasible solution to the original problem (i.e., $O^* = L_D^*$) in which the Lagrangian penalty term has zero contribution. Such a solution $(\overline{W^*}, b^*, \overline{\lambda^*})$ is referred to as the *saddle point* of the Lagrangian formulation. Note that zero Lagrangian penalty is achieved by a feasible solution only when each training data point $\overline{X_i}$ satisfies $\lambda_i \left[y_i(\overline{W} \cdot \overline{X_i} + b) - 1 \right] = 0$. These conditions are equivalent to the *Kuhn–Tucker optimality conditions*, and they imply that data points $\overline{X_i}$ with $\lambda_i > 0$ are support vectors. The Lagrangian formulation is solved using the following steps:

1. The Lagrangian objective L_P can be expressed more conveniently as a pure maximization problem by eliminating the minimization part from the awkward minimax formulation. This is achieved by eliminating the minimization variables $\overline{W}$ and b with gradient-based optimization conditions on these variables. By setting the gradient of L_P with respect to $\overline{W}$ to 0, we obtain the following:

$$\nabla L_P = \nabla \frac{||\overline{W}||^2}{2} - \nabla \sum_{i=1}^{n} \lambda_i \left[y_i(\overline{W} \cdot \overline{X_i} + b) - 1 \right] = 0 \tag{10.47}$$

$$\overline{W} - \sum_{i=1}^{n} \lambda_i y_i \overline{X_i} = 0. \tag{10.48}$$

Therefore, one can now derive an expression for $\overline{W}$ in terms of the Lagrangian multipliers and the training data points:

$$\overline{W} = \sum_{i=1}^{n} \lambda_i y_i \overline{X_i}. \tag{10.49}$$

Furthermore, by setting the partial derivative of L_P with respect to b to 0, we obtain $\sum_{i=1}^{n} \lambda_i y_i = 0$.

2. The optimization condition $\sum_{i=1}^{n} \lambda_i y_i = 0$ can be used to eliminate the term $-b\sum_{i=1}^{n} \lambda_i y_i$ from L_P. The expression $\overline{W} = \sum_{i=1}^{n} \lambda_i y_i \overline{X_i}$ from Eq. 10.49 can then be substituted in L_P to create a dual problem L_D in terms of only the *maximization* variables $\overline{\lambda}$. Specifically, the maximization objective function L_D for the Lagrangian dual is as follows:

$$L_D = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \overline{X_i} \cdot \overline{X_j}. \tag{10.50}$$

The dual problem maximizes L_D subject to the constraints $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$. Note that L_D is expressed only in terms of λ_i, the class labels, and the pairwise dot products $\overline{X_i} \cdot \overline{X_j}$ between training data points. Therefore, solving for the Lagrangian multipliers requires knowledge of only the class variables and dot products between training instances but it does not require *direct* knowledge of the feature values $\overline{X_i}$. The dot products between training data points can be viewed as a kind of similarity between the points, which can easily be defined for data types beyond numeric domains. This observation is important for generalizing linear SVMs to nonlinear decision boundaries and arbitrary data types with the kernel trick.

3. The value of b can be derived from the constraints in the original SVM formulation, for which the Lagrangian multipliers λ_r are *strictly* positive. For these training points, the margin constraint $y_r(\overline{W} \cdot \overline{X_r} + b) = +1$ is satisfied exactly according to the Kuhn–Tucker conditions. The value of b can be derived from *any* such training point $(\overline{X_r}, y_r)$ as follows:

$$y_r \left[\overline{W} \cdot \overline{X_r} + b\right] = +1 \qquad \forall r : \lambda_r > 0 \tag{10.51}$$

$$y_r \left[\left(\sum_{i=1}^{n} \lambda_i y_i \overline{X_i} \cdot \overline{X_r}\right) + b \right] = +1 \qquad \forall r : \lambda_r > 0. \tag{10.52}$$

The second relationship is derived by substituting the expression for $\overline{W}$ in terms of the Lagrangian multipliers according to Eq. 10.49. Note that this relationship is expressed only in terms of Lagrangian multipliers, class labels, and dot products between training instances. The value of b can be solved from this equation. To reduce numerical error, the value of b may be averaged over all the support vectors with $\lambda_r > 0$.

4. For a test instance $\overline{Z}$, its class label $F(\overline{Z})$ is defined by the decision boundary obtained by substituting for $\overline{W}$ in terms of the Lagrangian multipliers (Eq. 10.49):

$$F(\overline{Z}) = \text{sign}\{\overline{W} \cdot \overline{Z} + b\} = \text{sign}\{(\sum_{i=1}^{n} \lambda_i y_i \overline{X_i} \cdot \overline{Z}) + b\}. \tag{10.53}$$

> It is interesting to note that $F(\overline{Z})$ can be fully expressed in terms of the dot product between training instances and test instances, class labels, Lagrangian multipliers, and bias b. Because the Lagrangian multipliers λ_i and b can also be expressed in terms of the dot products between training instances, it follows that the classification can be fully performed using knowledge of only the dot product between different instances (training and test), without knowing the exact feature values of either the training or the test instances.

The observations about dot products are crucial in generalizing SVM methods to nonlinear decision boundaries and arbitrary data types with the use of a technique known as the *kernel trick*. This technique simply substitutes dot products with kernel similarities (cf. Sect. 10.6.4).

It is noteworthy from the derivation of $\overline{W}$ (see Eq. 10.49) and the aforementioned derivation of b, that only training data points that are support vectors (with $\lambda_r > 0$) are used to define the solution $\overline{W}$ and b in SVM optimization. As discussed in Chap. 11, this observation is leveraged by scalable SVM classifiers, such as *SVMLight*. Such classifiers shrink the size of the problem by discarding irrelevant training data points that are easily identified to be far away from the separating hyperplanes.

10.6.1.1 Solving the Lagrangian Dual

The Lagrangian dual L_D may be optimized by using the gradient ascent technique in terms of the n-dimensional parameter vector $\overline{\lambda}$.

$$\frac{\partial L_D}{\partial \lambda_i} = 1 - y_i \sum_{j=1}^{n} y_j \lambda_j \overline{X_i} \cdot \overline{X_j} \tag{10.54}$$

Therefore, as in logistic regression, the corresponding gradient-based update equation is as follows:

$$(\lambda_1 \ldots \lambda_n) \Leftarrow (\lambda_1 \ldots \lambda_n) + \alpha \left(\frac{\partial L_D}{\partial \lambda_1} \ldots \frac{\partial L_D}{\partial \lambda_n} \right). \tag{10.55}$$

The step size α may be chosen to maximize the improvement in objective function. The initial solution can be chosen to be the vector of zeros, which is also a feasible solution for $\overline{\lambda}$.

One problem with this update is that the constraints $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$ may be violated after an update. Therefore, the gradient vector is projected along the hyperplane $\sum_{i=1}^{n} \lambda_i y_i = 0$ before the update to create a modified gradient vector. Note that the projection of the gradient ∇L_D along the normal to this hyperplane is simply $\overline{H} = (\overline{y} \cdot \nabla L_D)\,\overline{y}$, where $\overline{y}$ is the unit vector $\frac{1}{\sqrt{n}}(y_1 \ldots y_n)$. This component is subtracted from ∇L_D to create a modified gradient vector $\overline{G} = \nabla L_D - \overline{H}$. Because of the projection, updating along the modified gradient vector $\overline{G}$ will not violate the constraint $\sum_{i=1}^{n} \lambda_i y_i = 0$. In addition, any negative values of λ_i after an update are reset to 0.

Note that the constraint $\sum_{i=1}^{n} \lambda_i y_i = 0$ is derived by setting the gradient of L_P with respect to b to 0. In some alternative formulations of SVMs, the bias vector b can be included within $\overline{W}$ by adding a synthetic dimension to the data with a constant value of 1. In such cases, the gradient vector update is simplified to Eq. 10.55 because one no longer needs to worry about the constraint $\sum_{i=1}^{n} \lambda_i y_i = 0$. This alternative formulation of SVMs is discussed in Chap. 13.

10.6.2 Support Vector Machines with Soft Margin for Nonseparable Data

The previous section discussed the scenario where the data points of the two classes are linearly separable. However, perfect linear separability is a rather contrived scenario, and real data sets usually will not satisfy this property. An example of such a data set is illustrated in Fig. 10.7b, where no linear separator may be found. Many real data sets may, however, be approximately separable, where *most* of the data points lie on correct sides of well-chosen separating hyperplanes. In this case, the notion of margin becomes a softer one because training data points are allowed to violate the margin constraints *at the expense of a penalty*. The two margin hyperplanes separate out "most" of the training data points but not all of them. An example is illustrated in Fig. 10.7b.

The level of violation of each margin constraint by training data point $\overline{X_i}$ is denoted by a slack variable $\xi_i \geq 0$. Therefore, the new set of soft constraints on the separating hyperplanes may be expressed as follows:

$$\begin{aligned} \overline{W} \cdot \overline{X_i} + b &\geq +1 - \xi_i \quad \forall i : y_i = +1 \\ \overline{W} \cdot \overline{X_i} + b &\leq -1 + \xi_i \quad \forall i : y_i = -1 \\ \xi_i &\geq 0 \quad \forall i. \end{aligned}$$

These slack variables ξ_i may be interpreted as the distances of the training data points from the separating hyperplanes, as illustrated in Fig. 10.7b, when they lie on the "wrong" side of the separating hyperplanes. The values of the slack variables are 0 when they lie on the correct side of the separating hyperplanes. It is not desirable for too many training data points to have positive values of ξ_i, and therefore such violations are penalized by $C \cdot \xi_i^r$, where C and r are user-defined parameters regulating the level of softness in the model. Small values of C would result in relaxed margins, whereas large values of C would minimize training data errors and result in narrow margins. Setting C to be sufficiently large would disallow any training data error in separable classes, which is the same as setting all slack variables to 0 and defaulting to the hard version of the problem. A popular choice of r is 1, which is also referred to as *hinge loss*. Therefore, the objective function for soft-margin SVMs, with hinge loss, is defined as follows:

$$O = \frac{||\overline{W}||^2}{2} + C \sum_{i=1}^{n} \xi_i. \tag{10.56}$$

As before, this is a convex quadratic optimization problem that can be solved using Lagrangian methods. A similar approach is used to set up the Lagrangian relaxation of the problem with penalty terms and additional multipliers $\beta_i \geq 0$ for the slack constraints $\xi_i \geq 0$:

$$L_P = \frac{||\overline{W}||^2}{2} + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \lambda_i \left[y_i(\overline{W} \cdot \overline{X_i} + b) - 1 + \xi_i \right] - \sum_{i=1}^{n} \beta_i \xi_i. \tag{10.57}$$

