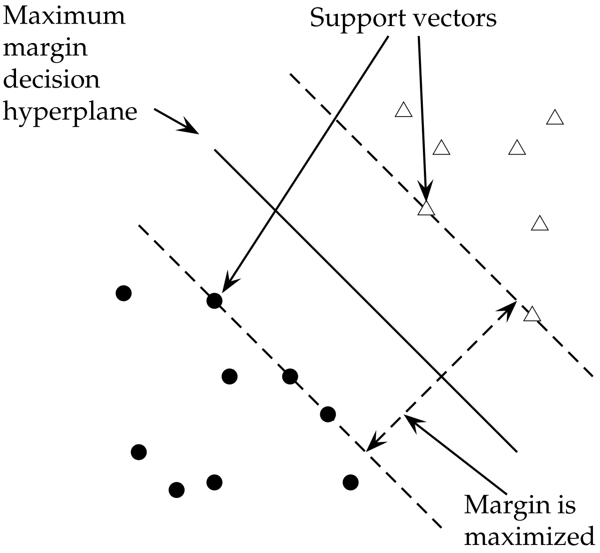
15 Support vector machines and machine learning on documents

Improving classifier effectiveness has been an area of intensive machine-learning research over the last two decades, and this work has led to a new generation of state-of-the-art classifiers, such as support vector machines, boosted decision trees, regularized logistic regression, neural networks, and random forests. Many of these methods, including support vector machines (SVMs), the main topic of this chapter, have been applied with success to information retrieval problems, particularly text classification. An SVM is a kind of large-margin classifier: it is a vector space based machine learning method where the goal is to find a decision boundary between two classes that is maximally far from any point in the training data (possibly discounting some points as outliers or noise).

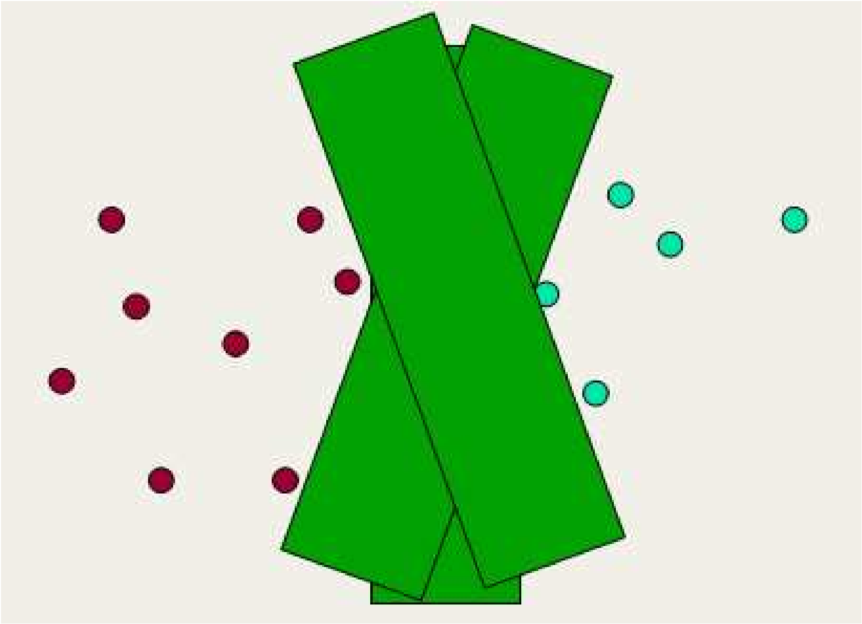
We will initially motivate and develop SVMs for the case of two-class data sets that are separable by a linear classifier (Section 15.1), and then extend the model in Section 15.2 to non-separable data, multi-class problems, and nonlinear models, and also present some additional discussion of SVM performance. The chapter then moves to consider the practical deployment of text classifiers in Section 15.3: what sorts of classifiers are appropriate when, and how can you exploit domain-specific text features in classification? Finally, we will consider how the machine learning technology that we have been building for text classification can be applied back to the problem of learning how to rank documents in ad hoc retrieval (Section 15.4). While several machine learning methods have been applied to this task, use of SVMs has been prominent. Support vector machines are not necessarily better than other machine learning methods (except perhaps in situations with little training data), but they perform at the state-of-the-art level and have much current theoretical and empirical appeal.



► **Figure 15.1** The support vectors are the 5 points right up against the margin of the classifier.

15.1 Support vector machines: The linearly separable case

For two-class, separable training data sets, such as the one in Figure 14.8 (page 301), there are lots of possible linear separators. Intuitively, a decision boundary drawn in the middle of the void between data items of the two classes seems better than one which approaches very close to examples of one or both classes. While some learning methods such as the perceptron algorithm (see references in Section 14.7, page 314) find just any linear separator, others, like Naive Bayes, search for the best linear separator according to some criterion. The SVM in particular defines the criterion to be looking for a decision surface that is maximally far away from any data point. This distance from the decision surface to the closest data point determines the MARGIN *margin* of the classifier. This method of construction necessarily means that the decision function for an SVM is fully specified by a (usually small) subset of the data which defines the position of the separator. These points are SUPPORT VECTOR referred to as the *support vectors* (in a vector space, a point can be thought of as a vector between the origin and that point). Figure 15.1 shows the margin and support vectors for a sample problem. Other data points play no part in determining the decision surface that is chosen.



► **Figure 15.2** An intuition for large-margin classification. Insisting on a large margin reduces the capacity of the model: the range of angles at which the fat decision surface can be placed is smaller than for a decision hyperplane (cf. Figure 14.8, page 301).

Maximizing the margin seems good because points near the decision surface represent very uncertain classification decisions: there is almost a 50% chance of the classifier deciding either way. A classifier with a large margin makes no low certainty classification decisions. This gives you a classification safety margin: a slight error in measurement or a slight document variation will not cause a misclassification. Another intuition motivating SVMs is shown in Figure 15.2. By construction, an SVM classifier insists on a large margin around the decision boundary. Compared to a decision hyperplane, if you have to place a fat separator between classes, you have fewer choices of where it can be put. As a result of this, the memory capacity of the model has been decreased, and hence we expect that its ability to correctly generalize to test data is increased (cf. the discussion of the bias-variance tradeoff in Chapter 14, page 312).

Let us formalize an SVM with algebra. A decision hyperplane (page 302) can be defined by an intercept term b and a decision hyperplane normal vector $\vec{w}$ which is perpendicular to the hyperplane. This vector is commonly

WEIGHT VECTOR

referred to in the machine learning literature as the *weight vector*. To choose among all the hyperplanes that are perpendicular to the normal vector, we specify the intercept term b. Because the hyperplane is perpendicular to the normal vector, all points $\vec{x}$ on the hyperplane satisfy $\vec{w}^{\mathrm{T}}\vec{x} = -b$. Now suppose that we have a set of training data points $\mathbb{D} = \{(\vec{x}_i, y_i)\}$, where each member is a pair of a point $\vec{x}_i$ and a class label y_i corresponding to it.[1] For SVMs, the two data classes are always named $+1$ and -1 (rather than 1 and 0), and the intercept term is always explicitly represented as b (rather than being folded into the weight vector $\vec{w}$ by adding an extra always-on feature). The math works out much more cleanly if you do things this way, as we will see almost immediately in the definition of functional margin. The linear classifier is then:

$$f(\vec{x}) = \text{sign}(\vec{w}^{\mathrm{T}}\vec{x} + b) \tag{15.1}$$

A value of -1 indicates one class, and a value of $+1$ the other class.

FUNCTIONAL MARGIN

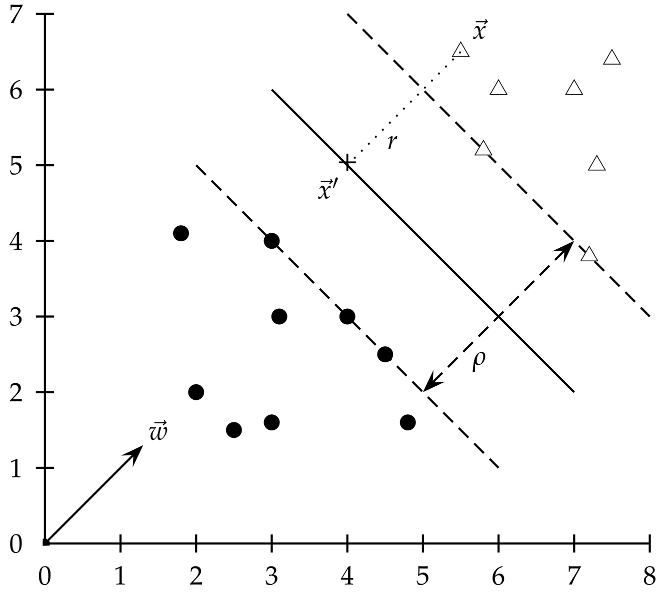
We are confident in the classification of a point if it is far away from the decision boundary. For a given data set and decision hyperplane, we define the *functional margin* of the i^{th} example $\vec{x}_i$ with respect to a hyperplane $\langle \vec{w}, b \rangle$ as the quantity $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b)$. The functional margin of a data set with respect to a decision surface is then twice the functional margin of any of the points in the data set with minimal functional margin (the factor of 2 comes from measuring across the whole width of the margin, as in Figure 15.3). However, there is a problem with using this definition as is: the value is underconstrained, because we can always make the functional margin as big as we wish by simply scaling up $\vec{w}$ and b. For example, if we replace $\vec{w}$ by $5\vec{w}$ and b by $5b$ then the functional margin $y_i(5\vec{w}^{\mathrm{T}}\vec{x}_i + 5b)$ is five times as large. This suggests that we need to place some constraint on the size of the $\vec{w}$ vector. To get a sense of how to do that, let us look at the actual geometry.

