RIPPER Algorithm

To illustrate the direct method, we consider a widely used rule induction algorithm called RIPPER. This algorithm scales almost linearly with the number of training examples and is particularly suited for building models from data sets with imbalanced class distributions. RIPPER also works well with noisy data sets because it uses a validation set to prevent model overfitting.

For two-class problems, RIPPER chooses the majority class as its default class and learns the rules for detecting the minority class. For multiclass problems, the classes are ordered according to their frequencies. Let $(y_1, y_2, \ldots, y_c)$ be the ordered classes, where y_1 is the least frequent class and y_c is the most frequent class. During the first iteration, instances that belong to y_1 are labeled as positive examples, while those that belong to other classes are labeled as negative examples. The sequential covering method is used to generate rules that discriminate between the positive and negative examples. Next, RIPPER extracts rules that distinguish y_2 from other remaining classes. This process is repeated until we are left with y_c, which is designated as the default class.

Rule Growing RIPPER employs a general-to-specific strategy to grow a rule and the FOIL's information gain measure to choose the best conjunct to be added into the rule antecedent. It stops adding conjuncts when the rule starts covering negative examples. The new rule is then pruned based on its performance on the validation set. The following metric is computed to determine whether pruning is needed: $(p-n)/(p+n)$, where p (n) is the number of positive (negative) examples in the validation set covered by the rule. This metric is monotonically related to the rule's accuracy on the validation set. If the metric improves after pruning, then the conjunct is removed. Pruning is done starting from the last conjunct added to the rule. For example, given a rule $ABCD \longrightarrow y$, RIPPER checks whether D should be pruned first, followed by CD, BCD, etc. While the original rule covers only positive examples, the pruned rule may cover some of the negative examples in the training set.

Building the Rule Set After generating a rule, all the positive and negative examples covered by the rule are eliminated. The rule is then added into the rule set as long as it does not violate the stopping condition, which is based on the minimum description length principle. If the new rule increases the total description length of the rule set by at least d bits, then RIPPER stops adding rules into its rule set (by default, d is chosen to be 64 bits). Another stopping condition used by RIPPER is that the error rate of the rule on the validation set must not exceed 50%.

RIPPER also performs additional optimization steps to determine whether some of the existing rules in the rule set can be replaced by better alternative rules. Readers who are interested in the details of the optimization method may refer to the reference cited at the end of this chapter.

5.1.5 Indirect Methods for Rule Extraction

This section presents a method for generating a rule set from a decision tree. In principle, every path from the root node to the leaf node of a decision tree can be expressed as a classification rule. The test conditions encountered along the path form the conjuncts of the rule antecedent, while the class label at the leaf node is assigned to the rule consequent. Figure 5.5 shows an example of a rule set generated from a decision tree. Notice that the rule set is exhaustive and contains mutually exclusive rules. However, some of the rules can be simplified as shown in the next example.

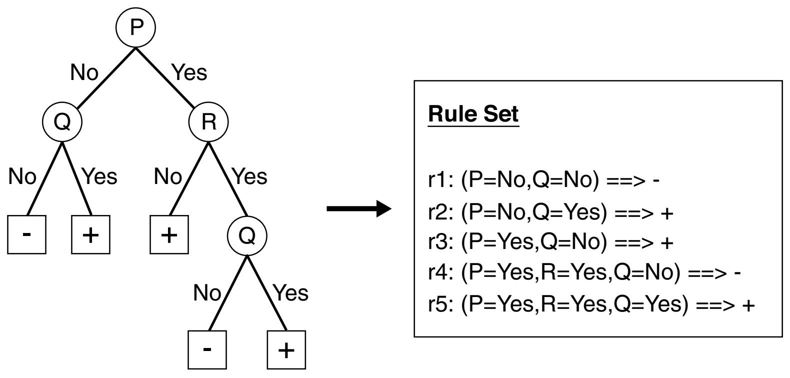


Figure 5.5. Converting a decision tree into classification rules.

Example 5.2. Consider the following three rules from Figure 5.5:

$r2$: (P = No) $\wedge$ (Q = Yes) $\longrightarrow$ +
$r3$: (P = Yes) $\wedge$ (R = No) $\longrightarrow$ +
$r5$: (P = Yes) $\wedge$ (R = Yes) $\wedge$ (Q = Yes) $\longrightarrow$ +

Observe that the rule set always predicts a positive class when the value of Q is Yes. Therefore, we may simplify the rules as follows:

$r2'$: (Q = Yes) $\longrightarrow$ +
$r3$: (P = Yes) $\wedge$ (R = No) $\longrightarrow$ +.

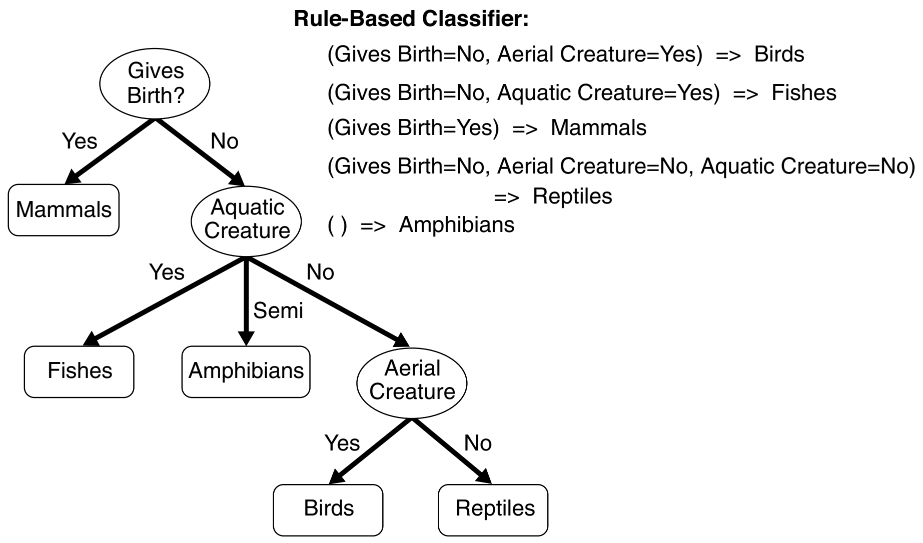

Rule-Based Classifier:
(Gives Birth=No, Aerial Creature=Yes) => Birds
(Gives Birth=No, Aquatic Creature=Yes) => Fishes
(Gives Birth=Yes) => Mammals
(Gives Birth=No, Aerial Creature=No, Aquatic Creature=No) => Reptiles
( ) => Amphibians
Gives Birth?
Yes
No
Mammals
Aquatic Creature
Yes
Semi
No
Fishes
Amphibians
Aerial Creature
Yes
No
Birds
Reptiles

Figure 5.6. Classification rules extracted from a decision tree for the vertebrate classification problem.

r_3 is retained to cover the remaining instances of the positive class. Although the rules obtained after simplification are no longer mutually exclusive, they are less complex and are easier to interpret. ■

In the following, we describe an approach used by the C4.5rules algorithm to generate a rule set from a decision tree. Figure 5.6 shows the decision tree and resulting classification rules obtained for the data set given in Table 5.2.

Rule Generation Classification rules are extracted for every path from the root to one of the leaf nodes in the decision tree. Given a classification rule $r : A \longrightarrow y$, we consider a simplified rule, $r' : A' \longrightarrow y$, where A' is obtained by removing one of the conjuncts in A. The simplified rule with the lowest pessimistic error rate is retained provided its error rate is less than that of the original rule. The rule-pruning step is repeated until the pessimistic error of the rule cannot be improved further. Because some of the rules may become identical after pruning, the duplicate rules must be discarded.

Rule Ordering After generating the rule set, C4.5rules uses the class-based ordering scheme to order the extracted rules. Rules that predict the same class are grouped together into the same subset. The total description length for each subset is computed, and the classes are arranged in increasing order of their total description length. The class that has the smallest description length is given the highest priority because it is expected to contain the best set of rules. The total description length for a class is given by $L_{\text{exception}} + g \times L_{\text{model}}$, where $L_{\text{exception}}$ is the number of bits needed to encode the misclassified examples, L_{model} is the number of bits needed to encode the model, and g is a tuning parameter whose default value is 0.5. The tuning parameter depends on the number of redundant attributes present in the model. The value of the tuning parameter is small if the model contains many redundant attributes.

5.1.6 Characteristics of Rule-Based Classifiers

A rule-based classifier has the following characteristics:

- The expressiveness of a rule set is almost equivalent to that of a decision tree because a decision tree can be represented by a set of mutually exclusive and exhaustive rules. Both rule-based and decision tree classifiers create rectilinear partitions of the attribute space and assign a class to each partition. Nevertheless, if the rule-based classifier allows multiple rules to be triggered for a given record, then a more complex decision boundary can be constructed.

- Rule-based classifiers are generally used to produce descriptive models that are easier to interpret, but gives comparable performance to the decision tree classifier.

- The class-based ordering approach adopted by many rule-based classifiers (such as RIPPER) is well suited for handling data sets with imbalanced class distributions.

5.2 Nearest-Neighbor classifiers

The classification framework shown in Figure 4.3 involves a two-step process: (1) an inductive step for constructing a classification model from data, and (2) a deductive step for applying the model to test examples. Decision tree and rule-based classifiers are examples of **eager learners** because they are designed to learn a model that maps the input attributes to the class label as soon as the training data becomes available. An opposite strategy would be to delay the process of modeling the training data until it is needed to classify the test examples. Techniques that employ this strategy are known as **lazy learners**. An example of a lazy learner is the **Rote classifier**, which memorizes the entire training data and performs classification only if the attributes of a test instance match one of the training examples exactly. An obvious drawback of

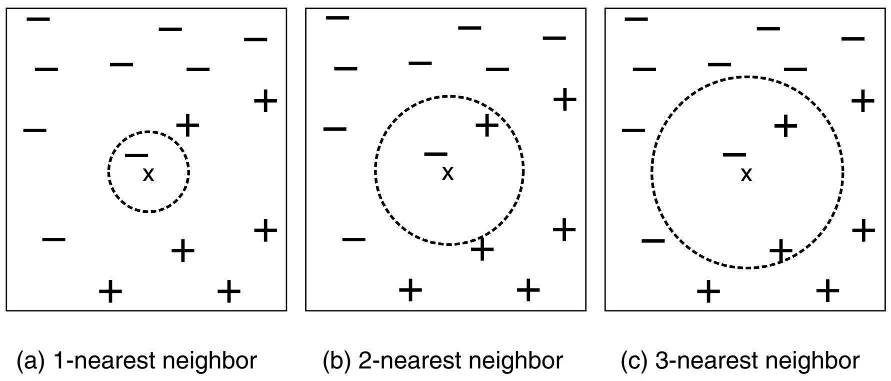


Figure 5.7. The 1-, 2-, and 3-nearest neighbors of an instance.

this approach is that some test records may not be classified because they do not match any training example.

One way to make this approach more flexible is to find all the training examples that are relatively similar to the attributes of the test example. These examples, which are known as **nearest neighbors**, can be used to determine the class label of the test example. The justification for using nearest neighbors is best exemplified by the following saying: *"If it walks like a duck, quacks like a duck, and looks like a duck, then it's probably a duck."* A nearest-neighbor classifier represents each example as a data point in a d-dimensional space, where d is the number of attributes. Given a test example, we compute its proximity to the rest of the data points in the training set, using one of the proximity measures described in Section 2.4 on page 65. The k-nearest neighbors of a given example z refer to the k points that are closest to z.

Figure 5.7 illustrates the 1-, 2-, and 3-nearest neighbors of a data point located at the center of each circle. The data point is classified based on the class labels of its neighbors. In the case where the neighbors have more than one label, the data point is assigned to the majority class of its nearest neighbors. In Figure 5.7(a), the 1-nearest neighbor of the data point is a negative example. Therefore the data point is assigned to the negative class. If the number of nearest neighbors is three, as shown in Figure 5.7(c), then the neighborhood contains two positive examples and one negative example. Using the majority voting scheme, the data point is assigned to the positive class. In the case where there is a tie between the classes (see Figure 5.7(b)), we may randomly choose one of them to classify the data point.

The preceding discussion underscores the importance of choosing the right value for k. If k is too small, then the nearest-neighbor classifier may be

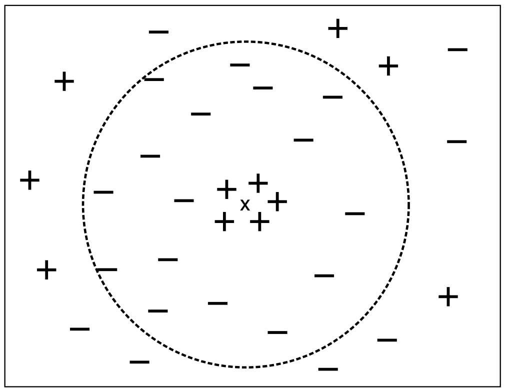


Figure 5.8. k-nearest neighbor classification with large k.

susceptible to overfitting because of noise in the training data. On the other hand, if k is too large, the nearest-neighbor classifier may misclassify the test instance because its list of nearest neighbors may include data points that are located far away from its neighborhood (see Figure 5.8).

5.2.1 Algorithm

A high-level summary of the nearest-neighbor classification method is given in Algorithm 5.2. The algorithm computes the distance (or similarity) between each test example $z = (\mathbf{x}', y')$ and all the training examples $(\mathbf{x}, y) \in D$ to determine its nearest-neighbor list, D_z. Such computation can be costly if the number of training examples is large. However, efficient indexing techniques are available to reduce the amount of computations needed to find the nearest neighbors of a test example.

Algorithm 5.2 The k-nearest neighbor classification algorithm.

```
1: Let $k$ be the number of nearest neighbors and $D$ be the set of training examples.
2: for each test example $z = (\mathbf{x}', y')$ do
3:     Compute $d(\mathbf{x}', \mathbf{x})$, the distance between $z$ and every example, $(\mathbf{x}, y) \in D$.
4:     Select $D_z \subseteq D$, the set of $k$ closest training examples to $z$.
5:     $y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i)$
6: end for
```

Once the nearest-neighbor list is obtained, the test example is classified based on the majority class of its nearest neighbors:

$$\text{Majority Voting: } y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i), \tag{5.7}$$

where v is a class label, y_i is the class label for one of the nearest neighbors, and $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

In the majority voting approach, every neighbor has the same impact on the classification. This makes the algorithm sensitive to the choice of k, as shown in Figure 5.7. One way to reduce the impact of k is to weight the influence of each nearest neighbor $\mathbf{x}_i$ according to its distance: $w_i = 1/d(\mathbf{x}', \mathbf{x}_i)^2$. As a result, training examples that are located far away from z have a weaker impact on the classification compared to those that are located close to z. Using the distance-weighted voting scheme, the class label can be determined as follows:

$$\text{Distance-Weighted Voting: } y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \times I(v = y_i). \tag{5.8}$$

5.2.2 Characteristics of Nearest-Neighbor Classifiers

The characteristics of the nearest-neighbor classifier are summarized below:

- Nearest-neighbor classification is part of a more general technique known as instance-based learning, which uses specific training instances to make predictions without having to maintain an abstraction (or model) derived from data. Instance-based learning algorithms require a proximity measure to determine the similarity or distance between instances and a classification function that returns the predicted class of a test instance based on its proximity to other instances.

- Lazy learners such as nearest-neighbor classifiers do not require model building. However, classifying a test example can be quite expensive because we need to compute the proximity values individually between the test and training examples. In contrast, eager learners often spend the bulk of their computing resources for model building. Once a model has been built, classifying a test example is extremely fast.

- Nearest-neighbor classifiers make their predictions based on local information, whereas decision tree and rule-based classifiers attempt to find a global model that fits the entire input space. Because the classification decisions are made locally, nearest-neighbor classifiers (with small values of k) are quite susceptible to noise.

- Nearest-neighbor classifiers can produce arbitrarily shaped decision boundaries. Such boundaries provide a more flexible model representation compared to decision tree and rule-based classifiers that are often constrained to rectilinear decision boundaries. The decision boundaries of nearest-neighbor classifiers also have high variability because they depend on the composition of training examples. Increasing the number of nearest neighbors may reduce such variability.

- Nearest-neighbor classifiers can produce wrong predictions unless the appropriate proximity measure and data preprocessing steps are taken. For example, suppose we want to classify a group of people based on attributes such as height (measured in meters) and weight (measured in pounds). The height attribute has a low variability, ranging from 1.5 m to 1.85 m, whereas the weight attribute may vary from 90 lb. to 250 lb. If the scale of the attributes are not taken into consideration, the proximity measure may be dominated by differences in the weights of a person.

5.3 Bayesian Classifiers

In many applications the relationship between the attribute set and the class variable is non-deterministic. In other words, the class label of a test record cannot be predicted with certainty even though its attribute set is identical to some of the training examples. This situation may arise because of noisy data or the presence of certain confounding factors that affect classification but are not included in the analysis. For example, consider the task of predicting whether a person is at risk for heart disease based on the person's diet and workout frequency. Although most people who eat healthily and exercise regularly have less chance of developing heart disease, they may still do so because of other factors such as heredity, excessive smoking, and alcohol abuse. Determining whether a person's diet is healthy or the workout frequency is sufficient is also subject to interpretation, which in turn may introduce uncertainties into the learning problem.

This section presents an approach for modeling probabilistic relationships between the attribute set and the class variable. The section begins with an introduction to the **Bayes theorem**, a statistical principle for combining prior knowledge of the classes with new evidence gathered from data. The use of the Bayes theorem for solving classification problems will be explained, followed by a description of two implementations of Bayesian classifiers: naïve Bayes and the Bayesian belief network.

5.3.1 Bayes Theorem

Consider a football game between two rival teams: Team 0 and Team 1. Suppose Team 0 wins 65% *of the time and Team 1 wins the remaining matches. Among the games won by Team 0, only* 30% *of them come from playing on Team 1's football field. On the other hand,* 75% *of the victories for Team 1 are obtained while playing at home. If Team 1 is to host the next match between the two teams, which team will most likely emerge as the winner?*

This question can be answered by using the well-known Bayes theorem. For completeness, we begin with some basic definitions from probability theory. Readers who are unfamiliar with concepts in probability may refer to Appendix C for a brief review of this topic.

Let X and Y be a pair of random variables. Their joint probability, $P(X = x, Y = y)$, refers to the probability that variable X will take on the value x and variable Y will take on the value y. A conditional probability is the probability that a random variable will take on a particular value given that the outcome for another random variable is known. For example, the conditional probability $P(Y = y|X = x)$ refers to the probability that the variable Y will take on the value y, given that the variable X is observed to have the value x. The joint and conditional probabilities for X and Y are related in the following way:

$$P(X, Y) = P(Y|X) \times P(X) = P(X|Y) \times P(Y). \tag{5.9}$$

Rearranging the last two expressions in Equation 5.9 leads to the following formula, known as the Bayes theorem:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}. \tag{5.10}$$

The Bayes theorem can be used to solve the prediction problem stated at the beginning of this section. For notational convenience, let X be the random variable that represents the team hosting the match and Y be the random variable that represents the winner of the match. Both X and Y can take on values from the set $\{0, 1\}$. We can summarize the information given in the problem as follows:

Probability Team 0 wins is $P(Y = 0) = 0.65$.
Probability Team 1 wins is $P(Y = 1) = 1 - P(Y = 0) = 0.35$.
Probability Team 1 hosted the match it won is $P(X = 1|Y = 1) = 0.75$.
Probability Team 1 hosted the match won by Team 0 is $P(X = 1|Y = 0) = 0.3$.

Our objective is to compute $P(Y = 1|X = 1)$, which is the conditional probability that Team 1 wins the next match it will be hosting, and compares it against $P(Y = 0|X = 1)$. Using the Bayes theorem, we obtain

$$\begin{aligned}
P(Y = 1|X = 1) &= \frac{P(X = 1|Y = 1) \times P(Y = 1)}{P(X = 1)} \\
&= \frac{P(X = 1|Y = 1) \times P(Y = 1)}{P(X = 1, Y = 1) + P(X = 1, Y = 0)} \\
&= \frac{P(X = 1|Y = 1) \times P(Y = 1)}{P(X = 1|Y = 1)P(Y = 1) + P(X = 1|Y = 0)P(Y = 0)} \\
&= \frac{0.75 \times 0.35}{0.75 \times 0.35 + 0.3 \times 0.65} \\
&= 0.5738,
\end{aligned}$$

where the law of total probability (see Equation C.5 on page 722) was applied in the second line. Furthermore, $P(Y = 0|X = 1) = 1 - P(Y = 1|X = 1) = 0.4262$. Since $P(Y = 1|X = 1) > P(Y = 0|X = 1)$, Team 1 has a better chance than Team 0 of winning the next match.

5.3.2 Using the Bayes Theorem for Classification

Before describing how the Bayes theorem can be used for classification, let us formalize the classification problem from a statistical perspective. Let $\mathbf{X}$ denote the attribute set and Y denote the class variable. If the class variable has a non-deterministic relationship with the attributes, then we can treat $\mathbf{X}$ and Y as random variables and capture their relationship probabilistically using $P(Y|\mathbf{X})$. This conditional probability is also known as the **posterior probability** for Y, as opposed to its **prior probability**, $P(Y)$.

During the training phase, we need to learn the posterior probabilities $P(Y|\mathbf{X})$ for every combination of $\mathbf{X}$ and Y based on information gathered from the training data. By knowing these probabilities, a test record $\mathbf{X}'$ can be classified by finding the class Y' that maximizes the posterior probability,

$P(Y'|\mathbf{X}')$. To illustrate this approach, consider the task of predicting whether a loan borrower will default on their payments. Figure 5.9 shows a training set with the following attributes: `Home Owner`, `Marital Status`, and `Annual Income`. Loan borrowers who defaulted on their payments are classified as `Yes`, while those who repaid their loans are classified as `No`.

	binary	categorical	continuous	class
Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Figure 5.9. Training set for predicting the loan default problem.

