Chapter 4
Inter-Class Boundaries: Linear and Polynomial Classifiers

When representing the training examples with points in an n-dimensional instance space, we may realize that positive examples tend to be clustered in regions different from those occupied by negative examples. This observation motivates yet another approach to classification. Instead of the probabilities and similarities employed by the earlier paradigms, we can try to identify the *decision surface* that separates the two classes. A very simple possibility is to use to this end a linear function. More flexible are high-order polynomials which are capable of defining very complicated inter-class boundaries. These, however, have to be handled with care.

The chapter introduces two simple mechanisms for induction of *linear classifiers* from examples described by boolean attributes, and then discusses how to use them in more general domains such as those with numeric attributes and more than just two classes. The whole idea is then extended to *polynomial classifiers*.

4.1 The Essence

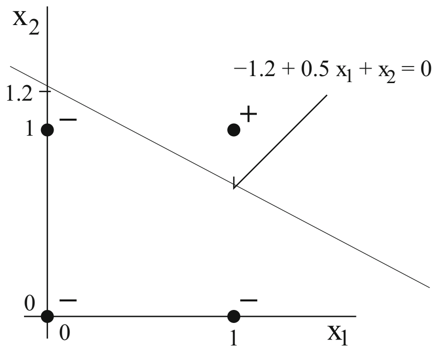
To begin with, let us constrain ourselves to boolean domains where each attribute is either *true* or *false*. To make it possible for these attributes to participate in algebraic functions, we will represent them by integers: *true* by 1, and *false* by 0.

The Linear Classifier In Fig. 4.1, one example is labeled as positive and the remaining three as negative. In this particular case, the two classes can be separated by the linear function defined by the following equation:

$$-1.2 + 0.5x_1 + x_2 = 0 \tag{4.1}$$

In the expression on the left-hand side, x_1 and x_2 represent attributes. If we substitute for x_1 and x_2 the concrete values of a given example, the expression, $-1.2 + 0.5x_1 + x_2$, will become either positive or negative. This sign then determines

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-63913-0_4



x_1	x_2	$-1.2+0.5x_1 + x_2$	Class
1	1	0.3	pos
1	0	−0.7	neg
0	1	−0.2	neg
0	0	−1.2	neg

Fig. 4.1 A linear classifier in a domain with two classes and two boolean attributes (using 1 instead of *true* and 0 instead of *false*)

the example's class. The table on the right shows how the four examples from the left have thus been classified.

Equation (4.1) is not the only one capable of doing the job. Other expressions, say, $-1.5 + x_1 + x_2$, will label the four examples in exactly the same way. As a matter of fact, the same can be accomplished by infinitely many different classifiers of the following generic form:

$$w_0 + w_1x_1 + w_2x_2 = 0$$

This is easy to generalize to domains with n attributes:

$$w_0 + w_1x_1 + \ldots + w_nx_n = 0 \tag{4.2}$$

If $n = 2$, Eq. (4.2) defines a line; if $n = 3$, a plane; and if $n > 3$, a hyperplane. By the way, if we artificially introduce a "zeroth" attribute, x_0, whose value is always fixed at $x_0 = 1$, the equation can be re-written in the following, more compact, form:

$$\sum_{i=0}^{n} w_ix_i = 0 \tag{4.3}$$

Some Practical Issues When writing a computer program implementing the classifier, the engineer must not forget to decide how to label the rare example that finds itself exactly on the hyperplane—which happens when the expression equals 0. Common practice either chooses the class randomly or gives preference to the one that has more representatives in the training set.

Also, we must not forget that no linear classifier can ever separate the positive and the negative examples if the two classes are *not* linearly separable. Thus if we change in Fig. 4.1 the class label of $\mathbf{x} = (x_1, x_2) = (0, 0)$ from minus to plus, no straight line will ever succeed. Let us defer further discussion of this issue till Sect. 4.5. For the time being, we will consider only domains where the classes *are* linearly separable.

The Parameters The classifier's behavior is determined by the coefficients, w_i. These are usually called *weights*. The task for machine learning is to find their appropriate values.

Not all the weights play the same role. Geometrically speaking, the coefficients $w_1, \ldots w_n$ define the angle of the hyperplane with respect to the system of coordinates; and the last, w_0, called *bias*, determines the *offset*, the hyperplane's distance from the origin of the system of coordinates.

The Bias and the Threshold In the case depicted in Fig. 4.1, the bias was $w_0 = -1.2$. A higher value would "shift" the classifier further from the origin, $[0, 0]$, whereas $w_0 = 0$ would make the classifier intersect the origin. Our intuitive grasp of the role played by bias in the classifier's behavior will improve if we re-write Eq. (4.2) as follows:

$$w_1x_1 + \ldots w_nx_n = \theta \tag{4.4}$$

The term on the right-hand side, $\theta = -w_0$, is the *threshold* that the weighted sum has to exceed if the example is to be deemed positive. Note that the threshold equals the negatively taken bias.

Simple Logical Functions Let us simplify our considerations by the (somewhat extreme) requirement that all attributes should have the same weight, $w_i = 1$. Even under this constraint, careful choice of the threshold will implement some useful functions. For instance, the reader will easily verify that the following classifier returns the positive class if and only if every single attribute has $x_i = 1$, a situation known as logical AND.

$$x_1 + \ldots + x_n = n - 0.5 \tag{4.5}$$

By contrast, the next classifier returns the positive class if *at least one* attribute is $x_i = 1$, a situation known as logical OR.

$$x_1 + \ldots + x_n = 0.5 \tag{4.6}$$

Finally, the classifier below returns the positive class if *at least k* attributes (out of the total of *n* attributes) are $x_i = 1$. This represents the so-called *k-of-n* function, of which AND and OR are special cases: AND is *n-of-n*, whereas OR is *1-of-n*.

$$x_1 + \ldots + x_n = k - 0.5 \tag{4.7}$$

Weights Now that we understand the impact of bias, let us abandon the restriction that all weights be equal, and take a look at the consequences of their concrete values. Consider the linear classifier defined by the following function:

$$2 + 3x_1 - 2x_2 + 0.1x_4 - 0.5x_6 = 0 \tag{4.8}$$

The first thing to notice in the expression on the left side is the absence of attributes x_3 and x_5. These are rendered *irrelevant* by their zero weights, $w_3 = w_5 = 0$. As for the other attributes, their impacts depend on their weights' absolute values as well as on the signs: if $w_i > 0$, then $x_i = 1$ increases the chances of the above expression being positive; and if $w_i < 0$, then $x_i = 1$ increases the chances of the expression being negative. Note that, in the case of the classifier defined by Eq. (4.8), x_1 supports the positive class more strongly than x_4 because $w_1 > w_4$. Likewise, the influence of x_2 is stronger than that of x_6—only in the opposite direction: reducing the value of the overall sum, this weight makes it more likely that an example with $x_2 = 1$ will be deemed negative. Finally, the very small value of w_4 renders attribute x_4 almost irrelevant.