What is the Euclidean distance from a point $\vec{x}$ to the decision boundary? In Figure 15.3, we denote by r this distance. We know that the shortest distance between a point and a hyperplane is perpendicular to the plane, and hence, parallel to $\vec{w}$. A unit vector in this direction is $\vec{w}/|\vec{w}|$. The dotted line in the diagram is then a translation of the vector $r\vec{w}/|\vec{w}|$. Let us label the point on the hyperplane closest to $\vec{x}$ as $\vec{x}'$. Then:

$$\vec{x}' = \vec{x} - yr\frac{\vec{w}}{|\vec{w}|} \tag{15.2}$$

where multiplying by y just changes the sign for the two cases of $\vec{x}$ being on either side of the decision surface. Moreover, $\vec{x}'$ lies on the decision boundary

[1] As discussed in Section 14.1 (page 291), we present the general case of points in a vector space, but if the points are length normalized document vectors, then all the action is taking place on the surface of a unit sphere, and the decision surface intersects the sphere's surface.



► **Figure 15.3** The geometric margin of a point (r) and a decision boundary (ρ).

and so satisfies $\vec{w}^{\mathrm{T}}\vec{x}' + b = 0$. Hence:

$$\vec{w}^{\mathrm{T}}(\vec{x} - yr\frac{\vec{w}}{|\vec{w}|}) + b = 0 \tag{15.3}$$

Solving for r gives:[2]

$$r = y\frac{\vec{w}^{\mathrm{T}}\vec{x} + b}{|\vec{w}|} \tag{15.4}$$

GEOMETRIC MARGIN

Again, the points closest to the separating hyperplane are support vectors. The *geometric margin* of the classifier is the maximum width of the band that can be drawn separating the support vectors of the two classes. That is, it is twice the minimum value over data points for r given in Equation (15.4), or, equivalently, the maximal width of one of the fat separators shown in Figure 15.2. The geometric margin is clearly invariant to scaling of parameters: if we replace $\vec{w}$ by $5\vec{w}$ and b by $5b$, then the geometric margin is the same, because it is inherently normalized by the length of $\vec{w}$. This means that we can impose any scaling constraint we wish on $\vec{w}$ without affecting the geometric margin. Among other choices, we could use unit vectors, as in Chapter 6, by

[2] Recall that $|\vec{w}| = \sqrt{\vec{w}^{\mathrm{T}}\vec{w}}$.

requiring that $|\vec{w}| = 1$. This would have the effect of making the geometric margin the same as the functional margin.

Since we can scale the functional margin as we please, for convenience in solving large SVMs, let us choose to require that the functional margin of all data points is at least 1 and that it is equal to 1 for at least one data vector. That is, for all items in the data:

$$y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1 \tag{15.5}$$

and there exist support vectors for which the inequality is an equality. Since each example's distance from the hyperplane is $r_i = y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b)/|\vec{w}|$, the geometric margin is $\rho = 2/|\vec{w}|$. Our desire is still to maximize this geometric margin. That is, we want to find $\vec{w}$ and b such that:

- $\rho = 2/|\vec{w}|$ is maximized
- For all $(\vec{x}_i, y_i) \in \mathbb{D}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$

Maximizing $2/|\vec{w}|$ is the same as minimizing $|\vec{w}|/2$. This gives the final standard formulation of an SVM as a minimization problem:

(15.6) Find $\vec{w}$ and b such that:

- $\frac{1}{2}\vec{w}^{\mathrm{T}}\vec{w}$ is minimized, and
- for all $\{(\vec{x}_i, y_i)\}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$

QUADRATIC PROGRAMMING

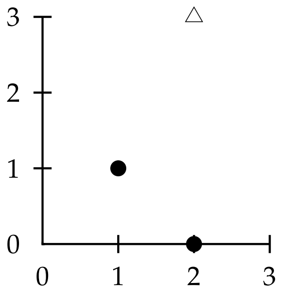
We are now optimizing a quadratic function subject to linear constraints. *Quadratic optimization* problems are a standard, well-known class of mathematical optimization problems, and many algorithms exist for solving them. We could in principle build our SVM using standard quadratic programming (QP) libraries, but there has been much recent research in this area aiming to exploit the structure of the kind of QP that emerges from an SVM. As a result, there are more intricate but much faster and more scalable libraries available especially for building SVMs, which almost everyone uses to build models. We will not present the details of such algorithms here.

However, it will be helpful to what follows to understand the shape of the solution of such an optimization problem. The solution involves constructing a dual problem where a Lagrange multiplier α_i is associated with each constraint $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$ in the primal problem:

(15.7) Find $\alpha_1, \ldots \alpha_N$ such that $\sum \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x}_i{}^{\mathrm{T}}\vec{x}_j$ is maximized, and

- $\sum_i \alpha_i y_i = 0$
- $\alpha_i \geq 0$ for all $1 \leq i \leq N$

The solution is then of the form:



► **Figure 15.4** A tiny 3 data point training set for an SVM.

$$\vec{w} = \sum \alpha_i y_i \vec{x}_i \qquad (15.8)$$
$$b = y_k - \vec{w}^{\mathrm{T}} \vec{x}_k \text{ for any } \vec{x}_k \text{ such that } \alpha_k \neq 0$$

In the solution, most of the α_i are zero. Each non-zero α_i indicates that the corresponding $\vec{x}_i$ is a support vector. The classification function is then:

$$f(\vec{x}) = \mathrm{sign}(\sum_i \alpha_i y_i \vec{x}_i{}^{\mathrm{T}} \vec{x} + b) \qquad (15.9)$$

Both the term to be maximized in the dual problem and the classifying function involve a *dot product* between pairs of points ($\vec{x}$ and $\vec{x}_i$ or $\vec{x}_i$ and $\vec{x}_j$), and that is the only way the data are used – we will return to the significance of this later.

To recap, we start with a training data set. The data set uniquely defines the best separating hyperplane, and we feed the data through a quadratic optimization procedure to find this plane. Given a new point $\vec{x}$ to classify, the classification function $f(\vec{x})$ in either Equation (15.1) or Equation (15.9) is computing the projection of the point onto the hyperplane normal. The sign of this function determines the class to assign to the point. If the point is within the margin of the classifier (or another confidence threshold t that we might have determined to minimize classification mistakes) then the classifier can return "don't know" rather than one of the two classes. The value of $f(\vec{x})$ may also be transformed into a probability of classification; fitting a sigmoid to transform the values is standard (Platt 2000). Also, since the margin is constant, if the model includes dimensions from various sources, careful rescaling of some dimensions may be required. However, this is not a problem if our documents (points) are on the unit hypersphere.

Example 15.1: Consider building an SVM over the (very little) data set shown in Figure 15.4. Working geometrically, for an example like this, the maximum margin weight vector will be parallel to the shortest line connecting points of the two classes, that is, the line between $(1,1)$ and $(2,3)$, giving a weight vector of $(1,2)$. The optimal decision surface is orthogonal to that line and intersects it at the halfway point.

Therefore, it passes through $(1.5, 2)$. So, the SVM decision boundary is:

$$y = x_1 + 2x_2 - 5.5$$

Working algebraically, with the standard constraint that $\text{sign}(y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b)) \geq 1$, we seek to minimize $|\vec{w}|$. This happens when this constraint is satisfied with equality by the two support vectors. Further we know that the solution is $\vec{w} = (a, 2a)$ for some a. So we have that:

$$\begin{aligned} a + 2a + b &= -1 \\ 2a + 6a + b &= 1 \end{aligned}$$

Therefore, $a = 2/5$ and $b = -11/5$. So the optimal hyperplane is given by $\vec{w} = (2/5, 4/5)$ and $b = -11/5$.

The margin ρ is $2/|\vec{w}| = 2/\sqrt{4/25 + 16/25} = 2/(2\sqrt{5}/5) = \sqrt{5}$. This answer can be confirmed geometrically by examining Figure 15.4.

?