A similar approach to the hard SVM case can be used to eliminate the minimization variables $\overline{W}$, ξ_i, and b from the optimization formulation and create a purely maximization dual formulation. This is achieved by setting the gradient of L_P with respect to these variables to 0. By setting the gradients of L_P with respect to $\overline{W}$ and b to 0, it can be respectively shown that the value of $\overline{W}$ is identical to the hard-margin case (Eq. 10.49), and the same multiplier constraint $\sum_{i=1}^{n} \lambda_i y_i = 0$ is satisfied. This is because the additional slack terms in L_P involving ξ_i do not affect the respective gradients with respect to $\overline{W}$ and b. Furthermore, it can be shown that the objective function L_D of the Lagrangian dual in the soft-margin case is identical to that of the hard-margin case, according to Eq. 10.50, because the linear terms involving each ξ_i evaluate[5] to 0. The *only* change to the dual optimization problem is that the nonnegative Lagrangian multipliers satisfy additional constraints of the form $C - \lambda_i = \beta_i \geq 0$. This constraint is derived by setting the partial derivative of L_P with respect to ξ_i to 0. One way of viewing this additional constraint $\lambda_i \leq C$ is that the influence of any training data point $\overline{X_i}$ on the weight vector $\overline{W} = \sum_{i=1}^{n} \lambda_i y_i \overline{X_i}$ is capped by C because of the softness of the margin. *The dual problem in soft SVMs maximizes L_D (Eq. 10.50) subject to the constraints $0 \leq \lambda_i \leq C$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$.*

The Kuhn–Tucker optimality conditions for the slack nonnegativity constraints are $\beta_i \xi_i = 0$. Because we have already derived $\beta_i = C - \lambda_i$, we obtain $(C - \lambda_i)\xi_i = 0$. In other words, training points $\overline{X_i}$ with $\lambda_i < C$ correspond to zero slack ξ_i and they might either lie on the margin or on the correct side of the margin. However, in this case, the support vectors are defined as data points that satisfy the *soft* SVM constraints exactly and some of them might have nonzero slack. Such points might lie on the margin, between the margin, or on the wrong side of the decision boundary. Points that satisfy $\lambda_i > 0$ are always support vectors. The support vectors that lie on the margin will therefore satisfy $0 < \lambda_i < C$. These points are very useful in solving for b. Consider any such support vector $\overline{X_r}$ with zero slack, which satisfies $0 < \lambda_r < C$. The value of b may be obtained as before:

$$y_r \left[(\sum_{i=1}^{n} \lambda_i y_i \overline{X_i} \cdot \overline{X_r}) + b \right] = +1. \tag{10.58}$$

Note that this expression is the same as for the case of hard SVMs, except that the relevant training points are identified by using the condition $0 < \lambda_r < C$. The gradient-ascent update is also identical to the separable case (cf. Sect. 10.6.1.1), except that any multiplier λ_i exceeding C because of an update needs to be reset to C. The classification of a test instance also uses Eq. 10.53 in terms of Lagrangian multipliers because the relationship between the weight vector and the Lagrangian multipliers is the same in this case. Thus, the soft SVM formulation with hinge loss is strikingly similar to the hard SVM formulation. This similarity is less pronounced for other slack penalty functions such as quadratic loss.

The soft version of SVMs also allows an *unconstrained* primal formulation by eliminating the margin constraints and slack variables simultaneously. This is achieved by substituting $\xi_i = \max\{0, 1 - y_i[\overline{W} \cdot \overline{X_i} + b]\}$ in the primal objective function of Eq. 10.56. This results in an unconstrained optimization (minimization) problem purely in terms of $\overline{W}$ and b:

$$O = \frac{||\overline{W}||^2}{2} + C \sum_{i=1}^{n} \max\{0, 1 - y_i[\overline{W} \cdot \overline{X_i} + b]\}. \tag{10.59}$$

One can use a gradient descent approach, which is analogous to the gradient ascent method used in logistic regression. The partial derivatives of nondifferentiable function O with respect to $w_1, \ldots w_d$ and b are approximated on a casewise basis, depending on whether or not the term inside the maximum function evaluates to a positive quantity. The precise derivation of the gradient descent steps is left as an exercise for the reader. While the dual

[5]The additional term in L_P involving ξ_i is $(C - \beta_i - \lambda_i)\xi_i$. This term evaluates to 0 because the partial derivative of L_P with respect to ξ_i is $(C - \beta_i - \lambda_i)$. This partial derivative must evaluate to 0 for optimality of L_P.

approach is more popular, the primal approach is intuitively simpler, and it is often more efficient when an approximate solution is desired.

10.6.2.1 Comparison with Other Linear Models

The normal vector to a linear separating hyperplane can be viewed as a direction along which the data points of the two classes are best separated. Fisher's linear discriminant also achieves this goal by maximizing the ratio of the between-class scatter to the within-class scatter along an optimally chosen vector. However, an important distinguishing feature of SVMs is that they focus extensively on the *decision boundary* region between the two classes because this is the most uncertain region, which is prone to classification error. Fisher's discriminant focuses on the global separation between the two classes and may not necessarily provide the best separation in the uncertain boundary region. This is the reason that SVMs often have better generalization behavior for noisy data sets that are prone to overfitting.

It is instructive to express logistic regression as a minimization problem by using the negative of the log-likelihood function and then comparing it with SVMs. The coefficients $(\theta_0, \ldots \theta_d)$ in logistic regression are analogous to the coefficients $(b, \overline{W})$ in SVMs. SVMs have a margin component to increase the generalization power of the classifier, just as logistic regression uses regularization. Interestingly, the margin component $||\overline{W}||^2/2$ in SVMs has an identical form to the regularization term $\sum_{i=1}^{d} \theta_i^2/2$ in logistic regression. SVMs have slack penalties just as logistic regression implicitly penalizes the *probability of* mistakes in the log-likelihood function. However, the slack is computed using *margin violations* in SVMs, whereas the penalties in logistic regression are computed as a smooth function of the distances from the *decision boundary.* Specifically, the log-likelihood function in logistic regression creates a smooth loss function of the form $log(1 + e^{-y_i[\theta_0 + \overline{\theta} \cdot \overline{X_i}]})$, whereas the hinge loss $max\{0, 1 - y_i[\overline{W} \cdot \overline{X_i} + b]\}$ in SVMs is not a smooth function. The nature of the misclassification penalty is the only difference between the two models. Therefore, there are several conceptual similarities among these models, but they emphasize different aspects of optimization. SVMs and regularized logistic regression show similar performance in many practical settings with poorly separable classes. However, SVMs and Fisher's discriminant generally perform better than logistic regression for the special case of well-separated classes. All these methods can also be extended to nonlinear decision boundaries in similar ways.

10.6.3 Nonlinear Support Vector Machines

In many cases, linear solvers are not appropriate for problems in which the decision boundary is not linear. To understand this point, consider the data distribution illustrated in Fig. 10.8. It is evident that no linear separating hyperplanes can delineate the two classes. This is because the two classes are separated by the following decision boundary:

$$8(x_1 - 1)^2 + 50(x_2 - 2)^2 = 1. \tag{10.60}$$

Now, if one already had some insight about the nature of the decision boundary, one might transform the training data into the new 4-dimensional space as follows:

$$\begin{aligned} z_1 &= x_1^2 \\ z_2 &= x_1 \\ z_3 &= x_2^2 \\ z_4 &= x_2. \end{aligned}$$

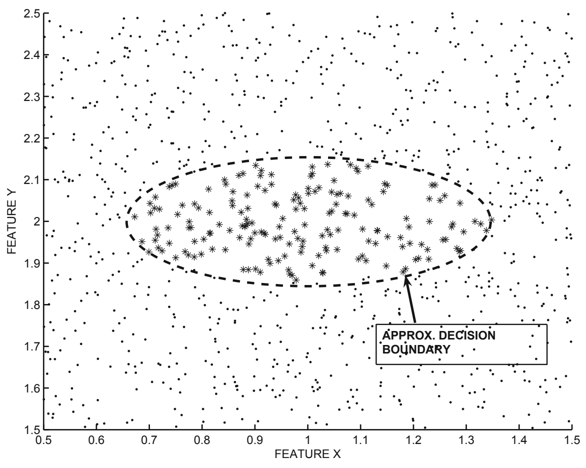


Figure 10.8: Nonlinear decision surface

The decision boundary of Eq. 10.60 can be expressed linearly in terms of the variables $z_1 \ldots z_4$, by expanding Eq. 10.60 in terms of x_1, x_1^2, x_2, and x_2^2:

$$\begin{aligned} 8x_1^2 - 16x_1 + 50x_2^2 - 200x_2 + 207 = 0 \\ 8z_1 - 16z_2 + 50z_3 - 200z_4 + 207 = 0. \end{aligned}$$

Thus, each training data point is now expressed in terms of these four newly transformed dimensions, and the classes will be linearly separable in this space. The SVM optimization formulation can then be solved in the transformed space as a linear model, and used to classify test instances that are also transformed to 4-dimensional space. It is important to note that the complexity of the problem effectively increased because of the increase in the size of the hyperplane coefficient vector $\overline{W}$.

In general, it is possible to approximate any polynomial decision boundary by adding an additional set of dimensions for each exponent of the polynomial. High-degree polynomials have significant expressive power in approximating many nonlinear functions well. This kind of transformation can be very effective in cases where one does not know whether the decision boundary is linear or nonlinear. This is because the additional degrees of freedom in the model, in terms of the greater number of coefficients to be learned, can determine the linearity or nonlinearity of the decision boundary in a data-driven way. In our previous example, if the decision boundary had been linear, the coefficients for z_1 and z_3 would automatically have been learned to be almost 0, given enough training data. The price for this additional flexibility is the increased computational complexity of the training problem, and the larger number of coefficients that need to be learned. Furthermore, if enough training data is not available, then this may result in overfitting where even a simple linear decision boundary is incorrectly approximated as a nonlinear one. A different approach, which is sometimes used to learn nonlinear decision boundaries, is known as the "kernel trick." This approach is able to learn arbitrary decision boundaries without performing the transformation explicitly.

10.6.4 The Kernel Trick

The kernel trick leverages the important observation that the SVM formulation can be fully solved in terms of dot products (or similarities) between pairs of data points. One does not need to know the feature values. Therefore, the key is to define the pairwise dot product (or similarity function) directly in the d'-dimensional transformed representation $\Phi(\overline{X})$, with the use of a kernel function $K(\overline{X_i}, \overline{X_j})$.

$$K(\overline{X_i}, \overline{X_j}) = \Phi(\overline{X_i}) \cdot \Phi(\overline{X_j}) \tag{10.61}$$

To effectively solve the SVM, recall that the transformed feature values $\Phi(\overline{X})$ need not be explicitly computed, as long as the dot product (or kernel similarity) $K(\overline{X_i}, \overline{X_j})$ is known. This implies that the term $\overline{X_i} \cdot \overline{X_j}$ may be replaced by the *transformed-space* dot product $K(\overline{X_i}, \overline{X_j})$ in Eq. 10.50, and the term $\overline{X_i} \cdot \overline{Z}$ in Eq. 10.53 can be replaced by $K(\overline{X_i}, \overline{Z})$ to perform SVM classification.

$$L_D = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j K(\overline{X_i}, \overline{X_j}) \tag{10.62}$$

$$F(\overline{Z}) = \text{sign}\{(\sum_{i=1}^{n} \lambda_i y_i K(\overline{X_i}, \overline{Z})) + b\} \tag{10.63}$$

Note that the bias b is also expressed in terms of dot products according to Eq. 10.58. These modifications are carried over to the update equations discussed in Sect. 10.6.1.1, all of which are expressed in terms of dot products.