As another example, consider the classifier defined by the following function:

$$2x_1 + x_2 + x_3 = 1.5 \tag{4.9}$$

Here the threshold 1.5 is exceeded either by the sole presence of $x_1 = 1$ (because then $2x_1 = 2 \cdot 1 > 1.5$) or by the combined contributions of $x_2 = 1$ and $x_3 = 1$. This means that x_1 will "outvote" the other two attributes even when x_2 and x_3 join their forces in the support of the negative class.

Low Computational Costs Note the relatively low computational costs of this approach. Whereas the 1-NN classifier had to evaluate many geometric distances, and then search for the smallest among them, the linear classifier only has to determine the sign of a relatively simple expression.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Write the general expression defining the linear classifier in a domain with four boolean attributes. Why do we prefer to represent the *true* and *false* values by 1 and 0, respectively? How does the classifier determine an example's class?
- How can a linear classifier implement functions AND, OR, and *k-of-n*?
- Explain and discuss the behavior of the linear classifier defined by the expression, $-2.5 + x_2 + 2x_3 = 0$. What do the weights tell us about the role of the individual attributes?
- Compare the computational costs of the linear classifier with those of the nearest-neighbor classifier.

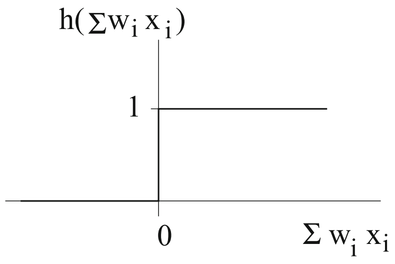


Fig. 4.2 The linear classifier outputs $h(\mathbf{x}) = 1$ when $\sum_{i=0}^{n} w_i x_i > 0$ and $h(\mathbf{x}) = 0$ when $\sum_{i=0}^{n} w_i x_i \leq 0$, signaling the example is positive or negative, respectively

4.2 The Additive Rule: Perceptron Learning

Having developed some understanding of how the linear classifier works, we are ready to take a closer look at how to induce the tool from training data.

The Learning Task We will assume that each training example, $\mathbf{x}$, is described by n binary attributes whose values are either $x_i = 1$ or $x_i = 0$. A positive example is labeled with $c(\mathbf{x}) = 1$, and a negative with $c(\mathbf{x}) = 0$. To make sure we never confuse an example's real class with the one returned by the classifier, we will denote the latter by $h(\mathbf{x})$ where h indicates that this is the classifier's *h*ypothesis. If $\sum_{i=0}^{n} w_i x_i > 0$, the classifier "hypothesizes" that the example is positive, and therefore returns $h(\mathbf{x}) = 1$. Conversely, if $\sum_{i=0}^{n} w_i x_i \leq 0$, the classifier returns $h(\mathbf{x}) = 0$. Figure 4.2 serves as a reminder that the classifier labels $\mathbf{x}$ as positive only if the cumulative evidence in favor of this class exceeds 0.

Finally, we will suppose that examples with $c(\mathbf{x}) = 1$ are linearly separable from those with $c(\mathbf{x}) = 0$. This means that there exists a linear classifier that will label correctly all training examples, $h(\mathbf{x}) = c(\mathbf{x})$. The task for machine learning is to find weights, w_i, that will make this happen.

Learning from Mistakes Here is the essence of the most common approach to the induction of linear classifiers. Suppose we already have an interim (even if imperfect) version of the classifier. When presented with a training example, $\mathbf{x}$, the classifier returns its label, $h(\mathbf{x})$. If this differs from the true class, $h(\mathbf{x}) \neq c(\mathbf{x})$, it is because the weights are less than perfect; they thus have to be modified in a way likely to correct this error.

Here is how to go about the weight modification. Let the true class be $c(\mathbf{x}) = 1$. In this event, $h(\mathbf{x}) = 0$ will only happen if $\sum_{i=0}^{n} w_i x_i < 0$, an indication that the weights are too small. If we *increase* the weights, then the sum, $\sum_{i=0}^{n} w_i x_i$, may exceed zero, making the returned label positive, and thus correct. Note that it is enough to increase only the weights of attributes with $x_i = 1$; when $x_i = 0$, then the value of w_i does not matter because anything multiplied by zero is still zero: $0 \cdot w_i = 0$.

Similarly, if $c(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 1$, then the weights of all attributes such that $x_i = 1$ should be *decreased* so as to give the sum the chance to drop below zero, $\sum_{i=0}^{n} w_i x_i < 0$.

The Weight-Adjusting Formula In summary, the presentation of a training example, $\mathbf{x}$, can result in three different situations. The technique based on "learning from mistakes" responds to them according to the following table:

Situation	Action
$c(\mathbf{x}) = 1$ while $h(\mathbf{x}) = 0$	Increase w_i for each attribute with $x_i = 1$
$c(\mathbf{x}) = 0$ while $h(\mathbf{x}) = 1$	Decrease w_i for each attribute with $x_i = 1$
$c(\mathbf{x}) = h(\mathbf{x})$	Do nothing

Interestingly, all these actions are carried out by a single formula:

$$w_i = w_i + \eta \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i \tag{4.10}$$

Let us take a look at the basic aspects of this weight-adjusting formula.

1. *Correct action*. If $c(\mathbf{x}) = h(\mathbf{x})$, the term in the brackets is $[c(\mathbf{x}) - h(\mathbf{x})] = 0$, which leaves w_i unchanged. If $c(\mathbf{x}) = 1$ and $h(\mathbf{x}) = 0$, the term in the brackets is 1, and the weights are thus increased. And if $c(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 1$, the term in the brackets is negative, and the weights are reduced.
2. *Affecting only relevant weights*. If $x_i = 0$, the term to be added to the i-th weight, $\eta \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot 0$, is zero. This means that the formula will affect w_i only when $x_i = 1$.
3. *Amount of change*. This is controlled by the *learning rate*, η, whose user-set value is chosen from the interval $\eta \in (0, 1]$.

Note that the modification of the weights is *additive* in the sense that a term is added to the previous value of the weight. In Sect. 4.3, we will discuss the other possibility: a *multiplicative* formula.

The Perceptron Learning Algorithm Equation (4.10) forms the core of the *Perceptron Learning Algorithm*.[1] The procedure is summarized by the pseudocode in Table 4.1. The principle is simple. Once the weights have been initialized to small random values, the training examples are presented, one at a time. After each example presentation, every weight of the classifier is subjected to Eq. (4.10). The last training example signals that one *training epoch* has been completed. Unless the classifier now labels correctly the entire training set, the learner returns to the first example, thus beginning the second epoch, then the third, and so on. Typically, several epochs are needed to reach the goal.

A Numeric Example Table 4.2 illustrates the procedure on a toy domain where the three training examples, $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$, are described by two binary attributes. After the presentation of $\mathbf{e}_1$, the weights (originally random) are reduced on account

[1]Its author, M. Rosenblatt, originally employed this learning technique in a device he called a *Perceptron*.