Exercise 15.1 [⋆]

What is the minimum number of support vectors that there can be for a data set (which contains instances of each class)?

Exercise 15.2 [⋆⋆]

The basis of being able to use kernels in SVMs (see Section 15.2.3) is that the classification function can be written in the form of Equation (15.9) (where, for large problems, most α_i are 0). Show explicitly how the classification function could be written in this form for the data set from Example 15.1. That is, write f as a function where the data points appear and the only variable is $\vec{x}$.

Exercise 15.3 [⋆⋆]

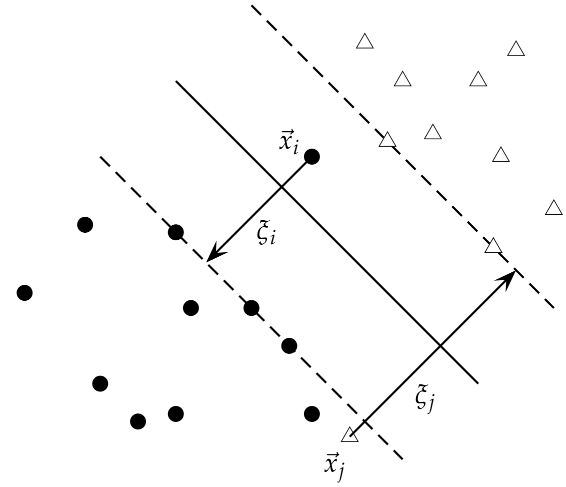
Install an SVM package such as SVMlight (http://svmlight.joachims.org/), and build an SVM for the data set discussed in Example 15.1. Confirm that the program gives the same solution as the text. For SVMlight, or another package that accepts the same training data format, the training file would be:

```
+1 1:2 2:3
−1 1:2 2:0
−1 1:1 2:1
```

The training command for SVMlight is then:

```
svm_learn -c 1 -a alphas.dat train.dat model.dat
```

The `-c 1` option is needed to turn off use of the slack variables that we discuss in Section 15.2.1. Check that the norm of the weight vector agrees with what we found in Example 15.1. Examine the file alphas.dat which contains the α_i values, and check that they agree with your answers in Exercise 15.2.



► **Figure 15.5** Large margin classification with slack variables.

15.2 Extensions to the SVM model

15.2.1 Soft margin classification

For the very high dimensional problems common in text classification, sometimes the data are linearly separable. But in the general case they are not, and even if they are, we might prefer a solution that better separates the bulk of the data while ignoring a few weird noise documents.

If the training set $\mathbb{D}$ is not linearly separable, the standard approach is to allow the fat decision margin to make a few mistakes (some points – outliers or noisy examples – are inside or on the wrong side of the margin). We then pay a cost for each misclassified example, which depends on how far it is from meeting the margin requirement given in Equation (15.5). To implement this, we introduce *slack variables* ξ_i. A non-zero value for ξ_i allows $\vec{x}_i$ to not meet the margin requirement at a cost proportional to the value of ξ_i. See Figure 15.5.

SLACK VARIABLES

The formulation of the SVM optimization problem with slack variables is:

(15.10) Find $\vec{w}$, b, and $\xi_i \geq 0$ such that:

- $\frac{1}{2}\vec{w}^{\mathrm{T}}\vec{w} + C\sum_i \xi_i$ is minimized
- and for all $\{(\vec{x}_i, y_i)\}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1 - \xi_i$

The optimization problem is then trading off how fat it can make the margin versus how many points have to be moved around to allow this margin. The margin can be less than 1 for a point $\vec{x}_i$ by setting $\xi_i > 0$, but then one pays a penalty of $C\xi_i$ in the minimization for having done that. The sum of the ξ_i gives an upper bound on the number of training errors. Soft-margin SVMs minimize training error traded off against margin. The parameter C is a *regularization* term, which provides a way to control overfitting: as C becomes large, it is unattractive to not respect the data at the cost of reducing the geometric margin; when it is small, it is easy to account for some data points with the use of slack variables and to have a fat margin placed so it models the bulk of the data.

REGULARIZATION

The dual problem for soft margin classification becomes:

(15.11) Find $\alpha_1, \ldots \alpha_N$ such that $\sum \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x}_i{}^{\mathrm{T}} \vec{x}_j$ is maximized, and

- $\sum_i \alpha_i y_i = 0$
- $0 \leq \alpha_i \leq C$ for all $1 \leq i \leq N$

Neither the slack variables ξ_i nor Lagrange multipliers for them appear in the dual problem. All we are left with is the constant C bounding the possible size of the Lagrange multipliers for the support vector data points. As before, the $\vec{x}_i$ with non-zero α_i will be the support vectors. The solution of the dual problem is of the form:

(15.12)
$$\vec{w} = \sum \alpha y_i \vec{x}_i$$
$$b = y_k(1 - \xi_k) - \vec{w}^{\mathrm{T}} \vec{x}_k \text{ for } k = \arg\max_k \alpha_k$$

Again $\vec{w}$ is not needed explicitly for classification, which can be done in terms of dot products with data points, as in Equation (15.9).

Typically, the support vectors will be a small proportion of the training data. However, if the problem is non-separable or with small margin, then every data point which is misclassified or within the margin will have a non-zero α_i. If this set of points becomes large, then, for the nonlinear case which we turn to in Section 15.2.3, this can be a major slowdown for using SVMs at test time.

The complexity of training and testing with linear SVMs is shown in Table 15.1.[3] The time for training an SVM is dominated by the time for solving the underlying QP, and so the theoretical and empirical complexity varies depending on the method used to solve it. The standard result for solving QPs is that it takes time cubic in the size of the data set (Kozlov et al. 1979). All the recent work on SVM training has worked to reduce that complexity, often by

[3] We write $\Theta(|\mathbb{D}|L_{\mathrm{ave}})$ for $\Theta(T)$ (page 262) and assume that the length of test documents is bounded as we did on page 262.

Classifier	Mode	Method	Time complexity
NB	training		$\Theta(\lvert\mathbb{D}\rvert L_{\text{ave}} + \lvert\mathbb{C}\rvert\lvert V\rvert)$
NB	testing		$\Theta(\lvert\mathbb{C}\rvert M_{\text{a}})$
Rocchio	training		$\Theta(\lvert\mathbb{D}\rvert L_{\text{ave}} + \lvert\mathbb{C}\rvert\lvert V\rvert)$
Rocchio	testing		$\Theta(\lvert\mathbb{C}\rvert M_{\text{a}})$
kNN	training	preprocessing	$\Theta(\lvert\mathbb{D}\rvert L_{\text{ave}})$
kNN	testing	preprocessing	$\Theta(\lvert\mathbb{D}\rvert M_{\text{ave}} M_{\text{a}})$
kNN	training	no preprocessing	$\Theta(1)$
kNN	testing	no preprocessing	$\Theta(\lvert\mathbb{D}\rvert L_{\text{ave}} M_{\text{a}})$
SVM	training	conventional	$O(\lvert\mathbb{C}\rvert\lvert\mathbb{D}\rvert^3 M_{\text{ave}})$; $\approx O(\lvert\mathbb{C}\rvert\lvert\mathbb{D}\rvert^{1.7} M_{\text{ave}})$, empirically
SVM	training	cutting planes	$O(\lvert\mathbb{C}\rvert\lvert\mathbb{D}\rvert M_{\text{ave}})$
SVM	testing		$O(\lvert\mathbb{C}\rvert M_{\text{a}})$

► **Table 15.1** Training and testing complexity of various classifiers including SVMs. Training is the time the learning method takes to learn a classifier over $\mathbb{D}$, while testing is the time it takes a classifier to classify one document. For SVMs, multiclass classification is assumed to be done by a set of $\lvert\mathbb{C}\rvert$ one-versus-rest classifiers. L_{ave} is the average number of tokens per document, while M_{ave} is the average vocabulary (number of non-zero features) of a document. L_{a} and M_{a} are the numbers of tokens and types, respectively, in the test document.

being satisfied with approximate solutions. Standardly, empirical complexity is about $O(\lvert\mathbb{D}\rvert^{1.7})$ (Joachims 2006a). Nevertheless, the super-linear training time of traditional SVM algorithms makes them difficult or impossible to use on very large training data sets. Alternative traditional SVM solution algorithms which are linear in the number of training examples scale badly with a large number of features, which is another standard attribute of text problems. However, a new training algorithm based on cutting plane techniques gives a promising answer to this issue by having running time linear in the number of training examples and the number of non-zero features in examples (Joachims 2006a). Nevertheless, the actual speed of doing quadratic optimization remains much slower than simply counting terms as is done in a Naive Bayes model. Extending SVM algorithms to nonlinear SVMs, as in the next section, standardly increases training complexity by a factor of $\lvert\mathbb{D}\rvert$ (since dot products between examples need to be calculated), making them impractical. In practice it can often be cheaper to materialize the higher-order features and to train a linear SVM.[4]

15.2.2 Multiclass SVMs

SVMs are inherently two-class classifiers. The traditional way to do multiclass classification with SVMs is to use one of the methods discussed in Section 14.5 (page 306). In particular, the most common technique in practice has been to build $|\mathbb{C}|$ one-versus-rest classifiers (commonly referred to as "one-versus-all" or OVA classification), and to choose the class which classifies the test datum with greatest margin. Another strategy is to build a set of one-versus-one classifiers, and to choose the class that is selected by the most classifiers. While this involves building $|\mathbb{C}|(|\mathbb{C}|-1)/2$ classifiers, the time for training classifiers may actually decrease, since the training data set for each classifier is much smaller.