Thus, all computations are performed in the *original* space, and the actual transformation $\Phi(\cdot)$ does not need to be known as long as the kernel similarity function $K(\cdot, \cdot)$ is known. By using kernel-based similarity with carefully chosen kernels, arbitrary nonlinear decision boundaries can be approximated. There are different ways of modeling similarity between $\overline{X_i}$ and $\overline{X_j}$. Some common choices of the kernel function are as follows:

Function	**Form**
Gaussian radial basis kernel	$K(\overline{X_i}, \overline{X_j}) = e^{-\|\overline{X_i}-\overline{X_j}\|^2/2\sigma^2}$
Polynomial kernel	$K(\overline{X_i}, \overline{X_j}) = (\overline{X_i} \cdot \overline{X_j} + c)^h$
Sigmoid kernel	$K(\overline{X_i}, \overline{X_j}) = \tanh(\kappa \overline{X_i} \cdot \overline{X_j} - \delta)$

Many of these kernel functions have parameters associated with them. In general, these parameters may need to be tuned by holding out a portion of the training data, and using it to test the accuracy of different choices of parameters. Many other kernels are possible beyond the ones listed in the table above. Kernels need to satisfy a property known as *Mercer's theorem* to be considered valid. This condition ensures that the $n \times n$ kernel similarity matrix $S = [K(\overline{X_i}, \overline{X_j})]$ is positive semidefinite, and similarities can be expressed as dot products in some transformed space. Why must the kernel similarity matrix always be positive semidefinite for similarities to be expressed as dot products? Note that if the $n \times n$ kernel similarity matrix S can be expressed as the $n \times n$ dot-product matrix AA^T of some $n \times r$ transformed representation A of the points, then for any n-dimensional column vector $\overline{V}$, we have $\overline{V}^T S \overline{V} = (A\overline{V})^T (A\overline{V}) \geq 0$. In other words, S is positive semidefinite. Conversely, if the kernel matrix S is positive semi-definite then it can be expressed as a dot product with the eigen decomposition $S = Q\Sigma^2 Q^T = (Q\Sigma)(Q\Sigma)^T$, where Σ^2 is an $n \times n$ diagonal matrix of nonnegative eigenvalues and Q is an $n \times n$ matrix containing the eigenvectors of S in columns. The matrix $Q\Sigma$ is the n-dimensional transformed representation of the points, and it also sometimes referred to as the *data-specific Mercer kernel map*. This map is data set-specific, and it is used in many nonlinear dimensionality reduction methods such as kernel *PCA*.

What kind of kernel function works best for the example of Fig. 10.8? In general, there are no predefined rules for selecting kernels. Ideally, if the similarity values $K(\overline{X_i}, \overline{X_j})$ were defined so that a space exists, in which points with this similarity structure are linearly separable, then a linear SVM in the transformed space $\Phi(\cdot)$ will work well.

To explain this point, we will revisit the example of Fig. 10.8. Let $\overline{X2_i}$ and $\overline{X2_j}$ be the d-dimensional vectors derived by squaring each coordinate of $\overline{X_i}$ and $\overline{X_j}$, respectively. In the case of Fig. 10.8, consider the transformation (z_1, z_2, z_3, z_4) in the previous section. It can be shown that the dot product between two transformed data points can be captured by the following kernel function:

$$\textit{Transformed-Dot-Product}(\overline{X_i}, \overline{X_j}) = \overline{X_i} \cdot \overline{X_j} + \overline{X2_i} \cdot \overline{X2_j}. \tag{10.64}$$

This is easy to verify by expanding the aforementioned expression in terms of the transformed variables $z_1 \dots z_4$ of the two data points. The kernel function *Transformed-Dot-Product*$(\overline{X_i}, \overline{X_j})$ would obtain the same Lagrangian multipliers and decision boundary as obtained with the explicit transformation $z_1 \dots z_4$. Interestingly, this kernel is closely related to the second-order polynomial kernel.

$$K(\overline{X_i}, \overline{X_j}) = (0.5 + \overline{X_i} \cdot \overline{X_j})^2 \tag{10.65}$$

Expanding the second-order polynomial kernel results in a superset of the additive terms in *Transformed-Dot-Product*$(\overline{X_i}, \overline{X_j})$. The additional terms include a constant term of 0.25 and some inter-dimensional products. These terms provide further modeling flexibility. In the case of the 2-dimensional example of Fig. 10.8, the use of the second-order polynomial kernel is equivalent to using an extra transformed variable $z_5 = \sqrt{2}x_1x_2$ representing the product of the values on the two dimensions and a constant dimension $z_6 = 0.5$. These variables are in addition to the original four variables (z_1, z_2, z_3, z_4). Since these additional variables are redundant in this case, they will not affect the ability to discover the correct decision boundary, although they might cause some overfitting. On the other hand, a variable such as $z_5 = \sqrt{2}x_1x_2$ would have come in handy, if the ellipse of Fig. 10.8 had been arbitrarily oriented with respect to the axis system. A full separation of the classes would not have been possible with a linear classifier on the original four variables (z_1, z_2, z_3, z_4). Therefore, the second-order polynomial kernel can discover more general decision boundaries than the transformation of the previous section. Using even higher-order polynomial kernels can model increasingly complex boundaries but at a greater risk of overfitting.

In general, different kernels have different levels of flexibility. For example, a transformed feature space that is implied by the Gaussian kernel of width σ can be shown to have an infinite number of dimensions by using the polynomial expansion of the exponential term. The parameter σ controls the relative scaling of various dimensions. A smaller value of σ results in a greater ability to model complex boundaries, but it may also cause overfitting. Smaller data sets are more prone to overfitting. Therefore, the optimal values of kernel parameters depend not only on the shape of the decision boundary but also on the size of the training data set. Parameter tuning is important in kernel methods. With proper tuning, many kernel functions can model complex decision boundaries. Furthermore, kernels provide a natural route for using SVMs in complex data types. This is because kernel methods only need the pairwise similarity between objects, and are agnostic to the feature values of the data points. Kernel functions have been defined for text, images, sequences, and graphs.

10.6.4.1 Other Applications of Kernel Methods

The use of kernel methods is not restricted to SVM methods. These methods can be extended to any technique in which the solutions are directly or indirectly expressed in terms of dot products. Examples include the Fisher's discriminant, logistic regression, linear regression (cf. Sect. 11.5.4 of Chap. 11), dimensionality reduction, and k-means clustering.

1. *Kernel k-means:* The key idea is that the Euclidean distance between a data point $\overline{X}$ and the cluster centroid $\overline{\mu}$ of cluster $\mathcal{C}$ can be computed as a function of the dot product between $\overline{X}$ and the data points in $\mathcal{C}$:

$$||\overline{X}-\overline{\mu}||^2 = ||\overline{X}-\frac{\sum_{\overline{X_i}\in\mathcal{C}}\overline{X_i}}{|\mathcal{C}|}||^2 = \overline{X}\cdot\overline{X}-2\frac{\sum_{\overline{X_i}\in\mathcal{C}}\overline{X}\cdot\overline{X_i}}{|\mathcal{C}|}+\frac{\sum_{\overline{X_i},\overline{X_j}\in\mathcal{C}}\overline{X_i}\cdot\overline{X_j}}{|\mathcal{C}|^2}. \quad (10.66)$$

   In kernel k-means, the dot products $\overline{X_i}\cdot\overline{X_j}$ are replaced with kernel similarity values $K(\overline{X_i},\overline{X_j})$. For the data point $\overline{X}$, the index of its assigned cluster is obtained by selecting the minimum value of the (kernel-based) distance in Eq. 10.66 over all clusters. Note that the cluster centroids in the transformed space do not need to be explicitly maintained over the different k-means iterations, although the cluster assignment indices for each data point need to be maintained for computation of Eq. 10.66. Because of its implicit nonlinear transformation approach, kernel k-means is able to discover arbitrarily shaped clusters like spectral clustering in spite of its use of the spherically biased Euclidean distance.

2. *Kernel PCA:* In conventional *SVD* and *PCA* of an $n \times d$ mean-centered data matrix D, the basis vectors are given by the eigenvectors of D^TD (columnwise dot product matrix), and the coordinates of the transformed points are extracted from the scaled eigenvectors of DD^T (rowwise dot product matrix). While the basis vectors can no longer be derived in kernel *PCA*, the coordinates of the transformed data can be extracted. The rowwise dot product matrix DD^T can be replaced with the kernel similarity matrix $S = [K(\overline{X_i},\overline{X_j})]_{n\times n}$. The similarity matrix is then adjusted for mean-centering of the data in the transformed space as $S \Leftarrow (I-\frac{U}{n})S(I-\frac{U}{n})$, where U is an $n\times n$ matrix containing all 1s (see Exercise 17). The assumption is that the matrix S can be approximately expressed as a dot product of the reduced data points in some k-dimensional transformed space. Therefore, one needs to approximately factorize S into the form AA^T to extract its reduced $n\times k$ embedding A in the transformed space. This is achieved by eigen-decomposition. Let Q_k be the $n \times k$ matrix containing the largest k eigenvectors of S, and Σ_k be the $k \times k$ diagonal matrix containing the square root of the corresponding eigenvalues. Then, it is evident that $S \approx Q_k\Sigma_k^2Q_k^T = (Q_k\Sigma_k)(Q_k\Sigma_k)^T$, and the k-dimensional embeddings of the data points are given[6] by the rows of the $n \times k$ matrix $A = Q_k\Sigma_k$. Note that this is a truncated version of the data-specific Mercer kernel map. This nonlinear embedding is similar to that obtained

[6] The original result [450] uses a more general argument to derive $S'Q_k\Sigma_k^{-1}$ as the $m \times k$ matrix of k-dimensional embedded coordinates of any *out-of-sample* $m \times d$ matrix D'. Here, $S' = D'D^T$ is the $m\times n$ matrix of kernel similarities between out-of-sample points in D' and in-sample points in D. However, when $D' = D$, this expression is (more simply) equivalent to $Q_k\Sigma_k$ by expanding $S' = S \approx Q_k\Sigma_k^2Q_k^T$.

by *ISOMAP*; however, unlike *ISOMAP*, out-of-sample points can also be transformed to the new space. It is noteworthy that the embedding of spectral clustering is also expressed in terms of the large eigenvectors[7] of a *sparsified* similarity matrix, which is better suited to preserving *local* similarities for clustering. In fact, most forms of nonlinear embeddings can be shown to be large eigenvectors of similarity matrices (cf. Table 2.3 of Chap. 2), and are therefore special cases of kernel *PCA*.