Table 4.1 The *perceptron learning* algorithm

Assumption: the two classes, $c(\mathbf{x}) = 1$ and $c(\mathbf{x}) = 0$, are linearly separable.

```
1. Initialize all weights, $w_i$, to small random numbers.
   Choose an appropriate learning rate, $\eta \in (0, 1]$.
2. For each training example, $\mathbf{x} = (x_1, \ldots, x_n)$, whose class is $c(\mathbf{x})$:
       (i) Let $h(\mathbf{x}) = 1$ if $\sum_{i=0}^{n} w_i x_i > 0$, and $h(\mathbf{x}) = 0$ otherwise.
       (ii) Update each weight using the formula, $w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i$
3. If $c(\mathbf{x}) = h(\mathbf{x})$ for all training examples, stop; otherwise, return to step 2.
```

Table 4.2 Illustration of *perceptron learning*

Let the learning rate be $\eta = 0.5$, and let the (randomly generated) initial weights be $w_0 = 0.1, w_1 = 0.3$, and $w_3 = 0.4$. Set $x_0 = 1$.

Task: Using the following training set, the perceptron learning algorithm is to learn how to separate the negative examples, $\mathbf{e}_1$ and $\mathbf{e}_3$, from the positive example, $\mathbf{e}_2$.

Example	x_1	x_2	$c(\mathbf{x})$
$\mathbf{e}_1$	1	0	0
$\mathbf{e}_2$	1	1	1
$\mathbf{e}_3$	0	0	0

The linear classifier's hypothesis about $\mathbf{x}$'s class is $h(\mathbf{x}) = 1$ if $\sum_{i=0}^{n} w_i x_i > 0$ and $h(\mathbf{x}) = 0$ otherwise. After each example presentation, all weights are subjected to the same formula: $w_i = w_i + 0.5 \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i$.

The table below shows, step by step, what happens to the weights in the course of learning.

	x_1	x_2	w_0	w_1	w_2	$h(\mathbf{x})$	$c(\mathbf{x})$	$c(\mathbf{x}) - h(\mathbf{x})$
Random classifier			0.1	0.3	0.4			
Example $\mathbf{e}_1$	1	0				1	0	−1
New classifier			−0.4	−0.2	0.4			
Example $\mathbf{e}_2$	1	1				0	1	1
New classifier			0.1	0.3	0.9			
Example $\mathbf{e}_3$	0	0				1	0	−1
Final classifier			−0.4	0.3	0.9			

The final version of the classifier, $-0.4 + 0.3x_1 + 0.9x_2 = 0$, no longer misclassifies any training example. The training has thus been completed in a single epoch.

of $h(\mathbf{e_1}) = 1$ and $c(\mathbf{e_1}) = 0$; however, this happens only to w_0 and w_1 because $x_2 = 0$. In response to $\mathbf{e}_2$, all weights of the classifier's new version are increased because $h(\mathbf{e_2}) = 0$ and $c(\mathbf{e_2}) = 1$, and all attributes have $x_i = 1$. And after $\mathbf{e}_3$, the fact that $h(\mathbf{e_3}) = 1$ and $c(\mathbf{e_1}) = 0$ results in the reduction of w_0, but not of the other weights because $x_1 = x_2 = 0$. From now on, the classifier correctly labels all training examples, and the process can thus be terminated.

Initial Weights and the Number of Attributes The training converged to the separating line in a single epoch, but this was only thanks to a few lucky choices that may have played quite a critical role. Let us discuss them briefly.

First of them is the set of (random) *initial weights*. Different initialization may result in a different number of epochs. Most of the time, the classifier's initial version will be almost useless, and a lot of training is needed before the process converges to something useful. Sometimes, however, the first version may be fairly good, and a single epoch will do the trick. And at the extreme case, there exists the possibility, however remote, that the random-number generator will create a classifier that labels all training examples without a single error, and no training is thus needed.

Another factor is the *length of the attribute vector*. As a rule of thumb, the number of the necessary epochs tends to grow linearly in the number of attributes (assuming the same learning rate, η, is used). For instance, the number of epochs needed in a domain with $3 \times n$ attributes is likely to be about three times the number of epochs that would be needed in a domain with n attributes.

Learning Rate A critical role is played by the *learning rate*, η. Returning to the example from Table 4.2, the reader will note the rapid weight changes. Thus w_0 "jumped" from 0.1 to –0.4 after $\mathbf{e}_1$, then back to 0.1 after $\mathbf{e}_2$, only to return to –0.4 after $\mathbf{e}_3$. Similar changes were experienced by w_1 and w_2. Figure 4.3 visualizes the phenomenon. The reader will easily verify that the four lines represent the four successive versions of the classifier. Note how dramatic, for instance, is the change from classifier 1 to classifier 2, and then from classifier 2 to classifier 3.

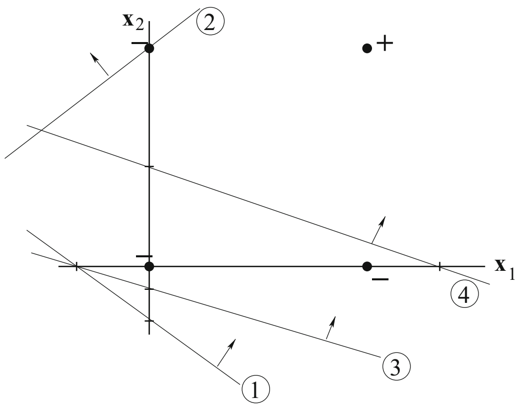


Fig. 4.3 The four classifiers from Table 4.2. The classifier defined by the initial weights is denoted by 1; numbers 2 and 3 represent the two intermediate stages; and 4, the final solution. The *arrows* indicate the half-space of positive examples

This remarkable sensitivity is explained by the high learning rate, $\eta = 0.5$. A smaller value, such as $\eta = 0.1$, would moderate the changes, thus "smoothing out" the learning. But if we overdo it by choosing, say, $\eta = 0.001$, the training process will become way too slow, and a great many epochs will have to pass before all training examples are correctly classified.

If the Solution Exists, It Will Be Found Whatever the initial weights, whatever the number of attributes, and whatever the learning rate, one thing is always guaranteed. If the positive and negative classes are linearly separable, the perceptron learning algorithm is guaranteed to find one version of the class-separating hyperplane in a finite number of steps.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Under what circumstances is *perceptron learning* guaranteed to find a classifier that perfectly labels all training examples?
- When does the algorithm reduce the classifier's weights, when does it increase them, and when does it leave them unchanged? Why does it modify w_i only if the corresponding attribute's value is $x_i = 1$?
- What circumstances influence the number of epochs needed by the algorithm to converge to a solution?

4.3 The Multiplicative Rule: WINNOW

Perceptron learning responds to the classifier's error by applying the *additive rule* that added to the weights a positive or negative term. An obvious alternative is the *multiplicative rule* where the weights are multiplied instead of being added to. Such approach has been adopted by WINNOW, an algorithm summarized by the pseudocode in Table 4.3.