Suppose we are given a test record with the following attribute set: $\mathbf{X} =$ (`Home Owner` = `No`, `Marital Status` = `Married`, `Annual Income` = \$120K). To classify the record, we need to compute the posterior probabilities $P(\texttt{Yes}|\mathbf{X})$ and $P(\texttt{No}|\mathbf{X})$ based on information available in the training data. If $P(\texttt{Yes}|\mathbf{X}) > P(\texttt{No}|\mathbf{X})$, then the record is classified as `Yes`; otherwise, it is classified as `No`.

Estimating the posterior probabilities accurately for every possible combination of class label and attribute value is a difficult problem because it requires a very large training set, even for a moderate number of attributes. The Bayes theorem is useful because it allows us to express the posterior probability in terms of the prior probability $P(Y)$, the **class-conditional** probability $P(\mathbf{X}|Y)$, and the evidence, $P(\mathbf{X})$:

$$P(Y|\mathbf{X}) = \frac{P(\mathbf{X}|Y) \times P(Y)}{P(\mathbf{X})}. \tag{5.11}$$

When comparing the posterior probabilities for different values of Y, the denominator term, $P(\mathbf{X})$, is always constant, and thus, can be ignored. The prior probability $P(Y)$ can be easily estimated from the training set by computing the fraction of training records that belong to each class. To estimate the class-conditional probabilities $P(\mathbf{X}|Y)$, we present two implementations of Bayesian classification methods: the naïve Bayes classifier and the Bayesian belief network. These implementations are described in Sections 5.3.3 and 5.3.5, respectively.

5.3.3 Naïve Bayes Classifier

A naïve Bayes classifier estimates the class-conditional probability by assuming that the attributes are conditionally independent, given the class label y. The conditional independence assumption can be formally stated as follows:

$$P(\mathbf{X}|Y = y) = \prod_{i=1}^{d} P(X_i|Y = y), \tag{5.12}$$

where each attribute set $\mathbf{X} = \{X_1, X_2, \ldots, X_d\}$ consists of d attributes.

Conditional Independence

Before delving into the details of how a naïve Bayes classifier works, let us examine the notion of conditional independence. Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ denote three sets of random variables. The variables in $\mathbf{X}$ are said to be conditionally independent of $\mathbf{Y}$, given $\mathbf{Z}$, if the following condition holds:

$$P(\mathbf{X}|\mathbf{Y}, \mathbf{Z}) = P(\mathbf{X}|\mathbf{Z}). \tag{5.13}$$

An example of conditional independence is the relationship between a person's arm length and his or her reading skills. One might observe that people with longer arms tend to have higher levels of reading skills. This relationship can be explained by the presence of a confounding factor, which is age. A young child tends to have short arms and lacks the reading skills of an adult. If the age of a person is fixed, then the observed relationship between arm length and reading skills disappears. Thus, we can conclude that arm length and reading skills are conditionally independent when the age variable is fixed.

The conditional independence between **X** and **Y** can also be written into a form that looks similar to Equation 5.12:

$$\begin{aligned}P(\mathbf{X},\mathbf{Y}|\mathbf{Z}) &= \frac{P(\mathbf{X},\mathbf{Y},\mathbf{Z})}{P(\mathbf{Z})}\\ &= \frac{P(\mathbf{X},\mathbf{Y},\mathbf{Z})}{P(\mathbf{Y},\mathbf{Z})}\times\frac{P(\mathbf{Y},\mathbf{Z})}{P(\mathbf{Z})}\\ &= P(\mathbf{X}|\mathbf{Y},\mathbf{Z})\times P(\mathbf{Y}|\mathbf{Z})\\ &= P(\mathbf{X}|\mathbf{Z})\times P(\mathbf{Y}|\mathbf{Z}), \qquad (5.14)\end{aligned}$$

where Equation 5.13 was used to obtain the last line of Equation 5.14.

How a Naïve Bayes Classifier Works

With the conditional independence assumption, instead of computing the class-conditional probability for every combination of **X**, we only have to estimate the conditional probability of each X_i, given Y. The latter approach is more practical because it does not require a very large training set to obtain a good estimate of the probability.

To classify a test record, the naïve Bayes classifier computes the posterior probability for each class Y:

$$P(Y|\mathbf{X}) = \frac{P(Y)\prod_{i=1}^{d} P(X_i|Y)}{P(\mathbf{X})}. \qquad (5.15)$$

Since $P(\mathbf{X})$ is fixed for every Y, it is sufficient to choose the class that maximizes the numerator term, $P(Y)\prod_{i=1}^{d} P(X_i|Y)$. In the next two subsections, we describe several approaches for estimating the conditional probabilities $P(X_i|Y)$ for categorical and continuous attributes.

Estimating Conditional Probabilities for Categorical Attributes

For a categorical attribute X_i, the conditional probability $P(X_i = x_i|Y = y)$ is estimated according to the fraction of training instances in class y that take on a particular attribute value x_i. For example, in the training set given in Figure 5.9, three out of the seven people who repaid their loans also own a home. As a result, the conditional probability for $P($`Home Owner`$=$`Yes`$|$`No`$)$ is equal to 3/7. Similarly, the conditional probability for defaulted borrowers who are single is given by $P($`Marital Status` $=$ `Single`$|$`Yes`$) = 2/3$.

Estimating Conditional Probabilities for Continuous Attributes

There are two ways to estimate the class-conditional probabilities for continuous attributes in naïve Bayes classifiers:

1. We can discretize each continuous attribute and then replace the continuous attribute value with its corresponding discrete interval. This approach transforms the continuous attributes into ordinal attributes. The conditional probability $P(X_i|Y = y)$ is estimated by computing the fraction of training records belonging to class y that falls within the corresponding interval for X_i. The estimation error depends on the discretization strategy (as described in Section 2.3.6 on page 57), as well as the number of discrete intervals. If the number of intervals is too large, there are too few training records in each interval to provide a reliable estimate for $P(X_i|Y)$. On the other hand, if the number of intervals is too small, then some intervals may aggregate records from different classes and we may miss the correct decision boundary.

2. We can assume a certain form of probability distribution for the continuous variable and estimate the parameters of the distribution using the training data. A Gaussian distribution is usually chosen to represent the class-conditional probability for continuous attributes. The distribution is characterized by two parameters, its mean, μ, and variance, σ^2. For each class y_j, the class-conditional probability for attribute X_i is

$$P(X_i = x_i|Y = y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp^{-\frac{(x_i-\mu_{ij})^2}{2\sigma_{ij}^2}}. \tag{5.16}$$

The parameter μ_{ij} can be estimated based on the sample mean of X_i ($\overline{x}$) for all training records that belong to the class y_j. Similarly, σ_{ij}^2 can be estimated from the sample variance (s^2) of such training records. For example, consider the annual income attribute shown in Figure 5.9. The sample mean and variance for this attribute with respect to the class `No` are

$$\overline{x} = \frac{125 + 100 + 70 + \ldots + 75}{7} = 110$$
$$s^2 = \frac{(125 - 110)^2 + (100 - 110)^2 + \ldots + (75 - 110)^2}{7(6)} = 2975$$
$$s = \sqrt{2975} = 54.54.$$

Given a test record with taxable income equal to \$120K, we can compute its class-conditional probability as follows:

$$P(\texttt{Income=120}|\texttt{No}) = \frac{1}{\sqrt{2\pi}(54.54)} \exp^{-\frac{(120-110)^2}{2\times 2975}} = 0.0072.$$

Note that the preceding interpretation of class-conditional probability is somewhat misleading. The right-hand side of Equation 5.16 corresponds to a **probability density function**, $f(X_i; \mu_{ij}, \sigma_{ij})$. Since the function is continuous, the probability that the random variable X_i takes a particular value is zero. Instead, we should compute the conditional probability that X_i lies within some interval, x_i and $x_i + \epsilon$, where ϵ is a small constant:

$$\begin{aligned} P(x_i \leq X_i \leq x_i + \epsilon | Y = y_j) &= \int_{x_i}^{x_i+\epsilon} f(X_i; \mu_{ij}, \sigma_{ij}) dX_i \\ &\approx f(x_i; \mu_{ij}, \sigma_{ij}) \times \epsilon. \end{aligned} \quad (5.17)$$

Since ϵ appears as a constant multiplicative factor for each class, it cancels out when we normalize the posterior probability for $P(Y|\mathbf{X})$. Therefore, we can still apply Equation 5.16 to approximate the class-conditional probability $P(X_i|Y)$.

Example of the Naïve Bayes Classifier

Consider the data set shown in Figure 5.10(a). We can compute the class-conditional probability for each categorical attribute, along with the sample mean and variance for the continuous attribute using the methodology described in the previous subsections. These probabilities are summarized in Figure 5.10(b).

To predict the class label of a test record $\mathbf{X}$ = (`Home Owner`=`No`, `Marital Status` = `Married`, `Income` = \$120K), we need to compute the posterior probabilities $P(\texttt{No}|\mathbf{X})$ and $P(\texttt{Yes}|\mathbf{X})$. Recall from our earlier discussion that these posterior probabilities can be estimated by computing the product between the prior probability $P(Y)$ and the class-conditional probabilities $\prod_i P(X_i|Y)$, which corresponds to the numerator of the right-hand side term in Equation 5.15.

The prior probabilities of each class can be estimated by calculating the fraction of training records that belong to each class. Since there are three records that belong to the class `Yes` and seven records that belong to the class

Tid	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	**No**
2	No	Married	100K	**No**
3	No	Single	70K	**No**
4	Yes	Married	120K	**No**
5	No	Divorced	95K	**Yes**
6	No	Married	60K	**No**
7	Yes	Divorced	220K	**No**
8	No	Single	85K	**Yes**
9	No	Married	75K	**No**
10	No	Single	90K	**Yes**

(a)

P(Home Owner=Yes|No) = 3/7
P(Home Owner=No|No) = 4/7
P(Home Owner=Yes|Yes) = 0
P(Home Owner=No|Yes) = 1
P(Marital Status=Single|No) = 2/7
P(Marital Status=Divorced|No) = 1/7
P(Marital Status=Married|No) = 4/7
P(Marital Status=Single|Yes) = 2/3
P(Marital Status=Divorced|Yes) = 1/3
P(Marital Status=Married|Yes) = 0

For Annual Income:
If class=No: sample mean=110
sample variance=2975
If class=Yes: sample mean=90
sample variance=25

(b)

Figure 5.10. The naïve Bayes classifier for the loan classification problem.

No, $P(\texttt{Yes}) = 0.3$ and $P(\texttt{No}) = 0.7$. Using the information provided in Figure 5.10(b), the class-conditional probabilities can be computed as follows:

$$\begin{aligned} P(\mathbf{X}|\texttt{No}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{No}) \times P(\texttt{Status} = \texttt{Married}|\texttt{No}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{No}) \\ &= 4/7 \times 4/7 \times 0.0072 = 0.0024. \end{aligned}$$

$$\begin{aligned} P(\mathbf{X}|\texttt{Yes}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{Yes}) \times P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{Yes}) \\ &= 1 \times 0 \times 1.2 \times 10^{-9} = 0. \end{aligned}$$

Putting them together, the posterior probability for class No is $P(\texttt{No}|\mathbf{X}) = \alpha \times 7/10 \times 0.0024 = 0.0016\alpha$, where $\alpha = 1/P(\mathbf{X})$ is a constant term. Using a similar approach, we can show that the posterior probability for class Yes is zero because its class-conditional probability is zero. Since $P(\texttt{No}|\mathbf{X}) > P(\texttt{Yes}|\mathbf{X})$, the record is classified as No.

M-estimate of Conditional Probability

The preceding example illustrates a potential problem with estimating posterior probabilities from training data. If the class-conditional probability for one of the attributes is zero, then the overall posterior probability for the class vanishes. This approach of estimating class-conditional probabilities using simple fractions may seem too brittle, especially when there are few training examples available and the number of attributes is large.

In a more extreme case, if the training examples do not cover many of the attribute values, we may not be able to classify some of the test records. For example, if $P(\texttt{Marital Status} = \texttt{Divorced}|\texttt{No})$ is zero instead of 1/7, then a record with attribute set $\mathbf{X} = (\texttt{Home Owner} = \texttt{Yes}, \texttt{Marital Status} = \texttt{Divorced}, \texttt{Income} = \$120\text{K})$ has the following class-conditional probabilities:

$$P(\mathbf{X}|\texttt{No}) = 3/7 \times 0 \times 0.0072 = 0.$$
$$P(\mathbf{X}|\texttt{Yes}) = 0 \times 1/3 \times 1.2 \times 10^{-9} = 0.$$

The naïve Bayes classifier will not be able to classify the record. This problem can be addressed by using the m-estimate approach for estimating the conditional probabilities:

$$P(x_i|y_j) = \frac{n_c + mp}{n + m}, \tag{5.18}$$

where n is the total number of instances from class y_j, n_c is the number of training examples from class y_j that take on the value x_i, m is a parameter known as the equivalent sample size, and p is a user-specified parameter. If there is no training set available (i.e., $n = 0$), then $P(x_i|y_j) = p$. Therefore p can be regarded as the prior probability of observing the attribute value x_i among records with class y_j. The equivalent sample size determines the tradeoff between the prior probability p and the observed probability n_c/n.

In the example given in the previous section, the conditional probability $P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) = 0$ because none of the training records for the class has the particular attribute value. Using the m-estimate approach with $m = 3$ and $p = 1/3$, the conditional probability is no longer zero:

$$P(\texttt{Marital Status} = \texttt{Married}|\texttt{Yes}) = (0 + 3 \times 1/3)/(3 + 3) = 1/6.$$

If we assume $p = 1/3$ for all attributes of class `Yes` and $p = 2/3$ for all attributes of class `No`, then

$$\begin{aligned} P(\mathbf{X}|\texttt{No}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{No}) \times P(\texttt{Status} = \texttt{Married}|\texttt{No}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{No}) \\ &= 6/10 \times 6/10 \times 0.0072 = 0.0026. \end{aligned}$$

$$\begin{aligned} P(\mathbf{X}|\texttt{Yes}) &= P(\texttt{Home Owner} = \texttt{No}|\texttt{Yes}) \times P(\texttt{Status} = \texttt{Married}|\texttt{Yes}) \\ &\quad \times P(\texttt{Annual Income} = \$120\text{K}|\texttt{Yes}) \\ &= 4/6 \times 1/6 \times 1.2 \times 10^{-9} = 1.3 \times 10^{-10}. \end{aligned}$$

The posterior probability for class `No` is $P(\texttt{No}|\mathbf{X}) = \alpha \times 7/10 \times 0.0026 = 0.0018\alpha$, while the posterior probability for class `Yes` is $P(\texttt{Yes}|\mathbf{X}) = \alpha \times 3/10 \times 1.3 \times 10^{-10} = 4.0 \times 10^{-11}\alpha$. Although the classification decision has not changed, the m-estimate approach generally provides a more robust way for estimating probabilities when the number of training examples is small.

Characteristics of Naïve Bayes Classifiers

Naïve Bayes classifiers generally have the following characteristics:

- They are robust to isolated noise points because such points are averaged out when estimating conditional probabilities from data. Naïve Bayes classifiers can also handle missing values by ignoring the example during model building and classification.

- They are robust to irrelevant attributes. If X_i is an irrelevant attribute, then $P(X_i|Y)$ becomes almost uniformly distributed. The class-conditional probability for X_i has no impact on the overall computation of the posterior probability.

- Correlated attributes can degrade the performance of naïve Bayes classifiers because the conditional independence assumption no longer holds for such attributes. For example, consider the following probabilities:

  $$\begin{aligned} &P(A = 0|Y = 0) = 0.4, \quad P(A = 1|Y = 0) = 0.6, \\ &P(A = 0|Y = 1) = 0.6, \quad P(A = 1|Y = 1) = 0.4, \end{aligned}$$

  where A is a binary attribute and Y is a binary class variable. Suppose there is another binary attribute B that is perfectly correlated with A

when $Y = 0$, but is independent of A when $Y = 1$. For simplicity, assume that the class-conditional probabilities for B are the same as for A. Given a record with attributes $A = 0, B = 0$, we can compute its posterior probabilities as follows:

$$\begin{aligned} P(Y=0|A=0,B=0) &= \frac{P(A=0|Y=0)P(B=0|Y=0)P(Y=0)}{P(A=0,B=0)} \\ &= \frac{0.16 \times P(Y=0)}{P(A=0,B=0)}. \end{aligned}$$

$$\begin{aligned} P(Y=1|A=0,B=0) &= \frac{P(A=0|Y=1)P(B=0|Y=1)P(Y=1)}{P(A=0,B=0)} \\ &= \frac{0.36 \times P(Y=1)}{P(A=0,B=0)}. \end{aligned}$$

If $P(Y = 0) = P(Y = 1)$, then the naïve Bayes classifier would assign the record to class 1. However, the truth is,

$$P(A=0,B=0|Y=0) = P(A=0|Y=0) = 0.4,$$

because A and B are perfectly correlated when $Y = 0$. As a result, the posterior probability for $Y = 0$ is

$$\begin{aligned} P(Y=0|A=0,B=0) &= \frac{P(A=0,B=0|Y=0)P(Y=0)}{P(A=0,B=0)} \\ &= \frac{0.4 \times P(Y=0)}{P(A=0,B=0)}, \end{aligned}$$

which is larger than that for $Y = 1$. The record should have been classified as class 0.

5.3.4 Bayes Error Rate

Suppose we know the true probability distribution that governs $P(\mathbf{X}|Y)$. The Bayesian classification method allows us to determine the ideal decision boundary for the classification task, as illustrated in the following example.

Example 5.3. Consider the task of identifying alligators and crocodiles based on their respective lengths. The average length of an adult crocodile is about 15 feet, while the average length of an adult alligator is about 12 feet. Assuming

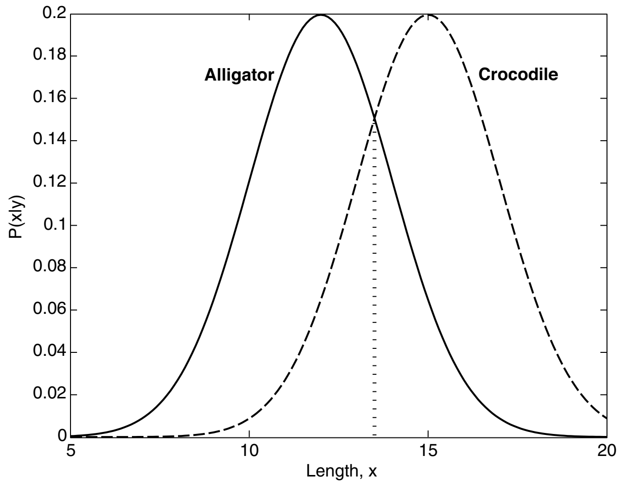


Figure 5.11. Comparing the likelihood functions of a crocodile and an alligator.

that their length x follows a Gaussian distribution with a standard deviation equal to 2 feet, we can express their class-conditional probabilities as follows:

$$P(X|\texttt{Crocodile}) = \frac{1}{\sqrt{2\pi}\cdot 2}\exp\left[-\frac{1}{2}\left(\frac{X-15}{2}\right)^2\right] \tag{5.19}$$

$$P(X|\texttt{Alligator}) = \frac{1}{\sqrt{2\pi}\cdot 2}\exp\left[-\frac{1}{2}\left(\frac{X-12}{2}\right)^2\right] \tag{5.20}$$

Figure 5.11 shows a comparison between the class-conditional probabilities for a crocodile and an alligator. Assuming that their prior probabilities are the same, the ideal decision boundary is located at some length $\hat{x}$ such that

$$P(X=\hat{x}|\texttt{Crocodile}) = P(X=\hat{x}|\texttt{Alligator}).$$

Using Equations 5.19 and 5.20, we obtain

$$\left(\frac{\hat{x}-15}{2}\right)^2 = \left(\frac{\hat{x}-12}{2}\right)^2,$$

which can be solved to yield $\hat{x} = 13.5$. The decision boundary for this example is located halfway between the two means. ∎

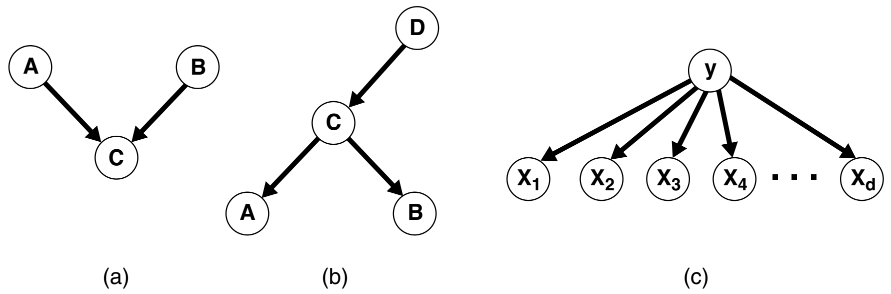


Figure 5.12. Representing probabilistic relationships using directed acyclic graphs.

When the prior probabilities are different, the decision boundary shifts toward the class with lower prior probability (see Exercise 10 on page 319). Furthermore, the minimum error rate attainable by any classifier on the given data can also be computed. The ideal decision boundary in the preceding example classifies all creatures whose lengths are less than $\hat{x}$ as alligators and those whose lengths are greater than $\hat{x}$ as crocodiles. The error rate of the classifier is given by the sum of the area under the posterior probability curve for crocodiles (from length 0 to $\hat{x}$) and the area under the posterior probability curve for alligators (from $\hat{x}$ to ∞):

$$\text{Error} = \int_0^{\hat{x}} P(\texttt{Crocodile}|X)dX + \int_{\hat{x}}^{\infty} P(\texttt{Alligator}|X)dX.$$

The total error rate is known as the **Bayes error rate**.