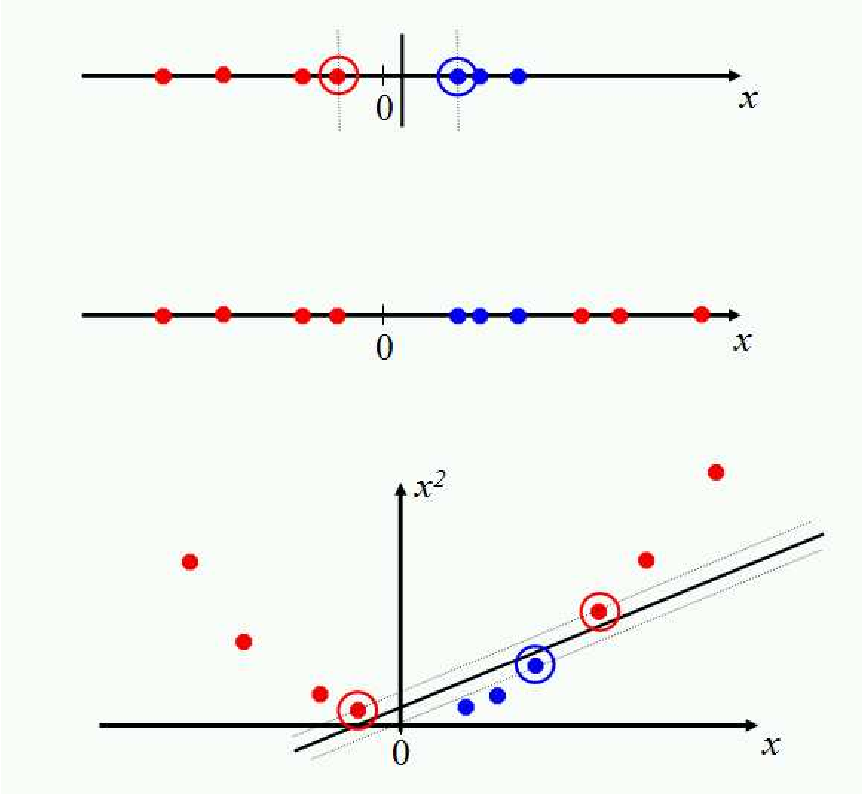
However, these are not very elegant approaches to solving multiclass problems. A better alternative is provided by the construction of multiclass SVMs, where we build a two-class classifier over a feature vector $\Phi(\vec{x}, y)$ derived from the pair consisting of the input features and the class of the datum. At test time, the classifier chooses the class $y = \arg\max_{y'} \vec{w}^{\mathrm{T}}\Phi(\vec{x}, y')$. The margin during training is the gap between this value for the correct class and for the nearest other class, and so the quadratic program formulation will require that $\forall i \;\; \forall y \neq y_i \;\; \vec{w}^{\mathrm{T}}\Phi(\vec{x}_i, y_i) - \vec{w}^{\mathrm{T}}\Phi(\vec{x}_i, y) \geq 1 - \xi_i$. This general method can be extended to give a multiclass formulation of various kinds of linear classifiers. It is also a simple instance of a generalization of classification where the classes are not just a set of independent, categorical labels, but may be arbitrary structured objects with relationships defined between them. In the SVM world, such work comes under the label of *structural SVMs*. We mention them again in Section 15.4.2.

STRUCTURAL SVMS

15.2.3 Nonlinear SVMs

With what we have presented so far, data sets that are linearly separable (perhaps with a few exceptions or some noise) are well-handled. But what are we going to do if the data set just doesn't allow classification by a linear classifier? Let us look at a one-dimensional case. The top data set in Figure 15.6 is straightforwardly classified by a linear classifier but the middle data set is not. We instead need to be able to pick out an interval. One way to solve this problem is to map the data on to a higher dimensional space and then to use a linear classifier in the higher dimensional space. For example, the bottom part of the figure shows that a linear separator can easily classify the data

[4] Materializing the features refers to directly calculating higher order and interaction terms and then putting them into a linear model.



► **Figure 15.6** Projecting data that is not linearly separable into a higher dimensional space can make it linearly separable.

if we use a quadratic function to map the data into two dimensions (a polar coordinates projection would be another possibility). The general idea is to map the original feature space to some higher-dimensional feature space where the training set is separable. Of course, we would want to do so in ways that preserve relevant dimensions of relatedness between data points, so that the resultant classifier should still generalize well.

KERNEL TRICK

SVMs, and also a number of other linear classifiers, provide an easy and efficient way of doing this mapping to a higher dimensional space, which is referred to as "the *kernel trick*". It's not really a trick: it just exploits the math that we have seen. The SVM linear classifier relies on a dot product between data point vectors. Let $K(\vec{x}_i, \vec{x}_j) = \vec{x}_i^{\mathrm{T}}\vec{x}_j$. Then the classifier we have seen so far is:

$$f(\vec{x}) = \mathrm{sign}(\sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b) \tag{15.13}$$

Now suppose we decide to map every data point into a higher dimensional space via some transformation $\Phi \colon \vec{x} \mapsto \phi(\vec{x})$. Then the dot product becomes $\phi(\vec{x}_i)^{\mathrm{T}}\phi(\vec{x}_j)$. If it turned out that this dot product (which is just a real number) could be computed simply and efficiently in terms of the original data points, then we wouldn't have to actually map from $\vec{x} \mapsto \phi(\vec{x})$. Rather, we could simply compute the quantity $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i)^{\mathrm{T}}\phi(\vec{x}_j)$, and then use the function's value in Equation (15.13). A *kernel function* K is such a function that corresponds to a dot product in some expanded feature space.

KERNEL FUNCTION

Example 15.2: The quadratic kernel in two dimensions. For 2-dimensional vectors $\vec{u} = (u_1 \quad u_2)$, $\vec{v} = (v_1 \quad v_2)$, consider $K(\vec{u}, \vec{v}) = (1 + \vec{u}^{\mathrm{T}}\vec{v})^2$. We wish to show that this is a kernel, i.e., that $K(\vec{u}, \vec{v}) = \phi(\vec{u})^{\mathrm{T}}\phi(\vec{v})$ for some ϕ. Consider $\phi(\vec{u}) = (1 \;\; u_1^2 \;\; \sqrt{2}u_1u_2 \;\; u_2^2 \;\; \sqrt{2}u_1 \;\; \sqrt{2}u_2)$. Then:

$$\begin{aligned} K(\vec{u}, \vec{v}) &= (1 + \vec{u}^{\mathrm{T}}\vec{v})^2 \\ &= 1 + u_1^2v_1^2 + 2u_1v_1u_2v_2 + u_2^2v_2^2 + 2u_1v_1 + 2u_2v_2 \\ &= (1 \;\; u_1^2 \;\; \sqrt{2}u_1u_2 \;\; u_2^2 \;\; \sqrt{2}u_1 \;\; \sqrt{2}u_2)^{\mathrm{T}}(1 \;\; v_1^2 \;\; \sqrt{2}v_1v_2 \;\; v_2^2 \;\; \sqrt{2}v_1 \;\; \sqrt{2}v_2) \\ &= \phi(\vec{u})^{\mathrm{T}}\phi(\vec{v}) \end{aligned} \tag{15.14}$$

KERNEL

MERCER KERNEL

In the language of functional analysis, what kinds of functions are valid *kernel functions*? Kernel functions are sometimes more precisely referred to as *Mercer kernels*, because they must satisfy Mercer's condition: for any $g(\vec{x})$ such that $\int g(\vec{x})^2 d\vec{x}$ is finite, we must have that:

$$\int K(\vec{x}, \vec{z})g(\vec{x})g(\vec{z})d\vec{x}d\vec{z} \geq 0\,. \tag{15.15}$$

A kernel function K must be continuous, symmetric, and have a positive definite gram matrix. Such a K means that there exists a mapping to a reproducing kernel Hilbert space (a Hilbert space is a vector space closed under dot products) such that the dot product there gives the same value as the function K. If a kernel does not satisfy Mercer's condition, then the corresponding QP may have no solution. If you would like to better understand these issues, you should consult the books on SVMs mentioned in Section 15.5. Otherwise, you can content yourself with knowing that 90% of work with kernels uses one of two straightforward families of functions of two vectors, which we define below, and which define valid kernels.