10.7 Neural Networks

Neural networks are a model of simulation of the human nervous system. The human nervous system is composed of cells, referred to as neurons. Biological neurons are connected to one another at contact points, which are referred to as synapses. Learning is performed in living organisms by changing the strength of synaptic connections between neurons. Typically, the strength of these connections change in response to external stimuli. Neural networks can be considered a simulation of this biological process.

As in the case of biological networks, the individual nodes in artificial neural networks are referred to as *neurons*. These neurons are units of computation that receive input from some other neurons, make computations on these inputs, and feed them into yet other neurons. The computation function at a neuron is defined by the weights on the input connections to that neuron. This weight can be viewed as analogous to the strength of a synaptic connection. By changing these weights appropriately, the computation function can be learned, which is analogous to the learning of the synaptic strength in biological neural networks. The "external stimulus" in artificial neural networks for learning these weights is provided by the training data. The idea is to incrementally modify the weights whenever incorrect predictions are made by the current set of weights.

The key to the effectiveness of the neural network is the *architecture* used to arrange the connections among nodes. A wide variety of architectures exist, starting from a simple single-layer *perceptron* to complex multilayer networks.

10.7.1 Single-Layer Neural Network: The Perceptron

The most basic architecture of a neural network is referred to as the *perceptron*. An example of the perceptron architecture is illustrated in Fig. 10.10a. The perceptron contains two layers of nodes, which correspond to the input nodes, and a single output node. The number of input nodes is exactly equal to the dimensionality d of the underlying data. Each input node receives and transmits a single numerical attribute to the output node. Therefore, the input nodes only *transmit* input values and do not perform any *computation* on these values. In the basic perceptron model, the output node is the only node that performs a mathematical function on its inputs. The individual features in the training data are assumed to be numerical. Categorical attributes are handled by creating a separate binary input for each value of the categorical attribute. This is logically equivalent to binarizing the categorical attribute into multiple attributes. For simplicity of further discussion, it will be assumed that all input variables are numerical. Furthermore, it will be assumed that the classification problem contains two possible values for the class label, drawn from $\{-1, +1\}$.

[7]Refer to Sect. 19.3.4 of Chap. 19. The small eigenvectors of the symmetric Laplacian are the same as the large eigenvectors of $S = \Lambda^{-1/2} W \Lambda^{-1/2}$. Here, W is often defined by the *sparsified* heat-kernel similarity between data points, and the factors involving $\Lambda^{-1/2}$ provide local normalization of the similarity values to handle clusters of varying density.

As discussed earlier, each input node is connected by a weighted connection to the output node. These weights define a function from the values transmitted by the input nodes to a binary value drawn from $\{-1, +1\}$. This value can be interpreted as the perceptron's prediction of the class variable of the test instance fed to the input nodes, for a binary-class value drawn from $\{-1, +1\}$. Just as learning is performed in biological systems by modifying synaptic strengths, the learning in a perceptron is performed by modifying the weights of the links connecting the input nodes to the output node whenever the predicted label does not match the true label.

The function learned by the perceptron is referred to as the *activation function*, which is a signed linear function. This function is very similar to that learned in SVMs for mapping training instances to binary class labels. Let $\overline{W} = (w_1 \dots w_d)$ be the weights for the connections of d different inputs to the output neuron for a data record of dimensionality d. In addition, a bias b is associated with the activation function. The output $z_i \in \{-1, +1\}$ for the feature set $(x_i^1 \dots x_i^d)$ of the ith data record $\overline{X_i}$, is as follows:

$$\begin{aligned} z_i &= \text{sign}\{\sum_{j=1}^{d} w_j x_i^j + b\} && (10.67) \\ &= \text{sign}\{\overline{W} \cdot \overline{X_i} + b\} && (10.68) \end{aligned}$$

The value z_i represents the *prediction* of the perceptron for the class variable of $\overline{X_i}$. It is, therefore, desired to learn the weights, so that the value of z_i is equal to y_i for as many training instances as possible. The error in prediction $(z_i - y_i)$ may take on any of the values of -2, 0, or $+2$. A value of 0 is attained when the predicted class is correct. The goal in neural network algorithms is to learn the vector of weights $\overline{W}$ and bias b, so that z_i approximates the true class variable y_i as closely as possible.

The basic perceptron algorithm starts with a random vector of weights. The algorithm then feeds the input data items $\overline{X_i}$ into the neural network one by one to create the prediction z_i. The weights are then updated, based on the error value $(z_i - y_i)$. Specifically, when the data point $\overline{X_i}$ is fed into it in the tth iteration, the weight vector $\overline{W}^t$ is updated as follows:

$$\overline{W}^{t+1} = \overline{W}^t + \eta(y_i - z_i)\overline{X_i}. \tag{10.69}$$

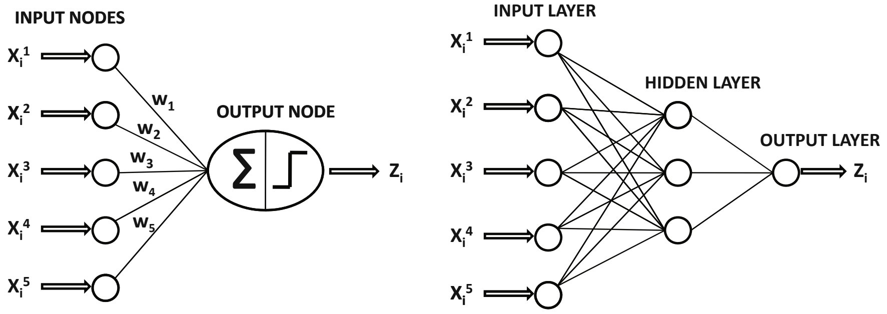
The parameter η regulates the learning rate of the neural network. The perceptron algorithm repeatedly cycles through all the training examples in the data and iteratively adjusts the weights until convergence is reached. The basic perceptron algorithm is illustrated in Fig. 10.9. Note that a single training data point may be cycled through many times. Each such cycle is referred to as an *epoch*.

Let us examine the incremental term $(y_i - z_i)\overline{X_i}$ in the update of Eq. 10.69, without the multiplicative factor η. It can be shown that this term is a heuristic approximation[8] of the negative of the gradient of the least-squares prediction error $(y_i - z_i)^2 = (y_i - \text{sign}(\overline{W} \cdot \overline{X_i} - b))^2$ of the class variable, with respect to the vector of weights $\overline{W}$. The update in this case is performed on a tuple-by-tuple basis, rather than globally, over the entire data set, as one would expect in a global least-squares optimization. Nevertheless, the basic perceptron algorithm can be considered a modified version of the gradient descent method, which implicitly minimizes the squared error of prediction. It is easy to see that nonzero updates are made to the weights only when errors are made in categorization. This is because the incremental term in Eq. 10.69 will be 0 whenever the predicted value z_i is the same as the class label y_i.

[8]The derivative of the sign function is replaced by only the derivative of its argument. The derivative of the sign function is zero everywhere, except at zero, where it is indeterminate.

```
Algorithm Perceptron(Training Data: $\mathcal{D}$)
begin
  Initialize weight vector $\overline{W}$ to random values;
  repeat
    Receive next training tuple $(\overline{X_i}, y_i)$;
    $z_i = \overline{W} \cdot \overline{X_i} + b$;
    $\overline{W} = \overline{W} + \eta(y_i - z_i)\overline{X_i}$;
  until convergence;
end
```

Figure 10.9: The perceptron algorithm



(a) Perceptron (b) Multilayer

Figure 10.10: Single and multilayer neural networks

A question arises as to how the learning rate η may be chosen. A high value of η will result in fast learning rates, but may sometimes result in suboptimal solutions. Smaller values of η will result in a convergence to higher-quality solutions, but the convergence will be slow. In practice, the value of η is initially chosen to be large and gradually reduced, as the weights become closer to their optimal values. The idea is that large steps are likely to be helpful early on, but may result in oscillation between suboptimal solutions at later stages. For example, the value of η is sometimes selected to be proportional to the inverse of the number of cycles through the training data (or epochs) so far.

10.7.2 Multilayer Neural Networks

The perceptron model is the most basic form of a neural network, containing only a single input layer and an output layer. Because the input layers only transmit the attribute values without actually applying any mathematical function on the inputs, the function learned by the perceptron model is only a simple linear model based on a single output node. In practice, more complex models may need to be learned with multilayer neural networks.

Multilayer neural networks have a *hidden layer*, in addition to the input and output layers. The nodes in the hidden layer can, in principle, be connected with different types of topologies. For example, the hidden layer can itself consist of multiple layers, and nodes in one layer might feed into nodes of the next layer. This is referred to as the *multilayer feed-forward network*. The nodes in one layer are also assumed to be fully connected to the nodes in the next layer. Therefore, the topology of the multilayer feed-forward network is automatically determined, after the number of layers, and the number of nodes in each layer, have been specified by the analyst. The basic perceptron may be viewed as a single-layer feed-forward network. A popularly used model is one in which a multilayer feed-forward network contains only a single hidden layer. Such a network may be considered a two-layer feed-forward network. An example of a two-layer feed-forward network is illustrated in Fig. 10.10b. Another aspect of the multilayer feed-forward network is that it is not restricted to the use of linear signed functions of the inputs. Arbitrary functions such as the logistic, sigmoid, or hyperbolic tangents may be used in different nodes of the hidden layer and output layer. An example of such a function, when applied to the training tuple $\overline{X_i} = (x_i^1 \dots x_i^d)$, to yield an output value of z_i, is as follows:

$$z_i = \sum_{j=1}^{d} w_j \frac{1}{1 + e^{-x_i^j}} + b. \tag{10.70}$$

The value of z_i is no longer a predicted output of the final class label in $\{-1, +1\}$, if it refers to a function computed at the hidden layer nodes. This output is then propagated forward to the next layer.

In the single-layer neural network, the training process was relatively straightforward because the *expected* output of the output node was known to be equal to the training label value. The known *ground truth* was used to create an optimization problem in least squares form, and update the weights with a gradient-descent method. Because the output node is the only neuron with weights in a single-layer network, the update process is easy to implement. In the case of multilayer networks, the problem is that the ground-truth output of the hidden layer nodes are not known because there are no training labels associated with the outputs of these nodes. Therefore, a question arises as to how the weights of these nodes should be updated when a training example is classified incorrectly. Clearly, when a classification error is made, some kind of "feedback" is required from the nodes in the forward layers to the nodes in earlier layers about the *expected* outputs (and corresponding errors). This is achieved with the use of the *backpropagation* algorithm. Although this algorithm is not discussed in detail in this chapter, a brief summary is provided here. The backpropagation algorithm contains two main phases, which are applied in the weight update process for each training instance:

1. *Forward phase:* In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The final predicted output can be compared to the class label of the training instance, to check whether or not the predicted label is an error.