5.3.5 Bayesian Belief Networks

The conditional independence assumption made by naïve Bayes classifiers may seem too rigid, especially for classification problems in which the attributes are somewhat correlated. This section presents a more flexible approach for modeling the class-conditional probabilities $P(\mathbf{X}|Y)$. Instead of requiring all the attributes to be conditionally independent given the class, this approach allows us to specify which pair of attributes are conditionally independent. We begin with a discussion on how to represent and build such a probabilistic model, followed by an example of how to make inferences from the model.

Model Representation

A Bayesian belief network (BBN), or simply, Bayesian network, provides a graphical representation of the probabilistic relationships among a set of random variables. There are two key elements of a Bayesian network:

1. A directed acyclic graph (dag) encoding the dependence relationships among a set of variables.
2. A probability table associating each node to its immediate parent nodes.

Consider three random variables, A, B, and C, in which A and B are independent variables and each has a direct influence on a third variable, C. The relationships among the variables can be summarized into the directed acyclic graph shown in Figure 5.12(a). Each node in the graph represents a variable, and each arc asserts the dependence relationship between the pair of variables. If there is a directed arc from X to Y, then X is the **parent** of Y and Y is the **child** of X. Furthermore, if there is a directed path in the network from X to Z, then X is an **ancestor** of Z, while Z is a **descendant** of X. For example, in the diagram shown in Figure 5.12(b), A is a descendant of D and D is an ancestor of B. Both B and D are also non-descendants of A. An important property of the Bayesian network can be stated as follows:

Property 1 (Conditional Independence). *A node in a Bayesian network is conditionally independent of its non-descendants, if its parents are known.*

In the diagram shown in Figure 5.12(b), A is conditionally independent of both B and D given C because the nodes for B and D are non-descendants of node A. The conditional independence assumption made by a naïve Bayes classifier can also be represented using a Bayesian network, as shown in Figure 5.12(c), where y is the target class and $\{X_1, X_2, \ldots, X_d\}$ is the attribute set.

Besides the conditional independence conditions imposed by the network topology, each node is also associated with a probability table.

1. If a node X does not have any parents, then the table contains only the prior probability $P(X)$.
2. If a node X has only one parent, Y, then the table contains the conditional probability $P(X|Y)$.
3. If a node X has multiple parents, $\{Y_1, Y_2, \ldots, Y_k\}$, then the table contains the conditional probability $P(X|Y_1, Y_2, \ldots, Y_k)$.

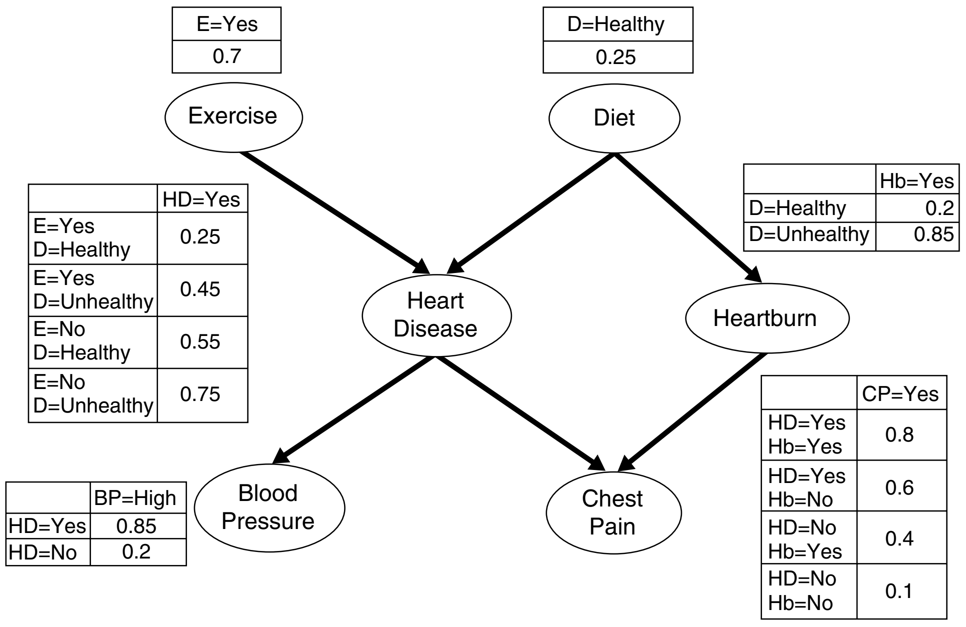


Figure 5.13. A Bayesian belief network for detecting heart disease and heartburn in patients.

Figure 5.13 shows an example of a Bayesian network for modeling patients with heart disease or heartburn problems. Each variable in the diagram is assumed to be binary-valued. The parent nodes for heart disease (HD) correspond to risk factors that may affect the disease, such as exercise (E) and diet (D). The child nodes for heart disease correspond to symptoms of the disease, such as chest pain (CP) and high blood pressure (BP). For example, the diagram shows that heartburn (Hb) may result from an unhealthy diet and may lead to chest pain.

The nodes associated with the risk factors contain only the prior probabilities, whereas the nodes for heart disease, heartburn, and their corresponding symptoms contain the conditional probabilities. To save space, some of the probabilities have been omitted from the diagram. The omitted probabilities can be recovered by noting that $P(X = \overline{x}) = 1 - P(X = x)$ and $P(X = \overline{x}|Y) = 1 - P(X = x|Y)$, where $\overline{x}$ denotes the opposite outcome of x. For example, the conditional probability

$$\begin{aligned}
& P(\texttt{Heart Disease} = \texttt{No}|\texttt{Exercise} = \texttt{No}, \texttt{Diet} = \texttt{Healthy}) \\
= \; & 1 - P(\texttt{Heart Disease} = \texttt{Yes}|\texttt{Exercise} = \texttt{No}, \texttt{Diet} = \texttt{Healthy}) \\
= \; & 1 - 0.55 = 0.45.
\end{aligned}$$

Model Building

Model building in Bayesian networks involves two steps: (1) creating the structure of the network, and (2) estimating the probability values in the tables associated with each node. The network topology can be obtained by encoding the subjective knowledge of domain experts. Algorithm 5.3 presents a systematic procedure for inducing the topology of a Bayesian network.

Algorithm 5.3 Algorithm for generating the topology of a Bayesian network.

```
1: Let $T = (X_1, X_2, \ldots, X_d)$ denote a total order of the variables.
2: for $j = 1$ to $d$ do
3:     Let $X_{T(j)}$ denote the $j^{th}$ highest order variable in $T$.
4:     Let $\pi(X_{T(j)}) = \{X_{T(1)}, X_{T(2)}, \ldots, X_{T(j-1)}\}$ denote the set of variables preceding $X_{T(j)}$.
5:     Remove the variables from $\pi(X_{T(j)})$ that do not affect $X_j$ (using prior knowledge).
6:     Create an arc between $X_{T(j)}$ and the remaining variables in $\pi(X_{T(j)})$.
7: end for
```

Example 5.4. Consider the variables shown in Figure 5.13. After performing Step 1, let us assume that the variables are ordered in the following way: (E, D, HD, Hb, CP, BP). From Steps 2 to 7, starting with variable D, we obtain the following conditional probabilities:

- $P(D|E)$ is simplified to $P(D)$.
- $P(HD|E, D)$ cannot be simplified.
- $P(Hb|HD, E, D)$ is simplified to $P(Hb|D)$.
- $P(CP|Hb, HD, E, D)$ is simplified to $P(CP|Hb, HD)$.
- $P(BP|CP, Hb, HD, E, D)$ is simplified to $P(BP|HD)$.

Based on these conditional probabilities, we can create arcs between the nodes (E, HD), (D, HD), (D, Hb), (HD, CP), (Hb, CP), and (HD, BP). These arcs result in the network structure shown in Figure 5.13. ■

Algorithm 5.3 guarantees a topology that does not contain any cycles. The proof for this is quite straightforward. If a cycle exists, then there must be at least one arc connecting the lower-ordered nodes to the higher-ordered nodes, and at least another arc connecting the higher-ordered nodes to the lower-ordered nodes. Since Algorithm 5.3 prevents any arc from connecting the lower-ordered nodes to the higher-ordered nodes, there cannot be any cycles in the topology.

Nevertheless, the network topology may change if we apply a different ordering scheme to the variables. Some topology may be inferior because it produces many arcs connecting between different pairs of nodes. In principle, we may have to examine all $d!$ possible orderings to determine the most appropriate topology, a task that can be computationally expensive. An alternative approach is to divide the variables into causal and effect variables, and then draw the arcs from each causal variable to its corresponding effect variables. This approach eases the task of building the Bayesian network structure.

Once the right topology has been found, the probability table associated with each node is determined. Estimating such probabilities is fairly straightforward and is similar to the approach used by naïve Bayes classifiers.

Example of Inferencing Using BBN

Suppose we are interested in using the BBN shown in Figure 5.13 to diagnose whether a person has heart disease. The following cases illustrate how the diagnosis can be made under different scenarios.

Case 1: No Prior Information

Without any prior information, we can determine whether the person is likely to have heart disease by computing the prior probabilities $P(\texttt{HD} = \texttt{Yes})$ and $P(\texttt{HD} = \texttt{No})$. To simplify the notation, let $\alpha \in \{\texttt{Yes}, \texttt{No}\}$ denote the binary values of `Exercise` and $\beta \in \{\texttt{Healthy}, \texttt{Unhealthy}\}$ denote the binary values of `Diet`.

$$\begin{aligned}
P(\texttt{HD} = \texttt{Yes}) &= \sum_\alpha \sum_\beta P(\texttt{HD} = \texttt{Yes} | E = \alpha, D = \beta) P(E = \alpha, D = \beta) \\
&= \sum_\alpha \sum_\beta P(\texttt{HD} = \texttt{Yes} | E = \alpha, D = \beta) P(E = \alpha) P(D = \beta) \\
&= 0.25 \times 0.7 \times 0.25 + 0.45 \times 0.7 \times 0.75 + 0.55 \times 0.3 \times 0.25 \\
&\quad + 0.75 \times 0.3 \times 0.75 \\
&= 0.49.
\end{aligned}$$

Since $P(\texttt{HD} = \texttt{no}) = 1 - P(\texttt{HD} = \texttt{yes}) = 0.51$, the person has a slightly higher chance of not getting the disease.

Case 2: High Blood Pressure

If the person has high blood pressure, we can make a diagnosis about heart disease by comparing the posterior probabilities, $P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High})$ against $P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High})$. To do this, we must compute $P(\texttt{BP} = \texttt{High})$:

$$\begin{aligned} P(\texttt{BP} = \texttt{High}) &= \sum_{\gamma} P(\texttt{BP} = \texttt{High}|\texttt{HD} = \gamma)P(\texttt{HD} = \gamma) \\ &= 0.85 \times 0.49 + 0.2 \times 0.51 = 0.5185. \end{aligned}$$

where $\gamma \in \{\texttt{Yes}, \texttt{No}\}$. Therefore, the posterior probability the person has heart disease is

$$\begin{aligned} P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High}) &= \frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes})P(\texttt{HD} = \texttt{Yes})}{P(\texttt{BP} = \texttt{High})} \\ &= \frac{0.85 \times 0.49}{0.5185} = 0.8033. \end{aligned}$$

Similarly, $P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High}) = 1 - 0.8033 = 0.1967$. Therefore, when a person has high blood pressure, it increases the risk of heart disease.

Case 3: High Blood Pressure, Healthy Diet, and Regular Exercise

Suppose we are told that the person exercises regularly and eats a healthy diet. How does the new information affect our diagnosis? With the new information, the posterior probability that the person has heart disease is

$$\begin{aligned} & P(\texttt{HD} = \texttt{Yes}|\texttt{BP} = \texttt{High}, D = \texttt{Healthy}, E = \texttt{Yes}) \\ = & \left[\frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes}, D = \texttt{Healthy}, E = \texttt{Yes})}{P(\texttt{BP} = \texttt{High}|D = \texttt{Healthy}, E = \texttt{Yes})}\right] \\ & \qquad \times\ P(\texttt{HD} = \texttt{Yes}|D = \texttt{Healthy}, E = \texttt{Yes}) \\ = & \frac{P(\texttt{BP} = \texttt{High}|\texttt{HD} = \texttt{Yes})P(\texttt{HD} = \texttt{Yes}|D = \texttt{Healthy}, E = \texttt{Yes})}{\sum_{\gamma} P(\texttt{BP} = \texttt{High}|\texttt{HD} = \gamma)P(\texttt{HD} = \gamma|D = \texttt{Healthy}, E = \texttt{Yes})} \\ = & \frac{0.85 \times 0.25}{0.85 \times 0.25 + 0.2 \times 0.75} \\ = & 0.5862, \end{aligned}$$

while the probability that the person does not have heart disease is

$$P(\texttt{HD} = \texttt{No}|\texttt{BP} = \texttt{High}, D = \texttt{Healthy}, E = \texttt{Yes}) = 1 - 0.5862 = 0.4138.$$

The model therefore suggests that eating healthily and exercising regularly may reduce a person's risk of getting heart disease.

Characteristics of BBN

Following are some of the general characteristics of the BBN method:

1. BBN provides an approach for capturing the prior knowledge of a particular domain using a graphical model. The network can also be used to encode causal dependencies among variables.

2. Constructing the network can be time consuming and requires a large amount of effort. However, once the structure of the network has been determined, adding a new variable is quite straightforward.

3. Bayesian networks are well suited to dealing with incomplete data. Instances with missing attributes can be handled by summing or integrating the probabilities over all possible values of the attribute.

4. Because the data is combined probabilistically with prior knowledge, the method is quite robust to model overfitting.

5.4 Artificial Neural Network (ANN)

The study of artificial neural networks (ANN) was inspired by attempts to simulate biological neural systems. The human brain consists primarily of nerve cells called **neurons**, linked together with other neurons via strands of fiber called **axons**. Axons are used to transmit nerve impulses from one neuron to another whenever the neurons are stimulated. A neuron is connected to the axons of other neurons via **dendrites**, which are extensions from the cell body of the neuron. The contact point between a dendrite and an axon is called a **synapse**. Neurologists have discovered that the human brain learns by changing the strength of the synaptic connection between neurons upon repeated stimulation by the same impulse.

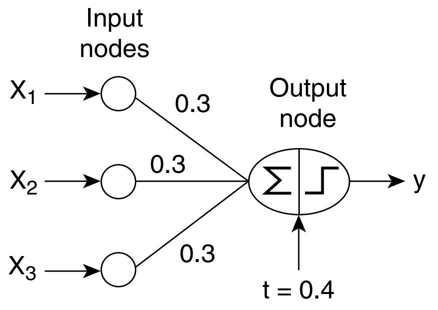
Analogous to human brain structure, an ANN is composed of an interconnected assembly of nodes and directed links. In this section, we will examine a family of ANN models, starting with the simplest model called **perceptron**, and show how the models can be trained to solve classification problems.

5.4.1 Perceptron

Consider the diagram shown in Figure 5.14. The table on the left shows a data set containing three boolean variables (x_1, x_2, x_3) and an output variable, y, that takes on the value -1 if at least two of the three inputs are zero, and $+1$ if at least two of the inputs are greater than zero.

X_1	X_2	X_3	y
1	0	0	–1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	–1
0	1	0	–1
0	1	1	1
0	0	0	–1

(a) Data set.



(b) Perceptron.

Figure 5.14. Modeling a boolean function using a perceptron.

Figure 5.14(b) illustrates a simple neural network architecture known as a perceptron. The perceptron consists of two types of nodes: input nodes, which are used to represent the input attributes, and an output node, which is used to represent the model output. The nodes in a neural network architecture are commonly known as neurons or units. In a perceptron, each input node is connected via a weighted link to the output node. The weighted link is used to emulate the strength of synaptic connection between neurons. As in biological neural systems, training a perceptron model amounts to adapting the weights of the links until they fit the input-output relationships of the underlying data.

A perceptron computes its output value, $\hat{y}$, by performing a weighted sum on its inputs, subtracting a bias factor t from the sum, and then examining the sign of the result. The model shown in Figure 5.14(b) has three input nodes, each of which has an identical weight of 0.3 to the output node and a bias factor of $t = 0.4$. The output computed by the model is

$$\hat{y} = \begin{cases} 1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 > 0; \\ -1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 < 0. \end{cases} \tag{5.21}$$

For example, if $x_1 = 1, x_2 = 1, x_3 = 0$, then $\hat{y} = +1$ because $0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4$ is positive. On the other hand, if $x_1 = 0, x_2 = 1, x_3 = 0$, then $\hat{y} = -1$ because the weighted sum subtracted by the bias factor is negative.

Note the difference between the input and output nodes of a perceptron. An input node simply transmits the value it receives to the outgoing link without performing any transformation. The output node, on the other hand, is a mathematical device that computes the weighted sum of its inputs, subtracts the bias term, and then produces an output that depends on the sign of the resulting sum. More specifically, the output of a perceptron model can be expressed mathematically as follows:

$$\hat{y} = sign\big(w_d x_d + w_{d-1}x_{d-1} + \ldots + w_2 x_2 + w_1 x_1 - t\big), \tag{5.22}$$

where $w_1, w_2, \ldots, w_d$ are the weights of the input links and $x_1, x_2, \ldots, x_d$ are the input attribute values. The sign function, which acts as an **activation function** for the output neuron, outputs a value $+1$ if its argument is positive and -1 if its argument is negative. The perceptron model can be written in a more compact form as follows:

$$\hat{y} = sign[w_d x_d + w_{d-1}x_{d-1} + \ldots + w_1 x_1 + w_0 x_0] = sign(\mathbf{w} \cdot \mathbf{x}), \tag{5.23}$$

where $w_0 = -t$, $x_0 = 1$, and $\mathbf{w} \cdot \mathbf{x}$ is the dot product between the weight vector $\mathbf{w}$ and the input attribute vector $\mathbf{x}$.

Learning Perceptron Model

During the training phase of a perceptron model, the weight parameters $\mathbf{w}$ are adjusted until the outputs of the perceptron become consistent with the true outputs of training examples. A summary of the perceptron learning algorithm is given in Algorithm 5.4.

The key computation for this algorithm is the weight update formula given in Step 7 of the algorithm:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\big(y_i - \hat{y}_i^{(k)}\big)x_{ij}, \tag{5.24}$$

where $w^{(k)}$ is the weight parameter associated with the i^{th} input link after the k^{th} iteration, λ is a parameter known as the **learning rate**, and x_{ij} is the value of the j^{th} attribute of the training example $\mathbf{x}_i$. The justification for the weight update formula is rather intuitive. Equation 5.24 shows that the new weight $w^{(k+1)}$ is a combination of the old weight $w^{(k)}$ and a term proportional

Algorithm 5.4 Perceptron learning algorithm.

```
1: Let $D = \{(\mathbf{x}_i, y_i) \mid i = 1, 2, \ldots, N\}$ be the set of training examples.
2: Initialize the weight vector with random values, $\mathbf{w}^{(0)}$
3: repeat
4:    for each training example $(\mathbf{x}_i, y_i) \in D$ do
5:       Compute the predicted output $\hat{y}_i^{(k)}$
6:       for each weight $w_j$ do
7:          Update the weight, $w_j^{(k+1)} = w_j^{(k)} + \lambda\big(y_i - \hat{y}_i^{(k)}\big)x_{ij}$.
8:       end for
9:    end for
10: until stopping condition is met
```

to the prediction error, $(y - \hat{y})$. If the prediction is correct, then the weight remains unchanged. Otherwise, it is modified in the following ways:

- If $y = +1$ and $\hat{y} = -1$, then the prediction error is $(y - \hat{y}) = 2$. To compensate for the error, we need to increase the value of the predicted output by increasing the weights of all links with positive inputs and decreasing the weights of all links with negative inputs.

- If $y_i = -1$ and $\hat{y} = +1$, then $(y - \hat{y}) = -2$. To compensate for the error, we need to decrease the value of the predicted output by decreasing the weights of all links with positive inputs and increasing the weights of all links with negative inputs.

In the weight update formula, links that contribute the most to the error term are the ones that require the largest adjustment. However, the weights should not be changed too drastically because the error term is computed only for the current training example. Otherwise, the adjustments made in earlier iterations will be undone. The learning rate λ, a parameter whose value is between 0 and 1, can be used to control the amount of adjustments made in each iteration. If λ is close to 0, then the new weight is mostly influenced by the value of the old weight. On the other hand, if λ is close to 1, then the new weight is sensitive to the amount of adjustment performed in the current iteration. In some cases, an adaptive λ value can be used; initially, λ is moderately large during the first few iterations and then gradually decreases in subsequent iterations.

The perceptron model shown in Equation 5.23 is linear in its parameters $\mathbf{w}$ and attributes $\mathbf{x}$. Because of this, the decision boundary of a perceptron, which is obtained by setting $\hat{y} = 0$, is a linear hyperplane that separates the data into two classes, -1 and $+1$. Figure 5.15 shows the decision boundary

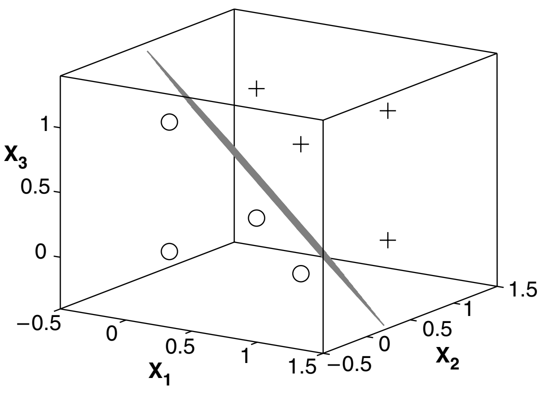


Figure 5.15. Perceptron decision boundary for the data given in Figure 5.14.

obtained by applying the perceptron learning algorithm to the data set given in Figure 5.14. The perceptron learning algorithm is guaranteed to converge to an optimal solution (as long as the learning rate is sufficiently small) for linearly separable classification problems. If the problem is not linearly separable, the algorithm fails to converge. Figure 5.16 shows an example of nonlinearly separable data given by the XOR function. Perceptron cannot find the right solution for this data because there is no linear hyperplane that can perfectly separate the training instances.