The Principle and the Formula The general scenario is the same as in the previous section. A training example, $\mathbf{x}$, is presented, and the classifier returns its hypothesis about the example's label, $h(\mathbf{x})$. The learner compares this hypothesis with the known label, $c(\mathbf{x})$. If the two differ, $c(\mathbf{x}) \neq h(\mathbf{x})$, the weights of the attributes with $x_i = 1$ are modified in the following manner (where $\alpha > 1$ is a user-set parameter):

Situation	Action
$c(\mathbf{x}) = 1$ while $h(\mathbf{x}) = 0$	$w_i = \alpha w_i$
$c(\mathbf{x}) = 0$ while $h(\mathbf{x}) = 1$	$w_i = w_i/\alpha$
$c(\mathbf{x}) = h(\mathbf{x})$	Do nothing

Table 4.3 The WINNOW algorithm

Assumption: the two classes, $c(\mathbf{x}) = 1$ and $c(\mathbf{x}) = 0$, are linearly separable.

1. Initialize the classifier's weights to $w_i = 1$.
2. Set the threshold to $\theta = n - 0.1$ (n being the number of attributes) and choose an appropriate value for parameter $\alpha > 1$ (usually $\alpha = 2$).
3. Present a training example, $\mathbf{x}$, whose class is $c(\mathbf{x})$. The classifier returns $h(\mathbf{x})$.
4. If $c(\mathbf{x}) \neq h(\mathbf{x})$, update the weights of each attribute whose value is $x_i = 1$:

   if $c(\mathbf{x}) = 1$ and $h(\mathbf{x}) = 0$, then $w_i = \alpha w_i$
   if $c(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 1$, then $w_i = w_i/\alpha$

5. If $c(\mathbf{x}) = h(\mathbf{x})$ for all training examples, stop; otherwise, return to step 3.

The reader is encouraged to verify that all these three actions can be carried out by the same formula:

$$w_i = w_i \cdot \alpha^{c(\mathbf{x})-h(\mathbf{x})} \tag{4.11}$$

A Numeric Example Table 4.4 illustrates the principle using a simple toy domain. The training set consists of all possible examples that can be described by three binary attributes. Those with $x_2 = x_3 = 1$ are labeled as positive and all others as negative, regardless of the value of the (irrelevant) attribute x_1.

In *perceptron learning*, the weights were initialized to small random values. In the case of WINNOW, however, they are all initially set to 1. As for the threshold, $\theta = n - 0.1$ is used, slightly less than the number of attributes. In the toy domain from Table 4.4, this means $\theta = 3 - 0.1 = 2.9$ because WINNOW of course has no a priori knowledge of one of the attributes being irrelevant.

When the first four examples are presented, the classifier's initial version labels them all correctly. The first mistake is made in the case of $\mathbf{e}_5$: for this positive example, the classifier incorrectly returns the negative label. The learner therefore increases the weights of attributes with $x_i = 1$ (that is, w_2 and w_3). This new classifier then classifies correctly all the remaining examples, $\mathbf{e}_6$ through $\mathbf{e}_8$. In the second epoch, the classifier errs on $\mathbf{e}_2$, causing a false positive. In response to this error, the algorithm reduces weights w_1 and w_2 (but not w_3 because $x_3 = 0$). After this last weight modification, the classifier labels correctly the entire training set.[2]

[2]Similarly as in the case of *perceptron learning*, we could have considered the 0-th attribute, $x_0 = 1$, whose initial weight is $w_0 = 1$.

Table 4.4 Illustration of the WINNOW's behavior

Task. Using the training examples from the table on the left (below), induce the linear classifier. Let $\alpha = 2$, and let $\theta = 2.9$.

Note that the training is here accomplished in two learning steps: presentation of $\mathbf{e}_5$ (false negative), and of $\mathbf{e}_2$ (false positive). After these two weight modifications, the resulting classifier correctly classifies all training examples.

	x_1	x_2	x_3	c($\mathbf{x}$)
$\mathbf{e}_1$	1	1	1	1
$\mathbf{e}_2$	1	1	0	0
$\mathbf{e}_3$	1	0	1	0
$\mathbf{e}_4$	1	0	0	0
$\mathbf{e}_5$	0	1	1	1
$\mathbf{e}_6$	0	1	0	0
$\mathbf{e}_7$	0	0	1	0
$\mathbf{e}_8$	0	0	0	0

	x_1	x_2	x_3	w_1	w_2	w_3	$h(\mathbf{x})$	$c(\mathbf{x})$
Init. class.				1	1	1		
Example $\mathbf{e}_5$	0	1	1				0	1
New weights				1	2	2		
Example $\mathbf{e}_2$	1	1	0				1	0
New weights				0.5	1	2		

Note that the weight of the irrelevant attribute, x_1, is now smaller than the weights of the relevant attributes. Indeed, the ability to penalize irrelevant attributes by significantly reducing their weights, thus "winnowing them out," is one of the main advantages of this technique.

The "Alpha" Parameter Parameter α controls the learner's sensitivity to errors in a manner reminiscent of the learning rate in perceptron learning. The main difference is the requirement that $\alpha > 1$. This guarantees an increase in weight w_i in the case of a false negative, and a decrease in w_i in the case of a false positive. The parameter's concrete value is not completely arbitrary. If it exceeds 1 by just a little (say, if $\alpha = 1.1$), then the weight-updates will be very small, resulting in slow convergence. Increasing α's value accelerates convergence, but risks overshooting the solution. The ideal value is best established experimentally; good results are often achieved with $\alpha = 2$.

No Negative Weights? Let us point out one fundamental difference between WINNOW and perceptron learning. Since the (originally positive) weights are always multiplied by α or $1/\alpha$, none of them can ever drop to zero, let alone become negative. This means that, unless appropriate measures have been taken, a whole class of linear classifiers will thus be eliminated: those with negative or zero coefficients.

The shortcoming is removed if we represent each of the original attributes by a pair of "new" attributes: one copying the original attribute's value, the other having the opposite value. In a domain that originally had n attributes, the total number of attributes will then be $2n$, the value of the $(n + i)$th attribute, x_{n+i}, being the opposite of x_i. For instance, suppose that an example is described by the following three attribute values:

$$x_1 = 1, x_2 = 0, x_3 = 1$$

In the new representation, the same example will be described by six attributes:

$$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0,$$

For these, WINNOW will have to find six weights, $w_1, \ldots, w_6$, or perhaps seven, if w_0 is used.

Comparing It with Perceptron In comparison with *perceptron learning*, WINNOW appears to converge faster in domains with irrelevant attributes whose weights are quickly reduced to small values. However, neither WINNOW nor *perceptron learning* is able to recognize (and eliminate) *redundant attributes*. In the event of two attributes always having the same value, $x_i = x_j$, the learning process will converge to the same weight for both, making them look equally important even though it is clear that only one of them is strictly needed.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What formula is used by the weight-updating mechanism in WINNOW? Why is the formula called *multiplicative*?
- What is the shortcoming of multiplying or dividing the weights by $\alpha > 1$? How is the situation remedied?
- Summarize the differences between WINNOW and *perceptron learning*.

4.4 Domains with More Than Two Classes

Having only two sides, a hyperplane may separate the positive examples from the negative—and that's it; when it comes to *multi-class domains*, the tool seems helpless. Or is it?