2. *Backward phase:* The main goal of the backward phase is to learn weights in the backward direction by providing an error estimate of the output of a node in the earlier layers from the errors in later layers. The error estimate of a node in the hidden layer is computed as a function of the error estimates and weights of the nodes in the layer ahead of it. This is then used to compute an error gradient with respect to the weights in the node and to update the weights of this node. The actual update equation is not very different from the basic perceptron at a conceptual level. The only differences that arise are due to the nonlinear functions commonly used in hidden layer nodes, and the fact that errors at hidden layer nodes are estimated via backpropagation, rather than directly computed by comparison of the output to a training label. This entire process is propagated backwards to update the weights of all the nodes in the network.

The basic framework of the multilayer update algorithm is the same as that for the single-layer algorithm illustrated in Fig. 10.9. The major difference is that it is no longer possible to use Eq. 10.69 for the hidden layer nodes. Instead, the update procedure is substituted with the forward–backward approach discussed above. As in the case of the single-layer network, the process of updating the nodes is repeated to convergence by repeatedly cycling through the training data in epochs. A neural network may sometimes require thousands of epochs through the training data to learn the weights at the different nodes.

A multilayer neural network is more powerful than a kernel SVM in its ability to capture arbitrary functions. A multilayer neural network has the ability to not only capture decision boundaries of arbitrary shapes, but also capture noncontiguous class distributions with different decision boundaries in different regions of the data. Logically, the different nodes in the hidden layer can capture the different decision boundaries in different regions of the data, and the node in the output layer can combine the results from these different decision boundaries. For example, the three different nodes in the hidden layer of Fig. 10.10b could conceivably capture three different nonlinear decision boundaries of different shapes in different localities of the data. With more nodes and layers, virtually any function can be approximated. This is more general than what can be captured by a kernel-based SVM that learns a single nonlinear decision boundary. In this sense, neural networks are viewed as *universal function approximators*. The price of this generality is that there are several implementation challenges in neural network design:

1. The initial design of the topology of the network presents many trade-off challenges for the analyst. A larger number of nodes and hidden layers provides greater generality, but a corresponding risk of overfitting. Little guidance is available about the design of the topology of the neural network because of poor interpretability associated with the multilayer neural network classification process. While some hill climbing methods can be used to provide a limited level of learning of the correct neural network topology, the issue of good neural network design still remains somewhat of an open question.

2. Neural networks are slow to train and sometimes sensitive to noise. As discussed earlier, thousands of epochs may be required to train a multilayer neural network. A larger network is likely to have a very slow learning process. While the training process of a neural network is slow, it is relatively efficient to classify test instances.

The previous discussion addresses only binary class labels. To generalize the approach to multiclass problems, a multiclass meta-algorithm discussed in the next chapter may be used. Alternatively, it is possible to modify both the basic perceptron model and the general neural network model to allow multiple output nodes. Each output node corresponds to the predicted value of a specific class label. The overall training process is exactly identical to the previous case, except that the weights of each output node now need to be trained.

10.7.3 Comparing Various Linear Models

Like neural networks, logistic regression also updates model parameters based on mistakes in categorization. This is not particularly surprising because both classifiers are linear classifiers but with different forms of the objective function for optimization. In fact, the use of some forms of logistic activation functions in the perceptron algorithm can be shown to be approximately equivalent to logistic regression. It is also instructive to examine the relationship of neural networks with SVM methods. In SVMs, the optimization function is based on the principle of maximum margin separation. This is different from neural networks, where the errors of predictions are directly penalized and then optimized with the use of a hill-climbing approach. In this sense, the SVM model has greater sophistication than the *basic* perceptron model by using the maximum margin principle to better focus on the more important decision boundary region. Furthermore, the generalization power of neural networks can be improved by using a (weighted) regularization penalty term $\lambda||\overline{W}||^2/2$ in the objective function. Note that this regularization term is similar to the maximum margin term in SVMs. The maximum margin term is, in fact, also referred to as the regularization term for SVMs. Variations of SVMs exist, in which the maximum margin term is replaced with an L_1 penalty $\sum_{i=1}^{d} |w_i|$. In such cases, the regularization interpretation is more natural than a margin-based interpretation. Furthermore, certain forms of the slack term in SVMs (e.g., quadratic slack) are similar to the main objective function in other linear models (e.g., least-squares models). The main difference is that the slack term is computed from the margin separators in SVMs rather than the decision boundary. This is consistent with the philosophy of SVMs that discourages training data points from not only being on the wrong side of the decision boundary, but also from being close to the decision boundary. Therefore, various linear models share a number of conceptual similarities, but they emphasize different aspects of optimization. This is the reason that maximum margin models are generally more robust to noise than linear models that use only distance-based penalties to reduce the number of data points on the wrong side of the separating hyperplanes. It has experimentally been observed that neural networks are sensitive to noise. On the other hand, multilayer neural networks can approximate virtually any complex function in principle.

10.8 Instance-Based Learning

Most of the classifiers discussed in the previous sections are *eager* learners in which the classification model is constructed *up front* and then used to classify a specific test instance. In instance-based learning, the training is delayed until the last step of classification. Such classifiers are also referred to as *lazy learners*. The simplest principle to describe instance-based learning is as follows:

Similar instances have similar class labels.

A natural approach for leveraging this general principle is to use nearest-neighbor classifiers. For a given test instance, the closest m training examples are determined. The dominant label among these m training examples is reported as the relevant class. In some variations of the model, an inverse distance-weighted scheme is used, to account for the varying importance of the m training instances that are closest to the test instance. An example of such an inverse weight function of the distance δ is $f(\delta) = e^{-\delta^2/t^2}$, where t is a user-defined parameter. Here, δ is the distance of the training point to the test instance. This weight is used as a vote, and the class with the largest vote is reported as the relevant label.

If desired, a nearest-neighbor index may be constructed up front, to enable more efficient retrieval of instances. The major challenge with the use of the nearest-neighbor classifier is the choice of the parameter m. In general, a very small value of m will not lead to robust classification results because of noisy variations within the data. On the other hand, large values of m will lose sensitivity to the underlying data locality. In practice, the appropriate value of m is chosen in a heuristic way. A common approach is to test different values of m for accuracy over the training data. While computing the m-nearest neighbors of a training instance $\overline{X}$, the data point $\overline{X}$ is not included[9] among the nearest neighbors. A similar approach can be used to learn the value of t in the distance-weighted scheme.

10.8.1 Design Variations of Nearest Neighbor Classifiers

A number of design variations of nearest-neighbor classifiers are able to achieve more effective classification results. This is because the Euclidean function is usually not the most effective distance metric in terms of its sensitivity to feature and class distribution. The reader is advised to review Chap. 3 on distance function design. Both unsupervised and supervised distance design methods can typically provide more effective classification results. Instead of using the Euclidean distance metric, the distance between two d-dimensional points $\overline{X}$ and $\overline{Y}$ is defined with respect to a $d \times d$ matrix A.

$$Dist(\overline{X}, \overline{Y}) = \sqrt{(\overline{X} - \overline{Y})A(\overline{X} - \overline{Y})^T} \tag{10.71}$$

This distance function is the same as the Euclidean metric when A is the identity matrix. Different choices of A can lead to better sensitivity of the distance function to the local and global data distributions. These different choices will be discussed in the following subsections.

10.8.1.1 Unsupervised Mahalanobis Metric

The Mahalanobis metric is introduced in Chap. 3. In this case, the value of A is chosen to be the inverse of the $d \times d$ covariance matrix Σ of the data set. The (i, j)th entry of the matrix Σ is the covariance between the dimensions i and j. Therefore, the Mahalanobis distance is defined as follows:

$$Dist(\overline{X}, \overline{Y}) = \sqrt{(\overline{X} - \overline{Y})\Sigma^{-1}(\overline{X} - \overline{Y})^T}. \tag{10.72}$$

The Mahalanobis metric adjusts well to the different scaling of the dimensions and the redundancies across different features. Even when the data is uncorrelated, the Mahalanobis metric is useful because it auto-scales for the naturally different ranges of attributes describing different physical quantities, such as age and salary. Such a scaling ensures that no single attribute dominates the distance function. In cases where the attributes are correlated, the Mahalanobis metric accounts well for the varying redundancies in different features. However, its major weakness is that it does not account for the varying shapes of the class distributions in the underlying data.

10.8.1.2 Nearest Neighbors with Linear Discriminant Analysis

To obtain the best results with a nearest-neighbor classifier, the distance function needs to account for the varying distribution of the different classes. For example, in the case of Fig. 10.11, there are two classes A and B, which are represented by "." and "*," respectively. The test instance denoted by X lies on the side of the boundary related to class A. However, the Euclidean metric does not adjust well to the arrangement of the class distribution, and a circle drawn around the test instance seems to include more points from class B than class A.

One way of resolving the challenges associated with this scenario, is to weight the most discriminating directions more in the distance function with an appropriate choice of the

[9]This approach is also referred to as leave-one-out cross-validation, and is described in detail in Sect. 10.9 on classifier evaluation.

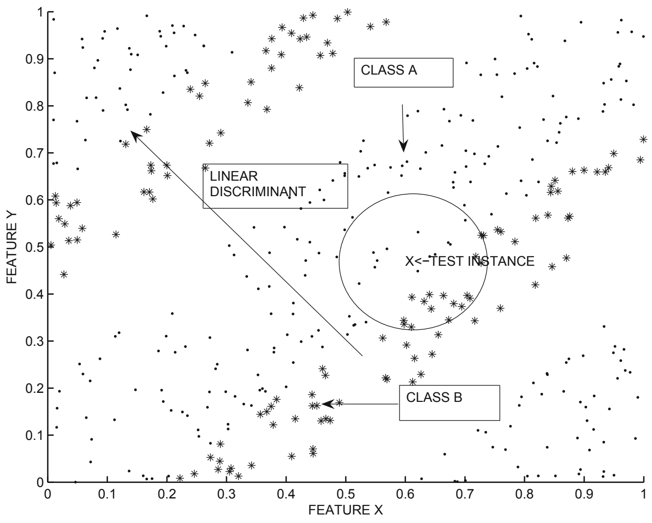


Figure 10.11: Importance of class sensitivity in distance function design

matrix A in Eq. 10.71. In the case of Fig. 10.11, the best discriminating direction is illustrated pictorially. Fisher's linear discriminant, discussed in Sect. 10.2.1.4, can be used to determine this direction, and map the data into a 1-dimensional space. In this 1-dimensional space, the different classes are separated out perfectly. The nearest-neighbor classifier will work well in this newly projected space. This is a very special example where only a 1-dimensional projection works well. However, it may not be generalizable to an arbitrary data set.