X_1	X_2	y
0	0	−1
1	0	1
0	1	1
1	1	−1

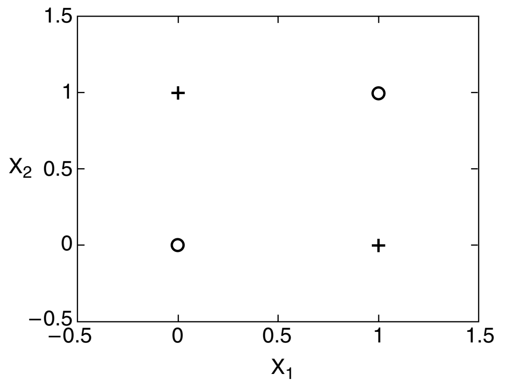


Figure 5.16. XOR classification problem. No linear hyperplane can separate the two classes.

5.4.2 Multilayer Artificial Neural Network

An artificial neural network has a more complex structure than that of a perceptron model. The additional complexities may arise in a number of ways:

1. The network may contain several intermediary layers between its input and output layers. Such intermediary layers are called **hidden layers** and the nodes embedded in these layers are called **hidden nodes**. The resulting structure is known as a multilayer neural network (see Figure 5.17). In a **feed-forward** neural network, the nodes in one layer

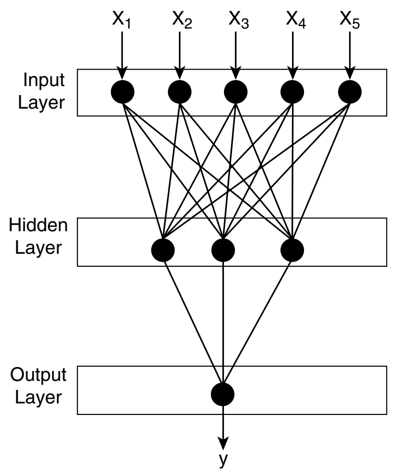


Figure 5.17. Example of a multilayer feed-forward artificial neural network (ANN).

are connected only to the nodes in the next layer. The perceptron is a single-layer, feed-forward neural network because it has only one layer of nodes—the output layer—that performs complex mathematical operations. In a **recurrent** neural network, the links may connect nodes within the same layer or nodes from one layer to the previous layers.

2. The network may use types of activation functions other than the sign function. Examples of other activation functions include linear, sigmoid (logistic), and hyperbolic tangent functions, as shown in Figure 5.18. These activation functions allow the hidden and output nodes to produce output values that are nonlinear in their input parameters.

These additional complexities allow multilayer neural networks to model more complex relationships between the input and output variables. For ex-

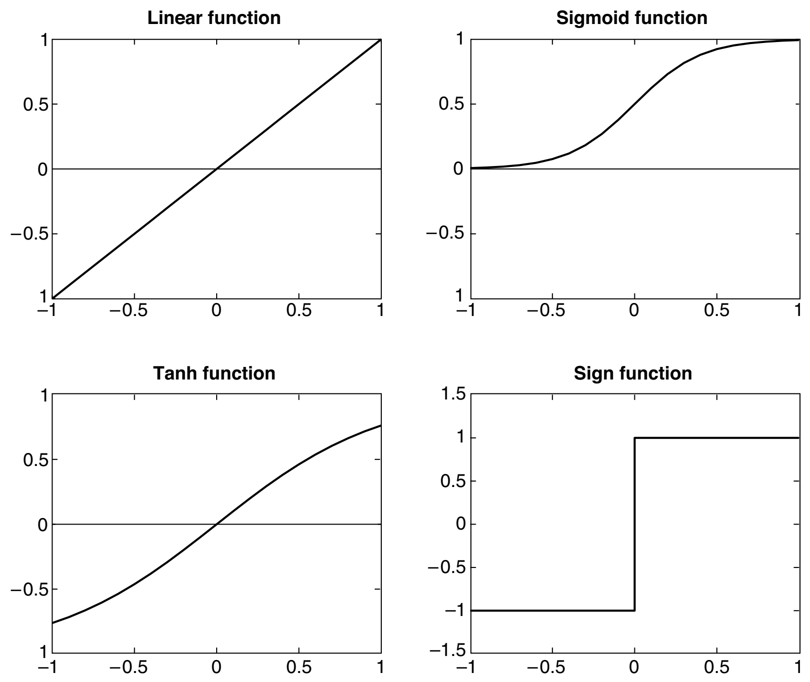
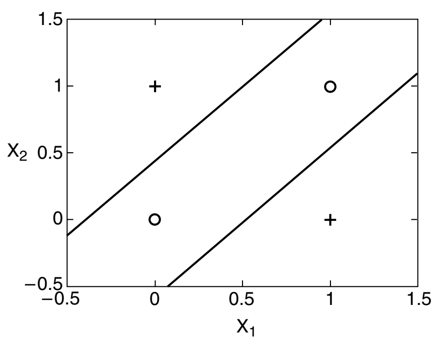


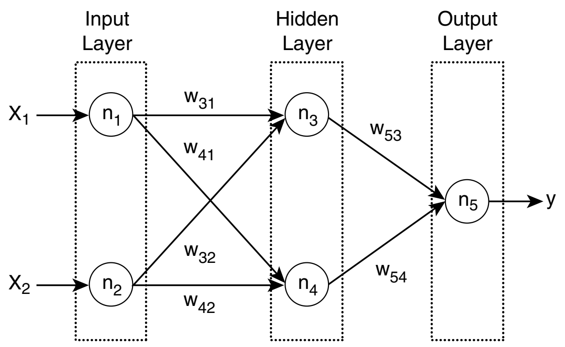
Figure 5.18. Types of activation functions in artificial neural networks.

ample, consider the XOR problem described in the previous section. The instances can be classified using two hyperplanes that partition the input space into their respective classes, as shown in Figure 5.19(a). Because a perceptron can create only one hyperplane, it cannot find the optimal solution. This problem can be addressed using a two-layer, feed-forward neural network, as shown in Figure 5.19(b). Intuitively, we can think of each hidden node as a perceptron that tries to construct one of the two hyperplanes, while the output node simply combines the results of the perceptrons to yield the decision boundary shown in Figure 5.19(a).

To learn the weights of an ANN model, we need an efficient algorithm that converges to the right solution when a sufficient amount of training data is provided. One approach is to treat each hidden node or output node in the network as an independent perceptron unit and to apply the same weight update formula as Equation 5.24. Obviously, this approach will not work because we lack *a priori* knowledge about the true outputs of the hidden nodes. This makes it difficult to determine the error term, $(y - \hat{y})$, associated



(a) Decision boundary.



(b) Neural network topology.

Figure 5.19. A two-layer, feed-forward neural network for the XOR problem.

with each hidden node. A methodology for learning the weights of a neural network based on the gradient descent approach is presented next.

Learning the ANN Model

The goal of the ANN learning algorithm is to determine a set of weights $\mathbf{w}$ that minimize the total sum of squared errors:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{5.25}$$

Note that the sum of squared errors depends on $\mathbf{w}$ because the predicted class $\hat{y}$ is a function of the weights assigned to the hidden and output nodes. Figure 5.20 shows an example of the error surface as a function of its two parameters, w_1 and w_2. This type of error surface is typically encountered when $\hat{y}_i$ is a linear function of its parameters, $\mathbf{w}$. If we replace $\hat{y} = \mathbf{w} \cdot \mathbf{x}$ into Equation 5.25, then the error function becomes quadratic in its parameters and a global minimum solution can be easily found.

In most cases, the output of an ANN is a nonlinear function of its parameters because of the choice of its activation functions (e.g., sigmoid or tanh function). As a result, it is no longer straightforward to derive a solution for $\mathbf{w}$ that is guaranteed to be globally optimal. Greedy algorithms such as those based on the gradient descent method have been developed to efficiently solve the optimization problem. The weight update formula used by the gradient

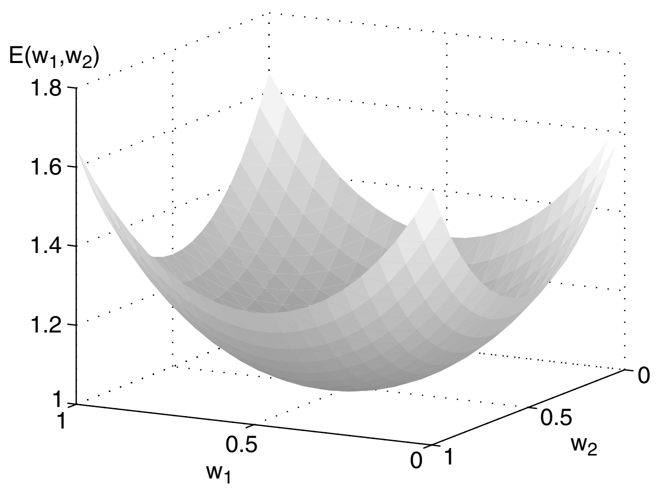


Figure 5.20. Error surface $E(w_1, w_2)$ for a two-parameter model.

descent method can be written as follows:

$$w_j \longleftarrow w_j - \lambda \frac{\partial E(\mathbf{w})}{\partial w_j}, \tag{5.26}$$

where λ is the learning rate. The second term states that the weight should be increased in a direction that reduces the overall error term. However, because the error function is nonlinear, it is possible that the gradient descent method may get trapped in a local minimum.

The gradient descent method can be used to learn the weights of the output and hidden nodes of a neural network. For hidden nodes, the computation is not trivial because it is difficult to assess their error term, $\partial E/\partial w_j$, without knowing what their output values should be. A technique known as **back-propagation** has been developed to address this problem. There are two phases in each iteration of the algorithm: the forward phase and the backward phase. During the forward phase, the weights obtained from the previous iteration are used to compute the output value of each neuron in the network. The computation progresses in the forward direction; i.e., outputs of the neurons at level k are computed prior to computing the outputs at level $k + 1$. During the backward phase, the weight update formula is applied in the reverse direction. In other words, the weights at level $k + 1$ are updated before the weights at level k are updated. This back-propagation approach allows us to use the errors for neurons at layer $k + 1$ to estimate the errors for neurons at layer k.

Design Issues in ANN Learning

Before we train a neural network to learn a classification task, the following design issues must be considered.

1. The number of nodes in the input layer should be determined. Assign an input node to each numerical or binary input variable. If the input variable is categorical, we could either create one node for each categorical value or encode the k-ary variable using $\lceil \log_2 k \rceil$ input nodes.

2. The number of nodes in the output layer should be established. For a two-class problem, it is sufficient to use a single output node. For a k-class problem, there are k output nodes.

3. The network topology (e.g., the number of hidden layers and hidden nodes, and feed-forward or recurrent network architecture) must be selected. Note that the target function representation depends on the weights of the links, the number of hidden nodes and hidden layers, biases in the nodes, and type of activation function. Finding the right topology is not an easy task. One way to do this is to start from a fully connected network with a sufficiently large number of nodes and hidden layers, and then repeat the model-building procedure with a smaller number of nodes. This approach can be very time consuming. Alternatively, instead of repeating the model-building procedure, we could remove some of the nodes and repeat the model evaluation procedure to select the right model complexity.

4. The weights and biases need to be initialized. Random assignments are usually acceptable.

5. Training examples with missing values should be removed or replaced with most likely values.

5.4.3 Characteristics of ANN

Following is a summary of the general characteristics of an artificial neural network:

1. Multilayer neural networks with at least one hidden layer are **universal approximators**; i.e., they can be used to approximate any target functions. Since an ANN has a very expressive hypothesis space, it is important to choose the appropriate network topology for a given problem to avoid model overfitting.

2. ANN can handle redundant features because the weights are automatically learned during the training step. The weights for redundant features tend to be very small.

3. Neural networks are quite sensitive to the presence of noise in the training data. One approach to handling noise is to use a validation set to determine the generalization error of the model. Another approach is to decrease the weight by some factor at each iteration.

4. The gradient descent method used for learning the weights of an ANN often converges to some local minimum. One way to escape from the local minimum is to add a momentum term to the weight update formula.

5. Training an ANN is a time consuming process, especially when the number of hidden nodes is large. Nevertheless, test examples can be classified rapidly.

5.5 Support Vector Machine (SVM)

A classification technique that has received considerable attention is support vector machine (SVM). This technique has its roots in statistical learning theory and has shown promising empirical results in many practical applications, from handwritten digit recognition to text categorization. SVM also works very well with high-dimensional data and avoids the curse of dimensionality problem. Another unique aspect of this approach is that it represents the decision boundary using a subset of the training examples, known as the **support vectors**.

To illustrate the basic idea behind SVM, we first introduce the concept of a **maximal margin hyperplane** and explain the rationale of choosing such a hyperplane. We then describe how a linear SVM can be trained to explicitly look for this type of hyperplane in linearly separable data. We conclude by showing how the SVM methodology can be extended to non-linearly separable data.

5.5.1 Maximum Margin Hyperplanes

Figure 5.21 shows a plot of a data set containing examples that belong to two different classes, represented as squares and circles. The data set is also linearly separable; i.e., we can find a hyperplane such that all the squares reside on one side of the hyperplane and all the circles reside on the other

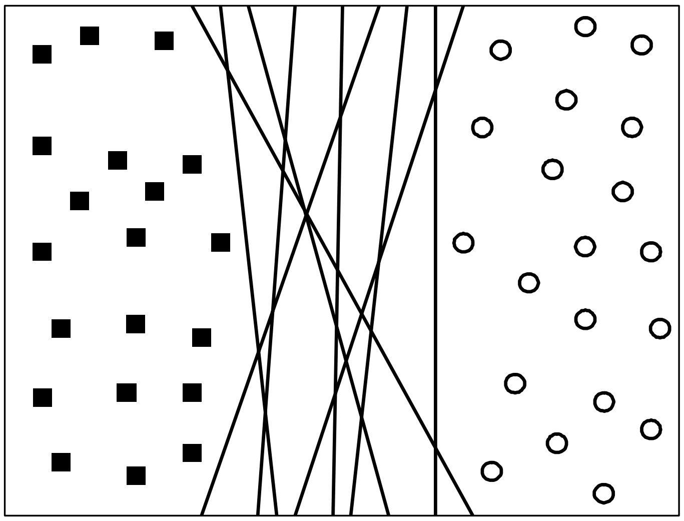


Figure 5.21. Possible decision boundaries for a linearly separable data set.

side. However, as shown in Figure 5.21, there are infinitely many such hyperplanes possible. Although their training errors are zero, there is no guarantee that the hyperplanes will perform equally well on previously unseen examples. The classifier must choose one of these hyperplanes to represent its decision boundary, based on how well they are expected to perform on test examples.

To get a clearer picture of how the different choices of hyperplanes affect the generalization errors, consider the two decision boundaries, B_1 and B_2, shown in Figure 5.22. Both decision boundaries can separate the training examples into their respective classes without committing any misclassification errors. Each decision boundary B_i is associated with a pair of hyperplanes, denoted as b_{i1} and b_{i2}, respectively. b_{i1} is obtained by moving a parallel hyperplane away from the decision boundary until it touches the closest square(s), whereas b_{i2} is obtained by moving the hyperplane until it touches the closest circle(s). The distance between these two hyperplanes is known as the margin of the classifier. From the diagram shown in Figure 5.22, notice that the margin for B_1 is considerably larger than that for B_2. In this example, B_1 turns out to be the maximum margin hyperplane of the training instances.

Rationale for Maximum Margin

Decision boundaries with large margins tend to have better generalization errors than those with small margins. Intuitively, if the margin is small, then

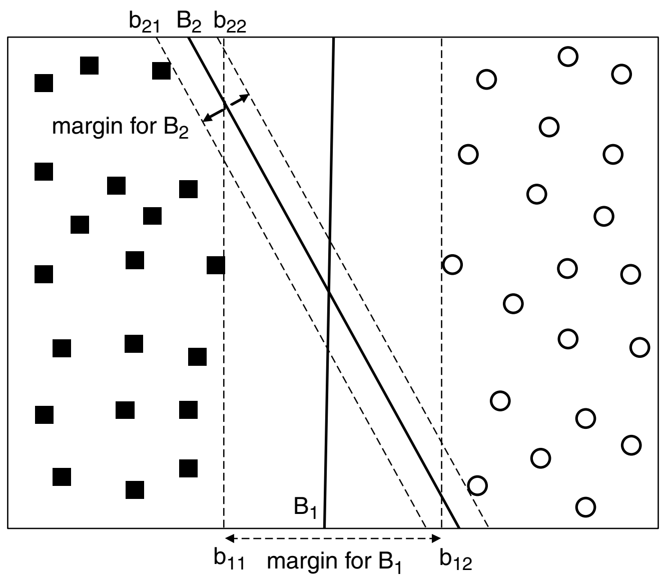


Figure 5.22. Margin of a decision boundary.

any slight perturbations to the decision boundary can have quite a significant impact on its classification. Classifiers that produce decision boundaries with small margins are therefore more susceptible to model overfitting and tend to generalize poorly on previously unseen examples.

A more formal explanation relating the margin of a linear classifier to its generalization error is given by a statistical learning principle known as **structural risk minimization** (SRM). This principle provides an upper bound to the generalization error of a classifier (R) in terms of its training error (R_e), the number of training examples (N), and the model complexity, otherwise known as its **capacity** (h). More specifically, with a probability of $1 - \eta$, the generalization error of the classifier can be at worst

$$R \leq R_e + \varphi\left(\frac{h}{N}, \frac{\log(\eta)}{N}\right), \tag{5.27}$$

where φ is a monotone increasing function of the capacity h. The preceding inequality may seem quite familiar to the readers because it resembles the equation given in Section 4.4.4 (on page 179) for the minimum description length (MDL) principle. In this regard, SRM is another way to express generalization error as a tradeoff between training error and model complexity.

The capacity of a linear model is inversely related to its margin. Models with small margins have higher capacities because they are more flexible and can fit many training sets, unlike models with large margins. However, according to the SRM principle, as the capacity increases, the generalization error bound will also increase. Therefore, it is desirable to design linear classifiers that maximize the margins of their decision boundaries in order to ensure that their worst-case generalization errors are minimized. One such classifier is the **linear SVM**, which is explained in the next section.

5.5.2 Linear SVM: Separable Case

A linear SVM is a classifier that searches for a hyperplane with the largest margin, which is why it is often known as a **maximal margin classifier**. To understand how SVM learns such a boundary, we begin with some preliminary discussion about the decision boundary and margin of a linear classifier.

Linear Decision Boundary

Consider a binary classification problem consisting of N training examples. Each example is denoted by a tuple $(\mathbf{x_i}, y_i)$ $(i = 1, 2, \ldots, N)$, where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T$ corresponds to the attribute set for the i^{th} example. By convention, let $y_i \in \{-1, 1\}$ denote its class label. The decision boundary of a linear classifier can be written in the following form:

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \tag{5.28}$$

where $\mathbf{w}$ and b are parameters of the model.

Figure 5.23 shows a two-dimensional training set consisting of squares and circles. A decision boundary that bisects the training examples into their respective classes is illustrated with a solid line. Any example located along the decision boundary must satisfy Equation 5.28. For example, if $\mathbf{x}_a$ and $\mathbf{x}_b$ are two points located on the decision boundary, then

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}_a + b &= 0, \\ \mathbf{w} \cdot \mathbf{x}_b + b &= 0. \end{aligned}$$

Subtracting the two equations will yield the following:

$$\mathbf{w} \cdot (\mathbf{x}_b - \mathbf{x}_a) = 0,$$

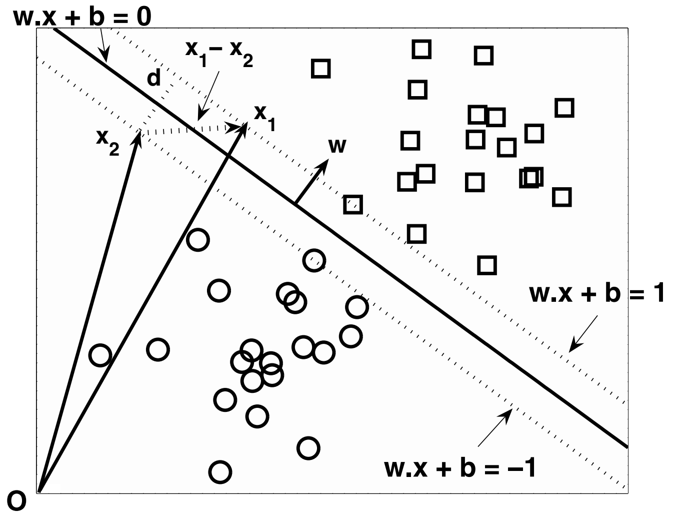


Figure 5.23. Decision boundary and margin of SVM.

where $\mathbf{x}_b - \mathbf{x}_a$ is a vector parallel to the decision boundary and is directed from $\mathbf{x}_a$ to $\mathbf{x}_b$. Since the dot product is zero, the direction for $\mathbf{w}$ must be perpendicular to the decision boundary, as shown in Figure 5.23.