The two commonly used families of kernels are polynomial kernels and radial basis functions. Polynomial kernels are of the form $K(\vec{x}, \vec{z}) = (1 +$

$\vec{x}^{\mathrm{T}}\vec{z})^d$. The case of $d = 1$ is a linear kernel, which is what we had before the start of this section (the constant 1 just changing the threshold). The case of $d = 2$ gives a quadratic kernel, and is very commonly used. We illustrated the quadratic kernel in Example 15.2.

The most common form of radial basis function is a Gaussian distribution, calculated as:

$$K(\vec{x}, \vec{z}) = e^{-(\vec{x}-\vec{z})^2/(2\sigma^2)} \tag{15.16}$$

A radial basis function (rbf) is equivalent to mapping the data into an infinite dimensional Hilbert space, and so we cannot illustrate the radial basis function concretely, as we did a quadratic kernel. Beyond these two families, there has been interesting work developing other kernels, some of which is promising for text applications. In particular, there has been investigation of string kernels (see Section 15.5).

The world of SVMs comes with its own language, which is rather different from the language otherwise used in machine learning. The terminology does have deep roots in mathematics, but it's important not to be too awed by that terminology. Really, we are talking about some quite simple things. A polynomial kernel allows us to model feature conjunctions (up to the order of the polynomial). That is, if we want to be able to model occurrences of pairs of words, which give distinctive information about topic classification, not given by the individual words alone, like perhaps *operating* AND *system* or *ethnic* AND *cleansing*, then we need to use a quadratic kernel. If occurrences of triples of words give distinctive information, then we need to use a cubic kernel. Simultaneously you also get the powers of the basic features – for most text applications, that probably isn't useful, but just comes along with the math and hopefully doesn't do harm. A radial basis function allows you to have features that pick out circles (hyperspheres) – although the decision boundaries become much more complex as multiple such features interact. A string kernel lets you have features that are character subsequences of terms. All of these are straightforward notions which have also been used in many other places under different names.

15.2.4 Experimental results

We presented results in Section 13.6 showing that an SVM is a very effective text classifier. The results of Dumais et al. (1998) given in Table 13.9 show SVMs clearly performing the best. This was one of several pieces of work from this time that established the strong reputation of SVMs for text classification. Another pioneering work on scaling and evaluating SVMs for text classification was (Joachims 1998). We present some of his results

		Roc-	Dec.		linear SVM		rbf-SVM
	NB	chio	Trees	kNN	$C = 0.5$	$C = 1.0$	$\sigma \approx 7$
earn	96.0	96.1	96.1	97.8	98.0	98.2	98.1
acq	90.7	92.1	85.3	91.8	95.5	95.6	94.7
money-fx	59.6	67.6	69.4	75.4	78.8	78.5	74.3
grain	69.8	79.5	89.1	82.6	91.9	93.1	93.4
crude	81.2	81.5	75.5	85.8	89.4	89.4	88.7
trade	52.2	77.4	59.2	77.9	79.2	79.2	76.6
interest	57.6	72.5	49.1	76.7	75.6	74.8	69.1
ship	80.9	83.1	80.9	79.8	87.4	86.5	85.8
wheat	63.4	79.4	85.5	72.9	86.6	86.8	82.4
corn	45.2	62.2	87.7	71.4	87.5	87.8	84.6
microavg.	72.3	79.9	79.4	82.6	86.7	87.5	86.4

► **Table 15.2** SVM classifier break-even F_1 from (Joachims 2002a, p. 114). Results are shown for the 10 largest categories and for microaveraged performance over all 90 categories on the Reuters-21578 data set.

from (Joachims 2002a) in Table 15.2.[5] Joachims used a large number of term features in contrast to Dumais et al. (1998), who used MI feature selection (Section 13.5.1, page 272) to build classifiers with a much more limited number of features. The success of the linear SVM mirrors the results discussed in Section 14.6 (page 308) on other linear approaches like Naive Bayes. It seems that working with simple term features can get one a long way. It is again noticeable the extent to which different papers' results for the same machine learning methods differ. In particular, based on replications by other researchers, the Naive Bayes results of (Joachims 1998) appear too weak, and the results in Table 13.9 should be taken as representative.

15.3 Issues in the classification of text documents

There are lots of applications of text classification in the commercial world; email spam filtering is perhaps now the most ubiquitous. Jackson and Moulinier (2002) write: "There is no question concerning the commercial value of being able to classify documents automatically by content. There are myriad

[5] These results are in terms of the break-even F_1 (see Section 8.4). Many researchers disprefer this measure for text classification evaluation, since its calculation may involve interpolation rather than an actual parameter setting of the system and it is not clear why this value should be reported rather than maximal F_1 or another point on the precision/recall curve motivated by the task at hand. While earlier results in (Joachims 1998) suggested notable gains on this task from the use of higher order polynomial or rbf kernels, this was with hard-margin SVMs. With soft-margin SVMs, a simple linear SVM with the default $C = 1$ performs best.

potential applications of such a capability for corporate Intranets, government departments, and Internet publishers."

Most of our discussion of classification has focused on introducing various machine learning methods rather than discussing particular features of text documents relevant to classification. This bias is appropriate for a textbook, but is misplaced for an application developer. It is frequently the case that greater performance gains can be achieved from exploiting domain-specific text features than from changing from one machine learning method to another. Jackson and Moulinier (2002) suggest that "Understanding the data is one of the keys to successful categorization, yet this is an area in which most categorization tool vendors are extremely weak. Many of the 'one size fits all' tools on the market have not been tested on a wide range of content types." In this section we wish to step back a little and consider the applications of text classification, the space of possible solutions, and the utility of application-specific heuristics.

15.3.1 Choosing what kind of classifier to use

When confronted with a need to build a text classifier, the first question to ask is how much training data is there currently available? None? Very little? Quite a lot? Or a huge amount, growing every day? Often one of the biggest practical challenges in fielding a machine learning classifier in real applications is creating or obtaining enough training data. For many problems and algorithms, hundreds or thousands of examples from each class are required to produce a high performance classifier and many real world contexts involve large sets of categories. We will initially assume that the classifier is needed as soon as possible; if a lot of time is available for implementation, much of it might be spent on assembling data resources.

If you have no labeled training data, and especially if there are existing staff knowledgeable about the domain of the data, then you should never forget the solution of using hand-written rules. That is, you write standing queries, as we touched on at the beginning of Chapter 13. For example:

IF (wheat OR grain) AND NOT (whole OR bread) THEN $c =$ grain

In practice, rules get a lot bigger than this, and can be phrased using more sophisticated query languages than just Boolean expressions, including the use of numeric scores. With careful crafting (that is, by humans tuning the rules on development data), the accuracy of such rules can become very high. Jacobs and Rau (1990) report identifying articles about takeovers with 92% precision and 88.5% recall, and Hayes and Weinstein (1990) report 94% recall and 84% precision over 675 categories on Reuters newswire documents. Nevertheless the amount of work to create such well-tuned rules is very large. A reasonable estimate is 2 days per class, and extra time has to go into maintenance of rules, as the content of documents in classes drifts over time (cf. page 269).

If you have fairly little data and you are going to train a supervised classifier, then machine learning theory says you should stick to a classifier with high bias, as we discussed in Section 14.6 (page 308). For example, there are theoretical and empirical results that Naive Bayes does well in such circumstances (Ng and Jordan 2001, Forman and Cohen 2004), although this effect is not necessarily observed in practice with regularized models over textual data (Klein and Manning 2002). At any rate, a very low bias model like a nearest neighbor model is probably counterindicated. Regardless, the quality of the model will be adversely affected by the limited training data.

SEMI-SUPERVISED LEARNING

Here, the theoretically interesting answer is to try to apply *semi-supervised training methods*. This includes methods such as bootstrapping or the EM algorithm, which we will introduce in Section 16.5 (page 368). In these methods, the system gets some labeled documents, and a further large supply of unlabeled documents over which it can attempt to learn. One of the big advantages of Naive Bayes is that it can be straightforwardly extended to be a semi-supervised learning algorithm, but for SVMs, there is also semi-supervised learning work which goes under the title of *transductive SVMs*. See the references for pointers.

TRANSDUCTIVE SVMS

Often, the practical answer is to work out how to get more labeled data as quickly as you can. The best way to do this is to insert yourself into a process where humans will be willing to label data for you as part of their natural tasks. For example, in many cases humans will sort or route email for their own purposes, and these actions give information about classes. The alternative of getting human labelers expressly for the task of training classifiers is often difficult to organize, and the labeling is often of lower quality, because the labels are not embedded in a realistic task context. Rather than getting people to label all or a random sample of documents, there has also been considerable research on *active learning*, where a system is built which decides which documents a human should label. Usually these are the ones on which a classifier is uncertain of the correct classification. This can be effective in reducing annotation costs by a factor of 2–4, but has the problem that the good documents to label to train one type of classifier often are not the good documents to label to train a different type of classifier.