A more general way of computing the distances in a class-sensitive way, is to use a soft weighting of different directions, rather than selecting specific dimensions in a hard way. This can be achieved with the use of an appropriate choice of matrix A in Eq. 10.71. The choice of matrix A defines the shape of the neighborhood of a test instance. A distortion of this neighborhood from the circular Euclidean contour corresponds to a soft weighting, as opposed to a hard selection of specific directions. A soft weighting is also more robust in the context of smaller training data sets where the optimal linear discriminant cannot be found without overfitting. Thus, the core idea is to "elongate" the neighborhoods along the less discriminative directions and "shrink" the neighborhoods along the more discriminative directions with the use of matrix A. Note that the elongation of a neighborhood in a direction by a particular factor $\alpha > 1$, is equivalent to de-emphasizing that direction by that factor because distance components in that direction need to be divided by α. This is also done in the case of the Mahalanobis metric, except that the Mahalanobis metric is an unsupervised approach that is agnostic to the class distribution. In the case of the unsupervised Mahalanobis metric, the level of elongation achieved by the matrix A is inversely dependent on the variance along the different directions. In the supervised scenario, the goal is to elongate the directions, so that the level of elongation is inversely dependent on the ratio of the interclass variance to intraclass variance along the different directions.

Let $\mathcal{D}$ be the full database, and $\mathcal{D}_i$ be the portion of the data set belonging to class i. Let $\overline{\mu}$ represent the mean of the entire data set. Let $p_i = |\mathcal{D}_i|/|\mathcal{D}|$ be the fraction of data points belonging to class i, $\overline{\mu_i}$ be the d-dimensional row vector of means of $\mathcal{D}_i$, and Σ_i be the

$d \times d$ covariance matrix of $\mathcal{D}_i$. Then, the scaled[10] within-class scatter matrix S_w is defined as follows:

$$S_w = \sum_{i=1}^{k} p_i \Sigma_i. \tag{10.73}$$

The between-class scatter matrix S_b may be computed as follows:

$$S_b = \sum_{i=1}^{k} p_i (\overline{\mu_i} - \overline{\mu})^T (\overline{\mu_i} - \overline{\mu}). \tag{10.74}$$

Note that the matrix S_b is a $d \times d$ matrix because it results from the product of a $d \times 1$ matrix with a $1 \times d$ matrix. Then, the matrix A (of Eq. 10.71), which provides the desired distortion of the distances on the basis of class distribution, can be shown to be the following:

$$A = S_w^{-1} S_b S_w^{-1}. \tag{10.75}$$

It can be shown that this choice of the matrix A provides an excellent discrimination between the different classes, where the elongation in each direction depends inversely on ratio of the between-class variance to within-class variance along the different directions. The reader is referred to the bibliographic notes for pointers to the derivation of the aforementioned steps.

10.9 Classifier Evaluation

Given a classification model, how do we quantify its accuracy on a given data set? Such a quantification has several applications, such as evaluation of classifier effectiveness, comparing different models, selecting the best one for a particular data set, parameter tuning, and several meta-algorithms such as *ensemble analysis*. The last of these applications will be discussed in the next chapter. This leads to several challenges, both in terms of *methodology* used for the evaluation, and the specific approach used for quantification. These two challenges are stated as follows:

1. *Methodological issues:* The methodological issues are associated with dividing the labeled data appropriately into training and test segments for evaluation. As will become apparent later, the choice of methodology has a direct impact on the evaluation process, such as the underestimation or overestimation of classifier accuracy. Several approaches are possible, such as *holdout*, *bootstrap*, and *cross-validation*.

2. *Quantification issues:* The quantification issues are associated with providing a numerical measure for the quality of the method after a specific methodology (e.g., cross-validation) for evaluation has been selected. Examples of such measures could include the accuracy, the cost-sensitive accuracy, or a receiver operating characteristic curve quantifying the trade-off between true positives and false positives. Other types of numerical measures are specifically designed to compare the relative performance of classifiers.

In the following, these different aspects of classifier evaluation will be studied in detail.

[10]The unscaled version may be obtained by multiplying S_w with the number of data points. There is no difference to the final result whether the scaled or unscaled version is used, within a constant of proportionality.

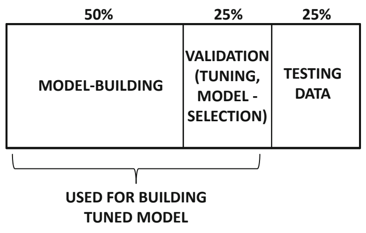

50%
25%
25%
MODEL-BUILDING
VALIDATION (TUNING, MODEL - SELECTION)
TESTING DATA
USED FOR BUILDING TUNED MODEL

Figure 10.12: Segmenting the labeled data for parameter tuning and evaluation

10.9.1 Methodological Issues

While the problem of classification is defined for unlabeled test examples, the evaluation process does need labels to be associated with the test examples as well. These labels correspond to the *ground truth* that is required in the evaluation process, but not used in the training. The classifier cannot use the same examples for both training and testing because such an approach will overestimate the accuracy of the classifier due to overfitting. It is desirable to construct models with high generalizability to *unseen* test instances.

A common mistake in the process of bench-marking classification models is that analysts often use the test set to tune the parameters of the classification algorithm or make other choices about classifier design. Such an approach might overestimate the true accuracy because *knowledge of the test set has been implicitly used in the training process.* In practice, the labeled data should be divided into three parts, which correspond to (a) the model-building part of the labeled data, (b) the validation part of the labeled data, and (c) the testing data. This division is illustrated in Fig. 10.12. The validation part of the data should be used for parameter tuning or *model selection.* Model selection (cf. Sect. 11.8.3.4 of Chap. 11) refers to the process of deciding which classification algorithm is best suited to a particular data set. The testing data should not even be looked at during this phase. After tuning the parameters, the classification model is sometimes reconstructed on the entire training data (including the validation but not test portion). Only at this point, the testing data can be used for evaluating the classification algorithm *at the very end.* Note that if an analyst uses insights gained from the resulting performance on the test data to again adjust the algorithm in some way, then the results will be contaminated with knowledge from the test set.

This section discusses how the labeled data may be divided into the data used for constructing the tuned model (i.e., first two portions) and testing data (i.e., third portion) to accurately estimate the classification accuracy. The methodologies discussed in this section are also used for dividing the first two portions into the first and second portions (e.g., for parameter tuning), although we consistently use the terminologies "training data" and "testing data" to describe the two portions of the division. One problem with segmenting the labeled data is that it affects the measured accuracy depending on how the segmentation is done. This is especially the case when the amount of labeled data is small because one might accidently sample a small test data set which is not an accurate representative of the training data. For cases in which the labeled data is small, careful methodological variations are required to prevent erroneous evaluations.

10.9.1.1 Holdout

In the holdout method, the labeled data is randomly divided into two disjoint sets, corresponding to the training and test data. Typically a majority (e.g., two-thirds or three-fourths) is used as the training data, and the remaining is used as the test data. The approach can be repeated several times with multiple samples to provide a final estimate. The problem with this approach is that classes that are overrepresented in the training data are also underrepresented in the test data. These random variations can have a significant impact when the original class distribution is imbalanced to begin with. Furthermore, because only a subset of the available labeled data is used for training, the full power of the training data is not reflected in the error estimate. Therefore, the error estimates obtained are pessimistic. By repeating the process over b different holdout samples, the mean and variance of the error estimates can be determined. The variance can be helpful in creating statistical confidence intervals on the error.

One of the challenges with using the holdout method robustly is the case when the classes are imbalanced. Consider a data set containing 1000 data points, with 990 data points belonging to one class and 10 data points belonging to the other class. In such cases, it is possible for a test sample of 200 data points to contain not even one data point belonging to the rare class. Clearly, in such cases, it will be difficult to estimate the classification accuracy, especially when cost-sensitive accuracy measures are used that weigh the various classes differently. Therefore, a reasonable alternative is to implement the holdout method by independently sampling the two classes at the same level. Therefore, exactly 198 data points will be sampled from the first class, and 2 data points will be sampled from the rare class to create the test data set. Such an approach ensures that the classes are represented to a similar degree in both the training and test sets.

10.9.1.2 Cross-Validation

In cross-validation, the labeled data is divided into m disjoint subsets of equal size n/m. A typical choice of m is around 10. One of the m segments is used for testing, and the other $(m-1)$ segments are used for training. This approach is repeated by selecting each of the m different segments in the data as a test set. The average accuracy over the different test sets is then reported. The size of the training set is $(m-1)n/m$. When m is chosen to be large, this is almost equal to the labeled data size, and therefore the estimation error is close to what would be obtained with the original training data, but only for a small set of test examples of size n/m. However, because every labeled instance is represented exactly once in the testing over the m different test segments, the overall accuracy of the cross-validation procedure tends to be a highly representative, but pessimistic estimate, of model accuracy. A special case is one where m is chosen to be n. Therefore, $(n-1)$ examples are used for training, and one example is used for testing. This is averaged over the n different ways of picking the test example. This is also referred to as *leave-one-out* cross-validation. This special case is rather expensive for large data sets because it requires the application of the training procedure n times. Nevertheless, such an approach is particularly natural for lazy learning methods, such as the nearest-neighbor classifier, where a training model does not need to be constructed up front. By repeating the process over b different random m-way partitions of the data, the mean and variance of the error estimates may be determined. The variance can be helpful in determining statistical confidence intervals on the error. *Stratified cross-validation* uses proportional representation of each class in the different folds and usually provides less pessimistic results.

10.9.1.3 Bootstrap

In the bootstrap method, the labeled data is sampled uniformly *with replacement*, to create a training data set, which might possibly contain duplicates. The labeled data of size n is sampled n times with replacement. This results in a training data with the same size as the original labeled data. However, the training typically contains duplicates and also misses some points in the original labeled data.

The probability that a particular data point is not included in a sample is given by $(1-1/n)$. Therefore, the probability that the data point is not included in n samples is given by $(1-1/n)^n$. For large values of n, this expression evaluates to approximately $1/e$, where e is the base of the natural logarithm. The fraction of the labeled data points included at least once in the training data is therefore $1-1/e \approx 0.632$. The training model $\mathcal{M}$ is constructed on the bootstrapped sample containing duplicates. The overall accuracy is computed using the original set of full labeled data as the test examples. The estimate is highly optimistic of the true classifier accuracy because of the large overlap between training and test examples. In fact, a 1-nearest neighbor classifier will always yield 100 % accuracy for the portion of test points included in the bootstrap sample and the estimates are therefore not realistic in many scenarios. By repeating the process over b different bootstrap samples, the mean and the variance of the error estimates may be determined.

A better alternative is to use *leave-one-out bootstrap*. In this approach, the accuracy $A(\overline{X})$ of each labeled instance $\overline{X}$ is computed using the classifier performance on only the subset of the b bootstrapped samples in which $\overline{X}$ is not a part of the bootstrapped sample of training data. The overall accuracy A_l of the leave-one-out bootstrap is the mean value of $A(\overline{X})$ over all labeled instances $\overline{X}$. This approach provides a pessimistic accuracy estimate. The 0.632-bootstrap further improves this accuracy with a “compromise” approach. The average *training-data* accuracy A_t over the b bootstrapped samples is computed. This is a highly optimistic estimate. For example, A_t will always be 100 % for a 1-nearest neighbor classifier. The overall accuracy A is a weighted average of the leave-one-out accuracy and the training-data accuracy.

$$A = (0.632) \cdot A_l + (0.368) \cdot A_t \tag{10.76}$$

In spite of the compromise approach, the estimates of 0.632 bootstrap are usually optimistic. The bootstrap method is more appropriate when the size of the labeled data is small.