For any square $\mathbf{x}_s$ located above the decision boundary, we can show that

$$\mathbf{w} \cdot \mathbf{x}_s + b = k, \tag{5.29}$$

where $k > 0$. Similarly, for any circle $\mathbf{x}_c$ located below the decision boundary, we can show that

$$\mathbf{w} \cdot \mathbf{x}_c + b = k', \tag{5.30}$$

where $k' < 0$. If we label all the squares as class $+1$ and all the circles as class -1, then we can predict the class label y for any test example $\mathbf{z}$ in the following way:

$$y = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{z} + b > 0; \\ -1, & \text{if } \mathbf{w} \cdot \mathbf{z} + b < 0. \end{cases} \tag{5.31}$$

Margin of a Linear Classifier

Consider the square and the circle that are closest to the decision boundary. Since the square is located above the decision boundary, it must satisfy Equation 5.29 for some positive value k, whereas the circle must satisfy Equation

5.30 for some negative value k'. We can rescale the parameters $\mathbf{w}$ and b of the decision boundary so that the two parallel hyperplanes b_{i1} and b_{i2} can be expressed as follows:

$$b_{i1}: \ \mathbf{w}\cdot\mathbf{x}+b=1, \tag{5.32}$$

$$b_{i2}: \ \mathbf{w}\cdot\mathbf{x}+b=-1. \tag{5.33}$$

The margin of the decision boundary is given by the distance between these two hyperplanes. To compute the margin, let $\mathbf{x}_1$ be a data point located on b_{i1} and $\mathbf{x}_2$ be a data point on b_{i2}, as shown in Figure 5.23. Upon substituting these points into Equations 5.32 and 5.33, the margin d can be computed by subtracting the second equation from the first equation:

$$\begin{aligned}
&\mathbf{w}\cdot(\mathbf{x}_1-\mathbf{x}_2)=2\\
&\|\mathbf{w}\|\times d=2\\
&\therefore d=\frac{2}{\|\mathbf{w}\|}.
\end{aligned} \tag{5.34}$$

Learning a Linear SVM Model

The training phase of SVM involves estimating the parameters $\mathbf{w}$ and b of the decision boundary from the training data. The parameters must be chosen in such a way that the following two conditions are met:

$$\begin{aligned}
\mathbf{w}\cdot\mathbf{x_i}+b &\geq \ \ 1 \text{ if } y_i=1,\\
\mathbf{w}\cdot\mathbf{x_i}+b &\leq -1 \text{ if } y_i=-1.
\end{aligned} \tag{5.35}$$

These conditions impose the requirements that all training instances from class $y=1$ (i.e., the squares) must be located on or above the hyperplane $\mathbf{w}\cdot\mathbf{x}+b=1$, while those instances from class $y=-1$ (i.e., the circles) must be located on or below the hyperplane $\mathbf{w}\cdot\mathbf{x}+b=-1$. Both inequalities can be summarized in a more compact form as follows:

$$y_i(\mathbf{w}\cdot\mathbf{x_i}+b)\geq 1, \quad i=1,2,\ldots,N. \tag{5.36}$$

Although the preceding conditions are also applicable to any linear classifiers (including perceptrons), SVM imposes an additional requirement that the margin of its decision boundary must be maximal. Maximizing the margin, however, is equivalent to minimizing the following objective function:

$$f(\mathbf{w})=\frac{\|\mathbf{w}\|^2}{2}. \tag{5.37}$$

Definition 5.1 (Linear SVM: Separable Case). The learning task in SVM can be formalized as the following constrained optimization problem:

$$\min_{\mathbf{w}} \frac{\|\mathbf{w}\|^2}{2}$$

$$\text{subject to} \qquad y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1, \quad i = 1, 2, \ldots, N.$$

Since the objective function is quadratic and the constraints are linear in the parameters **w** and b, this is known as a **convex** optimization problem, which can be solved using the standard **Lagrange multiplier** method. Following is a brief sketch of the main ideas for solving the optimization problem. A more detailed discussion is given in Appendix E.

First, we must rewrite the objective function in a form that takes into account the constraints imposed on its solutions. The new objective function is known as the Lagrangian for the optimization problem:

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \lambda_i \Big(y_i(\mathbf{w} \cdot \mathbf{x_i} + b) - 1 \Big), \tag{5.38}$$

where the parameters λ_i are called the Lagrange multipliers. The first term in the Lagrangian is the same as the original objective function, while the second term captures the inequality constraints. To understand why the objective function must be modified, consider the original objective function given in Equation 5.37. It is easy to show that the function is minimized when $\mathbf{w} = \mathbf{0}$, a null vector whose components are all zeros. Such a solution, however, violates the constraints given in Definition 5.1 because there is no feasible solution for b. The solutions for **w** and b are infeasible if they violate the inequality constraints; i.e., if $y_i(\mathbf{w} \cdot \mathbf{x_i} + b) - 1 < 0$. The Lagrangian given in Equation 5.38 incorporates this constraint by subtracting the term from its original objective function. Assuming that $\lambda_i \geq 0$, it is clear that any infeasible solution may only increase the value of the Lagrangian.

To minimize the Lagrangian, we must take the derivative of L_P with respect to **w** and b and set them to zero:

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0 \Longrightarrow \mathbf{w} = \sum_{i=1}^{N} \lambda_i y_i \mathbf{x}_i, \tag{5.39}$$

$$\frac{\partial L_p}{\partial b} = 0 \Longrightarrow \sum_{i=1}^{N} \lambda_i y_i = 0. \tag{5.40}$$

Because the Lagrange multipliers are unknown, we still cannot solve for $\mathbf{w}$ and b. If Definition 5.1 contains only equality instead of inequality constraints, then we can use the N equations from equality constraints along with Equations 5.39 and 5.40 to find the feasible solutions for $\mathbf{w}$, b, and λ_i. Note that the Lagrange multipliers for equality constraints are free parameters that can take any values.

One way to handle the inequality constraints is to transform them into a set of equality constraints. This is possible as long as the Lagrange multipliers are restricted to be non-negative. Such transformation leads to the following constraints on the Lagrange multipliers, which are known as the Karush-Kuhn-Tucker (KKT) conditions:

$$\lambda_i \geq 0, \tag{5.41}$$

$$\lambda_i \big[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \big] = 0. \tag{5.42}$$

At first glance, it may seem that there are as many Lagrange multipliers as there are training instances. It turns out that many of the Lagrange multipliers become zero after applying the constraint given in Equation 5.42. The constraint states that the Lagrange multiplier λ_i must be zero unless the training instance $\mathbf{x}_i$ satisfies the equation $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$. Such training instance, with $\lambda_i > 0$, lies along the hyperplanes b_{i1} or b_{i2} and is known as a support vector. Training instances that do not reside along these hyperplanes have $\lambda_i = 0$. Equations 5.39 and 5.42 also suggest that the parameters $\mathbf{w}$ and b, which define the decision boundary, depend only on the support vectors.

Solving the preceding optimization problem is still quite a daunting task because it involves a large number of parameters: $\mathbf{w}$, b, and λ_i. The problem can be simplified by transforming the Lagrangian into a function of the Lagrange multipliers only (this is known as the dual problem). To do this, we first substitute Equations 5.39 and 5.40 into Equation 5.38. This will lead to the following dual formulation of the optimization problem:

$$L_D = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j}. \tag{5.43}$$

The key differences between the dual and primary Lagrangians are as follows:

1. The dual Lagrangian involves only the Lagrange multipliers and the training data, while the primary Lagrangian involves the Lagrange multipliers as well as parameters of the decision boundary. Nevertheless, the solutions for both optimization problems are equivalent.

2. The quadratic term in Equation 5.43 has a negative sign, which means that the original minimization problem involving the primary Lagrangian, L_P, has turned into a maximization problem involving the dual Lagrangian, L_D.

For large data sets, the dual optimization problem can be solved using numerical techniques such as quadratic programming, a topic that is beyond the scope of this book. Once the λ_i's are found, we can use Equations 5.39 and 5.42 to obtain the feasible solutions for $\mathbf{w}$ and b. The decision boundary can be expressed as follows:

$$\left(\sum_{i=1}^{N} \lambda_i y_i \mathbf{x_i} \cdot \mathbf{x}\right) + b = 0. \tag{5.44}$$

b is obtained by solving Equation 5.42 for the support vectors. Because the λ_i's are calculated numerically and can have numerical errors, the value computed for b may not be unique. Instead it depends on the support vector used in Equation 5.42. In practice, the average value for b is chosen to be the parameter of the decision boundary.

Example 5.5. Consider the two-dimensional data set shown in Figure 5.24, which contains eight training instances. Using quadratic programming, we can solve the optimization problem stated in Equation 5.43 to obtain the Lagrange multiplier λ_i for each training instance. The Lagrange multipliers are depicted in the last column of the table. Notice that only the first two instances have non-zero Lagrange multipliers. These instances correspond to the support vectors for this data set.

Let $\mathbf{w} = (w_1, w_2)$ and b denote the parameters of the decision boundary. Using Equation 5.39, we can solve for w_1 and w_2 in the following way:

$$\begin{aligned} w_1 &= \sum_i \lambda_i y_i x_{i1} = 65.5621 \times 1 \times 0.3858 + 65.5621 \times -1 \times 0.4871 = -6.64. \\ w_2 &= \sum_i \lambda_i y_i x_{i2} = 65.5621 \times 1 \times 0.4687 + 65.5621 \times -1 \times 0.611 = -9.32. \end{aligned}$$

The bias term b can be computed using Equation 5.42 for each support vector:

$$\begin{aligned} b^{(1)} &= 1 - \mathbf{w} \cdot \mathbf{x}_1 = 1 - (-6.64)(0.3858) - (-9.32)(0.4687) = 7.9300. \\ b^{(2)} &= -1 - \mathbf{w} \cdot \mathbf{x}_2 = -1 - (-6.64)(0.4871) - (-9.32)(0.611) = 7.9289. \end{aligned}$$

Averaging these values, we obtain $b = 7.93$. The decision boundary corresponding to these parameters is shown in Figure 5.24. ■

x_1	x_2	y	Lagrange Multiplier
0.3858	0.4687	1	65.5261
0.4871	0.611	–1	65.5261
0.9218	0.4103	–1	0
0.7382	0.8936	–1	0
0.1763	0.0579	1	0
0.4057	0.3529	1	0
0.9355	0.8132	–1	0
0.2146	0.0099	1	0

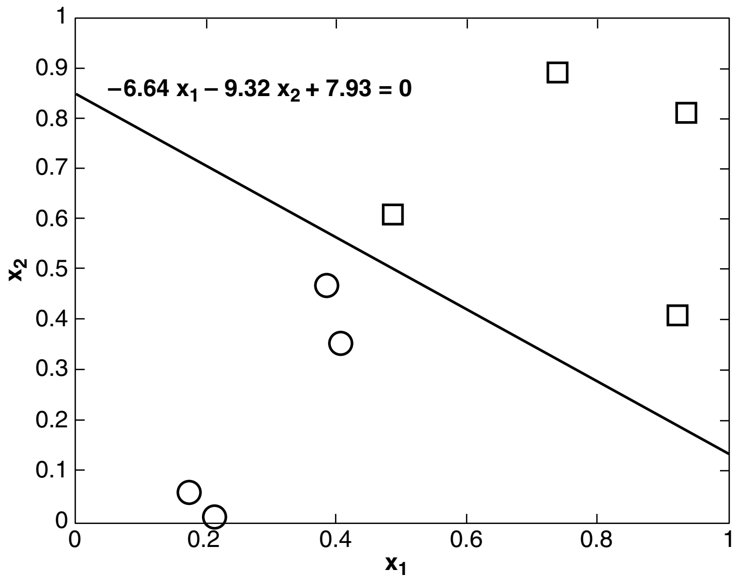


Figure 5.24. Example of a linearly separable data set.

Once the parameters of the decision boundary are found, a test instance $\mathbf{z}$ is classified as follows:

$$f(\mathbf{z}) = sign\big(\mathbf{w} \cdot \mathbf{z} + b\big) = sign\Bigg(\sum_{i=1}^{N} \lambda_i y_i \mathbf{x_i} \cdot \mathbf{z} + b\Bigg).$$

If $f(\mathbf{z}) = 1$, then the test instance is classified as a positive class; otherwise, it is classified as a negative class.

5.5.3 Linear SVM: Nonseparable Case

Figure 5.25 shows a data set that is similar to Figure 5.22, except it has two new examples, P and Q. Although the decision boundary B_1 misclassifies the new examples, while B_2 classifies them correctly, this does not mean that B_2 is a better decision boundary than B_1 because the new examples may correspond to noise in the training data. B_1 should still be preferred over B_2 because it has a wider margin, and thus, is less susceptible to overfitting. However, the SVM formulation presented in the previous section constructs only decision boundaries that are mistake-free. This section examines how the formulation can be modified to learn a decision boundary that is tolerable to small training errors using a method known as the **soft margin** approach. More importantly, the method presented in this section allows SVM to construct a linear decision boundary even in situations where the classes are not linearly separable. To do this, the learning algorithm in SVM must consider the trade-off between the width of the margin and the number of training errors committed by the linear decision boundary.

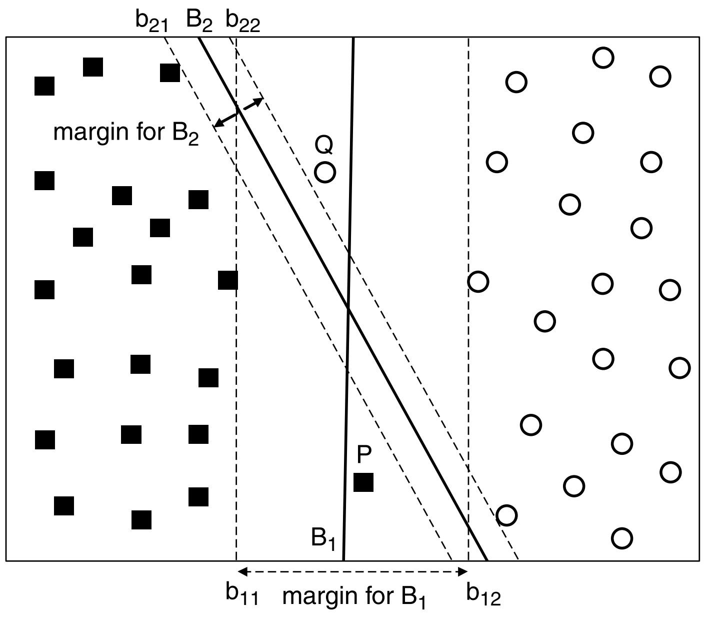


Figure 5.25. Decision boundary of SVM for the nonseparable case.

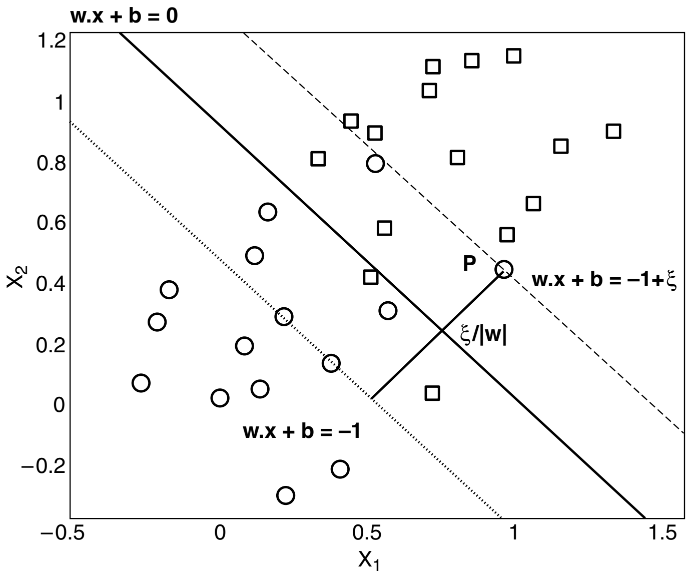


Figure 5.26. Slack variables for nonseparable data.

While the original objective function given in Equation 5.37 is still applicable, the decision boundary B_1 no longer satisfies all the constraints given in Equation 5.36. The inequality constraints must therefore be relaxed to accommodate the nonlinearly separable data. This can be done by introducing positive-valued **slack variables** (ξ) into the constraints of the optimization problem, as shown in the following equations:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x_i} + b &\geq 1 - \xi_i \quad \text{if } y_i = 1, \\ \mathbf{w} \cdot \mathbf{x_i} + b &\leq -1 + \xi_i \quad \text{if } y_i = -1, \end{aligned} \tag{5.45}$$

where $\forall i: \ \xi_i > 0$.

To interpret the meaning of the slack variables ξ_i, consider the diagram shown in Figure 5.26. The circle **P** is one of the instances that violates the constraints given in Equation 5.35. Let $\mathbf{w} \cdot \mathbf{x} + b = -1 + \xi$ denote a line that is parallel to the decision boundary and passes through the point **P**. It can be shown that the distance between this line and the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = -1$ is $\xi/\|\mathbf{w}\|$. Thus, ξ provides an estimate of the error of the decision boundary on the training example **P**.

In principle, we can apply the same objective function as before and impose the conditions given in Equation 5.45 to find the decision boundary. However,

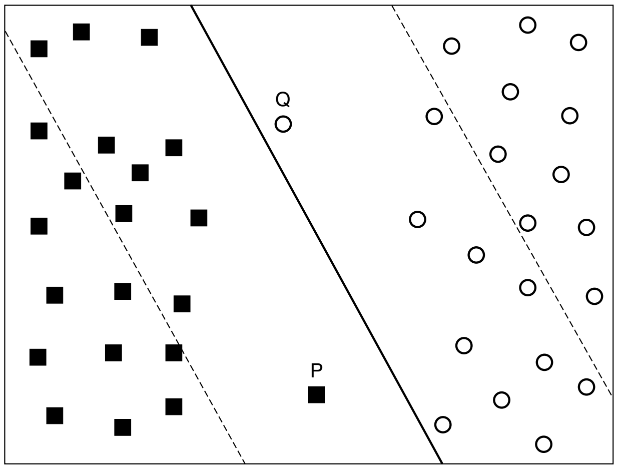


Figure 5.27. A decision boundary that has a wide margin but large training error.

since there are no constraints on the number of mistakes the decision boundary can make, the learning algorithm may find a decision boundary with a very wide margin but misclassifies many of the training examples, as shown in Figure 5.27. To avoid this problem, the objective function must be modified to penalize a decision boundary with large values of slack variables. The modified objective function is given by the following equation:

$$f(\mathbf{w}) = \frac{\|\mathbf{w}\|^2}{2} + C(\sum_{i=1}^{N} \xi_i)^k,$$

where C and k are user-specified parameters representing the penalty of misclassifying the training instances. For the remainder of this section, we assume $k = 1$ to simplify the problem. The parameter C can be chosen based on the model's performance on the validation set.

It follows that the Lagrangian for this constrained optimization problem can be written as follows:

$$L_P = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \lambda_i\{y_i(\mathbf{w}\cdot\mathbf{x_i} + b) - 1 + \xi_i\} - \sum_{i=1}^{N} \mu_i \xi_i, \quad (5.46)$$

where the first two terms are the objective function to be minimized, the third term represents the inequality constraints associated with the slack variables, and the last term is the result of the non-negativity requirements on the values of ξ_i's. Furthermore, the inequality constraints can be transformed into equality constraints using the following KKT conditions:

$$\xi_i \geq 0, \quad \lambda_i \geq 0, \quad \mu_i \geq 0, \tag{5.47}$$

$$\lambda_i\{y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 + \xi_i\} = 0, \tag{5.48}$$

$$\mu_i \xi_i = 0. \tag{5.49}$$

Note that the Lagrange multiplier λ_i given in Equation 5.48 is non-vanishing only if the training instance resides along the lines $\mathbf{w}\cdot\mathbf{x}_i + b = \pm 1$ or has $\xi_i > 0$. On the other hand, the Lagrange multipliers μ_i given in Equation 5.49 are zero for any training instances that are misclassified (i.e., having $\xi_i > 0$).

Setting the first-order derivative of L with respect to $\mathbf{w}$, b, and ξ_i to zero would result in the following equations:

$$\frac{\partial L}{\partial w_j} = w_j - \sum_{i=1}^{N} \lambda_i y_i x_{ij} = 0 \implies w_j = \sum_{i=1}^{N} \lambda_i y_i x_{ij}. \tag{5.50}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{N} \lambda_i y_i = 0 \implies \sum_{i=1}^{N} \lambda_i y_i = 0. \tag{5.51}$$

$$\frac{\partial L}{\partial \xi_i} = C - \lambda_i - \mu_i = 0 \implies \lambda_i + \mu_i = C. \tag{5.52}$$

Substituting Equations 5.50, 5.51, and 5.52 into the Lagrangian will produce the following dual Lagrangian:

$$\begin{aligned} L_D &= \frac{1}{2}\sum_{i,j} \lambda_i\lambda_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j + C\sum_i \xi_i \\ &\quad -\sum_i \lambda_i\{y_i(\sum_j \lambda_j y_j \mathbf{x}_i\cdot\mathbf{x}_j + b) - 1 + \xi_i\} \\ &\quad -\sum_i (C - \lambda_i)\xi_i \\ &= \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i,j} \lambda_i\lambda_j y_i y_j \mathbf{x_i}\cdot\mathbf{x_j}, \end{aligned} \tag{5.53}$$

which turns out to be identical to the dual Lagrangian for linearly separable data (see Equation 5.40 on page 262). Nevertheless, the constraints imposed on the Lagrange multipliers λ_i's are slightly different those in the linearly separable case. In the linearly separable case, the Lagrange multipliers must be non-negative, i.e., $\lambda_i \geq 0$. On the other hand, Equation 5.52 suggests that λ_i should not exceed C (since both μ_i and λ_i are non-negative). Therefore, the Lagrange multipliers for nonlinearly separable data are restricted to $0 \leq \lambda_i \leq C$.

The dual problem can then be solved numerically using quadratic programming techniques to obtain the Lagrange multipliers λ_i. These multipliers can be replaced into Equation 5.50 and the KKT conditions to obtain the parameters of the decision boundary.