Groups of Binary Classifiers What is beyond the powers of an individual can be solved by a team. One approach that is sometimes employed, in this context, is illustrated in Fig. 4.4. The "team" consists of four binary classifiers, each specializing on one of the four classes, C_1 through C_4. Ideally, the presentation of an example from C_i results in the i-th classifier returning $h_i(\mathbf{x}) = 1$, and all the other classifiers returning $h_j(\mathbf{x}) = 0$, assuming, again, that each class can be linearly separated from the other classes.

Modifying the Training Data To exhibit this behavior, however, the individual classifiers need to be properly trained. Here, one can rely on the algorithms that have been described in the previous sections. The only additional "trick" is that the engineer needs to modify the training data accordingly.

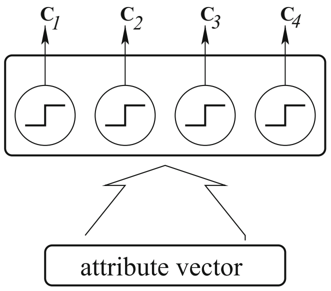


Fig. 4.4 Converting a 4-class problem into four 2-class problems

Table 4.5 A 4-class training set, T, converted to 4 binary training sets, $T_1 \dots T_4$

	T	T_1	T_2	T_3	T_4
e_1	C_2	0	1	0	0
e_2	C_1	1	0	0	0
e_3	C_3	0	0	1	0
e_4	C_4	0	0	0	1
e_5	C_2	0	1	0	0
e_6	C_4	0	0	0	1

Table 4.5 illustrates the principle. On the left is the original training set, T, where each example is labeled with one of the four classes. On the right are four "derived" sets, T_1 through T_4, each consisting of the same six examples which, however, have been re-labeled so that an example that in the original set, T, represents class C_i is labeled with $c(\mathbf{x}) = 1$ in T_i and with $c(\mathbf{x}) = 0$ in all other sets.

The Need for a Master Classifier The training sets, T_i, are presented to a learner which induces from each of them a linear classifier dedicated to the corresponding class. This is not the end of the story, though. The training examples may poorly represent the classes, they may be corrupted by noise, and even the requirement of linear separability may not be satisfied. As a result of these complications, the induced classifiers may "overlap" each other in the sense that two or more of them will respond to the same example, $\mathbf{x}$, with $h_i(\mathbf{x}) = 1$, leaving the wrong impression that $\mathbf{x}$ belongs to more than one class. This is why a "master classifier" is needed, its task being to choose from the returned classes the one most likely to be correct.

This is not difficult. The reader will recall that a linear classifier labels $\mathbf{x}$ as positive if the weighted sum of $\mathbf{x}$'s attribute values exceeds zero: $\Sigma_{i=0}^{n} w_i x_i > 0$. This sum (usually different in each of the classifiers that have returned $h_i(\mathbf{x}) = 1$) can be interpreted as the amount of evidence in support of the corresponding class. The master classifier then simply gives preference to the class whose binary classifier delivered the highest $\Sigma_{i=0}^{n} w_i x_i$.

A Numeric Example The principle is illustrated in Table 4.6. Here, each of the rows represents a different class (with the total of four classes). Of course, each

Table 4.6 Illustration of the master classifier's behavior: choosing the example's class from several candidates

Suppose we have four binary classifiers (the i-th classifier used for the i-th class) defined by the weights listed in the table below. How shall the master classifier label example $\mathbf{x} = (x_1, x_2, x_3, x_4) = (1, 1, 1, 0)$?

	Classifier						
Class	w_0	w_1	w_2	w_3	w_4	$\Sigma_{i=0}^{n} w_i x_i$	$h(\mathbf{x})$
C_1	−1.5	1	0.5	−1	−5	−1	0
C_2	0.5	1.5	−1	3	1	4	1
C_3	1	−2	4	−1	0.5	2	1
C_4	−2	1	1	−3	−1	−3	0

The rightmost column tells us that two classifiers, C_2 and C_3, return $h(\mathbf{x}) = 1$. From these, C_2 is supported by the higher value of $\Sigma_{i=0}^{n} w_i x_i$. Therefore, the master classifier labels $\mathbf{x}$ with C_2.

classifier has a different set of weights, each weight represented by one column in the table. When an example is presented, its attribute-values are in each classifier multiplied by the corresponding weights. We observe that in the case of two classifiers, C_2 and C_3, the weighted sums are positive, $\Sigma_{i=0}^{n} w_i x_i > 0$, which might mean that both classifiers return $h(\mathbf{x}) = 1$. Since each example is supposed to be labeled with one and only one class, we need a master classifier to make a decision. In this particular case, the master classifier gives preference to C_2 because this classifier's weighted sum is greater than that of C_3.

A Practical Limitation A little disclaimer is in place here. This method of employing linear classifiers in multi-class domains is reliable only if the number of classes is moderate, say, 3–5. In domains with many classes, the "derived" training sets, T_i, will be imbalanced in the sense that most examples will have $c(\mathbf{x}) = 0$ and only a few $c(\mathbf{x}) = 1$. As we will learn in Sect. 10.2, imbalanced training sets tend to cause difficulties in noisy domains unless appropriate measures have been taken.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- When trying to use N linear classifiers in an N-class domain, how will you create the training sets, T_i, for the induction of the individual binary classifiers?
- How can an example's class be decided upon in a situation where two or more binary classifiers return $h(\mathbf{x}) = 1$?

4.5 Polynomial Classifiers

It is now time to abandon the requirement that the positive examples be linearly separable from the negative; because fairly often, they are not. Not only can the linear separability be destroyed by noise. The very shape of the region occupied by one of the classes can make it impossible to separate it by a linear decision surface. Thus in the training set shown in Fig. 4.5, no linear classifier ever succeeds in separating the two classes, a feat that can only be accomplished by a non-linear curve such as the parabola shown in the picture.

Non-linear Classifiers The point having been made, we have to ask how to induce the more sophisticated *non-linear classifiers*. There is no doubt that they exist. For instance, math teaches us that *any* n-dimensional function can be approximated to arbitrary precision with a *polynomial* of a sufficiently high order. Let us therefore take a look at how to use—and induce—the polynomials for our classification purposes.

Polynomials of the Second Order The good news is that the coefficients of polynomials can be induced by the same techniques that we have used for linear classifiers. Let us explain how.

For the sake of clarity, we will begin by constraining ourselves to simple domains with only two boolean attributes, x_1 and x_2. The second-order polynomial function is then defined as follows:

$$w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 = 0 \tag{4.12}$$

The expression on the left is a sum of terms that all have one thing in common: a weight, w_i, that multiplies a product $x_1^k x_2^l$. In the first term, we have $k + l = 0$, because $w_0x_1^0x_2^0 = w_0$; next come the terms with $k + l = 1$, concretely, $w_1x_1^1x_2^0 = w_1x_1$ and $w_2x_1^0x_2^1 = w_1x_2$; and the sequence ends with the three terms that have $k + l = 2$: specifically, $w_3x_1^2$, $w_4x_1^1x_2^1$, and $w_5x_2^2$. The thing to remember is that the expansion of the second-order polynomial stops when the sum of the exponents reaches 2.