ACTIVE LEARNING

If there is a reasonable amount of labeled data, then you are in the perfect position to use everything that we have presented about text classification. For instance, you may wish to use an SVM. However, if you are deploying a linear classifier such as an SVM, you should probably design an application that overlays a Boolean rule-based classifier over the machine learning classifier. Users frequently like to adjust things that do not come out quite right, and if management gets on the phone and wants the classification of a particular document fixed right now, then this is much easier to do by hand-writing a rule than by working out how to adjust the weights of an SVM without destroying the overall classification accuracy. This is one reason why machine learning models like decision trees which produce user-interpretable Boolean-like models retain considerable popularity.

If a huge amount of data are available, then the choice of classifier probably has little effect on your results and the best choice may be unclear (cf. Banko and Brill 2001). It may be best to choose a classifier based on the scalability of training or even runtime efficiency. To get to this point, you need to have huge amounts of data. The general rule of thumb is that each doubling of the training data size produces a linear increase in classifier performance, but with very large amounts of data, the improvement becomes sub-linear.

15.3.2 Improving classifier performance

For any particular application, there is usually significant room for improving classifier effectiveness through exploiting features specific to the domain or document collection. Often documents will contain zones which are especially useful for classification. Often there will be particular subvocabularies which demand special treatment for optimal classification effectiveness.

Large and difficult category taxonomies

If a text classification problem consists of a small number of well-separated categories, then many classification algorithms are likely to work well. But many real classification problems consist of a very large number of often very similar categories. The reader might think of examples like web directories (the Yahoo! Directory or the Open Directory Project), library classification schemes (Dewey Decimal or Library of Congress) or the classification schemes used in legal or medical applications. For instance, the Yahoo! Directory consists of over 200,000 categories in a deep hierarchy. Accurate classification over large sets of closely related classes is inherently difficult.

HIERARCHICAL CLASSIFICATION

Most large sets of categories have a hierarchical structure, and attempting to exploit the hierarchy by doing *hierarchical classification* is a promising approach. However, at present the effectiveness gains from doing this rather than just working with the classes that are the leaves of the hierarchy remain modest.[6] But the technique can be very useful simply to improve the scalability of building classifiers over large hierarchies. Another simple way to improve the scalability of classifiers over large hierarchies is the use of aggressive feature selection. We provide references to some work on hierarchical classification in Section 15.5.

[6] Using the small hierarchy in Figure 13.1 (page 257) as an example, the leaf classes are ones like *poultry* and *coffee*, as opposed to higher-up classes like *industries*.

A general result in machine learning is that you can always get a small boost in classification accuracy by combining multiple classifiers, provided only that the mistakes that they make are at least somewhat independent. There is now a large literature on techniques such as voting, bagging, and boosting multiple classifiers. Again, there are some pointers in the references. Nevertheless, ultimately a hybrid automatic/manual solution may be needed to achieve sufficient classification accuracy. A common approach in such situations is to run a classifier first, and to accept all its high confidence decisions, but to put low confidence decisions in a queue for manual review. Such a process also automatically leads to the production of new training data which can be used in future versions of the machine learning classifier. However, note that this is a case in point where the resulting training data is clearly *not* randomly sampled from the space of documents.

Features for text

The default in both ad hoc retrieval and text classification is to use terms as features. However, for text classification, a great deal of mileage can be achieved by designing additional features which are suited to a specific problem. Unlike the case of IR query languages, since these features are internal to the classifier, there is no problem of communicating these features to an end user. This process is generally referred to as *feature engineering*. At present, feature engineering remains a human craft, rather than something done by machine learning. Good feature engineering can often markedly improve the performance of a text classifier. It is especially beneficial in some of the most important applications of text classification, like spam and porn filtering.

FEATURE ENGINEERING

Classification problems will often contain large numbers of terms which can be conveniently grouped, and which have a similar vote in text classification problems. Typical examples might be year mentions or strings of exclamation marks. Or they may be more specialized tokens like ISBNs or chemical formulas. Often, using them directly in a classifier would greatly increase the vocabulary without providing classificatory power beyond knowing that, say, a chemical formula is present. In such cases, the number of features and feature sparseness can be reduced by matching such items with regular expressions and converting them into distinguished tokens. Consequently, effectiveness and classifier speed are normally enhanced. Sometimes all numbers are converted into a single feature, but often some value can be had by distinguishing different kinds of numbers, such as four digit numbers (which are usually years) versus other cardinal numbers versus real numbers with a decimal point. Similar techniques can be applied to dates, ISBN numbers, sports game scores, and so on.

Going in the other direction, it is often useful to increase the number of features by matching parts of words, and by matching selected multiword patterns that are particularly discriminative. Parts of words are often matched by character *k*-gram features. Such features can be particularly good at providing classification clues for otherwise unknown words when the classifier is deployed. For instance, an unknown word ending in *-rase* is likely to be an enzyme, even if it wasn't seen in the training data. Good multiword patterns are often found by looking for distinctively common word pairs (perhaps using a mutual information criterion between words, in a similar way to its use in Section 13.5.1 (page 272) for feature selection) and then using feature selection methods evaluated against classes. They are useful when the components of a compound would themselves be misleading as classification cues. For instance, this would be the case if the keyword *ethnic* was most indicative of the categories *food* and *arts*, the keyword *cleansing* was most indicative of the category *home*, but the collocation *ethnic cleansing* instead indicates the category *world news*. Some text classifiers also make use of features from named entity recognizers (cf. page 195).

Do techniques like stemming and lowercasing (Section 2.2, page 22) help for text classification? As always, the ultimate test is empirical evaluations conducted on an appropriate test collection. But it is nevertheless useful to note that such techniques have a more restricted chance of being useful for classification. For IR, you often need to collapse forms of a word like *oxygenate* and *oxygenation*, because the appearance of either in a document is a good clue that the document will be relevant to a query about oxygenation. Given copious training data, stemming necessarily delivers no value for text classification. If several forms that stem together have a similar signal, the parameters estimated for all of them will have similar weights. Techniques like stemming help only in compensating for data sparseness. This can be a useful role (as noted at the start of this section), but often different forms of a word can convey significantly different cues about the correct document classification. Overly aggressive stemming can easily degrade classification performance.

Document zones in text classification

As already discussed in Section 6.1, documents usually have zones, such as mail message headers like the subject and author, or the title and keywords of a research article. Text classifiers can usually gain from making use of these zones during training and classification.

Upweighting document zones. In text classification problems, you can frequently get a nice boost to effectiveness by differentially weighting contributions from different document zones. Often, upweighting title words is particularly effective (Cohen and Singer 1999, p. 163). As a rule of thumb, it is often effective to double the weight of title words in text classification problems. You can also get value from upweighting words from pieces of text that are not so much clearly defined zones, but where nevertheless evidence from document structure or content suggests that they are important. Murata et al. (2000) suggest that you can also get value (in an ad hoc retrieval context) from upweighting the first sentence of a (newswire) document.

PARAMETER TYING

Separate feature spaces for document zones. There are two strategies that can be used for document zones. Above we upweighted words that appear in certain zones. This means that we are using the same features (that is, parameters are "tied" across different zones), but we pay more attention to the occurrence of terms in particular zones. An alternative strategy is to have a completely separate set of features and corresponding parameters for words occurring in different zones. This is in principle more powerful: a word could usually indicate the topic *Middle East* when in the title but *Commodities* when in the body of a document. But, in practice, tying parameters is usually more successful. Having separate feature sets means having two or more times as many parameters, many of which will be much more sparsely seen in the training data, and hence with worse estimates, whereas upweighting has no bad effects of this sort. Moreover, it is quite uncommon for words to have different preferences when appearing in different zones; it is mainly the strength of their vote that should be adjusted. Nevertheless, ultimately this is a contingent result, depending on the nature and quantity of the training data.

Connections to text summarization. In Section 8.7, we mentioned the field of text summarization, and how most work in that field has adopted the limited goal of extracting and assembling pieces of the original text that are judged to be central based on features of sentences that consider the sentence's position and content. Much of this work can be used to suggest zones that may be distinctively useful for text classification. For example Kołcz et al. (2000) consider a form of feature selection where you classify documents based only on words in certain zones. Based on text summarization research, they consider using (i) only the title, (ii) only the first paragraph, (iii) only the paragraph with the most title words or keywords, (iv) the first two paragraphs or the first and last paragraph, or (v) all sentences with a minimum number of title words or keywords. In general, these positional feature selection methods produced as good results as mutual information (Section 13.5.1), and resulted in quite competitive classifiers. Ko et al. (2004) also took inspiration from text summarization research to upweight sentences with either words from the title or words that are central to the document's content, leading to classification accuracy gains of almost 1%. This presumably works because most such sentences are somehow more central to the concerns of the document.