5.5.4 Nonlinear SVM

The SVM formulations described in the previous sections construct a linear decision boundary to separate the training examples into their respective classes. This section presents a methodology for applying SVM to data sets that have nonlinear decision boundaries. The trick here is to transform the data from its original coordinate space in $\mathbf{x}$ into a new space $\Phi(\mathbf{x})$ so that a linear decision boundary can be used to separate the instances in the transformed space. After doing the transformation, we can apply the methodology presented in the previous sections to find a linear decision boundary in the transformed space.

Attribute Transformation

To illustrate how attribute transformation can lead to a linear decision boundary, Figure 5.28(a) shows an example of a two-dimensional data set consisting of squares (classified as $y = 1$) and circles (classified as $y = -1$). The data set is generated in such a way that all the circles are clustered near the center of the diagram and all the squares are distributed farther away from the center. Instances of the data set can be classified using the following equation:

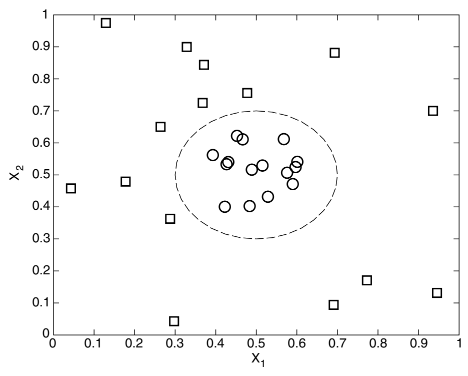
$$y(x_1, x_2) = \begin{cases} 1 & \text{if } \sqrt{(x_1 - 0.5)^2 + (x_2 - 0.5)^2} > 0.2, \\ -1 & \text{otherwise.} \end{cases} \tag{5.54}$$

The decision boundary for the data can therefore be written as follows:

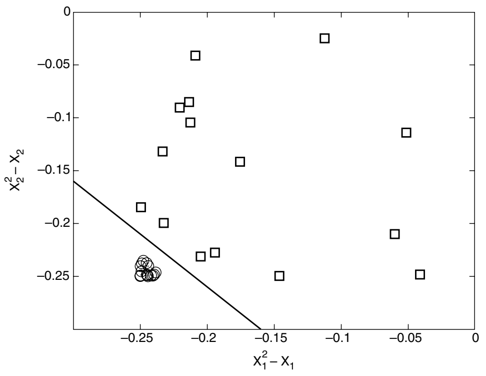
$$\sqrt{(x_1 - 0.5)^2 + (x_2 - 0.5)^2} = 0.2,$$

which can be further simplified into the following quadratic equation:

$$x_1^2 - x_1 + x_2^2 - x_2 = -0.46.$$



(a) Decision boundary in the original two-dimensional space.



(b) Decision boundary in the transformed space.

Figure 5.28. Classifying data with a nonlinear decision boundary.

A nonlinear transformation Φ is needed to map the data from its original feature space into a new space where the decision boundary becomes linear. Suppose we choose the following transformation:

$$\Phi : (x_1, x_2) \longrightarrow (x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, 1). \tag{5.55}$$

In the transformed space, we can find the parameters $\mathbf{w} = (w_0,\ w_1,\ \ldots,\ w_4)$ such that:

$$w_4 x_1^2 + w_3 x_2^2 + w_2\sqrt{2}x_1 + w_1\sqrt{2}x_2 + w_0 = 0.$$

For illustration purposes, let us plot the graph of $x_2^2 - x_2$ versus $x_1^2 - x_1$ for the previously given instances. Figure 5.28(b) shows that in the transformed space, all the circles are located in the lower right-hand side of the diagram. A linear decision boundary can therefore be constructed to separate the instances into their respective classes.

One potential problem with this approach is that it may suffer from the curse of dimensionality problem often associated with high-dimensional data. We will show how nonlinear SVM avoids this problem (using a method known as the kernel trick) later in this section.

Learning a Nonlinear SVM Model

Although the attribute transformation approach seems promising, it raises several implementation issues. First, it is not clear what type of mapping function should be used to ensure that a linear decision boundary can be constructed in the transformed space. One possibility is to transform the data into an infinite dimensional space, but such a high-dimensional space may not be that easy to work with. Second, even if the appropriate mapping function is known, solving the constrained optimization problem in the high-dimensional feature space is a computationally expensive task.

To illustrate these issues and examine the ways they can be addressed, let us assume that there is a suitable function, $\Phi(\mathbf{x})$, to transform a given data set. After the transformation, we need to construct a linear decision boundary that will separate the instances into their respective classes. The linear decision boundary in the transformed space has the following form: $\mathbf{w} \cdot \Phi(\mathbf{x}) + b = 0$.

Definition 5.2 (Nonlinear SVM). The learning task for a nonlinear SVM can be formalized as the following optimization problem:

$$\begin{aligned} & \min_{\mathbf{w}} \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \Phi(\mathbf{x}_i) + b) \geq 1, \quad i = 1, 2, \ldots, N. \end{aligned}$$

Note the similarity between the learning task of a nonlinear SVM to that of a linear SVM (see Definition 5.1 on page 262). The main difference is that, instead of using the original attributes $\mathbf{x}$, the learning task is performed on the transformed attributes $\Phi(\mathbf{x})$. Following the approach taken in Sections 5.5.2 and 5.5.3 for linear SVM, we may derive the following dual Lagrangian for the constrained optimization problem:

$$L_D = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) \tag{5.56}$$

Once the λ_i's are found using quadratic programming techniques, the parameters $\mathbf{w}$ and b can be derived using the following equations:

$$\mathbf{w} = \sum_{i} \lambda_i y_i \Phi(\mathbf{x}_i) \tag{5.57}$$

$$\lambda_i \{ y_i (\sum_{j} \lambda_j y_j \Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}_i) + b) - 1 \} = 0, \tag{5.58}$$

which are analogous to Equations 5.39 and 5.40 for linear SVM. Finally, a test instance z can be classified using the following equation:

$$f(\mathbf{z}) = sign\big(\mathbf{w} \cdot \Phi(\mathbf{z}) + b\big) = sign\bigg(\sum_{i=1}^{n} \lambda_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{z}) + b\bigg). \tag{5.59}$$

Except for Equation 5.57, note that the rest of the computations (Equations 5.58 and 5.59) involve calculating the dot product (i.e., similarity) between pairs of vectors in the transformed space, $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. Such computation can be quite cumbersome and may suffer from the curse of dimensionality problem. A breakthrough solution to this problem comes in the form of a method known as the **kernel trick**.

Kernel Trick

The dot product is often regarded as a measure of similarity between two input vectors. For example, the cosine similarity described in Section 2.4.5 on page 73 can be defined as the dot product between two vectors that are normalized to unit length. Analogously, the dot product $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$ can also be regarded as a measure of similarity between two instances, $\mathbf{x}_i$ and $\mathbf{x}_j$, in the transformed space.

The kernel trick is a method for computing similarity in the transformed space using the original attribute set. Consider the mapping function Φ given in Equation 5.55. The dot product between two input vectors $\mathbf{u}$ and $\mathbf{v}$ in the transformed space can be written as follows:

$$\begin{aligned}
\Phi(\mathbf{u}) \cdot \Phi(\mathbf{v}) &= (u_1^2, u_2^2, \sqrt{2}u_1, \sqrt{2}u_2, 1) \cdot (v_1^2, v_2^2, \sqrt{2}v_1, \sqrt{2}v_2, 1) \\
&= u_1^2 v_1^2 + u_2^2 v_2^2 + 2u_1 v_1 + 2u_2 v_2 + 1 \\
&= (\mathbf{u} \cdot \mathbf{v} + 1)^2.
\end{aligned} \tag{5.60}$$

This analysis shows that the dot product in the transformed space can be expressed in terms of a similarity function in the original space:

$$K(\mathbf{u}, \mathbf{v}) = \Phi(\mathbf{u}) \cdot \Phi(\mathbf{v}) = (\mathbf{u} \cdot \mathbf{v} + 1)^2. \tag{5.61}$$

The similarity function, K, which is computed in the original attribute space, is known as the **kernel function**. The kernel trick helps to address some of the concerns about how to implement nonlinear SVM. First, we do not have to know the exact form of the mapping function Φ because the kernel functions used in nonlinear SVM must satisfy a mathematical principle known as **Mercer's theorem**. This principle ensures that the kernel functions can always be expressed as the dot product between two input vectors in some high-dimensional space. The transformed space of the SVM kernels is called a **reproducing kernel Hilbert space** (RKHS). Second, computing the dot products using kernel functions is considerably cheaper than using the transformed attribute set $\Phi(\mathbf{x})$. Third, since the computations are performed in the original space, issues associated with the curse of dimensionality problem can be avoided.

Figure 5.29 shows the nonlinear decision boundary obtained by SVM using the polynomial kernel function given in Equation 5.61. A test instance $\mathbf{x}$ is classified according to the following equation:

$$\begin{aligned} f(\mathbf{z}) &= sign(\sum_{i=1}^{n} \lambda_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{z}) + b) \\ &= sign(\sum_{i=1}^{n} \lambda_i y_i K(\mathbf{x}_i, \mathbf{z}) + b) \\ &= sign(\sum_{i=1}^{n} \lambda_i y_i (\mathbf{x}_i \cdot \mathbf{z} + 1)^2 + b), \end{aligned} \tag{5.62}$$

where b is the parameter obtained using Equation 5.58. The decision boundary obtained by nonlinear SVM is quite close to the true decision boundary shown in Figure 5.28(a).

Mercer's Theorem

The main requirement for the kernel function used in nonlinear SVM is that there must exist a corresponding transformation such that the kernel function computed for a pair of vectors is equivalent to the dot product between the vectors in the transformed space. This requirement can be formally stated in the form of Mercer's theorem.

Theorem 5.1 (Mercer's Theorem). *A kernel function K can be expressed as*

$$K(\boldsymbol{u}, \boldsymbol{v}) = \Phi(\boldsymbol{u}) \cdot \Phi(\boldsymbol{v})$$

if and only if, for any function $g(x)$ such that $\int g(\boldsymbol{x})^2 d\boldsymbol{x}$ is finite, then

$$\int K(\boldsymbol{x}, \boldsymbol{y}) \; g(\boldsymbol{x}) \; g(\boldsymbol{y}) \; d\boldsymbol{x} \; d\boldsymbol{y} \geq 0.$$

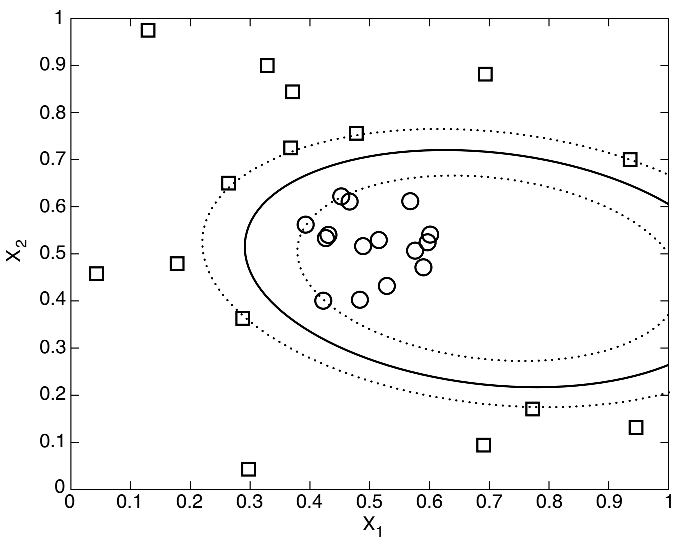


Figure 5.29. Decision boundary produced by a nonlinear SVM with polynomial kernel.

Kernel functions that satisfy Theorem 5.1 are called positive definite kernel functions. Examples of such functions are listed below:

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y}+1)^p \tag{5.63}$$

$$K(\mathbf{x},\mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|^2/(2\sigma^2)} \tag{5.64}$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(k\mathbf{x}\cdot\mathbf{y}-\delta) \tag{5.65}$$

Example 5.6. Consider the polynomial kernel function given in Equation 5.63. Let $g(x)$ be a function that has a finite L_2 norm, i.e., $\int g(\mathbf{x})^2 d\mathbf{x} < \infty$.

$$\begin{aligned}
&\int (\mathbf{x}\cdot\mathbf{y}+1)^p g(\mathbf{x})g(\mathbf{y})d\mathbf{x}d\mathbf{y} \\
= \;& \int \sum_{i=0}^{p} \binom{p}{i} (\mathbf{x}\cdot\mathbf{y})^i g(\mathbf{x})g(\mathbf{y})d\mathbf{x}d\mathbf{y} \\
= \;& \sum_{i=0}^{p} \binom{p}{i} \int \sum_{\alpha_1,\alpha_2,\ldots} \binom{i}{\alpha_1\alpha_2\ldots} \Big[(x_1y_1)^{\alpha_1}(x_2y_2)^{\alpha_2}(x_3y_3)^{\alpha_3}\ldots\Big] \\
&\qquad\qquad g(x_1,x_2,\ldots)\; g(y_1,y_2,\ldots)dx_1dx_2\ldots dy_1dy_2\ldots
\end{aligned}$$

$$= \sum_{i=0}^{p} \sum_{\alpha_1,\alpha_2,\ldots} \binom{p}{i} \binom{i}{\alpha_1 \alpha_2 \ldots} \left[\int x_1^{\alpha_1} x_2^{\alpha_2} \ldots g(x_1, x_2, \ldots) dx_1 dx_2 \ldots \right]^2.$$

Because the result of the integration is non-negative, the polynomial kernel function therefore satisfies Mercer's theorem. ∎

5.5.5 Characteristics of SVM

SVM has many desirable qualities that make it one of the most widely used classification algorithms. Following is a summary of the general characteristics of SVM:

1. The SVM learning problem can be formulated as a convex optimization problem, in which efficient algorithms are available to find the global minimum of the objective function. Other classification methods, such as rule-based classifiers and artificial neural networks, employ a greedy-based strategy to search the hypothesis space. Such methods tend to find only locally optimum solutions.

2. SVM performs capacity control by maximizing the margin of the decision boundary. Nevertheless, the user must still provide other parameters such as the type of kernel function to use and the cost function C for introducing each slack variable.

3. SVM can be applied to categorical data by introducing dummy variables for each categorical attribute value present in the data. For example, if `Marital Status` has three values {`Single`, `Married`, `Divorced`}, we can introduce a binary variable for each of the attribute values.

4. The SVM formulation presented in this chapter is for binary class problems. Some of the methods available to extend SVM to multiclass problems are presented in Section 5.8.

5.6 Ensemble Methods

The classification techniques we have seen so far in this chapter, with the exception of the nearest-neighbor method, predict the class labels of unknown examples using a single classifier induced from training data. This section presents techniques for improving classification accuracy by aggregating the predictions of multiple classifiers. These techniques are known as the **ensemble** or **classifier combination** methods. An ensemble method constructs a set of **base classifiers** from training data and performs classification by taking a vote on the predictions made by each base classifier. This section explains why ensemble methods tend to perform better than any single classifier and presents techniques for constructing the classifier ensemble.

5.6.1 Rationale for Ensemble Method

The following example illustrates how an ensemble method can improve a classifier's performance.

Example 5.7. Consider an ensemble of twenty-five binary classifiers, each of which has an error rate of $\epsilon = 0.35$. The ensemble classifier predicts the class label of a test example by taking a majority vote on the predictions made by the base classifiers. If the base classifiers are identical, then the ensemble will misclassify the same examples predicted incorrectly by the base classifiers. Thus, the error rate of the ensemble remains 0.35. On the other hand, if the base classifiers are independent—i.e., their errors are uncorrelated—then the ensemble makes a wrong prediction only if more than half of the base classifiers predict incorrectly. In this case, the error rate of the ensemble classifier is

$$e_{\text{ensemble}} = \sum_{i=13}^{25} \binom{25}{i} \epsilon^i (1-\epsilon)^{25-i} = 0.06, \tag{5.66}$$

which is considerably lower than the error rate of the base classifiers. ∎

Figure 5.30 shows the error rate of an ensemble of twenty-five binary classifiers (e_{ensemble}) for different base classifier error rates (ϵ). The diagonal line represents the case in which the base classifiers are identical, while the solid line represents the case in which the base classifiers are independent. Observe that the ensemble classifier performs worse than the base classifiers when ϵ is larger than 0.5.

The preceding example illustrates two necessary conditions for an ensemble classifier to perform better than a single classifier: (1) the base classifiers should be independent of each other, and (2) the base classifiers should do better than a classifier that performs random guessing. In practice, it is difficult to ensure total independence among the base classifiers. Nevertheless, improvements in classification accuracies have been observed in ensemble methods in which the base classifiers are slightly correlated.

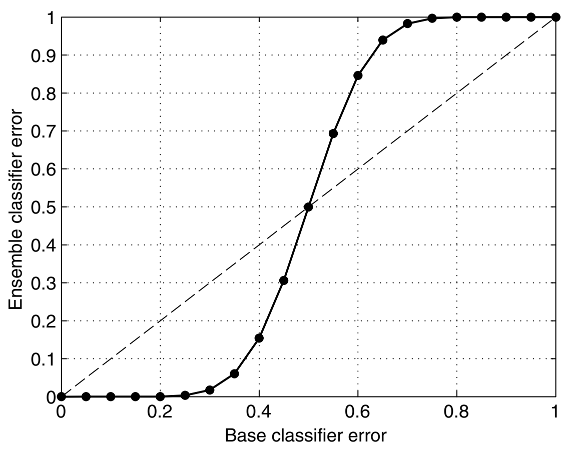
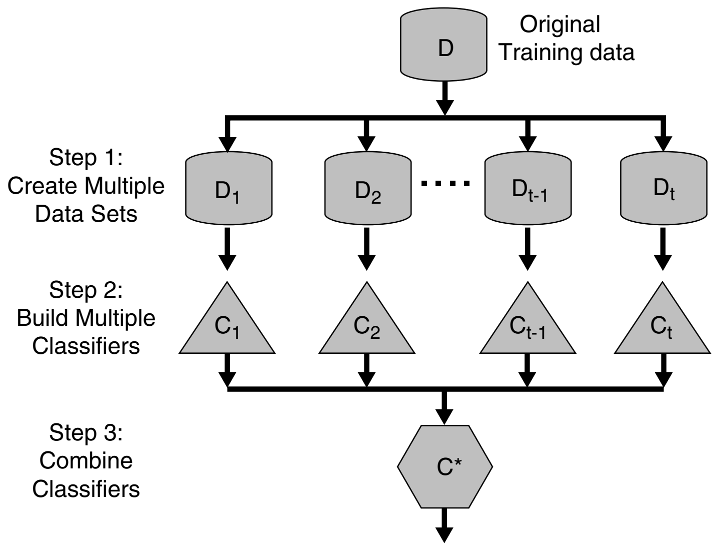


Figure 5.30. Comparison between errors of base classifiers and errors of the ensemble classifier.



D
Original Training data
Step 1: Create Multiple Data Sets
D_1
D_2
D_{t-1}
D_t
Step 2: Build Multiple Classifiers
C_1
C_2
C_{t-1}
C_t
Step 3: Combine Classifiers
C*

Figure 5.31. A logical view of the ensemble learning method.

5.6.2 Methods for Constructing an Ensemble Classifier

A logical view of the ensemble method is presented in Figure 5.31. The basic idea is to construct multiple classifiers from the original data and then aggregate their predictions when classifying unknown examples. The ensemble of classifiers can be constructed in many ways:

1. **By manipulating the training set.** In this approach, multiple training sets are created by resampling the original data according to some sampling distribution. The sampling distribution determines how likely it is that an example will be selected for training, and it may vary from one trial to another. A classifier is then built from each training set using a particular learning algorithm. **Bagging** and **boosting** are two examples of ensemble methods that manipulate their training sets. These methods are described in further detail in Sections 5.6.4 and 5.6.5.

2. **By manipulating the input features.** In this approach, a subset of input features is chosen to form each training set. The subset can be either chosen randomly or based on the recommendation of domain experts. Some studies have shown that this approach works very well with data sets that contain highly redundant features. **Random forest**, which is described in Section 5.6.6, is an ensemble method that manipulates its input features and uses decision trees as its base classifiers.

3. **By manipulating the class labels.** This method can be used when the number of classes is sufficiently large. The training data is transformed into a binary class problem by randomly partitioning the class labels into two disjoint subsets, A_0 and A_1. Training examples whose class label belongs to the subset A_0 are assigned to class 0, while those that belong to the subset A_1 are assigned to class 1. The relabeled examples are then used to train a base classifier. By repeating the class-relabeling and model-building steps multiple times, an ensemble of base classifiers is obtained. When a test example is presented, each base classifier C_i is used to predict its class label. If the test example is predicted as class 0, then all the classes that belong to A_0 will receive a vote. Conversely, if it is predicted to be class 1, then all the classes that belong to A_1 will receive a vote. The votes are tallied and the class that receives the highest vote is assigned to the test example. An example of this approach is the **error-correcting output coding** method described on page 307.

4. **By manipulating the learning algorithm.** Many learning algorithms can be manipulated in such a way that applying the algorithm several times on the same training data may result in different models. For example, an artificial neural network can produce different models by changing its network topology or the initial weights of the links between neurons. Similarly, an ensemble of decision trees can be constructed by injecting randomness into the tree-growing procedure. For

example, instead of choosing the best splitting attribute at each node, we can randomly choose one of the top k attributes for splitting.

The first three approaches are generic methods that are applicable to any classifiers, whereas the fourth approach depends on the type of classifier used. The base classifiers for most of these approaches can be generated sequentially (one after another) or in parallel (all at once). Algorithm 5.5 shows the steps needed to build an ensemble classifier in a sequential manner. The first step is to create a training set from the original data D. Depending on the type of ensemble method used, the training sets are either identical to or slight modifications of D. The size of the training set is often kept the same as the original data, but the distribution of examples may not be identical; i.e., some examples may appear multiple times in the training set, while others may not appear even once. A base classifier C_i is then constructed from each training set D_i. Ensemble methods work better with **unstable classifiers**, i.e., base classifiers that are sensitive to minor perturbations in the training set. Examples of unstable classifiers include decision trees, rule-based classifiers, and artificial neural networks. As will be discussed in Section 5.6.3, the variability among training examples is one of the primary sources of errors in a classifier. By aggregating the base classifiers built from different training sets, this may help to reduce such types of errors.