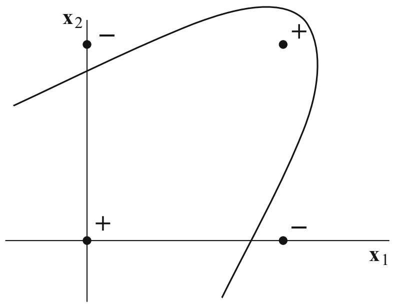


Fig. 4.5 In some domains, no linear classifier can separate the positive examples from the negative. Only a *non-linear classifier* can do so

Of course, some of the weights can be $w_i = 0$, rendering the corresponding terms "invisible" such as in $7 + 2x_1x_2 + 3x_2^2$ where the coefficients of x_1, x_2, and x_1^2 are zero.

Polynomials of the *r*-th Order More generally, the r-th order polynomial (still in a two-dimensional domain) will consist of a sum of weighted terms, $w_i x_1^k x_2^l$, such that $k + l = j$, where $j = 0, 1, \ldots r$.

The reader will easily make the next step and write down the general formula that defines the r-th order polynomial for domains with more than two attributes. A hint: the sum of the exponents in any single term never exceeds r.

Converting Them to Linear Classifiers Whatever the polynomial's order, and whatever the number of attributes, the task for machine learning is to find weights that make it possible to separate the positive examples from the negative. The seemingly unfortunate circumstance that the terms are non-linear (the sum of the exponents sometimes exceeds 1) is easily removed by *multipliers*, devices that return the logical conjunction of inputs: 1 if *all* inputs are 1; and 0 if *at least one* input is 0. With their help, we can replace each product of attributes with a new attribute, z_i, and thus re-write Eq. (4.12) in the following way:

$$w_0 + w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 = 0 \tag{4.13}$$

This means, for instance, that $z_3 = x_1^2$ and $z_4 = x_1 \cdot x_2$. Note that this "trick" has transformed the originally non-linear problem with two attributes, x_1 and x_2, into a linear problem with five newly created attributes, z_1 through z_5.

Figure 4.6 illustrates the situation where a second-order polynomial is used in a domain with three attributes.

Since the values of z_i in each example are known, the weights can be obtained without any difficulties using *perceptron learning* or WINNOW. Of course, we must not forget that these techniques will find the solution only if the polynomial of the chosen order is indeed capable of separating the two classes.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- When do we need non-linear classifiers? Specifically, what speaks in favor of polynomial classifiers?
- Write down the mathematical expression that defines a polynomial classifier. What "trick" allows us to use here the same learning techniques that were used in the case of linear classifiers?

The second-order polynomial function over three attributes is defined by the following function:

$$0 = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_1^2 + w_5x_1x_2 + w_6x_1x_3 \\ +w_7x_2^2 + w_8x_2x_3 + w_9x_3^2$$

Using the multipliers, we obtain the following:

$$0 = w_0 + w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + w_7z_7 + w_8z_8 + w_9z_9$$

Below is the schema of the whole "device" with multipliers. Before reaching the linear classifier, each signal z_i is multiplied by the corresponding weight, w_i.

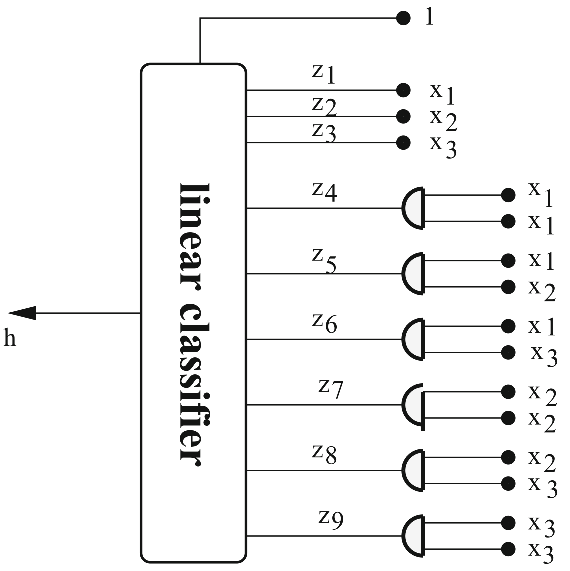


Fig. 4.6 A polynomial classifier can be converted into a linear classifier with the help of multipliers that pre-process the data

4.6 Specific Aspects of Polynomial Classifiers

To be able to use a machine-learning technique with success, the engineer must understand not only its strengths, but also its limitations, shortcomings, and pitfalls. In the case of polynomial classifiers, there are a few that deserve our attention. Let us briefly discuss them.

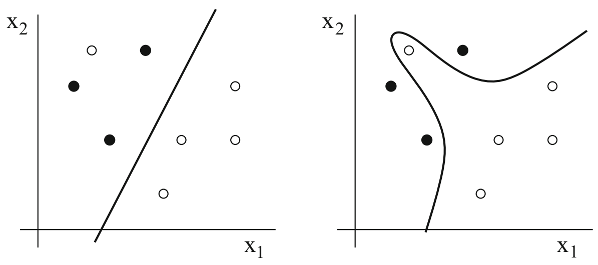


Fig. 4.7 The two classes are linearly separable, but noise has caused one negative example to be mislabeled as positive. The high-order polynomial on the *right overfits* the data, ignoring the possibility of noise

Overfitting Polynomial classifiers tend to *overfit* noisy training data. Since the problem of overfitting is encountered also in other machine-learning paradigms, we have to discuss its essence in some detail. For the sake of clarity, we will abandon the requirement that all attributes should be boolean; instead, we will rely on two-dimensional continuous domains that are easy to visualize.

The eight training examples in Fig. 4.7 fall into two groups. In one of them, all examples are positive; in the other, all save one are negative. Two attempts at separating the two classes are made. The one on the left is content with a linear classifier, shrugging off the minor inconvenience that one training example remains misclassified. The one on the right resorts to a polynomial classifier in an attempt to avoid any error being make on the training set.

An Inevitable Trade-Off Which of the two is to be preferred? This is not an easy question. On the one hand, the two classes may be linearly separable, and the only cause for one positive example to be found in the negative region is class-label noise. If this is the case, the single error made by the linear classifier on the training set is inconsequential, whereas the polynomial, cutting deep into the negative area, will misclassify examples that find themselves on the wrong side of the curve. On the other hand, there is also the chance that the outlier *does* represent some legitimate, even if rare, aspect of the positive class. In this event, using the polynomial will be justified. On the whole, however, the assumption that the single outlier is nothing but noise is more likely to be correct than the "special-aspect" alternative.

A realistic training set will contain not one, but quite a few, perhaps many examples that seem to find themselves in the wrong part of the instance space. And the inter-class boundary that our classifier attempts to model may indeed be curved, though *how much* curved is anybody's guess. The engineer may lay aside the linear classifier as too crude an approximation, and opt instead for the greater flexibility offered by the polynomials. This said, a very-high-order polynomial will avoid any error even in a very noisy training set—and then fail miserably on future data. The ideal solution is often somewhere between the extremes of linear classifiers and high-order polynomials. The best choice can be determined experimentally.