?

Exercise 15.4 [⋆⋆]

Spam email often makes use of various cloaking techniques to try to get through. One method is to pad or substitute characters so as to defeat word-based text classifiers. For example, you see terms like the following in spam email:

Rep1icaRolex   bonmus   Viiiaaaagra   pi11z
PHARlbdMACY   [LEV]i[IT]l[RA]   se∧xual   ClAfLlS

Discuss how you could engineer features that would largely defeat this strategy.

Exercise 15.5 [⋆⋆]

Another strategy often used by purveyors of email spam is to follow the message they wish to send (such as buying a cheap stock or whatever) with a paragraph of text from another innocuous source (such as a news article). Why might this strategy be effective? How might it be addressed by a text classifier?

Exercise 15.6 [⋆]

What other kinds of features appear as if they would be useful in an email spam classifier?

15.4 Machine learning methods in ad hoc information retrieval

Rather than coming up with term and document weighting functions by hand, as we primarily did in Chapter 6, we can view different sources of relevance signal (cosine score, title match, etc.) as features in a learning problem. A classifier that has been fed examples of relevant and nonrelevant documents for each of a set of queries can then figure out the relative weights of these signals. If we configure the problem so that there are pairs of a document and a query which are assigned a relevance judgment of *relevant* or *nonrelevant*, then we can think of this problem too as a text classification problem. Taking such a classification approach is not necessarily best, and we present an alternative in Section 15.4.2. Nevertheless, given the material we have covered, the simplest place to start is to approach this problem as a classification problem, by ordering the documents according to the confidence of a two-class classifier in its relevance decision. And this move is not purely pedagogical; exactly this approach is sometimes used in practice.

15.4.1 A simple example of machine-learned scoring

In this section we generalize the methodology of Section 6.1.2 (page 113) to *machine learning* of the scoring function. In Section 6.1.2 we considered a case where we had to combine Boolean indicators of relevance; here we consider more general factors to further develop the notion of machine-learned

Example	DocID	Query	Cosine score	ω	Judgment
Φ_1	37	linux operating system	0.032	3	*relevant*
Φ_2	37	penguin logo	0.02	4	*nonrelevant*
Φ_3	238	operating system	0.043	2	*relevant*
Φ_4	238	runtime environment	0.004	2	*nonrelevant*
Φ_5	1741	kernel layer	0.022	3	*relevant*
Φ_6	2094	device driver	0.03	2	*relevant*
Φ_7	3191	device driver	0.027	5	*nonrelevant*
...	...	...	...	...	...

► **Table 15.3** Training examples for machine-learned scoring.

relevance. In particular, the factors we now consider go beyond Boolean functions of query term presence in document zones, as in Section 6.1.2.

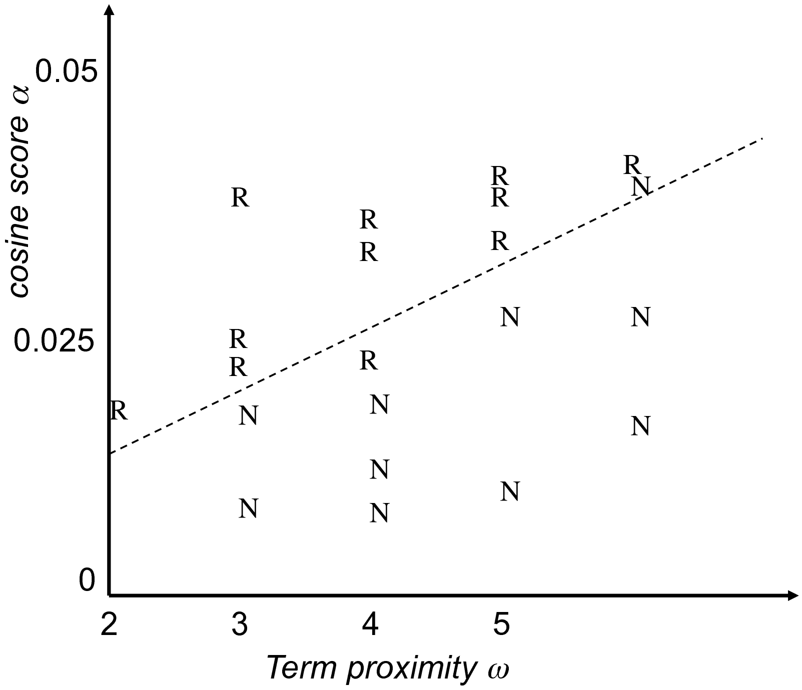
We develop the ideas in a setting where the scoring function is a linear combination of two factors: (1) the vector space cosine similarity between query and document and (2) the minimum window width ω within which the query terms lie. As we noted in Section 7.2.2 (page 144), query term proximity is often very indicative of a document being on topic, especially with longer documents and on the web. Among other things, this quantity gives us an implementation of implicit phrases. Thus we have one factor that depends on the statistics of query terms in the document as a bag of words, and another that depends on proximity weighting. We consider only two features in the development of the ideas because a two-feature exposition remains simple enough to visualize. The technique can be generalized to many more features.

As in Section 6.1.2, we are provided with a set of *training examples*, each of which is a pair consisting of a query and a document, together with a relevance judgment for that document on that query that is either *relevant* or *nonrelevant*. For each such example we can compute the vector space cosine similarity, as well as the window width ω. The result is a training set as shown in Table 15.3, which resembles Figure 6.5 (page 115) from Section 6.1.2.

Here, the two features (cosine score denoted α and window width ω) are real-valued predictors. If we once again quantify the judgment *relevant* as 1 and *nonrelevant* as 0, we seek a scoring function that combines the values of the features to generate a value that is (close to) 0 or 1. We wish this function to be in agreement with our set of training examples as far as possible. Without loss of generality, a linear classifier will use a linear combination of features of the form

$$Score(d,q) = Score(\alpha,\omega) = a\alpha + b\omega + c, \tag{15.17}$$

with the coefficients a, b, c to be learned from the training data. While it is



► **Figure 15.7** A collection of training examples. Each R denotes a training example labeled *relevant*, while each N is a training example labeled *nonrelevant*.

possible to formulate this as an error minimization problem as we did in Section 6.1.2, it is instructive to visualize the geometry of Equation (15.17). The examples in Table 15.3 can be plotted on a two-dimensional plane with axes corresponding to the cosine score α and the window width ω. This is depicted in Figure 15.7.

In this setting, the function $Score(\alpha, \omega)$ from Equation (15.17) represents a plane "hanging above" Figure 15.7. Ideally this plane (in the direction perpendicular to the page containing Figure 15.7) assumes values close to 1 above the points marked R, and values close to 0 above the points marked N. Since a plane is unlikely to assume only values close to 0 or 1 above the training sample points, we make use of *thresholding*: given any query and document for which we wish to determine relevance, we pick a value θ and if $Score(\alpha, \omega) > \theta$ we declare the document to be *relevant*, else we declare the document to be *nonrelevant*. As we know from Figure 14.8 (page 301), all points that satisfy $Score(\alpha, \omega) = \theta$ form a line (shown as a dashed line in Figure 15.7) and we thus have a linear classifier that separates relevant from nonrelevant instances. Geometrically, we can find the separating line as follows. Consider the line passing through the plane $Score(\alpha, \omega)$ whose height is θ above the page containing Figure 15.7. Project this line down onto Figure 15.7; this will be the dashed line in Figure 15.7. Then, any subsequent query/document pair that falls below the dashed line in Figure 15.7 is deemed *nonrelevant*; above the dashed line, *relevant*.

Thus, the problem of making a binary *relevant*/*nonrelevant* judgment given training examples as above turns into one of learning the dashed line in Figure 15.7 separating *relevant* training examples from the *nonrelevant* ones. Being in the α-ω plane, this line can be written as a linear equation involving α and ω, with two parameters (slope and intercept). The methods of linear classification that we have already looked at in Chapters 13–15 provide methods for choosing this line. Provided we can build a sufficiently rich collection of training samples, we can thus altogether avoid hand-tuning score functions as in Section 7.2.3 (page 145). The bottleneck of course is the ability to maintain a suitably representative set of training examples, whose relevance assessments must be made by experts.

15.4.2 Result ranking by machine learning

The above ideas can be readily generalized to functions of many more than two variables. There are lots of other scores that are indicative of the relevance of a document to a query, including static quality (PageRank-style measures, discussed in Chapter 21), document age, zone contributions, document length, and so on. Providing that these measures can be calculated for a training document collection with relevance judgments, any number of such measures can be used to train a machine learning classifier. For instance, we could train an SVM over binary relevance judgments, and order documents based on their probability of relevance, which is monotonic with the documents' signed distance from the decision boundary.