Finally, a test example $\mathbf{x}$ is classified by combining the predictions made by the base classifiers $C_i(\mathbf{x})$:

$$C^*(\mathbf{x}) = Vote(C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_k(\mathbf{x})).$$

The class can be obtained by taking a majority vote on the individual predictions or by weighting each prediction with the accuracy of the base classifier.

Algorithm 5.5 General procedure for ensemble method.

```
1: Let $D$ denote the original training data, $k$ denote the number of base classifiers,
   and $T$ be the test data.
2: for $i = 1$ to $k$ do
3:    Create training set, $D_i$ from $D$.
4:    Build a base classifier $C_i$ from $D_i$.
5: end for
6: for each test record $x \in T$ do
7:    $C^*(x) = Vote(C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_k(\mathbf{x}))$
8: end for
```

5.6.3 Bias-Variance Decomposition

Bias-variance decomposition is a formal method for analyzing the prediction error of a predictive model. The following example gives an intuitive explanation for this method.

Figure 5.32 shows the trajectories of a projectile launched at a particular angle. Suppose the projectile hits the floor surface at some location x, at a distance d away from the target position t. Depending on the force applied to the projectile, the observed distance may vary from one trial to another. The observed distance can be decomposed into several components. The first component, which is known as **bias**, measures the average distance between the target position and the location where the projectile hits the floor. The amount of bias depends on the angle of the projectile launcher. The second component, which is known as **variance**, measures the deviation between x and the average position $\overline{x}$ where the projectile hits the floor. The variance can be explained as a result of changes in the amount of force applied to the projectile. Finally, if the target is not stationary, then the observed distance is also affected by changes in the location of the target. This is considered the **noise** component associated with variability in the target position. Putting these components together, the average distance can be expressed as:

$$d_{f,\theta}(y,t) = \text{Bias}_\theta + \text{Variance}_f + \text{Noise}_t, \tag{5.67}$$

where f refers to the amount of force applied and θ is the angle of the launcher.

The task of predicting the class label of a given example can be analyzed using the same approach. For a given classifier, some predictions may turn out to be correct, while others may be completely off the mark. We can decompose the expected error of a classifier as a sum of the three terms given in Equation 5.67, where expected error is the probability that the classifier misclassifies a

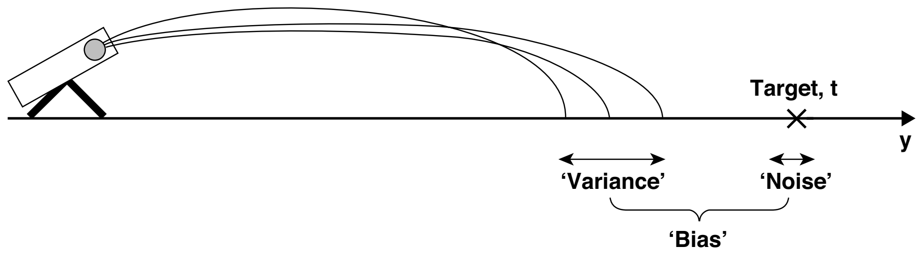


Figure 5.32. Bias-variance decomposition.

given example. The remainder of this section examines the meaning of bias, variance, and noise in the context of classification.

A classifier is usually trained to minimize its training error. However, to be useful, the classifier must be able to make an informed guess about the class labels of examples it has never seen before. This requires the classifier to generalize its decision boundary to regions where there are no training examples available—a decision that depends on the design choice of the classifier. For example, a key design issue in decision tree induction is the amount of pruning needed to obtain a tree with low expected error. Figure 5.33 shows two decision trees, T_1 and T_2, that are generated from the same training data, but have different complexities. T_2 is obtained by pruning T_1 until a tree with maximum depth of two is obtained. T_1, on the other hand, performs very little pruning on its decision tree. These design choices will introduce a bias into the classifier that is analogous to the bias of the projectile launcher described in the previous example. In general, the stronger the assumptions made by a classifier about the nature of its decision boundary, the larger the classifier's bias will be. T_2 therefore has a larger bias because it makes stronger assumptions about its decision boundary (which is reflected by the size of the tree) compared to T_1. Other design choices that may introduce a bias into a classifier include the network topology of an artificial neural network and the number of neighbors considered by a nearest-neighbor classifier.

The expected error of a classifier is also affected by variability in the training data because different compositions of the training set may lead to different decision boundaries. This is analogous to the variance in x when different amounts of force are applied to the projectile. The last component of the expected error is associated with the intrinsic noise in the target class. The target class for some domains can be non-deterministic; i.e., instances with the same attribute values can have different class labels. Such errors are unavoidable even when the true decision boundary is known.

The amount of bias and variance contributing to the expected error depend on the type of classifier used. Figure 5.34 compares the decision boundaries produced by a decision tree and a 1-nearest neighbor classifier. For each classifier, we plot the decision boundary obtained by "averaging" the models induced from 100 training sets, each containing 100 examples. The true decision boundary from which the data is generated is also plotted using a dashed line. The difference between the true decision boundary and the "averaged" decision boundary reflects the bias of the classifier. After averaging the models, observe that the difference between the true decision boundary and the decision boundary produced by the 1-nearest neighbor classifier is smaller than

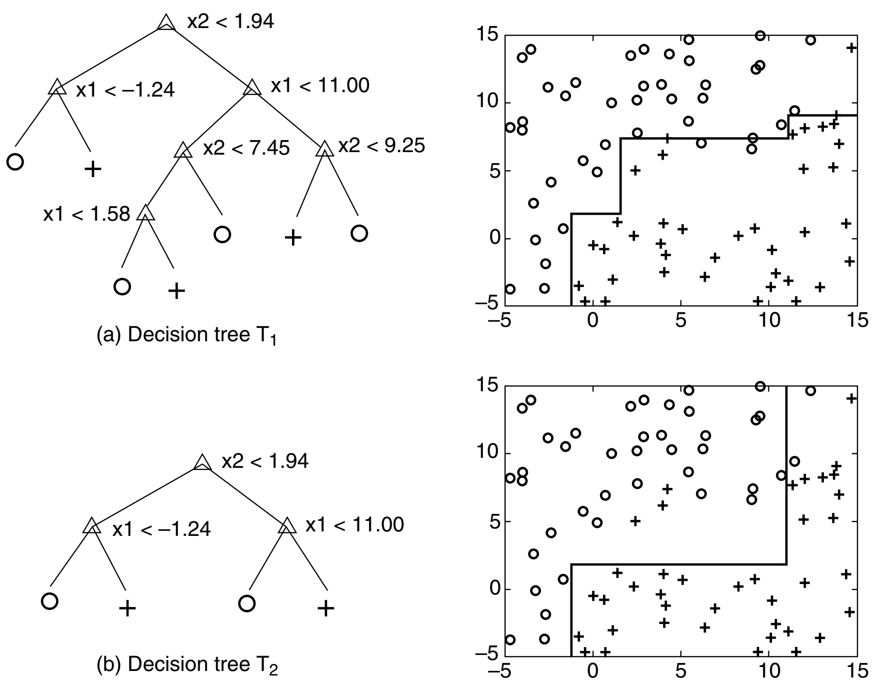


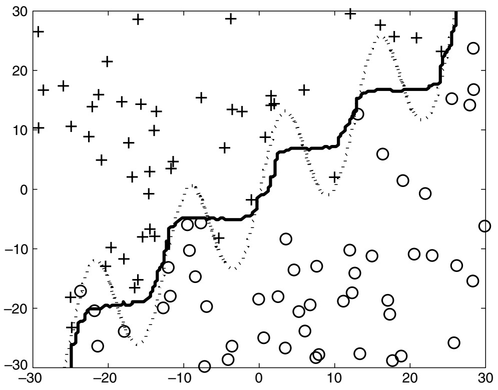
Figure 5.33. Two decision trees with different complexities induced from the same training data.

the observed difference for a decision tree classifier. This result suggests that the bias of a 1-nearest neighbor classifier is lower than the bias of a decision tree classifier.

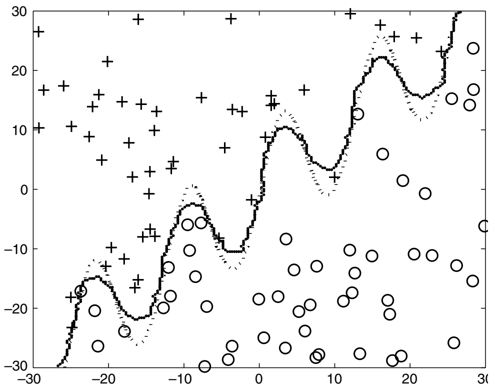
On the other hand, the 1-nearest neighbor classifier is more sensitive to the composition of its training examples. If we examine the models induced from different training sets, there is more variability in the decision boundary of a 1-nearest neighbor classifier than a decision tree classifier. Therefore, the decision boundary of a decision tree classifier has a lower variance than the 1-nearest neighbor classifier.

5.6.4 Bagging

Bagging, which is also known as bootstrap aggregating, is a technique that repeatedly samples (with replacement) from a data set according to a uniform probability distribution. Each bootstrap sample has the same size as the original data. Because the sampling is done with replacement, some instances may appear several times in the same training set, while others may be omitted from the training set. On average, a bootstrap sample D_i contains approxi-



(a) Decision boundary for decision tree.



(b) Decision boundary for 1-nearest neighbor.

Figure 5.34. Bias of decision tree and 1-nearest neighbor classifiers.

Algorithm 5.6 Bagging algorithm.

```
1: Let $k$ be the number of bootstrap samples.
2: **for** $i = 1$ to $k$ **do**
3:     Create a bootstrap sample of size $N$, $D_i$.
4:     Train a base classifier $C_i$ on the bootstrap sample $D_i$.
5: **end for**
6: $C^*(x) = \underset{y}{\operatorname{argmax}} \sum_i \delta\big(C_i(x) = y\big)$.
   $\{\delta(\cdot) = 1$ if its argument is true and 0 otherwise$\}$.
```

mately 63% of the original training data because each sample has a probability $1 - (1 - 1/N)^N$ of being selected in each D_i. If N is sufficiently large, this probability converges to $1 - 1/e \simeq 0.632$. The basic procedure for bagging is summarized in Algorithm 5.6. After training the k classifiers, a test instance is assigned to the class that receives the highest number of votes.

To illustrate how bagging works, consider the data set shown in Table 5.4. Let x denote a one-dimensional attribute and y denote the class label. Suppose we apply a classifier that induces only one-level binary decision trees, with a test condition $x \leq k$, where k is a split point chosen to minimize the entropy of the leaf nodes. Such a tree is also known as a **decision stump**.

Without bagging, the best decision stump we can produce splits the records at either $x \leq 0.35$ or $x \leq 0.75$. Either way, the accuracy of the tree is at

Table 5.4. Example of data set used to construct an ensemble of bagging classifiers.

x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
y	1	1	1	-1	-1	-1	-1	1	1	1

most 70%. Suppose we apply the bagging procedure on the data set using ten bootstrap samples. The examples chosen for training in each bagging round are shown in Figure 5.35. On the right-hand side of each table, we also illustrate the decision boundary produced by the classifier.

We classify the entire data set given in Table 5.4 by taking a majority vote among the predictions made by each base classifier. The results of the predictions are shown in Figure 5.36. Since the class labels are either -1 or $+1$, taking the majority vote is equivalent to summing up the predicted values of y and examining the sign of the resulting sum (refer to the second to last row in Figure 5.36). Notice that the ensemble classifier perfectly classifies all ten examples in the original data.

The preceding example illustrates another advantage of using ensemble methods in terms of enhancing the representation of the target function. Even though each base classifier is a decision stump, combining the classifiers can lead to a decision tree of depth 2.

Bagging improves generalization error by reducing the variance of the base classifiers. The performance of bagging depends on the stability of the base classifier. If a base classifier is unstable, bagging helps to reduce the errors associated with random fluctuations in the training data. If a base classifier is stable, i.e., robust to minor perturbations in the training set, then the error of the ensemble is primarily caused by bias in the base classifier. In this situation, bagging may not be able to improve the performance of the base classifiers significantly. It may even degrade the classifier's performance because the effective size of each training set is about 37% smaller than the original data.

Finally, since every sample has an equal probability of being selected, bagging does not focus on any particular instance of the training data. It is therefore less susceptible to model overfitting when applied to noisy data.

5.6.5 Boosting

Boosting is an iterative procedure used to adaptively change the distribution of training examples so that the base classifiers will focus on examples that are hard to classify. Unlike bagging, boosting assigns a weight to each training

Bagging Round 1:

x	0.1	0.2	0.2	0.3	0.4	0.4	0.5	0.6	0.9	0.9
y	1	1	1	1	-1	-1	-1	-1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 2:

x	0.1	0.2	0.3	0.4	0.5	0.8	0.9	1	1	1
y	1	1	1	-1	-1	1	1	1	1	1

x <= 0.65 ==> y = 1
x > 0.65 ==> y = 1

Bagging Round 3:

x	0.1	0.2	0.3	0.4	0.4	0.5	0.7	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 4:

x	0.1	0.1	0.2	0.4	0.4	0.5	0.5	0.7	0.8	0.9
y	1	1	1	-1	-1	-1	-1	-1	1	1

x <= 0.3 ==> y = 1
x > 0.3 ==> y = -1

Bagging Round 5:

x	0.1	0.1	0.2	0.5	0.6	0.6	0.6	1	1	1
y	1	1	1	-1	-1	-1	-1	1	1	1

x <= 0.35 ==> y = 1
x > 0.35 ==> y = -1

Bagging Round 6:

x	0.2	0.4	0.5	0.6	0.7	0.7	0.7	0.8	0.9	1
y	1	-1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 7:

x	0.1	0.4	0.4	0.6	0.7	0.8	0.9	0.9	0.9	1
y	1	-1	-1	-1	-1	1	1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 8:

x	0.1	0.2	0.5	0.5	0.5	0.7	0.7	0.8	0.9	1
y	1	1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 9:

x	0.1	0.3	0.4	0.4	0.6	0.7	0.7	0.8	1	1
y	1	1	-1	-1	-1	-1	-1	1	1	1

x <= 0.75 ==> y = -1
x > 0.75 ==> y = 1

Bagging Round 10:

x	0.1	0.1	0.1	0.1	0.3	0.3	0.8	0.8	0.9	0.9
y	1	1	1	1	1	1	1	1	1	1

x <= 0.05 ==> y = -1
x > 0.05 ==> y = 1

Figure 5.35. Example of bagging.

example and may adaptively change the weight at the end of each boosting round. The weights assigned to the training examples can be used in the following ways:

1. They can be used as a sampling distribution to draw a set of bootstrap samples from the original data.

2. They can be used by the base classifier to learn a model that is biased toward higher-weight examples.

Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	1	1	1	-1	-1	-1	-1	-1	-1	-1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	-1	-1	-1	-1	-1	-1	-1
4	1	1	1	-1	-1	-1	-1	-1	-1	-1
5	1	1	1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	1	1	1
7	-1	-1	-1	-1	-1	-1	-1	1	1	1
8	-1	-1	-1	-1	-1	-1	-1	1	1	1
9	-1	-1	-1	-1	-1	-1	-1	1	1	1
10	1	1	1	1	1	1	1	1	1	1
Sum	2	2	2	-6	-6	-6	-6	2	2	2
Sign	1	1	1	-1	-1	-1	-1	1	1	1
True Class	1	1	1	-1	-1	-1	-1	1	1	1

Figure 5.36. Example of combining classifiers constructed using the bagging approach.

This section describes an algorithm that uses weights of examples to determine the sampling distribution of its training set. Initially, the examples are assigned equal weights, $1/N$, so that they are equally likely to be chosen for training. A sample is drawn according to the sampling distribution of the training examples to obtain a new training set. Next, a classifier is induced from the training set and used to classify all the examples in the original data. The weights of the training examples are updated at the end of each boosting round. Examples that are classified incorrectly will have their weights increased, while those that are classified correctly will have their weights decreased. This forces the classifier to focus on examples that are difficult to classify in subsequent iterations.

The following table shows the examples chosen during each boosting round.

Boosting (Round 1):	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2):	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3):	4	4	8	10	4	5	4	6	3	4

Initially, all the examples are assigned the same weights. However, some examples may be chosen more than once, e.g., examples 3 and 7, because the sampling is done with replacement. A classifier built from the data is then used to classify all the examples. Suppose example 4 is difficult to classify. The weight for this example will be increased in future iterations as it gets misclassified repeatedly. Meanwhile, examples that were not chosen in the previous round, e.g., examples 1 and 5, also have a better chance of being selected in the next round since their predictions in the previous round were likely to be wrong. As the boosting rounds proceed, examples that are the hardest to classify tend to become even more prevalent. The final ensemble is obtained by aggregating the base classifiers obtained from each boosting round.

Over the years, several implementations of the boosting algorithm have been developed. These algorithms differ in terms of (1) how the weights of the training examples are updated at the end of each boosting round, and (2) how the predictions made by each classifier are combined. An implementation called AdaBoost is explored in the next section.

AdaBoost

Let $\{(\mathbf{x}_j, y_j) \mid j = 1, 2, \ldots, N\}$ denote a set of N training examples. In the AdaBoost algorithm, the importance of a base classifier C_i depends on its error rate, which is defined as

$$\epsilon_i = \frac{1}{N}\left[\sum_{j=1}^{N} w_j \; I\Big(C_i(\mathbf{x}_j) \neq y_j\Big)\right], \tag{5.68}$$

where $I(p) = 1$ if the predicate p is true, and 0 otherwise. The importance of a classifier C_i is given by the following parameter,

$$\alpha_i = \frac{1}{2}\ln\left(\frac{1-\epsilon_i}{\epsilon_i}\right).$$

Note that α_i has a large positive value if the error rate is close to 0 and a large negative value if the error rate is close to 1, as shown in Figure 5.37.

The α_i parameter is also used to update the weight of the training examples. To illustrate, let $w_i^{(j)}$ denote the weight assigned to example $(\mathbf{x}_i, y_i)$ during the j^{th} boosting round. The weight update mechanism for AdaBoost is given by the equation:

$$w_i^{(j+1)} = \frac{w_i^{(j)}}{Z_j} \times \begin{cases} \exp^{-\alpha_j} & \text{if } C_j(\mathbf{x_i}) = y_i \\ \exp^{\alpha_j} & \text{if } C_j(\mathbf{x_i}) \neq y_i \end{cases}, \tag{5.69}$$

where Z_j is the normalization factor used to ensure that $\sum_i w_i^{(j+1)} = 1$. The weight update formula given in Equation 5.69 increases the weights of incorrectly classified examples and decreases the weights of those classified correctly.

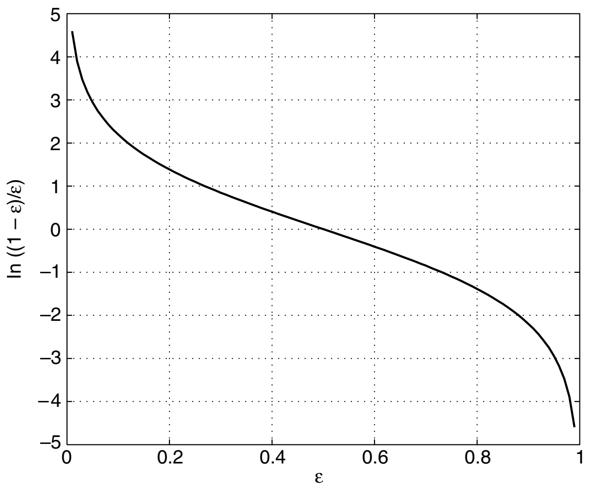


Figure 5.37. Plot of α as a function of training error ϵ.

Instead of using a majority voting scheme, the prediction made by each classifier C_j is weighted according to α_j. This approach allows AdaBoost to penalize models that have poor accuracy, e.g., those generated at the earlier boosting rounds. In addition, if any intermediate rounds produce an error rate higher than 50%, the weights are reverted back to their original uniform values, $w_i = 1/N$, and the resampling procedure is repeated. The AdaBoost algorithm is summarized in Algorithm 5.7.

Let us examine how the boosting approach works on the data set shown in Table 5.4. Initially, all the examples have identical weights. After three boosting rounds, the examples chosen for training are shown in Figure 5.38(a). The weights for each example are updated at the end of each boosting round using Equation 5.69.

Without boosting, the accuracy of the decision stump is, at best, 70%. With AdaBoost, the results of the predictions are given in Figure 5.39(b). The final prediction of the ensemble classifier is obtained by taking a weighted average of the predictions made by each base classifier, which is shown in the last row of Figure 5.39(b). Notice that AdaBoost perfectly classifies all the examples in the training data.