The Number of Weights The total number of the weights to be trained depends on the length of the attribute vector, and on the order of the polynomial. A simple analysis reveals that, in the case of n attributes and the r-th order polynomial, the number is determined by the following combinatorial expression:

$$N_W = \binom{n+r}{r} \tag{4.14}$$

Of course, N_W will be impractically high for large values of n. For instance, even for the relatively modest case of $n = 100$ attributes and a polynomial's order $r = 3$, the number of weights to be trained is $N_W = 176{,}851$ (103 choose 3). The computational costs thus incurred are not insurmountable for today's computers. What is worse is the danger of overfitting noisy training data; the polynomial is simply too flexible. The next paragraphs will tell us *how much* flexible.

Capacity The trivial domain in Fig. 4.1 consisted of four examples. Given that each of them can be labeled as either positive or negative, we have $2^4 = 16$ different ways of assigning labels to this training set. Of these sixteen, only two represent a situation where the two classes cannot be linearly separated—in this domain, linear inseparability is a rare event. But how typical is this situation in the more general case of m examples described by n attributes'? What are the chances that a random labeling of the examples will result in linearly separable classes?

Mathematics has found a simple guideline to be used in domains where n is "reasonably high" (say, ten or more attributes): if the number of examples, m, is less than twice the number of attributes ($m < 2n$), the probability that a random distribution of the two labels will result in linear separability is close to 100%. Conversely, this probability is almost zero when $m > 2n$. In this sense, "the *capacity* of a linear classifier is twice the number of attributes."

This result applies also to polynomial classifiers. The role of attributes is here played by the terms, z_i, obtained by the multipliers. Their number, N_W, is obtained by Eq. (4.14). We have seen that N_W can be quite high—and this makes the capacity high, too. In the case of $n = 100$ and $r = 3$, the number of weights is 176,851. This means that the third-order polynomial can separate the two classes (regardless of noise) as long as the number of examples is less than 353,702.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Elaborate on the term, "overfitting" and explain why this phenomenon (and its consequences) is in polynomial classifiers difficult to avoid.
- What is the upper bound on the number of weights to be trained in a polynomial of the r-th order in a domain that has n attributes?
- What is the *capacity* of the linear or polynomial classifier? What does capacity tell us about linear separability?

4.7 Numerical Domains and Support Vector Machines

Now that we have realized that polynomials do not call for new machine-learning algorithms, we can return to linear classifiers, a topic we have not yet exhausted. Time has come to abandon our self-imposed restriction to boolean attributes, and to start considering also the possibility of the attributes being continuous. Can we then still rely on the two training algorithms described above?

Perceptron in Numeric Domains In the case of *perceptron learning*, the answer is easy: yes, the same weight-modification formula can be used. Practical experience shows, however, that it is good to make all attribute values fall into the unit interval, $x_i \in [0, 1]$.

Let us repeat here, for the reader's convenience, the weight-adjusting formula:

$$w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})]x_i \tag{4.15}$$

While the learning rate, η, and the difference between the real and the hypothesized class label, $[c(\mathbf{x}) - h(\mathbf{x})]$, have the same meaning and impact as before, what has changed is the role of x_i. In the case of boolean attributes, the value of x_i decided whether or not the weight should change. Here, however, it rather says *how much* the weight should be affected: more in the case of higher attribute values.

The Multiplicative Rule In the case of WINNOW, too, essentially the same learning formula can be used as in the binary-attributes case:

$$w_i = w_i\alpha^{c(\mathbf{x})-h(\mathbf{x})} \tag{4.16}$$

This said, one has to be careful about *when* to apply the formula. Previously, one modified only the weights of attributes with $x_i = 1$. Now that the attribute values come from a continuous domain, some modification is needed. One possibility is the following rule:

"Update weight w_i only if the value of the i-th attribute is $x_i \geq 0.5$."

Let us remark that both of these algorithms (perceptron learning and WINNOW) usually find a relatively low-error solution even if the two classes are not linearly separable—for instance, in the presence of noise.

Which Linear Classifier Is Better? At this point, however, another important question needs to be discussed. Figure 4.8 shows three linear classifiers, each perfectly separating the positive training examples from the negative. Knowing that "good behavior" on the training set does not guarantee high performance in the future, we have to ask: which of the three is likely to score best on future examples?

The Support Vector Machine Mathematicians who studied this problem found an answer. When we take a look at Fig. 4.8, we can see that the dotted-line classifier all but touches the nearest examples on either side; we say that its *margin* is small.

Conversely, the margin is greater in the case of the solid-line classifier: the nearest positive example on one side of the line, and the nearest example on the other of the line are much farther than in the case of the other classifier. As it turns out, the greater the margin, the higher the chances that the classifier will do well on future data.

The technique of the *support vector machines* is illustrated in Fig. 4.9. The solid line is the best classifier. The graph shows also two thinner lines, parallel to the classifier, each at the same distance. The reader can see they pass through the examples nearest to the classifier. These examples are called *support vectors* (because, after all, each example is a vector of attributes).

The task for machine learning is to identify the support vectors that maximize the margin. The concrete mechanisms for finding the optimum set of support vectors exceed the ambition of an introductory text. The simplest technique would simply try all possible n-tuples of examples, and measure the margin implied by each such choice. This, however, is unrealistic in domains with many examples. Most of the time, therefore, engineers rely on some of the many software packages available for free on the internet.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Can *perceptron learning* and WINNOW be used in numeric domains? How?
- Given that there are infinitely many linear classifiers capable of separating the positive examples from the negative (assuming such separation exists), which of them can be expected to give the best results on future data?
- What is a *support vector*? What is meant by the *margin* to be maximized?

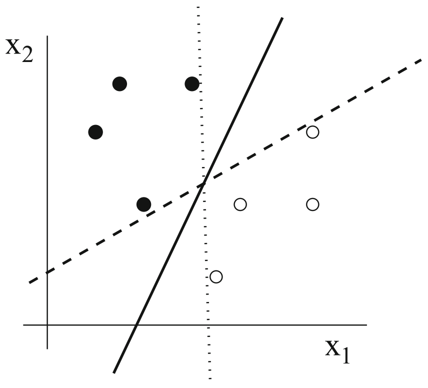


Fig. 4.8 Linearly separable classes can be separated in infinitely many different ways. Question is, which of the classifiers that are perfect on the training set will do best on future data

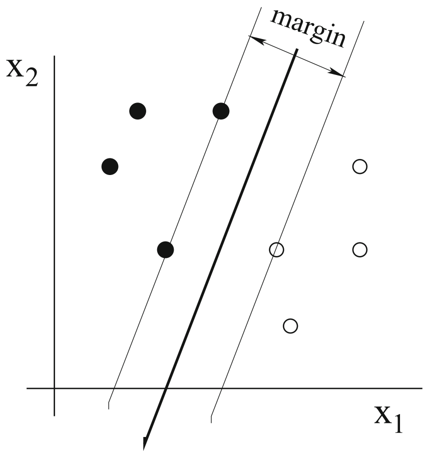


Fig. 4.9 The technique of the *support vector machine* looks for a separating hyperplane that has the maximum *margin*