REGRESSION
ORDINAL REGRESSION

However, approaching IR result ranking like this is not necessarily the right way to think about the problem. Statisticians normally first divide problems into classification problems (where a categorical variable is predicted) versus *regression* problems (where a real number is predicted). In between is the specialized field of *ordinal regression* where a ranking is predicted. Machine learning for ad hoc retrieval is most properly thought of as an ordinal regression problem, where the goal is to rank a set of documents for a query, given training data of the same sort. This formulation gives some additional power, since documents can be evaluated relative to other candidate documents for the same query, rather than having to be mapped to a global scale of goodness, while also weakening the problem space, since just a ranking is required rather than an absolute measure of relevance. Issues of ranking are especially germane in web search, where the ranking at the very top of the results list is exceedingly important, whereas decisions of relevance of a document to a query may be much less important. Such work can and has been pursued using the structural SVM framework which we mentioned in Section 15.2.2, where the class being predicted is a ranking of results for a query, but here we will present the slightly simpler ranking SVM.

RANKING SVM

The construction of a *ranking SVM* proceeds as follows. We begin with a set of judged queries. For each training query q, we have a set of documents returned in response to the query, which have been totally ordered by a person for relevance to the query. We construct a vector of features $\psi_j = \psi(d_j, q)$ for each document/query pair, using features such as those discussed in Section 15.4.1, and many more. For two documents d_i and d_j, we then form the vector of feature differences:

(15.18) $$\Phi(d_i, d_j, q) = \psi(d_i, q) - \psi(d_j, q)$$

By hypothesis, one of d_i and d_j has been judged more relevant. If d_i is judged more relevant than d_j, denoted $d_i \prec d_j$ (d_i should precede d_j in the results ordering), then we will assign the vector $\Phi(d_i, d_j, q)$ the class $y_{ijq} = +1$; otherwise -1. The goal then is to build a classifier which will return

(15.19) $$\vec{w}^{\mathrm{T}}\Phi(d_i, d_j, q) > 0 \quad \text{iff} \quad d_i \prec d_j$$

This SVM learning task is formalized in a manner much like the other examples that we saw before:

(15.20) Find $\vec{w}$, and $\xi_{i,j} \geq 0$ such that:

- $\frac{1}{2}\vec{w}^{\mathrm{T}}\vec{w} + C\sum_{i,j}\xi_{i,j}$ is minimized
- and for all $\{\Phi(d_i, d_j, q) : d_i \prec d_j\}$, $\vec{w}^{\mathrm{T}}\Phi(d_i, d_j, q) \geq 1 - \xi_{i,j}$

We can leave out y_{ijq} in the statement of the constraint, since we only need to consider the constraint for document pairs ordered in one direction, since $\prec$ is antisymmetric. These constraints are then solved, as before, to give a linear classifier which can rank pairs of documents. This approach has been used to build ranking functions which outperform standard hand-built ranking functions in IR evaluations on standard data sets; see the references for papers that present such results.

Both of the methods that we have just looked at use a linear weighting of document features that are indicators of relevance, as has most work in this area. It is therefore perhaps interesting to note that much of traditional IR weighting involves *nonlinear* scaling of basic measurements (such as log-weighting of term frequency, or idf). At the present time, machine learning is very good at producing optimal weights for features in a linear combination (or other similar restricted model classes), but it is not good at coming up with good nonlinear scalings of basic measurements. This area remains the domain of human feature engineering.

The idea of learning ranking functions has been around for a number of years, but it is only very recently that sufficient machine learning knowledge, training document collections, and computational power have come together to make this method practical and exciting. It is thus too early to write something definitive on machine learning approaches to ranking in information retrieval, but there is every reason to expect the use and importance of machine learned ranking approaches to grow over time. While skilled humans can do a very good job at defining ranking functions by hand, hand tuning is difficult, and it has to be done again for each new document collection and class of users.

Exercise 15.7

Plot the first 7 rows of Table 15.3 in the α-ω plane to produce a figure like that in Figure 15.7.

Exercise 15.8

Write down the equation of a line in the α-ω plane separating the Rs from the Ns.

Exercise 15.9

Give a training example (consisting of values for α, ω and the relevance judgment) that when added to the training set makes it impossible to separate the R's from the N's using a line in the α-ω plane.

15.5 References and further reading

The somewhat quirky name support vector machine originates in the neural networks literature, where learning algorithms were thought of as architectures, and often referred to as "machines". The distinctive element of this model is that the decision boundary to use is completely decided ("supported") by a few training data points, the support vectors.

For a more detailed presentation of SVMs, a good, well-known article-length introduction is (Burges 1998). Chen et al. (2005) introduce the more recent ν-SVM, which provides an alternative parameterization for dealing with inseparable problems, whereby rather than specifying a penalty C, you specify a parameter ν which bounds the number of examples which can appear on the wrong side of the decision surface. There are now also several books dedicated to SVMs, large margin learning, and kernels: (Cristianini and Shawe-Taylor 2000) and (Schölkopf and Smola 2001) are more mathematically oriented, while (Shawe-Taylor and Cristianini 2004) aims to be more practical. For the foundations by their originator, see (Vapnik 1998).

Some recent, more general books on statistical learning, such as (Hastie et al. 2001) also give thorough coverage of SVMs.

The construction of *multiclass SVMs* is discussed in (Weston and Watkins 1999), (Crammer and Singer 2001), and (Tsochantaridis et al. 2005). The last reference provides an introduction to the general framework of structural SVMs.

The kernel trick was first presented in (Aizerman et al. 1964). For more about string kernels and other kernels for structured data, see (Lodhi et al. 2002) and (Gaertner et al. 2002). The Advances in Neural Information Processing (NIPS) conferences have become the premier venue for theoretical machine learning work, such as on SVMs. Other venues such as SIGIR are much stronger on experimental methodology and using text-specific features to improve classifier effectiveness.

A recent comparison of most current machine learning classifiers (though on problems rather different from typical text problems) can be found in (Caruana and Niculescu-Mizil 2006). (Li and Yang 2003), discussed in Section 13.6, is the most recent comparative evaluation of machine learning classifiers on text classification. Older examinations of classifiers on text problems can be found in (Yang 1999, Yang and Liu 1999, Dumais et al. 1998). Joachims (2002a) presents his work on SVMs applied to text problems in detail. Zhang and Oles (2001) present an insightful comparison of Naive Bayes, regularized logistic regression and SVM classifiers.

Joachims (1999) discusses methods of making SVM learning practical over large text data sets. Joachims (2006a) improves on this work.

A number of approaches to hierarchical classification have been developed in order to deal with the common situation where the classes to be assigned have a natural hierarchical organization (Koller and Sahami 1997, McCallum et al. 1998, Weigend et al. 1999, Dumais and Chen 2000). In a recent large study on scaling SVMs to the entire Yahoo! directory, Liu et al. (2005) conclude that hierarchical classification noticeably if still modestly outperforms flat classification. Classifier effectiveness remains limited by the very small number of training documents for many classes. For a more general approach that can be applied to modeling relations between classes, which may be arbitrary rather than simply the case of a hierarchy, see Tsochantaridis et al. (2005).

Moschitti and Basili (2004) investigate the use of complex nominals, proper nouns and word senses as features in text classification.

Dietterich (2002) overviews ensemble methods for classifier combination, while Schapire (2003) focuses particularly on boosting, which is applied to text classification in (Schapire and Singer 2000).

Chapelle et al. (2006) present an introduction to work in semi-supervised methods, including in particular chapters on using EM for semi-supervised text classification (Nigam et al. 2006) and on transductive SVMs (Joachims

2006b). Sindhwani and Keerthi (2006) present a more efficient implementation of a transductive SVM for large data sets.

Tong and Koller (2001) explore active learning with SVMs for text classification; Baldridge and Osborne (2004) point out that examples selected for annotation with one classifier in an active learning context may be no better than random examples when used with another classifier.

Machine learning approaches to ranking for ad hoc retrieval were pioneered in (Wong et al. 1988), (Fuhr 1992), and (Gey 1994). But limited training data and poor machine learning techniques meant that these pieces of work achieved only middling results, and hence they only had limited impact at the time.

Taylor et al. (2006) study using machine learning to tune the parameters of the BM25 family of ranking functions (Section 11.4.3, page 232) so as to maximize NDCG (Section 8.4, page 163). Machine learning approaches to ordinal regression appear in (Herbrich et al. 2000) and (Burges et al. 2005), and are applied to clickstream data in (Joachims 2002b). Cao et al. (2006) study how to make this approach effective in IR, and Qin et al. (2007) suggest an extension involving using multiple hyperplanes. Yue et al. (2007) study how to do ranking with a structural SVM approach, and in particular show how this construction can be effectively used to directly optimize for MAP (Section 8.4, page 158), rather than using surrogate measures like accuracy or area under the ROC curve. Geng et al. (2007) study feature selection for the ranking problem.

Other approaches to learning to rank have also been shown to be effective for web search, such as (Burges et al. 2005, Richardson et al. 2006).