An important analytical result of boosting shows that the training error of the ensemble is bounded by the following expression:

$$e_{\text{ensemble}} \le \prod_i \left[\sqrt{\epsilon_i(1-\epsilon_i)}\right], \qquad (5.70)$$

Algorithm 5.7 AdaBoost algorithm.

```
1: $\mathbf{w} = \{w_j = 1/N \mid j = 1, 2, \ldots, N\}$.   {Initialize the weights for all $N$ examples.}
2: Let $k$ be the number of boosting rounds.
3: for $i = 1$ to $k$ do
4:     Create training set $D_i$ by sampling (with replacement) from $D$ according to $\mathbf{w}$.
5:     Train a base classifier $C_i$ on $D_i$.
6:     Apply $C_i$ to all examples in the original training set, $D$.
7:     $\epsilon_i = \frac{1}{N}\big[\sum_j w_j\ \delta\big(C_i(x_j) \neq y_j\big)\big]$   {Calculate the weighted error.}
8:     if $\epsilon_i > 0.5$ then
9:         $\mathbf{w} = \{w_j = 1/N \mid j = 1, 2, \ldots, N\}$.   {Reset the weights for all $N$ examples.}
10:        Go back to Step 4.
11:    end if
12:    $\alpha_i = \frac{1}{2} \ln \frac{1-\epsilon_i}{\epsilon_i}$.
13:    Update the weight of each example according to Equation 5.69.
14: end for
15: $C^*(\mathbf{x}) = \underset{y}{\operatorname{argmax}} \sum_{j=1}^{T} \alpha_j \delta\big(C_j(\mathbf{x}) = y\big)\big)$.
```

where ϵ_i is the error rate of each base classifier i. If the error rate of the base classifier is less than 50%, we can write $\epsilon_i = 0.5 - \gamma_i$, where γ_i measures how much better the classifier is than random guessing. The bound on the training error of the ensemble becomes

$$e_{\text{ensemble}} \leq \prod_i \sqrt{1 - 4\gamma_i^2} \leq \exp\left(-2 \sum_i \gamma_i^2 \right). \tag{5.71}$$

If $\gamma_i < \gamma*$ for all i's, then the training error of the ensemble decreases exponentially, which leads to the fast convergence of the algorithm. Nevertheless, because of its tendency to focus on training examples that are wrongly classified, the boosting technique can be quite susceptible to overfitting.

5.6.6 Random Forests

Random forest is a class of ensemble methods specifically designed for decision tree classifiers. It combines the predictions made by multiple decision trees, where each tree is generated based on the values of an independent set of random vectors, as shown in Figure 5.40. The random vectors are generated from a fixed probability distribution, unlike the adaptive approach used in AdaBoost, where the probability distribution is varied to focus on examples that are hard to classify. Bagging using decision trees is a special case of random forests, where randomness is injected into the model-building process

Boosting Round 1:

x	0.1	0.4	0.5	0.6	0.6	0.7	0.7	0.7	0.8	1
y	1	-1	-1	-1	-1	-1	-1	-1	1	1

Boosting Round 2:

x	0.1	0.1	0.2	0.2	0.2	0.2	0.3	0.3	0.3	0.3
y	1	1	1	1	1	1	1	1	1	1

Boosting Round 3:

x	0.2	0.2	0.4	0.4	0.4	0.4	0.5	0.6	0.6	0.7
y	1	1	-1	-1	-1	-1	-1	-1	-1	-1

(a) Training records chosen during boosting

Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	0.311	0.311	0.311	0.01	0.01	0.01	0.01	0.01	0.01	0.01
3	0.029	0.029	0.029	0.228	0.228	0.228	0.228	0.009	0.009	0.009

(b) Weights of training records

Figure 5.38. Example of boosting.

by randomly choosing N samples, with replacement, from the original training set. Bagging also uses the same uniform probability distribution to generate its bootstrapped samples throughout the entire model-building process.

It was theoretically proven that the upper bound for generalization error of random forests converges to the following expression, when the number of trees is sufficiently large.

$$\text{Generalization error} \leq \frac{\overline{\rho}(1-s^2)}{s^2}, \tag{5.72}$$

where $\overline{\rho}$ is the average correlation among the trees and s is a quantity that measures the "strength" of the tree classifiers. The strength of a set of classifiers refers to the average performance of the classifiers, where performance is measured probabilistically in terms of the classifier's margin:

$$\text{margin}, M(\mathbf{X}, Y) = P(\hat{Y}_\theta = Y) - \max_{Z \neq Y} P(\hat{Y}_\theta = Z), \tag{5.73}$$

where $\hat{Y}_\theta$ is the predicted class of $\mathbf{X}$ according to a classifier built from some random vector θ. The higher the margin is, the more likely it is that the

Round	Split Point	Left Class	Right Class	α
1	0.75	-1	1	1.738
2	0.05	1	1	2.7784
3	0.3	1	-1	4.1195

(a)

Round	x=0.1	x=0.2	x=0.3	x=0.4	x=0.5	x=0.6	x=0.7	x=0.8	x=0.9	x=1.0
1	-1	-1	-1	-1	-1	-1	-1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	-1	-1	-1	-1	-1	-1	-1
Sum	5.16	5.16	5.16	-3.08	-3.08	-3.08	-3.08	0.397	0.397	0.397
Sign	1	1	1	-1	-1	-1	-1	1	1	1

(b)

Figure 5.39. Example of combining classifiers constructed using the AdaBoost approach.

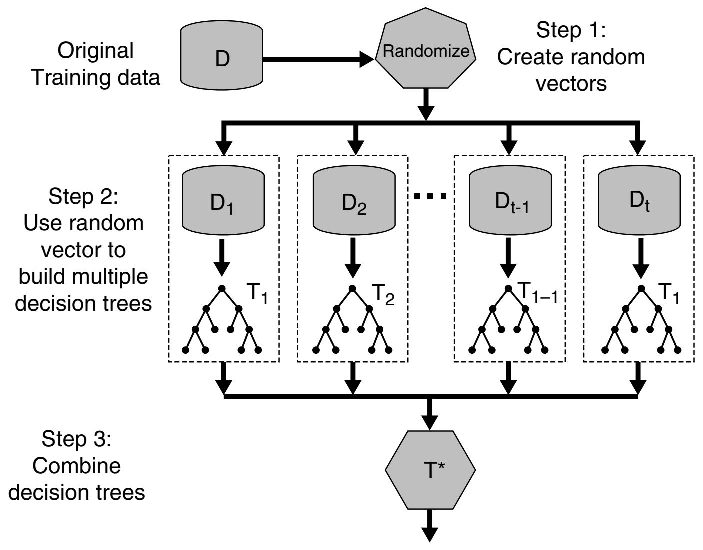

Original Training data
D
Randomize
Step 1: Create random vectors
Step 2: Use random vector to build multiple decision trees
D_1
D_2
D_{t-1}
D_t
T_1
T_2
T_{1-1}
T_1
Step 3: Combine decision trees
T*

Figure 5.40. Random forests.

classifier correctly predicts a given example $\mathbf{X}$. Equation 5.72 is quite intuitive; as the trees become more correlated or the strength of the ensemble decreases, the generalization error bound tends to increase. Randomization helps to reduce the correlation among decision trees so that the generalization error of the ensemble can be improved.

Each decision tree uses a random vector that is generated from some fixed probability distribution. A random vector can be incorporated into the tree-growing process in many ways. The first approach is to randomly select F input features to split at each node of the decision tree. As a result, instead of examining all the available features, the decision to split a node is determined from these selected F features. The tree is then grown to its entirety without any pruning. This may help reduce the bias present in the resulting tree. Once the trees have been constructed, the predictions are combined using a majority voting scheme. This approach is known as Forest-RI, where RI refers to random input selection. To increase randomness, bagging can also be used to generate bootstrap samples for Forest-RI. The strength and correlation of random forests may depend on the size of F. If F is sufficiently small, then the trees tend to become less correlated. On the other hand, the strength of the tree classifier tends to improve with a larger number of features, F. As a tradeoff, the number of features is commonly chosen to be $F = \log_2 d + 1$, where d is the number of input features. Since only a subset of the features needs to be examined at each node, this approach helps to significantly reduce the runtime of the algorithm.

If the number of original features d is too small, then it is difficult to choose an independent set of random features for building the decision trees. One way to increase the feature space is to create linear combinations of the input features. Specifically, at each node, a new feature is generated by randomly selecting L of the input features. The input features are linearly combined using coefficients generated from a uniform distribution in the range of $[-1, 1]$. At each node, F of such randomly combined new features are generated, and the best of them is subsequently selected to split the node. This approach is known as Forest-RC.

A third approach for generating the random trees is to randomly select one of the F best splits at each node of the decision tree. This approach may potentially generate trees that are more correlated than Forest-RI and Forest-RC, unless F is sufficiently large. It also does not have the runtime savings of Forest-RI and Forest-RC because the algorithm must examine all the splitting features at each node of the decision tree.

It has been shown empirically that the classification accuracies of random forests are quite comparable to the AdaBoost algorithm. It is also more robust to noise and runs much faster than the AdaBoost algorithm. The classification accuracies of various ensemble algorithms are compared in the next section.

Table 5.5. Comparing the accuracy of a decision tree classifier against three ensemble methods.

Data Set	Number of (Attributes, Classes, Records)	Decision Tree (%)	Bagging (%)	Boosting (%)	RF (%)
Anneal	(39, 6, 898)	92.09	94.43	95.43	95.43
Australia	(15, 2, 690)	85.51	87.10	85.22	85.80
Auto	(26, 7, 205)	81.95	85.37	85.37	84.39
Breast	(11, 2, 699)	95.14	96.42	97.28	96.14
Cleve	(14, 2, 303)	76.24	81.52	82.18	82.18
Credit	(16, 2, 690)	85.8	86.23	86.09	85.8
Diabetes	(9, 2, 768)	72.40	76.30	73.18	75.13
German	(21, 2, 1000)	70.90	73.40	73.00	74.5
Glass	(10, 7, 214)	67.29	76.17	77.57	78.04
Heart	(14, 2, 270)	80.00	81.48	80.74	83.33
Hepatitis	(20, 2, 155)	81.94	81.29	83.87	83.23
Horse	(23, 2, 368)	85.33	85.87	81.25	85.33
Ionosphere	(35, 2, 351)	89.17	92.02	93.73	93.45
Iris	(5, 3, 150)	94.67	94.67	94.00	93.33
Labor	(17, 2, 57)	78.95	84.21	89.47	84.21
Led7	(8, 10, 3200)	73.34	73.66	73.34	73.06
Lymphography	(19, 4, 148)	77.03	79.05	85.14	82.43
Pima	(9, 2, 768)	74.35	76.69	73.44	77.60
Sonar	(61, 2, 208)	78.85	78.85	84.62	85.58
Tic-tac-toe	(10, 2, 958)	83.72	93.84	98.54	95.82
Vehicle	(19, 4, 846)	71.04	74.11	78.25	74.94
Waveform	(22, 3, 5000)	76.44	83.30	83.90	84.04
Wine	(14, 3, 178)	94.38	96.07	97.75	97.75
Zoo	(17, 7, 101)	93.07	93.07	95.05	97.03

5.6.7 Empirical Comparison among Ensemble Methods

Table 5.5 shows the empirical results obtained when comparing the performance of a decision tree classifier against bagging, boosting, and random forest. The base classifiers used in each ensemble method consist of fifty decision trees. The classification accuracies reported in this table are obtained from ten-fold cross-validation. Notice that the ensemble classifiers generally outperform a single decision tree classifier on many of the data sets.

5.7 Class Imbalance Problem

Data sets with imbalanced class distributions are quite common in many real applications. For example, an automated inspection system that monitors products that come off a manufacturing assembly line may find that the number of defective products is significantly fewer than that of non-defective products. Similarly, in credit card fraud detection, fraudulent transactions are outnumbered by legitimate transactions. In both of these examples, there is a disproportionate number of instances that belong to different classes. The degree of imbalance varies from one application to another—a manufacturing plant operating under the six sigma principle may discover four defects in a million products shipped to their customers, while the amount of credit card fraud may be of the order of 1 in 100. Despite their infrequent occurrences, a correct classification of the rare class in these applications often has greater value than a correct classification of the majority class. However, because the class distribution is imbalanced, this presents a number of problems to existing classification algorithms.

The accuracy measure, which is used extensively to compare the performance of classifiers, may not be well suited for evaluating models derived from imbalanced data sets. For example, if 1% of the credit card transactions are fraudulent, then a model that predicts every transaction as legitimate has an accuracy of 99% even though it fails to detect any of the fraudulent activities. Additionally, measures that are used to guide the learning algorithm (e.g., information gain for decision tree induction) may need to be modified to focus on the rare class.

Detecting instances of the rare class is akin to finding a needle in a haystack. Because their instances occur infrequently, models that describe the rare class tend to be highly specialized. For example, in a rule-based classifier, the rules extracted for the rare class typically involve a large number of attributes and cannot be easily simplified into more general rules with broader coverage (unlike the rules for the majority class). Such models are also susceptible to the presence of noise in training data. As a result, many of the existing classification algorithms may not effectively detect instances of the rare class.

This section presents some of the methods developed for handling the class imbalance problem. First, alternative metrics besides accuracy are introduced, along with a graphical method called ROC analysis. We then describe how cost-sensitive learning and sampling-based methods may be used to improve the detection of rare classes.

5.7.1 Alternative Metrics

Since the accuracy measure treats every class as equally important, it may not be suitable for analyzing imbalanced data sets, where the rare class is considered more interesting than the majority class. For binary classification, the rare class is often denoted as the positive class, while the majority class is

Table 5.6. A confusion matrix for a binary classification problem in which the classes are not equally important.

		Predicted Class	
		$+$	$-$
Actual	$+$	f_{++} (TP)	f_{+-} (FN)
Class	$-$	f_{-+} (FP)	f_{--} (TN)

denoted as the negative class. A confusion matrix that summarizes the number of instances predicted correctly or incorrectly by a classification model is shown in Table 5.6.

The following terminology is often used when referring to the counts tabulated in a confusion matrix:

- True positive (TP) or f_{++}, which corresponds to the number of positive examples correctly predicted by the classification model.
- False negative (FN) or f_{+-}, which corresponds to the number of positive examples wrongly predicted as negative by the classification model.
- False positive (FP) or f_{-+}, which corresponds to the number of negative examples wrongly predicted as positive by the classification model.
- True negative (TN) or f_{--}, which corresponds to the number of negative examples correctly predicted by the classification model.

The counts in a confusion matrix can also be expressed in terms of percentages. The **true positive rate** (TPR) or **sensitivity** is defined as the fraction of positive examples predicted correctly by the model, i.e.,

$$TPR = TP/(TP + FN).$$

Similarly, the **true negative rate** (TNR) or **specificity** is defined as the fraction of negative examples predicted correctly by the model, i.e.,

$$TNR = TN/(TN + FP).$$

Finally, the **false positive rate** (FPR) is the fraction of negative examples predicted as a positive class, i.e.,

$$FPR = FP/(TN + FP),$$

while the **false negative rate** (FNR) is the fraction of positive examples predicted as a negative class, i.e.,

$$FNR = FN/(TP + FN).$$

Recall and **precision** are two widely used metrics employed in applications where successful detection of one of the classes is considered more significant than detection of the other classes. A formal definition of these metrics is given below.

$$\text{Precision, } p = \frac{TP}{TP + FP} \tag{5.74}$$

$$\text{Recall, } r = \frac{TP}{TP + FN} \tag{5.75}$$

Precision determines the fraction of records that actually turns out to be positive in the group the classifier has declared as a positive class. The higher the precision is, the lower the number of false positive errors committed by the classifier. Recall measures the fraction of positive examples correctly predicted by the classifier. Classifiers with large recall have very few positive examples misclassified as the negative class. In fact, the value of recall is equivalent to the true positive rate.

It is often possible to construct baseline models that maximize one metric but not the other. For example, a model that declares every record to be the positive class will have a perfect recall, but very poor precision. Conversely, a model that assigns a positive class to every test record that matches one of the positive records in the training set has very high precision, but low recall. Building a model that maximizes both precision and recall is the key challenge of classification algorithms.

Precision and recall can be summarized into another metric known as the F_1 measure.

$$F_1 = \frac{2rp}{r + p} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{5.76}$$

In principle, F_1 represents a harmonic mean between recall and precision, i.e.,

$$F_1 = \frac{2}{\frac{1}{r} + \frac{1}{p}}.$$

The harmonic mean of two numbers x and y tends to be closer to the smaller of the two numbers. Hence, a high value of F_1-measure ensures that both precision and recall are reasonably high. A comparison among harmonic, geometric, and arithmetic means is given in the next example.

Example 5.8. Consider two positive numbers $a = 1$ and $b = 5$. Their arithmetic mean is $\mu_a = (a + b)/2 = 3$ and their geometric mean is $\mu_g = \sqrt{ab} = 2.236$. Their harmonic mean is $\mu_h = (2\times1\times5)/6 = 1.667$, which is closer to the smaller value between a and b than the arithmetic and geometric means. ■

More generally, the F_β measure can be used to examine the tradeoff between recall and precision:

$$F_\beta = \frac{(\beta^2 + 1)rp}{r + \beta^2 p} = \frac{(\beta^2 + 1) \times TP}{(\beta^2 + 1)TP + \beta^2 FP + FN}. \tag{5.77}$$

Both precision and recall are special cases of F_β by setting $\beta = 0$ and $\beta = \infty$, respectively. Low values of β make F_β closer to precision, and high values make it closer to recall.

A more general metric that captures F_β as well as accuracy is the weighted accuracy measure, which is defined by the following equation:

$$\text{Weighted accuracy} = \frac{w_1 TP + w_4 TN}{w_1 TP + w_2 FP + w_3 FN + w_4 TN}. \tag{5.78}$$

The relationship between weighted accuracy and other performance metrics is summarized in the following table:

Measure	w_1	w_2	w_3	w_4
Recall	1	1	0	0
Precision	1	0	1	0
F_β	$\beta^2 + 1$	β^2	1	0
Accuracy	1	1	1	1

5.7.2 The Receiver Operating Characteristic Curve

A receiver operating characteristic (ROC) curve is a graphical approach for displaying the tradeoff between true positive rate and false positive rate of a classifier. In an ROC curve, the true positive rate (TPR) is plotted along the y axis and the false positive rate (FPR) is shown on the x axis. Each point along the curve corresponds to one of the models induced by the classifier. Figure 5.41 shows the ROC curves for a pair of classifiers, M_1 and M_2.

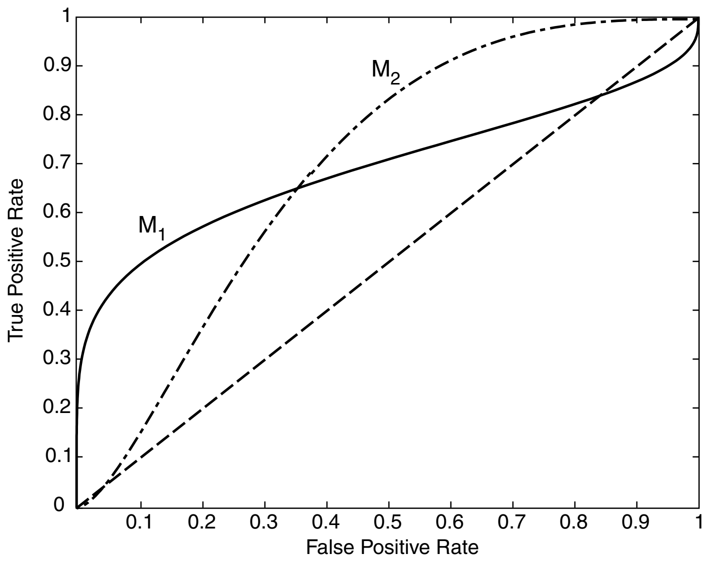


Figure 5.41. ROC curves for two different classifiers.

There are several critical points along an ROC curve that have well-known interpretations:

(TPR=0, FPR=0):	Model predicts every instance to be a negative class.
(TPR=1, FPR=1):	Model predicts every instance to be a positive class.
(TPR=1, FPR=0):	The ideal model.

A good classification model should be located as close as possible to the upper left corner of the diagram, while a model that makes random guesses should reside along the main diagonal, connecting the points $(TPR = 0, FPR = 0)$ and $(TPR = 1, FPR = 1)$. Random guessing means that a record is classified as a positive class with a fixed probability p, irrespective of its attribute set. For example, consider a data set that contains n_+ positive instances and n_- negative instances. The random classifier is expected to correctly classify pn_+ of the positive instances and to misclassify pn_- of the negative instances. Therefore, the TPR of the classifier is $(pn_+)/n_+ = p$, while its FPR is $(pn_-)/p = p$. Since the TPR and FPR are identical, the ROC curve for a random classifier always reside along the main diagonal.

An ROC curve is useful for comparing the relative performance among different classifiers. In Figure 5.41, M_1 is better than M_2 when FPR is less than 0.36, while M_2 is superior when FPR is greater than 0.36. Clearly, neither of these two classifiers dominates the other.

The area under the ROC curve (AUC) provides another approach for evaluating which model is better on average. If the model is perfect, then its area under the ROC curve would equal 1. If the model simply performs random guessing, then its area under the ROC curve would equal 0.5. A model that is strictly better than another would have a larger area under the ROC curve.

Generating an ROC curve

To draw an ROC curve, the classifier should be able to produce a continuous-valued output that can be used to rank its predictions, from the most likely record to be classified as a positive class to the least likely record. These outputs may correspond to the posterior probabilities generated by a Bayesian classifier or the numeric-valued outputs produced by an artificial neural network. The following procedure can then be used to generate an ROC curve:

1. Assuming that the continuous-valued outputs are defined for the positive class, sort the test records in increasing order of their output values.

2. Select the lowest ranked test record (i.e., the record with lowest output value). Assign the selected record and those ranked above it to the positive class. This approach is equivalent to classifying all the test records as positive class. Because all the positive examples are classified correctly and the negative examples are misclassified, $TPR = FPR = 1$.

3. Select the next test record from the sorted list. Classify the selected record and those ranked above it as positive, while those ranked below it as negative. Update the counts of TP and FP by examining the actual class label of the previously selected record. If the previously selected record is a positive class, the TP count is decremented and the FP count remains the same as before. If the previously selected record is a negative class, the FP count is decremented and TP count remains the same as before.

4. Repeat Step 3 and update the TP and FP counts accordingly until the highest ranked test record is selected.

5. Plot the TPR against FPR of the classifier.

Figure 5.42 shows an example of how to compute the ROC curve. There are five positive examples and five negative examples in the test set. The class