10.9.2 Quantification Issues

This section will discuss how the quantification of the accuracy of a classifier is performed after the training and test set for a classifier are fixed. Several measures of accuracy are used depending on the nature of the classifier output:

1. In most classifiers, the output is predicted in the form of a label associated with the test instance. In such cases, the ground-truth label of the test instance is compared with the predicted label to generate an overall value of the classifier accuracy.

2. In many cases, the output is presented as a numerical score *for each labeling possibility* for the test instance. An example is the Bayes classifier where a probability is reported for a test instance. As a convention, it will be assumed that higher values of the score imply a greater likelihood to belong to a particular class.

The following sections will discuss methods for quantifying accuracy in both scenarios.

10.9.2.1 Output as Class Labels

When the output is presented in the form of class labels, the ground-truth labels are compared to the predicted labels to yield the following measures:

1. *Accuracy:* The accuracy is the fraction of test instances in which the predicted value matches the ground-truth value.

2. *Cost-sensitive accuracy:* Not all classes are equally important in all scenarios while comparing the accuracy. This is particularly important in imbalanced class problems, which will be discussed in more detail in the next chapter. For example, consider an application in which it is desirable to classify tumors as *malignant* or *nonmalignant* where the former is much rarer than the latter. In such cases, the misclassification of the former is often much less desirable than misclassification of the latter. This is frequently quantified by imposing differential costs $c_1 \ldots c_k$ on the misclassification of the different classes. Let $n_1 \ldots n_k$ be the number of test instances belonging to each class. Furthermore, let $a_1 \ldots a_k$ be the accuracies (expressed as a fraction) on the subset of test instances belonging to each class. Then, the overall accuracy A can be computed as a weighted combination of the accuracies over the individual labels.

$$A = \frac{\sum_{i=1}^{k} c_i n_i a_i}{\sum_{i=1}^{k} c_i n_i} \tag{10.77}$$

The cost sensitive accuracy is the same as the unweighted accuracy when all costs $c_1 \ldots c_k$ are the same.

Aside from the accuracy, the statistical robustness of a model is also an important issue. For example, if two classifiers are trained over a small number of test instances and compared, the difference in accuracy may be a result of random variations, rather than a truly *statistically significant* difference between the two classifiers. Therefore, it is important to design statistical measures to quantify the specific advantage of one classifier over the other.

Most statistical methodologies such as holdout, bootstrap, and cross-validation use $b > 1$ different randomly sampled rounds to obtain multiple estimates of the accuracy. For the purpose of discussion, let us assume that b different rounds (i.e., b different m-way partitions) of cross-validation are used. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two models. Let $A_{i,1}$ and $A_{i,2}$ be the respective accuracies of the models $\mathcal{M}_1$ and $\mathcal{M}_2$ on the partitioning created by the ith round of cross-validation. The corresponding difference in accuracy is $\delta a_i = A_{i,1} - A_{i,2}$. This results in b estimates $\delta a_1 \ldots \delta a_b$. Note that δa_i might be either positive or negative, depending on which classifier provides superior performance on a particular round of cross-validation. Let the average difference in accuracy between the two classifiers be ΔA.

$$\Delta A = \frac{\sum_{i=1}^{b} \delta a_i}{b} \tag{10.78}$$

The standard deviation σ of the difference in accuracy may be estimated as follows:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{b} (\delta a_i - \Delta A)^2}{b-1}}. \tag{10.79}$$

Note that the sign of ΔA tells us which classifier is better than the other. For example, if $\Delta A > 0$ then model $\mathcal{M}_1$ has higher average accuracy than $\mathcal{M}_2$. In such a case, it is desired to determine a statistical measure of the confidence (or, a probability value) that $\mathcal{M}_1$ is truly better than $\mathcal{M}_2$.

The idea here is to assume that the different samples $\delta a_1 \dots \delta a_b$ are sampled from a normal distribution. Therefore, the estimated mean and standard deviations of this distribution are given by ΔA and σ, respectively. The standard deviation of the estimated mean ΔA of b samples is therefore $\sigma/\sqrt{b}$ according to the central-limit theorem. Then, the number of standard deviations s by which ΔA is different from the break-even accuracy difference of 0 is as follows:

$$s = \frac{\sqrt{b}|\Delta A - 0|}{\sigma}. \tag{10.80}$$

When b is large, the standard normal distribution with zero mean and unit variance can be used to quantify the probability that one classifier is truly better than the other. The probability in any one of the symmetric tails of the standard normal distribution, more than s standard deviations away from the mean, provides the probability that this variation is not significant, and it might be a result of chance. This probability is subtracted from 1 to determine the confidence that one classifier is truly better than the other.

It is often computationally expensive to use large values of b. In such cases, it is no longer possible to estimate the standard deviation σ robustly with the use of a small number b of samples. To adjust for this, the Student's t-distribution with $(b-1)$ degrees of freedom is used instead of the normal distribution. This distribution is very similar to the normal distribution, except that it has a heavier tail to account for the greater estimation uncertainty. In fact, for large values of b, the t-distribution with $(b-1)$ degrees of freedom converges to the normal distribution.

10.9.2.2 Output as Numerical Score

In many scenarios, the output of the classification algorithm is reported as a numerical score associated with each test instance and label value. In cases where the numerical score can be reasonably compared across test instances (e.g., the probability values returned by a Bayes classifier), it is possible to compare the different test instances in terms of their relative propensity to belong to a specific class. Such scenarios are more common when one of the classes of interest is rare. Therefore, for this scenario, it is more meaningful to use the binary class scenario where one of the classes is the positive class, and the other class is the negative class. The discussion below is similar to the discussion in Sect. 8.8.2 of Chap. 8 on external validity measures for outlier analysis. This similarity arises from the fact that outlier validation with class labels is identical to classifier evaluation.

The advantage of a numerical score is that it provides more flexibility in evaluating the overall trade-off between labeling a varying number of data points as positives. This is achieved by using a threshold on the numerical score for the positive class to define the binary label. If the threshold is selected too aggressively to minimize the number of declared positive class instances, then the algorithm will miss true-positive class instances (false negatives). On the other hand, if the threshold is chosen in a more relaxed way, this will lead to too many false positives. This leads to a trade-off between the false positives and false negatives. The problem is that the "correct" threshold to use is never known exactly in a real scenario. However, the entire trade-off curve can be quantified using a variety of measures, and two algorithms can be compared over the entire trade-off curve. Two examples of such curves are the *precision–recall* curve, and the *receiver operating characteristic (ROC)* curve.

For any given threshold t on the predicted positive-class score, the declared positive class set is denoted by $\mathcal{S}(t)$. As t changes, the size of $\mathcal{S}(t)$ changes as well. Let $\mathcal{G}$ represent the true set (ground-truth set) of positive instances in the data set. Then, for any given threshold t, the *precision* is defined as the percentage of *reported* positives that truly turn out to be positive.

$$Precision(t) = 100 * \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{S}(t)|}$$

The value of $Precision(t)$ is *not* necessarily monotonic in t because both the numerator and denominator may change with t differently. The *recall* is correspondingly defined as the percentage of *ground-truth* positives that have been reported as positives at threshold t.

$$Recall(t) = 100 * \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{G}|}$$

While a natural trade-off exists between precision and recall, this trade-off is not necessarily monotonic. One way of creating a single measure that summarizes both precision and recall is the F_1-measure, which is the harmonic mean between the precision and the recall.

$$F_1(t) = \frac{2 \cdot Precision(t) \cdot Recall(t)}{Precision(t) + Recall(t)} \tag{10.81}$$

While the $F_1(t)$ measure provides a better quantification than either precision or recall, it is still dependent on the threshold t, and is therefore still not a complete representation of the trade-off between precision and recall. It is possible to visually examine the entire trade-off between precision and recall by varying the value of t, and examining the trade-off between the two quantities, by plotting the precision versus the recall. As shown later with an example, the lack of monotonicity of the precision makes the results harder to intuitively interpret.

A second way of generating the trade-off in a more intuitive way is through the use of the ROC curve. The *true-positive rate*, which is the same as the recall, is defined as the percentage of ground-truth positives that have been predicted as positive instances at threshold t.

$$TPR(t) = Recall(t) = 100 * \frac{|\mathcal{S}(t) \cap \mathcal{G}|}{|\mathcal{G}|}$$

The false-positive rate $FPR(t)$ is the percentage of the falsely reported positives out of the ground-truth negatives. Therefore, for a data set $\mathcal{D}$ with ground-truth positives $\mathcal{G}$, this measure is defined as follows:

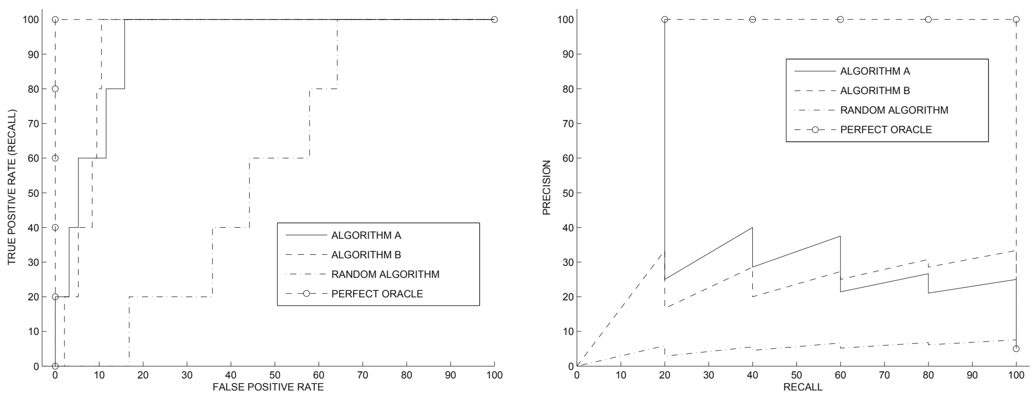
$$FPR(t) = 100 * \frac{|\mathcal{S}(t) - \mathcal{G}|}{|\mathcal{D} - \mathcal{G}|}. \tag{10.82}$$

The ROC curve is defined by plotting the $FPR(t)$ on the X-axis, and $TPR(t)$ on the Y-axis for varying values of t. Note that the end points of the ROC curve are always at $(0, 0)$ and $(100, 100)$, and a random method is expected to exhibit performance along the diagonal line connecting these points. The *lift* obtained above this diagonal line provides an idea of the accuracy of the approach. The area under the ROC curve provides a concrete quantitative evaluation of the effectiveness of a particular method.