4.8 Summary and Historical Remarks

- Linear and polynomial classifiers define a *decision surface* that separates the positive examples from the negative. Specifically, *linear* classifiers label the examples according to the sign of the following expression:

$$w_0 + w_1x_1 + \dots w_nx_n$$

  The concrete behavior is determined by the weights, w_i. The task for machine learning is to find appropriate values for these weights.
- The learning techniques from this chapter rely on the principle of "learning from mistakes." The training examples are presented, one by one, to the learner. Each time the learner misclassifies an example, the weights are modified. When the entire training set has been presented, one *epoch* has been completed. Usually, several epochs are needed.
- Two weight-modification techniques were considered here: the additive rule of *perceptron learning*, and the multiplicative rule of WINNOW.
- In domains with more than two classes, one can consider the use of a specialized classifier for each class. A "master classifier" then chooses the class whose classifier had the highest value of $\Sigma_{i=0}^{n} w_i x_i$.
- In domains with non-linear class boundaries, *polynomial* classifiers can sometimes be used. A second-order polynomial in a two-dimensional domain is defined by the following expression:

$$w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2$$

- The weights of the polynomial can be found by the same learning algorithms as in the case of linear classifiers, provided that the non-linear terms (e.g., x_1x_2)

have been replaced (with the help of *multipliers*) by newly created attributes such as $z_3 = x_1^2$ or $z_4 = x_1x_2$.

- Polynomial classifiers are prone to *overfit* noisy training data. This is explained by the excessive flexibility caused by the very high number of trainable weights.
- The potentially best class-separating hyperplane (among the infinitely many candidates) is identified by the technique of the *support vector machines (SVM)* that seek to maximize the distance of the nearest positive and the nearest negative example from the hyperplane.

Historical Remarks The principle of *perceptron learning* was developed by Rosenblatt [81], whereas WINNOW was proposed and analyzed by Littlestone [54]. The question of the capacity of linear and polynomial classifiers was analyzed by Cover [18]. The principle of Support Vector Machines was invented by Vapnik [93] as one of the consequences of the Computational Learning Theory which will be the subject of Chap. 7.

4.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Exercises

1. Write down equations for linear classifiers to implement the following functions:
   - At least two out of the boolean attributes $x_1, \ldots, x_5$ are *true*
   - At least three out of the boolean attributes $x_1, \ldots, x_6$ are *true*, and at least one of them is *false*.
2. Return to the examples from Table 4.2. Hand-simulate the perceptron learning algorithm's procedure, starting from a different initial set of weights than the one used in the table. Try also a different learning rate.
3. Repeat the same exercise, this time using WINNOW. Do not forget to introduce the additional "attributes" for what in perceptrons were the negative weights.
4. Write down the equation that defines a third-order polynomial in two dimensions. How many multipliers (each with up to three inputs) would be needed if we wanted to train the weights using the perceptron learning algorithm?

Give It Some Thought

1. How can induction of linear classifiers be used to identify irrelevant attributes? Hint: try to run the learning algorithm on different subsets of the attributes, and then observe the error rate achieved after a fixed number of epochs.
2. Explain in what way it is true that the 1-NN classifier applied to a pair of examples (one positive, the other negative) in a plane defines a linear classifier. Invent a machine learning algorithm that uses this observation for the induction of linear classifiers. Generalize the procedure to n-dimensional domains.
3. When is a linear classifier likely to lead to better classification performance on independent testing examples than a polynomial classifier?
4. Sometimes, a linearly separable domain becomes linearly non-separable on account of the class-label noise. Think of a technique capable of removing such noisy examples. Hint: you may rely on an idea we have already encountered in the field of k-NN classifiers.

Computer Assignments

1. Implement the perceptron learning algorithm and run it on the following training set where six examples (three positive and three negative) are described by four attributes:

x_1	x_2	x_3	x_4	Class
1	1	1	0	**pos**
0	0	0	0	**pos**
1	1	0	1	**pos**
1	1	0	0	**neg**
0	1	0	1	**neg**
0	0	0	1	**neg**

   Observe that the linear classifier fails to reach zero error rate because the two classes are not linearly separable.
2. Create a training set consisting of 20 examples described by five binary attributes, $x_1, \ldots, x_5$. Examples in which at least three attributes have values $x_i = 1$ are labeled as positive, all other examples are labeled as negative. Using this training set as input, induce a linear classifier using perceptron learning. Experiment with different values of the learning rate, η. Plot a function where the horizontal axis represents η, and the vertical axis represents the number of example-presentations needed for the classifier to correctly classify all training examples. Discuss the results.

3. Use the same domain as in the previous assignment (five boolean attributes, and the same definition of the positive class). Add to each example N additional boolean attributes whose values are determined by a random-number generator. Vary N from 1 to 20. Observe how the number of example-presentations needed to achieve the zero error rate depends on N.
4. Again, use the same domain, but add attribute noise by changing the values of randomly selected examples (while leaving class labels unchanged). Observe what minimum error rate can then be achieved.
5. Repeat the last three assignments for different sizes of the training set, evaluating the results on (always the same) testing set of examples that have not been seen during learning.
6. Repeat the last four assignments, using WINNOW instead of the perceptron learning. Compare the results in terms of the incurred computational costs. These costs can be measured by the number of epochs needed to converge to the zero error rate on the training set.
7. Define a domain with three numeric attributes with values from the unit interval, [0, 1]. Generate 100 training examples, labeling as positive those for which the expression $1 - x_1 + x2 + x3$ is positive. Use the "perceptron learning algorithm" modified so that the following versions of the weight-updating rule are used:

   (a) $w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})]x_i$
   (b) $w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})]$
   (c) $w_i = w_i + \eta[c(\mathbf{x}) - \sum w_i x_i]x_i$
   (d) $w_i = w_i + \eta[c(\mathbf{x}) - \sum w_i x_i]$

   Will all of them converge to zero error rate on the training set? Compare the speed of conversion.
8. Create a training set where each example is described by six boolean attributes, $x_i, \ldots, x_6$. Label each example with one of the four classes defined as follows:

   (a) C_1: at least five attributes have $x_i = 1$.
   (b) C_2: three or four attributes have $x_i = 1$.
   (c) C_3: two attributes have $x_i = 1$.
   (d) C_4: one attribute has $x_i = 1$.

   Use perceptron learning, applied in parallel to each of the four classes.
   As a variation, use different numbers of irrelevant attributes, varying their number from 0 to 20. See if the zero error rate on the training set can be achieved.
   Record the number of false positives and the number of false negatives observed on an independent testing set.
   Design an experiment showing that the performance of K binary classifiers, connected in parallel as in Fig. 4.4, will decrease if we increase the number of classes. How much is this observation pronounced in the presence of noise?

9. Run induction of linear classifiers on selected boolean domains from the UCI repository[3] and compare the results.
10. Experimenting with selected domains from the UCI repository, observe the impact of the learning rate, η, on the convergence speed of the perceptron learning algorithm.
11. Compare the behavior of linear and polynomial classifiers. Observe how the former wins in simple domains, and the latter in highly non-linear domains.

[3] `www.ics.uci.edu/~mlearn/MLRepository.html`.