To illustrate the insights gained from these different graphical representations, consider an example of a data set with 100 points from which 5 points belong to the positive class. Two algorithms A and B are applied to this data set that rank all data points from 1 to 100, with lower rank representing greater propensity to belong to the positive class. Thus, the true-positive rate and false-positive rate values can be generated by determining the ranks of the five ground-truth positive label points. In Table 10.2, some hypothetical ranks for the

Table 10.2: Rank of ground-truth positive instances

Algorithm	**Rank of positive class instances**
Algorithm A	1, 5, 8, 15, 20
Algorithm B	3, 7, 11, 13, 15
Random Algorithm	17, 36, 45, 59, 66
Perfect Oracle	1, 2, 3, 4, 5



(a) ROC (b) Precision-Recall

Figure 10.13: ROC curve and precision–recall curves

five ground-truth positive label instances have been illustrated for the different algorithms. In addition, ranks of the ground-truth positives for a random algorithm have been indicated. The random algorithm outputs a random score for each data point. Similarly, the ranks for a "perfect oracle" algorithm that ranks the correct top five points to belong to the positive class have also been illustrated in the table. The resulting ROC curves are illustrated in Fig. 10.13a. The corresponding precision–recall curves are illustrated in Fig. 10.13b. While the precision–recall curves are not quite as nicely interpretable as the ROC curves, it is easy to see that the *relative trends* between different algorithms, are the same in both cases. In general, ROC curves are used more frequently because of the ease in interpreting the quality of the algorithm with respect to a random classifier.

What do these curves really tell us? For cases in which one curve strictly dominates another, it is clear that the algorithm for the former curve is superior. For example, it is immediately evident that the oracle algorithm is superior to all algorithms, and the random algorithm is inferior to all the other algorithms. On the other hand, algorithms A and B show domination at different parts of the ROC curve. In such cases, it is hard to say that one algorithm is strictly superior. From Table 10.2, it is clear that Algorithm A ranks three of the correct positive instances very highly, but the remaining two positive instances are ranked poorly. In the case of Algorithm B, the highest ranked positive instances are not as well ranked as Algorithm A, though all five positive instances are determined much earlier in terms of rank threshold. Correspondingly, Algorithm A dominates on the earlier part of the ROC curve, whereas Algorithm B dominates on the later part. It is possible to use the area under the ROC curve as a proxy for the overall effectiveness of the algorithm.

10.10 Summary

The problem of data classification can be considered a supervised version of data clustering, in which a predefined set of groups is provided to a learner. This predefined set of groups is used for training the classifier to categorize unseen test examples into groups. A wide variety of models have been proposed for data classification.

Decision trees create a hierarchical model of the training data. For each test instance, the optimal path in the tree is used to classify unseen test instances. Each path in the tree can be viewed as a rule that is used to classify unseen test instances. Rule-based classifiers can be viewed as a generalization of decision trees, in which the classifier is not necessarily restricted to characterizing the data in a hierarchical way. Therefore, multiple conflicting rules can be used to cover the same training or test instance. Probabilistic classifiers map feature values to unseen test instances with probabilities. The naive Bayes rule or a logistic function may be used for effective estimation of probabilities. SVMs and neural networks are two forms of linear classifiers. The objective functions that are optimized are different. In the case of SVMs, the maximum margin principle is used, whereas for neural networks, the least squares error of prediction is approximately optimized. Instance-based learning methods are classifiers that delay learning to classification time as opposed to eager learners that construct the classification models up front. The simplest form of instance-based learning is the nearest-neighbor classifier. Many complex variations are possible by using different types of distance functions and locality-centric models.

Classifier evaluation is important for testing the relative effectiveness of different training models. Numerous models such as holdout, stratified sampling, bootstrap, and cross-validation have been proposed in the literature. Classifier evaluation can be performed either in the context of label assignment or numerical scoring. For label assignment, either the accuracy or the cost-sensitive accuracy may be used. For numerical scoring, the ROC curve is used to quantify the trade-off between the true-positive and false-positive rates.

10.11 Bibliographic Notes

The problem of data classification has been studied extensively by the data mining, machine learning, and pattern recognition communities. A number of books on these topics are available from these different communities [33, 95, 189, 256, 389]. Two surveys on the topic of data classification may be found in [286, 330]. A recent book [33] contains surveys on various aspects of data classification.

Feature selection is an important problem in data classification, to ensure the modeling algorithm is not confused by noise in the training data. Two books on feature selection may be found in [360, 366]. Fisher's discriminant analysis was first proposed in [207], although a slightly different variant with the assumption of normally distributed data used in linear discriminant analysis [379]. The most well-known decision tree algorithms include *ID3* [431], *C4.5* [430], and *CART* [110]. Decision tree methods are also used in the context of multivariate splits [116], though these methods are computationally more challenging. Surveys on decision tree algorithms may be found in [121, 393, 398]. Decision trees can be converted into rule-based classifiers where the rules are mutually exclusive. For example, the *C4.5* method has also been extended to the *C4.5rules* algorithm [430]. Other popular rule-based systems include *AQ* [386], *CN2* [177], and *RIPPER* [178]. Much of the discussion in this chapter was based on these algorithms. Popular associative classification algorithms include *CBA* [358], *CPAR* [529], and *CMAR* [349]. Methods for classification with discriminative patterns are discussed in [149]. A recent overview discussion of pattern-based classification algorithms may be found in [115]. The naive Bayes classifier has been discussed in detail in [187, 333, 344]. The work in [344] is particularly notable, in that it provides an understanding and justification of the naive Bayes assumption. A brief discussion of logistic regression models may be found in Chap. 3 of [33]. A more detailed discussion may be found in [275].

Numerous books are available on the topic of SVMs [155, 449, 478, 494]. An excellent tutorial on SVMs may be found in [124]. A detailed discussion of the Lagrangian relaxation technique for solving the resulting quadratic optimization problem may be found in [485]. It has been pointed out [133] that the advantages of the primal approach in SVMs seem to have been largely overlooked in the literature. It is sometimes mistakenly understood that the kernel trick can only be applied to the dual; the trick can be applied to the primal formulation as well [133]. A discussion of kernel methods for SVMs may be found in [451]. Other applications of kernels, such as nonlinear k-means and nonlinear PCA, are discussed in [173, 450]. The perceptron algorithm was due to Rosenblatt [439]. Neural networks are discussed in detail in several books [96, 260]. The back-propagation algorithm is described in detail in these books. The earliest work on instance-based classification was discussed in [167]. The method was subsequently extended to symbolic attributes [166]. Two surveys on instance-based classification may be found in [14, 183]. Local methods for nearest-neighbor classification are discussed in [216, 255]. Generalized instance-based learning methods have been studied in the context of decision trees [217], rule-based methods [347], Bayes methods [214], SVMs [105, 544], and neural networks [97, 209, 281]. Methods for classifier evaluation are discussed in [256].

10.12 Exercises

1. Compute the Gini index for the entire data set of Table 10.1, with respect to the two classes. Compute the Gini index for the portion of the data set with age at least 50.

2. Repeat the computation of the previous exercise with the use of the entropy criterion.

3. Show how to construct a (possibly overfitting) rule-based classifier that always exhibits 100 % accuracy on the training data. Assume that the feature variables of no two training instances are identical.

4. Design a univariate decision tree with a soft maximum-margin split criterion borrowed from SVMs. Suppose that this decision tree is generalized to the multivariate case. How does the resulting decision boundary compare with SVMs? Which classifier can handle a larger variety of data sets more accurately?

5. Discuss the advantages of a rule-based classifier over a decision tree.

6. Show that an SVM is a special case of a rule-based classifier. Design a rule-based classifier that uses SVMs to create an ordered list of rules.

7. Implement an associative classifier in which only maximal patterns are used for classification, and the majority consequent label of rules fired, is reported as the label of the test instance.

8. Suppose that you had d-dimensional numeric training data, in which it was known that the probability density of d-dimensional data instance $\overline{X}$ in each class i is proportional

to $e^{-||\overline{X}-\overline{\mu_i}||_1}$, where $||\cdot||_1$ is the Manhattan distance, and $\overline{\mu_i}$ is known for each class. How would you implement the Bayes classifier in this case? How would your answer change if $\overline{\mu_i}$ is unknown?

9. Explain the relationship of mutual exclusiveness and exhaustiveness of a rule set, to the need to order the rule set, or the need to set a class as the default class.

10. Consider the rules $Age > 40 \Rightarrow Donor$ and $Age \leq 50 \Rightarrow \neg Donor$. Are these two rules mutually exclusive? Are these two rules exhaustive?

11. For the example of Table 10.1, determine the prior probability of each class. Determine the conditional probability of each class for cases where the Age is at least 50.

12. Implement the naive Bayes classifier.

13. For the example of Table 10.1, provide a single linear separating hyperplane. Is this separating hyperplane unique?

14. Consider a data set containing four points located at the corners of the square. The two points on one diagonal belong to one class, and the two points on the other diagonal belong to the other class. Is this data set linearly separable? Provide a proof.

15. Provide a systematic way to determine whether two classes in a labeled data set are linearly separable.

16. For the soft SVM formulation with hinge loss, show that:

(a) The weight vector is given by the same relationship $\overline{W} = \sum_{i=1}^{n} \lambda_i y_i \overline{X_i}$, as for hard SVMs.

(b) The condition $\sum_{i=1}^{n} \lambda_i y_i = 0$ holds as in hard SVMs.

(c) The Lagrangian multipliers satisfy $\lambda_i \leq C$.

(d) The Lagrangian dual is identical to that of hard SVMs.

17. Show that it is possible to omit the bias parameter b from the decision boundary of SVMs by suitably preprocessing the data set. In other words, the decision boundary is now $\overline{W} \cdot \overline{X} = 0$. What is the impact of eliminating the bias parameter on the gradient ascent approach for Lagrangian dual optimization in SVMs?

18. Show that an $n \times d$ data set can be mean-centered by premultiplying it with the $n \times n$ matrix $(I - U/n)$, where U is a unit matrix of all ones. Show that an $n \times n$ kernel matrix K can be adjusted for mean centering of the data in the transformed space by adjusting it to $K' = (I - U/n)K(I - U/n)$.

19. Consider two classifiers A and B. On one data set, a 10-fold cross validation shows that classifier A is better than B by 3 %, with a standard deviation of 7 % over 100 different folds. On the other data set, classifier B is better than classifier A by 1 %, with a standard deviation of 0.1 % over 100 different folds. Which classifier would you prefer on the basis of this evidence, and why?

20. Provide a nonlinear transformation which would make the data set of Exercise 14 linearly separable.

21. Let S_w and S_b be defined according to Sect. 10.2.1.3 for the binary class problem. Let the fractional presence of the two classes be p_0 and p_1, respectively. Show that $S_w + p_0 p_1 S_b$ is equal to the covariance matrix of the data set.